САМОУЧИТЕЛЬ

bhv

MySQL 5

Максим Кузнецов
Игорь Симдянов

- Полный список функций MySQL 5
- Язык запросов SQL
- Хранимые процедуры
- Представления
- Безопасность
- Целостность данных

Изучение на примерах — гарантия успешного освоения MySQL 5

Максим Кузнецов
Игорь Симдянов

САМОУЧИТЕЛЬ MySQL 5

Санкт-Петербург
«БХВ-Петербург»
2006

УДК 681.3.06
ББК 32.973.26-018.2
К89

Кузнецов М. В., Симдянов И. В.

К89 Самоучитель MySQL 5. — СПб.: БХВ-Петербург, 2006. — 560 с.: ил.

ISBN 978-5-94157-754-5

Описывается пятая версия популярной бесплатной СУБД MySQL. В начале книги происходит знакомство читателя с MySQL и простейшими SQL-запросами, такими как создание баз данных и таблиц, их заполнение, извлечение и удаление записей. Далее рассматриваются сложные вопросы SQL-программирования: встроенные функции, полнотекстовый поиск, транзакции, временные таблицы. В заключительной части разбираются различные нововведения, появившиеся только в MySQL 5: вложенные запросы, хранимые процедуры и функции, представления, триггеры, курсоры, информационные схемы и т. п.

На компакт-диске размещена учебная база данных, на примере которой на протяжении всей книги демонстрируются особенности диалекта MySQL, и дистрибутивы MySQL версии 4.0, 4.1 и 5.0 для Windows и Linux, распространяемые в соответствии с лицензией GNU/GPL.

Для программистов и Web-разработчиков

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Наталья Сержантова*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 16.02.06.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 45,15.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-94157-754-5

© Кузнецов М. В., Симдянов И. В., 2006
© Оформление, издательство "БХВ-Петербург", 2006

Оглавление

Введение **1**

Для кого и о чем эта книга 1
Благодарности 2

ЧАСТЬ I. **3**

Глава 1. История развития баз данных. Понятие реляционной базы данных **5**

1.1. История развития СУБД. Реляционные базы данных 5
1.1.1. Иерархические базы данных 6
1.1.2. Сетевые базы данных 7
1.1.3. Реляционные базы данных 8
1.1.4. Объектно-ориентированные и гибридные базы данных 9
1.2. Особенности реляционных баз данных 10
1.2.1. Первичные ключи 10
1.2.2. Нормализация базы данных 12
1.2.3. Правила Кодда 14
1.3. СУБД и сети 15
1.3.1. Централизованная архитектура 16
1.3.2. Архитектура "клиент-сервер" 16
1.3.2.1. Два признака клиент-серверной архитектуры 17
1.3.3. Трехуровневая архитектура Интернета 18
1.4. Как работают базы данных и что такое SQL 19
1.5. Версии MySQL 21
1.5.1. Что нового в MySQL 4.1 23
1.5.2. Что нового в MySQL 5.0 23

Глава 2. Установка MySQL 5 ..... **25**

2.1. Получение дистрибутива 25

2.2. Установка на платформу Windows ... 26
2.3. Установка на платформу Linux ... 39

Глава 3. Работа с утилитами MySQL ... 41

3.1. Утилита *mysql* ... 42
3.2. Утилита *mysqldump* ... 50

Глава 4. Создание баз данных и таблиц. Типы данных ... 53

4.1. Создание базы данных ... 53
4.2. Создание таблицы ... 57
4.3. Типы данных ... 59
4.3.1. Числовые данные ... 60
4.3.2. Строковые данные ... 63
4.3.3. Календарные данные ... 66
4.3.4. Тип данных *NULL* ... 69
4.3.5. Выбор типа данных ... 71
4.4. Учебная база данных ... 72

Глава 5. Индексы ... 77

5.1. Первичный ключ ... 78
5.2. Обычный и уникальный индексы ... 81

Глава 6. Добавление данных ... 87

6.1. Однострочный оператор *INSERT* ... 87
6.2. Многострочный оператор *INSERT* ... 93
6.3. Пакетная загрузка данных ... 94

Глава 7. Выборка данных ... 99

7.1. Изменение количества и порядка следования столбцов ... 99
7.2. Условия ... 101
7.3. Сортировка ... 104
7.4. Ограничение выборки ... 107
7.5. Использование функций ... 108
7.6. Группировка записей ... 111
7.7. Объединение таблиц ... 116
7.8. Сохранение результатов во внешний файл ... 119

Глава 8. Удаление данных ... 123

8.1. Оператор *DELETE* ... 123
8.2. Оператор *TRUNCATE TABLE* ... 124

Глава 9. Обновление данных ... 125

9.1. Оператор *UPDATE* ... 125
9.2. Оператор *REPLACE* ... 129

Глава 10. Редактирование структуры таблиц 131

10.1. Добавление и удаление столбцов 131
10.2. Изменение уже существующих столбцов 133
10.3. Добавление и удаление индексов 135
10.4. Преобразование таблицы 137

ЧАСТЬ II 141

Глава 11. Типы и структура таблиц 143

11.1. MyISAM 143
11.2. MERGE 144
11.3. MEMORY (HEAP) 146
11.4. EXAMPLE 147
11.5. BDB (BerkeleyDB) 148
11.6. InnoDB 149
11.7. NDB Cluster 150
11.8. ARCHIVE 150
11.9. CSV 151
11.10. FEDERATED 152
11.11. Оператор *CREATE TABLE* 154
11.11.1. Структура таблицы 155
11.11.2. Параметры таблицы 161
ENGINE (TYPE) 161
AUTO_INCREMENT 162
AVG_ROW_LENGTH 162
[DEFAULT] CHARACTER SET 162
CHECKSUM 163
COMMENT 163
DATA DIRECTORY 164
DELAY_KEY_WRITE 164
INDEX DIRECTORY 164
INSERT_METHOD 164
MAX_ROWS 164
MIN_ROWS 165
PACK_KEYS 165
PASSWORD 165
ROW_FORMAT 165
UNION 166
11.12. Изменение типа таблицы 166

Глава 12. Приведение типов 167

12.1. Ключевое слово *BINARY* 167
12.2. Функция *CAST* 168
12.3. Функция *CONVERT* 170

Глава 13. Операторы и математические функции 171

13.1. Операторы 171
13.1.1. Арифметические операторы 171
13.1.2. Операторы сравнения 175
13.1.2.1. Равенство = ... 175
13.1.2.2. Оператор <=> 176
13.1.2.3. Оператор <> 177
13.1.2.4. Оператор < 177
13.1.2.5. Оператор <= 178
13.1.2.6. Оператор > 178
13.1.2.7. Оператор >= 179
13.1.2.8. Конструкция *IS NULL* 179
13.1.2.9. Конструкция *IS NOT NULL* 180
13.1.2.10. Конструкция *BETWEEN min AND max* 181
13.1.2.11. Конструкция *NOT BETWEEN min AND max* 182
13.1.2.12. Функция *COALESCE* 182
13.1.2.13. Функция *GREATEST* 183
13.1.2.14. Функция *LEAST* 183
13.1.2.15. Конструкция *IN* 184
13.1.2.16. Конструкция *NOT IN* 185
13.1.2.17. Функция *INTERVAL* 185
13.1.3. Логические операторы 186
13.1.3.1. Оператор *NOT* 186
13.1.3.2. Оператор *OR* 187
13.1.3.3. Оператор *AND* 187
13.1.3.4. Оператор *XOR* 188
13.1.4. Битовые операторы 188
13.1.4.1. Оператор & 188
13.1.4.2. Оператор | 189
13.1.4.3. Оператор ^ 190
13.1.4.4. Оператор ~ 191
13.1.4.5. Оператор << 191
13.1.4.6. Оператор >> 192
13.1.4.7. Функция *BIT_COUNT* 193
13.1.5. Приоритет операторов 193
13.2. Математические функции 194
13.2.1. Функция *ABS* 195
13.2.2. Функция *ACOS* 195
13.2.3. Функция *ASIN* 195
13.2.4. Функция *ATAN* 196
13.2.5. Функция *ATAN2* 196
13.2.6. Функция *CEILING* 196
13.2.7. Функция *COS* 197
13.2.8. Функция *COT* 198

13.2.9. Функция *CRC32* ... 198
13.2.10. Функция *DEGREES* ... 198
13.2.11. Функция *EXP* ... 199
13.2.12. Функция *FLOOR* ... 199
13.2.13. Функция *LOG* ... 199
13.2.14. Функция *LOG2* ... 200
13.2.15. Функция *LOG10* ... 200
13.2.16. Функция *MOD* ... 200
13.2.17. Функция *PI* ... 201
13.2.18. Функция *POW* ... 201
13.2.19. Функция *RADIANS* ... 201
13.2.20. Функция *RAND* ... 202
13.2.21. Функция *ROUND* ... 203
13.2.22. Функция *SIGN* ... 204
13.2.23. Функция *SIN* ... 204
13.2.24. Функция *SQPT* ... 204
13.2.25. Функция *TAN* ... 205
13.2.26. Функция *TRUNCATE* ... 205

Глава 14. Функции даты и времени ... 207

14.1. Функция *ADDDATE* ... 207
14.2. Функция *ADDTIME* ... 212
14.3. Функция *CONVERT_TZ* ... 213
14.4. Функция *CURDATE* ... 213
14.5. Функция *CURTIME* ... 214
14.6. Функция *DATE* ... 214
14.7. Функция *DATEDIFF* ... 215
14.8. Функция *DATE_FORMAT* ... 215
14.9. Функция *DAY* ... 218
14.10. Функция *DAYNAME* ... 218
14.11. Функция *DAYOFMONTH* ... 219
14.12. Функция *DAYOFWEEK* ... 220
14.13. Функция *DAYOFYEAR* ... 220
14.14. Функция *EXTRACT* ... 220
14.15. Функция *FROM_DAYS* ... 221
14.16. Функция *FROM_UNIXTIME* ... 222
14.17. Функция *GET_FORMAT* ... 223
14.18. Функция *HOUR* ... 225
14.19. Функция *LAST_DAY* ... 225
14.20. Функция *MAKEDATE* ... 226
14.21. Функция *MAKETIME* ... 227
14.22. Функция *MICROSECOND* ... 227
14.23. Функция *MINUTE* ... 228
14.24. Функция *MONTH* ... 228

14.25. Функция *MONTHNAME* ... 228
14.26. Функция *NOW* ... 229
14.27. Функция *PERIOD_ADD* ... 230
14.28. Функция *PERIOD_DIFF* ... 230
14.29. Функция *QUARTER* ... 231
14.30. Функция *SECOND* ... 231
14.31. Функция *SEC_TO_TIME* ... 231
14.32. Функция *STR_TO_DATE* ... 232
14.33. Функция *SUBDATE* ... 232
14.34. Функция *SUBTIME* ... 233
14.35. Функция *TIME* ... 234
14.36. Функция *TIMEDIFF* ... 234
14.37. Функция *TIMESTAMP* ... 235
14.38. Функция *TIMESTAMPADD* ... 235
14.39. Функция *TIMESTAMPDIFF* ... 236
14.40. Функция *TIME_FORMAT* ... 237
14.41. Функция *TIME_TO_SEC* ... 237
14.42. Функция *TO_DAYS* ... 238
14.43. Функция *UNIX_TIMESTAMP* ... 239
14.44. Функция *UTC_DATE* ... 239
14.45. Функция *UTC_TIME* ... 240
14.46. Функция *UTC_TIMESTAMP* ... 240
14.47. Функция *WEEK* ... 241
14.48. Функция *WEEKDAY* ... 242
14.49. Функция *WEEKOFYEAR* ... 242
14.50. Функция *YEAR* ... 243
14.51. Функция *YEARWEEK* ... 243

Глава 15. Строковые функции ... 245

15.1. Функция *ASCII* ... 245
15.2. Функция *BIN* ... 246
15.3. Функция *BIT_LENGTH* ... 246
15.4. Функция *CHAR* ... 246
15.5. Функция *CHAR_LENGTH* ... 247
15.6. Функция *CHARSET* ... 248
15.7. Функция *COLLATION* ... 249
15.8. Функция *COMPRESS* ... 249
15.9. Функция *CONCAT* ... 251
15.10. Функция *CONCAT_WS* ... 252
15.11. Функция *CONV* ... 252
15.12. Функция *ELT* ... 253
15.13. Функция *EXPORT_SET* ... 253
15.14. Функция *FIELD* ... 254
15.15. Функция *FIND_IN_SET* ... 255

15.16. Функция *FORMAT* ... 255
15.17. Функция *HEX* ... 256
15.18. Функция *INSERT* ... 256
15.19. Функция *INSTR* ... 257
15.20. Функция *LEFT* ... 257
15.21. Функция *LENGTH* ... 258
15.22. Функция *LOAD_FILE* ... 258
15.23. Функция *LOCATE* ... 259
15.24. Функция *LOWER* ... 259
15.25. Функция *LPAD* ... 260
15.26. Функция *LTRIM* ... 260
15.27. Функция *MAKE_SET* ... 261
15.28. Функция *MID* ... 261
15.29. Функция *OCT* ... 262
15.30. Функция *ORD* ... 263
15.31. Функция *POSITION* ... 263
15.32. Функция *QUOTE* ... 263
15.33. Функция *REPEAT* ... 264
15.34. Функция *REPLACE* ... 264
15.35. Функция *REVERSE* ... 266
15.36. Функция *RIGHT* ... 266
15.37. Функция *RDAP* ... 267
15.38. Функция *RTRIM* ... 267
15.39. Функция *SOUNDEX* ... 268
15.40. Функция *SPACE* ... 268
15.41. Функция *SUBSTRING* ... 268
15.42. Функция *SUBSTRING_INDEX* ... 269
15.43. Функция *TRIM* ... 270
15.44. Функция *UNCOMPRESS* ... 271
15.45. Функция *UNCOMPRESSED_LENGTH* ... 271
15.46. Функция *UNHEX* ... 272
15.47. Функция *UPPER* ... 272

Глава 16. Безопасность и MySQL ... 275

16.1. Функции *AES_ENCRYPT* и *AES_DECRYPT* ... 275
16.2. Функции *ENCODE* и *DECODE* ... 277
16.3. Функции *DES_ENCRYPT* и *DES_DECRYPT* ... 278
16.4. Функция *ENCRYPT* ... 279
16.5. Функция *MD5* ... 280
16.6. Функция *PASSWORD* ... 280
16.7. Функция *SHA1* ... 281

Глава 17. Поиск и регулярные выражения ... 283

17.1. Оператор *LIKE* ... 283
17.2. Оператор *NOT LIKE* ... 287

17.3. Оператор *SOUND LIKE* 288
17.4. Оператор *RLIKE (REGEXP)* 288
17.5. Оператор *NOT RLIKE* 299
17.6. Функция *STRCMP* 300

Глава 18. Полнотекстовый поиск 301

18.1. Индекс *FULLTEXT* 301
18.2. Конструкция *MATCH (...) AGAINST (...)* 303
18.3. Логический режим 307

Глава 19. Функции, применяемые вместе с конструкцией *GROUP BY* 313

19.1. Функция *AVG()* 313
19.2. Функция *BIT_AND()* 316
19.3. Функция *BIT_OR()* 317
19.4. Функция *BIT_XOR()* 317
19.5. Функция *COUNT()* 318
19.6. Функция *GROUP_CONCAT()* 321
19.7. Функция *MIN()* 322
19.8. Функция *MAX()* 323
19.9. Функция *STD()* 324
19.10. Функция *STDDEV_SAMP()* 325
19.11. Функция *SUM()* 326
19.12. Функция *VAR_POP()* 327
19.13. Функция *VAR_SAMP()* 327
19.14. Конструкция *WITH ROLLUP* 327

Глава 20. Разные функции 329

20.1. Функции управления потоком выполнения 329
20.1.1. Функция *CASE* 329
20.1.2. Функция *IF()* 330
20.1.3. Функция *IFNULL()* 331
20.1.4. Функция *NULLIF()* 333
20.2. Информационные функции 333
20.2.1. Функция *BENCHMARK()* 333
20.2.2. Функция *CONNECTION_ID()* 334
20.2.3. Функция *CURRENT_USER()* 334
20.2.4. Функция *DATABASE()* 335
20.2.5. Функция *FOUND_ROWS()* 335
20.2.6. Функция *LAST_INSERT_ID()* 336
20.2.7. Функция *USER()* 338
20.2.8. Функция *VERSION()* 339
20.3. Разные функции 339
20.3.1. Функция *DEFAULT()* 339

20.3.2. Функция *GET_LOCK()* ... 340
20.3.3. Функция *INET_ATON()* ... 341
20.3.4. Функция *INET_NTOA()* ... 342
20.3.5. Функция *IS_FREE_LOCK()* ... 342
20.3.6. Функция *IS_USED_LOCK()* ... 343
20.3.7. Функция *RELEASE_LOCK()* ... 343
20.3.8. Функция *UUID()* ... 343

Глава 21. Переменные и временные таблицы ... 345

21.1. Переменные SQL ... 345
21.2. Временные таблицы ... 348

Глава 22. Многотабличные запросы ... 353

22.1. Перекрестное объединение таблиц ... 353
22.2. Объединение таблиц при помощи *JOIN* ... 363
22.3. Обновление нескольких таблиц ... 369
22.4. Удаление из нескольких таблиц ... 372

Глава 23. Вложенные запросы ... 375

23.1. Вложенный запрос как скалярный операнд ... 377
23.2. Вложенные запросы, возвращающие несколько строк ... 380
23.2.1. Ключевое слово *IN* ... 381
23.2.2. Ключевое слово *ANY (SOME)* ... 382
23.2.3. Ключевое слово *ALL* ... 384
23.3. Проверка на существование ... 386
23.4. Коррелированные запросы ... 388
23.5. Вложенные запросы, возвращающие несколько столбцов ... 389
23.6. Подзапросы в конструкции *FROM* ... 391
23.7. Вложенные запросы в операторе *CREATE TABLE* ... 393
23.8. Вложенные запросы в операторе *INSERT* ... 396
23.9. Ссылочная целостность ... 398

Глава 24. Транзакции и блокировки ... 407

24.1. Транзакции ... 407
24.2. Блокировка таблиц ... 412

Глава 25. Управление учетными записями пользователей ... 417

25.1. Учетные записи СУБД MySQL ... 417
25.2. Оператор *CREATE USER* ... 420
25.3. Оператор *DROP USER* ... 422
25.4. Оператор *RENAME USER* ... 424
25.5. Оператор *GRANT* ... 425
25.6. Оператор *REVOKE* ... 434

Глава 26. Предотвращение катастроф и восстановление 437

26.1. Оператор *CHECK TABLE* 437
26.2. Оператор *ANALYZE TABLE* 440
26.3. Оператор *CHECKSUM TABLE* 440
26.4. Оператор *OPTIMIZE TABLE* 443
26.5. Оператор *REPAIR TABLE* 443
26.6. Оператор *BACKUP TABLE* 445
26.7. Оператор *RESTORE TABLE* 446

Глава 27. Административные команды 449

ЧАСТЬ III 451

Глава 28. Хранимые процедуры 453

28.1. Хранимые процедуры и привилегии 454
28.2. Создание хранимой процедуры 454
28.2.1. Тело процедуры 456
28.2.2. Параметры процедуры 459
28.2.3. Работа с таблицами базы данных 462
28.2.4. Хранимые функции 469
28.3. Группа характеристик хранимых процедур 471
28.4. Операторы управления потоком данных 473
28.4.1. Оператор *IF...THEN...ELSE* 474
28.4.2. Оператор *CASE* 477
28.4.3. Оператор *WHILE* 479
28.4.4. Оператор *REPEAT* 484
28.4.5. Оператор *LOOP* 485
28.4.6. Оператор *GOTO* 486
28.5. Метаданные 488
28.5.1. Оператор *SHOW PROCEDURE STATUS* 488
28.5.2. Оператор *SHOW CREATE* 490
28.5.3. Извлечение информации из таблицы *mysql.proc* 491
28.6. Удаление хранимых процедур 493
28.7. Редактирование хранимых процедур 494
28.8. Обработчики ошибок 495
28.9. Курсоры 500

Глава 29. Триггеры 505

29.1. Оператор *CREATE TRIGGER* 505
29.2. Оператор *DROP TRIGGER* 509

Глава 30. Представления 511

30.1. Создание представлений 512
30.2. Удаление представлений 524

30.3. Редактирование представлений 525
30.4. Оператор *SHOW CREATE VIEW* 525

Глава 31. Информационная схема 527

Заключение 529

Приложение. Описание компакт-диска 531

Предметный указатель 533

Введение

Для кого и о чем эта книга

Книга посвящена описанию СУБД MySQL и особенностям ее диалекта языка SQL. Популярность реляционных баз данных неуклонно возрастает в последние десятилетия. СУБД MySQL представляет собой СУБД с открытым кодом, разработкой которой занимается корпорация MySQL AB. СУБД MySQL успешно работает под управлением свыше 20 различных операционных систем (Windows, Linux, OS/2, Mac OS X, FreeBSD, Solaris, SunOS, OpenBSD, NetBSD, AIX, DEC Unix, Tru64 Unix и т. д.), а также предоставляет интерфейс для взаимодействия с множеством языков программирования: C, C++, Eiffel, Java, Perl, PHP, Python, Ruby и Tcl.

Принимая во внимание факт, что это одна из немногих СУБД с открытым кодом, и лицензионное соглашение, по которому она распространяется, не требует денежных отчислений (исключением является случай, когда MySQL входит в состав коммерческого продукта), не удивительно, что она получила широкое распространение в Российской Федерации. В частности, MySQL является стандартом де-факто у российских хост-провайдеров.

На сегодняшний день MySQL, наряду с СУБД Oracle, является одной из самых быстрых баз данных. В версии MySQL 5.0 была достигнута функциональность, доступная ранее только в коммерческих базах данных. Так, введены долгожданные вложенные запросы, хранимые процедуры, курсоры, обработчики ошибок, триггеры, представления и множество новых внутренних функций.

Книга будет полезна как начинающим разработчикам приложений с применением баз данных, которые хотят разобраться во всех тонкостях работы с MySQL, так и профессионалам, использовавшим в своей работе более ранние версии MySQL (3 и 4) и желающим познакомиться с нововведениями MySQL 5.0. Изложение в книге ведется таким образом, чтобы у читателя, незнакомого с языком запросов SQL, материал книги не вызвал затруднений. На протяжении всей книги нововведения версий 4.1 и 5.0 указываются явно в замечании, поэтому книга может быть полезна разработчикам, использующим более старые версии СУБД MySQL, но планирующим со временем использовать новые версии.

Первая часть книги посвящена основам SQL и взаимодействию с сервером MySQL. В основном она ориентирована на начинающих разработчиков. Во второй части рассматриваются более сложные вопросы, такие как внутренние функции, полнотекстовый поиск, регулярные выражения в MySQL, временные таблицы и переменные, вложенные и многотабличные запросы, транзакции и блокировки, учетные записи и администрирование. Третья часть книги полностью посвящена нововведениям 5-й версии СУБД MySQL — рассматриваются хранимые процедуры и функции, обработчики ошибок, курсоры, триггеры, представления и информационная схема.

На протяжении всей книги используется учебная база данных электронного магазина, которую можно найти в каталоге base прилагаемого к книге компакт-диска, а также загрузить по адресу **http://www.softtime.ru/sql/base.zip**. Помимо этого, компакт-диск содержит дистрибутивы СУБД MySQL версий 4.0, 4.1 и 5.0 для операционных систем Windows и Linux. Кроме того, на прилагаемом компакт-диске вы сможете найти документацию на русском и английском языках и клиенты с графическим интерфейсом для обращения к серверу MySQL.

По всем вопросам, которые могут возникнуть по мере чтения материала книги, вы можете обращаться на форум, расположенный на Web-сайте IT-студии SoftTime, сотрудниками которой являются авторы книги (**http://www.softtime.ru/forum/index.php?id_forum=3**). Авторы присутствуют на форуме каждый день и в меру своей компетенции постараются ответить на ваши вопросы.

Благодарности

Авторы выражают признательность сотрудникам издательства "БХВ-Петербург", благодаря которым наша рукопись увидела свет.

ЧАСТЬ I

Глава 1. История развития баз данных. Понятие реляционной базы данных

Глава 2. Установка MySQL 5

Глава 3. Работа с утилитами MySQL

Глава 4. Создание баз данных и таблиц. Типы данных

Глава 5. Индексы

Глава 6. Добавление данных

Глава 7. Выборка данных

Глава 8. Удаление данных

Глава 9. Обновление данных

Глава 10. Редактирование структуры таблиц

ГЛАВА 1

История развития баз данных. Понятие реляционной базы данных

Компьютерные технологии постоянно эволюционируют. Это связано с тем, что программы, которые от них требуются, становятся все более и более изощренными, а чем сложнее программа, тем больше кода необходимо для ее создания. За время существования этой отрасли программисты сменили машинные коды на ассемблер, ассемблер на языки высокого уровня. Каждый новый виток в технологии программирования обеспечивал одну простую вещь — программист мог создавать более сложные программы, содержащие большее число строк. Когда мощности языков высокого уровня стало не хватать и программы достигли уровня десятков тысяч строк, появилось структурное программирование, позволившее увеличить объем программы до 100 000 строк. Очень скоро и данного предела оказалось недостаточно, и был предложен объектно-ориентированный подход, который позволил создавать еще более сложные и объемные программы.

Программы — это процессы, которые манипулируют данными. Как же происходила эволюция данных? В далеких 60-х годах прошлого столетия программисты использовали машинные коды и файлы. В настоящий момент ни прикладное, ни даже системное программное обеспечение в машинных кодах не пишется, однако файлы используются повсеместно. Значит ли это, что эволюции не было? Ответу на этот вопрос посвящена данная глава.

1.1. История развития СУБД. Реляционные базы данных

Задача длительного хранения и обработки информации появилась практически сразу с созданием первых компьютеров. Для решения этой задачи в конце 60-х годов XX века были разработаны специализированные программы, получившие название *систем управления базами данных* (СУБД). СУБД проделали длительный путь эволюции от системы управления файлами через иерархические и сетевые базы данных к реляционным моделям. В конце 80-х годов XX века доминирующей стала *система управления реляционными базами данных* (СУРБД). С этого времени такие СУБД

стали стандартом де-факто. Каждая СУБД имела свой собственный язык запросов, и для того, чтобы унифицировать работу с ними, был разработан *структурированный язык запросов* SQL (Structured Query Language), который представляет собой язык управления реляционными базами данных. Цель данного раздела — рассмотреть реляционную модель данных, сравнив ее с более ранними СУБД: иерархическими и сетевыми.

Замечание

Взаимодействие с базой данных происходит при помощи *Системы Управления Базой Данных* (СУБД), которая расшифровывает запросы и производит операции с информацией в базе данных. Поэтому более правильно было бы говорить о запросе к СУБД и о взаимодействии приложения с СУБД. Но т. к. это несколько усложняет восприятие, далее везде мы будем говорить "база данных", подразумевая при этом СУБД.

Существуют следующие разновидности баз данных:

- иерархические;
- сетевые;
- реляционные;
- объектно-ориентированные;
- гибридные.

1.1.1. Иерархические базы данных

Первыми базами данных были иерархические. Иерархическая база данных основана на древовидной структуре хранения информации и в этом смысле напоминает файловую систему компьютера. Наиболее известным и распространенным примером иерархической базы данных является Information Management System (IMS) фирмы IBM, первая версия которой появилась в 1968 г. С точки зрения организации хранения информации иерархическая база данных состоит из упорядоченного набора деревьев одного типа, т. е. каждая запись в базе данных реализована виде отношений "предок-потомок". На рис. 1.1 показана схема организации такой иерархической базы данных.

Рис. 1.1. Схема организации иерархической базы данных на примере войсковой части

В этой схеме Войсковая часть является предком для объектов Командир и Бойцы, а объекты Командир и Бойцы — это потомки объекта Войсковая часть, и между этими типами записи поддерживаются связи.

Реальная иерархическая база данных войсковой части может выглядеть, к примеру, так, как показано на рис. 1.2.

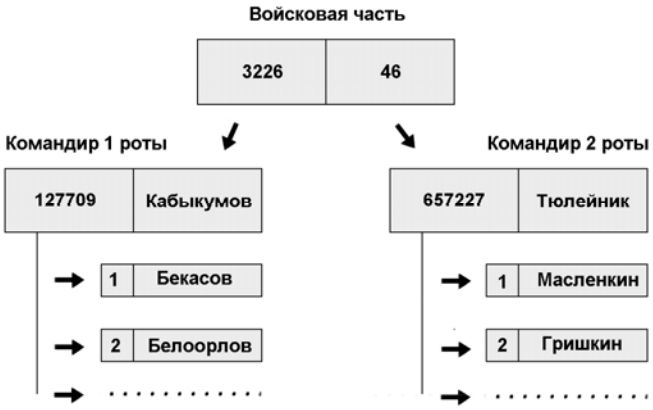

Рис. 1.2. Иерархическая база данных

Основной недостаток иерархической структуры базы данных — невозможность реализовать отношения "многие-ко-многим", а также ситуации, когда запись имеет несколько предков.

Иерархические базы данных наиболее пригодны для моделирования структур, являющихся иерархическими по своей природе. Самым простым примером такой структуры является воинское подразделение, имеющее очевидную естественную иерархию. Однако существует очень много структур, которые невозможно свести к простой иерархии, и в этом случае применение иерархической базы данных становится невозможным. Примером такой структуры является всем известное генеалогическое древо, которое, казалось бы, как нельзя лучше подходит для описания его в терминах иерархической базы данных, однако это сделать невозможно, поскольку каждый потомок имеет двух родителей.

Примечание

В настоящее время не разрабатываются СУБД, поддерживающие только иерархическую модель организации хранения данных. В лучшем случае проводится совмещение иерархической и сетевой модели *(см. далее)*, которые допускают связывание древовидных структур между собой и установление связей внутри них. Такие СУБД называются сетевыми даталогическими СУБД.

1.1.2. Сетевые базы данных

Иерархические базы данных в силу большого количества недостатков просуществовали недолго и были заменены сетевыми базами данных, являющимися, по сути,

расширением иерархических. То есть если в иерархических базах данных структура "запись-потомок" должна иметь только одного предка, то в сетевой базе данных потомок может иметь любое количество предков.

Примечание

Термин "сетевая" в данном случае не подразумевает никакого намека на локальные сети. Этим словом обозначается модель связей в базе данных, когда каждая запись может находиться в отношениях "многие-ко-многим" с другими записями, что делает графическую модель базы похожей на рыбацкую сеть (рис. 1.3).

Схема сетевой базы данных на примере организации сети железнодорожных путей показана на рис. 1.3. Как видно из этого рисунка, каждая крупная железнодорожная станция может иметь связи (пути сообщения) с несколькими другими станциями (связь "многие-ко-многим"). Примерно таким же образом выглядит графическая схема любой сетевой базы данных.

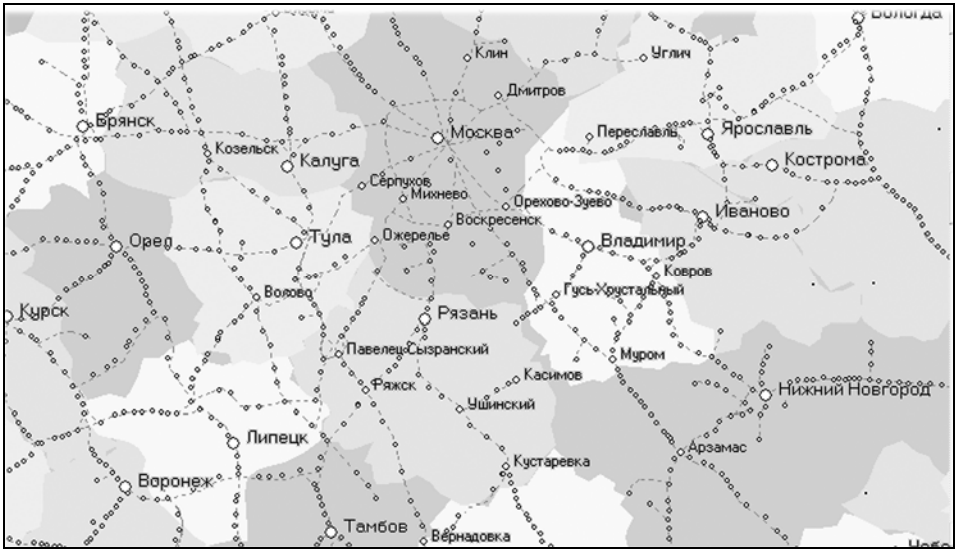

Рис. 1.3. Схема организации сетевой базы данных

Основным недостатком сетевых баз данных является сложность разработки серьезных приложений.

1.1.3. Реляционные базы данных

Настоящий прорыв в развитии теории баз данных произошел тогда, когда возросшая мощность компьютеров позволила реализовать реляционную модель данных. Теория реляционных баз данных была разработана доктором Коддом из компании IBM в 1970 году. Одной из задач реляционной модели была попытка упростить структуру базы данных. В ней отсутствовали явные указатели на предков и потомков, а все дан-

ные были представлены в виде простых таблиц, разбитых на строки и столбцы, на пересечении которых расположены данные.

Запросы к таким таблицам возвращают новые таблицы, которые сами могут становиться предметом дальнейших запросов. Даже если возвращается одно значение, это все равно считается таблицей, состоящей из одного столбца и одной строки. То, что SQL-запрос возвращает таблицу, очень важно, это означает, что результаты запроса можно записать обратно в базу данных в виде таблицы, а результаты двух или более запросов, имеющие одинаковую структуру, можно объединить в одну таблицу.

Каждая база данных может включать несколько таблиц, которые, как правило, связаны друг с другом, откуда и произошло название "реляционные".

ряд ⇩

строка ⇨

id_catalog	name	description
1	Мониторы	Информация о мониторах
2	Материнские платы	Информация о материнских платах
3	Жесткие диски	Информация о жестких дисках
4	Разное	Информация о различных комплектующих

Рис. 1.4. Таблица реляционной базы данных

На рис. 1.4 приведен пример таблицы `catalog` базы данных электронного магазина компьютерных комплектующих, которые распределены по разделам. Каждая строка этой таблицы представляет собой один вид товарных позиций, для описания которых используется поле `id_catalog` — уникальный номер раздела, `name` — название раздела и `description` — его описание.

Таким образом, можно дать следующее простое определение реляционной базы данных.

Определение

Реляционной базой данных называется такая база данных, в которой все данные организованы в виде таблиц, а все операции над этими данными сводятся к операциям над таблицами.

1.1.4. Объектно-ориентированные и гибридные базы данных

В *объектно-ориентированных* базах данных данные хранятся в виде объектов. С такими базами данных удобно работать, применяя объектно-ориентированный подход. Однако на сегодняшний день такие базы данных еще не достигли популярности реля-

ционных, поскольку пока значительно уступают им в производительности. *Гибридные (объектно-реляционные)* СУБД совмещают в себе возможности реляционных и объектно-ориентированных баз данных. В последнее время такие базы данных часто называют объектно-реляционными базами данных. Появление гибридных СУБД связано с тем, что объектные СУБД, пока что не получившие широкого применения, несомненно, будут развиваться в будущем. Поэтому разработчики реляционных баз данных включают в свои базы средства работы с объектными типами данных. Такие базы данных называются объектно-реляционными или гибридными. Примером объектно-реляционной СУБД является Oracle, бывшая ранее чисто реляционной базой данных. Возможность хранения и обработки объектов поддерживается в Oracle, начиная с восьмой версии.

1.2. Особенности реляционных баз данных

Можно кратко сформулировать особенности реляционной базы данных.

- Данные хранятся в таблицах, состоящих из столбцов и строк.
- На пересечении каждого столбца и строки находится только одно значение.
- У каждого столбца есть свое имя, которое служит его названием, и все значения в одном столбце имеют один тип. Например, в столбце `id_products` все значения имеют целочисленный тип, а в строке `name` — текстовый.
- Столбцы располагаются в определенном порядке, который задается при создании таблицы, в отличие от строк, которые располагаются в произвольном порядке. В таблице может не быть ни одной строчки, но обязательно должен быть хотя бы один столбец.
- Запросы к базе данных возвращают результат в виде таблиц, которые тоже могут выступать как объект запросов.

Более строгое определение реляционных баз данных основано на работе Кодда. В ней сформулированы 12 правил, которым должна соответствовать реляционная база данных. Сейчас эти правила так и называют: *правила Кодда* — и они являются определением понятия "реляционная база данных".

Примечание

Правила Кодда и их разъяснение приведены в конце данной главы. Они не являются догматическими — реляционные базы данных постоянно развиваются и совершенствуются, на рынке появляются новые продукты, которые поддерживают не все требования к реляционным базам данных. Например, MySQL долго не соответствовала многим стандартам SQL. Исторически сложилось так, что последним стандартам SQL не соответствует ни одна из баз данных.

1.2.1. Первичные ключи

Строки в реляционной базе данных неупорядочены. То есть в таблице нет "первой", "последней", "тридцать шестой" и "сорок третьей" строки, а есть просто неупорядо-

ченный набор строк. Возникает вопрос: "Каким же образом выбирать в таблице конкретную строку?" Для этого в правильно спроектированной базе данных для каждой таблицы создается один или несколько столбцов, значения которых во всех строках различны. Такой столбец называется *первичным ключом* таблицы. Первичный ключ обычно сокращенно обозначают как PK (primary key). Никакие из двух записей таблицы не могут иметь одинаковых значений первичного ключа — т. е., благодаря наличию первичных ключей, каждая строка таблицы обладает своим уникальным идентификатором.

Примечание

В математике таблица, все строки которой отличаются друг от друга, называется *отношением*, — по-английски relation. Именно этому английскому термину реляционные базы данных и обязаны своим названием.

По способу задания первичных ключей различают *логические* (естественные) ключи и *суррогатные* (искусственные).

Для логического задания первичного ключа нужно выбрать в базе данных то, что естественным образом определяет запись. Примером такого ключа является номер паспорта в базе данных о паспортных данных жителей. К примеру, в таблице базы данных, содержащей паспортные данные, уникальный индекс можно создать для поля "номер паспорта", поскольку каждый такой номер является единственным в своем роде. А вот дата рождения уже не уникальна, поэтому индекс по полю "Дата рождения" не может быть уникальным.

Если подходящих примеров для естественного задания первичного ключа не находится, пользуются суррогатным ключом. Суррогатный ключ представляет собой дополнительное поле в базе данных, предназначенное для обеспечения записей первичным ключом. Таким ключом, к примеру, является столбец `id_products`, базы данных товаров компьютерного магазина (см. рис. 1.4).

Совет

Однако даже если в базе данных содержится естественный первичный ключ, лучше использовать суррогатные ключи, поскольку их применение позволяет абстрагировать первичный ключ от реальных данных. В этом случае облегчается работа с таблицами, поскольку суррогатные ключи не связаны ни с какими фактическими данными этой таблицы.

Как уже упоминалось, в реляционных базах данных практически всегда разные таблицы логически связаны друг с другом. Одним из предназначений первичных ключей и является однозначная организация такой связи.

Например, рассмотрим базу данных какого-либо форума. В любом форуме обязательно присутствуют темы, в которых пользователи оставляют сообщения. Таким образом, в данном приближении мы имеем две таблицы: таблицу с названиями тем `themes` и таблицу `posts` с названиями сообщений. Еще раз обратим внимание на то, что строки в таблицах неупорядочены: т. е. у нас в таблице тем вперемешку содержатся строки с названиями тем, а в таблице сообщений — тексты сообщений. А как мы знаем, сообщения должны относиться к строго определенным темам. Следова-

тельно, встает задача установления взаимно однозначного соответствия между названиями тем и сообщениями в этих темах. Такое соответствие также устанавливается с помощью первичных ключей (рис. 1.5).

Рис. 1.5. Связь между таблицами `themes` и `posts`

Первичным ключом таблицы `themes` является `id_theme`, а таблицы `posts` — `id_post`. Обратите внимание, что поле `id_theme` присутствует и в таблице `posts`. Каждое значение этого поля в таблице `posts` является *внешним ключом* (в данном случае это внешний ключ для первичного ключа таблицы `themes`). Внешний ключ сокращенно обозначают как FK (foreign key).

Примечание

Внешний ключ также часто называют *вторичным ключом*.

Как видно из рис. 1.5, внешний ключ ссылается на первичный ключ таблицы `themes`, устанавливая однозначную логическую связь между записями таблиц `themes` и `posts`. Иначе говоря, если внешний ключ для записи (сообщения) с PK=1 в таблице `posts` имеет значение внешнего ключа равное 1, то это значит, что данное сообщение относится к теме с PK=1 таблицы `themes`.

1.2.2. Нормализация базы данных

Определение

Нормализацией схемы базы данных называется процедура, производимая над базой данных с целью удаления в ней избыточности.

Для того чтобы лучше уяснить приведенное определение нормализации, рассмотрим следующий пример. На рис. 1.6 показана таблица, в которой указаны фамилии сотрудников и их профессии.

Должность	Сотрудник
Инженер	Кузнецов
Инженер	Скворцов
Рабочий	Гришкин
Рабочий	Бекасов
Рабочий	Кабыкумов

Рис. 1.6. Пример избыточности в таблице базы данных

Эта таблица избыточна — для каждого из сотрудников повторяются одинаковые названия профессий. То есть схема такой базы данных не нормализована. Для небольшой базы данных это не критично, но в больших по объему базах это скажется на размере базы и, в конечном счете, на скорости доступа. Для нормализации необходимо разбить эту таблицу на две — профессий и фамилий сотрудников, как это показано на рис. 1.7.

Таблица должностей

Должность	Первичный ключ
"Инженер"	1
"Рабочий"	2

Таблица сотрудников

Должность (внешний ключ)	Сотрудник
1	Кузнецов
1	Скворцов
2	Гришкин
2	Бекасов
2	Кабыкумов

Рис. 1.7. Нормализованная схема соответствия профессий и сотрудников

Теперь каждый тип профессии обозначен уникальным числом, и строка в базе данных присутствует только в единственном экземпляре: в таблице профессий. Размер поля "профессия" уменьшился, т. к. строка занимает больше памяти, чем число.

Очевидно, что нормализация несет в себе немало преимуществ: в нормализованной базе данных уменьшается вероятность возникновения ошибок, она занимает меньше места на жестком диске и т. д.

Замечание

Хотя в теории баз данных и говорится о том, что схема базы данных должна быть полностью нормализована, в реальности при работе с полностью нормализованными базами данных необходимо применять весьма сложные SQL-запросы, что при-

водит к обратному эффекту — замедлению работы базы данных. Поэтому иногда для упрощения запросов даже прибегают к обратной процедуре — денормализации.

1.2.3. Правила Кодда

1. *Правило информации.* Вся информация в базе данных должна быть представлена только на логическом уровне и только в виде значений, содержащихся в таблицах. Это правило, по сути, является основным определением понятия "реляционная база данных".

2. *Правило гарантированного доступа.* Логический доступ к каждому элементу данных должен обеспечиваться использованием комбинации имени таблицы, первичного ключа и имени столбца. Данное правило определяет роль первичных ключей при поиске информации в базе данных и говорит о том, каким образом должен происходить этот поиск: по имени таблицы ищется нужная таблица, по имени столбца — нужный столбец, а первичный ключ позволяет найти ту строку, в которой содержится нужный элемент данных.

3. *Правило обработки отсутствующих значений.* Должна быть реализована поддержка отсутствующих значений, которые не являются ни строками нулевой длины, ни строками пробельных символов, ни нулем или каким-либо другим числом, а используются для представления отсутствующих данных, независимо от типа этих данных. Это правило говорит о том, что отсутствующие значения должны представляться значениями `NULL` (значения `NULL` описаны в *главе 4*).

4. *Правило динамического каталога.* Описание базы данных должно быть представлено в том же виде, что и описание самих данных. Это правило говорит о том, что реляционная базы данных должна сама себя описывать, т. е. содержать набор системных таблиц, описывающих саму структуру базы данных.

5. *Правило исчерпывающего подъязыка базы данных.* Реляционная база данных может поддерживать различные языки и способы взаимодействия с пользователем. Однако должен существовать по крайней мере один язык, инструкциями которого в соответствии с четко определенным синтаксисом можно описать следующие элементы:

   - определение данных;
   - обработку данных;
   - определение представлений;
   - идентификацию прав доступа;
   - условия целостности данных;
   - границы транзакций.

6. *Правило обновления представлений.* Все представления, которые теоретически возможно обновить, должны быть доступны для обновления.

Примечание

Представления, по сути, являются виртуальными таблицами, позволяющими показывать пользователям фрагменты структуры базы данных. В MySQL работа с представлениями возможна, только начиная с версии 5.0, и рассматривается в *главе 30*.

7. *Правило добавления, обновления и удаления.* Возможность работать с отношением как с одним операндом должна существовать не только при выборке данных, но и при операциях добавления, удаления и обновления данных. Это правило говорит о том, что все перечисленные операции можно было производить не только над одной строкой, а и над множествами строк.
8. *Правило физической независимости данных.* Прикладные программы для работы с данными должны оставаться нетронутыми при любых изменениях способов хранения данных или методов доступа к ним. Это правило, как и следующее, говорит о необходимости отделения пользователей и прикладных программ от низкоуровневой организации базы данных, т. е. конкретные способы реализации хранения данных и методы доступа к ним не должны влиять на возможность пользователя осуществлять работу с этими данными.
9. *Правило логической независимости данных.* Прикладные программы для работы с данными должны оставаться нетронутыми при внесении в таблицы базы данных любых изменений, которые позволяют сохранить нетронутыми содержащиеся в этих таблицах данные.
10. *Правило независимости условий целостности баз данных.* Должна существовать возможность определять условия целостности, специфичные для каждой конкретной базы данных, на подъязыке этой базы данных, и хранить их в динамическом каталоге, а не в прикладной программе. Это правило говорит о том, что язык управления базой данных должен поддерживать наложение ограничений и условий как на сами данные, так и на способы работы с ними.
11. *Правило независимости размещения.* База данных не должна зависеть от особенностей системы, на которой она расположена. Суть данного правила заключается в том, что язык управления базой данных должен обеспечивать возможность работы с распределенными данными.
12. Если в реляционной базе данных есть низкоуровневый язык, то должна отсутствовать возможность использования его для обхода правил и условий целостности данных, сформулированных на языке высокого уровня. Это правило запрещает иные средства работы с базой данных, кроме ее внутреннего языка.

Примечание

Низкоуровневым называется язык, обрабатывающий одну запись за один раз, а высокоуровневым, соответственно, язык, обрабатывающий несколько записей за один раз.

1.3. СУБД и сети

Развитие компьютерных сетей, а тем более рост популярности Интернета, сыграли серьезную роль в развитии СУБД. Если раньше и приложение, и СУБД находились на

одном центральном главном компьютере (мэйнфрейме), то теперь они отделены друг от друга: клиентская часть выполняется на рабочих машинах, а СУБД и собственно сама база данных находятся на сервере. В этой ситуации язык SQL играет важнейшую роль, являясь связующим звеном между приложением, выполняющимся на клиентских машинах, и находящейся на сервере СУБД. Рассмотрим основные этапы эволюции СУБД, связанной с развитием компьютерных сетей.

1.3.1. Централизованная архитектура

При централизованной архитектуре и приложение, и СУБД, и сама база данных размещаются на одном центральном мэйнфрейме[1], пользователи (иногда до сотни на один мэйнфрейм) подключались к нему посредством терминалов, как это показано на рис. 1.8. Сам терминал представлял собой клавиатуру, монитор и сетевую карту, посредством которой происходил обмен данных терминала с мэйнфреймом. Роль приложения в данном случае состоит в принятии вводимых данных с пользовательского терминала (клавиатура на рисунке) и передаче их мэйнфрейму, который передавал их на обработку СУБД, полученный от СУБД ответ отправлялся на монитор терминала.

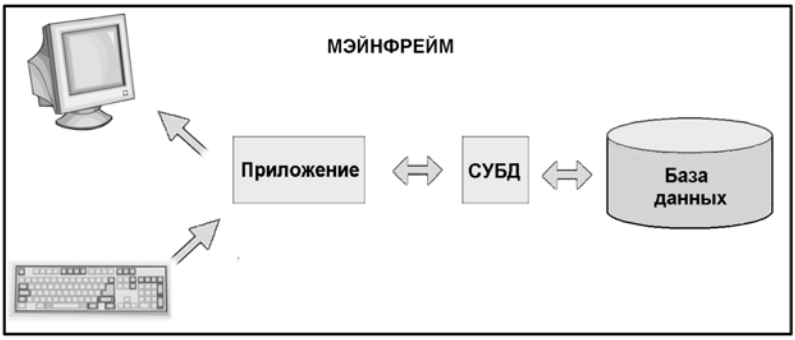

Рис. 1.8. Организация работы СУБД в централизованной архитектуре

Подчеркнем еще раз: основной признак централизованной архитектуры в том, что и приложение, и СУБД работают на одном компьютере. На этом же компьютере находятся и сами данные (база данных).

1.3.2. Архитектура "клиент-сервер"

В клиент-серверной архитектуре персональные компьютеры (клиенты) объединены в локальную сеть, в этой же сети находится и сервер баз данных, на котором содержатся общие для всех клиентов базы данные и СУБД. То есть отличие данной архитектуры от централизованной состоит в том, что СУБД и база данных (находящиеся на сервере) отделены от многочисленных приложений (выполняющихся на клиентских машинах). Схема организации архитектуры "клиент-сервер" изображена на рис. 1.9.

[1] От англ. mainframe — базовая универсальная вычислительная машина. — *Ред.*

Замечание

Следует понимать, что клиенты и серверы СУБД — это программы, а не машины, на которых они выполняются. Помимо сервера базы данных, на машине может выполняться множество других программ, как серверов, так и клиентов. Сервер представляет собой сложную ресурсоемкую программу, поэтому для него часто выделяют отдельную машину, которую тоже называют сервером. Здесь и далее под сервером мы будем понимать программную часть, а не аппаратную.

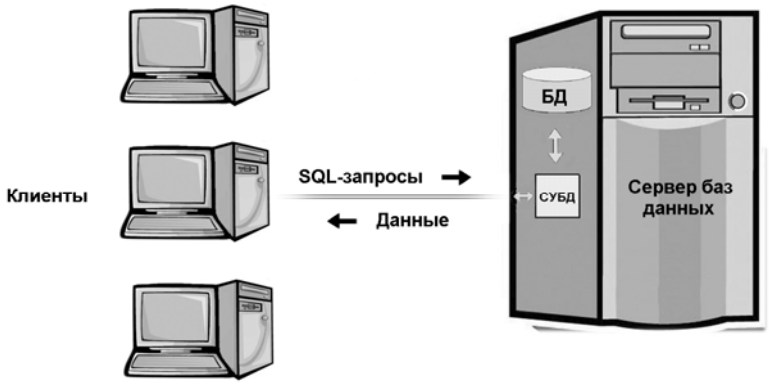

Рис. 1.9. Место СУБД в архитектуре "клиент-сервер"

О клиентах и серверах

В связи с тем, что очень часто возникает путаница, что называть клиентом, а что сервером, остановимся на этом вопросе подробнее. Попросту говоря, клиенты — это те, кто обращается (посылают запросы), а сервер — это тот, кто отвечает на эти запросы. Сервер постоянно прослушивает, не поступили ли запросы от клиентов, и если поступили — отвечает на эти запросы. Для прослушивания предназначены один или несколько портов, к которым и обращаются клиентские приложения. Жизненным примером такой организации может служить система "продавец—очередь из покупателей". Продавец за прилавком в данном случае является сервером, обслуживающим покупателей-клиентов. Точно так же, как и компьютер-сервер, продавец-сервер для того, чтобы обслужить покупателя-клиента, должен получить от него запрос, к примеру "Взвесьте мне килограмм яблок". Нужно также понимать, что сервер может одновременно и сам быть клиентом. В том случае, если он тоже обращается к другому серверу с каким-либо запросом. В этом случае наш первоначальный сервер по отношению к этому "другому серверу" уже будет клиентом, оставаясь при этом сервером для тех клиентов, которых он обслуживает. Продолжая аналогию с продавцами и покупателями, продавец-сервер тоже может стать клиентом. Например, когда сам пойдет в магазин и встанет в очередь за каким-то товаром, т. е. станет покупателем-клиентом.

1.3.2.1. Два признака клиент-серверной архитектуры

Во-первых, это то, что приложения и СУБД выполняются на разных компьютерах. Этим признаком клиент-серверная архитектура отличается от централизованной ар-

хитектуры. А во-вторых, это то, что клиент и сервер находятся в одной локальной сети. Это достаточно важный признак, о котором часто забывают. Между тем, в том числе и по этому признаку клиент-серверная архитектура отличается от трехуровневой архитектуры Интернета, которая рассмотрена ниже и в которой клиенты и серверы объединены не в локальной, а уже в глобальной сети.

Таким образом, в архитектуре "клиент-сервер" функции работы с пользователем, такие как обработка ввода и отображение данных, выполняются на персональном компьютере (клиенте), а функции работы с данными (выполнение запросов, дисковый ввод-вывод) выполняются сервером баз данных.

1.3.3. Трехуровневая архитектура Интернета

С развитием Интернета архитектура сетевого управления базами данных получила дальнейшее развитие. Ранний Интернет обеспечивал доступ к базам данных напрямую. Через некоторое время появилась среда Web, которая предоставила пользователям Интернета дружественный интерфейс. За формирование этого интерфейса несет ответственность Web-сервер, т. е. на пути между пользователем-клиентом и базой данных появился еще один посредник, который стал выполнять функции клиентской программы. Такой подход позволяет многочисленным пользователям Интернета иметь единственную программу Web-браузер для работы с различными базами данных, будь это интернет-магазин или форум, не прибегая к услугам специфических программ-клиентов.

Замечание

Клиенты-браузеры, в терминологии баз данных, часто называют "тонкими клиентами". В противовес "толстым клиентам", которые ориентированы на конкретную базу данных, тонкие клиенты способны лишь отображать данные в строго определенном формате и отправлять серверу лишь простейшие запросы. ICQ, WebMoney представляют примеры "толстых клиентов", а Internet Explorer, Opera, WAP-браузер сотового телефона являются "тонкими клиентами".

Схема работы трехуровневой архитектуры следующая. Допустим, клиенты какой-либо компании, находясь за тысячи километров от нее, желают ознакомиться со списком товаров, доступных на данный момент. В этом случае они используют браузер для посещения сайта этой компании. Страницу со списком товаров при этом формирует специальный модуль — скрипт, выполняющийся на Web-сервере компании. Для получения нужной информации этот скрипт обращается (посылает SQL-запросы) к СУБД, находящейся на сервере баз данных. Иллюстративно такая схема взаимодействия изображена на рис. 1.10.

Таким образом, в трехуровневой архитектуре Интернета интерфейсом пользователя является Web-браузер или другой "тонкий клиент". Это — клиентский уровень. Браузер взаимодействует с Web-сервером, посылая ему запросы на отображение той или иной Web-страницы. Уровень Web-сервера — прикладной уровень. Web-приложение, выполняющееся на Web-сервере, формирует SQL-запрос к СУБД, которая в свою очередь возвращает необходимые данные из базы данных. СУБД и база данных при

этом размещены на сервере баз данных и представляют собой третий, информационный уровень трехуровневой архитектуры Интернета.

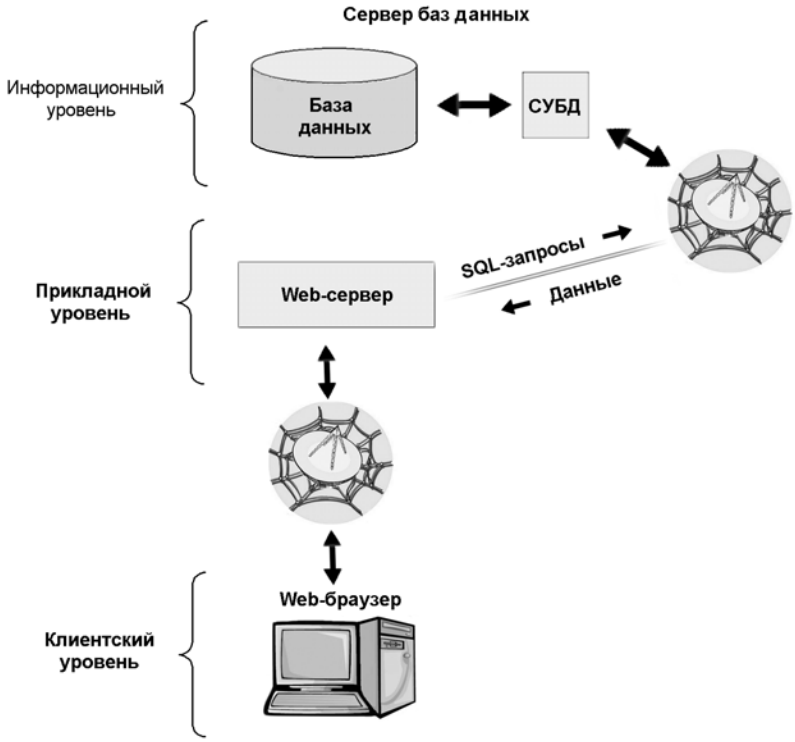

Рис. 1.10. Роль СУБД в трехуровневой архитектуре Интернета

1.4. Как работают базы данных и что такое SQL

Рассмотрим упрощенную схему работы пользователя с базой данных, показанную на рис. 1.11. Согласно этой схеме, в системе имеется база данных, в которой хранится некоторая информация, допустим, информация о работниках. По сути, база данных — это те же файлы, в которых хранится информация. Сами по себе базы данных не представляли бы никакого интереса, если бы не было систем управления базами данных, сокращенно — СУБД. СУБД — это программный комплекс, который управляет базой данных, т. е. берет на себя все низкоуровневые операции по работе с файлами, благодаря чему программисту при работе с базой данных нужно оперировать лишь логическими понятиями при помощи языка программирования, предназначенного для организации взаимодействия пользователя с базой данных.

Примечание

Итак, еще раз уточним понятия. База данных — это просто файловое хранилище информации, и не более. А программные продукты типа MySQL, Oracle, Dbase, Informix, PostgreSQL и др. — это системы управления базами данных. Ведь по сути базы данных везде одинаковы — это файлы с записанной в них информацией. Все вышеприведенные программные продукты отличаются друг от друга именно способом организации работы с файловой системой. Однако для краткости эти СУБД часто называют просто базами данных. Так будем поступать и мы. Следует помнить, что когда мы будем говорить "база данных" — речь идет именно о СУБД.

Рис. 1.11. Простейшая схема организации работы с базой данных

Язык программирования, с помощью которого пользователь общается с СУБД (или, как говорят, "осуществляет запросы к базе данных"), называется SQL (Structured Query Language, структурированный язык запросов).

Замечание

После публикации исследований Кодда компанией IBM был инициирован проект System/R по созданию первой реляционной базы данных, результатом которого стало создание минимального прототипа реляционной СУБД. Кроме разработки самой СУБД, проводилась работа над созданием языков запроса к базе данных. Один из этих языков был назван SEQUEL (Structured English Query Language, структурированный английский язык запросов). Позже по юридическим соображениям язык был переименован в SQL.

Таким образом, для того чтобы получить какую-либо информацию из базы данных, необходимо направить базе данных запрос, созданный с использованием SQL, результатом выполнения которого будет таблица с данными (см. рис. 1.11).

Несмотря на то, что SQL называется "языком запросов", в настоящее время этот язык представляет собой нечто большее, чем просто инструмент для создания запросов. Ведь запрос это, по сути, предложение вида "Выбрать из таких-то таблиц такие то данные". К примеру, на рис. 1.11 показан запрос, суть которого состоит в выборке всех фамилий из таблицы работников. С помощью SQL осуществляется реализация всех возможностей, которые представляются пользователям разработчиками СУБД, а именно:

- выборка данных (извлечение из базы данных содержащейся в ней информации);
- организация данных (определение структуры базы данных и установление отношений между ее элементами);
- обработка данных (добавление/изменение/удаление);
- управление доступом (ограничение возможностей ряда пользователей на доступ к некоторым категориям данных, защита данных от несанкционированного доступа);
- обеспечение целостности данных (защита базы данных от разрушения);
- управление состоянием СУБД.

SQL не является специализированным языком программирования, т. е. в отличие от языков высокого уровня (C++, Pascal и т. д.) с его помощью невозможно создать автономную программу. Все запросы выполняются либо в специализированных программах, либо из прикладных программ при помощи специальных библиотек.

Примечание

Несмотря на то, что SQL называют языком программирования баз данных, на самом деле это не совсем корректно, т. к. SQL работает только с базами данных определенного типа — реляционными базами данных. Однако не будет ошибкой сказать, что SQL на сегодняшний день это единственный язык программирования баз данных — по той причине, что реляционные базы данных фактически являются единственными типами баз данных, используемыми в настоящее время.

Несмотря на то, что язык запросов SQL строго стандартизирован, существует множество его диалектов, по сути, каждая база данных реализует свой собственный диалект со своими особенностями и ключевыми словами, недоступными в других базах данных. Такая ситуация связана с тем, что стандарты SQL появились достаточно поздно, в то время как компании-поставщики баз данных существуют давно и обслуживают большое число клиентов, для которых требуется обеспечить обратную совместимость со старыми версиями программного обеспечения. Кроме того, рынок реляционных баз данных оперирует сотнями миллиардов долларов в год, все компании находятся в жесткой конкуренции и постоянно совершенствуют свои продукты. Поэтому, когда дело доходит до принятия стандартов, базы данных уже имеют реализацию той или иной особенности, и комиссии по стандартам приходится в условиях жесткого давления выбирать в качестве стандарта решение одной из конкурирующих фирм. Так, хранимые процедуры, впервые появившиеся в MySQL 5 и чуть ранее в стандарте SQL, до недавнего времени были расширением SQL в языке запросов Transact-SQL базы данных MS SQL.

1.5. Версии MySQL

СУБД MySQL является развитием СУБД mSQL, которую разработчики шведской компании MySQL AB взяли за основу при создании своей собственной базы данных. Далее перечислены достоинства СУБД MySQL.

- Скорость выполнения запросов. Наряду с Oracle, MySQL считается одной из самых быстрых СУБД в мире.
- СУБД MySQL разработана с использованием языков C/C++ и оттестирована более чем на 23 платформах, среди которых Windows, Linux, FreeBSD, HP-UX, Mac OS X, OS/2, Solaris и др.
- Открытый код, который доступен для просмотра и модернизации всем желающим. Лицензия GPL (General Public License, общедоступная лицензия) позволяет постоянно улучшать программный продукт и быстро находить и устранять уязвимые места. Особенностью лицензии GPL является тот факт, что любой код, скомпилированный с GPL-кодом, попадает под GPL-лицензию, т. е. может свободно распространяться, и условием его распространения является предоставление исходных кодов. MySQL AB придерживается двойного лицензирования, позволяя пользователям бесплатно использовать СУБД MySQL под GPL-лицензией, а также предоставляет MySQL под коммерческой лицензией тем, кому необходимо использовать его в коммерческих целях, например, в составе программы.
- Высокое качество СУБД MySQL. Ни для кого не является секретом постоянно возрастающая сложность программных продуктов, что влечет за собой появление большого числа ошибок в конечных выпусках программного обеспечения. СУБД MySQL приятно удивит пользователей устойчивой работой.
- СУБД MySQL поддерживает API (Application Programming Interface, программный интерфейс приложения) для C, C++, Eiffel, Java, Perl, PHP, Python, Ruby и Tcl. MySQL можно успешно применять как для построения Web-страниц с использованием Perl, PHP и Java, так и для работы прикладной программы, созданной с использованием Builder C++ или платформы .NET.
- Наличие встроенного сервера. СУБД MySQL может быть использована как с внешним сервером, поддерживающим соединение с локальной машиной и с удаленным хостом, так и в качестве встроенного сервера. Достаточно скомпилировать программу с библиотекой встроенного сервера, и приложение будет содержать в себе полноценную СУБД MySQL с возможностью создания баз данных, таблиц и осуществления запросов к ним.
- Широкий выбор типов таблиц (типы таблиц описываются в *главе 11*), в том числе и сторонних разработчиков, что позволяет реализовать оптимальную для решаемой задачи производительность и функциональность. Кого-то привлечет наиболее быстрый тип таблиц MyISAM, а кому-то требуется поддержка объемов информации до 1 Тбайт с выполнением транзакций на уровне строк (таблицы типа InnoDB).
- Локализация в MySQL выполнена корректно. У пользователя не возникнет проблем из-за того, что в конце проекта обнаружится невозможность поддержки базой данных сортировки русских строк.
- MySQL, начиная с версии 5.0, практически полностью удовлетворяет стандарту SQL и, следовательно, хорошо совместима с другими базами данных.

Все эти особенности сделали MySQL стандартной базой данных, используемой на серверах хост-провайдеров и в качестве встроенной базы данных в прикладных программах.

На официальном сайте MySQL **http://www.mysql.com** поддерживается несколько версий СУБД MySQL, обычно это три версии. На момент написания данной книги это были следующие версии:

- MySQL 4.0 — устаревшая версия СУБД;
- MySQL 4.1 — стабильная (текущая) версия СУБД, рекомендуемая для загрузки;
- MySQL 5.0 — версия, находящаяся в стадии бета-тестирования.

Замечание

На компакт-диске, поставляемом вместе с данной книгой, можно найти дистрибутивы всех трех версий СУБД MySQL для операционных систем Windows и Linux.

По мере совершенствования MySQL 5.0 в ней будут устранены последние недочеты и она будет объявлена стабильной. При этом поддержка MySQL 4.0 будет прекращена, MySQL 4.1 будет объявлена устаревшей (но все еще будет доступна для загрузки), а в разработку поступит следующая версия — MySQL 5.1, которая пройдет через стадии гамма-, дельта-, бета- и альфа-тестирования.

1.5.1. Что нового в MySQL 4.1

Версия MySQL 4.1 стала стабильной в начале 2005 года и по сравнению с предыдущими версиями характеризуется приведенными далее нововведениями.

- Полная поддержка вложенных запросов. Данная тема рассматривается в *главе 23*.
- Увеличена производительность базы данных.
- Введен целый набор дополнительных функций.
- Изменен формат типа `TIMESTAMP`, а также введены дополнительные ключевые слова, которые позволяют управлять автоматическим обновлением времени при использовании операторов `INSERT` и `UPDATE`.
- Введена новая система поддержки кодировок, теперь можно назначить кодировку по умолчанию как таблице, так и отдельному столбцу в таблице.
- Введены новые типы таблиц, такие как EXAMPLE, NDB Cluster, ARCHIVE, CSV, которые подробно рассматриваются в *главе 11*.

1.5.2. Что нового в MySQL 5.0

Версия MySQL 5.0 является поворотной в линейке СУБД MySQL и содержит значимые нововведения, перечисленные далее.

- Представления, которые рассматриваются в *главе 30*.
- Хранимые процедуры и функции, рассматриваемые в *главе 28*.

- Курсоры, рассматриваемые в *главе 28*.
- Обработчики ошибок, рассматриваемые в *главе 28*.
- Триггеры, рассматриваемые в *главе 29*.
- Информационная схема, которая рассматривается в *главе 31*.
- Введен новый тип таблиц FEDERATED, который подробно рассматривается в *главе 11*.
- Введено множество новых функций, рассматриваемых в *части II* данной книги.

ГЛАВА 2

Установка MySQL 5

В данной главе описывается установка сервера MySQL на платформы ОС Windows и Linux.

2.1. Получение дистрибутива

Загрузить дистрибутив СУБД MySQL 5 можно с официального сайта компании AB MySQL по адресу **http://www.mysql.com/**. В разделе downloads (**http://dev.mysql.com/downloads/**) дистрибутив доступен в нескольких версиях: стабильной, устаревшей и будущей версии, находящейся в стадии альфа-, бета-, гамма- или дельта-тестирования.

> ***Замечание***
>
> На компакт-диске, прилагаемом к данной книге, можно найти дистрибутивы MySQL версий 4.0, 4.1 и 5.0 в бинарном виде (готовые к установке) — в директории binaries, а также в виде исходных кодов — в директории source для ОС Windows и Linux.

Для загрузки дистрибутива следует перейти по ссылке MySQL 5.0 (**http://dev.mysql.com/downloads/mysql/5.0.html**). На открывшейся странице будет представлен список дистрибутивов, скомпилированных под разные операционные системы. Для Linux имеются дистрибутивы в тарболах (.tar.gz) и в rpm-пакетах (.rpm), откомпилированные под 32- и 64-битную архитектуры. Для Windows представлены три вида дистрибутивов:

- Windows Essentials (x86) — урезанная версия дистрибутива, из которой удалены все вспомогательные утилиты, по сути, это "голый" сервер MySQL;
- Windows (x86) — полная версия дистрибутива MySQL, включающая автоматический установщик;
- Without installer (unzip in C:\) — полная версия MySQL без автоматического установщика.

Рекомендуется выбрать Windows (x86), поскольку, несмотря на то, что его объем превышает все остальные дистрибутивы из Windows-серии, он наиболее удобен в работе. Дальнейшее изложение материала будет основано именно на этом дистрибутиве.

Замечание

Полное справочное руководство на русском языке можно найти на официальном сайте MySQL по адресу **http://dev.mysql.com/doc/mysql/ru/index.html**. К сожалению, оно является устаревшим и описывает MySQL вплоть до версии 4.0. Полное справочное руководство на английском языке можно загрузить в различных форматах со страницы **http://dev.mysql.com/doc/** из раздела MySQL Reference Manual.

2.2. Установка на платформу Windows

При работе в Windows NT/2000/XP необходимо зарегистрироваться в системе с привилегиями администратора, разархивировать дистрибутив во временный каталог, после чего запустить файл setup.exe. В результате появится окно мастера установки, показанное на рис. 2.1.

Рис. 2.1. Стартовое окно автоматического установщика

Для продолжения установки следует нажать кнопку **Next**, после чего откроется окно, показанное на рис. 2.2, в котором предлагается выбрать различные способы установки:

- **Typical** (Стандартный);
- **Complete** (Полный);
- **Custom** (Выборочный).

В первом случае устанавливаются стандартные компоненты, во втором — все возможные компоненты, в третьем — по выбору пользователя.

Рис. 2.2. Выбор режима установки MySQL

В режиме **Custom** нажмите кнопку **Next** и в появившемся окне (рис. 2.3) выберите необходимые компоненты. Компоненты, которые по умолчанию отключены, перечеркнуты красным крестиком. Для того чтобы установить предлагаемые компоненты, необходимо выделить нужный компонент и выбрать в выпадающем меню пункт **This feature will be installed on local hard drive**. Если у вас отсутствует опыт установки MySQL, при первой установке лучше ничего не менять.

После того, как все компоненты выбраны, в этом же диалоговом окне (см. рис. 2.3) можно изменить директорию установки, нажав кнопку **Change**.

Замечание

По умолчанию директорией установки является C:\Program Files\MySQL\MySQL Server 5.0\.

Рекомендуется изменить путь по умолчанию на более короткий, например на C:\mysql5\, это необходимо для более комфортной работы с утилитами MySQL в командной строке. Нажав кнопку **Change**, введите путь `C:\mysql5\` (рис. 2.4).

Если вы устанавливаете MySQL поверх старой копии, например, MySQL 4.1, 4.0 или 3.23 — будьте осторожны. Такой способ вполне допустим и часто применяется для того, чтобы базы данных старой версии перешли в новую инсталляцию, но следует помнить, что системная база данных `mysql` в настоящий момент не совместима с предыдущими версиями. Поэтому если вы производите обновление MySQL, помимо остановки сервера, следует удалить каталог системной базы данных `mysql` C:\mysql\data\mysql. При первой инсталляции проблем, связанных с несовместимостью, возникать не должно.

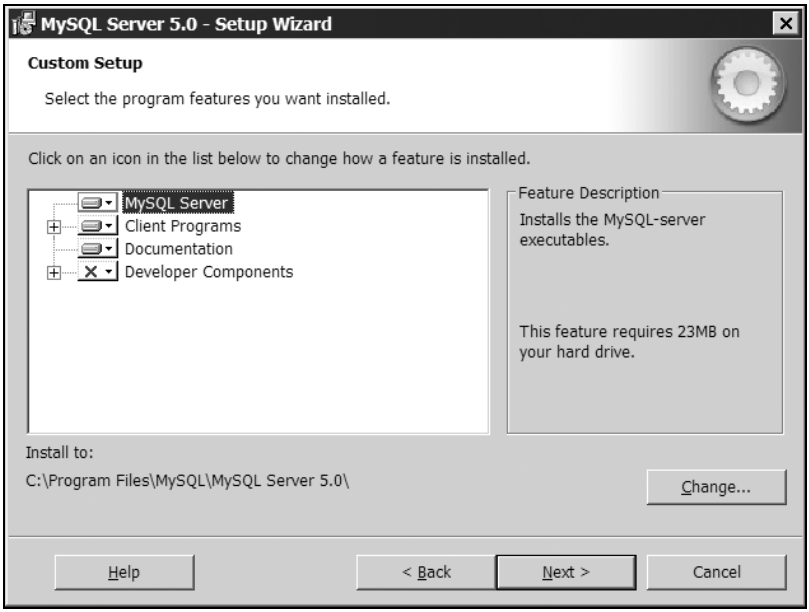

Рис. 2.3. Выбор компонентов для установки

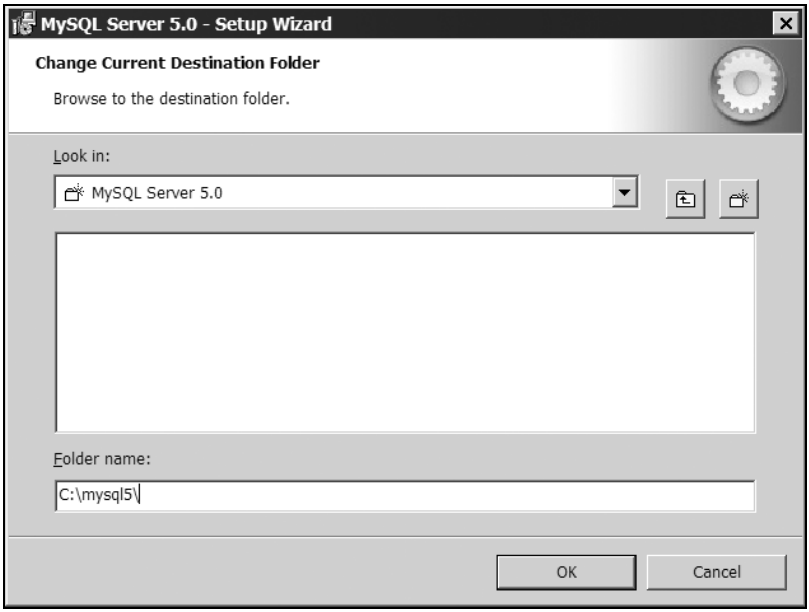

Рис. 2.4. Выбор директории, куда будет установлен MySQL

После завершения прединсталляционной настройки выводится окно, представленное на рис. 2.5. Нажатие кнопки **Install** начинает процесс копирования файлов, представленный на рис. 2.6.

После копирования файлов мастер установки предлагает зарегистрироваться на сайте **http://www.mysql.com**. Для того чтобы пропустить эту процедуру, выберите пункт **Skip Sign-Up** (рис. 2.7). Следующее окно (рис. 2.8) сообщает о завершении основного этапа установки MySQL.

Если в данном окне не отключать возможность немедленной настройки сервера, то после нажатия кнопки **Finish** запускается постинсталляционная настройка MySQL 5.

Замечание

К настройке всегда можно вернуться, выбрав пункт системного меню **Пуск | Программы | MySQL | MySQL Server 5.0 | MySQL Server Instance Config Wizard**. Тем не менее рекомендуется сразу произвести настройку системы.

Настройка начинается со стартового окна, изображенного на рис. 2.9. После нажатия кнопки **Next** открывается окно, представленное на рис. 2.10, в котором предлагается выбрать режим настройки:

- **Detailed Configuration** (Подробный);
- **Standard Configuration** (Стандартный).

Для более гибкой настройки системы следует выбрать первый пункт (**Detailed Configuration**). После нажатия кнопки **Next** открывается окно настройки производительности MySQL (см. рис. 2.11) со следующими опциями:

- **Developer Machine** (Машина разработчика);
- **Server Machine** (Сервер);
- **Dedicated MySQL Server Machine** (Выделенный сервер).

Все три опции различаются по интенсивности использования процессора, объему оперативной памяти и жесткого диска. Следует выбрать первый пункт, т. к. в этом случае MySQL занимает наименьший объем оперативной памяти, не мешая работе других приложений. Второй пункт предназначен для серверов, на который помимо MySQL будут работать другие серверы, например Web-сервер или транспортный почтовый агент. Третий пункт предназначен для выделенного сервера MySQL, на котором не будут выполняться никакие другие приложения.

Окно, представленное на рис. 2.12, позволяет выбрать предпочтительный тип для таблиц, который назначается по умолчанию. Следует оставить первый пункт.

Замечание

Результатом работы утилиты MySQL Server Instance Configuration Wizard является конфигурационный файл C:\mysql5\my.ini, который всегда можно отредактировать вручную, поэтому тип таблиц по умолчанию также несложно откорректировать позже.

Рис. 2.5. Завершающее окно перед процессом инсталляции

Рис. 2.6. Копирование файлов

Рис. 2.7. Регистрация на сайте **http://www.mysql.com**

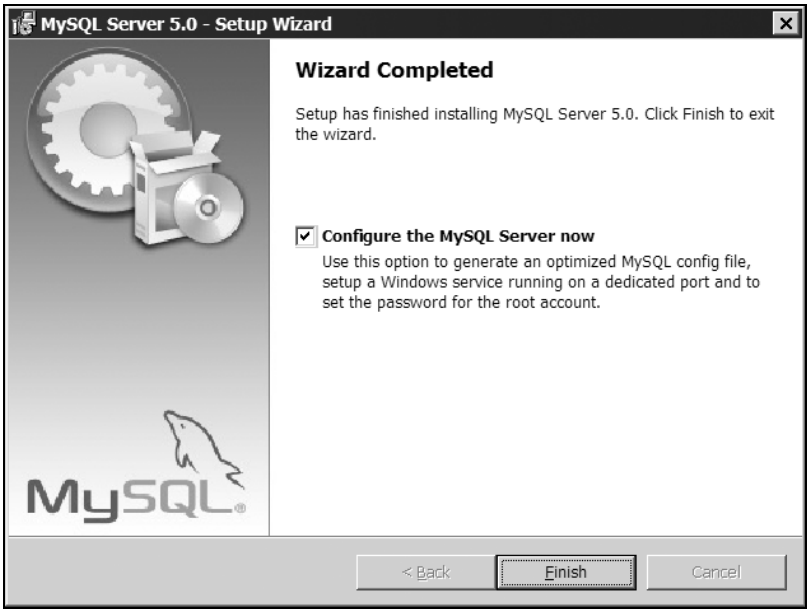

Рис. 2.8. Завершение основного этапа установки MySQL

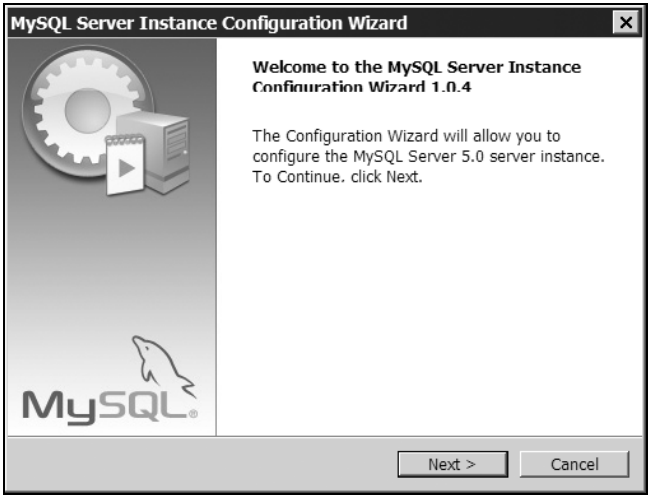

Рис. 2.9. Настройка MySQL при помощи мастера MySQL Server Instance Configuration Wizard

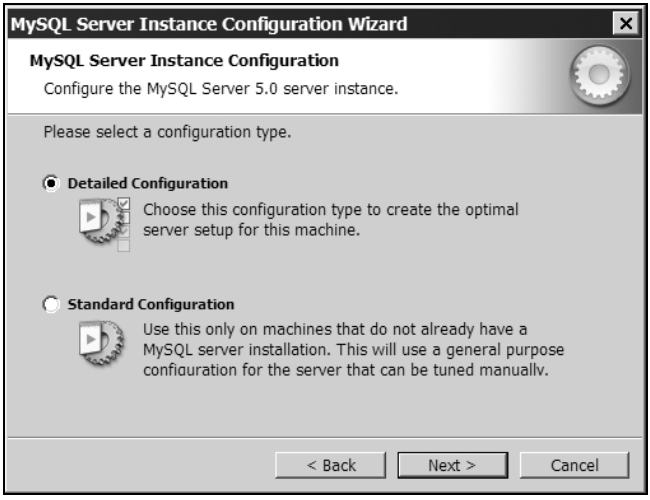

Рис. 2.10. Выбор режима настройки

В окне, показанном на рис. 2.12, предлагается выбор, по сути, между двумя типами таблиц: MyISAM и InnoDB, которые более подробно обсуждаются в *главе 11*. Главное различие заключается в том, что под каждую таблицу MyISAM создается отдельный файл, который хранится в соответствующей базе данных директории, а все таблицы InnoDB хранятся в едином табличном пространстве, под которое выделяется один файл. Выбор диска для хранения этого файла и пути, куда он будет установлен, осуществляется в следующем окне, представленном на рис. 2.13.

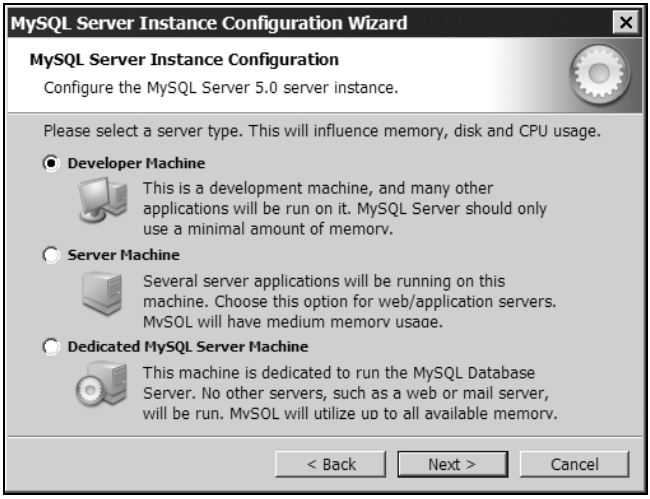

Рис. 2.11. Настройка производительности сервера MySQL

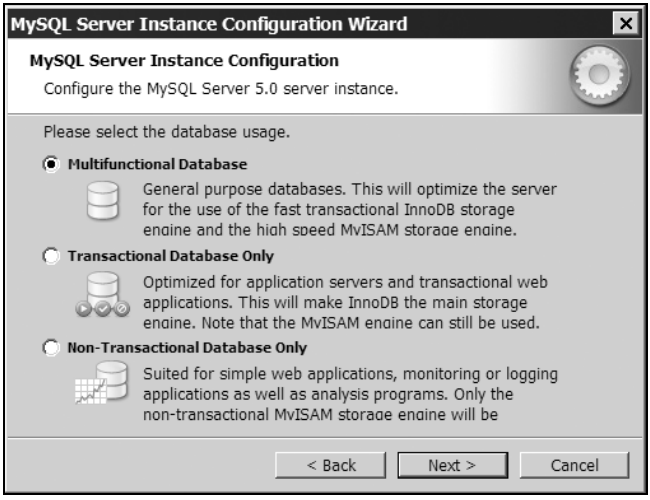

Рис. 2.12. Выбор типа таблиц по умолчанию

Следующее окно (рис. 2.14) предлагает выбрать максимальное число клиентов, которые могут одновременно подключиться к серверу. Первый пункт (рекомендуется выбрать именно его) предполагает, что число таких соединений не будет превышать 20, второй пункт допускает 500 соединений с сервером, третий пункт позволяет назначить собственный предел для числа активных соединений.

В следующем окне (рис. 2.15) устанавливается номер порта, по которому будет происходить соединение клиентов с MySQL-сервером (по умолчанию — 3306). Если на

компьютере не имеется других баз данных MySQL, рекомендуется оставить это значение по умолчанию, т. к. порт 3306 является стандартным для MySQL.

Примечание

В том случае, если необходимо запускать MySQL 5 совместно с другими версиями MySQL, можно выбрать другой номер порта, отличный от тех, по которым происходит обращение к прочим MySQL-серверам.

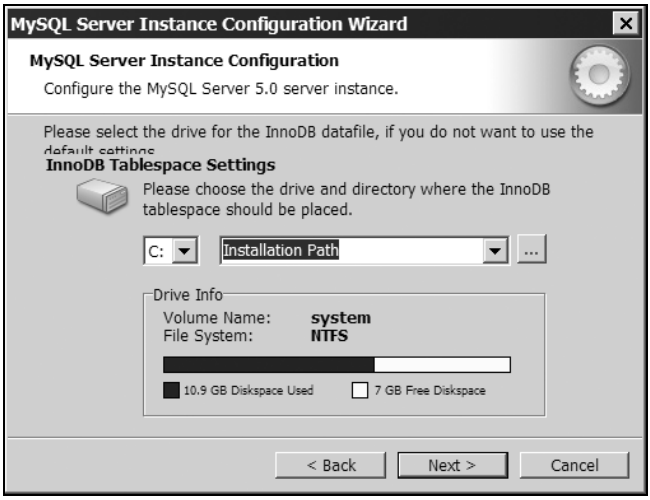

Рис. 2.13. Выбор пути для хранения файла под таблицы InnoDB

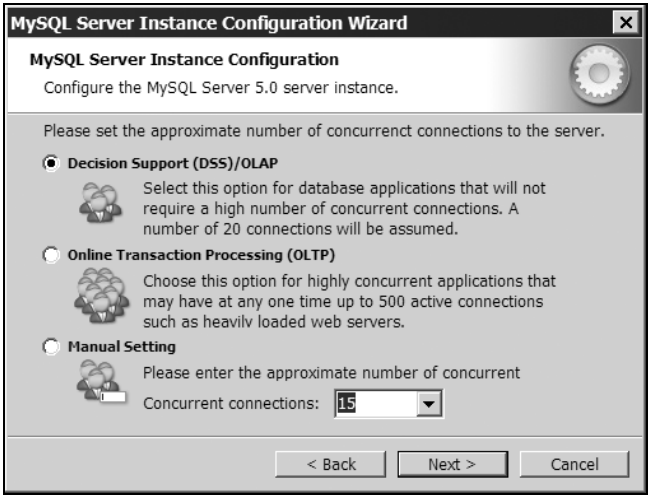

Рис. 2.14. Выбор максимального числа активных соединений с сервером

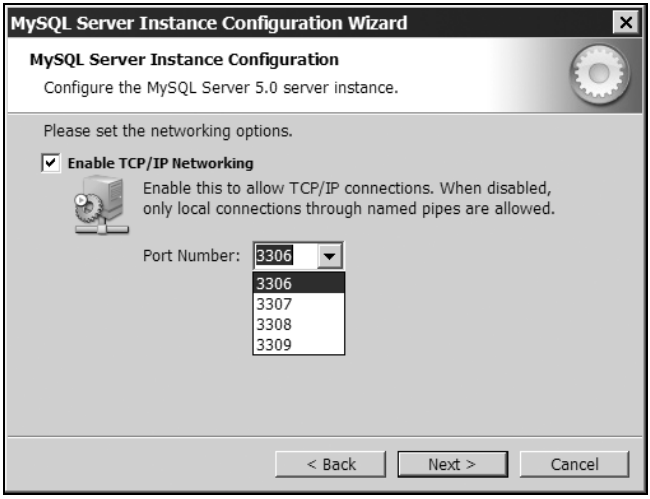

Рис. 2.15. Выбор порта, по которому сервер MySQL будет слушать клиентские запросы

В следующем окне предлагается указать кодировку по умолчанию. Необходимо выбрать третий пункт (ручной выбор кодировки) и в выпадающем окне выделить тип **cp1251**, соответствующий русской Windows-кодировке.

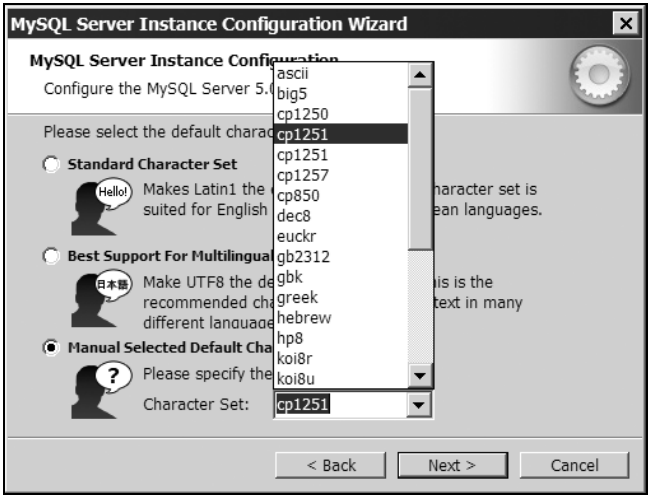

Рис. 2.16. Выбор кодировки на MySQL-сервере по умолчанию

При работе в среде Windows NT/2000/XP можно установить MySQL в качестве сервиса, что обеспечит запуск сервера mysqld при старте системы и корректное завершение работы сервера при выключении компьютера.

Окно, представленное на рис. 2.17, предназначено для настройки этого сервиса. Установка флажка **Install As Windows Service** позволяет установить сервис с именем,

которое следует выбрать в выпадающем списке **Service Name**. Можно изменить имя сервиса, особенно если в системе присутствует инсталлированный MySQL-сервер более ранней версии — это позволит избежать конфликтов при запуске серверов как MySQL 5, так и более ранней версии. Отметка флажка **Launch the MySQL Server automatic** позволяет настроить сервис на автоматический режим работы, когда запуск сервера производится со стартом системы, а останов — при завершении работы с системой. В противном случае потребуется ручной запуск и останов сервера. Как будет показано далее, можно изменить режим работы сервиса в любой момент в консоли управления сервисами.

Флажок **Include Bin Directory in Windows PATH** позволяет прописать путь к каталогу C:\mysql5\bin в системной директории `PATH`, что может быть удобным при частом использовании утилит из этой директории.

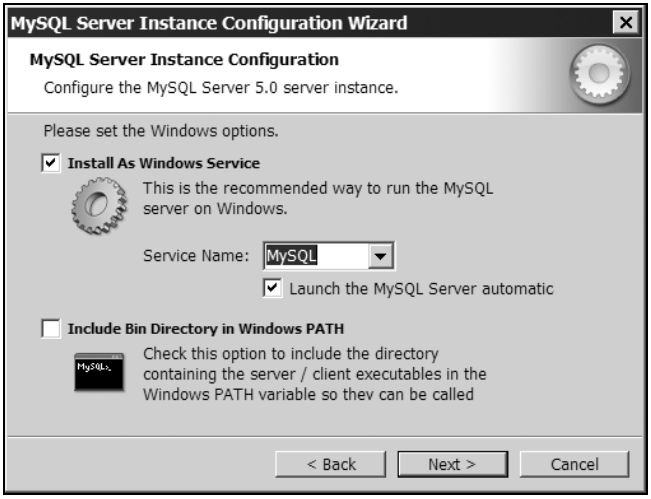

Рис. 2.17. Настройка сервиса для автоматического запуска MySQL-сервера

В следующем окне (рис. 2.18) производится настройка учетных записей. Если вы не знакомы с системой авторизации MySQL и производите установку первый раз, рекомендуется снять флажок **Modify Security Settings** и оставить данные настройки по умолчанию. Работа с учетными записями подробно рассматривается в *главах 3* и *25*.

Последняя страница утилиты настройки MySQL-сервера MySQL Server Instance Configuration Wizard представлена на рис. 2.19.

После нажатия кнопки **Execute** будет создан конфигурационный файл C:\mysql5\my.ini и запущен сервер MySQL.

Замечание

Если при установке сервера MySQL у вас возникли затруднения, вы можете обратиться на форум по адресу **http://www.softtime.ru/forum/index.php?id_forum=3**. Авторы присутствуют на форуме каждый день и помогают читателям с установкой

MySQL, Web-сервера Apache и PHP уже на протяжении полутора лет. На данном форуме можно также задавать любые вопросы, связанные как с администрированием, так и с программированием в среде MySQL.

Рис. 2.18. Настройка учетных записей

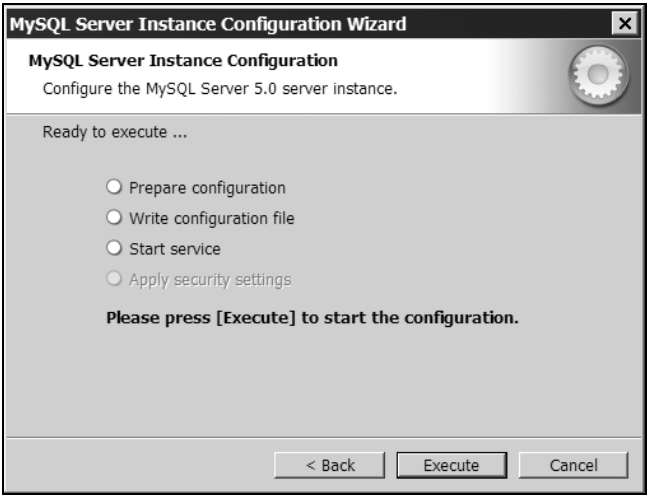

Рис. 2.19. Последняя страница утилиты MySQL Server Instance Configuration Wizard

После того как установка и конфигурирование MySQL завершены, необходимо убедиться в работоспособности сервера MySQL. Для этого следует запустить командную строку, выбрав в системном меню **Пуск | Программы | MySQL | MySQL Server 5.0 | MySQL Command Line Client**. Открывшееся окно может выглядеть так, как это по-

казано на рис. 2.20. На предложение ввести пароль нажмите клавишу <Enter>. После того как появилось приглашение `mysql>`, введите команду

```
SELECT VERSION();
```

В результате сервер должен показать текущую версию, в нашем случае — это `5.0.3-beta-nt` (рис. 2.21).

Замечание

Для выхода введите команду `EXIT`.

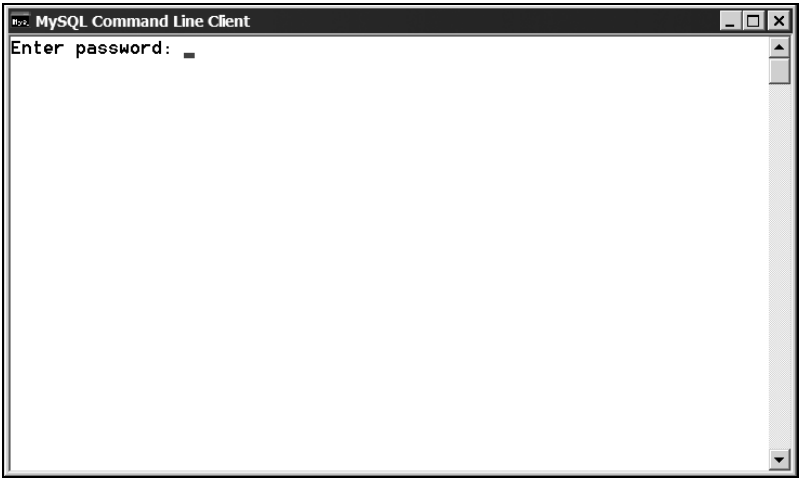

Рис. 2.20. Командная строка утилиты MySQL Command Line Client

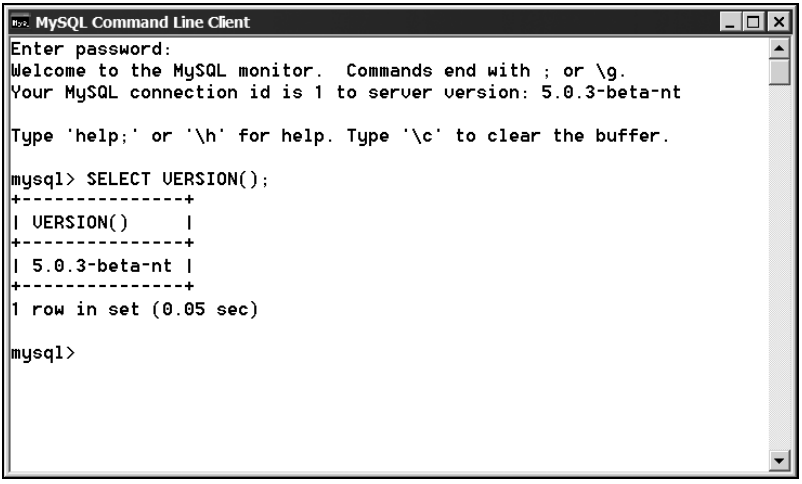

Рис. 2.21. Результат выполнения команды `SELECT VERSION()`

Если сервер не запущен, то после нажатия клавиши <Enter> окно будет закрыто. Для запуска сервера следует перейти в консоль управления сервисами, выполнив команду **Пуск | Настройка | Панель управления | Администрирование | Службы**. В результате этого откроется окно, представленное на рис. 2.22.

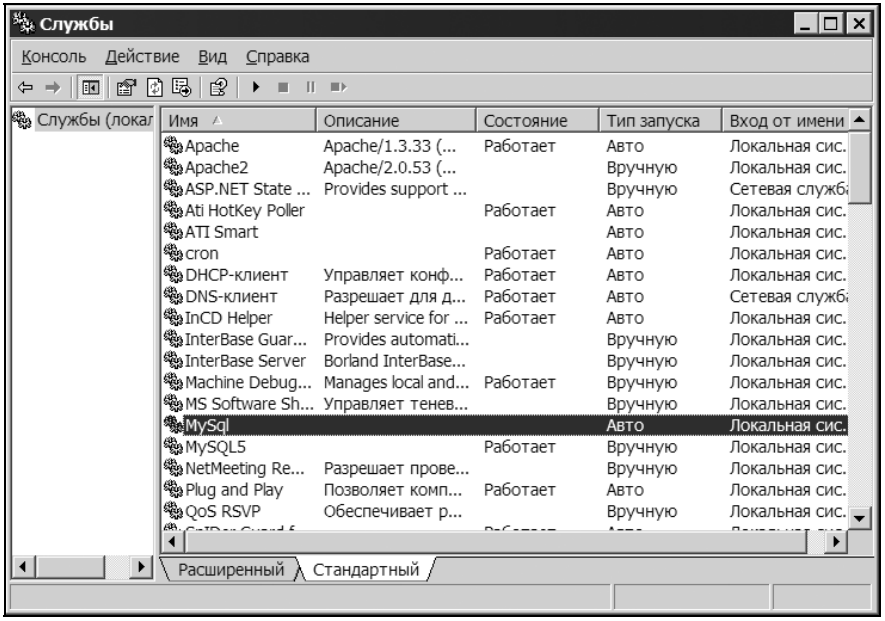

Рис. 2.22. Консоль управления сервисами

В консоли управления сервисами необходимо найти сервис MySQL. Если поле **Состояние** данного сервиса пусто, то он не запущен. Для его запуска следует выбрать в контекстном меню пункт **Пуск**. Для остановки сервиса необходимо выбрать пункт **Стоп**. Обратите внимание на столбец **Тип запуска**: значение **Авто** сообщает Windows о необходимости запускать сервис при старте операционной системы, значение **Вручную** означает необходимость запуска сервиса пользователем через консоль управления сервисами. Можно изменить режим запуска сервиса, выбрав в контекстном меню сервиса пункт **Свойства**.

2.3. Установка на платформу Linux

Большинство дистрибутивов Linux содержит в своем составе СУБД MySQL. В данном разделе будет рассмотрена установка сервера и клиента применительно к RedHat Linux (Fedora Core).

Несмотря на то, что в среде Linux принято устанавливать программное обеспечение из исходных кодов и MySQL, как база данных с открытым кодом позволяет делать это, официально рекомендуется устанавливать MySQL из бинарных пакетов, т. к. они

оптимизированы с учетом использования конкретной архитектуры для достижения максимальной эффективности.

Чтобы просмотреть все файлы в пакете RPM, выполните команду

```
rpm -qpl MySQL-VERSION.i386.rpm
```

Для выполнения стандартной установки войдите в каталог root и введите команду:

```
rpm -ihv MySQL-server-VERSION.i386.rpm MySQL-client-VERSION.i386.rpm
```

RPM помещает данные в каталог /var/lib/mysql/ и создает соответствующие вхождения в каталоге /etc/init.d/ для автоматического запуска сервера при загрузке.

При обновлении уже существующей версии MySQL можно воспользоваться командой:

```
rpm -Uhv MySQL-server-VERSION.i386.rpm MySQL-client-VERSION.i386.rpm
```

ГЛАВА 3

Работа с утилитами MySQL

Различные вспомогательные утилиты MySQL располагаются в подкаталоге bin корневого каталога MySQL. Далее приведен список наиболее часто используемых утилит:

- `myisampack` — сжимает таблицы типа MyISAM, уменьшая их в размере и делая доступными только для чтения;
- `mysql` — консольный клиент для доступа к MySQL-серверу, позволяет выполнять SQL-запросы и осуществлять администрирование сервера;
- `mysqladmin` — утилита для выполнения административных функций, таких как создание или удаление баз данных, получения информации с сервера о номере версии, процессах, состоянии сервера и т. п.;
- `mysqlbinlog` — данная утилита используется для чтения содержимого журнала двоичной регистрации при восстановлении данных в нештатных ситуациях;
- `mysqlcheck` — утилита, используемая для описания, проверки, оптимизации и восстановления таблиц;
- `mysqldump` — выводит содержимое базы данных MySQL в виде файла с SQL-операторами или в виде текстовых файлов с символом табуляции в качестве разделителя. Такие файлы часто называют дампом (dump) базы данных;
- `mysqlhotcopy` — утилита для создания резервной копии таблиц без остановки сервера MySQL, т. е. создания "горячей" копии базы данных;
- `mysqlimport` — выполняет перенос информации из текстового файла в таблицы базы данных;
- `mysqlshow` — отображает информацию о существующих базах данных, таблицах, полях и индексах.

На протяжении ближайших нескольких глав нам потребуются только две из названных выше утилит: `mysql` и `mysqldump`, которые и будут рассмотрены подробно. С описанием остальных утилит можно ознакомиться в документации MySQL.

Замечание

В стандартном дистрибутиве MySQL поставляются только консольные утилиты. Пользователям Windows и X Window часто трудно приспособиться к командной строке. Решением этой проблемы может стать выбор клиента с графическим интерфейсом. Так, на Web-странице **http://dev.mysql.com/downloads/** для свободной загрузки доступны графические клиенты MySQL Administrator, MySQL Query Browser и MySQL Control Center. Они также находятся на компакт-диске, прилагаемом к данной книге (директория clients).

3.1. Утилита *mysql*

Консольный клиент `mysql` часто называют "терминальным монитором" или просто "монитором". Пользователи операционной системы Linux могут набирать имя данной программы с параметрами из любой точки системы. Владельцам Windows для запуска mysql необходимо выбрать следующий пункт системного меню: **Пуск | Программы | MySQL | MySQL Server 5.0 | MySQL Command Line Client**. При такой форме запуска утилита `mysql` не позволяет вводить никакие параметры, кроме того, для работы с другими утилитами, входящими в поставку MySQL, необходим доступ к командной строке. Получить его можно, обратившись к следующему пункту: **Пуск | Программы | Стандартные | Командная строка**. В результате будет открыто окно, представленное на рис. 3.1.

Замечание

В данной книге цвет окна и его размеры выбраны отличными от тех, что выставляются по умолчанию системой. Можно самостоятельно изменить цветовую схему командной строки и ее размеры в свойствах системного меню окна.

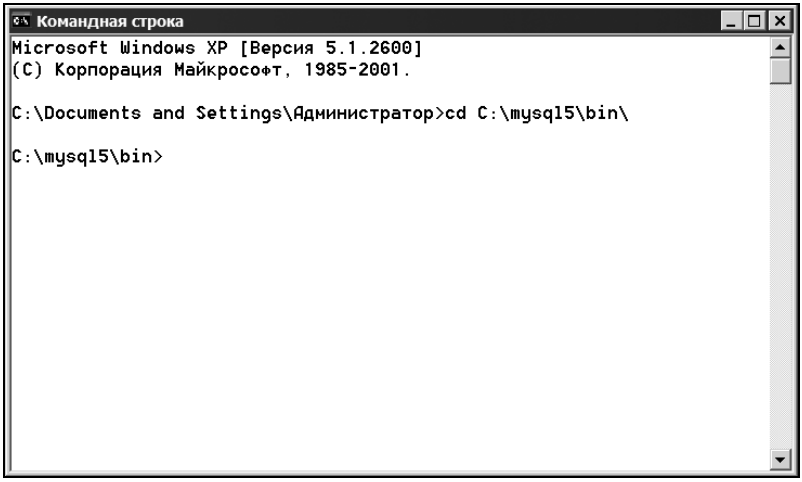

Рис. 3.1. Командная строка

После запуска командной строки необходимо перейти в поддиректорию bin корневого каталога MySQL. Для этого следует набрать команду

```
cd C:\mysql5\bin
```

В ответ будет выведено приглашение командной строки

```
C:\mysql5\bin
```

Замечание

Если корневой каталог MySQL расположен на другом диске, например D, необходимо перед выполнением команды `cd` сменить диск при помощи команды `D:`.

Теперь мы находимся в каталоге bin и можем запускать расположенные в нем утилиты. Для этого достаточно набрать в командной строке имя утилиты и, если необходимо, параметры. Параметры — это символы, начинающиеся с дефиса, например, `-u`, за которыми следует их значение. Использование различных параметров, которые будут рассмотрены в данной главе, позволяет изменять режим работы утилит.

Для соединения с сервером базы данных в параметрах mysql необходимо указать имя пользователя и его пароль. В только что установленной системе существует один пользователь `root`, наделенный правами администратора. Паролем по умолчанию выступает пустая строка. Поэтому для получения доступа к серверу достаточно набрать команду

```
mysql -u root
```

Результат выполнения команды представлен на рис. 3.2, после вывода версии сервера и информации о справочных ключах выводится приглашение `mysql>`, после которого можно набирать команды.

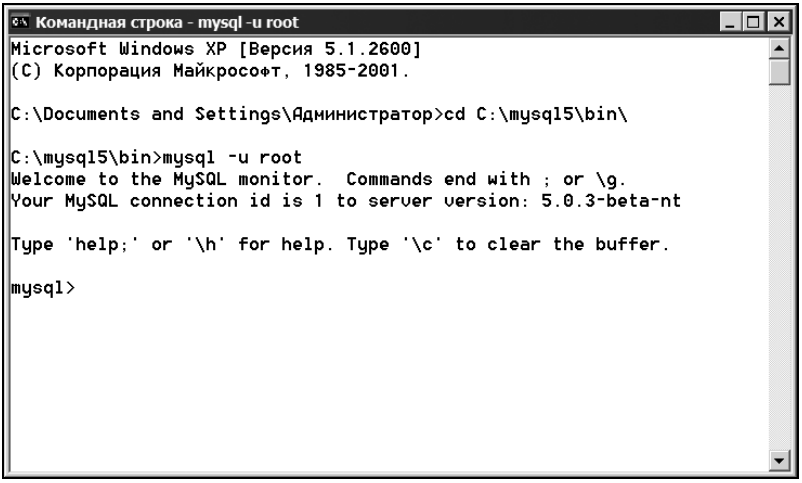

Рис. 3.2. Приглашение `mysql>` консольного клиента `mysql`

Символ `-u` является параметром, после которого следует имя пользователя, т. к. в системе нет пока пользователей, отличных от администратора, указывается `root`. В MySQL, начиная с версии 5.0, по умолчанию не устанавливается анонимного пользователя, т. е. пользователя, у которого в качестве имени выступает пустая строка. Поэтому необходимо каждый раз указывать его при помощи параметра `-u`. По умолчанию учетной записи `root` назначается в качестве пароля пустая строка, поэтому соединение с сервером базы данных MySQL не требует ввода пароля.

Для демонстрации полноценной авторизации создадим нового пользователя `user` с паролем `hello` при помощи инструкции `CREATE USER`, представленной в листинге 3.1. Эту инструкцию следует набрать сразу после приглашения `mysql>`, как это показано на рис. 3.3.

Замечание

Подробнее проблема безопасности, создания и удаления пользователей обсуждается в *главе 25*.

Листинг 3.1. Создание нового пользователя `user`

```
CREATE USER user IDENTIFIED BY 'hello';
```

После этого, выйдем их оболочки `mysql` и попробуем зайти из-под нового пользователя. Выход из оболочки `mysql` производится при помощи команды `exit` или `quit` (рис. 3.3).

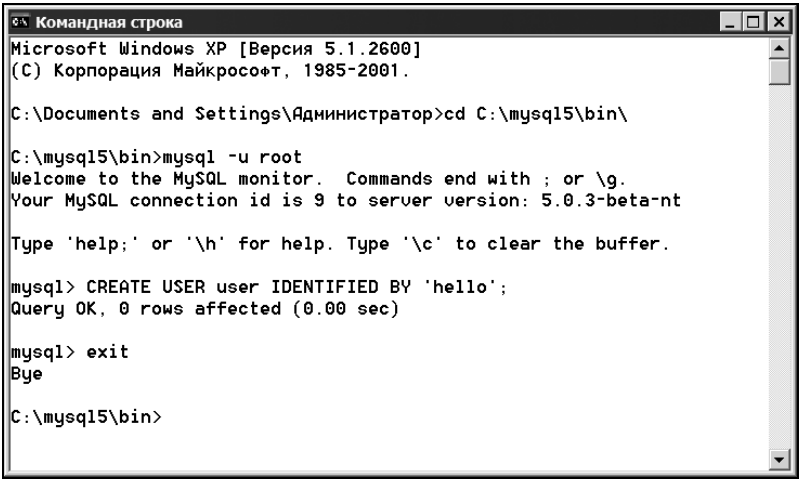

Рис. 3.3. Создание пользователя `user` с паролем `hello`

Теперь команда `mysql -u user` не проходит — сервер отказывается предоставить доступ, без подтверждения полномочия паролем (рис. 3.4). Для того чтобы подсоединиться к серверу из-под пользователя `user`, необходимо передать пароль. Для этого

используется параметр `-p`, сразу после которого без пробела вводится пароль. Для удобства в других параметрах допускается использование пробела между параметром и его значением, например следующие записи эквивалентны

```
mysql -uuser -phello
mysql -u user -phello
```

Параметр `-p` является исключением, т. к. все символы после него воспринимаются как часть пароля.

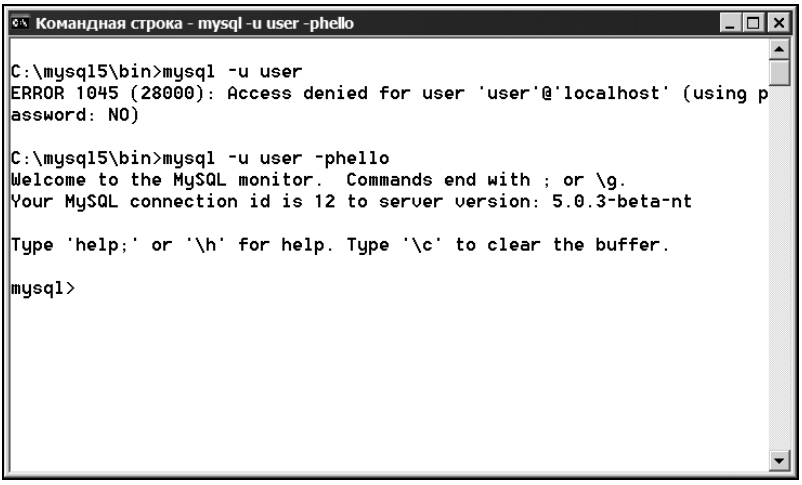

Рис. 3.4. Авторизация с использованием пароля

В некоторых ситуациях требуется скрыть пароль за символами звездочки, в этом случае допускается использование параметра `-p` без значения. Утилита `mysql` запросит пароль при помощи отдельной строки `Enter password:`, в которой вводимые символы будут скрыты символом звездочки (рис. 3.5).

Команды и SQL-инструкции, за редким исключением (`exit`, `use`), должны заканчиваться точкой с запятой. Например, на рис. 3.6 приведена команда, запрашивающая у сервера MySQL его версию и текущую дату.

Версия

Начиная с MySQL 5.0, изменить символ завершения запроса с точки запятой на новый символ, например, // можно либо при помощи команды `DELIMITER //`, либо указав разделитель при помощи параметра `--delimiter=name` *(см. раздел 28.2.1)*.

Когда пользователь вводит команду, она отправляется на сервер для выполнения, и если нет ошибок в синтаксисе, на экран выводится результат в виде таблицы, а на новой строке приглашение `mysql>`, после которого можно вводить новые команды.

В первой строке таблицы с результатами содержатся заголовки столбцов, а в следующих строках — ответ сервера на запрос. Обычно заголовками столбцов становят-

ся имена, полученные из таблиц базы. Если же извлекается не столбец таблицы, а значение выражения (как это происходит в приведенном выше примере), `mysql` дает столбцу имя запрашиваемого выражения. После этого сообщается количество возвращаемых строк (`1 row in set` — одна строка в результате) и время выполнения запроса.

Замечание

Для сокращения листингов в следующих главах книги число возвращаемых строк и время выполнения запроса будет опускаться.

```
Командная строка - mysql -u user -p

C:\mysql5\bin>mysql -u user -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 16 to server version: 5.0.3-beta-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Рис. 3.5. Альтернативный способ авторизации

```
Командная строка - mysql -u user -p
mysql> SELECT VERSION(), CURRENT_DATE;
+---------------+--------------+
| VERSION()     | CURRENT_DATE |
+---------------+--------------+
| 5.0.3-beta-nt | 2005-04-27   |
+---------------+--------------+
1 row in set (0.00 sec)

mysql> SELECT VERSION(),
    -> CURRENT_DATE
    ->
    ->
    ->
    -> ;
+---------------+--------------+
| VERSION()     | CURRENT_DATE |
+---------------+--------------+
| 5.0.3-beta-nt | 2005-04-27   |
+---------------+--------------+
1 row in set (0.00 sec)
```

Рис. 3.6. Выполнение команд

Для ввода ключевых слов можно использовать любой регистр символов. Так приведенные в листинге 3.2 запросы абсолютно идентичны.

Листинг 3.2. Имена инструкций и ключевые слова не зависят от регистра символов

```
mysql> SELECT VERSION(), CURRENT_DATE;
mysql> select version(), current_date;
mysql> SeLeCt vErSiOn(), current_DATE;
```

Если команда не убирается на одной строке, возможен переход на другую строку, после нажатия клавиши <Enter> — запрос отправляется серверу только после того, как консольный клиент `mysql` встретит символ точки с запятой. Приглашение командной строки после ввода первой строки этого запроса меняется с `mysql>` на `->` (см. рис. 3.6). Таким образом, программа `mysql` показывает, что завершенного выражения она пока что не получила и ожидает его полного ввода. Точно так же утилита `mysql` ведет себя когда ожидает завершение строки, заключенной в двойные (") или одинарные (') кавычки (рис. 3.7).

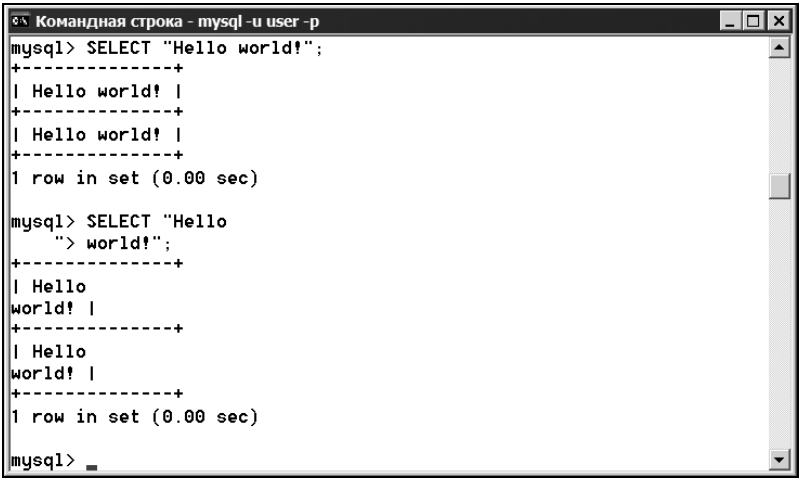

Рис. 3.7. Многострочный ввод текста, заключенного в двойные кавычки

Как видно из рис. 3.7, при переходе на следующую строку приглашение командной строки меняется с `mysql>` на `">`. Если строка заключена в одинарные кавычки, приглашение меняется на `'>`.

Уже введенные ранее команды не обязательно вводить снова, для этого достаточно их вызвать клавишами <↑> и <↓> (стрелка вверх и стрелка вниз), очистить строку запроса можно при помощи клавиши <Esc>.

Параметры в утилитах MySQL имеют две формы: полную, начинающуюся с двух дефисов `--user`, и краткую, которая начинается с одного дефиса `-u`. Можно применять оба варианта, но для ряда параметров имеется только полная форма. Ниже в списке

приведены стандартные параметры, применять которые можно со всеми утилитами MySQL:

- `--character-sets-dir=dir_name` — в данном параметре указывается имя каталога `dir_name`, в котором хранятся файлы кодировок;
- `--compress`, `-C` — указание данного параметра активирует сжатие данных при обмене по сети между клиентом и сервером;
- `--default-character-set=charset` — в данном параметре задается кодировка `charset` по умолчанию;
- `--database=name`, `-D` — база данных по умолчанию;
- `--delimiter=name` — разделитель по умолчанию;
- `--vertical`, `-E` — выводить таблицы в вертикальном стиле;
- `--no-beep`, `-b` — не издавать звуковой сигнал при возникновении ошибки;
- `--help`, `-I`, `-?` — требует от программы вывести справочную информацию и завершить работу;
- `--host=host_name`, `-h host_name` — в параметре указывается сетевое имя или IP-адрес компьютера `host_name`, на котором установлен MySQL-сервер;
- `--html`, `-H` — выводить результаты запросы в виде HTML-таблицы;
- `--xml`, `-X` — выводить результаты запроса в виде XML-дерева;
- `--password[=pas_val], -p[pas_val]` — в данном параметре указывается пароль `pas_val` для доступа к MySQL-серверу;
- `--port=port_name`, `-P port_name` — в данном параметре указывается номер порта `port_name`, по которому MySQL-сервер ожидает соединения с клиентом (по умолчанию 3306, если данный параметр не указан, производится попытка установить соединение именно по данному порту);
- `--user=user_name`, `-u user_name` — в данном параметре указывается имя пользователя `user_name`, от имени которого осуществляется соединение с MySQL-сервером;
- `--version`, `-V` — указание данного параметра требует от утилиты вывести строку с информацией о текущей версии и завершить работу.

Замечание

Консольный клиент `mysql` поддерживает множество других параметров, ознакомиться с которыми можно в документации к MySQL.

Интересна возможность установки соединения с удаленным хостом, так в домашней локальной сети автора MySQL 5 установлен на машине с IP-адресом 192.168.200.3. На рис. 3.8 показан процесс установки соединения с данным удаленным сервером.

На машине по определенному порту может быть установлено только одно приложение. Если необходимо установить еще один сервер, его требуется установить по дру-

гому порту, например 3307, в этом случае при установке соединения в параметре `-P` необходимо указывать данный порт (`-P 3307`).

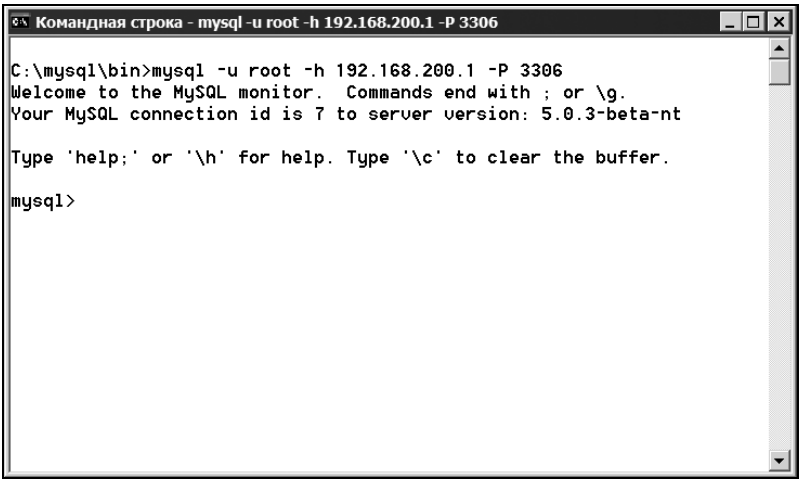

Рис. 3.8. Установка соединения с удаленным сервером 192.468.200.1

Замечание

Для успешной установки соединения на сервере должны быть прописаны IP-адреса, с которыми пользователь может обращаться к MySQL-серверу. Если планируется использовать MySQL-сервер в локальной сети, ознакомьтесь с *главой 25*, посвященной учетным записям MySQL.

Длинный ряд параметров не удобно всякий раз набирать при запуске консольного клиента `mysql`, поэтому они могут быть прописаны в конфигурационном файле `my.ini`. Данный файл разделен на разделы, которые относятся к разным частям, например, раздел `[mysqld]` относится к серверу MySQL, а раздел `[client]` — к консольному клиенту `mysql`. Так в каждом из разделов может быть указана кодировка по умолчанию

```
default-character-set=cp1251
```

В разделе `[mysqld]` может быть указан тип таблиц по умолчанию, например, InnoDB

```
default-storage-engine=INNODB
```

или MyISAM

```
default-storage-engine=MYISAM
```

или любой другой. Изменения вступают в силу после рестарта сервера, который можно осуществить в консоли управления сервисами (см. рис. 2.21).

Замечание

Подробнее ознакомиться с типами таблиц MySQL можно в *главе 11*.

В командной строке Windows по умолчанию используется кодировка DOS (cp866), поэтому, для того чтобы в консоли русский текст был читаемый, необходимо перед запуском клиента `mysql` изменить кодировку консоли при помощи команды

```
chcp 1251
```

Замечание

Использование команды `chcp` без параметра позволяет узнать текущую кодировку командной строки.

Замечание

Если в командной строке используются "Точечные шрифты" после смены кодировки на cp1251 в системном меню, следует изменить шрифт на Lucida Console, т. к. в данном семействе шрифтов имеются кириллические Windows-шрифты, в "Точечных шрифтах" имеется шрифт только для кодировки cp866 (DOS).

3.2. Утилита *mysqldump*

Данная утилита позволяет получить дамп содержимого базы данных или совокупности баз для создания резервной копии или пересылки данных на другой SQL-сервер баз данных (не обязательно MySQL-сервер). Дамп будет содержать набор команд SQL для создания и/или заполнения таблиц.

Для работы с данной утилитой следует сделать небольшое отступление. В MySQL базы данных представляют собой подкаталоги, расположенные в каталоге данных `C:\mysql5\data`, имена которых совпадают с именами баз данных. Создание в этой директории новой поддиректории аналогично процедуре создания новой базы данных при помощи оператора `CREATE DATABASE`. На компакт-диске, идущем вместе с книгой содержится учебная база данных `shop` как в виде готового дампа (`base\base.sql`), так и в бинарном виде в директории `base\shop`. Достаточно скопировать данную директорию в `C:\mysql5\data` и в системе появится новая база данных, которая автоматически распознается сервером MySQL.

Замечание

Подробное описание учебной базы данных `shop` приведено в *главе 4*.

Теперь можно приступать к созданию дампа базы данных `shop`. Для этого в командной строке необходимо выполнить команду, представленную в листинге 3.3.

Листинг 3.3. Создание дампа базы данных `shop`

```
C:\mysql5\bin> mysqldump -u root shop > shop.sql
```

Как видно из листинга 3.3, утилита `mysqldump` принимает имя пользователя при помощи параметра `-u`. Следует отметить, что `mysqldump` принимает все параметры, перечисленные в разделе, посвященному утилите `mysql`. Кроме этого, после всех параметров указывается имя базы данных `shop`, для которой осуществляется создание

дампа. Так как вывод данных осуществляется в стандартный поток (за которым по умолчанию закреплен экран монитора), его следует перенаправить в файл (в листинге 3.3 — это `shop.sql`). Перенаправление данных осуществляется при помощи оператора `>`. Если вместо оператора `>` использовать `>>`, то данные не будут перезаписывать уже существующий файл, а будут добавлены в конец файла.

Помимо параметров, представленных в предыдущем разделе, `mysqldump` поддерживает несколько специфических параметров. К ним относится параметр `--databases` или в сокращенной форме `-B`. Данный параметр позволяет создать дамп сразу нескольких баз данных, которые можно перечислить через пробел после него.

Листинг 3.4. Создание дампа нескольких баз данных

```
C:\mysql5\bin> mysqldump -u root -B shop mysql > shop_mysql.sql
```

Так команда, представленная в листинге 3.4, сохраняет дампы баз данных `shop` и `mysql` в файл `shop_mysql.sql`.

Если необходимо сохранить дамп всех баз данных MySQL-сервера, следует воспользоваться параметром `--all-databases` или в сокращенной форме `-A` (листинг 3.5).

Листинг 3.5. Создание дампа всех баз данных MySQL-сервера

```
mysqldump -u root --all-databases > all_databases.sql
```

Полученный в результате дамп базы данных представляет собой текстовый файл с SQL-инструкциями, выполнить которые можно при помощи утилиты `mysql`. При развертывании дампа базы данных удобно воспользоваться пакетным режимом (листинг 3.6).

Листинг 3.6. Развертывание дампа базы данных с использованием `mysql`

```
mysql -u root test < shop.sql
```

В листинге 3.6 данные из дампа `shop.sql` перенаправляются на стандартный вход утилиты `mysql`, которая размещает таблицы базы данных `shop` в базе данных `test`.

ГЛАВА 4

Создание баз данных и таблиц. Типы данных

Данные в любой СУБД организованы в многоуровневые структуры. На верхнем уровне расположена база данных, в состав которой входят таблицы. Каждая таблица разделена на столбцы и строки, на пересечении которых содержатся значения данных. Каждый столбец имеет свой собственный тип.

Замечание

MySQL не поддерживает каталоги баз данных, как это свойственно для некоторых СУБД.

4.1. Создание базы данных

Как было сказано в *главе 3*, в СУБД MySQL создание базы данных сводится к созданию нового подкаталога в каталоге данных C:\mysql5\data. Создание базы данных средствами SQL осуществляется при помощи оператора `CREATE DATABASE`. В листинге 4.1 приведен пример создания базы данных `wet`.

Листинг 4.1. Создание базы данных `wet`

```
mysql> CREATE DATABASE wet;
Query OK, 1 row affected (0.00 sec)
```

После выполнения запроса из листинга 4.1, заглянув в каталог C:\mysql5\data, можно обнаружить новый каталог wet. Максимальная длина имени базы данных составляет 64 знака и может включать литеры, цифры и символы '_' и '$'. Имя может начинаться с цифры, но не должно полностью состоять из цифр.

Замечание

Начиная с версии 4.1.1, в каталоге базы данных создается также файл db.opt, в котором указывается кодировка, используемая в базе данных по умолчанию `default-character-set`, и порядок сортировки символов `default-collation`.

Проконтролировать создание базы данных, а также узнать имена существующих баз данных можно при помощи оператора `SHOW DATABASES` (листинг 4.2).

Листинг 4.2. Использование оператора `SHOW DATABASES`

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| shop               |
| test               |
| wet                |
+--------------------+
```

Как видно из листинга 4.2, оператор `SHOW DATABASES` вернул имена пяти баз данных. Базы данных `information_schema` и `mysql` являются служебными и необходимы для поддержания сервера MySQL в работоспособном состоянии — в них хранится информация об учетных записях, региональных настройках и т. д.

База данных `shop` была создана в *главе 3*, а база данных `wet` — при помощи SQL-запроса в листинге 4.1. База данных `test` является пустой и создается при установке MySQL вместе с системными базами данных `information_schema` и `mysql`.

Попробуйте создать в директории C:\mysql5\data\ новую директорию и выполнить после этого запрос `SHOW DATABASES` — созданная таким образом база данных будет отображена в списке баз данных. Удаление директории приведет к исчезновению базы данных.

Удаление баз данных можно осуществить и штатными средствами при помощи оператора `DROP DATABASE`, за которым следует имя базы данных (листинг 4.3).

Листинг 4.3. Удаление базы данных `wet`

```
mysql> DROP DATABASE wet;
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| shop               |
| test               |
+--------------------+
```

После выполнения оператора `DROP DATABASE` можно убедиться, что из директории данных C:\mysql5\data\ была удалена директория wet.

Если производится попытка создания уже существующей базы данных, возвращается ошибка (листинг 4.4).

Листинг 4.4. Невозможно создать базу данных `wet`: база данных существует

```
mysql> CREATE DATABASE wet;
Query OK, 1 row affected (0.00 sec)
mysql> CREATE DATABASE wet;
ERROR 1007: Can't create database 'wet'; database exists
```

Часто такое поведение нежелательно, особенно в больших дампах, размещенных в sql-файлах, которые выполняются в пакетном режиме так, как это было продемонстрировано в *главе 3*. Для предотвращения ошибки оператор `CREATE DATABASE` можно снабдить конструкцией `IF NOT EXISTS`, при наличии которой база данных создается, если она еще не существует, если же существует — никаких действий не производится (листинг 4.5).

Листинг 4.5. Использование конструкции `IF NOT EXISTS`

```
mysql> DROP DATABASE wet;
Query OK, 0 rows affected (0.00 sec)
mysql> CREATE DATABASE IF NOT EXISTS wet;
Query OK, 1 row affected (0.00 sec)
mysql> CREATE DATABASE IF NOT EXISTS wet;
Query OK, 0 rows affected (0.00 sec)
```

Как видно из листинга 4.5, после удаления базы данных `wet` первый оператор `CREATE DATABASE` возвращает ответ, в котором говорится о том, что была задействована одна строка (`1 row affected`), т. е. база данных создается. Второй оператор `CREATE DATABASE` возвращает сообщение, что запрос не затронул ни одной строки (`0 rows affected`), т. е. база данных не создается, однако и ошибка не выдается.

Точно такой же механизм разработан для оператора `DROP DATABASE`, добавление ключевого слова `IF EXISTS` удаляет базу данных, если она существует, и не производит никаких действий, если база данных отсутствует (листинг 4.6).

Листинг 4.6. Использование конструкции `IF EXISTS`

```
mysql> DROP DATABASE IF EXISTS wet;
Query OK, 0 rows affected (0.03 sec)
mysql> DROP DATABASE IF EXISTS wet;
Query OK, 0 rows affected (0.00 sec)
```

Специального SQL-оператора, предназначенного для переименования базы данных, не существует, но эту операцию легко осуществить, переименовав каталог базы данных.

При создании базы данных можно указать кодировку, которая будет назначаться таблицам и столбцам по умолчанию. Для этого после имени базы данных следует указать ключевое слово `DEFAULT CHARACTER SET charset_name`, где `charset_name` — имя кодировки, например `cp1251`, которая обозначает русскую Windows-кодировку (листинг 4.7).

Листинг 4.7. Назначение кодировки по умолчанию

```
mysql> CREATE DATABASE wet DEFAULT CHARACTER SET cp1251;
```

Замечание

Подробнее кодировки обсуждаются в *главе 11*, а с полным списком кодировок можно ознакомиться в листинге 11.17.

Указание кодировки по умолчанию приводит к созданию в директории базы данных C:\mysql5\data\wet\ файла db.opt следующего содержания:

```
default-character-set=cp1251
default-collation=cp1251_general_ci
```

Если при конфигурировании MySQL утилитой MySQL Server Instance Configuration Wizard или при ручном редактировании конфигурационного файла my.ini в качестве кодировки по умолчанию для MySQL-сервера была выбрана другая кодировка, например `latain1`, то при отсутствии конструкции `DEFAULT CHARACTER SET` вместо `cp1251` будет использована именно `latain1`. Если затем кодировка не будет меняться при определении таблицы и столбцов, это может привести к неправильной сортировке русского текста.

Замечание

Подробное описание конфигурирования MySQL-сервера при помощи утилиты MySQL Server Instance Configuration Wizard приводится в *главе 2*.

Переименование базы данных можно осуществить, изменив имя директории, соответствующее базе данных, в каталоге C:\mysql5\data. Для изменения параметров базы данных можно воспользоваться оператором `ALTER DATABASE`, как это продемонстрировано в листинге 4.8.

Листинг 4.8. Изменение кодировки базы данных

```
ALTER DATABASE wet CHARACTER SET cp1251
```

В листинге 4.8 для базы данных `wet` в качестве кодировки по умолчанию устанавливается `cp1251`.

Замечание

Оператор `ALTER DATABASE` введен в версии MySQL 4.1.1. Начиная с версии 4.1.8, допускается не указывать имя базы данных, т. е. вполне допустима запись вида `ALTER DATABASE CHARACTER SET cp1251`.

4.2. Создание таблицы

Перед тем как создать таблицу, в клиентской программе следует выбрать базу данных, с которой будет производиться работа. Эта операция осуществляется при помощи инструкции `USE` (листинг 4.9).

Листинг 4.9. Выбор базы данных

```
mysql> USE test;
Database changed
```

Замечание

Прежде чем работать с таблицами, всегда производится выбор базы данных. Это относится и к прикладным программам, работающим с базой данных. Обычно с этой целью применяется функция с суффиксом `_select_db`.

Для создания таблицы используется оператор `CREATE TABLE`, после которого следует имя создаваемой таблицы и в круглых скобках — структура таблицы (листинг 4.10).

Листинг 4.10. Создание таблицы `tbl`

```
mysql> CREATE TABLE tbl (number INT, name TEXT);
Query OK, 0 rows affected (0.06 sec)
```

В круглых скобках через запятую указываются имена столбцов и их тип. Так, созданная таблица `tbl` имеет два поля: `number`, куда можно помещать целочисленные значения (`INT`), и поле `name`, в котором размещаются строки (`TEXT`). Максимальная длина имен таблиц и столбцов составляет 64 знака и может включать литеры, цифры и символы '_' и '$'. Имя может начинаться с цифры, но не должно полностью состоять из цифр.

Замечание

Более подробно типы столбцов обсуждаются далее в этой главе. Синтаксис SQL-оператора `CREATE TABLE` полностью рассматривается в *главе 11*.

Замечание

В операционной системе Windows имена таблиц и баз данных не зависят от регистра, но тогда как в UNIX-подобных операционных системах — зависят. Это связано с тем, что база данных является директорией, а таблицы — файлами, имена которых совпадают с именами баз данных и таблиц. В файловой системе Windows имена файлов не зависят от регистра, в то время как в UNIX имена, отличающиеся только регистром, считаются разными файлами.

Проконтролировать вновь созданную таблицу можно при помощи оператора `SHOW TABLES`, который возвращает список таблиц текущей базы данных.

Листинг 4.11. Использование оператора SHOW TABLES

```
mysql> SHOW TABLES;
+----------------+
| Tables_in_test |
+----------------+
| tbl            |
+----------------+
```

Для того чтобы просмотреть содержимое другой базы данных, следует предварительно выбрать ее с использованием оператора USE (листинг 4.12).

Листинг 4.12. Использование оператора USE

```
mysql> USE shop;
Database changed
mysql> SHOW TABLES;
+----------------+
| Tables_in_shop |
+----------------+
| catalogs       |
| orders         |
| products       |
| users          |
+----------------+
```

Как видно из листинга 4.12, учебная база данных содержит четыре таблицы: catalogs, orders, products и users.

Замечание

Подробнее структура учебной базы данных shop рассматривается в *разд. 4.4* данной главы.

Удаление таблицы производится при помощи оператора DROP TABLE (листинг 4.13).

Листинг 4.13. Использование оператора DROP TABLE

```
mysql> USE test;
Database changed
mysql> DROP TABLE tbl;
Query OK, 0 rows affected (0.02 sec)
mysql> SHOW TABLES;
Empty set (0.00 sec)
```

Замечание

Наряду с обращением к таблице по ее имени, например tbl, существует расширенное обращение, включающее в себя имя базы данных, например, test.tbl. Любой

SQL-оператор может обращаться к базе данных по этому имени. Так, оператор DROP TABLE из листинга 4.13 может выглядеть следующим образом: DROP TABLE test.tbl. При использовании расширенного синтаксиса допускается не производить выбор базы данных оператором USE, т. к. база данных явно указывается в запросе.

Для того чтобы получить описание столбцов базы данных, можно воспользоваться оператором DESCRIBE.

Листинг 4.14. Использование оператора DESCRIBE

```
mysql> DESCRIBE test.tbl;
+--------+---------+------+-----+---------+-------+
| Field  | Type    | Null | Key | Default | Extra |
+--------+---------+------+-----+---------+-------+
| number | int(11) | YES  |     | NULL    |       |
| name   | text    | YES  |     | NULL    |       |
+--------+---------+------+-----+---------+-------+
```

Как видно из листинга 4.14, имя таблицы приведено в расширенном формате test.tbl, однако вполне допустимо указание одного имени таблицы tbl, если в качестве текущей базы данных выбрана test.

Замечание

Оператор DESCRIBE не является стандартным оператором языка запросов SQL и характерен только для диалекта MySQL.

Оператор DESCRIBE возвращает таблицу, каждая строка которой соответствует столбцу таблицы tbl. Имена столбцов и их типы приводятся в первом и втором столбцах соответственно. К остальным столбам результирующей таблицы мы вернемся по мере освоения материала.

4.3. Типы данных

MySQL поддерживает несколько типов данных.

- Числовые данные — к ним относят целые числа, не содержащие дробной части (например, 124), а также вещественные числа, состоящие из последовательности цифр, разделенных точкой (например, 56.45).
- Строковые данные — последовательность символов, заключенных в одинарные или двойные кавычки: 'Hello world', '123', "MySQL". В качестве стандарта в SQL определяются одинарные кавычки, поэтому для совместимости с другими базами данных рекомендуется использовать именно их.
- Календарные данные — специальный тип для обозначения даты и времени, может принимать различную форму, например строковую "2005-04-28" или числовую 20050428. Основной характеристикой этого типа данных является их хранение в

единном внутреннем формате, позволяющем осуществлять операции сложения и вычитания, независимо от внешнего представления.

- NULL — специальный тип данных, обозначающий отсутствие информации.

4.3.1. Числовые данные

Числовые данные делятся на точечные (BOOLEAN, INTEGER и DECIMAL) и приближенные (FLOAT, REAL и DOUBLE PRECISION). Характеристики и занимаемый объем точечных типов приведены в табл. 4.1.

Замечание

Здесь и далее необязательные элементы синтаксиса будут заключаться в квадратные скобки, т. е. запись TINYINT[(M)] обозначает, что элемент может быть записан и как TINYINT, и как TINYINT(M).

Таблица 4.1. *Точечные типы*

Тип	Объем памяти	Диапазон
TINYINT[(M)]	1 байт	от -128 до 127 (от -2^7 до 2^7-1) от 0 до 255 (от 0 до 2^8-1)
SMALLINT[(M)]	2 байта	от -32768 до 32767 (от -2^{15} до 2^{15}-1) от 0 до 65535 (от 0 до 2^{16}-1)
MEDIUMINT[(M)]	3 байта	от -8388608 до 8388608 (от -2^{23} до 2^{23}-1) от 0 до 16777215 (от 0 до 2^{24}-1)
INT[(M)], INTEGER[(M)]	4 байта	от -2147683648 до 2147683648 (от -2^{31} до 2^{31}-1) от 0 до 4294967295 (от 0 до 2^{32}-1)
BIGINT[(M)]	8 байт	(от -2^{63} до 2^{63}-1) (от 0 до 2^{64})
BIT[(M)]	(M+7)/8 байт	От 1 до 64 битов, в зависимости от значения M
BOOL, BOOLEAN	1 байт	Либо 0, либо 1
DECIMAL[(M[,D])], DEC[(M[,D)]], NUMERUC[(M[,D)]]	M + 2 байта	Повышенная точность; зависит от параметров M и D

Как видно из табл. 4.1, СУБД MySQL имеет пять целых типов: `TINYINT`, `SMALLINT`, `MEDIUMINT`, `INT`, `BIGINT`. Различие между ними заключается в диапазоне величин, которые можно хранить в столбцах такого типа. Чем больше диапазон значений у типа данных, тем больше памяти для него требуется.

Замечание

Тип `INT` имеет синоним `INTEGER`.

Целые типы данных могут быть объявлены положительными. Для этого после объявления типа следует использовать ключевое слово `UNSIGNED`. В этом случае элементам данного столбца нельзя будет присвоить отрицательные значения, а допустимый диапазон, который может принимать тип, удваивается. Так, тип `TINYINT` может принимать значения от -128 до 127, а `TINYINT UNSIGNED` — от 0 до 255.

При объявлении целого типа задается количество отводимых под число символов `M` (от 1 до 255). Это необязательное указание количества выводимых символов используется для дополнения пробелами слева от выводимых значений символов, меньших, чем заданная ширина столбца. Однако ограничений ни на диапазон величин, ни на количество разрядов не налагается. Если число символов, необходимых для вывода числа, превышает `M`, под столбец будет выделено больше символов. Если дополнительно указан необязательный атрибут `ZEROFILL`, свободные позиции по умолчанию заполняются нулями слева. Например, для столбца, объявленного как `INT(5) ZEROFILL`, величина 4 отображается как 00004.

Тип `BIT[(M)]` предназначен для хранения битовых полей. Параметр `M` указывает число битовых значений, которое может принимать поле (от 1 до 64). Если параметр `M` не указан, то по умолчанию принимает значение 1.

Замечание

Тип `BIT` добавлен в MySQL, начиная с версии 5.0.3.

Тип `BOOLEAN` является синонимом для `TINYINT(1)`. Значение 1 рассматривается как истина (`true`), а 0 как ложь (`false`).

Тип `DECIMAL`, а также его синонимы `NUMERIC` и `DEC` предназначены для величин повышенной точности, например для денежных данных. Требуемая точность задается при объявлении столбца данных одного из этих типов, например:

```
salary DECIMAL(5,2)
```

В этом примере цифра 5 определяет общее число символов, отводимых под число, а цифра 2 задает количество знаков после запятой. Следовательно, в этом случае интервал величин, которые могут храниться в столбце `salary`, составляет от -99,99 до 99.99 (в действительности для данного столбца MySQL обеспечивает возможность хранения чисел вплоть до 999.99, поскольку допускается не хранить знак для положительных чисел).

Замечание

Первый параметр `M` может принимать максимальное значение, равное 64, второй параметр `D` — 30.

Величины типов DECIMAL, DEC и NUMERIC хранятся как строки, а не как двоичные числа с плавающей точкой, чтобы сохранить точность представления этих величин в десятичном виде. Если второй параметр равен 0, то величины DECIMAL и NUMERIC не содержат десятичного знака или дробной части.

Для представления вещественных типов в СУБД MySQL имеется три типа: FLOAT, DOUBLE и DECIMAL.

Замечание

Тип DOUBLE имеет два синонима: PRECISION и REAL. Тип DECIMAL имеет синоним NUMERIC.

Характеристики приближенных типов представлены в табл. 4.2.

Таблица 4.2. *Приближенные типы*

Тип	Объем памяти	Диапазон
FLOAT[(M,D)]	4 байта	Минимальное значение $\pm 1.175494351 \cdot 10^{-39}$ Максимальное значение $\pm 3.402823466 \cdot 10^{38}$
DOUBLE[(M,D)], REAL[(M,D)], DOUBLE PRECISION[(M,D)]	8 байт	Минимальное значение $\pm 2.2250738585072014 \cdot 10^{-308}$ Максимальное значение $\pm 1.797693134862315 \cdot 10^{308}$

Диапазон вещественных чисел помимо максимального значения имеет также минимальное значение, которое характеризует точность данного типа. Параметр M в табл. 4.2 задает число символов для отображения всего числа, а D — для его дробной части.

Числовые типы данных с плавающей точкой также могут иметь параметр UNSIGNED. Как и в целочисленных типах, этот атрибут предотвращает хранение в отмеченном столбце отрицательных величин, но, в отличие от целочисленных типов, максимальный интервал для величин столбца остается прежним.

Замечание

Приближенные числовые данные могут задаваться в обычной форме, например 45.67, и в так называемой научной нотации, например 5.456E-02 или 4.674E+04. Символ E указывает на то, что число перед ним следует умножить на 10 в степени, указанной после E. То есть приведенные в примерах числа можно записать как 0.05456 и 46740. Такой формат введен для удобства записи, т. к. числа в 300 степени неудобно представлять в обычном десятичном виде.

При выборе столбцов для формирования структуры таблицы необходимо обращать внимание на размер, занимаемый тем или иным типом данных: если значения, раз-

мещаемые в базе данных, никогда не будут выходить за пределы 100, не следует выбирать тип больше TINYINT. Если же в полях столбца предполагается хранить только целочисленные данные, то применение атрибута UNSIGNED позволит увеличить диапазон в два раза.

В листинге 4.15 представлен оператор CREATE TABLE, создающий таблицу tbl с двумя полями:

- num1 — целочисленное поле типа INT;
- num2 — поле для вещественных чисел, тип FLOAT;

Листинг 4.15. Использование оператора CREATE

```
CREATE TABLE tbl (
num1 INT DEFAULT 0,
num2 FLOAT DEFAULT 1.0
)
```

Как видно из листинга 4.15, после типа столбца указано ключевое слово DEFAULT. Оно позволяет задать значение поля по умолчанию, так, для поля num1 значение по умолчанию выбрано 0, а для num2 — 1.0. Ключевое слово DEFAULT является необязательным и может не указываться, т. е. вполне допустимо создание таблицы при помощи запроса, приведенного в листинге 4.16.

Листинг 4.16. Использование оператора CREATE без ключевого слова DEFAULT

```
CREATE TABLE tbl (
num1 INT,
num2 FLOAT
)
```

4.3.2. Строковые данные

Для строковых типов данных максимальный размер и требования к памяти приведены в табл. 4.3. Здесь L — это длина хранимой в ячейке строки, а байты, приплюсованные к L, — накладные расходы для хранения длины строки.

Таблица 4.3. *Строковые типы данных*

Тип	Объем памяти	Максимальный размер
CHAR(M)	M символов	M символов
VARCHAR(M)	L+1 символов	M символов
TINYBLOB, TINYTEXT	L+1 символов	2^8-1 символов

Таблица 4.3 *(окончание)*

Тип	Объем памяти	Максимальный размер
BLOB, TEXT	L+2 символов	2^{16}-1 символов
MEDIUMBLOB, MEDIUMTEXT	L+3 символов	2^{24}-1 символов
LONGBLOB, LONGTEXT	L+4 символов	2^{32}-1 символов
ENUM('value1','value2',...)	1 или 2 байта	65535 элементов
SET('value1','value2',...)	1, 2, 3, 4 или 8 байт	64 элемента

Тип CHAR позволяет хранить строку фиксированной длины M, его дополняет тип VARCHAR, позволяющий хранить переменные строки длиной L. Значение M может принимать значения от 0 до 65535.

Замечание

До версии MySQL 5.0.3 значение M для типов CHAR и VARCHAR могло принимать значения только от 0 до 255.

При выборе строкового типа данных для столбца следует принимать во внимание, что для переменных строк VARCHAR требуется количество символов, равное длине строки плюс один байт, в то время как тип CHAR(M), независимо от длины строки, использует для ее хранения все M символов. В то же время тип CHAR обрабатывается эффективнее переменных типов, т. к. всегда заранее известно, где заканчивается очередной блок данных. Сравнение типов CHAR и VARCHAR приведено в табл. 4.4.

Таблица 4.4. *Сравнение типов CHAR и VARCHAR*

Значение	CHAR(4)		VARCHAR(4)	
	представление	число байт	представление	число байт
''	' '	4 байта	''	1 байт
'ab'	' ab'	4 байта	'ab'	3 байта
'abcd'	'abcd'	4 байта	'abcd'	5 байт
'abcdefgh'	'abcd'	4 байта	'abcd'	5 байт

При создании таблицы нельзя смешивать столбцы типа CHAR и VARCHAR. Если такое происходит, СУБД MySQL изменит тип столбцов согласно правилу: в случае, когда в таблице присутствует хоть один столбец переменной длины, все столбцы типа CHAR приводятся к типу VARCHAR.

Замечание

Начиная с версии 4.1.2, типы `CHAR` и `VARCHAR` рассматривают строки как последовательности символов. Это означает, что при использовании многобайтных кодировок, например UNICODE, размер строки в байтах будет больше, чем в символах. Для совместимости со старыми версиями в MySQL введены два специальных типа: `BINARY` и `VARBINARY`, которые эквивалентны типам `CHAR` и `VARCHAR`, но строка в них рассматривается как последовательность байтов, а не символов. К `BINARY`-строкам не применимы кодировки и сортируются они как обычные последовательности байтов.

Типы `BLOB` и `TEXT` в СУБД MySQL во всем аналогичны и отличаются только в деталях. Например, при выполнении операций над столбцами типа `TEXT` учитывается кодировка, а типа `BLOB` — нет.

Тип `TEXT` обычно используется для хранения больших объемов текста, в то время как `BLOB` — для больших двоичных объектов, таких как электронные документы, изображения, звуки и т. д.

Замечание

Основным отличием типа `TEXT` от `CHAR` и `VARCHAR` является поддержка возможностей полнотекстового поиска, которые рассматриваются в *главе 18*.

К особым типам данных относятся `ENUM` и `SET`. Строки этих типов принимают значения из заранее заданного списка допустимых значений. Основное различие между ними заключается в том, что значение типа `ENUM` должно содержать точно одно значение из указанного множества, тогда как столбцы `SET` могут содержать любой или все элементы заранее заданного множества одновременно. Так, значения для столбца, объявленного как `ENUM('y','n')`, могут принимать только два значения: либо `'y'`, либо `'n'`.

Для типа `SET`, так же как и для типа `ENUM`, при объявлении задается список возможных значений, но ячейка может принимать любое значение из списка, а пустая строка означает, что ни один из элементов списка не выбран. Например, значения для столбца `SET('y','n')` могут принимать значения `('y','n')`, `('y')`, `('n')` и пустое множество `()`.

Типы `ENUM` и `SET` можно назвать строковыми лишь отчасти, т. к. при объявлении они задаются списком строк, но во внутреннем представлении базы данных элементы множеств сохраняются в виде чисел. Элементы типа `ENUM` нумеруются последовательно, начиная с 1. В зависимости от числа элементов в списке под столбец может отводиться 1 байт (до 256 элементов в списке) или 2 байта (от 257 до 65536 элементов в списке).

Элементы множества `SET` обрабатываются как биты, размер типа при этом также определяется числом элементов в списке: 1 байт (от 1 до 8 элементов), 2 байта (от 9 до 16 элементов), 3 байта (от 17 до 24 элементов), 4 байта (от 25 до 32 элементов) и 8 байт (от 33 до 64 элементов).

В листинге 4.17 представлен оператор `CREATE TABLE`, создающий таблицу `tbl` с тремя полями:

- text_field — текстовое поле типа TEXT;
- enum_field — поле типа ENUM, которое может принимать одно из трех значений: 'first', 'second' или 'third';
- set_field — поле типа SET, которое может принимать одну из комбинаций множества ('a','b','c','d').

Листинг 4.17. Создание таблицы tbl2 с тремя полями

```
CREATE TABLE tbl2
(
  text_field TEXT DEFAULT '',
  enum_field ENUM('first','second','third') DEFAULT 'first',
  set_field SET('a','b','c','d') DEFAULT 'a,b,c'
)
```

Как видно из листинга 4.17, в качестве значения по умолчанию для поля text_field используется пустая строка, для поля enum_field значение 'first', а для поля set_field список 'a,b,c'.

4.3.3. Календарные данные

СУБД MySQL имеет 5 видов столбцов для хранения календарных типов данных: DATE, DATETIME, TIME, TIMESTAMP и YEAR (табл. 4.5). Тип DATE предназначен для хранения даты, TIME для времени суток, а TIMESTAMP для представления и даты, и времени суток. Тип TIMESTAMP предназначен для представления даты и времени суток в виде числа секунд, прошедших с полуночи 1 января 1970 года. Тип данных YEAR позволяет хранить только год.

Таблица 4.5. *Календарные типы данных*

Тип	Объем памяти	Диапазон
DATE	3 байта	от '1000-01-01' до '9999-12-31'
TIME	3 байта	от '-828:59:59' до '828:59:59'
DATATIME	8 байт	от '1000-01-01 00:00:00' до '9999-12-31 00:00:00'
TIMESTAMP[(M)]	4 байта	от '1970-01-01 00:00:00' до '2038-12-31 59:59:59'
YEAR[(M)]	1 байт	от 1901 до 2155 для YEAR(4) от 1970 до 2069 для YEAR(2)

Для значений, имеющих тип DATE и DATETIME, в качестве первой цифры ожидается год либо в формате "YYYY", например '2005-10-15', либо в формате "YY", например

'05-10-15'. После года через дефис указывается месяц в формате "MM" (10), а затем день в формате "DD" (15).

Замечание

На протяжении всей книги мы будем обозначать формат даты как YYYY-MM-DD для дат вида 2005-10-15 и формат YY-MM-DD для дат вида 05-10-15. Для указания времени будет использоваться формат hh:mm:ss, где hh — часы, mm — минуты, а ss — секунды, например 10:48:56.

В типах `TIME` и `DATETIME` время приводится в привычном формате hh:mm:ss, где hh — часы, mm — минуты, а ss — секунды. Дни, месяцы, часы, минуты и секунды можно записывать как с ведущим нулем: 01, так и без него: 1. Например, все следующие записи идентичны:

```
'2005-04-06 02:04:08'
'2005-4-06 02:04:08'
'2005-4-6 02:04:08'
'2005-4-6 2:04:08'
'2005-4-6 2:4:08'
'2005-4-6 2:4:8'
```

В качестве разделителя между годами, месяцами, днями, часами, минутами, секундами может выступать любой символ, отличный от цифры. Так, следующие значения идентичны:

```
'05-12-31 11:30:45'
'05.12.31 11+30+45'
'05/12/31 11*30*45'
'05@12@31 11^30^45'
```

Дата и время суток могут также быть представлены в форматах 'YYYYMMDDhhmmss' и 'YYMMDDhhmmss'. Например, строки '2005091528' и '050523091528' аналогичны '2005-05-23 09:15:28', однако строка '051122129015' уже не может рассматриваться как дата и время суток, т. к. значение для минут равно 90 и выходит за допустимый интервал. Вместо строк допустимы и целочисленные значения, например, 2005091528 и 0523091528 рассматриваются как '2005-05-23 09:15:28'.

Начиная с версии MySQL 4.1.1, при указании времени суток после секунд через точку можно также указать микросекунды, т. е. использовать расширенный формат вида 'hh:mm:ss.ffffff', например '10:25:14.000001'. Кроме расширенного формата, можно использовать краткие форматы 'HH:MM' и 'HH' — вместо пропущенных величин будут подставлены нулевые значения.

Если время задается в недопустимом формате, то в поле записывается нулевое значение. Нулевое значение присваивается полям временного типа по умолчанию, когда им не присваивается инициирующее значение (табл. 4.6).

Формат типа `TIMESTAMP` совпадает с `DATETIME`, но во внутреннем представлении дата хранится в виде секунд, прошедших с полуночи 1 января 1970 года.

Таблица 4.6. *Календарные типы*

Тип	Нулевое значение
DATE	'0000-00-00'
TIME	'00:00:00'
DATATIME	'0000-00-00 00:00:00'
TIMESTAMP	00000000000000
YEAR	0000

Замечание

Время в данном формате хранится как число секунд, прошедшее с полуночи 1 января 1970 года. Такое исчисление принято в операционной системе UNIX, а дата 01.01.1970 часто называется "Началом эпохи UNIX" и считается днем рождения данной операционной системы.

Кроме того, поле TIMESTAMP может автоматически получать значения текущей даты и времени суток при создании новой записи и изменении уже существующей оператором UPDATE.

Замечание

В версиях MySQL 4.0 и более ранних формат типа TIMESTAMP представлял собой число YYYYMMDDhhmmss. Начиная с версии 4.1, формат представления был изменен на YYYY-MM-DD hh:mm:ss.

Если в таблице несколько столбцов TIMESTAMP, при модификации записи текущее время будет записываться только в один из столбцов (по умолчанию первый). Можно также указать явно столбец, которому необходимо назначать текущую дату при создании новой записи или изменении старой. Для того чтобы поля столбца принимали текущую дату при создании новой записи, следует после определения столбца добавить запись DEFAULT CURRENT_TIMESTAMP. Если требуется, чтобы текущее время выставлялось при модификации уже существующей записи, при использовании оператора UPDATE следует добавить конструкцию ON UPDATE CURRENT_TIMESTAMP. Так, три оператора CREATE TABLE, представленные в листинге 4.18, абсолютно идентичны.

Листинг 4.18. Задание временных параметров при создании таблицы

```
CREATE TABLE tbl(puttime TIMESTAMP);
CREATE TABLE tbl(puttime TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                                   ON UPDATE CURRENT_TIMESTAMP);
CREATE TABLE tbl(puttime TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
                                   DEFAULT CURRENT_TIMESTAMP);
```

Для столбца TIMESTAMP можно указать только одно из ключевых слов: либо только DEFAULT CURRENT_TIMESTAMP, либо только ON UPDATE CURRENT_TIMESTAMP.

Листинг 4.19. Задание текущего времени

```
CREATE TABLE tbl(
  puttime1 TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  puttime2 TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

В листинге 4.19 поле puttime1 получает текущее время при создании новой записи, а поле puttime2 при ее модификации.

Столбцы типа YEAR предназначены для хранения только года, указание параметра M позволяет задать формат года. Так, для YEAR(2) год представляется в виде двух чисел "YY", например 05 или 79, с диапазоном данных от 1970 года до 2069, а тип YEAR(4) позволяет задать тип в виде четырех чисел "YYYY", например 2005 или 1997 с диапазоном от 1901 до 2155 года. Если параметр M не указан, по умолчанию считается, что он равен 4.

4.3.4. Тип данных *NULL*

Реляционная база данных позволяет объединить многочисленные данные в одну таблицу и при помощи SQL-запросов проводить над ней различные манипуляции, получая результат в виде чисел и строк, а также новых таблиц. При создании таблицы неизбежны случаи, когда информации недостаточно и для части данных нельзя определить, какое значение они примут. Такие данные обозначаются специальным типом — NULL.

Например, в форме регистрации пользователь обязательно должен указать фамилию, имя, отчество. Кроме того, по желанию он может указать свой e-mail и URL домашней страницы. Для регистрации посетителей создается таблица users, оператор CREATE TABLE которой представлен в листинге 4.20.

Листинг 4.20. Создание таблицы users

```
CREATE TABLE users
(
  id_user INT,                                  # уникальный номер
  surname VARCHAR(255),                         # фамилия
  name VARCHAR(255),                            # имя
  patronymic VARCHAR(255),                      # отчество
  email VARCHAR(255),                           # e-mail
  url VARCHAR(255)                              # адрес домашней страницы
)
```

Замечание

Комментарий в SQL начинается с двух дефисов "--", все что расположено правее, считается комментарием. Непосредственно после "--" должен следовать пробел. Помимо стандартного комментария, MySQL содержит ряд собственных комментариев. Shell-комментарий # действует аналогично "--", все что расположено правее его, является текстом комментария. С-комментарий /* */ является многострочным — комментарий начинается с "/*" и заканчивается только тогда, когда встретится завершение "*/".

Посетитель обязан указать свои фамилию, имя и отчество, но может не указывать e-mail и адрес домашней страницы, даже несмотря на то, что они у него имеются в наличии. Таким образом, если для полей `email` и `url` нет информации, это не значит, что ее нет в природе, просто на данный момент она неизвестна. Такие поля принимают значение `NULL` — отсутствие информации, т. е. неопределенное значение. Выполнение арифметических операций с данными типа `NULL` всегда дает `NULL`, передача `NULL` в качестве аргумента функции также всегда приводит к значению типа `NULL`. Любые действия, производимые над неопределенным значением, приводят опять к неопределенному значению.

Для указания того факта, что поле может принимать значение `NULL`, в определении столбца, после типа данных следует указать ключевое слово `NULL`, если поле ни при каких обстоятельствах не должно принимать поле `NULL` (регистрация невозможна, если фамилия пользователя неизвестна), следует указать ключевое слово `NOT NULL` (листинг 4.21).

Листинг 4.21. Альтернативное определение таблицы `users`

```
CREATE TABLE users
(
  id_user INT NOT NULL,                          # уникальный номер
  surname VARCHAR(255) NOT NULL,                 # фамилия
  name VARCHAR(255) NOT NULL,                    # имя
  patronymic VARCHAR(255) NOT NULL,              # отчество
  email VARCHAR(255) NULL,                       # e-mail
  url VARCHAR(255) NULL                          # адрес домашней страницы
)
```

Передача значения `NULL` первым четырем столбцам таблицы `users` приведет к возникновению ошибки. Если при создании новой записи поля не инициируются и им будут присвоены значения по умолчанию, то `id_user` получит значение 0, `surname`, `name` и `patronymic` будет присвоена пустая строка, а `email` и `url` — значение `NULL`.

Замечание

Атрибут `NOT NULL` можно не указывать, т. к. он присваивается столбцу по умолчанию, если никакой из атрибутов не указан.

Совместно с атрибутами `NOT NULL` и `NULL` можно использовать `DEFAULT`, который имеет больший приоритет (листинг 4.22).

Листинг 4.22. Совместное использование типов NULL и NOT NULL с DEFAULT

```
CREATE TABLE users
(
  id_user INT NOT NULL DEFAULT 1,                # уникальный номер
  surname VARCHAR(255) NOT NULL DEFAULT 'Ф',     # фамилия
  name VARCHAR(255) NOT NULL DEFAULT 'И',        # имя
  patronymic VARCHAR(255) NOT NULL DEFAULT 'О', # отчество
  email VARCHAR(255) NULL DEFAULT 'None',        # e-mail
  url VARCHAR(255) NULL                          # адрес домашней страницы
)
```

Если при создании новой записи поля не инициируются и им будут присвоены значения по умолчанию, то `id_user` получит значение 1, `surname` — 'Ф', `name` — 'И', `patronymic` — 'О', `email` — 'None', а `url` — `NULL`. При этом по-прежнему поля `id_user`, `surname`, `name` и `patronymic` не могут принимать значение `NULL`, а для `email` и `url` это допустимо.

Применив к таблице `users` оператор `DESCRIBE`, получим результат, представленный в листинге 4.23.

Листинг 4.23. Структура таблицы users

```
mysql> DESCRIBE users;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| id_user    | int(11)      | NO   |     | 1       |       |
| surname    | varchar(255) | NO   |     | Ф       |       |
| name       | varchar(255) | NO   |     | И       |       |
| patronymic | varchar(255) | NO   |     | О       |       |
| email      | varchar(255) | YES  |     | None    |       |
| url        | varchar(255) | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
```

Третье поле результирующей таблицы (`Null`) показывает, какие поля могут принимать значение `NULL` (`YES`), а какие нет (`NO`). В пятом поле (`Default`) приводятся значения полей по умолчанию. Назначение поля `Key` будет объяснено в *главе 5*.

4.3.5. Выбор типа данных

При выборе типа данных следует помнить, что обработка числовых данных происходит быстрее строковых. Так как типы данных `ENUM` и `SET` имеют внутреннее числовое представление, им следует отдавать предпочтение перед другими видами строковых данных, если предоставляется такая возможность.

Производительность можно увеличить за счет представления строк в виде чисел. Примером может служить преобразование IP-адреса из строковой нотации в `BIGINT`.

Это позволит уменьшить размер таблицы и значительно увеличить скорость при сортировке и выборке данных, но потребует дополнительных преобразований.

Не следует забывать, что базы данных хранятся на жестком диске, и чем меньше места они занимают, тем быстрее происходит поиск и извлечение. Поэтому, там где есть возможность, предпочтение следует отдавать типам данных, занимающих меньше места.

Типы фиксированной длины обрабатываются быстрее типов переменной длины, т. к. в последнем случае при частых удалениях и модификациях таблицы происходит ее фрагментация.

Замечание

Если применение столбцов с данными переменной длины неизбежно, для дефрагментации таблицы следует применять команду `OPTIMIZE TABLE`, синтаксис которой подробно обсуждается в *главе 26*.

4.4. Учебная база данных

На протяжении всей книги описание особенностей диалекта SQL в СУБД MySQL будет вестись на примере учебной базы данных. В качестве такой базы данных выбрана база компьютерного магазина, торгующего комплектующими.

Замечание

На компакт-диске, поставляемом вместе с данной книгой, можно найти как текстовый дамп учебной базы данных \base\base.sql, так и бинарные файлы базы данных \shop\shop. О том, как подключить данные файлы к серверу MySQL, рассказывается в *главе 3*.

База данных компьютерного магазина будет состоять из четырех таблиц:

- `catalogs` — список торговых групп;
- `products` — товарные позиции;
- `users` — список зарегистрированных пользователей магазина;
- `orders` — список осуществленных сделок.

Таблица `catalogs` предназначена для хранения торговых групп, таких как "Материнские платы", "Процессоры", "Видеокарты" и т. п. Таблица состоит из двух полей:

- `id_catalog` — уникальный номер;
- `name` — имя раздела.

Оператор `CREATE TABLE`, создающий таблицу `catalogs`, представлен в листинге 4.24. Оба поля снабжены атрибутом `NOT NULL`, т. к. неопределенное значение для них недопустимо.

Листинг 4.24. Создание таблицы catalogs

```
CREATE TABLE catalogs (
  id_catalog INT(11) NOT NULL,
  name TINYTEXT NOT NULL
)
```

Таблица products содержит конкретные товарные позиции, такие как "Celeron 2.0GHz", "Intel Pentium 4 3.2GHz" и т. п. Таблица состоит из семи полей:

- id_product — уникальный номер товарной позиции;
- name — название товарной позиции;
- price — цена;
- count — количество товарных позиций на складе;
- mark — относительная оценка товара;
- description — описание;
- id_catalog — номер торговой группы из таблицы id_catalog, которой принадлежит товарная позиция.

Оператор CREATE TABLE, создающий таблицу products, представлен в листинге 4.25.

Листинг 4.25. Создание таблицы products

```
CREATE TABLE products (
  id_product INT(11) NOT NULL,
  name TINYTEXT NOT NULL,
  price DECIMAL(7,2) NULL default '0.00',
  count INT(11) NULL default '0',
  mark FLOAT(4,1) NOT NULL default '0.0',
  description TEXT NULL,
  id_catalog INT(11) NOT NULL
)
```

Цена товарной позиции price, количество товара на складе count и описание description снабжены атрибутом NULL. Несмотря на то, что в момент привоза зачастую бывает неизвестно количество товара на складе и его текущая цена, отразить факт наличия товара в прайс-листе необходимо. Точно так же и описание description может быть неизвестно из-за того, что товар слишком новый и имеет противоречивое описание в источниках.

Поле id_catalog устанавливает связь между таблицами catalog и products. Зная уникальный номер каталога id_catalog, всегда можно узнать, какие товарные позиции принадлежат данной товарной группе и наоборот, к какой товарной группе принадлежит товарная позиция с конкретным значением id_catalog. MySQL позволяет сделать это при помощи несложных SQL-запросов, которые будут рассмотрены в следующих главах.

Таблица users содержит записи с информацией о зарегистрированных покупателях и состоит из восьми полей:

- id_user — уникальный номер покупателя;
- surname — фамилия покупателя;
- patronymic — отчество покупателя;
- name — имя покупателя;
- phone — телефон покупателя (если имеется);
- email — e-mail покупателя (если имеется);
- url — домашняя страница покупателя (если имеется);
- userstatus — статус покупателя, поле типа ENUM, которое может принимать одно из четырех значений:
  - active — авторизованный покупатель, который может осуществлять покупки через Интернет;
  - passive — неавторизованный покупатель (значение по умолчанию), который осуществил процедуру регистрации, но не подтвердил ее и пока не может осуществлять покупки в магазине через Интернет, но ему доступны каталоги для просмотра;
  - lock — заблокированный покупатель, не может осуществлять покупки и просматривать каталоги магазина;
  - gold — активный покупатель с хорошей кредитной историей, которому предоставляется скидка при следующих покупках в магазине.

Оператор CREATE TABLE, создающий таблицу users, представлен в листинге 4.26.

Листинг 4.26. Создание таблицы users

```
CREATE TABLE users (
  id_user INT(11) NOT NULL,
  surname TINYTEXT,
  patronymic TINYTEXT,
  name TINYTEXT,
  phone VARCHAR(12) NULL,
  email TINYTEXT NULL,
  url TINYTEXT NULL,
  userstatus ENUM('active','passive','lock','gold') DEFAULT 'passive'
)
```

Как видно из листинга 4.26, поля phone, email и url снабжены атрибутом NULL, т. к. посетитель имеет право не указывать эту информацию. Если эта информация не указана, то она носит неопределенный характер: телефона, e-mail и домашней страницы у покупателя может просто не быть, с другой стороны, он мог не захотеть их указывать. Поэтому, чтобы указать неопределенность этой информации, в поле следует

поместить значение NULL. Все остальные поля получают атрибут NOT NULL, даже если он не указывается.

Таблица orders содержит информацию о покупках, совершенных в магазине, и включает пять полей:

- id_order — уникальный номер сделки;
- id_user — номер пользователя из таблицы users;
- ordertime — время совершения сделки;
- number — число приобретенных товаров;
- id_product — номер товарной позиции из таблицы products.

Оператор CREATE TABLE, создающий таблицу orders, представлен в листинге 4.27.

Листинг 4.27. Создание таблицы orders

```
CREATE TABLE orders (
  id_order INT(11) NOT NULL,
  id_user INT(11) NOT NULL,
  ordertime DATETIME NOT NULL,
  number INT(11) NOT NULL,
  id_product INT(11) NOT NULL
)
```

Как видно из листинга 4.27, все столбцы таблицы orders снабжены атрибутом NOT NULL, т. к. при совершении покупки вся информация должна быть доступна и занесена в таблицу.

В таблице orders устанавливается связь сразу с двумя таблицами: users, за счет поля id_user, и products, за счет поля id_product. Это позволит для каждой покупки восстановить покупателя и приобретенный товар. С другой стороны, можно узнать число покупок, совершенных каждым из покупателей, а также частоту покупки той или иной товарной позиции, что позволит увеличить или уменьшить закупки товара и тем самым произвести оптимизацию работы магазина.

ГЛАВА 5

Индексы

Индексы играют большую роль в базах данных, т. к. это основной способ ускорения их работы. Обычно записи в таблице располагаются в хаотическом порядке (рис. 5.1). Для того чтобы найти нужную запись, необходимо сканировать всю таблицу, на что уходит большое количество времени. Идея индексов состоит в том, чтобы создать для столбца копию, которая постоянно будет поддерживаться в отсортированном состоянии. Это позволяет очень быстро осуществлять поиск по такому столбцу, т. к. заранее известно, где необходимо искать значение.

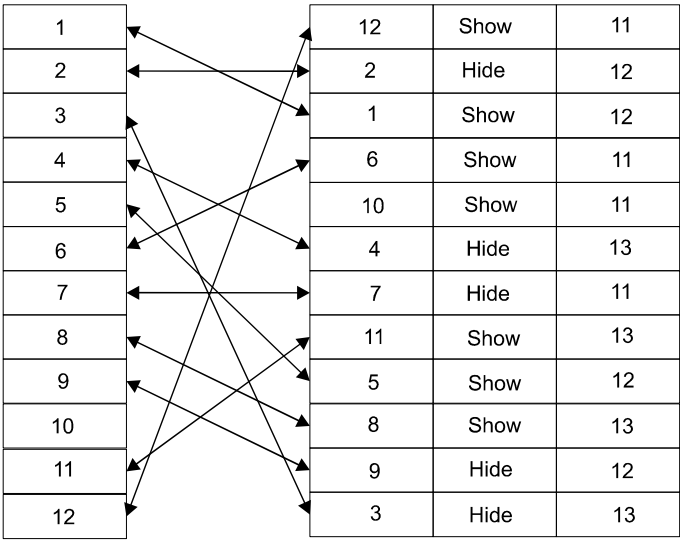

Рис. 5.1. Демонстрация работы индексированных столбцов

Обратной стороной медали является то, что добавление или удаление записи требует дополнительного времени на сортировку столбца, кроме того, создание копии увеличивает объем памяти, необходимый для размещения таблицы на жестком диске.

Существует несколько видов индексов.

- Первичный ключ — главный индекс таблицы. В таблице может быть только один первичный ключ, и все значения такого индекса должны отличаться друг от друга, т. е. являться уникальными.
- Обычный индекс — таких индексов может быть несколько.
- Уникальный индекс — уникальных индексов также может быть несколько, но значения индекса не должны повторяться.
- Полнотекстовый индекс — специальный вид индекса для столбцов типа `TEXT`, позволяющий производить полнотекстовый поиск, подробно описывается в *главе 18*.

Индексы могут иметь как свои собственные имена, так и имена индексируемых столбцов. Один индекс может охватывать один или несколько столбцов, причем тип столбца может быть любым, за исключением `ENUM` и `SET`.

Замечание

В таблице может быть не более 32 индексов, а многостолбцовый индекс может охватывать не более 16 столбцов.

5.1. Первичный ключ

Так как записи в реляционной таблице не упорядочены, нельзя выбрать строку по ее номеру (как в массиве), поэтому для идентификации записи в таблицу вводится так называемый первичный ключ. Первичный ключ является главным индексом таблицы и объявляется при помощи ключевого слова `PRIMARY KEY`, у таблицы может быть только один первичный ключ. Значение первичного ключа должно быть уникально и не повторяться в пределах таблицы. Кроме того, столбцы, помеченные атрибутом `PRIMARY KEY`, не могут принимать значение `NULL`. Для пометки поля таблицы в качестве первичного ключа достаточно поместить это ключевое слово `PRIMARY KEY` в определение столбца. В листинге 5.1 приведено определение таблицы `catalogs` (см. листинг 4.24), где в качестве первичного ключа назначен столбец `id_catalog`.

Замечание

Первичный ключ не является обязательной конструкцией, вполне возможно создание таблиц, не содержащих первичных ключей и индексов.

Листинг 5.1. Создание таблицы `catalogs`

```
CREATE TABLE catalogs (
  id_catalog INT(11) NOT NULL PRIMARY KEY,
  name TINYTEXT NOT NULL
)
```

Просмотр структуры таблицы `catalogs` при помощи оператора `DESCRIBE` показывает, что в четвертой колонке результирующей таблицы напротив поля `id_catalog` появился флаг `PRI`, который отмечает поле первичного ключа (листинг 5.2).

Листинг 5.2. `DESCRIBE catalogs`

```
mysql> DESCRIBE catalogs;
+------------+----------+------+-----+---------+-------+
| Field      | Type     | Null | Key | Default | Extra |
+------------+----------+------+-----+---------+-------+
| id_catalog | int(11)  |      | PRI | 0       |       |
| name       | tinytext |      |     |         |       |
+------------+----------+------+-----+---------+-------+
```

Существует альтернативный способ объявления первичного ключа, представленный в листинге 5.3.

Листинг 5.3. Альтернативный способ объявления первичного ключа

```
CREATE TABLE catalogs (
  id_catalog INT(11) NOT NULL,
  name TINYTEXT NOT NULL,
  PRIMARY KEY(id_catalog)
)
```

Как видно из листинга 5.3, объявление первичного ключа производится после объявления основных столбцов.

Замечание

Ключевое слово `PRIMARY KEY` может встречаться в таблице только один раз, т. к. в таблице разрешен только один первичный ключ, в противном случае сервер MySQL вернет ошибку `#1068 - Multiple primary key defined`.

В качестве первичного ключа может выступать не только целочисленный столбец, но и текстовые данные. В этом случае необходимо обязательно добавить в круглых скобках число символов, входящих в индекс.

Листинг 5.4. Создание первичного ключа на текстовом столбце

```
CREATE TABLE catalogs (
  id_catalog INT(11) NOT NULL,
  name TINYTEXT NOT NULL,
  PRIMARY KEY(name(10))
)
```

Как видно из листинга 5.4, первичный ключ создается по первым 10 символам столбца `name`. Можно индексировать от 1 до 1000 символов текстового столбца. Следует помнить, что индексирование по одному символу приведет к тому, что уникальные символы быстро исчерпаются, и будет невозможно добавить новую запись в таблицу. Индексирование по большому числу символов приведет к резкому увеличению объема жесткого диска, необходимого для хранения таблицы.

Замечание

До версии 4.1.2 префикс не мог содержать более 255 символов.

Индекс необязательно должен быть объявлен по одному столбцу, вполне допустимо объявление индекса сразу по двум или более (до 16) столбцам (листинг 5.5).

Листинг 5.5. Создание многостолбцового первичного ключа

```
CREATE TABLE catalogs (
  id_catalog INT(11) NOT NULL,
  name TINYTEXT NOT NULL,
  PRIMARY KEY(id_catalog, name(10))
)
```

Просмотр структуры таблицы `catalogs` при помощи оператора `DESCRIBE` показывает, что в четвертой колонке результирующей таблицы оба поля: и `id_catalog`, и `name` имеют флаг `PRI`, который отмечает первичный ключ (листинг 5.6).

Листинг 5.6. `DESCRIBE catalogs`

```
mysql> DESCRIBE catalogs;
+------------+----------+------+-----+---------+-------+
| Field      | Type     | Null | Key | Default | Extra |
+------------+----------+------+-----+---------+-------+
| id_catalog | int(11)  | NO   | PRI |         |       |
| name       | tinytext | NO   | PRI |         |       |
+------------+----------+------+-----+---------+-------+
```

В качестве первичного ключа часто выступает столбец типа `INT` или `BIGINT`. В таблицах учебной базы данных `shop`, созданных в *главе 4*, все поля, начинающиеся с префикса `id_`, предназначены для первичного ключа. Такой выбор связан с тем, что целочисленные типы данных обрабатываются быстрее всех и занимают небольшой объем данных. Другой причиной выбора целочисленного столбца является тот факт, что только данный тип столбца может быть снабжен атрибутом `AUTO_INCREMENT`, который обеспечивает автоматическое создание уникального индекса. Передача столбцу, снабженному этим атрибутом, значения `NULL` или 0 приводит к автоматическому присвоению ему максимального значения столбца плюс 1. Данный механизм является достаточно удобным и позволяет не заботиться о генерации уникального значения средствами прикладной программы, работающей с СУБД MySQL. В листинге 5.7 приводится пример использования ключевого слова `AUTO_INCREMENT`.

Листинг 5.7. Использование атрибута `AUTO_INCREMENT`

```
CREATE TABLE catalogs (
  id_catalog INT(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
  name TINYTEXT NOT NULL
)
```

Просматривая структуру вновь созданной таблицы catalogs при помощи оператора DESCRIBE (листинг 5.8), можно заметить, что для столбца id_catalog в шестой колонке Extra результирующей таблицы появилась запись auto_increment, сообщающая, что столбец снабжен этим атрибутом.

Листинг 5.8. Результат использования атрибута AUTO_INCREMENT

```
mysql> DESCRIBE catalogs;
+------------+----------+------+-----+---------+----------------+
| Field      | Type     | Null | Key | Default | Extra          |
+------------+----------+------+-----+---------+----------------+
| id_catalog | int(11)  | NO   | PRI | NULL    | auto_increment |
| name       | tinytext | NO   |     |         |                |
+------------+----------+------+-----+---------+----------------+
```

Как видно из листинга 5.8, для столбца id_catalog в качестве значения по умолчанию выставлено NULL, однако само поле не может принимать значение NULL. Передача NULL приведет к генерации уникального числа, которое и будет занесено в id_catalog. Если в таблице нет ни одного значения, то первое уникальное значение будет равным 1. Точно так же, как и в таблице может быть только один первичный ключ, столбцов, снабженных атрибутом AUTO_INCREMENT, также не должно быть больше одного.

Замечание

Это значение можно изменить, более подробно об этом см. *главу 11*, раздел, посвященный оператору CREATE TABLE.

К атрибуту AUTO_INCREMENT мы еще вернемся в *главе 6* при рассмотрении оператора INSERT — вставки данных в таблицу.

5.2. Обычный и уникальный индексы

В отличие от первичного ключа, таблица может содержать несколько обычных и уникальных индексов, для того чтобы их различать, индексы могут иметь собственные имена. Часто имена индексов совпадают с именами столбцов, которые они индексируют, но для индекса можно назначить и совершенно другое имя. Объявление индекса производится при помощи ключевого слова INDEX или KEY. Для уникальных индексов, которые не позволяют добавлять в таблицу записи с уже существующими значениями, вводится дополнительное ключевое слово UNIQUE, т. е. пишется UNIQUE INDEX и UNIQUE KEY.

Замечание

Ключевое слово KEY введено в MySQL начиная с версии 4.1.0.

Замечание

Создание обычных индексов, в отличие от первичного ключа и уникального индекса, допустимо для столбцов, снабженных атрибутом NULL.

В листинге 5.9. приводится оператор `CREATE TABLE`, создающий таблицу `orders` (см. листинг 4.27) из учебной базы данных `shop`.

Листинг 5.9. Создание таблицы `orders`

```
CREATE TABLE orders (
  id_order INT(11) NOT NULL AUTO_INCREMENT,
  id_user INT(11) NOT NULL DEFAULT '0',
  ordertime DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00',
  number INT(11) NOT NULL DEFAULT '0',
  id_product INT(11) NOT NULL DEFAULT '0',
  PRIMARY KEY (id_order),
  KEY id_product (id_product),
  KEY id_user (id_user)
)
```

Как видно из листинга 5.9, в таблице `orders` вводятся два индекса по столбцам `id_product` и `id_user`. Имена индексов совпадают с именами индексируемых столбцов, но вполне допустимо присвоить им другие имена, например:

```
KEY first (id_product),
KEY second (id_user)
```

Замечание

В отличие от первичного ключа, для обычных индексов недопустимо их определение вместе со столбцом. Индекс объявляется отдельно в конце таблицы при помощи ключевого слова `INDEX` или `KEY`.

Просмотр структуры таблицы `orders` при помощи оператора `DESCRIBE` приводит к результату, представленному в листинге 5.10.

Листинг 5.10. Оператор `DESCRIBE orders`

```
mysql> DESCRIBE orders;
+------------+----------+------+-----+------------+----------------+
| Field      | Type     | Null | Key | Default    | Extra          |
+------------+----------+------+-----+------------+----------------+
| id_order   | int(11)  | NO   | PRI | NULL       | auto_increment |
| id_user    | int(11)  | NO   | MUL | 0          |                |
| ordertime  | datetime | NO   |     | 0000-00-00 |                |
| number     | int(11)  | NO   |     | 0          |                |
| id_product | int(11)  | NO   | MUL | 0          |                |
+------------+----------+------+-----+------------+----------------+
```

Как видно из листинга 5.10, столбцы, проиндексированные обычным индексом, помечаются в поле `Key` флагом `MUL`.

Замечание

Индексы, ссылающиеся на первичные ключи другой таблицы, такие как `id_user` и `id_product`, называют *внешними ключами*. Так `id_user` в таблице `orders` является внешним ключом таблицы `users`, а `id_products` внешним ключом таблицы `products`. Подробнее с внешними ключами можно познакомиться в *главе 23*.

Объявление уникальных индексов в таблице `orders` приводит к тому, что проиндексированные столбцы помечаются флагом `UNI` (листинг 5.11).

Листинг 5.11. Оператор `DESCRIBE orders`

```
mysql> DESCRIBE orders;
+------------+----------+------+-----+------------+----------------+
| Field      | Type     | Null | Key | Default    | Extra          |
+------------+----------+------+-----+------------+----------------+
| id_order   | int(11)  | NO   | PRI | NULL       | auto_increment |
| id_user    | int(11)  | NO   | UNI | 0          |                |
| ordertime  | datetime | NO   |     | 0000-00-00 |                |
| number     | int(11)  | NO   |     | 0          |                |
| id_product | int(11)  | NO   | UNI | 0          |                |
+------------+----------+------+-----+------------+----------------+
```

Точно так же, как и первичный ключ, обычные индексы можно объявить на нескольких столбцах (до 16), как это продемонстрировано в листинге 5.12.

Листинг 5.12. Объявление индекса на двух столбцах

```
CREATE TABLE orders (
  id_order INT(11) NOT NULL AUTO_INCREMENT,
  id_user INT(11) NOT NULL DEFAULT '0',
  ordertime DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00',
  number INT(11) NOT NULL DEFAULT '0',
  id_product INT(11) NOT NULL DEFAULT '0',
  PRIMARY KEY (id_order),
  KEY id_product (id_product, id_user)
)
```

Интересно отметить, что оператор `DESCRIBE` в этом случае будет считать проиндексированным только поле `id_product` (листинг 5.13).

Листинг 5.13. Результат объявления индекса на двух столбцах

```
mysql> DESCRIBE orders;
+------------+----------+------+-----+------------+----------------+
| Field      | Type     | Null | Key | Default    | Extra          |
+------------+----------+------+-----+------------+----------------+
| id_order   | int(11)  | NO   | PRI | NULL       | auto_increment |
| id_user    | int(11)  | NO   |     | 0          |                |
```

```
| ordertime  | datetime | NO   |     | 0000-00-00 |                |
| number     | int(11)  | NO   |     | 0          |                |
| id_product | int(11)  | NO   | MUL | 0          |                |
+------------+----------+------+-----+------------+----------------+
```

Часто решение об индексировании столбцов принимается после того, как таблица создана и заполнена данными. В этом случае прибегают к изменению структуры таблицы, создавая индекс при помощи оператора `CREATE INDEX` или удаляя его при помощи оператора `DROP INDEX`.

Замечание

Для создания уникального индекса используется оператор `CREATE UNIQUE KEY`, в то же время удаление уникального индекса производится при помощи оператора `DROP INDEX` без дополнительного ключевого слова `UNIQUE`, т. к. характер индекса при удалении не имеет значения.

Замечание

Изменение структуры таблицы, в том числе и индексов, производится при помощи оператора `ALTER TABLE`, который рассматривается в *главе 10*. Операторы `CREATE INDEX` и `DROP INDEX` здесь являются, скорее, исключениями из правила.

Замечание

Создание и удаление первичного ключа при помощи операторов `CREATE INDEX` и `DROP INDEX` невозможно и осуществляется только при помощи оператора `ALTER TABLE`.

В листинге 5.14 демонстрируется создание индекса с именем `id_catalog_index` для столбца `id_catalog` таблицы `products` (см. листинг 4.25) учебной базы данных `shop`, рассмотренной в предыдущей главе.

Листинг 5.14. Создание индекса `id_catalog_index` в таблице `products`

```
CREATE INDEX id_catalog_index ON products (id_catalog)
```

Если необходимо создать индекс по нескольким столбцам, они перечисляются в круглых скобках через запятую.

Оператор `DROP INDEX`, который удаляет вновь созданный индекс, представлен в листинге 5.15.

Листинг 5.15. Удаление индекса `id_catalog_index` в таблице `products`

```
DROP INDEX id_catalog_index ON products
```

Точно так же, как и в случае первичного ключа, допускается индексирование текстовых столбцов, если указано число индексируемых символов (листинг 5.16).

Листинг 5.16. Создание текстового индекса `name` на 10 символах столбца `name`

```
CREATE INDEX name ON catalogs (name(10))
```

Допускается создание индекса на текстовых столбцах с использованием от 1 до 1000 первых символов.

Начиная с версии 4.1, в операторе `CREATE INDEX` можно при помощи ключевого слова `USING` указать тип индекса. В настоящее время допустимо создание индексов типа `BTREE` и `HASH`. Тип `BTREE` можно создавать в таблицах MyISAM, InnoDB и MEMORY, тип `HASH` доступен только в таблицах типа MEMORY.

Замечание

Подробнее типы таблиц и их характеристики рассматриваются в *главе 11*. Создание индексов в таблицах, чей тип не предназначен для данного вида индекса, приведет к его игнорированию (предупреждений и сообщений об ошибоке выдаваться не будет).

Листинг 5.17. Создание индексов типа `BTREE` и `HASH`

```
CREATE INDEX id_btree USING BTREE ON tbl (id)
CREATE INDEX id_hash USING HASH ON tbl (id)
```

Главное отличие этих двух видов индексов заключается в том, что `BTREE`-индексы построены на b-деревьях, а `HASH`-индексы — на хэшировании. Сравнение алгоритмов работы этих индексов выходит за рамки данной книги.

ГЛАВА 6

Добавление данных

После успешного создания базы данных и таблиц перед разработчиком встает задача заполнения таблиц данными. Традиционно в реляционных базах данных для осуществления этой операции применяют три подхода:

- однострочный оператор `INSERT` — добавляет в таблицу данных новую запись;
- многострочный оператор `INSERT` — добавляет в таблицу данных несколько новых записей;
- пакетная загрузка данных — служит для добавления в таблицу данных из внешнего файла.

6.1. Однострочный оператор *INSERT*

Однострочный оператор `INSERT` может использоваться в нескольких формах. Упрощенный синтаксис первой формы выглядит следующим образом:

```
INSERT [INTO] tbl [(col_name,...)] VALUES (expression,...)
```

Данный оператор вставляет новую запись в таблицу `tbl`, значения записи перечисляются в списке `expression`, порядок следования столбцов может задаваться списком `col_name`. Значения передаются в списке после ключевого слова `VALUES`. Рассмотрим процесс вставки на примере таблицы `catalogs` (см. листинг 4.24). Таблица имеет два поля:

- `id_catalog` — первичный ключ таблицы `catalogs`, снабженный атрибутом `AUTO_INCREMENT`, значение по умолчанию `NULL`;
- `name` — название каталога, значение по умолчанию пустая строка.

> ***Замечание***
>
> Просмотр данных, помещенных в таблицу `catalogs`, осуществляется при помощи запроса `SELECT * FROM catalogs`, который обсуждается подробнее в *главе 7*.

Добавить новую запись в таблицу `catalogs` можно при помощи запроса, представленного в листинге 6.1.

Листинг 6.1. Добавление новой записи в таблицу catalogs

```
mysql> INSERT INTO catalogs VALUES (1, 'Процессоры');
mysql> SELECT * FROM catalogs;
+------------+------------+
| id_catalog | name       |
+------------+------------+
|          1 | Процессоры |
+------------+------------+
```

Как видно из листинга 6.1, в таблицу catalogs добавилась новая запись с первичным ключом id_catalogs, равным единице, и именем name — "Процессоры". Строковые значения необходимо помещать в кавычки, в то время как числовые значения допускается использовать без них.

Замечание

Использование кавычек с числовыми значениями приведет к уменьшению производительности, т. к. на преобразование типа MySQL потребуется затратить дополнительное время.

Список столбцов col_name, размещенный после имени таблицы, позволяет изменить порядок следования столбцов при добавлении.

Листинг 6.2. Изменение порядка следования столбцов

```
mysql> INSERT INTO catalogs (name, id_catalog)
    -> VALUES ('Материнские платы', 2);
mysql> SELECT * FROM catalogs;
+------------+-------------------+
| id_catalog | name              |
+------------+-------------------+
|          1 | Процессоры        |
|          2 | Материнские платы |
+------------+-------------------+
```

Как видно из листинга 6.2, порядок следования столбцов был изменен, сначала было добавлено название раздела name и лишь затем первичный ключ таблицы id_catalog. Следует помнить, что первичный ключ таблицы является уникальным значением и добавление уже существующего значения приведет к ошибке (листинг 6.3).

Листинг 6.3. Значения первичного ключа должны быть уникальными

```
mysql> INSERT INTO catalogs VALUES (2, 'Видеоадаптеры');
ERROR 1062: Duplicate entry '2' for key 1
```

Если необходимо, чтобы новые записи с дублирующим ключом отбрасывались без генерации ошибки, нужно добавить после оператора INSERT ключевое слово IGNORE.

Листинг 6.4. Использование ключевого слова IGNORE

```
mysql> INSERT IGNORE catalogs VALUES (1, 'Процессоры');
mysql> SELECT * FROM catalogs;
+------------+-------------------+
| id_catalog | name              |
+------------+-------------------+
|          1 | Процессоры        |
|          2 | Материнские платы |
+------------+-------------------+
```

Как видно из листинга 6.4, генерации ошибки не происходит, тем не менее новая запись также не добавляется.

При добавлении новой записи с уникальными индексами выбор такого уникального значения может быть непростой задачей. Для того чтобы не осуществлять дополнительный запрос, направленный на выявление максимального значения первичного ключа для генерации нового уникального значения, в MySQL введен механизм его автоматической генерации. Для этого достаточно снабдить первичный ключ `id_catalog` атрибутом `AUTO_INCREMENT` *(см. главу 5)*. Тогда при создании новой записи в качестве значения `id_catalog` достаточно передать `NULL` или 0 — поле автоматически получит значение, равное максимальному значению столбца `id_catalog`, плюс единица (листинг 6.5).

Листинг 6.5. Действие механизма AUTO_INCREMENT

```
mysql> INSERT INTO catalogs VALUES (NULL, 'Видеоадаптеры');
mysql> SELECT * FROM catalogs;
+------------+-------------------+
| id_catalog | name              |
+------------+-------------------+
|          1 | Процессоры        |
|          2 | Материнские платы |
|          3 | Видеоадаптеры     |
+------------+-------------------+
```

При вставке оператором `INSERT` не обязательно указывать все столбцы. Те столбцы, которые не указаны в списке, получат значение по умолчанию. Так, в листинге 6.6 значение `id_catalog` не передается оператору `INSERT`.

Листинг 6.6. Значение id_catalog отсутствует

```
mysql> INSERT INTO catalogs (name) VALUES ('Жесткие диски');
mysql> SELECT * FROM catalogs;
+------------+-------------------+
| id_catalog | name              |
+------------+-------------------+
|          1 | Процессоры        |
```

```
|           2 | Материнские платы |
|           3 | Видеоадаптеры     |
|           4 | Жесткие диски     |
+-------------+-------------------+
```

Так как id_catalog по умолчанию принимает значение NULL, которое вызывает механизм AUTO_INCREMENT, то поле id_catalog автоматически получает значение 5 (максимальное значение 4 плюс 1). Допускается добавление полностью пустых строк, как это продемонстрировано в листинге 6.7.

Листинг 6.7. Добавление пустых строк

```
mysql> INSERT INTO catalogs () VALUES ();
mysql> SELECT * FROM catalogs;
+-------------+-------------------+
| id_catalog | name               |
+-------------+-------------------+
|           1 | Процессоры        |
|           2 | Материнские платы |
|           3 | Видеоадаптеры     |
|           4 | Жесткие диски     |
|           5 |                   |
+-------------+-------------------+
```

Так как текстовое поле name по умолчанию принимает пустую строку, новая запись содержит пустую запись. Значение по умолчанию можно подставить, если передать в качестве значения ключевое слово DEFAULT (листинг 6.8).

Листинг 6.8. Использование ключевого слова DEFAULT

```
mysql> INSERT catalogs (id_catalog, name) VALUES (DEFAULT, DEFAULT);
mysql> SELECT * FROM catalogs;
+-------------+-------------------+
| id_catalog | name               |
+-------------+-------------------+
|           1 | Процессоры        |
|           2 | Материнские платы |
|           3 | Видеоадаптеры     |
|           4 | Жесткие диски     |
|           5 |                   |
|           6 |                   |
+-------------+-------------------+
```

Часто в прикладных программах требуется узнать значение, присвоенное столбцу и снабженное атрибутом AUTO_INCREMENT. Это может потребоваться для того, чтобы использовать сгенерированное значение первичного ключа в качестве вторичного в другой таблице или для каких-то других целей. Только что сгенерированное значение

возвращает встроенная функция MySQL `LAST_INSERT_ID()` *(см. раздел 20.6)*, как это показано в листинге 6.9.

Листинг 6.9. Получение только что добавленного значения

```
mysql> SELECT LAST_INSERT_ID();
+------------------+
| LAST_INSERT_ID() |
+------------------+
|                6 |
+------------------+
```

Например, если необходимо, чтобы следующее значение поля `id_catalog` приняло значение 16, а не 7, можно воспользоваться запросом, представленным в листинге 6.10.

Замечание

Отличительной чертой MySQL является то, что при использовании функций пробелы между именем функции и круглыми скобками недопустимы, т. е. написание `LAST_INSERT_ID()` является правильным, а `LAST_INSERT_ID ()` уже нет.

Замечание

Результат функции можно вывести на экран при помощи SQL-оператора `SELECT`, который подробнее обсуждается в *главе 7*.

Листинг 6.10. Использование функции `LAST_INSERT_ID()`

```
mysql> INSERT INTO catalogs (is_catalog, name)
    -> VALUES (LAST_INSERT_ID() + 10, 'Оперативная память');
mysql> SELECT * FROM catalogs;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 |                    |
|          6 |                    |
|         16 | Оперативная память |
+------------+--------------------+
```

Замечание

При выборе столбца, снабженного атрибутом `AUTO_INCREMENT`, у начинающих программистов часто возникает вопрос, что происходит, когда достигается граница типа. В этом случае возникает ошибка при добавлении новой записи в таблицу, т. к. уникальных значений в пределах типа не остается. На практике достичь данного

предела можно только для очень коротких типов, например TINYINT. Выбор типа INT и BIGINT даже при ежедневном добавлении в таблицу 10 000 новых записей приведет к исчерпанию уникальных значений только через 588 и 2 500 000 000 000 лет соответственно. При этом даже если таблица будет содержать только один столбец, занимаемый объем данных достигнет 8 Гбайт для типа INT и 67 108 864 Тбайт для типа BIGINT.

Кроме констант, в списке значений expression могут выступать выражения с участием столбцов из списка col_name.

Листинг 6.11. Использование выражений

```
mysql> INSERT catalogs (id_catalog, name) VALUES (7, id_catalog*3);
mysql> INSERT catalogs (id_catalog, name) VALUES (name*3, 7);
mysql> SELECT * FROM catalogs;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 |                    |
|          6 |                    |
|         15 | Оперативная память |
|          7 | 21                 |
|         16 | 6                  |
+------------+--------------------+
```

Так, в листинге 6.11 для первой записи поле id_catalog принимает значение 6, а поле name значение 21 (7×3). Ссылаться в выражении можно только на столбцы, расположенные левее, ссылка на правые столбцы дает 0 (в случае столбца id_catalog это привело к автоматической генерации нового уникального значения 16 вместо ожидаемых 21).

Замечание

Оператор * является оператором умножения, более подробно операторы описываются в *главе 13*.

Листинг 6.12. Использование выражений совместно с AUTO_INCREMENT

```
mysql> INSERT catalogs (id_catalog, name) VALUES (NULL, id_catalog*3);
mysql> SELECT * FROM catalogs;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
```

```
|           3 | Видеоадаптеры      |
|           4 | Жесткие диски      |
|           5 |                    |
|           6 |                    |
|          15 | Оперативная память |
|           7 | 21                 |
|          16 | 6                  |
|          17 | 0                  |
+-------------+--------------------+
```

Если при вставке нового значения задействован механизм AUTO_INCREMENT, то воспользоваться вновь сгенерированным значением не удастся (листинг 6.12), так сначала полю id_catalog присваивается значение 0, и лишь после того, как запись добавлена — генерируется новое значение.

Еще одной формой оператора INSERT является синтаксис с ключевым словом SET:

```
INSERT [INTO] tbl SET col_name=expr, ...
```

Оператор заносит в таблицу tbl новую запись, столбец col_name в котором получает значение expr.

Листинг 6.13. Использование альтернативной формы оператора INSERT

```
mysql> INSERT INTO catalogs SET id_catalog = 18, name = 'Сетевые карты'
```

Запрос в листинге 6.13 аналогичен запросу из листинга 6.14.

Листинг 6.14. Полная форма оператора INSERT

```
mysql> INSERT INTO catalogs VALUES(18, 'Сетевые карты');
```

Форма оператора INSERT с ключевым словом SET позволяет более гибко добавлять записи. Если имя столбца после SET не встречается, полю присваивается значение по умолчанию.

6.2. Многострочный оператор *INSERT*

Многострочный оператор INSERT совпадает по форме с однострочным оператором. В нем используется ключевое слово VALUES, после которого добавляется не один, а несколько списков expression. Так, в листинге 6.15 добавляется сразу пять записей при помощи одного оператора INSERT.

Листинг 6.15. Многострочный оператор INSERT

```
mysql> INSERT INTO catalogs VALUES (0, 'Процессоры'),
    -> (0, 'Материнские платы'),
    -> (0, 'Видеоадаптеры'),
```

```
    -> (0, 'Жесткие диски'),
    -> (0, 'Оперативная память');
```

Точно так же, как и в случае однострочного варианта, допускается изменять порядок и состав списка добавляемых значений.

Листинг 6.16. Изменение состава столбцов в многострочном операторе INSERT

```
mysql> INSERT INTO catalogs (name) VALUES ('Процессоры'),
    -> ('Материнские платы'),
    -> ('Видеоадаптеры'),
    -> ('Жесткие диски'),
    -> ('Оперативная память');
```

6.3. Пакетная загрузка данных

Для пакетной загрузки данных из текстового файла в таблицу в СУБД MySQL предназначен специальный оператор LOAD DATA. Данный оператор работает быстрее, чем оператор INSERT, т. к. СУБД не требуется дополнительного времени на анализ синтаксиса оператора. Оператор имеет следующий синтаксис:

```
LOAD DATA [LOCAL] INFILE 'filename' INTO TABLE tbl
```

Если задано необязательное ключевое слово LOCAL, то файл читается с клиентского хоста. Если же LOCAL не указывается, то файл должен находиться на сервере.

Замечание

Оператор LOAD DATA является дополнительным к оператору SELECT ... INTO OUTFILE, который решает обратную задачу — формирование на основе таблицы текстового файла с данными. Синтаксис оператора SELECT ... INTO OUTFILE подробнее рассмотрен в *главе 7*.

Пусть в директории C:/mysql5/bin имеется текстовый файл catalogs.txt следующего содержания:

```
1       Процессоры
2       Материнские платы
3       Видеоадаптеры
4       Жесткие диски
5       Оперативная память
```

Поля базы данных должны быть разделены символом табуляции. Тогда выполнение запроса, представленного в листинге 6.17, приводит к заполнению базы таблицы catalogs.

Листинг 6.17. Использование оператора LOAD DATA

```
mysql> LOAD DATA INFILE 'C:\\mysql5\\bin\\catalogs.txt'
    -> INTO TABLE catalogs;
```

```
mysql> SELECT * FROM catalogs;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 | Оперативная память |
+------------+--------------------+
```

Замечание

При формировании пути к файлу в операционной системе Windows обратные слэши необходимо экранировать, этого можно избежать, если использовать прямые слэши: `'C:/mysql5/bin/catalogs.txt'`.

Перед конструкцией `INTO TABLE` можно разместить одно из двух ключевых слов, которые управляют обработкой ситуации, когда данные из текстового файла дублируют значения первичного или уникальных ключей:

- `IGNORE` — пропуск строк с дублирующими значениями;
- `REPLACE` — замена уже существующих записей новыми.

Создадим текстовый файл `newcatalog.txt` следующего содержания:

```
1       Сетевые адаптеры
2       Программное обеспечение
3       Мониторы
4       Периферия
5       CD-RW/DVD
```

Загрузка данных при помощи оператора `LOAD DATA` без использования ключевых слов `IGNORE` и `REPLACE` приведет к сообщению об ошибке (листинг 6.18). Однако использование оператора `REPLACE` приводит к полной замене содержимого таблицы `catalogs`.

Листинг 6.18. Использование ключевого слова `REPLACE`

```
mysql> LOAD DATA INFILE 'C:\\mysql5\\bin\\newcatalogs.txt'
    -> INTO TABLE catalogs;
ERROR 1062: Duplicate entry '1' for key 1
mysql> LOAD DATA INFILE 'C:\\mysql5\\bin\\newcatalogs.txt'
    -> REPLACE INTO TABLE catalogs;
mysql> SELECT * FROM catalogs;
+------------+-------------------------+
| id_catalog | name                    |
+------------+-------------------------+
|          1 | Сетевые адаптеры        |
|          2 | Программное обеспечение |
```

```
|           3 | Мониторы                  |
|           4 | Периферия                 |
|           5 | CD-RW/DVD                 |
+-------------+---------------------------+
```

Ключевое слово `IGNORE number LINES` позволяет задать количество `number` строк, которые необходимо пропустить от начала файла. Эта конструкция используется для того, чтобы пропустить заголовки столбцов, часто располагаемых на первой строке текстового файла (листинг 6.19).

Листинг 6.19. Использование конструкции `IGNORE number LINES`

```
mysql> LOAD DATA INFILE 'C:\\mysql5\\bin\\catalogs.txt'
    -> REPLACE INTO TABLE catalogs IGNORE 1 LINES;
```

Кроме базового синтаксиса, оператор `LOAD DATA` допускает дополнительные опции, позволяющие задать формат файла данных. Это осуществляется при помощи следующих ключевых слов:

- `LINES` — определяет формат строки данных, соответствующей записи;
- `FIELDS` — определяет формат поля данных.

Ключевое слово `LINES` позволяет задать символ начала и конца строки при помощи конструкций `STARTING BY` и `TERMINATED BY` соответственно. Так, по умолчанию строка в текстовом файле должна заканчиваться символом перевода строки `\n`. Однако при создании текстовых файлов в среде Windows строка заканчивается двумя символами `\r\n`, что может приводить к неправильной интерпретации файла данных и потере информации. При помощи конструкции `TERMINATED BY` можно переопределить символ конца строки. Представленный в листинге 6.20 SQL-запрос при использовании его в операционной системе Windows может выглядеть следующим образом.

Листинг 6.20. Использование конструкции `LINES TERMINATED BY`

```
mysql> LOAD DATA INFILE 'C:\\mysql5\\bin\\catalogs.txt'
    -> REPLACE INTO TABLE catalogs LINES TERMINATED BY '\r\n';
```

По умолчанию в качестве начальной строки задается пустая строка, таким образом, запросы в листинге 6.21 идентичны.

Листинг 6.21. Варианты использования оператора `LOAD DATA INFILE`

```
mysql> LOAD DATA INFILE 'C:\\mysql5\\bin\\catalogs.txt'
    -> REPLACE INTO TABLE catalogs;
mysql> LOAD DATA INFILE 'C:\\mysql5\\bin\\catalogs.txt'
    -> REPLACE INTO TABLE catalogs
    -> LINES STARTING BY '' TERMINATED BY '\n';
```

Ключевое слово `FIELDS` позволяет задать порядок обработки полей, совместно с ним применяются следующие конструкции:

- `TERMINATED BY` — данная конструкция определяет символ-разделитель между полями в строке, по умолчанию он равен символу табуляции `\t`, но может быть задан и любой другой символ;
- `ENCLOSED BY` — данная конструкция определяет символ кавычек, которыми ограничиваются поля, по умолчанию он равен пустой строке, т. е. кавычки не применяются;
- `ESCAPED BY` — используется для назначения символа экранирования в полях, по умолчанию принимает значение обратного слэша `"\"`.

Запросы, представленные в листинге 6.22, идентичны.

Листинг 6.22. Варианты использования оператора `LOAD DATA INFILE`

```
mysql> LOAD DATA INFILE 'C:\\mysql5\\bin\\catalogs.txt'
    -> REPLACE INTO TABLE catalogs;
mysql> LOAD DATA INFILE 'C:\\mysql5\\bin\\catalogs.txt'
    -> REPLACE INTO TABLE catalogs
    -> FIELDS TERMINATED BY '\t' ENCLOSED BY '' ESCAPED BY '\';
```

Пусть формат текстового файла данных таков, что поля разделены запятыми, значения обрамлены двойными кавычками, а строки заканчиваются символом `\r\n` (Windows стиль).

```
"1","Сетевые адаптеры"
"2","Программное обеспечение"
"3","Мониторы"
"4","Периферия"
"5","CD-RW/DVD"
```

Тогда запрос, загружающий данные из такого файла, может выглядеть так, как это представлено в листинге 6.23.

Листинг 6.23. Загрузка данных из csv-файла

```
mysql> LOAD DATA INFILE 'C:\\mysql5\\bin\\catalogs.txt'
    -> REPLACE INTO TABLE catalogs
    -> FIELDS TERMINATED BY ',' ENCLOSED BY '"'
    -> LINES TERMINATED BY '\r\n';
```

ГЛАВА 7

Выборка данных

В данной главе описывается оператор `SELECT`, позволяющий извлекать данные из таблиц базы данных. Упрощенный синтаксис оператора `SELECT` выглядит следующим образом:

```
SELECT select_expr FROM tbl
```

Каждый запрос начинается с ключевого слова `SELECT`, после которого следует список полей, разделенных запятыми `select_expr`, которые будут возвращены в результате запроса. Ключевое слово `FROM`, указывающее, из какой таблицы извлекаются данные, является необязательным и может быть опущено.

Результатом запроса `SELECT` всегда является таблица, которая ничем не отличается от таблиц, расположенных в базе данных, и называется *результирующей таблицей*. Более того, как будет показано далее, результаты запроса, выполненного при помощи оператора `SELECT`, могут быть использованы для создания новой таблицы. Если результаты двух запросов к разным таблицам имеют одинаковый формат, их можно объединить в одну таблицу.

Замечание

Таблица, полученная в результате запроса, может стать предметом дальнейших запросов. Техника создания таких запросов рассматривается в *главе 23*.

7.1. Изменение количества и порядка следования столбцов

Рассмотрим запросы на выборку данных на примере таблицы `catalogs` (см. листинг 4.24). Таблица имеет два поля: первичный ключ `id_catalog` и название каталога `name`. Выбрать все записи таблицы `tbl` можно при помощи запроса, представленного в листинге 7.1.

Листинг 7.1. Выборка данных при помощи оператора SELECT

```
mysql> SELECT id_catalog, name FROM catalogs;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          2 | Материнские платы  |
|          5 | Оперативная память |
|          1 | Процессоры         |
+------------+--------------------+
```

Если требуется вывести все столбцы таблицы, необязательно перечислять их имена после ключевого слова SELECT, достаточно заменить этот список символом "*" (все столбцы).

Листинг 7.2. Использование символа "*" (все столбцы)

```
mysql> SELECT * FROM catalogs;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          2 | Материнские платы  |
|          5 | Оперативная память |
|          1 | Процессоры         |
+------------+--------------------+
mysql> SELECT * FROM orders;
+----------+---------+---------------------+--------+------------+
| id_order | id_user | ordertime           | number | id_product |
+----------+---------+---------------------+--------+------------+
|        1 |       3 | 2005-01-04 10:39:38 |      1 |          8 |
|        2 |       6 | 2005-02-10 09:40:29 |      2 |         10 |
|        3 |       1 | 2005-02-18 13:41:05 |      4 |         20 |
|        4 |       3 | 2005-03-10 18:20:00 |      1 |         20 |
|        5 |       3 | 2005-03-17 19:15:36 |      1 |         20 |
+----------+---------+---------------------+--------+------------+
```

К списку столбцов в операторе SELECT прибегают в том случае, если необходимо изменить порядок следования столбцов в результирующей таблице или выбрать только часть столбцов.

Листинг 7.3. Изменение числа и порядка выводимых столбцов

```
mysql> SELECT name, id_catalog FROM catalogs;
+--------------------+------------+
| name               | id_catalog |
+--------------------+------------+
| Видеоадаптеры      |          3 |
```

```
| Жесткие диски      |          4 |
| Материнские платы  |          2 |
| Оперативная память |          5 |
| Процессоры         |          1 |
+--------------------+------------+
mysql> SELECT id_order, id_user, ordertime name FROM orders;
+----------+---------+--------------------+
| id_order | id_user | name               |
+----------+---------+--------------------+
|        1 |       3 | 2005-01-04 10:39:38 |
|        2 |       6 | 2005-02-10 09:40:29 |
|        3 |       1 | 2005-02-18 13:41:05 |
|        4 |       3 | 2005-03-10 18:20:00 |
|        5 |       3 | 2005-03-17 19:15:36 |
+----------+---------+--------------------+
```

В листинге 7.3 в запросе к таблице `catalogs` изменяется порядок следования столбцов, а в запросе к таблице `orders` из пяти столбцов таблицы извлекаются только три.

В списке `select_expr` наряду с именем столбца допустимо использование констант, например, числа 5 и строки `comments`. В этом случае каждая строка результирующей таблицы содержит поля 5 и `comments` (листинг 7.4).

Листинг 7.4. Использование констант в списке столбцов

```
mysql> SELECT name, id_catalog, 5, 'comments' FROM catalogs;
+--------------------+------------+---+----------+
| name               | id_catalog | 5 | comments |
+--------------------+------------+---+----------+
| Видеоадаптеры      |          3 | 5 | comments |
| Жесткие диски      |          4 | 5 | comments |
| Материнские платы  |          2 | 5 | comments |
| Оперативная память |          5 | 5 | comments |
| Процессоры         |          1 | 5 | comments |
+--------------------+------------+---+----------+
```

7.2. Условия

Ситуация, когда необходимо изменить количество выводимых строк, встречается гораздо чаще, чем когда требуется изменить число и порядок выводимых столбцов. Для ввода в SQL-запрос такого рода ограничений в операторе `SELECT` предназначено специальное ключевое слово `WHERE`, после которого следует логическое условие. Если запись удовлетворяет такому условию, она попадает в результат выборки, в противном случае такая запись отбрасывается.

Так, в листинге 7.5 приводится пример запроса, извлекающего из таблицы `catalog` записи, чей первичный ключ `id_catalog` больше (оператор ">") 2.

Листинг 7.5. Использование конструкции WHERE

```
mysql> SELECT * FROM catalogs WHERE id_catalog > 2;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 | Оперативная память |
+------------+--------------------+
```

Условие может быть составным и объединяться при помощи логических операторов. В листинге 7.6 используется составное условие: первичный ключ должен быть больше двух и меньше или равен 4. Для объединения этих двух условий используется оператор AND (И).

Замечание

Помимо оператора AND (И), для объединения логических выражений может использоваться оператор OR (ИЛИ). Подробнее логические операторы описаны в *главе 13*.

Листинг 7.6. Использование составного логического условия

```
mysql> SELECT * FROM catalogs
    -> WHERE id_catalog > 2 AND id_catalog <= 4;
+------------+---------------+
| id_catalog | name          |
+------------+---------------+
|          3 | Видеоадаптеры |
|          4 | Жесткие диски |
+------------+---------------+
```

Как видно из листинга 7.6, в результирующей таблице возвращаются записи в диапазоне от 3 до 4. Такого же эффекта можно добиться применением конструкции BETWEEN min AND max, возвращающей записи, значения столбца id_catalog для которых лежат в диапазоне от min до max (листинг 7.7).

Листинг 7.7. Использование конструкции BETWEEN

```
mysql> SELECT * FROM catalogs WHERE id_catalog BETWEEN 3 AND 4;
+------------+---------------+
| id_catalog | name          |
+------------+---------------+
|          3 | Видеоадаптеры |
|          4 | Жесткие диски |
+------------+---------------+
```

Существует конструкция, противоположенная конструкции BETWEEN — NOT BETWEEN, которая возвращает записи, не попадающие в интервал между min и max (листинг 7.8).

Листинг 7.8. Использование конструкции NOT BETWEEN

```
mysql> SELECT * FROM catalogs WHERE id_catalog NOT BETWEEN 3 AND 4;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          2 | Материнские платы  |
|          5 | Оперативная память |
|          1 | Процессоры         |
+------------+--------------------+
```

Иногда требуется извлечь записи, удовлетворяющие не диапазону, а списку, например, записи с id_catalog из списка (1,2,5), как показано в листинге 7.9. Для этого предназначена конструкция IN.

Листинг 7.9. Использование оператора IN

```
mysql> SELECT * FROM catalogs WHERE id_catalog IN (1,2,5);
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          2 | Материнские платы  |
|          5 | Оперативная память |
|          1 | Процессоры         |
+------------+--------------------+
```

Замечание

По аналогии с конструкцией BETWEEN, для IN существует обратная конструкция NOT IN, которая позволяет извлечь из таблицы записи, не удовлетворяющие списку. Подробнее операторы рассматриваются в *главе 13*.

В конструкции WHERE могут использоваться не только числовые столбцы. В листинге 7.10 из таблицы catalogs извлекается запись, соответствующая каталогу "Процессоры".

Листинг 7.10. Работа с текстовыми полями

```
mysql> SELECT * FROM catalogs WHERE name = 'процессоры';
+------------+------------+
| id_catalog | name       |
+------------+------------+
|          1 | Процессоры |
+------------+------------+
```

Как видно из листинга 7.10, сравнение строк в MySQL не зависит от регистра.

Замечание

Для работы с текстом СУБД MySQL предлагает широкие возможности, которые рассматриваются в *главах 15*, *17* и *18*.

В листинге 7.11 из таблицы `orders` извлекаются записи, соответствующие сделкам, осуществленным за февраль 2005 года.

Листинг 7.11. Работа с датой

```
mysql> SELECT * FROM orders
    -> WHERE ordertime >= '2005-02-01' AND ordertime < '2005-03-01';
+----------+---------+---------------------+--------+------------+
| id_order | id_user | ordertime           | number | id_product |
+----------+---------+---------------------+--------+------------+
|        2 |       6 | 2005-02-10 09:40:29 |      2 |         10 |
|        3 |       1 | 2005-02-18 13:41:05 |      4 |         20 |
+----------+---------+---------------------+--------+------------+
```

В качестве логических выражений могут выступать и константы. Любое число, отличное от нуля, считается истинным выражением, т. е. удовлетворяющим запросу, а 0 — ложью, т. е. не удовлетворяющим запросу.

Листинг 7.12. Использование констант в качестве логических выражений

```
mysql> SELECT * FROM catalogs WHERE 1;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          2 | Материнские платы  |
|          5 | Оперативная память |
|          1 | Процессоры         |
+------------+--------------------+
mysql> SELECT * FROM catalogs WHERE 0;
Empty set (0.00 sec)
```

Как видно из листинга 7.12, если первый запрос аналогичен `SELECT * FROM catalogs`, то второй запрос всегда возвращает пустую таблицу.

7.3. Сортировка

Как видно из листингов, результат выборки представляет собой записи, которые располагаются в порядке, в котором они хранятся в базе данных. Однако часто требуется отсортировать значения по одному из столбцов. Это осуществляется при помощи

конструкции ORDER BY, которая следует после выражения SELECT. После конструкции ORDER BY указывается столбец, по которому следует сортировать данные.

Листинг 7.13. Использование конструкции ORDER BY

```
mysql> SELECT * FROM catalogs ORDER BY id_catalog;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 | Оперативная память |
+------------+--------------------+
mysql> SELECT * FROM catalogs ORDER BY name;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          2 | Материнские платы  |
|          5 | Оперативная память |
|          1 | Процессоры         |
+------------+--------------------+
```

Как видно из листинга 7.13, первый запрос сортирует результат выборки по полю id_catalog, а второй по полю name.

Сортировку записей можно производить и по нескольким столбцам. Пусть требуется извлечь из таблицы products записи, соответствующие товарным позициям, количество которых на складе count от 4 до 8 с сортировкой по полю count. Для краткости выведем только столбцы count и mark (оценка товарной позиции).

Листинг 7.14. Вывод столбцов count и mark

```
mysql> SELECT count, mark FROM products
    -> WHERE count BETWEEN 4 AND 8 ORDER BY count;
+-------+------+
| count | mark |
+-------+------+
|     4 |  3.9 |
|     4 |  4.0 |
|     4 |  3.9 |
|     4 |  4.1 |
|     5 |  4.0 |
|     5 |  4.0 |
```

```
|     5 |  4.5 |
|     6 |  3.7 |
|     6 |  4.1 |
|     6 |  4.1 |
|     6 |  4.5 |
|     6 |  3.6 |
|     6 |  4.0 |
|     6 |  4.0 |
|     8 |  4.2 |
|     8 |  4.5 |
+-------+------+
```

Как видно из листинга 7.14, сортировка по полю `count` прошла успешно, однако оценки `mark` во втором столбце расположены в хаотическом порядке. Для сортировки результатов по двум столбцам их следует указать сразу после ключевого слова `ORDER BY` через запятую (листинг 7.15). Записи сортируются по полю `count` и, если встречается несколько записей с совпадающим полем `count`, сортируются по полю `mark`. Число столбцов, следующих за конструкцией `ORDER BY`, не ограничено.

Листинг 7.15. Сортировка по двум полям

```
mysql> SELECT count, mark FROM products
    -> WHERE count BETWEEN 4 AND 8 ORDER BY count, mark;
+-------+------+
| count | mark |
+-------+------+
|     4 |  3.9 |
|     4 |  3.9 |
|     4 |  4.0 |
|     4 |  4.1 |
|     5 |  4.0 |
|     5 |  4.0 |
|     5 |  4.5 |
|     6 |  3.6 |
|     6 |  3.7 |
|     6 |  4.0 |
|     6 |  4.0 |
|     6 |  4.1 |
|     6 |  4.1 |
|     6 |  4.5 |
|     8 |  4.2 |
|     8 |  4.5 |
+-------+------+
```

По умолчанию сортировка производится в прямом порядке, записи располагаются начиная с наименьшего значения и заканчивая наибольшим.

Листинг 7.16. Сортировка по умолчанию

```
mysql> SELECT ordertime FROM orders ORDER BY ordertime;
+---------------------+
| ordertime           |
+---------------------+
| 2005-01-04 10:39:38 |
| 2005-02-10 09:40:29 |
| 2005-02-18 13:41:05 |
| 2005-03-10 18:20:00 |
| 2005-03-17 19:15:36 |
+---------------------+
```

Изменить порядок сортировки на обратный можно при помощи ключевого слова `DESC` (листинг 7.17).

Листинг 7.17. Обратная сортировка

```
mysql> SELECT ordertime FROM orders ORDER BY ordertime DESC;
+---------------------+
| ordertime           |
+---------------------+
| 2005-03-17 19:15:36 |
| 2005-03-10 18:20:00 |
| 2005-02-18 13:41:05 |
| 2005-02-10 09:40:29 |
| 2005-01-04 10:39:38 |
+---------------------+
```

Для прямой сортировки также существует ключевое слово `ASC`, но поскольку по умолчанию записи сортируются в прямом порядке, данное ключевое слово часто опускают.

7.4. Ограничение выборки

Результат выборки может содержать сотни и тысячи записей. Их вывод и обработка занимают значительное время и серьезно загружают сервер базы данных. Поэтому информацию часто разбивают на страницы и предоставляют ее пользователю порциями. Извлечение только части запроса требует меньше времени и вычислений, кроме того, пользователю часто бывает достаточно просмотреть первые несколько записей. Постраничная навигация используется при помощи ключевого слова `LIMIT`, за которым следует число выводимых записей. В листинге 7.18 извлекаются первые пять записей, при этом одновременно осуществляется обратная сортировка по полю `count`.

Листинг 7.18. Использование ключевого слова `LIMIT`

```
mysql> SELECT id_product, count FROM products
    -> ORDER BY count DESC
    -> LIMIT 5;
+------------+-------+
| id_product | count |
+------------+-------+
|         28 |    20 |
|         25 |    20 |
|         26 |    15 |
|         29 |    12 |
|          9 |    12 |
+------------+-------+
```

Для того чтобы извлечь следующие пять записей, используется ключевое слово `LIMIT` с двумя цифрами, первая указывает позицию, начиная с которой необходимо вернуть результат, а вторая цифра — это число извлекаемых записей (листинг 7.19).

Листинг 7.19. Извлечение записей начиная с позиции 5

```
mysql> SELECT id_product, count FROM products
    -> ORDER BY count DESC
    -> LIMIT 5, 5;
+------------+-------+
| id_product | count |
+------------+-------+
|          1 |    10 |
|         27 |    10 |
|         24 |     8 |
|         30 |     8 |
|         20 |     6 |
+------------+-------+
```

Для извлечения следующих 5 записей необходимо использовать конструкцию `LIMIT 10, 5`.

7.5. Использование функций

При выполнении выборки из базы данных часто требуется выполнять специфические задачи, для решения которых удобно воспользоваться встроенными функциями MySQL. Каждая функция имеет уникальное имя и может иметь несколько аргументов (в том числе и ни одного), которые перечисляются через запятую в круглых скобках вслед за названием. Если аргументы у функции отсутствуют, круглые скобки все равно следует указывать, например, `NOW()`. Отличительной чертой MySQL является то, что при использовании функций пробелы между именем функции и круглыми скоб-

ками недопустимы, т. е. написание `NOW()` правильное, а `NOW ()` уже нет. Результат функции подставляется в место вызова функции.

В листинге 7.20 приведен пример использования функции `VERSION()`, которая возвращает версию сервера MySQL.

Листинг 7.20. Использование функции `VERSION()`

```
mysql> SELECT VERSION();
+---------------+
| VERSION()     |
+---------------+
| 5.0.3-beta-nt |
+---------------+
```

Как видно из листинга 7.20, для использования функции `VERSION()` не требуется применение ключевого слова `FROM`, т. к. параметры таблицы не нужны. Однако использование его не возбраняется (листинг 7.21).

Листинг 7.21. Использование функции `VERSION()`

```
mysql> SELECT VERSION() FROM catalogs;
+---------------+
| VERSION()     |
+---------------+
| 5.0.3-beta-nt |
| 5.0.3-beta-nt |
| 5.0.3-beta-nt |
| 5.0.3-beta-nt |
| 5.0.3-beta-nt |
+---------------+
```

В этом случае результат функции `VERSION()` выступает в качестве строковой константы (см. листинг 7.4) при выборке из таблицы `catalogs`.

Для того чтобы исключить повторение результата по числу строк, в таблице в MySQL применяется таблица `DUAL`, на самом деле не существующая, но которая может быть использована в предложении `FROM`.

Замечание

Таблица `DUAL` введена в MySQL, начиная с версии 4.1.

Листинг 7.22. Использование псевдотаблицы `DUAL`

```
mysql> SELECT VERSION(), 2+2 FROM DUAL;
+---------------+-----+
| VERSION()     | 2+2 |
+---------------+-----+
| 5.0.3-beta-nt |   4 |
+---------------+-----+
```

Чаще функция принимает параметры, например, имена столбцов. Функция `COUNT()` возвращает число записей в таблице и принимает в качестве аргумента имя столбца. Функция возвращает число строк в таблице, значения столбца для которых отличны от `NULL`.

Листинг 7.23. Использование функции `COUNT()`

```
mysql> SELECT COUNT(id_order) FROM orders;
+-----------------+
| COUNT(id_order) |
+-----------------+
|               5 |
+-----------------+
mysql> SELECT COUNT(id_product) FROM products;
+-------------------+
| COUNT(id_product) |
+-------------------+
|                30 |
+-------------------+
```

В качестве параметра функции наряду с именами столбцов может выступать символ звездочка "*". Одной из особенностей использования функций является то, что название столбца в результирующей таблицы совпадает с названием функции и ее параметрами. Часто это не удобно, особенно в прикладных программах, где после выполнения запроса обращение к результату происходит по имени столбца. В `SELECT`-запросе столбцу можно назначить новое имя, для этого предназначен оператор `AS`. В листинге 7.24 результату функции `COUNT()` присваивается новый псевдоним `total`.

Листинг 7.24. Использование оператора `AS`

```
mysql> SELECT COUNT(id_order) AS total FROM orders;
+-------+
| total |
+-------+
|     5 |
+-------+
```

Новое имя можно использовать в других частях SQL-запроса, в выражениях `WHERE` и `ORDER BY`. Данный прием часто используется при выполнении многотабличных запросов, рассмотренных в *главе 22*.

В качестве дополнительного примера можно привести функции `MIN()` и `MAX()`, возвращающие минимальное и максимальное значения столбца, имя которого передано в качестве параметра.

Листинг 7.25. Использование функций MIN() и MAX()

```
mysql> SELECT MIN(id_catalog) AS min,
    -> MAX(id_catalog) AS max FROM catalogs;
+------+------+
| min  | max  |
+------+------+
|    1 |    5 |
+------+------+
```

Однако использование функций MIN() и MAX() в выражении WHERE приведет к ошибке. В листинге 7.26 показана попытка извлечения записи из таблицы catalogs с максимальным значением поля id_catalog.

Листинг 7.26. Использование функций MIN() и MAX() в WHERE недопустимо

```
mysql> SELECT * FROM catalogs WHERE id_catalog = MAX(id_catalog);
ERROR 1111: Invalid use of group function
```

Решение поставленной выше задачи следует искать с привлечением выражения ORDER BY. В листинге 7.27 первый запрос извлекает запись с наименьшим значением поля id_catalog, а второй — с наибольшим.

Листинг 7.27. Использование функций MIN() и MAX() совместно с ORDER BY

```
mysql> SELECT * FROM catalogs ORDER BY id_catalog LIMIT 1;
+------------+------------+
| id_catalog | name       |
+------------+------------+
|          1 | Процессоры |
+------------+------------+
mysql> SELECT * FROM catalogs ORDER BY id_catalog DESC LIMIT 1;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          5 | Оперативная память |
+------------+--------------------+
```

СУБД MySQL имеет большое число встроенных функций, подробному их рассмотрению посвящена *часть II* книги.

7.6. Группировка записей

Пусть необходимо определить первичные ключи, соответствующие каталогам id_catalog, в которых есть хоть одна товарная позиция. Для этого достаточно осуществить запрос к таблице products, который извлечет поле id_catalog (листинг 7.28).

Листинг 7.28. Извлечение первичных ключей, соответствующих каталогам

```
mysql> SELECT id_catalog FROM products ORDER BY id_catalog;
+------------+
| id_catalog |
+------------+
|          1 |
|          1 |
|          1 |
|          1 |
|          1 |
|          1 |
|          1 |
|          1 |
|          1 |
|          2 |
|          2 |
|          2 |
|          2 |
|          2 |
|          2 |
|          3 |
|          3 |
|          3 |
|          3 |
|          4 |
|          4 |
|          4 |
|          4 |
|          4 |
|          5 |
|          5 |
|          5 |
|          5 |
|          5 |
|          5 |
+------------+
```

Как видно из листинга 7.28, результат не совсем удобен для восприятия. Было бы лучше, если бы запрос вернул уникальные значения столбца `id_catalog`. Для этого перед именем столбца можно использовать ключевое слово `DISTINCT`, которое предписывает MySQL извлекать только уникальные значения.

Замечание

Ключевое слово `DISTINCT` имеет синоним — `DISTINCTROW`.

Листинг 7.29. Выборка уникальных значений

```
mysql> SELECT DISTINCT id_catalog FROM products ORDER BY id_catalog;
+------------+
| id_catalog |
+------------+
|          1 |
|          2 |
|          3 |
|          4 |
|          5 |
+------------+
```

Как показано в листинге 7.29, результат запроса не содержит ни одного повторяющегося значения. Использование ключевого слова `DISTINCT` допускается совместно с функцией `COUNT()`. В листинге 7.30 первый запрос возвращает общее число записей в таблице `products`, а второй — число уникальных значений `id_catalog`.

Листинг 7.30. Совместное использование ключевого слова `DISTINCT` и функции `COUNT()`

```
mysql> SELECT COUNT(id_catalog) FROM products;
+-------------------+
| COUNT(id_catalog) |
+-------------------+
|                30 |
+-------------------+
mysql> SELECT COUNT(DISTINCT id_catalog) FROM products;
+----------------------------+
| COUNT(DISTINCT id_catalog) |
+----------------------------+
|                          5 |
+----------------------------+
```

Для ключевого слова `DISTINCT` имеется противоположное слово `ALL`, которое предписывает извлечение всех значений столбца, в том числе и повторяющихся. Поскольку такое поведение установлено по умолчанию, ключевое слово `ALL` часто опускают.

Для извлечения уникальных записей чаще прибегают к конструкции `GROUP BY`, за которой указывается имя столбца, по которому группируется результат.

Замечание

Конструкция `GROUP BY` располагается в `SELECT`-запросе перед `ORDER BY` и `LIMIT`.

Листинг 7.31. Использование конструкции `GROUP BY`

```
mysql> SELECT id_catalog FROM products
    -> GROUP BY id_catalog ORDER BY id_catalog;
```

```
+------------+
| id_catalog |
+------------+
|          1 |
|          2 |
|          3 |
|          4 |
|          5 |
+------------+
```

Однако, в отличие от ключевого слова `DISTINCT`, использование функции `COUNT()` совместно с `GROUP BY` приводит не к подсчету уникальных значений `id_catalog`, а к выводу числа записей, соответствующих каждому из уникальных значений `id_catalog`.

Листинг 7.32. Совместное использование `GROUP BY` и функции `COUNT()`

```
mysql> SELECT id_catalog, COUNT(id_catalog) FROM products
    -> GROUP BY id_catalog ORDER BY id_catalog;
+------------+-------------------+
| id_catalog | COUNT(id_catalog) |
+------------+-------------------+
|          1 |                 9 |
|          2 |                 6 |
|          3 |                 4 |
|          4 |                 5 |
|          5 |                 6 |
+------------+-------------------+
```

Как видно из листинга 7.32, каталог с первичным ключом `id_catalog = 1` содержит 9 товарных позиций, с `id_catalog = 2` — 6 и т. д.

Замечание

Помимо функции `COUNT()`, в выражениях с использованием конструкции `GROUP BY` может применяться целый класс функций, описанию которых посвящена *глава 9*.

При использовании конструкции `GROUP BY` точно так же возможно использование условия `WHERE`.

Листинг 7.33. Использование ключевого слова `WHERE` совместно с `GROUP BY`

```
mysql> SELECT id_catalog, COUNT(id_catalog) FROM products
    -> WHERE id_catalog > 2
    -> GROUP BY id_catalog
    -> ORDER BY id_catalog;
+------------+-------------------+
| id_catalog | COUNT(id_catalog) |
+------------+-------------------+
|          3 |                 4 |
```

```
|           4 |                   5 |
|           5 |                   6 |
+-------------+--------------------+
```

Чаще при составлении условий требуется ограничить выборку по результату функции, например, выбрать каталоги, где число товарных позиций больше пяти. Использование для этих целей конструкции WHERE приводит к ошибке.

Листинг 7.34. Ошибочное использование конструкции WHERE

```
mysql> SELECT id_catalog, COUNT(id_catalog) AS total FROM products
    -> WHERE total > 5
    -> GROUP BY id_catalog
    -> ORDER BY id_catalog;
ERROR 1054: Unknown column 'total' in 'where clause'
```

Для решения этой проблемы вместо ключевого слова WHERE используется ключевое слово HAVING, которое располагается вслед за конструкцией GROUP BY.

Листинг 7.35. Использование ключевого слова GROUP BY

```
mysql> SELECT id_catalog, COUNT(id_catalog) AS total FROM products
    -> GROUP BY id_catalog
    -> HAVING total > 5
    -> ORDER BY id_catalog;
+-------------+-------+
| id_catalog | total |
+-------------+-------+
|           1 |     9 |
|           2 |     6 |
|           5 |     6 |
+-------------+-------+
```

В условии HAVING можно использовать все столбцы результирующей таблицы, не только вычисляемые, например, в листинге 7.36 приводится пример запроса, извлекающего уникальные значения столбца id_catalog, большие двух.

Листинг 7.36. Использование в конструкции HAVING невычисляемых столбцов

```
mysql> SELECT id_catalog, COUNT(id_catalog) FROM products
    -> GROUP BY id_catalog
    -> HAVING id_catalog > 2
    -> ORDER BY id_catalog;
+-------------+--------------------+
| id_catalog | COUNT(id_catalog) |
+-------------+--------------------+
|           3 |                   4 |
```

```
|           4 |                   5 |
|           5 |                   6 |
+-------------+--------------------+
```

Отличие запросов, представленных в листингах 7.33 и 7.36, заключается в том, что в случае использования ключевого слова WHERE сначала производится выборка из таблицы с применением условия и лишь затем группировка результата, а в случае использования ключевого слова HAVING сначала происходит группировка таблицы и лишь затем выборка с применением условия.

Допускается использование условия HAVING без группировки GROUP BY.

Листинг 7.37. Использование ключевого слова HAVING без GROUP BY

```
mysql> SELECT id_catalog FROM products
    -> HAVING id_catalog > 2
    -> ORDER BY id_catalog;
+-------------+
| id_catalog |
+-------------+
|           3 |
|           3 |
|           3 |
|           3 |
|           4 |
|           4 |
|           4 |
|           4 |
|           4 |
|           5 |
|           5 |
|           5 |
|           5 |
|           5 |
|           5 |
+-------------+
```

В этом случае каждая строка таблицы рассматривается как отдельная группа.

7.7. Объединение таблиц

Как было сказано выше, оператор SELECT возвращает результат в виде таблицы. Если формат результирующих таблиц (число, порядок следования и тип столбцов) совпадает, то возможно объединение результатов выполнения двух операторов SELECT в одну результирующую таблицу. Это достигается использованием оператора UNION.

Пусть имеются два запроса, представленные в листинге 7.38.

Листинг 7.38. Одиночные SELECT-запросы

```
mysql> SELECT id_catalog FROM catalogs;
+------------+
| id_catalog |
+------------+
|          1 |
|          2 |
|          3 |
|          4 |
|          5 |
+------------+
mysql> SELECT id_order + 5 FROM orders;
+--------------+
| id_order + 5 |
+--------------+
|            6 |
|            7 |
|            8 |
|            9 |
|           10 |
+--------------+
```

Объединить результаты из этих двух таблиц можно, соединив два запроса SELECT при помощи ключевого слова UNION, как это продемонстрировано в листинге 7.39.

Листинг 7.39. Использование ключевого слова UNION

```
mysql> SELECT id_catalog FROM catalogs
    -> UNION
    -> SELECT id_order + 5 FROM orders;
+------------+
| id_catalog |
+------------+
|          1 |
|          2 |
|          3 |
|          4 |
|          5 |
|          6 |
|          7 |
|          8 |
|          9 |
|         10 |
+------------+
```

Во втором запросе SELECT, производящем выборку из таблицы orders, к значению первичного ключа id_orders добавляется значение 5, таким образом, все значения в

результирующей таблице становятся уникальными. Однако если результирующая таблица содержит повторяющиеся строки, СУБД MySQL автоматически отбрасывает дубликаты (листинг 7.40).

Листинг 7.40. Исключение дублирующих записей при использовании конструкции UNION

```
mysql> SELECT id_catalog FROM catalogs
    -> UNION
    -> SELECT id_order FROM orders;
+------------+
| id_catalog |
+------------+
|          1 |
|          2 |
|          3 |
|          4 |
|          5 |
+------------+
```

Как видно из листинга 7.40, вместо десяти записей выводится только пять. Изменить поведение по умолчанию можно при помощи ключевого слова ALL, которое добавляется после оператора UNION. Использование UNION ALL требует, чтобы возвращались все строки из обеих результирующих таблиц.

Листинг 7.41. Использование ключевого слова UNION ALL

```
mysql> SELECT id_catalog FROM catalogs
    -> UNION ALL
    -> SELECT id_order FROM orders;
+------------+
| id_catalog |
+------------+
|          1 |
|          2 |
|          3 |
|          4 |
|          5 |
|          1 |
|          2 |
|          3 |
|          4 |
|          5 |
+------------+
```

Для формы UNION ALL существует противоположенный оператор UNION DISTINCT, но т. к. оператор UNION по умолчанию отбрасывает не уникальные столбцы, ключевое слово DISTINCT часто опускают.

7.8. Сохранение результатов во внешний файл

Результирующую таблицу, получаемую в результате выполнения оператора SELECT, можно сохранить в текстовый файл. Это позволяет более внимательно изучить результаты, подвергнуть их дальнейшей обработки при помощи внешней программы или вставить их в таблицу при помощи оператора LOAD DATA INFILE, который описывается в *разделе 6.3*.

Для сохранения результатов в текстовом файле предназначен оператор SELECT в форме SELECT ... INTO OUTFILE 'file_name'. Эта разновидность команды осуществляет запись выбранных строк в файл, указанный в file_name.

Выполнение SQL-запроса, показанного в листинге 7.42, приведет к созданию файла text.sql в директории C:/mysql5/data/shop.

Замечание

Файл file_name не должен существовать, иначе будет возвращена ошибка.

Листинг 7.42. Сохранение результирующей таблицы во внешнем файле text.sql

```
mysql> SELECT * INTO OUTFILE 'text.sql' FROM catalogs;
Query OK, 5 rows affected (0.03 sec)
```

Важной деталью при составлении запроса, сохраняющего результат SELECT-запроса во внешнем текстовом файле, является то, что предложение INTO OUTFILE располагается до ключевого слова FROM. Открытие данного файла в текстовом редакторе (Notepad) Windows приведет к тому, что перевод строк не производится, а вместо символа перевода строки присутствует квадратик, которым обозначаются нечитаемые символы. Здесь, так же как и в случае оператора LOAD DATA INFILE, по умолчанию принят UNIX-формат, т. е. перевод строк осуществляется только символом перевода строки \n, без добавления символа перевода каретки \r\n, который требует Windows.

Точно так же, как и для оператора LOAD DATA INFILE, формат текстового файла задается при помощи ключевых слов:

- LINES — определяет формат строки данных, соответствующей записи;
- FIELDS — определяет формат поля данных.

Ключевое слово LINES позволяет задать символ начала и конца строки при помощи конструкций STARTING BY и TERMINATED BY соответственно. Для того чтобы строка начиналась со слова "TABLE ", а заканчивалась переводом строки в стиле Windows — \r\n, необходимо выполнить запрос, представленный в листинге 7.43.

Листинг 7.43. Изменение формата внешнего файла при помощи LINES

```
mysql> SELECT * INTO OUTFILE 'text.sql'
    -> LINES STARTING BY 'TABLE ' TERMINATED BY '\n'
```

```
    -> FROM catalogs
    -> ORDER BY id_catalog;
```

В результате будет создан файл следующего содержания:

```
TABLE 1 Процессоры
TABLE 2 Материнские платы
TABLE 3 Видеоадаптеры
TABLE 4 Жесткие диски
TABLE 5 Оперативная память
```

Конструкция `ORDER BY id_catalog` обеспечивает сортировку по первому полю таблицы `catalogs`.

Ключевое слово `FIELDS` позволяет задать порядок обработки полей, совместно с ним применяются следующие конструкции:

- `TERMINATED BY` — данная конструкция определяет символ-разделитель между полями в строке, по умолчанию это символ табуляции `\t`, но может быть задан любой другой символ;
- `ENCLOSED BY` — данная конструкция определяет символ кавычек, которыми ограничиваются поля, по умолчанию это пустая строка, т. е. кавычки не применяются;
- `ESCAPED BY` — используется для назначения символа экранирования в полях, по умолчанию принимает значение обратного слэша "\".

Замечание

Конструкция `FIELDS` размещается перед конструкцией `LINES`.

Листинг 7.44. Изменение формата внешнего файла при помощи `FIELDS`

```
mysql> SELECT * INTO OUTFILE 'text.sql'
    -> FIELDS TERMINATED BY ',' ENCLOSED BY '"'
    -> LINES STARTING BY 'TABLE ' TERMINATED BY '\r\n'
    -> FROM catalogs ORDER BY id_catalog;
```

Выполнение запроса из листинга 7.44 приводит к созданию файла `text.sql` следующего содержания:

```
TABLE "1","Процессоры"
TABLE "2","Материнские платы"
TABLE "3","Видеоадаптеры"
TABLE "4","Жесткие диски"
TABLE "5","Оперативная память"
```

Часто удобно формировать текстовый файл в виде инструкций SQL, как это делает утилита `mysqldump`. В листинге 7.45 приведен запрос, создающий текстовый файл, каждая строка в котором оформлена в виде запроса `INSERT`.

Листинг 7.45. Создание дампа таблицы

```
mysql> SELECT * INTO OUTFILE 'text.sql'
    -> FIELDS TERMINATED BY ',' ENCLOSED BY '"'
    -> LINES STARTING BY 'INSERT INTO tbl VALUES(' TERMINATED BY ');\r\n'
    -> FROM catalogs ORDER BY id_catalog;
```

В результате запроса, представленного в листинге 7.45, будет создан файл `text.sql` следующего содержания

```
INSERT INTO tbl VALUES("1","Процессоры");
INSERT INTO tbl VALUES("2","Материнские платы");
INSERT INTO tbl VALUES("3","Видеоадаптеры");
INSERT INTO tbl VALUES("4","Жесткие диски");
INSERT INTO tbl VALUES("5","Оперативная память");
```

ГЛАВА 8

Удаление данных

Время от времени возникает задача удаления записей из базы данных, например, комплектующие могут устаревать и быть снятыми с производства. Для того чтобы отразить этот факт в учебной базе данных, необходимо удалить соответствующую запись в таблице `products`.

Для удаления записей из таблиц предусмотрено два оператора:

- `DELETE`
- `TRUNCATE TABLE`

8.1. Оператор *DELETE*

Оператор `DELETE` имеет следующий синтаксис:

```
DELETE FROM tbl
WHERE where_definition
ORDER BY ...
LIMIT rows
```

Оператор удаляет из таблицы `tbl` записи, удовлетворяющие условию `where_definition`. В листинге 8.1 из таблицы `catalogs` удаляются записи, имеющие значение первичного ключа `id_catalog` больше двух.

Листинг 8.1. Удаление записей из таблицы `catalogs`

```
mysql> DELETE FROM catalogs WHERE id_catalog > 2;
mysql> SELECT * FROM catalogs;
+------------+-------------------+
| id_catalog | name              |
+------------+-------------------+
|          2 | Материнские платы |
|          1 | Процессоры        |
+------------+-------------------+
```

Если в операторе DELETE отсутствует условие WHERE, из таблицы удаляются все записи.

Листинг 8.2. Удаление всех записей таблицы catalogs

```
mysql> DELETE FROM catalogs;
mysql> SELECT * FROM catalogs;
Empty set (0.00 sec)
```

Применение ограничения LIMIT позволяет задать максимальное число записей, которые могут быть уничтожены. В листинге 8.3 удаляются все записи таблицы orders, но не более 3 записей.

Листинг 8.3. Удаление записей из таблицы orders

```
mysql> DELETE FROM orders LIMIT 3;
mysql> SELECT * FROM orders;
+----------+---------+---------------------+--------+------------+
| id_order | id_user | ordertime           | number | id_product |
+----------+---------+---------------------+--------+------------+
|        4 |       3 | 2005-03-10 18:20:00 |      1 |         20 |
|        5 |       3 | 2005-03-17 19:15:36 |      1 |         20 |
+----------+---------+---------------------+--------+------------+
```

Так как таблица содержит 5 записей, то в результате в таблице остаются две записи. Инструкция ORDER BY обычно применяется совместно с ключевым словом LIMIT. Например, если необходимо удалить 20 первых записей таблицы, то производится сортировка по полю DATETIME — это гарантирует, что удаляться будут в первую очередь самые старые таблицы.

8.2. Оператор *TRUNCATE TABLE*

Оператор TRUNCATE TABLE, в отличие от оператора DELETE, полностью очищает таблицу и не допускает условного удаления. То есть оператор TRUNCATE TABLE аналогичен оператору DELETE без условия WHERE и ограничения LIMIT. В отличие от оператора DELETE, удаление происходит гораздо быстрее, т. к. осуществляется не перебор каждой записи, а полное очищение таблицы.

Замечание

Оптимизатор запросов СУБД MySQL автоматически использует оператор TRUNCATE TABLE, если оператор DELETE не содержит условия WHERE или конструкции LIMIT.

Листинг 8.4. Удаление таблицы products

```
mysql> TRUNCATE TABLE products;
```

ГЛАВА 9

Обновление данных

Операция обновления позволяет менять значения полей в уже существующих записях, это может понадобиться при изменении цен комплектующих, их количества на складе и т. д. Для обновления данных предназначены операторы `UPDATE` и `REPLACE`. Первый позволяет обновлять отдельные поля в уже существующих записях, тогда как оператор `REPLACE` больше похож на `INSERT`, за исключением того, что если старая запись в данной таблице имеет то же значение индекса `UNIQUE` или `PRIMARY KEY`, что и новая, то старая запись перед занесением новой будет удалена.

9.1. Оператор *UPDATE*

Оператор `UPDATE` имеет следующий синтаксис:

```
UPDATE [IGNORE] tbl
SET col1=expr1 [, col2=expr2 ...]
[WHERE where_definition]
[ORDER BY ...]
[LIMIT rows]
```

В инструкции, сразу после ключевого слова `UPDATE`, указывается таблица `tbl`, которая подвергается изменению. В предложении `SET` указывается, какие столбцы подвергаются обновлению и устанавливаются их новые значения. Необязательное условие `WHERE` позволяет задать критерий отбора строк — обновлению будут подвергаться только те строки, которые удовлетворяют условию `where_definition`.

В листинге 9.1 запись в таблице `catalogs` "Процессоры" изменяется на "Процессоры (Intel)".

Листинг 9.1. Обновление таблицы `catalogs`

```
mysql> SELECT * FROM catalogs ORDER BY name;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          3 | Видеоадаптеры      |
```

```
|           4 | Жесткие диски      |
|           2 | Материнские платы  |
|           5 | Оперативная память |
|           1 | Процессоры         |
+------------+--------------------+
mysql> UPDATE catalogs SET name = 'Процессоры (Intel)'
    -> WHERE name = 'Процессоры';
mysql> SELECT * FROM catalogs ORDER BY name;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|           3 | Видеоадаптеры      |
|           4 | Жесткие диски      |
|           2 | Материнские платы  |
|           5 | Оперативная память |
|           1 | Процессоры (Intel) |
+------------+--------------------+
```

Изменения можно производить не только над выборочными записями, но и над всей таблицей в целом. Пусть требуется изменить цену комплектующих в учебной базе данных в связи с изменением курса доллара. Для этого следует разделить цену в рублях на старый курс (28.5), чтобы получить цену в долларах, и умножить результат на новый курс (28.0).

Листинг 9.2. Изменение курса доллара

```
mysql> UPDATE products SET price = (price/28.5)*28.0;
mysql> SELECT name, price FROM products ORDER BY price;
+------------------------------+---------+
| name                         | price   |
+------------------------------+---------+
| DDR-400 256MB PQI            | 883.23  |
| DDR-400 256MB Kingston       | 1065.96 |
| DDR-400 256MB Hynix Original | 1158.32 |
| Celeron 1.8                  | 1567.02 |
| ASUSTEK V9520X               | 1573.89 |
| DDR-400 512MB PQI            | 1660.35 |
| DDR-400 512MB Hynix          | 1686.88 |
| Celeron D 315 2.26GHz        | 1847.02 |
| Gigabyte GA-8I848P-RS        | 1862.74 |
| DDR-400 512MB Kingston       | 1898.11 |
| Celeron D 320 2.4GHz         | 1927.58 |
| Celeron 2.0GHz               | 1934.46 |
| Samsung SP0812C              | 2056.28 |
| Celeron 2.4GHz               | 2072.00 |
| Gigabyte GA-8IPE1000G        | 2248.84 |
| Epox EP-4PDA3I               | 2248.84 |
```

```
| Gigabyte GA-8IG1000           | 2377.54 |
| Maxtor 6Y120P0                | 2412.91 |
| Seagate ST3120026A            | 2424.70 |
| Asustek P4P800-VM\L i865G     | 2473.82 |
| SAPPHIRE 256MB RADEON 9550    | 2682.11 |
| Celeron D 325 2.53GHz         | 2698.81 |
| Seagate Barracuda ST3160023A  | 3083.93 |
| Maxtor 6B200P0                | 3526.04 |
| ASUSTEK A9600XT/TD            | 5065.54 |
| Asustek P4C800-E Delux        | 5300.35 |
| Intel Pentium 4 3.0GHz        | 5573.47 |
| GIGABYTE AGP GV-N59X128D      | 5782.74 |
| Intel Pentium 4 3.0GHz        | 6039.16 |
| Intel Pentium 4 3.2GHz        | 7131.65 |
+-------------------------------+---------+
```

Точно так же, как и в случае оператора `DELETE`, инструкции `LIMIT` и `ORDER BY` позволяют ограничить число записей, подвергающихся изменению. Пусть стоит задача для десяти самых дешевых товарных позиций вернуть цену с учетом старого курса доллара, в этом случае запрос к таблице `products` может выглядеть так, как это продемонстрировано в листинге 9.3.

Листинг 9.3. Изменение цены для 10 самых дешевых товарных позиций

```
mysql> UPDATE products SET price = (price/28.0)*28.5
    -> ORDER BY price LIMIT 10;
mysql> SELECT name, price, count FROM products ORDER BY price;
+-------------------------------+---------+-------+
| name                          | price   | count |
+-------------------------------+---------+-------+
| DDR-400 256MB PQI             | 899.00  |    10 |
| DDR-400 256MB Kingston        | 1085.00 |    20 |
| DDR-400 256MB Hynix Original  | 1179.00 |    15 |
| Celeron 1.8                   | 1595.00 |    10 |
| ASUSTEK V9520X                | 1602.00 |     6 |
| DDR-400 512MB PQI             | 1690.00 |    12 |
| DDR-400 512MB Hynix           | 1717.00 |     8 |
| Celeron D 315 2.26GHz         | 1880.00 |     6 |
| Gigabyte GA-8I848P-RS         | 1896.00 |     4 |
| Celeron D 320 2.4GHz          | 1927.58 |     1 |
| DDR-400 512MB Kingston        | 1932.00 |    20 |
| Celeron 2.0GHz                | 1934.46 |     2 |
| Samsung SP0812C               | 2056.28 |     5 |
| Celeron 2.4GHz                | 2072.00 |     4 |
| Gigabyte GA-8IPE1000G         | 2248.84 |     6 |
| Epox EP-4PDA3I                | 2248.84 |     5 |
| Gigabyte GA-8IG1000           | 2377.54 |     2 |
```

```
| Maxtor 6Y120P0                | 2412.91 |     6 |
| Seagate ST3120026A            | 2424.70 |     8 |
| Asustek P4P800-VM\L i865G     | 2473.82 |     6 |
| SAPPHIRE 256MB RADEON 9550    | 2682.11 |     3 |
| Celeron D 325 2.53GHz         | 2698.81 |     6 |
| Seagate Barracuda ST3160023A  | 3083.93 |     3 |
| Maxtor 6B200P0                | 3526.04 |     4 |
| ASUSTEK A9600XT/TD            | 5065.54 |     2 |
| Asustek P4C800-E Delux        | 5300.35 |     4 |
| Intel Pentium 4 3.0GHz        | 5573.47 |    12 |
| GIGABYTE AGP GV-N59X128D      | 5782.74 |     6 |
| Intel Pentium 4 3.0GHz        | 6039.16 |     1 |
| Intel Pentium 4 3.2GHz        | 7131.65 |     5 |
+-------------------------------+---------+-------+
```

За один SQL-запрос `UPDATE` обновлению может подвергаться несколько столбцов таблицы. Пусть теперь необходимо для десяти самых дешевых товарных позиций уменьшить число товарных позиций на складе на одну единицу, а цену — на 5 %. Решить эту задачу позволяет запрос, представленный в листинге 9.4.

Листинг 9.4. Уменьшение числа товарных позиций на складе, 5%-ная скидка на товары

```
mysql> SELECT name, price, count FROM products ORDER BY price;
+-------------------------------+---------+-------+
| name                          | price   | count |
+-------------------------------+---------+-------+
| DDR-400 256MB PQI             | 854.05  |     9 |
| DDR-400 256MB Kingston        | 1030.75 |    19 |
| DDR-400 256MB Hynix Original  | 1120.05 |    14 |
| Celeron 1.8                   | 1515.25 |     9 |
| ASUSTEK V9520X                | 1521.90 |     5 |
| DDR-400 512MB PQI             | 1605.50 |    11 |
| DDR-400 512MB Hynix           | 1631.15 |     7 |
| Celeron D 315 2.26GHz         | 1786.00 |     5 |
| Gigabyte GA-8I848P-RS         | 1801.20 |     3 |
| Celeron D 320 2.4GHz          | 1831.20 |     0 |
| DDR-400 512MB Kingston        | 1932.00 |    20 |
| Celeron 2.0GHz                | 1934.46 |     2 |
| Samsung SP0812C               | 2056.28 |     5 |
| Celeron 2.4GHz                | 2072.00 |     4 |
| Gigabyte GA-8IPE1000G         | 2248.84 |     6 |
| Epox EP-4PDA3I                | 2248.84 |     5 |
| Gigabyte GA-8IG1000           | 2377.54 |     2 |
| Maxtor 6Y120P0                | 2412.91 |     6 |
| Seagate ST3120026A            | 2424.70 |     8 |
| Asustek P4P800-VM\L i865G     | 2473.82 |     6 |
| SAPPHIRE 256MB RADEON 9550    | 2682.11 |     3 |
```

```
| Celeron D 325 2.53GHz         | 2698.81 |     6 |
| Seagate Barracuda ST3160023A  | 3083.93 |     3 |
| Maxtor 6B200P0                | 3526.04 |     4 |
| ASUSTEK A9600XT/TD            | 5065.54 |     2 |
| Asustek P4C800-E Delux        | 5300.35 |     4 |
| Intel Pentium 4 3.0GHz        | 5573.47 |    12 |
| GIGABYTE AGP GV-N59X128D      | 5782.74 |     6 |
| Intel Pentium 4 3.0GHz        | 6039.16 |     1 |
| Intel Pentium 4 3.2GHz        | 7131.65 |     5 |
+-------------------------------+---------+-------+
```

Если указывается необязательное ключевое слово `IGNORE`, то команда обновления не будет прервана, даже если при обновлении возникнет ошибка дублирования ключей. Строки, из-за которых возникают конфликтные ситуации, обновлены не будут.

9.2. Оператор *REPLACE*

Оператор `REPLACE` работает точно так же, как `INSERT`, за исключением того, что если старая запись в данной таблице имеет то же значение индекса `UNIQUE` или `PRIMARY KEY`, что и новая, то старая запись перед занесением новой будет удалена. Следует учитывать, что если не используются индексы `UNIQUE` или `PRIMARY KEY`, то применение команды `REPLACE` не имеет смысла, т. к. она работает просто как `INSERT`.

Синтаксис оператора `RENAME` аналогичен синтаксису оператора `INSERT`:

```
REPLACE [INTO] tbl [(col_name,...)] VALUES (expression,...),(...),...
```

В таблицу `tbl` вставляются значения, определяемые в списке после ключевого слова `VALUES`. Задать порядок столбцов можно при помощи необязательного списка `col_name`, следующего за именем таблицы `tbl`. Точно так же, как и в случае оператора `INSERT`, оператор `REPLACE` допускает многострочный формат. Пример использования оператора приведен в листинге 9.5, где в таблицу `catalogs` добавляется пять новых записей.

Листинг 9.5. Использование оператора `REPLACE`

```
mysql> SELECT * FROM catalogs ORDER BY id_catalog;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 | Оперативная память |
+------------+--------------------+
mysql> REPLACE INTO catalogs VALUES
    -> (4, 'Сетевые адаптеры'),
```

```
    -> (5, 'Программное обеспечение'),
    -> (6, 'Мониторы'),
    -> (7, 'Периферия'),
    -> (8, 'CD-RW/DVD');
mysql> SELECT * FROM catalogs ORDER BY id_catalog;
+------------+-------------------------+
| id_catalog | name                    |
+------------+-------------------------+
|          1 | Процессоры              |
|          2 | Материнские платы       |
|          3 | Видеоадаптеры           |
|          4 | Сетевые адаптеры        |
|          5 | Программное обеспеченье |
|          6 | Мониторы                |
|          7 | Периферия               |
|          8 | CD-RW/DVD               |
+------------+-------------------------+
```

ГЛАВА 10

Редактирование структуры таблиц

После того, как таблицы базы данных созданы, заполнены и введены в эксплуатацию, может встать вопрос об изменении их структуры с целью оптимизации работы базы данных и использующих ее приложений. Наиболее распространенные задачи изменения структуры таблицы включают изменение порядка следования столбцов, их названий, типов, добавление новых индексов и т. д. Все эти операции производятся при помощи оператора SQL `ALTER TABLE`, которому посвящена данная глава.

Оператор `ALTER TABLE` имеет следующий синтаксис

```
ALTER TABLE tbl alter_specification [,alter_specification] ...
```

Сразу после оператора `ALTER TABLE` следует имя таблицы `tbl`, которая подвергается изменению, и производимое изменение `alter_specification`, которых может быть несколько. Более подробное описание оператора производится в соответствующих разделах данной главы.

Замечание

Оператор `ALTER TABLE` во время работы создает временную копию исходной таблицы. Требуемое изменение выполняется на копии, затем исходная таблица удаляется, а новой переименовывается ее значение. Это делается для того, чтобы в новую таблицу автоматически попадали все обновления, кроме неудавшихся. Во время выполнения оператора `ALTER TABLE` исходная таблица доступна для чтения другим клиентам.

10.1. Добавление и удаление столбцов

Для добавления столбца используется форма оператора `ALTER TABLE – ADD [COLUMN]`, представленная в листинге 10.1, где в таблицу `products` (см. листинг 4.25) учебной базы данных `shop` добавляется целочисленный столбец `rebate`, в который можно поместить скидку на товар в процентах.

Листинг 10.1. Добавление нового столбца `rebate` в таблицу `products`

```
mysql> ALTER TABLE products ADD COLUMN rebate INT;
mysql> DESCRIBE products;
```

```
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| id_product  | int(11)      | NO   |     |         |       |
| name        | tinytext     | NO   |     |         |       |
| price       | decimal(7,2) | YES  |     | 0.00    |       |
| count       | int(11)      | YES  |     | 0       |       |
| mark        | float(4,1)   | NO   |     | 0.0     |       |
| description | text         | YES  |     | NULL    |       |
| id_catalog  | int(11)      | NO   |     |         |       |
| rebate      | int(11)      | YES  |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
```

Удаление столбца производится при помощи оператора `ALTER TABLE — DROP [COLUMN]`. В листинге 10.2 демонстрируется удаление вновь созданного столбца `rebate`.

Листинг 10.2. Удаление столбца `rebate` из таблицы `products`

```
mysql> ALTER TABLE products DROP COLUMN rebate;
mysql> DESCRIBE products;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| id_product  | int(11)      | NO   |     |         |       |
| name        | tinytext     | NO   |     |         |       |
| price       | decimal(7,2) | YES  |     | 0.00    |       |
| count       | int(11)      | YES  |     | 0       |       |
| mark        | float(4,1)   | NO   |     | 0.0     |       |
| description | text         | YES  |     | NULL    |       |
| id_catalog  | int(11)      | NO   |     |         |       |
+-------------+--------------+------+-----+---------+-------+
```

Замечание

Ключевое слово `COLUMN` является необязательным и может быть опущено как при добавлении нового столбца, так и при его удалении из таблицы. Здесь и далее элементы, заключенные в квадратные скобки, являются необязательными.

Замечание

Если таблица содержит только один столбец, то этот столбец не может быть удален. Вместо этого можно удалить данную таблицу, используя команду `DROP TABLE`.

Как видно из листинга 10.1, новый столбец `rebate` был добавлен в конец таблицы. Для того чтобы изменить позицию, в которую будет помещен столбец, совместно с `ALTER TABLE — ADD [COLUMN]` используются ключевые слова `FIRST` и `AFTER`. Ключевое слово `FIRST` требует, чтобы новый столбец был размещен первым, а `AFTER` позволяет

указать, после какого столбца следует поместить новый. В листинге 10.3 демонстрируется добавление столбца `id_first` в начало таблицы и столбца `rebate` — после `price`.

Листинг 10.3. Добавление столбцов

```
mysql> ALTER TABLE products ADD id_first INT FIRST,
    -> ADD rebate INT AFTER price;
mysql> DESCRIBE products;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| id_first    | int(11)      | YES  |     | NULL    |       |
| id_product  | int(11)      | NO   |     |         |       |
| name        | tinytext     | NO   |     |         |       |
| price       | decimal(7,2) | YES  |     | 0.00    |       |
| rebate      | int(11)      | YES  |     | NULL    |       |
| count       | int(11)      | YES  |     | 0       |       |
| mark        | float(4,1)   | NO   |     | 0.0     |       |
| description | text         | YES  |     | NULL    |       |
| id_catalog  | int(11)      | NO   |     |         |       |
+-------------+--------------+------+-----+---------+-------+
```

10.2. Изменение уже существующих столбцов

Помимо ввода в таблицу новых столбцов, часто встает задача изменения типа или названия уже существующего столбца. Для изменения типа уже существующего столбца используется ключевое слово `MODIFY`, после которого указывается имя модифицируемого столбца и его новый тип. В листинге 10.4 целочисленный тип столбца `id_first`, добавленного в листинге 10.3, меняется на текстовый.

Листинг 10.4. Изменение типа столбца `id_first`

```
mysql> ALTER TABLE products MODIFY id_first TEXT;
mysql> DESCRIBE products;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| id_first    | text         | YES  |     | NULL    |       |
| id_product  | int(11)      | NO   |     |         |       |
| name        | tinytext     | NO   |     |         |       |
| price       | decimal(7,2) | YES  |     | 0.00    |       |
| rebate      | int(11)      | YES  |     | NULL    |       |
| count       | int(11)      | YES  |     | 0       |       |
```

```
| mark        | float(4,1)   | NO   |     | 0.0     |       |
| description | text         | YES  |     | NULL    |       |
| id_catalog  | int(11)      | NO   |     |         |       |
+-------------+--------------+------+-----+---------+-------+
```

Точно так же, как и при создании столбца, использование ключевых слов `FIRST` и `AFTER` позволяет изменить порядок следования столбцов. В листинге 10.5 столбцы `id_first` и `rebate` меняются местами.

Листинг 10.5. Изменение порядка следования столбцов

```
mysql> ALTER TABLE products MODIFY id_first TEXT AFTER price;
mysql> ALTER TABLE products MODIFY rebate INT FIRST;
mysql> DESCRIBE products;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| rebate      | int(11)      | YES  |     | NULL    |       |
| id_product  | int(11)      | NO   |     |         |       |
| name        | tinytext     | NO   |     |         |       |
| price       | decimal(7,2) | YES  |     | 0.00    |       |
| id_first    | text         | YES  |     | NULL    |       |
| count       | int(11)      | YES  |     | 0       |       |
| mark        | float(4,1)   | NO   |     | 0.0     |       |
| description | text         | YES  |     | NULL    |       |
| id_catalog  | int(11)      | NO   |     |         |       |
+-------------+--------------+------+-----+---------+-------+
```

Ключевое слово `MODIFY` в операторе `ALTER TABLE` не позволяет изменить имя столбца, для этого предназначено ключевое слово `CHANGE`. После данного ключевого слова указывается имя столбца, затем следует новое имя и новый тип. Даже если тип остается прежним, его необходимо воспроизвести. В листинге 10.6 столбец `id_first` переименовывается в `id_second`.

Листинг 10.6. Переименование столбца

```
mysql> ALTER TABLE products CHANGE id_first id_second TEXT;
mysql> DESCRIBE products;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| rebate      | int(11)      | YES  |     | NULL    |       |
| id_product  | int(11)      | NO   |     |         |       |
| name        | tinytext     | NO   |     |         |       |
| price       | decimal(7,2) | YES  |     | 0.00    |       |
| id_second   | text         | YES  |     | NULL    |       |
| count       | int(11)      | YES  |     | 0       |       |
```

```
| mark        | float(4,1)   | NO   |     | 0.0     |       |
| description | text         | YES  |     | NULL    |       |
| id_catalog  | int(11)      | NO   |     |         |       |
+-------------+--------------+------+-----+---------+-------+
```

Как и в случае `MODIFY`, совместно с ключевым словом `CHANGE` допускается использование ключевых слов `FIRST` и `AFTER`, позволяющих изменить позицию столбца. Если при помощи `CHANGE` изменяется только тип столбца, но не его имя, название столбца записывается в операторе `ALTER TABLE` два раза. В листинге 10.7 решается задача смены типа столбца `id_second` с `TEXT` на `INT`, а типа столбца `rebate` с `INT` на `TEXT`, при этом столбцы размещаются в конце таблицы.

Листинг 10.7. Использование ключевого слова `CHANGE`

```
mysql> ALTER TABLE products
    -> CHANGE id_second id_second INT AFTER id_catalog,
    -> CHANGE rebate rebate TEXT AFTER id_catalog;
mysql> DESCRIBE products;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| id_product  | int(11)      | NO   |     |         |       |
| name        | tinytext     | NO   |     |         |       |
| price       | decimal(7,2) | YES  |     | 0.00    |       |
| count       | int(11)      | YES  |     | 0       |       |
| mark        | float(4,1)   | NO   |     | 0.0     |       |
| description | text         | YES  |     | NULL    |       |
| id_catalog  | int(11)      | NO   |     |         |       |
| rebate      | text         | YES  |     | NULL    |       |
| id_second   | int(11)      | YES  |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
```

10.3. Добавление и удаление индексов

Оператор `ALTER TABLE`, помимо добавления новых столбцов, позволяет создавать в таблице индексы. Для добавления индексов предназначены четыре ключевых слова:

- `ADD INDEX [index_name] (index_columns)` — создание обычного индекса;
- `ADD UNIQUE [index_name] (index_columns)` — создание уникального индекса;
- `ADD PRIMARY KEY (index_columns)` — создание первичного ключа;
- `ADD FULLTEXT [index_name] (index_columns)` — создание полнотекстового поиска.

Замечание

Полнотекстовый поиск подробно обсуждается в *главе 18*.

Все индексы за исключением PRIMARY KRY могут быть снабжены необязательным именем index_name, в скобках указывается столбец (или столбцы, если индекс многостолбцовый) index_columns.

В листинге 10.8 создается первичный ключ для таблицы products (см. листинг 4.25) и индексируется столбец id_catalog с созданием индекса id_catalog_index.

Листинг 10.8. Создание индексов при помощи оператора ALTER TABLE

```
mysql> ALTER TABLE products ADD PRIMARY KEY (id_product),
    -> ADD INDEX id_catalog_index (id_catalog);
mysql> DESCRIBE products;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| id_product  | int(11)      | NO   | PRI |         |       |
| name        | tinytext     | NO   |     |         |       |
| price       | decimal(7,2) | YES  |     | 0.00    |       |
| count       | int(11)      | YES  |     | 0       |       |
| mark        | float(4,1)   | NO   |     | 0.0     |       |
| description | text         | YES  |     | NULL    |       |
| id_catalog  | int(11)      | NO   | MUL |         |       |
+-------------+--------------+------+-----+---------+-------+
```

Удаление индексов осуществляется при помощи двух ключевых слов:

- DROP INDEX index_name — удаление индекса с именем index_name;
- DROP PRIMARY KEY — удаление первичного ключа таблицы.

Замечание

Как уникальные (UNIQUE), так и обычные индексы удаляются при помощи единой конструкции DROP INDEX. Для удаления первичного ключа предусмотрена отдельная конструкция DROP PRIMARY KEY, для которой не предусмотрена передача имени индекса index_name, т. к. первичный ключ в таблице всегда один и не имеет имени.

В листинге 10.9 происходит удаление индексов, созданных SQL-запросом из листинга 10.8.

Листинг 10.9. Удаление индексов при помощи оператора ALTER TABLE

```
mysql> ALTER TABLE products DROP PRIMARY KEY,
    -> DROP INDEX id_catalog_index;
mysql> DESCRIBE products;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| id_product  | int(11)      | NO   |     |         |       |
| name        | tinytext     | NO   |     |         |       |
| price       | decimal(7,2) | YES  |     | 0.00    |       |
```

```
| count       | int(11)      | YES  |     | 0       |       |
| mark        | float(4,1)   | NO   |     | 0.0     |       |
| description | text         | YES  |     | NULL    |       |
| id_catalog  | int(11)      | NO   |     |         |       |
+-------------+--------------+------+-----+---------+-------+
```

10.4. Преобразование таблицы

Помимо изменения структуры таблицы, оператор `ALTER TABLE` позволяет изменять параметры таблицы, например, ее название. Для этого используется конструкция `RENAME [TO] new_table`, в результате применения которой таблица получает новое имя `new_table`. В листинге 10.10 таблица `products` базы данных `shop` переименовывается в `components`.

Замечание

Необязательное ключевое слово `TO` в опции `RENAME [TO]` может быть опущено.

Листинг 10.10. Переименование таблицы `products` в `components`

```
mysql> SHOW TABLES;
+----------------+
| Tables_in_shop |
+----------------+
| catalogs       |
| orders         |
| products       |
| users          |
+----------------+
mysql> ALTER TABLE products RENAME TO components;
mysql> SHOW TABLES;
+----------------+
| Tables_in_shop |
+----------------+
| catalogs       |
| components     |
| orders         |
| users          |
+----------------+
```

Для переименования таблиц существует также отдельный оператор `RENAME TABLE`, который имеет следующий синтаксис:

```
RENAME TABLE tbl_name TO new_tbl_name[, tbl_name2 TO new_tbl_name2,...]
```

Результатом работы оператора является переименование таблицы `tbl_name` в `new_tbl_name`. В одном операторе можно переименовать сразу несколько таблиц. В листинге 10.11 таблицы `users` и `orders` меняются именами.

Листинг 10.11. Обмен именами таблиц `users` и `orders`

```
mysql> RENAME TABLE users TO backup_table,
    -> orders TO users,
    -> backup_table TO orders;
mysql> DESCRIBE users;
+------------+----------+------+-----+------------+----------------+
| Field      | Type     | Null | Key | Default    | Extra          |
+------------+----------+------+-----+------------+----------------+
| id_order   | int(11)  | NO   | PRI | NULL       | auto_increment |
| id_user    | int(11)  | NO   | MUL | 0          |                |
| ordertime  | datetime | NO   |     | 0000-00-00 |                |
| number     | int(11)  | NO   |     | 0          |                |
| id_product | int(11)  | NO   | MUL | 0          |                |
+------------+----------+------+-----+------------+----------------+
```

Так как в одной базе данных не может быть двух таблиц с одинаковыми именами, в листинге 10.11, таблице `users` потребовалось назначить временное имя `backup_table`, которое потом заменяется на `orders`.

Конструкция `ORDER BY field_name` позволяет отсортировать записи таблицы по столбцу `field_name`. Следует учитывать, что созданная таблица не будет сохранять этот порядок строк после операций вставки и удаления. Так в листинге 10.12 записи таблицы `catalogs` сортируются по полю `name`.

Листинг 10.12. Сортировка записей таблицы `catalogs` по полю `name`

```
mysql> SELECT * FROM catalogs;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 | Оперативная память |
+------------+--------------------+
mysql> ALTER TABLE catalogs ORDER BY name;
mysql> SELECT * FROM catalogs;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          2 | Материнские платы  |
|          5 | Оперативная память |
|          1 | Процессоры         |
+------------+--------------------+
```

В некоторых случаях эта возможность может облегчить операцию сортировки в MySQL, если таблица имеет такое расположение столбцов, которое вам нужно в дальнейшем. Эта опция в основном полезна, если заранее известен порядок, в котором преимущественно будут запрашиваться строки.

Замечание

В большинстве случаев предпочтительнее использовать ключевое слово `ORDER BY` в операторе `SELECT`.

Конструкция `AUTO_INCREMENT` позволяет назначить новое значение для параметра таблицы `AUTO_INCREMENT`. В листинге 10.13 для таблицы `catalogs` выставляется значение `AUTO_INCREMENT` равное 1000, в результате чего добавление нового значения приведет к тому, что первичный ключ таблицы `id_catalog` получит значение 1000.

Листинг 10.13. Изменение значений параметра `AUTO_INCREMENT`

```
mysql> ALTER TABLE catalogs AUTO_INCREMENT = 1000;
mysql> INSERT INTO catalogs () VALUES ();
mysql> SELECT * FROM catalogs;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 | Оперативная память |
|       1000 |                    |
+------------+--------------------+
```

Помимо параметра `AUTO_INCREMENT`, таблицы имеют множество других параметров, которые можно изменять при помощи оператора `ALTER TABLE`. Подробнее параметры таблиц рассматриваются в *главе 11*.

ЧАСТЬ II

Глава 11. Типы и структура таблиц

Глава 12. Приведение типов

Глава 13. Операторы и математические функции

Глава 14. Функции даты и времени

Глава 15. Строковые функции

Глава 16. Безопасность и MySQL

Глава 17. Поиск и регулярные выражения

Глава 18. Полнотекстовый поиск

Глава 19. Функции, применяемые вместе с конструкцией *GROUP BY*

Глава 20. Разные функции

Глава 21. Переменные и временные таблицы

Глава 22. Многотабличные запросы

Глава 23. Вложенные запросы

Глава 24. Транзакции и блокировки

Глава 25. Управление учетными записями пользователей

Глава 26. Предотвращение катастроф и восстановление

Глава 27. Административные команды

ГЛАВА 11

Типы и структура таблиц

В *главе 4* упоминалось, что СУБД MySQL поддерживает несколько видов таблиц, каждая из которых имеет свои возможности и ограничения. В данной главе типы таблиц будут рассмотрены более подробно.

Замечание

По мере рассмотрения различных типов таблиц будут рассмотрены элементы оператора `CREATE TABLE`, характерные для того или иного типа таблиц. Полное описание синтаксиса оператора `CREATE TABLE` приводится в *разделе 11.11*.

11.1. MyISAM

Тип таблиц MyISAM является "родным" для базы СУБД MySQL. Если в операторе `CREATE TABLE` *(см. главу 4)* тип таблицы не указывается — назначается именно этот тип.

Замечание

Долгое время наряду с MyISAM в MySQL присутствовал устаревший тип таблиц ISAM, но в версии MySQL 4.1 он был объявлен официально устаревшим, а в MySQL 5.0 его поддержка была исключена. Тип таблиц MyISAM является модифицированной и улучшенной версией ISAM.

Замечание

Тип таблиц MyISAM был введен в MySQL, начиная с версии 3.23.0.

База данных в MySQL организуется как поддиректория каталога mysql/data. Таблицы базы данных организуются как файлы данной директории. Каждая MyISAM-таблица хранится на диске в трех файлах, имена которых совпадают с названием таблицы, а расширение может принимать одно из следующих значений:

- `frm` — содержит структуру таблицы, в файле данного типа хранится информация об именах и типах столбцов и индексов;
- `MYD` — файл, в котором содержатся данные таблицы (MYData);

- `MYI` — файл, в котором содержатся индексы таблицы (MYIndex).

Таблицы MyISAM обладают рядом особенностей.

- Данные хранятся в кроссплатформенном формате, это позволяет переносить базы данных с сервера непосредственным копированием файлов, минуя промежуточные форматы.
- Максимальное число индексов в таблице составляет 64 (32 до появления версии 4.1.2). Каждый индекс может состоять максимум из 16 столбцов.
- Начиная с версии MySQL 4.1, для каждого из текстовых столбцов может быть назначена своя кодировка.
- Допускается индексирование текстовых столбцов, в том числе и переменной длины.
- Поддерживается полнотекстовый поиск.
- Каждая таблица имеет специальный флаг, указывающий правильность закрытия таблиц. Если сервер останавливается аварийно, то при его повторном старте незакрытые флаги сигнализируют о возможных сбойных таблицах, сервер автоматически проверяет их и пытается восстановить.
- Расширенная поддержка кодировок.

При создании таблиц с использованием оператора `CREATE TABLE` *(см. главу 4)* можно указать тип таблицы при помощи ключевого слова `ENGINE` или `TYPE`.

Листинг 11.1. Создание MyISAM-таблицы

```
CREATE TABLE tbl (i INT) ENGINE = MyISAM;
CREATE TABLE tbl (i INT) TYPE = MyISAM;
```

Использование ключевого слова `ENGINE` является более предпочтительным, но в отличие от `TYPE` не может быть применено в MySQL до версии 4.0.18. Для создания MyISAM-таблиц можно не указывать ключевое слово `ENGINE`, т. к. данный тип таблиц назначается по умолчанию.

11.2. MERGE

Тип таблиц MERGE позволяет сгруппировать несколько таблиц типа MyISAM в одну. Такой тип таблиц применяется главным образом для снятия ограничения на объем для таблиц MyISAM, например, если операционная или файловая системы не позволяют создавать таблицы больше 4 Гбайт. Таблицы MyISAM, которые подвергаются объединению в одну таблицу MERGE, должны иметь одинаковую структуру, т. е. одинаковые столбцы и индексы, а также порядок их следования.

При создании таблицы типа MERGE будут образованы файлы структуры таблицы с расширением frm и файлы с расширением MRG. Файл MRG содержит список индексных файлов (файлы MYI), работа с которыми должна осуществляться как с еди-

ным файлом. До версии MySQL 4.1.1 все используемые таблицы должны были размещаться в той же базе данных, что и таблица MERGE.

К полученной объединенной таблице можно применять команды `SELECT`, `DELETE` и `UPDATE`. Если же попытаться применить к таблице MERGE команду `DROP TABLE`, она уничтожит MERGE-таблицу, не затронув при этом MyISAM-таблицы, входящие в ее состав.

Замечание

Тип таблиц MERGE был введен в MySQL, начиная с версии 3.23.25.

В листинге 11.2 приведен пример создания MERGE-таблицы `total` из двух MyISAM таблиц: `tbl1` и `tbl2`. Каждая из этих таблиц заполняется при помощи многострочного оператора `INSERT` тремя значениями. Так как первичный ключ таблиц снабжен атрибутом `AUTO_INCREMENT`, он автоматически получает уникальное значение в пределах таблицы.

Листинг 11.2. Создание MERGE-таблицы

```
CREATE TABLE tbl1 (
id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
name CHAR(20));
CREATE TABLE tbl2 (
id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
name CHAR(20));
INSERT INTO tbl1 (name) VALUES ('12'), ('34'), ('56');
INSERT INTO tbl2 (name) VALUES ('78'), ('9a'), ('bc');
CREATE TABLE total (
id INT NOT NULL AUTO_INCREMENT,
name CHAR(20), INDEX(id))
ENGINE=MERGE UNION=(tbl1,tbl2) INSERT_METHOD=LAST;
```

Результат выборки из таблицы `total` может выглядеть так, как это представлено в листинге 11.3.

Листинг 11.3. Выборка из таблицы `total`

```
SELECT * FROM total;
+----+------+
| id | name |
+----+------+
|  1 | 12   |
|  2 | 34   |
|  3 | 56   |
|  1 | 78   |
|  2 | 9a   |
|  3 | bc   |
+----+------+
6 rows in set (0.00 sec)
```

Важно отметить, что в объединенной таблице `total` поле `id` уже не является уникальным.

Замечание

Таблицы MERGE не могут содержать уникальных ключей.

Замечание

В параметрах `ENGINE` и `TYPE` в качестве синонима `MERGE` можно применять `MRG_MYISAM`.

Как видно из листинга 11.2, помимо указания типа таблицы при помощи ключевого слова `TYPE`, в конце оператора `CREATE TABLE` ключевое слово `UNION` указывает имена таблиц, которые объединяются в конечную MERGE-таблицу. Кроме этого, ключевое слово `INSERT_METHOD` позволяет указывать метод добавления новых записей в MyISAM-таблицы. В качестве значения данного поля передается одна из трех констант:

- `FIRST` — при вставке новой записи в MERGE-таблицу запись размещается в первой таблице из списка, приведенного после ключевого слова `UNION`;
- `LAST` — при вставке новой записи в MERGE-таблицу запись размещается в последней таблице из списка, приведенного после ключевого слова `UNION`;
- `NO` — данное значение аналогично ситуации, когда ключевое слово `INSERT_METHOD` опускается из определения таблицы (`CREATE TABLE`). Использование оператора `INSERT` в такой таблице приведет к ошибке.

Так в листинге 11.4 демонстрируется вставка нового значения в таблицу `total`, в определении которой опущен параметр `INSERT_METHOD`.

Листинг 11.4. Ошибочное добавление новой записи в таблицу MERGE

```
INSERT INTO total VALUES (NULL, 'xx');
ERROR 1031: Table handler for 'total' doesn't have this option
```

11.3. MEMORY (HEAP)

Тип таблиц MEMORY хранится в оперативной памяти, поэтому все запросы к такой таблице выполняются очень быстро. Недостатком является полная потеря данных в случае сбоя работы сервера, поэтому в таблице данного типа хранят только временную информацию, которую можно легко восстановить заново.

Замечание

Тип HEAP является синонимом MEMORY. Последнее ключевое слово было введено в MySQL 4.1.0 и его использование является более предпочтительным по отношению к HEAP.

Замечание

Тип таблиц HEAP был введен в MySQL, начиная с версии 3.23.0.

При создании таблицы типа MEMORY она ассоциируется с одним-единственным файлом, имеющим расширение frm, в котором определяется структура таблицы. При остановке или перезапуске сервера данный файл остается в текущей базе данных, но содержимое таблицы, которое хранится в оперативной памяти машины, теряется. То есть при каждом рестарте MySQL-сервера создавать новую таблицу MEMORY не нужно.

Так как размещение данных происходит в оперативной памяти, MEMORY-таблицы имеют несколько ограничений.

- Индексы используются только в операциях сравнения совместно с операторами "=" и "<=>", с другими операторами, такими как ">" или "<", индексирование столбцов не имеет смысла.
- Так же, как и в случае MERGE-таблиц, возможно использование только неуникальных индексов.
- MEMORY-таблицы используют записи фиксированной длины, поэтому в них не допустимы столбцы типов `TEXT` и `BLOB`.
- В версиях, предшествующих MySQL 4.1, не поддерживается атрибут `AUTO_INCREMENT`.
- В версиях, предшествующих MySQL 4.0.2, не поддерживается индексирование столбцов, содержащих `NULL`-значения.

Для создания таблиц данного типа параметру `ENGINE` в операторе `CREATE TABLE` необходимо присвоить значение `MEMORY` или `HEAP`.

Листинг 11.5. Создание MEMORY-таблицы

```
CREATE TABLE test (i INT) ENGINE=MEMORY;
CREATE TABLE test (i INT) ENGINE=HEAP;
```

11.4. EXAMPLE

Данный тип таблиц является заглушкой, можно создать таблицу этого типа, но сохранить или получить из нее данные нельзя (листинг 11.6). При создании таблиц данного типа точно так же, как и в случае MEMORY, создается один файл с расширением frm, в котором определяется структура таблицы.

Тип EXAMPLE был введен для удобства сторонних разработчиков и демонстрирует, каким образом следует создавать свои собственные типы таблиц. Не нужно забывать, что MySQL является базой данных с открытым кодом и любой желающий может модифицировать ее код для своих нужд. Так типы таблиц InnoDB и BDB, рассматриваемые далее, были разработаны сторонними компаниями.

Замечание

Тип таблиц EXAMPLE был введен в MySQL, начиная с версии 4.1.3.

Для создания EXAMPLE-таблицы параметру `ENGINE` в операторе `CREATE TABLE` необходимо присвоить значение `EXAMPLE`.

Листинг 11.6. Создание EXAMPLE-таблицы

```
CREATE TABLE test (i INT) ENGINE = EXAMPLE;
Query OK, 0 rows affected (0.09 sec)
INSERT INTO test VALUES(1),(2),(3);
ERROR 1031 (HY000): Table storage engine for 'test' doesn't have this option
SELECT * FROM test;
Empty set (0.03 sec)
```

11.5. BDB (BerkeleyDB)

Таблицы типа BDB (сокращение от BerkeleyDB) обслуживаются транзакционным обработчиком BerkeleyDB, разработанным компанией Sleepycat. При создании таблиц данного типа формируются два файла. Первый с расширением frm, в котором определяется структура базы данных, а второй с расширением db, в котором размещаются данные и индексы.

Замечание

Тип таблиц BDB был введен в MySQL, начиная с версии 3.23.34.

Далее перечислены особенности таблиц BDB.

- Для каждой из таблиц ведется журнал. Это позволяет значительно повысить устойчивость базы и увеличить вероятность успешного восстановления после сбоя.
- Таблицы BDB хранятся в виде бинарных деревьев (b-деревьев). Такое представление замедляет сканирование таблицы (например, если извлекаются все строки) и увеличивает занимаемое место на жестком диске по сравнению с другими типами таблиц. С другой стороны, поиск отдельных значений в таких таблицах осуществляется быстрее. Так, для сравнения, остальные типы таблиц хранят в виде бинарных деревьев свои индексы.
- Каждая таблица BDB должна иметь первичный ключ, в случае его отсутствия создается скрытый первичный ключ, снабженный атрибутом `AUTO_INCREMENT`.
- Поддерживаются транзакции на уровне страниц.
- Подсчет числа строк в таблице при помощи встроенной функции `COUNT()` осуществляется медленнее, чем для MyISAM-таблиц, т. к. в отличие от последних, для BDB-таблиц не поддерживается подсчет количества строк в таблице и MySQL вынуждена каждый раз сканировать таблицу заново.
- Сканирование осуществляется медленнее, чем в таблицах MyISAM, т. к. данные в таблицах BDB хранятся в b-деревьях, а не в отдельных файлах данных.
- Ключи не являются упакованными, как в MyISAM, т. е. ключи в таблицах BDB занимают больше места по сравнению с ключами таблиц MyISAM.

- Если таблица BDB займет все пространство на диске, то будет выведено сообщение об ошибке и выполнен откат транзакции. В отличие от BDB, таблицы MyISAM и ISAM будут ожидать пока не появится свободное место, а потом продолжат работу, т. е. произойдет зависание.
- Для обеспечения блокировок таблиц на уровне операционной системы в файл db в момент создания таблицы записывается путь к файлу. Это приводит к тому, что файлы нельзя перемещать из текущего каталога в другой.
- При создании резервных копий таблиц BDB необходимо использовать утилиту mysqldump или создать резервные копии всех файлов db и файлов журналов BDB. Файлы журналов BDB — это файлы в каталоге mysql/data с именами log.XXXXXX (6 цифр). Обработчик таблицы BDB хранит незавершенные транзакции в файлах журналов, их наличие требуется при запуске сервера MySQL.

Для создания BDB-таблицы необходимо указать в качестве значения параметра `ENGINE` в операторе `CREATE TABLE` константу BDB.

Листинг 11.7. Создание BDB-таблиц

```
CREATE TABLE t (i INT) ENGINE = BDB;
CREATE TABLE t (i INT) TYPE = BDB;
```

Замечание

В параметрах `ENGINE` и `TYPE` в качестве синонима `BDB` можно применять `BerkeleyDB`.

11.6. InnoDB

Данный тип таблиц обслуживается обработчиком InnoDB, разработанным компанией Innobase Oy. Таблицы данного типа гарантируют высокую производительность, обеспечивая устойчивое хранение данных в таблицах объемом вплоть до 1 Тбайт и нагрузкой на сервер до 800 вставок/обновлений в секунду.

Замечание

Тип таблиц InnoDB был введен в MySQL, начиная с версии 3.23.34.

Замечание

Начиная с версии 4.1.5, автоматическая программа установки MySQL под Windows (Windows installer) может назначить таблицы InnoDB типом по умолчанию.

К особенностям таблиц InnoDB относят следующие.

- В отличие от MyISAM и BDB, таблицы для InnoDB не создаются в базах данных и для каждой из таблиц не выделяется отдельного файла данных. Исключение — файл определения с расширением frm, который все же создается (по умолчанию в

базе данных `test`). Все таблицы хранятся в едином табличном пространстве, поэтому имена таблиц должны быть уникальными.

- Хранение данных в едином табличном пространстве позволяет снять ограничения на объем таблиц, т. к. файл с таблицами может быть разбит на несколько частей и распределен по нескольким дискам или даже хостам.
- Данный тип таблиц поддерживает автоматическое восстановление после сбоев.
- Обеспечивается поддержка транзакций.
- Это единственный тип таблиц в MySQL, поддерживающий внешние ключи и каскадное удаление.
- Выполняется блокировка на уровне отдельных записей.
- Расширенная поддержка кодировок.

Для создания InnoDB-таблицы необходимо указать в качестве значения параметра `ENGINE` в операторе `CREATE TABLE` константу InnoDB.

Листинг 11.8. Создание InnoDB-таблиц

```
CREATE TABLE CUSTOMER (A INT, B CHAR (20), INDEX (A)) ENGINE = InnoDB;
CREATE TABLE CUSTOMER (A INT, B CHAR (20), INDEX (A)) TYPE = InnoDB;
```

11.7. NDB Cluster

Тип таблиц NDB Cluster предназначен для организации кластеров MySQL, когда таблицы распределены между несколькими компьютерами, объединенными в локальную сеть.

Замечание

Тип таблиц NDB Cluster был введен в MySQL, начиная с версии 4.1.2.

11.8. ARCHIVE

Тип таблиц ARCHIVE введен для хранения большого объема данных в сжатом формате. При создании таблиц данного типа MySQL, так же как и для таблиц любого другого типа, создает файл с именем, совпадающим с именем таблицы и расширением frm, в котором хранится определение структуры таблицы. Помимо этого, создаются два файла с расширениями ARZ и ARM, в которых хранятся данные и мета-данные соответственно. Кроме этого, при оптимизации таблицы может появиться файл с расширением ARN.

Замечание

Тип таблиц ARCHIVE был введен в MySQL, начиная с версии 4.1.3.

Замечание

MySQL в бинарных кодах распространяется без поддержки данного типа таблиц, для его включения необходима перекомпиляция MySQL с включенной опцией `--with-archive-storage-engine`.

ARCHIVE-таблицы поддерживают только два SQL-оператора: `INSERT` и `SELECT`. Причем оператор `SELECT` выполняется по методу полного сканирования таблицы. Любые другие операторы, включая `DELETE`, `UPDATE` и `REPLACE`, не поддерживаются.

11.9. CSV

Формат CSV, представляющий собой обычный текстовый файл, записи в котором хранятся в строках, а поля разделены точкой с запятой, широко распространен в компьютерном мире. Следует отметить, что MS Excel позволяет сохранять таблицы в этом формате.

Замечание

Тип таблиц CSV был введен в MySQL, начиная с версии 4.1.4.

Замечание

MySQL в бинарных кодах распространяется без поддержки данного типа таблиц, для его включения необходима перекомпиляция MySQL с включенной опцией `--with-csv-storage-engine`.

При создании CSV-таблицы в директории с текущей базой данных формируются два файла с именами, совпадающими с именем таблицы, и расширениями frm и csv. В первом файле хранится определение структуры таблицы, а во втором — данные в CSV-формате. Любая другая программа, поддерживающая CSV-формат, например, MS Excel, может открыть данный файл.

В листинге 11.9 приведены SQL-запросы, создающие и заполняющие CSV-таблицу.

Листинг 11.9. Создание CSV-таблицы

```
CREATE TABLE test(i INT, c CHAR(10)) ENGINE = CSV;
INSERT INTO test VALUES(1,'record one'),(2,'record two');
```

Результат выборки при помощи оператора `SELECT` представлен в листинге 11.10.

Листинг 11.10. Выборка из таблицы

```
SELECT * FROM test;
+------+------------+
| i    | c          |
+------+------------+
|    1 | record one |
|    2 | record two |
+------+------------+
2 rows in set (0.00 sec)
```

11.10. FEDERATED

Тип таблиц FEDERATED позволяет хранить данные в удаленных таблицах, расположенных на другой машине сети. Во время создания таблицы в локальной директории данных mysql/data создается только файл определения структуры таблицы с расширением frm, никаких других файлов не создается, т. к. все данные хранятся на удаленной машине.

Замечание

Тип таблиц FEDERATED был введен в MySQL, начиная с версии 5.0.3.

Замечание

MySQL в бинарных кодах распространяется без поддержки данного типа таблиц, для его включения необходима перекомпиляция MySQL с включенной опцией `--with-federated-storage-engine`.

При работе с обычными локальными таблицами MySQL читает структуру таблицы из файла с расширением frm и обращается к файлам данных MYD для разбора данных и выполнения операций, которые необходимы для реализации SQL-запроса. При работе с FEDERATED-таблицами синтаксис запроса проверяется на локальной машине, после чего он отправляется на удаленную машину, где он выполняется, далее результаты отправляются обратно на локальную машину — сервер MySQL, — которой возвращает их клиенту.

Для создания таблицы типа FEDERATED необходимо сначала создать таблицу на удаленной машине, например, в базе данных `federated` (листинг 11.11).

Листинг 11.11. Создание MyISAM-таблицы на удаленной машине

```
CREATE TABLE test_table (
    id     INT(20) NOT NULL auto_increment,
    name   VARCHAR(32) NOT NULL default '',
    other  INT(20) NOT NULL default '0',
    PRIMARY KEY (id),
    KEY name (name),
    KEY other_key (other)
) ENGINE = MyISAM;
```

В качестве типа таблицы (`ENGINE`) может выступать любой тип, не обязательно MyISAM. После создания таблицы на удаленном сервере можно приступать к развертыванию FEDERATED-таблицы на локальной машине (листинг 11.12).

Листинг 11.12. Создание FEDERATED-таблицы

```
CREATE TABLE federated_table (
    id     INT(20) NOT NULL auto_increment,
    name   VARCHAR(32) NOT NULL default '',
```

```
    other  INT(20) NOT NULL default '0',
    PRIMARY KEY (id),
    KEY name (name),
    KEY other_key (other)
) ENGINE = FEDERATED
COMMENT = 'mysql://user_name@remote_host:3306/federated/test_table';
```

Как видно из листинга 11.12, параметр `ENGINE` принимает значение `FEDERATED`, а в комментарии `COMMENT` к таблице указываются параметры соединения с удаленным сервером. В результате выполнения SQL-запроса из листинга 11.12 в директории federated локальной машины будет создан один-единственный файл federated_table.frm.

Замечание

Структура таблиц на обоих серверах, как локальном, так и удаленном, должна быть идентична.

Общий вид строки комментария при создании FEDERATED-таблиц выглядит следующим образом:

```
scheme://user_name[:password]@host_name[:port_num]:/db_name/tbl_name
```

Протокол соединения `scheme` может принимать только одно значение — `mysql`, параметр `user_name` передает имя пользователя, от имени которого на удаленном сервере запускается запрос, через необязательный параметр `password` передается пароль пользователя. Параметр `host_name` принимает имя или IP-адрес удаленного хоста, а необязательный параметр `port_num` — порт, который прослушивается сервером MySQL на `host_name`. Параметр `db_name` определяет имя базы данных на удаленном хосте, а `tbl_name`, соответственно, имя таблицы.

Ниже приведено несколько примеров того, как может выглядеть строка комментария в FEDERATED-таблице:

```
COMMENT='mysql://username:password@hostname:port/database/tablename'
COMMENT='mysql://username@hostname/database/tablename'
COMMENT='mysql://username:password@hostname/database/tablename'
```

Замечание

Формат строки `COMMENT` в определении FEDERATED-таблицы признан не оптимальным и может претерпеть изменения в версии MySQL 5.1.

Таблицы типа FEDERATED обладают следующими ограничениями.

- Удаленным сервером может быть только MySQL-сервер. Поддержку других СУБД создатели MySQL обещают ввести в следующих версиях.
- Таблица на удаленном сервере обязательно должна существовать, иначе не удастся создать FEDERATED-таблицу на локальном сервере.
- Создание FEDERATED-таблицы, указывающей на другую FEDERATED-таблицу, возможно, но следует внимательно следить за тем, чтобы не образовывалось циклических ссылок, последствия которых непредсказуемы.

- В таблицах данного типа не поддерживаются транзакции.
- FEDERATED-таблицы позволяют работать с SQL-операторами SELECT, INSERT, UPDATE, DELETE, а также с индексированными столбцами, но операторы ALTER TABLE и DROP TABLE не поддерживаются.

11.11. Оператор *CREATE TABLE*

В предыдущих разделах данной главы, а также в *главе 4* рассматривался оператор CREATE DATABASE, позволяющий создать таблицу в текущей базе данных. Полный синтаксис данного оператора выглядит следующим образом (листинг 11.13).

Листинг 11.13. Синтаксис оператора CREATE TABLE

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    [(create_definition,...)]
    [table_options] [select_statement]
```

Необязательное ключевое слово TEMPORARY позволяет создавать временные таблицы, которые более подробно рассматриваются в *главе 21*.

> ***Замечание***
>
> Элементы в квадратных скобках являются необязательными и могут быть опущены.

Необязательная конструкция IF NOT EXISTS сообщает MySQL, что таблицу необходимо создавать только в том случае, если она не существует, в противном случае никаких действий предпринимать не следует (листинг 11.14).

Листинг 11.14. Создание уже существующей таблицы

```
mysql> CREATE TABLE tbl_name (k INT);
Query OK, 0 rows affected (0.11 sec)
mysql> CREATE TABLE tbl_name (k INT);
ERROR 1050: Table 'tbl_name' already exists
mysql> CREATE TABLE IF NOT EXISTS tbl_name (k INT);
Query OK, 0 rows affected (0.02 sec)
```

Как видно из листинга 11.14, повторное создание таблицы без конструкции IF NOT EXISTS приводит к возникновению ошибки 1050: "Таблица 'tbl_name' уже существует", однако ее применение позволяет выполнить запрос корректно. Такое поведение базы данных бывает удобным при выполнении большого числа SQL-инструкций в пакетном режиме (из файла), где ошибка вызовет остановку выполнения всего пакетного файла, несмотря на то, что дальнейшее его выполнение в такой ситуации не приведет к сбою.

Определение таблицы состоит из структуры таблицы create_definition, параметров таблицы table_options и необязательного вложенного запроса select_statement,

позволяющего создать таблицу из результата, возвращаемого оператором SELECT. Все эти конструкции будут рассмотрены далее.

Замечание

Создание таблицы с использованием вложенного запроса select_statement подробно рассматривается в *главе 23*.

11.11.1. Структура таблицы

Структура таблицы, следующая за именем таблицы tbl_name, представляет собой список create_definition, который определяет имена столбцов и индексов. В качестве элемента списка могут выступать:

- описание столбца, например, counter BIGINT;
- определение индекса.

Описание столбца включает имя столбца, следующий за ним тип и несколько необязательных ключевых слов. Типы столбцов подробно рассматриваются в *главе 4*, здесь же будут описаны ключевые слова, которые могут следовать за типом столбца.

- NULL — означает, что столбец может содержать значение NULL (по умолчанию).
- NOT NULL — столбец не может содержать значение NULL.
- DEFAULT default_value — определяет значение столбца, который он принимает по умолчанию, если ему не передано значение при вставке записи оператором INSERT. Это ключевое слово нельзя использовать совместно с типами данных BLOB и TEXT. Значения по умолчанию должны быть константами, т. е. использование встроенных функций, таких как NOW() или CURRENT_DATE(), недопустимо. Если ключевое слово DEFAULT опущено, то MySQL автоматически присваивает полю значение. В качестве значения по умолчанию в этом случае выступает значение NULL, если для столбца допустимо использование данного типа. Если использование NULL запрещено, то значение по умолчанию присваивается согласно следующим правилам:
  - для числовых типов, за исключением объявленных с атрибутом AUTO_INCREMENT, значение по умолчанию равно 0. Для столбца AUTO_INCREMENT значением по умолчанию является следующее значение в последовательности для этого столбца;
  - для типов даты и времени, отличных от TIMESTAMP, значение по умолчанию равно соответствующей нулевой величине для данного типа, например, для datetime в качестве значения по умолчанию будет назначено '0000-00-00 00:00:00'. Для столбца TIMESTAMP значение по умолчанию представляет собой текущее значение даты и времени;
  - для строковых типов, кроме ENUM, значением по умолчанию является пустая строка. Для ENUM значение по умолчанию равно первой перечисляемой величине.

- AUTO_INCREMENT — при вставке в таблицу новой записи передача в поле, снабженное данным атрибутом, величины 0 или NULL приводит к назначению ему величины, равной максимальному значению столбца плюс единица. Таблица может содержать только один столбец, снабженный атрибутом AUTO_INCREMENT, а значения первичного ключа должны быть уникальными, т. е. повторения в пределах одной таблицы не допускаются. Также для этого столбца не допускается использование ключевого слова DEFAULT.
- KEY — или INDEX, помечает столбец как обычный индекс.
- PRIMARY KEY — наличие данного индекса позволяет пометить столбец как первичный ключ.
- UNIQUE [KEY] — помечает столбец как уникальный индекс, в котором недопустимы повторяющиеся значения.
- FOREIGN KEY — внешний ключ применим только для таблиц типа InnoDB, для всех других таблиц игнорируется. Данный параметр подробно рассмотрен в *главе 23*.
- COMMENT — комментарий к столбцу, который отображается операторами SHOW CREATE TABLE и SHOW FULL COLUMNS.
- CHARACTER SET — определяет кодировку текстового столбца.
- COLLATE — определяет правила сравнения текстовых столбцов (сортировку).

Рассмотрим несколько примеров, демонстрирующих различные примеры таблиц. В листинге 11.15 представлено определение таблицы пользователей форума.

Листинг 11.15. Оператор CREATE TABLE для создания таблицы authors

```
CREATE TABLE authors (
  id_author INT(6) NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name TINYTEXT NOT NULL,
  email TINYTEXT,
  about TEXT,
  photo MEDIUMBLOB,
  puttime DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00',
  kod INT(10) NOT NULL DEFAULT '20' UNIQUE KEY,
  id BIGINT(20) NOT NULL DEFAULT '0',
  statususer ENUM('user','moderator','admin') NOT NULL DEFAULT 'user'
);
```

Таблица authors состоит из 9 столбцов.

- id_author — целочисленный столбец типа INT, который снабжен атрибутом AUTO_INCREMENT и является первичным ключом таблицы (PRIMARY KEY). При вставке новой записи при помощи оператора INSERT поле id_author можно не заполнять или передавать ему значение NULL — MySQL автоматически присвоит ему увеличенное на единицу максимальное значение столбца id_author. В таблице больше не может быть первичных ключей и столбцов, снабженных атрибутом

AUTO_INCREMENT. Кроме того, поле id_author обязательно должно быть помечено атрибутом NOT NULL.

- name — имя пользователя, поле типа TINYTEXT, способное принимать в качестве значения строку до 255 символов. Поле помечено атрибутом NOT NULL, т. е. не может принимать значение NULL. Если значение для данного поля не указывается, оно принимает в качестве значения по умолчанию пустую строку. Как было сказано выше, применять ключевое слово DEFAULT для типов TEXT/BLOB нельзя.
- email — адрес электронной почты пользователя, поле типа TINYTEXT, по умолчанию, если никаких дополнительных параметров ни указывается, оно может принимать значение NULL.
- about — "несколько слов о себе", поле типа TEXT, способное принимать в качестве значения строку до 65 535 символов.
- photo — фотография пользователя, поле типа MEDIUMBLOB, способное принимать в качестве значения бинарную информацию объемом до 16777215 байт. Поля типа BLOB, в отличие от TEXT, не имеют кодировки, поэтому данные, помещенные в них, не повреждаются.
- puttime — время регистрации пользователя, поле типа DATETIME, в котором хранится дата и время суток. Ключевое слово DEFAULT назначает полю по умолчанию значение '0000-00-00 00:00:00', можно выставить любую другую дату, например '2005-12-01 00:00:00'.
- kod — поле типа INT, принимающее по умолчанию значение 20. Ключевое слово UNIQUE KEY требует, чтобы в пределах таблицы не было двух записей с полями kod, имеющими одинаковое значение.
- id — поле типа BIGINT, принимающее по умолчанию значение 0.
- statususer — поле типа ENUM, принимающее три значения: 'user', 'moderator' и 'admin'. Поле данного типа не может иметь значение NULL, поэтому снабжается ключевым словом NOT NULL и default, которое определяет в качестве значения по умолчанию строку 'user'.

В листинге 11.16 представлен оператор CREATE TABLE для таблицы test, который демонстрирует применение конструкций COMMENT и CHARACTER SET.

Листинг 11.16. Оператор CREATE TABLE для создания таблицы test

```
CREATE TABLE test (
  dos_name TEXT CHARACTER SET cp866 COMMENT 'Кодировка DOS',
  win_name TEXT CHARACTER SET cp1251 COMMENT 'Кодировка Windows',
  koi8r_name TEXT CHARACTER SET koi8r COMMENT 'Кодировка KOI8-R',
  utf8_name TEXT CHARACTER SET utf8 COMMENT 'Юникод'
);
```

Как видно из листинга 11.16, таблица test состоит из четырех полей типа TEXT:

- win_name — текст в кодировке cp866 (русская кодировка DOS);
- win_name — текст в кодировке cp1251 (русская кодировка Windows);

- `koi8r_name` — текст в кодировке KOI8-R (кодировка, распространенная в Интернете);
- `utf8_name` — текст в кодировке UTF-8 (Юникод).

В полях таблицы `test` теперь можно хранить текст в соответствующих кодировках, и он будет сортироваться правильно. Если же в поле с кодировкой cp866 поместить текст в кодировке cp1251, то о правильной сортировке записей при помощи конструкции `ORDER BY` не может быть и речи.

Замечание

При указании кодировки столбец не должен иметь атрибут `NOT NULL`, иначе это вызовет синтаксическую ошибку.

При помощи оператора `SHOW CHARACTER SET` можно посмотреть, какие кодировки поддерживает сервер MySQL (листинг 11.17). Для различных версий их число может быть различным, например, в тестовые версии MySQL обычно не включают национальные кодировки и поддерживаются только кодировки `latain`.

Листинг 11.17. Результат работы оператора `SHOW CHARACTER SET`

```
mysql> SHOW CHARACTER SET;
+----------+-----------------------------+---------------------+--------+
| Charset  | Description                 | Default collation   | Maxlen |
+----------+-----------------------------+---------------------+--------+
| big5     | Big5 Traditional Chinese    | big5_chinese_ci     |      2 |
| dec8     | DEC West European           | dec8_swedish_ci     |      1 |
| cp850    | DOS West European           | cp850_general_ci    |      1 |
| hp8      | HP West European            | hp8_english_ci      |      1 |
| koi8r    | KOI8-R Relcom Russian       | koi8r_general_ci    |      1 |
| latin1   | ISO 8859-1 West European    | latin1_swedish_ci   |      1 |
| latin2   | ISO 8859-2 Central European | latin2_general_ci   |      1 |
| swe7     | 7bit Swedish                | swe7_swedish_ci     |      1 |
| ascii    | US ASCII                    | ascii_general_ci    |      1 |
| ujis     | EUC-JP Japanese             | ujis_japanese_ci    |      3 |
| sjis     | Shift-JIS Japanese          | sjis_japanese_ci    |      2 |
| hebrew   | ISO 8859-8 Hebrew           | hebrew_general_ci   |      1 |
| tis620   | TIS620 Thai                 | tis620_thai_ci      |      1 |
| euckr    | EUC-KR Korean               | euckr_korean_ci     |      2 |
| koi8u    | KOI8-U Ukrainian            | koi8u_general_ci    |      1 |
| gb2312   | GB2312 Simplified Chinese   | gb2312_chinese_ci   |      2 |
| greek    | ISO 8859-7 Greek            | greek_general_ci    |      1 |
| cp1250   | Windows Central European    | cp1250_general_ci   |      1 |
| gbk      | GBK Simplified Chinese      | gbk_chinese_ci      |      2 |
| latin5   | ISO 8859-9 Turkish          | latin5_turkish_ci   |      1 |
| armscii8 | ARMSCII-8 Armenian          | armscii8_general_ci |      1 |
| utf8     | UTF-8 Unicode               | utf8_general_ci     |      3 |
| ucs2     | UCS-2 Unicode               | ucs2_general_ci     |      2 |
```

```
| cp866    | DOS Russian                 | cp866_general_ci    |      1 |
| keybcs2  | DOS Kamenicky Czech-Slovak  | keybcs2_general_ci  |      1 |
| macce    | Mac Central European        | macce_general_ci    |      1 |
| macroman | Mac West European           | macroman_general_ci |      1 |
| cp852    | DOS Central European        | cp852_general_ci    |      1 |
| latin7   | ISO 8859-13 Baltic          | latin7_general_ci   |      1 |
| cp1251   | Windows Cyrillic            | cp1251_general_ci   |      1 |
| cp1256   | Windows Arabic              | cp1256_general_ci   |      1 |
| cp1257   | Windows Baltic              | cp1257_general_ci   |      1 |
| binary   | Binary pseudo charset       | binary              |      1 |
| geostd8  | GEOSTD8 Georgian            | geostd8_general_ci  |      1 |
+----------+-----------------------------+---------------------+--------+
```

Как видно из листинга 11.17, первый столбец возвращаемой таблицы содержит название кодировки, используемое в SQL-запросе `CREATE TABLE`. Второй столбец содержит официальное название кодировки, третий столбец — правила сравнения, которые назначаются для данной кодировки по умолчанию. Дело в том, что сортировка столбцов требует задать порядок сравнения букв, "А = а < Б = б < В = в" и т. д. Так как код буквы в каждой кодировке различается, необходимо составить для каждой кодировки свои правила сравнения, которые, вообще говоря, можно переопределить, указав после кодировки ключевое слово `COLLATE` (листинг 11.18). Последний столбец содержит максимальное число байт, отводимых под одну букву в кодировке.

Листинг 11.18. Применение ключевого слова `COLLATE`

```
CREATE TABLE test (
  dos_name TEXT CHARACTER SET cp866 COLLATE cp866_general_ci,
  win_name TEXT CHARACTER SET cp1251 COLLATE cp1251_general_ci,
  koi8r_name TEXT CHARACTER SET koi8r COLLATE koi8r_general_ci,
  utf8_name TEXT CHARACTER SET utf8 COLLATE utf8_general_ci
);
```

Однако в повседневной практике переопределение правил сравнения текстовых строк практически не требуется.

Комментарий столбца, который задается после ключевого слова `COMMENT`, позволяет закрепить за столбцом текст длиной не более 60 символов. Просмотреть комментарий к каждому из столбцов можно при помощи оператора `SHOW FULL COLUMNS FROM tbl`.

Листинг 11.19. Просмотр комментария таблицы

```
mysql> SHOW FULL COLUMNS FROM test;
+------------+------+------+---------+--------+------------------+
| Field      | Type | Null | Default | Privileges | Comment       |
+------------+------+------+---------+--------+------------------+
| dos_name   | text | YES  | NULL    | select | Кодировка DOS    |
| win_name   | text | YES  | NULL    | select | Кодировка Windows |
```

```
| koi8r_name | text | YES  | NULL    | select | Кодировка KOI8-R  |
| utf8_name  | text | YES  | NULL    | select | Юникод            |
+------------+------+------+---------+--------+-------------------+
```

Для удобочитаемости из конечного вывода оператора `SHOW FULL COLUMNS` исключены столбцы `Collation`, `Key` и `Extra`.

Помимо представленного в листинге 11.15 синтаксиса определения индексов, допускается их определение после перечисления всех столбцов. Такой синтаксис применяется в том случае, если индексу требуется задать имя, отличное от имени столбца, или индекс должен быть определен по нескольким столбцам. К ключевым словам для определения индексов, приведенных в начале данного раздела, добавляется ключевое слово `FULLTEXT`, объявление которого допустимо только в конце таблицы (листинг 11.20).

Листинг 11.20. Оператор `CREATE TABLE` для создания таблицы `authors`

```
CREATE TABLE authors (
  id_author INT(6) NOT NULL AUTO_INCREMENT,
  name TINYTEXT NOT NULL,
  email TINYTEXT,
  about TEXT,
  photo MEDIUMBLOB,
  puttime DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00',
  kod INT(10) NOT NULL DEFAULT '20',
  id BIGINT(20) NOT NULL DEFAULT '0',
  statususer ENUM('user', 'moderator', 'admin') NOT NULL DEFAULT 'user',
  PRIMARY KEY (id_author, id),
  UNIQUE KEY kod (kod),
  FULLTEXT KEY name (name),
  FULLTEXT KEY mailsearch (email),
  FULLTEXT KEY search (about, name)
) TYPE = MyISAM;
```

Как видно из листинга 11.20, оператор `CREATE TABLE` создает таблицу `authors`, аналогичную представленной в листинге 11.15. Отличие заключается в том, что определение ключей вынесено из определения столбцов в конец таблицы — такой синтаксис позволяет переименовать ключи или определить ключи сразу по нескольким столбцам. Так, первичный ключ определен по двум столбцам: `id_author` и `id`, комбинация которых должна быть уникальной. В таблице определено три ключа `FULLTEXT` для полнотекстового поиска по столбцам типа `TEXT`. Первый ключ `name` позволяет осуществлять поиск по столбцу `name`, второй `mailsearch` по полю `email`, а индекс `search` позволяет осуществлять поиск сразу по двум полям: `about` и `name`.

Замечание

Следует помнить, что ключевое слово `PRIMARY KEY` должно появляться в определении таблицы только один раз: либо в качестве дополнительных ключевых слов столбца, либо в конце таблицы в списке индексов.

Замечание

Полнотекстовый поиск подробно рассмотрен в *главе 18*.

11.11.2. Параметры таблицы

Параметры таблицы, обозначенные в схеме оператора `CREATE TABLE` как `table_options`, были введены, начиная с MySQL 3.23. Ниже приводятся ключевые слова, которые могут быть перечислены после структуры таблицы.

ENGINE (TYPE)

Данное ключевое слово определяет тип таблицы. Использование ключевого слова `ENGINE` является более предпочтительным, но в отличие от `TYPE` не может быть использовано в версиях MySQL младше, чем 4.0.18.

Листинг 11.21. Использование ключевого слова `ENGINE`

```
CREATE TABLE tbl_myisam (i INT) ENGINE = MyISAM;
CREATE TABLE tbl_innodb (i INT) TYPE = InnoDB;
```

Параметр `ENGINE` может принимать значения, приведенные в табл. 11.1.

Таблица 11.1. *Значения параметра* `ENGINE`

Имя таблицы	Описание
ARCHIVE	Хранения большого объема данных в сжатом формате
BDB	Транзакционные таблицы с блокировкой на уровне страниц
BerkeleyDB	Синоним для BDB
CSV	Формат CSV, записи в котором хранятся в строках, а поля разделены точкой с запятой
EXAMPLE	Данный тип таблиц является заглушкой, демонстрирующей, каким образом создаются новые типы таблиц
FEDERATED	Позволяет хранить данные в удаленных таблицах, расположенных на другой машине сети
HEAP	Синоним для MEMORY
InnoDB	Транзакционные таблицы с блокировкой на уровне записей и поддержкой внешних ключей и каскадного удаления
MEMORY	Высокоскоростная таблица, данные в которой располагаются в оперативной памяти
MERGE	Объединение нескольких MyISAM-таблиц, выступающих как одна таблица
MRG_MyISAM	Синоним для MERGE

Таблица 11.1 *(окончание)*

Имя таблицы	Описание
MyISAM	Переносимая кроссплатформенная таблица, поддерживающая полнотекстовый поиск
NDB	Таблица для кластеров
NDBCLUSTER	Синоним NDB

Более подробно типы таблиц описаны в начале данной главы.

AUTO_INCREMENT

Параметр AUTO_INCREMENT позволяет задать число, начиная с которого будет происходить увеличение счетчика для столбца, снабженного этим атрибутом (листинг 11.22).

Листинг 11.22. Использование параметра AUTO_INCREMENT

```
CREATE TABLE name (
  id_name INT(6) NOT NULL AUTO_INCREMENT,
  name TINYTEXT NOT NULL,
) ENGINE = MyISAM AUTO_INCREMENT = 144;
```

При вставке новой записи поле id_name получит значение 145, а не 1. Такой прием удобно использовать для пропуска некоторого количества значений, а также для воссоздания базы данных из дампа.

AVG_ROW_LENGTH

Приблизительная средняя длина записи для таблиц типа MyISAM. Установка данного параметра имеет смысл только для таблиц с записями переменной длины.

Во время создания таблицы MySQL оценивает ее размер по произведению AVG_ROW_LENGTH и максимального числа записей MAX_ROWS в таблице. Чем больше таблица, тем большую разрядность должен иметь дескриптор для адресации данных в таблице. Так по умолчанию применяется 32-разрядный дескриптор (4 байта), который позволяет адресовать 4 Гбайт данных. Именно на столько рассчитана по умолчанию таблица MyISAM, если не указаны параметры AVG_ROW_LENGTH и MAX_ROWS. Если по оценке MySQL планируемый размер таблицы превысит 4 Гбайт, будет автоматически выбран дескриптор большей разрядности.

[DEFAULT] CHARACTER SET

Данный параметр позволяет задать кодировку для всех текстовых полей таблицы и может принимать две формы: краткую CHARACTER SET и полную DEFAULT CHARACTER SET. Пример использования данного параметра приведен в листинге 11.23.

Листинг 11.23. Использование параметра CHARACTER SET

```
CREATE TABLE collection (
  coll_koi8r TINYTEXT CHARACTER SET koi8r,
  name TINYTEXT,
  description TINYTEXT
) ENGINE = MyISAM CHARACTER SET cp1251;
```

В листинге 11.23 столбец `coll_koi8r` обрабатывает помещенный текст в кодировке KOI8-R, значения всех остальных столбцов обрабатываются как текст в кодировке Windows-1251.

Параметр `CHARACTER SET` может быть задан в нескольких формах, которые эквивалентны между собой:

```
CHARACTER SET charset
CHARSET = charset
CHARSET charset
```

Здесь `charset` является названием кодировки, например, cp1251, koi8r и т. п. Все три формы можно также применять для определения кодировки столбца.

CHECKSUM

Параметр `CHECKSUM` является флагом, который принимает значение 1 или 0. Если устанавливается значение 1, MySQL рассчитывает контрольную сумму для каждой записи таблицы (листинг 11.24). Это приводит к небольшому снижению скорости обновления таблицы, но значительно повышает вероятность ее восстановления в случае сбоя.

Листинг 11.24. Использование параметра CHECKSUM

```
CREATE TABLE test (
  name TINYTEXT
) ENGINE = MyISAM CHECKSUM = 1;
```

COMMENT

Данный параметр позволяет задать комментарий к таблице, максимальная длина которого не должна превышать 60 символов (листинг 11.25).

Листинг 11.25. Использование параметра COMMENT

```
CREATE TABLE test (
  name TINYTEXT
) ENGINE = MyISAM COMMENT = 'Тестовая таблица';
```

Этот комментарий отображается операторами `SHOW CREATE TABLE` и `SHOW TABLE STATUS`.

DATA DIRECTORY

Параметр `DATA DIRECTORY` используется только для таблиц типа MyISAM и определяет каталог, куда помещается файл с данными (.MYD). Путь должен быть абсолютным (листинг 11.26).

Листинг 11.26. Использование параметра `DATA DIRECTORY`

```
CREATE TABLE test (
  name TINYTEXT
) ENGINE = MyISAM DATA DIRECTORY = 'C:/mysql/data/test';
```

DELAY_KEY_WRITE

Параметр `DELAY_KEY_WRITE` принимает значение либо 1, либо 0. Если установлено значение 1, кэш индекса периодически обновляется для всей таблицы, а не после каждой операции вставки. Данный параметр применяется только для таблиц типа MyISAM и игнорируется для таблиц другого типа.

INDEX DIRECTORY

Параметр `INDEX DIRECTORY` используется только для таблиц типа MyISAM и определяет каталог, куда помещается индексный файл (.MYI). Путь должен быть абсолютным (листинг 11.27).

Листинг 11.27. Использование параметра `INDEX DIRECTORY`

```
CREATE TABLE test (
  name TINYTEXT
) ENGINE = MyISAM INDEX DIRECTORY = 'C:/mysql/data/test';
```

INSERT_METHOD

Данный параметр используется в таблицах типа MERGE для определения правил добавления строк. Подробное описание этого параметра приведено в *разделе "MERGE"* данной главы.

MAX_ROWS

Параметр `MAX_ROWS` определяет максимальное число записей, которое планируется хранить в таблице (листинг 11.28). Использование данного параметра имеет смысл только в таблицах MyISAM.

Листинг 11.28. Использование параметра `MAX_ROWS`

```
CREATE TABLE test (
  name TINYTEXT
) ENGINE = MyISAM MAX_ROWS = 10000000 AVG_ROW_LENGTH = 100;
```

Следует отметить, что параметр MAX_ROWS не ограничивает число записей в таблице. Он лишь служит для подсказки оптимизатору MySQL, как наиболее эффективно хранить данные.

MIN_ROWS

Параметр MIN_ROWS определяет минимальное число записей, которое планируется хранить в таблице. Использование данного параметра имеет смысл только в таблицах MyISAM и HEAP. Точно так же, как и MAX_ROWS, данный параметр не несет ограничительных функций и служит лишь для оценки объема таблицы.

PACK_KEYS

Параметр PACK_KEYS принимает значение либо 1, либо 0. Установка значения 1 для данного параметра включает механизм сжатия текстовых (CHAR и VARCHAR) и числовых индексов. Это приводит к замедлению операции вставки, но увеличивает скорость извлечения данных.

PASSWORD

Данная опция, показанная в листинге 11.29, задает пароль для шифрования файла описания таблицы (с расширением frm).

Листинг 11.29. Использование параметра PASSWORD

```
CREATE TABLE test (
  name TINYTEXT
) ENGINE = MyISAM PASSWORD = 'root';
```

К сожалению, в текущих версиях MySQL данная опция лишь зарезервирована, но не работает.

ROW_FORMAT

Параметр ROW_FORMAT применим только к таблицам типа MyISAM и определяет способ хранения записей (листинг 11.30). Параметр может принимать следующие значения:

- FIXED — в таблицах, не использующих типы данных BLOB и TEXT, все столбцы, имеющие тип VARCHAR, рассматриваются как CHAR.
- DYNAMIC — в таблицах, не использующих типы BLOB и TEXT, все столбцы, имеющие тип CHAR, рассматриваются как VARCHAR;
- DEFAULT — таблица хранится в неупакованном виде;
- COMPRESSED — таблица хранится в упакованном виде.

Листинг 11.30. Использование параметра ROW_FORMAT

```
CREATE TABLE test (
  name VARCHAR(100)
) ENGINE = MyISAM ROW_FORMAT = FIXED;
```

UNION

Параметр UNION определяет список таблиц MyISAM, которые составляют MERGE-таблицу. Синтаксис данного параметра рассмотрен в *разделе "MERGE"* данной главы.

11.12. Изменение типа таблицы

Тип таблицы и ее параметры можно изменить при помощи рассмотренного в *главе 10* оператора ALTER TABLE. В листинге 11.31 приводится пример SQL-оператора ALTER TABLE, изменяющего тип таблицы total на MyISAM и устанавливающего режим подсчета контрольной суммы для записей.

Листинг 11.31. Изменение параметров таблицы

```
ALTER TABLE total TYPE = MYISAM CHECKSUM = 1;
```

ГЛАВА 12

Приведение типов

В данной главе рассматривается приведение типов данных в СУБД MySQL, а также встроенные функции, предназначенные для приведения типов. В большинстве случаев приведение типов осуществляется автоматически (листинг 12.1).

Листинг 12.1. Автоматическое приведение типов

```
mysql> SELECT 1 + '1';
+---------+
| 1 + '1' |
+---------+
|       2 |
+---------+
1 row in set (0.01 sec)
```

Тем не менее отдельные случаи требуют специального приведения типов.

12.1. Ключевое слово *BINARY*

Сравнение текстовых строк в MySQL не зависит от регистра сравниваемых строк. Это означает, что строки "`Hello`" и "`HELLO`", с точки зрения MySQL, являются идентичными (листинг 12.2).

Листинг 12.2. Сравнение текстовых строк

```
mysql> SELECT "Hello" = "HELLO";
+-------------------+
| "Hello" = "HELLO" |
+-------------------+
|                 1 |
+-------------------+
1 row in set (0.02 sec)
```

Ключевое слово BINARY позволяет сравнивать строки как двоичные последовательности. В этом случае строки будут равны только при условии полной эквивалентности.

Листинг 12.3. Двоичное сравнение строк

```
mysql> SELECT BINARY "Hello" = "HELLO";
+--------------------------+
| BINARY "Hello" = "HELLO" |
+--------------------------+
|                        0 |
+--------------------------+
1 row in set (0.01 sec)
```

Использование ключевого слова BINARY допускается как слева от оператора сравнения:

```
SELECT BINARY "Hello" = "HELLO";
```

так и справа:

```
SELECT "Hello" = BINARY "HELLO";
```

Аналогичного результата можно добиться, если в качестве типа столбцов использовать BLOB.

12.2. Функция *CAST*

Встроенная функция CAST() предназначена для преобразования выражения из одного типа в другой. Функция имеет следующий синтаксис:

```
CAST(expr AS type)
```

Результатом работы функции является преобразование выражения expr в тип type. Значением type может быть одно из следующих:

- BINARY (двоичная строка)
- CHAR
- DATE
- DATETIME
- SIGNED [INTEGER]
- TIME
- UNSIGNED [INTEGER]

В листинге 12.4 продемонстрирована работа функции CAST().

Листинг 12.4. Функция CAST()

```
mysql> SELECT 20051201 AS first, CAST(20051201 AS DATE) AS second;
+----------+------------+
| first    | second     |
+----------+------------+
| 20051201 | 2005-12-01 |
+----------+------------+
1 row in set (0.00 sec)
```

Замечание

Между именем функции и открывающей круглой скобкой не должно быть пробела, это связано с тем, что имена функций не являются зарезервированными словами и могут выступать в качестве имени столбца. В такой ситуации MySQL может отличить столбец от функции, только если за именем функции сразу следует открывающая круглая скобка.

Замечание

Конструкция `CAST(str AS BINARY)` аналогична конструкции `BINARY str`.

Функция `CAST()` часто применяется для сортировки полей типа `ENUM` не по численному значению, а по алфавиту. Возвращаясь к учебной базе данных, отсортируем поля таблицы `users` по полю `status` (листинг 12.5).

Листинг 12.5. Сортировка по ENUM-полю status

```
mysql> SELECT surname, status FROM users ORDER BY CAST(status AS CHAR);
+-----------+---------+
| surname   | status  |
+-----------+---------+
| Иванов    | active  |
| Симдянов  | active  |
| Кузнецов  | active  |
| Корнеев   | gold    |
| Нехорошев | lock    |
| Лосев     | passive |
+-----------+---------+
6 rows in set (0.00 sec)
```

Еще один пример использования функции `CAST()` приведен в листинге 12.6.

Листинг 12.6. Преобразование текущего времени в формат DATE

```
mysql> SELECT NOW(), CAST(NOW() AS DATE);
+---------------------+---------------------+
| NOW()               | CAST(NOW() AS DATE) |
+---------------------+---------------------+
| 2005-04-10 02:01:47 | 2005-04-10          |
+---------------------+---------------------+
1 row in set (0.00 sec)
```

В листинге 12.6 демонстрируется приведение текущего времени в полном формате, возвращаемого функцией `NOW()` *(см. главу 14)* к усеченному формату типа `DATE`.

12.3. Функция *CONVERT*

Встроенная функция `CONVERT()` имеет следующий синтаксис:

```
CONVERT(expr, type)
CONVERT(expr USING charset)
```

Первая форма функции `CONVERT()` аналогична по действию функции `CAST()`. Функция `CAST()` является стандартной функцией SQL и поддерживается другими базами данных. Функция `CONVERT()` в первом варианте (без `USING`) является функцией интерфейса ODBC, который получил широкое распространение. Аргументы `expr` и `type` имеют то же значение, что и для функции `CAST()`.

Вторая форма функции `CONVERT()` (с использованием `USING`) предназначена для преобразования текста из одной кодировки в другую (листинг 12.7).

Листинг 12.7. Преобразование кодировки

```
SELECT CONVERT('База данных MySQL' USING koi8r);
+----------------------------------------+
| CONVERT('База данных MySQL' USING koi8r) |
+----------------------------------------+
| аЮГЮ ДЮММШУ MySQL                       |
+----------------------------------------+
1 row in set (0.00 sec)
```

Для быстрого преобразования кодировки строки существует альтернативный синтаксис, пример которого приведен в листинге 12.8.

Листинг 12.8. Альтернативное преобразование кодировки

```
mysql> SELECT _koi8r'База данных MySQL';
+-------------------+
| аЮГЮ ДЮММШУ MySQL |
+-------------------+
| аЮГЮ ДЮММШУ MySQL |
+-------------------+
1 row in set (0.00 sec)
```

Имя кодировки, которое предваряется знаком подчеркивания, помещается перед преобразуемой строкой — результатом является строка в требуемой кодировке.

ГЛАВА 13

Операторы и математические функции

В данной главе будут рассмотрены основные функции для работы с числовыми данными. Здесь же подробно описываются операторы, которые можно использовать при составлении запросов в MySQL.

13.1. Операторы

Под *операторами* подразумеваются конструкции языка, которые производят преобразование данных, например, операция сложения "+", вычитания "-" и т. п. Данные, над которыми совершается операция, называются *операндами*.

13.1.1. Арифметические операторы

К арифметическим операциям относят сложение +, вычитание -, умножение *, деления / и целочисленное деление `DIV`. Если в качестве аргументов используются числа типа `INT`, то результатом является число типа `BIGINT`. В то же время, если аргументы имеют тип `UNSIGNED INT`, то результат также будет иметь тип `UNSIGNED INT`.

> ***Замечание***
>
> Операция целочисленного деления `DIV` введена в СУБД MySQL, начиная с версии 4.1.0.

Применение операции сложения "+" демонстрируется в листинге 13.1.

Листинг 13.1. Сложение

```
mysql> SELECT 3 + 5;
+-------+
| 3 + 5 |
+-------+
|     8 |
+-------+
```

Применять операцию сложения можно не только к целым числам, но и к столбцам. Возвращаясь к учебной базе данных, смоделируем ситуацию привоза, когда на склад поступает продукция от поставщиков. Пусть партия комплектующих включает по 10 штук каждой товарной позиции. Тогда изменить число товара можно при помощи оператора `UPDATE`, представленного в листинге 13.2.

Листинг 13.2. Обновление числа товарных позиций на складе

```
UPDATE products SET count = count + 10;
```

В результате выполнения запроса число товарных позиций для каждой записи будет увеличено на 10. Если нужно изменить число только для одной или нескольких товарных позиций, необходимо применить конструкцию `WHERE`, которая ограничит число затрагиваемых запросом записей.

Важно отметить, что операция сложения числа с `NULL` дает `NULL` (листинг 13.3).

Листинг 13.3. Сложение с `NULL`

```
mysql> SELECT 3 + NULL;
+----------+
| 3 + NULL |
+----------+
|     NULL |
+----------+
```

Такое поведение вполне оправдано. `NULL` обозначает данные, для которых значение неизвестно, поэтому прибавление к такому значению числа приводит опять к неизвестному значению, т. е. к `NULL`.

MySQL автоматически приводит типы данных, так, если в качестве слагаемых будут выступать строки, они будут автоматически приведены к числам. Результатом в этом случае также будет число (листинг 13.4).

Листинг 13.4. Приведение строк к числам при сложении

```
mysql> SELECT '3' + '5';
+-----------+
| '3' + '5' |
+-----------+
|         8 |
+-----------+
```

При этом если строка не может быть приведена к числу, она интерпретируется как 0 (листинг 13.5).

Листинг 13.5. Сложение строк

```
mysql> SELECT 'abc' + 'dfe';
+---------------+
| 'abc' + 'dfe' |
+---------------+
|             0 |
+---------------+
```

Замечание

Для объединения строк в MySQL используется встроенная функция `CONCAT()`, ознакомиться с синтаксисом которой можно в *главе 15*.

Операция вычитания "-", продемонстрированная в листинге 13.6, имеет те же особенности и ограничения, что и операция сложения.

Листинг 13.6. Вычитание

```
mysql> SELECT 8 - 3;
+-------+
| 8 - 3 |
+-------+
|     5 |
+-------+
```

Помимо бинарного оператора "-", который производит вычитание, существует унарный оператор, который меняет знак операнда (листинг 13.7).

Листинг 13.7. Смена знака

```
mysql> SELECT -7;
+----+
| -7 |
+----+
| -7 |
+----+
```

Операции умножения "*" и деления "/" продемонстрированы в листинге 13.8. Пусть требуется изменить цену комплектующих в учебной базе данных в связи с изменением курса доллара. Для этого следует разделить цену в рублях на старый курс (28.5), чтобы получить цену в долларах, и затем умножить на новый (28.0).

Листинг 13.8. Изменение цены в таблице `products`

```
UPDATE products SET price = (price/28.5)*28.0;
```

Как видно из листинга 13.8, точно так же, как и в арифметике, порядок вычисления можно задавать при помощи круглых скобок. В приведенном запросе круглые скобки вполне можно опустить — `price/28.5*28.0` — результат будет тот же: каждое значение из столбца `price` будет разделено на 28.5, а потом умножено на 28.0.

Если результат умножения выходит за пределы типа `BIGINT`, то возвращается 0 (листинг 13.9).

Листинг 13.9. Выход за границы типа `BIGINT` при умножении

```
mysql> SELECT 18014398509481984*18014398509481984;
+-------------------------------------+
| 18014398509481984*18014398509481984 |
+-------------------------------------+
|                                   0 |
+-------------------------------------+
```

В отличие от других языков программирования, деление на ноль не вызывает ошибки синтаксиса и остановки вычислений. В качестве результата возвращается `NULL` (листинг 13.10).

Листинг 13.10. Деление на ноль

```
mysql> SELECT 8/0;
+------+
| 8/0  |
+------+
| NULL |
+------+
```

Кроме обычного деления, начиная с версии 4.1.0, в MySQL введен оператор целочисленного деления `DIV`, работа которого представлена в листинге 13.11.

Листинг 13.11. Целочисленное деление при помощи `DIV`

```
mysql> SELECT 5 DIV 2, 5/2;
+---------+------+
| 5 DIV 2 | 5/2  |
+---------+------+
|       2 | 2.50 |
+---------+------+
```

Как видно из листинга 13.11, результат деления при помощи `DIV` является целочисленным. Дробная часть просто отбрасывается и округления результата не производится. Для того чтобы получить остаток от деления, необходимо воспользоваться оператором "%" (листинг 13.12).

Листинг 13.12. Взятие остатка от деления

```
mysql> SELECT 5 % 2;
+-------+
| 5 % 2 |
+-------+
|     1 |
+-------+
```

Результат запроса равен 1, т. к. без остатка на 2 делится только 4 (5 – 4 = 1). Оператор взятия остатка от деления имеет еще две альтернативные формы написания:

- замена % на `MOD`;
- встроенная функция `MOD()`(листинг 13.13).

Листинг 13.13. Альтернативная запись "%"

```
mysql> SELECT MOD(5,2), 5 MOD 2;
+----------+---------+
| MOD(5,2) | 5 MOD 2 |
+----------+---------+
|        1 |       1 |
+----------+---------+
```

Точно так же, как и в случае обычного деления, при использовании `DIV` и `MOD`, если делитель равен нулю, возвращается `NULL`.

13.1.2. Операторы сравнения

Данная группа операторов интенсивно используется для создания условных запросов с применением инструкции `WHERE`. Результатом операции сравнения может быть 0 (ложь), 1 (истина) и `NULL`. Сравнивать можно как числа, так и строки.

Замечание

При сравнении со значением `NULL` результат всегда равен `NULL`. Исключение составляет оператор "<=>", который специально введен для сравнения `NULL`.

Замечание

Если один из операндов имеет тип `TIMESTAMP` или `DATETIME`, а другой является константой, операнды сравниваются как `TIMESTAMP`-типы.

13.1.2.1. Равенство =

Результат равенства равен 1, если операнды равны, и равен 0 в противном случае. В листинге 13.14 демонстрируется сравнение строк и чисел.

Листинг 13.14. Сравнение строк и чисел

```
mysql> SELECT 1=1, 1=2, '1'=1, 'ab'='ab', 'ab' = 'AB', 'ab' = 0;
+-----+-----+-------+-----------+-------------+----------+
| 1=1 | 1=2 | '1'=1 | 'ab'='ab' | 'ab' = 'AB' | 'ab' = 0 |
+-----+-----+-------+-----------+-------------+----------+
|   1 |   0 |     1 |         1 |           1 |        1 |
+-----+-----+-------+-----------+-------------+----------+
```

Как видно из листинга 13.14, при сравнении строк с числами последние автоматически преобразуются в числа, если же такое преобразование выполнить невозможно, то строка имеет значение 0, поэтому сравнение `'ab' = 0` дает истину — 1. Строка `'14L'` будет преобразована в `'14'`, а `'L14'` в `'0'`.

Сравнение строк происходит без учета регистра. Если необходимо, чтобы регистр учитывался, строки следует сравнивать как бинарные (сравнение производится не посимвольно, а побайтно). Для этого строка предваряется ключевым словом `BINARY` (листинг 13.15).

Листинг 13.15. Бинарное сравнение строк

```
mysql> SELECT BINARY 'abc'='ABC', BINARY 'abc'='abc';
+--------------------+--------------------+
| BINARY 'abc'='ABC' | BINARY 'abc'='abc' |
+--------------------+--------------------+
|                  0 |                  1 |
+--------------------+--------------------+
```

Сравнивать можно не только целочисленные значения и строки, но также числа с плавающей точкой (листинг 13.16).

Листинг 13.16. Сравнение чисел с плавающей точкой

```
mysql> SELECT 0.01 = 0, 0.01 = .01;
+----------+------------+
| 0.01 = 0 | 0.01 = .01 |
+----------+------------+
|        0 |          1 |
+----------+------------+
```

Как видно из листинга 13.16, при написании чисел с плавающей точкой ведущий ноль можно не указывать.

13.1.2.2. Оператор <=>

Оператор эквивалентности "<=>" аналогичен обычному равенству, но возвращает он только два значения: 1 (истина) и 0 (ложь). Результата `NULL` он не возвращает. Пример, демонстрирующий работу данного оператора, приведен в листинге 13.17.

Листинг 13.17. Сравнение с использованием оператора эквивалентности "<=>"

```
mysql> SELECT 1<=>1, 1<=>2, NULL<=>1, NULL<=>NULL;
+-------+-------+----------+-------------+
| 1<=>1 | 1<=>2 | NULL<=>1 | NULL<=>NULL |
+-------+-------+----------+-------------+
|     1 |     0 |        0 |           1 |
+-------+-------+----------+-------------+
```

Как видно из листинга 13.17, сравнение `NULL` с `NULL` дает 1 (истина), в отличие от оператора равенства "=", который бы дал в этом случае `NULL`.

13.1.2.3. Оператор <>

Оператор неравенства "<>" равен 1 (истина), если операнды не равны, и 0 (ложь) в противном случае. Листинг 13.18 демонстрирует работу данного оператора.

Замечание

Помимо записи "<>", данный оператор может принимать форму "!=".

Листинг 13.18. Использование оператора неравенства "<>"

```
mysql> SELECT 0.01<>0.01, 'ab'!='AB', BINARY 'ab'<>'AB', 'ab'!='cd';
+------------+------------+------------------+------------+
| 0.01<>0.01 | 'ab'!='AB' | BINARY 'ab'<>'AB' | 'ab'!='cd' |
+------------+------------+------------------+------------+
|          0 |          0 |                1 |          1 |
+------------+------------+------------------+------------+
```

Данный оператор возвращает значения, противоположенные оператору равенства "=".

13.1.2.4. Оператор <

Оператор "меньше" "<" возвращает 1 (истина), если левый операнд меньше правого (листинг 13.19). В противном случае возвращается 0 (ложь).

Листинг 13.19. Использования оператора "<"

```
mysql> SELECT 3<10, 5<1, 'big number'<1;
+------+-----+----------------+
| 3<10 | 5<1 | 'big number'<1 |
+------+-----+----------------+
|    1 |   0 |              1 |
+------+-----+----------------+
```

Как видно из листинга 13.19, при сравнении числа со строкой строка автоматически приводится к числу, т. к. в приведенном примере извлечь числовое значение из строки нельзя, вместо нее подставляется значение 0, что приводит к 1 (истина).

Возвращаясь к учебной базе данных, выясним названия товарных позиций, количество которых на складе меньше 5 экземпляров (листинг 13.20).

Листинг 13.20. Выборка из таблицы `products` по цене

```
mysql> SELECT name, count FROM products WHERE count < 5;
+-----------------------------+-------+
| name                        | count |
+-----------------------------+-------+
| Celeron 2.0GHz              |     2 |
| Celeron 2.4GHz              |     4 |
| Celeron D 320 2.4GHz        |     1 |
| Intel Pentium 4 3.0GHz      |     1 |
| Gigabyte GA-8I848P-RS       |     4 |
| Gigabyte GA-8IG1000         |     2 |
| Asustek P4C800-E Delux      |     4 |
| ASUSTEK A9600XT/TD          |     2 |
| SAPPHIRE 256MB RADEON 9550  |     3 |
| Maxtor 6B200P0              |     4 |
| Seagate Barracuda ST3160023A |     3 |
+-----------------------------+-------+
```

13.1.2.5. Оператор <=

Оператор "меньше или равно" "<=" возвращает 1 (истина), если левый операнд меньше правого или оба оператора равны (листинг 13.21), иначе возвращается 0 (ложь).

Листинг 13.21. Использование оператора "<="

```
mysql> SELECT 3.4 <= '3.4', 10 <= 1;
+--------------+---------+
| 3.4 <= '3.4' | 10 <= 1 |
+--------------+---------+
|            1 |       0 |
+--------------+---------+
```

13.1.2.6. Оператор >

Оператор "больше" ">" возвращает 1 (истина), если левый операнд больше, чем правый (листинг 13.22), иначе возвращается ложь.

Листинг 13.22. Использование оператора ">"

```
mysql> SELECT 65 > 34, 65 > 65;
+---------+---------+
| 65 > 34 | 65 > 65 |
+---------+---------+
|       1 |       0 |
+---------+---------+
```

Возвращаясь к учебной базе данных, выясним названия товарных позиций, цена которых превышает 2000 рублей (листинг 13.23).

Листинг 13.23. Выборка из таблицы `products` по цене

```
mysql> SELECT name, price FROM products WHERE price > 2000;
+------------------------------+---------+
| name                         | price   |
+------------------------------+---------+
| Celeron 2.4GHz               | 2109.00 |
| Celeron D 325 2.53GHz        | 2747.00 |
| Intel Pentium 4 3.2GHz       | 7259.00 |
| Intel Pentium 4 3.0GHz       | 6147.00 |
| Intel Pentium 4 3.0GHz       | 5673.00 |
| Gigabyte GA-8IG1000          | 2420.00 |
| Gigabyte GA-8IPE1000G        | 2289.00 |
| Asustek P4C800-E Delux       | 5395.00 |
| Asustek P4P800-VM\L i865G    | 2518.00 |
| Epox EP-4PDA3I               | 2289.00 |
| ASUSTEK A9600XT/TD           | 5156.00 |
| SAPPHIRE 256MB RADEON 9550   | 2730.00 |
| GIGABYTE AGP GV-N59X128D     | 5886.00 |
| Maxtor 6Y120P0               | 2456.00 |
| Maxtor 6B200P0               | 3589.00 |
| Samsung SP0812C              | 2093.00 |
| Seagate Barracuda ST3160023A | 3139.00 |
| Seagate ST3120026A           | 2468.00 |
+------------------------------+---------+
```

13.1.2.7. Оператор >=

Оператор "больше или равно" ">=" возвращает 1 (истина), если левый операнд больше правого или они равны, и 0 в противном случае (листинг 13.24).

Листинг 13.24. Использование оператора" >="

```
mysql> SELECT 65 >= 34, 65 >= 65;
+----------+----------+
| 65 >= 34 | 65 >= 65 |
+----------+----------+
|        1 |        1 |
+----------+----------+
```

13.1.2.8. Конструкция *IS NULL*

Данная конструкция позволяет определить, равняется ли проверяемое значение `NULL` или нет (листинг 13.25). Если значение равно `NULL`, оператор возвращает 1 (истина), в противном случае возвращается 0 (ложь).

Листинг 13.25. Использование конструкции `IS NULL`

```
mysql> SELECT NULL IS NULL, 0 IS NULL;
+--------------+-----------+
| NULL IS NULL | 0 IS NULL |
+--------------+-----------+
|            1 |         0 |
+--------------+-----------+
```

Рассмотрим запрос к учебной базе данных. В таблице `users` необязательные поля, такие как телефон, e-mail и URL пользователя, могут принимать значение `NULL`, если отсутствует достоверная информация о наличии или отсутствии этих данных о пользователе. Получим имена, фамилии и отчества пользователей, у которых не указан телефон (листинг 13.26).

Листинг 13.26. Запрос к таблице `users` с использованием конструкции `IS NULL`

```
mysql> SELECT surname, name, patronymic FROM users WHERE phone IS NULL;
+-----------+----------+------------+
| surname   | name     | patronymic |
+-----------+----------+------------+
| Кузнецов  | Максим   | Валерьевич |
| Нехорошев | Анатолий | Юрьевич    |
+-----------+----------+------------+
```

Замечание

Конструкция `IS NULL` также возвращает 1 (истина) для нулевых полей типа `DATETIME` — '0000-00-00 00:00:00'. Это необходимо для ODBC-приложений, т. к. этот интерфейс не поддерживает нулевое время, вместо него используется `NULL`.

Существует альтернативная форма записи конструкции `IS NULL` — `ISNULL()`, применение которой демонстрируется в листинге 13.27.

Листинг 13.27. Использование функции `ISNULL`

```
mysql> SELECT ISNULL(NULL), ISNULL(0);
+--------------+-----------+
| ISNULL(NULL) | ISNULL(0) |
+--------------+-----------+
|            1 |         0 |
+--------------+-----------+
```

13.1.2.9. Конструкция *IS NOT NULL*

Данная конструкция является противопоставлением оператора `IS NULL`. Если значение равно `NULL`, оператор возвращает 0 (ложь), в противном случае возвращается 1 (истина).

Листинг 13.28. Использование конструкции `IS NOT NULL`

```
mysql> SELECT NULL IS NOT NULL, 0 IS NOT NULL;
+------------------+---------------+
| NULL IS NOT NULL | 0 IS NOT NULL |
+------------------+---------------+
|                0 |             1 |
+------------------+---------------+
```

Выберем из таблицы `users` фамилии, имена и отчества всех пользователей, в данных которых имеется URL (листинг 13.29).

Листинг 13.29. Запрос к таблице `users` с использованием конструкции `IS NOT NULL`

```
mysql> SELECT surname, name, patronymic FROM users WHERE url IS NOT NULL;
+----------+--------+--------------+
| surname  | name   | patronymic   |
+----------+--------+--------------+
| Симдянов | Игорь  | Вячеславович |
| Кузнецов | Максим | Валерьевич   |
+----------+--------+--------------+
```

13.1.2.10. Конструкция *BETWEEN min AND max*

Конструкция `BETWEEN` возвращает 1 (истина), если проверяемое значение расположено между `min` и `max`, в противном случае возвращается 0 (ложь). В листинге 13.30 представлен запрос, извлекающий товарные позиции, цена которых располагается между 1000 и 2000 рублей.

Листинг 13.30. Использование конструкции `BETWEEN`

```
mysql> SELECT name, price FROM products
mysql> WHERE price BETWEEN 1000 AND 2000;
+------------------------------+---------+
| name                         | price   |
+------------------------------+---------+
| Celeron 1.8                  | 1595.00 |
| Celeron 2.0GHz               | 1969.00 |
| Celeron D 320 2.4GHz         | 1962.00 |
| Celeron D 315 2.26GHz        | 1880.00 |
| Gigabyte GA-8I848P-RS        | 1896.00 |
| ASUSTEK V9520X               | 1602.00 |
| DDR-400 256MB Kingston       | 1085.00 |
| DDR-400 256MB Hynix Original | 1179.00 |
| DDR-400 512MB Kingston       | 1932.00 |
| DDR-400 512MB PQI            | 1690.00 |
| DDR-400 512MB Hynix          | 1717.00 |
+------------------------------+---------+
```

Форма записи `price BERWEEN 1000 AND 2000` эквивалентна `1000 <= price AND price <= 2000`. Если типы сравниваемого значения и границ интервала сравнения не совпадают, то происходит приведение типов.

13.1.2.11. Конструкция *NOT BETWEEN min AND max*

Конструкция `NOT BETWEEN` возвращает 0 (ложь), если проверяемое значение расположено между `min` и `max`, в противном случае возвращается 1 (истина). Выясним, какие товарные позиции не вошли в ценовой интервал между 1000 и 2000 рублей, т. е. решим задачу, противоположную представленной в листинге 13.30.

Листинг 13.31. Использование конструкции `NOT BETWEEN`

```
mysql> SELECT name, price FROM products
mysql> WHERE price NOT BETWEEN 1000 AND 2000;
+-----------------------------+---------+
| name                        | price   |
+-----------------------------+---------+
| Celeron 2.4GHz              | 2109.00 |
| Celeron D 325 2.53GHz       | 2747.00 |
| Intel Pentium 4 3.2GHz      | 7259.00 |
| Intel Pentium 4 3.0GHz      | 6147.00 |
| Intel Pentium 4 3.0GHz      | 5673.00 |
| Gigabyte GA-8IG1000         | 2420.00 |
| Gigabyte GA-8IPE1000G       | 2289.00 |
| Asustek P4C800-E Delux      | 5395.00 |
| Asustek P4P800-VM\L i865G   | 2518.00 |
| Epox EP-4PDA3I              | 2289.00 |
| ASUSTEK A9600XT/TD          | 5156.00 |
| SAPPHIRE 256MB RADEON 9550  | 2730.00 |
| GIGABYTE AGP GV-N59X128D    | 5886.00 |
| Maxtor 6Y120P0              | 2456.00 |
| Maxtor 6B200P0              | 3589.00 |
| Samsung SP0812C             | 2093.00 |
| Seagate Barracuda ST3160023A | 3139.00 |
| Seagate ST3120026A          | 2468.00 |
| DDR-400 256MB PQI           |  899.00 |
+-----------------------------+---------+
```

13.1.2.12. Функция *COALESCE*

Функция `COALESCE()` возвращает первый элемент из списка, который не равен `NULL`. Применение данного оператора демонстрируется в листинге 13.32.

Листинг 13.32. Использование функции COALESCE()

```
mysql> SELECT COALESCE(1 + NULL, 4 - NULL, 10, NULL, 2);
+-------------------------------------------+
| COALESCE(1 + NULL, 4 - NULL, 10, NULL, 2) |
+-------------------------------------------+
|                                        10 |
+-------------------------------------------+
```

Как видно из листинга 13.32, функция возвращает третий элемент списка, равный 10, все предыдущие элементы равны NULL, т. к. сложение и вычитание с NULL всегда дает NULL.

Замечание

Списком называется набор однородных значений, следующих через запятую и помещенных в круглые скобки. Такие списки можно получить в результате выполнения вложенных запросов, которые рассмотрены в *главе 23*.

13.1.2.13. Функция *GREATEST*

Данная функция возвращает максимальное значение из списка (листинг 13.33).

Листинг 13.33. Использование функции GREATEST()

```
mysql> SELECT GREATEST(4,3,-10);
+-------------------+
| GREATEST(4,3,-10) |
+-------------------+
|                 4 |
+-------------------+
mysql> SELECT GREATEST(34.0,3.0,5.0,767.0);
+-----------------------------+
| GREATEST(34.0,3.0,5.0,767.0) |
+-----------------------------+
|                       767.0 |
+-----------------------------+
mysql> SELECT GREATEST('B','A','C');
+-----------------------+
| GREATEST('B','A','C') |
+-----------------------+
| C                     |
+-----------------------+
```

13.1.2.14. Функция *LEAST*

Функция LEAST() является противоположной GREATEST() и возвращает минимальное значение из списка.

Листинг 13.34. Использование функции `LEAST()`

```
mysql> SELECT LEAST(-10,4,3);
+----------------+
| LEAST(-10,4,3) |
+----------------+
|            -10 |
+----------------+
```

13.1.2.15. Конструкция *IN*

Конструкция `IN` возвращает 1 (истина), если проверяемое значение входит в качестве элемента в список, в противном случае возвращается 0 (ложь). Использование конструкции `IN` демонстрируется в листинге 13.35.

Листинг 13.35. Использование конструкции `IN`

```
mysql> SELECT 2 IN (0,3,5,'wefwf');
+----------------------+
| 2 IN (0,3,5,'wefwf') |
+----------------------+
|                    0 |
+----------------------+
mysql> SELECT 'wefwf' IN (0,3,5,'wefwf');
+----------------------------+
| 'wefwf' IN (0,3,5,'wefwf') |
+----------------------------+
|                          1 |
+----------------------------+
```

Если проверяемое значение равно `NULL`, результат всегда равен `NULL` (листинг 13.36).

Листинг 13.36. `NULL` в качестве проверяемого значения

```
mysql> SELECT NULL IN (0,3,5,'wefwf');
+-------------------------+
| NULL IN (0,3,5,'wefwf') |
+-------------------------+
|                    NULL |
+-------------------------+
```

Для совместимости со стандартом SQL, в MySQL, начиная с версии 4.1.0, конструкция `IN` также возвращает `NULL`, если один из элементов списка равен `NULL` (листинг 13.37).

Листинг 13.37. NULL в качестве элемента списка

```
mysql> SELECT 2 IN (0,NULL,5,'wefwf');
+-------------------------+
| 2 IN (0,NULL,5,'wefwf') |
+-------------------------+
|                    NULL |
+-------------------------+
```

Важно отметить, что в предыдущих версиях MySQL в этом случае возвращался либо 0 (ложь), либо 1 (истина).

13.1.2.16. Конструкция *NOT IN*

Конструкция `NOT IN` является противоположной оператору `IN` и возвращает 1 (истина), если проверяемое значение не входит в список, и 0 (ложь), если оно присутствует в списке.

Листинг 13.38. Использование конструкции NOT IN

```
mysql> SELECT 2 NOT IN (0,3,5,'wefwf');
+--------------------------+
| 2 NOT IN (0,3,5,'wefwf') |
+--------------------------+
|                        1 |
+--------------------------+
```

Помимо представленного в листинге 13.38 синтаксиса конструкции `NOT IN`, существует альтернативная форма записи этого оператора, которая представлена в листинге 13.39.

Листинг 13.39. Альтернативная форма записи конструкции NOT IN

```
mysql> SELECT NOT 2 IN (0,3,5,'wefwf');
+---------------------------+
| NOT 2 IN (0,3,5,'wefwf') |
+---------------------------+
|                         1 |
+---------------------------+
```

13.1.2.17. Функция *INTERVAL*

Функция `INTERVAL(N,N1,N2,N3,...)` возвращает 0, если `N < N1`, и 1, если `N < N2`, и т. д. Все аргументы трактуются как целые числа (листинг 13.40).

Листинг 13.40. Использование функции `INTERVAL()`

```
mysql> SELECT INTERVAL(23, 1, 15, 17, 30, 44, 200);
+-------------------------------------+
| INTERVAL(23, 1, 15, 17, 30, 44, 200) |
+-------------------------------------+
|                                   3 |
+-------------------------------------+
mysql> SELECT INTERVAL(10, 1, 10, 100, 1000);
+--------------------------------+
| INTERVAL(10, 1, 10, 100, 1000) |
+--------------------------------+
|                              2 |
+--------------------------------+
mysql> SELECT INTERVAL(22, 23, 30, 44, 200);
+------------------------------+
| INTERVAL(22, 23, 30, 44, 200) |
+------------------------------+
|                            0 |
+------------------------------+
```

13.1.3. Логические операторы

В SQL логические операторы проверяют истинность или ложность выражений. Для обозначения истинного выражения используется любое не нулевое число, чаще 1 (истина), а для ложного — 0 (ложь). Помимо указанных двух значений, логические операторы могут возвращать `NULL`.

Замечание

Логические операторы называют также булевыми в честь математика Джорджа Буля, создателя булевой алгебры.

Замечание

Логические операторы применяют к числовым данным. Если в качестве операнда выступает строка, то она преобразуется к числу согласно правилам, описанным ранее.

13.1.3.1. Оператор *NOT*

Это оператор логического отрицания. Если операнд, на который он действует, истинен (например, 1), он возвращает 0 (ложь), если операнд ложен (0), он возвращает 1 (истина).

Замечание

Помимо `NOT` данный оператор может быть записан как "!".

Листинг 13.41. Использование оператора NOT

```
mysql> SELECT NOT 10, NOT 0, NOT NULL, !(1 + 1), ! 1 + 1;
+--------+-------+----------+----------+---------+
| NOT 10 | NOT 0 | NOT NULL | !(1 + 1) | ! 1 + 1 |
+--------+-------+----------+----------+---------+
|      0 |     1 |     NULL |        0 |       1 |
+--------+-------+----------+----------+---------+
```

Как видно из листинга 13.41, применение оператора NOT к NULL возвращает NULL. Последний пример на самом деле сводится к (!1) + 1 = 0 + 1, т. е. сначала выполняется оператор отрицания "!", а лишь затем сложения. В таком случае говорят, что приоритет оператора "!" выше, чем оператора "+". Уровни приоритетов приводятся в *разделе 13.1.5*.

13.1.3.2. Оператор *OR*

Это оператор логического ИЛИ, который сравнивает два операнда и возвращает 1 (истина), если хотя бы один из операндов истинен (не 0 и не NULL), и 0 (ложь) в противном случае. Пример использования оператора OR приведен в листинге 13.42.

Замечание

Помимо OR данный оператор может быть записан как "||".

Листинг 13.42. Использование оператора OR

```
mysql> SELECT 1 OR 1, 1 OR 0, 0 OR 0, 0 || NULL, 1 || NULL;
+--------+--------+--------+-----------+-----------+
| 1 OR 1 | 1 OR 0 | 0 OR 0 | 0 || NULL | 1 || NULL |
+--------+--------+--------+-----------+-----------+
|      1 |      1 |      0 |      NULL |         1 |
+--------+--------+--------+-----------+-----------+
```

13.1.3.3. Оператор *AND*

Это оператор логического И, который сравнивает два операнда и возвращает 1 (истина), если оба операнда истины (не 0 и не NULL), и 0 (ложь) в противном случае. Пример использования оператора AND приведен в листинге 13.43.

Замечание

Помимо AND данный оператор может быть записан как "&&".

Листинг 13.43. Использование оператора AND

```
mysql> SELECT 1 AND 1, 1 AND 0, 1 AND NULL, 0 && NULL;
+---------+---------+------------+-----------+
| 1 AND 1 | 1 AND 0 | 1 AND NULL | 0 && NULL |
+---------+---------+------------+-----------+
|       1 |       0 |       NULL |         0 |
+---------+---------+------------+-----------+
```

13.1.3.4. Оператор *XOR*

Это оператор исключающего логического ИЛИ. Выражение `a XOR b` эквивалентно `(a AND (NOT b)) OR ((NOT a) and b)`. Пример использования оператора `XOR` приведен в листинге 13.44.

Листинг 13.44. Использование оператора XOR

```
mysql> SELECT 1 XOR 1, 0 XOR 0, 1 XOR 0, 0 XOR 1, 1 XOR NULL, 0 XOR NULL;
+---------+---------+---------+---------+------------+------------+
| 1 XOR 1 | 0 XOR 0 | 1 XOR 0 | 0 XOR 1 | 1 XOR NULL | 0 XOR NULL |
+---------+---------+---------+---------+------------+------------+
|       0 |       0 |       1 |       1 |       NULL |       NULL |
+---------+---------+---------+---------+------------+------------+
```

13.1.4. Битовые операторы

Битовые операции позволяют манипулировать отдельными битами в двоичной последовательности. Операции производятся над значениями типа `BIGINT` (64-разрядные целые). Если в качестве одного из операндов выступает `NULL`, результат также равен `NULL`. Использование битовых последовательностей позволяет значительно уменьшить объем таблиц при хранении наборов логических флагов, состояний и т. п.

13.1.4.1. Оператор &

Данный оператор соответствует побитовому И (пересечение). При использовании битовых операторов следует помнить, что все операции выполняются над числами в двоичном представлении. Пусть имеется два числа 6 и 10, применение к ним оператора "&" дает 2 (листинг 13.45).

Листинг 13.45. Использование оператора "&"

```
mysql> SELECT 6 & 10;
+--------+
| 6 & 10 |
+--------+
|      2 |
+--------+
```

Этот, казалось бы, парадоксальный результат можно легко понять, если перевести цифры 6 и 10 в двоичное представление, в котором они равны 0110 и 1010 соответственно. Сложение разрядов при использовании оператора "&" происходит согласно правилам, представленным в табл. 13.1. Учитывая их, можно легко получить результат, который равен 0010 (в десятичной системе — 2).

```
0110 (6)
1010 (10)
0010 (2)
```

Таблица 13.1. *Правила сложения разрядов при использовании оператора "&"*

Первый операнд	1	1	0	0
Второй операнд	1	0	1	0
Результат	1	0	0	0

В листинге 13.46 приведен более сложный пример, использование оператора "&" для чисел 113 и 45.

Листинг 13.46. Побитовое сложение "&"

```
mysql> SELECT 113 & 45;
+----------+
| 113 & 45 |
+----------+
|       33 |
+----------+
```

Для наглядности представим складываемые числа в двоичном виде:

```
1110001 (113)
0101101 (45)
0100001 (33)
```

13.1.4.2. Оператор |

Данный оператор соответствует побитовому ИЛИ (пересечение). В табл. 13.2 приведены правила сложения разрядов при использовании оператора "|".

Таблица 13.2. *Правила сложения разрядов при использовании оператора "|"*

Первый операнд	1	1	0	0
Второй операнд	1	0	1	0
Результат	1	1	1	0

Применяя операцию "|" к числам, рассмотренным в предыдущем разделе, получим следующий результат, приведенный в листинге 13.47.

Листинг 13.47. Использование оператора "|"

```
mysql> SELECT 6 | 10, 113 | 45;
+--------+----------+
| 6 | 10 | 113 | 45 |
+--------+----------+
|     14 |      125 |
+--------+----------+
```

Дадим двоичное представление чисел, чтобы действие оператора было более понятно:

```
0110 (6)
1010 (10)
1110 (14)
```

Для второй пары чисел:

```
1110001 (113)
0101101 (45)
1111101 (125)
```

13.1.4.3. Оператор ^

Данный оператор соответствует побитовому исключающему ИЛИ (XOR). В табл. 13.3 приведены правила сложения разрядов при использовании оператора "^".

Таблица 13.3. *Правила сложения разрядов при использовании оператора "^"*

Первый операнд	1	1	0	0
Второй операнд	1	0	1	0
Результат	0	1	1	0

В листинге 13.48 приведен пример использования данного оператора применительно к цифрам 6, 10 и 113, 45.

Листинг 13.48. Использование оператора "^"

```
mysql> SELECT 6 ^ 10, 113 ^ 45;
+--------+----------+
| 6 ^ 10 | 113 ^ 45 |
+--------+----------+
|     12 |       92 |
+--------+----------+
```

Двоичное представление первой пары чисел:

```
0110 (6)
1010 (10)
1100 (12)
```

Для второй пары чисел:

```
1110001 (113)
0101101 (45)
1011100 (92)
```

13.1.4.4. Оператор ~

Данный оператор является побитовой инверсией и заменяет все 0 на 1 и наоборот. В листинге 13.49 демонстрируется использование данного оператора.

Листинг 13.49. Использование оператора "~"

```
mysql> SELECT ~45, ~92;
+----------------------+----------------------+
| ~45                  | ~92                  |
+----------------------+----------------------+
| 18446744073709551570 | 18446744073709551523 |
+----------------------+----------------------+
```

Такие огромные числа, получившиеся в результате, являются следствием того, что все побитовые операции производятся над значениями типа `BIGINT` (64-разрядные целые). В двоичных числах ведущие нули обычно не указывают, но т. к. операция "~" производит их инвертирование — они заменяются единицами, приводя к очень большим величинам. Так, операция `~45` в двоичном представлении выглядит следующим образом:

```
0000000000000000000000000000000000000000000000000000000000101101 (45)
1111111111111111111111111111111111111111111111111111111111010010
```

Для ~92:

```
0000000000000000000000000000000000000000000000000000000001011100 (92)
1111111111111111111111111111111111111111111111111111111110100011
```

13.1.4.5. Оператор <<

Данный оператор сдвигает влево все биты первого операнда на число позиций, указанных во втором операнде. Сдвиг на отрицательное значение приводит к нулевому значению.

Листинг 13.50. Использование оператора "<<"

```
mysql> SELECT 6 << 1, 6 << 2, 6 << 10, 6 << -1, 6 << 64;
+--------+--------+---------+---------+---------+
| 6 << 1 | 6 << 2 | 6 << 10 | 6 << -1 | 6 << 64 |
+--------+--------+---------+---------+---------+
|     12 |     24 |    6144 |       0 |       0 |
+--------+--------+---------+---------+---------+
```

В двоичном представлении эти операции выглядят следующим образом (6 << 1):

```
0110 (6)
1100 (12)
```

(6 << 2):

```
00110 (6)
11000 (24)
```

(6 << 10):

```
0000000000110 (6)
1100000000000 (6144)
```

Как видно из листинга 13.50, к нулевому результату приводит не только сдвиг на отрицательное число позиций, но и сдвиг на 64 позиции, т. к. это приводит к тому, что все 1 уходят за границу 64-битного числа и все разряды принимают значение 0.

```
0000000000000000000000000000000000000000000000000000000000000000 (0)
```

13.1.4.6. Оператор >>

Данный оператор сдвигает вправо все биты первого операнда на число позиций, указанных во втором операнде. Сдвиг на отрицательное значение приводит к нулевому значению.

Листинг 13.51. Использование оператора ">>"

```
mysql> SELECT 6144 >> 10, 45 >> 2, 45 > 2;
+------------+---------+--------+
| 6144 >> 10 | 45 >> 2 | 45 > 2 |
+------------+---------+--------+
|          6 |      11 |      1 |
+------------+---------+--------+
```

В двоичном представлении эти операции выглядят следующим образом (6144 >> 10):

```
1100000000000 (6144)
0000000000110 (6)
```

(45 >> 2):

```
101101 (45)
001011 (6)
```

Как видно из листинга 13.51, при использовании операторов сдвига очень легко ошибиться, указав только один знак ">". В этом случае, как было описано ранее, MySQL сравнит два числа и вернет либо 1 (истина), либо 0 (ложь). За такими опечатками следует следить очень внимательно, т. к. MySQL не генерирует предупреждений, в то же время результат будет отличаться от ожидаемого.

13.1.4.7. Функция *BIT_COUNT*

Функция `BIT_COUNT(N)` возвращает число единиц в числе `N`.

Листинг 13.52. Использование функции `BIT_COUNT()`

```
mysql> SELECT BIT_COUNT(4), BIT_COUNT(12), BIT_COUNT(478);
+--------------+---------------+----------------+
| BIT_COUNT(4) | BIT_COUNT(12) | BIT_COUNT(478) |
+--------------+---------------+----------------+
|            1 |             2 |              7 |
+--------------+---------------+----------------+
```

Пример использования функции `BIT_COUNT` представлен в листинге 13.52.

Для того чтобы лучше понять логику данного оператора, представим числа 4, 12 и 478 в двоичной форме:

```
000000100 (4)
000001100 (12)
111011110 (478)
```

13.1.5. Приоритет операторов

Рассмотренные операторы обладают различными приоритетами, что демонстрируется в листинге 13.53. Оператор отрицания "!" обладает большим приоритетом, чем сложения "+", поэтому, если нужно сначала сложить числа, а лишь затем применить к результату оператор "!", необходимо прибегать к круглым скобкам.

Листинг 13.53. Приоритет операторов

```
mysql> SELECT ! (1 + 1), ! 1 + 1;
+-----------+---------+
| ! (1 + 1) | ! 1 + 1 |
+-----------+---------+
|         0 |       1 |
+-----------+---------+
```

Уровни приоритетов отображает список, представленный ниже — сверху вниз уровень приоритета уменьшается. Записанные на одной строке операторы обладают одним уровнем приоритета и выполняются в выражении слева направо.

```
BINARY, COLLATE
!, NOT
- (унарный минус), ~
^
*, /, DIV, %, MOD
-, +
<<, >>
&
|
=, <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN
BETWEEN, CASE, WHEN, THEN, ELSE
NOT
&&, AND
||, OR, XOR
:=
```

Часть операторов из списка не была рассмотрена в данной главе. Операторы `LIKE`, `REGEXP` рассматриваются в *главе 17*, а операторы `CASE`, `WHEN`, `IF`, `THEN`, `ELSE` — в *главе 20*. Оператор ":=" для работы с пользовательскими переменными рассматривается в *главе 21*.

Замечание

Уровень приоритетов отдельных операторов можно не запоминать, т. к. в спорных ситуациях всегда можно указать порядок вычислений при помощи круглых скобок.

В списке приоритетов операций, представленном выше, есть несколько особенностей. Унарный минус (применяемый для смены знака числа –7, – (1 + 5)) имеет приоритет на три порядка выше, чем бинарный (применяемый для вычитания 5 – 8).

13.2. Математические функции

Все числовые функции возвращают `NULL`, если при их работе возникает ошибка, например, если аргументы выходят за границы допустимых значений или в качестве аргументов передается `NULL`.

MySQL, как и любая другая база данных, обладает большим числом встроенных функций, которые будут рассматриваться на протяжении следующих нескольких глав. Так же, как и в любом другом языке программирования, функции характеризуются именем и аргументами, которые перечисляются через запятую за именем в круглых скобках. Если аргументы у функции отсутствуют, круглые скобки все равно следует указывать, например, `NOW()`. Отличительной чертой MySQL является то, что при использовании функций пробелы между именем функции и круглыми скобками недопустимы, т. е. написание `NOW()` правильное, а `NOW ()` уже нет. Результат функции подставляется в место вызова функции.

13.2.1. Функция *ABS*

Функция `ABS(X)` возвращает абсолютное значение аргумента `X` (листинг 13.54).

Листинг 13.54. Использование функции `ABS()`

```
mysql> SELECT ABS(-4.3), ABS(4.3), ABS(NULL);
+-----------+----------+-----------+
| ABS(-4.3) | ABS(4.3) | ABS(NULL) |
+-----------+----------+-----------+
|       4.3 |      4.3 |      NULL |
+-----------+----------+-----------+
```

13.2.2. Функция *ACOS*

Функция `ACOS(X)` возвращает арккосинус числа `X` или `NULL`, если значение `X` не находится в диапазоне от –1 до 1 (листинг 13.55).

Листинг 13.55. Использование функции `ACOS()`

```
mysql> SELECT ACOS(1), ACOS(0), ACOS(1.001), ACOS(NULL);
+---------+-----------------+-------------+------------+
| ACOS(1) | ACOS(0)         | ACOS(1.001) | ACOS(NULL) |
+---------+-----------------+-------------+------------+
|       0 | 1.5707963267949 |        NULL |       NULL |
+---------+-----------------+-------------+------------+
```

13.2.3. Функция *ASIN*

Функция `ASIN(X)` возвращает арксинус числа `X` или `NULL`, если значение `X` не находится в диапазоне от –1 до 1 (листинг 13.56).

Листинг 13.56. Использование функции `ASIN()`

```
mysql> SELECT ASIN(1), ASIN(0), ASIN(1.001), ASIN(NULL);
+-----------------+---------+-------------+------------+
| ASIN(1)         | ASIN(0) | ASIN(1.001) | ASIN(NULL) |
+-----------------+---------+-------------+------------+
| 1.5707963267949 |       0 |        NULL |       NULL |
+-----------------+---------+-------------+------------+
```

13.2.4. Функция *ATAN*

Функция `ATAN(X)` возвращает арктангенс числа `X`.

Листинг 13.57. Использование функции `ATAN(X)`

```
mysql> SELECT ATAN(0), ATAN(1), ATAN('единица'), ATAN(NULL);
+---------+------------------+-----------------+------------+
| ATAN(0) | ATAN(1)          | ATAN('единица') | ATAN(NULL) |
+---------+------------------+-----------------+------------+
|       0 | 0.78539816339745 |               0 |       NULL |
+---------+------------------+-----------------+------------+
```

Как видно из листинга 13.57, передача в качестве аргумента строки приводит к ее автоматическому приведению к числу (в данном случае к 0).

Функция `ATAN()` может принимать также два аргумента `ATAN(X,Y)`, как это показано в листинге 13.58. Данная функция аналогична взятию арктангенса от частного `X` и `Y`, т. е. `ATAN(X/Y)`.

Листинг 13.58. Использование функции `ATAN(X,Y)`

```
mysql> SELECT ATAN(3,2), ATAN(3/2);
+------------------+------------------+
| ATAN(3,2)        | ATAN(3/2)        |
+------------------+------------------+
| 0.98279372324733 | 0.98279372324733 |
+------------------+------------------+
```

13.2.5. Функция *ATAN2*

Функция `ATAN2(X,Y)` аналогична `ATAN(X,Y)`, однако для вычисления квадранта возвращаемого значения в ней учитываются знаки аргументов (листинг 13.59).

Листинг 13.59. Использование функции `ATAN2()`

```
mysql> SELECT ATAN2(1,1), ATAN2(1/-1), ATAN2(-1,-1);
+------------------+-------------------+------------------+
| ATAN2(1,1)       | ATAN2(1/-1)       | ATAN2(-1,-1)     |
+------------------+-------------------+------------------+
| 0.78539816339745 | -0.78539816339745 | -2.3561944901923 |
+------------------+-------------------+------------------+
```

13.2.6. Функция *CEILING*

Функция `CEILING(X)` принимает дробное число `X` и возвращает наименьшее целое, не меньше чем `X`.

Замечание

Для функции `CEILING()` существует более короткий синоним `CEIL()`.

Листинг 13.60. Использование функции `CEILING()`

```
mysql> SELECT CEILING(0.49), CEILING(1.51), CEIL(-0.49), CEIL(-1.51);
+---------------+---------------+-------------+-------------+
| CEILING(0.49) | CEILING(1.51) | CEIL(-0.49) | CEIL(-1.51) |
+---------------+---------------+-------------+-------------+
|             1 |             2 |           0 |          -1 |
+---------------+---------------+-------------+-------------+
```

Функция `CEILING()` не производит округления, как может показаться на первый взгляд. Она возвращает первое целое число, которое встречает справа от значения аргумента (рис. 13.1). Пример использования функции `CEILING()` приведен в листинге 13.60.

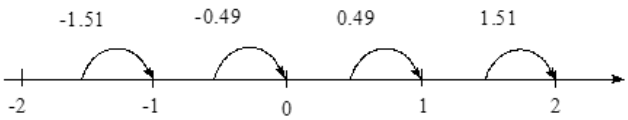

Рис. 13.1. Принцип действия функции `CEILING()`

Замечание

Результат, возвращаемый функцией `CEILING()`, имеет тип `BIGINT`.

13.2.7. Функция *COS*

Функция `COS(X)` вычисляет косинус угла `X`, заданного в радианах (листинг 13.61).

Листинг 13.61. Использование функции `COS()`

```
mysql> SELECT COS(0), COS(1), COS(PI());
+--------+------------------+-----------+
| COS(0) | COS(1)           | COS(PI()) |
+--------+------------------+-----------+
|      1 | 0.54030230586814 |        -1 |
+--------+------------------+-----------+
```

13.2.8. Функция *COT*

Функция `COT(X)` вычисляет котангенс угла `X`, заданного в радианах (листинг 13.62).

Листинг 13.62. Использование функции `COT()`

```
mysql> SELECT COT(0), COT(1), COT(10);
+--------+------------------+-----------------+
| COT(0) | COT(1)           | COT(10)         |
+--------+------------------+-----------------+
|   NULL | 0.64209261593433 | 1.5423510453569 |
+--------+------------------+-----------------+
```

13.2.9. Функция *CRC32*

Функция `CRC32(str)` вычисляет значение кода циклической проверки избыточности строки `str` и возвращает 32-битное значение в диапазоне от 0 до 2^{32}-1. Если в качестве аргумента функции передано значение `NULL`, функция возвращает `NULL` (листинг 13.63).

Листинг 13.63. Использование функции `CRC32()`

```
mysql> SELECT CRC32('MySQL'), CRC32('0'), CRC32(0), CRC32(NULL);
+----------------+------------+------------+-------------+
| CRC32('MySQL') | CRC32('0') | CRC32(0)   | CRC32(NULL) |
+----------------+------------+------------+-------------+
|     3259397556 | 4108050209 | 4108050209 |        NULL |
+----------------+------------+------------+-------------+
```

Замечание

Функция `CRC32()` введена в MySQL, начиная с версии 4.1.0.

13.2.10. Функция *DEGREES*

Функция `DEGREES(X)` возвращает значение угла `X`, преобразованное из радиан в градусы (листинг 13.64).

Листинг 13.64. Использование функции `DEGREES()`

```
mysql> SELECT DEGREES(PI()), DEGREES(PI()/4), DEGREES(NULL);
+---------------+-----------------+---------------+
| DEGREES(PI()) | DEGREES(PI()/4) | DEGREES(NULL) |
+---------------+-----------------+---------------+
|           180 |              45 |          NULL |
+---------------+-----------------+---------------+
```

13.2.11. Функция *EXP*

Функция `EXP(X)` возвращает значение степени числа `X`: e^X, где `e` — основание натурального логарифма (листинг 13.65).

Листинг 13.65. Использование функции `EXP()`

```
mysql> SELECT EXP(1), EXP(2), EXP('10');
+----------------+-----------------+-----------------+
| EXP(1)         | EXP(2)          | EXP('10')       |
+----------------+-----------------+-----------------+
| 2.718281828459 | 7.3890560989307 | 22026.465794807 |
+----------------+-----------------+-----------------+
```

Как видно из листинга 13.65, строки автоматически преобразуются в числа (в данном случае 10).

13.2.12. Функция *FLOOR*

Функция `FLOOR(X)` принимает дробное число `X` и возвращает наибольшее целое значение, не больше чем `X`. Эта функция сходна по действию с функцией `CEILING(X)`, но сдвиг происходит в другую сторону. Пример использования функции `FLOOR()` представлен в листинге 13.66.

Листинг 13.66. Использование функции `FLOOR()`

```
mysql> SELECT FLOOR(0.49), FLOOR(1.51), FLOOR(-0.49), FLOOR(-1.51);
+-------------+-------------+--------------+--------------+
| FLOOR(0.49) | FLOOR(1.51) | FLOOR(-0.49) | FLOOR(-1.51) |
+-------------+-------------+--------------+--------------+
|           0 |           1 |           -1 |           -2 |
+-------------+-------------+--------------+--------------+
```

13.2.13. Функция *LOG*

Функция `LOG(X)` возвращает натуральный логарифм (с основанием e) числа `X`.

Замечание

Кроме `LOG(X)` допускается запись `LN(X)`.

Листинг 13.67. Использование функции `LOG(X)`

```
mysql> SELECT LOG(0), LOG(1), LN(2), LN(EXP(1));
+--------+--------+------------------+------------+
| LOG(0) | LOG(1) | LN(2)            | LN(EXP(1)) |
+--------+--------+------------------+------------+
|   NULL |      0 | 0.69314718055995 |          1 |
+--------+--------+------------------+------------+
```

Кроме формата с одним аргументом, показанного в листинге 13.67, функция допускает передачу ей двух аргументов — `LOG(B,X)`, в этом случае функция возвращает логарифм числа `X` по основанию `B` (листинг 13.68).

Листинг 13.68. Использование функции `LOG(B,X)`

```
mysql> SELECT LOG(10,100), LOG(2,256);
+-------------+------------+
| LOG(10,100) | LOG(2,256) |
+-------------+------------+
|           2 |          8 |
+-------------+------------+
```

13.2.14. Функция *LOG2*

Функция `LOG2(X)`, пример которой показан в листинге 13.69, возвращает логарифм числа `X` по основанию 2.

Листинг 13.69. Использование функции `LOG2()`

```
mysql> SELECT LOG2(255), LOG2(32768);
+-----------------+-------------+
| LOG2(255)       | LOG2(32768) |
+-----------------+-------------+
| 7.9943534368589 |          15 |
+-----------------+-------------+
```

13.2.15. Функция *LOG10*

Функция `LOG10(X)` возвращает логарифм числа `X` по основанию 10 (листинг 13.70).

Листинг 13.70. Использование функции `LOG10()`

```
mysql> SELECT LOG10(0), LOG10(10), LOG10(100);
+----------+-----------+------------+
| LOG10(0) | LOG10(10) | LOG10(100) |
+----------+-----------+------------+
|     NULL |         1 |          2 |
+----------+-----------+------------+
```

13.2.16. Функция *MOD*

Функция `MOD(M, N)` аналогична оператору `M % N` и возвращает остаток от деления `M` на `N`. Более подробно применение данной функции описано в *разделе 13.1.1*.

13.2.17. Функция *PI*

Функция `PI()` не принимает аргументов и возвращает значение числа π (листинг 13.71).

Листинг 13.71. Использование функции `PI()`

```
mysql> SELECT PI()*2, PI(), PI()/2, PI()/4;
+----------+----------+------------+------------+
| PI()*2   | PI()     | PI()/2     | PI()/4     |
+----------+----------+------------+------------+
| 6.283185 | 3.141593 | 1.57079633 | 0.78539816 |
+----------+----------+------------+------------+
```

13.2.18. Функция *POW*

Функция `POW(X,Y)` возвращает значение числа `X`, возведенного в степень `Y` — X^Y.

Замечание

Для функции `POW()` существует синоним `POWER()`.

Листинг 13.72. Использование функции `POW()`

```
mysql> SELECT POW(2,3), POW(2,-3), POWER(4,.5), POWER(16,.25);
+----------+-----------+-------------+---------------+
| POW(2,3) | POW(2,-3) | POWER(4,.5) | POWER(16,.25) |
+----------+-----------+-------------+---------------+
|        8 |     0.125 |           2 |             2 |
+----------+-----------+-------------+---------------+
```

Как видно из листинга 13.72, функция `POW` допускает и дробные аргументы, что позволяет вычислять корень числа. Так в третьем столбце вычисляется квадратный корень, а в четвертом — корень четвертой степени.

13.2.19. Функция *RADIANS*

Функция `RADIANS(X)` возвращает значение угла `X`, преобразованное из градусов в радианы (листинг 13.73).

Листинг 13.73. Использование функции `RADIANS()`

```
mysql> SELECT RADIANS(0), RADIANS(180), RADIANS(360);
+------------+-----------------+-----------------+
| RADIANS(0) | RADIANS(180)    | RADIANS(360)    |
+------------+-----------------+-----------------+
|          0 | 3.1415926535898 | 6.2831853071796 |
+------------+-----------------+-----------------+
```

13.2.20. Функция *RAND*

Функция RAND() возвращает случайное значение с плавающей точкой в диапазоне от 0.0 до 1.0. Данная функция может принимать в качестве аргумента целое число N, которым инициализируется генератор случайных чисел. Каждый раз, когда функция принимает один и тот же параметр N, она возвращает один и тот же результат. Обычно функция первый раз вызывается с параметром, а все последующие разы без параметров. Пример использования функции RAND() показан в листинге 13.74.

Листинг 13.74. Использование функции RAND()

```
mysql> SELECT RAND(100), RAND(), RAND(100), RAND();
+-----------------+-----------------+-----------------+-----------------+
| RAND(100)       | RAND()          | RAND(100)       | RAND()          |
+-----------------+-----------------+-----------------+-----------------+
| 0.17353134804734| 0.10621183841136| 0.17353134804734| 0.65180650321004|
+-----------------+-----------------+-----------------+-----------------+
```

Задание начального значения для всех клиентов производится независимо друг от друга и в следующем сеансе при том же инициализирующем значении будет получен другой результат.

Функцию RAND() можно использовать после ключевого слова ORDER BY для выборки записей в случайном порядке. В качестве примера выберем из таблицы products 10 случайных товарных позиций (листинг 13.75).

Листинг 13.75. Случайная выборка из таблицы с использованием функции RAND()

```
mysql> SELECT name FROM products ORDER BY RAND() LIMIT 10;
+-----------------------------+
| name                        |
+-----------------------------+
| Asustek P4P800-VM\L i865G   |
| ASUSTEK A9600XT/TD          |
| Gigabyte GA-8IG1000         |
| Celeron D 325 2.53GHz       |
| Asustek P4C800-E Delux      |
| DDR-400 512MB Kingston      |
| Celeron D 315 2.26GHz       |
| Gigabyte GA-8IPE1000G       |
| Seagate Barracuda ST3160023A |
| Intel Pentium 4 3.0GHz      |
+-----------------------------+
```

Часто требуется осуществить выборку одной случайной записи, для этого достаточно указать после ключевого слова LIMIT цифру 1 (листинг 13.76).

Листинг 13.76. Выборка одной случайной записи

```
mysql> SELECT name FROM products ORDER BY RAND() LIMIT 1;
+-------------+
| name        |
+-------------+
| Celeron 1.8 |
+-------------+
```

13.2.21. Функция *ROUND*

Функция `ROUND(X)` возвращает округленное до ближайшего целого значение числа `X`. Данная функция может принимать два аргумента — `ROUND(X,D)`, в этом случае округление производится до числа, имеющего `D` знаков после запятой.

Листинг 13.77. Использование функции `ROUND()`

```
mysql> SELECT ROUND(4.3), ROUND(4.7), ROUND(-3.56847,2), ROUND(1,3);
+------------+------------+-------------------+------------+
| ROUND(4.3) | ROUND(4.7) | ROUND(-3.56847,2) | ROUND(1,3) |
+------------+------------+-------------------+------------+
|          4 |          5 |             -3.57 |      1.000 |
+------------+------------+-------------------+------------+
```

До версии 5.0.3 округление чисел вида 4.5, 10.5 зависело от реализации библиотеки C. Такие значения могли быть округлены как в большую, так и в меньшую сторону. Так как, начиная с версии 5.0.3, MySQL использует собственную точную математическую библиотеку, округление производится согласно следующим правилам:

- для чисел, значение которых однозначно определяется дробной частью, округление чисел со значением дробной части .5 и выше производится в сторону большего числа, для чисел со значением строго меньше .5 (например, .49999) округление производится в сторону меньшего числа;
- для приближенных чисел (записанных в научной форме 25E-1) результат попрежнему зависит от реализации библиотеки C.

В листинге 13.78 показан пример использования функции `ROUND()`.

Листинг 13.78. Округление с дробной частью 2.5

```
mysql> SELECT ROUND(2.5), ROUND(25E-1);
+------------+--------------+
| ROUND(2.5) | ROUND(25E-1) |
+------------+--------------+
| 3          |            2 |
+------------+--------------+
```

13.2.22. Функция *SIGN*

Функция `SIGN(X)` позволяет определить знак числа `X` и возвращает -1, 0 или 1, если `X` отрицательно, равно 0 или положительно соответственно (листинг 13.79).

Листинг 13.79. Использование функции `SIGN()`

```
mysql> SELECT SIGN(4.3), SIGN(0), SIGN(-4.3);
+-----------+---------+------------+
| SIGN(4.3) | SIGN(0) | SIGN(-4.3) |
+-----------+---------+------------+
|         1 |       0 |         -1 |
+-----------+---------+------------+
```

13.2.23. Функция *SIN*

Функция `SIN(X)` вычисляет синус угла `X`, заданного в радианах (листинг 13.80).

Листинг 13.80. Использование функции `SIN()`

```
mysql> SELECT SIN(0), SIN(1), SIN(PI()/2);
+--------+-----------------+-------------+
| SIN(0) | SIN(1)          | SIN(PI()/2) |
+--------+-----------------+-------------+
|      0 | 0.8414709848079 |           1 |
+--------+-----------------+-------------+
```

13.2.24. Функция *SQPT*

Функция `SQRT(X)` вычисляет квадратный корень числа `X`.

Листинг 13.81. Использование функции `SQRT()`

```
mysql> SELECT SQRT(4), SQRT(20), SQRT(-5), SQRT(NULL);
+---------+-----------------+----------+------------+
| SQRT(4) | SQRT(20)        | SQRT(-5) | SQRT(NULL) |
+---------+-----------------+----------+------------+
|       2 | 4.4721359549996 |     NULL |       NULL |
+---------+-----------------+----------+------------+
```

Как видно из листинга 13.81, передача в качестве аргумента отрицательного значения приводит к значению `NULL`.

13.2.25. Функция *TAN*

Функция `TAN(X)` вычисляет тангенс угла `X`, заданного в радианах (листинг 13.82).

Листинг 13.82. Использование функции `TAN()`

```
mysql> SELECT TAN(0), TAN(PI()/4), TAN(NULL);
+--------+-------------+-----------+
| TAN(0) | TAN(PI()/4) | TAN(NULL) |
+--------+-------------+-----------+
|      0 |           1 |      NULL |
+--------+-------------+-----------+
```

13.2.26. Функция *TRUNCATE*

Функция `TRUNCATE(X,D)` возвращает число `X` с дробной частью, имеющей `D` знаков после запятой. Если количество знаков после запятой в числе `X` больше `D`, лишние разряды усекаются, если меньше, то в конец числа добавляются нули (листинг 13.83).

Листинг 13.83. Использование функции `TRUNCATE()`

```
mysql> SELECT TRUNCATE(1.284,0), TRUNCATE(1.284,1), TRUNCATE(1.284,4);
+-------------------+-------------------+-------------------+
| TRUNCATE(1.284,0) | TRUNCATE(1.284,1) | TRUNCATE(1.284,4) |
+-------------------+-------------------+-------------------+
|                 1 |               1.2 |            1.2840 |
+-------------------+-------------------+-------------------+
```

ГЛАВА 14

Функции даты и времени

Данная группа функций предназначена для работы с данными временного типа: DATETIME, DATE, TIMESTAMP, TIME и YEAR, которые подробно рассматриваются в *главе 4*.

> ***Замечание***
>
> Отличительной чертой MySQL является то, что при использовании функций пробелы между именем функции и круглыми скобками недопустимы, т. е. написание NOW() правильное, а NOW () уже нет. Результат функции подставляется в место вызова функции.

Функции даты и времени могут принимать данные в различных форматах:

- в виде строки, содержащей полную дату и время '2005-12-27 08:30:00';
- строки, содержащей только дату '2005-12-27';
- в виде числа 20051227083000 или 20051227.

MySQL корректно распознает время во всех форматах и переводит его во внутреннее представление.

Ряд функций позволяет вернуть текущую дату или время суток. К ним относятся NOW(), CURDATE(), CURTIME(), UTC_DATE(), UTC_TIME(), UTC_TIMESTAMP(). Вычисление текущего времени в рамках одного SQL-запроса производится только один раз, сколько бы раз они не вызывались на протяжении данного запроса. Это приводит к тому, что время в рамках всего запроса остается постоянным.

14.1. Функция *ADDDATE*

Функция ADDDATE(date, INTERVAL expr type) возвращает время date, к которому прибавлен временной интервал, определяемый вторым параметром.

> ***Замечание***
>
> Функция ADDDATE() имеет синоним DATE_ADD().

Второй параметр начинается с ключевого слова INTERVAL, за которым следует число и единицы измерения type. Значения, которые может принимать параметр type, приведены в первом столбце табл. 14.1.

Листинг 14.1. Использование функции ADDDATE()

```
mysql> SELECT ADDDATE('2005-03-20',INTERVAL 10 DAY);
+-------------------------------------+
| ADDDATE('2005-03-20',INTERVAL 10 DAY) |
+-------------------------------------+
| 2005-03-30                          |
+-------------------------------------+
```

В листинге 14.1 приводится пример использования функции ADDDATE(). К дате 20 марта 2005 года добавляется 10 дней, в результате чего возвращается дата 30 марта 2005 года.

Таблица 14.1. *Типы временных интервалов функции ADDDATE()*

Тип	Описание	Формат назначения
MICROSECOND	Микросекунды	xxxxxx
SECOND	Секунды	ss
MINUTE	Минуты	mm
HOUR	Часы	hh
DAY	Дни	DD
WEEK	Недели	WW
MONTH	Месяцы	MM
QUARTER	Квартал	QQ
YEAR	Год	YY
SECOND_MICROSECOND	Секунды и микросекунды	'ss.xxxxxx'
MINUTE_MICROSECOND	Минуты, секунды и микросекунды	'mm:ss.xxxxxx'
MINUTE_SECOND	Минуты и секунды	'mm:ss'
HOUR_MICROSECOND	Часы, минуты, секунды и микросекунды	'hh:mm:ss.xxxxxx'
HOUR_SECOND	Часы, минуты и секунды	'hh:mm:ss'
HOUR_MINUTE	Часы и минуты	'hh:mm'
DAY_MICROSECOND	Дни, часы, минуты, секунды и микросекунды	'DD hh:mm:ss.xxxxxx'
DAY_SECOND	Дни, часы, минуты и секунды	'DD hh:mm:ss'
DAY_MINUTE	Дни, часы и минуты	'DD hh:mm'

Таблица 14.1 *(окончание)*

Тип	Описание	Формат назначения
DAY_HOUR	Дни и часы	'DD hh'
YEAR_MONTH	Года и месяцы	'YY-MM'

Функция ADDDATE() допускает и отрицательные значения (листинг 14.2). В этом случае происходит вычитание временного интервала.

Листинг 14.2. Работа с отрицательными значениями

```
mysql> SELECT ADDDATE('2005-03-20',INTERVAL -10 DAY);
+---------------------------------------+
| ADDDATE('2005-03-20',INTERVAL -10 DAY) |
+---------------------------------------+
| 2005-03-10                            |
+---------------------------------------+
```

Остановимся подробнее на табл. 14.1. Вместо единиц измерения DAY, которые использовались в листингах 14.1 и 14.2, можно применять любой параметр из первого столбца таблицы. Выражение expr можно задавать как в виде числа, так и в виде строки, expr содержит и нечисловые значения (пробел, дефис, двоеточие). Вместо пробела, двоеточия или дефиса можно устанавливать любой разделитель символов.

Замечание

Типы временных интервалов DAY_MICROSECOND, HOUR_MICROSECOND, MINUTE_MICROSECOND, SECOND_MICROSECOND и MICROSECOND были введены в MySQL 4.1.1, а QUARTER и WEEK — начиная с 5.0.0.

Работа с типом временного интервала MICROSECOND продемонстрирована в листинге 14.3.

Листинг 14.3. Использование типа MICROSECOND

```
mysql> SELECT ADDDATE('2005-03-20 00:00:00',INTERVAL 1 MICROSECOND);
+-----------------------------------------------------+
| ADDDATE('2005-03-20 00:00:00',INTERVAL 1 MICROSECOND) |
+-----------------------------------------------------+
| 2005-03-20 00:00:00.000001                          |
+-----------------------------------------------------+
mysql> SELECT ADDDATE('2005-03-20 00:00:00',
    -> INTERVAL 1000000 MICROSECOND);
+-------------------------------------------------------------+
| ADDDATE('2005-03-20 00:00:00',INTERVAL 1000000 MICROSECOND) |
+-------------------------------------------------------------+
| 2005-03-20 00:00:01                                         |
+-------------------------------------------------------------+
```

Как видно из листинга 14.4, если используемого формата времени недостаточно для отображения изменений, полученных в результате сложения, то MySQL автоматически расширяет тип данных.

Листинг 14.4. Использование типа MINUTE_MICROSECOND

```
mysql> SELECT ADDDATE('2005-03-20',
    -> INTERVAL '59:10.20' MINUTE_MICROSECOND);
+-------------------------------------------------------------+
| ADDDATE('2005-03-20',INTERVAL '59:10.20' MINUTE_MICROSECOND) |
+-------------------------------------------------------------+
| 2005-03-20 00:59:10.200000                                  |
+-------------------------------------------------------------+
```

При использовании типа WEEK (неделя), показанного в листинге 14.5, и QUARTER (квартал), пример которого приведен в листинге 14.6, происходит увеличение даты на интервалы, кратные семи дням и трем месяцам соответственно.

Листинг 14.5. Использование типа WEEK

```
mysql> SELECT ADDDATE('2005-03-20',INTERVAL 52 WEEK);
+---------------------------------------+
| ADDDATE('2005-03-20',INTERVAL 52 WEEK) |
+---------------------------------------+
| 2006-03-19                            |
+---------------------------------------+
mysql> SELECT ADDDATE('2005-03-20',INTERVAL 1 WEEK);
+--------------------------------------+
| ADDDATE('2005-03-20',INTERVAL 1 WEEK) |
+--------------------------------------+
| 2005-03-27                           |
+--------------------------------------+
```

Листинг 14.6. Использование типа QUARTER

```
mysql> SELECT ADDDATE('2005-03-20',INTERVAL 1 QUARTER);
+----------------------------------------+
| ADDDATE('2005-03-20',INTERVAL 1 QUARTER) |
+----------------------------------------+
| 2005-06-20                             |
+----------------------------------------+
mysql> SELECT ADDDATE('2005-03-20',INTERVAL 3 QUARTER);
+----------------------------------------+
| ADDDATE('2005-03-20',INTERVAL 3 QUARTER) |
+----------------------------------------+
| 2005-12-20                             |
+----------------------------------------+
```

Помимо функции ADDDATE(), для сложения даты с временным интервалом допустимо использование оператора "+" (листинг 14.7), который действует аналогично функции ADDDATE().

Листинг 14.7. Использование оператора "+" с временными типами данных

```
mysql> SELECT '2005-03-20' + INTERVAL '10 12:59:59.14' DAY_MICROSECOND;
+-----------------------------------------------------------+
| '2005-03-20' + INTERVAL '10 12:59:59.14' DAY_MICROSECOND |
+-----------------------------------------------------------+
| 2005-03-30 12:59:59.140000                                |
+-----------------------------------------------------------+
```

При использовании составных форматов, если значение одного из параметров выходит за границы, например, указывается 100 минут (листинг 14.8), избыток автоматически прибавляется к старшему разряду (1 час 40 минут).

Замечание

Наряду с оператором "+" можно использовать оператор "–" для вычитания временного интервала из даты.

Листинг 14.8. Выход за границы часа

```
mysql> SELECT '2005-03-20' + INTERVAL '10 12:100:59.14' DAY_MICROSECOND;
+-----------------------------------------------------------+
| '2005-03-20' + INTERVAL '10 12:100:59.14' DAY_MICROSECOND |
+-----------------------------------------------------------+
| 2005-03-30 13:40:59.140000                                |
+-----------------------------------------------------------+
mysql> SELECT '2005-03-20' + INTERVAL 30 MONTH;
+----------------------------------+
| '2005-03-20' + INTERVAL 30 MONTH |
+----------------------------------+
| 2007-09-20                       |
+----------------------------------+
```

Помимо развернутой формы синтаксиса, у функции ADDDATE() имеется краткая форма ADDDATE(expr, days). Во втором параметре days отсутствует ключевое слово INTERVAL, а тип временного интервала приравнивается к DAY (листинг 14.9).

Замечание

Краткая форма функции ADDDATE() появилась в MySQL, начиная с версии 4.1.1.

Листинг 14.9. Использование краткой формы функции ADDDATE()

```
mysql> SELECT ADDDATE('2005-03-20',6), ADDDATE('2005-03-20',-6);
+-------------------------+-------------------------+
| ADDDATE('2005-03-20',6) | ADDDATE('2005-03-20',-6) |
+-------------------------+-------------------------+
| 2005-03-26              | 2005-03-14              |
+-------------------------+-------------------------+
```

Возвращаясь к учебной базе данных, смоделируем ситуацию перехода на летнее время и вычтем из времени оформления заказа в таблице `orders` один час (листинг 14.10).

Листинг 14.10. Перевод времени на один час

```
mysql> SELECT * FROM orders;
+----------+---------+---------------------+--------+------------+
| id_order | id_user | ordertime           | number | id_product |
+----------+---------+---------------------+--------+------------+
|        1 |       3 | 2005-01-04 10:39:38 |      1 |          8 |
|        2 |       6 | 2005-02-10 09:40:29 |      2 |         10 |
|        3 |       1 | 2005-02-18 13:41:05 |      4 |         20 |
|        4 |       3 | 2005-03-10 18:20:00 |      1 |         20 |
|        5 |       3 | 2005-03-17 19:15:36 |      1 |         20 |
+----------+---------+---------------------+--------+------------+
mysql> UPDATE orders SET ordertime = ordertime - INTERVAL 1 HOUR;
mysql> SELECT * FROM orders;
+----------+---------+---------------------+--------+------------+
| id_order | id_user | ordertime           | number | id_product |
+----------+---------+---------------------+--------+------------+
|        1 |       3 | 2005-01-04 09:39:38 |      1 |          8 |
|        2 |       6 | 2005-02-10 08:40:29 |      2 |         10 |
|        3 |       1 | 2005-02-18 12:41:05 |      4 |         20 |
|        4 |       3 | 2005-03-10 17:20:00 |      1 |         20 |
|        5 |       3 | 2005-03-17 18:15:36 |      1 |         20 |
+----------+---------+---------------------+--------+------------+
```

Замечание

На нулевых значениях временного типа '0000-00-00 00:00:00' вычитание не отражается.

14.2. Функция *ADDTIME*

Функция `ADDTIME(expr1, expr2)` возвращает результат сложения двух временных значений `expr1` и `expr2`.

Замечание

Функция `ADDTIME()` появилась в MySQL в версии 4.1.1.

Листинг 14.11. Использование функции `ADDTIME()`

```
mysql> SELECT ADDTIME('1997-12-31 23:59:59.999999','1 1:1:1.000002');
+-------------------------------------------------------+
| ADDTIME('1997-12-31 23:59:59.999999','1 1:1:1.000002') |
+-------------------------------------------------------+
| 1998-01-02 01:01:01.000001                            |
+-------------------------------------------------------+
mysql> SELECT ADDTIME('01:00:00.999999', '02:00:00.999998');
+----------------------------------------------+
| ADDTIME('01:00:00.999999', '02:00:00.999998') |
+----------------------------------------------+
| 03:00:01.999997                              |
+----------------------------------------------+
```

Как видно из листинга 14.11, дата и время суток в обоих аргументах может задаваться произвольно, результат автоматически преобразуется к максимальному формату.

14.3. Функция *CONVERT_TZ*

Функция `CONVERT_TZ(dt, from_tz, to_tz)` переводит `DATETIME`-значение `dt` из часового пояса `from_tz` в `to_tz`.

Замечание

Функция `CONVERT_TZ()` появилась в MySQL в версии 4.1.3.

14.4. Функция *CURDATE*

Функция `CURDATE()`, показанная в листинге 14.12, возвращает текущую дату в формате '`YYYY-MM-DD`' или '`YYYYDDMM`' в зависимости от того, вызывается функция в текстовом или числовом контексте.

Листинг 14.12. Использование функции `CURDATE()`

```
mysql> SELECT CURDATE(), CURDATE() + 0, CURRENT_DATE(), CURRENT_DATE;
+------------+---------------+----------------+--------------+
| CURDATE()  | CURDATE() + 0 | CURRENT_DATE() | CURRENT_DATE |
+------------+---------------+----------------+--------------+
| 2005-04-17 |      20050417 | 2005-04-17     | 2005-04-17   |
+------------+---------------+----------------+--------------+
```

Замечание

Функция `CURDATE()` имеет два синонима: `CURRENT_DATE()` и `CURRENT_DATE`. То есть вариант `CURRENT_DATE` допускает написание без круглых скобок.

В листинге 4.13 демонстрируется вставка новой записи в таблицу `orders` учебной базы данных `shop`, в которой имеется `DATETIME`-поле `ordertime`. Подстановка вместо значения этого поля в SQL-операторе `INSERT` функции `CURDATE()` приводит к автоматическому назначению данному полю текущей даты.

Листинг 14.13. Вставка записи в таблицу `orders`

```
mysql> INSERT INTO orders VALUES (NULL, 3, CURDATE(), 1, 20);
mysql> SELECT * FROM orders WHERE id_order = 6;
+----------+---------+---------------------+--------+------------+
| id_order | id_user | ordertime           | number | id_product |
+----------+---------+---------------------+--------+------------+
|        6 |       3 | 2005-05-27 00:00:00 |      1 |         20 |
+----------+---------+---------------------+--------+------------+
```

Как видно из листинга 14.13, часам, минутам и секундам автоматически присваиваются нулевые значения.

14.5. Функция *CURTIME*

Функция `CURTIME()` возвращает текущее время суток в виде '`hh-mm-ss`' или '`hhmmss`' в зависимости от того, вызывается функция в текстовом или числовом контексте.

Листинг 14.14. Использование функции `CURTIME()`

```
mysql> SELECT CURTIME(), CURTIME() + 0, CURRENT_TIME(), CURRENT_TIME;
+-----------+---------------+----------------+--------------+
| CURTIME() | CURTIME() + 0 | CURRENT_TIME() | CURRENT_TIME |
+-----------+---------------+----------------+--------------+
| 00:26:08  |          2608 | 00:26:08       | 00:26:08     |
+-----------+---------------+----------------+--------------+
```

Замечание

Функции `CURTIME()` соответствуют два синонима: `CURRENT_TIME()` и `CURRENT_TIME`. То есть вариант `CURRENT_TIME` допускает написание без круглых скобок.

14.6. Функция *DATE*

Функция `DATE()` извлекает из значения даты и времени суток в формате `DATETIME` '2005-04-17 00:26:08' дату, отсекая часы, минуты и секунды (листинг 14.15).

Листинг 14.15. Использование функции DATE()

```
mysql> SELECT DATE('2005-04-17 00:26:08');
+----------------------------+
| DATE('2005-04-17 00:26:08') |
+----------------------------+
| 2005-04-17                 |
+----------------------------+
```

Замечание

Функция DATE() появилась в MySQL в версии 4.1.1.

14.7. Функция *DATEDIFF*

Функция DATEDIFF(begin, end) вычисляет разницу в днях между датами begin и end. Аргументы функции могут иметь тип DATE или DATETIME, однако при вычислении разницы используется только DATE-часть.

Листинг 14.15. Использование функции DATEDIFF()

```
mysql> SELECT DATEDIFF('2005-12-31 23:59:59','2005-12-30');
+---------------------------------------------+
| DATEDIFF('2005-12-31 23:59:59','2005-12-30') |
+---------------------------------------------+
|                                           1 |
+---------------------------------------------+
mysql> SELECT DATEDIFF('2005-11-30 23:59:59','2005-12-31');
+---------------------------------------------+
| DATEDIFF('2005-11-30 23:59:59','2005-12-31') |
+---------------------------------------------+
|                                         -31 |
+---------------------------------------------+
```

Как видно из листинга 14.15, если первый аргумент begin меньше, чем второй end, результат получается отрицательным.

Замечание

Функция DATEDIFF() появилась в MySQL, начиная с версии 4.1.1.

14.8. Функция *DATE_FORMAT*

Функция DATE_FORMAT(date,format) форматирует время date в соответствии со строкой format. В строке format могут использоваться определители, представленные в табл. 14.2.

Таблица 14.2. *Определители строки форматирования функции* `DATE_FORMAT()`

Определитель	Описание
`%a`	Сокращенное наименование дня недели (Sun...Sat)
`%b`	Сокращенное наименование месяца (Jan...Dec)
`%c`	Месяц в числовой форме (1.12)
`%D`	День месяца с английским суффиксом (1st, 2nd, 3rd и т. д.)
`%d`	День месяца в числовой форме с ведущим нулем (01..31)
`%e`	День месяца в числовой форме (1..31)
`%f`	Микросекунды (000000..999999)
`%H`	Час с ведущим нулем (00..23)
`%h`	Час с ведущим нулем (01..12)
`%I`	Час с ведущим нулем (01..12)
`%i`	Минуты с ведущим нулем (00..59)
`%j`	День года (001..366)
`%k`	Час с ведущим нулем (0..23)
`%l`	Час без ведущего нуля (1..12)
`%M`	Название месяца (January...December)
`%m`	Месяц в числовой форме с ведущим нулем (01..12)
`%p`	AM или PM (для 12-часового формата)
`%r`	Время, 12-часовой формат (hh:mm:ss AM|hh:mm:ss PM)
`%S`	Секунды (00..59)
`%s`	Секунды (00..59)
`%T`	Время, 24-часовой формат (hh:mm:ss)
`%U`	Неделя (00..52), где воскресенье считается первым днем недели
`%u`	Неделя (00..52), где понедельник считается первым днем недели
`%V`	Неделя (01..53), где воскресенье считается первым днем недели. Используется с '`%X`'
`%v`	Неделя (01..53), где понедельник считается первым днем недели. Используется с '`%x`'
`%W`	Название дня недели (Sunday...Saturday)
`%w`	День недели (0...6) 0 — Воскресенье, 6 — Суббота
`%X`	Год для недели, где воскресенье считается первым днем недели, 4 разряда, используется с '`%V`'

Таблица 14.2 *(окончание)*

Определитель	Описание
%x	Год для недели, где воскресенье считается первым днем недели, число, 4 разряда, используется с '%v'
%Y	Год, 4 разряда (YYYY)
%y	Год, 2 разряда (YY)
%%	Литерал '%'

Все другие символы, которые не входят в табл. 14.2, выводятся без изменений. Пример использования функции DATE_FORMAT() представлен в листинге 14.16.

Листинг 14.16. Использование функции DATE_FORMAT()

```
mysql> SELECT DATE_FORMAT('2005-01-01 01:59:59','На дворе %Y год');
+----------------------------------------------------+
| DATE_FORMAT('2005-01-01 01:59:59','На дворе %Y год') |
+----------------------------------------------------+
| На дворе 2005 год                                  |
+----------------------------------------------------+
```

В СУБД MySQL по умолчанию в представлении даты сначала следует год, затем месяц и день, т. к. в таком порядке удобнее сортировать результат. Это не всегда является удобным, поскольку часто требуется вначале разместить дату, затем месяц и лишь потом год. Именно для таких задач и предназначена функция DATE_FORMAT() (листинг 14.17).

Листинг 14.17. Преобразование даты из формата YYYY-MM-DD в DD.MM.YYYY

```
mysql> SELECT NOW(), DATE_FORMAT(NOW(),'%d.%m.%Y');
+---------------------+------------------------------+
| NOW()               | DATE_FORMAT(NOW(),'%d.%m.%Y') |
+---------------------+------------------------------+
| 2005-04-18 15:42:03 | 18.04.2005                   |
+---------------------+------------------------------+
```

Замечание

Определитель %f для микросекунд появился в MySQL, начиная с версии 4.1.0.

При помощи функции DATE_FORMAT() время и дату из одного формата можно преобразовывать в любой другой формат. Если входной аргумент date содержит лишнюю информацию, она будет отброшена, если информации недостаточно, недостающие элементы будут дополнены с присвоением им нулевых значений (листинг 14.18).

Листинг 14.18. Преобразование формата при помощи функции DATE_FORMAT()

```
mysql> SELECT NOW(), DATE_FORMAT(NOW(),'%T');
+---------------------+-------------------------+
| NOW()               | DATE_FORMAT(NOW(),'%T') |
+---------------------+-------------------------+
| 2005-04-18 15:51:16 | 15:51:16                |
+---------------------+-------------------------+
mysql> SELECT NOW(), DATE_FORMAT(NOW(),'%T');
+---------------------+-------------------------+
| NOW()               | DATE_FORMAT(NOW(),'%T') |
+---------------------+-------------------------+
| 2005-04-18 15:51:16 | 15:51:16                |
+---------------------+-------------------------+
mysql> SELECT DATE_FORMAT('2005-04-18', '%Y-%m-%d %T');
+-----------------------------------------+
| DATE_FORMAT('2005-04-18', '%Y-%m-%d %T') |
+-----------------------------------------+
| 2005-04-18 00:00:00                     |
+-----------------------------------------+
```

14.9. Функция *DAY*

Функция `DAY(date)` является синонимом функции `DAYOFMONTH(date)`, принимает в качестве аргумента дату `date` и возвращает порядковый номер дня в месяце (от 1 до 31).

Листинг 14.19. Использование функции DAY()

```
mysql> SELECT DAY('2005-12-03');
+-------------------+
| DAY('2005-12-03') |
+-------------------+
|                 3 |
+-------------------+
```

Как видно из листинга 14.19, результат возвращается без ведущего нуля.

Замечание

Функция `DAY()` введена в MySQL, начиная с версии 4.1.1. В более ранних реализациях MySQL следует использовать `DAYOFMONTH()`.

14.10. Функция *DAYNAME*

Функция `DAYNAME(date)` принимает в качестве аргумента дату `date` и возвращает день недели в виде полного английского названия (листинг 14.20).

Листинг 14.20. Использование функции DAYNAME()

```
mysql> SELECT DAYNAME('2006-01-01');
+-----------------------+
| DAYNAME('2006-01-01') |
+-----------------------+
| Sunday                |
+-----------------------+
```

Обычно перевод английских названий производят уже в прикладной программе. Соответствие английских и русских названий дней недели приведено в табл. 14.3.

Таблица 14.3. *Соответствие английских и русских названий дней недели*

Номер	Английское название	Сокращенный вариант	Русское название
0	Sunday	Sun	Воскресенье
1	Monday	Mon	Понедельник
2	Tuesday	Tue	Вторник
3	Wednesday	Wed	Среда
4	Thursday	Thu	Четверг
5	Friday	Fri	Пятница
6	Saturday	Sat	Суббота

Следует помнить, что в западных странах неделя начинается с воскресенья (Sunday), а не с понедельника (Monday), как в Российской Федерации, поэтому нумерация дней недели в MySQL начинается именно с воскресенья.

14.11. Функция *DAYOFMONTH*

Функция `DAYOFMONTH(date)`, показанная в листинге 14.21, принимает в качестве аргумента дату `date` и возвращает порядковый номер дня в месяце (от 1 до 31).

Листинг 14.21. Использование функции DAYOFMONTH()

```
mysql> SELECT DAYOFMONTH('2006-01-08'), DAYOFMONTH('2006-01-20');
+-------------------------+-------------------------+
| DAYOFMONTH('2006-01-08') | DAYOFMONTH('2006-01-20') |
+-------------------------+-------------------------+
|                       8 |                      20 |
+-------------------------+-------------------------+
```

14.12. Функция *DAYOFWEEK*

Функция `DAYOFWEEK(date)` принимает в качестве аргумента дату `date` и возвращает порядковый номер дня недели. Следует помнить, что в западных странах неделя начинается с воскресенья, для которого функция возвращает 1, для последнего дня недели, субботы, возвращается 7.

Листинг 14.22. Использование функции `DAYOFWEEK()`

```
mysql> SELECT DAYOFWEEK('1998-02-03'), DAYNAME('1998-02-03');
+-------------------------+-----------------------+
| DAYOFWEEK('1998-02-03') | DAYNAME('1998-02-03') |
+-------------------------+-----------------------+
|                       3 | Tuesday               |
+-------------------------+-----------------------+
```

В листинге 14.22 результат соответствует вторнику.

14.13. Функция *DAYOFYEAR*

Функция `DAYOFYEAR(date)` принимает в качестве аргумента дату `date` и возвращает порядковый номер дня в году (от 1 до 366). Пример использования функции `DAYOFYEAR()` показан в листинге 14.23.

Листинг 14.23. Использование функции `DAYOFYEAR()`

```
mysql> SELECT DAYOFYEAR('2004-02-23'), DAYOFYEAR('2004-12-31');
+-------------------------+-------------------------+
| DAYOFYEAR('2004-02-23') | DAYOFYEAR('2004-12-31') |
+-------------------------+-------------------------+
|                      54 |                     366 |
+-------------------------+-------------------------+
```

14.14. Функция *EXTRACT*

Функция `EXTRACT(type FROM datetime)` принимает дату и время суток `datetime` и возвращает часть, определяемую параметром `type`, который может принимать значения из первого столбца табл. 4.1 (листинг 14.24).

Листинг 14.24. Использование функции `EXTRACT()`

```
mysql> SELECT EXTRACT(YEAR FROM '2005-12-31 14:30:10');
+-----------------------------------------+
| EXTRACT(YEAR FROM '2005-12-31 14:30:10') |
+-----------------------------------------+
|                                    2005 |
```

```
+-----------------------------------------+
mysql> SELECT EXTRACT(MINUTE FROM '2005-12-31 14:30:10');
+-----------------------------------------+
| EXTRACT(MINUTE FROM '2005-12-31 14:30:10') |
+-----------------------------------------+
|                                       30 |
+-----------------------------------------+
mysql> SELECT EXTRACT(DAY FROM '2005-12-31 14:30:10');
+---------------------------------------+
| EXTRACT(DAY FROM '2005-12-31 14:30:10') |
+---------------------------------------+
|                                    31 |
+---------------------------------------+
mysql> SELECT EXTRACT(HOUR_SECOND FROM '2005-12-31 14:30:10');
+-----------------------------------------------+
| EXTRACT(HOUR_SECOND FROM '2005-12-31 14:30:10') |
+-----------------------------------------------+
|                                        143010 |
+-----------------------------------------------+
```

Помимо полной формы времени 'YYYY-MM-DD hh:mm:ss', допускается использование укороченной формы, включающей только дату 'YYYY-MM-DD' (листинг 14.25).

Листинг 14.25. Использование укороченного формата YYYY-MM-DD

```
mysql> SELECT EXTRACT(DAY FROM '2005-12-31');
+-------------------------------+
| EXTRACT(DAY FROM '2005-12-31') |
+-------------------------------+
|                            31 |
+-------------------------------+
```

Извлечение при помощи функции `EXTRACT()` микросекунд, секунд, минут и часов из укороченного варианта времени, включающего только дату 'YYYY-MM-DD', приводит к непредсказуемому результату.

14.15. Функция *FROM_DAYS*

Функция `FROM_DAYS(N)` (листинг 14.26) принимает число дней `N`, прошедших с нулевого года, и возвращает дату в формате 'YYYY-MM-DD'.

Листинг 14.26. Использование функции `FROM_DAYS()`

```
mysql> SELECT FROM_DAYS(0), FROM_DAYS(732419);
+--------------+-------------------+
| FROM_DAYS(0) | FROM_DAYS(732419) |
+--------------+-------------------+
| 0000-00-00   | 2005-04-18        |
+--------------+-------------------+
```

Обычно функция FROM_DAYS() используется совместно с функцией TO_DAYS(date), которая решает обратную задачу — преобразует дату date из формата 'YYYY-MM-DD' в число дней, прошедших с нулевого года (листинг 14.27).

Листинг 14.27. Совместное использование функций FROM_DAYS() и TO_DAYS()

```
mysql> SELECT TO_DAYS('2010-10-08');
+-----------------------+
| TO_DAYS('2010-10-08') |
+-----------------------+
|                734418 |
+-----------------------+
mysql> SELECT FROM_DAYS(734418 + 10);
+------------------------+
| FROM_DAYS(734418 + 10) |
+------------------------+
| 2010-10-18             |
+------------------------+
```

14.16. Функция *FROM_UNIXTIME*

Функция FROM_UNIXTIME(unix_timestamp) принимает число секунд, прошедшее с полуночи 1 января 1970 года и возвращает дату и время суток в виде строки 'YYYY-MM-DD hh:mm:ss' или в виде числа YYYYMMDDhhmmss в зависимости от того, вызвана функция в строковом или числовом контексте.

Листинг 14.28. Использование функции FROM_UNIXTIME()

```
mysql> SELECT FROM_UNIXTIME(1113837695), FROM_UNIXTIME(0);
+---------------------------+---------------------+
| FROM_UNIXTIME(1113837695) | FROM_UNIXTIME(0)    |
+---------------------------+---------------------+
| 2005-04-18 19:21:35       | 1970-01-01 03:00:00 |
+---------------------------+---------------------+
```

Как видно из листинга 14.28, передача функции значения 0 в качестве аргумента приводит к выводу вместо полуночи 3 часов ночи. Это связано с тем, что время на машине, где расположен сервер MySQL, настроено для третьей временной зоны.

Обычно функцию FROM_UNIXTIME() используют совместно с функцией UNIX_TIMESTAMP(date), которая решает обратную задачу, принимая дату и время суток date и возвращая число секунд, прошедшее с полуночи 1 января 1970 года (листинг 14.29).

Листинг 14.29. Совместное использование функций `UNIX_TIMESTAMP()` и `FROM_UNIXTIME()`

```
mysql> SELECT UNIX_TIMESTAMP('2010-10-08 00:00:00');
+---------------------------------------+
| UNIX_TIMESTAMP('2010-10-08 00:00:00') |
+---------------------------------------+
|                            1286481600 |
+---------------------------------------+
mysql> SELECT FROM_UNIXTIME(1286481600 + 3600 * 24 * 30);
+-------------------------------------------+
| FROM_UNIXTIME(1286481600 + 3600 * 24 * 30) |
+-------------------------------------------+
| 2010-11-06 23:00:00                       |
+-------------------------------------------+
```

Функция `FROM_UNIXTIME(unix_timestamp, format)` может принимать второй параметр `format`, который представляет собой строку, содержащую определители из табл. 4.2. Использование определителей позволяет отформатировать строку, например, так, как это представлено в листинге 14.30.

Листинг 14.30. Форматирование результата в функции `FROM_UNIX()`

```
mysql> SELECT
    -> FROM_UNIXTIME(1286481600,'В %Y году контракт заканчивается');
+-------------------------------------------------------------+
| FROM_UNIXTIME(1286481600,'В %Y году контракт заканчивается') |
+-------------------------------------------------------------+
| В 2010 году контракт заканчивается                          |
+-------------------------------------------------------------+
```

14.17. Функция *GET_FORMAT*

Данная функция возвращает строку форматирования, соответствующую одному из пяти стандартов времени. Полученную строку можно затем использовать в функциях `DATE_FORMAT()` и `STR_TO_DATE()`. Функция имеет следующий синтаксис:

```
GET_FORMAT (DATE|TIME|DATETIME, 'EUR'|'USA'|'JIS'|'ISO'|'INTERNAL')
```

В качестве первого параметра выступает одно из трех ключевых слов:

- `DATE` — получить формат для даты;
- `TIME` — получить формат для времени;
- `DATETIME` — получить формат для даты и времени.

Второй параметр задает название стандарта, согласно которому следует отформатировать время, и может принимать одну из строк:

- 'EUR' — европейский стандарт IBM;
- 'USA' — американский стандарт IBM;
- 'JIS' — японский индустриальный стандарт;
- 'ISO' — стандарт ISO (Международная организация стандартов);
- 'INTERNAL' — интернациональный стандарт.

Пример использования функции GET_FORMAT() приведен в листинге 14.31.

Замечание

Функция GET_FORMAT() введена в MySQL, начиная с версии 4.1.1.

Листинг 14.31. Использование функции GET_FORMAT()

```
mysql> SELECT GET_FORMAT(DATE,'EUR');
+------------------------+
| GET_FORMAT(DATE,'EUR') |
+------------------------+
| %d.%m.%Y               |
+------------------------+
```

Представление времени во всех пяти стандартах прриведено в табл. 14.4.

Таблица 14.4. *Представление времени в различных стандартах*

Вызов функции	Результат
GET_FORMAT(DATE,'USA')	'%m.%d.%Y'
GET_FORMAT(DATE,'JIS')	'%Y-%m-%d'
GET_FORMAT(DATE,'ISO')	'%Y-%m-%d'
GET_FORMAT(DATE,'EUR')	'%d.%m.%Y'
GET_FORMAT(DATE,'INTERNAL')	'%Y%m%d'
GET_FORMAT(DATETIME,'USA')	'%Y-%m-%d-%H.%i.%s'
GET_FORMAT(DATETIME,'JIS')	'%Y-%m-%d %H:%i:%s'
GET_FORMAT(DATETIME,'ISO')	'%Y-%m-%d %H:%i:%s'
GET_FORMAT(DATETIME,'EUR')	'%Y-%m-%d-%H.%i.%s'
GET_FORMAT(DATETIME,'INTERNAL')	'%Y%m%d%H%i%s'
GET_FORMAT(TIME,'USA')	'%h:%i:%s %p'
GET_FORMAT(TIME,'JIS')	'%H:%i:%s'
GET_FORMAT(TIME,'ISO')	'%H:%i:%s'
GET_FORMAT(TIME,'EUR')	'%H.%i.%S'
GET_FORMAT(TIME,'INTERNAL')	'%H%i%s'

Пример использования функции GET_FORMAT() совместно с DATE_FORMAT() приведен в листинге 14.32.

Листинг 14.32. Совместное использование GET_FORMAT() и DATE_FORMAT()

```
mysql> SELECT DATE_FORMAT('2003-10-03',GET_FORMAT(DATE,'EUR'));
+-------------------------------------------------+
| DATE_FORMAT('2003-10-03',GET_FORMAT(DATE,'EUR')) |
+-------------------------------------------------+
| 03.10.2003                                      |
+-------------------------------------------------+
```

14.18. Функция *HOUR*

Функция HOUR(datetime) возвращает значение часа (от 0 до 23) для времени datetime (листинг 14.33).

Листинг 14.33. Использование функции HOUR()

```
mysql> SELECT HOUR('12:10:45'), HOUR('2005-08-20 12:10:45');
+------------------+-----------------------------+
| HOUR('12:10:45') | HOUR('2005-08-20 12:10:45') |
+------------------+-----------------------------+
|               12 |                          12 |
+------------------+-----------------------------+
```

Одной из особенностей функции HOUR() является тот факт, что если время суток выходит за 24-часовую границу, функция вернет большее число (листинг 14.34).

Листинг 14.34. Выход за границу 24-часового дня

```
mysql> SELECT HOUR('248:10:45');
+-------------------+
| HOUR('248:10:45') |
+-------------------+
|               248 |
+-------------------+
```

14.19. Функция *LAST_DAY*

Функция LAST_DAY(datetime) (листинг 14.35) принимает в качестве параметра значение даты datetime в кратком 'YYYY-MM-DD' или расширенном формате 'YYYY-MM-DD hh:mm:ss' и возвращает дату в кратком формате 'YYYY-MM-DD', день в которой выставлен на последний день текущего месяца.

Листинг 14.35. Использование функции LAST_DAY()

```
mysql> SELECT LAST_DAY('2003-02-05'), LAST_DAY('2004-02-05');
+------------------------+------------------------+
| LAST_DAY('2003-02-05') | LAST_DAY('2004-02-05') |
+------------------------+------------------------+
| 2003-02-28             | 2004-02-29             |
+------------------------+------------------------+
mysql> SELECT LAST_DAY('2004-01-01 01:01:01');
+---------------------------------+
| LAST_DAY('2004-01-01 01:01:01') |
+---------------------------------+
| 2004-01-31                      |
+---------------------------------+
mysql> SELECT LAST_DAY('2003-03-32');
+------------------------+
| LAST_DAY('2003-03-32') |
+------------------------+
| NULL                   |
+------------------------+
```

Замечание

Функция `LAST_DAY()` введена в MySQL, начиная с версии 4.1.1.

14.20. Функция *MAKEDATE*

Функция `MAKEDATE(year, dayofyear)` принимает в качестве параметров год `year`, номер дня в году `dayofyear` и возвращает дату в формате 'YYYY-MM-DD' (листинг 14.36).

Листинг 14.36. Использование функции MAKEDATE()

```
mysql> SELECT MAKEDATE(2001,31), MAKEDATE(2001,32);
+-------------------+-------------------+
| MAKEDATE(2001,31) | MAKEDATE(2001,32) |
+-------------------+-------------------+
| 2001-01-31        | 2001-02-01        |
+-------------------+-------------------+
mysql> SELECT MAKEDATE(2001,365), MAKEDATE(2004,365);
+--------------------+--------------------+
| MAKEDATE(2001,365) | MAKEDATE(2004,365) |
+--------------------+--------------------+
| 2001-12-31         | 2004-12-30         |
+--------------------+--------------------+
```

```
mysql> SELECT MAKEDATE(2001,0), MAKEDATE(2001,400);
+------------------+--------------------+
| MAKEDATE(2001,0) | MAKEDATE(2001,400) |
+------------------+--------------------+
| NULL             | 2002-02-04         |
+------------------+--------------------+
```

Параметр `dayofyear` должен быть строго больше нуля, в противном случае возвращается `NULL`.

Замечание

Функция `MAKEDATE()` введена в MySQL, начиная с версии 4.1.1.

14.21. Функция *MAKETIME*

Функция `MAKETIME(hour, minute, second)` принимает три параметра: час `hour`, минуты `minute` и секунды `second`. В качестве результата функция возвращает время суток в формате "hh:mm:ss" (листинг 14.37).

Листинг 14.37. Использование функции `MAKETIME()`

```
mysql> SELECT MAKETIME(12,15,30), MAKETIME(12,60,0), MAKETIME(120,15,30);
+--------------------+-------------------+---------------------+
| MAKETIME(12,15,30) | MAKETIME(12,60,0) | MAKETIME(120,15,30) |
+--------------------+-------------------+---------------------+
| 12:15:30           | NULL              | 120:15:30           |
+--------------------+-------------------+---------------------+
```

Выход параметра `hour` за границу 24-часового диапазона допускается, в то время как выход за границу параметров `minute` и `second` приводит к `NULL`.

Замечание

Функция `MAKETIME()` введена в MySQL, начиная с версии 4.1.1.

14.22. Функция *MICROSECOND*

Функция `MICROSECOND(datetime)` извлекает из временного значения `datetime` микросекунды (листинг 14.38).

Листинг 14.38. Использование функции `MICROSECOND()`

```
mysql> SELECT MICROSECOND('12:00:00.123456');
+--------------------------------+
| MICROSECOND('12:00:00.123456') |
+--------------------------------+
|                         123456 |
```

```
+-------------------------------+
mysql> SELECT MICROSECOND('1997-12-31 23:59:59.000010');
+-----------------------------------------+
| MICROSECOND('1997-12-31 23:59:59.000010') |
+-----------------------------------------+
|                                      10 |
+-----------------------------------------+
```

Замечание

Функция `MICROSECOND()` введена в MySQL, начиная с версии 4.1.1.

14.23. Функция *MINUTE*

Функция `MINUTE(datetime)` возвращает значение минут (от 0 до 59) для времени суток `datetime` (листинг 14.39).

Листинг 14.39. Использование функции `MINUTE()`

```
mysql> SELECT MINUTE('98-02-03 10:05:03'), MINUTE('11:59:59');
+----------------------------+--------------------+
| MINUTE('98-02-03 10:05:03') | MINUTE('11:59:59') |
+----------------------------+--------------------+
|                          5 |                 59 |
+----------------------------+--------------------+
```

14.24. Функция *MONTH*

Функция `MONTH(datetime)` возвращает числовое значение месяца года (от 1 до 12) для даты `datetime` (листинг 14.40).

Листинг 14.40. Использование функции `MONTH()`

```
mysql> SELECT MONTH('2005-02-03'), MONTH('2005-08-12 10:00:00');
+---------------------+-----------------------------+
| MONTH('2005-02-03') | MONTH('2005-08-12 10:00:00') |
+---------------------+-----------------------------+
|                   2 |                           8 |
+---------------------+-----------------------------+
```

14.25. Функция *MONTHNAME*

Функция `MONTHNAME(datetime)` возвращает строку с названием месяца для даты `datetime` (листинг 14.41).

Листинг 14.41. Использование функции MONTHNAME()

```
mysql> SELECT MONTHNAME('2005-02-05'), MONTHNAME('2005-08-12 10:10:00');
+-------------------------+----------------------------------+
| MONTHNAME('2005-02-05') | MONTHNAME('2005-08-12 10:10:00') |
+-------------------------+----------------------------------+
| February                | August                           |
+-------------------------+----------------------------------+
```

Обычно перевод английских названий производят уже в прикладной программе. Соответствие английских и русских названий месяцев приведено в табл. 14.5.

Таблица 14.5. *Соответствие английских и русских названий месяцев*

Номер	Английское название	Сокращенный вариант	Русское название
1	January	Jan	Январь
2	February	Feb	Февраль
3	March	Mar	Март
4	April	Apr	Апрель
5	May	May	Май
6	June	Jun	Июнь
7	Jule	Jul	Июль
8	August	Aug	Август
9	September	Sep	Сентябрь
10	October	Oct	Октябрь
11	November	Nov	Ноябрь
12	December	Dec	Декабрь

14.26. Функция *NOW*

Функция `NOW()` возвращает текущие дату и время в виде строки в формате 'YYYY-MM-DD hh:mm:ss' или в виде числа YYYYMMDDhhmmss, в зависимости от того, вызывается функция в строковом или числовом контексте (листинг 14.42).

Замечание

Функция `NOW()` имеет пять синонимов: `LOCALTIME()`, `LOCALTIME`, `LOCALTIMESTAMP()`, `LOCALTIMESTAMP`, `SYSDATE()`.

Листинг 14.42. Использование функции NOW()

```
mysql> SELECT NOW()+0, NOW(), DATE_FORMAT(NOW(),'%Y');
+----------------+---------------------+------------------------+
| NOW()+0        | NOW()               | DATE_FORMAT(NOW(),'%Y') |
+----------------+---------------------+------------------------+
| 20050419184838 | 2005-04-19 18:48:38 | 2005                   |
+----------------+---------------------+------------------------+
```

14.27. Функция *PERIOD_ADD*

Функция PERIOD_ADD(period, N) добавляет N месяцев к значению даты period. Аргумент period должен быть представлен в числовом формате 'YYYYMM' или YYMM. Передача в качестве аргумента даты в любом другом формате приводит к непредсказуемому результату (листинг 14.43).

Листинг 14.43. Использование функции PERIOD_ADD()

```
mysql> SELECT PERIOD_ADD(200510,2), PERIOD_ADD(0510,6),
    -> PERIOD_ADD(0510,-6);
+----------------------+--------------------+--------------------+
| PERIOD_ADD(200510,2) | PERIOD_ADD(0510,6) | PERIOD_ADD(0510,-6) |
+----------------------+--------------------+--------------------+
|               200512 |             200604 |             200504 |
+----------------------+--------------------+--------------------+
```

Результат представляет собой число в формате YYYYMM.

14.28. Функция *PERIOD_DIFF*

Функция PERIOD_DIFF(period1, period2) вычисляет разницу в месяцах между двумя датами period1 и period2, которые представлены в числовом формате 'YYYYMM' или 'YYMM' (листинг 14.44). Передача в качестве аргументов даты в любом другом формате приводит к непредсказуемому результату.

Листинг 14.44. Использование функции PERIOD_DIFF()

```
mysql> SELECT PERIOD_DIFF(200503, 200403), PERIOD_DIFF(200411, 0502);
+----------------------------+---------------------------+
| PERIOD_DIFF(200503, 200403) | PERIOD_DIFF(200411, 0502) |
+----------------------------+---------------------------+
|                         12 |                        -3 |
+----------------------------+---------------------------+
```

Результат представляет собой целое число: положительное, если первый аргумент больше второго, и отрицательное, если первый аргумент меньше второго.

14.29. Функция *QUARTER*

Функция `QUARTER(datetime)` возвращает значение квартала года (от 1 до 4) для даты `datatime`, которая передается в формате 'YYYY-MM-DD' или 'YYYY-MM-DD hh:mm:ss' (листинг 14.45).

Листинг 14.45. Использование функции `QUARTER()`

```
mysql> SELECT QUARTER('2005-01-01'), QUARTER('2005-12-01 23:00:00');
+-----------------------+-------------------------------+
| QUARTER('2005-01-01') | QUARTER('2005-12-01 23:00:00') |
+-----------------------+-------------------------------+
|                     1 |                             4 |
+-----------------------+-------------------------------+
```

14.30. Функция *SECOND*

Функция `SECOND(time)` возвращает число секунд для времени суток `time`, которое задается либо в виде строки 'hh:mm:ss', либо числа hhmmss (листинг 14.46). Передача в качестве аргумента даты в любом другом формате приводит к непредсказуемому результату.

Листинг 14.46. Использование функции `SECOND()`

```
mysql> SELECT SECOND(120012), SECOND('10:10:10');
+----------------+--------------------+
| SECOND(120012) | SECOND('10:10:10') |
+----------------+--------------------+
|             12 |                 10 |
+----------------+--------------------+
```

14.31. Функция *SEC_TO_TIME*

Функция `SEC_TO_TIME(seconds)` принимает число секунд `seconds`, прошедшее от начала суток и возвращает время в формате 'hh:mm:ss' или hhmmss, в зависимости от того, вызывается функция в строковом или числовом контексте (листинг 14.47).

Листинг 14.47. Использование функции `SEC_TO_TIME()`

```
mysql> SELECT SEC_TO_TIME(1), SEC_TO_TIME(36584), SEC_TO_TIME(100584);
+----------------+--------------------+---------------------+
| SEC_TO_TIME(1) | SEC_TO_TIME(36584) | SEC_TO_TIME(100584) |
+----------------+--------------------+---------------------+
| 00:00:01       | 10:09:44           | 27:56:24            |
+----------------+--------------------+---------------------+
```

Замечание

Обратную задачу, т. е. преобразование времени 'hh:mm:ss' в число секунд, прошедших от начала суток, выполняет функция `TIME_TO_SEC()`.

14.32. Функция *STR_TO_DATE*

Функция `STR_TO_DATE(str, format)` является противоположенной функцией `DATE_FORMAT`. Данная функция принимает дату и время суток в виде строки `str`, соответствующей формату, задаваемому параметром `format`, которая содержит определители из табл. 4.2. В результате функция возвращает время в формате MySQL 'YYYY-MM-DD hh:mm:ss' (листинг 14.48).

Замечание

Функция `STR_TO_DATE()` появилась в MySQL, начиная с версии 4.1.1.

Листинг 14.48. Использование функции `STR_TO_DATE()`

```
mysql> SELECT STR_TO_DATE('03.10.2003 09.20','%d.%m.%Y %H.%i');
+-------------------------------------------------+
| STR_TO_DATE('03.10.2003 09.20','%d.%m.%Y %H.%i') |
+-------------------------------------------------+
| 2003-10-03 09:20:00                             |
+-------------------------------------------------+
mysql> SELECT STR_TO_DATE('10arp', '%carp');
+-------------------------------+
| STR_TO_DATE('10arp', '%carp') |
+-------------------------------+
| 0000-10-00                    |
+-------------------------------+
mysql> SELECT STR_TO_DATE('2003-15-10 00:00:00','%Y-%m-%d %H:%i:%s');
+-------------------------------------------------------+
| STR_TO_DATE('2003-15-10 00:00:00','%Y-%m-%d %H:%i:%s') |
+-------------------------------------------------------+
| NULL                                                  |
+-------------------------------------------------------+
```

Значение `NULL` в последнем примере вызвано передачей функции некорректного значения месяца (15).

14.33. Функция *SUBDATE*

Функция `SUBDATE(date, INTERVAL expr interval)` возвращает дату `date`, из которой вычитается временной интервал, определяемый вторым параметром. Функция полностью аналогична `ADDDATE()`, за исключением того, что `SUBDATE()` производит над аргументом `date` операцию вычитания, а не сложения (листинг 14.49).

Замечание

Подробное описание параметров функции приведено в *разделе "Функция ADDDATE"*.

Замечание

Функция `SUBDATE()` имеет синоним `DATE_SUB()`.

Листинг 14.49. Использование функции `SUBDATE()`

```
mysql> SELECT ADDDATE('05-10-10', INTERVAL -5 DAY);
+-------------------------------------+
| ADDDATE('05-10-10', INTERVAL -5 DAY) |
+-------------------------------------+
| 2005-10-05                          |
+-------------------------------------+
mysql> SELECT SUBDATE('05-10-10', INTERVAL 5 DAY);
+-------------------------------------+
| SUBDATE('05-10-10', INTERVAL 5 DAY) |
+-------------------------------------+
| 2005-10-05                          |
+-------------------------------------+
```

14.34. Функция *SUBTIME*

Функция `SUBTIME(datetime, time)` вычитает из временной величины `datetime` время суток `time` и возвращает результат (листинг 14.50). Величина `datetime` может быть задана либо в полном формате 'YYYY-MM-DD hh:mm:ss', либо в кратком формате 'hh:mm:ss', величину `time` следует задавать только в кратком формате 'hh:mm:ss'.

Версия

Функция `SUBTIME()` появилась в MySQL, начиная с версии 4.1.1.

Листинг 14.50. Использование функции `SUBTIME()`

```
mysql> SELECT SUBTIME('1997-12-31 23:59:59.999999','1:1:1.000002');
+-----------------------------------------------------+
| SUBTIME('1997-12-31 23:59:59.999999','1:1:1.000002') |
+-----------------------------------------------------+
| 1997-12-31 22:58:58.999997                          |
+-----------------------------------------------------+
mysql> SELECT SUBTIME('01:00:00', '02:00:00');
+--------------------------------+
| SUBTIME('01:00:00', '02:00:00') |
+--------------------------------+
| -01:00:00                      |
+--------------------------------+
```

14.35. Функция *TIME*

Функция `TIME(datetime)` извлекает время суток 'hh:mm:ss' из даты и времени `datetime` (листинг 14.51).

Замечание

Функция `TIME()` появилась в MySQL, начиная с версии 4.1.1.

Листинг 14.51. Использование функции `TIME()`

```
mysql> SELECT TIME('2003-12-31 01:02:03');
+----------------------------+
| TIME('2003-12-31 01:02:03') |
+----------------------------+
| 01:02:03                   |
+----------------------------+
mysql> SELECT TIME('2003-12-31 01:02:03.000123');
+-----------------------------------+
| TIME('2003-12-31 01:02:03.000123') |
+-----------------------------------+
| 01:02:03.000123                   |
+-----------------------------------+
```

14.36. Функция *TIMEDIFF*

Функция `TIMEDIFF(expr,expr2)` возвращает разницу между временными значениями, заданными параметрами `expr` и `expr2` (листинг 14.52).

Версия

Функция `TIMEDIFF()` появилась в MySQL, начиная с версии 4.1.1.

Листинг 14.52. Использование функции `TIMEDIFF()`

```
mysql> SELECT TIMEDIFF('2000:01:01 00:00:00','2000:01:01 00:00:00.000001');
+-------------------------------------------------------------+
| TIMEDIFF('2000:01:01 00:00:00','2000:01:01 00:00:00.000001') |
+-------------------------------------------------------------+
| -00:00:00.000001                                            |
+-------------------------------------------------------------+
mysql> SELECT TIMEDIFF('1997-12-31 23:59:59','1997-12-30 01:01:01');
+-----------------------------------------------------+
| TIMEDIFF('1997-12-31 23:59:59','1997-12-30 01:01:01') |
+-----------------------------------------------------+
| 46:58:58                                            |
+-----------------------------------------------------+
```

14.37. Функция *TIMESTAMP*

Функция `TIMESTAMP(expr)` принимает в качестве аргумента дату и время суток в полном или кратком форматах и возвращает полный вариант в формате 'YYYY-MM-DD hh:mm:ss' (листинг 14.53).

Замечание

Функция `TIMESTAMP()` появилась в MySQL, начиная с версии 4.1.1.

Листинг 14.53. Использование функции `TIMESTAMP()`

```
mysql> SELECT TIMESTAMP('2005-12-31'), TIMESTAMP('2005-12-31 15:00:10');
+-------------------------+---------------------------------+
| TIMESTAMP('2005-12-31') | TIMESTAMP('2005-12-31 15:00:10') |
+-------------------------+---------------------------------+
| 2005-12-31 00:00:00     | 2005-12-31 15:00:10              |
+-------------------------+---------------------------------+
```

Как видно из листинга 14.53, для даты и времени суток в полном формате 'YYYY-MM-DD hh:mm:ss' функция возвращает значение параметра `expr` без изменений.

Данная функция может принимать второй необязательный аргумент `time`, который представляет собой время в формате 'hh:mm:ss'. Если второй аргумент присутствует, то функция осуществляет сложение аргументов (листинг 14.54).

Листинг 14.54. Использование второго параметра в функции `TIMESTAMP()`

```
mysql> SELECT TIMESTAMP('2005-12-31','01:00:10');
+-----------------------------------+
| TIMESTAMP('2005-12-31','01:00:10') |
+-----------------------------------+
| 2005-12-31 01:00:10               |
+-----------------------------------+
```

14.38. Функция *TIMESTAMPADD*

Функция `TIMESTAMPADD(interval, int_expr, datetime_expr)` прибавляет к дате и времени суток `datetime_expr` в полном или кратком формате временной интервал `int_expr`, единицы измерения которого задаются параметром `interval`, принимающим значения из табл. 14.6.

Замечание

Для каждого из приведенных в списке значений существует синоним, отличающийся от них префиксом `SQL_TSI_`. Например, для `DAY` вполне допустимо использование синонима `SQL_TSI_DAY`.

Таблица 14.6. *Значения параметра* `interval`

Значение	Единицы измерения
FRAC_SECOND	Микросекунды
SECOND	Секунды
MINUTE	Минуты
HOUR	Часы
DAY	Дни
WEEK	Недели
MONTH	Месяцы
QUARTER	Кварталы
YEAR	Года

Замечание

Функция `TIMESTAMPADD()` появилась в MySQL, начиная с версии 5.0.0.

Функция возвращает результат в полном формате 'YYYY-MM-DD hh:mm:ss' (листинг 14.55).

Листинг 14.55. Использование функции `TIMESTAMPADD()`

```
mysql> SELECT TIMESTAMPADD(FRAC_SECOND,1,'2005-01-02');
+-----------------------------------------+
| TIMESTAMPADD(FRAC_SECOND,1,'2005-01-02') |
+-----------------------------------------+
| 2005-01-02 00:00:00.000001              |
+-----------------------------------------+
```

14.39. Функция *TIMESTAMPDIFF*

Функция `TIMESTAMPDIFF(interval, datetime_expr1, datetime_expr2)` возвращает разницу между двумя датами `datetime_expr1` или `datetime_expr2`, заданных в кратком 'YYYY-MM-DD' или полном 'YYYY-MM-DD hh:mm:ss' форматах. Единицы измерения, в которых функция возвращает результат, задаются параметром `interval`, принимающим значения из табл. 14.6.

Версия

Функция `TIMESTAMPADD()` появилась в MySQL, начиная с версии 5.0.0.

Листинг 14.56. Использование функции TIMESTAMPDIFF()

```
mysql> SELECT TIMESTAMPDIFF(MONTH,'2005-02-01','2005-05-01');
+-----------------------------------------------+
| TIMESTAMPDIFF(MONTH,'2005-02-01','2005-05-01') |
+-----------------------------------------------+
|                                             3 |
+-----------------------------------------------+
mysql> SELECT TIMESTAMPDIFF(YEAR,'2005-05-01','2004-01-01');
+----------------------------------------------+
| TIMESTAMPDIFF(YEAR,'2005-05-01','2004-01-01') |
+----------------------------------------------+
|                                           -1 |
+----------------------------------------------+
```

Как видно из листинга 14.56, функция возвращает положительную величину, если datetime_expr2 больше datetime_expr1, и отрицательную — в противном случае.

14.40. Функция *TIME_FORMAT*

Функция TIME_FORMAT(time, format) форматирует время time, которое задается в виде 'hh:mm:ss' согласно строке форматирования fomat, содержащей определители из табл. 4.2.

Замечание

Строка формата format функции TIME_FORMAT() аналогична используемой в синтаксисе функции DATE_FORMAT(), но в TIME_FORMAT() используются только определители, относящиеся к часам, минутам, секундам и микросекундам. Использование остальных определителей приводит к результату равному NULL или 0.

Листинг 14.57. Использование функции TIME_FORMAT()

```
mysql> SELECT TIME_FORMAT('100:45:10', '%H.%i.%s');
+-------------------------------------+
| TIME_FORMAT('100:45:10', '%H.%i.%s') |
+-------------------------------------+
| 100.45.10                           |
+-------------------------------------+
```

Как видно из листинга 14.57, для часов допустимо использование величин, превышающих 24-часовой интервал, для минут и секунд величины больше 59 не допустимы.

14.41. Функция *TIME_TO_SEC*

Функция TIME_TO_SEC(time) принимает в качестве аргумента время суток time 'hh:mm:ss' и возвращает число секунд, прошедшее с начала суток. Данная функция

часто используется совместно с функцией `SEC_TO_TIME()`, которая решает обратную задачу (листинг 14.58).

Листинг 14.58. Использование функции `TIME_TO_SEC()`

```
mysql> SELECT TIME_TO_SEC('22:23:00');
+-------------------------+
| TIME_TO_SEC('22:23:00') |
+-------------------------+
|                   80580 |
+-------------------------+
mysql> SELECT SEC_TO_TIME(80580 + 3600);
+---------------------------+
| SEC_TO_TIME(80580 + 3600) |
+---------------------------+
| 23:23:00                  |
+---------------------------+
```

14.42. Функция *TO_DAYS*

Функция `TO_DAYS(date)` принимает дату `date` в полном 'YYYY-MM-DD hh:mm:ss' или кратком 'YYYY-MM-DD' формате и возвращает число дней, прошедших с нулевого года (листинг 14.59).

Листинг 14.59. Использование функции `TO_DAYS()`

```
mysql> SELECT TO_DAYS('1582-01-01'), TO_DAYS('2005-10-10');
+-----------------------+-----------------------+
| TO_DAYS('1582-01-01') | TO_DAYS('2005-10-10') |
+-----------------------+-----------------------+
|                577814 |                732594 |
+-----------------------+-----------------------+
```

Использование функции `TO_DAYS()` обычно происходит совместно с функцией `FROM_DAYS()`, которая решает обратную задачу — преобразует число дней, прошедших с нулевого года, в дату 'YYYY-MM-DD' (листинг 14.60).

Листинг 14.60. Совместное использование функций `TO_DAYS()` и `FROM_DAYS()`

```
mysql> SELECT TO_DAYS('2005-10-10');
+-----------------------+
| TO_DAYS('2005-10-10') |
+-----------------------+
|                732594 |
+-----------------------+
```

```
mysql> SELECT FROM_DAYS(732594 + 31);
+------------------------+
| FROM_DAYS(732594 + 31) |
+------------------------+
| 2005-11-10             |
+------------------------+
```

14.43. Функция *UNIX_TIMESTAMP*

Функция `UNIX_TIMESTAMP()` возвращает число секунд, прошедших с полуночи 1 января 1970 года (листинг 14.61). Кроме этого, функция может принимать необязательный параметр `datetime`, определяющий дату в кратком 'YYYY-MM-DD' или полном 'YYYY-MM-DD hh:mm:ss' формате. В этом случае возвращает разницу в секундах между 1 января 1970 года и указанной в `datetime` датой.

Листинг 14.61. Использование функции `UNIX_TIMESTAMP()`

```
mysql> SELECT UNIX_TIMESTAMP(),UNIX_TIMESTAMP('1997-10-04 22:23:00');
+------------------+---------------------------------------+
| UNIX_TIMESTAMP() | UNIX_TIMESTAMP('1997-10-04 22:23:00') |
+------------------+---------------------------------------+
|       1113999060 |                             875989380 |
+------------------+---------------------------------------+
```

Замечание

Если функции `UNIX_TIMESTAMP()` через параметр `datetime` передается величина с датой до 1 января 1970 года или после 1 января 2038 года, возвращается 0.

14.44. Функция *UTC_DATE*

Функция `UTC_DATE()` возвращает текущую дату в виде строки 'YYYY-MM-DD' или числа YYYYMMDD в зависимости от того, в каком контексте вызывается функция (листинг 14.62). В отличие от функций `NOW()` или `CURDATE()`, возвращающих локальную дату, в `UTC_DATE()` время исчисляется по Гринвичу.

Замечание

Функция `UTC_DATE()` появилась в MySQL, начиная с версии 4.1.1.

Листинг 14.62. Использование функции `UTC_DATE()`

```
mysql> SELECT UTC_DATE(), UTC_DATE() + 0;
+------------+----------------+
| UTC_DATE() | UTC_DATE() + 0 |
+------------+----------------+
| 2005-04-20 |       20050420 |
+------------+----------------+
```

Замечание

Для функции `UTC_DATE()` имеется синоним `UTC_DATE`, т. е. возможно использование функции без круглых скобок.

14.45. Функция *UTC_TIME*

Функция `UTC_TIME()` возвращает текущее время суток в виде строки 'hh:mm:ss' или числа hhmmss в зависимости от того, в каком контексте вызывается функция (листинг 14.63). В отличие от функций `NOW()` или `CURTIME()`, возвращающих локальную дату, в `UTC_DATE()` время исчисляется по Гринвичу.

Замечание

Функция `UTC_TIME()` появилась в MySQL, начиная с версии 4.1.1.

Листинг 14.63. Использование функции `UTC_TIME()`

```
mysql> SELECT UTC_TIME(), UTC_TIME + 0, NOW();
+------------+--------------+---------------------+
| UTC_TIME() | UTC_TIME + 0 | NOW()               |
+------------+--------------+---------------------+
| 12:40:29   |       124029 | 2005-04-20 16:40:29 |
+------------+--------------+---------------------+
```

Замечание

Для функции `UTC_TIME()` имеется синоним `UTC_TIME`, т. е. возможно использование функции без круглых скобок.

14.46. Функция *UTC_TIMESTAMP*

Функция `UTC_TIMESTAMP()` возвращает дату и время суток в виде строки 'YYYY-MM-DD hh:mm:ss' или в виде числа 'YYYYMMDDhhmmss' в зависимости от того, в каком контексте вызывается функция (листинг 14.64). В отличие от функции `NOW()`, возвращающей локальную дату, в `UTC_DATE()` время исчисляется по Гринвичу.

Замечание

Функция `UTC_TIMESTAMP()` появилась в MySQL, начиная с версии 4.1.1.

Листинг 14.64. Использование функции `UTC_TIMESTAMP()`

```
mysql> SELECT UTC_TIMESTAMP(), UTC_TIMESTAMP + 0, NOW();
+---------------------+------------------+---------------------+
| UTC_TIMESTAMP()     | UTC_TIMESTAMP + 0 | NOW()               |
+---------------------+------------------+---------------------+
| 2005-04-20 13:00:27 |    20050420130027 | 2005-04-20 17:00:27 |
+---------------------+------------------+---------------------+
```

Замечание

Для функции `UTC_TIMESTAMP()` имеется синоним `UTC_TIMESTAMP`, т. е. возможно использование функции без круглых скобок.

14.47. Функция *WEEK*

Функция `WEEK(date)` возвращает номер недели (от 0 до 53) в году для даты `date` (листинг 14.65). Предполагается, что неделя начинается с воскресенья.

Листинг 14.65. Использование функции `WEEK()`

```
mysql> SELECT WEEK('2005-01-31'), WEEK('2006-12-31');
+--------------------+--------------------+
| WEEK('2005-01-31') | WEEK('2006-12-31') |
+--------------------+--------------------+
|                  5 |                 53 |
+--------------------+--------------------+
```

Функция также допускает наличие второго необязательного параметра `WEEK(date, mode)` (листинг 14.66), который представляет собой число из первого столбца табл. 14.7 и определяет порядок вычисления номера недели.

Таблица 14.7. *Значение второго параметра функции `WEEK()`*

Значение `mode`	Первый день недели	Диапазон	Неделя с номером 1 — это первая неделя
0	Воскресенье	0-53	с воскресеньем в этом году
1	Понедельник	0-53	с более чем 3 днями в этом году
2	Воскресенье	1-53	с воскресеньем в этом году
3	Понедельник	1-53	с более чем 3 днями в этом году
4	Воскресенье	0-53	с более чем 3 днями в этом году
5	Понедельник	0-53	с понедельником в этом году
6	Воскресенье	1-53	с более чем 3 днями в этом году
7	Понедельник	1-53	с понедельником в этом году

По умолчанию, когда второй аргумент `mode` не указывается, ему присваивается значение 4.

Замечание

До версии MySQL 4.0.17 вычисления по умолчанию проводились в режиме 3.

Листинг 14.66. Использование второго параметра в функции `WEEK()`

```
mysql> SELECT WEEK('2005-04-24',1), WEEK('2005-04-24',0),
    -> DAYNAME('2005-04-24');
```

```
+----------------------+----------------------+-----------------------+
| WEEK('2005-04-24',1) | WEEK('2005-04-24',0) | DAYNAME('2005-04-24') |
+----------------------+----------------------+-----------------------+
|                   16 |                   17 | Sunday                |
+----------------------+----------------------+-----------------------+
```

14.48. Функция *WEEKDAY*

Функция `WEEKDAY(date)` принимает дату `date` в полном 'YYYY-MM-DD hh:mm:ss' или кратком 'YYYY-MM-DD' формате (листинг 14.67) и возвращает номер дня недели (0 для понедельника, 1 для вторника, ..., 6 для воскресенья).

Замечание

Функция `WEEKDAY()` появилась в MySQL, начиная с версии 4.1.1.

Листинг 14.67. Использование функции `WEEKDAY()`

```
mysql> SELECT WEEKDAY('2005-02-03 22:23:00'),
    -> DAYNAME('2005-02-03 22:23:00')
+--------------------------------+-------------------------------+
| WEEKDAY('2005-02-03 22:23:00') | DAYNAME('2005-02-03 22:23:00') |
+--------------------------------+-------------------------------+
|                              3 | Thursday                      |
+--------------------------------+-------------------------------+
mysql> SELECT WEEKDAY('2005-11-05'), DAYNAME('2005-11-05');
+-----------------------+-----------------------+
| WEEKDAY('2005-11-05') | DAYNAME('2005-11-05') |
+-----------------------+-----------------------+
|                     5 | Saturday              |
+-----------------------+-----------------------+
```

14.49. Функция *WEEKOFYEAR*

Функция `WEEKOFYEAR(datetime)` возвращает порядковый номер недели в году (от 1 до 53) для даты `datetime` (листинг 14.68). Порядок вычисления номера соответствует режиму 3 функции `WEEK(date,3)` из табл. 4.7.

Листинг 14.68. Использование функции `WEEKOFYEAR()`

```
mysql> SELECT WEEKOFYEAR('1998-02-20');
+--------------------------+
| WEEKOFYEAR('1998-02-20') |
+--------------------------+
|                        8 |
+--------------------------+
```

14.50. Функция *YEAR*

Функция `YEAR(datetime)` возвращает год (от 1000 до 9999) для даты `datetime` (листинг 14.69).

Листинг 14.69. Использование функции `YEAR()`

```
mysql> SELECT YEAR('05-02-03'), YEAR('2005-02-03'),
    -> YEAR('05-02-03 10:23:15');
+------------------+--------------------+---------------------------+
| YEAR('05-02-03') | YEAR('2005-02-03') | YEAR('05-02-03 10:23:15') |
+------------------+--------------------+---------------------------+
|             2005 |               2005 |                      2005 |
+------------------+--------------------+---------------------------+
```

14.51. Функция *YEARWEEK*

Функция `YEARWEEK(date)` возвращает число в формате 'YYYYWW', представляющее год и номер недели (от 0 до 53) в году, соответствующее дате `date`. Предполагается, что неделя начинается с воскресенья (как это принято на Западе).

Листинг 14.70. Использование функции `YEARWEEK()`

```
mysql> SELECT YEARWEEK('2006-01-01');
+------------------------+
| YEARWEEK('2006-01-01') |
+------------------------+
|                 200601 |
+------------------------+
mysql> SELECT YEARWEEK('2006-01-01',0), YEARWEEK('2006-01-01',1);
+--------------------------+--------------------------+
| YEARWEEK('2006-01-01',0) | YEARWEEK('2006-01-01',1) |
+--------------------------+--------------------------+
|                   200601 |                   200552 |
+--------------------------+--------------------------+
```

Как видно из листинга 14.70, функция может принимать дополнительный аргумент, задающий день, с которого начинается неделя. Передача в качестве такого аргумента значения 0 задает в качестве первого дня воскресенье, 1 — понедельник и т. д. Функция `YAERWEEK()` никогда не возвращает для недели значение 0, если же такое значение выпадает, то она возвращает предыдущий год и число 52 для недели.

ГЛАВА 15

Строковые функции

Данная группа функций позволяет осуществлять различные преобразования строк. Все строковые функции условно можно поделить на функции общего назначения, которые будут рассмотрены в данной главе, и поисковые функции, описываемые в *главах 17 и 18*.

> ***Замечание***
>
> Отличительной чертой MySQL является то, что при использовании функций пробелы между именем функции и круглыми скобками недопустимы, т. е. написание `NOW()` правильное, а `NOW ()` уже нет. Результат функции подставляется в место вызова функции.

При использовании строковых функций следует помнить, что позиция в строке начинается с 1, а не с 0, как, например, в С-подобных языках программирования.

15.1. Функция *ASCII*

Функция `ASCII(str)` возвращает значение ASCII-кода первого символа строки `str` (листинг 15.1). Для пустой строки возвращается значение 0 или `NULL`, если в качестве `str` передано значение `NULL`.

Листинг 15.1. Использование функции `ASCII()`

```
mysql> SELECT ASCII('2'), ASCII(2), ASCII(0), ASCII(NULL);
+------------+----------+----------+-------------+
| ASCII('2') | ASCII(2) | ASCII(0) | ASCII(NULL) |
+------------+----------+----------+-------------+
|         50 |       50 |       48 |        NULL |
+------------+----------+----------+-------------+
```

Возвращаемые функцией символы лежат в диапазоне от 0 до 256.

15.2. Функция *BIN*

Функция `BIN(N)` принимает десятичное число `N` и возвращает его двоичное представление (листинг 15.2).

> ***Замечание***
>
> Вызов функции `BIN(N)` эквивалентен вызову функции `CONV(N,10,2)`.

Листинг 15.2. Использование функции `BIN()`

```
mysql> SELECT BIN(12), BIN(95);
+---------+---------+
| BIN(12) | BIN(95) |
+---------+---------+
| 1100    | 1011111 |
+---------+---------+
```

15.3. Функция *BIT_LENGTH*

Функция `BIT_LENGTH(str)` принимает строку `str` и возвращает ее длину в битах (листинг 15.3).

Листинг 15.3. Использование функции `BIT_LENGTH()`

```
mysql> SELECT BIT_LENGTH('text'), BIT_LENGTH(1);
+--------------------+---------------+
| BIT_LENGTH('text') | BIT_LENGTH(1) |
+--------------------+---------------+
|                 32 |             8 |
+--------------------+---------------+
```

Обычно каждый символ представлен 8 битами (1 байт), но в ряде кодировок под символ отводится более одного бита. Узнать максимальное число бит, отводимое под символ, можно при помощи SQL-запроса `SHOW CHARACTER SET` (см. листинг 11.17).

15.4. Функция *CHAR*

Функция `CHAR(N1,N2,...)` принимает последовательность из ASCII-кодов и возвращает строку, созданную путем объединения соответствующих им символов (листинг 15.4).

Листинг 15.4. Использование функции CHAR()

```
mysql> SELECT CHAR(77,121,83,81,76);
+-----------------------+
| CHAR(77,121,83,81,76) |
+-----------------------+
| MySQL                 |
+-----------------------+
```

Замечание

Аргументы со значениями NULL игнорируются.

15.5. Функция *CHAR_LENGTH*

Функция CHAR_LENGTH(str) возвращает число символов в строке str, в отличие от функции LENGTH() расчет ведется не по байтам, а по символам, поэтому функция корректно работает с многобайтовыми кодировками (листинг 15.5).

Замечание

Для функции CHAR_LENGTH() имеется синоним CHARACTER_LENGTH().

Листинг 15.5. Использование функции CHAR_LENGTH()

```
mysql> SELECT CHAR_LENGTH(CONVERT('MySQL' USING ucs2));
+-----------------------------------------+
| CHAR_LENGTH(CONVERT('MySQL' USING ucs2)) |
+-----------------------------------------+
|                                       5 |
+-----------------------------------------+
mysql> SELECT LENGTH(CONVERT('MySQL' USING ucs2));
+-------------------------------------+
| LENGTH(CONVERT('MySQL' USING ucs2)) |
+-------------------------------------+
|                                  10 |
+-------------------------------------+
```

Возвращаясь к учебной базе данных, подсчитаем число символов в полях name и description таблицы products (листинг 15.6).

Листинг 15.6. Подсчет числа символов в текстовом поле

```
mysql> SELECT name, CHAR_LENGTH(name) AS lname,
    -> CHAR_LENGTH(description) AS ldescription FROM products;
```

```
+-------------------------------+-------+--------------+
| name                          | lname | ldescription |
+-------------------------------+-------+--------------+
| Celeron 1.8                   |    11 |           59 |
| Celeron 2.0GHz                |    14 |           55 |
| Celeron 2.4GHz                |    14 |           55 |
| Celeron D 320 2.4GHz          |    20 |           60 |
| Celeron D 325 2.53GHz         |    21 |           62 |
| Celeron D 315 2.26GHz         |    21 |           62 |
| Intel Pentium 4 3.2GHz        |    22 |           78 |
| Intel Pentium 4 3.0GHz        |    22 |           80 |
| Intel Pentium 4 3.0GHz        |    22 |           77 |
| Gigabyte GA-8I848P-RS         |    21 |          117 |
| Gigabyte GA-8IG1000           |    19 |          124 |
| Gigabyte GA-8IPE1000G         |    21 |          139 |
| Asustek P4C800-E Delux        |    22 |          145 |
| Asustek P4P800-VM\L i865G     |    25 |          136 |
| Epox EP-4PDA3I                |    14 |          116 |
| ASUSTEK A9600XT/TD            |    18 |          107 |
| ASUSTEK V9520X                |    14 |           95 |
| SAPPHIRE 256MB RADEON 9550    |    26 |           56 |
| GIGABYTE AGP GV-N59X128D      |    24 |           49 |
| Maxtor 6Y120P0                |    14 |           56 |
| Maxtor 6B200P0                |    14 |           56 |
| Samsung SP0812C               |    15 |          133 |
| Seagate Barracuda ST3160023A  |    28 |           70 |
| Seagate ST3120026A            |    18 |           60 |
| DDR-400 256MB Kingston        |    22 |           42 |
| DDR-400 256MB Hynix Original  |    29 |           48 |
| DDR-400 256MB PQI             |    17 |           37 |
| DDR-400 512MB Kingston        |    22 |           42 |
| DDR-400 512MB PQI             |    17 |           37 |
| DDR-400 512MB Hynix           |    19 |           39 |
+-------------------------------+-------+--------------+
```

15.6. Функция *CHARSET*

Функция `CHARSET(str)` возвращает имя кодировки, в которой представлена строка (листинг 15.7).

Замечание

Функция `CHARSET()` введена в СУБД MySQL, начиная с версии 4.1.0.

Листинг 15.7. Использование функции `CHARSET()`

```
mysql> SELECT CHARSET('Программирование'),
    -> CHARSET(CONVERT('MySQL' USING utf8));
```

```
+----------------------------+-------------------------------------+
| CHARSET('Программирование') | CHARSET(CONVERT('MySQL' USING utf8)) |
+----------------------------+-------------------------------------+
| cp1251                     | utf8                                |
+----------------------------+-------------------------------------+
```

15.7. Функция *COLLATION*

Функция `COLLATION(str)` возвращает порядок сортировки, установленный для кодировки аргумента `str` (листинг 15.8).

Замечание

Функция `COLLATION()` введена в MySQL, начиная с версии 4.1.0.

Листинг 15.8. Использование функции `COLLATION()`

```
mysql> SELECT COLLATION('Программирование'),
    -> COLLATION(CONVERT('MySQL' USING utf8));
+------------------------------+--------------------------------------+
| COLLATION('Программирование') | COLLATION(CONVERT('MySQL' USING utf8))|
+------------------------------+--------------------------------------+
| cp1251_general_ci            | utf8_general_ci                      |
+------------------------------+--------------------------------------+
```

15.8. Функция *COMPRESS*

Функция `COMPRESS(string_to_compress)` сжимает строку `string_to_compress`.

Замечание

Данная функция работает только в том случае, если MySQL скомпилирована совместно с библиотекой zlib, в противном случае всегда возвращается значение `NULL`.

Замечание

Функция `COMPRESS()` введена в MySQL, начиная с версии 4.1.1.

Листинг 15.9. Использование функции `COMPRESS()`

```
mysql> SELECT LENGTH(COMPRESS(REPEAT('a',1000))) AS a1000,
    -> LENGTH(COMPRESS(REPEAT('a',16))) AS a16,
    -> LENGTH(COMPRESS('')) AS a0;
+-------+-----+----+
| a1000 | a16 | a0 |
+-------+-----+----+
|    21 |  15 |  0 |
+-------+-----+----+
```

Пустая строка не подвергается сжатию и возвращается без изменений. Если функция заканчивается пробельными символами, необходимо добавить завершающую точку ".", в противном случае они будут отброшены.

Так как сжатые данные представляют собой бинарный код, их рекомендуется хранить в столбцах типа BLOB, а не TEXT, CHAR или VARCHAR, где они могут быть повреждены кодировкой.

Следует отметить, что эффект сжатия заметен только для длинных текстовых строк с большим числом повторяющихся элементов.

Листинг 15.10. Сжатие описания товарных позиций в таблице products

```
mysql> select CHAR_LENGTH(description) AS ldescription,
    -> CHAR_LENGTH(COMPRESS(description)) AS compress from products;
+--------------+----------+
| ldescription | compress |
+--------------+----------+
|           59 |       70 |
|           55 |       65 |
|           55 |       66 |
|           60 |       72 |
|           62 |       74 |
|           62 |       74 |
|           78 |       90 |
|           80 |       92 |
|           77 |       89 |
|          117 |      123 |
|          124 |      134 |
|          139 |      149 |
|          145 |      154 |
|          136 |      143 |
|          116 |      127 |
|          107 |      115 |
|           95 |      107 |
|           56 |       69 |
|           49 |       62 |
|           56 |       68 |
|           56 |       66 |
|          133 |      136 |
|           70 |       82 |
|           60 |       71 |
|           42 |       57 |
|           48 |       63 |
|           37 |       52 |
|           42 |       57 |
|           37 |       52 |
|           39 |       54 |
+--------------+----------+
```

Как видно из листинга 15.10, применение функции COMPRESS() к коротким текстовым строкам приводит только к увеличению их длины. Это связано с тем, что при сжатии данных добавляются заголовки, позволяющие произвести разархивирование. Для коротких строк длина заголовков превышает эффект от сжатия.

15.9. Функция *CONCAT*

Функция CONCAT(str1,str2, ...) возвращает строку, созданную путем объединения всех аргументов, количество которых не ограничено (листинг 15.11). Если хотя бы один из аргументов равен NULL, возвращается значение NULL.

Листинг 15.11. Использование функции CONCAT()

```
mysql> SELECT CONCAT('ab'), CONCAT('ab','cd'), CONCAT('ab',NULL);
+--------------+-------------------+-------------------+
| CONCAT('ab') | CONCAT('ab','cd') | CONCAT('ab',NULL) |
+--------------+-------------------+-------------------+
| ab           | abcd              | NULL              |
+--------------+-------------------+-------------------+
mysql> SELECT CONCAT('Hello',' ','world','!');
+---------------------------------+
| CONCAT('Hello',' ','world','!') |
+---------------------------------+
| Hello world!                    |
+---------------------------------+
```

Замечание

Числовые аргументы конвертируются в строки.

Если аргументы функции имеют различные кодировки, результирующая строка будет иметь кодировку первого аргумента.

Помимо функции CONCAT(), возможно объединение строк при задании их одной за другой, как это показано в листинге 15.12.

Листинг 15.12. Альтернативный способ объединения строк

```
mysql> SELECT 'ab' 'cd' 'ef', 'ab' 'cd' = 'abcd';
+--------+--------------------+
| ab     | 'ab' 'cd' = 'abcd' |
+--------+--------------------+
| abcdef |                  1 |
+--------+--------------------+
```

Как видно из листинга 15.12, сравнение следующих друг за другом строк 'ab' и 'cd' с конечной строкой 'abcd' приводит к значению 1 (истина).

15.10. Функция *CONCAT_WS*

Функция `CONCAT_WS(separator, str1, str2,...)`, как и предыдущая строка, объединяет аргументы `str1`, `str2` и т. д., помещая между ними разделитель `separator` (листинг 15.13).

В отличие от функции `CONCAT()`, данная функция игнорирует параметры, равные `NULL`, за исключением `separator`, равенство которого `NULL` обращает результат в `NULL`.

Листинг 15.13. Использование функции `CONCAT_WS()`

```
mysql> SELECT CONCAT_WS(NULL,'ab','bc'), CONCAT_WS(',','ab',NULL,'bc');
+---------------------------+-------------------------------+
| CONCAT_WS(NULL,'ab','bc') | CONCAT_WS(',','ab',NULL,'bc') |
+---------------------------+-------------------------------+
| NULL                      | ab,bc                         |
+---------------------------+-------------------------------+
```

Если в качестве параметров `str1`, `str2` и т. д. передаются пустые строки, они также игнорируются.

15.11. Функция *CONV*

Функция `CONV(N, from_base, to_base)` преобразует число `N` из одной системы исчисления `from_base` в другую `to_base`. Параметры `from_base` и `to_base` могут принимать значения от 2 до 36.

Замечание

Функция `CONV()` возвращает `NULL`, если хотя бы один из аргументов равен `NULL`.

Листинг 15.14. Использование функции `CONV()`

```
mysql> SELECT CONV('E',16,2), CONV('E',16,10), CONV('E',16,8);
+----------------+-----------------+----------------+
| CONV('E',16,2) | CONV('E',16,10) | CONV('E',16,8) |
+----------------+-----------------+----------------+
| 1110           | 14              | 16             |
+----------------+-----------------+----------------+
```

В листинге 15.14 шестнадцатеричное число E (14) переводится в двоичную, десятичную и восьмеричную системы исчисления.

15.12. Функция *ELT*

Функция `ELT(N, str1, str2, str3,...)` возвращает `N`-ю строку из списка аргументов `str1`, `str2`, `str3` и т. д. Так для `N` = 1 возвращается `str1`, для `N` = 2 — `str2` и т. д. Если строка с номером `N` отсутствует или равна `NULL`, возвращается `NULL` (листинг 15.15).

Листинг 15.15. Использование функции `ELT()`

```
mysql> SELECT ELT(3,'Hello','this','is','a','base');
+-------------------------------------+
| ELT(3,'Hello','this','is','a','base') |
+-------------------------------------+
| is                                  |
+-------------------------------------+
```

15.13. Функция *EXPORT_SET*

Функция `EXPORT_SET(bits, on, off[, separator[, number_of_bits]])` возвращает число `bits` в двоичном представлении, 1 в котором заменяется на `on`, а 0 на `off` (листинг 15.16). Необязательный параметр `separator` задает разделитель, которым по умолчанию является запятая. Параметр `number_of_bits` позволяет ограничить число возвращаемых функцией символов (по умолчанию равно 64). Если данный параметр не задан, будут выведены все 64 символа, недостающие символы будут установлены в 0 (`off`).

Листинг 15.16. Использование функции `EXPORT_SET()`

```
mysql> SELECT EXPORT_SET(7,'+','-','');
+-----------------------------------------------------------------+
| EXPORT_SET(7,'+','-','')                                        |
+-----------------------------------------------------------------+
| +++------------------------------------------------------------ |
+-----------------------------------------------------------------+
mysql> SELECT EXPORT_SET(7,'+','-','',5);
+----------------------------+
| EXPORT_SET(7,'+','-','',5) |
+----------------------------+
| +++--                      |
+----------------------------+
```

Следует отметить, что в отличие от обычно двоичного числа символы располагаются слева направо, а не справа налево. Так в листинге 15.17 число 4, в двоичном представлении равное 100, представляется как 001.

Листинг 15.17. Порядок следования символов

```
mysql> SELECT EXPORT_SET(4,'1','0','');
+-----------------------------------------------------------------+
| EXPORT_SET(4,'1','0','')                                        |
+-----------------------------------------------------------------+
| 0010000000000000000000000000000000000000000000000000000000000000 |
+-----------------------------------------------------------------+
mysql> SELECT EXPORT_SET(4,'1','0','',3);
+----------------------------+
| EXPORT_SET(4,'1','0','',3) |
+----------------------------+
| 001                        |
+----------------------------+
```

Замечание

Функция `EXPORT_SET()` возвращает значение `NULL`, если один из аргументов равен `NULL`.

15.14. Функция *FIELD*

Функция `FIELD(str, str1, str2, ...)` (листинг 15.18) находит строку `str` в списке строк `str1`, `str2`, ... и возвращает номер строки в этом списке (нумерация начинается с 1).

Листинг 15.18. Использование функции `FIELD()`

```
mysql> SELECT FIELD('ej', 'Hej', 'ej', 'Heja', NULL, 'foo');
+-----------------------------------------------+
| FIELD('ej', 'Hej', 'ej', 'Heja', NULL, 'foo') |
+-----------------------------------------------+
|                                             2 |
+-----------------------------------------------+
mysql> SELECT FIELD(NULL, 'Hej', 'ej', 'Heja', NULL, 'foo');
+-----------------------------------------------+
| FIELD(NULL, 'Hej', 'ej', 'Heja', NULL, 'foo') |
+-----------------------------------------------+
|                                             0 |
+-----------------------------------------------+
```

Наличие в списке строк значения `NULL` игнорируется, если же искомая строка `str` равна `NULL`, возвращается 0.

15.15. Функция *FIND_IN_SET*

Функция `FIND_IN_SET(str, str_list)` ищет вхождение строки `str` в список `str_list` и возвращает номер строки в этом списке (нумерация начинается с 1). Если вхождение не найдено, возвращается 0. Параметр `str_list` представляет собой набор строк, разделенных запятыми (листинг 15.19).

Замечание

Функция `FIND_IN_SET()` идеально подходит для определения, входит ли значение в строковый набор поля `SET`.

Листинг 15.19. Использование функции `FIND_IN_SET()`

```
mysql> SELECT FIND_IN_SET('show','hide,lock,show');
+-------------------------------------+
| FIND_IN_SET('show','hide,lock,show') |
+-------------------------------------+
|                                   3 |
+-------------------------------------+
mysql> SELECT FIND_IN_SET('active','hide,lock,show');
+----------------------------------------+
| FIND_IN_SET('active','hide,lock,show') |
+----------------------------------------+
|                                      0 |
+----------------------------------------+
```

Если один из аргументов функции `FIND_IN_SET()` равен `NULL`, функция возвращает `NULL`.

15.16. Функция *FORMAT*

Функция `FORMAT(X,D)` преобразует число `X` в строку с `D` знаками после запятой (листинг 15.20). Каждые три разряда отделяются запятой.

Листинг 15.20. Использование функции `FORMAT()`

```
mysql> SELECT FORMAT(12345.12345,4), FORMAT(12345.1,4),
    -> FORMAT(12345.1,0);
+-----------------------+-------------------+-------------------+
| FORMAT(12345.12345,4) | FORMAT(12345.1,4) | FORMAT(12345.1,0) |
+-----------------------+-------------------+-------------------+
| 12,345.1234           | 12,345.1000       | 12,345            |
+-----------------------+-------------------+-------------------+
```

Замечание

Если аргумент `D` имеет нулевое значение, возвращаемый результат не включает десятичную точку и дробную часть.

15.17. Функция *HEX*

Функция `HEX(N_or_S)` возвращает значение аргумента `N_or_S` в виде шестнадцатеричного числа. Аргумент `N_or_S` может быть как числом, так и строкой. В первом случае возвращается обычное шестнадцатеричное число (листинг 15.21).

Листинг 15.21. Использование функции `HEX()` с числовым параметром

```
mysql> SELECT HEX(255), HEX(5678);
+----------+-----------+
| HEX(255) | HEX(5678) |
+----------+-----------+
| FF       | 162E      |
+----------+-----------+
```

Когда в качестве аргумента выступает строка, то функция переводит в шестнадцатеричное представление каждый символ строки и объединяет результат (листинг 15.22).

Листинг 15.22. Использование функции `HEX()` со строковым параметром

```
mysql> SELECT HEX('MySQL'), HEX('M'), HEX('y'), HEX('S'), HEX('Q'),
    -> HEX('L');
+--------------+----------+----------+----------+----------+----------+
| HEX('MySQL') | HEX('M') | HEX('y') | HEX('S') | HEX('Q') | HEX('L') |
+--------------+----------+----------+----------+----------+----------+
| 4D7953514C   | 4D       | 79       | 53       | 51       | 4C       |
+--------------+----------+----------+----------+----------+----------+
mysql> SELECT HEX('255'), HEX('5678');
+------------+-------------+
| HEX('255') | HEX('5678') |
+------------+-------------+
| 323535     | 35363738    |
+------------+-------------+
```

Замечание

Для преобразования чисел с использованием других систем исчисления можно воспользоваться функцией `CONV()`.

15.18. Функция *INSERT*

Функция `INSERT(str, pos, len, new_str)` возвращает строку `str`, в которой подстрока, начинающаяся с позиции `pos` и имеющая длину `len` символов, заменена подстрокой `new_str`. Функция возвращает строку `str` без изменений, если значение `pos` находится за пределами строки (листинг 15.23).

Листинг 15.23. Использование функции `INSERT()`

```
mysql> SELECT INSERT('Quadratic', 3, 4, 'What');
+-----------------------------------+
| INSERT('Quadratic', 3, 4, 'What') |
+-----------------------------------+
| QuWhattic                         |
+-----------------------------------+
mysql> SELECT INSERT('Quadratic', -1, 4, 'What');
+------------------------------------+
| INSERT('Quadratic', -1, 4, 'What') |
+------------------------------------+
| Quadratic                          |
+------------------------------------+
mysql> SELECT INSERT('Quadratic', 3, 100, 'What');
+-------------------------------------+
| INSERT('Quadratic', 3, 100, 'What') |
+-------------------------------------+
| QuWhat                              |
+-------------------------------------+
```

15.19. Функция *INSTR*

Функция `INSTR(str, substr)` возвращает позицию первого вхождения подстроки `substr` в строку `str` (листинг 15.24).

Замечание

Функция `INSTR()` эквивалентна функции `LOCATE()` с двумя параметрами, за исключением того факта, что в `LOCATE(substr,str)` порядок параметров обратный.

Листинг 15.24. Использование функции `INSTR()`

```
mysql> SELECT INSTR('foobarbar', 'bar'), INSTR('xbar', 'foobar');
+---------------------------+-------------------------+
| INSTR('foobarbar', 'bar') | INSTR('xbar', 'foobar') |
+---------------------------+-------------------------+
|                         4 |                       0 |
+---------------------------+-------------------------+
```

15.20. Функция *LEFT*

Функция `LEFT(str,len)` возвращает `len` крайних левых символов строки `str` (листинг 15.25). Если аргумент `len` превышает число символов в строке `str`, она возвращается без изменений. Если `len` равно `NULL` или меньше 0, возвращается пустая строка.

Листинг 15.25. Использование функции `LEFT()`

```
mysql> SELECT LEFT('программирование', 5), LEFT('программирование', -1);
+----------------------------+-----------------------------+
| LEFT('программирование', 5) | LEFT('программирование', -1) |
+----------------------------+-----------------------------+
| прогр                      |                             |
+----------------------------+-----------------------------+
```

15.21. Функция *LENGTH*

Функция `LENGTH(str)` (листинг 15.26) возвращает длину строки `str`. Данная функция может работать некорректно с многобайтными кодировками, т. к. фактически возвращается число байт в строке.

Листинг 15.26. Использование функции `LENGTH()`

```
mysql> SELECT LENGTH('MySQL');
+-----------------+
| LENGTH('MySQL') |
+-----------------+
|               5 |
+-----------------+
```

15.22. Функция *LOAD_FILE*

Функция `LOAD_FILE(file_name)` читает файл `file_name` и возвращает его содержимое в виде строки. Файл должен быть расположен на сервере, а в параметре `file_name` необходимо указывать абсолютный путь к файлу.

Если файл не существует или не может быть прочитан из-за того, что MySQL не обладает достаточными правами доступа, то возвращается `NULL`. Создадим файл "`hello`" следующего содержания:

```
This is a text from file "hello"
```

Файл расположим в директории C:\mysql\bin\, тогда получить доступ к нему можно при помощи SQL-запроса, представленного в листинге 15.27.

Листинг 15.27. Использование функции `LOAD_FILE()`

```
mysql> SELECT LOAD_FILE("C:/mysql/bin/hello"), LOAD_FILE("hello");
+----------------------------------+--------------------+
| LOAD_FILE("C:/mysql/bin/hello")  | LOAD_FILE("hello") |
+----------------------------------+--------------------+
| This is a text from file "hello" | NULL               |
+----------------------------------+--------------------+
```

Обычно данную функцию используют для заполнения BLOB-полей в операторах INSERT и UPDATE (листинг 15.28).

Листинг 15.28. Использование функции LOAD_FILE() при заполнении BLOB-полей

```
UPDATE tbl_name
SET blob_column = LOAD_FILE('C:/tmp/picture')
WHERE id=1;
```

15.23. Функция *LOCATE*

Функция LOCATE(substr, str [, pos]) (листинг 15.29) возвращает позицию первого вхождения подстроки substr в строку str. Если ни одного вхождения не найдено, возвращается 0. Если хотя бы один из аргументов функции равен NULL, функция возвращает NULL. При наличии необязательного аргумента pos поиск начинается с позиции, указанной в этом аргументе.

Листинг 15.29. Использование функции LOCATE()

```
mysql> SELECT LOCATE('bar','foobarbar'), LOCATE('xbar', 'foobar');
+---------------------------+-------------------------+
| LOCATE('bar','foobarbar') | LOCATE('xbar', 'foobar') |
+---------------------------+-------------------------+
|                         4 |                       0 |
+---------------------------+-------------------------+
mysql> SELECT LOCATE('bar','foobarbar',5);
+-----------------------------+
| LOCATE('bar','foobarbar',5) |
+-----------------------------+
|                           7 |
+-----------------------------+
```

15.24. Функция *LOWER*

Функция LOWER(str) возвращает строку str, в которой все символы записаны строчными символами (листинг 15.30).

Замечание

Для функции LOWER() имеется синоним — LCASE().

Листинг 15.30. Использование функции LOWER()

```
mysql> SELECT LOWER('Hello World!');
```

```
+-----------------------+
| LOWER('Hello World!') |
+-----------------------+
| hello world!          |
+-----------------------+
```

Замечание

Функция `LOWER()` корректно работает с многобайтными кодировками.

15.25. Функция *LPAD*

Функция `LPAD(str, len, padstr)` (листинг 15.31) возвращает строку `str`, дополненную слева строкой `padstr` до длины `len`.

Листинг 15.31. Использование функции `LPAD()`

```
mysql> SELECT LPAD('111',10,'123456789'), LPAD('111',10,'57');
+----------------------------+---------------------+
| LPAD('111',10,'123456789') | LPAD('111',10,'57') |
+----------------------------+---------------------+
| 1234567111                 | 5757575111          |
+----------------------------+---------------------+
```

Если строка `str` содержит более `len` символов, она усекается до `len` (листинг 15.32).

Листинг 15.32. Усечение строки при помощи `LPAD()`

```
mysql> SELECT LPAD('Hello world!',10,'1');
+----------------------------+
| LPAD('Hello world!',10,'1') |
+----------------------------+
| Hello worl                 |
+----------------------------+
```

15.26. Функция *LTRIM*

Функция `LTRIM(str)` возвращает строку `str`, в которой удалены все начальные пробелы (листинг 15.33).

Листинг 15.33. Использование функции `LTRIM()`

```
mysql> SELECT '  111  ' AS ST,LTRIM('  111  ') AS LT;
+---------+-------+
| ST      | LT    |
+---------+-------+
|   111   | 111   |
+---------+-------+
```

Замечание

Функция корректно работает с многобайтными кодировками.

15.27. Функция *MAKE_SET*

Функция `MAKE_SET(N, bit0_str, bit1_str,...)` представляет число `N` в двоичной форме, создавая список для типа `SET`, заменяя 1 значениями из списка `bit0_str`, `bit1_str`, ..., при этом для первого разряда используется `bit0_str`, для второго `bit1_str` и т. д.

Листинг 15.34. Использование функции `MAKE_SET()`

```
mysql> SELECT MAKE_SET(8,'a','b','c','d','e'),
    -> MAKE_SET(7,'a','b','c','d');
+---------------------------------+----------------------------+
| MAKE_SET(8,'a','b','c','d','e') | MAKE_SET(7,'a','b','c','d') |
+---------------------------------+----------------------------+
| d                               | a,b,c                      |
+---------------------------------+----------------------------+
```

Результат в листинге 15.34 легко понять, если вспомнить, что цифра 8 в двоичном представлении записывается как 1000, а 7 как 0111.

15.28. Функция *MID*

Функция `MID(str, pos [, len])` возвращает подстроку строки `str`, которая начинается с позиции `pos` и имеет длину `len` символов (листинг 15.35). Если третий параметр `len` не указывается, то подстрока возвращается, начиная с позиции `pos` и до конца строки `str`.

Замечание

Для функции `MID()` имеется синоним — `SUBSTRING()`.

Листинг 15.35. Использование функции `MID()`

```
mysql> SELECT MID('Hello world! This is a database!',7,5);
+---------------------------------------------+
| MID('Hello world! This is a database!',7,5) |
+---------------------------------------------+
| world                                       |
+---------------------------------------------+
mysql> SELECT MID('Hello world! This is a database!',7);
+--------------------------------------------+
| MID('Hello world! This is a database!',7) |
+--------------------------------------------+
| world! This is a database!                 |
+--------------------------------------------+
```

Рассмотрим более сложный пример, связанный с учебной базой данных shop. Пусть стоит задача вывести первые буквы фамилий покупателей из таблицы users, причем в результирующий список каждая буква должна входить только один раз.

Листинг 15.36. Вывод первых букв фамилий покупателей из таблицы users

```
mysql> SELECT id_user, surname FROM users ORDER BY id_user;
+---------+-----------+
| id_user | surname   |
+---------+-----------+
|       1 | Иванов    |
|       2 | Лосев     |
|       3 | Симдянов  |
|       4 | Кузнецов  |
|       5 | Нехорошев |
|       6 | Корнеев   |
+---------+-----------+
mysql> SELECT MID(surname,1,1) AS first_char FROM users
    -> GROUP BY first_char ORDER BY first_char;
+------------+
| first_char |
+------------+
| И          |
| К          |
| Л          |
| Н          |
| С          |
+------------+
```

Как видно из листинга 15.36, в таблице имеются две фамилии на букву "К": Кузнецов и Корнеев, однако, благодаря конструкции GROUP BY, в результирующий список попадает только одна буква "К". Для удобства ссылки на новый столбец при помощи ключевого слова AS результат функции переименовывается в столбец first_char.

15.29. Функция *OCT*

Функция OCT(N) возвращает число N в восьмеричной системе исчисления (листинг 15.37).

Листинг 15.37. Использование функции OCT()

```
mysql> SELECT OCT('255'), OCT(255), OCT(5678);
+------------+----------+-----------+
| OCT('255') | OCT(255) | OCT(5678) |
+------------+----------+-----------+
| 377        | 377      | 13056     |
+------------+----------+-----------+
```

В отличие от функции `HEX()`, функция `OCT()` не делает различий между строковыми и числовыми типами параметра `N`.

Замечание

Для преобразования чисел с использованием других систем исчисления можно воспользоваться функцией `CONV()`.

15.30. Функция *ORD*

Функция `ORD(str)` (листинг 15.38) возвращает ASCII-код крайнего левого символа строки `str`, аналогична функции `ASCII()`, но в отличие от последней корректно работает с многобайтными кодировками.

Листинг 15.38. Использование функции `ORD()`

```
mysql> SELECT ORD('MySQL'),ORD('База');
+--------------+-------------+
| ORD('MySQL') | ORD('База') |
+--------------+-------------+
|           77 |         129 |
+--------------+-------------+
```

15.31. Функция *POSITION*

Функция `POSITION(substr IN str)` полностью аналогична функции `LOCATE(substr, str)`.

15.32. Функция *QUOTE*

Функция `QUOTE(str)` экранирует строку `str` с тем, чтобы получить корректное значение для SQL-выражения (листинг 15.39). Строка заключается в одинарные кавычки, и каждое вхождение одинарной кавычки ('), обратного слэша (\), значения `ASCII NUL` и <Ctrl>+<Z> экранируются обратным слэшем.

Замечание

Если аргумент `str` равен `NULL`, то тогда результатом будет слово `NULL` без окружающих кавычек.

Листинг 15.39. Использование функции `QUOTE()`

```
mysql> SELECT QUOTE("Don't!");
+-----------------+
| QUOTE("Don't!") |
+-----------------+
| 'Don\'t!'       |
+-----------------+
```

15.33. Функция *REPEAT*

Функция `REPEAT(str, count)` (листинг 15.40) возвращает строку, полученную из `count` повторений строки `str`. Если аргумент `count` имеет отрицательное значение или ноль, возвращается пустая строка.

Листинг 15.40. Использование функции `REPEAT()`

```
mysql> SELECT REPEAT('MySQL ',3);
+--------------------+
| REPEAT('MySQL ',3) |
+--------------------+
| MySQL MySQL MySQL  |
+--------------------+
```

15.34. Функция *REPLACE*

Функция `REPLACE(str, from_str, to_str)` (листинг 15.41) возвращает строку `str`, в которой все подстроки `from_str` заменены `to_str`.

Листинг 15.41. Использование функции `REPLACE()`

```
mysql> SELECT REPLACE('MS SQL - лучшая база данных', 'MS SQL', 'MySQL');
+-------------------------------------------------------------+
| REPLACE('MS SQL - лучшая база данных', 'MS SQL', 'MySQL') |
+-------------------------------------------------------------+
| MySQL - лучшая база данных                                  |
+-------------------------------------------------------------+
```

Замечание

Функция `REPLACE()` корректно работает с многобайтными кодировками.

Часто результат, возвращаемый одной функцией, может быть использован в качестве аргумента второй функции. В случае функции `REPLACE()` часто прибегают к каскадному выполнению этой функции, т. е. возвращаемый функцией результат передается в качестве параметра `str` следующей функции `REPLACE()`. В листинге 15.42 в одном запросе к таблице `products` учебной базы данных `shop` подстрока "`Pentium 4`" заменяется на "`P IV`", а "`Celeron`" на "`Cel`".

Листинг 15.42. Каскадное использование функции `REPLACE()`

```
mysql> SELECT id_product, REPLACE(name,'Pentium 4', 'P IV')
    -> FROM products ORDER BY id_product LIMIT 10;
```

```
+------------+-----------------------------------+
| id_product | REPLACE(name,'Pentium 4', 'P IV') |
+------------+-----------------------------------+
|          1 | Celeron 1.8                       |
|          2 | Celeron 2.0GHz                    |
|          3 | Celeron 2.4GHz                    |
|          4 | Celeron D 320 2.4GHz              |
|          5 | Celeron D 325 2.53GHz             |
|          6 | Celeron D 315 2.26GHz             |
|          7 | Intel P IV 3.2GHz                 |
|          8 | Intel P IV 3.0GHz                 |
|          9 | Intel P IV 3.0GHz                 |
|         10 | Gigabyte GA-8I848P-RS             |
+------------+-----------------------------------+
mysql> SELECT id_product,
    -> REPLACE(REPLACE(name,'Pentium 4','P IV'),'Celeron','Cel') AS res
    -> FROM products ORDER BY id_product LIMIT 10;
+------------+-----------------------+
| id_product | res                   |
+------------+-----------------------+
|          1 | Cel 1.8               |
|          2 | Cel 2.0GHz            |
|          3 | Cel 2.4GHz            |
|          4 | Cel D 320 2.4GHz      |
|          5 | Cel D 325 2.53GHz     |
|          6 | Cel D 315 2.26GHz     |
|          7 | Intel P IV 3.2GHz     |
|          8 | Intel P IV 3.0GHz     |
|          9 | Intel P IV 3.0GHz     |
|         10 | Gigabyte GA-8I848P-RS |
+------------+-----------------------+
```

Как видно из листинга 15.42, обе замены удалось провести в рамках одного запроса. Важно помнить, что изменения касаются только результата и не затрагивают данных самой таблицы.

Функция `REPLACE()` может так же использоваться для удаления подстрок, с этой целью достаточно в качестве третьего параметра `to_str` передать пустую строку. Так, в листинге 15.43 удаляются все пробельные символы из результата запроса.

Листинг 15.43. Удаление пробельных символов

```
mysql> SELECT id_product, REPLACE(name,' ','') FROM products
    -> ORDER BY id_product LIMIT 10;
+------------+----------------------+
| id_product | REPLACE(name,' ','') |
+------------+----------------------+
|          1 | Celeron1.8           |
|          2 | Celeron2.0GHz        |
```

```
|          3 | Celeron2.4GHz        |
|          4 | CeleronD3202.4GHz    |
|          5 | CeleronD3252.53GHz   |
|          6 | CeleronD3152.26GHz   |
|          7 | IntelPentium43.2GHz  |
|          8 | IntelPentium43.0GHz  |
|          9 | IntelPentium43.0GHz  |
|         10 | GigabyteGA-8I848P-RS |
+------------+----------------------+
```

15.35. Функция *REVERSE*

Функция `REVERSE(str)` возвращает строку `str`, записанную в обратном порядке (листинг 15.44).

Листинг 15.44. Использование функции `REVERSE()`

```
mysql> SELECT REVERSE("Hello, I'm MySQL");
+----------------------------+
| REVERSE("Hello, I'm MySQL") |
+----------------------------+
| LQSyM m'I ,olleH           |
+----------------------------+
```

Замечание

Функция `REVERSE()` корректно работает с многобайтными кодировками.

15.36. Функция *RIGHT*

Функция `RIGHT(str, len)` (листинг 15.45) возвращает `len` крайних правых символов строки `str`, или всю строку `str`, если ее длина короче `len`. Функция возвращает пустую строку, если аргумент `len` равен `NULL` или меньше 1. Если строка `str` равна `NULL`, функция `RIGHT()` также возвращает `NULL`.

Листинг 15.45. Использование функции `RIGHT()`

```
mysql> SELECT RIGHT('MySQL',NULL),RIGHT('MySQL',0), RIGHT('MySQL',2);
+---------------------+------------------+------------------+
| RIGHT('MySQL',NULL) | RIGHT('MySQL',0) | RIGHT('MySQL',2) |
+---------------------+------------------+------------------+
|                     |                  | QL               |
+---------------------+------------------+------------------+
```

Замечание

Функция `RIGHT()` корректно работает с многобайтными кодировками.

15.37. Функция *RDAP*

Функция `RPAD(str, len, padstr)` (листинг 15.46) возвращает строку `str`, дополненную справа строкой `padstr` до длины `len`.

Листинг 15.46. Использование функции `RDAP()`

```
mysql> SELECT RPAD('111',10,'23456789'), RPAD('111',10,'57');
+---------------------------+---------------------+
| RPAD('111',10,'23456789') | RPAD('111',10,'57') |
+---------------------------+---------------------+
| 1112345678                | 1115757575          |
+---------------------------+---------------------+
```

Если строка `str` содержит более `len` символов, она усекается до `len` (листинг 15.47).

Листинг 15.47. Отбрасывание лишних символов

```
mysql> SELECT RPAD('Hello world!',10,'1');
+----------------------------+
| RPAD('Hello world!',10,'1') |
+----------------------------+
| Hello worl                  |
+----------------------------+
```

15.38. Функция *RTRIM*

Функция `RTRIM(str)` возвращает строку `str`, в которой удалены все конечные пробелы (листинг 15.48).

Листинг 15.48. Использование функции `RTRIM()`

```
mysql> SELECT '  111  ' AS ST,RTRIM('  111  ') AS RT;
+---------+-------+
| ST      | RT    |
+---------+-------+
|   111   |   111 |
+---------+-------+
```

Замечание

Функция `RTRIM()` корректно работает с многобайтными кодировками.

15.39. Функция *SOUNDEX*

Функция `SOUNDEX(str)` (листинг 15.49) возвращает произношение строки `str` по алгоритму, описанному Д. Кнутом. Все не алфавитные символы в строке игнорируются, а национальные символы рассматриваются как гласные.

Листинг 15.49. Использование функции `SOUNDEX()`

```
mysql> SELECT SOUNDEX('Hello'), SOUNDEX('Howl'), SOUNDEX('Cow'),
    -> SOUNDEX('Cowl');
+------------------+-----------------+----------------+-----------------+
| SOUNDEX('Hello') | SOUNDEX('Howl') | SOUNDEX('Cow') | SOUNDEX('Cowl') |
+------------------+-----------------+----------------+-----------------+
| H400             | H400            | C000           | C400            |
+------------------+-----------------+----------------+-----------------+
```

15.40. Функция *SPACE*

Функция `SPACE(N)` возвращает строку, состоящую из `N` пробелов, или пустую строку, если `N` имеет отрицательное значение (листинг 15.50).

Замечание

Если `N` принимает значение `NULL`, то функция также возвращает `NULL`.

Листинг 15.50. Использование функции `SPACE()`

```
mysql> SELECT SPACE(NULL), SPACE(0), SPACE(20);
+-------------+----------+----------------------+
| SPACE(NULL) | SPACE(0) | SPACE(20)            |
+-------------+----------+----------------------+
| NULL        |          |                      |
+-------------+----------+----------------------+
```

15.41. Функция *SUBSTRING*

Функция `SUBSTRING(str, pos [, len])` возвращает подстроку строки `str`, начинающуюся с позиции `pos` и имеющую длину в `len` символов (листинг 15.51). Если третий параметр `len` не указывается, то подстрока возвращается, начиная с позиции `pos` и до конца строки `str`.

Замечание

Для функции `SUBSTRING()` имеется синоним — `MID()`.

Листинг 15.51. Использование функции SUBSTRING()

```
mysql> SELECT SUBSTRING('Hello world! This is a database!',7,5);
+-----------------------------------------------+
| SUBSTRING('Hello world! This is a database!',7,5) |
+-----------------------------------------------+
| world                                         |
+-----------------------------------------------+
mysql> SELECT SUBSTRING('Hello world! This is a database!',7);
+-----------------------------------------------+
| SUBSTRING('Hello world! This is a database!',7) |
+-----------------------------------------------+
| world! This is a database!                    |
+-----------------------------------------------+
```

Помимо указанного выше синтаксиса, в функции SUBSTRING можно использовать следующий синтаксис: SUBSTRING(str FROM pos [FOR len]), как показано в листинге 15.52.

Листинг 15.52. Альтернативный синтаксис функции SUBSTRING()

```
mysql> SELECT SUBSTRING('Hello world! This is a database!' FROM 7 FOR 5);
+-------------------------------------------------------------+
| SUBSTRING('Hello world! This is a database!' FROM 7 FOR 5) |
+-------------------------------------------------------------+
| world                                                       |
+-------------------------------------------------------------+
mysql> SELECT SUBSTRING('Hello world! This is a database!' FROM 7);
+-------------------------------------------------------+
| SUBSTRING('Hello world! This is a database!' FROM 7) |
+-------------------------------------------------------+
| world! This is a database!                            |
+-------------------------------------------------------+
```

Замечание

Функция SUBSTRING() корректно работает с многобайтными кодировками.

15.42. Функция *SUBSTRING_INDEX*

Функция SUBSTRING_INDEX(str, delim, N) (листинг 15.53) возвращает подстроку строки str. Если параметр N имеет положительное значение, то функция SUBSTRING_INDEX() находит N-е вхождение (отсчет слева) подстроки delim в строку str и возвращает всю часть строки, расположенную слева от подстроки delim. Если N имеет отрицательное значение, то находится N-е вхождение (отсчет справа) подстроки delim в строку str и возвращается часть строки, расположенная справа от подстроки delim.

Листинг 15.53. Использование функции SUBSTRING_INDEX()

```
mysql> SELECT SUBSTRING_INDEX('www.mysql.com', '.', 2);
+-----------------------------------------+
| SUBSTRING_INDEX('www.mysql.com', '.', 2) |
+-----------------------------------------+
| www.mysql                                |
+-----------------------------------------+
mysql> SELECT SUBSTRING_INDEX('www.mysql.com', '.', -2);
+-----------------------------------------+
| SUBSTRING_INDEX('www.mysql.com', '.', -2) |
+-----------------------------------------+
| mysql.com                                |
+-----------------------------------------+
```

Замечание

Функция `SUBSTRING_INDEX()` корректно работает с многобайтными кодировками.

15.43. Функция *TRIM*

Функция `TRIM([[BOTH | LEADING | TRAILING] [remstr] FROM] str)` (листинг 15.54) удаляет из строки `str` расположенные в начале (в конце) символы, указанные в строке `remstr`. Если указано ключевое слово `LEADING`, удаляются расположенные в начале символы, если `TRAILING` — в конце, если `BOTH` — и в начале, и в конце. Если ни одно из ключевых слово не задано, то по умолчанию устанавливается `BOTH`. Если строка `remstr` не задана, в качестве удаляемых символов выступают пробелы.

Листинг 15.54. Использование функции TRIM()

```
mysql> SELECT '  111  ' AS ST,TRIM('  111  ') AS BT;
+---------+-----+
| ST      | BT  |
+---------+-----+
|   111   | 111 |
+---------+-----+
mysql> SELECT TRIM(LEADING 'x' FROM 'xxxbarxxx');
+-----------------------------------+
| TRIM(LEADING 'x' FROM 'xxxbarxxx') |
+-----------------------------------+
| barxxx                            |
+-----------------------------------+
mysql> SELECT TRIM(BOTH 'x' FROM 'xxxbarxxx');
+--------------------------------+
| TRIM(BOTH 'x' FROM 'xxxbarxxx') |
+--------------------------------+
| bar                            |
+--------------------------------+
```

```
mysql> SELECT TRIM(TRAILING 'xyz' FROM 'barxxyz');
+------------------------------------+
| TRIM(TRAILING 'xyz' FROM 'barxxyz') |
+------------------------------------+
| barx                               |
+------------------------------------+
```

Замечание

Функция `TRIM()` корректно работает с многобайтными кодировками.

15.44. Функция *UNCOMPRESS*

Функция `UNCOMPRESS(string_to_uncompress)` (листинг 15.55) разархивирует строку `string_to_uncompress`, сжатую при помощи функции `COMPRESS()`. Если в качестве аргумента функции передается обычная, не сжатая строка, возвращается `NULL`.

Замечание

Данная функция работает только в том случае, если пакет MySQL скомпилирован с библиотекой zlib, в противном случае возвращается всегда `NULL`.

Замечание

Функция `UNCOMPRESS()` введена в MySQL, начиная с версии 4.1.1.

Листинг 15.55. Использование функции `UNCOMPRESS()`

```
mysql> SELECT UNCOMPRESS(COMPRESS('any string'));
+------------------------------------+
| UNCOMPRESS(COMPRESS('any string')) |
+------------------------------------+
| any string                         |
+------------------------------------+
mysql> SELECT UNCOMPRESS('any string');
+--------------------------+
| UNCOMPRESS('any string') |
+--------------------------+
| NULL                     |
+--------------------------+
```

15.45. Функция *UNCOMPRESSED_LENGTH*

Функция `UNCOMPRESSED_LENGTH(compressed_string)` возвращает длину строки `compressed_string` сжатой функцией `COMPRESS()` до применения данной функции (листинг 15.56). То есть при помощи функции `UNCOMPRESSED_LENGTH()` можно выяснить, какой длина будет строки после разархивирования ее функцией `UNCOMPRESS()`.

Замечание

Функция `UNCOMPRESS()` введена в MySQL, начиная с версии 4.1.1.

Листинг 15.56. Использование функции `UNCOMPRESSED_LENGTH()`

```
mysql> SELECT UNCOMPRESSED_LENGTH(COMPRESS(REPEAT('a',1000)));
+-------------------------------------------------+
| UNCOMPRESSED_LENGTH(COMPRESS(REPEAT('a',1000))) |
+-------------------------------------------------+
|                                            1000 |
+-------------------------------------------------+
mysql> SELECT LENGTH(COMPRESS(REPEAT('a',1000)));
+------------------------------------+
| LENGTH(COMPRESS(REPEAT('a',1000))) |
+------------------------------------+
|                                 21 |
+------------------------------------+
```

15.46. Функция *UNHEX*

Функция `UNHEX(str)` является обратной функции `HEX()` и интерпретирует каждую пару символов строки `str` как шестнадцатеричный код, который необходимо преобразовать в символ (листинг 15.57).

Замечание

Функция `UNCOMPRESS()` введена в MySQL, начиная с версии 4.1.2.

Листинг 15.57. Использование функции `UNHEX()`

```
mysql> SELECT UNHEX('4D7953514C'), 0x4D7953514C;
+---------------------+--------------+
| UNHEX('4D7953514C') | 0x4D7953514C |
+---------------------+--------------+
| MySQL               | MySQL        |
+---------------------+--------------+
mysql> SELECT UNHEX(HEX('string')), UNHEX(HEX('1234'));
+----------------------+--------------------+
| UNHEX(HEX('string')) | UNHEX(HEX('1234')) |
+----------------------+--------------------+
| string               | 1234               |
+----------------------+--------------------+
```

15.47. Функция *UPPER*

Функция `UPPER(str)` переводит все символы строки `str` в верхний регистр (листинг 15.58). Если строка `str` имеет значение `NULL`, возвращается `NULL`.

Замечание

Функции `UPPER()` имеет синоним — `UCASE()`.

Листинг 15.58. Использование функции `UPPER()`

```
mysql> SELECT UPPER('Hello world!');
+-----------------------+
| UPPER('Hello world!') |
+-----------------------+
| HELLO WORLD!          |
+-----------------------+
```

ГЛАВА 16

Безопасность и MySQL

В данной главе рассматриваются многочисленные функции шифрования данных, которые входят в состав СУБД MySQL. Чаще всего шифрованию подвергаются пароли. В случае кражи базы данных злоумышленник получит доступ только к зашифрованным записям, на расшифровку которых потребуется время, за которое пароли перестанут быть актуальными.

Различают необратимое шифрование, когда данные, подвергшиеся шифрованию, невозможно расшифровать, и так называемое симметричное шифрование — когда данные шифруются с использованием секретного ключа, обладая которым можно произвести дешифровку данных. Как это на первый взгляд ни удивительно, но чаще прибегают именно к необратимому шифрованию данных. Введенный пользователем пароль также подвергается необратимому шифрованию, и между собой сравниваются зашифрованные строки — если они совпадают, пароль верен. Благодаря этой схеме никто, кроме пользователя, даже администратор базы данных, не имеет доступа к настоящему паролю.

16.1. Функции *AES_ENCRYPT* и *AES_DECRYPT*

Функции `AES_ENCRYPT()` и `AES_DECRYPT()` позволяют симметрично зашифровать и расшифровать данные, используя официальный алгоритм AES (Advanced Encryption Standard). В нем применяется кодирование с 128-битным ключом, однако при помощи добавки (патча, patch) к исходному коду СУБД MySQL длину ключа можно увеличить до 256 бит. В MySQL выбран 128-битный ключ, поскольку он работает намного быстрее и обеспечивает вполне достаточную защиту. Функции имеют следующий синтаксис:

```
AES_ENCRYPT(str,key_str)
AES_DECRYPT(crypt_str,key_str)
```

Функция `AES_ENCRYPT()` принимает в качестве аргумента `str` строку, которую необходимо подвергнуть шифрованию, и секретный ключ `key_str`. Входные аргументы

могут быть любой длины. Если один из аргументов равен NULL, то результат этой функции также будет иметь значение NULL.

Функция AES_DECRYPT() принимает в качестве первого аргумента зашифрованную при помощи AES_ENCRYPT() строку. Ключ key_str при этом должен совпадать как в первой, так и во второй строках. Если функция AES_DECRYPT() обнаруживает некорректные данные или некорректное заполнение строки, должно возвращаться значение NULL. Однако AES_DECRYPT() вполне может вернуть величину, отличную от NULL, или, возможно, просто "мусор".

Замечание

Функции AES_ENCRYPT() и AES_DECRYPT() были добавлены в версию 4.0.2 и могут рассматриваться как наиболее криптографически защищенные шифрующие функции, доступные в настоящее время в СУБД MySQL.

В листинге 16.1 приводится пример использования функций AES_ENCRYPT() и AES_DECRYPT().

Листинг 16.1. Использование функций AES_ENCRYPT() и AES_DECRYPT()

```
mysql> SELECT AES_ENCRYPT('MySQL','секретный ключ');
+--------------------------------------+
| AES_ENCRYPT('MySQL','секретный ключ') |
+--------------------------------------+
| ...ЗАьZЭj9к•-еП™Мh                    |
+--------------------------------------+
mysql> SELECT AES_DECRYPT('...ЗАьZЭj9к•-еП™Мh','секретный ключ');
+-------------------------------------------------+
| AES_DECRYPT('...ЗАьZЭj9к•-еП™Мh','секретный ключ') |
+-------------------------------------------------+
| MySQL                                           |
+-------------------------------------------------+
```

В качестве секретного ключа следует использовать фразу, которую сложно подобрать при помощи перебора или словаря, желательно, чтобы фраза была длинной, запоминающейся, и в то же время состояла из несуществующих в словарях слов, например "бызэ даннух".

Рассмотрим шифрование поля email в таблице users учебной базы данных shop. Шифрование данного поля позволит уберечь пользователей от спамеров в случае, если база данных попадет им в руки. Так как поля таблицы уже созданы и существуют, воспользуемся для шифрования данных этих полей оператором UPDATE (листинг 16.2).

Листинг 16.2. Шифрование e-mail

```
mysql> SELECT email FROM users;
```

```
+-----------------------+
| email                 |
+-----------------------+
| ivanov@email.ru       |
| losev@email.ru        |
| simdyanov@softtime.ru |
| kuznetsov@softtime.ru |
| NULL                  |
| korneev@domen.ru      |
+-----------------------+
mysql> UPDATE users SET email = AES_ENCRYPT(email,'бызэ даннух');
mysql> SELECT email FROM users;
+----------------------------------+
| email                            |
+----------------------------------+
| □хтЫl—lЬ}'.r®™Ц                  |
| №.LЙъУЕ□ŕ¦ФUIЪ«н                 |
| ЂIgПуLA □2W□TI,,jeE:гУ@ддS
пСгц· |
| =-"рО26ецЩгу□jeE:гУ@ддS
пСгц· |
| NULL                             |
V1' |ЂчЎЧ_ПLЌЕЩ•БМ...ъtb©л»пж¬□я
+----------------------------------+
```

Как видно из листинга 16.2, прямой просмотр поля `email` приводит к выводу "мусора". Для того чтобы узнать настоящие адреса `email`, необходимо воспользоваться функцией дешифровки `AES_DECRYPT()` и знать секретный ключ.

Листинг 16.3. Дешифровка данных

```
mysql> SELECT AES_DECRYPT(email,'бызэ даннух') FROM users;
+----------------------------------+
| AES_DECRYPT(email,'бызэ даннух') |
+----------------------------------+
| ivanov@email.ru                  |
| losev@email.ru                   |
| simdyanov@softtime.ru            |
| kuznetsov@softtime.ru            |
| NULL                             |
| korneev@domen.ru                 |
+----------------------------------+
```

16.2. Функции *ENCODE* и *DECODE*

Функции `ENCODE()` и `DECODE()` имеют синтаксис, аналогичный рассмотренным выше функциям `AES_ENCRYPT()` и `AES_DECRYPT()`.

```
ENCODE(str,pass_str)
DECODE(crypt_str,pass_str)
```

Функция ENCODE() шифрует строку str, используя аргумент pass_str как секретный ключ. Для расшифровки результата следует использовать функцию DECODE(), которая принимает в качестве параметра crypt_str зашифрованную строку и возвращает исходную. Результирующая строка, которую возвращает функция ENCODE(), представляет собой двоичную строку той же длины, что и str. Для хранения результата в столбце следует использовать столбец типа BLOB — это позволит не повредить строку кодировкой. Пример использования функций ENCODE() и DECODE() приведен в листинге 16.4.

Листинг 16.4. Использование функций ENCODE() и DECODE()

```
mysql> SELECT DECODE(ENCODE('MySQL','бызэ даннух'),'бызэ даннух');
+-----------------------------------------------------+
| DECODE(ENCODE('MySQL','бызэ даннух'),'бызэ даннух') |
+-----------------------------------------------------+
| MySQL                                               |
+-----------------------------------------------------+
```

16.3. Функции *DES_ENCRYPT* и *DES_DECRYPT*

Функции DES_ENCRYPT() и DES_DECRYPT() (листинг 16.5) осуществляют симметричное шифрование и дешифрование данных при помощи алгоритма 3DES (Triple-DES) и имеют следующий синтаксис:

```
DES_ENCRYPT(string_to_encrypt [, (key_number | key_string) ])
DES_DECRYPT(string_to_decrypt [, key_string])
```

Функция DES_ENCRYPT() в качестве первого параметра принимает строку string_to_encrypt, которая подвергается шифрованию. В качестве второго необязательного параметра может выступать строка key_string, задающая секретный ключ. В случае использования секретного ключа необходимо приводить его в качестве второго параметра и в функции дешифровки DES_DECRYPT(). Вместо секретного ключа можно указать номер key_number, принимающий значения от 0 до 9. Номер указывает на запись в ключевом DES-файле сервера, местоположение которого можно указать при старте сервера MySQL в параметре --des-key-file. Важно отметить, что повторная передача номера при использовании функции DES_DECRYPT() уже не требуется, т. к. он прописывается в зашифрованную строку. Такой подход, когда секретный ключ хранится на сервере и не передается через сетевое соединение, значительно безопаснее, т. к. значение ключа невозможно извлечь из сетевого трафика. Если второй параметр не указывается, то предполагается, что используется первая строка DES-файла.

Замечание

Если в MySQL не включена поддержка протокола SSL, функции `DES_ENCRYPT()` и `DES_DECRYPT()` возвращают `NULL`.

Листинг 16.5. Использование функций `DES_ENCRYPT()` и `DES_DECRYPT()`

```
mysql> SELECT DES_DECRYPT(DES_ENCRYPT('MySQL'));
+-----------------------------------+
| DES_DECRYPT(DES_ENCRYPT('MySQL')) |
+-----------------------------------+
| MySQL                             |
+-----------------------------------+
```

16.4. Функция *ENCRYPT*

Функция `ENCRYPT()` подвергает данные необратимому шифрованию, используя вызов системной функции `crypt() UNIX`, и имеет следующий синтаксис:

```
ENCRYPT(str [, salt])
```

Строка `str` подвергается необратимому шифрованию, т. е. функции, расшифровывающей зашифрованные данные, уже нет. Если второй необязательный параметр `salt` не указывается, то результат каждый раз получается новым (листинг 16.6). Если в качестве второго параметра передается определенная строка, то результат будет воспроизводиться при каждом вызове функции (листинг 16.7).

Листинг 16.6. Использование функции `ENCRYPT()`

```
mysql> SELECT ENCRYPT('MySQL'), ENCRYPT('MySQL');
+------------------+------------------+
| ENCRYPT('MySQL') | ENCRYPT('MySQL') |
+------------------+------------------+
| qnh1AW/pwGY2A    | snshG/AgkhKoU    |
+------------------+------------------+
```

Замечание

Если системный вызов `crypt()` отсутствует в системе, например, при использовании в качестве операционной системы Windows, функции `ENCRYPT()` всегда возвращается `NULL`.

Листинг 16.7. Использование функции `ENCRYPT()` совместно с параметром `salt`

```
mysql> SELECT ENCRYPT('MySQL','password'), ENCRYPT('MySQL','password');
+-----------------------------+-----------------------------+
| ENCRYPT('MySQL','password') | ENCRYPT('MySQL','password') |
+-----------------------------+-----------------------------+
| pa6Fh2dhjP10c               | pa6Fh2dhjP10c               |
+-----------------------------+-----------------------------+
```

В листинге 16.7, так же как и в листинге 16.6, необратимому шифрованию подвергается строка 'MySQL', только на этот раз в качестве параметра salt передается секретный ключ 'password'. Как видно из листинга, это приводит к тому, что результат, возвращаемый функцией ENCRYPT(), воспроизводится при повторном вызове функции.

16.5. Функция *MD5*

Функция MD5() (листинг 16.8) осуществляет необратимое шифрование данных по алгоритму MD5 (Message-Digest Algorithm) и имеет следующий синтаксис:

```
MD5(str)
```

Функция принимает строковый параметр str и возвращает 128-битную контрольную сумму, вычисленную по алгоритму MD5. Возвращаемая величина представляет собой 32-разрядное шестнадцатеричное число, которое уникально для строки, т. е. для строк, отличающихся хотя бы одним символом, результат функции MD5() будет отличаться. В то же время, для двух одинаковых строк всегда возвращается одинаковый результат.

Листинг 16.8. Использование функции MD5()

```
mysql> SELECT MD5('MySQL'), MD5('MySQL');
+----------------------------------+----------------------------------+
| MD5('MySQL')                     | MD5('MySQL')                     |
+----------------------------------+----------------------------------+
| 62a004b95946bb97541afa471dcca73a | 62a004b95946bb97541afa471dcca73a |
+----------------------------------+----------------------------------+
mysql> SELECT MD5('MySQL1'), MD5('MySQL');
+----------------------------------+----------------------------------+
| MD5('MySQL1')                    | MD5('MySQL')                     |
+----------------------------------+----------------------------------+
| fc3dd4ddcb132de1f9552818344e1b09 | 62a004b95946bb97541afa471dcca73a |
+----------------------------------+----------------------------------+
```

Замечание

Алгоритм MD5 также часто используется для создания уникального хэш-кода объемных файлов, которые передаются по сети. Загрузив файл, всегда можно проверить его целостность, вычислив код по алгоритму MD5 и сравнив полученный результат с хэш-кодом, предоставляемым распространителем. Это позволяет отследить повреждения файла, вызванные передачей через сеть, а также предотвращает фальсификацию дистрибутива.

16.6. Функция *PASSWORD*

Функция PASSWORD() (листинг 16.9) подвергает необратимому шифрованию данные str и имеет следующий синтаксис:

```
PASSWORD(str)
```

Именно эта функция используется для шифрования паролей MySQL в столбце `password` в таблице привилегий `user` системной базы данных `mysql`.

Листинг 16.9. Использование функции `PASSWORD()`

```
mysql> SELECT PASSWORD('MySQL');
+-------------------------------------------+
| PASSWORD('MySQL')                         |
+-------------------------------------------+
| *1799AB5202FE2E9958365F9B3ECBBF53657254C7 |
+-------------------------------------------+
```

До версии MySQL 4.1 использовался менее защищенный алгоритм шифрования, результат работы которого представлен в листинге 16.10.

Листинг 16.10. Результат работы функции `PASSWORD()` в старых версиях MySQL

```
mysql> SELECT PASSWORD('MySQL');
+-------------------+
| PASSWORD('MySQL') |
+-------------------+
| 7c651ebc23eebf89  |
+-------------------+
```

Так как в настоящее время используется большое число баз данных, поля в которых зашифрованы при помощи старой версии функции `PASSWORD()`, в MySQL 4.1 была введена функция `OLD_PASSWORD()`, которая эмулирует работу старой версии (листинг 16.11).

Листинг 16.11. Использование функции `OLD_PASSWORD()`

```
mysql> SELECT PASSWORD('MySQL'), OLD_PASSWORD('MySQL');
+-------------------------------------------+-----------------------+
| PASSWORD('MySQL')                         | OLD_PASSWORD('MySQL') |
+-------------------------------------------+-----------------------+
| *1799AB5202FE2E9958365F9B3ECBBF53657254C7 | 7c651ebc23eebf89      |
+-------------------------------------------+-----------------------+
```

Функция `PASSWORD()` используется в системе аутентификации в сервер MySQL, поэтому ее не следует использовать в приложениях — лучше воспользоваться функциями `MD5()` и `SHA1()`.

16.7. Функция *SHA1*

Функция `SHA1()` (листинг 16.12) вычисляет 160-битную контрольную сумму по алгоритму SHA1 (Secure Hash Algorithm) для строки `str` и имеет следующий синтаксис:

```
SHA1(str)
```

Возвращаемая величина представляет собой 40-разрядное шестнадцатеричное число или NULL (в том случае, если входной аргумент был равен NULL).

Листинг 16.12. Использование функции SHA1()

```
mysql> SELECT SHA('MySQL');
+------------------------------------------+
| SHA('MySQL')                             |
+------------------------------------------+
| deaa0c393a6613972aaccbf1fecfdad67aa21e88 |
+------------------------------------------+
```

Замечание

Для функции SHA1() имеется синоним SHA().

Так же, как и в случае алгоритма MD5, получаемые строки являются уникальными для каждой строки.

ГЛАВА 17

Поиск и регулярные выражения

Одной из сильных сторон баз данных по сравнению с обычными и плоскими файлами, являющимися примитивными базами данных, являются широкие возможности поиска. СУБД MySQL предоставляет многочисленные инструменты для осуществления поиска, среди которых оператор `LIKE`, осуществляющий простейшие операции поиска, оператор `RLIKE`, предоставляющий возможности поиска по регулярным выражениям, и механизм полнотекстового поиска, который подробно рассматривается в *главе 18*. В данной главе рассматривается поиск с использованием операторов `LIKE` и `RLIKE`.

17.1. Оператор *LIKE*

Оператор `LIKE` предназначен для сравнения строк с использованием простейших регулярных выражений и имеет следующий синтаксис:

```
expr LIKE pat
```

Оператор часто используется в конструкции `WHERE` и возвращает 1, если выражение `expr` соответствует выражению `pat`, и 0 в противном случае.

Листинг 17.1. Использование оператора `LIKE`

```
mysql> SELECT 'MySQL' LIKE 'MySQL', 'MySQL' LIKE 'mysql';
+----------------------+----------------------+
| 'MySQL' LIKE 'MySQL' | 'MySQL' LIKE 'mysql' |
+----------------------+----------------------+
|                    1 |                    1 |
+----------------------+----------------------+
mysql> SELECT 'MySQL' LIKE 'MySQL1', 'MySQL' LIKE BINARY 'mysql';
+------------------------+-----------------------------+
| 'MySQL' LIKE 'MySQL1' | 'MySQL' LIKE BINARY 'mysql' |
+------------------------+-----------------------------+
|                      0 |                           0 |
+------------------------+-----------------------------+
```

Замечание

Важно помнить, что сравнение строк не зависит от регистра, если не используется ключевое слово `BINARY`, указывающее, что строку следует рассматривать как двоичную последовательность.

Замечание

Передача в качестве любого операнда значения `NULL` приводит к тому, что оператор `LIKE` возвращает `NULL`.

Рассмотренные в листинге 17.1 операции аналогичны оператору "=". Главное преимущество оператора `LIKE` перед оператором равенства заключается в использовании спецсимволов, приведенных в табл. 17.1.

Таблица 17.1. *Спецсимволы, используемые в операторе* `LIKE`

Символ	Описание
%	Соответствует любому количеству символов, даже их отсутствию
_	Соответствует ровно одному символу

При помощи спецсимволов, представленных в табл. 17.1, можно задать различные шаблоны соответствия.

Листинг 17.2. Использование спецсимвола "%"

```
mysql> SELECT 'Программист' LIKE 'Программ%';
+--------------------------------+
| 'Программист' LIKE 'Программ%' |
+--------------------------------+
|                              1 |
+--------------------------------+
mysql> SELECT 'Программа' LIKE 'Программ%';
+------------------------------+
| 'Программа' LIKE 'Программ%' |
+------------------------------+
|                            1 |
+------------------------------+
mysql> SELECT 'Программ' LIKE 'Программ%';
+----------------------------+
| 'Программ' LIKE 'Программ%' |
+----------------------------+
|                          1 |
+----------------------------+
```

Как видно из листинга 17.2, спецсимвол "%" заменяет собой любую последовательность символов, в том числе и пустую строку, поэтому шаблон `'Программ%'` удовлетво-

ряет словам 'Программист', 'Программа' и 'Программ' и будет удовлетворять любому слову, начинающемуся с выражения 'Программ'.

Спецсимвол "%" может быть размещен в любом месте шаблона, как в начале, так и в середине слова (листинг 17.3).

Листинг 17.3. Использование спецсимвола "%" в начале и середине шаблона

```
mysql> SELECT 'Программирование' LIKE 'П%е','Печенье' LIKE 'П%е';
+-------------------------------+----------------------+
| 'Программирование' LIKE 'П%е' | 'Печенье' LIKE 'П%е' |
+-------------------------------+----------------------+
|                             1 |                    1 |
+-------------------------------+----------------------+
mysql> SELECT 'Программирование' LIKE '%ние','Кодирование' LIKE '%ние';
+--------------------------------+---------------------------+
| 'Программирование' LIKE '%ние' | 'Кодирование' LIKE '%ние' |
+--------------------------------+---------------------------+
|                              1 |                         1 |
+--------------------------------+---------------------------+
```

Другой спецсимвол "_" соответствует одному любому символу. Так, шаблон из трех знаков подчеркивания '___' будет соответствовать любому слову, состоящему из трех букв 'код', 'рот', 'абв'.

Листинг 17.4. Использование спецсимвола "_"

```
mysql> SELECT 'код' LIKE '___','рот' LIKE '___', 'абв' LIKE '___';
+------------------+------------------+------------------+
| 'код' LIKE '___' | 'рот' LIKE '___' | 'абв' LIKE '___' |
+------------------+------------------+------------------+
|                1 |                1 |                1 |
+------------------+------------------+------------------+
```

Ограничений на позицию спецсимвола "_", так же как и в случае "%", нет, можно располагать его в любой части шаблона. Пример использования символа "_" приведен в листингах 17.4 и 17.5.

Листинг 17.5. Использование спецсимвола "_" в разных частях шаблона

```
mysql> SELECT 'код' LIKE '_од', 'код' LIKE 'к_д', 'код' LIKE 'ко_';
+------------------+------------------+------------------+
| 'код' LIKE '_од' | 'код' LIKE 'к_д' | 'код' LIKE 'ко_' |
+------------------+------------------+------------------+
|                1 |                1 |                1 |
+------------------+------------------+------------------+
```

Для того чтобы поместить в шаблон сами символы "%" и "_" без их специальной интерпретации, необходимо экранировать их при помощи обратного слэша, как это продемонстрировано в листинге 17.6.

Листинг 17.6. Поиск символов "%" и "_"

```
mysql> SELECT '15 %' LIKE '15 \%', 'my_sql' LIKE 'my\_sql';
+---------------------+-------------------------+
| '15 %' LIKE '15 \%' | 'my_sql' LIKE 'my\_sql' |
+---------------------+-------------------------+
|                   1 |                       1 |
+---------------------+-------------------------+
mysql> SELECT '15' LIKE '15 \%', 'my sql' LIKE 'my\_sql';
+-------------------+-------------------------+
| '15' LIKE '15 \%' | 'my sql' LIKE 'my\_sql' |
+-------------------+-------------------------+
|                 0 |                       0 |
+-------------------+-------------------------+
```

В заключение данного раздела рассмотрим несколько примеров к учебной базе данных `shop`. Первый запрос, представленный в листинге 17.7, извлекает из таблицы `products` записи, в которых имеется слово '`Celeron`'.

Листинг 17.7. Извлечение из таблицы `products` записей, относящихся к '`Celeron`'

```
mysql> SELECT name FROM products WHERE name LIKE 'Celeron%';
+-----------------------+
| name                  |
+-----------------------+
| Celeron 1.8           |
| Celeron 2.0GHz        |
| Celeron 2.4GHz        |
| Celeron D 320 2.4GHz  |
| Celeron D 325 2.53GHz |
| Celeron D 315 2.26GHz |
+-----------------------+
```

Следующий запрос, который представлен в листинге 17.8, извлекает из таблицы `products` записи, соответствующие товарным позициям, цена которых находится между 1000 и 2000 рублей.

Замечание

Для оперирования цифрами лучше использовать математические операторы, рассмотренные в *главе 13*, т. к. в этом случае действия с числами выполняются быстрее. Кроме того, СУБД не будет тратить время на преобразование типа.

Листинг 17.8. Извлечение из таблицы `products` записей

```
mysql> SELECT name, price FROM products
    -> WHERE price LIKE '1___.00'
    -> ORDER BY price;
+------------------------------+---------+
| name                         | price   |
+------------------------------+---------+
| DDR-400 256MB Kingston       | 1085.00 |
| DDR-400 256MB Hynix Original | 1179.00 |
| Celeron 1.8                  | 1595.00 |
| ASUSTEK V9520X               | 1602.00 |
| DDR-400 512MB PQI            | 1690.00 |
| DDR-400 512MB Hynix          | 1717.00 |
| Celeron D 315 2.26GHz        | 1880.00 |
| Gigabyte GA-8I848P-RS        | 1896.00 |
| DDR-400 512MB Kingston       | 1932.00 |
| Celeron D 320 2.4GHz         | 1962.00 |
| Celeron 2.0GHz               | 1969.00 |
+------------------------------+---------+
```

17.2. Оператор *NOT LIKE*

Оператор `NOT LIKE`, показанный в листинге 17.9, противоположен по действию оператору `LIKE`, и имеет следующий синтаксис:

```
expr NOT LIKE pat
```

Оператор возвращает 0, если выражение `expr` соответствует выражению `pat`, и 1 в противном случае. То есть с его помощью можно извлечь записи, которые не удовлетворяют указанному условию.

> ***Замечание***
>
> Оператор `NOT LIKE` можно представить в виде `NOT (expr LIKE pat)`.

Возвращаясь к учебной базе данных `shop`, извлечем из таблицы `products` записи, не вошедшие в список, который вернул запрос из листинга 17.8.

Листинг 17.9. Использование оператора `NOT LIKE`

```
mysql> SELECT name, price FROM products
    -> WHERE price NOT LIKE '1___.00'
    -> ORDER BY price;
+-----------------------------+---------+
| name                        | price   |
+-----------------------------+---------+
| DDR-400 256MB PQI           |  899.00 |
| Samsung SP0812C             | 2093.00 |
```

```
| Celeron 2.4GHz               | 2109.00 |
| Gigabyte GA-8IPE1000G        | 2289.00 |
| Epox EP-4PDA3I               | 2289.00 |
| Gigabyte GA-8IG1000          | 2420.00 |
| Maxtor 6Y120P0               | 2456.00 |
| Seagate ST3120026A           | 2468.00 |
| Asustek P4P800-VM\L i865G    | 2518.00 |
| SAPPHIRE 256MB RADEON 9550   | 2730.00 |
| Celeron D 325 2.53GHz        | 2747.00 |
| Seagate Barracuda ST3160023A | 3139.00 |
| Maxtor 6B200P0               | 3589.00 |
| ASUSTEK A9600XT/TD           | 5156.00 |
| Asustek P4C800-E Delux       | 5395.00 |
| Intel Pentium 4 3.0GHz       | 5673.00 |
| GIGABYTE AGP GV-N59X128D     | 5886.00 |
| Intel Pentium 4 3.0GHz       | 6147.00 |
| Intel Pentium 4 3.2GHz       | 7259.00 |
+------------------------------+---------+
```

17.3. Оператор *SOUND LIKE*

Оператор `SOUNDS LIKE` аналогичен `SOUNDEX(expr1) = SOUNDEX(expr2)` и имеет следующий синтаксис:

```
expr1 SOUNDS LIKE expr2
```

Оператор позволяет сравнивать слова `expr1` и `expr2` по звучанию и возвращает 1, если слова звучат одинаково, и 0 в противном случае.

Замечание

Оператор `SOUND LIKE` появился в СУБД MySQL, начиная с версии 4.1.

17.4. Оператор *RLIKE (REGEXP)*

Оператор `RLIKE` (или его синоним `REGEXP`) позволяет производить поиск в соответствии с регулярными выражениями, которые предоставляют значительно более гибкие средства для поиска по сравнению с оператором `LIKE`. Обратной стороной медали является более медленное выполнение операции поиска с использованием регулярных выражений по сравнению с оператором `LIKE`.

Регулярные выражения — это специализированный язык поиска и осуществления манипуляций с подстроками в тексте. Для работы с регулярными выражениями в СУБД MySQL предназначены конструкции `RLIKE` и `REGEXP`, которые являются синонимами и имеют следующий синтаксис:

```
expr REGEXP pat
expr RLIKE pat
```

Данные операторы часто используются в выражении WHERE и возвращают 1, если выражение expr соответствует выражению pat, и 0 в противном случае.

Замечание

В настоящий момент существует несколько диалектов регулярных выражений. В СУБД MySQL реализована расширенная версия предложенной Генри Спенсером реализации регулярных выражений, которая ориентирована на соответствие стандарту POSIX.

Регулярное выражение можно рассматривать как шаблон, применяемый к заданному тексту слева направо. Большая часть символов сохраняет свое значение в шаблоне и означает совпадение с соответствующим символом. Так, регулярное выражение, содержащее обычный текст, например 'грам', соответствует строке, содержащей такую подстроку ('грам'). Например, этому регулярному выражению будут соответствовать строки 'программирование', 'грамм', 'грампластинка' и т. п.

Листинг 17.10. Использование оператора RLIKE

```
mysql> SELECT 'грамм' RLIKE 'грам', 'грампластинка' RLIKE 'грам';
+----------------------+-----------------------------+
| 'грамм' RLIKE 'грам' | 'грампластинка' RLIKE 'грам' |
+----------------------+-----------------------------+
|                    1 |                           1 |
+----------------------+-----------------------------+
mysql> SELECT 'программирование' RLIKE 'грам';
+---------------------------------+
| 'программирование' RLIKE 'грам' |
+---------------------------------+
|                               1 |
+---------------------------------+
```

Как видно из листинга 17.10, регулярное выражение 'грам' осуществляет поиск по всему тексту, независимо от того, находится ли подстрока 'грам' в начале, середине или конце слова.

Часто необходимо привязать регулярное выражение к началу слова, т. е. чтобы регулярное выражение 'грам' соответствовало строке, начинающемуся с подстроки 'грам', например, 'грампластинка', но не соответствовало слову 'программирование'. Для этого используется символ, "^", соответствующий началу строки (листинг 17.11).

Листинг 17.11. Использование спецсимвола "^"

```
mysql> SELECT 'программирование' RLIKE '^грам',
    -> 'грампластинка' RLIKE '^грам';
+----------------------------------+-----------------------------+
| 'программирование' RLIKE '^грам' | 'грампластинка' RLIKE '^грам' |
+----------------------------------+-----------------------------+
|                                0 |                           1 |
+----------------------------------+-----------------------------+
```

Спецсимвол "$" позволяет привязать регулярное выражение к концу строки (листинг 17.12), т. е. применение символов "^" и "$" позволяет указать, что регулярное выражение должно в точности соответствовать всей строке поиска от начала до конца. Регулярное выражение '^грампластинка$' соответствует строке 'грампластинка', но не соответствует строке 'грампластинка — это вам не программирование', т. к. после искомого слова идет текст.

Листинг 17.12. Использование спецсимвола "$"

```
mysql> SELECT 'грампластинка' RLIKE '^грампластинка$';
+----------------------------------------+
| 'грампластинка' RLIKE '^грампластинка$' |
+----------------------------------------+
|                                      1 |
+----------------------------------------+
mysql> SELECT 'грампластинка - это вам не программирование'
    -> RLIKE '^грампластинка$';
+-----------------------------------------------------------------------+
| 'грампластинка - это вам не программирование' RLIKE '^грампластинка$' |
+-----------------------------------------------------------------------+
|                                                                     0 |
+-----------------------------------------------------------------------+
```

Замечание

Регулярное выражение '^$' соответствует пустой строке.

Возвращаясь к учебной базе данных shop, извлечем из таблицы products записи, поля name у которых заканчиваются подстрокой 'HGz' (листинг 17.13).

Листинг 17.13. Извлечение всех записей, содержащих подстроку 'HGz'

```
mysql> SELECT name FROM products WHERE name RLIKE 'GHz$';
+------------------------+
| name                   |
+------------------------+
| Celeron 2.0GHz         |
| Celeron 2.4GHz         |
| Celeron D 320 2.4GHz   |
| Celeron D 325 2.53GHz  |
| Celeron D 315 2.26GHz  |
| Intel Pentium 4 3.2GHz |
| Intel Pentium 4 3.0GHz |
| Intel Pentium 4 3.0GHz |
+------------------------+
```

Символы "^" и "$" соответствуют началу и концу строки, но гораздо чаще встает задача привязки регулярного выражения к началу и концу слова. Для этого предназна-

чены последовательности `[[:<:]]` и `[[:>:]]`, соответствующие началу и концу слова (листинг 17.14).

Листинг 17.14. Использование последовательностей `[[:<:]]` и `[[:>:]]`

```
mysql> SELECT 'a word a' REGEXP '[[:<:]]word[[:>:]]';
+---------------------------------------+
| 'a word a' REGEXP '[[:<:]]word[[:>:]]' |
+---------------------------------------+
|                                     1 |
+---------------------------------------+
mysql> SELECT 'a xword a' REGEXP '[[:<:]]word[[:>:]]';
+----------------------------------------+
| 'a xword a' REGEXP '[[:<:]]word[[:>:]]' |
+----------------------------------------+
|                                      0 |
+----------------------------------------+
```

Символ вертикальной черты "|" (листинг 17.15) применяется в регулярном выражении для задания альтернативных масок:

```
'abc|абв'
```

Этому регулярному выражению соответствует любая строка, содержащая подстроки 'abc' или 'абв'.

Листинг 17.15. Использование спецсимвола "|"

```
mysql> SELECT 'abc' RLIKE 'abc|абв', 'абв' RLIKE 'abc|абв';
+----------------------+----------------------+
| 'abc' RLIKE 'abc|абв' | 'абв' RLIKE 'abc|абв' |
+----------------------+----------------------+
|                    1 |                    1 |
+----------------------+----------------------+
```

Спецсимвол "|" можно интерпретировать как оператор ИЛИ, при его использовании удовлетворяющими шаблону признаются записи, соответствующие либо правой, либо левой частям.

Если шаблон должен включать символ "|" или любой другой специальный символ, например, рассмотренные выше "^" и "$", то их необходимо экранировать при помощи двойного обратного слэша \\ — в этом случае они теряют свое специальное значение и рассматриваются как обычные символы.

Возвращаясь к нашей учебной базе данных `shop`, можно привести следующий пример. Пусть необходимо извлечь из таблицы `products` записи, поле `name` которых содержит в своем составе подстроки '`Pentium`' или '`Celeron`'. В этом случае запрос может выглядеть так, как это представлено в листинге 17.16.

Листинг 17.16. Извлечение всех записей, содержащих подстроки 'Pentium' и 'Celeron'

```
mysql> SELECT name FROM products WHERE name RLIKE ' Pentium|Celeron';
+------------------------+
| name                   |
+------------------------+
| Celeron 1.8            |
| Celeron 2.0GHz         |
| Celeron 2.4GHz         |
| Celeron D 320 2.4GHz   |
| Celeron D 325 2.53GHz  |
| Celeron D 315 2.26GHz  |
| Intel Pentium 4 3.2GHz |
| Intel Pentium 4 3.0GHz |
| Intel Pentium 4 3.0GHz |
+------------------------+
```

Выражение 'Intel Pentium|Celeron' будет искать строки, в состав которых входит подстрока 'Intel Pentium' или 'Celeron'. Для того чтобы производился поиск строк 'Intel Pentium' или 'Intel Celeron', необходимо прибегнуть к круглым скобкам:

'Intel (Pentium|Celeron)'

Для задания класса символов используются квадратные скобки (листинг 17.17). Квадратные скобки ограничивают поиск теми символами, которые в них заключены, например, '[abc]'.

Листинг 17.17. Использование класса символов

```
mysql> SELECT 'a' RLIKE '[abc]' AS a,
    -> 'b' RLIKE '[abc]' AS b,
    -> 'c' RLIKE '[abc]' AS c;
+---+---+---+
| a | b | c |
+---+---+---+
| 1 | 1 | 1 |
+---+---+---+
```

Регулярному выражению '[abc]' соответствует подстрока, содержащая один символ: либо a, либо b, либо c.

Так, для создания регулярного выражения, соответствующего всем буквам русского алфавита, можно, конечно, перечислить все буквы в квадратных скобках. Это допустимо, но утомительно и неэлегантно. Более кратко такое регулярное выражение можно записать следующим образом:

'[а-я]'

Это выражение соответствует всем буквам русского алфавита, поскольку любые два символа, разделяемые дефисом, задают соответствие диапазону символов, находящихся между ними.

> ***Замечание***
>
> Строки в СУБД MySQL не зависят от регистра, в том числе и в регулярных выражениях. Для того чтобы СУБД MySQL интерпретировала строки как регистрозависимые, их необходимо предварять ключевым словом `BINARY`.

Точно таким же образом задается регулярное выражение, соответствующее любому числу:

```
'[0-9]'
```

Это выражение эквивалентно

```
'[0123456789]'
```

В листинге 17.18 приводится пример сравнения русской буквы 'Л' и английской 'z' с регулярным выражением '[а-я]'.

Листинг 17.18. Использование диапазонов в классах символов

```
mysql> SELECT 'Л' RLIKE '[а-я]' AS a;
+---+
| a |
+---+
| 1 |
+---+
mysql> SELECT 'z' RLIKE '[а-я]' AS a;
+---+
| a |
+---+
| 0 |
+---+
```

Выражение вида

```
'[а-я0-9 ]'
```

соответствует либо букве русского алфавита, либо цифре, либо пробелу.

Если в начале класса помещается символ "^", то смысл выражения инвертируется (листинг 17.19). Так, выражение

```
'[^0-9]'
```

соответствует любому символу, кроме цифры.

Листинг 17.19. Отрицание в классах символов

```
mysql> SELECT '7' RLIKE '[^0-9]' AS number, 'a' RLIKE '[^0-9]' AS str;
+--------+-----+
| number | str |
+--------+-----+
|      0 |   1 |
+--------+-----+
```

Для определения специальных последовательностей внутри строк в СУБД MySQL используется С-нотация, в которой экранирование обычных символов приводит к их специальной интерпретации:

- `\t` — символ табуляции;
- `\f` — конец файла;
- `\n` — символ перевода строки;
- `\r` — символ возврата каретки;
- `\\` — символ обратного слэша `\`.

Кроме классов, которые могут создать разработчики, путем комбинирования отдельных символов и их диапазонов в стандарте POSIX предусмотрены специальные конструкции классов, представленные в табл. 17.2.

Таблица 17.2. *Классы символов POSIX регулярных выражений*

Класс	Описание
`[:alnum:]`	Алфавитно-цифровые символы
`[:alpha:]`	Алфавитные символы
`[:blank:]`	Символы пробела или табуляции
`[:cntrl:]`	Управляющие символы
`[:digit:]`	Десятичные цифры (0—9)
`[:graph:]`	Графические (видимые) символы
`[:lower:]`	Символы алфавита в нижнем регистре
`[:print:]`	Графические или невидимые символы
`[:punct:]`	Знаки препинания
`[:space:]`	Символы пробела, табуляции, новой строки или возврата каретки
`[:upper:]`	Символы алфавита в верхнем регистре
`[:xdigit:]`	Шестнадцатеричные цифры

Классы в табл. 17.2 определяют диапазоны символов, так, например, регулярное выражение `'[0-9]'` эквивалентно `'[[:digit:]]'`. В листинге 17.20 демонстрируется при-

менение классов символов `[:digit:]`, `[:alpha:]` и `[:alnum:]` с цифрой 1 и символом `'a'`. Важно отметить, что алфавитные символы удовлетворяют всем алфавитным символам, включая русские и английские.

Листинг 17.20. Использование классов символов

```
mysql> SELECT '1' RLIKE '[[:digit:]]', 'a' RLIKE '[[:digit:]]';
+-------------------------+-------------------------+
| '1' RLIKE '[[:digit:]]' | 'a' RLIKE '[[:digit:]]' |
+-------------------------+-------------------------+
|                       1 |                       0 |
+-------------------------+-------------------------+
mysql> SELECT '1' RLIKE '[[:alpha:]]', 'a' RLIKE '[[:alpha:]]';
+-------------------------+-------------------------+
| '1' RLIKE '[[:alpha:]]' | 'a' RLIKE '[[:alpha:]]' |
+-------------------------+-------------------------+
|                       0 |                       1 |
+-------------------------+-------------------------+
mysql> SELECT '1' RLIKE '[[:alnum:]]', 'a' RLIKE '[[:alnum:]]';
+-------------------------+-------------------------+
| '1' RLIKE '[[:alnum:]]' | 'a' RLIKE '[[:alnum:]]' |
+-------------------------+-------------------------+
|                       1 |                       1 |
+-------------------------+-------------------------+
```

Выражение в квадратных скобках соответствует только одному символу и часто применяется совместно с так называемыми квантификаторами. Это символы "?", "+" и "*", которые следуют сразу за символом и изменяют число вхождений этого символа в строку:

- ? — символ либо входит в строку один раз, либо вообще в нее не входит;
- * — любое число вхождений символа в строку, в том числе и 0;
- + — одно или более число вхождений символа в строку.

Символ "?" позволяет сократить выражения вида

`'^СУБД|СУБД MySQL$'`

до

`'^СУБД( MySQL)?$'`(см. листинг 17.21).

Листинг 17.21. Использование квантификатора "?"

```
mysql> SELECT 'СУБД MySQL' RLIKE '^СУБД( MySQL)?$',
    -> 'СУБД' RLIKE '^СУБД( MySQL)?$';
+-------------------------------------+-------------------------------+
| 'СУБД MySQL' RLIKE '^СУБД( MySQL)?$' | 'СУБД' RLIKE '^СУБД( MySQL)?$' |
+-------------------------------------+-------------------------------+
|                                   1 |                             1 |
+-------------------------------------+-------------------------------+
```

Метасимвол "+" обозначает один или несколько экземпляров элемента непосредственно предшествующего элемента. Так, если необходимо найти подстроку, содержащую одну или большее число цифр, можно воспользоваться выражением вида

```
'[[:digit:]]+'
```

В листинге 17.22 демонстрируется использование квантификатора "+", который удовлетворяет последовательности из одного и большего числа символов.

Листинг 17.22. Использование квантификатора "+"

```
mysql> SELECT '1' RLIKE '^[[:digit:]]+$',
    -> '453455234' RLIKE '^[[:digit:]]+$';
+---------------------------+-----------------------------------+
| '1' RLIKE '^[[:digit:]]+$' | '453455234' RLIKE '^[[:digit:]]+$' |
+---------------------------+-----------------------------------+
|                         1 |                                 1 |
+---------------------------+-----------------------------------+
mysql> SELECT '' RLIKE '^[[:digit:]]+$',
    -> '45.3455234' RLIKE '^[[:digit:]]+$';
+---------------------------+-----------------------------------+
| '' RLIKE '^[[:digit:]]+$' | '45.3455234' RLIKE '^[[:digit:]]+$' |
+---------------------------+-----------------------------------+
|                         0 |                                 0 |
+---------------------------+-----------------------------------+
```

Символ "*" используется для любого числа вхождений строки в подстроку, в том числе и нулевого, т. е. регулярное выражение

```
'^[[:digit:]]*$'
```

соответствует либо пустой строке, либо строке, содержащей только числа, причем их число не ограничено (листинг 17.23).

Листинг 17.23. Использование квантификатора "*"

```
mysql> SELECT '1' RLIKE '^[[:digit:]]*$',
    -> '453455234' RLIKE '^[[:digit:]]*$';
+---------------------------+-----------------------------------+
| '1' RLIKE '^[[:digit:]]*$' | '453455234' RLIKE '^[[:digit:]]*$' |
+---------------------------+-----------------------------------+
|                         1 |                                 1 |
+---------------------------+-----------------------------------+
mysql> SELECT '' RLIKE '^[[:digit:]]*$',
    -> '45.3455234' RLIKE '^[[:digit:]]*$';
+---------------------------+-----------------------------------+
| '' RLIKE '^[[:digit:]]*$' | '45.3455234' RLIKE '^[[:digit:]]*$' |
+---------------------------+-----------------------------------+
|                         1 |                                 0 |
+---------------------------+-----------------------------------+
```

Помимо круглых и квадратных скобок, в регулярных выражениях также применяются фигурные скобки. Они предназначены для указания числа или диапазона чисел повторения элемента:

- `'xy{2}'` — соответствует строке, в которой за `x` следуют два символа `y`, т. е. `xyy`;
- `'xy{2,}'` — соответствует строке, в которой за `x` следует не менее двух `y` (но может быть и больше);
- `'xy{2,6}'` — соответствует строке, в которой за `x` следует от двух до шести `y`.

Для указания количества вхождений не одного символа, а их последовательности, используются круглые скобки:

- `'x(yz){2,6}'` — соответствует строке, в которой за `x` следует от двух до шести последовательностей `yz`;
- `'x(yz)*'` — соответствует строке, в которой за `x` следует ноль и более последовательностей `yz`.

В листинге 17.24 демонстрируется регулярное выражение, удовлетворяющее числу формата ###.##. Целая часть может состоять из любого числа цифр, а дробная часть всегда состоит из двух.

Листинг 17.24. Поиск чисел формата ###.##

```
mysql> SELECT '123.90' RLIKE '^[[:digit:]]*\\.[[:digit:]]{2}$';
+-------------------------------------------------+
| '123.90' RLIKE '^[[:digit:]]*\\.[[:digit:]]{2}$' |
+-------------------------------------------------+
|                                               1 |
+-------------------------------------------------+
mysql> SELECT '123' RLIKE '^[[:digit:]]*\\.[[:digit:]]{2}$';
+----------------------------------------------+
| '123' RLIKE '^[[:digit:]]*\\.[[:digit:]]{2}$' |
+----------------------------------------------+
|                                            0 |
+----------------------------------------------+
```

Как видно из листинга 17.24, если числу '`123.90`' удовлетворяет представленное регулярное выражение, то обычное целое число '`123`' данным запросом отбрасывается. Для того чтобы регулярное выражение удовлетворяло сразу обоим числам, необходимо видоизменить запрос так, как это показано в листинге 17.25.

Листинг 17.25. Поиск чисел формата ### и ###.##

```
mysql> SELECT '123' RLIKE '^[[:digit:]]*(\\.[[:digit:]]{2})?$';
+-------------------------------------------------+
| '123' RLIKE '^[[:digit:]]*(\\.[[:digit:]]{2})?$' |
+-------------------------------------------------+
|                                               1 |
+-------------------------------------------------+
```

```
mysql> SELECT '123.90' RLIKE '^[[:digit:]]*(\\.[[:digit:]]{2})?$';
+-----------------------------------------------------+
| '123.90' RLIKE '^[[:digit:]]*(\\.[[:digit:]]{2})?$' |
+-----------------------------------------------------+
|                                                   1 |
+-----------------------------------------------------+
```

Как видно из листинга 17.25, элемент регулярного выражения, ответственного за дробную часть числа `\\.[[:digit:]]{2}`, группируется при помощи круглых скобок, за которыми следует спецсимвол "?", требующий вхождения подстроки 0 или 1 раз.

Следующий пример, приведенный в листинге 17.26, демонстрирует поиск строк в таблице `users` учебной базы данных `shop`, которые содержат e-mail вида someone@somwhere.dom.

Листинг 17.26. Поиск e-mail

```
mysql> SELECT email FROM users;
+-----------------------+
| email                 |
+-----------------------+
| ivanov@email.ru       |
| losev@email.ru        |
| simdyanov@softtime.ru |
| kuznetsov@softtime.ru |
| NULL                  |
| korneev@domen.ru      |
+-----------------------+
mysql> SELECT email FROM users
    -> WHERE email RLIKE '^[-0-9a-z_]+@[-0-9a-z_^\\.]+\\.[a-z]{2,3}$';
+-----------------------+
| email                 |
+-----------------------+
| ivanov@email.ru       |
| losev@email.ru        |
| simdyanov@softtime.ru |
| kuznetsov@softtime.ru |
| korneev@domen.ru      |
+-----------------------+
```

У адреса имеются две составляющие — имя пользователя и имя домена, которые разделены знаком `"@"`. В имени пользователя могут присутствовать буквы нижнего и верхнего регистров, цифры, знаки подчеркивания, минуса и точки. Для проверки разделителя между именем пользователя и именем домена в выражение требуется добавить `"@"`. Таким образом, регулярное выражение, проверяющее имя пользователя и наличие разделителя, имеет следующий вид:

```
'[-0-9a-z_]+@[-0-9a-z_^\\.]+"
```

Для проверки последнего доменного имени (.ru, .com) необходимо добавить следующее выражение:

```
'\\.[a-z]{2,3}'
```

Символ "." в регулярных выражениях используется для обозначения любого символа, поэтому для поиска соответствия точке в регулярном выражении этот символ экранируется: "\\.".

Объединяя эти строки и привязывая начало и конец регулярного выражения к началу и концу строки при помощи спецсимволов "^" и "$", получаем выражение, представленное в листинге 17.26.

Можно создавать достаточно сложные SQL-запросы, комбинируя несколько операторов `RLIKE` при помощи логических операторов `AND` и `OR`. Пусть стоит задача выбора процессора марки '`Celeron`', стоящего около 1900 рублей. Решение этой задачи может выглядеть так, как это представлено в листинге 17.27.

Листинг 17.27. Выбор процессоров марки '`Celeron`' стоимостью около 1900 рублей

```
mysql> SELECT name, price FROM products
    -> WHERE name RLIKE '^Celeron' AND
    ->       price RLIKE '^19[[:digit:]]{2}\\.[[:digit:]]{2}';
+----------------------+---------+
| name                 | price   |
+----------------------+---------+
| Celeron 2.0GHz       | 1969.00 |
| Celeron D 320 2.4GHz | 1962.00 |
+----------------------+---------+
```

17.5. Оператор *NOT RLIKE*

Оператор `NOT RLIKE` аналогичен по действию оператору `RLIKE` и имеет следующий синтаксис:

```
expr NOT RLIKE pat
```

Оператор возвращает 0, если выражение `expr` соответствует выражению `pat`, и 1 в противном случае. То есть с его помощью можно извлечь записи, которые не удовлетворяют заданному условию.

Замечание

Оператор `NOT RLIKE` имеет синоним `NOT REGEXP`.

Замечание

Оператор `NOT RLIKE` можно представить в виде `NOT (expr RLIKE pat)`.

Возвращаясь к учебной базе данных `shop`, извлечем из таблицы `products` записи, за исключением товарных позиций фирм '`Asustek`' и '`Gigabyte`' (листинг 17.28).

Листинг 17.28. Использование оператора NOT RLIKE

```
mysql> SELECT name, price FROM products
    -> WHERE name NOT RLIKE '^Asustek' AND name NOT RLIKE '^Gigabyte';
+------------------------------+---------+
| name                         | price   |
+------------------------------+---------+
| Celeron 1.8                  | 1595.00 |
| Celeron 2.0GHz               | 1969.00 |
| Celeron 2.4GHz               | 2109.00 |
| Celeron D 320 2.4GHz         | 1962.00 |
| Celeron D 325 2.53GHz        | 2747.00 |
| Celeron D 315 2.26GHz        | 1880.00 |
| Intel Pentium 4 3.2GHz       | 7259.00 |
| Intel Pentium 4 3.0GHz       | 6147.00 |
| Intel Pentium 4 3.0GHz       | 5673.00 |
| Epox EP-4PDA3I               | 2289.00 |
| SAPPHIRE 256MB RADEON 9550   | 2730.00 |
| Maxtor 6Y120P0               | 2456.00 |
| Maxtor 6B200P0               | 3589.00 |
| Samsung SP0812C              | 2093.00 |
| Seagate Barracuda ST3160023A | 3139.00 |
| Seagate ST3120026A           | 2468.00 |
| DDR-400 256MB Kingston       | 1085.00 |
| DDR-400 256MB Hynix Original | 1179.00 |
| DDR-400 256MB PQI            |  899.00 |
| DDR-400 512MB Kingston       | 1932.00 |
| DDR-400 512MB PQI            | 1690.00 |
| DDR-400 512MB Hynix          | 1717.00 |
+------------------------------+---------+
```

17.6. Функция *STRCMP*

Функция `STRCMP()` предназначена для сравнения двух строк и имеет следующий синтаксис:

```
STRCMP(expr1,expr2)
```

Функция возвращает 0, если строки `expr1` и `expr2` идентичны, -1 — если `expr1` больше `expr2`, и 1 — если `expr1` меньше `expr2`.

Замечание

Начиная с версии MySQL 4.0, для сравнения строк используется текущая кодировка, тогда как до версии 4.0 строки сравнивались как двоичные последовательности. То есть в текущих версиях сравнение не зависит от регистра, а в более ранних — зависит.

ГЛАВА 18

Полнотекстовый поиск

Помимо рассмотренного в предыдущей главе поиска по регулярным выражениям, в СУБД MySQL предусмотрен так называемый режим полнотекстового поиска, который не требует применения шаблонов. Данный режим предоставляет широкие возможности поиска в тексте и выполняется гораздо быстрее поиска с использованием регулярных выражений благодаря специальному индексу `FULLTEXT`. Следует помнить, что индексация столбца, в том числе и индексом `FULLTEXT`, требует дополнительного объема памяти для хранения индексов, иногда превышающего объем основных данных в несколько раз, и приводит к замедлению операций вставки и удаления при помощи операторов `INSERT` и `DELETE`.

Полнотекстовый поиск в СУБД MySQL на сегодняшний день поддерживается только для таблиц типа MyISAM и столбцов `CHAR`, `VARCHAR` и `TEXT`.

Замечание

Полнотекстовый поиск в СУБД MySQL не чувствителен к регистру. Кроме того, при поиске игнорируются так называемые "общеупотребительные" слова. К ним относятся слишком короткие слова (по умолчанию состоящие меньше чем из четырех символов), а также слова, встречающиеся, по крайней мере, в половине записей. Так, если в таблице имеются только две записи, то поиск не даст результатов, т. к. каждое слово будет присутствовать в половине записей.

18.1. Индекс *FULLTEXT*

Для использования возможностей полнотекстового поиска необходимо проиндексировать текстовые столбцы таблицы при помощи индекса `FULLTEXT`. На необходимость индексирования текстового столбца можно указать при создании таблицы в операторе `CREATE TABLE`.

Замечание

Синтаксис оператора `CREATE TABLE` подробно рассматривается в *главе 11*.

Определение индекса начинается с ключевого слова FULLTEXT, после которого следует необязательное ключевое слово INDEX. Далее указывается имя индекса (которое может совпадать с именем столбца) и имя индексируемого столбца.

В листинге 18.1 демонстрируется создание таблицы catalogs учебной базы данных shop. Столбец name в таблице catalogs индексируется индексом FULLTEXT.

Замечание

Если индекс создается только по одному столбцу и его имя совпадает с именем столбца, то имя индекса, размещаемое после ключевого слова FULLTEXT, можно опустить.

Листинг 18.1. Индексирование столбца name индексом FULLTEXT

```
mysql> CREATE TABLE catalogs (
    ->   id_catalog INT(11) NOT NULL AUTO_INCREMENT,
    ->   name TINYTEXT NOT NULL,
    ->   PRIMARY KEY (id_catalog),
    ->   FULLTEXT INDEX name (name)
    -> ) ENGINE=MyISAM;
mysql> DESCRIBE catalogs;
+------------+----------+------+-----+---------+----------------+
| Field      | Type     | Null | Key | Default | Extra          |
+------------+----------+------+-----+---------+----------------+
| id_catalog | int(11)  | NO   | PRI | NULL    | auto_increment |
| name       | tinytext | NO   | MUL |         |                |
+------------+----------+------+-----+---------+----------------+
```

Индекс FULLTEXT можно создать сразу по нескольким столбцам. Так, если поиск следует осуществлять сразу по нескольким столбцам, то в скобках после имени индекса следует перечислить имена индексируемых столбцов через запятую. В листинге 18.2 приводится оператор CREATE TABLE, создающий таблицу products из учебной базы данных shop, в которой проиндексированы столбцы name и description.

Листинг 18.2. Индексирование нескольких столбцов

```
mysql> CREATE TABLE products (
    ->   id_product INT(11) NOT NULL AUTO_INCREMENT,
    ->   name TINYTEXT NOT NULL,
    ->   price DECIMAL(7,2) DEFAULT '0.00',
    ->   count INT(11) DEFAULT '0',
    ->   mark FLOAT(4,1) NOT NULL DEFAULT '0.0',
    ->   description TEXT,
    ->   id_catalog INT(11) NOT NULL DEFAULT '0',
    ->   PRIMARY KEY  (id_product),
    ->   INDEX id_catalog (id_catalog),
    ->   FULLTEXT INDEX search (name, description)
    -> ) ENGINE=MyISAM DEFAULT CHARSET=cp1251;
```

Имя индекса может быть любым, в листинге 18.2 в качестве такого имени выбрано `search`.

Часто задача индексирования столбцов возникает после того, как данные в таблицу уже добавлены. Это может быть вызвано необходимостью добавления поисковых возможностей в уже готовое приложение или переносом таблицы при помощи текстового дампа с ускоренным заполнением таблицы, после которого требуется повторное построение индекса.

Для добавления индекса `FULLTEXT` в уже готовую таблицу предназначен оператор `ALTER TABLE`. Индекс `FULLTEXT`, как и любой другой индекс, создается при помощи ключевого слова `ADD`, за которым следует определение индекса. В листинге 18.3 приводятся операторы `ALTER TABLE` для создания индекса в рассмотренных выше таблицах `catalogs` и `products`.

Замечание

Синтаксис оператора `ALTER TABLE` подробно рассматривается в *главе 10*.

Листинг 18.3. Создание индекса `FULLTEXT` при помощи `ALTER TABLE`

```
ALTER TABLE catalogs ADD FULLTEXT name (name);
ALTER TABLE products ADD FULLTEXT search (name, description);
```

В таблице не накладывается ограничений на число индексов, допустимо создание нескольких индексов, как показано в листинге 18.4.

Листинг 18.4. Создание нескольких индексов

```
ALTER TABLE products ADD FULLTEXT (name),
                     ADD FULLTEXT (description),
                     ADD FULLTEXT search (name, description);
```

Помимо оператора `ALTER TABLE`, для создания индекса `FULLTEXT` в уже существующей таблице можно использовать оператор `CREATE INDEX`, показанный в листинге 18.5.

Листинг 18.5. Использование оператора `CREATE INDEX`

```
CREATE FULLTEXT INDEX name ON catalogs (name);
CREATE FULLTEXT INDEX search ON catalogs (name, description);
```

18.2. Конструкция *MATCH (...) AGAINST (...)*

Полнотекстовый поиск выполняется с помощью конструкции `MATCH (...) AGAINST (...)`, которая помещается в конструкцию `WHERE` оператора `SELECT`. В круглых скобках после ключевого слова `MATCH` указываются имена столбцов, по которым

производится поиск, а в скобках после AGAINST указывается фраза, которую необходимо найти.

Для того чтобы осуществлять поиск по столбцу или по комбинации столбцов, столбец и каждая из комбинаций должны быть проиндексированы. То есть создание индекса FULLTEXT по двум столбцам name и description (листинг 18.5) позволит производить поиск одновременно по этим двум столбцам:

```
SELECT * FROM products
WEHRE MATCH (name, description) AGAINST ('Celeron')
```

На запросы по отдельным столбцам будет возвращена ошибка 1191 — отсутствие проиндексированного столбца:

```
SELECT * FROM products
WEHRE MATCH (name) AGAINST ('Celeron')
SELECT * FROM products
WEHRE MATCH (description) AGAINST ('Celeron')
```

Для осуществления данных запросов необходимо создать отдельные индексы по столбцам name и description, помимо уже существующего индекса по обоим столбцам.

В листинге 18.6 демонстрируется поиск слова '256MB' в таблице products учебной базы данных shop.

Листинг 18.6. Использование конструкции

```
mysql> SELECT name FROM products
    -> WHERE MATCH (name, description) AGAINST ('256MB');
+------------------------------+
| name                         |
+------------------------------+
| DDR-400 256MB PQI            |
| DDR-400 256MB Kingston       |
| DDR-400 256MB Hynix Original |
| SAPPHIRE 256MB RADEON 9550   |
+------------------------------+
```

Для каждой строки конструкция MATCH (...) AGAINST (...) возвращает коэффициент релевантности, т. е. степень сходства между искомой строкой и текстом в таблице.

Когда конструкция MATCH (...) AGAINST (...) используется в выражении WHERE, возвращенные строки столбцов автоматически сортируются с помещением в начало списка наиболее релевантных записей. Величина релевантности представляет собой неотрицательное число с плавающей точкой. Релевантность вычисляется на основе количества слов в данной строке столбца, количества уникальных слов в этой строке, общего количества слов в тексте и числа строк, содержащих отдельное слово.

Таблица products учебной базы данных shop проиндексирована по двум столбцам: name и description. Каждое из полей содержит название процессора, а поле description содержит также слово 'Процессор', поэтому вначале размещаются записи для процессоров 'Pentium' и лишь затем 'Celeron' (листинг 18.7).

Листинг 18.7. Использование конструкции MATCH (...) AGAINST (...)

```
mysql> SELECT name FROM products
    -> WHERE MATCH(name, description) AGAINST ('Процессор Pentium');
+------------------------+
| name                   |
+------------------------+
| Intel Pentium 4 3.0GHz |
| Intel Pentium 4 3.2GHz |
| Intel Pentium 4 3.0GHz |
| Celeron 1.8            |
| Celeron 2.0GHz         |
| Celeron 2.4GHz         |
| Celeron D 320 2.4GHz   |
| Celeron D 325 2.53GHz  |
| Celeron D 315 2.26GHz  |
+------------------------+
```

Если поместить конструкцию в список столбцов, следующих за ключевым словом SELECT, можно получить численные значения коэффициента релевантности (листинг 18.8).

Листинг 18.8. Численные значения коэффициента релевантности

```
mysql> SELECT name,
    ->        MATCH(name, description)
    ->        AGAINST ('Процессор Pentium') AS coefficient
    -> FROM products
    -> WHERE MATCH(name, description) AGAINST ('Процессор Pentium');
+------------------------+------------------+
| name                   | coefficient      |
+------------------------+------------------+
| Intel Pentium 4 3.0GHz |  3.3197972520964 |
| Intel Pentium 4 3.2GHz |  3.2590784068991 |
| Intel Pentium 4 3.0GHz |  3.2590784068991 |
| Celeron 1.8            | 0.70367693948263 |
| Celeron 2.0GHz         | 0.62730217952272 |
| Celeron 2.4GHz         | 0.62730217952272 |
| Celeron D 320 2.4GHz   | 0.62730217952272 |
| Celeron D 325 2.53GHz  | 0.62730217952272 |
| Celeron D 315 2.26GHz  | 0.62730217952272 |
+------------------------+------------------+
```

Как видно из приведенных выше примеров, в результат полнотекстового поиска входят записи, содержащие хотя бы одно слово из строки AGAINST. Для разбивки текста на слова СУБД MySQL использует очень простой синтаксический анализатор. "Словом" является любая последовательность символов, состоящая из букв, чисел, знаков "'" и "_". Любое "слово" меньше 4-х символов игнорируется.

Каждое слово в наборе проверяемых текстов оценивается в соответствии с его важностью в этом запросе или наборе текстов. Таким образом, слово, присутствующее во многих документах, будет иметь меньший вес (и даже, возможно, нулевой, если слово присутствует в более чем половине записей таблицы), как имеющее более низкое смысловое значение в данном конкретном наборе текстов. С другой стороны, редко встречающееся слово получит более высокий вес. Затем полученные значения весов слов объединяются для вычисления релевантности данной строки столбца.

Можно составлять достаточно сложные SQL-запросы, прибегая к комбинации нескольких конструкций MATCH (...) AGAINST (...) при помощи логических операторов AND, OR и NOT. В листинге 18.9 приводится пример запроса, в котором слова 'Процессор' и 'Pentium' ищутся при помощи двух отдельных конструкций MATCH (...) AGAINST (...), объединенных логическим оператором AND.

Листинг 18.9. Использование логических операторов при полнотекстовом поиске

```
mysql> SELECT name FROM products
    -> WHERE MATCH (name, description) AGAINST ('Процессор') AND
    ->       MATCH (name, description) AGAINST ('Pentium');
+------------------------+
| name                   |
+------------------------+
| Intel Pentium 4 3.0GHz |
| Intel Pentium 4 3.2GHz |
| Intel Pentium 4 3.0GHz |
+------------------------+
```

Как видно из листинга 18.9, возвращаются только те строки, которые содержат и слово 'Процессор' (в данном случае оно входит в состав поля description, начинающегося фразами 'Процессор Celeron', 'Процессор Pentium'), и слово 'Pentium'. По сравнению с запросом, представленным в листинге 18.8, возвращаются только три записи.

При использовании комбинации нескольких конструкций MATCH (...) AGAINST (...), объединенных логическими операторами, теряется преимущество сортировки результатов по коэффициенту релевантности, однако в большинстве случаев такая сортировка не нужна и требуется просто отсортировать результат либо по алфавиту, либо по дате. Если все-таки сортировка по релевантности необходима, следует воспользоваться так называемым "логическим режимом".

18.3. Логический режим

Логический режим позволяет более гибко управлять системой поиска, для его активизации в конструкции AGAINST после строки с ключевыми словами следует поместить конструкцию IN BOOLEAN MODE, как это продемонстрировано в листинге 18.10.

Листинг 18.10. Поиск в логическом режиме

```
mysql> SELECT name FROM products
    -> WHERE MATCH (name, description)
    -> AGAINST ('Процессор +Pentium' IN BOOLEAN MODE);
+------------------------+
| name                   |
+------------------------+
| Intel Pentium 4 3.2GHz |
| Intel Pentium 4 3.0GHz |
| Intel Pentium 4 3.0GHz |
+------------------------+
```

По сравнению с листингом 18.8, запрос в листинге 18.10 выдал только три записи, это связано с тем, что ключевому слову 'Pentium' предшествует знак "+", который требует, чтобы слово входило в каждую запись. Поэтому записи, содержащие в поле description только слово 'Процессор', но не содержащие в индексированных столбцах слова 'Pentium', отбрасываются.

Листинг 18.11. Исключение слов из выборки

```
mysql> SELECT name FROM products
    -> WHERE MATCH (name, description)
    -> AGAINST ('Процессор -Intel' IN BOOLEAN MODE);
+------------------------+
| name                   |
+------------------------+
| Celeron 1.8            |
| Celeron 2.0GHz         |
| Celeron 2.4GHz         |
| Celeron D 320 2.4GHz   |
| Celeron D 325 2.53GHz  |
| Celeron D 315 2.26GHz  |
+------------------------+
```

Если слову предшествует знак "-", то СУБД MySQL вернет только те записи, которые не содержат данного слова. В листинге 18.11 представлен запрос, который возвращает все записи, в состав которых входит слово 'Процессор', но из выборки исключаются записи, содержащие слово 'Intel'.

Важно отметить, что запрос, состоящий из одних слов, которые предваряет минус, вернет пустую выборку (листинг 18.12).

Листинг 18.12. Использование символа "-"

```
mysql> SELECT name FROM products
    -> WHERE MATCH (name, description)
    -> AGAINST ('-Процессор -Intel' IN BOOLEAN MODE);
Empty set (0.00 sec)
```

По умолчанию (если ни плюс, ни минус не указаны) данное слово является не обязательным, но содержащие его строки будут оцениваться более высоко. Это имитирует поведение команды `MATCH (...) AGAINST(...)` без модификатора `IN BOOLEAN MODE`.

Листинг 18.13. Имитация обычного полнотекстового режима

```
mysql> SELECT description FROM products
    -> WHERE MATCH (name, description)
    ->       AGAINST ('Celeron Pentium' IN BOOLEAN MODE);
+-------------------------------------------------------------------+
| description                                                       |
+-------------------------------------------------------------------+
| Процессор Celeron® 1.8GHz, 128kb, 478-PGA, 400Mhz, OEM 0.18       |
| Процессор Celeron® 2.0GHz, 128KB, 478-PGA, 400MHz, OEM            |
| Процессор Celeron® 2.4GHz, 128kb, 478-PGA, 400Mhz, OEM            |
| Процессор Celeron® D 320 2.4GHz, 256kb, 478-PGA, 533Mhz, OEM      |
| Процессор Celeron® D 325 2.53GHz, 256kb, 478-PGA, 533Mhz, OEM     |
| Процессор Celeron® D 315 2.26GHz, 256kb, 478-PGA, 533Mhz, OEM     |
| Процессор Intel® Pentium®4 3.2GHz, 1Mb, 478-PGA, 800Mhz, HT, OEM  |
| Процессор Intel® Pentium®4 3.0GHz, 512Kb, 478-PGA, 800Mhz, HT, OEM |
| Процессор Intel® Pentium®4 3.0GHz, 1Mb, 478-PGA, 800Mhz, HT, OEM  |
+-------------------------------------------------------------------+
```

В листинге 18.13 выборка содержит записи, в которых встречаются как слово '`Celeron`', так и '`Pentium`'.

Помимо "+" и "–", в логическом режиме предусмотрено еще несколько специальных символов, описание которых приведено в табл. 18.1.

Таблица 18.1. *Специальные символы логического режима*

Символ	Описание
+	Предшествующий слову знак "плюс" показывает, что это слово должно присутствовать в каждой возвращенной строке
-	Предшествующий слову знак "минус" означает, что это слово не должно присутствовать в какой-либо возвращенной строке

Таблица 18.1 *(окончание)*

Символ	Описание
<	Этот оператор используется для того, чтобы уменьшить вклад слова в величину релевантности, которое приписывается строке
>	Этот оператор используется для того, чтобы увеличить вклад слова в величину релевантности, которое приписывается строке
( )	Круглые скобки группируют слова в подвыражения
~	Предшествующий слову знак "тильда" воздействует как оператор отрицания, обуславливая негативный вклад данного слова в релевантность строки. Им отмечают нежелательные слова. Строка, содержащая такое слово, будет оценена ниже других, но не будет исключена совершенно, как в случае оператора "минус"
*	Звездочка является оператором усечения. В отличие от остальных операторов, она должна добавляться в конце слова, а не в начале
"	Фраза, заключенная в двойные кавычки, соответствует только строкам, содержащим эту фразу, написанную буквально

В листинге 18.14 демонстрируется применение специальных символов ">" и "<" для управления коэффициентом релевантности.

Листинг 18.14. Управление коэффициентом релевантности

```
mysql> SELECT name FROM products
    -> WHERE MATCH (name, description)
    -> AGAINST ('Процессор (>Celeron <Pentium)' IN BOOLEAN MODE);
+------------------------+
| name                   |
+------------------------+
| Celeron 1.8            |
| Celeron 2.0GHz         |
| Celeron 2.4GHz         |
| Celeron D 320 2.4GHz   |
| Celeron D 325 2.53GHz  |
| Celeron D 315 2.26GHz  |
| Intel Pentium 4 3.2GHz |
| Intel Pentium 4 3.0GHz |
| Intel Pentium 4 3.0GHz |
+------------------------+
```

SQL-запрос в листинге 18.14 ищет комбинацию слов '`Процессор`' и '`Celeron`' или '`Процессор`' и '`Pentium`', но ранг '`Процессор Celeron`' выше, чем '`Процессор Pentium`'.

Выведем числовые значения коэффициента релевантности для данного запроса (листинг 18.15).

Листинг 18.15. Числовые значения коэффициента релевантности

```
mysql> SELECT name,
    ->        MATCH (name, description)
    ->        AGAINST ('Процессор (>Celeron <Pentium)' IN BOOLEAN MODE)
    ->        AS coefficient
    -> FROM products
    -> WHERE MATCH (name, description)
    ->       AGAINST ('Процессор (>Celeron <Pentium)' IN BOOLEAN MODE);
+------------------------+-----------------+
| name                   | coefficient     |
+------------------------+-----------------+
| Celeron 1.8            |             2.5 |
| Celeron 2.0GHz         |             2.5 |
| Celeron 2.4GHz         |             2.5 |
| Celeron D 320 2.4GHz   |             2.5 |
| Celeron D 325 2.53GHz  |             2.5 |
| Celeron D 315 2.26GHz  |             2.5 |
| Intel Pentium 4 3.2GHz | 1.6666667461395 |
| Intel Pentium 4 3.0GHz | 1.6666667461395 |
| Intel Pentium 4 3.0GHz | 1.6666667461395 |
+------------------------+-----------------+
```

Однако без использования логического режима и символов управления коэффициентом релевантности порядок сортировки прямо противоположный: сначала следуют записи, соответствующие процессору '`Pentium`' и лишь затем — процессору '`Celeron`' (листинг 18.16).

Листинг 18.16. Сортировка записей в полнотекстовом режиме

```
mysql> SELECT name,
    ->        MATCH (name, description)
    ->        AGAINST ('Процессор Celeron Pentium') AS coefficient
    -> FROM products
    -> WHERE MATCH (name, description)
    -> AGAINST ('Процессор Celeron Pentium');
+------------------------+-----------------+
| name                   | coefficient     |
+------------------------+-----------------+
| Intel Pentium 4 3.0GHz | 3.3197972520964 |
| Intel Pentium 4 3.2GHz | 3.2590784068991 |
| Intel Pentium 4 3.0GHz | 3.2590784068991 |
| Celeron 1.8            | 2.6530160544406 |
| Celeron 2.0GHz         | 2.3650664294898 |
| Celeron 2.4GHz         | 2.3650664294898 |
| Celeron D 320 2.4GHz   | 2.3650664294898 |
| Celeron D 325 2.53GHz  | 2.3650664294898 |
| Celeron D 315 2.26GHz  | 2.3650664294898 |
+------------------------+-----------------+
```

Важно отметить, что при использовании логического режима без управляющих символов ">" и "<" коэффициент релевантности для всех строк одинаковый и составляет величину 2 (листинг 18.17).

Листинг 18.17. Числовые значения коэффициента релевантности

```
mysql> SELECT name,
    ->        MATCH (name, description)
    ->        AGAINST ('Процессор Celeron Pentium' IN BOOLEAN MODE)
    ->        AS coefficient
    -> FROM products
    -> WHERE MATCH (name, description)
    -> AGAINST ('Процессор Celeron Pentium' IN BOOLEAN MODE);
+------------------------+-------------+
| name                   | coefficient |
+------------------------+-------------+
| Celeron 1.8            |           2 |
| Celeron 2.0GHz         |           2 |
| Celeron 2.4GHz         |           2 |
| Celeron D 320 2.4GHz   |           2 |
| Celeron D 325 2.53GHz  |           2 |
| Celeron D 315 2.26GHz  |           2 |
| Intel Pentium 4 3.2GHz |           2 |
| Intel Pentium 4 3.0GHz |           2 |
| Intel Pentium 4 3.0GHz |           2 |
+------------------------+-------------+
```

В отличие от предыдущих спецсимволов, символ "*" всегда располагается в конце слова и указывает на то, что после слова может следовать любое число других символов, составляющих слово. Например, последовательности '`файл*`' будут соответствовать слова '`файл`', '`файловый`', '`файлообмен`' и т. п.

Замечание

Однако символ "*" не может использоваться для поиска слов, длина которых меньше четырех символов.

Использование двойных кавычек " требует поиска неразрывной фразы. В листинге 18.18 производится поиск фразы '`Процессор Celeron`'

Листинг 18.18. Поиск неразрывной фразы

```
mysql> SELECT description FROM products
    -> WHERE MATCH (name, description)
    -> AGAINST ('"Процессор Celeron"' IN BOOLEAN MODE);
+-------------------------------------------------------------+
| description                                                 |
+-------------------------------------------------------------+
| Процессор Celeron® 1.8GHz, 128kb, 478-PGA, 400Mhz, OEM 0.18 |
| Процессор Celeron® 2.0GHz, 128KB, 478-PGA, 400MHz, OEM      |
```

```
| Процессор Celeron® 2.4GHz, 128kb, 478-PGA, 400Mhz, OEM         |
| Процессор Celeron® D 320 2.4GHz, 256kb, 478-PGA, 533Mhz, OEM   |
| Процессор Celeron® D 325 2.53GHz, 256kb, 478-PGA, 533Mhz, OEM  |
| Процессор Celeron® D 315 2.26GHz, 256kb, 478-PGA, 533Mhz, OEM  |
+----------------------------------------------------------------+
```

Несмотря на то, что записи, соответствующие процессору 'Pentium', содержат слово 'Процессор', они в выборку не попадают, т. к. в них отсутствует фраза 'Процессор Celeron'. Более того, поиск фразы "Процессор Pentium" с использованием двойных кавычек вернет пустую результирующую таблицу (листинг 18.19).

Листинг 18.19. Использование двойных кавычек

```
mysql> SELECT description FROM products
    -> WHERE MATCH (name, description)
    -> AGAINST ('"Процессор Pentium"' IN BOOLEAN MODE);
Empty set (0.00 sec)
```

К верному результату приведет лишь фраза "Процессор Intel Pentium", которая встречается в поле description соответствующего процессора (листинг 18.20).

Замечание

Важно отметить, что графический символ "®", не относящийся к числовым и алфавитным символам, воспринимается как символ-разделитель и не воспринимается при полнотекстовом поиске.

Листинг 18.20. Поиск полной фразы

```
mysql> SELECT description FROM products
    -> WHERE MATCH (name, description)
    -> AGAINST ('"Процессор Intel Pentium"' IN BOOLEAN MODE);
+--------------------------------------------------------------------+
| description                                                        |
+--------------------------------------------------------------------+
| Процессор Intel® Pentium®4 3.2GHz, 1Mb, 478-PGA, 800Mhz, HT, OEM   |
| Процессор Intel® Pentium®4 3.0GHz, 512Kb, 478-PGA, 800Mhz, HT, OEM |
| Процессор Intel® Pentium®4 3.0GHz, 1Mb, 478-PGA, 800Mhz, HT, OEM   |
+--------------------------------------------------------------------+
```

ГЛАВА 19

Функции, применяемые вместе с конструкцией *GROUP BY*

Функции, применяемые совместно с конструкцией `GROUP BY`, часто называют агрегатными или суммирующими функциями. Они вычисляют одно значение для каждой группы, которые создает конструкция `GROUP BY` *(см. главу 7)*. Функции данной группы позволяют узнать число строк, входящих в группу, подсчитать среднее значение или получить сумму значений столбцов. Результирующее значение рассчитывается только для значений, не равных `NULL` (исключение составляет лишь функция `COUNT(*)`, которая подсчитывает число всех строк). Несмотря на то, что функции позиционируются как функции, применяемые совместно с конструкцией `GROUP BY`, вполне допустимо их использование в запросах без группировки. В этом случае вся выборка выступает как одна большая группа.

19.1. Функция *AVG()*

Функция `AVG()` возвращает среднее значение аргумента `expr` и имеет следующий синтаксис:

```
AVG([DISTINCT] expr)
```

В качестве аргумента `expr` обычно выступает имя столбца. Необязательное ключевое слово `DISTINCT` позволяет дать указание СУБД MySQL обрабатывать только уникальные значения столбца `expr`.

> ***Замечание***
>
> Ключевое слово `DISTINCT` в функции `AVG()` добавлено, начиная с версии 5.0.3.

В листинге 19.1 приводится запрос к таблице `products` учебной базы данных `shop`, который возвращает среднюю цену товарной позиции в магазине.

Листинг 19.1. Использование функции `AVG()`

```
mysql> SELECT AVG(price) FROM products;
```

```
+-------------+
| AVG(price)  |
+-------------+
| 2858.966667 |
+-------------+
```

При использовании конструкции `GROUP BY` функция `AVG()` вычисляет среднее значение для каждой из группы. Пусть группировка в таблице `products` происходит по полю `id_catalog`, тогда при помощи встроенной функции `AVG()` можно узнать среднюю цену по каждому из пяти каталогов электронного магазина. В листинге 19.2 выводится результирующая таблица с уникальными номерами `id_catalog` и соответствующие каждому из каталогов средние цены.

Листинг 19.2. Использование функции `AVG()` совместно с `GROUP BY`

```
mysql> SELECT id_catalog, AVG(price) FROM products GROUP BY id_catalog;
+------------+-------------+
| id_catalog | AVG(price)  |
+------------+-------------+
|          1 | 3482.333333 |
|          2 | 2801.166667 |
|          3 | 3843.500000 |
|          4 | 2749.000000 |
|          5 | 1417.000000 |
+------------+-------------+
```

Средние значения, полученные при помощи функции `AVG()`, могут использоваться в вычисляемых столбцах. Например, для того чтобы увеличить средние значения для каждого из каталогов на 20 %, достаточно умножить либо столбец `price`, либо функцию `AVG()` на 1.2 (листинг 19.3).

Листинг 19.3. Использование функции `AVG()` в выражениях

```
mysql> SELECT id_catalog, AVG(price*1.2) FROM products
    -> GROUP BY id_catalog;
+------------+----------------+
| id_catalog | AVG(price*1.2) |
+------------+----------------+
|          1 | 4178.8000000   |
|          2 | 3361.4000000   |
|          3 | 4612.2000000   |
|          4 | 3298.8000000   |
|          5 | 1700.4000000   |
+------------+----------------+
5 rows in set (0.03 sec)
mysql> SELECT id_catalog, AVG(price)*1.2 FROM products
    -> GROUP BY id_catalog;
```

```
+------------+----------------+
| id_catalog | AVG(price)*1.2 |
+------------+----------------+
|          1 | 4178.8000000   |
|          2 | 3361.4000000   |
|          3 | 4612.2000000   |
|          4 | 3298.8000000   |
|          5 | 1700.4000000   |
+------------+----------------+
5 rows in set (0.00 sec)
```

В первом случае сначала каждое из значений столбца `price` умножается на 1.2, а затем производится группировка и вычисление результата при помощи функции `AVG()`. Во втором сначала производится группировка значений и вычисление средних значений, и лишь после этого результат умножается на 1.2. Как следствие, первый запрос выполняется 0.03 секунды, а второй 0.00 секунды.

По полученным в результате выполнения агрегатных функций значениям может выполняться сортировка результирующей таблицы. Для этого столбцу назначается псевдоним при помощи оператора `AS`, который передается конструкции `ORDER BY` (листинг 19.4).

Листинг 19.4. Сортировка результирующей таблицы

```
mysql> SELECT id_catalog, AVG(price) AS price FROM products
    -> GROUP BY id_catalog ORDER BY price DESC;
+------------+-------------+
| id_catalog | price       |
+------------+-------------+
|          3 | 3843.500000 |
|          1 | 3482.333333 |
|          2 | 2801.166667 |
|          4 | 2749.000000 |
|          5 | 1417.000000 |
+------------+-------------+
```

Если необходимо использовать в условии вычисляемые столбцы, следует использовать конструкцию `HAVING`, а не `WHERE`.

Листинг 19.5. Использование условия `HAVING`

```
mysql> SELECT id_catalog, AVG(price) AS price FROM products
    -> GROUP BY id_catalog HAVING price > 3000 ORDER BY price DESC;
+------------+-------------+
| id_catalog | price       |
+------------+-------------+
|          3 | 3843.500000 |
|          1 | 3482.333333 |
+------------+-------------+
```

19.2. Функция *BIT_AND()*

Функция `BIT_AND()` возвращает побитовое И для всех битов в `expr` и имеет следующий синтаксис:

```
BIT_AND(expr)
```

Вычисление производится с 64-битовой (BIGINT) точностью (листинг 19.6).

Листинг 19.6. Использование функции `BIT_AND()`

```
mysql> SELECT BIT_AND(id_catalog) FROM catalogs;
+---------------------+
| BIT_AND(id_catalog) |
+---------------------+
|                   0 |
+---------------------+
```

Результат можно легко понять, если представить значения `id_catalog` в двоичном виде и обратиться к табл. 13.1, где приведены правила двоичного сложения разрядов при использовании логики И.

```
001 (1)
010 (2)
011 (3)
100 (4)
101 (5)
-------
000 (0)
```

При группировке по полю `id_catalog`, когда каждая запись образует свою собственную группу, функция возвращает исходные значения полей `id_catalog` (листинг 19.7).

Листинг 19.7. Использование функции `BIT_AND()` совместно с `GROUP BY`

```
mysql> SELECT BIT_AND(id_catalog) FROM catalogs GROUP BY id_catalog;
+---------------------+
| BIT_AND(id_catalog) |
+---------------------+
|                   1 |
|                   2 |
|                   3 |
|                   4 |
|                   5 |
+---------------------+
```

19.3. Функция *BIT_OR()*

Функция `BIT_OR()` (листинг 19.8) возвращает побитовое ИЛИ для всех битов в `expr` и имеет следующий синтаксис:

```
BIT_OR(expr)
```

Вычисление производится с 64-битовой (BIGINT) точностью.

Листинг 19.8. Использование функции `BIT_OR()`

```
mysql> SELECT BIT_OR(id_catalog) FROM catalogs;
+--------------------+
| BIT_OR(id_catalog) |
+--------------------+
|                  7 |
+--------------------+
```

Результат можно легко понять, если представить значения `id_catalog` в двоичном представлении и обратиться к табл. 13.2, где приведены правила двоичного сложения разрядов при использовании логики ИЛИ.

```
001 (1)
010 (2)
011 (3)
100 (4)
101 (5)
-------
111 (7)
```

Точно так же, как и в случае с функцией `BIT_AND()`, при группировке по полю `id_catalog` возвращаются исходные значения полей `id_catalog`, т. к. в этом случае аргумент один и сложения как такового не происходит.

19.4. Функция *BIT_XOR()*

Функция `BIT_OR()` (листинг 19.9) возвращает исключающее побитовое ИЛИ для всех битов в `expr` и имеет следующий синтаксис:

```
BIT_XOR(expr)
```

Вычисление производится с 64-битовой (BIGINT) точностью.

Замечание

Функция `BIT_XOR()` введена в СУБД MySQL, начиная с версии 4.1.1.

Листинг 19.9. Использование функции BIT_XOR()

```
mysql> SELECT BIT_XOR(id_catalog) FROM catalogs;
+---------------------+
| BIT_XOR(id_catalog) |
+---------------------+
|                   1 |
+---------------------+
```

Результат можно легко понять, если представить значения id_catalog в двоичном представлении и обратиться к табл. 13.3, где приведены правила двоичного сложения разрядов при использовании логики исключающего ИЛИ.

```
001 (1)
010 (2)
011 (3)
100 (4)
101 (5)
-------
001 (7)
```

Точно так же, как и в случае с функцией BIT_AND(), при группировке по полю id_catalog возвращаются исходные значения полей id_catalog, т. к. в этом случае аргумент один и сложения как такового не происходит.

19.5. Функция *COUNT()*

Функция COUNT() подсчитывает число записей и имеет несколько форм, синтаксис которых представлен далее:

```
COUNT(expr)
COUNT(*)
COUNT(DISTINCT expr1, expr2, ...)
```

Первая форма возвращает число записей в таблице, поле expr для которых не равно NULL. В листинге 19.10 представлен запрос, возвращающий число непустых полей id_user, email и url таблицы users учебной базы данных shop.

Листинг 19.10. Использование функции COUNT()

```
mysql> SELECT id_user, email, url FROM users;
+---------+----------------------+-------------------------+
| id_user | email                | url                     |
+---------+----------------------+-------------------------+
|       1 | ivanov@email.ru      | NULL                    |
|       2 | losev@email.ru       | NULL                    |
|       3 | simdyanov@softtime.ru | http://www.softtime.ru/ |
```

```
|        4 | kuznetsov@softtime.ru | http://www.softtime.ru  |
|        5 | NULL                  | NULL                    |
|        6 | korneev@domen.ru      | NULL                    |
+---------+----------------------+------------------------+
mysql> SELECT COUNT(id_user), COUNT(email), COUNT(url) FROM users;
+----------------+--------------+------------+
| COUNT(id_user) | COUNT(email) | COUNT(url) |
+----------------+--------------+------------+
|              6 |            5 |          2 |
+----------------+--------------+------------+
```

Как видно из листинга 19.10, для полей `id_user`, `email` и `url` возвращаются различные значения. Это связано с тем, что число `NULL`-полей во всех трех столбцах различается.

Форма функции `COUNT(*)` возвращает общее число строк в таблице независимо от того, принимает какое-либо поле значение `NULL` или нет (листинг 19.11). Запись таблицы учитывается в результате, даже если все поля записи равны `NULL`.

Листинг 19.11. Использование функции `COUNT(*)`

```
mysql> SELECT COUNT(*) FROM users;
+----------+
| COUNT(*) |
+----------+
|        6 |
+----------+
```

Функция `COUNT(*)` оптимизирована для очень быстрого возврата результата при условии, что команда `SELECT` извлекает данные из одной таблицы, никакие другие столбцы не обрабатываются и запрос не содержит условия `WHERE`.

Функция `COUNT()` может быть использована не только для подсчета общего числа записей в таблице, но и для подсчета числа строк в выборке с условием `WHERE`.

Листинг 19.12. Использование функции `COUNT()` совместно с `WHERE`

```
mysql> SELECT COUNT(*) FROM products WHERE price < 2000;
+----------+
| COUNT(*) |
+----------+
|       12 |
+----------+
```

В листинге 19.12 представлен запрос, извлекающий из таблицы `products` учебной базы `shop` записи, соответствующие товарным позициям, цена которых не превышает 2000 рублей.

Разумеется, как агрегатная функция, `COUNT()` может быть использована для вычисления числа записей в каждой из групп, полученных в результате применения группировки результата с помощью конструкции `GROUP BY`.

В листинге 19.13 представлен запрос, сообщающий о числе товарных позиций в каждом из каталогов электронного магазина.

Листинг 19.13. Подсчет числа товарных позиций в каждом из каталогов

```
mysql> SELECT id_catalog, COUNT(*) AS total FROM products
    -> GROUP BY id_catalog ORDER BY total;
+------------+-------+
| id_catalog | total |
+------------+-------+
|          3 |     4 |
|          4 |     5 |
|          2 |     6 |
|          5 |     6 |
|          1 |     9 |
+------------+-------+
```

Сортировка строк в результирующей таблице производится по числу товарных позиций в каталоге. Для этого столбцу `COUNT(*)` при помощи ключевого слова `AS` назначается псевдоним `total`, который затем используется в выражении `ORDER BY`.

Третий вариант функции `COUNT()` позволяет использовать ключевое слово `DISTINCT`, которое позволяет подсчитать только уникальные значения столбца (листинг 19.14).

Листинг 19.14. Использование ключевого слова `DISTINCT` совместно с `COUNT()`

```
mysql> SELECT id_product FROM orders;
+------------+
| id_product |
+------------+
|          8 |
|         10 |
|         20 |
|         20 |
|         20 |
+------------+
mysql> SELECT COUNT(id_product), COUNT(DISTINCT id_product) FROM orders;
+-------------------+---------------------------+
| COUNT(id_product) | COUNT(DISTINCT id_product) |
+-------------------+---------------------------+
|                 5 |                         3 |
+-------------------+---------------------------+
```

Так как в поле id_product таблицы orders значение 20 встречается три раза, то функция COUNT(DISTINCT id_product) возвращает только три уникальных значения вместо пяти.

19.6. Функция *GROUP_CONCAT()*

Функция GROUP_CONCAT() объединяет значения отдельных групп, полученных в результате применения конструкции GROUP BY, в одну строку и имеет следующий синтаксис:

```
GROUP_CONCAT([DISTINCT] expr [,expr ...]
             [ORDER BY {unsigned_integer | col_name | expr}
             [ASC | DESC] [,col_name ...]]
             [SEPARATOR str_val])
```

В простейшем случае функция принимает имя столбца expr и возвращает строку со значениями столбца, разделенными запятыми.

Замечание

Функция появилась в MySQL, начиная с версии 4.1.0.

Листинг 19.15. Использование функции GROUP_CONCAT()

```
mysql> SELECT GROUP_CONCAT(id_catalog) FROM catalogs;
+--------------------------+
| GROUP_CONCAT(id_catalog) |
+--------------------------+
| 1,2,3,4,5                |
+--------------------------+
```

При передаче в качестве аргумента имени числового столбца значения автоматически преобразуются к текстовому типу.

Ключевое слово DISTINCT позволяет потребовать вернуть только уникальные значения столбца, а ключевое слово SEPARATOR позволяет задать в качестве разделителя значений произвольный символ. В листинге 19.16 представлен запрос, извлекающий уникальные значения столбца id_product таблицы orders учебной базы данных shop с использованием в качестве разделителя символа '-'.

Листинг 19.16. Использование ключевых слов DISTINCT и SEPARATOR

```
mysql> SELECT GROUP_CONCAT(DISTINCT id_product SEPARATOR '-')
    -> FROM orders;
+-------------------------------------------------+
| GROUP_CONCAT(DISTINCT id_product SEPARATOR '-') |
+-------------------------------------------------+
| 8-10-20                                         |
+-------------------------------------------------+
```

Ключевое слово `ORDER BY` позволяет отсортировать значения в рамках возвращаемой строки. Отсортируем значения, представленные в листинге 19.16, в обратном порядке (листинг 19.17).

Листинг 19.17. Использование ключевого слова `ORDER BY`

```
mysql> SELECT GROUP_CONCAT(DISTINCT id_product
    ->                     ORDER BY id_product DESC
    ->                     SEPARATOR '-') AS str
    -> FROM orders;
+---------+
| str     |
+---------+
| 20-10-8 |
+---------+
```

Возвратим строки с ценами (в порядке убывания) на товарные позиции для каждого из каталогов учебного электронного магазина (листинг 19.18).

Листинг 19.18. Список цен каталогов

```
mysql> SELECT GROUP_CONCAT(price ORDER BY id_product DESC)
    -> FROM products GROUP BY id_catalog;
+----------------------------------------------------------------------+
| str                                                                  |
+----------------------------------------------------------------------+
|5673.00,6147.00,7259.00,1880.00,2747.00,1962.00,2109.00,1969.00,1595.00|
|2289.00,2518.00,5395.00,2289.00,2420.00,1896.00                       |
|5886.00,2730.00,1602.00,5156.00                                       |
|2468.00,3139.00,2093.00,3589.00,2456.00                               |
|1717.00,1690.00,1932.00,899.00,1179.00,1085.00                        |
+----------------------------------------------------------------------+
```

19.7. Функция *MIN()*

Возвращает минимальное значение среди всех непустых значений выбранных строк в столбце `expr` и имеет следующий синтаксис:

```
MIN([DISTINCT] expr)
```

В качестве аргумента `expr` обычно выступает имя столбца. Необязательное ключевое слово `DISTINCT` позволяет дать указание СУБД MySQL обрабатывать только уникальные значения столбца `expr`.

Замечание

Ключевое слово `DISTINCT` в функции `MIN()` добавлено, начиная с версии 5.0.0.

Листинг 19.19. Использование функции `MIN()`

```
mysql> SELECT MIN(price) FROM products;
+------------+
| MIN(price) |
+------------+
| 899.00     |
+------------+
```

В листинге 19.19 запрос возвращает минимальную цену в таблице `products`. Использование конструкции `GROUP BY id_catalog` позволяет найти минимальную цену для каждого из каталогов.

Листинг 19.20. Использование функции `MIN()` совместно с конструкцией `GROUP BY`

```
mysql> SELECT id_catalog, MIN(price) FROM products GROUP BY id_catalog;
+------------+------------+
| id_catalog | MIN(price) |
+------------+------------+
|          1 | 1595.00    |
|          2 | 1896.00    |
|          3 | 1602.00    |
|          4 | 2093.00    |
|          5 | 899.00     |
+------------+------------+
```

Как видно из результатов запроса, товарная позиция с минимальной для всей таблицы ценой находится в пятом каталоге.

Функцию `MIN()` можно также использовать со строковыми столбцами. В этом случае возвращается минимальное лексикографическое значение (листинг 19.21).

Листинг 19.21. Использование функции `MIN()` совместно со строковыми столбцами

```
mysql> SELECT MIN(name) FROM products;
+--------------------+
| MIN(name)          |
+--------------------+
| ASUSTEK A9600XT/TD |
+--------------------+
```

19.8. Функция *MAX()*

Возвращает максимальное значение среди всех непустых значений выбранных строк в столбце `expr` и имеет следующий синтаксис:

```
MAX([DISTINCT] expr)
```

В качестве аргумента `expr` обычно выступает имя столбца. Необязательное ключевое слово `DISTINCT` позволяет дать указание СУБД MySQL обрабатывать только уникальные значения столбца `expr`.

Замечание

Ключевое слово `DISTINCT` в функции `MAX()` добавлено, начиная с версии 5.0.0.

Листинг 19.22. Использование функции `MAX()`

```
mysql> SELECT MAX(price) FROM products;
+------------+
| MAX(price) |
+------------+
| 7259.00    |
+------------+
```

В листинге 19.22 запрос возвращает максимальную цену в таблице `products`. Использование конструкции `GROUP BY id_catalog` позволяет найти максимальную цену для каждого из каталогов.

Листинг 19.23. Использование функции `MIN()` совместно с конструкцией `GROUP BY`

```
mysql> SELECT id_catalog, MAX(price) FROM products GROUP BY id_catalog;
+------------+------------+
| id_catalog | MAX(price) |
+------------+------------+
|          1 | 7259.00    |
|          2 | 5395.00    |
|          3 | 5886.00    |
|          4 | 3589.00    |
|          5 | 1932.00    |
+------------+------------+
```

19.9. Функция *STD()*

Функция `STD()` (листинг 19.24) возвращает стандартное среднеквадратичное отклонение значения в аргументе `expr` и имеет следующий синтаксис:

```
STD(expr)
```

Замечание

Функция `STD()` имеет два синонима: первый — `STDDEV()` обеспечивает совместимость с СУБД Oracle и второй — `STDDEV_POP()` обеспечивает совместимость со стандартом SQL.

Замечание

Синоним `STDDEV_POP()` появился в СУБД MySQL, начиная с версии 5.0.3.

Листинг 19.24. Использование функции `STD()`

```
mysql> SELECT STD(price) FROM products;
+-------------+
| STD(price)  |
+-------------+
| 1653.531584 |
+-------------+
mysql> SELECT id_catalog, STD(price) FROM products GROUP BY id_catalog;
+------------+-------------+
| id_catalog | STD(price)  |
+------------+-------------+
|          1 | 2090.202808 |
|          2 | 1175.967604 |
|          3 | 1743.464009 |
|          4 |  539.044711 |
|          5 |  379.681006 |
+------------+-------------+
```

Функция возвращает квадратный корень значения, вычисленный функцией `VAR_POP()`.

19.10. Функция *STDDEV_SAMP()*

Функция `STDDEV_SAMP()` (листинг 19.25) возвращает выборочное среднеквадратическое отклонение и имеет следующий синтаксис:

```
STDDEV_SAMP(expr)
```

Замечание

Функция `STDDEV_SAMP()` появилась в СУБД MySQL, начиная с версии 5.0.3.

Листинг 19.25. Использование функции `STDDEV_SAMP()`

```
mysql> SELECT STDDEV_SAMP(price) FROM products;
+--------------------+
| STDDEV_SAMP(price) |
+--------------------+
|        1681.799129 |
+--------------------+
mysql> SELECT id_catalog, STDDEV_SAMP(price) FROM products
    -> GROUP BY id_catalog;
```

```
+------------+--------------------+
| id_catalog | STDDEV_SAMP(price) |
+------------+--------------------+
|          1 |        2216.994869 |
|          2 |        1288.207967 |
|          3 |        2013.178830 |
|          4 |         602.670308 |
|          5 |         415.919704 |
+------------+--------------------+
```

Функция возвращает квадратный корень значения, вычисленный функцией `VAR_SAMP()`.

19.11. Функция *SUM()*

Функция `SUM()` (листинг 19.26) возвращает сумму величин в столбце `expr` и имеет следующий синтаксис:

```
SUM([DISTINCT] expr)
```

Если возвращаемый набор данных не содержит ни одной строки, то функция возвращает `NULL`. Необязательное ключевое слово `DISTINCT` позволяет указать СУБД MySQL обрабатывать только уникальные значения столбца `expr`.

Замечание

Ключевое слово `DISTINCT` в функции `SUM()` добавлено, начиная с версии 5.0.0.

Листинг 19.26. Использование функции `SUM()`

```
mysql> SELECT SUM(price) FROM products;
+------------+
| SUM(price) |
+------------+
| 85769.00   |
+------------+
mysql> SELECT id_catalog, SUM(price) FROM products GROUP BY id_catalog;
+------------+------------+
| id_catalog | SUM(price) |
+------------+------------+
|          1 | 31341.00   |
|          2 | 16807.00   |
|          3 | 15374.00   |
|          4 | 13745.00   |
|          5 | 8502.00    |
+------------+------------+
```

19.12. Функция *VAR_POP()*

Функция `VAR_POP()` возвращает стандартное отклонение значения в аргументе `expr` и имеет следующий синтаксис:

```
VAR_POP(expr)
```

Замечание

Функция имеет синоним `VARIANCE()`.

Замечание

Функция `VARIANCE()` введена в СУБД MySQL, начиная с версии 4.1.0, в то время как функция `VAR_POP()` появилась лишь в версии 5.0.3 для совместимости со стандартом SQL.

19.13. Функция *VAR_SAMP()*

Функция `VAR_SAMP()` возвращает выборочное отклонение значения в аргументе `expr` и имеет следующий синтаксис:

```
VAR_SAMP(expr)
```

Замечание

Функция `VAR_SAMP()` введена в СУБД MySQL, начиная с версии 5.0.3.

19.14. Конструкция *WITH ROLLUP*

При выборке с применением одной из агрегатных функций и конструкции `GROUP BY` выводится список строк с результатами для каждой из групп. Конструкция `WITH ROLLUP` позволяет добавить еще одну строку с суммой значений всех предыдущих строк.

Листинг 19.27. Использование конструкции `WITH ROLLUP`

```
mysql> SELECT id_catalog, SUM(price) FROM products
    -> GROUP BY id_catalog WITH ROLLUP;
+------------+------------+
| id_catalog | SUM(price) |
+------------+------------+
|          1 | 31341.00   |
|          2 | 16807.00   |
|          3 | 15374.00   |
|          4 | 13745.00   |
|          5 | 8502.00    |
|       NULL | 85769.00   |
+------------+------------+
```

Как видно из листинга 19.27, в результирующую таблицу добавлена дополнительная строка с суммой значений столбца `SUM(price)` и значением `NULL` для столбца `id_catalog`. Значение `NULL` присваивается всем столбцам, кроме сгенерированного агрегатной функцией (в данном случае функцией `SUM()`).

Замечание

При использовании конструкции `WITH ROLLUP` нельзя использовать конструкцию `ORDER BY`.

ГЛАВА 20

Разные функции

Данная глава завершает рассмотрение встроенных функций СУБД MySQL. Здесь будут представлены функции, которые не были описаны в предыдущих главах.

20.1. Функции управления потоком выполнения

Функции управления потоком предназначены для принятия решений в ходе выполнения SQL-инструкций.

20.1.1. Функция *CASE*

Функция `CASE` (листинг 20.1) имеет две формы записи, первая из которых выглядит следующим образом:

```
CASE value WHEN [compare-value] THEN result
          [WHEN [compare-value] THEN result ...]
          [ELSE result] END
```

Функция возвращает значение `result`, когда значение `value` совпадает со значением `compare-value`. Если значения не совпадают, управление передается следующей конструкции `WHEN-THEN`, число которых не ограничено, в конечном итоге управление передается конструкции `ELSE`. Если конструкция `ELSE` отсутствует, функция возвращает `NULL`.

Листинг 20.1. Использование функции `CASE`

```
mysql> SELECT CASE 1 WHEN 1 THEN 'one' WHEN 2 THEN 'two' ELSE 'more' END;
+-------------------------------------------------------------+
| CASE 1 WHEN 1 THEN 'one' WHEN 2 THEN 'two' ELSE 'more' END |
+-------------------------------------------------------------+
| one                                                         |
+-------------------------------------------------------------+
```

```
mysql> SELECT CASE 'C' WHEN 'a' THEN 1 WHEN 'b' THEN 2 END;
+---------------------------------------------+
| CASE 'C' WHEN 'a' THEN 1 WHEN 'b' THEN 2 END |
+---------------------------------------------+
|                                         NULL |
+---------------------------------------------+
```

Вторая форма функции `CASE` имеет следующий синтаксис:

```
CASE WHEN [condition] THEN result
          [WHEN [condition] THEN result ...]
          [ELSE result] END
```

Вторая форма функции `CASE` (листинг 20.2) аналогична первой, только после ключевого слова `WHEN` следует логическое условие `condition`. Если оно равно 1 (истина), возвращается значение `result`, если 0 (ложь) — управление передается следующей инструкции `WHEN` или `ELSE`. Так же как и в первом случае, если отсутствует подходящее значение `condition`, возвращается `NULL`.

Листинг 20.2. Альтернативное использование функции `CASE`

```
mysql> SELECT CASE WHEN 1>0 THEN 'true' ELSE 'false' END;
+-------------------------------------------+
| CASE WHEN 1>0 THEN 'true' ELSE 'false' END |
+-------------------------------------------+
| true                                      |
+-------------------------------------------+
mysql> SELECT CASE WHEN 1<0 THEN 'true' END;
+------------------------------+
| CASE WHEN 1<0 THEN 'true' END |
+------------------------------+
| NULL                         |
+------------------------------+
```

20.1.2. Функция *IF()*

Функция `IF()` (листинг 20.3) предназначена для логического выбора и имеет следующий синтаксис:

```
IF(expr1,expr2,expr3)
```

Если выражение `expr1` истинно (не равно 0 или `NULL`), функция `IF()` возвращает `expr2`, в противном случае — `expr3`. Функция возвращает числовое или строковое значение в зависимости от контекста, в котором выполняется.

Листинг 20.3. Использование функции `IF()`

```
mysql> SELECT IF(1>2,2,3), IF(1<2,'yes','no');
+-------------+--------------------+
| IF(1>2,2,3) | IF(1<2,'yes','no') |
+-------------+--------------------+
|           3 | yes                |
+-------------+--------------------+
```

Наиболее часто данная функция применяется для условного вывода. Пусть стоит задача мониторинга числа комплектующих на складе в учебном электронном магазине `shop`. Если число одного вида товаров не менее 5, то напротив данной товарной позиции будет выведена надпись "Достаточно", в противном случае выведем надпись "Заканчивается" (листинг 20.4).

Листинг 20.4. Определение числа товарных позиций на складе

```
mysql> SELECT name,
    ->        count,
    ->        IF(count>5,'Достаточно','Заканчивается') AS status
    -> FROM products LIMIT 10;
+------------------------+-------+---------------+
| name                   | count | status        |
+------------------------+-------+---------------+
| Celeron 1.8            |    10 | Достаточно    |
| Celeron 2.0GHz         |     2 | Заканчивается |
| Celeron 2.4GHz         |     4 | Заканчивается |
| Celeron D 320 2.4GHz   |     1 | Заканчивается |
| Celeron D 325 2.53GHz  |     6 | Достаточно    |
| Celeron D 315 2.26GHz  |     6 | Достаточно    |
| Intel Pentium 4 3.2GHz |     5 | Заканчивается |
| Intel Pentium 4 3.0GHz |     1 | Заканчивается |
| Intel Pentium 4 3.0GHz |    12 | Достаточно    |
| Gigabyte GA-8I848P-RS  |     4 | Заканчивается |
+------------------------+-------+---------------+
```

В листинге 20.4 информация выводится только для первых 10 товарных позиций из таблицы `products`.

20.1.3. Функция *IFNULL()*

Функция `IFNULL()`, так же как `IF()`, предназначена для логического выбора и имеет следующий синтаксис:

```
IFNULL(expr1,expr2)
```

Если выражение `expr1` не равно `NULL`, функция возвращает `expr1`, в противном случае возвращается `expr2`. Функция `IFNULL()` возвращает число или строку в зависимости

от контекста, в котором вызывается. В листинге 20.5 приводятся примеры использования функции — везде, где в качестве первого аргумента передается NULL или выражение, приводящее к NULL, — функция возвращает значение второго аргумента.

Листинг 20.5. Использование функции IFNULL()

```
mysql> SELECT IFNULL(0,1), IFNULL(NULL,1);
+-------------+----------------+
| IFNULL(0,1) | IFNULL(NULL,1) |
+-------------+----------------+
|           0 | 1              |
+-------------+----------------+
mysql> SELECT IFNULL(1/0,1), IFNULL(0/1,1);
+---------------+---------------+
| IFNULL(1/0,1) | IFNULL(0/1,1) |
+---------------+---------------+
| 1             | 0.00000       |
+---------------+---------------+
```

Возвращаясь к учебной базе данных shop, заменим при выводе из таблицы users ключевое слово NULL на более понятное неискушенному пользователю слово 'отсутствует' (листинг 20.6).

Листинг 20.6. Замена ключевого слова NULL на 'отсутствует'

```
mysql> SELECT phone, email, url FROM users;
+------------+----------------------+------------------------+
| phone      | email                | url                    |
+------------+----------------------+------------------------+
| 58-98-78   | ivanov@email.ru      | NULL                   |
| 9057777777 | losev@email.ru       | NULL                   |
| 9056666100 | simdyanov@softtime.ru | http://www.softtime.ru/ |
| NULL       | kuznetsov@softtime.ru | http://www.softtime.ru  |
| NULL       | NULL                 | NULL                   |
| 89-78-36   | korneev@domen.ru     | NULL                   |
+------------+----------------------+------------------------+
mysql> SELECT IFNULL(phone,'отсутствует') AS phone,
    ->        IFNULL(email,'отсутствует') AS email,
    ->        IFNULL(url,'отсутствует') AS url
    -> FROM users;
+-------------+----------------------+------------------------+
| phone       | email                | url                    |
+-------------+----------------------+------------------------+
| 58-98-78    | ivanov@email.ru      | отсутствует            |
| 9057777777  | losev@email.ru       | отсутствует            |
| 9056666100  | simdyanov@softtime.ru | http://www.softtime.ru/ |
```

```
| отсутствует | kuznetsov@softtime.ru | http://www.softtime.ru  |
| отсутствует | отсутствует           | отсутствует             |
| 89-78-36    | korneev@domen.ru      | отсутствует             |
+-------------+-----------------------+-------------------------+
```

20.1.4. Функция *NULLIF()*

Функция `NULLIF(expr1,expr2)` (листинг 20.7) возвращает `NULL`, если значение `expr1` совпадает с `expr2`, в противном случае возвращается значение `expr1`.

> ***Замечание***
>
> Функция `NULLIF(expr1,expr2)` аналогична выражению `CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END`.

Листинг 20.7. Использование функции `NULLIF()`

```
mysql> SELECT NULLIF(1,1),NULLIF(1,2);
+-------------+-------------+
| NULLIF(1,1) | NULLIF(1,2) |
+-------------+-------------+
|        NULL |           1 |
+-------------+-------------+
```

20.2. Информационные функции

Информационной функции возвращают различную служебную и системную информацию.

20.2.1. Функция *BENCHMARK()*

Функция `BENCHMARK(count, expr)` (листинг 20.8) повторяет выполнение выражения `expr` заданное количество раз, указанное в аргументе `count`. Она может использоваться для определения того, насколько быстро MySQL обрабатывает данное выражение. Значение результата, возвращаемого функцией, всегда равно 0. Функция предназначена для использования в клиенте mysql, который сообщает о времени выполнения запроса.

Указанное в отчете время представляет собой время, подсчитанное на стороне клиента, а не затраченное центральным процессором (CPU time) на сервере. Может оказаться целесообразным выполнить `BENCHMARK()` несколько раз, чтобы выяснить, насколько интенсивно загружен серверный компьютер.

Листинг 20.8. Использование функции BENCHMARK()

```
mysql> SELECT BENCHMARK(1000000,ENCODE("hello","goodbye"));
+---------------------------------------------+
| BENCHMARK(1000000,ENCODE("hello","goodbye")) |
+---------------------------------------------+
|                                           0 |
+---------------------------------------------+
1 row in set (0.61 sec)
```

20.2.2. Функция *CONNECTION_ID()*

Функция CONNECTION_ID() возвращает идентификатор (thread_id) для данного соединения.

Листинг 20.9. Использование функции CONNECTION_ID()

```
mysql> SELECT CONNECTION_ID();
+----------------+
| CONNECTION_ID() |
+----------------+
|               1 |
+----------------+
```

Каждое соединение имеет свой собственный уникальный идентификатор. В листинге 20.9 функция вернула идентификатор, равный единице. Если открыть еще одно консольная окно и вызвать в нем функцию CONNECTION_ID(), она вернет уже цифру два.

20.2.3. Функция *CURRENT_USER()*

Функция CURRENT_USER() возвращает текущее имя, под которым пользователь аутентифицировался в текущей сессии, и имя хоста, с которого он обращается к серверу (листинг 20.10).

Листинг 20.10. Использование функции CURRENT_USER()

```
mysql> SELECT CURRENT_USER();
+---------------+
| CURRENT_USER() |
+---------------+
| root@localhost |
+---------------+
```

20.2.4. Функция *DATABASE()*

Функция DATABASE() (листинг 20.11) возвращает имя текущей базы данных.

Листинг 20.11. Использование функции DATABASE()

```
mysql> USE test;
mysql> SELECT DATABASE();
+------------+
| DATABASE() |
+------------+
| test       |
+------------+
```

Если в данное время нет активной базы данных, то функция DATABASE() возвращает NULL.

Замечание

До версии СУБД MySQL 4.1.1 вместо NULL функция DATABASE() возвращает пустую строку.

20.2.5. Функция *FOUND_ROWS()*

Оператор SELECT может включать инструкцию LIMIT, которая ограничивает число записей, возвращаемых СУБД MySQL. Тем не менее часто, особенно при построении постраничной навигации, требуется определить число строк, которое бы возвратил оператор SELECT в отсутствие инструкции LIMIT. Для решения этой задачи предназначена функция FOUND_ROWS(). Чтобы получить корректное значение, в состав оператора SELECT необходимо включить опцию SQL_CALC_FOUND_ROWS. После этого можно вызывать функцию FOUND_ROWS().

Замечание

Функция FOUND_ROWS() и опция SQL_CALC_FOUND_ROWS доступны, начиная с версии MySQL 4.0.0.

Листинг 20.12. Использование функции FOUND_ROWS()

```
mysql> SELECT SQL_CALC_FOUND_ROWS name FROM products LIMIT 10;
+------------------------+
| name                   |
+------------------------+
| Celeron 1.8            |
| Celeron 2.0GHz         |
| Celeron 2.4GHz         |
| Celeron D 320 2.4GHz   |
| Celeron D 325 2.53GHz  |
```

```
| Celeron D 315 2.26GHz  |
| Intel Pentium 4 3.2GHz |
| Intel Pentium 4 3.0GHz |
| Intel Pentium 4 3.0GHz |
| Gigabyte GA-8I848P-RS  |
+------------------------+
mysql> SELECT FOUND_ROWS();
+--------------+
| FOUND_ROWS() |
+--------------+
|           30 |
+--------------+
```

Если опция SQL_CALC_FOUND_ROWS в операторе SELECT будет пропущена, функция вернет число строк, возвращенное оператором SELECT. Применительно к листингу 20.12 это число равно 10.

Замечание

Следует отметить, что когда используется опция SQL_CALC_FOUND_ROWS, то СУБД MySQL приходится выполнять дополнительные вычисления для того, чтобы подсчитать число строк в полном результирующем наборе (в отсутствие инструкции LIMIT).

20.2.6. Функция *LAST_INSERT_ID()*

Функция LAST_INSERT_ID() возвращает последнее автоматически сгенерированное значение для столбца, снабженного атрибутом AUTO_INCREMENT.

Замечание

Функция LAST_INSERT_ID() обсуждается также в *главе 6* при разборе синтаксиса оператора INSERT.

Следует отметить, что функция LAST_INSERT_ID() возвращает последнее сгенерированное значение в рамках данного сеанса. То есть функция вернет значение только в том случае, если новое значение было сгенерировано непосредственно перед вызовом функции LAST_INSERT_ID().

Последнее сгенерированное значение сохраняется на сервере для данного конкретного соединения и не может быть изменено другим клиентом. Оно не изменится даже при обновлении столбца AUTO_INCREMENT конкретной величиной (т. е. не равной NULL и 0). При использовании многострочного варианта оператора INSERT функция LAST_INSERT_ID() возвращает значение для первой внесенной строки. Причина этого заключается в том, что можно легко воспроизвести точно такую же команду INSERT на другом сервере. Пример использования функции LAST_INSERT_ID() приведен в листинге 20.13.

Листинг 20.13. Использование функции LAST_INSERT_ID()

```
mysql> INSERT INTO catalogs VALUES(NULL, 'Принтеры');
mysql> SELECT LAST_INSERT_ID();
+------------------+
| LAST_INSERT_ID() |
+------------------+
|                6 |
+------------------+
```

Функция LAST_INSERT_ID() может принимать параметр expr, который возвращается функцией и запоминается как следующее значение, которое функция LAST_INSERT_ID() вернет при повторном вызове. Это можно использовать для создания последовательностей (листинг 20.14).

Листинг 20.14. Создание последовательностей

```
mysql> CREATE TABLE sequence (id INT NOT NULL);
mysql> INSERT INTO sequence VALUES (0);
mysql> UPDATE sequence SET id=LAST_INSERT_ID(id+1);
```

Оператор UPDATE увеличивает счетчик последовательности и заставляет следующий вызов LAST_INSERT_ID() возвращать измененное значение. Для столбца, снабженного атрибутом AUTO_INCREMENT, последовательность может быть сгенерирована автоматически без вызова функции LAST_INSERT_ID().

Замечание

Управлять значениями, которые будут возвращаться функцией LAST_INSERT_ID(), можно при помощи системных переменных IDENTITY, INSERT_ID и LAST_INSERT_ID, значения которых устанавливаются при помощи оператора SET *(см. раздел 27.1)*. Просмотреть текущее значение системных переменных можно при помощи запроса вида SELECT @@IDENTITY, в котором значение системной переменной предваряется двойным знаком @.

Вернуть последнюю вставленную при помощи оператора INSERT запись можно при помощи запроса, представленного в листинге 20.15.

Листинг 20.15. Последняя вставленная запись

```
mysql> SELECT * FROM catalogs WHERE id_catalog = LAST_INSERT_ID();
+------------+----------+
| id_catalog | name     |
+------------+----------+
|          6 | Принтеры |
+------------+----------+
```

Однако механизм AUTO_INCREMENT позволяет также в этом случае использовать условие id_catalog IS NULL (листинг 20.16).

Листинг 20.16. Альтернативный способ извлечения последней записи

```
mysql> SELECT * FROM catalogs WHERE id_catalog IS NULL;
+------------+----------+
| id_catalog | name     |
+------------+----------+
|          6 | Принтеры |
+------------+----------+
```

Такое поведение СУБД MySQL организовано для совместимости с другими СУБД. Оно может быть отключено установкой системной переменной SQL_AUTO_IS_NULL в 0 (листинг 20.17).

Листинг 20.17. Управление системной переменной SQL_AUTO_IS_NULL

```
mysql> SELECT @@SQL_AUTO_IS_NULL;
+--------------------+
| @@sql_auto_is_null |
+--------------------+
|                  1 |
+--------------------+
mysql> SET SQL_AUTO_IS_NULL = 1;
```

Для того чтобы извлечь текущую системную переменную, ее имя следует предварить двумя символами @. Более подробно системные переменные и синтаксис оператора SET обсуждается в *главе 27*.

20.2.7. Функция *USER()*

Функция USER() (листинг 20.18) возвращает имя текущего пользователя и имя хоста, с которого он обращается к СУБД MySQL.

Замечание

Функция USER() имеет два синонима: SESSION_USER() и SYSTEM_USER().

Листинг 20.18. Использование функции USER()

```
mysql> SELECT USER();
+----------------+
| USER()         |
+----------------+
| root@localhost |
+----------------+
```

20.2.8. Функция *VERSION()*

Функция `VERSION()` возвращает строку, содержащую информацию о версии сервера MySQL. Если текущая версия находится в стадии альфа-, бета- или гамма-тестирования, это будет отмечено в отчете функции (листинг 20.19).

Листинг 20.19. Использование функции `VERSION()`

```
mysql> SELECT VERSION();
+---------------+
| VERSION()     |
+---------------+
| 5.0.6-beta-nt |
+---------------+
```

20.3. Разные функции

В англоязычной литературе этот вид функций обозначается как Miscellaneous-функции, что переводится как смешанные функции, т. е. функции, которые не поддаются классификации.

20.3.1. Функция *DEFAULT()*

Функция `DEFAULT(col_name)` возвращает значение по умолчанию для столбца `col_name`, которое назначается при помощи ключевого слова `DEFAULT` *(см. главу 11)*.

> ***Замечание***
>
> Функция `DEFAULT()` введена в СУБД MySQL, начиная с версии 4.1.0. В версиях, начиная с 5.0.2, генерируется ошибка, если функция применяется к столбцу без значения по умолчанию.

Листинг 20.20. Использование функции `DEFAULT()`

```
mysql> DESCRIBE orders;
+------------+----------+------+-----+---------------------+------------+
| Field      | Type     | Null | Key | Default             | Extra      |
+------------+----------+------+-----+---------------------+------------+
| id_order   | int(11)  | NO   | PRI | NULL               |auto_increment|
| id_user    | int(11)  | NO   | MUL | 0                   |            |
| ordertime  | datetime | NO   |     | 0000-00-00 00:00:00 |            |
| number     | int(11)  | NO   |     | 0                   |            |
| id_product | int(11)  | NO   | MUL | 0                   |            |
+------------+----------+------+-----+---------------------+------------+
```

```
mysql> SELECT DEFAULT(ordertime) FROM orders LIMIT 1;
+---------------------+
| DEFAULT(ordertime)  |
+---------------------+
| 0000-00-00 00:00:00 |
+---------------------+
mysql> SELECT DEFAULT(id_order) FROM orders LIMIT 1;
+-------------------+
| DEFAULT(id_order) |
+-------------------+
|                 0 |
+-------------------+
```

Как видно из листинга 20.20, при использовании функции `DEFAULT()` совместно с оператором `SELECT` требуется ограничивать вывод при помощи инструкции `LIMIT`. В противном случае число строк в результирующей таблице будет совпадать с числом строк в таблице `orders`.

20.3.2. Функция *GET_LOCK()*

Функция `GET_LOCK(str, timeout)` предназначена для получения блокировки для имени `str` со временем ожидания ответа сервера `timeout` секунд и возвращает 1 (истина) в случае успешной установки блокировки на имя `str` и 0 (ложь), если время ожидания ответа превысило величину `timeout`.

Замечание

Функция может вернуть значение `NULL`, если произошла ошибка, такая как переполнение памяти или уничтожение потока.

Снятие блокировки осуществляется при помощи функции `RELEASE_LOCK()`, которая рассматривается далее, повторным вызовом функции `GET_LOCK()`, а также в результате разрыва соединения с сервером.

Эта функция может использоваться для осуществления блокировок уровня приложения или для эмуляции блокировок записей. Имена блокируются в глобальном контексте сервера. Если имя блокировано одним клиентом, `GET_LOCK()` блокирует любой запрос другого клиента на получение блокировки с тем же именем. Это позволяет клиентам согласовать попытки доступа к общим ресурсам. Пример использования функции `GET_LOCK()` приведен в листинге 20.21.

Замечание

Имена блокировок `str` могут быть любыми и вовсе не должны совпадать с именами баз данных и таблиц, хотя это не запрещается.

Листинг 20.21. Использование функции `GET_LOCK()`

```
mysql> SELECT GET_LOCK('lock1',10), GET_LOCK('lock2',10);
```

```
+----------------------+----------------------+
| GET_LOCK('lock1',10) | GET_LOCK('lock2',10) |
+----------------------+----------------------+
|                    1 |                    1 |
+----------------------+----------------------+
mysql> SELECT RELEASE_LOCK('lock1'), RELEASE_LOCK('lock2');
+-----------------------+-----------------------+
| RELEASE_LOCK('lock1') | RELEASE_LOCK('lock2') |
+-----------------------+-----------------------+
|                  NULL |                     1 |
+-----------------------+-----------------------+
```

Следует отметить, что вызов функции `RELEASE_LOCK('lock1')` возвращает `NULL`, т. к. блокировка `'lock1'` была автоматически снята установкой блокировки `GET_LOCK('lock2',10)`.

20.3.3. Функция *INET_ATON()*

Функция `INET_ATON(address)` (листинг 20.22) принимает IP-адрес `address` и представляет его в виде целого числа. Для числа `'XXX.YYY.ZZZ.WWW'` результат функции вычисляется по формуле $XXX*256^3 + YYY*256^2 + ZZZ*256 + WWW$.

> ***Замечание***
>
> Для того чтобы можно было поместить в целочисленное поле весь диапазон IP-адресов, следует использовать тип `BIGINT`.

Листинг 20.22. Использование функции `INET_ATON()`

```
mysql> SELECT INET_ATON('62.145.69.10'), INET_ATON('127.0.0.1');
+---------------------------+------------------------+
| INET_ATON('62.145.69.10') | INET_ATON('127.0.0.1') |
+---------------------------+------------------------+
|                1049707786 |             2130706433 |
+---------------------------+------------------------+
```

Начиная с версии 4.1.2, функция `INET_ATON()` способна принимать IP-адреса в сокращенной форме, как показано в листинге 20.23.

Листинг 20.23. Работа с сокращенной формой IP-адреса

```
mysql> SELECT INET_ATON('127.0.01'), INET_ATON('127.1');
+-----------------------+--------------------+
| INET_ATON('127.0.01') | INET_ATON('127.1') |
+-----------------------+--------------------+
|            2130706433 |         2130706433 |
+-----------------------+--------------------+
```

20.3.4. Функция *INET_NTOA()*

Функция `INET_NTOA(address)` принимает IP-адрес в виде числа (результат выполнения функции `INET_ATON()`) и возвращает адрес в виде строки, состоящий из четырех чисел, разделенных точкой (листинг 20.24).

Листинг 20.24. Работа с функцией `INET_NTOA()`

```
mysql> SELECT INET_NTOA(1049707786), INET_NTOA(2130706433);
+-----------------------+-----------------------+
| INET_NTOA(1049707786) | INET_NTOA(2130706433) |
+-----------------------+-----------------------+
| 62.145.69.10          | 127.0.0.1             |
+-----------------------+-----------------------+
```

20.3.5. Функция *IS_FREE_LOCK()*

Функция `IS_FREE_LOCK(str)` (листинг 20.25) проверяет, свободна ли блокировка с именем `str`, которая устанавливается при помощи функции `GET_LOCK()`, рассмотренной ранее. Функция возвращает 1 (истина), если блокировка свободна (никем не используется), и 0 (ложь), если занята.

Замечание

Функция может вернуть значение `NULL`, если произошла ошибка, такая как переполнение памяти или уничтожение потока.

Листинг 20.25. Использование функции `IS_FREE_LOCK()`

```
mysql> SELECT GET_LOCK('lock1',10), GET_LOCK('lock2',10);
+----------------------+----------------------+
| GET_LOCK('lock1',10) | GET_LOCK('lock2',10) |
+----------------------+----------------------+
|                    1 |                    1 |
+----------------------+----------------------+
mysql> SELECT IS_FREE_LOCK('lock1'), IS_FREE_LOCK('lock2');
+-----------------------+-----------------------+
| IS_FREE_LOCK('lock1') | IS_FREE_LOCK('lock2') |
+-----------------------+-----------------------+
|                     1 |                     0 |
+-----------------------+-----------------------+
```

Следует отметить, что вызов функции `IS_FREE_LOCK('lock1')` возвращает 1 (истина), т. к. блокировка `'lock1'` была автоматически снята установкой блокировки `GET_LOCK('lock2',10)`.

20.3.6. Функция *IS_USED_LOCK()*

Функция `IS_USED_LOCK(str)` (листинг 20.26) проверяет, установлена ли блокировка с именем `str`. В случае, если блокировка установлена, функция возвращает идентификатор соединения клиента, который удерживает блокировку. Если блокировка не установлена, возвращается `NULL`.

Листинг 20.26. Использование функции `IS_USED_LOCK()`

```
mysql> SELECT GET_LOCK('lock1',10), GET_LOCK('lock2',10);
+----------------------+----------------------+
| GET_LOCK('lock1',10) | GET_LOCK('lock2',10) |
+----------------------+----------------------+
|                    1 |                    1 |
+----------------------+----------------------+
mysql> SELECT IS_USED_LOCK('lock1'), IS_USED_LOCK('lock2');
+-----------------------+-----------------------+
| IS_USED_LOCK('lock1') | IS_USED_LOCK('lock2') |
+-----------------------+-----------------------+
|                  NULL |                     2 |
+-----------------------+-----------------------+
```

Замечание

Функция `IS_USED_LOCK()` введена в СУБД MySQL, начиная с версии 4.1.0.

20.3.7. Функция *RELEASE_LOCK()*

Функция `RELEASE_LOCK(str)` снимает блокировку с именем `str`, которая была установлена функцией `GET_LOCK()`. Функция возвращает 1 (истина), если блокировка успешно снята, 0 (ложь), если блокировка установлена другим потоком и не может быть снята, и `NULL`, если блокировки с таким именем не существует.

20.3.8. Функция *UUID()*

Функция `UUID()` (листинг 20.27) возвращает Универсальный Уникальный Идентификатор (Universal Unique Identifier — UUID), сгенерированный в соответствии со спецификацией "DCE 1.1: Удаленный вызов процедур" (Приложение A), опубликованной Open Group в 1987 году.

Идентификатор UUID реализован в виде числа, которое является глобально уникальным во времени и пространстве. Два вызова функции `UUID()` вернут два разных значения, если они производятся одновременно на двух разных компьютерах или на одном и том же компьютере в разное время.

Замечание

Функция `UUID()` введена в СУБД MySQL, начиная с версии 4.1.2.

Листинг 20.27. Использование функции `UUID()`

```
mysql> SELECT UUID();
+--------------------------------------+
| UUID()                               |
+--------------------------------------+
| 6627c9e4-2fec-1028-9007-551f9bad91cf |
+--------------------------------------+
mysql> SELECT UUID();
+--------------------------------------+
| UUID()                               |
+--------------------------------------+
| 6975f814-2fec-1028-9007-551f9bad91cf |
+--------------------------------------+
```

Идентификатор UUID — это 128-разрядное число, представленное в виде строки, состоящей из пяти шестнадцатеричных чисел в формате 'aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee'.

- Первые три числа генерируются на основе текущего времени.
- Четвертое число предохраняет темпоральную уникальность в случае, если значение временной метки теряет монотонность (постоянное увеличение со временем), например, из-за перехода на летнее время и обратно.
- Пятое число — это номер узла IEEE 802, т. е. аппаратный MAC-адрес платы Ethernet, который уникален для каждой платы. Если хост не имеет сетевой платы или отсутствует возможность его получения, подставляется случайное число.

Замечание

В настоящее время MAC-адрес интерфейса принимается во внимание только в средах FreeBSD и Linux. В других операционных системах MySQL использует случайно сгенерированное 48-разрядное число.

ГЛАВА 21

Переменные и временные таблицы

Часто результаты запроса необходимо использовать в последующих запросах. Для этого полученные данные необходимо сохранить во временных структурах. Эту задачу решают переменные SQL и временные таблицы, синтаксис которых рассматривается в данной главе.

21.1. Переменные SQL

СУБД MySQL предоставляет возможность сохранения результатов текущего запроса для использования в следующих запросах в переменных SQL. Объявление переменной начинается с символа @, за которым следует имя переменной. Значения переменным присваиваются посредством оператора `SELECT` с использованием оператора присваивания `":="`.

Листинг 21.1. Объявление и использование переменных SQL

```
mysql> SELECT @total := COUNT(*) FROM products;
+--------------------+
| @total := COUNT(*) |
+--------------------+
|                 30 |
+--------------------+
mysql> SELECT @total;
+--------+
| @total |
+--------+
|     30 |
+--------+
```

В листинге 21.1 объявляется переменная `@total`, которой присваивается число записей в таблице `products` учебной базы данных `shop`. Затем в рамках текущего сеанса в последующих запросах появляется возможность использования данной переменной.

Замечание

Переменная действует только в рамках одного сеанса соединения с сервером MySQL и прекращает свое существование после разрыва соединения.

В листинге 21.2 из таблицы `products` извлекается информация о товарной позиции, обладающей самой высокой ценой.

Листинг 21.2. Извлечение товарной позиции с самой высокой ценой

```
mysql> SELECT @price := MAX(price) FROM products;
+----------------------+
| @price := MAX(price) |
+----------------------+
|              7259.00 |
+----------------------+
mysql> SELECT id_product, name, price, count
    -> FROM products WHERE price = @price;
+------------+-----------------------+---------+-------+
| id_product | name                  | price   | count |
+------------+-----------------------+---------+-------+
|          7 | Intel Pentium 4 3.2GHz | 7259.00 |     5 |
+------------+-----------------------+---------+-------+
```

Если в качестве значения переменной передается имя столбца, то переменная получит последнее значение (листинг 21.3).

Листинг 21.3. Передача в качестве значения переменной имени столбца

```
mysql> SELECT @id := id_catalog FROM catalogs;
+-------------------+
| @id := id_catalog |
+-------------------+
|                 1 |
|                 2 |
|                 3 |
|                 4 |
|                 5 |
+-------------------+
mysql> SELECT @id;
+------+
| @id  |
+------+
|    5 |
+------+
mysql> SELECT * FROM catalogs WHERE id_catalog = @id;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          5 | Оперативная память |
+------------+--------------------+
```

В качестве значения переменной необязательно должны выступать результаты выполнения функций и значения столбцов таблицы, можно использовать произвольные числовые и строковые значения (листинг 21.4).

Листинг 21.4. Использование для инициализации переменной числового значения

```
mysql> SELECT @id := 3;
+----------+
| @id := 3 |
+----------+
|        3 |
+----------+
mysql> SELECT * FROM catalogs WHERE id_catalog = @id;
+------------+---------------+
| id_catalog | name          |
+------------+---------------+
|          3 | Видеоадаптары |
+------------+---------------+
```

Следует помнить, что имена переменных чувствительны к регистру (листинг 21.5).

Листинг 21.5. Имена переменных чувствительны к регистру

```
mysql> SELECT @id := 5, @ID := 3;
+----------+----------+
| @id := 5 | @ID := 3 |
+----------+----------+
|        5 |        3 |
+----------+----------+
mysql> SELECT @id, @ID;
+------+------+
| @id  | @ID  |
+------+------+
|    5 |    3 |
+------+------+
```

Переменные также могут объявляться при помощи оператора `SET`, как это продемонстрировано в листинге 21.6.

Замечание

При использовании оператора `SET` в качестве оператора присваивания может выступать как "`:=`", так и обычный знак равенства "=".

Замечание

Синтаксис оператора `SET` рассматривается в *главе 27*.

Листинг 21.6. Использование оператора SET для объявления переменной

```
mysql> SET @last = CURDATE() - INTERVAL 7 DAY;
mysql> SELECT CURDATE(), @last;
+------------+------------+
| CURDATE()  | @last      |
+------------+------------+
| 2005-06-18 | 2005-06-11 |
+------------+------------+
```

Удобство использования оператора SET заключается в том, что он, в отличие от оператора SELECT, не возвращает результирующую таблицу.

При использовании переменных следует помнить, что не рекомендуется одновременно присваивать и использовать переменную в одном запросе, т. к. это может приводить к ошибкам, которые трудно обнаружить.

Листинг 21.7. Особенности присвоения и использования переменных в одном запросе

```
mysql> SET @a = 'test';
mysql> SELECT @a, (@a := 20);
+------+------------+
| @a   | (@a := 20) |
+------+------------+
| test |         20 |
+------+------------+
mysql> SET @a = 'test';
mysql> SELECT (@a := 20), @a;
+------------+------+
| (@a := 20) | @a   |
+------------+------+
|         20 | 20   |
+------------+------+
```

Как видно из листинга 21.7, порядок следования выражений с участием переменных имеет значение для результата, возвращаемого запросом.

21.2. Временные таблицы

Переменная SQL позволяет сохранить одно промежуточное значение. Когда необходимо сохранить результирующую таблицу, прибегают к временным таблицам.

Создание временных таблиц осуществляется при помощи оператора CREATE TEMPORARY TABLE, синтаксис которого ничем не отличается от синтаксиса оператора CREATE TABLE *(см. главу 11)*.

Временная таблица автоматически удаляется по завершении соединения с сервером, а ее имя действительно только в течение данного соединения. Это означает, что два разных клиента могут использовать временные таблицы с одинаковыми именами без конфликта друг с другом или с существующей таблицей с тем же именем (существующая таблица остается скрытой, пока не будет удалена временная таблица).

Листинг 21.8. Создание временной таблицы `temp`

```
mysql> CREATE TEMPORARY TABLE temp (id_catalog INT, name TINYINT);
mysql> SHOW TABLES;
+----------------+
| Tables_in_shop |
+----------------+
| catalogs       |
| orders         |
| products       |
| users          |
+----------------+
mysql> DESCRIBE temp;
+------------+------------+------+-----+---------+-------+
| Field      | Type       | Null | Key | Default | Extra |
+------------+------------+------+-----+---------+-------+
| id_catalog | int(11)    | YES  |     | NULL    |       |
| name       | tinyint(4) | YES  |     | NULL    |       |
+------------+------------+------+-----+---------+-------+
```

В листинге 21.8 демонстрируется создание таблицы `temp`. Если возникает необходимость удалить таблицу до момента завершения соединения с сервером, можно воспользоваться оператором `DROP TABLE` (листинг 21.9).

Листинг 21.9. Удаление временной таблицы `temp`

```
mysql> DROP TABLE temp;
```

Временные таблицы можно заполнять традиционными способами, описанными в *главе 6*, но наиболее удобно производить заполнение таких таблиц при помощи вложенных запросов, когда результат запроса `SELECT` непосредственно помещается во временную таблицу (листинг 21.10).

Замечание

Более подробно вложенные запросы рассматриваются в *главе 23*.

Листинг 21.10. Создание временной таблицы `temp` при помощи вложенного запроса

```
mysql> CREATE TEMPORARY TABLE temp
    -> SELECT id_catalog, COUNT(id_catalog) AS total
    -> FROM products GROUP BY id_catalog;
```

```
mysql> SELECT * FROM temp;
+------------+-------+
| id_catalog | total |
+------------+-------+
|          1 |     9 |
|          2 |     6 |
|          3 |     4 |
|          4 |     5 |
|          5 |     6 |
+------------+-------+
```

Следует обратить внимание на тот факт, что при таком способе создания таблицы не требуется определять структуру таблицы `temp` — она автоматически принимает структуру результирующей таблицы оператора `SELECT`.

Временная таблица `temp` может выступать в качестве предмета запроса. В листинге 21.11 подсчитывается сумма значений столбца `total`.

Листинг 21.11. Запрос к временной таблице `temp`

```
mysql> SELECT SUM(total) FROM temp;
+------------+
| SUM(total) |
+------------+
|         30 |
+------------+
```

Поместив в переменную `@total` общее число товарных позиций в таблице `products`, можно получить процентное соотношение числа товарных позиций в разных отделах учебного электронного магазина (листинг 21.12).

Листинг 21.12. Процентное отношение числа товарных позиций по отделам

```
mysql> SELECT @total := SUM(total) FROM temp;
+----------------------+
| @total := SUM(total) |
+----------------------+
|                   30 |
+----------------------+
mysql> SELECT id_catalog, FORMAT((total/@total)*100,2) AS percent
    -> FROM temp;
+------------+---------+
| id_catalog | percent |
+------------+---------+
|          1 | 30.00   |
|          2 | 20.00   |
|          3 | 13.33   |
```

```
|          4 | 16.67   |
|          5 | 20.00   |
+------------+---------+
```

Системная переменная `BIG_TABLES` позволяет сообщить СУБД MySQL, где должны храниться временные таблицы: в оперативной памяти или на диске. Если переменная устанавливается в 1, все временные таблицы сохраняются на диске, если в 0, то в оперативной памяти. Установить новое значение переменной можно при помощи оператора `SET` (листинг 21.13).

Листинг 21.13. Управление системной переменной `BIG_TABLES`

```
mysql> SELECT @@BIG_TABLES;
+--------------+
| @@BIG_TABLES |
+--------------+
|            0 |
+--------------+
mysql> SET BIG_TABLES = 1;
```

Для того чтобы извлечь текущее значение системной переменной, ее имя следует предварить двумя символами @. Более подробно системные переменные и синтаксис оператора `SET` обсуждаются в *главе 27*.

Замечание

Начиная с версии MySQL 4.0.0, ручное управление этой переменной, скорее всего, не потребуется, т. к. MySQL сама принимает решение о том, где размещать временную таблицу.

Замечание

В MySQL 5.0.1 появилась серьезная альтернатива временным таблицам в виде представлений *(см. главу 30)*. На самом деле представления часто также являются временными таблицами, только об их создании и уничтожении заботится СУБД MySQL.

ГЛАВА 22

Многотабличные запросы

Рассмотренные до этой главы SQL-запросы использовали одну таблицу. Но в реальных приложениях часто требуется использовать сразу несколько таблиц базы данных. Запросы, которые обращаются одновременно к нескольким таблицам, называются многотабличными запросами. Именно в таких запросах проявляется одно из преимуществ реляционных баз данных — связь таблиц друг с другом.

22.1. Перекрестное объединение таблиц

Для обсуждения многотабличных запросов создадим две таблицы: `tbl1` и `tbl2`, каждая из которых будет содержать два столбца — числовой и текстовый. В листинге 22.1 представлены операторы `CREATE TABLE`, создающие указанные таблицы.

Листинг 22.1. Создание таблиц `tbl1` и `tbl2`

```
CREATE TABLE tbl1 (
  id int(11) NOT NULL default '0',
  name tinytext NOT NULL
) ENGINE=MyISAM;
INSERT INTO tbl1 VALUES (1, 'А');
INSERT INTO tbl1 VALUES (2, 'Б');
INSERT INTO tbl1 VALUES (3, 'В');
CREATE TABLE tbl2 (
  id int(11) NOT NULL default '0',
  letter tinytext NOT NULL
) ENGINE=MyISAM;
INSERT INTO tbl2 VALUES (2, 'Г');
INSERT INTO tbl2 VALUES (3, 'Д');
INSERT INTO tbl2 VALUES (4, 'Е');
```

После создания таблиц заполним каждую таблицу тремя записями, состоящими из целого числа и соответствующей ему буквы русского алфавита.

Для осуществления запроса сразу к нескольким таблицам их имена перечисляются после ключевого слова `FROM` через запятую. В листинге 22.2 представлена выборка из каждой таблицы, а также двухтабличный запрос с использованием оператора `SELECT` к таблицам `tbl1`, `tbl2`.

Листинг 22.2. Обычные и двухтабличный `SELECT`-запрос к таблицам `tbl1` и `tbl2`

```
mysql> SELECT * FROM tbl1;
+----+------+
| id | name |
+----+------+
|  1 | А    |
|  2 | Б    |
|  3 | В    |
+----+------+
mysql> SELECT * FROM tbl2;
+----+--------+
| id | letter |
+----+--------+
|  2 | Г      |
|  3 | Д      |
|  4 | Е      |
+----+--------+
mysql> SELECT * FROM tbl1, tbl2;
+----+------+----+--------+
| id | name | id | letter |
+----+------+----+--------+
|  1 | А    |  2 | Г      |
|  2 | Б    |  2 | Г      |
|  3 | В    |  2 | Г      |
|  1 | А    |  3 | Д      |
|  2 | Б    |  3 | Д      |
|  3 | В    |  3 | Д      |
|  1 | А    |  4 | Е      |
|  2 | Б    |  4 | Е      |
|  3 | В    |  4 | Е      |
+----+------+----+--------+
```

Двухтабличный запрос, представленный в листинге 22.2, называют также *перекрестным объединением*. При таком объединении каждая строка одной таблицы объединяется с каждой строкой другой таблицы, создавая тем самым все возможные комбинации строк обеих таблиц. Результирующая таблица содержит число столбцов, равное сумме столбцов в объединяемых таблицах. Таблицы `tbl1` и `tbl2` содержат по два столбца, поэтому результирующая таблица содержит 4 столбца (2 + 2 = 4). Число записей в результирующей таблице определяется произведением числа записей в таблицах, участвующих в многотабличном запросе (3*3 = 9).

Точно так же, как и в обычном операторе SELECT, в многотабличном операторе SELECT можно управлять числом столбцов в результирующей таблице. Для этого столбцы, которые должны войти в результирующую таблицу, перечисляются после ключевого слова SELECT через запятую (листинг 22.3).

Листинг 22.3. Управление столбцами результирующей таблицы

```
mysql> SELECT name, letter FROM tbl1, tbl2;
+------+--------+
| name | letter |
+------+--------+
| А    | Г      |
| Б    | Г      |
| В    | Г      |
| А    | Д      |
| Б    | Д      |
| В    | Д      |
| А    | Е      |
| Б    | Е      |
| В    | Е      |
+------+--------+
```

Однако если в результирующую таблицу добавить столбец id, который входит в состав обеих таблиц, СУБД MySQL вернет ошибку — неоднозначность поля id в списке полей (листинг 22.4).

Листинг 22.4. Неоднозначность поля id

```
mysql> SELECT id, name, letter FROM tbl1, tbl2;
ERROR 1052: Column: 'id' in field list is ambiguous
```

Для того чтобы исключить неоднозначность, т. е. определить, поле id какой таблицы имеется в виду в запросе из листинга 22.4, прибегают к полным именам столбцов. Полное имя включает имена таблицы и столбца, разделенные точкой (листинг 22.5).

Листинг 22.5. Использование полных имен

```
mysql> SELECT tbl1.id, tbl1.name, tbl2.letter FROM tbl1, tbl2;
+----+------+--------+
| id | name | letter |
+----+------+--------+
|  1 | А    | Г      |
|  2 | Б    | Г      |
|  3 | В    | Г      |
|  1 | А    | Д      |
|  2 | Б    | Д      |
```

```
|  3 | В    | Д      |
|  1 | А    | Е      |
|  2 | Б    | Е      |
|  3 | В    | Е      |
+----+------+--------+
```

Для символа "*", указывающего на необходимость выборки всех столбцов таблицы, также можно использовать полное имя: `tbl1.*` и `tbl2.*` (листинг 22.6).

Листинг 22.6. Использование полного имени совместно с символом "*"

```
mysql> SELECT tbl1.*, tbl2.* FROM tbl1, tbl2;
+----+------+----+--------+
| id | name | id | letter |
+----+------+----+--------+
|  1 | А    |  2 | Г      |
|  2 | Б    |  2 | Г      |
|  3 | В    |  2 | Г      |
|  1 | А    |  3 | Д      |
|  2 | Б    |  3 | Д      |
|  3 | В    |  3 | Д      |
|  1 | А    |  4 | Е      |
|  2 | Б    |  4 | Е      |
|  3 | В    |  4 | Е      |
+----+------+----+--------+
mysql> SELECT tbl1.* FROM tbl1, tbl2;
+----+------+
| id | name |
+----+------+
|  1 | А    |
|  2 | Б    |
|  3 | В    |
|  1 | А    |
|  2 | Б    |
|  3 | В    |
|  1 | А    |
|  2 | Б    |
|  3 | В    |
+----+------+
```

Замечание

Полные имена можно использовать для обращения не только в пределах одной базы данных, но и для объединения таблиц из двух разных баз данных. Для этого полное имя столбца предваряется именем базы данных. Если таблицы `tbl1` и `tbl2` расположены в базах данных `db1` и `db2`, то к столбцам `name` и `letter` можно обратиться по именам `db1.tbl1.name` и `db2.tbl2.letter` соответственно. При этом названия таблиц следует также записывать полным именем: `db1.tbl1` и `db2.tbl2`.

В реальных запросах редко требуется выводить всевозможные комбинации строк объединяемых таблиц. Чаще число строк в результирующей таблице ограничивается при помощи условия `WHERE`. В листинге 22.7 производится выборка из таблиц `tbl1` и `tbl2` при условии совпадения полей `id` этих таблиц.

Листинг 22.7. Ограничение числа строк при помощи условия `WHERE`

```
mysql> SELECT tbl1.id, tbl1.name, tbl2.letter FROM tbl1, tbl2
    -> WHERE tbl1.id = tbl2.id;
+----+------+--------+
| id | name | letter |
+----+------+--------+
|  2 | Б    | Г      |
|  3 | В    | Д      |
+----+------+--------+
```

Еще одним эффективным способом ограничения числа столбцов является группировка результата выборки по одному из полей при помощи конструкции `GROUP BY` (листинг 22.8).

Листинг 22.8. Ограничение числа строк при помощи конструкции `GROUP BY`

```
mysql> SELECT tbl1.id, tbl1.name, tbl2.letter FROM tbl1, tbl2
    -> GROUP BY tbl1.id;
+----+------+--------+
| id | name | letter |
+----+------+--------+
|  1 | А    | Г      |
|  2 | Б    | Г      |
|  3 | В    | Г      |
+----+------+--------+
```

При работе с результирующей таблицей в прикладных программах часто бывает неудобно работать с полными именами столбцов. Поэтому прибегают к назначению псевдонимов при помощи оператора `AS`. В листинге 22.9 полному имени `tbl1.id` назначается псевдоним `id`, имени `tbl1.name` — `name`, а `tbl2.letter` — `letter`.

Листинг 22.9. Назначение псевдонимов

```
mysql> SELECT tbl1.id AS id, tbl1.name AS name, tbl2.letter As letter
    -> FROM tbl1, tbl2 WHERE tbl1.id = tbl2.id;
+----+------+--------+
| id | name | letter |
+----+------+--------+
|  2 | Б    | Г      |
|  3 | В    | Д      |
+----+------+--------+
```

Оператор AS может использоваться не только для назначения псевдонимов столбцам, но и для назначения псевдонимов таблицам. В листинге 22.10 таблицам tbl1 и tbl2 назначаются псевдонимы t1 и t2 соответственно. Такой подход позволяет использовать в качестве имен таблиц более короткие имена, что приводит к более короткому SQL-запросу.

Листинг 22.10. Назначение псевдонимов таблицам

```
mysql> SELECT t1.id, t1.name, t2.letter
    -> FROM tbl1 AS t1, tbl2 AS t2 WHERE t1.id = t2.id;
+----+------+--------+
| id | name | letter |
+----+------+--------+
|  2 | Б    | Г      |
|  3 | В    | Д      |
+----+------+--------+
```

Замечание

После назначения псевдонимов таблицам, участвующим в запросе, использовать исходные имена таблиц в конструкциях WHERE, GROUP BY, ORDER BY и списке столбцов после ключевого слова SELECT уже не допускается.

Кроме того, такое назначение псевдонимов позволяет осуществлять многотабличные запросы к одной таблице. Такие запросы называют еще самообъединением таблицы. Для этого достаточно назначить одной и той же таблице разные псевдонимы (листинг 22.11).

Листинг 22.11. Многотабличный запрос к одной таблице

```
mysql> SELECT t1.id, t2.name FROM tbl1 AS t1, tbl1 AS t2
    -> WHERE t1.id = t2.id;
+----+------+
| id | name |
+----+------+
|  1 | А    |
|  2 | Б    |
|  3 | В    |
+----+------+
```

Рассмотрим более реальный пример. Пусть требуется вывести названия и цены всех товарных позиций каталога, которому принадлежит товарная позиция 'Maxtor 6Y120P0'. То есть, обнаружив в нашем магазине один жесткий диск, мы хотим выяснить, какие еще жесткие диски имеются в продаже и сравнить их по цене. Данную задачу решает запрос, представленный в листинге 22.12.

Листинг 22.12. Список жестких дисков

```
mysql> SELECT products.name, products.price
    -> FROM products, products AS prd
    -> WHERE products.id_catalog = prd.id_catalog AND
    ->       prd.name = 'Maxtor 6Y120P0';
+-----------------------------+---------+
| name                        | price   |
+-----------------------------+---------+
| Maxtor 6Y120P0              | 2456.00 |
| Maxtor 6B200P0              | 3589.00 |
| Samsung SP0812C             | 2093.00 |
| Seagate Barracuda ST3160023A | 3139.00 |
| Seagate ST3120026A          | 2468.00 |
+-----------------------------+---------+
```

В листинге 22.12 происходит самообъединение таблицы `products`, при этом в полных именах списка столбцов после ключевого слова `SELECT` следует использовать имя таблицы, не участвующей в сравнении со строкой '`Maxtor 6Y120P0`', иначе будет возвращено пять строк с данным названием (листинг 22.13).

Листинг 22.13. Ошибочное выполнение запроса

```
mysql> SELECT prd.name, prd.price FROM products, products AS prd
    -> WHERE products.id_catalog = prd.id_catalog AND
    -> prd.name = 'Maxtor 6Y120P0';
+----------------+---------+
| name           | price   |
+----------------+---------+
| Maxtor 6Y120P0 | 2456.00 |
| Maxtor 6Y120P0 | 2456.00 |
| Maxtor 6Y120P0 | 2456.00 |
| Maxtor 6Y120P0 | 2456.00 |
| Maxtor 6Y120P0 | 2456.00 |
+----------------+---------+
```

Другим применением самообъединения является вычисление разницы между последовательными строками. В листинге 22.14 представлено содержимое таблицы `orders`.

Листинг 22.14. Содержимое таблицы `orders`

```
mysql> SELECT id_order, ordertime FROM orders ORDER BY ordertime;
+----------+---------------------+
| id_order | ordertime           |
+----------+---------------------+
|        1 | 2005-01-04 10:39:38 |
|        2 | 2005-02-10 09:40:29 |
```

```
|         3 | 2005-02-18 13:41:05 |
|         4 | 2005-03-10 18:20:00 |
|         5 | 2005-03-17 19:15:36 |
+-----------+---------------------+
```

Пусть требуется вычислить разницу в днях между заказами в электронном магазине. Решить эту задачу можно при помощи запроса, представленного в листинге 22.15.

Листинг 22.15. Вычисления разницы в днях между заказами

```
mysql> SELECT ord1.id_order AS id1,
    -> ord2.id_order AS id2,
    -> ord1.ordertime, ord2.ordertime,
    -> TO_DAYS(ord2.ordertime) - TO_DAYS(ord1.ordertime) AS different
    -> FROM orders AS ord2, orders AS ord1
    -> WHERE ord1.id_order + 1 = ord2.id_order
    -> ORDER BY ord1.ordertime;
+-----+-----+---------------------+---------------------+-----------+
| id1 | id2 | ordertime           | ordertime           | different |
+-----+-----+---------------------+---------------------+-----------+
|   1 |   2 | 2005-01-04 10:39:38 | 2005-02-10 09:40:29 |        37 |
|   2 |   3 | 2005-02-10 09:40:29 | 2005-02-18 13:41:05 |         8 |
|   3 |   4 | 2005-02-18 13:41:05 | 2005-03-10 18:20:00 |        20 |
|   4 |   5 | 2005-03-10 18:20:00 | 2005-03-17 19:15:36 |         7 |
+-----+-----+---------------------+---------------------+-----------+
```

Как видно из листинга 22.15, при самообъединении таблицы `orders` копии таблицы смещаются друг относительно друга на одну строку за счет условия `ord1.id_order + 1 = ord2.id_order`. При помощи функции `TO_DAYS()` время приводится к дням, после чего производится вычитание полученных результатов. Полученное значение `different` и представляет разницу в днях между заказами.

Рассмотрим еще несколько запросов на примере учебной базы данных `shop`. Пусть требуется решить задачу подсчета числа товарных позиций, а также общего числа товара на складе для каждого из каталогов и вывести результирующую таблицу. В *главе 19* для осуществления этой задачи предлагался запрос, представленный в листинге 22.16.

Листинг 22.16. Вывод числа товарных позиций и запасов на складе

```
mysql> SELECT id_catalog, COUNT(id_product), SUM(count) FROM products
    -> GROUP BY id_catalog;
+------------+------------------+------------+
| id_catalog | COUNT(id_product) | SUM(count) |
+------------+------------------+------------+
|          1 |                9 |         47 |
|          2 |                6 |         27 |
|          3 |                4 |         17 |
```

```
|           4 |                   5 |          26 |
|           5 |                   6 |          85 |
+-------------+--------------------+-------------+
```

Первый столбец представляет собой внешние ключи таблицы catalogs, по которым можно восстановить названия каталогов. При помощи многотабличного запроса к таблицам products и catalogs вместо данного столбца можно вывести названия каталогов, тем самым представив результат запроса в более удобном виде (листинг 22.17).

Листинг 22.17. Замена первичного ключа id_catalog на имя каталога

```
mysql> SELECT catalogs.name, COUNT(id_product), SUM(count)
    -> FROM products, catalogs
    -> WHERE products.id_catalog = catalogs.id_catalog
    -> GROUP BY products.id_catalog;
+--------------------+-------------------+------------+
| name               | COUNT(id_product) | SUM(count) |
+--------------------+-------------------+------------+
| Процессоры         |                 9 |         47 |
| Материнские платы  |                 6 |         27 |
| Видеоадаптеры      |                 4 |         17 |
| Жесткие диски      |                 5 |         26 |
| Оперативная память |                 6 |         85 |
+--------------------+-------------------+------------+
```

Обязательным условием в данном запросе является равенство полей id_catalog таблиц products и catalogs (products.id_catalog = catalogs.id_catalog). Именно при помощи данного условия осуществляется связь этих двух таблиц.

> ***Замечание***
>
> Если имя столбца является уникальным, то при перечислении его в списке столбцов и конструкциях WHERE, GROUP BY, ORDER BY можно не использовать полное имя столбца.

Среди таблиц учебной базы данных shop примечательной таблицей является таблица orders (листинг 22.18), которая содержит внешний ключ id_product для связи с таблицей products и внешний ключ id_user для связи с таблицей users. Поэтому, чтобы результирующая таблица содержала данные заказа, фамилию покупателя и название товара, требуется осуществить трехтабличный запрос.

Листинг 22.18. Содержимое таблицы orders

```
mysql> SELECT * FROM orders;
+----------+---------+---------------------+--------+------------+
| id_order | id_user | ordertime           | number | id_product |
+----------+---------+---------------------+--------+------------+
|        1 |       3 | 2005-01-04 10:39:38 |      1 |          8 |
```

```
|          2 |         6 | 2005-02-10 09:40:29 |       2 |          10 |
|          3 |         1 | 2005-02-18 13:41:05 |       4 |          20 |
|          4 |         3 | 2005-03-10 18:20:00 |       1 |          20 |
|          5 |         3 | 2005-03-17 19:15:36 |       1 |          20 |
+----------+---------+---------------------+--------+------------+
```

Запрос, результирующая таблица в результате выполнения которого содержит вместо внешних ключей `id_user` и `id_product` фамилию покупателя и название товарной позиции, представлен в листинге 22.19.

Листинг 22.19. Трехтабличный запрос

```
mysql> SELECT id_order, users.surname, ordertime, products.name
    -> FROM users, orders, products
    -> WHERE orders.id_user = users.id_user AND
    ->       orders.id_product = products.id_product;
+----------+----------+---------------------+-----------------------+
| id_order | surname  | ordertime           | name                  |
+----------+----------+---------------------+-----------------------+
|        1 | Симдянов | 2005-01-04 10:39:38 | Intel Pentium 4 3.0GHz |
|        2 | Корнеев  | 2005-02-10 09:40:29 | Gigabyte GA-8I848P-RS  |
|        3 | Иванов   | 2005-02-18 13:41:05 | Maxtor 6Y120P0         |
|        4 | Симдянов | 2005-03-10 18:20:00 | Maxtor 6Y120P0         |
|        5 | Симдянов | 2005-03-17 19:15:36 | Maxtor 6Y120P0         |
+----------+----------+---------------------+-----------------------+
```

Помимо того, что после ключевого слова `FROM` требуется указать три таблицы: `users`, `orders` и `products`, `WHERE`-условие должно включать две связи: таблицы `orders` с `users` и `orders` с `products`.

Если, кроме представленной в листинге 2.19 информации, в результирующую таблицу необходимо поместить столбец с названием каталога, к которому относится выбранный товар, потребуется осуществить четырехтабличный запрос, представленный в листинге 22.20.

Листинг 22.20. Четырехтабличный запрос

```
mysql> SELECT id_order, users.surname, products.name, catalogs.name
    -> FROM users, orders, products, catalogs
    -> WHERE orders.id_user = users.id_user AND
    ->       orders.id_product = products.id_product AND
    ->       products.id_catalog = catalogs.id_catalog;
+----------+----------+-----------------------+------------------+
| id_order | surname  | name                  | name             |
+----------+----------+-----------------------+------------------+
|        1 | Симдянов | Intel Pentium 4 3.0GHz | Процессоры       |
|        2 | Корнеев  | Gigabyte GA-8I848P-RS  | Материнские платы |
```

```
|         3 | Иванов   | Maxtor 6Y120P0         | Жесткие диски     |
|         4 | Симдянов | Maxtor 6Y120P0         | Жесткие диски     |
|         5 | Симдянов | Maxtor 6Y120P0         | Жесткие диски     |
+----------+----------+-----------------------+-------------------+
```

Здесь, помимо связи таблицы `orders` с таблицами `users` и `products`, добавляется третья связь таблицы `products` с таблицей `catalogs` (`products.id_catalog = catalogs.id_catalog`).

С увеличением количества таблиц в запросе резко возрастает объем работы, необходимой для выполнения запроса, поэтому по возможности в запросе не следует использовать больше трех-четырех таблиц.

22.2. Объединение таблиц при помощи *JOIN*

Оператор `JOIN` позволяет объединять таблицы и имеет многочисленные варианты использования, которые будут рассмотрены далее на протяжении данного раздела. В SQL-запросе данный оператор располагается между именами объединяемых таблиц после ключевого слова `FROM`. Без дополнительных ключевых слов объединение при помощи ключевого слова `JOIN` аналогично перекрестному объединению таблиц, рассмотренному в *разделе 22.1*.

Листинг 22.21. Использование ключевого слова `JOIN`

```
mysql> SELECT * FROM tbl1, tbl2 WHERE tbl1.id = tbl2.id;
+----+------+----+--------+
| id | name | id | letter |
+----+------+----+--------+
|  2 | Б    |  2 | Г      |
|  3 | В    |  3 | Д      |
+----+------+----+--------+
mysql> SELECT * FROM tbl1 JOIN tbl2 WHERE tbl1.id = tbl2.id;
+----+------+----+--------+
| id | name | id | letter |
+----+------+----+--------+
|  2 | Б    |  2 | Г      |
|  3 | В    |  3 | Д      |
+----+------+----+--------+
```

Запросы, представленные в листинге 22.21, полностью идентичны. Для формирования условия в запросах, использующих объединение `JOIN`, вместо ключевого слова `WHERE` предпочтительно использовать ключевое слово `ON`, как это продемонстрировано в листинге 22.22.

Листинг 22.22. Использование ключевого слова ON

```
mysql> SELECT * FROM tbl1 JOIN tbl2 ON tbl1.id = tbl2.id;
+----+------+----+--------+
| id | name | id | letter |
+----+------+----+--------+
|  2 | Б    |  2 | Г      |
|  3 | В    |  3 | Д      |
+----+------+----+--------+
```

Замечание

Ключевое слово JOIN имеет два синонима: CROSS JOIN и INNER JOIN.

Помимо перекрестного объединения таблиц, предусмотрено левое и правое объединение таблиц, которое осуществляется при помощи конструкций LEFT JOIN и RIGHT JOIN соответственно.

В листинге 22.21 продемонстрировано перекрестное объединение таблиц tbl1 и tbl2. При этом результирующая таблица содержит комбинации строк обеих таблиц, удовлетворяющих условию tbl1.id = tbl2.id. Левое объединение (LEFT JOIN) позволяет включить в результирующую таблицу строки "левой" таблицы tbl1, которым не нашлось соответствия в "правой" таблице tbl2 (листинг 22.23).

Листинг 22.23. Левое объединение таблиц tbl1 и tbl2

```
mysql> SELECT * FROM tbl1 LEFT JOIN tbl2 ON tbl1.id = tbl2.id;
+----+------+------+--------+
| id | name | id   | letter |
+----+------+------+--------+
|  1 | А    | NULL | NULL   |
|  2 | Б    |    2 | Г      |
|  3 | В    |    3 | Д      |
+----+------+------+--------+
```

Как видно из листинга 22.23, записи в таблице tbl1 со значением id = 1 не нашлось соответствия в таблице tbl2, т. к. поле id в ней принимает значения 2, 3, 4. Тем не менее в результирующую таблицу запись включена, при этом значения полей из таблицы tbl2 принимают значение NULL. Следует заметить, что для задания условия вместо ключевого слова WHERE при левом и правом объединениях используется ключевое слово ON.

В листинге 22.24 демонстрируется "правое" объединение при помощи конструкции RIGHT JOIN.

Листинг 22.24. Правое объединение таблиц tbl1 и tbl2

```
mysql> SELECT * FROM tbl1 RIGHT JOIN tbl2 ON tbl1.id = tbl2.id;
```

```
+------+------+----+--------+
| id   | name | id | letter |
+------+------+----+--------+
|    2 | Б    |  2 | Г      |
|    3 | В    |  3 | Д      |
| NULL | NULL |  4 | Е      |
+------+------+----+--------+
```

Как видно из листинга 22.24, при правом объединении возвращаются строки, удовлетворяющие условию `tbl1.id = tbl2.id`, и строки "правой" таблицы `tbl2`, которым не нашлось соответствия в "левой" таблице `tbl1`.

Замечание

Ключевые слова `LEFT JOIN` и `RIGHT JOIN` имеют синонимы `LEFT OUTER JOIN` и `RIGHT OUTER JOIN` соответственно.

Другим способом установки связи между таблицами `tbl1` и `tbl2` при правом и левом объединениях является использование ключевого слова `USING()`. В круглых скобках, следующих за этим ключевым словом, перечисляются имена столбцов, которые должны присутствовать в обеих таблицах и для которых необходимо соблюдение равенства. Данный оператор предназначен для создания более компактных SQL-запросов. Так, следующие два выражения идентичны:

```
a LEFT JOIN b USING(c1,c2,c3)
a LEFT JOIN b ON a.c1 = b.c1 AND a.c2 = b.c2 AND a.c3 = b.c3
```

Принимая во внимание синтаксис ключевого слова `USING()`, левое и правое объединения, показанные в листингах 22.23 и 22.24, можно представить так, как это сделано в листинге 22.25.

Листинг 22.25. Использование ключевого слова `USING`

```
mysql> SELECT * FROM tbl1 LEFT JOIN tbl2 USING(id);
+----+------+------+--------+
| id | name | id   | letter |
+----+------+------+--------+
|  1 | А    | NULL | NULL   |
|  2 | Б    |    2 | Г      |
|  3 | В    |    3 | Д      |
+----+------+------+--------+
mysql> SELECT * FROM tbl1 RIGHT JOIN tbl2 USING(id);
+------+------+----+--------+
| id   | name | id | letter |
+------+------+----+--------+
|    2 | Б    |  2 | Г      |
|    3 | В    |  3 | Д      |
| NULL | NULL |  4 | Е      |
+------+------+----+--------+
```

Для каждой из таблиц, участвующих в объединении с использованием SQL-оператора `JOIN`, можно ввести подсказку о том, как СУБД MySQL должна использовать индексы при извлечении данных из таблицы. Указав после имени таблицы ключевое слово `USE INDEX(list)`, в скобках можно задать список индексов `list`, которые СУБД MySQL должна использовать при поиске записей в таблице. Ключевое слово `IGNORE INDEX(list)` предназначено для того, чтобы запретить СУБД MySQL использовать какой-то отдельный индекс. Ключевое слово `FORCE INDEX(list)` подобно `USE INDEX(list)`, но с тем отличием, что сканирование таблицы расценивается как очень дорогая операция. Это учитывается оптимизатором MySQL, и полное сканирование таблицы производится только в том случае, если нет возможности использовать индекс.

Возвращаясь к учебной базе данных `shop`, рассмотрим несколько запросов к таблицам, входящим в ее состав. В листинге 22.26 при помощи конструкции `JOIN ... USING()` извлекается число товарных позиций в каталогах.

Листинг 22.26. Извлечение числа товарных позиций в каталогах

```
mysql> SELECT catalogs.name, COUNT(id_product)
    -> FROM catalogs JOIN products USING(id_catalog)
    -> GROUP BY products.id_catalog;
+--------------------+-------------------+
| name               | COUNT(id_product) |
+--------------------+-------------------+
| Процессоры         |                 9 |
| Материнские платы  |                 6 |
| Видеоадаптеры      |                 4 |
| Жесткие диски      |                 5 |
| Оперативная память |                 6 |
+--------------------+-------------------+
```

Допустим, происходит расширение ассортимента товарных позиций и в списке каталогов появляется новый каталог `'Периферия'`. SQL-запрос, добавляющий данный каталог, представлен в листинге 22.27.

Листинг 22.27. Добавление нового каталога `'Периферия'`

```
mysql> INSERT INTO catalogs VALUES (NULL, 'Периферия');
mysql> SELECT * FROM catalogs;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 | Оперативная память |
|          6 | Периферия          |
+------------+--------------------+
```

Однако запрос из листинга 22.26 не отражает наличие нового каталога в электронном магазине, т. к. таблица `products` еще не содержит ни одной записи, относящейся к новому каталогу. Для того чтобы название каталога `'Периферия'` появилось в результирующей таблице, необходимо провести левое объединение таблиц `catalogs` и `products`, причем таблица `catalogs` должна выступать в качестве "левой" таблицы.

Листинг 22.28. Полный список каталогов

```
mysql> SELECT catalogs.name, COUNT(id_product)
    -> FROM catalogs LEFT JOIN products USING(id_catalog)
    -> GROUP BY products.id_catalog;
+--------------------+-------------------+
| name               | COUNT(id_product) |
+--------------------+-------------------+
| Периферия          |                 0 |
| Процессоры         |                 9 |
| Материнские платы  |                 6 |
| Видеоадаптеры      |                 4 |
| Жесткие диски      |                 5 |
| Оперативная память |                 6 |
+--------------------+-------------------+
```

Как видно из листинга 22.28, левое объединение позволило включить каталог `'Периферия'`, товарные позиции в котором пока отсутствуют.

Пусть требуется вывести список покупателей и число осуществленных ими покупок, причем покупателей необходимо отсортировать в порядке убывания числа оплаченных ими заказов. Для решения этой задачи можно воспользоваться запросом, представленным в листинге 22.29.

Листинг 22.29. Список покупателей и число оплаченных ими заказов

```
mysql> SELECT users.surname, users.name,
    -> users.patronymic, COUNT(orders.id_order) AS total
    -> FROM users JOIN orders USING(id_user)
    -> GROUP BY users.id_user
    -> ORDER BY total DESC;
+----------+-----------+---------------+-------+
| surname  | name      | patronymic    | total |
+----------+-----------+---------------+-------+
| Симдянов | Игорь     | Вячеславович  |     3 |
| Корнеев  | Александр | Александрович |     1 |
| Иванов   | Александр | Валерьевич    |     1 |
+----------+-----------+---------------+-------+
```

Как видно из листинга 22.29, в списке присутствуют только те покупатели, которые оплатили хотя бы одну покупку, покупатели, на счету у которых нет ни одной покуп-

ки, в список не входят. Для того чтобы вывести полный список покупателей, необходимо вместо перекрестного объединения таблиц users и orders воспользоваться левым объединением, где в качестве "левой" таблицы выступит таблица users (листинг 22.30).

Листинг 22.30. Полный список покупателей

```
mysql> SELECT users.surname, users.name,
    -> users.patronymic, COUNT(orders.id_order) AS total
    -> FROM users LEFT JOIN orders USING(id_user)
    -> GROUP BY users.id_user
    -> ORDER BY total DESC;
+-----------+-----------+---------------+-------+
| surname   | name      | patronymic    | total |
+-----------+-----------+---------------+-------+
| Симдянов  | Игорь     | Вячеславович  |     3 |
| Иванов    | Александр | Валерьевич    |     1 |
| Корнеев   | Александр | Александрович |     1 |
| Лосев     | Сергей    | Иванович      |     0 |
| Кузнецов  | Максим    | Валерьевич    |     0 |
| Нехорошев | Анатолий  | Юрьевич       |     0 |
+-----------+-----------+---------------+-------+
```

Помимо условия ON или USING, в запросах на объединение могут использоваться традиционные конструкции условия. Например, запрос, представленный в листинге 22.31, извлекает список покупателей и число их покупок при условии, что покупателя зовут "Александр".

Листинг 22.31. Использование ключевого слова WHERE

```
mysql> SELECT users.surname, users.name,
    -> users.patronymic, COUNT(orders.id_order) AS total
    -> FROM users LEFT JOIN orders USING(id_user)
    -> WHERE name = 'Александр'
    -> GROUP BY users.id_user
    -> ORDER BY total DESC;
+---------+-----------+---------------+-------+
| surname | name      | patronymic    | total |
+---------+-----------+---------------+-------+
| Иванов  | Александр | Валерьевич    |     1 |
| Корнеев | Александр | Александрович |     1 |
+---------+-----------+---------------+-------+
```

Для этого используется условие WHERE name = 'Александр'. Однако использовать столбец total в условии WHERE уже не получится, т. к. это групповой столбец, сформированный агрегатной функцией COUNT() и конструкцией GROUP BY. Для формирования условия с его участием необходимо использовать условие HAVING. Пусть требу-

ется извлечь всех покупателей магазина, число покупок (total) у которых меньше трех (листинг 22.32).

Листинг 22.32. Использование ключевого слова HAVING

```
mysql> SELECT users.surname, users.name,
    -> users.patronymic, COUNT(orders.id_order) AS total
    -> FROM users LEFT JOIN orders USING(id_user)
    -> GROUP BY users.id_user
    -> HAVING total < 3
    -> ORDER BY total DESC;
+-----------+-----------+---------------+-------+
| surname   | name      | patronymic    | total |
+-----------+-----------+---------------+-------+
| Иванов    | Александр | Валерьевич    |     1 |
| Корнеев   | Александр | Александрович |     1 |
| Лосев     | Сергей    | Иванович      |     0 |
| Кузнецов  | Максим    | Валерьевич    |     0 |
| Нехорошев | Анатолий  | Юрьевич       |     0 |
+-----------+-----------+---------------+-------+
```

22.3. Обновление нескольких таблиц

Помимо оператора SELECT, в многотабличных запросах можно использовать оператор UPDATE. Для этого имена таблиц перечисляются через запятую. Пусть требуется изменить первичный ключ id_catalog таблицы catalogs для каталога 'Оперативная память' с 5 на 10. Для этого можно использовать запрос, представленный в листинге 22.33.

Листинг 22.33. Редактирование таблицы catalogs

```
mysql> UPDATE catalogs SET id_catalog = 10 WHERE id_catalog = 5
```

Однако изменения коснулись только таблицы catalogs, значения внешних ключей в таблице products не изменились. Исправить ситуацию может многотабличный запрос, представленный в листинге 22.34.

Замечание

Синтаксис оператора UPDATE описывается в *главе 9*.

Листинг 22.34. Обновление сразу двух таблиц

```
mysql> UPDATE catalogs, products
    -> SET catalogs.id_catalog = 10, products.id_catalog = 10
    -> WHERE catalogs.id_catalog = 5 AND products.id_catalog = 5;
```

```
mysql> SELECT * FROM catalogs;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|         10 | Оперативная память |
+------------+--------------------+
mysql> SELECT id_catalog FROM products GROUP BY id_catalog;
+------------+
| id_catalog |
+------------+
|          1 |
|          2 |
|          3 |
|          4 |
|         10 |
+------------+
```

Пусть в таблицу `products` требуется добавить столбец `catalog`, который необходимо заполнить названиями каталогов из таблицы `catalogs`. Добавить новый столбец можно при помощи оператора `ALTER TABLE` (листинг 22.35), синтаксис которого подробно рассматривается в *главе 10*.

Листинг 22.35. Добавление нового столбца `catalog` в таблицу `products`

```
mysql> ALTER TABLE products ADD catalog TINYTEXT AFTER id_catalog;
mysql> DESCRIBE products;
+-------------+--------------+------+-----+---------+----------------+
| Field       | Type         | Null | Key | Default | Extra          |
+-------------+--------------+------+-----+---------+----------------+
| id_product  | int(11)      |      | PRI | NULL    | auto_increment |
| name        | tinytext     |      | MUL |         |                |
| price       | decimal(7,2) |      |     | 0.00    |                |
| count       | int(11)      |      |     | 0       |                |
| mark        | float(4,1)   |      |     | 0.0     |                |
| description | text         |      |     |         |                |
| id_catalog  | int(11)      |      | MUL | 0       |                |
| catalog     | tinytext     | YES  |     | NULL    |                |
+-------------+--------------+------+-----+---------+----------------+
```

Однако поле `catalog` в настоящий момент является пустым, для того чтобы заполнить его, можно прибегнуть к многотабличному запросу `UPDATE` (листинг 22.36).

Листинг 22.36. Заполнение поля `catalog`

```
mysql> SELECT id_catalog, catalog FROM products GROUP BY id_catalog;
+------------+---------+
| id_catalog | catalog |
+------------+---------+
|          1 | NULL    |
|          2 | NULL    |
|          3 | NULL    |
|          4 | NULL    |
|         10 | NULL    |
+------------+---------+
mysql> UPDATE products, catalogs SET products.catalog = catalogs.name
    -> WHERE products.id_catalog = catalogs.id_catalog;
mysql> SELECT id_catalog, catalog FROM products GROUP BY id_catalog;
+------------+--------------------+
| id_catalog | catalog            |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|         10 | Оперативная память |
+------------+--------------------+
```

Как видно из листинга 22.36, таблица `catalogs` не подвергается изменениям, но активно участвует в многотабличном запросе с участием оператора `UPDATE`.

Замечание

В многотабличных запросах с участием SQL-оператора `UPDATE` не допускается использование конструкций `ORDER BY` и `LIMIT`.

Замечание

Многотабличные запросы с участием SQL-оператора `UPDATE` введены в СУБД MySQL, начиная с версии 4.0.0.

Рассмотренные выше примеры используют перекрестное объединение, но многотабличные запросы с участием оператора `UPDATE` допускают любой тип объединения, продемонстрируемые ранее на примере оператора `SELECT`. В листинге 22.37 приводятся три запроса, идентичные представленному в листинге 22.36.

Листинг 22.37. Многотабличные запросы с использованием ключевого слова `JOIN`

```
mysql> UPDATE products JOIN catalogs SET products.catalog = catalogs.name
    -> WHERE products.id_catalog = catalogs.id_catalog;
```

```
mysql> UPDATE products JOIN catalogs
    -> ON products.id_catalog = catalogs.id_catalog
    -> SET products.catalog = catalogs.name;
mysql> UPDATE products JOIN catalogs USING(id_catalog)
    -> SET products.catalog = catalogs.name;
```

Точно так же, как и в случае оператора SELECT, допускается использование псевдонимов для таблиц, которые назначаются при помощи оператора AS (листинг 22.38).

Листинг 22.38. Использование псевдонимов

```
mysql> UPDATE products AS p, catalogs AS c SET p.catalog = c.name
    -> WHERE p.id_catalog = c.id_catalog;
```

22.4. Удаление из нескольких таблиц

Многотабличное удаление из таблиц при помощи оператора DELETE во многом аналогично операторам SELECT и UPDATE.

Замечание

В многотабличных запросах с участием SQL-оператора DELETE не допускается использование конструкций ORDER BY и LIMIT.

Замечание

Многотабличные запросы с участием SQL-оператора DELETE введены в СУБД MySQL, начиная с версии 4.0.0.

При использовании многотабличных запросов с помощью оператора DELETE допустимы две формы, представленные в листинге 22.39, где из таблицы products удаляются все записи, для которых имеется соответствие в таблице catalogs.

Листинг 22.39. Многотабличное удаление из таблицы products

```
mysql> DELETE products FROM products, catalogs
    -> WHERE products.id_catalog = catalogs.id_catalog;
mysql> DELETE FROM products USING products, catalogs
    -> WHERE products.id_catalog = catalogs.id_catalog;
```

Следует отметить, что записи касаются только таблицы products — удаление производится лишь из таблиц, перечисленных после ключевого слова DELETE в первой форме запроса, и после ключевого слова FROM во второй форме запроса. Запросы, представленные в листинге 22.40, затронут уже обе таблицы.

Листинг 22.40. Многотабличное удаление из таблиц products и catalogs

```
mysql> DELETE products, catalogs FROM products, catalogs
    -> WHERE products.id_catalog = catalogs.id_catalog;
```

```
mysql> DELETE FROM products, catalogs USING products, catalogs
    -> WHERE products.id_catalog = catalogs.id_catalog;
```

Продемонстрированные выше примеры используют перекрестное объединение, хотя многотабличные запросы с участием оператора DELETE допускают любой тип объединения, рассмотренные ранее на примере оператора SELECT.

Для достижения совместимости с Access в многотабличных операторах DELETE допускается использование символа ".*" после имени таблицы (листинг 22.41).

Листинг 22.41. Использование символа ".*"

```
mysql> DELETE products.*, catalogs.* FROM products, catalogs
    -> WHERE products.id_catalog = catalogs.id_catalog;
mysql> DELETE FROM products.*, catalogs.* USING products, catalogs
    -> WHERE products.id_catalog = catalogs.id_catalog;
```

При использовании псевдонимов таблиц, назначаемых оператором AS, в СУБД MySQL 4.0 необходимо обращаться к таблицам с помощью реальных имен (листинг 22.42).

Листинг 22.42. Использование ключевого слова AS в СУБД MySQL 4.0

```
mysql> DELETE products, catalogs FROM products AS p, catalogs AS c
    -> WHERE p.id_catalog = c.id_catalog;
```

Начиная с версии СУБД MySQL 4.1, псевдонимы требуется использовать в тексте всего оператора DELETE (листинг 22.43).

Листинг 22.43. Использование ключевого слова AS в СУБД MySQL 4.1

```
mysql> DELETE p, c FROM products AS p, catalogs AS c
    -> WHERE p.id_catalog = c.id_catalog;
```

ГЛАВА 23

Вложенные запросы

Вложенный запрос позволяет использовать результат, возвращаемый одним запросом, в другом запросе. Возможность применения одного запроса внутри другого и была причиной появления слова "структурированный" в названии "структурированный язык запросов" (SQL). Так как результат возвращает только оператор `SELECT`, то в качестве "вложенного" запроса всегда выступает `SELECT`-запрос (информационные SQL-запросы с использованием нестандартных операторов `SHOW`, `DESCRIBE` во внимание не принимаются). В качестве внешнего запроса может выступать запрос с участием любого SQL-оператора: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE TABLE` и др.

Замечание

Термин "вложенный запрос" имеет различные варианты в русскоязычной литературе, такого рода запросы часто называют подчиненными запросами и подзапросами.

Замечание

Вложенные запросы появились в СУБД MySQL, начиная с версии 4.1.

Пусть требуется вывести название товарных позиций из таблицы `products` для раздела "Процессоры" таблицы `catalogs`. Опираясь на предыдущие главы, решить эту задачу можно, например, при помощи двух SQL-запросов (листинг 23.1). Первый запрос извлекает первичный ключ (`id_catalog`) записи из таблицы `catalogs`, соответствующей каталогу '`Процессоры`'. Второй запрос выводит названия товарных позиций и их цены для данного каталога.

Листинг 23.1. Товарные позиции каталога '`Процессоры`'

```
mysql> SELECT @id := id_catalog FROM catalogs WHERE name = 'Процессоры';
+-------------------+
| @id := id_catalog |
+-------------------+
|                 1 |
+-------------------+
```

```
mysql> SELECT name, price FROM products
    -> WHERE id_catalog = @id
    -> ORDER BY price;
+------------------------+---------+
| name                   | price   |
+------------------------+---------+
| Celeron 1.8            | 1595.00 |
| Celeron D 315 2.26GHz  | 1880.00 |
| Celeron D 320 2.4GHz   | 1962.00 |
| Celeron 2.0GHz         | 1969.00 |
| Celeron 2.4GHz         | 2109.00 |
| Celeron D 325 2.53GHz  | 2747.00 |
| Intel Pentium 4 3.0GHz | 5673.00 |
| Intel Pentium 4 3.0GHz | 6147.00 |
| Intel Pentium 4 3.2GHz | 7259.00 |
+------------------------+---------+
```

Замечание

Синтаксис переменных и их использование подробно рассматривается в *главе 21*.

Как видно из листинга, промежуточное значение помещается в переменную `@id`. Вложенные запросы позволяют осуществить подобную выборку в одном запросе (листинг 23.2).

Листинг 23.2. Применение вложенного запроса

```
mysql> SELECT name, price FROM products
    -> WHERE id_catalog = (SELECT id_catalog FROM catalogs
    ->                     WHERE name = 'Процессоры')
    -> ORDER BY price;
+------------------------+---------+
| name                   | price   |
+------------------------+---------+
| Celeron 1.8            | 1595.00 |
| Celeron D 315 2.26GHz  | 1880.00 |
| Celeron D 320 2.4GHz   | 1962.00 |
| Celeron 2.0GHz         | 1969.00 |
| Celeron 2.4GHz         | 2109.00 |
| Celeron D 325 2.53GHz  | 2747.00 |
| Intel Pentium 4 3.0GHz | 5673.00 |
| Intel Pentium 4 3.0GHz | 6147.00 |
| Intel Pentium 4 3.2GHz | 7259.00 |
+------------------------+---------+
```

Здесь в качестве внешнего запроса выступает `SELECT name, price FROM products`, а в качестве вложенного запроса используется запрос `SELECT id_catalog FROM catalogs WHERE name = 'Процессоры'`. Таким образом, результат вложенного запроса

(id_catalog), минуя промежуточные переменные в MySQL или внешней программы, применяется в качестве элемента внешнего запроса. Вложенный запрос всегда помещается в круглые скобки.

Получить аналогичный результат можно при помощи многотабличного запроса, представленного в листинге 23.3.

Листинг 23.3. Решение задачи при помощи многотабличного запроса

```
mysql> SELECT products.name, products.price FROM catalogs, products
    -> WHERE catalogs.id_catalog = products.id_catalog AND
    ->       catalogs.name = 'Процессоры'
    -> ORDER BY price;
+------------------------+---------+
| name                   | price   |
+------------------------+---------+
| Celeron 1.8            | 1595.00 |
| Celeron D 315 2.26GHz  | 1880.00 |
| Celeron D 320 2.4GHz   | 1962.00 |
| Celeron 2.0GHz         | 1969.00 |
| Celeron 2.4GHz         | 2109.00 |
| Celeron D 325 2.53GHz  | 2747.00 |
| Intel Pentium 4 3.0GHz | 5673.00 |
| Intel Pentium 4 3.0GHz | 6147.00 |
| Intel Pentium 4 3.2GHz | 7259.00 |
+------------------------+---------+
```

В отличие от вложенного запроса, многотабличный запрос менее нагляден, хотя в ряде случаев выполняется быстрее, т. к. оптимизация многотабличных запросов происходит быстрее вложенных. В большинстве случаев запросы можно представить как при помощи одного из объединений (перекрестного, левого, правого), так и при помощи вложенных запросов, но все же имеется ряд задач, которые решаются только при помощи вложенных запросов. Они будут рассмотрены в данной главе.

Вложенный запрос может применяться не только с условием WHERE, но также в конструкциях DISTINCT, GROUP BY, ORDER BY, LIMIT, JOIN, UNION, в функциях и т. д.

23.1. Вложенный запрос как скалярный операнд

В данном виде запроса вложенный запрос возвращает одно-единственное значение, которое используется во внешнем запросе. Такой вложенный запрос можно использовать в любом месте, где допустимо использование скалярного значения или литерала.

Пусть требуется определить название каталога, в котором присутствует самая дорогая товарная позиция (листинг 23.4).

Листинг 23.4. Определение названий каталога с самой дорогой товарной позицией

```
mysql> SELECT name FROM catalogs
    -> WHERE id_catalog = (SELECT id_catalog FROM products
    ->                     WHERE price = (SELECT MAX(price)
    ->                                    FROM products));
+------------+
| name       |
+------------+
| Процессоры |
+------------+
```

Как видно из листинга 23.4, степень вложения запросов не ограничена вторым уровнем, могут использоваться трехуровневые и многоуровневые вложенные запросы. Первый внутренний запрос определяет максимальную цену в таблице `products`:

```
SELECT MAX(price) FROM products
```

Внешний по отношению к нему запрос определяет значение первичного ключа (`id_catalog`) для товарной позиции с максимальным значением цены (7259.00):

```
SELECT id_catalog FROM products WHERE price = 7259.00
```

Полученное значение поля `id_catalog` (равное 1) подставляется во внешний запрос:

```
SELECT name FROM catalogs WHERE id_catalog = 1
```

Замечание

На практике редко прибегают к вложенным запросам со степенью вложения больше трех, т. к. большая степень вложенности быстро приводит к увеличению времени выполнения запроса.

Вложенный запрос, возвращающий единственное значение, просто подставляет результат на место своего выполнения.

Листинг 23.5. Использование вложенного запроса в списке столбцов

```
mysql> SELECT id_catalog, (SELECT MAX(price) FROM products)
    -> FROM catalogs;
+------------+----------------------------------+
| id_catalog | (SELECT MAX(price) FROM products) |
+------------+----------------------------------+
|          1 | 7259.00                          |
|          2 | 7259.00                          |
|          3 | 7259.00                          |
|          4 | 7259.00                          |
|          5 | 7259.00                          |
|          6 | 7259.00                          |
+------------+----------------------------------+
```

В листинге 23.5 вложенный запрос `SELECT MAX(price) FROM products` возвращает максимальное значение цены из таблицы `products` (7259.00) и подставляет это значение на место запроса, поэтому запрос из листинга 23.5 можно представить как запрос со скалярной величиной в качестве столбца (листинг 23.6).

Листинг 23.6. Запрос со скалярной величиной в качестве столбца

```
mysql> SELECT id_catalog, 7259.00 FROM catalogs;
+------------+---------+
| id_catalog | 7259.00 |
+------------+---------+
|          1 | 7259.00 |
|          2 | 7259.00 |
|          3 | 7259.00 |
|          4 | 7259.00 |
|          5 | 7259.00 |
|          6 | 7259.00 |
+------------+---------+
```

Допустимы также запросы вида, представленного в листинге 23.7. Вложенный запрос возвращает название каталога с первичным ключом `id_catalog = 1`, а внешний запрос, состоящий из одного оператора `SELECT`, выводит его.

Листинг 23.7. Использование вложенного запроса как скалярной величины

```
mysql> SELECT (SELECT name FROM catalogs WHERE id_catalog = 1);
+-------------------------------------------------+
| (SELECT name FROM catalogs WHERE id_catalog = 1) |
+-------------------------------------------------+
| Процессоры                                      |
+-------------------------------------------------+
```

При использовании вложенных запросов в качестве аргументов встроенных функций MySQL следует помнить о необходимости помещения вложенного запроса в дополнительные круглые скобки. В листинге 23.8 название каталога с первичным ключом `id_catalog = 1` из таблицы `catalogs` передается в качестве аргумента функции `UPPER()`, которая переводит строку в верхний регистр. При этом используются двойные круглые скобки — одни для обозначения вложенного запроса, другие — для обозначения признака функции.

Листинг 23.8. Использование вложенного запроса в качестве аргумента функции

```
mysql> SELECT UPPER((SELECT name FROM catalogs WHERE id_catalog = 1));
+---------------------------------------------------------+
| UPPER((SELECT name FROM catalogs WHERE id_catalog = 1)) |
+---------------------------------------------------------+
| ПРОЦЕССОРЫ                                              |
+---------------------------------------------------------+
```

Наиболее часто вложенные запросы используются в операциях сравнения в условиях, которые задаются ключевыми словами WHERE, HAVING или ON. Для этого задействуются шесть операторов сравнения ("=", "<>", "<", "<=", ">", ">="), подробное описание которых можно найти в *главе 13*. Вложенный запрос, участвующий в операции сравнения, должен возвращать в качестве результата единичное значение.

Листинг 23.9. Использование вложенных запросов в операциях сравнения

```
mysql> SELECT name FROM catalogs
    -> WHERE id_catalog > (SELECT id_catalog FROM products
    ->                     WHERE id_product = 10);
+--------------------+
| name               |
+--------------------+
| Видеоадаптеры      |
| Жесткие диски      |
| Оперативная память |
+--------------------+
mysql> SELECT name FROM catalogs
    -> WHERE (SELECT id_catalog FROM products
    ->        WHERE id_product = 10) < id_catalog;
+--------------------+
| name               |
+--------------------+
| Видеоадаптеры      |
| Жесткие диски      |
| Оперативная память |
+--------------------+
```

В листинге 23.9 возвращаются названия каталогов, значение первичного ключа id_catalog для которых больше значения первичного ключа каталога, в котором находится товарная позиция с id_product = 10.

Замечание

В листинге 23.9 показано, что вложенный запрос может располагаться как справа, так и слева от оператора сравнения. Однако предпочтительнее располагать его в правой части, как это продемонстрировано в первом запросе, для совместимости с другими СУБД, не удовлетворяющими стандарту SQL2, в котором было устранено подобное ограничение.

23.2. Вложенные запросы, возвращающие несколько строк

В предыдущем разделе рассмотрены запросы, где вложенный запрос возвращает единственное значение. Если вложенный запрос возвращает несколько строк, СУБД

MySQL возвращает ошибку 1242 — вложенный запрос возвращает более чем одну строку (листинг 23.10).

Листинг 23.10. Вложенный запрос возвращает более одной строки

```
mysql> SELECT name FROM catalogs
    -> WHERE id_catalog = (SELECT id_catalog FROM products);
ERROR 1242: Subquery returns more than 1 row
```

23.2.1. Ключевое слово *IN*

Для того чтобы выбрать строки за таблицы `catalogs`, у которых первичный ключ `id_catalog` совпадает с одним из значений, возвращаемых вложенным запросом, следует воспользоваться конструкцией `IN`, которая подробно рассмотрена в *главе 7*.

Листинг 23.11. Использование конструкции `IN`

```
mysql> SELECT name FROM catalogs
    -> WHERE id_catalog IN (SELECT id_catalog FROM products
    ->                      GROUP BY id_catalog);
+--------------------+
| name               |
+--------------------+
| Процессоры         |
| Материнские платы  |
| Видеоадаптеры      |
| Жесткие диски      |
| Оперативная память |
+--------------------+
```

Запрос, представленный в листинге 23.11, аналогичен запросу, показанному в листинге 23.12.

Листинг 23.12. Использование конструкции `IN` совместно со списком скалярных величин

```
mysql> SELECT name FROM catalogs WHERE id_catalog IN (1,2,3,4,5);
+--------------------+
| name               |
+--------------------+
| Процессоры         |
| Материнские платы  |
| Видеоадаптеры      |
| Жесткие диски      |
| Оперативная память |
+--------------------+
```

Для того чтобы возвратить строки, которые отсутствуют в результирующей таблице, возвращаемой вложенным запросом, следует воспользоваться оператором NOT IN, представленным в листинге 23.13.

Листинг 23.13. Использование оператора NOT IN

```
mysql> SELECT name FROM catalogs
    -> WHERE id_catalog NOT IN (SELECT DISTINCT id_catalog
    ->                          FROM products WHERE id_catalog<3);
+--------------------+
| name               |
+--------------------+
| Видеоадаптеры      |
| Жесткие диски      |
| Оперативная память |
+--------------------+
```

23.2.2. Ключевое слово *ANY (SOME)*

Конструкция IN позволяет осуществить поиск величины в списке и выражает логику запроса, представленного в листинге 23.10. Однако на месте оператора "=" в этом запросе может стоять другой оператор сравнения (листинг 23.14).

Листинг 23.14. Вложенный запрос возвращает более одной строки

```
mysql> SELECT name FROM catalogs
    -> WHERE id_catalog > (SELECT id_catalog FROM products);
ERROR 1242: Subquery returns more than 1 row
```

Данный запрос также завершается неудачей, но сформулировать его с использованием конструкции IN уже не получится. К счастью, язык запросов SQL обладает средствами решения подобных задач. Для этого используется ключевое слово ANY (листинг 23.15).

Замечание

Ключевое слово ANY имеет синоним SOME, однако предпочтительнее использовать именно ANY.

Листинг 23.15. Использование ключевого слова ANY

```
mysql> SELECT id_catalog, name FROM catalogs
    -> WHERE id_catalog > ANY (SELECT id_catalog FROM products);
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          2 | Материнские платы  |
```

```
|           3 | Видеоадаптеры      |
|           4 | Жесткие диски      |
|           5 | Оперативная память |
+-------------+--------------------+
```

Ключевое слово ANY может применяться для сравнения значений с использованием одного из шести операторов сравнения ("=", "<>", "<", "<=", ">", ">="). Проверяемое значение id_catalog поочередно сравнивается с каждым элементом, который возвращает вложенный запрос. Если хотя бы одно из сравнений возвращает 1 (истина), строка выводится запросом. В листинге 23.15 происходит сравнение значений первичного ключа id_catalog (1, 2, 3, 4, 5), которые присутствуют в таблице catalogs, со значениями поля id_catalog (1, 2, 3, 4, 5) из таблицы products. Значение id_catalog = 1 не удовлетворяет ни одному условию:

```
1 > 1 -- 0 (ложь)
1 > 2 -- 0 (ложь)
1 > 3 -- 0 (ложь)
1 > 4 -- 0 (ложь)
1 > 5 -- 0 (ложь)
```

Поэтому в результаты эта строка не попадает, в то же время все остальные цифры удовлетворяют хотя бы одному условию и попадают в результирующую таблицу:

```
3 > 1 -- 1 (истина)
3 > 2 -- 1 (истина)
3 > 3 -- 0 (ложь)
3 > 4 -- 0 (ложь)
3 > 5 -- 0 (ложь)
```

То есть запрос вида

"WHERE X > ANY (SELECT Y...)"

можно интерпретировать как "где X больше хотя бы одного выбранного Y".

А запрос вида

"WHERE X < ANY (SELECT Y...)"

следует читать "где X меньше, чем любой выбранный Y...".

Запрос, представленный в листинге 23.16, возвращает имя и фамилию покупателей из таблицы users, совершивших хотя бы одну покупку.

Листинг 23.16. Извлечение списка покупателей, совершивших хотя бы одну покупку

```
mysql> SELECT name, surname FROM users
    -> WHERE id_user = ANY (SELECT id_user FROM orders);
+-----------+----------+
| name      | surname  |
+-----------+----------+
| Александр | Иванов   |
```

```
| Игорь      | Симдянов |
| Александр | Корнеев  |
+-----------+----------+
```

На практике сравнение с использованием ключевого слова ANY может приводить к ошибкам, которые трудно выявить, особенно когда применяется оператор сравнения "не равно" ("<>" или "!="). Так, если потребуется вернуть список покупателей, не совершавших ни одной покупки, возникнет искушение заменить оператор равенства "=" (листинг 23.16) на оператор неравенства "<>" (листинг 23.17).

Листинг 23.17. Извлечение списка покупателей, не совершавших покупок

```
mysql> SELECT name, surname FROM users
    -> WHERE id_user <> ANY (SELECT id_user FROM orders);
+-----------+-----------+
| name      | surname   |
+-----------+-----------+
| Александр | Иванов    |
| Сергей    | Лосев     |
| Игорь     | Симдянов  |
| Максим    | Кузнецов  |
| Анатолий  | Нехорошев |
| Александр | Корнеев   |
+-----------+-----------+
```

Результатом является полный список покупателей как совершавших, так и не совершавших покупок, т. к. значение id_user сравнивается с каждым значением, которое возвращает вложенный запрос. Среди этих значений всегда найдется число, которое не будет равно текущему значению id_user. Решить данную задачу можно при помощи ключевых слов EXISTS и NOT EXISTS, которые рассмотрены в *разделе 23.3*.

23.2.3. Ключевое слово *ALL*

Вместо ключевого слова ANY может быть использовано ключевое слово ALL, которое точно так же используется совместно с одним из шести операторов сравнения ("=", "<>", "<", "<=", ">", ">="). В этом случае проверяемое значение также поочередно сравнивается с каждым элементом, который возвращает вложенный запрос, но строка возвращается только тогда, когда все сравнения возвращают 1 (истина).

Замечание

Если в выражениях с ключевым словом ANY используется логика ИЛИ, т. е. достаточно, чтобы срабатывало хотя бы одно из многих условий, то в случае ALL используется логика И — должны срабатывать все условия.

В листинге 23.18 представлен запрос, возвращающий все товарные позиции из таблицы products базы данных shop, цена которых превышает среднюю цену каждого из каталогов.

Листинг 23.18. Использование ключевого слова ALL

```
mysql> SELECT name, price FROM products
    -> WHERE price > ALL (SELECT AVG(price) FROM products
    ->                    GROUP BY id_catalog);
+--------------------------+---------+
| name                     | price   |
+--------------------------+---------+
| Intel Pentium 4 3.2GHz   | 7259.00 |
| Intel Pentium 4 3.0GHz   | 6147.00 |
| Intel Pentium 4 3.0GHz   | 5673.00 |
| Asustek P4C800-E Delux   | 5395.00 |
| ASUSTEK A9600XT/TD       | 5156.00 |
| GIGABYTE AGP GV-N59X128D | 5886.00 |
+--------------------------+---------+
```

Для того чтобы понять логику запроса, представленного в листинге 23.18, полезно выполнить вложенный запрос отдельно.

Листинг 23.19. Вложенный запрос

```
mysql> SELECT AVG(price) FROM products GROUP BY id_catalog;
+-------------+
| AVG(price)  |
+-------------+
| 3482.333333 |
| 2801.166667 |
| 3843.500000 |
| 2749.000000 |
| 1417.000000 |
+-------------+
```

Условие ALL в листинге 23.18 эквивалентно требованию вывести все товарные позиции, цена которых больше каждого из значений результирующей таблицы запроса, приведенного в листинге 23.19.

То есть запрос вида

"WHERE X > ALL (SELECT Y...)"

можно толковать как "где X больше любого выбранного Y".

А запрос вида

"WHERE X < ALL (SELECT Y...)"

следует читать "где X меньше, чем все выбранные Y...".

Ошибки, которые могут возникнуть, если проверка ANY содержит оператор "<>", происходят и при проверке ALL.

23.3. Проверка на существование

Результирующая таблица, которая возвращается вложенным запросом, может быть пустой, т. е. не содержать ни одной строки. Для проверки данного факта используются ключевые слова EXISTS и NOT EXISTS. Данные ключевые слова не требуют левого операнда, они просто сообщают, имеются ли в результирующей таблице строки или нет. Если вложенный запрос возвращает более одной строки, EXISTS возвращает 1 (истина), в противном случае ключевое слово возвращает 0 (ложь). Ключевое слово NOT EXISTS действует противоположным образом.

Замечание

Проверка на существование допустима только во вложенных запросах.

Пусть требуется вывести названия товарных позиций из таблицы products, которые были выбраны покупателями, т. е. для которых имеется отметка о покупке в таблице orders (листинг 23.20).

Листинг 23.20. Использование ключевого слова EXISTS

```
mysql> SELECT id_product, name, price FROM products
    -> WHERE EXISTS (SELECT * FROM orders
    ->               WHERE orders.id_product = products.id_product);
+------------+-----------------------+---------+
| id_product | name                  | price   |
+------------+-----------------------+---------+
|          8 | Intel Pentium 4 3.0GHz | 6147.00 |
|         10 | Gigabyte GA-8I848P-RS  | 1896.00 |
|         20 | Maxtor 6Y120P0         | 2456.00 |
+------------+-----------------------+---------+
```

Внешний запрос последовательно перебирает все строки таблицы products и для каждой товарной позиции выполняет вложенный запрос, который проверяет, имеется ли в таблице orders текущая товарная позиция. Число записей, которые возвращает вложенный запрос, может быть более одной (листинг 23.21).

Листинг 23.21. Содержимое таблицы orders

```
mysql> SELECT id_product FROM orders;
+------------+
| id_product |
+------------+
|          8 |
|         10 |
|         20 |
|         20 |
|         20 |
+------------+
```

Как видно из листинга 23.21, для первых двух строк результирующей таблицы листинга 23.20 вложенный запрос вернет по одной строке (id_product = 8, 10), для последней их будет три (id_product = 20). То есть ключевое слово EXISTS в действительности не использует результаты вложенного запроса, проверяется только число возвращаемых строк. Это означает, что в списке столбцов, следующих после ключевого слова SELECT вложенного запроса, вместо символа "*" может быть расположено любое допустимое имя — название столбца или просто цифра, на результатах это не отражается (листинг 23.22).

Листинг 23.22. Использование скалярного значения во вложенном запросе

```
mysql> SELECT id_product, name, price FROM products
    -> WHERE EXISTS (SELECT 7 FROM orders
    ->               WHERE orders.id_product = products.id_product);
+------------+-----------------------+---------+
| id_product | name                  | price   |
+------------+-----------------------+---------+
|          8 | Intel Pentium 4 3.0GHz | 6147.00 |
|         10 | Gigabyte GA-8I848P-RS  | 1896.00 |
|         20 | Maxtor 6Y120P0         | 2456.00 |
+------------+-----------------------+---------+
```

Важно отметить, что в WHERE-условии вложенного запроса допускается использование столбцов внешнего запроса при обращении к ним по полному имени. В листинге 23.22 к текущей товарной позиции из внешнего запроса во вложенном запросе можно обратиться по имени products.id_product.

Замечание

Обращение к текущей строке внешнего запроса из вложенного запроса называется внешней ссылкой и может быть использовано не только совместно с ключевым словом EXISTS, но и во всех других вложенных запросах. Запросы с использованием внешних ссылок называются коррелированными запросами и рассматриваются в *разделе 23.4*.

Задачу на формирование списка покупателей, совершивших хотя бы одну покупку (листинг 23.16), можно решить при помощи запроса, представленного в листинге 23.23.

Листинг 23.23. Извлечение списка покупателей, совершивших хотя бы одну покупку

```
mysql> SELECT name, surname FROM users
    -> WHERE EXISTS (SELECT * FROM orders
    ->               WHERE orders.id_user = users.id_user);
+-----------+----------+
| name      | surname  |
+-----------+----------+
| Александр | Иванов   |
```

```
| Игорь     | Симдянов |
| Александр | Корнеев  |
+-----------+----------+
```

Обратную задачу, т. е. извлечение списка покупателей, еще не совершавших ни одной покупки, можно легко решить, воспользовавшись отрицанием — `NOT EXISTS` (листинг 23.24) (в отличие от ключевого слова `ANY` (и `ALL`)).

Листинг 23.24. Извлечение списка покупателей, не совершавших покупок

```
mysql> SELECT name, surname FROM users
    -> WHERE NOT EXISTS (SELECT * FROM orders
    ->                   WHERE orders.id_user = users.id_user);
+----------+-----------+
| name     | surname   |
+----------+-----------+
| Сергей   | Лосев     |
| Максим   | Кузнецов  |
| Анатолий | Нехорошев |
+----------+-----------+
```

23.4. Коррелированные запросы

Во вложенном запросе часто требуется ссылаться на значение столбца в текущей строке внешнего запроса. Рассмотрим запрос, извлекающий из таблицы `products` названия приобретенных товаров, для которых число купленных товарных позиций совпадает с запасами на складе (листинг 23.25).

Листинг 23.25. Товарные позиции и их запас

```
mysql> SELECT name, count FROM products
    -> WHERE count = (SELECT SUM(count) FROM orders
    ->                WHERE orders.id_product = products.id_product);
+------------------------+-------+
| name                   | count |
+------------------------+-------+
| Intel Pentium 4 3.0GHz |     1 |
| Gigabyte GA-8I848P-RS  |     4 |
+------------------------+-------+
```

Во вложенном запросе вместо значения `products.id_product` подставляется текущее значение `id_product` из внешнего запроса. Вложенный запрос содержит ссылку на столбец таблицы `products`, несмотря на то, что конструкция `FROM` вложенного запроса не упоминает таблицу `products`. Такая связь внешнего запроса с внутренним называется внешней ссылкой. Вложенный запрос, содержащий внешнюю ссылку, называет-

ся коррелированным вложенным запросом, т. к. его результаты оказываются коррелированными с каждой строкой таблицы во внешнем запросе. По той же причине внешняя ссылка называется иногда коррелирующей ссылкой.

Вложенный запрос может иметь внешнюю ссылку на таблицу в конструкции `FROM` любого запроса, который содержит данный вложенный запрос, независимо от его уровня вложенности. Например, имя столбца во вложенном запросе четвертого уровня может относиться к одной из таблиц, указанных в конструкции `FROM` внешнего запроса, или к таблице в любом вложенном запросе, содержащем данный вложенный запрос четвертого уровня.

23.5. Вложенные запросы, возвращающие несколько столбцов

До сих пор рассматривались вложенные запросы, возвращающие единственный столбец, однако в СУБД MySQL реализованы так называемые строчные запросы, которые возвращают более одного столбца. Примеры таких запросов представлены в листинге 23.26.

Листинг 23.26. Пример строчных запросов

```
mysql> SELECT * FROM tbl1 WHERE (1,2) = (SELECT col1, col2 FROM tbl2);
mysql> SELECT * FROM tbl1 WHERE ROW(1,2) = (SELECT col1, col2 FROM tbl2);
```

Запрос возвращает строку, если в таблице `tbl2` присутствует запись, для которой справедливо равенство `col1 = 1` и `col2 = 2`.

Выражения "`(1,2)`" и "`ROW(1,2)`" называются конструкторами строки. Оба выражения являются эквивалентными, но в целях увеличения читабельности запроса предпочтительно использовать второй вариант.

Следует отметить, что конструктор строки может быть использован не только с вложенными запросами, но и в обычных запросах, как это представлено в листинге 23.27, где из таблицы `products` извлекаются параметры товарных позиций, число которых на складе равно 6, а оценка 4.

Листинг 23.27. Применение конструктора строки

```
mysql> SELECT name, price, count, mark FROM products
    -> WHERE (6,4) = (count,mark);
+--------------------------+---------+-------+------+
| name                     | price   | count | mark |
+--------------------------+---------+-------+------+
| Asustek P4P800-VM\L i865G | 2518.00 |     6 |  4.0 |
| ASUSTEK V9520X           | 1602.00 |     6 |  4.0 |
+--------------------------+---------+-------+------+
```

Запрос из листинга 23.27 можно переписать так, как это представлено в листинге 23.28.

Листинг 23.28. Альтернативный запрос без применения конструктора строки

```
mysql> SELECT name, price, count, mark FROM products
    -> WHERE count = 5 AND mark = 4;
+-----------------+---------+-------+------+
| name            | price   | count | mark |
+-----------------+---------+-------+------+
| Epox EP-4PDA3I  | 2289.00 |     5 |  4.0 |
| Samsung SP0812C | 2093.00 |     5 |  4.0 |
+-----------------+---------+-------+------+
```

Пусть требуется выяснить время оформления заказа на товар, цена которого превышает 1000 рублей, всеми посетителями со статусом 'gold' (листинг 23.29).

Листинг 23.29. Использование конструктора строки совместно с вложенным запросом

```
mysql> SELECT ordertime FROM orders
    -> WHERE (id_user, id_product) IN (SELECT users.id_user,
    ->                                        products.id_product
    ->                                 FROM users, products
    ->                                 WHERE products.price > 1000 AND
    ->                                       users.status = 'gold');
+---------------------+
| ordertime           |
+---------------------+
| 2005-02-10 09:40:29 |
+---------------------+
```

Как видно из запроса, представленного в листинге 23.29, после конструктора строки используется конструкция IN. Это связано с тем, что вложенный запрос может вернуть более одной строки и применение знака равенства "=" приведет к возникновению ошибки 1242 — вложенный запрос вернул более одной строки.

Замечание

При сравнении конструктора строки и результатов, которые вернул вложенный запрос, не следует применять операторы, отличные от равенства "=" и IN, т. к. результат может быть непредсказуемым.

Строчные запросы часто используются для сравнения таблиц друг с другом. Пусть таблицы tbl1 и tbl2 состоят из трех столбцов col1, col2 и col3. Тогда найти все строки таблицы tbl1, которые имеются также и в таблице tbl2, можно при помощи запроса, представленного в листинге 23.30.

Листинг 23.30. Вывод строк таблицы tbl1, присутствующих в таблице tbl2

```
mysql> SELECT col1, col2, col3 FROM tbl1
WHERE ROW(col1, col2, col3) IN (SELECT col1, col2, col3 FROM tbl2);
```

23.6. Подзапросы в конструкции *FROM*

Так как вложенные запросы возвращают результирующую таблицу, которая становится предметом дальнейших запросов, стандарт SQL разрешает использование вложенных запросов везде, где допускаются ссылки на таблицы. В частности, вложенный запрос может указываться вместо имени таблицы в предложении FROM. В листинге 23.31 представлен запрос, в котором в качестве одной из таблиц используется результирующая таблица вложенного запроса.

Листинг 23.31. Использование вложенных запросов в конструкции FROM

```
mysql> SELECT usr.surname, usr.name, orders.ordertime
    -> FROM orders,
    ->      (SELECT * FROM users) AS usr
    -> WHERE orders.id_user = usr.id_user;
+----------+-----------+---------------------+
| surname  | name      | ordertime           |
+----------+-----------+---------------------+
| Симдянов | Игорь     | 2005-01-04 10:39:38 |
| Корнеев  | Александр | 2005-02-10 09:40:29 |
| Иванов   | Александр | 2005-02-18 13:41:05 |
| Симдянов | Игорь     | 2005-03-10 18:20:00 |
| Симдянов | Игорь     | 2005-03-17 19:15:36 |
+----------+-----------+---------------------+
```

Ключевое слово AS, назначающее псевдоним результирующей таблице, является обязательным, т. к. каждая таблица в конструкции FROM должна иметь имя. Например, запрос, выводящий список всех сделок, с фамилиями и инициалами покупателей, наименованием товарных позиций и каталога, может выглядеть так, как это представлено в листинге 23.32. Запрос является трехтабличным, причем в качестве двух таблиц (users и goods) выступают вложенные запросы. Результирующая таблица users содержит лишь два столбца: id_user и name, в которые заносится фамилия покупателя и его инициалы, выделенные при помощи функции SUBSTRING() и объединенные в одну строку при помощи функции CONCAT(). Таблица goods имеет три поля:

- id_products — уникальный номер товарной позиции;
- productname — название товарной позиции;
- catalogname — название каталога.

Интересно отметить, что таблица goods сама сформирована при помощи двухтабличного запроса из таблиц products и catalogs.

Листинг 23.32. Использование вложенных запросов в конструкции FROM

```
mysql> SELECT users.name,
    ->        goods.productname,
    ->        goods.catalogname
    -> FROM orders,
    ->      (
    ->          SELECT id_user,
    ->                 CONCAT(surname, " ", SUBSTRING(name,1,1), ".",
    ->                        SUBSTRING(patronymic,1,1), ".") AS name
    ->          FROM users
    ->      ) AS users,
    ->      (
    ->          SELECT products.id_product AS id_product,
    ->                 products.name AS productname,
    ->                 catalogs.name AS catalogname
    ->          FROM products, catalogs
    ->          WHERE products.id_catalog = catalogs.id_catalog
    ->      ) AS goods
    -> WHERE orders.id_user = users.id_user AND
    ->       orders.id_product = goods.id_product;
+---------------+-----------------------+-------------------+
| name          | productname           | catalogname       |
+---------------+-----------------------+-------------------+
| Симдянов И.В. | Intel Pentium 4 3.0GHz | Процессоры        |
| Корнеев А.А.  | Gigabyte GA-8I848P-RS  | Материнские платы |
| Иванов А.В.   | Maxtor 6Y120P0         | Жесткие диски     |
| Симдянов И.В. | Maxtor 6Y120P0         | Жесткие диски     |
| Симдянов И.В. | Maxtor 6Y120P0         | Жесткие диски     |
+---------------+-----------------------+-------------------+
```

Рассмотрим другой пример. Пусть имеется запрос, который возвращает общее число товаров на складе для каждого из каталогов (листинг 23.33).

Листинг 23.33. Число товаров на складе для каждого из каталогов

```
mysql> SELECT id_catalog, SUM(count) FROM products GROUP BY id_catalog;
+------------+------------+
| id_catalog | SUM(count) |
+------------+------------+
|          1 | 47         |
|          2 | 27         |
|          3 | 17         |
|          4 | 26         |
|          5 | 85         |
+------------+------------+
```

Требуется определить среднее значение сумм в сгруппированной таблице. Вариант, представленный в листинге 23.34, недопустим.

Листинг 23.34. Ошибочный запрос

```
mysql> SELECT id_catalog, AVG(SUM(count))
    -> FROM products GROUP BY id_catalog;
ERROR 1111: Invalid use of group function
```

Выходом из подобной ситуации является использование запроса, представленного в листинге 23.33, в качестве вложенного запроса в предложении `FROM` (листинг 23.35).

Листинг 23.35. Среднее значение сумм

```
mysql> SELECT AVG(sumtbl.sum) FROM (SELECT id_catalog, SUM(count) AS sum
    ->                              FROM products
    ->                              GROUP BY id_catalog) AS sumtbl;
+-----------------+
| AVG(sumtbl.sum) |
+-----------------+
| 40.4000         |
+-----------------+
```

23.7. Вложенные запросы в операторе *CREATE TABLE*

При создании таблицы при помощи оператора `CREATE TABLE` ее можно заполнить, используя вложенный запрос, обращающийся к другой таблице.

Замечание

Синтаксис оператора `CREATE TABLE` подробно рассматривается в *главе 11*.

В листинге 23.36 представлен запрос, создающий таблицу `new_orders`, в которой в отличие от таблицы `orders` вместо внешних ключей, связывающих таблицу с другими таблицами (`users`, `products`), размещены столбцы для фамилии (`surname`), имени (`name`), отчества (`patronymic`) покупателя и названия товарной позиции (`product`).

Листинг 23.36. Заполнение таблицы при создании

```
mysql> CREATE TABLE new_orders (
    -> id_order INT(11) NOT NULL AUTO_INCREMENT,
    -> user TINYTEXT,
    -> ordertime DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00',
    -> number INT(11) NOT NULL DEFAULT '0',
    -> product TINYTEXT,
```

```
    -> PRIMARY KEY (id_order))
    -> ENGINE=MyISAM DEFAULT CHARSET=cp1251
    -> SELECT orders.id_order AS id_order,
    ->        CONCAT(users.surname, " ", SUBSTRING(users.name,1,1), ".",
    ->               SUBSTRING(users.patronymic,1,1), ".") AS user,
    ->        orders.ordertime AS ordertime,
    ->        orders.number AS number,
    ->        products.name AS product
    -> FROM orders, users, products
    -> WHERE orders.id_user = users.id_user AND
    ->       orders.id_product = products.id_product;
mysql> SELECT * FROM new_orders;
+---+---------------+---------------------+---+-----------------------+
| id_order | user   | ordertime           | number | product            |
+---+---------------+---------------------+---+-----------------------+
| 1 | Симдянов И.В. | 2005-01-04 10:39:38 | 1 | Intel Pentium 4 3.0GHz |
| 2 | Корнеев А.А.  | 2005-02-10 09:40:29 | 2 | Gigabyte GA-8I848P-RS  |
| 3 | Иванов А.В.   | 2005-02-18 13:41:05 | 4 | Maxtor 6Y120P0         |
| 4 | Симдянов И.В. | 2005-03-10 18:20:00 | 1 | Maxtor 6Y120P0         |
| 5 | Симдянов И.В. | 2005-03-17 19:15:36 | 1 | Maxtor 6Y120P0         |
+---+---------------+---------------------+---+-----------------------+
```

Как видно из листинга 23.36, для того чтобы заполнить только что созданную таблицу `new_orders`, достаточно в конце оператора `CREATE TABLE` поместить вложенный запрос. Столбцам во вложенном запросе необходимо назначать псевдонимы при помощи оператора `AS`. Если этого не сделать, запрос может завершиться аварийно или же будут получены нежелательные имена в конечной таблице.

Замечание

Данный вид запросов появился в версии 3.23, поэтому синтаксис не требует заключать вложенный запрос в обязательные круглые скобки.

Следует отметить, что столбцы, которые присутствуют в определении таблицы, но отсутствуют во вложенном запросе, получают значение по умолчанию (листинг 23.37).

Листинг 23.37. Пропуск столбца `number` во вложенном запросе

```
mysql> CREATE TABLE new_orders (
    -> id_order INT(11) NOT NULL AUTO_INCREMENT,
    -> user TINYTEXT,
    -> ordertime DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00',
    -> number INT(11) NOT NULL DEFAULT '0',
    -> product TINYTEXT,
    -> PRIMARY KEY (id_order))
    -> ENGINE=MyISAM DEFAULT CHARSET=cp1251
    -> SELECT orders.id_order AS id_order,
```

```
    ->        CONCAT(users.surname, " ", SUBSTRING(users.name,1,1), ".",
    ->               SUBSTRING(users.patronymic,1,1), ".") AS user,
    ->        orders.ordertime AS ordertime,
    ->        products.name AS product
    -> FROM orders, users, products
    -> WHERE orders.id_user = users.id_user AND
    ->       orders.id_product = products.id_product;
+---+---------------+---------------------+---+-----------------------+
| id_order | user   | ordertime           | number | product            |
+---+---------------+---------------------+---+-----------------------+
| 1 | Симдянов И.В. | 2005-01-04 10:39:38 | 0 | Intel Pentium 4 3.0GHz |
| 2 | Корнеев А.А.  | 2005-02-10 09:40:29 | 0 | Gigabyte GA-8I848P-RS  |
| 3 | Иванов А.В.   | 2005-02-18 13:41:05 | 0 | Maxtor 6Y120P0         |
| 4 | Симдянов И.В. | 2005-03-10 18:20:00 | 0 | Maxtor 6Y120P0         |
| 5 | Симдянов И.В. | 2005-03-17 19:15:36 | 0 | Maxtor 6Y120P0         |
+---+---------------+---------------------+---+-----------------------+
```

В листинге 23.37 в структуре таблицы `new_order` присутствует столбец `number`, однако во вложенном запросе, заполняющем только что созданную таблицу, столбец `number` пропущен. Это приводит к тому, что он получает значение по умолчанию — 0.

Важной особенностью применения вложенных запросов совместно с оператором `CREATE TABLE` является тот факт, что столбцы, не определенные в структуре оператора `CREATE TABLE`, но присутствующие в результирующей таблице вложенного запроса, добавляются в создаваемую таблицу (листинг 23.38).

Листинг 23.38. Использование оператора `CREATE TABLE`

```
mysql> CREATE TABLE new_orders (
    -> id_order INT(11) NOT NULL AUTO_INCREMENT,
    -> PRIMARY KEY (id_order))
    -> ENGINE=MyISAM DEFAULT CHARSET=cp1251
    -> SELECT orders.id_order AS id_order,
    ->        CONCAT(users.surname, " ", SUBSTRING(users.name,1,1), ".",
    ->               SUBSTRING(users.patronymic,1,1), ".") AS user,
    ->        orders.ordertime AS ordertime,
    ->        products.name AS product
    -> FROM orders, users, products
    -> WHERE orders.id_user = users.id_user AND
    ->       orders.id_product = products.id_product;
mysql> SELECT * FROM new_orders;
+--------+---------------+---------------------+-----------------------+
| id_order | user        | ordertime           | product               |
+--------+---------------+---------------------+-----------------------+
|      1 | Симдянов И.В. | 2005-01-04 10:39:38 | Intel Pentium 4 3.0GHz |
|      2 | Корнеев А.А.  | 2005-02-10 09:40:29 | Gigabyte GA-8I848P-RS  |
|      3 | Иванов А.В.   | 2005-02-18 13:41:05 | Maxtor 6Y120P0         |
```

```
|       4 | Симдянов И.В. | 2005-03-10 18:20:00 | Maxtor 6Y120P0         |
|       5 | Симдянов И.В. | 2005-03-17 19:15:36 | Maxtor 6Y120P0         |
+--------+---------------+---------------------+------------------------+
```

Несмотря на то, что оператор CREATE TABLE определяет один-единственный столбец id_order, созданная таблица new_orders содержит четыре столбца, три из которых появились благодаря присутствию в результирующей таблице вложенного SELECT-запроса.

Вложенный SELECT-запрос можно предварять ключевыми словами IGNORE или REPLACE, чтобы указать, как следует обрабатывать записи с дублированными значениями уникальных ключей (в случае их присутствия в таблице). Если указывается ключевое слово IGNORE, новые записи отбрасываются при обнаружении в таблице записей с аналогичным значением уникального ключа. Если же указано ключевое слово REPLACE, новые записи заменяют собой существующие. Если не указано ни IGNORE, ни REPLACE, то появление записей с дублированными значениями ключей приводит к ошибке.

23.8. Вложенные запросы в операторе *INSERT*

С помощью вложенного SELECT-запроса оператор INSERT позволяет быстро вставлять множество записей из другой (других) таблицы.

> ***Замечание***
>
> Данный вид запросов появился в версии 3.23, поэтому синтаксис не требует заключать вложенный запрос в обязательные круглые скобки.

Представленные в предыдущем разделе вложенные запросы позволяли заполнять таблицу new_orders только в момент создания. Однако если уже после создания таблицы покупатели оформляют сделки, то информация в таблице new_orders теряет свою актуальность. Выходом из ситуации будет создание таблицы new_orders при помощи обычного оператора CREATE TABLE (листинг 23.39) с последующим ее обновлением.

Листинг 23.39. Создание таблицы new_orders

```
mysql> CREATE TABLE new_orders (
    -> id_order INT(11) NOT NULL AUTO_INCREMENT,
    -> user TINYTEXT,
    -> ordertime DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00',
    -> number INT(11) NOT NULL DEFAULT '0',
    -> product TINYTEXT,
    -> PRIMARY KEY (id_order))
    -> ENGINE=MyISAM DEFAULT CHARSET=cp1251;
```

Теперь, когда таблица new_orders создана, ее можно заполнить при помощи оператора INSERT, представленного в листинге 23.40.

Листинг 23.40. Использование вложенного запроса в операторе INSERT

```
mysql> INSERT INTO new_orders
    -> SELECT orders.id_order AS id_order,
    ->        CONCAT(users.surname, " ", SUBSTRING(users.name,1,1), ".",
    ->               SUBSTRING(users.patronymic,1,1), ".") AS user,
    ->        orders.ordertime AS ordertime,
    ->        orders.number AS number,
    ->        products.name AS product
    -> FROM orders, users, products
    -> WHERE orders.id_user = users.id_user AND
    ->       orders.id_product = products.id_product;
mysql> SELECT * FROM new_orders;
+---+---------------+---------------------+---+-----------------------+
| id_order | user   | ordertime           | number | product           |
+---+---------------+---------------------+---+-----------------------+
| 1 | Симдянов И.В. | 2005-01-04 10:39:38 | 1 | Intel Pentium 4 3.0GHz |
| 2 | Корнеев А.А.  | 2005-02-10 09:40:29 | 2 | Gigabyte GA-8I848P-RS  |
| 3 | Иванов А.В.   | 2005-02-18 13:41:05 | 4 | Maxtor 6Y120P0         |
| 4 | Симдянов И.В. | 2005-03-10 18:20:00 | 1 | Maxtor 6Y120P0         |
| 5 | Симдянов И.В. | 2005-03-17 19:15:36 | 1 | Maxtor 6Y120P0         |
+---+---------------+---------------------+---+-----------------------+
```

После того, как таблица утратит свою актуальность, можно повторно выполнить запрос, добавив в условие WHERE ограничение по временному столбцу ordertime, которое отбросит записи, уже присутствующие в таблице.

Можно избежать дополнительного условия ordertime, воспользовавшись ключевыми словами REPLACE и IGNORE. В случае REPLACE все записи с дублирующим первичным ключом id_order будут заменены новыми. При использовании ключевого слова IGNORE записи в результирующей таблице вложенного запроса с первичным ключом id_order, которые уже присутствуют в таблице new_orders, будут отбрасываться (листинг 23.41).

Листинг 23.41. Использование ключевого слова IGNORE

```
mysql> INSERT IGNORE INTO new_orders
    -> SELECT orders.id_order AS id_order,
    ->        CONCAT(users.surname, " ", SUBSTRING(users.name,1,1), ".",
    ->               SUBSTRING(users.patronymic,1,1), ".") AS user,
    ->        orders.ordertime AS ordertime,
    ->        orders.number AS number,
    ->        products.name AS product
    -> FROM orders, users, products
    -> WHERE orders.id_user = users.id_user AND
    ->       orders.id_product = products.id_product;
```

В предложении FROM вложенного запроса может появляться имя таблицы, в которую производится добавление новых записей. Эта возможность появилась в СУБД MySQL, начиная с версии 4.0.14 (листинг 23.42).

Листинг 23.42. Вставка строк из текущей таблицы

```
mysql> INSERT IGNORE INTO orders SELECT * FROM orders;
```

Если число столбцов в таблице, в которую происходит вставка новых записей, и во вложенном запросе не совпадает и часть столбцов получает значения по умолчанию, или если порядок следования столбцов во вложенном запросе не совпадает с порядком следования столбцов в целевой таблице, можно воспользоваться конструкцией VALUES оператора INSERT, которая позволяет задать имена столбцов и их порядок при добавлении (листинг 23.43).

Листинг 23.43. Использование конструкции VALUES

```
mysql> INSERT INTO new_orders
    -> VALUES (id_order, ordertime, product, user)
    -> SELECT orders.id_order AS id_order,
    ->        orders.ordertime AS ordertime,
    ->        products.name AS product
    ->        CONCAT(users.surname, " ", SUBSTRING(users.name,1,1), ".",
    ->               SUBSTRING(users.patronymic,1,1), ".") AS user
    -> FROM orders, users, products
    -> WHERE orders.id_user = users.id_user AND
    ->       orders.id_product = products.id_product;
```

23.9. Ссылочная целостность

В *главе 5* рассмотрены понятия первичного ключа и внешнего ключа, которые образуют связь между таблицами. Так, ключ catalogs.id_catalog является первичным ключом таблицы catalogs, а ключ products.id_catalog — внешним ключом. Совокупность ключей catalogs.id_catalog и products.id_catalog образуют связь между таблицами catalogs и products.

Содержимое таблицы catalogs представлено в листинге 23.44.

Листинг 23.44. Содержимое таблицы catalogs

```
mysql> SELECT * FROM catalogs;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
```

```
|           3 | Видеоадаптеры      |
|           4 | Жесткие диски      |
|           5 | Оперативная память |
+-------------+--------------------+
```

Пусть в связи с реорганизацией электронного магазина требуется удалить каталог 'Жесткие диски'. Для этого можно прибегнуть к оператору DELETE, подобному приведенному в листинге 23.45.

Листинг 23.45. Удаление каталога 'Жесткие диски'

```
mysql> DELETE FROM catalogs WHERE name = 'Жесткие диски';
mysql> SELECT * FROM catalogs;
+-------------+--------------------+
| id_catalog | name                |
+-------------+--------------------+
|           1 | Процессоры         |
|           2 | Материнские платы  |
|           3 | Видеоадаптеры      |
|           5 | Оперативная память |
+-------------+--------------------+
```

После удаления раздела с первичным ключом id_catalog, равным 4, в таблице products остались товарные позиции, которые принадлежат удаленному разделу (листинг 23.46).

Листинг 23.46. Товарные позиции, которые не принадлежат ни одному каталогу

```
mysql> SELECT id_product, id_catalog, name FROM products
    -> WHERE id_catalog = 4;
+-------------+-------------+-----------------------------+
| id_product | id_catalog | name                          |
+-------------+-------------+-----------------------------+
|          20 |           4 | Maxtor 6Y120P0              |
|          21 |           4 | Maxtor 6B200P0              |
|          22 |           4 | Samsung SP0812C             |
|          23 |           4 | Seagate Barracuda ST3160023A |
|          24 |           4 | Seagate ST3120026A          |
+-------------+-------------+-----------------------------+
```

Такую ситуацию называют нарушением ссылочной целостности. В рамках реляционной модели таблицу catalogs называют предком, а products — потомком. Всего существует четыре типа изменений базы данных, которые могут нарушить ссылочную целостность.

1. *Добавление новой строки-потомка.* Добавление в электронный магазин товарной позиции, не принадлежащей ни одному из каталогов, т. е. если поле id_catalog

таблицы products принимает, например, значение 104. Следует отметить, что добавление в таблицу catalogs строки, для которой не имеется соответствия в таблице products, не вызывает противоречий, т. к. существование каталога без товарных позиций вполне допустимое явление.

2. *Обновление внешнего ключа в строке-потомке.* Это та же проблема, что и в предыдущей ситуации, но выраженная в иной форме. При обновлении поля id_catalog таблицы products, т. е. при переносе товарной позиции из одного каталога в другой, необходимо, чтобы каталог с новым значением id_catalog существовал в таблице catalogs, иначе товарная позиция опять повисает в воздухе.
3. *Удаление строки-предка.* Эта ситуация описана ранее в листингах 23.45 и 23.46. Удаление каталога из таблицы catalogs приводит к тому, что ряд товарных позиций принадлежат к несуществующему каталогу.
4. *Обновление первичного ключа в строке-предке.* Эта иная форма проблемы, рассмотренной в предыдущем пункте. При обновлении первичного ключа каталога у товарных позиций, которые относятся к данному каталогу, внешний ключ не меняется.

Средства поддержки ссылочной целостности языка SQL позволяют обрабатывать каждую из четырех описываемых ситуаций.

Замечание

В СУБД MySQL ссылочная целостность поддерживается, начиная с версии 3.23.44. Однако поддержка реализована только для таблиц InnoDB.

Первая проблема (добавление новой строки-потомка) решается путем проверки значений в столбцах внешнего ключа перед выполнением оператора INSERT. Если передаваемые оператору INSERT значения не равны ни одному из значений первичного ключа, то возвращается ошибка добавления записи.

Вторая проблема (обновление таблицы-потомка) решается аналогично, путем проверки нового значения внешнего ключа. Если такого значения среди первичных ключей таблицы-предка не обнаружено, возвращается ошибка.

Третья проблема (удаление строки-предка) может потребовать различной реакции в зависимости от контекста и предметной области. Так, применительно к электронному магазину при удалении каталога из таблицы catalogs могут быть применены следующие действия:

- не удалять каталог до тех пор, пока принадлежащие ему товарные позиции из таблицы products не будут перенесены в другой каталог;
- автоматически удалить все товарные позиции из каталога products, принадлежащие удаляемому каталогу;
- для товарных позиций, принадлежащих удаляемому каталогу, в поле id_catalog поместить значение NULL, которое будет сигнализировать о том, что каталог для этих товарных позиций не известен;

- для товарных позиций, принадлежащих удаляемому каталогу, в поле `id_catalog` поместить значение по умолчанию, отличное от `NULL`, например, значение `id_catalog` для каталога '`Разное`'.

Точно такие же проблемы возникают в четвертой ситуации (обновление первичного ключа в строке-предке). Для задания всех этих действий в стандарте SQL предусмотрены специальные ключевые слова. До этого момента при определении структуры таблицы явным образом обозначался только первичный ключ. Однако для того, чтобы задать правила удаления и обновления, требуется также явное указание вторичного ключа, которое задается конструкцией `FOREIGN KEY`, имеющей следующий синтаксис:

```
FOREIGN KEY [name_key] (col1, ...) REFERENCES tbl (tbl_col, ...)
[ON DELETE {CASCADE|SET NULL|NO ACTION|RESTRICT|SET DEFAULT}]
[ON UPDATE {CASCADE|SET NULL|NO ACTION|RESTRICT|SET DEFAULT}]
```

Данная конструкция позволяет задать внешний ключ с необязательным именем `name_key` на столбцах, которые задаются в круглых скобках (один или несколько). Ключевое слово `REFERENCES` указывает таблицу `tbl`, на которую ссылается внешний ключ, в круглых скобках указываются имена столбцов. Необязательные конструкции `ON DELETE` и `ON UPDATE` позволяют задать поведение СУБД при удалении и обновлении строк из таблицы-предка, соответственно. Параметры, следующие за этими ключевыми словами, имеют следующие значения.

- `CASCADE` — при удалении или обновлении записи, содержащей первичный ключ, записи со ссылками на это значение в таблице-потомке удаляются или обновляются автоматически.
- `SET NULL` — при удалении или обновлении записи, содержащей первичный ключ, в таблице-потомке значения внешнего ключа, ссылающегося на таблицу-предка, устанавливаются в `NULL`.
- `NO ACTION` — при удалении или обновлении записей, содержащих первичный ключ, с таблицей-потомком никаких дополнительных действий не производится.
- `RESTRICT` — если в таблице-потомке имеются записи, ссылающиеся на первичный ключ таблицы-предка, при удалении или обновлении записей с таким первичным ключом возвращается ошибка. СУБД не позволяет изменять или удалять запись с первичным ключом, пока не останется ни одной ссылки из таблицы-потомка.
- `SET DEFAULT` — согласно стандарту SQL, при удалении или обновлении первичного ключа в таблице-потомке для ссылающихся на него записей в поле внешнего ключа должно устанавливаться значение по умолчанию. Однако в СУБД MySQL это ключевое слово зарезервировано, но не обрабатывается.

Снабдим таблицы учебной базы данных `shop` внешними ключами. Следует помнить, что внешние ключи поддерживаются только таблицами типа InnoDB *(см. главу 11)*, поэтому все таблицы должны быть преобразованы к данному типу. Это можно осуществить либо при помощи оператора `ALTER TABLE`, либо создав таблицы заново с использованием оператора `CREATE TABLE`.

Листинг 23.47. Создание внешних ключей

```
mysql> CREATE TABLE catalogs (
    ->   id_catalog int(11) NOT NULL auto_increment,
    ->   name tinytext NOT NULL,
    ->   PRIMARY KEY (id_catalog)
    ->  ) ENGINE=InnoDB DEFAULT CHARSET=cp1251;
mysql> CREATE TABLE products (
    ->   id_product int(11) NOT NULL auto_increment,
    ->   name tinytext NOT NULL,
    ->   price decimal(7,2) default '0.00',
    ->   count int(11) default '0',
    ->   mark float(4,1) NOT NULL default '0.0',
    ->   description text,
    ->   id_catalog int(11) NOT NULL default '0',
    ->   PRIMARY KEY  (id_product),
    ->   FOREIGN KEY (id_catalog) REFERENCES catalogs (id_catalog)
    ->       ON DELETE CASCADE
    ->       ON UPDATE CASCADE
    -> ) ENGINE=InnoDB DEFAULT CHARSET=cp1251;
```

В листинге 23.47 в табличном пространстве InnoDB создаются таблицы `catalogs` и `products`. В таблице `products` поле `id_catalog` объявлено как внешний ключ (`FOREIGN KEY`) с правилом удаления и обновления `CASCADE`. Это означает, что обновление таблицы `catalogs` приводит к автоматическому обновлению таблицы `products` (листинг 23.48).

Листинг 23.48. Каскадное обновление

```
mysql> UPDATE catalogs SET id_catalog = 10 WHERE id_catalog = 4;
mysql> SELECT id_product, id_catalog, name FROM products;
+------------+------------+------------------------------+
| id_product | id_catalog | name                         |
+------------+------------+------------------------------+
|          1 |          1 | Celeron 1.8                  |
|          2 |          1 | Celeron 2.0GHz               |
|          3 |          1 | Celeron 2.4GHz               |
|          4 |          1 | Celeron D 320 2.4GHz         |
|          5 |          1 | Celeron D 325 2.53GHz        |
|          6 |          1 | Celeron D 315 2.26GHz        |
|          7 |          1 | Intel Pentium 4 3.2GHz       |
|          8 |          1 | Intel Pentium 4 3.0GHz       |
|          9 |          1 | Intel Pentium 4 3.0GHz       |
|         10 |          2 | Gigabyte GA-8I848P-RS        |
|         11 |          2 | Gigabyte GA-8IG1000          |
|         12 |          2 | Gigabyte GA-8IPE1000G        |
```

```
|          13 |           2 | Asustek P4C800-E Delux        |
|          14 |           2 | Asustek P4P800-VM\L i865G     |
|          15 |           2 | Epox EP-4PDA3I                |
|          16 |           3 | ASUSTEK A9600XT/TD            |
|          17 |           3 | ASUSTEK V9520X                |
|          18 |           3 | SAPPHIRE 256MB RADEON 9550    |
|          19 |           3 | GIGABYTE AGP GV-N59X128D      |
|          20 |          10 | Maxtor 6Y120P0                |
|          21 |          10 | Maxtor 6B200P0                |
|          22 |          10 | Samsung SP0812C               |
|          23 |          10 | Seagate Barracuda ST3160023A  |
|          24 |          10 | Seagate ST3120026A            |
|          25 |           5 | DDR-400 256MB Kingston        |
|          26 |           5 | DDR-400 256MB Hynix Original  |
|          27 |           5 | DDR-400 256MB PQI             |
|          28 |           5 | DDR-400 512MB Kingston        |
|          29 |           5 | DDR-400 512MB PQI             |
|          30 |           5 | DDR-400 512MB Hynix           |
+-------------+-------------+-------------------------------+
```

А удаление любого из каталогов в таблице `catalogs` приведет к тому, что все записи таблицы `products`, поле `id_catalog` которых равно первичному ключу удаляемого каталога, также будет удалено из таблицы.

Листинг 23.49. Каскадное удаление

```
mysql> DELETE FROM catalogs WHERE id_catalog > 2;
mysql> SELECT * FROM catalogs;
+-------------+--------------------+
| id_catalog | name               |
+-------------+--------------------+
|           1 | Процессоры         |
|           2 | Материнские платы  |
+-------------+--------------------+
mysql> SELECT id_product, id_catalog, name FROM products;
+-------------+-------------+--------------------------+
| id_product | id_catalog | name                      |
+-------------+-------------+--------------------------+
|           1 |           1 | Celeron 1.8               |
|           2 |           1 | Celeron 2.0GHz            |
|           3 |           1 | Celeron 2.4GHz            |
|           4 |           1 | Celeron D 320 2.4GHz      |
|           5 |           1 | Celeron D 325 2.53GHz     |
|           6 |           1 | Celeron D 315 2.26GHz     |
|           7 |           1 | Intel Pentium 4 3.2GHz    |
|           8 |           1 | Intel Pentium 4 3.0GHz    |
|           9 |           1 | Intel Pentium 4 3.0GHz    |
```

```
|         10 |          2 | Gigabyte GA-8I848P-RS     |
|         11 |          2 | Gigabyte GA-8IG1000       |
|         12 |          2 | Gigabyte GA-8IPE1000G     |
|         13 |          2 | Asustek P4C800-E Delux    |
|         14 |          2 | Asustek P4P800-VM\L i865G |
|         15 |          2 | Epox EP-4PDA3I            |
+------------+------------+---------------------------+
```

Как видно из листинга 23.49, удаление записей с первичным ключом приводит к автоматическому удалению ссылающихся на него записей с внешним ключом. Записи с внешними ключами сами могут выступать в качестве ссылок для внешних ключей других таблиц, и один-единственный запрос может осуществлять достаточно сложные преобразования базы данных. Удаление и обновление базы данных, когда таблицы связаны внешним ключом с правилом `CASCADE`, называется каскадным удалением или обновлением.

> ***Замечание***
>
> Внешний ключ может рекурсивно ссылаться на записи своей же таблицы. Это часто необходимо, когда требуется моделировать реферальные системы и вложенные каталоги при помощи одной и той же таблицы.

СУБД MySQL позволяет отключить проверку ограничений внешнего ключа. Для этого достаточно установить для системной переменной `FOREIGN_KEY_CHECKS` значение 0. Установка системных переменных осуществляется при помощи оператора `SET` (листинг 23.50).

Листинг 23.50. Управление системной переменной `FOREIGN_KEY_CHECKS`

```
mysql> SELECT @@FOREIGN_KEY_CHECKS;
+----------------------+
| @@FOREIGN_KEY_CHECKS |
+----------------------+
|                    1 |
+----------------------+
mysql> FOREIGN_KEY_CHECKS = 1;
```

Для того чтобы извлечь текущее значение системной переменной, ее имя следует предварить двумя символами @. Более подробно системные переменные и синтаксис оператора `SET` обсуждаются в *главе 27*.

К сожалению, внешние ключи доступны только при использовании таблиц типа InnoDB, которые уступают в скорости таблицам MyISAM. Однако каскадные удаления и обновления можно эмулировать и для других типов таблиц, используя вложенные запросы совместно с операторами `DELETE` и `UPDATE`. В листинге 23.51 представлен запрос, удаляющий из таблицы `orders` учебной базы данных `shop` все заказы, принадлежащие покупателю с фамилией "`Симдянов`".

Листинг 23.51. Использование вложенного запроса в операторе DELETE

```
mysql> SELECT id_order, surname FROM orders, users
    -> WHERE orders.id_user = users.id_user;
+----------+----------+
| id_order | surname  |
+----------+----------+
|        1 | Симдянов |
|        2 | Корнеев  |
|        3 | Иванов   |
|        4 | Симдянов |
|        5 | Симдянов |
+----------+----------+
mysql> DELETE FROM orders WHERE id_user = (SELECT id_user FROM users
    ->                                     WHERE surname = 'Симдянов');
mysql> SELECT id_order, surname FROM orders, users
    -> WHERE orders.id_user = users.id_user;
+----------+---------+
| id_order | surname |
+----------+---------+
|        2 | Корнеев |
|        3 | Иванов  |
+----------+---------+
```

Вложенный запрос возвращает первичный ключ таблицы users — id_user для покупателя 'Симдянов' и подставляет его в условие WHERE внешнего запроса, по которому производится удаление из таблицы orders.

ГЛАВА 24

Транзакции и блокировки

Как правило, изменения базы данных, обусловленные внешними явлениями, требуют выполнения нескольких запросов. Так при покупке товара в электронном магазине `shop` требуется добавить запись в таблицу `orders` для данного заказа и уменьшить число товарных позиций на складе. В промышленных базах данных одно событие может затрагивать гораздо большее число таблиц и требовать выполнения многочисленных запросов.

Если на этапе выполнения одного из запросов происходит сбой, это может нарушить целостность базы данных. Например, товар может быть продан, а число товарных позиций на складе не обновлено; деньги могут быть сняты с одного счета, но не переведены на другой и т. п. Чтобы сохранить целостность базы данных, все изменения должны выполняться как единое целое — либо все изменения успешно выполняются, либо, в случае сбоя, хотя бы одно из изменений базы данных принимает состояние, которое было до начала изменений. Это обеспечивается средствами обработки транзакций, описываемых в данной главе.

24.1. Транзакции

Транзакция — это последовательность операторов SQL, выполняющихся как единая операция, которая не прерывается другими клиентами. То есть, пока происходит работа с записями таблицы (их обновление или удаление), никто другой не может получить доступ к этим записям, т. к. СУБД MySQL автоматически блокирует доступ к ним.

Замечание

Таблицы ISAM, MyISAM и HEAP не поддерживают транзакции. В настоящий момент их поддержка осуществляется только в таблицах BDB и InnoDB.

Транзакции позволяют объединять операторы в группу и гарантировать, что все операции внутри группы будут выполнены успешно. Если часть транзакции выполняется со сбоем, результаты выполнения всех операторов транзакции до места сбоя отменяются, приводя базу данных к виду, в котором она была до выполнения транзакции.

Системы, поддерживающие возможности транзакций, часто еще характеризуют как обеспечивающие свойства ACID (ACID является сокращением Atomic (атомарный), Consistent (целостный), Isolated (изолированный), Durable (длительный)).

- Атомарность. Операторы транзакции формируют единый логический блок, каждый элемент которого невозможно выполнить без выполнения всех остальных элементов блока.
- Целостность. База данных является целостной до и после выполнения транзакции.
- Изоляция. Отдельные транзакции не влияют на работу друг друга.
- Длительность. При выполнении транзакции все ее результаты сохраняются в базе данных.

Для выяснения механизма транзакций рассмотрим ситуацию, которая возможна при работе с электронным магазином `shop`. Предположим, что два покупателя одновременно начинают покупку процессора Celeron D 320 2.4GHz, который остался на складе в одном экземпляре.

Листинг 24.1. Покупка Celeron D 320 2.4GHz

```
mysql> SELECT id_product, name, @total := count FROM products
    -> WHERE name = 'Celeron D 320 2.4GHz';
+------------+----------------------+-------+
| id_product | name                 | count |
+------------+----------------------+-------+
|          4 | Celeron D 320 2.4GHz |     1 |
+------------+----------------------+-------+
```

Первый покупатель оформляет покупку, при этом происходят следующие события:

1. При помощи оператора `SELECT` запрашивается число товарных позиций на складе (листинг 24.1), которое помещается в переменную `@total` — дальнейшая регистрация покупки разрешена, если число процессоров больше нуля.
2. Регистрируется факт покупки в таблице `orders` — для этого в таблицу вставляется новая запись при помощи оператора `INSERT`.
3. Обновляется количество товарных позиций, доступных на складе, с первичным ключом `id_product` = 8 (таблица `products`). Для этого из переменной `@total` вычитается единица, и полученная цифра присваивается полю `count` товарной позиции (листинг 24.2).

Листинг 24.2. Обновление числа товарных позиций

```
mysql> UPDATE products SET count = @total - 1
    -> WHERE name = 'Celeron D 320 2.4GHz';
```

Одновременно второй покупатель начинает оформление покупки. Запрос на выборку количества процессоров для него также возвращает значение 1, которое помещается в

локальную переменную `@total`, видимую только в рамках текущего соединения с базой данных.

После регистрации покупки первым покупателем число товарных позиций, оставшихся на складе, обновляется с 1 на 0 (листинг 24.2).

После регистрации покупки вторым покупателем также происходит обновление таблицы `products` (листинг 24.2), но т. к. текущее соединение ничего не знает о запросе, параллельно выполняющемся в другом соединении, то баланс по результату двух покупок будет 1 процессор, в то время как покупатели заплатят за 2 и будут ожидать поставки двух процессоров.

Для предотвращения такой ситуации выполнение всех трех этапов покупки должно выполняться как одна транзакция.

Другая проблема, связанная с выполнением группы операций, заключается в том, что при возникновении ошибки в середине группы операторов база данных останется в наполовину модифицированном состоянии.

Если при оплате покупки происходит перевод денежной суммы со счета клиента на счет электронного магазина, то счет клиента должен уменьшиться на сумму `sum`, а счет электронного магазина увеличиться на ту же сумму (листинг 24.3).

Листинг 24.3. Обновление счетов

```
UPDATE account SET balance = balance - sum WHERE name = 'client';
UPDATE account SET balance = balance + sum WHERE name = 'bookshop';
```

Если в момент завершения первого запроса и перед началом выполнения второго произойдет сбой, транзакция окажется незавершенной.

Следует обратить внимание, что транзакции имеют смысл только в случае с типами таблиц, которые их поддерживают: InnoDB и BDB. Если существующие таблицы имеют другой тип, например MyISAM, для работы с транзакциями его следует изменить при помощи оператора `ALTER TABLE` (листинг 24.4).

Листинг 24.4. Изменение типа таблиц

```
mysql> ALTER TABLE catalogs ENGINE = InnoDB;
mysql> ALTER TABLE products ENGINE = InnoDB;
```

Следует особенно внимательно следить, чтобы при изменении типа MyISAM на любой другой в таблице отсутствовали `FULLTEXT`-индексы, т. к. никакой другой тип таблиц их не поддерживает.

По умолчанию СУБД MySQL работает в режиме автоматического завершения транзакций, т. е. как только выполняется оператор обновления данных, который модифицирует таблицу, MySQL тут же сохраняет изменения на диске. Для объединения в транзакцию нескольких операторов необходимо отключить этот режим. Это можно осуществить при помощи системной переменной `AUTOCOMMIT`, значение которой выставляется при помощи оператора `SET` (листинг 24.5).

Листинг 24.5. Отключение режима автоматического завершения транзакций

```
mysql> SET AUTOCOMMIT=0;
```

После отключения режима автоматического завершения транзакций следует использовать оператор COMMIT, чтобы сохранять изменения на диске, либо ROLLBACK, чтобы отменять изменения, выполненные с момента начала транзакции (листинг 24.6). Для того чтобы включить режим автоматического завершения транзакций, необходимо выполнить оператор SET AUTOCOMMIT=1.

Листинг 24.6. Использование транзакций

```
mysql> SET AUTOCOMMIT=0;
mysql> INSERT INTO catalogs VALUES(NULL,'Периферия');
mysql> INSERT INTO catalogs VALUES(NULL,'Разное');
mysql> SELECT * FROM catalogs;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 | Оперативная память |
|          6 | Периферия          |
|          7 | Разное             |
+------------+--------------------+
mysql> ROLLBACK;
mysql> SELECT * FROM catalogs;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 | Оперативная память |
+------------+--------------------+
mysql> INSERT INTO catalogs VALUES(NULL,'Периферия');
mysql> INSERT INTO catalogs VALUES(NULL,'Разное');
mysql> COMMIT;
mysql> SELECT * FROM catalogs;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
```

```
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 | Оперативная память |
|          9 | Периферия          |
|         10 | Разное             |
+------------+--------------------+
mysql> SET AUTOCOMMIT=1;
```

В листинге 24.6 в таблицу `catalogs` добавляются новые записи, при этом выполнение оператора `ROLLBACK` позволяет отменить все изменения, которые были произведены над таблицей `catalogs`. Напротив, оператор `COMMIT` сохраняет изменения на диске и уже следующий оператор `ROLLBACK` вернет последнее сохраненное состояние.

Для того чтобы включить режим автоматического завершения транзакций только для отдельной последовательности операторов, можно воспользоваться оператором `START TRANSACTION` (листинг 24.7).

Листинг 24.7. Использование оператора `START TRANSACTION`

```
mysql> START TRANSACTION;
mysql> SELECT @total := count FROM products
    -> WHERE name = 'Celeron D 320 2.4GHz';
+-----------------+
| @total := count |
+-----------------+
|               1 |
+-----------------+
mysql> UPDATE products SET count = @total - 1
    -> WHERE name = 'Celeron D 320 2.4GHz';
mysql> COMMIT;
```

После выполнения оператора `START TRANSACTION` режим автоматического завершения транзакций остается включенным до явного завершения транзакции с помощью оператора `COMMIT` или отката транзакции посредством `ROLLBACK`.

Замечание

Оператор `SET TRANSACTION` появился в СУБД MySQL, начиная с версии 4.0.11, и имеет два синонима: `BEGIN` и `BEGIN WORK`, появившихся еще в версии 3.23. Однако рекомендуется использовать именно `SET TRANSACTION`.

Для некоторых операторов нельзя выполнить откат при помощи оператора `ROLLBACK`. К их числу относят `CREATE INDEX`, `DROP INDEX`, `CREATE TABLE`, `DROP TABLE`, `TRUNCATE TABLE`, `ALTER TABLE`, `RENAME TABLE`, `CREATE DATABASE`, `DROP DATABASE`, `ALTER DATABASE`. Следует избегать помещать их в транзакции с другими операторами.

Кроме того, существует ряд операторов, которые неявно завершают транзакцию, как если бы был вызван оператор `COMMIT`. К их числу относятся: `ALTER TABLE`, `BEGIN`, `CREATE INDEX`, `CREATE TABLE`, `CREATE DATABASE`, `DROP DATABASE`, `DROP INDEX`, `DROP`

TABLE, DROP DATABASE, LOAD MASTER DATA, LOCK TABLES, RENAME, SET AUTOCOMMIT=1, START TRANSACTION, TRUNCATE TABLE.

Замечание

Операторы CREATE TABLE, TRUNCATE TABLE, DROP DATABASE и CREATE DATABASE приводят к неявному завершению транзакции, начиная с версии MySQL 4.1.13 и 5.0.8.

Транзакции не могут быть вложенными, что связано с тем, что любой оператор, начинающий транзакцию, приводит к завершению предыдущей транзакции.

Начиная с версий MySQL 4.0.14 и 4.1.1, для таблиц типа InnoDB поддерживаются операторы SAVEPOINT и ROLLBACK TO SAVEPOINT, которые позволяют работать с именованными точками начала транзакции (листинг 24.8).

Листинг 24.8. Работа с именованными точками начала транзакции

```
mysql> SAVEPOINT cheops;
mysql> INSERT INTO catalogs VALUES(NULL,'Периферия');
mysql> INSERT INTO catalogs VALUES(NULL,'Разное');
mysql> ROLLBACK TO SAVEPOINT cheops;
```

В листинге 24.8 оператор SAVEPOINT устанавливает именованную точку начала транзакции с именем cheops. Если текущая транзакция имеет точку сохранения с таким же именем, старая точка удаляется и устанавливается новая. Оператор ROLLBACK TO SAVEPOINT cheops откатывает транзакцию к состоянию, в котором находилась база данных на момент установки именованной точки, с помощью оператора SAVEPOINT. Все точки сохранения транзакций удаляются, если выполняется оператор COMMIT или ROLLBACK без указания имени точки сохранения.

24.2. Блокировка таблиц

Для таблиц типа MyISAM использование транзакций недоступно. Однако их можно эмулировать при помощи операторов LOCK TABLES и UNLOCK TABLES. В отличие от полноценных транзакций, данные операторы блокируют всю таблицу, в результате чего никто не может работать с таблицами до тех пор, пока они остаются заблокированными. Оператор LOCK TABLES выполняет блокировку таблиц, а UNLOCK TABLES снимает блокировку.

Замечание

Все таблицы, заблокированные в текущем соединении, разблокируются при повторном вызове оператора LOCK TABLES.

Замечание

Операторы LOCK TABLES и UNLOCK TABLES имеют синонимы LOCK TABLE и UNLOCK TABLE соответственно.

Листинг 24.9. Использование операторов LOCK TABLES и UNLOCK TABLES

```
mysql> LOCK TABLES catalogs WRITE;
mysql> INSERT INTO catalogs VALUES(NULL,'Периферия');
mysql> INSERT INTO catalogs VALUES(NULL,'Разное');
mysql> UNLOCK TABLES;
```

Листинг 24.9 демонстрирует блокировку таблицы catalogs на время добавления данных в базу данных. Следует обратить внимание, что после оператора LOCK TABLES указывается имя блокируемой таблицы, при снятии блокировки при помощи оператора UNLOCK TABLES указания имени таблицы уже не требуется. Основная причина применения блокировки таблицы при помощи LOCK TABLES — это увеличение скорости обновления таблиц и добавления больших объемов данных.

При использовании блокировок можно явно указать тип блокировки — на чтение (READ) или на запись (WRITE) (листинг 24.10).

Листинг 24.10. Детализация типа блокировки

```
mysql> LOCK TABLES catalogs WRITE;
mysql> INSERT INTO catalogs VALUES(NULL,'Периферия');
mysql> INSERT INTO catalogs VALUES(NULL,'Разное');
mysql> UNLOCK TABLES catalogs;
mysql> LOCK TABLES catalogs READ;
mysql> INSERT INTO catalogs VALUES(NULL,'Периферия');
mysql> INSERT INTO catalogs VALUES(NULL,'Разное');
mysql> UNLOCK TABLES;
```

Различие между блокировками на чтение (READ) и запись (WRITE) заключается в том, что клиент, установивший блокировку на чтение, и остальные клиенты могут только читать из таблицы данные. При блокировке на запись (WRITE) установивший ее клиент может как вносить записи в таблицу, так и читать, в то время как доступ других клиентов к таблице блокируется. Причем все остальные клиенты ожидают, когда блокировка будет отменена оператором UNLOCK TABLES. Следует отметить, что время ожидания может быть значительным и это следует предусмотреть в клиентской программе (листинг 24.11).

Листинг 24.11. Время блокировки может быть значительным

```
mysql> LOCK TABLES products READ;
Query OK, 0 rows affected (331.67 sec)
```

Один оператор LOCK TABLES может блокировать сразу несколько таблиц (листинг 24.12), причем рекомендуется блокировать все таблицы, которые участвуют в запросах внутри блокировки. Это связано с тем, что пока блокировка, установленная LOCK TABLES,

активна, невозможно получить доступ ни к каким таблицам, которые не были блокированы этим оператором.

Листинг 24.12. Блокировка нескольких таблиц

```
mysql> LOCK TABLES catalogs WRITE, products WRITE;
mysql> SELECT * FROM catalogs, products
    -> WHERE products.id_catalog = catalogs.id_catalogs
mysql> INSERT INTO catalogs VALUES(NULL,'Разное');
mysql> UNLOCK TABLES catalogs;
```

Если запрос обращается к таблице через псевдоним, требуется блокировать таблицу, используя тот же псевдоним. Блокировка таблицы не будет работать, если не указан псевдоним (листинг 24.13).

Листинг 24.13. Ошибка при использовании псевдонима

```
mysql> LOCK TABLES products READ;
mysql> SELECT * FROM products AS p;
ERROR 1100: Table 'p' was not locked with LOCK TABLES
```

И наоборот, если таблица блокирована по псевдониму, к ней следует обращаться, используя этот псевдоним (листинг 24.14).

Листинг 24.14. Обращение к таблице по псевдониму

```
mysql> LOCK TABLE products AS p READ;
mysql> SELECT * FROM products;
ERROR 1100: Table 'products' was not locked with LOCK TABLES
```

Блокировки по записи (`WRITE`) обычно имеют более высокий приоритет, чем блокировки по чтению (`READ`), чтобы гарантировать, что обновления данных пройдут как можно быстрее. Это означает, что если один или несколько клиентов устанавливают блокировку на чтение (`READ`), а затем другой клиент устанавливает блокировку на запись (`WRITE`) по этим же таблицам, то все остальные клиенты будут ожидать, пока блокировка по записи не будет снята.

Можно использовать ключевое слово `LOW_PRIORITY` совместно с ключевым словом `WRITE`, чтобы позволить другим клиентам устанавливать блокировку по чтению, пока клиент ожидает возможности установки блокировки по записи (листинг 24.15).

Листинг 24.15. Использование ключевого слова `LOW_PRIORITY`

```
mysql> LOCK TABLES catalogs LOW_PRIORITY WRITE;
mysql> INSERT INTO catalogs VALUES(NULL,'Периферия');
mysql> UNLOCK TABLES catalogs;
```

Однако при интенсивном обращении к серверу, когда таблицы постоянно блокированы по чтению, клиент, установивший блокировку с низким приоритетом, может не дождаться своей очереди.

Замечание

При установке блокировки по чтению использование ключевого слова `LOW_PRIORITY` не предусмотрено.

ГЛАВА 25

Управление учетными записями пользователей

СУБД MySQL является многопользовательской средой, это означает, что для доступа к таблицам базы данных могут быть созданы различные учетные записи с различным уровнем привилегий. Так, учетной записи редактора можно предоставить привилегии на просмотр таблицы, добавление новых записей и обновление уже существующих. Администратору базы данных можно предоставить более широкие полномочия, которые будут включать возможность создания таблиц, редактирования и удаления уже существующих. Для пользователя базы данных достаточно лишь просмотра таблиц. Данная глава посвящена созданию, удалению учетных записей, а также изменению уровня их привилегий.

25.1. Учетные записи СУБД MySQL

Следует отменить, что учетная запись является составной и принимает форму `'username'@'host'`, где `username` — имя, а `host` — наименование хоста, с которого пользователю `username` разрешено обращаться к серверу MySQL. Так, учетные записи `'root'@'127.0.0.1'` и `'wet'@'62.78.56.34'` означают, что пользователь с именем `root` может обращаться с хоста, на котором расположен сервер, а `wet` — только с хоста с IP-адресом `62.78.56.34`, ни с какого другого хоста обратиться к серверу MySQL с этим именем нельзя, в том числе и из-под IP-адреса `127.0.0.1`, относящегося к локальному хосту.

Замечание

Если имя пользователя и хост не содержат специальных символов "-" и "%", их необязательно брать в кавычки — `root@127.0.0.1`.

Замечание

IP-адрес `127.0.0.1` всегда относится к локальному хосту — если сервер и клиент установлены на одном хосте и сеть не используется, то сервер слушает соединения по этому адресу, а клиент отправляет на него SQL-запросы. Во всех операционных системах IP-адрес `127.0.0.1` имеет псевдоним `localhost`, поэтому вместо IP-адреса можно указывать этот псевдоним, а учетные записи вида `'root'@'127.0.0.1'` записывать в виде `'root'@'localhost'`.

Если к IP-адресу хоста привязано какое-то доменное имя, например `'softtime.ru'`, то вместо данного хоста можно использовать доменное имя `'root'@'softtime.ru'` — такая учетная запись означает, что из-под пользователя `root` можно обращаться к серверу с хоста, доменным именем которого является `softtime.ru`.

Если в дополнение к двум хостам `127.0.0.1` и `62.78.56.34` требуется разрешить пользователю `wet` обращаться к серверу MySQL с третьего хоста `158.0.55.62`, то потребуется создать три учетных записи: `'wet'@'127.0.0.1'`, `'wet'@'62.78.56.34'` и `'wet'@'158.0.55.62'`. Однако число адресов, с которых необходимо обеспечить доступ пользователю, может быть значительным и включать целые диапазоны. Для задания диапазона в имени хоста используется специальный символ "%", выполняющий ту же функцию, что и в операторе `LIKE`. Так, учетная запись `'wet'@'%'` позволяет пользователю `wet` обращаться к серверу MySQL с любых компьютеров сети. Символ "%" позволяет задавать диапазоны, поэтому учетная запись `'wet'@'%.softtime.ru'` разрешает обращаться к серверу MySQL поддоменам домена `softtime.ru`. Учетная запись `'wet'@'62.78.56.%'` позволяет пользователю `wet` получить доступ с хостов, обладающих IP-адресами в диапазоне от `62.78.56.0` до `62.78.56.255`.

Отсутствие части `host` в учетной записи аналогично использованию "`%`", таким образом, учетные записи `'wet'@'%'` и `'wet'` эквивалентны.

Все учетные записи хранятся в таблице `user` системной базы данных с именем `mysql`. После первой инсталляции содержимое таблицы `user` выглядит так, как это представлено в листинге 25.1.

Листинг 25.1. Содержимое таблицы `mysql.user`

```
mysql> SELECT Host, User, Password FROM mysql.user;
+---------------+------+----------+
| Host          | User | Password |
+---------------+------+----------+
| localhost     | root |          |
| %             | root |          |
+---------------+------+----------+
```

Если при инсталляции СУБД MySQL был отмечен флажок **Создание анонимного пользователя**, то к этим двум записям добавятся записи для анонимного пользователя — в качестве его имени выступает пустая строка (листинг 25.2).

Листинг 25.2. Анонимный пользователь и пользователь `root`

```
mysql> SELECT Host, User, Password FROM mysql.user;
+---------------+------+----------+
| Host          | User | Password |
+---------------+------+----------+
| localhost     | root |          |
| %             | root |          |
```

```
| localhost     |       |          |
| %             |       |          |
+---------------+-------+----------+
```

Пользователь root является уникальным пользователем, обладающим всеми привилегиями, анонимный пользователь по умолчанию обладает минимальными привилегиями, проще говоря, он может только авторизоваться и выполнять простейшие запросы вроде SELECT VERSION(). Изменение уровня привилегий будет рассмотрено далее в *разделе 25.5*, посвященном оператору GRANT.

Как видно из листингов 25.1 и 25.2, по умолчанию учетные записи не защищены паролем, что позволяет не указывать его при авторизации. Однако, если к хосту, на котором установлен сервер MySQL, имеют доступ несколько человек, разумнее защитить учетные записи паролем. Это можно сделать при помощи оператора SET PASSWORD, который имеет следующий синтаксис:

```
SET PASSWORD = PASSWORD('pass')
SET PASSWORD FRO user = PASSWORD('pass')
```

В первой форме оператор SET PASSWORD устанавливает пароль для текущего пользователя. Любой клиент, подключенный к серверу с использованием не анонимной учетной записи, может изменять свой пароль.

Таким образом, пользователь wet, осуществив авторизацию, может поменять свой пароль при помощи запроса, представленного в листинге 25.3.

Замечание

Пароль будет изменен для учетной записи с использованием того хоста, с которого авторизовался пользователь. То есть если пользователь меняет пароль на локальном хосте, это не приводит к смене пароля для сетевой учетной записи.

Листинг 25.3. Смена пароля текущим пользователем

```
mysql> SET PASSWORD = PASSWORD('Ht76W9yOWa');
```

Вторая форма оператора SET PASSWORD позволяет менять пароль пользователя для любой учетной записи (листинг 25.4). Только клиенты, имеющие доступ к базе данных mysql, имеют право использовать этот оператор. В качестве такого клиента, как правило, выступает root. Значение user задается в форме 'username'@'host'.

Листинг 25.4. Смена пароля произвольного пользователя

```
mysql> SET PASSWORD FOR 'wet'@'%' = PASSWORD('Ht76W9yOWa');
```

Следует отметить, что для всех учетных записей пароль обрабатывается при помощи функции PASSWORD(), это связано с тем, что MySQL не позволяет хранить пароли в незашифрованном виде. Пропуск функции PASSWORD() может привести к тому, что учетная запись будет недоступна для использования. Особенно это касается прямого

редактирования таблицы `user` при помощи оператора `UPDATE`, при котором опустить вызов функции `PASSWORD()` очень легко (листинг 25.5).

Листинг 25.5. Прямое редактирование системной таблицы `user`

```
mysql> UPDATE mysql.user SET Password = PASSWORD('Ht76W9yOWa')
    -> WHERE User = 'wet' AND Host = '%';
    -> FLUSH PRIVILEGES;
```

Запросы, показанные в листингах 25.4 и 25.5, эквивалентны, но настоятельно рекомендуется использовать запрос, представленный в листинге 25.4. В листинге 25.5 используется оператор `FLUSH PRIVILEGES` — это связано с тем, что таблицы системной базы данных `mysql` хранятся в памяти, и для того чтобы изменения вступили в силу, необходимо сохранить их на жестком диске.

Из листингов 25.1 и 25.2 видно, что по умолчанию для учетных записей суперпользователя (`root`) и анонимного пользователя создается также сетевой вариант учетной записи ("`%`"), позволяющий получить доступ к серверу базы данных с любого хоста локальной сети. В целях безопасности рекомендуется удалить эти учетные записи при помощи операторов `REVOKE` и `DROP USER`, которые рассматриваются далее.

25.2. Оператор *CREATE USER*

Оператор `CREATE USER` позволяет создать новую учетную запись и имеет следующий синтаксис:

```
CREATE USER user [IDENTIFIED BY [PASSWORD] 'password']
```

Оператор создает учетную запись `user` с необязательным паролем `password`. В листинге 25.6 представлен запрос, создающий пользователя `softtime`.

> ***Замечание***
>
> Оператор `CREATE USER` появился в СУБД MySQL, начиная с версии 5.0.2.

Листинг 25.6. Создание новой учетной записи при помощи оператора `CREATE USER`

```
mysql> CREATE USER softtime;
```

Если пароль не указывается, по умолчанию в его качестве выступает пустая строка. После того как новая учетная запись создана, в любом клиенте можно авторизоваться из пользователя `softtime`.

Листинг 25.7. Авторизация пользователя `softtime`

```
C:\mysql\bin>mysql -u softtime
```

Для того чтобы задать пароль, следует использовать ключевое слово `IDENTIFIED BY`, за которым в одиночных кавычках следует пароль.

Листинг 25.8. Защита учетной записи паролем

```
mysql> CREATE USER softtime IDENTIFIED BY '1234567';
```

Замечание

На данный момент оператор `CREATE USER` поддерживает имена хостов до 60 символов. Имя пользователя не может быть длиннее 16 символов.

Пароль, заданный в листинге 25.8 для учетной записи `softtime`, будет храниться в виде обычного текста. Разумнее хранить пароль в виде хэш-кода, полученного в результате необратимого шифрования. При авторизации пароль, вводимый пользователем, также подвергается необратимому шифрованию, после чего сравнению подвергаются хэш-коды. При такой системе авторизации злоумышленник, получивший в руки хэш-коды, вынужден будет потратить время на расшифровку паролей пользователя путем перебора. Если пароли выбраны правильно, т. е. не являются осмысленной фразой, а представляют набор букв и цифр, время, которое может потребоваться на их расшифровку, сделает их неактуальными. Для того чтобы воспользоваться таким механизмом авторизации, необходимо поместить между ключевым словом `IDENTIFIED BY` и паролем ключевое слово `PASSWORD`, которое сообщает, что пароль перед сохранением следует пропустить через функцию `PASSWORD()` (*см. главу 16* и листинг 25.9).

Замечание

При необратимом шифровании можно использовать только функцию `PASSWORD()`, использование других функций не предусмотрено.

Листинг 25.9. Использование ключевого слова `PASSWORD`

```
mysql> CREATE USER wet IDENTIFIED BY PASSWORD '123456';
```

При попытке создания учетной записи с уже существующим именем СУБД MySQL возвращает ошибку (листинг 25.10).

Листинг 25.10. Ошибка при создании уже существующей учетной записи

```
mysql> CREATE USER wet IDENTIFIED BY '123456';
ERROR 1396: Operation CREATE USER failed for 'wet'@'%'
```

Однако созданные при помощи оператора `CREATE USER` учетные записи не обладают никакими привилегиями — с использованием такой учетной записи невозможно просматривать таблицы и осуществлять запросы. Для наделения учетной записи привилегиями необходимо воспользоваться оператором `GRANT`.

Один оператор CREATE USER позволяет создать сразу несколько записей. Для этого необходимо перечислить определения учетных записей через запятую, как это продемонстрировано в листинге 25.11, где создаются сразу три учетных записи.

Листинг 25.11. Создание нескольких учетных записей

```
mysql> CREATE USER wet IDENTIFIED BY PASSWORD '123456',
    ->             ret IDENTIFIED BY PASSWORD '654321',
    ->             tor IDENTIFIED BY PASSWORD '678451';
```

По умолчанию, если не указывается хост, создается сетевая учетная запись, т. е. доступ разрешается с любого хоста сети. Для того чтобы создать учетную запись, разрешающую обращаться к серверу MySQL лишь с определенного хоста, например только с локального хоста (localhost), следует явно указать это в операторе CREATE USER (листинг 25.12).

Листинг 25.12. Создание нескольких учетных записей

```
mysql> CREATE USER 'wet'@'localhost' IDENTIFIED BY PASSWORD '123456',
    ->             'ret'@'localhost' IDENTIFIED BY PASSWORD '654321',
    ->             'tor'@'localhost' IDENTIFIED BY PASSWORD '678451';
```

25.3. Оператор *DROP USER*

Оператор DROP USER позволяет удалить учетную запись user и имеет следующий синтаксис:

```
DROP USER user
```

В качестве учетной записи user используется стандартная для MySQL запись вида 'username'@'host'.

Замечание

Оператор DROP USER появился в СУБД MySQL, начиная с версии 4.1.1.

Листинг 25.13. Использование оператора DROP USER

```
mysql> DROP USER 'wet'@'%';
```

Оператор DROP USER (листинг 25.13) не закрывает автоматически соединение удаляемого пользователя, который сможет работать с базой данных до тех пор, пока не будет закрыто его соединение или связь с сервером не оборвется.

Точно так же, как и оператор CREATE USER, оператор DROP USER позволяет удалять сразу несколько учетных записей (листинг 25.14).

Листинг 25.14. Удаление нескольких учетных записей

```
mysql> DROP USER 'wet'@'%', 'ret'@'%', 'tor'@'%';
```

Первоначально оператор `DROP USER` удалял только тех пользователей, которые не обладают никакими правами, однако, начиная с версии 5.0.2, порядок работы оператора `DROP USER` был изменен, и он приобрел возможность удалять любого пользователя с любым уровнем привилегий. До версии 5.0.2 перед использованием оператора `DROP USER` следовало убедиться, что пользователь не обладает никакими привилегиями. Для этого необходимо было выполнить оператор `SHOW GRANTS` (листинг 25.15). Если в результирующей таблице учетной записи нет, пользователь не обладает никакими правами и его учетная запись может быть удалена.

Замечание

Привилегии подробнее рассматриваются в *разделе 25.5*, посвященном оператору `GRANT`.

Замечание

Синтаксис оператора `SHOW` рассматривается в *главе 27*.

Листинг 25.15. Использование оператора `SHOW GRANTS`

```
mysql> SHOW GRANTS;
+-------------------------------------------------------------------+
| Grants for root@localhost                                         |
+-------------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' WITH GRANT OPTION |
+-------------------------------------------------------------------+
```

Как видно из листинга 25.15, только пользователь `'root'@'localhost'` обладает привилегиями, все остальные пользователи могут быть удалены при помощи оператора `DROP USER` (листинг 25.16).

Листинг 25.16. Текущие пользователи

```
mysql> SELECT Host, User, Password FROM mysql.user;
+-----------+----------+-------------------------------------------+
| Host      | User     | Password                                  |
+-----------+----------+-------------------------------------------+
| localhost | root     |                                           |
| %         | wet      | *6BB4837EB74329105EE4568DDA7DC67ED2CA2AD9 |
| localhost | softtime | *6A7A490FB9DC8C33C2B025A91737077A7E9CC5E5 |
+-----------+----------+-------------------------------------------+
```

До версии 4.1.1 удаление пользователя проводилось прямым редактированием таблицы `mysql.user` (листинг 25.17). Для этого также требовалось удалить все привилегии для удаляемого пользователя.

Листинг 25.17. Удаление пользователя прямым редактированием таблицы `mysql`

```
mysql> DELETE FROM mysql.user WHERE User = 'wet' AND Host = '%';
mysql> FLUSH PRIVILEGES;
```

25.4. Оператор *RENAME USER*

Оператор `RENAME USER` позволяет изменять имя пользователя в учетной записи и имеет следующий синтаксис

```
RENAME USER old_user TO new_user
```

Здесь `old_user` — старое имя пользователя, а `new_user` — новое имя пользователя. Пример использования оператора `RENAME USER` приведен в листинге 25.18.

Замечание

Оператор `RENAME USER` введен в СУБД MySQL, начиная с версии 5.0.2.

Листинг 25.18. Использование оператора `RENAME USER`

```
mysql> RENAME USER wet TO wet1;
mysql> SELECT Host, User, Password FROM mysql.user;
+-----------+----------+-------------------------------------------+
| Host      | User     | Password                                  |
+-----------+----------+-------------------------------------------+
| localhost | root     |                                           |
| %         | wet1     | *6BB4837EB74329105EE4568DDA7DC67ED2CA2AD9 |
| localhost | softtime | *6A7A490FB9DC8C33C2B025A91737077A7E9CC5E5 |
+-----------+----------+-------------------------------------------+
```

Замечание

На данный момент оператор `RENAME USER` поддерживает имена хостов до 60 символов. Имя пользователя не может быть длиннее 16 символов.

До версии 5.0.2 изменение имени пользователя в учетной записи проводилось прямым редактированием таблицы `mysql.user` (листинг 25.19).

Листинг 25.19. Изменение имени пользователя прямым редактированием

```
mysql> UPDATE mysql.user SET Uset = 'wet1'
    -> WHERE User = 'wet' AND Host = '%';
mysql> FLUSH PRIVILEGES;
```

25.5. Оператор *GRANT*

Рассмотренные выше операторы позволяют создавать, удалять и редактировать учетные записи, однако они не позволяют изменять привилегии пользователя, т. е. сообщать СУБД MySQL, какой пользователь имеет право только на чтение информации, какой на чтение и редактирование, а кому предоставлены права изменять структуру базы данных и создавать учетные записи для остальных пользователей.

Для решения этих задач предназначены операторы `GRANT` и `REVOKE`, оператор `GRANT` назначает привилегии пользователю, а `REVOKE` — удаляет. Если учетной записи, которая появляется в операторе `GRANT`, не существует, то она автоматически создается. Однако удаление всех привилегий с помощью оператора `REVOKE` не приводит к автоматическому уничтожению учетной записи — для полного удаления пользователя необходимо либо воспользоваться оператором `DROP USER`, либо удалить соответствующую запись из таблицы `mysql.user`.

В простейшем случае оператор `GRANT` выглядит так, как это представлено в листинге 25.20.

Листинг 25.20. Использование оператора `GRANT`

```
mysql> GRANT ALL ON *.* TO 'wet'@'localhost' IDENTIFIED BY 'pass';
```

Запрос в листинге 25.20 создает пользователя с именем `wet` и паролем `pass`, который может обращаться к серверу MySQL с локального хоста (`localhost`), и имеет все права (`ALL`) для всех баз данных (`*.*`). Если такой пользователь уже существует, то его привилегии будут изменены на `ALL`.

Замечание

На данный момент оператор `GRANT` поддерживает имена хостов, баз данных, таблиц и столбцов размером до 60 символов. Имя пользователя не может быть длиннее 16 символов.

Отобрать права у пользователя `'wet'@'localhost'` можно при помощи оператора `REVOKE`, представленного в листинге 25.21.

Листинг 25.21. Использование оператора `REVOKE`

```
mysql> REVOKE ALL ON *.* FROM 'wet'@'localhost';
```

Вместо ключевого слова `ALL`, которое обозначает все системные привилегии (кроме `GRANT OPTION` — передача привилегий другим пользователям), можно использовать любое из ключевых слов, представленных в табл. 25.1.

Замечание

Привилегии `CREATE TEMPORARY TABLES`, `EXECUTE`, `LOCK TABLES`, `REPLICATION CLIENT`, `REPLICATION SLAVE`, `SHOW DATABASES` и `SUPER` были добавлены в версии 4.0.2 (привилегия `EXECUTE` не выполняла никаких действий до версии 5.0.3).

Привилегии CREATE VIEW и SHOW VIEW были добавлены в версии 5.0.1, а CREATE USER, CREATE ROUTINE и ALTER ROUTINE в версии 5.0.3.

Таблица 25.1. *Системные привилегии, используемые в операторах GRANT и REVOKE*

Привилегия	Операция, разрешенная привилегией
ALL [PRIVILEGES]	Комбинация всех привилегий, за исключением привилегии GRANT OPTION, которая всегда задается отдельно или с предложением WITH GRANT OPTION
ALTER	Позволяет редактировать таблицу при помощи оператора ALTER TABLE
ALTER ROUTINE	Позволяет редактировать или удалять хранимую процедуру
CREATE	Позволяет создавать таблицу при помощи оператора CREATE TABLE
CREATE ROUTINE	Позволяет создавать хранимую процедуру
CREATE TEMPORARY TABLES	Позволяет создавать временные файлы
CREATE USER	Позволяет работать с учетными записями при помощи операторов CREATE USER, DROP USER, RENAME USER и REVOKE ALL PRIVILEGES
CREATE VIEW	Позволяет создавать представление при помощи оператора CREATE VIEW
DELETE	Позволяет удалять записи при помощи оператора DELETE
DROP	Позволяет удалять таблицы при помощи оператора DROP TABLE
EXECUTE	Позволяет выполнять хранимые процедуры
FILE	Позволяет работать с файлами при помощи операторов SELECT ... INTO OUTFILE и LOAD DATA INFILE
INDEX	Позволяет работать с индексами, в частности, использовать операторы CREATE INDEX и DROP INDEX
INSERT	Позволяет добавлять в таблицу новые записи при помощи оператора INSERT
LOCK TABLES	Позволяет осуществлять блокировки таблиц при помощи операторов LOCK TABLES и UNLOCK TABLES. Для вступления в действие этой привилегии должна быть установлена привилегия SELECT
PROCESS	Позволяет использовать оператор SHOW FULL PROCESSLIST
REFERENCES	Зарезервировано, но не реализовано
RELOAD	Позволяет использовать оператор FLUSH для обновления таблиц разрешений, кэша и журналов

Таблица 25.1 *(окончание)*

Привилегия	Операция, разрешенная привилегией
REPLICATION CLIENT	Позволяет запрашивать, является ли сервер главным или подчиненным в репликации
REPLICATION SLAVE	Данная привилегия необходима для подчиненных серверов репликации
SELECT	Позволяет осуществлять выборки таблиц при помощи оператора SELECT
SHOW DATABASES	Позволяет просматривать список всех таблиц на сервере MySQL при помощи оператора SHOW DATABASES
SHOW VIEW	Позволяет использовать оператор SHOW CREATE VIEW
SHUTDOWN	Позволяет завершать работу сервера при помощи mysqladmin shutdown
SUPER	Позволяет выполнять административные функции при помощи операторов CHANGE MASTER, KILL, PURGE MASTER LOGS и SET GLOBAL
UPDATE	Позволяет обновлять содержимое таблиц при помощи оператора UPDATE
USAGE	Синоним для статуса "отсутствуют привилегии"
GRANT OPTION	Позволяет управлять привилегиями других пользователей, без данной привилегии невозможно выполнить операторы GRANT и REVOKE

Рассмотрим несколько примеров. Для того чтобы разрешить пользователю editor полный доступ на просмотр, заполнение, редактирование и удаление таблиц, необходимо воспользоваться запросом, представленным в листинге 25.22.

Листинг 25.22. Предоставление привилегий для пользователя editor

```
mysql> GRANT SELECT, INSERT, DELETE, UPDATE ON *.* TO editor;
```

Если в качестве пользователя выступает приложение, которое добавляет новые записи или обновляет текущие, то его права можно ограничить лишь привилегиями INSERT и UPDATE (листинг 25.23), это позволит избежать выполнения операций DELETE и SELECT, которыми может воспользоваться злоумышленник при помощи инъекционного кода.

Листинг 25.23. Предоставление привилегий INSERT и UPDATE

```
mysql> GRANT INSERT, UPDATE ON *.* TO program;
```

Для того чтобы назначить все привилегии сразу, следует воспользоваться запросами, представленными в листинге 25.24.

Листинг 25.24. Предоставление всех привилегий

```
mysql> GRANT ALL ON *.* TO superuser;
mysql> GRANT GRANT OPTION ON *.* TO superuser;
```

Выполнить операции по наделению пользователя всеми правами в одном запросе не получится, т. к. ключевое слово ALL всегда употребляется отдельно и не должно использоваться совместно с другими ключевыми словами из табл. 25.1 (листинг 25.25).

Листинг 25.25. Ошибка при совместном использовании ALL и GRANT OPTION

```
mysql> GRANT ALL, GRANT OPTION ON *.* TO superuser;
ERROR 1064: You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near
' GRANT OPTION ON *.* TO superuser' at line 1
```

Однако использование ключевого слова USEAGE, означающего полное отсутствие привилегий, совместно с другими ключевыми словами из табл. 25.1, вполне допустимо (листинг 25.26).

Листинг 25.26. Совместное использование ключевого слова USAGE с SELECT

```
mysql> GRANT USAGE, SELECT ON *.* TO superuser;
```

Следует отметить, что один оператор GRANT может быть использован для предоставления привилегий сразу нескольким пользователям (листинг 25.27).

Листинг 25.27. Предоставление привилегий сразу нескольким пользователям

```
mysql> GRANT ALL OPTION ON *.* TO superuser, wet, toor;
```

Ключевое слово ON в операторе GRANT задает уровень привилегий, которые могут быть заданы на одном из четырех уровней, представленных в табл. 25.2.

Таблица 25.2. *Уровни привилегий*

Ключевое слово ON	Уровень
ON *.*	Глобальный уровень — касается всех баз данных и таблиц, входящих в их состав, таким образом, пользователь с полномочиями на глобальном уровне может обращаться ко всем базам данных
ON *	Если текущая база данных не была выбрана при помощи оператора USE, данное предложение эквивалентно ON *.*, если произведен выбор текущей базы данных, то устанавливаемые при помощи оператора GRANT привилегии относятся ко всем таблицам текущей базы данных

Таблица 25.2 *(окончание)*

Ключевое слово ON	Уровень
ON db.*	Уровень базы данных — данное предложение означает, что привилегии распространяются на таблицы базы данных db
ON db.tbl	Уровень таблицы — данное предложение означает, что привилегии распространяются на таблицу tbl базы данных db
ON db.tbl	Уровень столбца — привилегии уровня столбца касаются отдельных столбцов в таблице tbl базы данных db. Список столбцов указывается в скобках через запятую после ключевых слов SELECT, INSERT, UPDATE

Привилегии EXECUTION, FILE, PROCESS, RELOAD, REPLICATION CLIENT, REPLICATION SLAVE, SHOW DATABASES, SHUTDOWN и SUPER могут быть установлены лишь на глобальном уровне. Для таблиц можно установить только следующие типы привилегий: SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, GRANT OPTION, INDEX и ALTER. Это следует учитывать при использовании конструкции GRANT ALL, которая назначает привилегии на текущем уровне. Так, запрос уровня базы данных GRANT ALL ON db.* не предоставляет никаких глобальных привилегий, таких как FILE или RELOAD.

Замечание

Не обязательно запоминать контекст применения привилегий. Доступные в текущей версии привилегии, а также контекст их применения, можно уточнить при помощи оператора SHOW PRIVILEGES. Данный оператор введен в MySQL, начиная с версии 4.1.0.

Допустим, для работы с учебной базой электронного магазина требуется создать учетную запись. В этом случае можно воспользоваться запросом, представленным в листинге 25.28, где пользователю shop_editor предоставляются привилегии уровня базы данных.

Листинг 25.28. Предоставление привилегий уровня базы данных

```
mysql> GRANT ALL ON shop.* TO shop_editor;
```

Замечание

Следует отметить, что отсутствие конструкции IDENTIFIED BY приводит к тому, что в качестве пароля пользователя назначается пустая строка.

Если у пользователя нет привилегий на доступ к базе данных, имя базы данных не отображается в ответ на запрос SHOW DATABASES, если только у пользователя нет специальной привилегии SHOW DATABASES.

Следующий уровень, который мы рассмотрим, — это привилегии на уровне таблицы. Пусть объем работ возрос настолько, что потребовалась новая учетная запись для

менеджера по работе с клиентами, которому требуется только одна таблица — users. В этом случае запрос на создание такой учетной записи может выглядеть так, как это представлено в листинге 25.29.

Листинг 25.29. Предоставление привилегий уровня таблицы

```
mysql> GRANT ALL ON shop.user TO client_manager;
```

Если у пользователя нет привилегий на доступ к таблице, то эта таблица не отображается в ответ на запрос списка таблиц базы данных — SHOW TABLES.

В именах баз данных и таблиц допускается использование символов "%" и "_", оба символа имеют тот же смысл, что и в операторе LIKE, т. е. заменяют произвольное число и один символ соответственно. Это означает, что если требуется использовать символ "_" как часть имени базы данных, его необходимо экранировать обратным слэшем, иначе можно нечаянно предоставить доступ к базам данных, соответствующим введенному шаблону. Например, для базы данных list_user оператор GRANT может выглядеть так, как это представлено в листинге 25.30.

Листинг 25.30. Символ "_" в имени базы данных

```
mysql> GRANT ALL ON 'list\_user'.* TO 'wet'@'localhost';
```

Еще более гибкие средства для назначения привилегий предоставляются на уровне столбца. Для того чтобы предоставить доступ для столбца, требуется перечислить имена столбцов через запятую после ключевых слов SELECT, INSERT и UPDATE — использовать другие ключевые слова для предоставления привилегий на уровне столбца нельзя. Пусть пользователю wet предоставляется привилегия просмотра (SELECT) столбца name таблицы catalogs базы данных shop. В этом случае запрос, создающий такого пользователя, может выглядеть так, как это представлено в листинге 25.31.

Листинг 25.31. Предоставление привилегий уровня столбца

```
mysql> GRANT SELECT(name) ON shop.catalogs TO 'wet'@'localhost';
```

После этого, авторизовавшись с использованием учетной записи wet, просмотреть запросы с явным или неявным участием столбцов, отличных от name, в таблице shop.catalogs уже не удастся (листинг 25.32).

Листинг 25.32. Пользователь wet имеет право просматривать только столбец name

```
mysql> SELECT * FROM catalogs;
ERROR 1143: SELECT command denied to user 'wet'@'localhost' for column
'id_catalog' in table 'catalogs'
mysql> SELECT name FROM catalogs;
```

```
+--------------------+
| name               |
+--------------------+
| Процессоры         |
| Материнские платы  |
| Видеоадаптеры      |
| Жесткие диски      |
| Оперативная память |
+--------------------+
```

Как видно из листинга 25.32, запрос `SELECT * FROM catalogs` заканчивается неудачей с сообщением о том, что пользователь `'wet'@'localhost'` не имеет прав на просмотр содержимого столбца `id_catalog`. Однако запрос `SELECT name FROM catalogs` завершается успешно, т. к. в списке столбцов присутствует только столбец `name`, который пользователь `wet` имеет право просматривать. Такой вид прав доступа удобно использовать для скрытия конфиденциальной информации, например, пароля или счета.

Особняком в привилегиях стоит привилегия `GRANT OPTION` — способность наделять других пользователей правами на передачу пользовательских прав. Обычно эту привилегию назначают при помощи отдельного запроса (см. листинг 25.24), но язык запросов SQL предоставляет специальное предложение `WITH GRANT OPTION`, которое позволяет наделять учетную запись этой привилегией. Предложение `WITH GRANT OPTION` помещается в конце запроса (листинг 25.33).

Листинг 25.33. Использование предложения `WITH GRANT OPTION`

```
mysql> GRANT ALL ON shop.* TO 'wet'@'localhost' WITH GRANT OPTION;
```

В результате выполнения запроса из листинга 25.33 пользователь `wet` наделяется правами использования оператора `GRANT ALL` для предоставления привилегий другим пользователям на базу данных `shop` и ее таблиц. Ни на какие другие базы данных пользователь `wet` не имеет права выдавать привилегии, другими словами, пользователь, обладающий правами `GRANT OPTION`, может передавать другим пользователям только те права, которые принадлежат ему самому.

Следует крайне осторожно обращаться с привилегией `GRANT OPTION`, т. к. она распространяется не только на те привилегии, которыми пользователь обладает на настоящий момент, но и на все привилегии, которые он может получить в будущем.

Ключевое слово `WITH`, помимо привилегии `GRANT OPTION`, позволяет наложить ограничения на число подключений, запросов, обновлений и параллельных запросов в час, это осуществляется при помощи опций `MAX_CONNECTIONS_PER_HOUR`, `MAX_QUERIES_PER_HOUR`, `MAX_UPDATES_PER_HOUR` и `MAX_USER_CONNECTIONS` соответственно. Запрос в листинге 25.34 устанавливает для пользователя `wet` полный доступ к базе данных `shop`, но с ограничением — не больше 10 подключений к серверу MySQL в час и не более 1000 запросов, из которых только 200 могут быть операциями обновления `UPDATE`, а 3 запроса протекать одновременно.

Замечание

Опции MAX_QUERIES_PER_HOUR, MAX_UPDATES_PER_HOUR, MAX_CONNECTIONS_PER_HOUR появились в СУБД MySQL, начиная с версии 4.0.2. Опция MAX_USER_CONNECTIONS появилась в СУБД MySQL, начиная с версии 5.0.3.

Листинг 25.34. Ограничение учетной записи

```
mysql> GRANT ALL ON shop.* TO 'wet'@'localhost' IDENTIFIED BY 'pass'
    -> WITH MAX_CONNECTIONS_PER_HOUR 10
    ->      MAX_QUERIES_PER_HOUR 1000
    ->      MAX_UPDATES_PER_HOUR 10
    ->      MAX_USER_CONNECTIONS 3;
```

Если значение каждой из опции равно 0 (по умолчанию), это означает, что у пользователя нет ограничений по этому параметру.

MySQL позволяет шифровать весь сетевой трафик между сервером и клиентом при помощи протокола SSL.

Замечание

Протоколы Internet, в том числе и транспортный протокол TCP, который используется в СУБД MySQL, разрабатывались достаточно давно и не были изначально предназначены для коммерции и передачи конфиденциальных сведений. Для повышения безопасности в сети компания Netscape в свое время разработала протокол обмена данными, который гарантирует защиту транзакций. Этот протокол она назвала протоколом защищенных сокетов (SSL, Secure Sockets Layer). Это промежуточный протокол, расположенный между транспортным и прикладным уровнем, перед отправкой по сети он шифрует данные, отправляемые прикладными протоколами (HTTP, FTP, SMTP, SQL). Поэтому если злоумышленник с сетевым анализатором перехватит трафик — он получит не чистый текст, а зашифрованный. На другом конце происходит расшифровка.

Для того, чтобы явно потребовать от пользователя использовать те или иные средства шифрования данных, оператор GRANT снабжен ключевым словом REQUIRE, которое может принимать следующие формы.

- REQUIRE SSL — сообщает серверу, что использование данной учетной записи разрешается только для шифрованных SSL-подключений, попытки установить соединение по обычному нешифрованному каналу будут отвергаться.
- REQUIRE X509 — данная опция сообщает серверу, что клиент должен иметь действительный сертификат, но какой именно, кем и кому выданный — не имеет значения.
- REQUIRE ISSUER 'provider' — сообщает серверу, что клиент должен обладать сертификатом, выданным сертификационным центром provider. Даже если сертификат действительный, но выдан кем-то другим, то сервер откажет в предоставлении доступа.
- REQUIRE SUBJECT 'subject' — сообщает серверу, что сертификат должен быть выдан на имя subject.

- `REQUIRE CIPHER 'cod'` — данная опция необходима для того, чтобы гарантировать, что будет использоваться достаточно строгий шифр и длина ключа. SSL сам по себе может быть ослаблен, если применяются старые алгоритмы с короткими ключами шифрования. Используя эту опцию, можно потребовать конкретный метод шифрования, чтобы разрешить соединение.

Для того чтобы потребовать от пользователя `root` обращаться к серверу только по зашифрованному SSL-каналу, следует воспользоваться запросом, представленным в листинге 25.35.

Листинг 25.35. Использование опции `REQUIRE SSL`

```
mysql> GRANT ALL ON *.* TO 'root'@'*'
    -> IDENTIFIED BY 'secret' REQUIRE SSL;
```

Требование использования сертификата представлено в листинге 25.36.

Листинг 25.36. Использование опции `REQUIRE X509`

```
mysql> GRANT ALL ON *.* TO 'root'@'*'
    -> IDENTIFIED BY 'secret' REQUIRE X509;
```

Если сертификат должен быть выдан конкретным сертификационным центром, оператор `GRANT` может выглядеть так, как это представлено в листинге 25.37. Важным требованием в этом случае является то, что строка '`provider`' должна быть представлена как единое целое и не может разрываться символом перевода строки.

Листинг 25.37. Использование опции `REQUIRE ISSUER 'provider'`

```
mysql> GRANT ALL ON *.* TO 'root'@'*'
    -> IDENTIFIED BY 'secret'
    -> REQUIRE ISSUER '/C=FI/ST=Some-State/L=Helsinki/ O=MySQL Finland
AB/CN=Tonu Samuel/Email=tonu@example.com';
```

Если сертификат должен быть выдан на конкретное лицо, то оператор `GRANT` может выглядеть так, как это представлено в листинге 25.38. Здесь, так же как и в предыдущем примере, не допускается разрыв символом перевода строки.

Листинг 25.38. Использование опции `REQUIRE SUBJECT 'subject'`

```
mysql> GRANT ALL ON *.* TO 'root'@'*'
    -> IDENTIFIED BY 'secret'
    -> REQUIRE SUBJECT '/C=EE/ST=Some-State/L=Tallinn/O=MySQL demo client
certificate/CN=Tonu Samuel/Email=tonu@example.com';
```

Применение опции `REQUIRE CIPHER 'cod'` демонстрируется в листинге 25.39.

Листинг 25.39. Использование опции REQUIRE CIPHER 'cod'

```
mysql> GRANT ALL ON *.* TO 'root'@'*'
    -> IDENTIFIED BY 'secret'
    -> REQUIRE CIPHER 'EDH-RSA-DES-CBC3-SHA';
```

Опции SUBJECT, ISSURE и CIPHER могут комбинироваться в конструкции REQUIRE так, как это представлено в листинге 25.40.

Листинг 25.40. Комбинирование опций SUBJECT, ISSURE и CIPHER

```
mysql> GRANT ALL ON *.* TO 'root'@'*'
    -> IDENTIFIED BY 'secret'
    -> REQUIRE SUBJECT '/C=EE/ST=Some-State/L=Tallinn/O=MySQL demo client
certificate/CN=Tonu Samuel/Email=tonu@example.com';
    -> AND REQUIRE ISSUER '/C=FI/ST=Some-State/L=Helsinki/ O=MySQL Finland
AB/CN=Tonu Samuel/Email=tonu@example.com';
    -> AND REQUIRE CIPHER 'EDH-RSA-DES-CBC3-SHA';
```

Замечание

Начиная с версии MySQL 4.0.4, допускается необязательное слово AND между опциями REQUIRE.

25.6. Оператор *REVOKE*

Для отмены привилегий учетной записи используется оператор REVOKE (листинг 25.41). Его синтаксис похож на синтаксис оператора GRANT с той лишь разницей, что ключевое слово TO заменено на FROM, а предложения IDENTIFIED BY, REQUIRE и WITH GRANT OPTION отсутствуют.

Листинг 25.41. Использование оператора REVOKE

```
mysql> REVOKE DELETE, UPDATE ON shop.* FROM 'wet'@'localhost';
```

Следует помнить, что оператор REVOKE отменяет привилегии, но не удаляет учетные записи, для их удаления необходимо воспользоваться либо оператором DROP USER, либо удалить пользователя из таблицы mysql.user при помощи оператора DELETE.

До версии 4.1.2 нельзя было удалить сразу все привилегии, необходимо было выполнить два оператора (листинг 25.42).

Листинг 25.42. Удаление всех привилегий

```
mysql> REVOKE ALL ON shop.* FROM 'wet'@'localhost';
mysql> REVOKE GRANT OPTION ON shop.* FROM 'wet'@'localhost';
```

Начиная с версии 4.1.2, был добавлен синтаксис, позволяющий удалять сразу все привилегии при помощи одного запроса (листинг 25.43).

Листинг 25.43. Альтернативный синтаксис удаления всех привилегий

```
mysql> REVOKE ALL, GRANT OPTION ON shop.* FROM 'wet'@'localhost';
```

Синтаксис оператора `REVOKE` позволяет удалять привилегии сразу для нескольких пользователей (листинг 25.44).

Листинг 25.44. Синтаксис удаления всех привилегий для нескольких пользователей

```
mysql> REVOKE ALL, GRANT OPTION ON shop.*
    -> FROM 'wet'@'localhost',
    ->      'toor'@'localhost',
    ->      'ret'@'localhost';
```

ГЛАВА 26

Предотвращение катастроф и восстановление

СУБД является сложным образованием и требует постоянной поддержки и сопровождения. Поэтому проблеме восстановления данных следует уделить повышенное внимание. Для облегчения этой задачи база данных СУБД MySQL предоставляет развитые средства для анализа, восстановления и резервного копирования баз данных, которые будут рассмотрены в этой главе.

26.1. Оператор *CHECK TABLE*

Оператор `CHECK TABLE` проверяет таблицу или группу таблиц на наличие ошибок. Для таблиц MyISAM кроме всего прочего обновляется статистическая информация о ключах. Оператор `CHECK TABLE` работает только с таблицами MyISAM и BDB.

Замечание

Наряду с формой `CHECK TABLE`, может использоваться форма `CHECK TABLES`, которая является синонимом и производит те же действия, что и оператор `CHECK TABLE`.

Листинг 26.1. Использование оператора `CHECK TABLE`

```
mysql> CHECK TABLE orders, users, products, catalogs;
+---------------+-------+----------+-----------------------------------+
| Table         | Op    | Msg_type | Msg_text                          |
+---------------+-------+----------+-----------------------------------+
| shop.orders   | check | warning  | 1 client is using or hasn't closed
                                     the table properly                |
| shop.orders   | check | status   | OK                                |
| shop.users    | check | warning  | 1 client is using or hasn't closed
                                     the table properly                |
| shop.users    | check | status   | OK                                |
| shop.products | check | warning  | Table is marked as crashed        |
| shop.products | check | status   | OK                                |
| shop.catalogs | check | status   | OK                                |
+---------------+-------+----------+-----------------------------------+
```

Как видно из листинга 26.1, результирующая таблица состоит из четырех столбцов. Столбец `Table` содержит расширенное имя таблицы, над которой была произведена операция.

> ***Замечание***
>
> Операторы `ANALYZE TABLE`, `OPTIMIZE TABLE`, `REPAIR TABLE` и `BACKUP TABLE` возвращают результат своей работы точно в таком же формате. Второй столбец результирующей таблицы `Op` сообщает, какой из операторов был использован.

Столбец `Op` отражает тип операции над таблицей и может принимать одно из шести значений:

- `check` — сообщает о том, что над таблицей проведена операция проверки при помощи оператора `CHECK TABLE`;
- `analyze` — таблица проверена при помощи оператора `ANALYZE TABLE`;
- `optimize` — таблица была оптимизирована при помощи оператора `OPTIMIZE TABLE`;
- `repair` — была произведена операция восстановления поврежденных таблиц при помощи оператора `REPAIR TABLE`;
- `backup` — была произведена операция резервного копирования таблицы при помощи оператора `BACKUP TABLE`;
- `restore` — была произведена операция восстановления из резервной копии при помощи оператора `RESTORE TABLE`.

Третье поле `Msg_type` содержит информацию о типе сообщения, а четвертое `Msg_text` — само сообщение. Поле `Msg_type` может принимать одно из следующих значений:

- `status` — статус операции, который, как правило, равен `OK`. Если статус возвращает значение `Table already up to data` (Таблица уже обновлена), это означает, что с момента последней проверки изменений в таблице не проводилось, т. е. в ее проверке не было надобности;
- `error` — сообщает об ошибке;
- `info` — информационное сообщение;
- `warning` – предупреждение.

Для каждой из таблиц может быть выведено несколько сообщений, так в листинге 26.1, помимо того, что выводится статусное сообщение (`status`), для таблиц `orders` и `users` выводится предупреждение (`warning`), что один из клиентов использует таблицы или они не были корректно закрыты. При открытии таблицы СУБД MySQL помечает ее специальным флагом, который сбрасывается при корректном завершении работы сервера. Не сброшенный флаг сообщает серверу о необходимости проверки таблицы, именно на такой флаг и реагирует оператор `CHECK TABLE`.

Для таблицы `products` выводится предупреждение, что таблица может быть повреждена. Если оператор `CHECK TABLE` не находит никаких повреждений, он снимает эти

отметки (листинг 26.2), однако если найдено существенное повреждение, оператор CHECK TABLE сообщает о повреждении и при повторном использовании. В этом случае следует попытаться восстановить таблицу при помощи оператора REPAIR TABLE.

Замечание

В большинстве случаев повреждение таблицы связано с повреждениями индексов, а не данных — такого рода повреждения легко устраняются.

Листинг 26.2. Повторный вызов оператора CHECK TABLE

```
mysql> CHECK TABLE orders, users, products, catalogs;
+---------------+-------+----------+---------------------------+
| Table         | Op    | Msg_type | Msg_text                  |
+---------------+-------+----------+---------------------------+
| shop.orders   | check | status   | OK                        |
| shop.users    | check | status   | OK                        |
| shop.products | check | warning  | Table is marked as crashed |
| shop.products | check | status   | OK                        |
| shop.catalogs | check | status   | OK                        |
+---------------+-------+----------+---------------------------+
```

В конце оператора CHECK TABLE может быть указана одна из следующих опций или их комбинация (листинг 26.3):

- QUICK — не сканировать строки в поисках некорректных связей;
- FAST — проверять только те таблицы, которые были некорректно закрыты;
- CHANGED — проверять только те таблицы, которые изменились с момента последней проверки или не были корректно закрыты;
- MEDIUM — сканировать строки для проверки того, что удаленные связи в порядке. При этом также вычисляется контрольная сумма ключа строк и сверяется с контрольной суммой всех ключей;
- EXTENDED — выполнять полный поиск всех ключей во всех строках. Гарантирует стопроцентную целостность таблиц, но требует значительное время для проверки таблиц.

Листинг 26.3. Использование опций в операторе CHECK TABLE

```
mysql> CHECK TABLE orders, users, products, catalogs QUICK;
```

Если не указана ни одна из опций QUICK, MEDIUM или EXTENDED, то по умолчанию выбирается MEDIUM. Можно комбинировать несколько опций, как это продемонстрировано в листинге 26.4.

Листинг 26.4. Использование комбинации опций

```
mysql> CHECK TABLE orders, users, products, catalogs FAST QUICK;
```

Запрос в листинге 26.4 выполняет быструю проверку таблицы на предмет того, была ли она закрыта правильно.

Опции FAST и CHANGED наиболее подходят для применения в сценариях, например, запускаемых по расписанию при помощи UNIX-демона cron.

26.2. Оператор *ANALYZE TABLE*

Оператор ANALYZE TABLE анализирует и сохраняет распределение индексов таблицы (листинг 26.5). Это позволяет MySQL использовать в дальнейшем сохраненные данные для того, чтобы принимать решения относительно порядка объединения таблиц в многотабличных запросах.

Замечание

Данный оператор может применяться к таблицам типа MyISAM, BDB и InooDB. На время выполнения оператора ANALYZE TABLE доступ к таблице блокируется.

Замечание

Наряду с формой ANALYZE TABLE, может использоваться форма ANALYZE TABLES, которая является синонимом и производит те же действия, что и оператор ANALYZE TABLE.

Листинг 26.5. Использование оператора ANALYZE TABLE

```
mysql> ANALYZE TABLES catalogs, products;
+---------------+---------+----------+----------------------------+
| Table         | Op      | Msg_type | Msg_text                   |
+---------------+---------+----------+----------------------------+
| shop.catalogs | analyze | status   | OK                         |
| shop.products | analyze | status   | Table is already up to date |
+---------------+---------+----------+----------------------------+
```

Формат результирующей таблицы оператора ANALYZE TABLE совпадает с описанным в предыдущем разделе форматом для оператора CHECK TABLE. В случае успешного обновления индексов в четвертом столбце Msg_text появляется сообщение OK. Если с момента предыдущего применения оператора ANALYZE TABLE таблица не изменилась, то в столбце Msg_text будет присутствовать сообщение Table is already up to date.

26.3. Оператор *CHECKSUM TABLE*

Оператор CHECKSUM TABLE (листинг 26.6) возвращает контрольную сумму таблицы (или таблиц).

Замечание

Оператор CHECKSUM TABLE добавлен в MySQL, начиная с версии 4.1.1.

Листинг 26.6. Использование оператора CHECKSUM TABLE

```
mysql> CHECKSUM TABLE catalogs;
+---------------+-----------+
| Table         | Checksum  |
+---------------+-----------+
| shop.catalogs | 219379133 |
+---------------+-----------+
mysql> INSERT INTO catalogs VALUES(NULL, 'Переферия');
mysql> CHECKSUM TABLE catalogs;
+---------------+------------+
| Table         | Checksum   |
+---------------+------------+
| shop.catalogs | 1559532263 |
+---------------+------------+
mysql> UPDATE catalogs SET id_catalog = 10 WHERE id_catalog = 6;
mysql> CHECKSUM TABLE catalogs;
+---------------+------------+
| Table         | Checksum   |
+---------------+------------+
| shop.catalogs | 1559532263 |
+---------------+------------+
mysql> UPDATE catalogs SET name = '' WHERE name = 'Переферия';
mysql> CHECKSUM TABLE catalogs;
+---------------+------------+
| Table         | Checksum   |
+---------------+------------+
| shop.catalogs | 1895537962 |
+---------------+------------+
```

Изменения, проводимые над таблицей catalogs, такие как вставка новых записей, редактирование или удаление уже существующих, приводят к изменению контрольной суммы таблицы, поэтому, делая отметки, всегда можно контролировать, подвергалась таблица изменению или нет. Следует заметить, что изменение числового значения с 6 на 10 не приводит к изменению контрольной суммы таблицы (листинг 26.6). Оператор CHECKSUM TABLE может подсчитывать контрольную сумму сразу для нескольких таблиц — для этого их необходимо перечислить через запятую сразу после предложения CHECKSUM TABLE (листинг 26.7).

Листинг 26.7. Подсчет контрольной суммы для нескольких таблиц

```
mysql> CHECKSUM TABLE catalogs, products, users;
+---------------+------------+
| Table         | Checksum   |
+---------------+------------+
| shop.catalogs | 1895537962 |
```

```
| shop.products | 1706424754 |
| shop.users    | 1587572300 |
+---------------+------------+
```

Оператор CHECKSUM TABLE имеет две взаимоисключающие опции — QUICK и EXTENDED, которые располагаются сразу после списка таблиц. При создании таблицы при помощи оператора CREATE TABLE или в результате ее изменения при помощи ALTER TABLE можно потребовать от СУБД MySQL автоматически подсчитывать контрольную сумму, выставив опцию таблицы CHECKSUM в 1. Опция QUICK оператора CHECKSUM TABLE (листинг 26.8) позволяет извлекать контрольную сумму из счетчика, который ведется для таблицы — если учет суммы не ведется, оператор вернет NULL. Опция EXNTENDED игнорирует счетчик контрольной суммы и требует от оператора CHECKSUM TABLE провести полное сканирование таблицы.

Замечание

Использование включенной опции CHECKSUM в таблице приводит к замедлению операций вставок, обновления и удаления записей. Кроме того, в настоящий момент автоматический счетчик контрольной суммы поддерживает только один тип таблиц — MyISAM.

Листинг 26.8. Использование опции QUICK

```
mysql> CHECKSUM TABLE catalogs, products, users QUICK;
+---------------+----------+
| Table         | Checksum |
+---------------+----------+
| shop.catalogs |     NULL |
| shop.products |     NULL |
| shop.users    |     NULL |
+---------------+----------+
mysql> ALTER TABLE catalogs CHECKSUM = 1;
mysql> ALTER TABLE products CHECKSUM = 1;
mysql> ALTER TABLE users CHECKSUM = 1;
mysql> CHECKSUM TABLE catalogs, products, users QUICK;
+---------------+------------+
| Table         | Checksum   |
+---------------+------------+
| shop.catalogs | 1895537962 |
| shop.products | 1706424754 |
| shop.users    | 2410629935 |
+---------------+------------+
```

Если ни одна опция не указана, то MySQL возвращает контрольную сумму для таблиц с включенной опцией CHECKSUM и выполняет полное сканирование таблицы в противном случае.

26.4. Оператор *OPTIMIZE TABLE*

Оператор `OPTIMIZE TABLE` дефрагментирует таблицу, освобождая неиспользуемое свободное пространство. Такое свободное пространство может появляться в результате выполнения операторов `DELETE`, `REPLACE` и `UPDATE`. Ситуация усугубляется, если в таблице используются столбцы переменной длины.

Замечание

Данный оператор может применяться только к таблицам типа MyISAM и BDB. На время выполнения оператора `OPTIMIZE TABLE` доступ к таблице блокируется.

Замечание

Наряду с формой `OPTIMIZE TABLE`, может использоваться форма `OPTIMIZE TABLES`, которая является синонимом и производит те же действия, что и оператор `OPTIMIZE TABLE`.

Листинг 26.9. Использование оператора `OPTIMIZE TABLE`

```
mysql> OPTIMIZE TABLE catalogs, products, users, orders;
+---------------+----------+----------+----------+
| Table         | Op       | Msg_type | Msg_text |
+---------------+----------+----------+----------+
| shop.catalogs | optimize | status   | OK       |
| shop.products | optimize | status   | OK       |
| shop.users    | optimize | status   | OK       |
| shop.orders   | optimize | status   | OK       |
+---------------+----------+----------+----------+
```

Оператор `OPTIMIZE TABLE` выполняет следующие действия:

- дефрагментирует таблицы для уменьшения неиспользуемого пространства и уменьшения размера таблиц;
- объединяет содержимое строк переменной длины, которые стали фрагментированными (разбиты на несколько частей);
- по мере необходимости сортирует индексные страницы;
- обновляет внутреннюю табличную статистику.

В большинстве случаев вообще нет необходимости вызывать оператор `OPTIMIZE TABLE`. Даже если в таблицу постоянно вносятся изменения, а сама таблица содержит строки переменной длины, не рекомендуется проводить оптимизацию таблиц чаще, чем раз в неделю или месяц.

26.5. Оператор *REPAIR TABLE*

Оператор `REPAIR TABLE` (листинг 26.10) предназначен для выполнения восстановления поврежденных таблиц.

Замечание

Данный оператор может применяться только к таблицам типа MyISAM. На время выполнения оператора `REPAIR TABLE` доступ к таблице блокируется.

Замечание

Наряду с формой `REPAIR TABLE`, может использоваться форма `REPAIR TABLES`, которая является синонимом и производит те же действия, что и оператор `REPAIR TABLE`.

Листинг 26.10. Использование оператора `REPAIR TABLE`

```
mysql> REPAIR TABLE catalogs, products, users, orders;
+---------------+--------+----------+----------------------------------+
| Table         | Op     | Msg_type | Msg_text                         |
+---------------+--------+----------+----------------------------------+
| shop.catalogs | repair | note     | The storage engine for the table
                                       doesn't support repair           |
| shop.products | repair | status   | OK                               |
| shop.users    | repair | status   | OK                               |
| shop.orders   | repair | status   | OK                               |
+---------------+--------+----------+----------------------------------+
```

Если восстановление прошло успешно или таблица не являлась поврежденной, в четвертый столбец результирующей таблицы `Msg_text` помещается значение `OK`. Сообщение '`The storage engine for the table doesn't support repair`' появляется в том случае, если таблица имеет тип, отличный от MyISAM.

Оператор `REPAIR TABLE` имеет следующие опции:

- `QUICK` — быстрое восстановление, при котором восстанавливается только дерево индекса;
- `EXTENDED` — полное восстановление, в котором оператор просматривает таблицу строку за строкой;
- `USE_FRM` — используется (листинг 26.11), если отсутствует файл индексов с расширением MYI или заголовок данного файла поврежден. В этом случае СУБД MySQL создает файл MYI заново, используя информацию из файла с расширением frm.

Листинг 26.11. Использование опции `USE_FRM`

```
mysql> REPAIR TABLE catalogs, products, users, orders USE_FRM;
+---------------+--------+----------+----------------------------------+
| Table         | Op     | Msg_type | Msg_text                         |
+---------------+--------+----------+----------------------------------+
| shop.catalogs | repair | note     |The storage engine for the table
                                      doesn't support repair            |
```

```
| shop.products | repair | warning  |Number of rows changed from 0 to 31|
| shop.products | repair | status   | OK                                |
| shop.users    | repair | warning  | Number of rows changed from 0 to 6|
| shop.users    | repair | status   | OK                                |
| shop.orders   | repair | warning  | Number of rows changed from 0 to 2|
| shop.orders   | repair | status   | OK                                |
+---------------+--------+----------+-----------------------------------+
```

Если сервер аварийно завершает работу во время выполнения оператора REPAIR TABLE, важно после его перезапуска немедленно снова запустить REPAIR TABLE, прежде чем выполнять какие-то манипуляции с таблицей. В худшем случае можно получить пустой индексный файл и следующая операция может переписать файл данных. Это маловероятно, но все же возможно.

26.6. Оператор *BACKUP TABLE*

Оператор BACKUP TABLE копирует в резервный каталог минимальный набор файлов таблицы, необходимый для ее последующего восстановления (листинг 26.12). Копируются только файлы данных с расширением MYD и определения структуры таблицы с расширением frm. Индексный файл MYI может быть построен на основе первых двух.

Замечание

Данный оператор может применяться только к таблицам типа MyISAM.

Замечание

Наряду с формой BACKUP TABLE, может использоваться форма BACKUP TABLES, которая является синонимом и производит те же действия, что и оператор BACKUP TABLE.

Листинг 26.12. Использование оператора BACKUP TABLE

```
mysql> BACKUP TABLE products, users, orders TO 'd:\tmp';
+---------------+--------+----------+----------+
| Table         | Op     | Msg_type | Msg_text |
+---------------+--------+----------+----------+
| shop.products | backup | status   | OK       |
| shop.users    | backup | status   | OK       |
| shop.orders   | backup | status   | OK       |
+---------------+--------+----------+----------+
```

Важной деталью является то обстоятельство, что допускается использование только абсолютного пути к резервному каталогу. Резервный каталог, указанный после ключевого слова TO, должен существовать, иначе оператор BACKUP TABLE возвращает ошибку (листинг 26.13).

Листинг 26.13. Резервный каталог `'d:/tmp'` должен существовать

```
mysql> BACKUP TABLE products, users, orders TO 'd:/tmp';
+---------------+--------+----------+-----------------------------------+
| Table         | Op     | Msg_type | Msg_text                          |
+---------------+--------+----------+-----------------------------------+
| shop.products | backup | error    |Failed copying .frm file (errno: 2)|
| shop.products | backup | status   |Operation failed                   |
| shop.users    | backup | error    |Failed copying .frm file (errno: 2)|
| shop.users    | backup | status   |Operation failed                   |
| shop.orders   | backup | error    |Failed copying .frm file (errno: 2)|
| shop.orders   | backup | status   |Operation failed                   |
+---------------+--------+----------+-----------------------------------+
```

При резервном копировании каждая таблица последовательно блокируется по чтению (`READ`). Если в резервной копии необходимо получить согласованный набор данных, для предотвращения изменения таблиц во время резервирования требуется перед вызовом оператора `BACKUP TABLE` вызвать блокировку таблиц по чтению (листинг 26.14). Следует помнить, что в операторе `LOCK TABLES` необходимо перечислить все таблицы, которые появляются в запросах после установки блокировки, в противном случае запрос выполнить не удастся.

Замечание

Подробнее синтаксис оператора `LOCK TABLES` рассматривается в *главе 24*.

Листинг 26.14. Предварительная блокировка таблиц

```
mysql> LOCK TABLES products READ, users READ, orders READ;
mysql> BACKUP TABLE products, users, orders TO 'd:/tmp';
+---------------+--------+----------+----------+
| Table         | Op     | Msg_type | Msg_text |
+---------------+--------+----------+----------+
| shop.products | backup | status   | OK       |
| shop.users    | backup | status   | OK       |
| shop.orders   | backup | status   | OK       |
+---------------+--------+----------+----------+
```

26.7. Оператор *RESTORE TABLE*

Оператор `RESTORE TABLE` (листинг 26.15) восстанавливает таблицу из резервной копии, созданной при помощи оператора `BACKUP TABLE`.

Замечание

Данный оператор может применяться только к таблицам типа MyISAM.

Замечание

Наряду с формой RESTORE TABLE, может использоваться форма RESTORE TABLES, которая является синонимом и производит те же действия, что и оператор RESTORE TABLE.

Листинг 26.15. Использование оператора RESTORE TABLE

```
mysql> RESTORE TABLE products, users, orders FROM 'd:/tmp';
+---------------+---------+----------+----------+
| Table         | Op      | Msg_type | Msg_text |
+---------------+---------+----------+----------+
| shop.products | restore | status   | OK       |
| shop.users    | restore | status   | OK       |
| shop.orders   | restore | status   | OK       |
+---------------+---------+----------+----------+
```

Существующие таблицы не перезаписываются. Если производится попытка восстановить таблицы поверх существующих таблиц, то возвращается сообщение об ошибке 'table exists, will not overwrite on restore' (таблица существует, невозможно осуществить перезапись при восстановлении) (листинг 26.16).

Листинг 26.16. Восстанавливаемых таблиц в базе данных не должно быть

```
mysql> RESTORE TABLE products, users, orders FROM 'd:/tmp';
+----------+---------+----------+-----------------------------------+
| Table    | Op      | Msg_type | Msg_text                          |
+----------+---------+----------+-----------------------------------+
| products | restore | error    | table exists, will not overwrite on
                                  restore                           |
| users    | restore | error    | table exists, will not overwrite on
                                  restore                           |
| orders   | restore | error    | table exists, will not overwrite on
                                  restore                           |
+----------+---------+----------+-----------------------------------+
```

Точно так же, как и в случае оператора BACKUP TABLE, оператор RESTORE TABLE требует указания абсолютного пути к резервному каталогу.

Резервная копия каждой таблицы состоит из его файлов данных MYD и файла структуры таблицы frm. При восстановлении таблицы эти файлы копируются в каталог базы данных, после чего по ним восстанавливается индексный файл с расширением MYI. Восстановление при помощи оператора RESTORE TABLE занимает больше времени, чем резервное копирование при помощи файла BACKUP TABLE, т. к. требуется воссоздание индексов. Чем больше у таблицы индексов, тем больше нужно времени на восстановление.

ГЛАВА 27

Административные команды

Язык запросов SQL, помимо операторов для работы с базами данных, включает в себя административные команды, которые предоставляют средства для управления СУБД, а не только базами данных. Частично рассмотрение таких команд начато в *главе 26*. Данная глава рассматривает остальные команды. Среди команд администрирования следует упомянуть оператор `SET`, который позволяет устанавливать системные и пользовательские переменные, и команду `SHOW`, которая позволяет получать статистическую информацию.

> ***Замечание***
>
> В данной главе административные команды рассматриваются кратко. За более подробным описанием следует обратиться к справочной документации.

Оператор `SET` позволяет устанавливать системные переменные, которые влияют на работу сервера и клиента. В *главе 21* оператор `SET` уже рассматривался при определении пользовательских переменных (листинг 27.1).

Листинг 27.1. Использование оператора `SET`

```
mysql> set @total = 5;
mysql> SELECT @total;
+--------+
| @total |
+--------+
| 5      |
+--------+
```

Оператор `SHOW` может принимать самые разнообразные формы, например, `SHOW CHARACTER SET` выводит список всех кодировок, которые поддерживает текущая версия MySQL (см. листинг 11.17). Оператор `SHOW COLUMNS` выводит список столбцов заданной таблицы, `SHOW CREATE DATABASE` — выводит оператор `CREATE DATABASE` для базы данных, `SHOW DATABASES` — выводит список баз данных, находящихся под управлением сервера MySQL.

ЧАСТЬ III

Глава 28. Хранимые процедуры

Глава 29. Триггеры

Глава 30. Представления

Глава 31. Информационная схема

ГЛАВА 28

Хранимые процедуры

Часто при выполнении рутинных операций требуется производить последовательность одинаковых запросов. Хранимые процедуры позволяют объединить последовательность таких запросов и сохранить их на сервере. После этого клиентам не придется посылать серверу последовательность запросов, достаточно послать одни запрос на выполнение хранимой процедуры.

Замечание

Хранимые процедуры появились в СУБД MySQL, начиная с версии 5.0.0.

Хранимые процедуры обладают приведенными далее преимуществами.

- Повторное использование кода — после того, как хранимая процедура создана, ее можно вызывать из любых приложений и SQL-запросов, больше не требуется каждый раз программировать одни и те же действия, благодаря чему уменьшается риск проникновения в приложения программных ошибок и уменьшается время разработки программы.
- Сокращение сетевого трафика — при использовании хранимых процедур вместо того, чтобы отсылать по сети каждый запрос и получать на каждый из запросов ответ, гораздо экономнее послать серверу запрос на выполнение хранимой процедуры и сразу получить ответ.
- Безопасность — хранимые процедуры используются для всех стандартных банковских операций. Процедура гарантирует, что последовательность действий будет согласованной и не приведет к нарушению целостности данных из-за того, что один оператор не будет выполнен. Кроме того, для выполнения хранимой процедуры пользователь должен иметь соответствующую привилегию. При этом прав доступа к таблицам иметь вовсе не обязательно. Таким образом, администратор базы данных получает более широкие возможности в плане защиты данных и управления доступом пользователей к объектам базы данных.
- Простота доступа — основным императивом разработки программного обеспечения провозглашено уменьшение сложности кода. Хранимые процедуры позволяют инкапсулировать сложный код и оформить его в виде простого вызова с осмыс-

ленным именем. При создании нового каталога гораздо проще оперировать процедурой CREATE_NEW_CATALOG(), чем несколькими операторами непонятного назначения. При регистрации сделки, требующей поиска в таблицах catalogs, products, users, и редактировании таблиц orders и users гораздо проще оформить последовательность SQL-операторов для оформления сделки в хранимую процедуру ORDERING(). Даже если последовательность операторов хорошо знакома программисту, применение хранимых процедур позволяет перейти на более высокий уровень абстракции и оперировать не таблицами, а сущностями реального мира, такими как сделка, каталог продукции и др.

- Выполнение деловой логики — хранимые процедуры могут выполнять проверку условий выполнения заказа, например, подсчитывая число товарных позиций на складе, и отклонять заказ в случае, если его недостаточно. Это позволяет перенести код сохранения целостности базы данных из прикладной программы на сервер базы данных. Так как прикладная программа может выполняться на сотне машин сети, а сервер, как правило, один, устойчивость системы при перемещении логики на сервер резко возрастает. Кроме того, в этом случае легче сменить язык разработки внешнего приложения, т. к. большая часть логики оформлена в виде хранимых процедур и не зависит от языка разработки внешнего приложения.

28.1. Хранимые процедуры и привилегии

Начиная с версии 5.0.3, СУБД MySQL требует от пользователей наличия следующих привилегий при работе с хранимыми процедурами:

- для создания хранимых процедур необходимо наличие привилегии CREATE ROUTINE;
- для редактирования и удаления хранимой процедуры необходимо наличие привилегии ALTER ROUTINE, пользователь, создающий хранимую процедуру, автоматически наделяется этой привилегией;
- для вызова хранимой процедуры необходимо наличие привилегии EXECUTE, эта привилегия также автоматически передается пользователю, создавшему хранимую процедуру.

Замечание

Более подробно с системой привилегий СУБД MySQL можно ознакомиться в *главе 25*.

28.2. Создание хранимой процедуры

Для создания хранимой процедуры существует два оператора: CREATE PROCEDURE и CREATE FUNCTION, которые имеют следующий синтаксис:

```
CREATE PROCEDURE sp_name ([parameter[,...]])
    [characteristic ...] routine_body
```

```
CREATE FUNCTION sp_name ([parameter[,...]])
    RETURNS type
    [characteristic ...] routine_body
```

Здесь sp_name имя хранимой процедуры, в скобках, следующих за именем, передается необязательный список параметров, перечисленных через запятую. Каждый параметр parameter позволяет передать в процедуру или из нее входные данные или результат работы функции и имеет следующий синтаксис:

```
[ IN | OUT | INOUT ] param_name type
```

Параметр param_name предваряет одно из ключевых слов IN, OUT, INOUT, которые позволяют задать направление передачи данных:

- IN — данные передаются строго внутрь хранимой процедуры, но если параметру с данным модификатором внутри функции присваивается новое значение, по выходу из нее оно не сохраняется и параметр принимает значение, которое он имел до вызова процедуры;
- OUT — данные передаются строго из хранимой процедуры, даже если параметр имеет какое-то начальное значение, внутри хранимой процедуры это значение не принимается во внимание. С другой стороны, если параметр изменяется внутри процедуры, после вызова процедуры параметр имеет значение, присвоенное ему внутри процедуры;
- INOUT — значение этого параметра как принимается во внимание внутри процедуры, так и сохраняет свое значение по выходу из нее.

Замечание

Список аргументов, заключенных в круглые скобки, должен присутствовать всегда. Если аргументы отсутствуют, следует использовать пустой список аргументов ().

После имени параметра param_name указывается его тип, который может быть одним из типов, рассмотренных в *главе 4*. По умолчанию, если ни один из модификаторов не указан, считается, что параметр объявлен с ключевым словом IN.

Оператор CREATE FUNCTION позволяет задать пользовательскую функцию, т. е. такой вид хранимой процедуры, который возвращает единственное значение. Тип этого значения позволяет задать оператор RETURN.

Характеристика characteristic может принимать одно из следующих значений или их комбинацию:

```
    LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | SQL SECURITY {DEFINER | INVOKER}
  | COMMENT 'string'
```

Подробнее эти ключевые слова рассматриваются в *разделе 28.3*.

28.2.1. Тело процедуры

Тело процедуры routine_body состоит из составного оператора BEGIN ... END, внутри которого могут располагаться другие операторы, в том числе и другие составные операторы BEGIN ... END. Оператор имеет следующий синтаксис:

```
[label:] BEGIN
  statements
END [label]
```

Если оператор начинается с необязательной метки label, в качестве которой может выступать любое уникальное имя, то он может заканчиваться выражением END label.

Оператор BEGIN ... END может выглядеть так, как это представлено в листинге 28.1.

Листинг 28.1. Использование составного оператора BEGIN ... END

```
BEGIN
  UPDATE tbl1 SET col1 = '1234.56';
  UPDATE tbl2 SET col2 = '1234.56';
END
```

В качестве одного из операторов внутри составного оператора BEGIN ... END может выступать другой составной оператор (листинг 28.2).

Листинг 28.2. Вложенный составной оператор BEGIN ... END

```
BEGIN
  UPDATE tbl1 SET col1 = '1234.56';
  inner: BEGIN
    UPDATE tbl2 SET col2 = '1234.56';
    UPDATE tbl3 SET col3 = '1234.56';
  END inner
END
```

Как видно из листинга 28.2, во внутреннем составном операторе BEGIN ... END перед оператором BEGIN используется метка inner, которая обязательно упоминается и в операторе END. Следует отметить, что если хранимая процедура содержит только один запрос, то можно не использовать составной оператор BEGIN ... END.

Замечание

После оператора END можно как помещать, так и не помещать точку с запятой. Здесь и далее точка с запятой после END помещаться не будет.

Основная трудность, которая возникает при работе с хранимыми процедурами, заключается в том, что обязательный символ точки с запятой ";" в конце каждого запроса воспринимается консольным клиентом как сигнал к отправке запроса на сер-

вер. Для того чтобы избежать этого, при работе с хранимыми процедурами следует переопределить разделитель запросов при помощи параметра `--delimiter=//` консольного клиента `mysql` (листинг 28.3). В этом случае для обозначения окончания ввода вместо точки с запятой необходимо будет использовать последовательность "//".

Листинг 28.3. Переопределение символа окончания запроса

```
mysql -u root --delimiter=//
```

Кроме того, можно менять разделитель в любой момент в консольном клиенте `mysql`. Для этого необходимо воспользоваться командой `DELIMITER`, после которой указать разделитель (листинг 28.4).

Листинг 28.4. Смена разделителя запросов при помощи команды `DELIMITER`

```
mysql> DELIMITER //
mysql> SELECT VERSION()//
+---------------+
| VERSION()     |
+---------------+
| 5.0.6-beta-nt |
+---------------+
mysql> DELIMITER ;
mysql> SELECT VERSION();
+---------------+
| VERSION()     |
+---------------+
| 5.0.6-beta-nt |
+---------------+
```

Имя хранимой процедуры не может превышать 64 символа и не зависит от регистра, т. е. имена `numcatalogs()`, `Numcatalogs()` и `NUMCATALOGS()` являются эквивалентными. Пример создания простейшей хранимой процедуры демонстрируется в листинге 28.5.

Листинг 28.5. Создание хранимой процедуры `my_version()`

```
mysql> CREATE PROCEDURE my_version ()
    -> BEGIN
    ->   SELECT VERSION();
    -> END
    -> //
```

Если тело процедуры содержит единственный запрос, можно не использовать составной оператор `BEGIN ... END`. Определения хранимой процедуры `my_version()` в листингах 28.5 и 28.6 эквивалентны.

Листинг 28.6. Альтернативное определение хранимой процедуры `my_version()`

```
mysql> CREATE PROCEDURE my_version ()
    ->   SELECT VERSION();
    -> //
```

Функция `my_version()` ничего не делает, кроме того, что выводит версию сервера MySQL. Каким образом можно воспользоваться результатами запроса? Для того чтобы вызвать хранимую процедуру, необходимо применить оператор `CALL`, после которого помещается имя процедуры и ее параметры в круглых скобках (листинг 28.7).

Листинг 28.7. Вызов хранимой процедуры при помощи ключевого слова `CALL`

```
mysql> CALL my_version ();
+---------------+
| VERSION()     |
+---------------+
| 5.0.6-beta-nt |
+---------------+
```

При вызове хранимой процедуры, в отличие от встроенных функций, между именем функции и круглыми скобками допускается пробел. В листинге 28.8 оба запроса являются эквивалентными.

Листинг 28.8. Пробел после имени функции допустим

```
mysql> CALL my_version();
mysql> CALL my_version ();
```

При именовании функций следует избегать названий, совпадающих с именами внутренних функций MySQL (листинг 28.9).

Листинг 28.9. Хранимая процедура `pi ()`

```
mysql> CREATE PROCEDURE pi ()
    -> BEGIN
    ->   SELECT VERSION();
    -> END
    -> //
```

В листинге 28.9 объявляется хранимая процедура `pi ()`, которая выводит текущую версию сервера MySQL. Следует обратить внимание, что между именем процедуры и круглыми скобками помещен пробел — в противном случае СУБД MySQL интерпретирует последовательность `pi()` как вызов встроенной функции, возвращающей число π. При вызове хранимой процедуры использование пробела в этом случае также является обязательным (листинг 28.10). Перегрузка функций в MySQL не предусмотрена.

Листинг 28.10. Вызов хранимой процедуры `pi ()`

```
mysql> CALL pi();
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use
near 'pi()' at line 1
mysql> CALL pi ();
+---------------+
| VERSION()     |
+---------------+
| 5.0.6-beta-nt |
+---------------+
```

Во избежание подобных накладок рекомендуется избегать использования названий хранимых процедур, совпадающих с именами встроенных функций MySQL. Если все же это необходимо, то и в определении функции, и при ее вызове следует использовать пробел между именем хранимой процедуры и круглыми скобками.

Какие операторы допустимы в теле хранимых процедур? Любые, включая `INSERT`, `UPDATE`, `DELETE`, `SELECT`, `DROP`, `REPLACE` и др. (листинг 28.11).

Листинг 28.11. Примеры определения хранимых процедур

```
mysql> CREATE PROCEDURE p () DELETE FROM t; //
mysql> CREATE PROCEDURE p () SET @x = 5; //
mysql> CREATE PROCEDURE p () DROP TABLE t; //
mysql> CREATE PROCEDURE p () SELECT 'A' ; //
```

В теле хранимой процедуры можно использовать многострочный комментарий в стиле языка C, который начинается с последовательности `"/*"` и заканчивается последовательностью `"*/"` (листинг 28.12).

Листинг 28.12. Использование комментария в теле функции

```
mysql> CREATE PROCEDURE my_version ()
    -> BEGIN
    ->  /* Многострочный комментарий внутри
    ->     функции my_version()
    ->  */
    ->  SELECT VERSION(); /* Вызов единственного оператора */
    -> END
    -> //
```

28.2.2. Параметры процедуры

В предыдущем разделе хранимые процедуры не содержали параметров. Как упоминалось ранее, каждый параметр может быть снабжен одним из модификаторов `IN`, `OUT`

или INOUT. В листинге 28.13 приводится пример функции, которая присваивает пользовательской переменной @x новое значение.

Листинг 28.13. Использование ключевого слова IN

```
mysql> USE test//
Database changed
mysql> CREATE PROCEDURE set_x (IN value INT)
    -> BEGIN
    ->  SET @x = value;
    -> END
    -> //
mysql> CALL set_x(123456)//
mysql> SELECT @x//
+--------+
| @x     |
+--------+
| 123456 |
+--------+
```

Как видно из листинга 28.13, через параметр value функции передается числовое значение 123456, которое она присваивает пользовательской переменной @x. Модификатор IN сообщает СУБД MySQL, что при помощи параметра value пользователи передают данные внутрь функции.

Замечание

В отличие от пользовательской переменной @x, которая является глобальной и доступна как внутри хранимой процедуры set_x(), так и вне ее, параметры функции являются локальными и доступны для использования только внутри функции.

Использование ключевого слова IN не является обязательным — если ни один из модификаторов не указан, СУБД MySQL считает, что параметр объявлен с модификатором IN.

Листинг 28.14. Использование ключевого слова IN

```
mysql> USE test//
Database changed
mysql> CREATE PROCEDURE set_y (value INT)
    -> BEGIN
    ->  SET value = 7;
    ->  SET @x = value;
    -> END
    -> //
mysql> SET @val = 123456//
mysql> CALL set_y(@val)//
mysql> SELECT @x, @val//
```

```
+------+--------+
| @x   | @val   |
+------+--------+
| 7    | 123456 |
+------+--------+
```

Замечание

Следует отметить, что имена параметров при объявлении хранимой процедуры и при вызове не обязательно должны совпадать. Внутри хранимой процедуры все локальные переменные используются без символа "@", в то время как для глобальных пользовательских переменных символ "@" перед именем обязателен.

Замечание

Подробнее пользовательские переменные обсуждаются в *главе 21*.

Хранимая процедура `set_y()` принимает единственный `IN`-параметр `value`, при помощи оператора `SET` значение параметра изменяется внутри функции. Однако, как видно из листинга 28.14, после выполнения хранимой процедуры значение пользовательской переменной `@val`, переданной функции в качестве параметра, не изменяется. Если требуется, чтобы значение переменной подвергалось изменению, необходимо объявить параметр процедуры с модификатором `OUT` (листинг 28.15).

Листинг 28.15. Использование ключевого слова `OUT`

```
mysql> USE test//
Database changed
mysql> CREATE PROCEDURE set_y (OUT value INT)
    -> BEGIN
    ->  SET @x = value;
    ->  SET value = 7;
    -> END
    -> //
mysql> SET @val = 123456//
mysql> CALL set_y(@val)//
mysql> SELECT @x, @val//
+------+------+
| @x   | @val |
+------+------+
| NULL | 7    |
+------+------+
```

Как видно из листинга 28.15, при использовании модификатора `OUT` любые изменения параметра внутри процедуры отражаются на параметре. Передача в качестве значения пользовательской переменной позволяет использовать результат процедуры для дальнейших вычислений. Однако передать значение внутрь функции при помощи `OUT`-параметра уже не получится.

Замечание

Если локальная или пользовательская переменные не инициируются при помощи оператора SET или ключевого слова DEFAULT, они получают значение NULL.

Для того чтобы через параметр можно было и передать значение внутрь процедуры, и получить значение, которое попадает в параметр в результате вычислений внутри процедуры, его следует объявить с модификатором INOUT (листинг 28.16).

Листинг 28.16. Использование ключевого слова INOUT

```
mysql> USE test//
Database changed
mysql> CREATE PROCEDURE set_y (INOUT value INT)
    -> BEGIN
    ->  SET @x = value;
    ->  SET value = 7;
    -> END
    -> //
mysql> SET @val = 123456//
mysql> CALL set_y(@val)//
mysql> SELECT @x, @val//
+--------+------+
| @x     | @val |
+--------+------+
| 123456 | 7    |
+--------+------+
```

Теперь через параметр value можно как передавать значения внутрь процедуры, так и извлекать значения, которые получает параметр внутри процедуры. Тем не менее рекомендуется использовать только IN- и OUT-параметры, не прибегая к комбинированным INOUT-параметрам, т. к. это приводит к нечитабельному и непоследовательному коду.

28.2.3. Работа с таблицами базы данных

Все процедуры, рассмотренные выше, не использовали в своей работе таблицы, далее рассмотрим несколько процедур, осуществляющих запросы к таблицам базы данных. Создадим функцию numcatalogs(), которая подсчитывает число записей в таблице catalogs учебной базы данных shop (листинг 28.17).

Листинг 28.17. Создание хранимой процедуры numcatalogs()

```
mysql> USE shop
Database changed
mysql> CREATE PROCEDURE numcatalogs (OUT total INT)
```

```
    -> BEGIN
    ->   SELECT COUNT(*) INTO total FROM catalogs;
    -> END
    -> //
```

Хранимая процедура `numcatalogs()` имеет один целочисленный (`INT`) параметр `total`, в который сохраняется число записей в таблице `catalogs`. Осуществляется это при помощи оператора `SELECT ... INTO ... FROM`, который позволяет сохранять результаты непосредственно в выходном параметре `total` функции `numcatalogs`. Данный оператор позволяет оперировать сразу несколькими столбцами (листинг 28.18).

Листинг 28.18. Использование оператора `SELECT ... INTO ... FROM`

```
SELECT id, data INTO x, y FROM test LIMIT 1;
```

Таким образом, параметр `total` после выполнения хранимой процедуры содержит число записей в таблице `catalogs`.

Листинг 28.19. Вызов хранимой процедуры `numcatalogs()`

```
mysql> USE shop;
mysql> CALL numcatalogs(@a);
mysql> SELECT @a;
+------+
| @a   |
+------+
| 5    |
+------+
```

Как видно из листинга 28.19, в качестве параметра функции `numcatalogs()` передается пользовательская переменная `@a`.

СУБД поддерживает контекст вызова хранимой процедуры для базы данных по умолчанию. Это означает, что если хранимая процедура, созданная в базе данных `shop`, будет вызвана в тот момент, когда текущей базой данных является база `test` — СУБД MySQL вернет ошибку (листинг 28.20).

Листинг 28.20. Вызов хранимой процедуры `numcatalogs()` в контексте базы `test`

```
mysql> USE test;
mysql> CALL numcatalogs(@a);
ERROR 1305: PROCEDURE test.numcatalogs does not exist
```

Хранимая процедура наследует базу данных по умолчанию от вызывающего оператора, поэтому при обращении к таблицам других баз данных необходимо использовать расширенные имена.

Замечание

Использование оператора `USE` в хранимых процедурах запрещено.

Создадим хранимую процедуру `catalogname()`, которая будет возвращать по первичному ключу `id_catalog` название каталога `name` (листинг 28.21). Для этого потребуется определить параметр `id_catalog` с атрибутом `IN`, а `name` с атрибутом `OUT`.

Листинг 28.21. Создание хранимой процедуры `catalogname()`

```
mysql> CREATE PROCEDURE catalogname (IN id INT, OUT catalog TINYTEXT)
    -> BEGIN
    ->   SELECT name INTO catalog FROM catalogs
    ->   WHERE id_catalog = id;
    -> END
    -> //
mysql> SET @id := 5//
mysql> CALL catalogname(@id, @name)//
mysql> SELECT @id, @name//
+------+--------------------+
| @id  | @name              |
+------+--------------------+
| 5    | Оперативная память |
+------+--------------------+
mysql> CALL catalogname(1, @name)//
mysql> SELECT @name//
+------------+
| @name      |
+------------+
| Процессоры |
+------------+
```

Примером функции, использующей параметр типа `INOUT`, может послужить функция `count_by_id()`, которая, принимая в качестве параметра первичный ключ каталога из таблицы `catalogs`, возвращает число товарных позиций в данном каталоге.

Листинг 28.22. Создание хранимой процедуры `count_by_id()`

```
mysql> CREATE PROCEDURE count_by_id (INOUT id INT)
    -> BEGIN
    ->   SELECT COUNT(*) INTO id FROM products
    ->   WHERE id_catalog = id;
    -> END
    -> //
mysql> SET @id = 3//
mysql> CALL count_by_id(@id)//
mysql> SELECT @id//
```

```
+------+
| @id  |
+------+
| 4    |
+------+
```

Однако применения INOUT-параметров следует всячески избегать, т. к. при их использовании возникает соблазн нарушить правила хорошего стиля программирования, как в листинге 28.22, где пользовательская переменная @id сначала содержала первичный ключ каталога, а после вызова содержит уже число товарных позиций в данном каталоге. Использование переменных в этом духе непременно приводит к возникновению ошибок. Лучше вообще отказаться от использования INOUT-переменных даже в том случае, когда функция принимает и возвращает однотипные данные (например, первичный ключ), лучше создать два параметра: один для входного значения, другой — для выходного.

Оператор SELECT ... INTO ... FROM не возвращает результат запроса клиенту непосредственно, в отличие от обычного оператора SELECT ... FROM, который отправляет результат клиенту, даже если вызывается внутри хранимой процедуры (листинг 28.23).

Листинг 28.23. Использование обычного оператора SELECT

```
mysql> CREATE PROCEDURE numcatalogsview ()
    -> BEGIN
    ->   SELECT COUNT(*) FROM catalogs;
    -> END
    -> //
mysql> CALL numcatalogsview ()//
+----------+
| COUNT(*) |
+----------+
|        5 |
+----------+
```

Как видно из листинга 28.23, функция numcatalogsview() позволяет добиться тех же результатов, что и numcatalogs() (см. листинг 28.17) без использования параметров и пользовательских переменных. Однако использование параметров оправдано, т. к. позволяет создавать более читабельный код и использовать результат функции при вызове ее из другой функции. Создадим функцию catalog_by_product(), которая по имени товарной позиции возвращает имя каталога, к которому относится данная товарная позиция. В ходе создания новой функции будем опираться на ранее созданную функцию catalogname(), которая возвращает имя каталога по его первичному ключу id_catalog (листинг 28.21). Код новой функции представлен в листинге 28.24.

Листинг 28.24. Создание хранимой процедуры `catalog_by_product()`

```
mysql> CREATE PROCEDURE catalog_by_product (IN product TINYTEXT,
    ->                                      OUT catalog TINYTEXT)
    -> BEGIN
    ->   DECLARE id INT;
    ->   SELECT id_catalog INTO id FROM products
    ->   WHERE name = product LIMIT 1;
    ->   CALL catalogname(id, catalog);
    -> END
    -> //
mysql> CALL catalog_by_product('Celeron 1.8', @catalogname)//
mysql> SELECT @catalogname//
+--------------+
| @catalogname |
+--------------+
| Процессоры   |
+--------------+
```

Как видно из листинга 28.24, функция `catalog_by_product()` имеет два параметра:

- `product` — входной параметр, через который передается название товарной позиции;
- `catalog` — выходной параметр, через который можно получить результат работы функции — название каталога, в который входит товарная позиция.

По названию товарной позиции `product` при помощи `SELECT`-запроса определяется первичный ключ каталога. Полученное значение помещается во временную переменную `id`, которая передается в качестве первого аргумента функции `catalogname()`, возвращающей название каталога и помещающей результат работы в переменную `catalog`, являющуюся выходным параметром функции `catalog_by_product()`.

В функции `catalog_by_product()` потребовалась временная переменная `id`. Для использования любой переменной в функции требуется ее объявление при помощи оператора `DECLARE`, который имеет следующий синтаксис:

```
DECLARE var_name[,...] type [DEFAULT value]
```

Один оператор `DECLARE` позволяет объявить сразу несколько переменных одного типа, причем необязательное слово `DEFAULT` позволяет назначить инициирующее значение.

Листинг 28.25. Использование оператора `DECLARE`

```
mysql> CREATE PROCEDURE declare_var ()
    -> BEGIN
    ->   DECLARE id, num INT(11) DEFAULT 0;
    ->   DECLARE name, hello, temp TINYTEXT;
    -> END
    -> //
```

В листинге 28.25 объявляются две переменные типа INT(11) — id и num, инициированные значением 0, и три текстовые переменные name, hello и temp, объявленные без дополнительной инициализации. Инициировать локальные переменные можно и позже при помощи оператора SET (см. листинг 28.14).

Оператор DECLARE может появляться только внутри блока BEGIN ... END, область видимости объявленной переменной также ограничена этим блоком. Это означает, что в разных блоках BEGIN ... END могут быть объявлены переменные с одинаковым именем, и действовать они будут только в рамках данного блока, не пересекаясь с переменными других блоков.

Листинг 28.26. Область видимости локальных переменных

```
mysql> CREATE PROCEDURE declare_var ()
    -> outer: BEGIN
    ->   DECLARE var TINYTEXT DEFAULT 'внешняя переменная';
    ->   inner: BEGIN
    ->     DECLARE var TINYTEXT DEFAULT 'внутренняя переменная';
    ->     SELECT var;
    ->   END inner;
    ->   SELECT var;
    -> END outer
    -> //
mysql> CALL declare_var()//
+-----------------------+
| var                   |
+-----------------------+
| внутренняя переменная |
+-----------------------+
+--------------------+
| var                |
+--------------------+
| внешняя переменная |
+--------------------+
```

В листинге 28.26 переменная var объявляется сначала со значением 'внешняя переменная' во внешнем блоке BEGIN ... END, после чего во вложенном блоке объявляется вторая переменная var со значением 'внутренняя переменная', которая экранирует первую переменную.

Однако переменная, объявленная во внешнем блоке BEGIN ... END, будет доступна во вложенном блоке, если не будет объявлено экранирующей ее переменной (листинг 28.27).

Листинг 28.27. Область видимости локальных переменных

```
mysql> CREATE PROCEDURE one_declare_var ()
    -> BEGIN
    ->   DECLARE var TINYTEXT DEFAULT 'внешняя переменная';
```

```
    ->   BEGIN
    ->     SELECT var;
    ->   END;
    ->   SELECT var;
    -> END
    -> //
mysql> CALL one_declare_var()//
+--------------------+
| var                |
+--------------------+
| внешняя переменная |
+--------------------+
+--------------------+
| var                |
+--------------------+
| внешняя переменная |
+--------------------+
```

Обратное не верно, переменная, объявленная во вложенном блоке, недоступна во внешнем (листинг 28.28).

Листинг 28.28. Область видимости локальных переменных

```
mysql> CREATE PROCEDURE inner_declare_var ()
    -> BEGIN
    ->   BEGIN
    ->     DECLARE var TINYTEXT DEFAULT 'внутренняя переменная';
    ->     SELECT var;
    ->   END;
    ->   SELECT var;
    -> END
    -> //
mysql> CALL inner_declare_var()//
+------------------------+
| var                    |
+------------------------+
| внутренняя переменная |
+------------------------+
ERROR 1054 (42S22): Unknown column 'var' in 'field list'
```

Как видно из листинга 28.28, вызов оператора `SELECT var` во внешнем блоке `BEGIN ... END` приводит к ошибке, т. к. время жизни переменной `var` ограничено ее блоком и после выхода из этого блока ее существование прекращается.

В заключение раздела следует отметить, что не допускается и повторное объявление переменной в рамках одного блока `BEGIN ... END` (листинг 28.29). Это приводит к возникновению ошибки 1331: "Повторное объявление переменной".

Листинг 28.29. Повторное объявление переменной var

```
mysql> CREATE PROCEDURE dbl_declare_var ()
    -> BEGIN
    ->   DECLARE var TINYTEXT DEFAULT 'внешняя переменная';
    ->   DECLARE var TINYTEXT DEFAULT 'внутренняя переменная';
    ->   SELECT var;
    -> END
    -> //
ERROR 1331 (42000): Duplicate variable: var
```

28.2.4. Хранимые функции

Помимо формы CREATE PROCEDURE, создающей хранимую процедуру, допускается использование CREATE FUNCTION, которая создает хранимую функцию. Функция в отличие от процедуры может вызываться непосредственно, без использования оператора CALL и возвращать одно значение, которое подставляется в место вызова функции, как в случае встроенных функций MySQL.

Создадим простейшую хранимую функцию say_hello(), которая будет принимать единственный входной параметр с именем name и возвращать фразу "Hello, name!", где вместо подстроки name будет подставлено значение параметра name (листинг 28.30).

Листинг 28.30. Использование оператора CREATE FUNCTION

```
mysql> CREATE FUNCTION say_hello (name CHAR(20)) RETURNS CHAR(50)
    -> BEGIN
    ->  RETURN CONCAT('Hello, ',name,'!');
    -> END
    -> //
mysql> SELECT say_hello('world'), say_hello('softtime')//
+--------------------+-----------------------+
| say_hello('world') | say_hello('softtime') |
+--------------------+-----------------------+
| Hello, world!      | Hello, softtime!      |
+--------------------+-----------------------+
```

После объявления параметров функции следует оператор RETURNS, который задает тип возвращаемого функцией значения. Вернуть значение из функции можно при помощи оператора RETURN (листинг 28.31), который может быть вызван в любой точке функции. Вызов оператора RETURN означает, что функция должна немедленно завершить выполнение и вернуть значение, переданное в качестве аргумента оператора RETURN.

Замечание

При объявлении параметров функции использование ключевых слов IN, INOUT и OUT недопустимо. Все параметры, передаваемые функции, являются входными.

Замечание

Функция обязательно должна содержать оператор RETURNS, устанавливающий тип возвращаемого значения, и хотя бы один оператор RETURN в теле функции, который возвращает это значение.

Листинг 28.31. Использование оператора RETURN

```
mysql> CREATE FUNCTION func_catalog (id INT)
    -> RETURNS TINYTEXT
    -> BEGIN
    ->   DECLARE catalog TINYTEXT;
    ->   SELECT name INTO catalog FROM catalogs
    ->   WHERE id_catalog = id LIMIT 1;
    ->   RETURN catalog;
    ->   SELECT name INTO catalog FROM catalogs
    ->   WHERE id_catalog = id + 1 LIMIT 1;
    ->   RETURN catalog;
    -> END
    -> //
mysql> SELECT func_catalog(1)//
+-----------------+
| func_catalog(1) |
+-----------------+
| Процессоры      |
+-----------------+
```

Хранимая функция func_catalog() принимает единственный параметр id — первичный ключ таблицы catalogs. Приняв в качестве параметра id значение 1, функция возвращает результат ("Процессоры"), достигнув первого оператора RETURN. При этом второй оператор SELECT и RETURN не достигаются никогда (иначе возвращалось бы значение "Оперативная память"). Это не значит, что двух операторов RETURN в теле функции не должно встречаться. Ниже будут рассмотрены условные конструкции, позволяющие в зависимости от условий выбирать вариант, который должен возвращаться — в такой ситуации использование множественного выхода из функции при помощи нескольких операторов RETURN не избежать.

Если последовательность операторов, которые решено оформить в виде хранимой процедуры, возвращает одно-единственное значение, гораздо удобнее оформить их в виде хранимой функции, т. к. работать с ними в выражениях гораздо удобнее. Создадим две функции: count_product_in_catalog() и count_product(), которые будут возвращать общее число товарных позиций в каталоге и общее число товарных позиций в учебном электронном магазине shop.

Листинг 28.32. Подсчет процентного вклада товарных позиций каталога

```
mysql> CREATE FUNCTION count_product_in_catalog (id INT)
    -> RETURNS INT
```

```
    -> BEGIN
    ->   DECLARE total INT;
    ->   SELECT SUM(count) INTO total FROM products
    ->   WHERE id_catalog = id LIMIT 1;
    ->   RETURN total;
    -> END
    -> //
mysql> CREATE FUNCTION count_product ()
    -> RETURNS INT
    -> BEGIN
    ->   DECLARE total INT;
    ->   SELECT SUM(count) INTO total FROM products;
    ->   RETURN total;
    -> END
    -> //
mysql> SELECT count_product_in_catalog(1) AS total,
    -> count_product_in_catalog(1)/count_product()*100 AS persent//
+-------+---------+
| total | persent |
+-------+---------+
|    56 | 26.5403 |
+-------+---------+
```

Как видно из листинга 28.32, функции удобно использовать в выражениях, например, для подсчета процентного вклада товарных позиций каталога в общее число товаров в электронном магазине `shop`.

28.3. Группа характеристик хранимых процедур

Синтаксис хранимых процедур допускает использование следующих характеристик `characteristic` в определении `CREATE PROCEDURE` и `CREATE FUNCTION` *(см. раздел 28.2)*:

```
    LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | SQL SECURITY {DEFINER | INVOKER}
  | COMMENT 'string'
```

Данные ключевые слова описывают характеристики хранимых процедур и функций. Они размещаются после списка параметров, но до начала тела хранимой процедуры. Характеристика `LANGUAGE SQL` пока не имеет особого смысла и сообщает, что хранимая процедура написана на языке SQL (листинг 28.33).

Листинг 28.33. Использование ключевого слова `LANGUAGE SQL`

```
mysql> CREATE PROCEDURE characteristics ()
    -> LANGUAGE SQL
```

```
    -> BEGIN
    ->   SELECT RAND();
    -> END
    -> //
```

В следующих версиях MySQL планируется предоставить возможность создания хранимых процедур с использованием языка, отличного от SQL. Скорее всего, одним из первых поддерживаемых языков станет PHP, т. к. базовый механизм PHP невелик по размерам, безопасен в отношении потоков и легко встраивается. Рекомендуется использовать ключевое слово `LANGUAGE SQL` уже сейчас для обеспечения совместимости с будущими версиями MySQL и другими СУБД.

Ключевое слово `DETERMINISTIC` позволяет сообщить оптимизатору, что процедура всегда возвращает один и тот же результат для одних и тех же входных параметров, в противном случае следует использовать ключевое слово `NOT DETERMINISTIC` (листинг 28.34).

Листинг 28.34. Использование ключевого слова `DETERMINISTIC`

```
mysql> CREATE PROCEDURE characteristics ()
    -> LANGUAGE SQL
    -> NOT DETERMINISTIC
    -> BEGIN
    ->   SELECT RAND();
    -> END
    -> //
```

Замечание

В настоящий момент ключевое слово `DETERMINISTIC` распознается, но еще не используется оптимизатором MySQL.

Ключевое слово `SQL SECURITY` может быть записано в двух формах: `SQL SECURITY DEFINER` и `SQL SECURITY INVOKER`. Если используется форма `SQL SECURITY DEFINER`, то хранимая процедура вызывается с привилегиями пользователя, создавшего ее, при использовании `SQL SECURITY INVOKER` процедура вызывается с привилегиями пользователя, вызывающего процедуру оператором `CALL` (листинг 28.35).

Листинг 28.35. Использование ключевого слова `SQL SECURITY`

```
mysql> CREATE PROCEDURE characteristics ()
    -> LANGUAGE SQL
    -> NOT DETERMINISTIC
    -> SQL SECURITY INVOKER
    -> BEGIN
    ->   SELECT RAND();
    -> END
    -> //
```

Замечание

Если ключевое слово SQL SECURITY не указано, по умолчанию устанавливается режим SQL SECURITY DEFINER — хранимая процедура выполняется с привилегиями создавшего ее пользователя.

Для выполнения хранимой процедуры и ее создатель, и ее пользователь должны иметь доступ к базе данных, в которой сохранена процедура, кроме того, они оба должны иметь привилегию EXECUTE, независимо от того, в какой форме использовано ключевое слово SQL SECURITY.

Ключевое слово COMMENT (листинг 28.36) позволяет снабжать хранимую процедуру кратким описанием, которое отображается операторами SHOW CREATE PROCEDURE и SHOW CREATE FUNCTION.

Замечание

Ключевое слово COMMENT является расширением MySQL и может не поддерживаться другими СУБД.

Листинг 28.36. Использование ключевого слова COMMENT

```
mysql> CREATE PROCEDURE characteristics ()
    -> LANGUAGE SQL
    -> NOT DETERMINISTIC
    -> SQL SECURITY INVOKER
    -> COMMENT 'Функция возвращает случайное значение'
    -> BEGIN
    ->   SELECT RAND();
    -> END
    -> //
mysql> SHOW CREATE PROCEDURE characteristics\G//
*************************** 1. row ***************************
       Procedure: characteristics
        sql_mode:
Create Procedure: CREATE PROCEDURE 'shop'.'characteristics'()
    SQL SECURITY INVOKER
    COMMENT 'Функция возвращает случайное значение'
BEGIN
  SELECT RAND();
END
```

28.4. Операторы управления потоком данных

Хранимые процедуры это не просто удобные контейнеры для группы запросов, они позволяют реализовать достаточно сложную логику, используя операторы ветвления и циклы.

Замечание

Вне хранимых процедур описываемые в данном разделе операторы применять нельзя.

28.4.1. Оператор *IF...THEN...ELSE*

Оператор `IF` позволяет реализовать ветвление программы по условию и имеет следующий синтаксис:

```
IF search_condition THEN statement_list
    [ELSEIF search_condition THEN statement_list] ...
    [ELSE statement_list]
END IF
```

Логическое выражение `search_condition` может принимать два значения: 0 (ложь) и значение, отличное от нуля (истина). Если логическое выражение истинно, то оператор `statement_list` после ключевого слова выполняется, иначе выполняется список операторов в блоке `ELSE` (если блок `ELSE` имеется). В качестве `statement_list` может выступать составной оператор `BEGIN ... END`.

Замечание

Следует отметить, что в СУБД MySQL, кроме оператора `IF`, описанного в этом разделе, существует функция `IF()`, описание синтаксиса которой можно найти в *разделе 20.1.2*.

Рассмотрим простейший пример использования оператора `IF`. Хранимая процедура `pricelist()`, представленная в листинге 28.37, выводит список товарных позиций с каталогом, первичный ключ которых равен `id`. Вторым параметром функции является число `cur`, которое может принимать значение 0, если требуется вывести цены в рублях, и 1, если цены должны быть пересчитаны в доллары с курсом 28.8 рублей.

Замечание

Для создания логических выражений можно использовать все операторы сравнения ("=", ">", ">=", "<>", "<", "<="), описанные в *разделе 13.1.2*. Кроме того, логические выражения можно комбинировать между собой при помощи операторов "&&" (И), а также "||" (ИЛИ).

Листинг 28.37. Использование оператора `IF`

```
mysql> CREATE PROCEDURE pricelist (id INT, cur INT)
    -> LANGUAGE SQL
    -> BEGIN
    ->   IF(cur = 0) THEN
    ->     SELECT name, price FROM products WHERE id_catalog = id;
    ->   END IF;
    ->   IF(cur = 1) THEN
    ->     SELECT name, price/28.8 FROM products WHERE id_catalog = id;
```

```
    ->   END IF;
    -> END
    -> //
mysql> CALL pricelist(1,1)//
+-----------------------+------------+
| name                  | price/28.8 |
+-----------------------+------------+
| Celeron 1.8           | 55.381944  |
| Celeron 2.0GHz        | 68.368056  |
| Celeron 2.4GHz        | 73.229167  |
| Celeron D 320 2.4GHz  | 68.125000  |
| Celeron D 325 2.53GHz | 95.381944  |
| Celeron D 315 2.26GHz | 65.277778  |
| Intel Pentium 4 3.2GHz | 252.048611 |
| Intel Pentium 4 3.0GHz | 213.437500 |
| Intel Pentium 4 3.0GHz | 196.979167 |
+-----------------------+------------+
```

Оператор `IF` может быть снабжен дополнительным блоком `ELSE`, после которого выполняются операторы, если условие оказалось ложным. Функцию `pricelist()` можно переписать так, как это показано в листинге 28.38.

Замечание

После ключевого слова `ELSE` ставить точку с запятой не нужно, т. к. этим самым единый оператор `IF` разбивается на части — точку с запятой ставят после ключевого слова `END IF`.

Листинг 28.38. Использование оператора `IF...ELSE`

```
mysql> CREATE PROCEDURE pricelist (id INT, cur INT)
    -> LANGUAGE SQL
    -> BEGIN
    ->   IF(cur) THEN
    ->     SELECT name, price/28.8 FROM products WHERE id_catalog = id;
    ->   ELSE
    ->     SELECT name, price FROM products WHERE id_catalog = id;
    ->   END IF;
    -> END
    -> //
```

С точки зрения функциональности, хранимые процедуры из листингов 28.37 и 28.38 абсолютно одинаковы, но в последнем случае понадобился лишь один оператор `IF`, который принимает в качестве логического выражения параметр `cur`. Если `cur` равен 1, что является истиной, выполняется первый оператор, выводящий цены в долларах, если параметр `cur` равен 0, что является ложью, выполняется запрос в блоке `ELSE`.

Оператор IF позволяет выбрать и большее число альтернатив. Пусть необходимо помимо рублевого и долларового прайс-листа, выводить цены в евро по курсу 34.5 рублей за евро. Таким образом, параметр cur может принимать следующие значения:

- 0 — рубли;
- 1 — доллары;
- 2 и выше — евро.

Листинг 28.39. Использование оператора IF...ELSEIF...ELSE

```
mysql> CREATE PROCEDURE pricelist (id INT, cur INT)
    -> LANGUAGE SQL
    -> BEGIN
    ->   IF(cur = 0) THEN
    ->     SELECT name, price FROM products WHERE id_catalog = id;
    ->   ELSEIF (cur = 1) THEN
    ->     SELECT name, price/28.8 FROM products WHERE id_catalog = id;
    ->   ELSE
    ->     SELECT name, price/34.5 FROM products WHERE id_catalog = id;
    ->   END IF;
    -> END
    -> //
```

Оператор IF в листинге 28.39 проверяет, не равен ли параметр cur нулю, если это так, срабатывает первый запрос и процедура выходит из IF. Если параметр cur не равен нулю, проверка перемещается к блоку ELSEIF, где происходит сравнение параметра cur с единицей. Равенство cur единице приводит к выполнению второго оператора SELECT. Если оператор cur принимает любое другое значение отличное от 0 и 1, выполняется третий запрос. Третий запрос будет выполнен, даже если cur является отрицательным значением.

Число блоков ELSEIF не ограничено — можно использовать любое их число. Добавим в процедуру pricelist() английский фунт стерлингов по курсу 50.4 рубля за фунт — будем использовать значение параметра cur, равное 3 и выше, для обозначения этой валюты (листинг 28.40).

Листинг 28.40. Использование оператора IF...ELSEIF...ELSE

```
mysql> CREATE PROCEDURE pricelist (id INT, cur INT)
    -> LANGUAGE SQL
    -> BEGIN
    ->   IF(cur = 0) THEN
    ->     SELECT name, price FROM products WHERE id_catalog = id;
    ->   ELSEIF (cur = 1) THEN
    ->     SELECT name, price/28.8 FROM products WHERE id_catalog = id;
    ->   ELSEIF (cur = 2) THEN
    ->     SELECT name, price/34.5 FROM products WHERE id_catalog = id;
```

```
    ->   ELSE
    ->     SELECT name, price/50.4 FROM products WHERE id_catalog = id;
    ->   END IF;
    -> END
    -> //
```

Однако злоупотреблять блоками ELSEIF не рекомендуется, т. к. большое число блоков ELSEIF снижает читабельность кода.

Если в блоках IF, ELSEIF и ELSE используется два или более операторов, для наглядности можно прибегать к составному оператору BEGIN ... END. В листинге 28.41 приводится вариант процедуры pricelist() с выводом прайс-листов, содержащих цены в рублях, долларах и евро. Однако для каждого списка будет выводиться общее число товарных позиций и их общая стоимость в выбранной валюте для каталога с первичным ключом id.

Листинг 28.41. Использование оператора IF...ELSEIF...ELSE

```
mysql> CREATE PROCEDURE pricelist (id INT, cur INT)
    -> LANGUAGE SQL
    -> BEGIN
    ->   IF(cur = 0) THEN
    ->     BEGIN
    ->       SELECT name, price FROM products WHERE id_catalog = id;
    ->       SELECT SUM(count), SUM(price*count) FROM products
    ->       WHERE id_catalog = id;
    ->     END;
    ->   ELSEIF (cur = 1) THEN
    ->     BEGIN
    ->       SELECT name, price/28.8 FROM products WHERE id_catalog = id;
    ->       SELECT SUM(count), SUM(price*count)/28.8  FROM products
    ->       WHERE id_catalog = id;
    ->     END;
    ->   ELSE
    ->     BEGIN
    ->       SELECT name, price/34.5 FROM products WHERE id_catalog = id;
    ->       SELECT SUM(count), SUM(price*count)/34.5  FROM products
    ->       WHERE id_catalog = id;
    ->     END;
    ->   END IF;
    -> END
    -> //
```

28.4.2. Оператор *CASE*

Оператор CASE позволяет осуществить множественный выбор и имеет две формы. Синтаксис первой формы оператора выглядит следующим образом:

```
CASE case_value
    WHEN when_value THEN statement_list
    [WHEN when_value THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

Синтаксис второй формы:

```
CASE
    WHEN search_condition THEN statement_list
    [WHEN search_condition THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

Замечание

Синтаксис оператора CASE внутри хранимой процедуры немного отличается от синтаксиса SQL-выражения CASE. Оператор CASE не может содержать конструкцию ELSE NULL, и его выполнение завершается с помощью выражения END CASE, а не END.

В первой форме оператор CASE сравнивает выражение case_value с when_value. Как только соответствие найдено, выполняется соответствующий оператор statement_list. Если ни одного соответствия не найдено, выполняется оператор statement_list, размещенный после ключевого слова ELSE (если оно, конечно, присутствует). Перепишем хранимую процедуру pricelist() из листинга 28.40 при помощи оператора CASE (листинг 28.42).

Листинг 28.42. Использование оператора CASE (первая форма)

```
mysql> CREATE PROCEDURE pricelist (id INT, cur INT)
    -> LANGUAGE SQL
    -> BEGIN
    ->   CASE cur
    ->     WHEN 0 THEN
    ->       SELECT name, price FROM products WHERE id_catalog = id;
    ->     WHEN 1 THEN
    ->       SELECT name, price/28.8 FROM products WHERE id_catalog = id;
    ->     WHEN 2 THEN
    ->       SELECT name, price/34.5 FROM products WHERE id_catalog = id;
    ->     WHEN 3 THEN
    ->       SELECT name, price/50.4 FROM products WHERE id_catalog = id;
    ->     ELSE
    ->       SELECT 'Ошибка в параметре cur';
    ->   END CASE;
    -> END
    -> //
```

```
mysql> CALL pricelist(1, 5)//
+------------------------+
| Ошибка в параметре cur |
+------------------------+
| Ошибка в параметре cur |
+------------------------+
```

В листинге 28.42 ситуация, когда в качестве параметра `cur` передано ошибочное значение, обработана специально в блоке `ELSE`. Подход с применением оператора `IF` это также допускает, но в операторе `CASE` ключевое слово `ELSE` лучше выделяется на фоне последовательности ключевых слов `WHEN` по сравнению с ключевыми словами `ELSEIF` в операторе `IF`.

Вторая форма оператора `CASE` позволяет осуществлять сравнение непосредственно в конструкции `WHEN` — как только будет найдено первое истинное значение, выполняется оператор `statement_list` и процедура выходит из оператора `CASE`. В листинге 28.43 представлена процедура `pricelist()`, реализованная с применением второй формы оператора `CASE`.

Листинг 28.43. Использование оператора `CASE` (вторая форма)

```
mysql> CREATE PROCEDURE pricelist (id INT, cur INT)
    -> LANGUAGE SQL
    -> BEGIN
    ->   CASE
    ->     WHEN cur = 0 THEN
    ->       SELECT name, price FROM products WHERE id_catalog = id;
    ->     WHEN cur = 1 THEN
    ->       SELECT name, price/28.8 FROM products WHERE id_catalog = id;
    ->     WHEN cur = 2 THEN
    ->       SELECT name, price/34.5 FROM products WHERE id_catalog = id;
    ->     WHEN cur = 3 THEN
    ->       SELECT name, price/50.4 FROM products WHERE id_catalog = id;
    ->     ELSE
    ->       SELECT 'Ошибка в параметре cur';
    ->   END CASE;
    -> END
    -> //
```

Если в одном блоке `WHEN` необходимо выполнить несколько запросов, следует использовать блок `BEGIN ... END`.

28.4.3. Оператор *WHILE*

Оператор `WHILE` выполняет цикл и имеет следующий синтаксис:

```
[label:] WHILE search_condition DO
    statement_list
END WHILE [label]
```

Цикл WHILE выполняет операторы statement_list до тех пор, пока условие search_condition истинно. При каждой итерации условие search_condition проверяется, и если при очередной проверке оно будет ложным (0), цикл завершит свое выполнение. Это означает, что если условие search_condition ложно с самого начала, цикл не выполнит ни одной итерации.

Если в цикле требуется выполнить более одного оператора, не обязательно заключать их в блок BEGIN ... END, т. к. эту функцию выполняет сам оператор WHILE. Выведем 3 раза текущую дату при помощи цикла WHILE (листинг 28.44).

Листинг 28.44. Использование оператора WHILE

```
mysql> CREATE PROCEDURE NOW5 ()
    -> LANGUAGE SQL
    -> BEGIN
    ->   DECLARE i INT DEFAULT 3;
    ->   WHILE i > 0 DO
    ->     SELECT NOW();
    ->     SET i = i - 1;
    ->   END WHILE;
    -> END
    -> //
mysql> CALL NOW5()//
+---------------------+
| NOW()               |
+---------------------+
| 2005-07-18 12:36:59 |
+---------------------+
+---------------------+
| NOW()               |
+---------------------+
| 2005-07-18 12:36:59 |
+---------------------+
+---------------------+
| NOW()               |
+---------------------+
| 2005-07-18 12:36:59 |
+---------------------+
```

Первый оператор в цикле WHILE выводит текущую дату, а второй вычитает из локальной переменной i единицу. Если единицу не вычитать, то образуется бесконечный цикл, из которого процедура никогда не выйдет, а будет бесполезно нагружать сервер, пока сеанс с ним не прекратится. Следует очень внимательно проектировать циклы, чтобы предотвратить бесконечные циклы.

В листинге 28.45 представлен код хранимой процедуры, которая выводит текущую дату num число раз, где num — параметр, задаваемый пользователем.

Листинг 28.45. Использование оператора WHILE

```
mysql> CREATE PROCEDURE NOWN (IN num INT)
    -> LANGUAGE SQL
    -> BEGIN
    ->   DECLARE i INT DEFAULT 0;
    ->   IF (num > 0) THEN
    ->     wet : WHILE i < num DO
    ->       SELECT NOW();
    ->       SET i = i + 1;
    ->     END WHILE wet;
    ->   ELSE
    ->     SELECT 'Ошибочное значение параметра';
    ->   END IF;
    -> END
    -> //
mysql> CALL NOWN(2)//
+---------------------+
| NOW()               |
+---------------------+
| 2005-07-18 12:51:27 |
+---------------------+
+---------------------+
| NOW()               |
+---------------------+
| 2005-07-18 12:51:27 |
+---------------------+
```

Как видно из листинга 28.45, цикл `WHILE` снабжен меткой `wet`. Метка в цикле предназначена не только для того, чтобы облегчить чтение кода при очень длинных циклах, она позволяет осуществлять досрочный выход из цикла.

Для досрочного выхода из цикла предназначен оператор `LEAVE`, который имеет следующий синтаксис:

```
LEAVE label
```

Оператор `LEAVE` прекращает выполнение блока, помеченного меткой `label`.

Замечание

Оператор `LEAVE` эквивалентен оператору `break` в С-подобных языках программирования.

Хранимая процедура `NOWN()` обладает недостатком — если задать очень большое значение аргумента `num`, можно создать псевдобесконечный цикл, который позволит злоумышленнику загрузить сервер бесполезной работой. Для предотвращения такой ситуации можно воспользоваться оператором `LEAVE`, который прекратит выполнение цикла по достижении критического числа итераций. В листинге 28.46 приводится пример хранимой процедуры, где число итераций ограничено двумя.

Листинг 28.46. Досрочный выход из цикла WHILE

```
mysql> CREATE PROCEDURE NOWN (IN num INT)
    -> LANGUAGE SQL
    -> BEGIN
    ->   DECLARE i INT DEFAULT 0;
    ->   IF (num > 0) THEN
    ->     wet : WHILE i < num DO
    ->       IF i > 2 THEN LEAVE wet;
    ->       END IF;
    ->       SELECT NOW();
    ->       SET i = i + 1;
    ->     END WHILE wet;
    ->   ELSE
    ->     SELECT 'Ошибочное значение параметра';
    ->   END IF;
    -> END
    -> //
mysql> CALL NOWN(10)//
+---------------------+
| NOW()               |
+---------------------+
| 2005-07-18 12:51:27 |
+---------------------+
+---------------------+
| NOW()               |
+---------------------+
| 2005-07-18 12:51:27 |
+---------------------+
```

Условие `IF i > 2 THEN LEAVE wet;` проверяет, не превысило ли значение счетчика `i` числа 2, и если это так, происходит прекращение цикла `WHILE`. Использование меток позволяет точно указать, какой цикл необходимо прервать. Если имеется вложенный цикл, можно явно указать, какой из двух циклов требуется прервать (листинг 28.47).

Листинг 28.47. Досрочный выход из вложенного цикла

```
first : WHILE i < num DO
  second : WHILE j < num DO
    IF i > 2 && j > 2 THEN LEAVE first;
    END IF;
    SELECT NOW();
    SET j = j + 1;
  END WHILE second;
  SET i = i + 1;
END WHILE first;
```

При достижении условия `i > 2 && j > 2` оператор `LEAVE` прервет не вложенный цикл `second`, а внешний цикл `first`, т. к. метка внешнего цикла явно указана после оператора.

При использовании досрочного выхода `LEAVE` можно даже использовать бесконечные циклы, т. к. рано или поздно внешний цикл будет завершен (листинг 28.48).

Листинг 28.48. Досрочный выход из вложенного цикла

```
first : WHILE 1 DO
  second : WHILE 1 DO
    IF i > 2 && j > 2 THEN LEAVE first;
    END IF;
    SELECT NOW();
    SET j = j + 1;
  END WHILE second;
  SET i = i + 1;
END WHILE first;
```

Цифра 1 в условии цикла всегда будет истинной и цикл не прекратится до тех пор, пока его не прервет оператор `LEAVE`.

Еще одним оператором, выполняющим досрочное прекращение цикла, является оператор `ITERATE`, который имеет следующий синтаксис:

```
ITERATE label
```

В отличие от оператора `LEAVE`, оператор `ITERATE` не прекращает выполнение цикла, он лишь выполняет досрочное прекращение текущей итерации.

Замечание

Оператор `ITERATE` эквивалентен оператору `continue` в С-подобных языках программирования.

Рассмотрим программу, которая в цикле формирует бинарную последовательность, добавляя к строке две единицы на четных итерациях и две единицы и два нуля на нечетных (листинг 28.49).

Листинг 28.49. Использование оператора `ITERATE`

```
mysql> CREATE PROCEDURE binstring (IN num INT)
    -> LANGUAGE SQL
    -> BEGIN
    ->   DECLARE i INT DEFAULT 0;
    ->   DECLARE bin TINYTEXT DEFAULT '';
    ->   IF (num > 0) THEN
    ->     wet : WHILE i < num DO
    ->       SET i = i + 1;
    ->       SET bin = CONCAT(bin, '11');
```

```
    ->       IF !(i/2 - CEILING(i/2)) THEN ITERATE wet;
    ->       END IF;
    ->       SET bin = CONCAT(bin, '00');
    ->     END WHILE wet;
    ->     SELECT bin;
    ->   ELSE
    ->     SELECT 'Ошибочное значение параметра';
    ->   END IF;
    -> END
    -> //
mysql> CALL binstring(10)//
+--------------------------------+
| bin                            |
+--------------------------------+
| 11001111001111001111001111 0011 |
+--------------------------------+
```

Бинарная последовательность хранится во временной строке `bin`, которая обязательно должна быть инициирована пустой строкой:

```
DECLARE bin TINYTEXT DEFAULT '';
```

Если инициализация не проведена, переменная получит значение `NULL`, и все операции с этой переменной также будут приводить к `NULL`. На каждой итерации переменной `bin` при помощи функции `CONCAT()` прибавляется последовательность '`11`'. Если индекс `i` является нечетным (1, 3, 5, 7, 9), текущий цикл прекращается при помощи ключевого слова `ITERATE`, если индекс является четным (0, 2, 4, 6, 8), то итерация выполняется до конца, т. е. к временной строке `bin` добавляется еще и последовательность '`00`'. На четность индекс `i` проверяется при помощи строки

```
i/2 - CEILING(i/2)
```

Если индекс `i` делится на 2 без остатка, это выражение вернет 0 (ложь), если число является нечетным, то выражение вернет 0.5 (истина).

> ***Замечание***
>
> Необходимо следить, чтобы оператор `SET`, увеличивающий значение счетчика `i` на единицу, на каждой итерации был расположен до оператора `ITERATE`, иначе это приведет к созданию бесконечного цикла — значение счетчика будет оставаться нечетным и увеличиваться не будет, т. к. оператор `ITERATE` будет прекращать выполнение итерации цикла досрочно.

28.4.4. Оператор *REPEAT*

Оператор `REPEAT`, так же как и оператор `WHILE`, реализует цикл:

```
[label:] REPEAT
    statement_list
UNTIL search_condition
END REPEAT [label]
```

Отличительной особенностью данного цикла является тот факт, что условие цикла search_condition проверяется не в начале, как в цикле WHILE, а в конце оператора (ключевое слово UNTIL). Таким образом, цикл выполняет, по крайней мере, одну итерацию независимо от условия. Следует отметить, что цикл REPEAT выполняется, пока условие search_condition ложно.

Оператор REPEAT может быть снабжен необязательной меткой label, по которой можно осуществлять досрочный выход из цикла при помощи операторов LEAVE и ITERATE, рассмотренных в предыдущей главе.

В листинге 28.50 представлена хранимая процедура binrand(), которая генерирует и выводит случайную бинарную последовательность из 20 символов. Для формирования бинарной последовательности используется цикл REPEAT (листинг 28.50).

Листинг 28.50. Использование цикла REPEAT

```
mysql> CREATE PROCEDURE binrand ()
    -> LANGUAGE SQL
    -> BEGIN
    ->   DECLARE i INT DEFAULT 0;
    ->   DECLARE bin TINYTEXT DEFAULT '';
    ->   REPEAT
    ->       SET i = i + 1;
    ->       IF RAND() > 0.5 THEN SET bin = CONCAT(bin, '1');
    ->       ELSE SET bin = CONCAT(bin, '0');
    ->       END IF;
    ->   UNTIL i >= 20
    ->   END REPEAT;
    ->   SELECT bin;
    -> END
    -> //
mysql> CALL binrand()//
+------------------------+
| bin                    |
+------------------------+
| 10101000111100011111110 |
+------------------------+
```

28.4.5. Оператор *LOOP*

Оператор LOOP предназначен для реализации циклов и имеет следующий синтаксис:

```
[label:] LOOP
    statement_list
END LOOP [label]
```

Цикл LOOP, в отличие от операторов WHILE и REPEAT, не имеет условий выхода. Поэтому данный вид цикла должен обязательно иметь в своем составе оператор LEAVE.

Хранимую процедуру binrand(), приведенную в листинге 28.50, можно переписать с использованием цикла LOOP, как это сделано в листинге 28.51.

Листинг 28.51. Использование оператора LOOP

```
mysql> CREATE PROCEDURE binrand ()
    -> LANGUAGE SQL
    -> BEGIN
    ->   DECLARE i INT DEFAULT 0;
    ->   DECLARE bin TINYTEXT DEFAULT '';
    ->   wet : LOOP
    ->       SET i = i + 1;
    ->       IF RAND() > 0.5 THEN SET bin = CONCAT(bin, '1');
    ->       ELSE SET bin = CONCAT(bin, '0');
    ->       END IF;
    ->       IF i >= 20 THEN LEAVE wet;
    ->       END IF;
    ->   END LOOP wet;
    ->   SELECT bin;
    -> END
    -> //
mysql> CALL binrand()//
+----------------------+
| bin                  |
+----------------------+
| 01000111011110001110 |
+----------------------+
```

28.4.6. Оператор *GOTO*

Оператор GOTO позволяет осуществлять безусловный переход и имеет следующий синтаксис:

```
GOTO label;
```

Оператор GOTO осуществляет переход к оператору, помеченному меткой label. Это может быть как оператор BEGIN, так и любой из циклов: WHILE, REPEAT и LOOP. Кроме того, метка может быть не привязана ни к одному из операторов процедуры, а объявлена при помощи оператора LABEL, который имеет следующий синтаксис:

```
LABEL label;
```

В общем случае использование операторов GOTO и LABEL может выглядеть так, как это представлено в листинге 28.52.

Листинг 28.52. Использование операторов GOTO и LABEL

```
CREATE PROCEDURE binrand ()
BEGIN
...
```

```
LABEL labelname;
...
GOTO labelname;
...
END
```

Оператор LABEL не выполняет никаких действий — задача этого оператора просто объявить метку labelname. На месте этого оператора может быть помещен любой оператор, допускающий использование в своем составе метки (BEGIN, WHILE, REPEAT и LOOP). Достигая оператора GOTO, процедура следует по ссылке и переходит обратно к метке labelname. Таким образом, в листинге 28.52 реализован бесконечный цикл. Обычно переход по метке GOTO используют совместно с оператором IF, т. к. это продемонстрировано в листинге 28.53, где представлена реализация хранимой процедуры binrand() с использованием оператора безусловного перехода GOTO.

Листинг 28.53. Использование операторов GOTO и IF

```
mysql> CREATE PROCEDURE binrand ()
    -> LANGUAGE SQL
    -> BEGIN
    ->   DECLARE i INT DEFAULT 0;
    ->   DECLARE bin TINYTEXT DEFAULT '';
    ->   LABEL wet;
    ->   SET i = i + 1;
    ->   IF RAND() > 0.5 THEN SET bin = CONCAT(bin, '1');
    ->   ELSE SET bin = CONCAT(bin, '0');
    ->   END IF;
    ->   IF(i < 20) THEN GOTO wet;
    ->   END IF;
    ->   SELECT bin;
    -> END
    -> //
mysql> CALL binrand()//
+----------------------+
| bin                  |
+----------------------+
| 00001010001110111100 |
+----------------------+
```

Однако использовать оператор GOTO для реализации циклов не рекомендуется, т. к. обычные циклы гораздо нагляднее и проще поддаются модификации, к тому же в них сложнее допустить логическую ошибку.

Часто вообще рекомендуется не использовать оператор GOTO, т. к. он подталкивает к созданию плохо структурированного кода. Это не исключает возможности создания читабельного и структурированного кода с использованием оператора GOTO, но соблазн неправильной реализации этого оператора слишком велик. Кроме того, любой

код можно переписать без использования GOTO, некоторые современные языки программирования (например, PHP), вообще не включают его в свой состав.

28.5. Метаданные

Существует четыре способа просмотреть данные, относящиеся к хранимым процедурам или функциям:

1. Операторы SHOW PROCEDURE STATUS / SHOW FUNCTION STATUS.
2. Операторы SHOW CREATE PROCEDURE / SHOW CREATE FUNCTION.
3. Запрос SELECT FROM mysq.proc.
4. Запрос SELECT FROM information_schema.

Первые три рассматриваются в данном разделе, последний способ с привлечением информационной схемы рассматривается в *главе 31*.

28.5.1. Оператор *SHOW PROCEDURE STATUS*

Просмотреть список уже созданных хранимых процедур можно при помощи оператора SHOW PROCEDURE STATUS, который имеет следующий синтаксис:

```
SHOW PROCEDURE STATUS [LIKE 'pattern'];
```

Оператор возвращает список хранимых процедур, который не включает хранимых функций. При использовании ключевого слова LIKE можно вывести информацию только о тех процедурах, имена которых удовлетворяют шаблону pattern.

Листинг 28.54. Использование оператора SHOW PROCEDURE STATUS

```
mysql> SHOW PROCEDURE STATUS LIKE 'bin%'\G;
*************************** 1. row ***************************
           Db: shop
         Name: binrand
         Type: PROCEDURE
      Definer: root@localhost
     Modified: 2005-07-18 23:43:23
      Created: 2005-07-18 23:43:23
Security_type: DEFINER
      Comment:
*************************** 2. row ***************************
           Db: shop
         Name: binstring
         Type: PROCEDURE
      Definer: root@localhost
     Modified: 2005-07-18 14:09:55
      Created: 2005-07-18 14:09:55
Security_type: DEFINER
      Comment:
```

Как видно из листинга 28.54, оператор возвращает результирующую таблицу, в которой каждая строка соответствует одной хранимой процедуре. При этом таблица содержит 8 следующих столбцов:

- `Db` — имя базы данных, в которую сохранена процедура;
- `Name` — имя процедуры;
- `Type` — тип хранимой процедуры, принимает значение `PROCEDURE` для оператора хранимой процедуры и `FUNCTION` для хранимой функции (список хранимых функций возвращается оператором `SHOW PROCEDURE STATUS`, который рассматривается ниже);
- `Definer` — учетная запись, от имени которой была создана хранимая процедура;
- `Modified` — дата последней модификации хранимой процедуры;
- `Created` — дата создания хранимой процедуры;
- `Security_type` — режим выполнения хранимой процедуры. Если это поле принимает значение `DEFINER`, то хранимая процедура выполняется с правами доступа пользователя, создавшего данную процедуру. Если поле `Security_type` принимает значение `INVOKER`, то хранимая процедура выполняется с правами доступа пользователя, вызывающего процедуру при помощи оператора `CALL`.
- `Comment` — комментарий к хранимой процедуре.

Для просмотра списка хранимых функций предназначен оператор `SHOW FUNCTION STATUS`. Оператор имеет следующий синтаксис:

```
SHOW FUNCTION STATUS [LIKE 'pattern'];
```

Оператор выводит список хранимых функций, который не включает в свой состав хранимых процедур (листинг 28 55). При использовании ключевого слова `LIKE` можно вывести информацию только о тех функциях, имена которых удовлетворяют шаблону `pattern`.

Листинг 28.55. Использование оператора `SHOW FUNCTION STATUS`

```
mysql> SHOW FUNCTION STATUS LIKE 'count%'\G;
*************************** 1. row ***************************
           Db: shop
         Name: count_product
         Type: FUNCTION
      Definer: root@localhost
     Modified: 2005-07-17 01:26:41
      Created: 2005-07-17 01:26:41
Security_type: DEFINER
      Comment:
*************************** 2. row ***************************
           Db: shop
         Name: count_product_in_catalog
         Type: FUNCTION
```

```
      Definer: root@localhost
     Modified: 2005-07-17 01:26:40
      Created: 2005-07-17 01:26:40
Security_type: DEFINER
      Comment:
```

Формат вывода оператора `SHOW FUNCTION STATUS` совпадает с форматом `SHOW PROCEDURE STATUS`, рассмотренном ранее.

28.5.2. Оператор *SHOW CREATE*

Еще одним оператором, позволяющим получить информацию о хранимых процедурах, является оператор `SHOW CREATE PROCEDURE`, который имеет следующий синтаксис:

```
SHOW CREATE PROCEDURE procname;
```

Оператор выводит оператор `CREATE PROCEDURE`, при помощи которого была создана хранимая процедура `procname` (листинг 28.56).

Листинг 28.56. Использование оператора `SHOW CREATE PROCEDURE`

```
mysql> SHOW CREATE PROCEDURE binstring\G;
*************************** 1. row ***************************
       Procedure: binstring
        sql_mode:
Create Procedure: CREATE PROCEDURE 'shop'.'binstring'(IN num INT)
BEGIN
  DECLARE i INT DEFAULT 0;
  DECLARE bin TINYTEXT DEFAULT '';
  IF (num > 0) THEN
    wet : WHILE i < num DO
      SET i = i + 1;
      SET bin = CONCAT(bin, '11');
      IF !(i/2 - CEILING(i/2)) THEN ITERATE wet;
      END IF;
      SET bin = CONCAT(bin, '00');
    END WHILE wet;
    SELECT bin;
  ELSE
    SELECT 'Ошибочное значение параметра';
  END IF;
END
```

Оператор `SHOW CREATE PROCEDURE` выводит информацию только для хранимых процедур, для хранимых функций необходимо воспользоваться оператором `SHOW CREATE FUNCTION`, который имеет следующий синтаксис:

```
SHOW CREATE FUNCTION funcname;
```

Оператор выводит оператор CREATE FUNCTION, при помощи которого была создана хранимая процедура funcname.

28.5.3. Извлечение информации из таблицы *mysql.proc*

Помимо представленных выше операторов SHOW, существует еще один способ извлечь информацию о хранимых процедурах — извлечь строки таблицы proc системной базы данных mysql, куда сохраняются все хранимые процедуры. В листинге 28.57 приводится пример SELECT-запроса, извлекающего запись, соответствующую хранимой функции count_product_in_catalog() (см. листинг 28.32).

Листинг 28.57. Извлечение информации из таблицы mysql.proc

```
mysql> SELECT * FROM mysql.proc
    -> WHERE name = 'count_product_in_catalog'\G;
*************************** 1. row ***************************
              db: shop
            name: count_product_in_catalog
            type: FUNCTION
   specific_name: count_product_in_catalog
        language: SQL
 sql_data_access: CONTAINS_SQL
is_deterministic: NO
   security_type: DEFINER
      param_list: id INT
         returns: int(11)
            body: BEGIN
  DECLARE total INT;
  SELECT SUM(count) INTO total FROM products
  WHERE id_catalog = id LIMIT 1;
  RETURN total;
END
         definer: root@localhost
         created: 2005-07-17 01:26:40
        modified: 2005-07-17 01:26:40
        sql_mode:
         comment:
```

Как видно из листинга 28.57, результирующая таблица имеет 16 полей:

- db — имя базы данных, в которую сохранена процедура;
- name — имя процедуры;
- type — тип хранимой процедуры, может принимать два значения: PROCEDURE или FUNCTION, для хранимой процедуры или функции соответственно;
- specific_name — имя процедуры;

- language — язык, на котором написана хранимая процедура, в настоящий момент принимает единственное значение — SQL;
- sql_data_access — поле показывает, насколько зависит хранимая процедура от данных, и может принимать следующие значения: CONTAINS_SQL, NO_SQL, READS_SQL_DATA, MODIFIES_SQL_DATA, если процедура соответственно использует константы, не использует SQL, читает данные или модифицирует их;
- is_deterministic — данное поле определяет, является ли хранимая процедура детерминированной (YES) или нет (NO). На состояние данного поля оказывает влияние характеристика DETERMINISTIC;
- security_type — режим выполнения хранимой процедуры. Если это поле принимает значение DEFINER, то хранимая процедура выполняется с правами доступа создавшего ее пользователя. Если поле Security_type принимает значение INVOKER, то хранимая процедура выполняется с правами доступа пользователя, вызывающего процедуру при помощи оператора CALL;
- param_list — список параметров хранимой процедуры;
- returns — тип результата, возвращаемого хранимой функцией, для хранимых процедур поле принимает пустую строку;
- body — тело хранимой процедуры от оператора BEGIN до оператора END;
- definer — учетная запись, из-под которой была создана хранимая процедура;
- created — время создания хранимой процедуры;
- modified — время последней модификации хранимой процедуры;
- sql_mode — режимы выполнения хранимой процедуры;
- comment — комментарий к хранимой процедуре.

Удобство данного подхода заключается в том, что при формировании отчета можно использовать всю гибкость, предоставляемую оператором SELECT, и извлекать только ту информацию, которая действительно необходима. В листинге 28.58 формируется список всех хранимых процедур и функций, причем выводится только название и признак — процедура или функция.

Листинг 28.58. Извлечение списка хранимых процедур

```
mysql> SELECT name, type FROM proc;
+-------------------------+-----------+
| name                    | type      |
+-------------------------+-----------+
| binrand                 | PROCEDURE |
| binstring               | PROCEDURE |
| catalogname             | PROCEDURE |
| catalog_by_product      | PROCEDURE |
| characteristics         | PROCEDURE |
| count_by_id             | PROCEDURE |
```

```
| count_product             | FUNCTION  |
| count_product_in_catalog | FUNCTION  |
| declare_var               | PROCEDURE |
| func_catalog              | FUNCTION  |
| inner_declare_var         | PROCEDURE |
| NOW5                      | PROCEDURE |
| NOWN                      | PROCEDURE |
| numcatalogs               | PROCEDURE |
| numcatalogsview           | PROCEDURE |
| one_declare_var           | PROCEDURE |
| pricelist                 | PROCEDURE |
| say_hello                 | FUNCTION  |
| set_x                     | PROCEDURE |
| set_y                     | PROCEDURE |
+--------------------------+-----------+
```

28.6. Удаление хранимых процедур

Для удаления хранимых процедур используется оператор `DROP PROCEDURE`, который имеет следующий синтаксис:

```
DROP PROCEDURE [IF EXISTS] nameproc
```

Оператор `DROP ROCEDURE` позволяет удалить хранимую процедуру `nameproc`. Если процедуры с таким именем не существует, оператор возвращает ошибку, которую можно подавить, если использовать необязательное ключевое слово `IF EXISTS`.

Создадим в базе данных `test` хранимую процедуру `test()` и применим к ней оператор `DROP PROCEDURE` (листинг 28.59).

Листинг 28.59. Использование оператора `DROP PROCEDURE`

```
mysql> use test
Database changed
mysql> CREATE PROCEDURE test() SELECT VERSION();
mysql> SELECT name, type FROM mysql.proc WHERE db = 'test';
+------+-----------+
| name | type      |
+------+-----------+
| test | PROCEDURE |
+------+-----------+
mysql> DROP PROCEDURE test;
mysql> SELECT name, type FROM mysql.proc WHERE db = 'test';
Empty set (0.00 sec)
```

Однако использование оператора `DROP PROCEDURE` применительно к хранимой функции заканчивается ошибкой (листинг 28.60).

Листинг 28.60. Удаление хранимой функции

```
mysql> CREATE FUNCTION test() RETURNS TINYTEXT RETURN VERSION();
mysql> SELECT name, type FROM mysql.proc WHERE db = 'test';
+------+----------+
| name | type     |
+------+----------+
| test | FUNCTION |
+------+----------+
mysql> DROP PROCEDURE test;
ERROR 1305 (42000): PROCEDURE test.test does not exist
mysql> SELECT name, type FROM mysql.proc WHERE db = 'test';
+------+----------+
| name | type     |
+------+----------+
| test | FUNCTION |
+------+----------+
```

Для удаления хранимых функций необходимо использовать специальный оператор `DROP FUNCTION` (листинг 28.61).

Листинг 28.61. Использование оператора `DROP FUNCTION`

```
mysql> SELECT name, type FROM mysql.proc WHERE db = 'test';
+------+----------+
| name | type     |
+------+----------+
| test | FUNCTION |
+------+----------+
mysql> DROP FUNCTION test;
mysql> SELECT name, type FROM mysql.proc WHERE db = 'test';
Empty set (0.00 sec)
```

28.7. Редактирование хранимых процедур

Для изменения характеристик хранимой процедуры предназначен оператор `ALTER PROCEDURE`. Редактирования хранимой функции выполняется с помощью оператора `ALTER FUNCTION`. Операторы имеют следующий синтаксис:

```
ALTER PROCEDURE sp_name [characteristic ...]
ALTER FUNCTION sp_name [characteristic ...]
```

Характеристика `characteristic` может принимать следующие значения:

- `SQL SECURITY {DEFINER | INVOKER}` — данное предложение определяет режим выполнения: хранимая процедура выполняется либо с правами создавшего ее пользователя, (`DEFINER`), либо с правами пользователя, вызвавшего ее (`INVOKER`);

- `COMMENT 'string'` — данное предложение позволяет назначить комментарий для хранимой процедуры.

> ***Замечание***
>
> Для выполнения операторов `ALTER PROCEDURE` и `ALTER FUNCTION` необходимо обладать привилегией `ALTER ROUTINE`. Данная привилегия автоматически передается пользователю, создавшему хранимую процедуру.

Листинг 28.62. Использование оператора `ALTER PROCEDURE`

```
mysql> CREATE PROCEDURE test() SELECT VERSION();
mysql> SELECT name, type, security_type, comment FROM mysql.proc
    -> WHERE db = 'test';
+------+-----------+---------------+---------+
| name | type      | security_type | comment |
+------+-----------+---------------+---------+
| test | PROCEDURE | DEFINER       |         |
+------+-----------+---------------+---------+
mysql> ALTER PROCEDURE test
    -> SQL SECURITY INVOKER
    -> COMMENT 'Функция возвращает версию сервера';
mysql> SELECT name, type, security_type, comment FROM mysql.proc
    -> WHERE db = 'test';
+------+-----------+---------------+-----------------------------------+
| name | type      | security_type | comment                           |
+------+-----------+---------------+-----------------------------------+
| test | PROCEDURE | INVOKER       | Функция возвращает версию сервера |
+------+-----------+---------------+-----------------------------------+
```

Как видно из листинга 28.62, режим выполнения и комментарий к хранимой процедуре были успешно изменены.

28.8. Обработчики ошибок

Во время выполнения хранимых процедур и функций могут происходить самые разнообразные ошибки. При интерактивном выполнении запросов отследить возникающие ошибки и предпринять действия, направленные на их предотвращение, гораздо легче, чем при пакетном выполнении десятков запросов в хранимой процедуре. Поэтому СУБД MySQL поддерживает обработчики ошибок, которые позволяют каждой возникающей в хранимой процедуре ошибке назначить свой собственный обработчик. Кроме того, обработчик в зависимости от ситуации и серьезности ошибки может как прекратить, так и продолжить выполнение процедуры.

Для объявления такого обработчика предназначен оператор `DECLARE ... HANDLER FOR`, который имеет следующий синтаксис:

```
DECLARE handler_type HANDLER FOR condition_value[,...] sp_statement
```

Тип обработчика handler_type может принимать одно из трех значений:

- CONTINUE — выполнение текущей операции продолжается после выполнения оператора обработчика;
- EXIT — выполнение составного оператора BEGIN ... END, в котором объявлен обработчик, прекращается;
- UNDO — данный вид обработчика пока не поддерживается.

Конструкция condition_value задает код ошибки, для которой будет происходить срабатывание обработчика. Обработчик может быть привязан сразу к нескольким ошибкам, для этого их коды следует перечислить через запятую. Конструкция condition_value может принимать одно из следующих значений:

- SQLSTATE [VALUE] sqlstate_value — значение SQLSTATE является пятисимвольным кодом ошибки в шестнадцатеричном формате и является стандартом в SQL. В СУБД MySQL данный код поддерживается, начиная с версии 4.1. Примером таких кодов являются 'HY000', 'HY001', '42000' и т. п. Следует отметить, что один код обозначает сразу несколько ошибок MySQL;
- SQLWARNING — любое предупреждение MySQL. Данное ключевое слово позволяет назначить обработчик для всех предупреждений MySQL. В терминах SQLSTATE обрабатываются любые события, для которых код SQLSTATE начинается с '01';
- NOT FOUND — любая ошибка MySQL, связанная с отсутствием или невозможностью найти объект (таблицу, процедуру, функцию, столбец и т. п.). Данное ключевое слово позволяет назначить обработчик для всех ошибок такого рода. В терминах SQLSTATE обрабатываются любые события, для которых код SQLSTATE начинается с '02';
- SQLEXCEPTION — ошибки, не охваченные ключевыми словами SQLWARNING и NOT FOUND;
- mysql_error_code — обычные четырехзначные ошибки MySQL, такие как '1020', '1232', '1324' и т. п.;
- condition_name — имя условия, которое объявляется при помощи оператора DECLARE ... CONDITON FOR, рассмотренного далее.

Выражение sp_statement содержит SQL-запрос, который выполняется при срабатывании обработчика.

Для демонстрации приемов работы с обработчиками ошибок создадим таблицу tbl, содержащую единственный столбец id, являющийся первичным ключом (листинг 28.63).

Листинг 28.63. Создание таблицы tbl

```
mysql> CREATE TABLE tbl (id INT, PRIMARY KEY(id));
```

Так как поле id объявлено первичным ключом, его значения обязаны быть строго уникальными. Добавление в таблицу значений, совпадающих с одним из тех, которые уже существуют в таблице, приведет к возникновению ошибочной ситуации. Созда-

дим хранимую процедуру `handler_key()`, которая содержит обработчик такой ситуации, и попытаемся добавить в таблицу `tbl` два одинаковых значения.

Листинг 28.64. Использование оператора `DECLARE ... HANDLER FOR`

```
mysql> CREATE PROCEDURE handler_key ()
    -> LANGUAGE SQL
    -> BEGIN
    ->   DECLARE CONTINUE HANDLER
    ->   FOR SQLSTATE '23000' SET @error = 'Ошибка';
    ->   INSERT INTO tbl VALUES(1);
    ->   INSERT INTO tbl VALUES(1);
    ->   INSERT INTO tbl VALUES(1);
    ->   SELECT VERSION();
    -> END
    -> //
mysql> SELECT @error//
+--------+
| @error |
+--------+
| NULL   |
+--------+
mysql> CALL handler_key()//
+---------------+
| VERSION()     |
+---------------+
| 5.0.6-beta-nt |
+---------------+
mysql> SELECT @error//
+--------+
| @error |
+--------+
| Ошибка |
+--------+
```

Как видно из листинга 28.64, в результате обработки ошибки не происходит остановки работы процедуры и последний запрос `SELECT VERSION()` успешно выполняется, несмотря на то, что предпринимается две попытки присвоить первичному ключу не уникальное значение. В результате срабатывания обработчика пользовательская переменная `@error` получает значение '`Ошибка`'.

В качестве обработчика ошибки можно использовать и составной оператор (листинг 28.65).

Листинг 28.65. Использование составного оператора в обработчике

```
mysql> CREATE PROCEDURE handler_key ()
    -> LANGUAGE SQL
```

```
    -> BEGIN
    ->   DECLARE CONTINUE HANDLER
    ->   FOR SQLSTATE '23000'
    ->   BEGIN
    ->     SET @error = 'Ошибка';
    ->     SELECT 'Ошибка при вставке нового значения в таблицу tbl';
    ->   END;
    ->   INSERT INTO tbl VALUES(2);
    ->   INSERT INTO tbl VALUES(2);
    ->   INSERT INTO tbl VALUES(2);
    -> END
    -> //
mysql> CALL handler_key()//
+-------------------------------------------------+
| Ошибка при вставке нового значения в таблицу tbl |
+-------------------------------------------------+
| Ошибка при вставке нового значения в таблицу tbl |
+-------------------------------------------------+
+-------------------------------------------------+
| Ошибка при вставке нового значения в таблицу tbl |
+-------------------------------------------------+
| Ошибка при вставке нового значения в таблицу tbl |
+-------------------------------------------------+
mysql> SELECT @error//
+--------+
| @error |
+--------+
| Ошибка |
+--------+
```

Как видно из листинга 28.65, процедура выполняет все запросы, не прерываясь и лишь вызывая операторы из обработчика при возникновении каждой ошибки. Для того чтобы прекратить выполнение функции при возникновении ошибки, необходимо воспользоваться обработчиком `DECLARE EXIT` (листинг 28.66).

Листинг 28.66. Использование обработчика `EXIT`

```
mysql> CREATE PROCEDURE handler_key ()
    -> LANGUAGE SQL
    -> BEGIN
    ->   DECLARE EXIT HANDLER
    ->   FOR SQLSTATE '23000'
    ->   BEGIN
    ->     SET @error = 'Ошибка';
    ->     SELECT 'Ошибка при вставке нового значения в таблицу tbl';
    ->   END;
```

```
    ->   INSERT INTO tbl VALUES(3);
    ->   INSERT INTO tbl VALUES(3);
    ->   INSERT INTO tbl VALUES(3);
    ->   SELECT VERSION();
    -> END
    -> //
mysql> CALL handler_key()//
+-------------------------------------------------+
| Ошибка при вставке нового значения в таблицу tbl  |
+-------------------------------------------------+
| Ошибка при вставке нового значения в таблицу tbl |
+-------------------------------------------------+
mysql> SELECT @error//
+--------+
| @error |
+--------+
| Ошибка |
+--------+
```

Как видно из листинга 28.66, если используется обработчик `EXIT`, то при возникновении первой же ошибки функция прекращает свою работу.

При указании кода ошибки можно использовать не только целочисленные коды, но и именованные условия, которые объявляются при помощи оператора `DECLARE ... CONDITION FOR`, имеющего следующий синтаксис:

```
DECLARE condition_name CONDITION FOR condition_value
```

Оператор объявляет именованное условие `condition_name` для ошибки `condition_value`, которая может принимать одно из следующих значений:

- `SQLSTATE [VALUE] sqlstate_value` — значение `SQLSTATE` является пятисимвольным кодом ошибки в шестнадцатеричном формате и является стандартом в SQL. В СУБД MySQL данный код поддерживается, начиная с версии 4.1. Примером таких кодов являются '`HY000`', '`HY001`', '`42000`' и т. п. Следует отметить, что один код обозначает сразу несколько ошибок MySQL;
- `mysql_error_code` — обычные четырехзначные ошибки MySQL, такие как '`1020`', '`1232`', '`1324`' и т. п.

Так, для обрабатываемой ранее ошибки 1062 (23000) — дублирование уникального индекса, оператор `DECLARE ... CONDITION FOR` может выглядеть так, как это представлено в листинге 28.67.

Листинг 28.67. Использование оператор `DECLARE ... CONDITION FOR`

```
DECLARE 'violation' CONDITION FOR SQLSTATE '23000';
DECLARE 'violation' CONDITION FOR 1062;
```

Первое объявление является более широким и охватывает все ошибки со статусом '23000', второй вид ошибок более узкий и включает только дублирование уникального индекса.

В листинге 28.68 демонстрируется использование именованных условий, при этом следует обратить внимание, что название именованного условия заключается в необязательные обратные кавычки.

Листинг 28.68. Использование именованных условий

```
mysql> CREATE PROCEDURE handler_key ()
    -> LANGUAGE SQL
    -> BEGIN
    ->   DECLARE 'violation' CONDITION FOR 1062;
    ->   DECLARE EXIT HANDLER FOR 'violation' ROLLBACK;
    ->   START TRANSACTION;
    ->     INSERT INTO tbl VALUES(3);
    ->     INSERT INTO tbl VALUES(3);
    ->     INSERT INTO tbl VALUES(3);
    ->   COMMIT;
    -> END
    -> //
```

В листинге 28.68 хранимая процедура выполняет транзакцию. При возникновении ошибки дублирования записей состояние базы данных откатывается при помощи оператора `ROLLBACK`.

Замечание

Транзакции подробнее описываются в *главе 24*.

28.9. Курсоры

Если результирующий запрос возвращает одну запись, поместить результаты в промежуточные переменные можно при помощи оператора `SELECT ... INTO ... FROM`. Однако результирующие таблицы чаще содержат несколько записей и использование такого запроса совместно с оператором `SELECT ... INTO ... FROM` приводит к возникновению ошибки 1172: "Результат содержит более чем одну строку" (листинг 28.69).

Листинг 28.69. Ошибочное выполнение хранимой процедуры

```
mysql> CREATE PROCEDURE idcatalogs (OUT total INT)
    -> BEGIN
    ->   SELECT id_catalog INTO total FROM catalogs;
    -> END
    -> //
```

```
mysql> CALL idcatalogs(@total)//
ERROR 1172 (42000): Result consisted of more than one row
```

Избежать возникновения ошибки можно, добавив предложение LIMIT 1 или назначив CONTINUE-обработчик ошибок. Однако функция будет реализовывать совсем не то поведение, которое ожидает пользователь. Кроме того, существуют ситуации, когда требуется обработать именно многострочную результирующую таблицу.

Например, пусть требуется вернуть записи таблицы catalogs учебной базы данных shop и на основании этих записей создать новую таблицу letter_catalogs, в которой названия каталогов будут представлены в верхнем регистре.

Решить эту задачу можно при помощи курсоров, которые позволяют в цикле просмотреть каждую строку результирующей таблицы запросов. Работа с курсорами происходит по следующему алгоритму:

1. При помощи инструкции DECLARE CURSOR связывается имя курсора с выполняемым запросом.
2. Оператор OPEN выполняет запрос, связанный с курсором, и устанавливает курсор перед первой записью результирующей таблицы.
3. Оператор FETCH помещает курсор на первую запись результирующей таблицы и извлекает данные из записи в локальные переменные хранимой процедуры. Повторный вызов оператора FETCH приводит к перемещению курсора к следующей записи и так до тех пор, пока записи в результирующей таблице не будут исчерпаны. Эту операцию удобно осуществлять в цикле.
4. Оператор CLOSE прекращает доступ к результирующей таблице и ликвидирует связь между курсором и результирующей таблицей.

Замечание

Работа с курсорами похожа на работу с файлами — сначала происходит открытие курсора, затем чтение и после закрытие.

Оператор DECLARE CURSOR объявляет курсор и имеет следующий синтаксис:

```
DECLARE cursor_name CURSOR FOR select_statement
```

Оператор объявляет курсор с именем cursor_name для SELECT-запроса select_statement. В рамках хранимой процедуры имя cursor_name должно быть уникальным.

Замечание

В SELECT-запросе select_statement не допускается использовать запрос вида SELECT ... INTO ... FROM.

В момент объявления курсора при помощи оператора DECLARE CURSOR SELECT-запрос select_statement не выполняется. Его выполнение откладывается до момента вызова оператора OPEN, который имеет следующий синтаксис:

```
OPEN cursor_name
```

Оператор OPEN принимает имя курсора cursor_name. Далее записи извлекаются при помощи оператора FETCH, который имеет следующий синтаксис:

```
FETCH cursor_name INTO var, var1, ...
```

После ключевого слова INTO должно быть приведено столько локальных переменных, сколько полей возвращает SELECT-запрос select_statement.

Оператор CLOSE имеет синтаксис, схожий с оператором OPEN:

```
CLOSE cursor_name
```

Вернемся к поставленной ранее задаче — создания новой таблицы letter_catalogs с именами каталогов в верхнем регистре. Создадим таблицу letter_catalogs при помощи запроса, представленного в листинге 28.70.

Листинг 28.70. Создание таблицы letter_catalogs

```
mysql> CREATE TABLE IF NOT EXISTS letter_catalogs
    -> (id_catalog INT,
    ->  name TINYTEXT);
```

Тогда процедура, выполняющая поставленную задачу, может выглядеть так, как это представлено в листинге 28.71.

Листинг 28.71. Использование курсоров

```
mysql> CREATE PROCEDURE curcatalogs ()
    -> BEGIN
    ->   /* Объявляем локальные переменные */
    ->   DECLARE id INT;
    ->   DECLARE is_end INT DEFAULT 0;
    ->   DECLARE cat TINYTEXT;
    ->
    ->   /* Объявляем курсор */
    ->   DECLARE curcat CURSOR FOR SELECT * FROM catalogs;
    ->
    ->   /* Объявляем обработчик для ситуации, когда курсор достигает
    ->      конца результирующей таблицы */
    ->   DECLARE CONTINUE HANDLER FOR NOT FOUND SET is_end = 1;
    ->
    ->   /* Открываем курсор */
    ->   OPEN curcat;
    ->
    ->   /* В цикле читаем данные и курсора и добавляем записи */
    ->   wet : LOOP
    ->     FETCH curcat INTO id, cat;
    ->     IF is_end THEN LEAVE wet;
    ->     END IF;
```

```
    ->     INSERT INTO letter_catalogs VALUES(id, UPPER(cat));
    ->   END LOOP wet;
    ->
    ->   /* Закрываем курсор */
    ->   CLOSE curcat;
    -> END
    -> //
mysql> CALL curcatalogs()//
mysql> SELECT * FROM letter_catalogs//
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          1 | ПРОЦЕССОРЫ         |
|          2 | МАТЕРИНСКИЕ ПЛАТЫ  |
|          3 | ВИДЕОАДАПТЕРЫ      |
|          4 | ЖЕСТКИЕ ДИСКИ      |
|          5 | ОПЕРАТИВНАЯ ПАМЯТЬ |
+------------+--------------------+
```

Как видно из листинга 28.71, хранимая процедура `curcatalogs()` заполняет таблицу `letter_catalogs` записями из таблицы `catalogs`, переводя названия каталогов в верхний регистр. Нетрудно переделать хранимую процедуру таким образом, чтобы она изменяла записи существующей таблицы `catalogs`. Для этого запрос

```
INSERT INTO letter_catalogs VALUES(id, UPPER(cat));
```

следует заменить `UPDATE`-запросом

```
UPDATE catalogs SET name = UPPER(cat) WHERE id_catalog = id.
```

ГЛАВА 29

Триггеры

Триггер — эта та же хранимая процедура, но привязанная к событию изменения содержимого таблицы. Возможны три события изменения таблицы, к которым можно привязать триггер: это изменение содержимого таблицы при помощи операторов `INSERT`, `DELETE` и `UPDATE`. Например, при оформлении нового заказа, т. е. при добавлении новой записи в таблицу `orders`, можно создать триггер, который автоматически будет вычитать число заказанных товарных позиций в таблице `products`.

> ***Замечание***
>
> Поддержка триггеров введена в СУБД MySQL, начиная с версии 5.0.2.

В данной главе рассматривается создание, удаление и использование триггеров.

29.1. Оператор *CREATE TRIGGER*

Оператор `CREATE TRIGGER` позволяет создать новый триггер и имеет следующий синтаксис:

```
CREATE TRIGGER trigger_name trigger_time trigger_event
    ON tbl_name FOR EACH ROW trigger_stmt
```

Оператор `CREATE TRIGGER` создает `trigger_name`, привязанный к таблице `tbl_name`. Таблица должна существовать физически, т. е. не допускается привязка триггера к временной таблице или представлению.

Конструкция `trigger_time` указывает момент выполнения триггера и может принимать два значения:

- `BEFORE` — действия триггера производятся до выполнения операции изменения таблицы;
- `AFTER` — действия триггера производятся после выполнения операции изменения таблицы.

Конструкция `trigger_event` показывает, на какое из событий должен реагировать триггер, и может принимать три значения:

- `INSERT` — триггер привязан к событию вставки новой записи в таблицу;
- `UPDATE` — триггер привязан к событию обновления записи таблицы;
- `DELETE` — триггер привязан к событию удаления записей таблицы.

Замечание

Для таблицы `tbl_name` может быть создан только один триггер для каждого из событий `trigger_event` и момента `trigger_time`. То есть для каждой из таблиц может быть создано всего шесть триггеров.

Конструкция `trigger_stmt` представляет тело триггера, т. е. оператор, который необходимо выполнить при возникновении события `trigger_event` в таблице `tbl_name`. Если требуется выполнить несколько операторов, то следует прибегать к составному оператору `BEGIN ... END`, в котором размещаются все требуемые запросы.

Вообще синтаксис и допустимые операторы совпадают с телом хранимых процедур *(см. главу 28)*. Внутри составного оператора `BEGIN ... END` допускаются все специфичные для хранимых процедур операторы и конструкции:

- другие составные операторы `BEGIN ... END`;
- операторы управления потоком (`IF`, `CASE`, `WHILE`, `LOOP`, `REPEAT`, `LEAVE`, `ITERATE`);
- объявления локальных переменных при помощи оператора `DECLARE` и назначение им значений при помощи оператора `SET`;
- именованные условия и обработчики ошибок.

Замечание

Для создания триггера при помощи оператора `CREATE TRIGGER` требуется наличие привилегии `SUPER`. Подробнее с системой привилегий СУБД MySQL можно познакомиться в *главе 25*.

Замечание

В СУБД MySQL триггеры нельзя привязать к каскадному обновлению или удалению записей из таблицы типы InnoDB по связи первичный ключ/внешний ключ.

Триггеры очень сложно использовать, не имея доступа к новым записям, которые вставляются в таблицу, или старым записям, которые обновляются или удаляются. Для доступа к новым и старым записям используются префиксы `NEW` и `OLD` соответственно. То есть если в таблице обновляется поле `total`, то получить доступ к старому значению можно по имени `OLD.total`, а к новому — `NEW.total`.

Создадим простейший триггер, который при оформлении нового заказа (добавление новой записи в таблицу `orders`) будет присваивать значение 1 пользовательской переменной `@tot` (листинг 29.1).

Листинг 29.1. Создание простейшего триггера

```
mysql> CREATE TRIGGER sub_count AFTER INSERT ON orders
    -> FOR EACH ROW
    -> BEGIN
    ->   SET @tot = 1;
    -> END
    -> //
mysql> select @tot//
+------+
| @tot |
+------+
| NULL |
+------+
mysql> INSERT INTO orders VALUES (NULL,1,NOW(),1,10)//
mysql> select @tot//
+------+
| @tot |
+------+
| 1    |
+------+
```

Как видно из листинга 29.1, в результате добавления новой записи в таблицу `orders` пользовательской переменной присваивается значение 1. Отредактируем триггер `sub_count` таким образом, чтобы к пользовательской переменной `@tot` прибавлялось каждый раз число заказанных товарных позиций `number` (листинг 29.2).

Листинг 29.2. Новый вариант триггера `sub_count`

```
mysql> CREATE TRIGGER sub_count AFTER INSERT ON orders
    -> FOR EACH ROW
    -> BEGIN
    ->   SET @tot = @tot + NEW.number;
    -> END
    -> //
mysql> select @tot//
+------+
| @tot |
+------+
| 1    |
+------+
mysql> INSERT INTO orders VALUES (NULL,1,NOW(),5,10)//
mysql> select @tot//
+------+
| @tot |
+------+
| 6    |
+------+
```

Как видно из листинга 29.2, для того чтобы получить число товарных позиций, внутри триггера происходит обращение к переменной `NEW.number`, которая связана с полем `number` `INSERT`-запроса.

Замечание

Для корректной работы триггера необходимо, чтобы пользовательская переменная `@tot` имела значение, отличное от `NULL`, т. к. операции сложения с `NULL` приводят к `NULL`.

Замечание

При создании триггеров для таблицы в каталоге данных создается файл, название которого совпадает с именем таблицы. Файл является текстовым и имеет расширение trg.

Предыдущие два примера демонстрировали работу триггеров после добавления записи в таблицу (`AFTER`) без вмешательства в запрос. Рассмотрим триггер, который будет включаться до (`BEFORE`) вставки новых записей в таблицу `orders`. Основная задача триггера заключается в ограничении числа заказываемых товаров до 1 (листинг 29.3).

Листинг 29.3. Использование ключевого слова `BEFORE`

```
mysql> CREATE TRIGGER restrict_count BEFORE INSERT ON orders
    -> FOR EACH ROW
    -> BEGIN
    ->   SET NEW.number = 1;
    -> END
    -> //
mysql> INSERT INTO orders VALUES (NULL,1,NOW(),2,10)//
mysql> SELECT * FROM orders WHERE id_order = LAST_INSERT_ID()//
+----------+---------+---------------------+--------+------------+
| id_order | id_user | ordertime           | number | id_product |
+----------+---------+---------------------+--------+------------+
|       16 |       1 | 2005-07-22 21:40:02 |      1 |         10 |
+----------+---------+---------------------+--------+------------+
```

Как видно из листинга 29.3, несмотря на то, что заказ оформлялся на две товарные позиции, триггер `restrict_count` изменил значение поля `number` на 1.

Часто при обновлении одних полей таблицы операторы баз данных забывают обновить связанные поля таблицы или производится попытка добавления некорректных значений. Пусть при добавлении нового покупателя необходимо преобразовать имена и отчества покупателей в инициалы. Триггер для обработки этой ситуации может выглядеть так, как это представлено в листинге 29.4.

Листинг 29.4. Контролирующий триггер

```
mysql> CREATE TRIGGER restrict_user BEFORE INSERT ON users
    -> FOR EACH ROW
```

```
    -> BEGIN
    ->   SET NEW.name = LEFT(NEW.name,1);
    ->   SET NEW.patronymic = LEFT(NEW.patronymic,1);
    -> END
    -> //
    -> INSERT INTO users VALUES (NULL, 'Ремизов', 'Алексеевич', 'Сергей',
    ->                           '83-89-00', NULL, NULL, 'active')//
mysql> SELECT surname, patronymic, name FROM users
    -> WHERE id_user = LAST_INSERT_ID()//
+---------+------------+------+
| surname | patronymic | name |
+---------+------------+------+
| Ремизов | А          | С    |
+---------+------------+------+
```

Как видно из листинга 29.4, имя и отчество любого нового посетителя урезается до одной буквы. Для того чтобы имя и отчество не было отредактировано при помощи оператора `UPDATE`, следует также создать триггер, привязанный к событию `UPDATE`.

29.2. Оператор *DROP TRIGGER*

Оператор `DROP TRIGGER` позволяет удалять существующие триггеры и имеет следующий синтаксис:

```
DROP TRIGGER tbl_name.trigger_name
```

Оператор удаляет триггер с именем `trigger_name` таблицы `tbl_name`. В листинге 29.5 демонстрируется удаление триггера `restrict_user` таблицы `users`, созданного в листинге 29.4.

Листинг 29.5. Удаление триггера `restrict_user`

```
mysql> DROP TRIGGER users.restrict_user;
```

ГЛАВА 30

Представления

Основной структурной единицей в реляционных базах данных являются таблицы. Однако язык запросов SQL предоставляет еще один способ организации данных — представления. *Представление* — это запрос на выборку (`SELECT`), которому присваивается уникальное имя и который можно сохранять или удалять из базы данных как обычную хранимую процедуру. Представления позволяют увидеть результаты сохраненного запроса, таким образом, как будто это полноценная таблица базы данных.

> ***Замечание***
>
> Представления и все связанные с ними операторы доступны в СУБД MySQL, начиная с версии 5.0.1.

Когда СУБД MySQL встречает в запросе ссылку на представление, она отыскивает его определение, сохраненное в базе данных. Затем происходит преобразование пользовательского запроса с участием представления в эквивалентный запрос к исходным таблицам. После чего происходит выполнение запроса. Таким образом, клиент может работать с представлениями так, как если бы они существовали как независимые таблицы. Если определение представления простое, то каждая строка представления формируется "на лету". Если определение сложное, СУБД MySQL "материализует" представление в виде временной таблицы в оперативной памяти или на жестком диске.

Клиент, обращаясь к представлению, будет видеть только столбцы результирующей таблицы. При этом не имеет значение, сколько столбцов в исходной таблице и является ли `SELECT`-запрос, лежащий в основе представления, одно- или многотабличным. Кроме того, клиенту можно запретить обращаться к исходным таблицам, но снабдить привилегиями обращения к представлениям. Таким образом, на одном и том же наборе таблиц можно создать гибкие системы доступа к таблицам базы данных.

Учитывая все вышесказанное, можно выделить следующие преимущества представлений:

- безопасность — каждому пользователю можно разрешить доступ к небольшому числу представлений, содержащих только ту информацию, которую ему позволено знать;

- простота запросов — с помощью представления можно извлечь данные из нескольких таблиц и представить их как одну таблицу, заменяя запрос ко многим таблицам в однотабличный запрос к представлению;
- простота структуры — представления позволяют создать для каждого пользователя собственную "структуру" базы данных, отображая только те данные, которые ему нужны, и "не засоряя" результаты вспомогательными столбцами, которые пользователю заведомо не пригодятся;
- защита от изменений — в связи с оптимизацией скорости, таблицы и их структура могут постоянно претерпевать изменения и переименовываться. Представления позволяют создавать виртуальные таблицы со старыми именами и структурой, позволяя избежать модификации внешней прикладной программы.

Помимо преимуществ, описанных выше, представления обладают рядом недостатков:

- производительность — представления создают лишь видимость существования соответствующей таблицы, и СУБД MySQL приходится преобразовывать запрос к представлению в запрос к исходным таблицам. Если представление отображает многотабличный запрос, то простой запрос к представлению становится сложным объединением и на его выполнение может потребоваться много времени;
- ограничение на обновление — когда пользователь пытается обновить строки представления, СУБД MySQL необходимо обновить строки в исходных таблицах. Это возможно только для простых представлений; сложные представления обновить нельзя, они доступны только для выборки.

Указанные недостатки означают, что не стоит без разбора применять представления вместо исходных таблиц. В каждом конкретном случае необходимо учитывать перечисленные преимущества и недостатки.

30.1. Создание представлений

Создание представлений осуществляется при помощи оператора `CREATE VIEW`, который имеет следующий синтаксис:

```
CREATE [OR REPLACE] [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    VIEW view_name [(column_list)]
    AS select_statement
    [WITH [CASCADED | LOCAL] CHECK OPTION]
```

Оператор создает представление `view_name` со столбцами, перечисленными в `column_list`, на основании `SELECT`-запроса `select_statement`. В листинге 30.1 приводится пример создания представления `cat`, которое дублирует таблицу `catalogs` учебной базы данных `shop`.

Листинг 30.1. Создание представления `cat`

```
mysql> CREATE VIEW cat AS SELECT * FROM catalogs;
mysql> SELECT * FROM cat;
```

```
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 | Оперативная память |
+------------+--------------------+
```

В каталоге данных для представления `cat` будет создан файл cat.frm, который, в отличие от файлов определения структуры таблиц, является не бинарным, а текстовым.

Листинг 30.2. Структура файла cat.frm представления `cat`

```
TYPE=VIEW
query=select 'shop'.'catalogs'.'id_catalog' AS
'id_catalog','shop'.'catalogs'.'name' AS 'name' from 'shop'.'catalogs'
md5=d0f2f9ade597ab03f3c876837262b4d6
updatable=1
algorithm=0
with_check_option=0
revision=1
timestamp=2005-07-14 14:51:37
create-version=1
source=CREATE VIEW cat AS SELECT * FROM catalogs
```

Представление рассматривается СУБД MySQL как полноценная таблица и может быть просмотрено в списке таблиц базы данных при помощи оператора `SHOW TABLES` (листинг 30.3).

Листинг 30.3. Просмотр представления при помощи оператора `SHOW TABLES`

```
mysql> SHOW TABLES;
+----------------+
| Tables_in_test |
+----------------+
| cat            |
| catalogs       |
| forums         |
| goods          |
| products       |
+----------------+
```

В листинге 30.1 список столбцов представления не уточняется, поэтому представление принимает структуру таблицы `catalogs`. Однако при создании представления можно явно указать список столбцов и даже изменить их названия и порядок следования (листинг 30.4).

Листинг 30.4. Изменения названия столбцов и порядка их следования

```
mysql> CREATE VIEW cat (catalog, id)
    -> AS SELECT name, id_catalog FROM catalogs;
mysql> SELECT * FROM cat;
+--------------------+----+
| catalog            | id |
+--------------------+----+
| Процессоры         |  1 |
| Материнские платы  |  2 |
| Видеоадаптеры      |  3 |
| Жесткие диски      |  4 |
| Оперативная память |  5 |
+--------------------+----+
```

Как видно из листинга 30.4, название столбца `name` меняется на `catalog`, а `id_catalog` на `id`. При этом порядок их следования меняется на обратный. При формировании списка столбцов представления задаются только имена столбцов, тип данных, длина и другие характеристики берутся из определения столбца исходной таблицы.

В качестве столбцов представления могут выступать вычисляемые столбцы. Пусть для таблицы `products` следует создать представление `goods`, содержащее имя товарной позиции, а также ее цену в рублях, долларах и евро. Пусть курс доллара будет составлять 28,8 рублей, а евро 34,5 рублей, тогда запрос на создание представления может выглядеть так, как это представлено в листинге 30.5.

Листинг 30.5. Создание представления с вычисляемыми столбцами

```
mysql> CREATE OR REPLACE VIEW goods (name, rubl, doll, euro)
    -> AS SELECT name, price, price/28.8, price/34.5 FROM products;
mysql> SELECT * FROM goods;
+------------------------------+---------+------------+------------+
| name                         | rubl    | doll       | euro       |
+------------------------------+---------+------------+------------+
| Celeron 1.8                  | 1595.00 | 55.381944  | 46.231884  |
| Celeron 2.0GHz               | 1969.00 | 68.368056  | 57.072464  |
| Celeron 2.4GHz               | 2109.00 | 73.229167  | 61.130435  |
| Celeron D 320 2.4GHz         | 1962.00 | 68.125000  | 56.869565  |
| Celeron D 325 2.53GHz        | 2747.00 | 95.381944  | 79.623188  |
| Celeron D 315 2.26GHz        | 1880.00 | 65.277778  | 54.492754  |
| Intel Pentium 4 3.2GHz       | 7259.00 | 252.048611 | 210.405797 |
| Intel Pentium 4 3.0GHz       | 6147.00 | 213.437500 | 178.173913 |
| Intel Pentium 4 3.0GHz       | 5673.00 | 196.979167 | 164.434783 |
| Gigabyte GA-8I848P-RS        | 1896.00 | 65.833333  | 54.956522  |
| Gigabyte GA-8IG1000          | 2420.00 | 84.027778  | 70.144928  |
| Gigabyte GA-8IPE1000G        | 2289.00 | 79.479167  | 66.347826  |
| Asustek P4C800-E Delux       | 5395.00 | 187.326389 | 156.376812 |
```

```
| Asustek P4P800-VM\L i865G     | 2518.00 | 87.430556  | 72.985507  |
| Epox EP-4PDA3I                | 2289.00 | 79.479167  | 66.347826  |
| ASUSTEK A9600XT/TD            | 5156.00 | 179.027778 | 149.449275 |
| ASUSTEK V9520X                | 1602.00 | 55.625000  | 46.434783  |
| SAPPHIRE 256MB RADEON 9550    | 2730.00 | 94.791667  | 79.130435  |
| GIGABYTE AGP GV-N59X128D      | 5886.00 | 204.375000 | 170.608696 |
| Maxtor 6Y120P0                | 2456.00 | 85.277778  | 71.188406  |
| Maxtor 6B200P0                | 3589.00 | 124.618056 | 104.028986 |
| Samsung SP0812C               | 2093.00 | 72.673611  | 60.666667  |
| Seagate Barracuda ST3160023A  | 3139.00 | 108.993056 | 90.985507  |
| Seagate ST3120026A            | 2468.00 | 85.694444  | 71.536232  |
| DDR-400 256MB Kingston        | 1085.00 | 37.673611  | 31.449275  |
| DDR-400 256MB Hynix Original  | 1179.00 | 40.937500  | 34.173913  |
| DDR-400 256MB PQI             | 899.00  | 31.215278  | 26.057971  |
| DDR-400 512MB Kingston        | 1932.00 | 67.083333  | 56.000000  |
| DDR-400 512MB PQI             | 1690.00 | 58.680556  | 48.985507  |
| DDR-400 512MB Hynix           | 1717.00 | 59.618056  | 49.768116  |
+-------------------------------+---------+------------+------------+
```

Следует отметить, что представление `goods`, кроме того, не содержит дополнительных столбцов, таких как описание каждой товарной позиции, внешний ключ для связи с таблицей `catalogs` и пр., способных затруднить работу с прайс-листом. Такие представления называются *вертикальными представлениями*. Вертикальные представления широко применяются для ограничения доступа пользователей к столбцам таблицы. Например, можно создать представление `list_user`, которое будет отображать фамилию и инициалы пользователя, скрывая поля с контактными телефонами и адресами электронной почты (листинг 30.6).

Листинг 30.6. Создание представления списка пользователей `list_user`

```
mysql> CREATE OR REPLACE VIEW list_user
    -> AS SELECT CONCAT(surname, " ",
    ->                  SUBSTRING(name,1,1), ".",
    ->                  SUBSTRING(patronymic,1,1), ".") AS name
    -> FROM users ORDER BY name;
mysql> SELECT * FROM list_user;
+----------------+
| name           |
+----------------+
| Иванов А.В.    |
| Корнеев А.А.   |
| Кузнецов М.В.  |
| Лосев С.И.     |
| Нехорошев А.Ю. |
| Симдянов И.В.  |
+----------------+
```

Таким образом, псевдотаблица list_user всегда содержит текущий список покупателей, отсортированный по алфавиту. Запросы к представлениям сами могут содержать условия WHERE и собственные сортировки, правильное составление запроса к исходным таблицам — это забота MySQL. В листинге 30.7 выводится список пользователей, отсортированный в обратном порядке, который получен с использованием представления list_user.

Листинг 30.7. Список пользователей, отсортированный в обратном порядке

```
mysql> SELECT * FROM list_user ORDER BY name DESC;
+----------------+
| name           |
+----------------+
| Симдянов И.В.  |
| Нехорошев А.Ю. |
| Лосев С.И.     |
| Кузнецов М.В.  |
| Корнеев А.А.   |
| Иванов А.В.    |
+----------------+
```

При сортировке результатов запроса к представлению list_user предложение ORDER BY самого представления игнорируется.

Наряду с вертикальными представлениями используются *горизонтальные представления*, которые ограничивают доступ пользователей к строкам таблиц, делая видимыми только те строки, с которыми они работают. Допустим, в учебном электронном магазине shop, содержащем пять каталогов, работают три менеджера, ответственность которых распределяется по каталогам следующим образом:

- Видеоадаптеры — первый менеджер;
- Жесткие диски — второй менеджер;
- Материнские платы — первый менеджер;
- Оперативная память — третий менеджер;
- Процессоры — третий менеджер.

Для того чтобы при работе с таблицей products менеджер видел только те товарные позиции, за которые он отвечает, можно создать три представления: first, second и third для первого, второго и третьего менеджеров соответственно (листинг 30.8). Затем учетные записи менеджеров следует лишить привилегии доступа к таблице products и разрешить просматривать только свои представления. Таким образом, с одной стороны, менеджерам не потребуется искать свои товарные позиции среди большого числа других товарных позиций, не имеющих к ним отношения, с другой стороны, они не смогут случайно внести изменения в товарные позиции других менеджеров.

Листинг 30.8. Создание представлений `first`, `second` и `third`

```
mysql> CREATE VIEW first
    -> AS SELECT * FROM products
    ->    WHERE id_catalog IN (SELECT id_catalog
    ->                         FROM catalogs
    ->                         WHERE name = 'Видеоадаптеры' OR
    ->                               name = 'Материнские платы')
    ->    ORDER BY name;
mysql> SELECT name, price, count FROM first;
+----------------------------+---------+-------+
| name                       | price   | count |
+----------------------------+---------+-------+
| ASUSTEK A9600XT/TD         | 5156.00 |     2 |
| Asustek P4C800-E Delux     | 5395.00 |     4 |
| Asustek P4P800-VM\L i865G  | 2518.00 |     6 |
| ASUSTEK V9520X             | 1602.00 |     6 |
| Epox EP-4PDA3I             | 2289.00 |     5 |
| GIGABYTE AGP GV-N59X128D   | 5886.00 |     6 |
| Gigabyte GA-8I848P-RS      | 1896.00 |     4 |
| Gigabyte GA-8IG1000        | 2420.00 |     2 |
| Gigabyte GA-8IPE1000G      | 2289.00 |     6 |
| SAPPHIRE 256MB RADEON 9550 | 2730.00 |     3 |
+----------------------------+---------+-------+
mysql> CREATE VIEW second
    -> AS SELECT * FROM products
    ->    WHERE id_catalog IN (SELECT id_catalog
    ->                         FROM catalogs
    ->                         WHERE name = 'Жесткие диски')
    ->    ORDER BY name;
mysql> SELECT name, price, count FROM second;
+------------------------------+---------+-------+
| name                         | price   | count |
+------------------------------+---------+-------+
| Maxtor 6B200P0               | 3589.00 |     4 |
| Maxtor 6Y120P0               | 2456.00 |     6 |
| Samsung SP0812C              | 2093.00 |     5 |
| Seagate Barracuda ST3160023A | 3139.00 |     3 |
| Seagate ST3120026A           | 2468.00 |     8 |
+------------------------------+---------+-------+
mysql> CREATE VIEW third
    -> AS SELECT * FROM products
    ->    WHERE id_catalog IN (SELECT id_catalog
    ->                         FROM catalogs
    ->                         WHERE name = 'Процессоры' OR
    ->                               name = 'Оперативная память')
    ->    ORDER BY name;
```

```
mysql> SELECT name, price, count FROM third;
+------------------------------+---------+-------+
| name                         | price   | count |
+------------------------------+---------+-------+
| Celeron 1.8                  | 1595.00 |    10 |
| Celeron 2.0GHz               | 1969.00 |     2 |
| Celeron 2.4GHz               | 2109.00 |     4 |
| Celeron D 315 2.26GHz        | 1880.00 |     6 |
| Celeron D 320 2.4GHz         | 1962.00 |     0 |
| Celeron D 325 2.53GHz        | 2747.00 |     6 |
| DDR-400 256MB Hynix Original | 1179.00 |    15 |
| DDR-400 256MB Kingston       | 1085.00 |    20 |
| DDR-400 256MB PQI            | 899.00  |    10 |
| DDR-400 512MB Hynix          | 1717.00 |     8 |
| DDR-400 512MB Kingston       | 1932.00 |    20 |
| DDR-400 512MB PQI            | 1690.00 |    12 |
| Intel Pentium 4 3.0GHz       | 6147.00 |     1 |
| Intel Pentium 4 3.0GHz       | 5673.00 |    12 |
| Intel Pentium 4 3.2GHz       | 7259.00 |     5 |
+------------------------------+---------+-------+
```

Использование вложенных запросов при формировании представлений не совсем рационально, т. к. на представления `first`, `second` и `third` могут накладываться внешние условия. Это приведет к тому, что скорость выполнения запроса будет невелика. При формировании запросов лучше поступиться их читабельностью и вычислить заранее первичные ключи каталогов, возвращаемые вложенными запросами (листинг 30.9). Это позволит не выполнять многократно дополнительный вложенный запрос для каждой строки.

Листинг 30.9. Представления, оптимизированные по скорости

```
mysql> CREATE VIEW first
    -> AS SELECT * FROM products
    ->    WHERE id_catalog IN (2, 3)
    ->    ORDER BY name;
mysql> CREATE VIEW second
    -> AS SELECT * FROM products
    ->    WHERE id_catalog IN (4)
    ->    ORDER BY name;
mysql> CREATE VIEW third
    -> AS SELECT * FROM products
    ->    WHERE id_catalog IN (1, 5)
    ->    ORDER BY name;
```

В реальной практике могут встречаться смешанные представления, которые ограничивают таблицу и по горизонтали, и по вертикали. Термины *горизонтальное* и *вер-*

тикальное представления являются условными и предназначены для того, чтобы лучше понять, как из исходной таблицы формируется представление.

Предложение ALGORITHM определяет алгоритм формирования конечного запроса с участием представления и может принимать три значения.

- MERGE — при использовании данного алгоритма запрос объединяется с представлением таким образом, что представление заменяет собой соответствующие части в запросе.
- TEMPTABLE — результирующая таблица представления помещается во временную таблицу, которая затем используется в конечном запросе.
- UNDEFINED — в данном случае СУБД MySQL самостоятельно пытается выбрать алгоритм, предпочитая использовать подход MERGE и прибегая к алгоритму TEMPTABLE (создание временной таблицы) только в случае необходимости, т. к. метод MERGE более эффективен.

Замечание

Если ни одно из значений ALGORITHM не указано, по умолчанию назначается UNDEFINED.

Листинг 30.10. Использование ключевого слова ALGORITHM

```
mysql> CREATE ALGORITHM = TEMPTABLE VIEW cat AS SELECT * FROM catalogs;
```

Запрос в листинге 30.10 требует от MySQL при каждом обращении к представлению cat создавать временную таблицу. При создании представления с помощью запроса, представленного в листинге 30.11, временная таблица для результирующей таблицы SELECT-запроса не будет создаваться никогда.

Листинг 30.11. Использование ключевого слова MERGE

```
mysql> CREATE ALGORITHM = MERGE VIEW cat AS SELECT * FROM catalogs;
```

Наиболее удобно использовать представления для формирования сгруппированных таблиц. При работе с такими таблицами СУБД MySQL самостоятельно формирует временную таблицу. Создадим представление price с общей стоимостью товарных позиций в каждом из каталогов (листинг 30.12).

Листинг 30.12. Создание представления price

```
mysql> CREATE VIEW price
    -> AS SELECT id_catalog, SUM(price*count) AS price
    ->    FROM products
    ->    GROUP BY id_catalog
    ->    ORDER BY price;
```

```
mysql> SELECT * FROM price;
+------------+-----------+
| id_catalog | price     |
+------------+-----------+
|          3 | 63430.00  |
|          4 | 68718.00  |
|          2 | 74291.00  |
|          5 | 121031.00 |
|          1 | 182554.00 |
+------------+-----------+
mysql> SELECT catalogs.name, price.price
    -> FROM price, catalogs
    -> WHERE price.id_catalog = catalogs.id_catalog
    -> ORDER BY catalogs.name;
+--------------------+-----------+
| name               | price     |
+--------------------+-----------+
| Видеоадаптеры      | 63430.00  |
| Жесткие диски      | 68718.00  |
| Материнские платы  | 74291.00  |
| Оперативная память | 121031.00 |
| Процессоры         | 182554.00 |
+--------------------+-----------+
mysql> SELECT MIN(price), MAX(price), SUM(price) FROM price;
+------------+------------+------------+
| MIN(price) | MAX(price) | SUM(price) |
+------------+------------+------------+
| 63430.00   | 182554.00  | 510024.00  |
+------------+------------+------------+
```

К сгруппированному представлению `price` нельзя применить операторы `UPDATE` и `DELETE`, т. к. суммарная стоимость товара для каждого из каталогов хранится во временной таблице, а сами операции обновления и удаления суммарной стоимости, учитывая структуру базы данных `shop`, не имеют смысла. Использование операторов `UPDATE` и `DELETE` совместно с представлениями не допускается, если в `SELECT`-запрос, лежащий в основе представления, входят следующие предложения:

- агрегатные функции (`SUM()`, `MIN()`, `MAX()`, `COUNT()` и др);
- `DISTINCT`;
- `GROUP BY`;
- `HAVING`;
- `UNION` или `UNION ALL`;
- вложенные запросы в списке `SELECT`;
- перекрестное, левое или правое объединение;

- представления в предложении FROM, к которым нельзя применять операторы UPDATE и DELETE;
- вложенные запросы в предложении WHERE, которые ссылаются на таблицы из предложения FROM;
- простые литеральные значения, такие как цифра 5 или строка 'Hello' в списке столбцов на выборку;
- ALGORITHM = TEMPTABLE (при явном использовании временных таблиц).

Необязательное ключевое слово OR REPLACE позволяет заменить представление, если представление с таким именем уже существует. Если это ключевое слово не используется, попытка создания представления поверх существующего приводит к ошибке 1050: "Таблица 'cat' уже существует" (листинг 30.13).

Листинг 30.13. Использование ключевого слова

```
mysql> CREATE VIEW cat AS SELECT * FROM catalogs;
ERROR 1050: Table 'cat' already exists
mysql> CREATE OR REPLACE VIEW cat AS SELECT * FROM catalogs;
Query OK, 0 rows affected (0.02 sec)
```

Замечание

Использование оператора CREATE VIEW требует привилегии CREATE VIEW, а также привилегий SELECT для каждого столбца, используемого в представлении. При применении ключевого слова OR REPLACE также требуется привилегия DELETE для представления.

При создании представления при помощи оператора CREATE VIEW представление сохраняется в текущей базе данных. Однако представление можно создать и в произвольной базе данных, указав имя базы данных при помощи расширенного имени таблицы dbase.tbl (листинг 30.14).

Листинг 30.14. Создание представления в базе данных test

```
mysql> CREATE VIEW test.cat AS SELECT * FROM catalogs;
mysql> USE test;
Database changed
mysql> SELECT * FROM cat;
+------------+--------------------+
| id_catalog | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 | Оперативная память |
+------------+--------------------+
```

Теперь получить доступ к таблице catalogs можно и в базе данных test, не прибегая к расширенному имени — достаточно обратиться к представлению cat. Таблицы и представления разделяют пространство имен базы данных, поэтому имена представлений и таблиц не могут совпадать и должны быть уникальными в пределах одной базы данных.

В качестве SELECT-запроса к представлению может выступать не только запрос к таблице, но и запрос к системной функции MySQL (листинг 30.15).

Листинг 30.15. Представление функции NOW()

```
mysql> CREATE VIEW time AS SELECT NOW() AS time;
mysql> SELECT * FROM time;
+---------------------+
| time                |
+---------------------+
| 2005-07-14 18:15:44 |
+---------------------+
```

Запрос к представлению time будет выдавать текущее время, а не время создания представления. Несмотря на то, что для создания представлений в качестве SELECT-запроса могут выступать практически любые запросы, имеется несколько ограничений.

- SELECT-запрос не может содержать подзапрос в предложении FROM.
- SELECT-запрос не может ссылаться на системную или пользовательскую переменную.
- Любые таблицы или представления, на которые в свою очередь ссылается представление, должны существовать физически.
- Внутри хранимых процедур представление не может ссылаться на параметры процедуры или локальные переменные процедуры.
- Представления не могут ссылаться на временные таблицы и сами представления не могут быть объявлены временными при помощи ключевого слова TEMPORARY.
- Триггер нельзя ассоциировать с представлением.

Предложение WITH CHECK OPTION добавляется для представлений, к которым могут быть применены операторы INSERT и UPDATE. В этом случае происходит проверка, чтобы вставляемые данные удовлетворяли WHERE-условию SELECT-запроса, лежащего в основе представления.

Замечание

Предложение WITH CHECK OPTION было добавлено в СУБД MySQL, начиная с версии 5.0.2.

Ключевые слова LOCAL и CASCADED в предложении WITH CHECK OPTION определяют область видимости ограничения применительно к другим представлениям. Ключевое слово LOCAL сообщает, что ограничение WITH CHECK OPTION распространяется только на текущее представление, а WHERE-ограничения представлений, на которые ссылается текущее представление, не влияют на процесс вставки новых данных. Тогда как ключевое слово CASCADED требует, чтобы проверка на соответствие вставляемых данных учитывала WHERE-условия и тех представлений, на которые ссылается текущее представление.

Замечание

Если ни одно из ключевых слов LOCAL или CASCADED не указано, считается, что выбран вариант LOCAL.

Для демонстрации эффекта предложения WITH CHECK OPTION создадим таблицу tbl с одним-единственным целочисленным столбцом a (листинг 13.16).

Листинг 13.16. Использование предложения WITH CHECK OPTION

```
mysql> CREATE TABLE tbl (a INT);
mysql> CREATE VIEW v1 AS SELECT * FROM tbl WHERE a < 2
    -> WITH CHECK OPTION;
mysql> CREATE VIEW v2 AS SELECT * FROM v1 WHERE a > 0
    -> WITH LOCAL CHECK OPTION;
mysql> CREATE VIEW v3 AS SELECT * FROM v1 WHERE a > 0
    -> WITH CASCADED CHECK OPTION;
```

Далее в листинге 13.16 создается представление v1, ограничивающее выборку значениями меньше двух. На представление v1 ссылаются два представления: v2 и v3, которые ограничивают выборку снизу значениями больше нуля. Представление v2 ограничивает вставку новых значений при помощи предложения WITH LOCAL CHECK OPTION, и, следовательно, не учитывает ограничение a < 2 из родительского представления. Представление v3 ограничивает вставку новых данных при помощи предложения WITH CASCADED CHECK OPTION, и, следовательно, реально вставка новых значений ограничена условиями 0 < a < 2 (листинг 13.17).

Листинг 13.17. Вставка новых значений в представления v1, v2 и v3

```
mysql> INSERT INTO v1 VALUES (-1);
Query OK, 1 row affected (0.03 sec)
mysql> INSERT INTO v1 VALUES (2);
ERROR 1369: CHECK OPTION failed 'test.v1'
mysql> INSERT INTO v2 VALUES (-1);
ERROR 1369: CHECK OPTION failed 'test.v2'
mysql> INSERT INTO v2 VALUES (2);
Query OK, 1 row affected (0.05 sec)
```

```
mysql> INSERT INTO v3 VALUES (-1);
ERROR 1369: CHECK OPTION failed 'test.v3'
mysql> INSERT INTO v3 VALUES (2);
ERROR 1369: CHECK OPTION failed 'test.v3'
```

30.2. Удаление представлений

Для удаления представлений предназначен оператор `DROP VIEW`, который имеет следующий синтаксис:

```
DROP VIEW [IF EXISTS]
    view_name [, view_name] ...
```

Оператор `DROP VIEW` позволяет уничтожить представление или сразу несколько представлений по их именам (листинг 30.18).

Листинг 30.18. Использование оператора `DROP VIEW`

```
mysql> DROP VIEW cat, tbl1, tbl2;
mysql> SHOW TABLES;
+----------------+
| Tables_in_test |
+----------------+
| catalogs       |
| forums         |
| goods          |
| products       |
+----------------+
```

Замечание

Применение по отношению к представлению оператора `DROP TABLE` приводит к возникновению ошибки 1051: `Unknown table 'cat'` (Неизвестная таблица '`cat`').

Необязательное ключевое слово `IF EXISTS` позволяет избежать ошибки при использовании оператора `DROP VIEW` к уже удаленному представлению (листинг 30.19). Это особенно удобно при пакетном выполнении группы операторов, когда эта незначительная ошибка может привести к остановке выполнения всех оставшихся операторов в пакетном файле.

Листинг 30.19. Использование ключевого слова `IF EXISTS`

```
mysql> DROP VIEW cat;
ERROR 1051: Unknown table 'test.cat'
mysql> DROP VIEW IF EXISTS cat;
Query OK, 0 rows affected (0.00 sec)
```

Если в прикладной программе все же требуется выяснить статус завершения операции удаления представления, удобнее воспользоваться оператором SHOW WARNINGS (листинг 30.20).

Листинг 30.20. Возвращение статуса удаления представления

```
mysql> SHOW WARNINGS;
+-------+------+-------------------------+
| Level | Code | Message                 |
+-------+------+-------------------------+
| Note  | 1051 | Unknown table 'test.cat' |
+-------+------+-------------------------+
```

30.3. Редактирование представлений

Для редактирования представлений предназначен оператор ALTER VIEW, который имеет следующий синтаксис:

```
ALTER [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    VIEW view_name [(column_list)]
    AS select_statement
    [WITH [CASCADED | LOCAL] CHECK OPTION]
```

Оператор ALTER TABLE практически полностью копирует структуру оператора CREATE VIEW и позволяет переопределить структуру. На самом деле эффект операторов ALTER VIEW и CREATE OR REPLACE VIEW один и тот же. В листинге 30.21 оба запроса являются эквивалентными.

Листинг 30.21. Использование оператора ALTER VIEW

```
mysql> ALTER VIEW cat AS SELECT * FROM catalogs;
mysql> CREATE OR REPLACE VIEW cat AS SELECT * FROM catalogs;
```

Замечание

Использование оператора ALTER VIEW требует привилегии CREATE VIEW и DELETE, а также привилегий SELECT для каждого столбца, используемого в представлении.

30.4. Оператор *SHOW CREATE VIEW*

Оператор SHOW CREATE VIEW позволяет просмотреть структуру оператора CREATE VIEW, при помощи которого было создано представление. Следует отметить, что оператор возвращает полную форму оператора CREATE VIEW, включая все опции и расширенные имена (листинг 30.22). Оператор возвращает содержимое файла с расширением frm, соответствующего представлению.

Листинг 30.22. Использование оператора `SHOW CREATE VIEW`

```
mysql> CREATE OR REPLACE VIEW cat AS SELECT * FROM catalogs;
mysql> SHOW CREATE VIEW cat\G;
*************************** 1. row ***************************
       View: cat
Create View: CREATE ALGORITHM=UNDEFINED VIEW 'test'.'cat' AS select
'test'.'catalogs'.'id_catalog' AS 'id_catalog','test'.'catalogs'.'name' AS
'name' from 'test'.'catalogs'
```

ГЛАВА 31

Информационная схема

В *главе 30* были рассмотрены представления, позволяющие создать единый интерфейс для Web-приложений. Реальная структура базы данных может изменяться, но при помощи представлений можно воссоздавать интерфейс к базе данных, который ожидает внешнее приложение.

С другой стороны, каждая база данных имеет системную базу данных, в которой хранится самая разнообразная информация, начиная с привилегий и заканчивая хранимыми процедурами. Недостатком системной базы данных является ее непереносимость — каждая СУБД организует системную базу данных по-своему.

Замечание

В СУБД MySQL системная база данных называется `mysql`.

Устранить недостаток несовместимости призван стандартный набор представлений системной таблицы, который называют *информационной схемой*. При использовании запросов к информационной схеме можно не опасаться, что они будут несовместимыми с другой базой данных.

Замечание

Представления информационной схемы поддерживаются СУБД MySQL, начиная с версии 5.0.2. В более ранних версиях для просмотра информации о СУБД и базах данных необходимо использовать оператор `SHOW`, который подробно рассматривается в *главе 27*. В то время как оператор `SHOW` поддерживается только СУБД MySQL, информационная схема является стандартной для всех реляционных СУБД.

Информационная схема оформлена в виде специального каталога `INFORMATION_SCHEMA`. В отличие от системного каталога, в СУБД MySQL отсутствует физический каталог, а также файлы для базы данных `INFORMATION_SCHEMA` и ее таблиц — база данных полностью располагается в оперативной памяти. Это означает, что невозможно выбрать базу данных `INFORMATION_SCHEMA` при помощи оператора `USE`, как и выполнить по отношению к таблицам этой базы данных запросы с участием операторов `INSERT`, `UPDATE` и `DELETE`. Допускается использование только оператора `SELECT`. Кроме

того, база данных информационной схемы information_schema присутствует в результирующей таблице оператора SHOW DATABASES (листинг 31.1).

Замечание

Доступ к базе данных INFORMATION_SCHEMA не требует специальных привилегий. Каждый пользователь снабжается привилегией SELECT на каждую таблицу базы данных INFORMATION_SCHEMA.

Листинг 31.1. База данных information_schema

```
mysql> SHOW DATABASES//
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| qwert              |
| shop               |
| test               |
| wet                |
+--------------------+
```

Заключение

Чтобы ни говорилось, чтение книги — это всегда в том или ином виде слушание авторского монолога. Это, в общем-то, не очень хорошо, поскольку быстрая обратная связь в равной мере необходима как читателям, так и авторам. Для того чтобы авторский монолог превратить в полноценный диалог с читателем, мы создали на своем сайте форум, на котором можно задать свои вопросы. Адрес форума **http://www.softtime.ru/forum/index.php?id_forum=3**. Авторы искренне надеются, что после того, как вы закроете последнюю страницу этой книги, мы с вами не расстанемся и снова встретимся на нашем форуме.

Успехов вам и всего доброго!

ПРИЛОЖЕНИЕ

Описание компакт-диска

Папки	Описание	Главы
\base	Учебная база данных	Все
\base\base.sql	Текстовый дамп учебной базы данных `shop`	Все
\base\shop	Бинарные файлы учебной базы данных `shop`	Все
\binaries	Бинарные дистрибутивы MySQL для Windows и Linux	2
\binaries\mysql40	Инсталлятор MySQL 4.0	2
\binaries\mysql41	Инсталлятор MySQL 4.1	2
\binaries\mysql50	Инсталлятор MySQL 5.0	2
\clients	Графические клиенты для СУБД MySQL	2
\docs	Документация на русском и английском языках	2
\source	Исходные коды MySQL	2
\source\mysql40	Исходные коды MySQL 4.0	2
\source\mysql41	Исходные коды MySQL 4.1	2
\source\mysql50	Исходные коды MySQL 5.0	2

Предметный указатель

A

ACID 408
Advanced Encryption Standard (AES) 275

C

Cron 440

I

IP-адрес 341, 417

M

MAC-адрес 344
Message-Digest Algorithm (MD5) 280

R

RPM-пакет 40

S

Secure Hash Algorithm (SHA1) 281
SSL 432
Structured Query Language (SQL) 20

T

Triple-DES (3DES) 278

U

Universal Unique Identifier (UUID) 343

W

Web-браузер 18
Web-сервер 18

А

Административные команды 449

Б

База данных:
◊ information_schema 54, 528
◊ mysql 54, 281, 418, 491, 527
◊ test 54
◊ восстановление 437
◊ гибридная 10
◊ дамп 50
◊ денормализация 14
◊ иерархическая 6
◊ изменение 56
◊ имя 53
◊ нормализация схемы 12
◊ объектно-ориентированная 9
◊ привилегии 429
◊ реляционная 8—11, 20, 21
◊ сетевая 7
◊ система управления 5, 19
◊ системная 527
◊ создание 53
◊ текущая 335
◊ удаление 54
◊ целостность 407
Библиотека zlib 249
Блокировка:
◊ получение 340
◊ проверка 342
◊ снятие 340, 343

В

Версия сервера 339
Внешняя ссылка 388
Встроенный сервер 22

Д

Дамп 50, 162
Деление на ноль 174
Доменное имя 418

З

Запись:
◊ максимальное число 164
◊ максимальное число в таблице 162
◊ минимальное число 165
◊ средняя длина 162
Запрос:
◊ вложенный 23, 349, 375, 520, 522
◊ выборка данных 99
◊ двухтабличный 354
◊ коррелированный 389
◊ левое объединение 364, 520
◊ многотабличный 353, 377
◊ перекрестное объединение 354, 520
◊ подчиненный 375
◊ правое объединение 364, 520
◊ самообъединение таблицы 358
◊ строчный 389
◊ трехтабличный 361
◊ четырехтабличный 362

И

Именованное условие 499
Индекс:
◊ FULLTEXT 301
◊ PRIMARY KEY 129
◊ UNIQUE 129, 136
◊ анализ распределения 440
◊ внешний ключ 12, 83, 156, 398
◊ вторичный ключ 401
◊ изменение 84
◊ логические ключи 11
◊ максимальное число 144
◊ обычный 81, 136
◊ первичный ключ 11, 78, 79, 88,156, 398
◊ повреждение 439
◊ полнотекстовый 78
◊ сжатие 165
◊ создание 84, 135
◊ суррогатные ключи 11
◊ удаление 84, 136
◊ уникальный 81, 156

Информационная схема 24, 527
Информационное сообщение 438

К

Каталог данных 50, 53, 143, 152, 164
Клиент-серверная архитектура 16
Кодировка 23, 48, 56, 156, 170, 300
Команда DELIMITER 45
Комментарий 70, 459
Компьютерная сеть 15, 18, 152, 418
Конструкция:
◊ ADD FULLTEXT 135
◊ ADD INDEX 135
◊ ADD PRIMARY KEY 135
◊ ADD UNIQUE 135
◊ AFTER 132, 134
◊ AGAINST 304, 307
◊ ALL 113, 118, 384
◊ ANY 382, 384
◊ AS 110, 262, 315, 357, 372, 373, 391, 394
◊ AUTO_INCREMENT 80, 81, 87, 89—91, 93, 139, 145, 147, 148, 155, 156, 162, 336
◊ AVG_ROW_LENGTH 162
◊ BETWEEN 102, 181
◊ BINARY 168, 176, 284, 293
◊ CASCADE 401
◊ CHANGE 134
◊ CHANGED 439
◊ CHARACTER SET 156, 157, 162, 163
◊ CHECKSUM 163, 442
◊ COLLATE 156, 159
◊ COLUMN 132
◊ COMMENT 153, 156, 157, 159, 163
◊ DATA DIRECTORY 164
◊ DEFAULT 70, 90, 155, 156
◊ DEFAULT CHARACTER SET 56, 162
◊ DELAY_KEY_WRITE 164
◊ DISTINCT 112—114, 118, 313, 320, 321, 322, 324, 326, 520
◊ DISTINCTROW 112
◊ DROP INDEX 136
◊ DROP PRIMARY KEY 136
◊ ELSE 329
◊ ENCLOSED BY 97, 120
◊ ENGINE 144, 146—150, 152, 153, 161
◊ ESCAPED BY 97, 120
◊ EXISTS 384, 386
◊ EXTENDED 439, 442, 444
◊ FAST 439
◊ FIELDS 119, 120
◊ FIRST 132, 134, 146
◊ FOREIGN KEY 156, 401, 402
◊ FROM 99, 109, 119, 354, 363, 389, 391, 521
◊ FULLTEXT 160
◊ GROUP BY 113—116, 262, 313, 320, 357, 358, 368, 520
◊ HAVING 115, 116, 368, 380, 520
◊ IDENTIFIED BY 421, 429
◊ IF EXISTS 493, 524
◊ IF NOT EXISTS 55, 154
◊ IGNORE 88, 95, 129, 396, 397
◊ IN 103, 184, 381, 390, 455, 460
◊ IN BOOLEAN MODE 307
◊ INDEX 81, 82, 156
◊ INDEX DIRECTORY 164
◊ INOUT 455, 462
◊ INSERT_METHOD 146, 164
◊ INTERVAL 208
◊ INTO OUTFILE 119
◊ INTO TABLE 95
◊ IS NOT NULL 180
◊ IS NULL 179
◊ KEY 81, 82, 156
◊ LAST 146
◊ LIKE 194, 488, 489
◊ LIMIT 107, 108, 124, 127, 335, 340, 371, 372
◊ LINES 96, 119, 120
◊ LOCAL 94
◊ LOW_PRIORITY 414
◊ MATCH 303
◊ MATCH (…) AGAINST (…) 303, 306
◊ MAX_CONNECTIONS_PER_HOUR 431
◊ MAX_QUERIES_PER_HOUR 431

Конструкция *(прод.)*:
◊ MAX_ROWS 164, 165
◊ MAX_UPDATES_PER_HOUR 431
◊ MAX_USER_CONNECTIONS 431
◊ MEDIUM 439
◊ MIN_ROWS 165
◊ MODIFY 133—135
◊ NO 146
◊ NO ACTION 401
◊ NOT BETWEEN 103, 182
◊ NOT EXISTS 384, 386
◊ NOT IN 103, 185, 382
◊ NOT NULL 70, 72, 75, 155, 157
◊ NULL 70, 74, 81, 155, 156
◊ ON 363, 368, 380, 428
◊ ORDER BY 105, 106, 110, 111, 120, 124, 127, 138, 139, 158, 202, 315, 322, 328, 358, 371, 372
◊ ORDER BY ... ASC 107
◊ ORDER BY ... DESC 107
◊ OUT 455, 461
◊ PACK_KEYS 165
◊ PASSWORD 165, 421
◊ PRIMARY KEY 78, 79, 156
◊ QUICK 439, 442, 444
◊ READ 413
◊ REFERENCES 401
◊ REGEXP 194
◊ RENAME 137
◊ REPLACE 95, 396, 397
◊ REQUIRE 432
◊ REQUIRE CIPHER 433
◊ REQUIRE ISSUER 432
◊ REQUIRE SSL 432
◊ REQUIRE SUBJECT 432
◊ REQUIRE X509 432
◊ RESTRICT 401
◊ ROW 389
◊ ROW_FORMAT 165
◊ ROW_FORMAT=COMPRESSED 165
◊ ROW_FORMAT=DEFAULT 165
◊ ROW_FORMAT=FIXED 165
◊ SEPARATOR 321
◊ SET 93, 125
◊ SET DEFAULT 401
◊ SET NULL 401
◊ SOME 382
◊ SQL_CALC_FOUND_ROWS 335
◊ STARTING BY 96, 119
◊ TEMPORARY 154
◊ TERMINATED BY 96, 97, 119, 120
◊ THEN 329
◊ TO 445
◊ TYPE 144, 146, 161
◊ UNION 146, 166
◊ UNIQUE 81, 84, 156
◊ UNIQUE INDEX 81
◊ UNIQUE KEY 81
◊ UNSIGNED 61, 62
◊ USE_FRM 444
◊ USEAGE 428
◊ USING 365, 368
◊ VALUES 87, 93, 129, 398
◊ WHEN 329
◊ WHERE 101, 110, 114—116, 124, 125, 175, 283, 303, 315, 319, 357, 358, 363, 380, 397, 521
◊ WITH GRANT OPTION 431
◊ WRITE 413
◊ ZEROFILL 61
Коррелирующая ссылка 389
Коэффициент релевантности 304
Курсоры 24, 501

Л

Лицензия GPL 22
Логический режим 307
◊ " 311
◊ * 311
◊ + 307
◊ < 309
◊ > 309

М

Мэйнфрейм 16

Н

Нарушение ссылочной целостности 399

О

Обработчик ошибок 24, 495
◊ CONTINUE 496
◊ EXIT 496
◊ NOT FOUND 496
◊ SQLEXCEPTION 496
◊ SQLSTATE 496
◊ SQLWARNING 496
◊ UNDO 496
Общеупотребительные слова 301
Операнд 171
Оператор:
◊ ! 186, 193
◊ != 177, 384
◊ % 174
◊ & 188
◊ && 187, 474
◊ * 92, 171, 173
◊ / 171, 173
◊ := 194, 345, 347
◊ ^ 190
◊ | 189
◊ || 187, 474
◊ ~ 191
◊ + 171, 193, 211
◊ < 177, 474
◊ << 191
◊ <= 178, 474
◊ <=> 176
◊ <> 177, 384, 474
◊ = 175, 347, 384, 474
◊ > 101, 178, 474
◊ >= 179, 474
◊ >> 192
◊ ALTER DATABASE 56, 411
◊ ALTER FUNCTION 494
◊ ALTER PROCEDURE 494
◊ ALTER TABLE 131, 134, 135, 137, 139, 154, 166, 303, 370, 401, 411, 442
◊ ALTER VIEW 525
◊ ANALYZE TABLE 440
◊ AND 102, 187, 299, 306
◊ BACKUP TABLE 445
◊ BEGIN 411
◊ BEGIN … END 456, 457, 467, 468, 474, 480, 506
◊ BEGIN WORK 411
◊ CALL 458, 472
◊ CASE 477, 506
◊ CHECK TABLE 437
◊ CHECKSUM TABLE 440
◊ CLOSE 501, 502
◊ COMMIT 410, 412
◊ CREATE DATABASE 50, 53, 154, 411, 449
◊ CREATE FUNCTION 454, 469, 471
◊ CREATE INDEX 84, 85, 303, 411
◊ CREATE PROCEDURE 454, 469, 471
◊ CREATE TABLE 57, 72, 74, 82, 143, 144, 146—150, 157, 161, 301, 348, 353, 375, 393, 395, 396, 401, 411, 442
◊ CREATE TEMPORARY TABLE 348
◊ CREATE TRIGGER 505, 506
◊ CREATE USER 44, 420, 421
◊ CREATE VIEW 512, 521, 525
◊ DECLARE 466, 506
◊ DECLARE … CONDITION FOR 496, 499
◊ DECLARE … HANDLER FOR 495
◊ DECLARE CURSOR 501
◊ DELETE 123, 124, 127, 151, 154, 301, 372, 375, 399, 404, 427, 434, 443, 459, 505, 520, 521
◊ DELIMITER 457
◊ DESCRIBE 59, 71, 78, 81, 82
◊ DIV 171, 174
◊ DROP DATABASE 54, 411, 412
◊ DROP FUNCTION 494
◊ DROP INDEX 84, 411
◊ DROP PROCEDURE 493
◊ DROP TABLE 58, 132, 154, 349, 411, 412
◊ DROP TRIGGER 509
◊ DROP USER 420, 422, 434
◊ DROP VIEW 524

Оператор *(прод.)*:
◊ ELSEIF 476
◊ FETCH 501, 502
◊ FLUSH PRIVILEGES 420
◊ GOTO 486
◊ GRANT 425
◊ IF 474, 476, 479, 506
◊ INSERT 23, 81, 88, 89, 93, 120, 129, 151, 154, 259, 301, 375, 396, 398, 400, 408, 427, 459, 505, 508, 522
◊ INSERT, многострочный 93
◊ INSERT, однострочный 87
◊ ITERATE 483, 485, 506
◊ JOIN 363, 366
◊ LABEL 486
◊ LEAVE 481, 483, 485, 506
◊ LEFT JOIN 364
◊ LEFT OUTER JOIN 365
◊ LIKE 283
◊ LOAD DATA 94, 96
◊ LOAD DATA INFILE 119
◊ LOAD MASTER DATA 412
◊ LOCK TABLE 412
◊ LOCK TABLES 412, 446
◊ LOOP 485, 486, 506
◊ MOD 175
◊ NOT 186, 306
◊ NOT LIKE 287
◊ NOT REGEXP 299
◊ NOT RLIKE 299
◊ OPEN 501, 502
◊ OPTIMIZE TABLE 443
◊ OR 102, 187, 299, 306
◊ REGEXP 288
◊ RENAME 412
◊ RENAME TABLE 137, 411
◊ RENAME USER 424
◊ REPAIR TABLE 443
◊ REPAIR TABLES 444
◊ REPEAT 484, 486, 506
◊ REPLACE 129, 151, 443, 459
◊ RESTORE TABLE 446
◊ RESTORE TABLES 447
◊ RETURN 455, 469
◊ RETURNS 469
◊ REVOKE 420, 425, 434
◊ RIGHT JOIN 364
◊ RIGHT OUTER JOIN 365
◊ RLIKE 288
◊ ROLLBACK 410, 411, 412
◊ ROLLBACK TO SAVEPOINT 412
◊ SAVEPOINT 412
◊ SELECT 91, 99—101, 113, 116, 117, 139, 151, 154, 303, 319, 335, 345, 349, 354, 355, 375, 379, 427, 459, 491, 492, 501, 520
◊ SELECT ... INTO OUTFILE 94, 119
◊ SELECT … INTO … FROM 463, 465, 500
◊ SET 337, 347, 351, 404, 409, 449, 462, 506
◊ SET PASSWORD 419
◊ SET TRANSACTION 411
◊ SHOW 527
◊ SHOW CHARACTER SET 158, 246, 449
◊ SHOW COLUMNS 449
◊ SHOW CREATE DATABASE 449
◊ SHOW CREATE FUNCTION 490
◊ SHOW CREATE PROCEDURE 490
◊ SHOW CREATE TABLE 156, 163
◊ SHOW CREATE VIEW 525
◊ SHOW DATABASES 54, 429, 449, 528
◊ SHOW FULL COLUMNS 156, 159, 160
◊ SHOW FUNCTION STATUS 489, 490
◊ SHOW GRANTS 423
◊ SHOW PRIVILEGES 429
◊ SHOW PROCEDURE STATUS 488, 489, 490
◊ SHOW TABLE STATUS 163
◊ SHOW TABLES 57, 430
◊ SHOW WARNINGS 525
◊ SOUNDS LIKE 288
◊ START TRANSACTION 411, 412
◊ TRUNCATE TABLE 124, 411, 412
◊ UNION 116, 117, 118, 520
◊ UNION ALL 118

◊ UNION DISTINCT 118
◊ UNLOCK TABLE 412
◊ UNLOCK TABLES 412
◊ UPDATE 23, 125, 128, 151, 154, 172, 259, 369, 371, 375, 404, 420, 427, 443, 459, 505, 520—522
◊ USE 57, 464
◊ WHILE 479, 486, 506
◊ XOR 188
◊ битовый 188
◊ логический 102, 186
◊ приоритет 193
Ошибка операции 438

П

Пакетная загрузка данных 94
Параметр:
◊ -? 48
◊ –A 51
◊ --all-databases 51
◊ -b 48
◊ –B 51
◊ -C 48
◊ --character-sets-dir 48
◊ --compress 48
◊ -D 48
◊ --database 48
◊ --databases 51
◊ --default-character-set 48
◊ --delimiter 48, 457
◊ --delimiter=name 45
◊ --des-key-file 278
◊ -E 48
◊ -h 48
◊ -H 48
◊ --help 48
◊ --host 48
◊ --html 48
◊ -I 48
◊ --no-beep 48
◊ -p 45, 48
◊ --password 48
◊ --port 48
◊ -u 43, 44, 48
◊ --user 48
◊ -V 48
◊ --version 48
◊ --vertical 48
◊ -X 48
◊ --xml 48
Переменная 376
◊ AUTOCOMMIT 409
◊ BIG_TABLES 351
◊ FOREIGN_KEY_CHECKS 404
◊ IDENTITY 337
◊ LAST_INSERT_ID 337
◊ SESSION_USER() 338
◊ SQL_AUTO_IS_NULL 338
◊ SYSTEM_USER() 338
◊ пользовательская 345, 449, 461, 522
◊ серверная 449, 522
Полнотекстовый поиск 144, 161, 301
Пользователь:
◊ root 43, 417
◊ анонимный 418
◊ изменение 424
◊ имя 43, 153, 334, 338
◊ максимальная длина 421
◊ пароль 43, 44, 153, 275
◊ создание 44
Порт 33, 153
Последовательность:
◊ % 284, 430
◊ * 100, 110, 356
◊ .* 373
◊ /* 459
◊ @ 345, 461
◊ @@ 351
◊ _ 285, 430
◊ ® 312
◊ \n 96, 119
◊ \r\n 96, 97, 119
◊ \t 97, 120
◊ бинарная 484
◊ двоичная 168
Постраничная навигация 335
Представление 15, 23, 511, 527
◊ ALGORITHM 519
◊ CASCADED 523

Представление *(прод.)*:
◊ LOCAL 523
◊ MERGE 519
◊ OR REPLACE 521
◊ TEMPTABLE 519
◊ UNDEFINED 519
◊ WITH CHECK OPTION 522
◊ вертикальное 515
◊ горизонтальное 516
◊ редактирование 525
◊ смешанное 518
◊ создание 512
◊ удаление 524
Предупреждение 438
Привилегия:
◊ ALTER ROUTINE 454
◊ CREATE ROUTINE 454
◊ CREATE VIEW 521, 525
◊ DELETE 521, 525
◊ EXECUTE 454
◊ SELECT 521, 525, 528
◊ SUPER 506
Протокол соединения 153

Р

Расширенное имя 58
Регулярные выражения 288
◊ $ 290
◊ * 295
◊ . 299
◊ ? 295
◊ [[:<:]] 291
◊ [[:>:]] 291
◊ [] 292
◊ \\ 291, 294
◊ ^ 289, 293
◊ {} 297
◊ | 291
◊ + 295
◊ e-mail 298
◊ \f 294
◊ \n 294
◊ \r 294
◊ \t 294
◊ число формата ###.## 297

С

Секретный ключ 275
Сервис 35, 39
Соединение с удаленным хостом 48
Стандарт языка SQL 21
Статус операции 438
Столбец:
◊ добавление 131
◊ изменение 133
◊ имя 57
◊ кодировка 156, 157, 163
◊ полное имя 355
◊ привилегии 430
◊ псевдоним 110, 358
◊ сортировка 156
◊ удаление 132
◊ уникальные значения 112
Структурированный язык запросов 6, 20

Т

Таблица:
◊ ALTER TABLE 409
◊ ARCHIVE 23, 150
◊ BDB 147, 148, 407, 409, 437, 440, 443
◊ CSV 23, 151
◊ DUAL 109
◊ EXAMPLE 23, 147, 148
◊ FEDERATED 24, 152, 153
◊ HEAP 146, 165, 407
◊ InnoDB 22, 32, 49, 85, 147, 149, 156, 400, 401, 404, 407, 409, 412, 440
◊ ISAM 143, 407
◊ MEMORY 85, 146, 147
◊ MERGE 144—146, 164, 166
◊ MRG_MYISAM 146
◊ MyISAM 22, 32, 49, 85, 143, 144, 152, 162, 164—166, 301, 404, 407, 409, 412, 437, 440, 443—446
◊ NDB Cluster 23, 150
◊ блокировка 412

◊ восстановление 443
◊ восстановление из резервной копии 446
◊ временная 154, 348, 522
◊ дефрагментация 443
◊ изменение структуры 131
◊ имя 57
◊ контрольная сумма 163, 440
◊ объединение 116
◊ повреждение 438
◊ привилегии 429
◊ проверка 437
◊ псевдоним 358
◊ резервное копирование 445
◊ результирующая 99, 100
◊ самообъединение 358
◊ связь 398
◊ создание 57
◊ структура 155
Текущая дата 207
Текущее время суток 207
Тип данных:
◊ BIGINT 61, 80, 92, 171, 174, 188, 316, 317, 341
◊ BINARY 65, 168
◊ BIT 61
◊ BLOB 65, 147, 155, 165, 250, 259, 278
◊ BOOLEAN 61
◊ CHAR 64, 165, 168, 250, 301
◊ DATE 66, 168, 207
◊ DATETIME 66, 124, 168, 207
◊ DEC 61
◊ DECIMAL 61
◊ DOUBLE 62
◊ ENUM 65
◊ FLOAT 62
◊ INT 61, 80, 92, 171
◊ INTEGER 61
◊ MEDIUMINT 61
◊ NULL 69
◊ NUMERIC 61
◊ PRECISION 62
◊ REAL 62
◊ SET 65, 255
◊ SIGNED INTEGER 168
◊ SMALLINT 61
◊ TEXT 65, 147, 155, 165, 250, 301
◊ TIME 66, 168, 207
◊ TIMESTAMP 23, 66, 68, 155, 175, 207
◊ TINYINT 61, 92
◊ UNSIGNED INT 171
◊ UNSIGNED INTEGER 168
◊ VARBINARY 65
◊ VARCHAR 64, 165, 250, 301
◊ YEAR 66, 207
◊ календарный 66
◊ приближенный 60
◊ приведение 167
◊ строковый 63, 88
◊ точечный 60
◊ числовой 60
Тонкий клиент 18
Транзакция 407
◊ атомарность 408
◊ длительность 408
◊ изоляция 408
◊ режим автоматического завершения 409
◊ целостность 408
Трехуровневая архитектура 18
Триггер 24, 505, 522
◊ AFTER 505
◊ BEFORE 505
◊ DELETE 506
◊ INSERT 506
◊ NEW 506
◊ OLD 506
◊ UPDATE 506
◊ создание 505
◊ удаление 509

У

Удаленный хост 48, 152, 153
Универсальный Уникальный Идентификатор 343
Уникальный идентификатор соединения 334

Утилита:
◊ myisampack 41
◊ mysql 41, 42, 333
◊ MySQL Administrator 42
◊ MySQL Command Line Client 37
◊ MySQL Control Center 42
◊ MySQL Query Browser 42
◊ MySQL Server Instance Configuration Wizard 29
◊ mysqladmin 41
◊ mysqlbinlog 41
◊ mysqlcheck 41
◊ mysqldump 41, 50, 120
◊ mysqlhotcopy 41
◊ mysqlimport 41
◊ mysqlshow 41
Учебная база данных 50, 58, 72
Учетная запись 417
◊ % 418
◊ ограничения 431
◊ пароль 419
◊ привилегии 419, 421, 425
◊ редактирование 424
◊ создание 420
◊ удаление 422
◊ удаление привилегий 434
◊ форма 417

Ф

Файл:
◊ ARM 150
◊ ARN 150
◊ ARZ 150
◊ csv 151
◊ db 148, 149
◊ frm 143, 144, 147—152, 165, 444, 445, 447, 525
◊ MRG 144
◊ MYD 143, 152, 164, 445, 447
◊ MYI 144, 164, 444, 445, 447
◊ db.opt 53
◊ DES 278
◊ my.ini 49
◊ сохранение результатов во внешний файл 119
◊ текстовый 94, 95
Функция:
◊ ABS() 195
◊ ACOS() 195
◊ ADDDATE() 207
◊ ADDTIME() 212
◊ AES_DECRYPT() 275
◊ AES_ENCRYPT() 275
◊ ASCII() 245
◊ ASIN() 195
◊ ATAN() 196
◊ ATAN2() 196
◊ AVG() 313
◊ BENCHMARK() 333
◊ BIN() 246
◊ BIT_AND() 316
◊ BIT_COUNT() 193
◊ BIT_LENGTH() 246
◊ BIT_OR() 317
◊ CASE 329
◊ CAST() 168, 169
◊ CEIL() 197
◊ CEILING() 196
◊ CHAR() 246
◊ CHAR_LENGTH() 247
◊ CHARACTER_LENGTH() 247
◊ CHARSET() 248
◊ COALESCE() 182
◊ COLLATION() 249
◊ COMPRESS() 249
◊ CONCAT() 251, 391, 484
◊ CONCAT_WS() 252
◊ CONNECTION_ID() 334
◊ CONV() 252
◊ CONVERT() 170
◊ CONVERT_TZ() 213
◊ COS() 197
◊ COT() 198
◊ COUNT() 110, 113, 114, 148, 318, 368
◊ CRC32() 198
◊ CURDATE() 213
◊ CURRENT_DATE 214

◊ CURRENT_DATE() 214
◊ CURRENT_TIME 214
◊ CURRENT_TIME() 214
◊ CURRENT_USER() 334
◊ CURTIME() 214
◊ DATABASE() 335
◊ DATE() 214
◊ DATE_ADD() 207
◊ DATE_FORMAT() 215, 223
◊ DATE_SUB() 233
◊ DATEDIFF() 215
◊ DAY() 218
◊ DAYNAME() 218
◊ DAYOFMONTH() 219
◊ DAYOFWEEK() 220
◊ DAYOFYEAR() 220
◊ DECODE() 277
◊ DEFAULT() 339
◊ DEGREES() 198
◊ DES_DECRYPT() 278
◊ DES_ENCRYPT() 278
◊ ELT() 253
◊ ENCODE() 277
◊ ENCRYPT() 279
◊ EXP() 199
◊ EXPORT_SET() 253
◊ EXTRACT() 220
◊ FIELD() 254
◊ FIND_IN_SET() 255
◊ FLOOR() 199
◊ FORMAT() 255
◊ FOUND_ROWS() 335
◊ FROM_DAYS() 221
◊ FROM_UNIXTIME() 222
◊ GET_FORMAT() 223
◊ GET_LOCK() 340
◊ GREATEST() 183
◊ GROUP_CONCAT() 321
◊ HEX() 256
◊ HOUR() 225
◊ IF() 330
◊ IFNULL() 331
◊ INET_ATON() 341
◊ INET_NTOA() 342
◊ INSERT() 256
◊ INSTR() 257
◊ INTERVAL() 185
◊ IS_FREE_LOCK() 342
◊ IS_USED_LOCK() 343
◊ LAST_DAY() 225
◊ LAST_INSERT_ID() 91, 336
◊ LCASE() 259
◊ LEAST() 183
◊ LEFT() 257
◊ LENGTH() 258
◊ LN() 199
◊ LOAD_FILE() 258
◊ LOCALTIME 229
◊ LOCALTIME() 229
◊ LOCALTIMESTAMP 229
◊ LOCALTIMESTAMP() 229
◊ LOCATE() 259
◊ LOG() 199
◊ LOG10() 200
◊ LOG2() 200
◊ LOWER() 259
◊ LPAD() 260
◊ LTRIM() 260
◊ MAKE_SET() 261
◊ MAKEDATE() 226
◊ MAKETIME() 227
◊ MAX() 110, 111, 324
◊ MD5() 280
◊ MICROSECOND() 227
◊ MID() 261
◊ MIN() 110, 111, 322
◊ MINUTE() 228
◊ MOD() 175, 200
◊ MONTH() 228
◊ MONTHNAME() 228
◊ NOW() 229
◊ NULLIF() 333
◊ OCT() 262
◊ OLD_PASSWORD() 281
◊ ORD() 263
◊ PASSWORD() 280, 419, 421
◊ PERIOD_ADD() 230
◊ PERIOD_DIFF() 230

Функция *(прод.)*:
◊ PI() 201
◊ POSITION() 263
◊ POW() 201
◊ POWER() 201
◊ QUARTER() 231
◊ QUOTE() 263
◊ RADIANS() 201
◊ RAND() 202
◊ RELEASE_LOCK() 340, 343
◊ REPEAT() 264
◊ REPLACE() 264
◊ REVERSE() 266
◊ RIGHT() 266
◊ ROUND() 203
◊ RPAD() 267
◊ RTRIM() 267
◊ SEC_TO_TIME() 231
◊ SECOND() 231
◊ SHA() 282
◊ SHA1() 281
◊ SIGN() 204
◊ SIN() 204
◊ SOUNDEX() 268
◊ SQRT() 204
◊ STD() 324
◊ STDDEV() 324
◊ STDDEV_POP() 324
◊ STDDEV_SAMP() 325
◊ STR_TO_DATE() 223, 232
◊ STRCMP() 300
◊ SUBDATE() 232
◊ SUBSTRING() 261, 268, 391
◊ SUBSTRING_INDEX() 269
◊ SUBTIME() 233
◊ SUM() 326
◊ SYSDATE() 229
◊ TAN() 205
◊ TIME() 234
◊ TIME_FORMAT() 237
◊ TIME_TO_SEC() 237
◊ TIMEDIFF() 234
◊ TIMESTAMP() 235
◊ TIMESTAMPADD() 235
◊ TIMESTAMPDIFF() 236
◊ TO_DAYS() 222, 238, 360
◊ TRIM 270
◊ TRUNCATE() 205
◊ UCASE() 273
◊ UNCOMPRESS() 271
◊ UNCOMPRESSED_LENGTH() 271
◊ UNHEX() 272
◊ UNIX_TIMESTAMP() 222, 239
◊ UPPER() 272, 379
◊ USER() 338
◊ UTC_DATE 240
◊ UTC_DATE() 239
◊ UTC_TIME() 240
◊ UTC_TIMESTAMP 241
◊ UTC_TIMESTAMP() 240
◊ UUID() 343
◊ VAR_POP() 327
◊ VAR_SAMP() 327
◊ VARIANCE() 327
◊ VERSION() 109, 339
◊ WEEK() 241
◊ WEEKDAY() 242
◊ WEEKOFYEAR() 242
◊ YEAR() 243
◊ YEARWEEK() 243
◊ агрегатная 313, 520
◊ временная 207
◊ встроенная 108
◊ информационная 333
◊ использование 108
◊ конструкции GROUP BY 313
◊ miscellaneous 339
◊ математическая 194
◊ строковая 245
◊ суммирующая 313
◊ хранимая 23, 469
◊ шифрования 275

Х

Хост локальный 417
Хранимая процедура 21, 23, 453, 522
◊ COMMENT 473, 495

◊ DETERMINISTIC 472
◊ LANGUAGE SQL 471
◊ NOT DETERMINISTIC 472
◊ SQL SECURITY 472, 494
◊ групповые характеристики 471
◊ изменение 494
◊ имя 457, 489, 491
◊ метка 456
◊ параметр 455, 459
◊ создание 454
◊ тело 456
◊ тип 491
◊ удаление 493

Ц

Централизованная архитектура 16

Ш

Шифрование:
◊ необратимое 275, 279
◊ симметричное 275

Я

Язык SQL 16, 20, 21