Для тех, кто хочет стать профессионалом

Р. Р. Аклимов

БАЗЫ ДАННЫХ

SQL, MySQL, MS SQL и Postgre/PL/pgSQL

Проектирование
и практическая реализация

+ виртуальный диск с примерами

Аклимов Р. Р.

Базы данных на SQL, MySQL, MS SQL и Postgre/PL/pgSQL

Проектирование и практическая реализация

"Издательство Наука и Техника"

Санкт-Петербург

УДК 004.42
ББК 32.973
ISBN 978-5-907592-91-9

Аклимов Р. Р.

Базы данных на SQL, MySQL, MS SQL и Postgre/PL/pgSQL. Проектирование и практическая реализация — СПб.: Издательство Наука и Техника, 2026. — 512 с., ил.

Серия "Для тех, кто хочет стать профессионалом"

У вас в руках — исчерпывающий гид по языку SQL, призванный помочь не просто освоить синтаксис, а глубоко понять принципы работы с данными и стать востребованным специалистом. Вы начнёте с абсолютного нуля, и каждый шаг — от установки СУБД до сложных запросов, — будет подкреплён практическими примерами (интернет-магазин, фитнес-клуб, банковская система, компьютерная игра, таксопарк, система регистрации в аэропорту, мониторинг состояния пациента в больнице и т.п.) и детальными объяснениями.

Понимая, что в мире не существует идеальной СУБД, книга разбирает каждую концепцию на примере трёх самых популярных систем: MySQL, PostgreSQL и Microsoft SQL Server. Такой подход позволит вам на практике увидеть их сильные и слабые стороны и избавиться от страха перед «незнакомой» средой, став по-настоящему гибким специалистом.

Особое внимание уделено «больным» темам, которые часто остаются за кадром в учебниках, но критически важны в работе: тонкости типов данных, кодировок и коллаций, механизмы обеспечения целостности, транзакции, уровни изоляции и блокировки. Вы поймёте не только «как это работает», но и «почему это работает именно так». Также книга выходит за рамки чистого SQL и рассматривает архитектурные вопросы: стратегии «мягкого» удаления, плюсы и минусы переноса бизнес-логики в хранимые процедуры, принципы безопасности и управления доступом и т.д.

Отдельный фокус сделан на проектировании таблиц, индексах и анализе планов выполнения запросов. Вы научитесь писать не только рабочие, но и эффективные запросы. Именно это умение — ключевой навык в реальной работе и главное отличие любителя от профессионала.

Эта книга является не только идеальным стартом для начинающих разработчиков, аналитиков и будущих администраторов БД, но также будет полезна и специалистам, уже работающим с СУБД: помимо глубоких теоретических знаний, книга даёт настоящую прикладную уверенность в работе с тремя основными СУБД современного рынка.

ISBN 978-5-907592-91-9

9 785907 592919 >

Контактные телефоны издательства:
(812) 412 70 26
Официальный сайт: www.nit.com.ru

© Аклимов Р. Р.
© Издательство Наука и Техника

Содержание

https://t.me/IT_Portal

ГЛАВА 1. **SQL для всех. Основы реляционных баз данных**.....21

1.1. ЗАЧЕМ НУЖЕН *SQL*.....22

SQL — это мост между вами и миром данных.....23

SQL проще, чем кажется?.....24

1.2. ИСТОКИ И РОЖДЕНИЕ *SQL*.....24

1.3. СТАНДАРТЫ И ДИАЛЕКТЫ.....25

1.4. ТАК ЧТО ЖЕ ВЫБРАТЬ?.....26

1.5. СИСТЕМНЫЕ ТРЕБОВАНИЯ.....28

1.6. УСТАНОВКА И ПОДКЛЮЧЕНИЕ.....29

1.6.1. MS SQL Server.....29

1.6.2. MySQL.....31

Что надо скачать?.....31

1.6.3. PostgreSQL.....32

1.6.4. Альтернативные графические оболочки.....33

1.7. ПОДКЛЮЧЕНИЕ К БАЗЕ, ПЕРВЫЙ ЗАПРОС.....34

1.7.1. Подключение к MS SQL Server.....35

1.7.2. Подключение к MySQL.....37

Подключение через командную строку.....37

1.7.3. Подключение к PostgreSQL.....38

ЧТО УЗНАЛИ И ЧЕМУ МЫ НАУЧИЛИСЬ?.....40

ГЛАВА 2. **Типы данных**.....43

2.1. ПОНЯТИЕ ТИПА ДАННЫХ.....44

Почему важно знать размер данных?.....44

2.2. ТИПЫ ДЛЯ ЦЕЛЫХ ЧИСЕЛ.....45

О знаковых и беззнаковых целых в MySQL.....45

2.3. ТИПЫ ДЛЯ ДЕЙСТВИТЕЛЬНЫХ ЧИСЕЛ.....46

Как быть, если нужна абсолютная точность?.....47

2.4. ТИПЫ ДЛЯ ТЕКСТА.....49

CHAR(n) / NCHAR(n).....50

CHARACTER VARYING(n) / VARCHAR(n) / NVARCHAR(n).....50

2.4.1. Про кодировки и Unicode.....50

Рекомендации.....51

2.4.2. Collation, или головная боль разработчика.....52

2.5. ДАТА И ВРЕМЯ.....53

2.6. ЛОГИЧЕСКИЙ ТИП ДАННЫХ (BOOLEAN).....54

2.7. NULL: ОТСУТСТВИЕ ЗНАЧЕНИЯ.....55

2.8. ДРУГИЕ ТИПЫ ДАННЫХ.....56

Специализированные типы: для нестандартных задач.....56

2.9. ЧТО ДАЛЬШЕ?.....57

2.10. ПРИЛОЖЕНИЕ К ГЛАВЕ 2: ТИПЫ ДАННЫХ СТАНДАРТА SQL.....58

ГЛАВА 3. **Основы проектирования баз данных на практике**.....61

3.1. ТАБЛИЧНОЕ ПРЕДСТАВЛЕНИЕ 62

3.2. ПРИМЕР 1. ИНТЕРНЕТ-МАГАЗИН. НОРМАЛИЗАЦИЯ 63

Как распознать ненормализованные данные? 64

3.3. ФОРМЫ НОРМАЛИЗАЦИИ 64

Первая нормальная форма (1NF) 65

Вторая нормальная форма (2NF) 65

Третья нормальная форма (3NF) 65

А все ли мы учли? 67

3.4. СОЗДАНИЕ БАЗЫ ДАННЫХ, ПРОЕКТИРОВАНИЕ ТАБЛИЦ 68

3.5. *CREATE TABLE* 75

3.5.1. О верхнем и нижнем регистре 76

MySQL 77

PostgreSQL 77

MS SQL Server 78

3.5.2. Работа в графических оболочках 80

3.6. ДОБАВЛЯЕМ ОТНОШЕНИЯ, ОГРАНИЧЕНИЯ И ВНЕШНИЕ КЛЮЧИ 81

3.6.1. Ссылочная целостность 84

Как это работает "под капотом"? 85

3.6.2. Автонумерация. Умные идентификаторы 87

MySQL 87

PostgreSQL 89

Microsoft SQL Server 91

3.6.3. Какие еще бывают ограничения? 93

UNIQUE 93

DEFAULT 94

CHECK 94

Когда что использовать? 94

3.6.4. Вычислимые столбцы 95

3.7. ПРИМЕР 2. ФИТНЕС-КЛУБ 96

3.7.1. Нормализация 98

3.7.2. Все ли мы учли? 100

3.7.3. Подбираем типы данных и создаем таблицы 100

***ЧТО УЗНАЛИ И ЧЕМУ МЫ НАУЧИЛИСЬ?* 103**

ГЛАВА 4. **Вставка, вывод и чтение данных** 105

4.1. ДОБАВЛЕНИЕ ДАННЫХ. ОСНОВЫ 106

4.1.1. Простая вставка. INSERT на примере Интернет-магазина 106

4.1.2. Массовое добавление данных 109

MySQL 110

PostgreSQL 111

MS SQL 111

4.2. ЧТЕНИЕ ДАННЫХ ИЗ БАЗЫ. *SELECT* 112

4.2.1. Псевдонимы столбцов (ALIAS) 114

4.3. ФИЛЬТРАЦИЯ ВЫВОДА. *WHERE* 114

4.3.1. Операторы сравнения 115

4.3.2. Логические операторы 116

4.3.3. Поиск по шаблону. Оператор LIKE 119

4.3.4. Поиск по сложному шаблону 121

4.3.5. Условный вывод с оператором CASE 123

4.3.6. Приоритет операторов 126

4.4. НЕЯВНОЕ АВТОМАТИЧЕСКОЕ ПРИВЕДЕНИЕ ТИПОВ В *SQL* 127

Сравнение разных типов 128

Арифметические операции 128

4.4.1. Преобразование даты в строку и обратно 129

4.4.2. Особенности СУБД 130
MySQL 130
PostgreSQL 130
SQL Server 131
4.4.3. Потенциальные риски потери данных 131

4.5. ПОСТРАНИЧНЫЙ ВЫВОД, ОГРАНИЧЕНИЕ ВЫВОДА, СОРТИРОВКА ...132

4.5.1. Сортировка. ORDER BY 132
MySQL, MS SQL Server 135
PostgreSQL 135
4.5.2. Особенности сортировки NULL 136
4.5.3. Использование колляции при сортировке 136
4.5.4. Постраничный вывод 139
Вопросы производительности и советы 142

ЧТО УЗНАЛИ И ЧЕМУ МЫ НАУЧИЛИСЬ? **....142**

ГЛАВА 5. **Объединение данных. Работа с несколькими таблицами**....145

5.1. ЕЩЕ НЕМНОГО О ПРЕИМУЩЕСТВАХ НОРМАЛИЗАЦИИ НА ПРИМЕРЕ ФИТНЕС-КЛУБА 146

Примечание о вставке данных 150

5.2. ПРОСТОЕ ОБЪЕДИНЕНИЕ 151

5.2.1. Как присоединять таблицу 151
5.2.2. Присоединение нескольких таблиц 156
5.2.3. Использование псевдонимов таблиц (aliases) 157

5.3. ВИДЫ ОБЪЕДИНЕНИЙ НА ПРИМЕРЕ СИСТЕМЫ РЕГИСТРАЦИИ В АЭРОПОРТУ 158

5.3.1. Внутреннее объединение INNER JOIN 158

5.3.2. Левое внешнее объединение LEFT OUTER JOIN.....159
5.3.3. Правое внешнее объединение RIGHT OUTER JOIN.....163
5.3.4. Полное объединение (FULL OUTER JOIN).....164
5.3.5. Перекрестное объединение CROSS JOIN.....165
5.3.6. Слияние (UNION и UNION ALL).....167
5.3.7. Полная карта объединений 169
5.3.8. Порядок объединений, или как ничего не потерять.....171
Вариант 1 171
Вариант 2 172

ЧТО УЗНАЛИ И ЧЕМУ МЫ НАУЧИЛИСЬ? **173**

ПРИЛОЖЕНИЕ К ГЛАВЕ 5. ВАРИАНТЫ РЕШЕНИЯ ЗАДАНИЙ.....174

ГЛАВА 6. **Группировка данных и агрегатные функции**.....177

6.1. ОСНОВЫ АНАЛИТИКИ НА ПРИМЕРЕ РЫБОЛОВЕЦКОЙ КОМПАНИИ 178

6.2. АГРЕГАТНЫЕ ФУНКЦИИ 181

6.2.1. DISTINCT — вывод уникальных записей.....181
6.2.2. COUNT — подсчет количества записей.....182
6.2.3. SUM — сумма 183
6.2.4. MIN/MAX — минимальное и максимальное значения.....183
6.2.5. AVG — среднее значение 184
6.2.6. COALESCE — замещение NULL.....185

6.3. ГРУППИРОВКА РЕЗУЛЬТАТОВ *GROUP BY*.....185

6.4. ФИЛЬТРАЦИЯ РЕЗУЛЬТАТОВ ПО ГРУППАМ — *HAVING* 188

6.5. НЕКОТОРЫЕ НЕСТАНДАРТНЫЕ АГРЕГАТНЫЕ ФУНКЦИИ В РАЗНЫХ СУБД 190

6.5.1. GROUP_CONCAT (MySQL) / STRING_AGG (PostgreSQL/MySQL).191
6.5.2. STDDEV_POP / STDEVP.....192

6.5.3. STDEV_SAMP / STDEV 192
6.5.4. VAR_SAMP / VAR 192
6.5.5. PostgreSQL: WITHIN GROUP и PERCENTILE_CONT/ PERCENTILE_DISC 193

ЧТО УЗНАЛИ И ЧЕМУ МЫ НАУЧИЛИСЬ? **....194**

ГЛАВА 7. **Подзапросы и обобщённые табличные выражения (CTE)**....195

7.1. ДАННЫЕ НА ОСНОВАНИИ ДРУГИХ ДАННЫХ. ПРАКТИЧЕСКИЙ ПРИМЕР КОМПЬЮТЕРНОЙ ИГРЫ 196

7.2. ПОДЗАПРОСЫ....199

7.2.1. Подзапрос, вложенный в SELECT....199
7.2.2. Подзапрос, вложенный в WHERE 200
7.2.3. Подзапрос, вложенный в FROM 200
7.2.4. Коррелируемый подзапрос 201
7.2.5. Проверка на наличие в наборе значений IN 202
7.2.6. Проверка на условие с наборами значений ANY / ALL....203
7.2.7. Быстрая проверка наличия WHERE EXISTS 204
7.2.8. Когда использовать подзапросы 206

7.3. ПРОБЛЕМЫ ПОДЗАПРОСОВ 207

7.3.1. Сложность для восприятия....207
7.3.2. Повторяющаяся логика 207
7.3.3. Производительность 208

7.4. ОБОБЩЕННЫЕ ТАБЛИЧНЫЕ ВЫРАЖЕНИЯ....208

7.4.1. Решение неэффективности подзапросов 208
7.4.2. Как это работает 209
7.4.3. Простые CTE....210

7.4.4. Рекурсия в CTE 211
7.4.5. Глубина рекурсии и опасность петель 215
7.4.6. Техники обнаружения петель в разных СУБД 217
7.4.7. Где рекурсия может быть полезна, а где — вредна 221

ЧТО УЗНАЛИ И ЧЕМУ МЫ НАУЧИЛИСЬ? **....222**

ГЛАВА 8. **Оконные функции. Анализируем данные**.....225

8.1. ДАННЫЕ В КОНТЕКСТЕ "ОКНА". ПРИМЕР МОНИТОРИНГА СОСТОЯНИЯ ПАЦИЕНТА В БОЛЬНИЦЕ 226

8.2. ЧТО ТАКОЕ ОКОННЫЕ ФУНКЦИИ 228

8.2.1. Синтаксис 228
8.2.2. Агрегатные функции: SUM, AVG, COUNT, MAX, MIN 231
8.2.3. Ранжирующие функции: ROW_NUMBER, RANK, DENSE_RANK ..234
Когда какую функцию использовать? 236
8.2.4. Функции распределения / относительного ранга: NTILE, PERCENT_RANK, CUME_DIST 237
PERCENT_RANK 237
CUME_DIST 238
NTILE(n) 239
8.2.5. Аналитические функции: LAG, LEAD, FIRST_VALUE, LAST_VALUE 240
LAG — доступ к предыдущим данным 241
LEAD — доступ к последующим данным 242
FIRST_VALUE — первое значение окна 243
LAST_VALUE — последнее значение окна 244

8.3. ВОПРОСЫ ПРОИЗВОДИТЕЛЬНОСТИ И АНТИПАТТЕРНЫ 244

8.3.1. Вложенные оконные функции 244
8.3.2. ROWS вместо RANGE 245

8.3.3. Фильтрация до оконных функций 246
8.3.4. Индексы по столбцам PARTITION BY и ORDER BY 246
8.3.5. Антипаттерн: избыточное секционирование 247
8.3.6. Антипаттерн: оконные функции вместо простых агрегатов .247

ЧТО УЗНАЛИ И ЧЕМУ МЫ НАУЧИЛИСЬ? **247**

ГЛАВА 9. **Изменение и удаление данных**....249

9.1. МАНИПУЛЯЦИЯ ДАННЫМИ НА ПРИМЕРЕ ТАКСОПАРКА....**250**

9.2. ЕЩЕ НЕМНОГО О ВСТАВКЕ ДАННЫХ....**252**

9.2.1. Вставка данных из подзапроса 252
9.2.2. Возврат данных при добавлении 255
Microsoft SQL Server (OUTPUT)....255
PostgreSQL (RETURNING)....256
MySQL 257

9.3. ИЗМЕНЕНИЕ ДАННЫХ....**257**

9.3.1. Простое изменение 258
9.3.2. О важности WHERE 259
9.3.3. Обновление данных из подзапроса/CTE....260
9.3.4. Обновление данных через присоединение JOIN....262
Microsoft SQL Server 262
MySQL 263
PostgreSQL....263

9.4. РАЗРЕШЕНИЕ КОНФЛИКТОВ С ОГРАНИЧЕНИЯМИ....**264**

9.4.1. Техника UPSERT 264
Microsoft SQL Server 265
MySQL 265
PostgreSQL....267

9.4.2. Объединение таблиц (MERGE) 268

9.5. УДАЛЕНИЕ ДАННЫХ 271

9.5.1. Простое удаление DELETE 271

9.5.2. Удаление по данным подзапроса или CTE 272

9.5.3. Удаление через присоединение 273

9.6. КОНТРОЛЬ ЦЕЛОСТНОСТИ ПРИ УДАЛЕНИИ ДАННЫХ 273

9.6.1. Действия ON DELETE 274

9.6.2. Таблица сравнения действий 276

9.7. ПОДХОДЫ К ХРАНЕНИЮ НЕВОСТРЕБОВАННЫХ ДАННЫХ 278

9.7.1. Немедленное удаление (IMMEDIATE DELETE) 278

9.7.2. "Мягкое" удаление (SOFT DELETE) 279

9.7.3. Отсроченное удаление (DELAYED DELETE) 279

9.7.4. Постоянное хранение (NO DELETE) 280

9.8. ПРОИЗВОДИТЕЛЬНОСТЬ ЗАПРОСОВ *UPDATE* И *DELETE* 280

9.8.1. Временное отключение проверок и триггеров 280

9.8.2. Дробление операций на части 282

Microsoft SQL Server 282

MySQL 282

PostgreSQL 282

9.8.3. Дополнительные рекомендации по производительности 283

***ЧТО УЗНАЛИ И ЧЕМУ МЫ НАУЧИЛИСЬ?* 283**

ГЛАВА 10. **Индексы и оптимизация. Улучшаем производительность**.....285

10.1. ЧТО ТАКОЕ ИНДЕКС И ЗАЧЕМ ОН НУЖЕН? 286

10.2. ВИДЫ ИНДЕКСОВ 287

10.3. ИДЕНТИФИКАТОРЫ СТРОК В СУБД 288

10.4. ТИПЫ ИНДЕКСОВ В РАЗНЫХ СУБД 290

10.4.1. MS SQL Server 291

10.4.2. MySQL (InnoDB) 293

10.4.3. PostgreSQL 294

10.4.4. Обобщенная информация по индексам для разных СУБД . 295

10.5. КАК ИНДЕКСЫ УСКОРЯЮТ ЗАПРОСЫ 297

10.5.1. Генерация больших объемов данных 297

Microsoft SQL Server 303

MySQL 304

PostgreSQL 305

10.5.2. Медленные запросы 306

Запрос 1. Количество посещений и средняя скидка для ..... регулярных пользователей 306

Запрос 2. Разные пользователи с одинаковым IP 308

Анализ рефералов по группам 309

10.5.3. Добавляем индексы 311

Запрос 1 (количество посещений и средняя скидка) 311

Запрос 2 (общие IP-адреса для разных пользователей) ... 312

Запрос 3 (статистика посещений по реферальным группам) . 313

10.5.4. Особенности синтаксиса 315

MS SQL Server — управление заполняемостью страниц . 316

MySQL — префиксные индексы 318

MySQL, PostgreSQL — выбор типа индекса 319

Построение индекса без блокировки таблицы в PostgreSQL . 319

Альтернативы для других СУБД 320

10.5.5. Фрагментация индексов и их перестройка 320

MS SQL Server 320

MySQL 322

PostgreSQL 323

10.5.6. Удаление и модификация индексов 325

10.6. АНАЛИЗ ПЛАНОВ ВЫПОЛНЕНИЯ 326

10.6.1. Планировщик запросов 326

10.6.2. Получение плана выполнения 327

MS SQL Server 327

MySQL 330

PostgreSQL 332

10.6.3. Как читать план выполнения? 332

Операции доступа к данным 333

Операции соединения (JOIN) 334

Операции сортировки и агрегации 336

10.7. ОПТИМИЗАЦИЯ ЗАПРОСОВ С ПОМОЩЬЮ ИНДЕКСОВ НА ПРАКТИКЕ..336

10.7.1. MS SQL Server 337

1-й этап. Подготовка CTE 338

2-й этап. Соединение с основной таблицей 338

3-й этап. Соединение с таблицей ***users*** 339

Финальный этап. Агрегация и сортировка 339

CTE 340

Присоединение CTE 340

Присоединение ***users*** 341

Финальный этап 341

10.7.2. MySQL 341

Этап 1. Обработка CTE 341

Этап 2. Присоединение CTE 342

Этап 3. Агрегация и сортировка 342

10.7.3. PostgreSQL 344

Ресурсоемкая сортировка 345

10.8. ДИАГНОСТИКА ПРОБЛЕМ ПРОИЗВОДИТЕЛЬНОСТИ 348

10.8.1. Неэффективные индексы 348

10.8.2. Статистика и ее роль 350

10.9. ЛУЧШИЕ ПРАКТИКИ И АНТИПАТТЕРНЫ 350

10.9.1. Рекомендации 351

10.9.2. Чего избегать 351

***ЧТО УЗНАЛИ И ЧЕМУ МЫ НАУЧИЛИСЬ?* 352**

ГЛАВА 11. **Представления (Views)** 353

11.1. ЗАЧЕМ НУЖНЫ ПРЕДСТАВЛЕНИЯ? 354

11.2. СОЗДАНИЕ ТЕСТОВЫХ ДАННЫХ. ПРИМЕР РАСПРЕДЕЛЕНИЯ ГОНОРАРОВ И ВОЗНАГРАЖДЕНИЙ 356

11.3. РАБОТА С ПРЕДСТАВЛЕНИЯМИ 357

11.3.1. Синтаксис создания и абстракция сложности 358

11.3.2. Безопасность и логическое разделение данных 359

11.3.3. Обновляемые представления 360

11.3.4. Материализованные представления 361

11.4. КОГДА ПРЕДСТАВЛЕНИЯ МОГУТ МЕШАТЬ? 364

***ЧТО УЗНАЛИ И ЧЕМУ МЫ НАУЧИЛИСЬ?* 365**

ГЛАВА 12. **Транзакции, изоляция и блокировки** 367

12.1. ПРИНЦИПЫ ACID 368

Atomicity (атомарность) 369

Consistency (Согласованность) 369

Isolation (Изолированность) 369

Durability (Долговечность) 369

12.2. ПРОЦЕСС ВЫПОЛНЕНИЯ ТРАНЗАКЦИИ 370

12.3. УРОВНИ ИЗОЛЯЦИИ ТРАНЗАКЦИЙ 371

12.4. РЕАЛИЗАЦИЯ УРОВНЕЙ ИЗОЛЯЦИИ В РАЗНЫХ СУБД. ПРИМЕР АВИАКАССЫ В ЧАС ПИК 373

12.4.1. Microsoft SQL Server 375

12.4.2. MySQL (InnoDB) 376

12.4.3. PostgreSQL 376

12.5. ПРАКТИЧЕСКАЯ РАБОТА С ТРАНЗАКЦИЯМИ. ПРИМЕР УПРОЩЕНОЙ БАНКОВСКОЙ СИСТЕМЫ 377

Задания для самостоятельной работы 380

12.6. БЛОКИРОВКИ 381

12.6.1. Виды блокировок 382

12.6.2. Как избегать взаимных блокировок? 384

12.6.3. Явное управление блокировками 385

12.6.4. Отключение блокировок (READ UNCOMMITTED) 388

12.6.5. Оптимистичная блокировка 389

ЧТО УЗНАЛИ И ЧЕМУ МЫ НАУЧИЛИСЬ? **....391**

ГЛАВА 13. **Бизнес-логика. Хранимые процедуры и функции**....393

13.1. СИНТАКСИС СОЗДАНИЯ 395

MS SQL Server 395

MySQL 396

PostgreSQL 397

13.2. ПЕРЕМЕННЫЕ 399

13.3. ПАРАМЕТРЫ 403

MS SQL Server 404
MySQL 404
PostgreSQL 404

13.4. ТИПЫ ПАРАМЕТРОВ, ФУНКЦИИ 405

13.4.1. Входные (IN), выходные (OUT) и смешанные (INOUT) параметры 405
Перегрузка в PostgreSQL 411
13.4.2. Функции 412
MS SQL Server 413
MySQL 414
PostgreSQL 416
13.4.3. Скалярные и табличные результаты 417
MS SQL Server 417
MySQL 417
PostgreSQL 418

13.5. ПРОВЕРКА УСЛОВИЙ, ВЕТВЛЕНИЯ 420

MS SQL Server 422
MySQL 423
PostgreSQL 424

13.6. ЦИКЛЫ 425

13.6.1. Организация циклов в разных СУБД 425
13.6.2. Досрочный выход из цикла, пропуск итерации 428
MS SQL Server 429
MySQL 429
PostgreSQL 430
13.6.3. Модификация процедуры записи на прием 430
MS SQL Server 431
MySQL 432
PostgreSQL 433

13.6.4. Курсоры. Построчная обработка запроса....434
13.6.5. Пример поднятия цен в книжном магазине....435
MS SQL Server....437
MySQL....439
PostgreSQL....440

13.7. ОБРАБОТКА ОШИБОК....442

13.7.1. MS SQL Server....443
13.7.2. MySQL....444
13.7.3. PostgreSQL....445

13.8. ТРИГГЕРЫ....447

13.8.1. Создание триггера....448
MS SQL Server....449
MySQL....450
PostgreSQL....451
13.8.2. Изменение триггеров....452
13.8.3. Вопросы производительности....452
MS SQL Server....453
MySQL....453
PostgreSQL....453
13.8.4. Советы....454

13.9. ПЛЮСЫ И МИНУСЫ ПЕРЕНОСА БИЗНЕС-ЛОГИКИ В СУБД....454

13.9.1. Что вы получаете, реализуя бизнес-логику в СУБД....454
13.9.2. Что плохого вы получите "в нагрузку"....455
13.9.3. Переносить или не переносить?....456

ГЛАВА 14. **Безопасность данных и управление пользователями**....459

14.1. КЛЮЧЕВЫЕ КОНЦЕПЦИИ....461

14.1.1. Принцип наименьших привилегий....461

14.1.2. Субъекты безопасности 461
14.1.3. Схемы (Schemas)....463

14.2. ПРЕДОПРЕДЕЛЕННЫЕ (СИСТЕМНЫЕ) РОЛИ 464

14.3. ТИПЫ ПРИВИЛЕГИЙ. ЧТО МОЖНО РАЗРЕШИТЬ 465

14.4. УПРАВЛЕНИЕ ПОЛЬЗОВАТЕЛЯМИ И ПАРОЛЯМИ....466

14.4.1. Основные команды 466
MS SQL Server 466
MySQL 467
PostgreSQL....468
14.4.2. Смена пароля 469
MS SQL Server 469
MySQL 469
PostgreSQL....469

14.5. УПРАВЛЕНИЕ РОЛЯМИ 469

14.6. ПРЕДОСТАВЛЕНИЕ И ОТЗЫВ ПРИВИЛЕГИЙ 470

14.6.1. Правила указания объектов 471
MS SQL Server 471
MySQL 472
PostgreSQL....473
14.6.2. Наследование ролей в PostgreSQL....475
14.6.3. Антипаттерны....475

14.7. АУДИТ И ПРОФИЛАКТИКА 476

14.7.1. SQL Server....476
sys.database_role_members 477
sys.database_principals 477
14.7.2. MySQL 479
14.7.3. PostgreSQL....480

ЧТО УЗНАЛИ И ЧЕМУ МЫ НАУЧИЛИСЬ? 481

Заключение.....483

Куда двигаться дальше?.....484

Смежные технологии и направления.....485

ГЛОССАРИЙ ТЕРМИНОВ.....487

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ ИНФОРМАЦИИ.....509

Глава 1.

SQL для всех. Основы реляционных баз данных

https://t.me/IT_Portal

1.1. Зачем нужен SQL

Представьте, что у вас есть ключ, который открывает доступ к любой информации в мире баз данных.

Этот ключ — **SQL** (Structured Query Language, *структурный язык запросов*). Он нужен не только программистам: сегодня данные стали универсальным языком бизнеса, науки и технологий.

Знание SQL позволяет задавать вопросы данным и получать ответы, которые меняют решения, открывают тренды и экономят миллионы.

Почему SQL нужен вам? Данные окружают нас везде: от онлайн-заказов до медицинских исследований. SQL — это инструмент, который превращает «сырые» цифры в смыслы.

И вот как он поможет **вам**, независимо от профессии:

- Если вы работаете бизнес-аналитиком, освоив SQL, вы обретете суперспособности, недоступные простым пользователям Excel, сможете при помощи написания одного запроса делать то, над чем ваши коллеги будут корпеть полдня. Такие задачи, как подготовка и сведение отчетов, сводные таблицы, фильтрация по множеству условий сразу станут для вас пустяком, что, несомненно, отразится на вашей производительности. Вы станете тем, кто превращает таблицы в стратегии.
- Если вы — менеджер, то станете мастером фактов, не нужно будет проводить долгие часы, анализируя отчеты подчиненных о проделанной работе, или ждать, когда наконец будет готов тот отчет, который вы сказали

подготовить. Вы сможете упростить себе жизнь, моментально получая необходимые сведения (кто выполнил план, где наибольшие издержки по логистике, с какой рекламной площадки приходит больше клиентов и пр.).

- Если вы — исследователь и занимаетесь наукой, то получите мощный аналитический инструмент, позволяющий вам быстро находить аномалии в результатах экспериментов, выявлять закономерности (есть ли корреляция между дозой препарата и наличием побочных эффектов, например), игнорировать результаты, где условия не подходят по критериям.
- Маркетологам знания SQL помогут лучше понять своих клиентов, давая возможность, к примеру, сегментировать аудиторию (кто чаще покупает после email-рассылки), считать ROI кампаний (какие каналы приносят больше всего целевых лидов), провести A/B-тестирование (сравнить конверсию версии сайта A и B).
- Даже если вы просто работаете секретарем в учебном заведении, знание SQL позволит вам составить расписание занятий гораздо быстрее.

Разумеется, кто-то может возразить, что на все перечисленные задачи существуют специализированные программы. Однако, подобные аналитические инструменты весьма дороги и доступны далеко не всем, а «под капотом» скрывают все те же SQL-запросы в обертке из красивого интерфейса.

Ну и, разумеется, если вы планируете свою карьеру в ИТ, знание SQL для вас не является опцией, оно жизненно необходимо. В настоящее время мало просто уметь программировать на каком-либо языке, важно уметь манипулировать данными профессионально и эффективно, и обойтись без знаний SQL вам никак невозможно.

SQL — это мост между вами и миром данных

Неважно, кто вы — биолог, менеджер проекта или начинающий разработчик. SQL даст вам способности:

- задавать вопросы там, где другие видят только цифры;
- автоматизировать рутину (прощай, копирование из Excel в Excel!);

- говорить на одном языке с ИТ-специалистами.

SQL проще, чем кажется?

- SQL читается, как английские слова: SELECT name FROM employees WHERE salary > 50000 (ВЫБЕРИ имя ИЗ работников ГДЕ зарплата > 50000).
- Не нужно быть математиком: логика условий (AND/OR) интуитивна.
- Результат виден сразу: ввели запрос — получили таблицу.

Начать можно с малого. Спросите себя: «Что я хочу узнать из данных, с которыми работаю?» — и напишите первый запрос. Уже через неделю вы удивитесь, как жили без этого. И эта книга поможет вам в этом.

1.2. Истоки и рождение *SQL*

Одной из главных задач, которую люди хотели решить, изобретая компьютеры, была **быстрая обработка данных**. С самого начала самые первые компьютеры уже стали накапливать данные на магнитных лентах. Первые базы данных имели мало общего. Чтобы хотя бы начать работать с ними, специалистам необходимо было для начала изучить документацию о том, как именно физически организовано хранение информации, затем выяснить, где именно расположены нужные им данные, а затем — создать программу, которая эти данные извлечет. Все это было медленно и крайне неудобно.

В 1970 году Эдгар Кодд предложил модель абстракции данных в виде таблиц, упростив доступ и манипуляции, а в 1974 году фирма IBM в своем проекте System R создала язык SEQUEL (Structured English Query Language — *структурированный [английский] язык запросов*), позже переименованный в SQL.

Это был не единственный на то время язык доступа к данным, можно вспомнить, например, QUEL, но популярность SQL получил за свой декларативный подход, когда пользователь просто писал, что ему нужно получить, а не как это сделать (выбери все заказы за апрель — не нужно описывать процесс получения этих заказов).

Закрепила успех компания Oracle, которая в 1979 году выпустила первую коммерческую СУБД с поддержкой SQL. С тех пор SQL царствует в мире обработки данных и не собирается сдавать свои позиции, оставаясь все таким же актуальным, как и половину столетия назад.

1.3. Стандарты и диалекты

В 1986 году национальный институт стандартов США и международная организация ISO в 1987 стандартизировали SQL, превратив его в универсальный, используемый по всему миру инструмент. С тех пор, с какой бы вы не столкнулись СУБД, синтаксис базовых запросов остается неизменным.

Не важно, что вы используете — бесплатные MySQL или PostgreSQL либо коммерческие Microsoft SQL Server или Oracle, вы сможете выполнить практически любой запрос в соответствии со стандартом SQL, и в подавляющем большинстве случаев вам не потребуются знания, специфические конкретно для выбранной СУБД.

Тем не менее, задачи, которые в современном мире решают СУБД, а также стремление производителей ПО к получению конкурентных преимуществ в избранных нишах делают строгий стандарт языка SQL, который был разработан в прошлом веке, слишком тесным.

В современном мире появились потребности хранить более сложные типы данных, чем предлагается стандартом, и Postgres расширила этот

набор, например, добавив специализированную поддержку для геоинформационных систем, в то же время уделяя внимание соблюдению классического стандарта, а MySQL пошел другим путем — сосредоточился на производительности и доступности, сделав синтаксис максимально интуитивным для массового пользователя.

Эти различия отражают философию разработки: PostgreSQL стремится к строгости и расширяемости, MySQL — к скорости и простоте внедрения. Так со временем закрепился стереотип, что PostgreSQL создан для инженеров, а MySQL — для разработчиков, жертвуя сложными функциями ради низкого порога входа.

Гиганты вроде Microsoft и Oracle не отставали, предлагая широкий набор расширений для разработчиков, обитающих в их экосистемах, и расширения для языка, недоступные или труднореализуемые у конкурентов.

1.4. Так что же выбрать?

Различия в реализации языка разными производителями называют «диалектами». Диалекты — плата за гибкость и развитие технологий, но ядро SQL остается единым уже более 50 лет. Для понимания основ SQL различия между диалектами не имеют принципиального значения. Примеры запросов из этой книги будут работать практически идентично в любой СУБД. Если в каких-то случаях требуются специфические конструкции, это обязательно будет оговорено особо.

Бесплатные СУБД, такие как PostgreSQL и MySQL, наиболее доступны для обучения. Однако даже коммерческие системы, вроде Microsoft SQL Server, предоставляют бесплатные версии для некоммерческого использования.

Выбор СУБД всегда зависит от конкретных задач и часто становится предметом споров. Ниже вы найдете обобщенное сравнение трех систем, которые мы будем изучать: MySQL, PostgreSQL и Microsoft SQL Server.

Если какие-то термины покажутся вам незнакомыми — не переживайте, все станет понятнее по мере изучения материала.

Таблица 1.1. Обобщенное сравнение MySQL, PostgreSQL и Microsoft SQL Server

Параметр	MySQL	PostgreSQL	Microsoft SQL Server
Легкость установки	Очень простая. Часто предустановлен на хостингах	Средняя. Требует настройки, но есть подробные инструкции	Средняя. Интегрирован с Windows, установка может занять время
Доступность	Бесплатная (Community Edition)	Бесплатная (Open Source)	Бесплатные версии (Express Edition) с ограничениями. Коммерческие лицензии дорогие
Типовые задачи	Веб-приложения, малый и средний бизнес, стартапы	Сложные типы данных, аналитика, научные проекты, машинное обучение	Корпоративные приложения, тесная интеграция с Windows, аналитика
Плохо подходит*	Сложная аналитика, многопоточные транзакции	Высоконагруженные веб-приложения (требует оптимизации)	Небольшие проекты из-за стоимости и сложности
Поддержка	Активное сообщество, много материалов для веб-разработчиков	Сильное сообщество, документация на высоком уровне	Отличная поддержка Microsoft, но меньше open-source инициатив
Распространенность	Лидер в веб-разработке (WordPress, Joomla)	Популярен в научных сообществах и аналитике	Очень распространен в корпоративной среде
Языки программирования**	PHP, Python, JavaScript (Node.js)	Python, Ruby, Java, JavaScript	C#, .NET, Java

ПРИМЕЧАНИЕ.

* «Плохо подходит» не означает, что использовать в данной сфере нельзя, скорее, это следует понимать, что имеются более подходящие альтернативы.

** Любой язык программирования может работать с любой СУБД через драйверы (библиотеки). Но перечисленные пары — это «классика», где интеграция максимально удобна, а сообщество предлагает больше готовых решений «из коробки».

1.5. Системные требования

При выборе СУБД важно учитывать не только задачи, но и **ресурсы вашего компьютера или сервера**:

- **MySQL** — самая «легкая» из трех. Она хорошо работает даже на слабых машинах: для локальной разработки или небольшого сайта хватит 1-2 Гб оперативной памяти и простого процессора. Это делает ее идеальной для обучения, стартапов и проектов с ограниченным бюджетом. Например, MySQL часто используют на дешевых хостингах.
- **PostgreSQL** — часто требует больше ресурсов, особенно для сложных аналитических запросов или работы с большими объемами данных. Минимум 2–4 Гб оперативной памяти и современный процессор. Однако для базовых задач (учебные проекты, небольшие приложения) он тоже запустится на обычном ноутбуке.
- **Microsoft SQL Server** — самый «тяжелый» в коммерческих версиях (Enterprise). Для работы с большими базами и аналитикой в реальном времени потребуется сервер с 4+ Гб памяти и мощным CPU. Но если вы только учитесь, бесплатная версия Express Edition (с ограничением в 10 ГБ на базу) легко встанет на домашний ПК с Windows.

Если у вас старый компьютер или вы работаете в облаке с ограниченными ресурсами — выбирайте MySQL или PostgreSQL.

Для Windows-ориентированных проектов или интеграции с облаком Microsoft (Azure) подойдет SQL Server Express.

Все три СУБД доступны в облачных сервисах (AWS, Google Cloud, Azure), где требования к локальному железу не важны: вы арендуете мощности удаленно, однако, если вы изучаете SQL, рекомендуется установить одну (или все три, если хотите) СУБД на ваш компьютер.

Совет для новичков: не гонитесь за «мощными» решениями для простых задач. Начните с минимальных конфигураций — этого хватит, чтобы освоить основы SQL и создать первый проект.

1.6. Установка и подключение

В данном разделе предоставлены подробные инструкции по установке СУБД. Если вы уже сделали это, умеете это делать, можете смело пропустить этот раздел.

С СУБД можно работать несколькими способами:

- через командную строку;
- через графическую оболочку;
- через программный код.

Наиболее удобной для учебных целей, конечно, будет работа через графическую оболочку, тем не менее, в ряде случаев умение обходиться без нее может оказаться полезным. Здесь приводятся инструкции как для подключения через терминал, так и посредством графической оболочки.

В связи с тем, что с момента написания настоящего материала конкретные ссылки и версии продуктов могут измениться, для нахождения текущей страницы для скачивания той или иной СУБД рекомендуется воспользоваться поисковым запросом.

Еще одним общим для всех рассматриваемых СУБД является совет скачивать стабильные версии на долгосрочной поддержке и не использовать по возможности beta-версии, в которых обкатываются какие-то новые возможности. Не переживайте, что вы не используете самую последнюю версию. Для освоения материала и практики это не имеет значения.

Даже если что-то пойдет не так, помните, вы всегда можете удалить установку и начать все сначала.

Если вы определились, какую именно СУБД вы хотите использовать, смело можете пропускать разделы, посвященные альтернативам.

1.6.1. MS SQL Server

Поскольку Microsoft периодически меняет структуру своего сайта, проще всего воспользоваться поисковым запросом **MS SQL Express download**.

В будущем для экономии места под сокращенным наименованием MS-SQL будет пониматься Microsoft SQL Server (как Express, так и полная версия).

Использование MSSQL на других ОС, кроме Windows, хотя и теоретически возможно, но зачастую не имеет большого смысла, так как на других ОС становятся неочевидными все ее преимущества и возможности для интеграции с другими продуктами от Microsoft.

Кроме того, поскольку MSSQL для бесплатного использования предоставляет только ограниченную по функциональности версию, стоит остановиться подробнее на этих ограничениях (перечислены только основные):

- Версия доступна **только для некоммерческого использования**.
- Размер базы ограничен 10 Гб, при превышении работа СУБД блокируется.
- Ограничение по памяти 1,4 Гб, что недостаточно для высоко- и даже средненагруженных систем.
- Используется до 4 ядер процессора.
- Количество пользователей: версия Express рассчитана 50 пользователей. Строгого ограничения нет, но при большем количестве возникают серьезные задержки.
- Невозможно создавать регламентные задания (автоматическое резервное копирование, оптимизация индексов и т.д.).

Если вы просто изучаете SQL, вы без проблем сможете уложиться в данные ограничения, однако, если вы планируете изучать и одновременно разрабатывать свой проект, примите данную информацию к сведению.

После того как вы скачаете установщик, вам будет предложено ввести имя для экземпляра вашего сервера. Рекомендуется оставить его без изменений: **SQLEXPRESS**.

В разделе **режим аутентификации** (authentication mode) укажите **смешанный режим** (Mixed mode) и задайте пароль суперпользователя **sa** (сокращенно от *system administrator*).

Стоит повторить: помните, что использование учетной записи суперпользователя для каких бы то ни было задач, кроме нужд администрирования, не рекомендуется из соображений безопасности.

MSSQL также имеет графическую оболочку, которая называется **SQL Server Management Studio (SSMS)**, однако она не входит в пакет Express и требует отдельного скачивания и установки.

Она бесплатна и легко находится по соответствующему поисковому запросу: **SSMS download**.

Если у вас есть желание работать с MSSQL через терминал, вам также потребуется утилита sqlcmd, которая входит в пакет **SQL Server Command Line Tools**.

1.6.2. MySQL

Текущая версия MySQL доступна на сайте https://www.mysql.com

`Поисковый запрос:` **MySQL community edition**.

Что надо скачать?

Вам обязательно понадобится **MySQL community server**.

- Для **Windows** найдите в списке **MySQL Installer for Windows**, затем выберите версию **MySQL Community Server** в процессе установки.

  Этот установщик автоматически загрузит сервер и предложит установить дополнительные инструменты, например **Workbench** — инструмент для визуального управления базами данных, написания запросов и проектирования схем. Вы можете установить его либо воспользоваться альтернативами, которые будут рассмотрены позднее. Он есть в составе MySQL Installer для Windows или доступен отдельно для macOS/Linux.

- **Для macOS/Linux:** выберите **MySQL Community Server**, скачайте версию под вашу ОС (например, .dmg для macOS, .deb или .rpm для Linux).

  Если вы планируете использовать терминал (оболочку командной строки) для работы с MySQL, вам также потребуется **MySQL Shell**.

  Если у вас возникают трудности при установке, оставьте настройки по умолчанию — они подходят для большинства учебных задач.

Рекомендуется оставить настройки типа конфигурации (config type) Developer Computer, в настройках связности (connectivity) оставьте метод TCP/IP и порт 3306.

Во время установки система запросит пароль для учетной записи **root** — это главный администратор вашей СУБД. Для обучения такой аккаунт допустим, но в реальных проектах никогда не используйте **root** в работе. Создавайте отдельных пользователей с ограниченными правами — это защитит данные от случайных ошибок и злоупотреблений.

Более подробно о настройке безопасности, ролях и привилегиях мы поговорим в следующих главах.

1.6.3. PostgreSQL

Официальный сайт проекта https://www.postgresql.org

Поисковый запрос: **postgres download.**

Раздел для скачивания сразу предлагает выбрать вашу операционную систему и в режиме мастера проводит вас через необходимые шаги для скачивания нужного пакета для установки.

Для работы с СУБД в терминале отметьте для установки драйверы командной строки. Также рекомендуется установить графическую оболочку **pgAdmin**.

Остальные параметры установки рекомендуется оставить без изменений (порт 5432).

При установке вам будет предложено ввести пароль для суперпользователя (в данном случае это **postgres** — обратите внимание — с одним «s» на конце). Не оставляйте пароль пустым.

Если вы не читали раздел по установке MySQL, стоит повторить: для учебных проектов можно использовать учетную запись *postgres*, но в рабочих задачах создавайте отдельных пользователей.

1.6.4. Альтернативные графические оболочки

Помимо «штатных» средств подключения к СУБД через графический интерфейс, существует и несколько независимых разработок.

Основным плюсом использования многих из них является универсальность — их можно использовать с любой СУБД, что бывает удобно.

- **DBeaver**

  Бесплатный кроссплатформенный инструмент с открытым исходным кодом, поддерживающий более 80 СУБД. Прекрасный инструмент, одним из единственных его недостатков, пожалуй, является частота выходящих обновлений, которая может раздражать. На больших нагрузках начинаются проблемы с производительностью.

- **HeidiSQL**

  Бесплатная оболочка с открытым исходным кодом. Также поддерживает большинство известных СУБД. Ранее была доступна только на Windows, теперь получила и полноценную версию для Linux (на момент написания доступна только альфа-версия).

- **Beekeeper Studio**

  Подкупает своим интерфейсом, поддерживает множество СУБД, однако в бесплатной версии имеет ряд серьезных ограничений.

- **Adminer**

  Подходит там, где нужна легкость и минимализм, для простых задач. Непригоден для сложной аналитики. Тем не менее, популярен из-за хорошей расширяемости за счет плагинов.

- **OmniDB**

  Заслуживает внимания как веб-инструмент, работающий из браузера и не требующий установки, что является и плюсом (легкодоступность), и минусом (низкая производительность) одновременно.

1.7. Подключение к базе, первый запрос

Мы установили сервер баз данных, теперь нам надо научиться подключаться к самой базе данных.

Для доступа к базе данных фундаментально требуются следующие реквизиты:

- **Адрес сервера** — имя или IP-адрес компьютера в сети, на котором установлен сервер СУБД. В реальных проектах обычно адрес отличается от адреса текущей машины, в нашей учебной настройке этот адрес будет localhost или IP 127.0.0.1.

ПРИМЕЧАНИЕ.

Если вы при установке указали TCP-порт не по умолчанию, при подключении важно будет указывать его тоже.

- **Имя для базы данных** — поскольку один сервер может обслуживать сразу несколько баз данных, мы можем создавать новые базы данных, изменять и удалять их, поэтому имя необходимо, чтобы их различать.
- **Логин** и **пароль** — здесь все просто, это реквизиты пользователя, который подключается к базе данных. Нужно помнить, что у разных пользователей может иметься разный набор прав и привилегий, одни пользователи могут просматривать данные, другие нет, одни могут изменять что-то в базе, другие могут только считывать информацию. В нашем случае мы будем пользоваться всеми правами, так как будем использовать учетную запись администратора (**root** для MySQL, **postgres** для PostgreSQL или **sa** для MS SQL), однако в реальных системах использование этих учетных записей представляет собой очень серьезную угрозу безопасности и целостности данных.

СУБД имеют и другие опции для подключения, если требуется более тонкая настройка, но в данный момент они нам не потребуются. Ниже приведены инструкции по подключению к разным СУБД.

Если вы осуществляете подключение из командной строки и при выполнении команды получите сообщение о том, что файл или команда не найдены, скорее всего, это значит, что путь к СУБД не прописан в переменных окружения вашего терминала. Добавьте путь к СУБД к переменной PATH либо выполите команду из той директории, где она установлена.

1.7.1. Подключение к MS SQL Server

MSSQL по умолчанию не поставляется со средствами работы в командной строке. Чтобы воспользоваться данным функционалом, вам необходимо будет установить пакет **sqlcmd** либо, если вы используете менеджер пакетов **pip**, — средство **mssql-cli**.

Для подключения к локальному серверу с именем SQLEXPRESS (по умолчанию) вам необходимо будет воспользоваться командой:

```
sqlcmd -S .\SQLEXPRESS -E
```

Здесь ключ *-S* предполагает, что далее будет имя сервера (для локальной машины используется подстановка в виде точки (.), затем через обратный слеш следует имя экземпляра базы данных (SQLEXPRESS по умолчанию или то, что вы указали при установке). Ключ *-E* говорит, что для аутентификации следует использовать встроенный механизм ОС Windows (уже на этом этапе проявляются преимущества интеграции с продуктами Microsoft).

Принцип ввода команд для MS SQL несколько отличается от предыдущих двух СУБД. Вам сразу предлагается вводить многострочную команду, начиная с первой строки (приглашение 1>).

Вводите на первых двух строках:

```
1> SELECT 1+1
2> GO
```

Результатом будет:

```
-----------
          2
(1 rows affected)
```

Стоит пояснить, что команда GO указывает среде, что необходимо выполнить все введенные до этого команды.

Для интерактивной работы с СУБД разумнее использовать графическую оболочку, которая в данном случае называется **SQL Server Management Studio** или сокращенно SSMS. Она тоже не поставляется вместе с SQL Server Express, и вам придется скачать и установить ее отдельно.

При запуске программы вам также будет предложено выбрать, к какому серверу и какой базе данных нужно подключиться, и, возможно, вид аутентификации (SQL Server потребует ввода логина и пароля пользователя либо аутентификации операционной системы Windows, использующей реквизиты, которые вы указали при входе в систему).

После подключения нажмите на кнопку создания запроса (New Query) или нажмите CTRL + N, после чего в появившемся окне введите тот же запрос и нажмите кнопку **Execute** (выполнить):

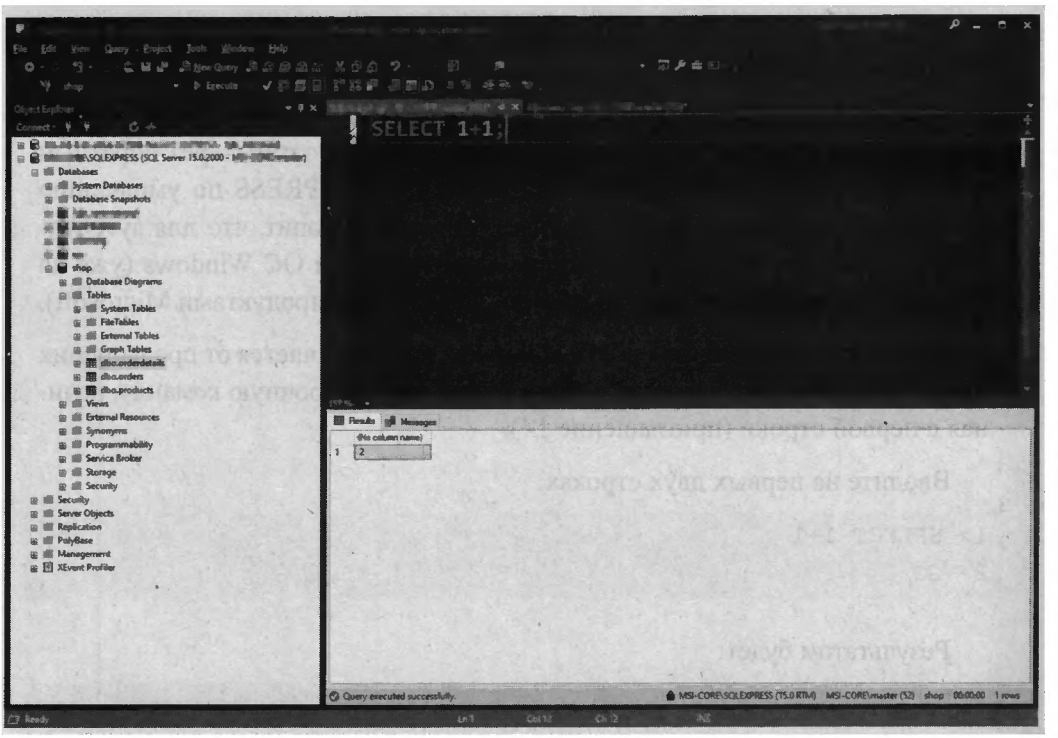


Рис. 1.1. *Окно SSMS с введенным запросом*

1.7.2. Подключение к MySQL

Подключение через командную строку

Откройте окно терминала в Linux или окно командной строки (*cmd* или *powershell* в Windows), затем введите команду:

```
mysql -u root -p
```

Здесь мы вводим имя пользователя **root** при помощи ключа *-u* (от *user*), а дальше ключ *–p*, указывающий на то, что пароль надо спросить при выполнении команды (так этот пароль не останется в истории введенных команд и не отобразится на экране).

Введите команду (не забудьте точку с запятой (;) в конце):

```
SELECT 1+1;
```

Мы просим сложить 1 и 1. Если вы все сделали правильно, результатом будет такой ответ с результатом запроса 2:

```
mysql> SELECT 1+1;
+-----+
| 1+1 |
+-----+
|   2 |
+-----+
1 row in set (0.00 sec)
```

Если вы используете графическую оболочку **MySQL Workbench**, просто запустите ее, на начальном экране выберите подключение, введите пароль и затем в окне запроса введите тот же запрос.

ПРИМЕЧАНИЕ.

Чтобы выйти из среды mysql, вам необходимо воспользоваться командой **выхода** (quit), которую можно увидеть ниже:

```
\q
```

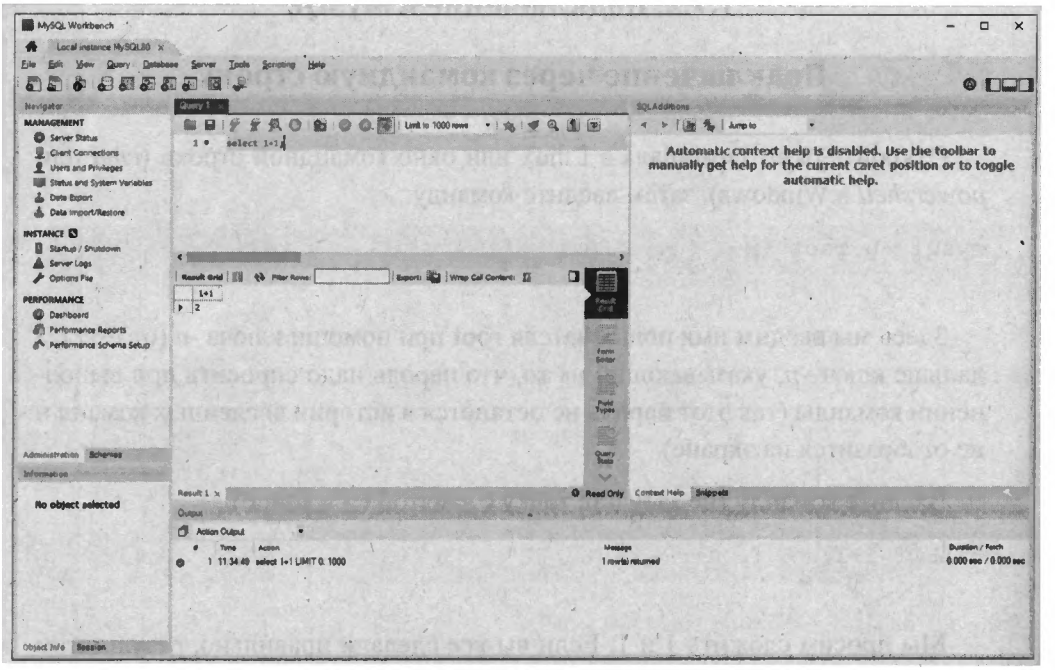


Рис. 1.2. *Окно MySQL Workbench с введенным запросом*

1.7.3. Подключение к PostgreSQL

Откройте окно терминала в Linux или окно командной строки (*cmd* или *powershell* в Windows), затем введите команду:

```
psql -U postgres
```

postgres, как уже упоминалось, — это название учетной записи по умолчанию.

Важно: ключ -U должен быть с заглавной буквы!

После ввода команды (помните про добавление директории СУБД в переменную окружения PATH) вам будет предложено ввести пароль (по умолчанию тоже postgres, если вы не указывали иной пароль при установке).

После появления в окне приглашения СУБД введите:

```
SELECT 1+1;
```

Не забудьте указать точку запятой в конце, иначе СУБД посчитает, что вы еще не закончили вводить команду. Точка с запятой в конце команды не обязательна при использовании графических оболочек, но необходима при работе в режиме командной строки.

Результат ваших усилий будет выглядеть примерно так:

```
postgres=# SELECT 1+1;
 ?column?
----------
        2
```

ПРИМЕЧАНИЕ.

Для пользователей Windows: если вы используете русскоязычную версию PostgreSQL, то в режиме работы с командной строкой вы сможете столкнуться с проблемой неверного отображения кириллических символов. Если это происходит, вам необходимо удостовериться, что внутри СУБД используется кодировка **UTF8**, для чего вам надо выполнить команду:

```
\encoding
```

в результате чего вам будет выдана текущая кодировка, которая используется интерпретатором psql. Если она отличается от UTF8, введите:

```
\encoding UTF8
```

Если же это не помогло, то для начала вам необходимо выйти из среды psql, воспользовавшись командой:

```
\q
```

Затем, вернувшись в среду операционной системы, ввести:

```
chcp 65001
```

Если вы используете графическую оболочку **pgAdmin**, то при запуске вам также надо будет указать адрес сервера (в нашем случае *localhost*) и параметры подключения: база данных **postgres**, имя пользователя **postgres**, пароль **postgres**.

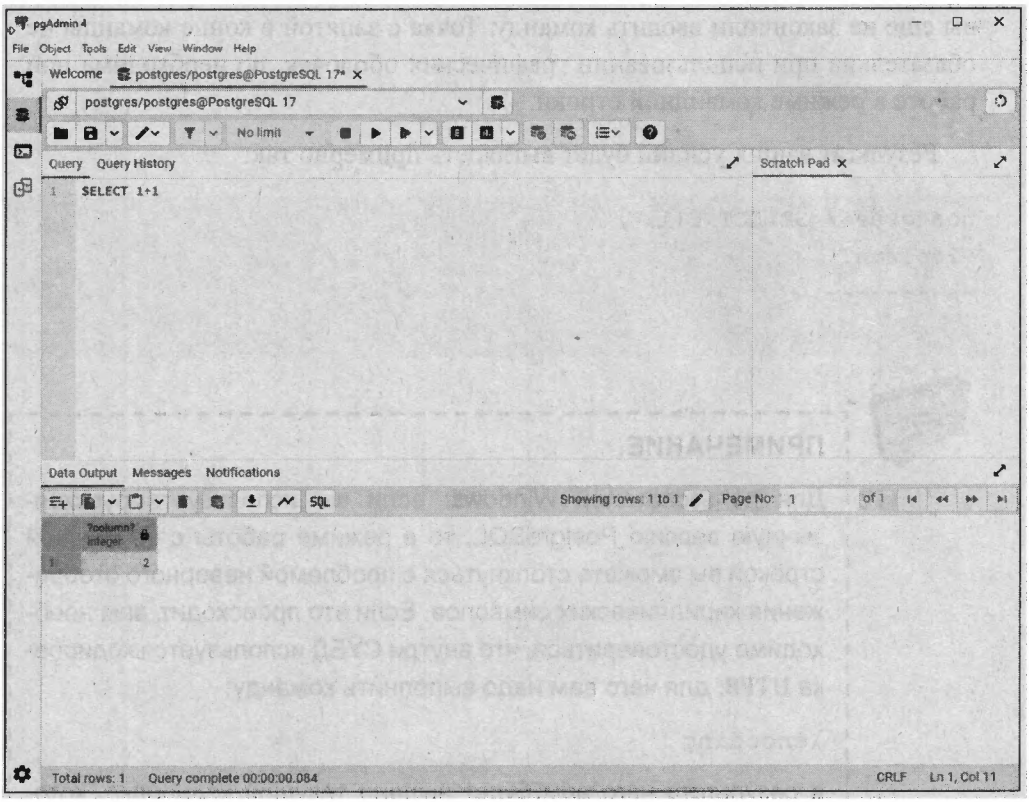


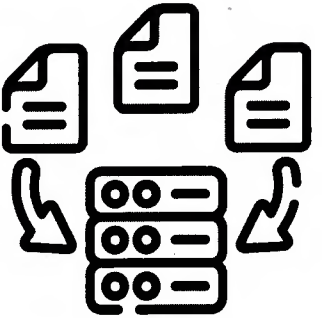
Рис. 1.3. *Окно pgAdmin с введенным запросом*

Что узнали и чему мы научились?

Если вы проделали все описанные действия, то поздравляю вас — вы научились подключаться к базе данных и выполнили команду SQL SELECT 1+1, попросив вычислить сумму двух чисел.

Дальнейшее повествование пойдет без указания, каким именно способом вам надо подключиться к базе данных и где именно вводить команду, так как предполагается, что вы уже умеете это делать.

Вы можете использовать любой понравившийся вам способ и любую из описанных СУБД для упражнений.



Глава 2.

Типы данных

https://t.me/IT_Portal

2.1. Понятие типа данных

Данные — это не беспорядочный набор цифр и слов. Это правила, которые определяют структуру, хранение и обработку информации.

Представьте, что вы заполняете анкету. Каждое поле требует конкретного формата:

- **Текст** для имени,
- **Дата** для даты рождения,
- **Логическое значение** для пола (М/Ж),
- **Число** для количества детей.

Эти правила аналогичны типам данных в SQL. Они не только задают формат («здесь должно быть число»), но и определяют, сколько места займет информация в базе.

Почему важно знать размер данных?

Рассматривая типы данных, необходимо очень хорошо понимать не только, какую именно информацию мы хотим хранить (число, дату, текст), но и какого размера (в байтах) будут эти данные.

Каждый тип данных резервирует фиксированный объем памяти.

Например: в двух байтах можно хранить числа от –32 768 до +32 767. Такой размер идеален для счетчиков, которые вряд ли превысят 30 тысяч (например, подписчики в небольшом сообществе), в то же время, если отвести под хранение 8 байт, мы сможем вместить в него до 9 квинтиллионов значений. Для подсчета книг в библиотеке, где количество томов вряд ли превышает несколько сотен тысяч, это избыточно, но для YouTube, где счетчики просмотров исчисляются миллиардами, такой размер оправдан.

Стандартом языка SQL предусмотрен набор базовых типов данных, которые обеспечивают целостность и эффективность хранения информации. Полный перечень типов, включенных в стандарт, наряду с их описанием можно будет найти в приложении к данной главе, здесь же мы рассмотрим самые распространенные из них.

2.2. Типы для целых чисел

Целочисленные типы (INTEGER) используются для хранения количественных данных: товары на складе, дни до дедлайна, лайки под постом. Название происходит от английского *integer* — отсюда сокращение INT в типах данных.

О знаковых и беззнаковых целых в MySQL

Среди рассматриваемых нами СУБД поддержку знаковых и беззнаковых представлений целых имеет только MySQL.

Название «знаковый» происходит от того, что вместе с числом хранится его знак (+ или –), позволяя хранить как положительные, так и отрицательные значения.

Выбор между **знаковым** (SIGNED, от англ. *sign* — знак) и **беззнаковым** (UNSIGNED) представлениями становится значимым на практике довольно редко, тем не менее в определенных случаях позволит сократить размер занимаемых данных. По умолчанию всегда используется знаковый тип.

Так, например, объем в 1 байт имеет емкость 256 значений. В знаковом представлении эти значения могут быть в пределах от –128 до +127, в беззнаковом — от 0 до 255. Если мы знаем, что значение не может (не должно) быть отрицательным и также нам необходимо сохранить значение, к примеру, 200, что невозможно в знаковом представлении, мы можем использовать беззнаковое. В противном случае нам необходимо будет увеличить размер до 2 байт.

Поскольку английский язык был основой для SEQUEL, который стал впоследствии SQL, его знание очень пригодится для того, чтобы различать целочисленные типы интуитивно, так они названы в порядке возрастания размерности:

Таблица 2.1. Целочисленные типы SQL

Тип	Перевод	Размер, байт	Диапазон со знаком (SIGNED)
TINYINT	tiny — крохотный	1	–128 … + 127
SMALLINT	small — маленький	2	–32 768 … +32 767
INT	от integer — целый	4	–2 147 483 648 … 2 147 483 647
BIGINT	big — большой	8	–9 223 372 036 854 775 808 … 9 223 372 036 854 775 807

ПРИМЕЧАНИЕ.

Названия типов приведены в соответствии со стандартом SQL. В документации к СУБД PostgreSQL, например, можно встретить написание INTEGER вместо INT. Ошибки не будет, если вы будете использовать любое написание.

2.3. Типы для действительных чисел

Действительные (REAL) **числа**, которые нельзя представить как целое, — это дроби, проценты, курс валюты или температура воздуха.

Но в базах данных их нельзя хранить «как есть» — компьютеры округляют значения, как мы округляем число π до 3.14. Задача архитектора баз данных — выбрать, **насколько точно** нужно это сделать.

Для представления действительных чисел компьютерными инженерами разработан стандарт IEEE 754, в котором определяется два стандарта хранения чисел «с плавающей точкой» (floating point numbers): **одинарной точности** (*single precision* или просто *single*) или **двойной** (*double precision* или **double**), правила округления и арифметических операций с ними.

Из-за особенностей реализации данного стандарта в компьютерной архитектуре арифметические операции с данными типами не гарантируют точности, поэтому при хранении действительных чисел надо найти компромисс между требуемой точностью и размером, который данные будут занимать в памяти.

Для хранения действительных чисел в SQL предусмотрено два типа.

Таблица 2.2. Типы хранения действительных чисел в SQL

Тип данных	Размер, байт	Примерная точность
REAL	4	до 6 десятичных разрядов после запятой, пример для числа π — 3.1415927
DOUBLE	8	до 15 десятичных разрядов после запятой (π = 3.141592653589793)

Как быть, если нужна абсолютная точность?

Многие практические задачи требуют абсолютной точности. В учетной системе банка мы не можем позволить себе надеяться на то, что компьютер проведет правильное округление и не накопит ошибку, а если мы на бирже, даже сотые и тысячные доли копейки могут оказаться значимыми.

Главное правило: **никогда не использовать REAL или DOUBLE, если надо хранить денежные суммы или цены.**

Здесь нам на помощь приходит тип **NUMERIC** (цифровой), он же **DECIMAL** (десятичный). Появление двух идентичных типов данных в стандарте — результат желания некоторых производителей СУБД учесть предпочтения пользователей и требования рынка, так что во всех практических смыслах типы NUMERIC и DECIMAL идентичны.

Этот тип замечателен тем, что мы можем настроить требуемую нам точность, определив ее в явном виде:

```
NUMERIC(p, s)
```

где:

p — точность (от *precision*) — общее количество знаков в хранимом числе;

s — масштаб (от *scale*) — количество знаков после десятичной точки.

Например, NUMERIC(10, 2) означает, что всего цифр 10, из них 2 знака после запятой, и максимальное значение для чисел с такой заданной точностью 99999999,99. Для биржевых сводок необходима точность побольше, например NUMERIC(15, 8) сможет сохранить число 0.00001523.

Некоторые реализации СУБД позволяют указывать отрицательное значение для *s*, так, например, запись NUMERIC(10, -2) позволит представлять числа с точностью до 10 значащих цифр, из которых две позиции находятся левее предполагаемой десятичной точки. Такое определение может показаться странным, но фактически оно увеличивает разрядность целой части числа. Максимальной положительной величиной для такого определения будет, соответственно, 9999999900, а минимальной — -9999999900. Подобная запись создает дополнительные сложности в понимании и интерпретации данных, поэтому лучше всего выбирать положительное значение масштаба или вообще оставлять его равным нулю, если дробная часть не нужна.

Если не указывать масштаб, например NUMERIC(8), то он принимается равным 0. Можно и вовсе не указывать масштаб и точность, просто указав NUMERIC, но в данном случае разные СУБД интерпретируют это поразному:

SKLADCHIK.ORG

- **PostgreSQL** — по умолчанию принимается неопределенная точность и масштаб, что ведет к автоматическому выбору максимально доступной точности, ограниченной конкретной реализацией конкретной версии СУБД. Масштаб при этом не определен (зависит от реализации).
- **MySQL** — если не указаны точность и масштаб, по умолчанию считается точность равной 10, а масштаб — 0.
- **MS SQL Server** — стандартом считается точность 18 и масштаб 0; если не указаны конкретные значения (в некоторых реализациях MS SQL Server требуется явно указывать масштаб и точность), то такая запись будет генерировать ошибку.

Во избежание проблем с миграцией и для лучшего понимания того, какие именно данные должны храниться в данном типе, рекомендуется всегда явно задавать точность и масштаб.

ВАЖНОЕ ПРЕДОСТЕРЕЖЕНИЕ!

Некоторые СУБД могут предлагать иные типы данных для хранения денежных значений, такие как **MONEY**, **CURRENCY** и пр. Эти типы имеют весьма узкую нишу для применения и обладают рядом ограничений. Перед их использованием тщательно прочитайте документацию и взвесьте все «за» и «против». В большинстве практических случаев для хранения денежных значений лучше использовать тип **NUMERIC**, он же **DECIMAL**.

2.4. Типы для текста

Для хранения текстовой информации используется 2 основных типа: **CHAR** (от *character* — «символ») — для текста, длина которого известна и не изменяется, и **VARCHAR** (от *character varying* — изменяющийся текст). Для поддержки символов Unicode (кириллица, иероглифы) используются расширенные версии **NCHAR** и **NVARCHAR** (буква N от *national* — национальный).

CHAR(n) / NCHAR(n)

Данный тип хранит строки с фиксированной длиной, которая определяется числом n. Например, CHAR(2) хранит строки длиной 2 символа. Максимальный размер составляет приблизительно 2000 символов (точный лимит зависит от конкретной СУБД). Обычно используется для данных с постоянной длиной (обычно это какие-либо коды: коды стран RU, US, DE или, может быть, государственные регистрационные знаки для автомобилей).

CHARACTER VARYING(n) / VARCHAR(n) / NVARCHAR(n)

Этот тип хранит текст неопределенной длины, размером до 65535 символов (зависит от конкретной реализации СУБД).

2.4.1. Про кодировки и Unicode

Хранение текстовой информации на разных языках — не такая тривиальная задача, как может показаться. Разбор всех нюансов в хранении текста выходит далеко за рамки данной книги.

Для хранения текстовой информации стандартом SQL изначально было предусмотрено два основных типа — **CHAR** и **VARCHAR**. CHAR и VARCHAR работали с однобайтовой кодировкой ASCII, что ограничивало их только символами латиницы.

В стандарте ANSI SQL когда-то текстовые данные были представлены в одной единственной кодировке ASCII, где на 1 символ приходился 1 байт. Разумеется, такой стандарт не был совместим с алфавитами других языков. Так, например, эта кодировка не была совместима с кириллицей, не говоря уже о сложных наборах иероглифов китайского или японского языка.

Русскоязычные разработчики, пользуясь литературой на иностранном языке, очень часто сталкиваются с проблемами хранения и отображения

текстовой информации, тогда как для англоязычных авторов проблемы с кодировками бывают не так очевидны.

В эпоху глобализации, когда возникла потребность хранить текстовую информацию на разных языках, типы CHAR и VARCHAR были расширены до NCHAR и NVARCHAR (буква N здесь от *National* — национальный), которые были способны хранить символы Unicode. Платой за универсальность послужил размер данных. В кодировке UTF-8 латинские символы (например, Hello) занимают 1 байт на символ, поэтому VARCHAR и NVARCHAR будут использовать одинаковый объем (5 байт). Но для кириллицы или иероглифов NVARCHAR в UTF-16 займет 2 байта на символ, а VARCHAR в UTF-8 — от 2 до 4 байт.

Современные СУБД (PostgreSQL, MySQL) по умолчанию используют UTF-8, где VARCHAR поддерживает Unicode, однако SQL Server для подобной поддержки требует отдельной настройки. Помните об этом, когда будете проектировать реальные базы данных: если вы точно знаете, что в строковой информации будут использоваться только буквы латинского алфавита из набора ASCII, вы сможете сократить объем занимаемого места для таких данных, просто используя типы CHAR и VARCHAR.

Если вам не так уж критичен объем базы данных, надежнее использовать «национальные» типы NCHAR и NVARCHAR.

Рекомендации

- Используйте NCHAR/NVARCHAR, если:
  - база должна поддерживать несколько языков;
  - вы работаете с **SQL Server** (безопасный выбор);
  - вы рассматриваете миграцию в будущем на другую СУБД.
- Выбирайте CHAR/VARCHAR, если:
  - данные только на латинице (экономия места);
  - вы используете **PostgreSQL/MySQL с UTF-8** (Unicode поддерживается в VARCHAR).

Таблица 2.3.

	CHAR / VARCHAR	NCHAR / NVARCHAR
Кодировка	Зависит от настроек базы	Unicode (UTF16 / UTF8)
Размер символа	1 байт (ASCII) 1–4 байта (UTF-8)	1–4 байта (UTF-8) 2 байта (UTF-16)
Поддержка нац. символов	ASCII: нет UTF-8: да	Да
Поддержка СУБД	Для MS SQL поддержка нац. символов требует доп. настройки	Все

2.4.2. Collation, или головная боль разработчика

Когда мы используем текст только на родном языке, задача по простой сортировке по алфавиту, скажем, имен, кажется нам тривиальной, но сможете ли вы сходу ответить на вопрос о том, как правильно сортировать по алфавиту, к примеру, шведские слова *Älg, Zebra Ödla*?

Если мы попробуем применить общее правило сортировки для Unicode, то слова с этими буквами в начале будут отсортированы так: [«Älg», «Ödla», «Zebra»], так как будет использоваться такой алфавитный порядок: A, Å, Ä, B, C, …, Ö, ..., Z. Но в шведском языке буквы в алфавите расположены в другом порядке (A, B, C, ..., Z, Å, Ä, Ö), и правильным вариантом сортировки будет [«Zebra», «Älg», «Ödla»].

В датском языке буквы Æ ≠ Ä, Ø ≠ Ö, соответственно, *Æble* и *Äble* — это разные слова, и правильной будет такая алфавитная последовательность: Æble, Älg, Øl, Ödla. Если же воспользоваться общими же правилами Unicode, слова *Æble* и *Äble* будут считаться идентичными, а сортировка будет такой: Älg, Æble, Ödla, Øl.

Даже в пределах родного языка необходимо указать базе данных, как, например, различать строчные и заглавные буквы, ведь одну и ту же последовательность можно отсортировать в зависимости от этого либо так: АаБбВв, либо так: АБВабв.

Правила, по которым ваша база будет сортировать и сравнивать текстовые данные, называются *collation* (сопоставление).

В одной и той же базе могут использоваться разные правила сопоставления. В СУБД MySQL и PostgresSQL на этапах обучения вам вряд ли понадобится изменять правила сопоставления, в то же время в некоторых случаях MS SQL может потребовать настройки *collation* для русского языка.

2.5. Дата и время

Может показаться, что хранение даты и времени не представляет особых сложностей, но это ровно до тех пор, пока мы не столкнемся с часовыми поясами, високосными годами или разницей в форматах. Вот базовые типы данных:

- **DATE** — хранит только дату: ГГГГ-ММ-ДД (например, 2023-10-05). Используется для хранения дней рождений, сроков задач, календарных событий.
- **TIME** — хранит только время: ЧЧ:ММ:СС (например, 14:30:00). Подходит для указания времени сеансов в кино, графика работы.
- **TIMESTAMP/DATETIME** — хранит и дату, и время: 2023-10-05 14:30:00. Подходит для записи момента регистрации пользователя, оплаты заказа.

Тип **TIMESTAMP** в разных СУБД ведет себя по-разному:

- для **MYSQL**: при указании TIMESTAMP время при записи автоматически конвертируется в UTC и обратно — при чтении, в то время как DATETIME хранится как есть, без привязки к часовому поясу;

  TIMESTAMP в MySQL работает только до 2038 года — для долгосрочных проектов берите **DATETIME**.

- в **PostgreSQL** для хранения даты и времени с учетом часового пояса используется тип **TIMESTAMP WITH TIME ZONE**, в то время как TIMESTAMP (или *timestamp without time zone*) хранит данные как есть (заданное местное время). Рекомендуется использовать тип с указанием зоны.

- **Microsoft SQL Server** интерпретирует тип TIMESTAMP совсем по-другому, поэтому его использование может вызвать путаницу у пользователей, которые приходят с других СУБД.

  В MS SQL Server **TIMESTAMP** — это не тип для даты и времени, а **бинарное число**, которое автоматически обновляется при изменении строки. Оно используется для отслеживания версий данных (например, чтобы избежать конфликтов при одновременном обновлении). Для хранения времени в SQL Server выбирайте **DATETIME**, **DATETIME2** можно задавать дробные доли секунды), **SMALLDATETIME** (минуты округляются до 0 или 30, например, 14:30:00 вместо 14:29:03, что позволит сэкономить место, если точность до секунды не важна) или **DATETIMEOFFSET** (дата и время со смещением относительно UTC). В других СУБД (MySQL, PostgreSQL) TIMESTAMP действительно хранит дату и время, поэтому не путайте их!

  Для облегчения понимания ниже дана сравнительная таблица 2.4.

Таблица 2.4.

Что хранить	MySQL	PostgreSQL	MSSQL
Время заказа в интернет-магазине	DATETIME	TIMESTAMP	DATETIME (SMALLDATETIME)
Время прилета международного рейса	TIMESTAMP	TIMESTAMP WITH TIME ZONE	DATETIMEOFFSET

Предостережение: никогда не используйте строковые типы данных для хранения даты и времени!

2.6. Логический тип данных (Boolean)

Как следует из названия, способен хранить одно из состояний — либо истину, либо ложь. К сожалению, в разных СУБД определение этого типа происходит по-разному, эту информацию следует иметь в виду:

- **MySQL** — использует название BOOL, которое на самом деле является псевдонимом TINYINT(1), где 1 — необязательная спецификация того, сколько байт надо выделить для хранения этой информации. Принимает значения 1, 0 или NULL (о NULL мы поговорим чуть дальше).

  Помимо 1 или 0, можно также использовать, соответственно, константы TRUE (истина) или FALSE (ложь), вне зависимости от регистра написания.

- **PostgreSQL** — использует название BOOLEAN. Занимает 1 байт, принимает значения:
  - » TRUE (также допустимы значения t, yes, on, 1);
  - » FALSE (или синонимы: f, no, off, 0);
  - » NULL (значение отсутствует).

- **Microsoft SQL Server** — отдельного булева типа не имеет. Вместо этого используется имя **BIT**, но, вопреки названию, занимает целый байт в памяти (или даже больше, если столбцов в таблице больше 8, это надо учитывать при проектировании). Также в MSSQL отсутствуют встроенные константы TRUE и FALSE, вместо них надо использовать числовые значения 1 или 0.

Предостережение: избегайте хранения логических значений в виде строк (TRUE как текстовая строка), это напрасная трата ресурсов и места в хранилище данных.

2.7. NULL: отсутствие значения

В анкете, если ответа на какой-то вопрос не существует (или не может существовать), мы ставим прочерк или оставляем поле пустым. Это значит «данных нет». Значение NULL в SQL имеет такой же смысл. Оно определяет отсутствие данных.

Важно понимать, что NULL — это не 0, не пустая строка и не пробел. Это вообще не значение. Даже если мы попросим СУБД сравнить NULL с

NULL, ответом будет не истина или ложь, а тот же NULL. Это не значение, это его отсутствие.

Если мы допускаем, что каких-то данных может не быть (например, в графе «супруг/супруга»), мы говорим о таких данных, что они обнуляемые (nullable), иными словами, они могут не иметь определенного значения.

Обнуляемым может быть даже логический тип, который может принимать, казалось бы, только значения 1 или 0.

Например, для вопроса анкеты «Согласны ли вы на обработку персональных данных?» хранимыми значениями могут быть 1 (согласен); 0 — (не согласен) и NULL (еще не ответил).

Для целых чисел там, где 0, тоже вполне допустимое и значимое значение (например, неделя беременности), NULL указывает на случаи, когда значения не может быть вовсе (например, если пациент — мужчина).

2.8. Другие типы данных

Теперь вы знакомы с основными типами данных, которые используются в SQL для работы с числами, текстом, датами и логическими значениями. Это фундамент, который поможет вам создавать структуры таблиц и понимать, как хранится информация. Но на этом мир типов данных не заканчивается!

Специализированные типы: для нестандартных задач

Иногда базовых типов недостаточно. Например:

- **Стандарт SQL** предлагает:
  - **INTERVAL** — для хранения промежутков времени («2 часа 30 минут», «3 месяца»). Это удобно для расчетов длительности событий.
  - **XML** — для работы с XML-документами, если ваша система использует такой формат данных.
  - **JSON** — для работы с данными в формате JSON. Хотя официально данный тип данных был включен в стандарт лишь в 2016 году, каждая из рассматриваемых нами СУБД имеет собственную реализацию данного типа.

- В **MySQL** вы встретите:
  - **ENUM** — позволяет создать список допустимых значений (например, статусы заказа: «В обработке», «Отправлен», «Доставлен»).
  - **GEOMETRY** — геометрические типы данных для пространственных данных (точки, линии, полигоны). Позволяют создавать геоприложения, карты и проводить геометрические операции. Благодаря такому типу можно выполнять пространственные запросы и анализировать координаты объектов.

- В **Microsoft SQL Server** есть:
  - **UNIQUEIDENTIFIER** — уникальный идентификатор (UUID/GUID), который гарантирует, что каждая запись будет отличаться даже при объединении данных из разных систем.
  - **HIERARCHYID** — специальный тип для работы с иерархиями (например, структура отделов в компании).

- **В PostgreSQL** добавлены:
  - **JSONB** — улучшенная версия JSON для быстрого поиска внутри документов.
  - **UUID** — аналог **UNIQUEIDENTIFIER** для MS SQL.
  - **ARRAY** — позволяет хранить списки значений (например, теги к статье: [«техника», «советы», «SQL»]).

PostgreSQL обладает самым широким списком расширенных типов данных для самых разнообразных задач, поэтому, если вам необходимо хранить что-то необычное, для начала посмотрите, не реализован ли нужный вам тип в PostgreSQL.

2.9. Что дальше?

Мы научились использовать корректные типы данных для разных видов значений. Ниже вы найдете удобную таблицу со всеми типами данных, которые мы разобрали, включая их краткое описание и примеры использования.

А сейчас — давайте создавать базы данных!

2.10. Приложение к главе 2: типы данных стандарта SQL

В таблице перечислены только стандартные типы SQL, специфичные для СУБД (например, SERIAL, TINYINT) опущены.

Размеры указаны в соответствии с реализацией стандарта. Для типов с переменным размером (например, VARCHAR) указан максимальный предел.

Примеры использования приведены для распространенных сценариев.

Особенности реализации выделяют ключевые различия между СУБД.

Тип данных	Размер (если применимо)	Примеры использования	Особенности реализации (MySQL)
CHARACTER(n) синоним: CHAR(n)	Фиксированный (n)	Код страны (CHAR(2)), Статус (CHAR(1))	MySQL: максимум 255 символов. PostgreSQL: допускает CHAR без указания длины (по умолчанию 1). MS SQL: максимум 8000 символов
CHARACTER VARYING(n) синоним: VARHAR(n)	Максимальный (n)	Имя (VARCHAR(50)), Описание (VARCHAR(255))	MySQL: максимум 65535 символов. PostgreSQL: поддерживает TEXT для неограниченного размера. MS SQL: максимум 8000 символов (VARCHAR(max) для больших данных)
NATIONAL CHARACTER(n) синоним: NCHAR(n)	Фиксированный (n)	Локализованный текст (NCHAR(10))	MySQL: аналог CHAR с кодировкой utf8mb4. PostgreSQL: не поддерживается (используйте CHAR с указанием COLLATE). MS SQL: аналог CHAR с кодировкой NVARCHAR
NATIONAL CHARACTER VARYING(n) синоним: NVARCHAR(N)	Максимальный (n)	Многоязычный текст (NVARCHAR(100))	MySQL: аналог VARCHAR с utf8mb4. PostgreSQL: не поддерживается (используйте VARCHAR с COLLATE). MS SQL: стандартный тип для Unicode (NVARCHAR)
INTEGER синоним: INT	4 байта	ID пользователя (INT), Количество (INT)	MySQL: INT(11) по умолчанию. PostgreSQL: INT4 (синоним INTEGER). MS SQL: INT (диапазон от -2^31 до 2^31-1)

Тип данных	Размер (если применимо)	Примеры использования	Особенности реализации (MySQL)
SMALLINT	2 байта	Возраст (SMALLINT), Рейтинг (SMALLINT)	MySQL: диапазон от -32768 до 32767. PostgreSQL: INT2 (синоним SMALLINT). MS SQL: диапазон от -32768 до 32767
BIGINT	8 байт	Крупные суммы (BIGINT), Уникальные идентификаторы (BIGINT)	MySQL: диапазон от -2^63 до 2^63-1. PostgreSQL: INT8 (синоним BIGINT) MS SQL: диапазон от -2^63 до 2^63-1
DECIMAL(p, s) синоним: NUMERIC(p,s)	Зависит от p (точности) и s (масштаба)	Цена (DECIMAL(10,2)), Научные расчеты (DECIMAL(20,10))	MySQL: DECIMAL(p, s) без ограничений по умолчанию. PostgreSQL: точность до 1000 цифр. MS SQL: DECIMAL(p, s) (макс. p = 38)
REAL	4 байта	Температура (REAL), Координаты (REAL)	MySQL: аналог FLOAT(24). PostgreSQL: REAL (6 значащих цифр). MS SQL: FLOAT(24) (7 значащих цифр)
DOUBLE PRECISION синоним: DOUBLE	8 байт	Сложные вычисления (DOUBLE), Статистика (DOUBLE)	MySQL: аналог FLOAT(53). PostgreSQL: DOUBLE PRECISION (15 значащих цифр). MS SQL: FLOAT(53) (15 значащих цифр)
FLOAT(p)	Зависит от p	Универсальные числа с плавающей точкой	MySQL: FLOAT(p) (p <= 53). PostgreSQL: реализуется как DOUBLE PRECISION. MS SQL: FLOAT(p) (p <= 53)
DATE	3 байта	Дата рождения (DATE), Срок действия (DATE)	MySQL: формат YYYY-MM-DD. PostgreSQL: поддерживает диапазон от 4713 г. до н.э. MS SQL: диапазон от 0001-01-01 до 9999-12-31
TIME(p)	3–5 байт	Время события (TIME), Продолжительность (TIME)	MySQL: формат HH:MM:SS[.доли]. PostgreSQL: поддерживает TIME WITH TIME ZONE. MS SQL: точность до 100 наносекунд
TIMESTAMP(p)	8 байт	Метка времени (TIMESTAMP), Логирование (TIMESTAMP)	MySQL: автоматическое обновление при вставке. PostgreSQL: включает дату и время с таймзоной. MS SQL: TIMESTAMP (может конфликтовать с rowversion)

Тип данных	Размер (если применимо)	Примеры использования	Особенности реализации (MySQL)
INTERVAL	Зависит от типа	Периоды времени (INTERVAL YEAR TO MONTH)	MySQL: не поддерживается. PostgreSQL: INTERVAL YEAR TO MONTH / DAY TO SECOND. MS SQL: не поддерживается (используйте DATEDIFF)
BINARY(n)	Фиксированный (n)	Хеши (BINARY(32)), Шифрованные данные	MySQL: максимум 255 байт. PostgreSQL: не поддерживается (используйте BYTEA). MS SQL: максимум 8000 байт
BINARY VARYING(n)	Максимальный (n)	Динамические бинарные данные (VARBINARY(100))	MySQL: максимум 65535 байт. PostgreSQL: не поддерживается (используйте BYTEA). MS SQL: VARBINARY(max) для больших данных
BLOB	Изменяемый	Файлы (BLOB), Изображения (BLOB)	MySQL: LONGBLOB (до 4 ГБ). PostgreSQL: BYTEA (неограниченный размер). MS SQL: VARBINARY(max) (до 2 ГБ)
BOOLEAN	1 байт	Флаги (BOOLEAN), Статус (TRUE/FALSE)	MySQL: синоним TINYINT(1) (0/1). PostgreSQL: отдельный тип BOOLEAN. MS SQL: BIT (0/1, но не полностью соответствует стандарту)
XML	Изменяемый	XML-данные (XML), Конфигурации (XML)	MySQL: поддерживает XML-валидацию. PostgreSQL: XML с проверкой DTD. MS SQL: XML (поддержка XSD-схем)
JSON	Изменяемый	JSON-объекты (JSON), Настройки (JSON)	MySQL: JSON-валидация с MySQL 5.7+. PostgreSQL: JSONB (бинарный формат для оптимизации). MS SQL: JSON (поддержка с SQL Server 2016)

Глава 3.

Основы проектирования баз данных на практике

https://t.me/IT_Portal

3.1. Табличное представление

В реляционных СУБД, таких как MySQL, PostgreSQL и Microsoft SQL Server, любые данные представлены в виде **таблиц**.

Табличное представление люди стали использовать, судя по археологическим раскопкам, еще в древней Месопотамии, где торговцы записывали количество зерна, масла или скота в столбцы, чтобы следить за запасами. Это были первые "базы данных" на физических носителях.

В средние века купцы использовали таблицы для записи доходов, расходов и долгов. Двойная запись в бухгалтерии (дебет/кредит) — прямой предок современных отношений между таблицами.

Некоторое время из-за крайне высокой стоимости компьютерной памяти, организовывать данные в виде таблиц в компьютерах казалось крайне расточительной затеей, даже создание каталога данных требовало места под хранение этого каталога, что приводило к удорожанию стоимости хранения. В 1970 году британский ученый **Эдгар Кодд** предложил, несмотря на дороговизну, использовать табличное представление данных и в компьютерах, так как не такое уж и значительное увеличение стоимости позволяло сократить скорость поиска необходимых данных в десятки раз, а также появлялась возможность изменять существующие данные без риска сломать всю систему.

Вот и сейчас почти в любой сфере деятельности мы используем данные в таблицах.

3.2. Пример 1. Интернет-магазин. Нормализация

Представим, мы устроились на работу в небольшой интернет-магазин, и менеджер нам показывает список текущих заказов в таблице Excel, которая выглядит примерно так:

Район доставки	Имя клиента	e-mail клиента	Адрес доставки	Артикул, наименование	Цена	Кол-во
Юг	Кольцов П.А.	adjutant@ white.mov	ул. Центральная, д. 12, кв. 23	002, Ручка шариковая	199,50	12
Запад	Исаев М. М.	stirlitz@ hustpo.ger	ул. Цветочная, д. 6, кв. 8	006, Блокнот, 96 листов	80,00	4
Запад	Белов А. А.	j.weiss@ abwehr.ger	ул. Тополиная, д. 21, кв. 116	007, Бумага копировальная, пачка 10 листов	260,00	2
Центр	Тульев М. А.	resident@ cia.org	ул. Ленинградская, д. 16, кв. 72	006, Блокнот, 96 листов	72,00	12
Восток	Славин В. В.	nagonia@ tass.age	ул. Горького, д. 84, кв. 307	002, Ручка шариковая	210,00	3
Юг	Кольцов П. А.	adjtant@ white.mov	ул. Центральная, д. 12, кв. 23	001, Арифмометр "Феликс", реплика, Китай	4 400,00	1

Выглядит здорово! Вся информация о каждом заказе собрана в одном месте, не надо ничего искать в других источниках.

Но давайте приглядимся внимательно:

- Клиент Кольцов П. А. повторяется дважды, но, если человеку и кажется, что записи "Кольцов П.А." и "Кольцов П. А." идентичны, компьютер посчитает их разными, так как во втором случае между инициалами вставлен пробел, кроме того, во втором случае есть еще и опечатка в адресе электронной почты, клиент не получит уведомление о втором заказе.
- Если кто-то из клиентов изменит адрес электронной почты, придется заменять во всей таблице.

- Если клиент сделает 100 заказов, имя и адрес электронной почты повторятся 100 раз. Это бесполезный "лишний вес" в таблице.
- Помимо артикула, в каждой строчке сохранено еще и наименование товара, также добавляя ненужный объем к размеру таблицы.

Мы столкнулись с классическим примером **ненормализованных** данных: вся информация хранится в одной таблице, смешивая разные сущности (клиентов, заказы, товары), и это создает проблемы.

Как распознать ненормализованные данные?

1. **Повторяющиеся значения:**
   - » В таблице много дублирующихся данных (например, имя и e-mail клиента в каждом заказе).
2. **Столбцы — "мусорные ведра":**
   - » В таблице есть поля, которые логически относятся к разным объектам (клиенты + товары + заказы).
3. **Сложность изменений:**
   - » Чтобы обновить одну запись (например, исправить e-mail клиента), нужно менять ее в десятках строк.

3.3. Формы нормализации

В нашем случае, чтобы избавиться от найденных проблем, нам потребуется разделить (**нормализовать**) одну большую таблицу на несколько маленьких, каждая из которых отвечает за одну **сущность**: **Клиенты**, **Товары**, **Заказы**.

Нормализация данных — это процесс реорганизации структуры базы данных для устранения избыточности и улучшения целостности данных. Для проектирования нормальной структуры таблиц, проектировщики руководствуются т. н. "**нормальными формами**" (Normal Forms, NF).

Первая нормальная форма (1NF)

Первая нормальная форма (First Normal Form, 1NF) требует, чтобы каждая ячейка таблицы содержала единственное атомарное значение. Другими словами, данные должны быть разбиты на самые мелкие составляющие элементы и не должно быть массивов или множественных значений в одной ячейке, или, проще говоря, **в каждой таблице присутствуют только однозначные значения**.

Вторая нормальная форма (2NF)

Вторая нормальная форма (Second Normal Form, 2NF) строится поверх 1NF и требует, чтобы все неключевые атрибуты зависели только от идентификатора (№ заказа, ИД клиента и т. д.; забегая вперед, запомните, что такие идентификаторы называются **первичными ключами**), а не от отдельных частей составного ключа. Это устраняет частичную зависимость, простыми словами: **все неключевые атрибуты зависят от полного первичного ключа**.

Третья нормальная форма (3NF)

Третья нормальная форма (Third Normal Form, 3NF) основана на 2NF и утверждает, что никакие неключевые атрибуты не должны зависеть от других неключевых атрибутов. Это избавляет таблицу от транзитивных зависимостей. Простыми словами: **никакие неключевые атрибуты не зависят друг от друга.**

Давайте применим последовательно все три формы нормализации к нашей таблице. Обычно опытный архитектор проводит все этапы одновременно, но мы для наглядности будем преобразовывать нашу таблицу постепенно.

Цель 1NF: убрать повторяющиеся группы, обеспечить атомарность данных.

1. Создадим поле "Номер заказа", чтобы каждый заказ можно было однозначно определить по единственному значению.

2. Столбец "Артикул, наименование товара" страдает отсутствием атомарности, надо определить артикул и наименование товара в разные столбцы.
3. Вынесем товары в отдельную таблицу, так как один заказ может содержать несколько товаров.

Таблица **Заказы** (Orders):

Номер заказа	Район доставки	Имя клиента	e-mail клиента	Адрес доставки

Таблица **Параметры заказов** (OrderDetails):

Номер заказа	Артикул	Наименование	Цена	Количество

Мы убрали повторяющиеся группы, каждая строка в таблице **Параметры заказов** (OrderDetails) соответствует одному значению в таблице **Заказы** (Orders).

Цель 2NF: устранить частичные зависимости (поля в таблице должны зависеть от всего первичного ключа).

1. Выносим клиентов в отдельную таблицу, так как их данные зависят только от клиента, а не от заказа.
2. Выносим товары в отдельную таблицу, так как их наименование и цена зависят от артикула, а не от заказа.
3. Вводим идентификаторы "Ид клиента" и "Ид товара" для однозначной идентификации клиентов и товаров.

Таблица **Клиенты** (Customers):

Ид клиента	Имя клиента	E-mail клиента	Адрес доставки	Район доставки

Таблица **Товары** (Products):

Ид товара	Артикул	Наименование	Цена

Таблица **Заказы** (Orders) у нас упрощается до следующего вида:

Номер заказа	Ид клиента

Таблица **Параметры заказа** (OrderDetails) тоже претерпела изменения:

Номер заказа	Ид товара	Количество

Теперь данные товаров и клиентов не дублируются, а цена хранится в одном месте (в **Products**).

Но не все еще хорошо. В таблице **Customers** есть зависимость **Район доставки → Адрес доставки** (если район определяется адресом, это нарушает 3NF).

Цель 3NF: устранить транзитивные зависимости (поля не должны зависеть от других неключевых полей).

С этой целью мы выносим районы доставки в отдельную таблицу, так как район зависит от адреса, а не от клиента, создав таблицу **Районы доставки** (DeliveryAreas).

Адрес	Район доставки

Соответственно, мы убираем это поле из таблицы **Клиенты** (Customers):

Ид клиента	Имя клиента	E-mail клиента

Район доставки больше не хранится в **Customers** — он определяется через адрес.

На этом этапе мы успешно провели все три формы нормализации.

А все ли мы учли?

Окидывая напоследок взглядом исходную таблицу, мы замечаем один странный факт: цены на одни и те же товары разнятся от заказа к заказу. Мы спрашиваем менеджера об природе этого несоответствия, и он говорит нам,

что действительно, если клиент заказывает более 10 товаров одного артикула, к этому заказу применяется скидка. Так, для шариковых ручек (артикул 002) применяется скидка 5% при заказе от 10 штук, а для блокнотов (артикул 006) — 10% при количестве от 5 штук.

Ход наших рассуждений такой: скидка индивидуальна для товара, соответственно, зависит от товара и от заказа, так как скидка для разных заказов будет разной.

Получается, сперва нам надо дополнительно создать таблицу с правилами применения скидок (**Discount**), которая может выглядеть примерно так:

Артикул	Мин. количество	Процент скидки
002	10	0,05
006	5	0,1

Таблица **Параметры заказа** (OrderDetails) получила дополнительное поле с информацией о скидке.

Номер заказа	Ид товара	Количество	Примененная скидка

Наконец, сейчас структура данных выглядит как надо. В качестве самостоятельной работы подумайте, какие изменения надо будет внести в эту структуру, если потом выяснится, что скидки регулируются маркетинговыми акциями и могут применяться только в пределах срока действия таких акций.

3.4. Создание базы данных, проектирование таблиц

Давайте создадим базу данных для нашего магазина (для начала мы создадим более упрощенную структуру без учета скидок, так как их использование предполагает более глубокое знание материала, но мы вернемся к скидкам позднее). Для создания чего бы то ни было в SQL используется инструкция, начинающаяся с инструкции **CREATE** (создать). Для начала да-

вайте создадим базу данных (database). Для этого просто выполним инструкцию:

```
CREATE DATABASE shop;
```

После успешного выполнения команды мы должны сообщить СУБД, что далее мы хотим работать именно с этой базой данных (так как баз может быть несколько). Для этого мы должны выполнить инструкцию USE <имя базы данных> (для MySQL или MSSQL):

```
USE shop;
```

Если вы работаете в PosgreSQL, вам надо подключиться к базе, используя команду подключения \c (connect):

```
\c shop;
```

Альтернативно вы можете подключаться к базе напрямую, если будете запускать **psql** с указанием базы данных (ключ *-d*):

```
psql -d shop
```

Теперь СУБД будет знать, что все последующие команды, которые мы будем вводить, должны применяться к базе данных **shop**.

ПРИМЕЧАНИЕ.

Чтобы удалить какой-то объект из базы данных, необходимо будет воспользоваться инструкцией DROP (выбросить). Используйте с осторожностью, потому что никаких подтверждений вашего намерения при выполнении этой команды СУБД спрашивать не будет!

Теперь давайте вспомним все, что мы узнали о типах данных, и посмотрим, какие типы данных мы должны использовать для каждой таблицы. При проектировании следуйте простому правилу — использовать самый простой тип данных, который вы только можете себе позволить.

Таблица **Orders** (Заказы):

Столбец	Тип данных	Обнуляемость	Пояснения
id	int	НЕТ	Нам достаточно одного целого числа. Идентификатор заказа не может быть пустым (NULL), иначе мы его не сможем найти
customerid	int	НЕТ	Аналогично, присвоив номер покупателю, мы сможем всегда однозначно его определять. Точно так же это поле не может иметь значение NULL, так как у заказа обязательно должен быть покупатель

ПРИМЕЧАНИЕ.

Строго говоря, для магазинов с небольшим количеством клиентов может подойти и тип SMALLINT (до 32 тыс. положительных значений, 65 тыс. всего), однако всегда есть "риск" расширения бизнеса, кроме того, даже для миллиона записей мы получим экономию места всего в $2 * 10^6 \approx 2$ Мб, что не стоит усилий. Кроме того, быстрее всего современные процессоры обрабатывают 32-битные (4 байта) или 64-битные (8 байт) целые числа, это связано с шириной системной шины, поэтому при подборе целочисленного типа данных в качестве идентификатора (первичного ключа) лучше всего всегда выбирать **int** (для большинства случаев) или **bigint** (если количество значений пойдет на миллиарды).

Поле **id** здесь является **первичным ключом** (**primary key**, или сокращенно **PK**). Первичный ключ должен удовлетворять следующим требованиям:

- Он должен быть **уникальным** идентификатором, то есть ни разу не повторяться, однозначно идентифицируя запись в таблице.
- Он обязателен — поле первичного ключа не может быть пустым (**NULL**).

Первичные ключи — основа любой СУБД. Данные специально организовываются таким образом, чтобы поиск по значению первичного ключа происходил максимально быстро. Сложность поиска составляет O(log n), где *n* — количество записей. Если данных в таблице немного и они носят вспомогательный характер, вы можете в отдельных редких случаях обходиться без первичного ключа. Не будет ошибкой всегда в любой таблице определять поле, которое будет являться первичным ключом.

Поле первичного ключа, соответственно, должно занимать как можно меньший размер в памяти для экономии места. Целое число для идентификации заказов и покупателей в нашем случае прекрасно подходит для этой цели.

Таблица **Клиенты** (Customers):

Столбец	Тип данных	Обнуляемость (nullability)	Пояснения
id	int	НЕТ	Это **первичный ключ** для таблицы. Соответственно, он не может быть NULL
Имя клиента **name**	nvarchar(100)	НЕТ	Имена могут иметь разную длину, но, наверное, 100 знаков будет достаточно, чтобы записать самое сложное имя. В базу данных будут заноситься имена со слов клиента, однако какое-то имя должно быть, поэтому NULL-значения мы здесь не разрешаем
E-mail клиента **email**	nvarchar(50)	ДА	Для большинства e-mail длины в 50 знаков может быть достаточно. В то же время у клиента может не быть адреса электронной почты или он может не захотеть сообщить его, поэтому данное поле может быть пустым
Адрес **address**	nvarchar(300)	ДА	Не скупимся на длину, адреса могут быть очень длинными, поэтому отводим целых 300 знаков, чтобы уместить даже самый сложный и длинный адрес. Клиент может не предоставлять свой адрес, предпочитая самовывоз, поэтому данное поле тоже может быть пустым

Таблица **Товары** (Products):

Столбец	Тип данных	Обнуляемость (nullability)	Пояснения
id	int	НЕТ	**PK** (Первичный ключ)
Артикул **article**	nvarchar(10)	НЕТ	* см. примечание
Наименование **name**	nvarchar(300)	НЕТ	Будем щедры, выделим до 300 знаков для хранения наименования товара. Разумеется, название должно быть, поэтому пустых значений мы здесь не допускаем
Цена **price**	numeric(8,2)	НЕТ	Сомнительно, чтобы в магазине канцтоваров базовая цена на какой-либо товар будет выше миллиона рублей. Мы указываем, что всего значащих разрядов у нас будет 8, из них 2 — после запятой, поэтому максимально допустимым значением здесь будет 999999.99

ПРИМЕЧАНИЕ.

Существует 2 "философских" школы учений по поводу того, стоит ли использовать в качестве первичных ключей "натуральные" или "искусственные" данные. Поклонники первой в данном случае сказали бы, что в качестве первичного ключа в этой таблице следует использовать поле "Артикул" (натуральные данные, уже присутствующие в наборе), однако давайте рассмотрим такое решение более подробно.

Если внутри одного артикула появляются вариации (цвет, размер), артикул перестает быть уникальным.

Это делает его непригодным для первичного ключа, так как нарушается главное правило: *одна запись = один уникальный идентификатор*.

Например:

Артикул: А-123
Товар: Футболка
Цвет: Красный

Артикул: А-123
Товар: Футболка
Цвет: Синий

Артикул, кроме того, может быть изменен в будущем (например, из-за реорганизации номенклатуры), что вызовет необходимость в каскадном обновлении всех таблиц, где используются артикулы.

Мы предпочли использовать искусственные данные (искусственно создали дополнительное поле с кодами товаров, никак не привязанные к "натуральным" данным).

Этот выбор позволяет в долгосрочной перспективе большую гибкость за счет незначительного увеличения объема хранимых данных, что в данном случае — вполне приемлемая цена за гибкость.

Искусственный ключ не зависит от бизнес-логики (артикул, название). Даже если артикул изменится или появится новая вариация товара, структура базы останется стабильной, так как искусственный ключ остается неизменным.

Кроме того, числовые идентификаторы (INT, BIGINT) работают быстрее в операциях соединения и индексирования, чем строковые ключи (артикул). Это важно для больших баз данных.

Таблица **Параметры заказа** (OrderDetails):

Столбец	Тип данных	Обнуляемость (nullability)	Пояснения
Номер заказа **orderid**	int	НЕТ	**PK** (Первичный ключ). Да, мы вполне можем использовать несколько значений для формирования первичного ключа. Поскольку по отдельности ни заказ, ни товар не могут однозначно определять запись в данной таблице, зато могут это делать в совокупности, мы используем комбинацию этих столбцов в качестве первичного ключа
Ид товара **productid**	int	НЕТ	
Количество **quantity**	int	НЕТ	Мы используем **int** для указания количества. Можно, разумеется, использовать и **smallint**, но вдруг нам попадется особо крупный оптовый заказ и емкости **smallint** не хватит. Конечно же, NULL здесь быть не может

ПРИМЕЧАНИЕ.

Добавление искусственной сущности здесь не оправдано, потому что у нас, в отличие от таблицы **Products**, уже имеется набор значений, который однозначно определяет строку в таблице, а главное правило при проектировании баз данных созвучно знаменитому принципу "Бритвы Оккама", который звучит: **не плоди сущности сверх необходимости!**

В то же время могут возникнуть ситуации, когда наличие отдельного первичного ключа в данной таблице может стать необходимым. Например, если для работы с базой будет использована какая-либо система ORM (допустим, Hibernate) или присутствуют каскадные связи, где проще ссылаться на одно

поле, а не на несколько, но эти аспекты относятся к более глубоким концепциям, чем те, что рассматриваются в данной главе.

3.5. CREATE TABLE

Мы спроектировали структуру таблиц и теперь приступаем к их непосредственному созданию.

Как уже было сказано, для создания чего бы то ни было в СУБД используется инструкция CREATE. Мы уже создали базу данных, теперь создадим таблицу заказов:

```
CREATE TABLE orders(
  id INT PRIMARY KEY,
  customerid INT NOT NULL
);
```

Сначала мы пишем инструкцию **Создать таблицу** (CREATE TABLE), дальше пишем название таблицы (orders), потом в скобках перечисляем все столбцы, которые СУБД должна будет добавить в эту таблицу. Перечисление столбцов идет через запятую. Завершается команда символом точки с запятой (после нее можно писать следующую команду).

Внутри мы пишем имя столбца, потом тип данных: id INT, далее пишем, что этот столбец является первичным ключом для таблицы, для этого добавляем ключевые слова **PRIMARY KEY**. Традиционно столбец первичного ключа указывается первым, но, строго говоря, это не обязательно. Поскольку одним из требований к первичному ключу является уникальность и необнуляемость, дополнительно ничего сообщать не требуется.

Описание следующего столбца также начинается с его названия и типа данных: customerid INT, после чего мы указываем **ограничение** (constraint): NOT NULL (не обнуляемый). Если бы мы хотели, чтобы столбец был обнуляемым, можно было бы указать NULL или не указывать ничего (все столбцы считаются обнуляемыми по умолчанию, если не указано обратное, тем не менее для лучшего понимания и наглядности рекомендуется все равно указывать ключевое слово NULL, хоть это и не обязательно). В SQL существуют и другие виды ограничений, они упомянуты в конце главы.

Инструкции можно вводить в несколько строк (именно поэтому в оболочках командной строки нужно завершать команду точкой с запятой), при этом вы можете разбивать команду на любое количество строк, СУБД попросту игнорирует перенос на другую строку.

3.5.1. О верхнем и нижнем регистре

Примечания к синтаксису: в листинге SQL-выражений ключевые слова используют верхний регистр. Строго говоря, это необязательно. Традиция использовать верхний регистр пошла еще с тех времен, когда в компьютерных терминалах такого понятия, как регистр букв, еще не было. Абсолютно не обязательно использовать заглавные буквы при указании ключевых слов. Никакой разницы не будет, если вы напишете:

```
create table orders(
  id int primary key,
  customerid int not null
);
```

Что же касается объектов (имен таблиц, столбцов и пр.), ситуация немного сложнее. Современный SQL поддерживает Unicode, поэтому вы можете использовать абсолютно любые символы для названий объектов (даже эмодзи, если хотите). Тем не менее, стремитесь делать названия объектов понятными и лаконичными: с одной стороны, объект должен иметь название, по которому другой человек должен получить представление о том, что именно содержится внутри, а с другой — оно должно быть как можно короче. Думайте о людях (и о себе в будущем), которым придется набирать названия вручную.

ПРИМЕЧАНИЕ.

Придерживайтесь одного стиля в именовании объектов (таблиц и столбцов) — либо используйте нижний регистр, либо заглавный. Иногда может возникнуть желание использовать именования в стиле т.н. CamelCase (OrderId, ProductId и пр.), но в силу того, что разные СУБД имеют разные настройки по умолчанию, зависимые от регистра, это может привести к ошибкам. Очень распространенная ошибка — это путаница в написании названий типа OrderID или OrderId.

Название объекта (таблицы, столбца, индекса, отношения и пр.) может, строго говоря, содержать и пробелы, и состоять из нескольких слов. Однако пользоваться такой возможностью не стоит по целому ряду причин. Во-первых, использование названий, состоящих из нескольких слов, ухудшает понимание кода, во-вторых, синтаксис использования подобных названий в разных СУБД разный, и это серьезно усложнит процесс миграции на другую СУБД. В самом крайнем случае старайтесь пользоваться символом подчеркивания, например order_id.

Ниже приведена сводная информация по тому, как разные СУБД воспринимают регистр и пробелы в названиях:

MySQL

- **Регистрозависимость**: имена столбцов в **MySQL** нечувствительны к регистру по умолчанию. То есть столбцы с именами **Name** и **NAME** считаются одинаковыми.
- **Синтаксис**: столбцы, состоящие из нескольких слов, заключаются в обратные апострофы (') для защиты от ключевых слов и пробелов.

  Например:

```
CREATE TABLE users (
  'User Name' VARCHAR(100),
  'Email Address' VARCHAR(255)
);
```

  Обратные апострофы необязательны, если имя столбца не содержит пробелы или ключевые слова SQL.

PostgreSQL

- **Регистрозависимость**: PostgreSQL не чувствителен к регистру с оговорками. Если имя указано без обрамляющих двойных кавычек (""), столбцы **Name** и **NAME** рассматриваются как одинаковые имена столбцов. В то же время, если указать имя в кавычках, **"Name"**и **"NAME"** будут считаться разными именами. Если кавычки не указаны, PostgreSQL всегда преобразует имя к нижнему регистру.

- **Синтаксис**: для многосоставных названий столбцов или для имен, содержащих ключевые слова, используется фигурные кавычки ("").

  Пример:

```
CREATE TABLE users (
  "User Name" VARCHAR(100),
  "Email Address" VARCHAR(255)
);
```

  Названия столбцов без пробелов и спецсимволов не обязательно заключать в кавычки.

MS SQL Server

- **Регистрозависимость**: настраивается на уровне базы данных. По умолчанию имена столбцов нечувствительны к регистру, но можно настроить чувствительность к регистру при создании базы данных.
- **Синтаксис**: многосоставные имена столбцов берутся в квадратные скобки ([]). Пример:

```
CREATE TABLE users (
  [User Name] VARCHAR(100),
  [Email Address] VARCHAR(255)
);
```

  Скобки обязательны, если имя содержит пробелы или совпадает с ключевыми словами SQL.

Мы будем придерживаться соглашения, что все объекты будут именоваться в нижнем регистре, и не использовать внутри пробелы или спецсимволы.

Таблица Customers (Клиенты)

Создаем таблицу:

```
CREATE TABLE customers(
  id INT PRIMARY KEY,
  name NVARCHAR(100) NOT NULL,
```

SKLADCHIK.ORG


```
  email NVARCHAR(50) NULL,
  address NVARCHAR(300) NULL
);
```

ПРИМЕЧАНИЕ.

Если вы работаете в PostgreSQL, используйте VARCHAR вместо NVARCHAR, так как PostgreSQL по умолчанию поддерживает Unicode и в ней отсутствует тип NVARCHAR.

Мы определили таблицу покупателей, в котором определены столбцы **id** в качестве первичного ключа, необнуляемое имя клиента длиной до 100 символов, а также обнуляемые поля: адрес электронной почты длиной до 50 символов и адрес длиной до 300. Как уже говорилось, указание NULL не является обязательным и указывается только для наглядности.

В листингах CREATE TABLE часто последнюю закрывающую скобку выносят на отдельную строку, чтобы описание столбцов было легко читать или изменять впоследствии.

Таблица Products (Товары)

```
CREATE TABLE products (
  id INT PRIMARY KEY,
  article NVARCHAR(10) NOT NULL,
  name NVARCHAR(300) NOT NULL,
  price NUMERIC(8, 2) NOT NULL
);
```

Таблица OrderDetails (Параметры заказа)

```
CREATE TABLE orderdetails (
  id INT,
  productid INT,
  quantity INT NOT NULL,
  PRIMARY KEY(id, productid)
);
```

В данном запросе мы используем другой синтаксис для указания первичного ключа. Конструкция **PRIMARY KEY** указана отдельно после перечисления всех столбцов. В скобках мы указываем имена столбцов, комбинацию которых мы хотим использовать в качестве первичного ключа.

3.5.2. Работа в графических оболочках

Какую бы графическую оболочку вы ни использовали, они все имеют схожий интерфейс:

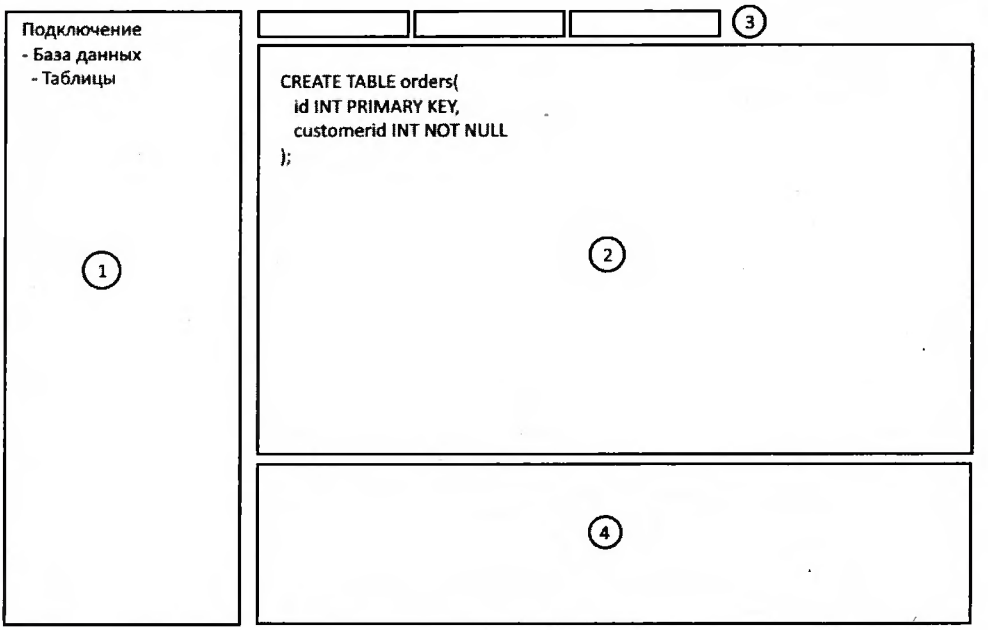


Рис. 3.1. *Структура интерфейса графической оболочки*

- Слева (обозначено цифрой 1) расположено иерархическое представление всех объектов СУБД, к которой вы подключены. Обычно эта область называется **Object Explorer** (Исследователь объектов) или **Navigator**. Здесь вы увидите список серверов СУБД, о которых известно оболочке, внутри каждого сервера будет список баз данных внутри этого сервера, уровнем ниже — таблицы, представления и прочие объекты базы данных.

Если вы уже создали все 4 таблицы, то вы увидите их в разделе **Tables** (Таблицы) внутри базы данных **shop**. Если вы не видите наши таблицы или базу, попробуйте щелкнуть правой кнопкой мыши по серверу СУБД и в появившемся меню выберите пункт **Refresh** (Обновить).

- Под цифрами 2 и 3 обозначена главная рабочая область (2) и вкладки (3). Вы можете одновременно работать с разными объектами и писать разные запросы. При помощи вкладок вы легко можете переключаться между рабочими областями.
- В области 4 обычно отображаются результаты ваших запросов и перечень ошибок, если они возникли.
- В верхней части окна расположено меню, в котором вы можете найти пункт **New Query** (Новый запрос). В зависимости от выбранной оболочки интерфейс может иметь особенности, однако общее расположение областей сохраняется неизменным, вне зависимости от того, какую именно графическую оболочку вы используете.

3.6. Добавляем отношения, ограничения и внешние ключи

Мы создали 4 таблицы для наших данных (см. рис. 3.2.).

При создании такой структуры мы подразумевали, что в качестве **productid** в таблице **orderdetails** мы будем указывать идентификатор одной из строк в таблице **products**, а в качестве **customerid** в таблице **orders** мы будем указывать идентификатор, отсылающий нас к определенной строке в таблице **customers**. Но сейчас ничего не мешает нам указать в любом из этих полей ссылку на несуществующую запись, ведь СУБД пока не знает наших намерений. Мы должны рассказать ей об **отношениях** между таблицами.

Вы могли уже сталкиваться со словом "реляционная" применительно к СУБД. Само слово "реляционные" происходит от английского ***relation*** — "**отношение**". Это не случайно. В реляционных базах данных (как PostgreSQL, MySQL, MSSQL и других) информация хранится в таблицах, которые могут быть связаны между собой через **отношения**. Эти связи и превращают разрозненные таблицы в единую логичную систему, где данные "общаются" друг с другом и зависят друг от друга.

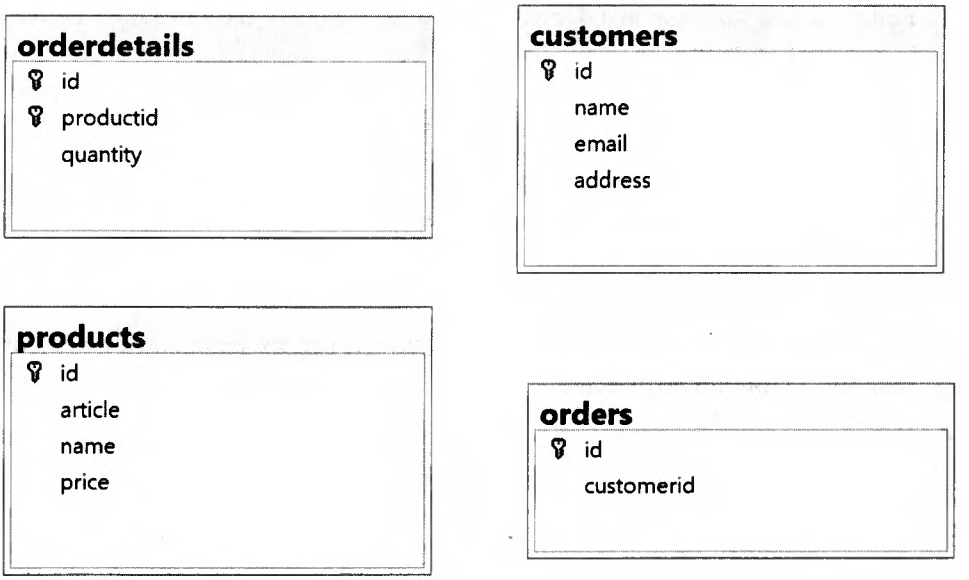


Рис. 3.2. *Диаграмма созданных таблиц*

Для установления отношения между таблицами служит такой инструмент, как **внешний ключ**. Мы уже неявно использовали его, когда проектировали таблицы для нашего магазина. Внешний ключ — это значение, которое ссылается на другой первичный ключ в другой таблице. В таблице **orderdetails** мы используем внешний ключ **productid**, *ссылающийся* на первичный ключ **id** в таблице **products**.

Внешний ключ (Foreign Key) — это "мост" между таблицами. Он гарантирует, что значение в одном столбце (например, **customerid** в таблице **orders**) всегда будет соответствовать **реальной записи** в другой таблице (например, **id** в таблице **customers**).

Давайте попробуем рассказать СУБД, что мы хотим установить связи между нашими таблицами, используя внешние ключи. Вот что мы хотим:

1. **Изменить таблицу *orders*.**

   Добавим ограничение, назовем его fk_customer_id (fk — от foreign key) с внешним ключом (customerid), который ссылается на поле **id** таблицы **customers**.

Как это сделать в SQL? Да просто перевести эту фразу на английский язык, добавив некоторые формальности:

```
ALTER TABLE orders
ADD CONSTRAINT fk_customer_id FOREIGN KEY (customerid)
REFERENCES customers(id);
```

ALTER TABLE означает Изменить таблицу. Вообще в SQL всегда, когда вы хотите что-то изменить, вы будете использовать инструкцию **ALTER** (хотя MySQL иногда отступает от этого правила). Далее мы указываем название таблицы (orders), после чего пишем, как именно мы хотим изменить таблицу: мы хотим добавить (**ADD**) ограничение (**CONSTRAINT**) [с названием] fk_customer_id. После этого мы пишем, какое именно ограничение: мы добавляем внешний ключ **FOREIGN KEY**, после чего указываем в скобках имя столбца, который будет служить нашим внешним ключом (customerid). Далее мы пишем, что этот внешний ключ ссылается (**REFERENCES**) на таблицу customers, поле **id**.

Когда создавали язык SQL, его специально продумывали так, чтобы инструкции были похожи на законченные предложения на английском языке. К сожалению, по этой же причине тем, кто не владеет английским, осваивать SQL значительно сложнее.

После выполнения этой инструкции, при добавлении новой записи в таблицу заказов мы сможем указать только существующего покупателя (запись с таким **id** должна присутствовать в таблице покупателей).

2. Изменить таблицу *orderdetails*.

Добавить ограничение с названием fk_products_id как внешний ключ от **productid** со ссылкой на первичный ключ (id) таблицы **products**, а также добавить ограничение с названием fk_orders_id как внешний ключ на таблицу **orders**.

Или то же самое на SQL:

```
ALTER TABLE orderdetails
ADD CONSTRAINT fk_products_id FOREIGN KEY (productid)
REFERENCES products(id),
ADD CONSTRAINT fk_orders_id FOREIGN KEY (id)
REFERENCES orders(id);
```

Как видите, мы можем в одной инструкции ALTER передавать сразу несколько инструкций по добавлению (а также модификации или удалению) объектов и свойств таблицы.

ПРИМЕЧАНИЕ.

Для MS SQL: чтобы этот запрос прошел, вам надо убрать вторую инструкцию ADD, выполнив такой запрос:

```
ALTER TABLE orderdetails
ADD
CONSTRAINT fk_products_id FOREIGN KEY (productid)
REFERENCES products(id),
CONSTRAINT fk_orders_id FOREIGN KEY (id)
REFERENCES orders(id);
```

Вообще для MS SQL старайтесь выполнять каждое изменение отдельным запросом. Такова "философия" его разработчиков.

3.6.1. Ссылочная целостность

Мы добавили связи между таблицами, но не учли важный вопрос: что произойдет, если удалить запись, на которую ссылаются другие таблицы? Например:

- что будет, если удалить товар, который есть в заказах;
- удалить клиента, у которого есть активные заказы.

Каждая СУБД имеет для этого **механизм контроля ссылочной целостности**.

Ссылочная целостность — это "правила дорожного движения" для данных. Она следит за тем, чтобы:

- **Не было "мертвых" ссылок.**

  Например, если в таблице **orders** есть заказ с **customerid** = 5, то в таблице **customers** обязательно должна быть запись с **id** = 5.

- **Данные не теряли смысла.**

  Если удалить клиента из **customers**, все его заказы в **orders** не превратятся в "беспризорников": СУБД либо запретит удаление, либо обработает его по вашему сценарию (например, удалит заказы автоматически).

Как это работает "под капотом"?

Когда вы объявляете внешний ключ, СУБД начинает **проверять каждое ваше действие**:

- При вставке новой записи — убеждается, что связанная запись существует.
- При удалении записи — решает, что делать с зависимыми данными (запретить удаление? Удалить их тоже?).
- При изменении ключа — обновляет ссылки автоматически (если вы так настроите).

Чтобы избежать ошибок, СУБД предлагает четыре действия для внешних ключей:

1. **RESTRICT** — запрещает удаление, если есть зависимые записи. Проверка выполняется сразу.
2. **NO ACTION** — для **PostgreSQL** и **MSSQL** проверка ссылочной целостности откладывается до конца транзакции. Может быть медленнее **RESTRICT** из-за этого. Для **MySQL RESTRICT** и **NO ACTION** являются синонимами.
3. **SET NULL** — устанавливает NULL в поле внешнего ключа (только для обнуляемых полей).
4. **CASCADE** — автоматически удаляет или обновляет зависимые записи "каскадом".

Если мы удалим заказ, то нет смысла в существовании соответствующей записи в параметрах заказа, поэтому, наверное, стоит указать, что изменения в **orders** должны каскадно распространяться и на **orderdetails**.

В остальных случаях имеет смысл разрешать удаление, только если на данную запись нет ссылок из других таблиц. Такое поведение заложено по умолчанию (NO ACTION). Чтобы контроль проходил быстрее, можно поставить действие RESTRICT. В нашем случае это не прибавит нам скорости исполнения, но в качестве практики мы внесем изменения.

Некоторые СУБД предлагают способы изменить существующие ограничения, однако универсальным способом в SQL для этого является удаление существующих ограничений и добавление новых.

Выполните следующую команду:

```
ALTER TABLE orderdetails
DROP CONSTRAINT fk_orders_id,
DROP CONSTRAINT fk_products_id;
```

Для MySQL, возможно, вам понадобится такой синтаксис:

```
ALTER TABLE orderdetails
DROP FOREIGN KEY fk_orders_id,
DROP FOREIGN KEY fk_products_id;
```

Если вы внимательно читали, то уже знаете, что командой на удаление объекта является инструкция **DROP** (Выбросить). Данной командой мы "выбрасываем" ограничения fk_orders_id и fk_products_id.

Для MS SQL синтаксис такой:

```
ALTER TABLE orderdetails
DROP
 CONSTRAINT fk_orders_id,
 CONSTRAINT fk_products_id;
```

Теперь выполните добавление ограничений со указанием стратегий контроля ссылочной целостности:

```
ALTER TABLE orderdetails
ADD CONSTRAINT fk_orders_id FOREIGN KEY (id)
    REFERENCES orders (id)
    ON UPDATE CASCADE
```

```
    ON DELETE CASCADE,
ADD CONSTRAINT fk_products_id FOREIGN KEY (productid)
    REFERENCES products (id)
    ON UPDATE RESTRICT
    ON DELETE RESTRICT;
```

(Для MS SQL — уберите вторую инструкцию ADD, в MS SQL каждое ограничение добавляется отдельной командой).

Добавление после описания ограничения инструкций ON UPDATE (при обновлении) и ON DELETE (при удалении) описывает инструкции, которые СУБД должна выполнять при нахождении активных ссылок на обновляемые или удаляемые данные. Можно указывать разные действия на случай удаления и обновления. Можно не указывать обе части (и ON UPDATE и ON DELETE), достаточно какой-то одной из них.

Изменять ограничение fk_customer_id в нашем случае нет большого смысла, но вы можете попробовать сделать это самостоятельно.

Помните, не бойтесь экспериментировать. Вы всегда можете удалить таблицу или ограничение и создать их заново.

3.6.2. Автонумерация. Умные идентификаторы

Мы предусмотрели целочисленные идентификаторы для заказов, товаров и покупателей. Каждый раз, прежде чем добавить новую запись, нам придется уточнять, а какой последний номер по порядку мы использовали, чтобы не ошибиться со значением первичного ключа.

К счастью, данную работу СУБД может взять на себя, если мы будем использовать поля с автонумерацией. Этот механизм позволяет автоматически генерировать уникальные значения для столбцов и гарантирует, что каждая запись получит уникальный номер.

К сожалению, каждая из СУБД имеет свой синтаксис для добавления поля с автонумерацией.

MySQL

Хотя стандартом SQL для поля с автонумерацией предусмотрена конструкция GENERATED AS IDENTITY, MySQL разрабатывался в эпоху,

когда стандарт SQL еще не включал эту конструкцию (появился в SQL:2003). Для решения задачи автонумерации было введено ключевое слово AUTO_INCREMENT, которое стало частью синтаксиса MySQL.

Синтаксис для CREATE TABLE:

```
CREATE TABLE orders
id INT PRIMARY KEY AUTO_INCREMENT
…
```

Чтобы изменить существующие таблицы в MySQL, выполните следующие команды (прежде чем менять поля первичного ключа, необходимо сперва удалить существующие внешние связи, а затем их восстановить):

```
-- Убираем ограничения для таблицы параметров заказа
ALTER TABLE orderdetails
  DROP CONSTRAINT fk_orders_id,
  DROP CONSTRAINT fk_products_id;

-- добавляем автонумерацию на столбец id
ALTER TABLE products
  MODIFY COLUMN id INT AUTO_INCREMENT;

-- меняем таблицу заказов, убираем внешний ключ и
-- одновременно добавляем автонумерацию на id
ALTER TABLE orders
  DROP CONSTRAINT fk_customers_id,
  MODIFY COLUMN id INT AUTO_INCREMENT;

-- меняем таблицу покупателей, добавляем автонумерацию
ALTER TABLE customers
  MODIFY COLUMN id INT AUTO_INCREMENT;

-- восстанавливаем внешний ключ от заказов на покупателей
ALTER TABLE orders
  ADD CONSTRAINT fk_customers_id FOREIGN KEY (customerid)
REFERENCES customers(id);
```

```
-- восстанавливаем внешние ключи из параметров заказов
-- на заказы и продукцию
ALTER TABLE orderdetails
  ADD CONSTRAINT fk_orders_id FOREIGN KEY (id) REFERENCES orders(id),
  ADD CONSTRAINT fk_products_id FOREIGN KEY (productid) REFERENCES
products(id);
```

В MySQL для изменения столбца используется слово MODIFY вместо ALTER. Это особенности диалекта.

Двумя дефисами подряд (--) в SQL отделяются комментарии. Все, что следует после этого маркера до конца строки, игнорируется.

PostgreSQL

Разработчики PostgreSQL так же, как и в случае с MySQL, создали собственный механизм для поля с автонумерацией, тем не менее, данная СУБД также поддерживает и стандартный механизм.

Нестандартный механизм использует ключевое слово **SERIAL** с таким синтаксисом:

```
CREATE TABLE orders (
    id SERIAL PRIMARY KEY, -- Автоматически создает SEQUENCE
    …
);
```

Особенностью данного механизма является возможность ручного добавления значения (не NULL) в поле SERIAL.

Стандартный механизм (начиная с PosrgreSQL 10) использует ключевое слово IDENTITY:

```
CREATE TABLE orders (
    id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    …
);
```

В этой записи используется конструкция GENERATED ALWAYS (генерируется всегда). Эта настройка говорит СУБД о том, что вставлять собственные значения в это поле нельзя, значение всегда генерируется автоматически.

Альтернативно можно использовать конструкцию GENERATED BY DEFAULT (генерируется по умолчанию). С этой настройкой значение можно добавлять свое (при условии, что оно уникальное), либо значение будет генерироваться автоматически, если при добавлении соответствующее значение не будет указано.

Ключевое слово IDENTITY указывает на то, что данное поле является идентификатором.

- **SERIAL** — просто и быстро, но нестандартно
- **IDENTITY** — стандартно, современно и предоставляет большую гибкость (за счет параметров, которые можно задавать: значение, с которого начинать, шаг увеличения, минимум, максимум и т. д.).

Чтобы внести изменения в наши таблицы в PostgresSQL, выполните следующие инструкции:

```
-- Убираем ограничения для таблицы параметров заказа
ALTER TABLE orderdetails
    DROP CONSTRAINT fk_orders_id,
    DROP CONSTRAINT fk_products_id;

-- Добавляем автонумерацию (IDENTITY)
-- на столбец id таблицы products
ALTER TABLE products
    ALTER COLUMN id ADD GENERATED BY DEFAULT AS IDENTITY;

-- Меняем таблицу заказов:
-- убираем внешний ключ и добавляем IDENTITY
ALTER TABLE orders
    DROP CONSTRAINT fk_customers_id;

ALTER TABLE orders
    ALTER COLUMN id ADD GENERATED BY DEFAULT AS IDENTITY;
```

```
-- Добавляем автонумерацию (IDENTITY)
-- на столбец id таблицы customers
ALTER TABLE customers
    ALTER COLUMN id ADD GENERATED BY DEFAULT AS IDENTITY;

-- Восстанавливаем внешний ключ от заказов на покупателей
ALTER TABLE orders
    ADD CONSTRAINT fk_customers_id
    FOREIGN KEY (customerid) REFERENCES customers(id);

-- Восстанавливаем внешние ключи в таблице orderdetails
ALTER TABLE orderdetails
    ADD CONSTRAINT fk_orders_id
    FOREIGN KEY (id) REFERENCES orders(id),

    ADD CONSTRAINT fk_products_id
    FOREIGN KEY (productid) REFERENCES products(id);
```

Microsoft SQL Server

MS SQL также использует ключевое слово IDENTITY для создания столбца с автонумерацией. В отличие от PostgreSQL, значения параметров START WITH (Начать с...) и INCREMENT BY (Увеличивать на...) нужно указывать явно: INCREMENT(1,1) означает, что начать нумерацию надо с 1 и продолжить нумерацию, увеличивая текущий счетчик на 1.

При создании таблицы синтаксис для создания поля с автонумерацией будет таким:

```
CREATE TABLE orders (
    id INT PRIMARY KEY IDENTITY(1,1), -- старт с 1, шаг 1
    …
);
```

Еще одним отличием для MSSQL служит тот факт, что существующее поле нельзя изменить, добавив к нему автонумерацию. Мы должны удалить существующий столбец и добавить новый, но, поскольку существующий столбец так же является еще и первичным ключом, проще удалить суще-

ствующие таблицы и создать их заново (поскольку наши таблицы еще не содержат никаких данных, мы можем себе это позволить).

Чтобы внести изменения в наши таблицы, выполните следующие инструкции:

```
-- удаляем таблицы
-- порядок перечисления таблиц для удаления важен
-- сначала идут таблицы, которые имеют зависимости
-- затем - таблицы, от которых зависят другие таблицы
DROP TABLE orderdetails;
DROP TABLE orders;
DROP TABLE products;
DROP TABLE customers

-- таблица товаров
CREATE TABLE products(
   id INT PRIMARY KEY IDENTITY(1,1),
   article NVARCHAR(10) NOT NULL,
   name NVARCHAR(300) NOT NULL,
   price NUMERIC(8,2) NOT NULL
   );

-- таблица покупателей
CREATE TABLE customers(
   id INT PRIMARY KEY IDENTITY(1,1),
   name NVARCHAR(100) NOT NULL,
   email NVARCHAR(50) NULL,
   address NVARCHAR(300) NULL
   );

-- таблица заказов
CREATE TABLE orders(
   id INT PRIMARY KEY IDENTITY(1,1),
   customerid INT NOT NULL
   CONSTRAINT fk_customers_id
   FOREIGN KEY REFERENCES customers(id)
   );
```

```
-- таблица параметров заказов
CREATE TABLE orderdetails(
   id INT,
   productid INT NOT NULL,
   CONSTRAINT pk_products PRIMARY KEY(id, productid),
   CONSTRAINT fk_products_id
   FOREIGN KEY(productid) REFERENCES products(id),
   CONSTRAINT fk_orders_id
   FOREIGN KEY(id) REFERENCES orders(id)
   ON UPDATE CASCADE
   ON DELETE CASCADE
);
```

ПРИМЕЧАНИЕ.

Мы не добавляем автонумерацию в **orderdetails**, потому что это подчиненная таблица к **orders**. При добавлении мы должны будем указывать тот же идентификатор, что и в таблице **orders**.

3.6.3. Какие еще бывают ограничения?

Помимо уже известных нам PRIMARY KEY, FOREIGN KEY и NOT NULL, в SQL существуют и другие виды ограничений, условия которых будут автоматически проверяться для добавления данных в таблицу:

UNIQUE

Гарантирует, что значения в столбце не повторяются (кроме NULL, если поле обнуляемое).

Пример для всех СУБД:

```
CREATE TABLE products (
    id INT PRIMARY KEY,
    article NCHAR(10) UNIQUE -- Уникальный артикул
);
```

Особенности:

- В PostgreSQL можно создать ограничение для группы столбцов:

```
UNIQUE (column1, column2)
```

DEFAULT

Устанавливает значение по умолчанию, если поле не указано при вставке.

Пример:

```
CREATE TABLE customers (
   id INT PRIMARY KEY IDENTITY(1,1),
   name NVARCHAR(100) NOT NULL DEFAULT 'Нет данных',
   …
   );
```

CHECK

Проверяет условие перед вставкой/обновлением.

```
CREATE TABLE employees (
   id INT PRIMARY KEY,
   age INT CHECK (age >= 18) -- Возраст не меньше 18
);
```

Особенности:

- В MS SQL можно давать имя ограничению:

```
CONSTRAINT age_check CHECK (age >= 18)
```

Когда что использовать?

- **UNIQUE:** для логинов, адресов электронной почты, артикулов.
- **DEFAULT:** для дат создания, статусов ("в обработке"), любых значений по умолчанию.
- **CHECK:** для возрастных ограничений, диапазонов цен, в ситуациях, когда следует исключить ошибочный ввод данных.

3.6.4. Вычислимые столбцы

Вычислимые (или вычисляемые) столбцы — это столбцы в таблице, значения которых автоматически рассчитываются на основе других столбцов этой же таблицы.

Например, если в таблице есть *цена товара* (price) и *количество* (quantity), можно создать столбец *total*, который будет вычислять *сумму*:

```
CREATE TABLE order_details(
…
price NUMERIC(10,2),
quantity INT,
total = price * quantity -- формула расчета цена * кол-во
…
```

Если значение часто используется в условиях или агрегациях, вычисление на уровне таблицы избавляет от необходимости писать формулы в каждом запросе. Например, для хранения возраста пользователя можно использовать хранимое значение в таблице, которое будет вычисляться по формуле на основании даты его рождения, а на основании отдельных элементов адреса (город, улица, дом, …) формировать общую адресную строку по заданному шаблону.

Вычисляемые столбцы бывают двух видов:

1. **VIRTUAL** (виртуальные) — значение вычисляется "на лету", когда обрабатывается запрос к такому столбцу. Данный тип экономит место, но замедляет запрос к такой таблице. Эта настройка используется по умолчанию.

2. **PERSISTED/STORED** (сохраняемый) — значение рассчитывается при добавлении или изменении исходных данных и сохраняется в таблице. Соответственно, требует места на диске для хранения, но ускоряет запросы. Используйте эту настройку для часто запрашиваемых данных.

Синтаксис для PostgreSQL и MySQL (используется GENERATED ALWAYS AS... и STORED, если надо указать, что столбец сохраняемый):

```
CREATE TABLE order_details (
    id SERIAL PRIMARY KEY,
    price NUMERIC(10, 2),
    quantity INT,
    total NUMERIC(10, 2) GENERATED ALWAYS AS (price * quantity)
STORED
);
```

Синтаксис для MS SQL Server (используется ключевое слово PERSISTED):

```
CREATE TABLE order_details (
    id INT PRIMARY KEY,
    price DECIMAL(10, 2),
    quantity INT,
    total AS price * quantity PERSISTED -- Сохраняется на диске
);
```

ВАЖНО!

Используйте вычисляемые столбцы, когда данные, на основе которых вычисляются значения, не изменяются часто, чтобы не создавать избыточную нагрузку, также не используйте вычисляемые столбцы со сложными формулами (особенно если в них есть другие запросы), так как это может очень серьезно замедлить работу вашей базы данных.

3.7. Пример 2. Фитнес-клуб

Для закрепления материала давайте разберем еще один пример. Ниже в таблице представлено расписание индивидуальных тренировок в фитнес-клубе. Для упрощения и экономии места примем, что длительность любой тренировки составляет 1 час и тренировка назначается только на время ЧЧ:00.

Посетитель	Тренер, тренировка	Время	Место	Абонемент	Контактный тел.
Калугина Людмила Петровна	Васильев (Силовая)	11:00	Общий зал	VIP	+7-717-882-13-12
Кислицына Антонина Николаевна	Борисова (Пилатес)	12:00	Малый зал	Базовый	+7-732-325-55-45
Лукашин Евгений Павлович	Васильев (Кардио)	16:00	Общий зал	Гостевой	+7-782-811-46-79
Саахов Джабраил Сахмедович	Абрамов (Плавание)	12:00	Бассейн	VIP	+7-717-442-13-12
Калугина Людмила Петровна	Васильев (Кардио)	14:00	Общий зал	VIP	+7-717-882-13-12
Новосельцев Анатолий Ефремович	Васильев (Силовая)	12:00	Общий зал	Базовый	+7-782-412-18-01
Саахов Джабраил Сахмедович	Борисова (Танцы)	14:00	Малый зал	VIP	+7-717-442-13-12
Горбунков Семен Семенович	Абрамов (Функциональная)	16:00	Общий зал	Базовый	+7-719-322-86-12
Бузыкин Андрей Иванович	Васильев (Функциональная)	15:00	Общий зал	Базовый	+7-722-342-18-16
Клюева Надежда Ивановна	Борисова (Йога)	16:00	Малый зал	Расширенный	+7-719-900-16-11
Новосельцев Анатолий Ефремович	Абрамов (Плавание)	14:00	Бассейн	Базовый	+7-782-412-18-01
Кислицына Антонина Николаевна	Васильев (Силовая)	17:00	Общий зал	Базовый	+7-732-325-55-45

Посетитель	Тренер, тренировка	Время	Место	Абонемент	Контактный тел.
Бузыкин Андрей Иванович	Абрамов (Функциональная)	17:00	Общий зал	Базовый	+7-722-342-18-16
Новосельцев Анатолий Ефремович	Васильев (Силовая)	18:00	Общий зал	Базовый	+7-782-412-18-01

3.7.1. Нормализация

Проводим нормализацию по трем формам.

1NF: не все поля атомарны, поле "Тренер, тренировка", нарушает атомарность. Выделяем **вид тренировки** в отдельное поле:

Посетитель	Тренер	Вид тренировки	Время	Место	Абонемент	Контактный тел.

2NF: выявим и устраним частичные зависимости. Первая — простая и очевидная: Посетитель → Телефон и абонемент.

id	ФИО	Контактный тел.	Тип абонемента

Типы абонементов:

id	Тип абонемента

Еще две зависимости неявные, с ними надо порассуждать. Подсказкой может служить место проведения тренировок по плаванию. Очевидно, что такую тренировку можно провести только в бассейне, соответственно, мы имеем зависимость места проведения от типа тренировки.

SKLADCHIK.ORG

Третья зависимость неявная, и ее невозможно выявить прямым анализом таблицы. Эта зависимость может скрываться за ответом на вопрос: а все ли тренеры могут проводить все типы тренировок? Скорее всего, нет, поэтому третьей зависимостью здесь служит зависимость вида тренировки от тренера.

Создаем таблицу **Виды тренировок**:

id	Вид тренировки	id места проведения

Таблица **Места проведения**:

id	Место проведения

Таблица **Компетенции тренеров**:

id	id вида тренировок

Создаем таблицу **Тренеры**:

id	Тренер

Итоговая таблица **Расписание**:

id	id посетителя	id тренера	id вида тренировки	Время

3NF

Проверяем:

- В Тренировках место зависит только от ID_тренировки — все хорошо.
- В Компетенциях_тренеров нет неключевых атрибутов — все хорошо.
- В Расписании все поля зависят от составного ключа (id_посетителя + id_тренера + id_тренировки + Время) — все хорошо.

3.7.2. Все ли мы учли?

Проверяем еще раз структуру. Текущая структура без проблем позволяет назначить в расписании одного и того же тренера на две разные индивидуальные тренировки одновременно. Что мы можем сделать?

Введем сущность **Сеансы тренировок**:

id	id тренера	id вида тренировки	Время начала

Одного наличия этой сущности мало. Мы должны ввести дополнительное ограничение уникальности (**UNIQUE**) на поля (id тренера, время начала). В реальности нужно будет ввести ограничение CHECK, чтобы проверять, чтобы время начала у нас всегда соответствовало формату ЧЧ:00, однако эта проверка реализуется через еще не пройденный материал, поэтому просто доверимся разработчикам фронтенда и предположим, что это ограничение сделано на уровне пользовательского интерфейса.

Изменим сущность **Расписания**:

id	id посетителя	id сеанса

3.7.3. Подбираем типы данных и создаем таблицы

ПРИМЕЧАНИЕ.

Для пользователей MS SQL Server: убедитесь, что ваша база данных использует колляцию, поддерживающую UTF-8 (например, Latin1_General_100_CI_AS_SC_UTF8). В качестве альтернативы, используйте тип NVARCHAR вместо VARCHAR, но тогда снабдите строки в инструкциях INSERT префиксом N, например N'Строка для добавления'.

Посетители (visitors)

Столбец	Тип данных	Ограничения
id	INT	PRIMARY KEY (PK)

name	VARCHAR(150)	NOT NULL
phone	VARCHAR(20)	*
membership_type	INT	FOREIGN KEY (FK) (membership_types) **

* Телефон может быть не указан.
** У некоторых посетителей гостевой тип посещения.

В обоих случаях NULL разрешен.

Типы абонементов (membership_types)

Столбец	Тип данных	Ограничения
id	INT	PK
name	VARCHAR(30)	NOT NULL

Места проведения (places)

Столбец	Тип данных	Ограничения
id	INT	PK
name	VARCHAR(60)	NOT NULL

Виды тренировок (workout_types)

Столбец	Тип данных	Ограничения
id	INT	PK
name	VARCHAR(60)	NOT NULL
place_id	INT	FK (places), NOT NULL

Тренеры (coaches):

Столбец	Тип данных	Ограничения
id	INT	PRIMARY KEY (PK)
name	VARCHAR(50)	NOT NULL

Компетенции тренеров (coach_skills)

Столбец	Тип данных	Ограничения
coach_id	INT	FK (coaches), NOT NULL
workout_id	INT	FK (workout_types), NOT NULL
		PK (coach_id, workout_id)*

Мы используем составной первичный ключ, обеспечивающий связь "многие ко многим" между тренерами и видами тренировок.

Сеансы (sessions)

Столбец	Тип данных	Ограничения
id	INT	PRIMARY KEY (PK) IDENTITY (AUTO_INCREMENT)
time	TIME	PK, (ЧЧ:00)
coach_id	INT	FK (coaches), NOT NULL
workout_type_id	INT	FK (workout_types), NOT NULL
visitor_id	INT	FK (visitors), NOT NULL
		UNIQUE (coach_id, time)

Попробуйте самостоятельно создать базу данных **Fitness** и создать в ней структуру таблиц для фитнес-клуба.

(Не забудьте указать автонумерацию для первичного ключа таблицы сеансов! Для остальных таблиц указывать автонумерацию для первичного ключа не следует).

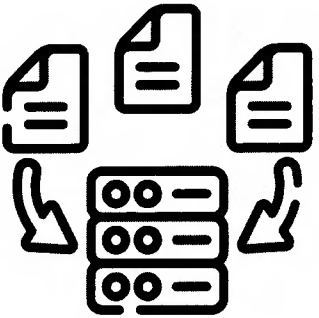
Если у вас не получится, вы можете воспользоваться готовыми листингами для каждой из СУБД из электронного приложения к данной главе, которое можно скачать на сайте издательства.

Заполнение таблиц данными мы продолжим в последующих главах.

Что узнали и чему мы научились?

Ключевые моменты, которые мы изучили в этой главе:

- Мы научились формам нормализации данных, используя 1NF, 2NF и 3NF, которые позволяют нам проектировать эффективную структуру данных.
- Мы освоили создание (CREATE), модификацию (ALTER) и удаление (DROP) таблиц — основы **DDL** — языка описания данных (Data Definition Language).
- Мы научились создавать связи (*relations* — **отношения**) между таблицами через **внешние ключи.**
- Мы научились создавать столбцы с автонумерацией, узнали, что для ввода данных в таблицу можно настраивать различные ограничения.



Глава 4.

Вставка, вывод и чтение данных

https://t.me/IT_Portal

4.1. Добавление данных. Основы

4.1.1. Простая вставка. INSERT на примере Интернет-магазина

В предыдущей главе мы создали структуру базы данных **shop**. Теперь давайте наполним ее данными.

Процесс добавления данных в SQL традиционно называют **вставкой** (англ. **INSERT**).

Давайте вспомним ту таблицу, которую менеджер нам показывал в предыдущей главе.

Район доставки	Имя клиента	e-mail клиента	Адрес доставки	Артикул, наименование	Цена	Кол-во
Юг	Кольцов П. А.	adjutant@ white.mov	ул. Центральная, д. 12, кв. 23	002, Ручка шариковая	199,50	12
Запад	Исаев М. М.	stirlitz@ hustpo.ger	ул. Цветочная, д. 6, кв. 8	006, Блокнот, 96 листов	80,00	4
Запад	Белов А. А.	j.weiss@ abwehr.ger	ул. Тополиная, д. 21, кв. 116	007, Бумага копировальная, пачка 10 листов	260,00	2
Центр	Тульев М. А.	resident@ cia.org	ул. Ленинградская, д. 16, кв. 72	006, Блокнот, 96 листов	72,00	12
Восток	Славин В. В.	nagonia@ tass.age	ул. Горького, д. 84, кв. 307	002, Ручка шариковая	210,00	3
Юг	Кольцов П. А.	adjtant@ white.mov	ул. Центральная, д. 12, кв. 23	001, Арифмометр "Феликс", реплика, Китай	4 400,00	1

Начнем с клиентов. Для них мы предусмотрели таблицу **customers**, в которой есть 4 столбца: **id**, **name**, **email**, **address**. Id нам заполнять не надо, так как он будет генерировать идентификаторы автоматически, таким образом, нам надо внести имя, электронную почту и адрес.

Все три столбца являются строковыми (тип VARCHAR/NVARCHAR), поэтому каждое значение по правилам SQL надо заключить в одинарные кавычки ('текст').

```
'Кольцов П.А.', 'adjutant@white.mov',  'ул. Центральная, д. 12, кв. 23'
'Исаев М.М.',   'stirlitz@hustpo.ger', 'ул. Цветочная, д. 6, кв. 8 '
'Белов А. А.',  'j.weiss@abwehr.ger',  'ул. Тополиная, д. 21, кв. 116'
'Тульев М. А.', 'resident@cia.org',    'ул. Ленинградская, д. 16, кв. 72'
'Славин В. В.', 'nagonia@tass.age',    'ул. Горького, д. 84, кв. 307'
```

ПРИМЕЧАНИЕ.

Если в нашем тексте встречается символ одинарной кавычки ('), нужно применить т. н. **экранирование** — указать этот символ дважды. Например: 'О"Генри — мой любимый автор' (MySQL допускает также другой вид экранирования через знак обратного слеша: 'О\'Генри — мой любимый автор', но это решение нестандартное и не поддерживается в других СУБД). Двойные кавычки (") экранировать не требуется.

Чтобы добавить данные в таблицу, напишем начало нашего выражения:

```
INSERT INTO customers (name, email, address) VALUES
```

(ВСТАВИТЬ В [таблицу] customers (список столбцов, куда вставляем данные) ЗНАЧЕНИЯ...)

Дальше мы перечислим, какие значения мы будем добавлять.

ВАЖНО!

Значения нужно перечислять именно в том порядке, в каком мы указали имена столбцов. Сейчас мы указали порядок: *имя*, *почта*, *адрес*. Мы можем указывать любой порядок, например *адрес*, *имя*, *почта*, но тогда и значения мы должны будем указывать именно в этом порядке.

Далее мы должны в скобках указать три значения для перечисленных столбцов:

```
('Кольцов П.А.', 'adjutant@white.mov',  'ул. Центральная, д. 12, кв. 23')
```

и далее, через запятую, можем указать дополнительные столбцы.

Команда целиком получилась такая:

```
INSERT INTO customers (name, email, address) VALUES
('Кольцов П.А.', 'adjutant@white.mov',  'ул. Центральная, д. 12, кв. 23'),
('Исаев М.М.',   'stirlitz@hustpo.ger', 'ул. Цветочная, д. 6, кв. 8 '),
('Белов А. А.',  'j.weiss@abwehr.ger',  'ул. Тополиная, д. 21, кв. 116'),
('Тульев М. А.', 'resident@cia.org',    'ул. Ленинградская, д. 16, кв. 72'),
('Славин В. В.', 'nagonia@tass.age',    'ул. Горького, д. 84, кв. 307');
```

Обратите внимание, что набор значений для каждого из столбцов указан в скобках, наборы перечисляются через запятую.

Таким образом, можно вставлять множество столбцов сразу, если это необходимо, однако в реализации разных СУБД существуют нюансы:

- **MySQL** — по умолчанию максимальная длина пакета составляет около 1 ГБ (определяется параметром *max_allowed_packet*). Но фактическое ограничение связано больше с максимальным количеством столбцов в таблице (около 4096).
- **MSSQL** — официально ограничением является примерно 1024 столбца на одну команду INSERT, хотя фактически лимит сильно зависит от длины отдельных полей.
- **PostgreSQL** — позволяет создавать таблицы с большим числом столбцов (до нескольких тысяч), однако запросы могут столкнуться с проблемами производительности и длиной строки SQL-команды.

Вставка подобным образом оправдана, если мы добавляем сравнительно небольшое количество записей (в практическом смысле — до нескольких десятков). Если операцию вставки нужно провести однократно (как в нашем случае) можно подготовить и большой объем данных для вставки, однако следует учитывать, что вставка подобным образом множества записей

SKLADCHIK.ORG

является довольно медленным процессом (чем больше записей, тем, соответственно, медленнее будет выполнение) и на больших объемах данных, а также в реальной системе использоваться не должна.

Добавим таким же образом номенклатуру товаров в таблицу **products**. Вспомним, что эта таблица имеет столбцы: **id**, **article**, **name price**, при этом **id** у нас генерируется автоматически.

```
INSERT INTO products (article, name, price) VALUES
('001', 'Арифмометр "Феликс", реплика, Китай', 210),
('002', 'Ручка шариковая', 210),
('006', 'Блокнот, 96 листов', 80),
('007', 'Бумага копировальная, пачка 10 листов', 260);
```

Обратите внимание, что артикулы мы также заключаем в одинарные кавычки, так как этот артикул является текстом, а не числом. Если бы мы отвели для хранения этих данных числовое поле, мы не смогли бы сохранить значение '002', оно бы сократилось до 2 и не подошло бы нам вовсе, если в номенклатуре присутствуют артикулы, содержащие буквы.

Цены являются числами, поэтому их мы в кавычки не заключаем.

В большинстве случаев на практике команда INSERT используется для вставки одной записи в таблицу, после ввода данных пользователем, например, при заполнении формы заказа на сайте.

4.1.2. Массовое добавление данных

В некоторых случаях добавлять данные вручную при помощи инструкции INSERT непрактично. Массовые вставки требуют создания больших запросов, которые долго набирать вручную и сложно формировать средствами автоматизации, кроме того, операции вставки довольно медленные, что может стать проблемой при больших объемах данных.

Когда требуется добавить в таблицу крупный набор данных, эти данные уже существуют где-то (в информационной системе, в другой базе данных или в файле). Одним из самых универсальных (а потому — широко используемых) форматов для переноса данных являются так называемые файлы с разделителями (CSV — comma separated values). Это обычные текстовые

файлы, внутри которых содержится набор значений в том порядке, в каком он будет добавляться в таблицу, по одному на каждую строку (строки в файле разделяются стандартным символом перевода строки '\n'). В качестве разделителя между значениями, вопреки названию (*comma* — запятая), в таком файле может служить какой угодно символ, предпочтительно, чтобы он не встречался в основном наборе, поэтому, помимо запятой, используют табуляцию (\t), точку с запятой (;) или вертикальный разделитель (|).

Например, наш набор товаров в CSV-варианте будет выглядеть так:

```
001;Арифмометр "Феликс", реплика, Китай;210
002;Ручка шариковая;210
006;Блокнот, 96 листов;80
007;Бумага копировальная, пачка 10 листов;260
```

Вместо точки с запятой можно использовать любой разделитель, основной рекомендацией служит то, чтобы этот символ не встречался в основных данных.

К сожалению, сам стандарт SQL не предлагает универсального решения для импорта данных из CSV-файла, вместо этого каждая СУБД предлагает свои способы добавления больших данных в таблицу:

MySQL

1. Утилита **mysqlimport** (исполняемый файл находится в директории bin, где установлена СУБД).
2. Команда LOAD DATA:

```
LOAD DATA INFILE 'путь к файлу/имяфайла.csv'
INTO TABLE <таблица>
FIELDS TERMINATED BY ';' -- указываем символ-разделитель полей
LINES TERMINATED BY '\n'  -- указываем разделитель строк
(столбцы)
```

В скобках вы должны указать имена столбцов в таблице назначения через запятую **в том же порядке, в каком они указаны в файле**, например (name, article, price).

PostgreSQL

1. Утилита **copy** (доступна из оболочки **psql**). Утилита входит в состав стандартной поставки PostgreSQL и предназначена для быстрой передачи данных из внешних файлов в таблицы БД и обратно.
2. Команда **COPY**, которая работает примерно так же, но уже в форме SQL-выражения:

```
COPY <имя таблицы>(столбцы)
FROM 'путь к файлу\имяфайла.csv'
DELIMITER ';' -- разделитель
CSV ;
```

MS SQL

1. Утилита **bcp** поставляется вместе с SQL Server и служит инструментом высокоскоростной загрузки данных в таблицы или выгрузки данных из них. Она позволяет эффективно управлять большими объемами данных и поддерживает гибкую настройку форматов файлов.
2. Команда BULK INSERT:

```
BULK INSERT <таблица>
FROM 'путь к файлу\имяфайла.csv'
WITH
(
   FIELDTERMINATOR = ';', -- разделитель значений
   ROWTERMINATOR = '\n' - разделитель строк
)
```

ВАЖНО!

Как вы помните, наши таблицы имеют поле **id** с автонумерацией. Обычно в таких случаях в качестве значения при вставке столбец с генерируемым значением вообще не указывают. В вашем CSV-файле, возможно, придется указать null-значение в качестве значения для первого столбца. При импорте СУБД заменит его на то, которое сгенерирует счетчик автонумерации.

Чтобы добавить NULL-значение, просто поставьте разделитель в самом начале строки:

```
;001;Арифмометр "Феликс", реплика, Китай;4400
;002;Ручка шариковая;210
;006;Блокнот, 96 листов;80
;007;Бумага копировальная, пачка 10 листов;260
```

К сожалению, универсального правила здесь нет, так как каждая СУБД обрабатывает подобную ситуацию по-своему, поэтому, если вы столкнетесь с ошибкой в ходе импорта данных, обратите внимание на этот аспект.

Мы еще вернемся к вставке данных, когда будем рассматривать подзапросы, а пока давайте посмотрим, что же у нас получилось.

4.2. Чтение данных из базы. *Select*

Для чтения данных в SQL используется инструкция **SELECT** (выбрать). Ей вы будете пользоваться больше всего, и изучать ее особенности мы будем на протяжении почти всей оставшейся книги.

Для выборки данных мы должны указать, какие столбцы надо выбрать из таблицы.

Например, нам необходимо знать, какие именно идентификаторы были присвоены клиентам и товарам. Давайте попросим СУБД предоставить нам эту информацию. Для этого выполним запрос:

```
SELECT id, name FROM customers;
```

(ВЫБРАТЬ идентификатор, имя ИЗ [таблицы] customers [покупатели];)

Результатом этой команды будет примерно такой вывод:

id	name
1	Кольцов П.А.
2	Исаев М.М.

3	Белов А. А.
4	Тульев М. А.
5	Славин В. В.

Если не указано обратное, поле автонумерации начинает счет с 1 и продолжает нумерацию, увеличивая счетчик на 1. Однако, если изначально производилась вставка, скажем, 4 строк, которым были присвоены идентификаторы, соответственно, 1, 2, 3, 4, а затем были удалены, например, записи 2 и 4 и остались только идентификаторы 1 и 3, то следующая вставка будет все равно иметь идентификатор 5, так как счетчик не уменьшается при удалении данных. В зависимости от ваших действий вы можете получить иные значения идентификаторов, чем приведены здесь. Это нормально, однако вам придется учитывать эти отличия в некоторых примерах запросов.

ПРИМЕЧАНИЕ.

Для тех, кто работает в MSSQL: если вместо кириллицы вы видите знаки вопроса (????), вам потребуется сменить колляцию (collation) для базы данных. Для этого выполните следующую команду:

```
ALTER DATABASE shop COLLATE Cyrillic_General_CI_AI;
```

Collation, как уже говорилось во второй главе, — набор правил для сортировки и сравнения строк. Для работы с кириллицей здесь мы воспользуемся колляцией, игнорирующей регистр (CI — Case Insensitive), делающей при сравнении строк равенство между 'Иван' и 'ИВАН', а также нечувствительной к ударениям (AI — Accent Insensitive), делающей идентичными символы 'е' и 'ё'.

К сожалению, общего стандарта для названия колляций не существует, поэтому для уточнения, какая именно колляция вам потребуется, необходимо сверяться с документацией.

Если нам нужно получить все столбцы в таблице, можно воспользоваться символом звездочки (*), который говорит СУБД, что необходимо вывести все столбцы в таблице.

Давайте попробуем с товарами:

```
SELECT * FROM products;
```

(ВЫБЕРИ "ВСЕ" ИЗ [таблицы] products)

Результат запроса выведет вам все столбцы из таблицы товаров:

id	article	name	price
1	001	Арифмометр "Феликс", реплика, Китай	4 400.00
2	002	Ручка шариковая	210.00
3	006	Блокнот, 96 листов	80.00
4	007	Бумага копировальная, пачка 10 листов	260.00

4.2.1. Псевдонимы столбцов (ALIAS)

При работе с SQL возникают ситуации, когда необходимо изменить наименование столбца в результате запроса. Для этого можно прибегнуть к возможности подстановки другого имени в результат. Для этого достаточно указать за именем столбца в запросе ключевое слово AS (как), а за ним — нужное вам имя.

```
SELECT article AS Артикул, name As Наименование, price As Цена
FROM products;
```

Допускается не указывать AS, а сразу писать псевдоним:

```
SELECT article Артикул, name Наименование, price Цена
FROM products;
```

Результат будет выводиться с такими заголовками столбцов:

Артикул	Наименование	Цена

4.3. Фильтрация вывода. *WHERE*

Пока что мы научились извлекать содержимое таблицы полностью, что, согласитесь, будет не очень удобно, когда счет идет более чем на 10 записей,

а в современных базах данных могут содержаться миллионы и даже миллиарды строк. В реальности нам очень редко когда требуется считывать все сразу. Обычно мы хотим ограничить вывод данных некоторыми условиями.

Условия выборки в SQL задаются при помощи ключевого слова WHERE (где), после которого идет выражение, по которому будет проверяться каждая строка. Если условие не соблюдается, строка пропускается.

Выражение, таким образом, читается как "Выбрать (столбцы) где (условие)".

Например, чтобы найти все товары дешевле 200 рублей, необходимо выполнить запрос:

```
SELECT article, name FROM products WHERE price < 200;
```

Выбрать артикул, наименование из товаров, где цена < 200. СУБД производит вычисление условия (в данном случае "цена меньше 200"). Любое записанное условие сводится в конечном счете к его истинности или ложности.

4.3.1. Операторы сравнения

Выражения формируются при помощи операторов. Использование операторов, в целом, интуитивно понятно. Если вы работали с формулами в Excel, их использование не должно вызвать серьезных затруднений. Тем не менее, новой информацией для вас может оказаться взаимодействие с NULL.

Для построения условий в SQL используются операторы сравнения, которые возвращают истину (TRUE), ложь (FALSE) или NULL (если данные отсутствуют или неизвестны).

Основные операторы:

= ("равно")

Проверяет равенство двух операндов (слева и справа). Например, выражение ниже выберет всех сотрудников из IT-отдела:

```
SELECT * FROM employees WHERE department = 'IT';
```

<> ("не равно")

В некоторых СУБД так же доступен вариант **!=** .

Обратен оператору равенства, возвращает TRUE, если операнды не равны друг другу. В примере ниже мы выбираем всех посетителей фитнес-клуба, которые не имеют абонемент статуса VIP:

```
SELECT * FROM visitors WHERE membership <> 'VIP';
```

ВАЖНО!

Результатом проверки NULL <> NULL также будет UNKNOWN.

Также доступны и другие стандартные операторы сравнения:

< ("меньше");

> ("больше");

<= ("меньше либо равно");

>= ("больше либо равно").

Если при выборке встретится строка со значением NULL, результатом сравнения будет специальный результат (UNKNOWN — неизвестно). Функционально в данном случае это эквивалентно FALSE, то есть в отборку такая строка не попадет.

Находим товары с ценой меньше 200 рублей:

```
SELECT * FROM products WHERE price < 200;
```

Находим клиентов, у которых вид абонемента не VIP:

```
SELECT * from visitors WHERE membership_type <> 'VIP';
```

4.3.2. Логические операторы

Если требуется отбор по нескольким условиям, применяются логические операторы, при помощи которых можно их комбинировать.

AND — логическое "И", истинно, когда оба условия истинны:

```
SELECT * FROM products WHERE price > 200 AND price < 300;
```

Данное условие выберет только те строки, где цена больше 200 **И** одновременно — меньше 300.

OR — логическое "ИЛИ", истинно, если истинно хотя бы одно из условий:

```
SELECT name
    FROM visitors
    WHERE
      membership_type = 'расширенный' OR
      membership_type = 'VIP';
```

Найти посетителей с расширенным **ИЛИ** VIP-абонементом.

NOT — логическая инверсия (НЕ), возвращает *false*, если операнд, следующий за ним, истинный, или *true*, если операнд ложный.

```
SELECT * FROM visitors
WHERE NOT membership_type = 'VIP'
```

Найти всех посетителей, кроме тех, у кого есть VIP-абонемент.

Данный запрос эквивалентен:

```
SELECT * FROM visitors
WHERE membership_type <> 'VIP'
```

NOT очень часто применяется с оператором **IN**, проверяющим вхождение в другой список.

IS (NOT) NULL

С NULL логические операторы работают следующим образом:

```
NULL AND TRUE → NULL
NULL OR FALSE → NULL
NOT NULL → NULL
```

Для сравнения с NULL (для проверки на NULL) используется конструкция IS NULL или IS NOT NULL (операторы = и <> с NULL не работают!).

Найти клиентов, у которых отсутствуют сведения о телефонном номере:

```
SELECT * FROM visitors WHERE phone IS NULL;
```

Выражение	Результат	Примечание
10 = NULL	UNKNOWN	Любое сравнение с NULL → UNKNOWN
NULL = NULL	UNKNOWN	NULL не равен NULL (!)
10 IS NULL	FALSE	Явная проверка на NULL
NULL IS NULL	TRUE	Корректная проверка
10 IS NOT NULL	TRUE	
NULL IS NOT NULL	FALSE	

IN — "входит В", возвращает TRUE, если проверяемое значение входит в заданное множество.

Вывести только товары из стран-участниц Таможенного Союза:

```
SELECT * FROM products
WHERE
country_of_origin IN ('RU', 'BE', 'AM', 'KZ', 'TM');
```

BETWEEN — "между", возвращает TRUE, если проверяемое значение находится в пределах заданного диапазона.

Найти всех членов клуба, купивших абонемент в январе:

SKLADCHIK.ORG


```
SELECT name FROM members
WHERE
   purchase_date >= '2025-01-01' AND
   purchase_date <= '2025-01-31';
```

Эквивалентно:

```
SELECT name FROM members
WHERE
purchase_date BETWEEN '2025-01-01' AND '2025-01-31';
```

4.3.3. Поиск по шаблону. Оператор LIKE

Оператор `LIKE` используется для поиска текста по заданному шаблону.

В шаблоне мы можем использовать символ %, обозначающий любое количество символов, включая пустое множество '', а также символ _, обозначающий один любой символ.

Как это работает, лучше посмотреть на примере.

Создадим таблицу с набором тестовых данных (листинг доступен в электронном виде на сайте издательства):

```
CREATE TABLE users (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    email VARCHAR(50)
);
```

Добавим данные:

```
INSERT INTO users (id, name, email) VALUES
(1, 'Анна', 'anna@example.com'),
(2, 'андрей', 'andrey@test.org'),
(3, 'Мария', 'maria123@mayl.ru'),
```

```
(4, 'Иван', 'ivan_2024@gmayl.com'),
(5, 'Елена', 'elena.pro@yandax.ru'),
(6, 'alex', 'alex@test.com'),
(7, 'Ольга25', 'olga25@example.net'),
(8, 'Сергей_Иванов', 'sergio@mayl.com'),
(9, 'тест', 'test@domain.com'),
(10, 'Кирилл', 'kirill-2024@example.com');
```

Предположим, мы хотим найти всех пользователей, чье имя начинается на **А**.

Шаблоном для такого поиска будет 'А%' (начинается на нужную букву, а дальше может следовать любой текст). Выполним запрос:

```
SELECT name FROM users WHERE name LIKE 'А%';
```

Ответ, строго говоря, зависит от настроек колляции. По умолчанию MySQL и MSSQL выдадут строки с именами 'Анна' и 'андрей'. Однако в PostgreSQL вы получите только 'Анна', так как оператор LIKE в этой СУБД является регистрозависимым и имя 'андрей' не удовлетворяет условиям шаблона. Для регистронезависимого поиска в PostgreSQL нужно использовать оператор **ILIKE**.

Если мы хотим найти все имена, которые заканчиваются на 'а', создадим шаблон '%а' (сначала любые символы, а на конце строки 'а').

```
SELECT name FROM users WHERE name LIKE '%а';
```

Результатом будут 'Анна' и 'Елена'.

Если мы хотим произвести поиск по букве в середине, зная ее позицию (например, 2), используем подстановочный символ _. Например, для поиска имен, где вторая буква 'н', шаблоном будет '_н%' (один любой символ, затем 'н', а затем что угодно):

```
SELECT * FROM users WHERE name LIKE '_н%'; -- Анна, андрей
```

Если нам необходимо найти строку, где встречается символ подчеркивания или процента, необходимо воспользоваться экранированием. Для этого

используется дополнительный параметр ESCAPE (в данном контексте переводится как *экранировать*).

Например, такой запрос задает символ для экранирования '\':

```
SELECT * FROM users WHERE name LIKE '%\_%' ESCAPE '\';
```

СУБД, встретив такую конструкцию, будет рассматривать символ, который идет непосредственно за символом экранирования, не в качестве символа-подстановки, а как обычный строковой литерал.

ПРИМЕЧАНИЕ.

В MySQL символ для экранирования по умолчанию '\', и параметр ESCAPE использовать не требуется.

4.3.4. Поиск по сложному шаблону

Найдем пользователей, в чьем адресе электронной почты присутствует домен *example*.

Шаблон для адреса электронной почты должен выглядеть так: любой текст, затем символ @, затем 'example', затем точка, затем любой текст:

```
SELECT * FROM users
WHERE email LIKE '%@example.%';
```

Результат будет такой:

id	name	email
1	Анна	anna@example.com
7	Ольга25	olga25@example.net
10	Кирилл	kirill-2024@example.com

Microsoft SQL Server также позволяет внедрять в выражения LIKE элементы регулярных выражений.

Например, чтобы найти адреса *email*, содержащие цифры, можно использовать подобный запрос:

```
SELECT * FROM users WHERE email LIKE '%[0-9]%';
```

От PostgreSQL и MySQL также можно добиться аналогичного поведения, используя регулярные выражения:

Для PostgreSQL:

```
-- Поиск цифр в строке:
SELECT * FROM users WHERE email ~ '[0-9]';

-- Регистронезависимый поиск:
SELECT * FROM users WHERE name ~* '^анн';
```

Для MySQL:

```
-- Поиск цифр:
SELECT * FROM users WHERE email REGEXP '[0-9]';

-- Поиск имен, начинающихся на "A" или "a":
SELECT * FROM users WHERE name REGEXP '^[Aa]';
```

Однако эти возможности не входят в стандарт SQL, кроме того, тема регулярных выражений настолько огромна, что по ней написаны отдельные книги.

Оператор LIKE — очень мощный инструмент, однако его использование имеет свою цену: его использование существенно снижает производительность. Если вы можете не использовать этот оператор, не используйте его. Существенно замедляет поиск ведущий %, поэтому по возможности избегайте таких шаблонов.

Рассматривайте альтернативы: полнотекстовый поиск или регулярные выражения. Если LIKE — единственный вариант, всегда тестируйте запросы на реалистичных данных!

Оптимизировать запросы мы научимся в следующих главах.

4.3.5. Условный вывод с оператором CASE

Иногда нам требуется вывод разных результатов в зависимости от значений в запрашиваемой таблице. Например, у нас есть база данных сети кинотеатров, и мы хотим разбить все фильмы на группы: "До 2000 года", "2000–2010" и "После 2010 года".

Создадим тестовую таблицу (создайте таблицу самостоятельно, если возникнут затруднения, листинги для каждой СУБД доступны в электронном виде):

- **film_id**: уникальный идентификатор фильма.
- **title**: название фильма.
- **genre**: жанр фильма.
- **release_year**: год выхода фильма.
- **box_office**: сумма сборов фильма в рублях.

film_id	title	genre	release_year	box_office
1	Интерстеллар	Фантастика	2014	320000000.00
2	Форрест Гамп	Драма	1994	280000000.00
3	Титаник	Мелодрама	1997	450000000.00
4	Матрица	Экшн	1999	300000000.00
5	Игра престолов	Фэнтези	2011	250000000.00
6	Гарри Поттер	Приключения	2001	350000000.00
7	Хоббит	Фэнтези	2012	310000000.00
8	Терминатор	Научная фантастика	1984	290000000.00
9	Начало	Психологический триллер	2010	330000000.00
10	Крепкий орешек	Боевик	1988	270000000.00

Чтобы получить желаемый результат, нам потребуется оператор условного выражения **CASE**. С помощью него мы можем прямо в запросе реализовывать логику ЕСЛИ ИНАЧЕ (IF ELSE).

Синтаксис у команды может быть двух видов:

1. Простая форма, где выводятся разные значения в зависимости от результата вычисления определенного выражения:

```
CASE выражение
    WHEN значение1 THEN результат1
    WHEN значение2 THEN результат2
    ...
    ELSE результат_по_умолчанию
END
```

2. Поисковая форма, где для разных результатов вывода можно задавать разные условия:

```
CASE
    WHEN условие1 THEN результат1
    WHEN условие2 THEN результат2
    ...
    ELSE результат_по_умолчанию
END
```

Читать это так: "В СЛУЧАЕ КОГДА условие 1, ТОГДА результат 1".

Нетрудно догадаться, что первый вариант работает быстрее, поэтому, если можно обойтись простой формой, используйте ее.

Поскольку для нашей задачи надо проверить три разных условия, нам необходимо воспользоваться вторым вариантом.

Создадим три условия:

```
WHEN release_year < 2000 THEN 'До 2000 года'
WHEN release_year BETWEEN 2000 AND 2010 THEN '2000 - 2010'
WHEN release_year > 2010 THEN 'После 2010 года'.
```

Весь запрос целиком, соответственно, выглядит так:

```
SELECT
title,
CASE
```

```
  WHEN release_year < 2000 THEN 'До 2000 года'
  WHEN release_year BETWEEN 2000 AND 2010 THEN '2000 - 2010'
  WHEN release_year > 2010 THEN 'После 2010 года'
END AS era
FROM films;
```

Результат:

title	era
Интерстеллар	После 2010 года
Форрест Гамп	До 2000 года
Титаник	До 2000 года
Матрица	До 2000 года
Игра престолов	После 2010 года
Гарри Поттер	2000–2010
Хоббит	После 2010 года
Терминатор	До 2000 года
Начало	2000–2010
Крепкий орешек	До 2000 года

Выражения обрабатываются в порядке их указания (сверху вниз), в результат будет выведено то выражение THEN, условие которого было удовлетворено первым.

Например, если изменить предыдущий запрос следующим образом:

```
SELECT
title,
CASE
  WHEN release_year < 2000 THEN 'До 2000 года'
  WHEN release_year BETWEEN 2000 AND 2011 THEN '2000 - 2011'
  WHEN release_year > 2010 THEN 'После 2010 года'
END AS era
FROM films;
```

"Игра престолов" войдет в группу 2000–2011, а не в группу "После 2010 года", так как условие BETWEEN 2000 AND 2011 будет проверено первым.

Предупреждения по использованию:

- В **PostgreSQL** все выражения в блоке **THEN** должны иметь одинаковый тип данных.
- Избегайте вложенных **CASE**: SQL позволяет вам вкладывать один оператор CASE внутрь другого, однако это является плохой практикой и этого следует избегать.
- Используйте **ELSE** для неучтенных сценариев. При написании запроса СЕЙЧАС вы, возможно, учли все возможные варианты, однако это не значит, что в будущем в таблице не появятся данные, которые все **WHEN** не учитывают. Наличие **ELSE** (например, ELSE 'Неизвестно') гарантирует, что результат будет выведен непременно.
- Комментируйте условия, если логика нетривиальная. Заботьтесь о людях, кто будет читать ваш запрос в будущем.
- При работе с NULL помните, что NULL = NULL → FALSE. Используйте **IS NULL**.
- Не забывайте про ключевое слово **END** (конец), закрывающее конструкцию **CASE**.
- В **MS SQL** можно пользоваться альтернативами в виде функций IIF и CHOOSE:

```
-- Аналог CASE с двумя ветками
SELECT IIF(price > 100, 'Дорогой', 'Дешевый') FROM products;

-- Можно указать несколько значений
-- СУБД выберет нужное в зависимости от первого аргумента:
SELECT CHOOSE(month, 'Январь', 'Февраль', 'Март') FROM sales;
```

4.3.6. Приоритет операторов

Выражения вычисляются с таким приоритетом:

1. Выражения внутри круглых скобок () вычисляются в первую очередь, они имеют наивысший приоритет.
2. Затем вычисляются результаты операций сравнения (=, >, < и т. д.).
3. Далее проверяется соответствие шаблону с оператором **LIKE**.
4. Следом вычисляются результаты логических операций. Обратите внимание, что высший приоритет среди них имеет **NOT**, затем **AND** и далее — **OR**. При написании сложных выражений **рекомендуется явно указывать приоритетные операции в скобках, даже когда это необязательно.** Это облегчает чтение выражения и впоследствии помогает избежать ошибок.

Пример запроса со сложными условиями:

```
SELECT name, email FROM users
WHERE (reg_date >= '2025-01-01' AND country = 'Russia')
 OR (status = 'vip' AND NOT is_banned);
```

Выбор всех незаблокированных пользователей из России со статусом VIP, кто зарегистрировался после 1 января 2025 года.

Порядок вычислений будет такой:

```
Результат 1: reg_date >= '2025-01-01'
Результат 2: country = 'Russia'
Результат 3: Результат 1 AND Результат 2
Результат 4: status = 'VIP'
Результат 5: NOT is_banned
Результат 6: Результат 4 AND Результат 5
Результат 7: Результат 3 OR Результат 7
```

4.4. Неявное автоматическое приведение типов в *SQL*

Давайте вернемся к нашей базе **Shop** и выполним запрос:

```
SELECT * FROM products WHERE article = 7;
```

Несмотря на то, что в нашей таблице артикул имеет строковой тип, мы все равно получили запись, где артикул равен '007'. Это произошло, потому что СУБД автоматически привела данные в таблице к сравниваемому типу (INT) и, поскольку в одной из строк выполнилось условие '007' = 7, эта строка и была выведена в результаты запроса.

Автоматическое приведение типов (type coercion) — это процесс, при котором СУБД неявно преобразует данные одного типа в другой, чтобы выполнить операцию (сравнение, арифметику и т.д.). Это упрощает написание запросов, но может приводить к неочевидным ошибкам, если не учитывать правила работы с типами данных.

Как это работает?

Сравнение разных типов

Если в операции участвуют данные разных типов, СУБД пытается привести их к совместимому типу.

В нашем примере СУБД преобразовала строку '007' в число 7.

Вы можете в этом легко убедиться, если выполните запрос:

```
SELECT '007' = 7;
```

СУБД выполнит сравнение и выдаст ненулевой (истинный) результат. Разные СУБД реализуют эту логику по-разному, поэтому вы можете получить 1, t или 7 в зависимости от того, какую именно СУБД вы используете.

В случае если приведение типов выполнить не удалось, СУБД выдаст ошибку:

```
SELECT * FROM products WHERE price > '220₽';
```

Арифметические операции

Если типы операндов не совпадают, СУБД выбирает наиболее подходящий тип.

SKLADCHIK.ORG

Например:

```
SELECT 10 + '5';
```

Результат: 15 (строка '5' → число 5).

4.4.1. Преобразование даты в строку и обратно

Наверное, самым часто используемым вариантом приведения типов является преобразование строк в дату и наоборот.

Предположим, мы хотим получить заказы за 15 мая 2025 года. СУБД в теории может преобразовывать даты из разных форматов, результат этого преобразования не всегда очевиден. Например, допустимо использовать форматы MM/DD/YYYY (американский формат) или DD/MM/YYYY (европейский формат), однако, в связи с тем, что конкретный порядок следования в строке даты месяца и дня зависит от настроек сервера, лучше сразу привыкнуть задавать дату в стандарте ISO, то есть **YYYY-MM-DD** (год-месяц-число).

```
SELECT * FROM orders
WHERE order_date = '2025-05-15';
```

В СУБД так же присутствуют функции для явного преобразования типов, но к ним мы вернемся позднее.

Полный формат стандарта ISO 8601 для строки, задающей дату и время, выглядит так:

```
YYYY-MM-DDThh:mm:ss.sss±hh:mm
```

Также возможен компактный формат, когда дефисы и двоеточия опускаются:

```
YYYYMMDDThhmmss.sss±hhmm
```

где:

- **YYYY** — четырехзначный год (например, 2023);
- **MM** — двузначный месяц (например, 10 для октября);
- **DD** — двузначный день месяца (например, 05);

- **T** — латинская буква T служит разделителем между частью даты и временем;
- **Hh** — двузначные часы (формат 24 часа, например, 14 означает 2 часа дня);
- **Mm** — двузначные минуты (например, 30 минут);
- **ss** — двузначные секунды (например, 15 секунд);
- **.sss** — необязательная дробная секунда (например, .123);
- **±hh:mm** — необязательное смещение временной зоны относительно UTC (например, +03:00 для московского времени зимой).

4.4.2. Особенности СУБД

MySQL

Более гибок в преобразованиях, например:

```
SELECT '10' + 'abc';  -- Результат: 10
```

Может автоматически преобразовывать число в BOOL:

```
SELECT FALSE OR 1; -- 1;
```

или даже так:

```
SELECT TRUE AND '555'; -- 1;
```

Пустая строка '' преобразуется в 0 при работе с числами (или в FALSE):

```
SELECT '10' * 0; -- Результат: 0
```

PostgreSQL

Может приводить автоматически числа, даты BOOLEAN и UUID, также имеет специальный синтаксис для явного приведения типов:

```
SELECT '100'::INTEGER + 50; -- 150
```

Строка сначала явно приводится к целому числу, затем выполняется арифметическая операция.

```
SELECT '1'::BOOL AND TRUE; -- TRUE
```

SQL Server

Позволяет неявное преобразование строк в числа/даты и в **UNIQUEIDENTIFIER**, если формат корректен.

4.4.3. Потенциальные риски потери данных

Автоматическое приведение типов — мощный инструмент, но его неконтролируемое использование может привести к ошибкам. Всегда проверяйте типы данных в условиях WHERE и используйте явные преобразования, когда это необходимо.

> **Основное правило**: избегайте неявного приведения типов во всех случаях, кроме преобразования дат и GUID/UUID.
>
> Для безопасного преобразования строк в дату используйте формат **ISO 8601**.

Поскольку MySQL особенно толерантен к неявным преобразованиям, при использовании данной СУБД следует учитывать эту особенность.

Чтобы проиллюстрировать данную проблему, создадим таблицу example:

```
CREATE TABLE example(data INT NOT NULL DEFAULT 0);
```

Теперь добавим данные:

```
INSERT INTO example (data) VALUES ('3.1415926');
```

Посмотрим, что сохранилось:

```
SELECT * FROM example;
```

data
3

Запросы были выполнены, MySQL спокойно преобразовал строку с действительным числом к типу INT и записал результат в таблицу.

(Чтобы удалить таблицу, выполните запрос `DROP TABLE example;`).

PostgreSQL и MS SQL не позволят неявно преобразовать типы, однако такой запрос в PostgreSQL вполне допустим:

```
SELECT '123.45'::INTEGER;  -- Результат: 123
```

4.5. Постраничный вывод, ограничение вывода, сортировка

4.5.1. Сортировка. ORDER BY

Для сортировки результатов по одному или нескольким столбцам используется инструкция ORDER BY (Упорядочить по).

```
SELECT столбец1, столбец2
FROM таблица
ORDER BY столбец1 [ASC|DESC], столбец2 [ASC|DESC];
```

ASC — сокращение от Ascending (По возрастанию). Этот режим используется по умолчанию, и если вам нужна сортировка от меньшего к большему, ее можно не указывать (в отдельных случаях это полезно для наглядности).

DESC — сокращение от Descending (По убыванию). Используйте этот режим, если вам нужно отсортировать записи по убыванию — от большего к меньшему.

Вы можете использовать сортировку по нескольким столбцам в разных направлениях.

Создадим тестовый набор для системы учета кадров:

```
CREATE TABLE employees (name VARCHAR(50), entry_date DATETIME);
```

ПРИМЕЧАНИЕ.

Для PostgreSQL используйте тип TIMESTAMP.

```
INSERT INTO EMPLOYEES (name, entry_date)
VALUES
('Иванов А. Б.', '2019-07-12'),
('Петрова В. К.', '2021-12-06'),
('Сергеев И. В.', '2025-02-06'),
('Быков Д. И.', '2022-11-12'),
('Андрияди П. В.', '2024-06-23'),
('Терехов А. С.', '2021-12-06'),
('Владимирова С. Д.', '2024-11-20');
```

Обратите внимание. Сотрудники Петрова и Терехов устроились на работу в один и тот же день.

Отсортируем данные по дате выхода на работу:

```
SELECT * FROM employees ORDER BY entry_date;
```

Результат будет отсортирован по дате:

name	entry_date
Иванов А. Б.	2019-07-12 00:00:00.000
Терехов А. С.	2021-12-06 00:00:00.000
Петрова В. К.	2021-12-06 00:00:00.000
Быков Д. И.	2022-11-12 00:00:00.000
Андрияди П. В.	2024-06-23 00:00:00.000
Владимирова С. Д.	2024-11-20 00:00:00.000

Если теперь мы захотим, чтобы имена сотрудников, поступивших на работу в один день, были отсортированы по алфавиту, надо уточнить это в запросе:

```
SELECT * FROM employees ORDER BY entry_date, name;
```

Теперь результат будет таким:

name	entry_date
...	
Петрова В. К.	2021-12-06 00:00:00.000
Терехов А. С.	2021-12-06 00:00:00.000
...	

НАПОМИНАНИЕ!

Помните о *колляции* (collation), когда сортируете текст.

Если теперь мы хотим отсортировать таблицу от самой свежей даты до самой ранней, но при этом сохранить алфавитную сортировку по имени, необходимо указать ключевое слово DESC в запросе:

```
SELECT * FROM employees ORDER BY entry_date desc, name;
```

Получим соответствующий результат:

name	entry_date
Сергеев И. В.	2025-02-06 00:00:00.000
Владимирова С. Д.	2024-11-20 00:00:00.000
Андрияди П. В.	2024-06-23 00:00:00.000
Быков Д. И.	2022-11-12 00:00:00.000
Петрова В. К.	2021-12-06 00:00:00.000
Терехов А. С.	2021-12-06 00:00:00.000

Вместо имен столбцов можно указывать их порядковые номера. Предыдущий запрос эквивалентен этому:

```
SELECT * FROM employees ORDER BY 2 desc, 1;
```

Использовать такую возможность можно для экономии времени при ручном вводе запросов, однако для понимания кода, который будут читать другие, рекомендуется указывать имена столбцов.

В выражении для сортировки можно также использовать встроенные функции СУБД. Ниже примеры для разных СУБД, где сортировка идет по году начала работы, а затем — по алфавиту.

MySQL, MS SQL Server

```
SELECT
    name,
    YEAR(entry_date) AS entry_year
FROM employees
ORDER BY entry_year, name;
```

Здесь мы используем встроенную в MySQL функцию YEAR, которая возвращает компоненту года из даты.

PostgreSQL

```
SELECT
    name,
    EXTRACT(YEAR FROM entry_date) AS entry_year
FROM employees
ORDER BY entry_year, name;
```

PostgreSQL использует функцию EXTRACT для этих же целей (в MySQL тоже можно использовать эту функцию).

Результат будет таким:

name	entry_year
"Иванов А. Б."	2019
"Петрова В. К."	2021
"Терехов А. С."	2021
"Быков Д. И."	2022
"Андрияди П. В."	2024
"Владимирова С. Д."	2024

4.5.2. Особенности сортировки NULL

В MySQL и MSSQL значения NULL считаются минимальным значением и появляются в сортировке по возрастанию (ASC) первыми.

В PostgreSQL можно задать параметр, как сортировать значения NULL: NULLS FIRST (сначала NULL) или NULLS LAST (NULL потом). По умолчанию значения NULL идут первыми.

```
SELECT <столбцы> FROM <таблица>
ORDER BY <столбцы> NULLS LAST
```

Такой запрос поместит значения NULL в конец списка.

4.5.3. Использование колляции при сортировке

Даже если сама СУБД настроена на использование какой-то одной колляции, иногда необходимо отсортировать значения с использованием других настроек. Для этого необходимо указать ключевое слово COLLATE, за которым следует имя требуемой колляции.

Мы можем посмотреть, как это работает, используя таблицу users, которую мы создали ранее в пункте 4.3.3.

```
SELECT name FROM users ORDER BY name;
```

Если используется колляция по умолчанию, то, скорее всего, вы получите такой список:

name
alex
андрей
Анна
Елена
Иван
Кирилл

Мария
Ольга25
Сергей_Иванов

Но если вы укажете колляцию для регистрозависимого поиска

```
SELECT name FROM users ORDER BY name COLLATE <имя колляции>
```

вы получите совсем другой результат:

name
alex
Анна
Елена
Иван
Кирилл
Мария
Ольга25
Сергей_Иванов
андрей

К сожалению, единого механизма получения списка доступных колляций в СУБД нет.

В **MySQL** имена колляций, которые установлены в вашей СУБД, можно посмотреть при помощи запроса:

```
SHOW COLLATION;
```

В **PostgreSQL** список доступных колляций находится в служебной таблице **pg_collation**. Его можно получить запросом:

```
SELECT * FROM pg_collation;
```

В **MS SQL** Server информацию о колляциях можно получить из системной функции fn_helpcollations():

```
SELECT * FROM fn_helpcollations();
```

Обозначение в имени	Пояснение
AI (Accent Insensitive)	*Нечувствительная к акцентам* — колляция не учитывает акценты в символах. Например, символы *a* и *á* будут считаться одинаковыми
AS (Accent Sensitive)	*Чувствительная к акцентам* — колляция учитывает акценты. В этом случае *a* и *á* считаются разными символами
CS (Case Sensitive)	*Чувствительная к регистру* — колляция делает различия между заглавными и строчными буквами. Например, *a* и *A* считаются разными символами
CI (Case Insensitive)	*Нечувствительная к регистру* — колляция, при которой не учитывается регистр букв. Таким образом, *a* и *A* будут считаться одинаковыми
S (Sort Order)	*Порядок сортировки* — этот элемент в имени колляции может указывать на метод сортировки, используемый в колляции, например, определяет, какая последовательность применяется для сравнения текстов
Binary	*Двоичная* — обозначает колляцию, которая сравнивает данные на основе их двоичных значений, что означает, что символы сравниваются в соответствии с их ASCII- или Unicode-представлением. Это приводит к учету регистра (CS) и акцентов (AS)

SKLADCHIK.ORG

Обозначение в имени	Пояснение
Non-Binary	*Недвоичная* — аналогична бинарной колляции, но при этом не учитывает регистр и акценты. Это может быть полезно, когда необходимо произвести более "гуманизированное" сравнение строк

4.5.4. Постраничный вывод

Очень часто можно наблюдать при выводе каких-либо данных в интерфейсе, например, на веб-сайте вам предлагается выбрать, сколько записей следует выводить на странице.

Такой подход оправдан, так как в случае, если в результате запроса, содержание которого отображается пользователю, присутствует много записей, вывод таких данных может занять продолжительное время, а то и вовсе привести к исчерпанию отведенной под эту цель памяти.

Хотя стандартом SQL:2008 предусмотрен один вариант синтаксиса, до публикации этого стандарта разбивка по страницам разными СУБД реализовывалась по-своему. На момент написания этого материала MySQL не поддерживает стандартный синтаксис. К счастью, данные различия незначительные, общая же суть не меняется.

При необходимости вывода в страничном режиме мы должны сообщить СУБД 3 вещи:

1. Параметры сортировки (без явного указания, в каком порядке должны быть отсортированы записи, нет гарантии, что порядок строк сохранится при переходе со страницы на страницу).
2. Сколько записей всего нужно вывести (например, если пользователь выбрал выводить по 25 записей на страницу).
3. Сколько записей из результата надо пропустить (при переходе с первой на вторую страницу первые 25 записей надо будет пропустить и выводить, начиная с 26-й).

Обобщенная информация по различиям в синтаксисе собрана здесь:

Действие	Упрощенный синтаксис MySQL / PostgreSQL	Стандарт SQL:2008 MSSQL / PostgreSQL
Вывести N записей	LIMIT N	FETCH NEXT N ROWS ONLY
Пропустить N записей	OFFSET N	OFFSET N ROWS

Может возникнуть вопрос, почему синтаксис стандарта более громоздкий и перегруженный ключевыми словами (например, зачем нужны слова NEXT и ONLY). Дело в том, что стандартом также предусмотрены расширения страничного вывода, и вместо NEXT (следующие записи) стандарт допускает, например, вывод предшествующих (PRIOR), первых (FIRST) или последних записей (LAST), а вместо ONLY допускается конструкция WITH TIES (примерный перевод — *с равными*), которая включит в результат также и одноранговые по результатам сортировки записи.

Из трех СУБД, которые мы рассматриваем, поддержки иных режимов, кроме NEXT, не имеет ни одна из них, а поддержку WITH TIES имеет только PostgreSQL. Для постраничного вывода, однако, настройка WITH TIES непригодна, так как не гарантирует вывода точного количества записей на страницу. Если вы используете PostgreSQL, который поддерживает оба варианта, лучше придерживаться стандартного синтаксиса, несмотря на то что он более громоздок.

Например, таблицу с фильмами и кассовыми сборами можно разбить по 3 фильма на страницу, выполнив следующий запрос (упрощенный синтаксис для MySQL/PostgreSQL):

```
-- первая страница
SELECT
title, release_year
FROM films
ORDER BY release_year
LIMIT 3 OFFSET 0;
-- вторая страница
SELECT
title, release_year
```

```
FROM films
ORDER BY release_year
LIMIT 3 OFFSET 3;
```

Выбрать название, год выхода из таблицы фильмов, упорядочить вывод по году выхода (ORDER BY), ограничиться (LIMIT) 3 записями, пропустить (OFFSET) 0.

Указывать OFFSET 0 для первой страницы не обязательно, однако обычно подобные запросы выполняются из скриптов, где значения для LIMIT и OFFSET передаются параметрами.

Для вывода второй страницы мы указываем, что пропустить надо первые 3 записи. Те же два запроса в стандартном синтаксисе (PostgreSQL, MS SQL Server) выглядят так:

```
-- первая страница
SELECT
title, release_year
FROM films
ORDER BY release_year
OFFSET 0 ROWS FETCH NEXT 3 ROWS ONLY
-- вторая страница
SELECT
title, release_year
FROM films
ORDER BY release_year
OFFSET 3 ROWS FETCH NEXT 3 ROWS ONLY
```

В первой странице будет выведено:

title	release_year
Терминатор	1984
Крепкий орешек	1988
Форрест Гамп	1994

Во второй:

title	release_year
Титаник	1997
Матрица	1999
Гарри Поттер	2001

Вопросы производительности и советы

Проблема больших OFFSET: запросы вида **OFFSET 100000** заставляют СУБД перебирать все предыдущие строки, что медленно. Чтобы улучшить производительность, используйте фильтрацию по последнему значению:

```
-- Вместо OFFSET 3 из предыдущего запроса
SELECT title, release_year
FROM films
WHERE release_year > 1994
ORDER BY release_year
OFFSET 0 ROWS
FETCH NEXT 3 ROWS ONLY
```

Здесь мы, зная последнее ключевое значение предыдущей страницы (1994), которое, разумеется, необходимо получить заранее, заставляем СУБД найти все записи, где значение года выхода больше указанного, и вывести первые три записи из них.

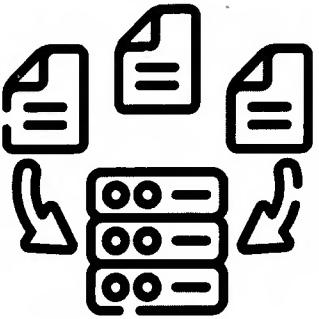
При проведении оптимизации, используя эту технику, вам необходимо предварительно удостовериться, что в исходной таблице только одна запись с подходящим годом выпуска. Обычно для таких целей либо используются значения первичного ключа, если они идут по возрастанию, либо набор значений, которые обеспечивают уникальность каждой из записей.

Что узнали и чему мы научились?

В данной главе вы узнали:

- как добавлять данные в таблицы при помощи инструкции **INSERT**;
- как добавлять большие объемы данных в пакетном режиме для всех трех изучаемых СУБД;
- как получать информацию из таблиц при помощи инструкции **SELECT**;
- как ограничивать результаты при помощи инструкции **WHERE**;
- как сортировать полученный результат при помощи инструкции **ORDER BY** и выводить данные постранично.

Но что делать, если данные, которые нам нужны, хранятся в нескольких таблицах? Об этом мы поговорим в следующей главе.



Глава 5.

Объединение данных. Работа с несколькими таблицами

5.1. Еще немного о преимуществах нормализации на примере фитнес-клуба

Ранее мы изучили **принципы нормализации данных**, которые помогают структурировать информацию, устранять избыточность и поддерживать целостность базы данных. Благодаря разделению данных на логические таблицы мы избегаем дублирования и упрощаем управление сущностями. Однако на практике часто возникает задача, которая кажется парадоксальной: как работать с данными, которые намеренно разнесены по разным таблицам?

Вернемся к нашему фитнес-клубу, структуру базы данных для которого мы создали в конце 3-й главы. Вот повтор ненормализованного расписания занятий:

Посетитель	Тренер, тренировка	Время	Место	Абонемент	Контактный тел.
Калугина Людмила Петровна	Васильев (Силовая)	11:00	Общий зал	VIP	+7-717-882-13-12
Кислицына Антонина Николаевна	Борисова (Пилатес)	12:00	Малый зал	Базовый	+7-732-325-55-45
Лукашин Евгений Павлович	Васильев (Кардио)	16:00	Общий зал	Гостевой	+7-782-811-46-79
Саахов Джабраил Сахмедович	Абрамов (Плавание)	12:00	Бассейн	VIP	+7-717-442-13-12

Посетитель	Тренер, тренировка	Время	Место	Абонемент	Контактный тел.
Калугина Людмила Петровна	Васильев (Кардио)	14:00	Общий зал	VIP	+7-717-882-13-12
Новосельцев Анатолий Ефремович	Васильев (Силовая)	12:00	Общий зал	Базовый	+7-782-412-18-01
Саахов Джабраил Сахмедович	Борисова (Танцы)	14:00	Малый зал	VIP	+7-717-442-13-12
Горбунков Семен Семенович	Абрамов (Функциональная)	16:00	Общий зал	Базовый	+7-719-322-86-12
Бузыкин Андрей Иванович	Васильев (Функциональная)	15:00	Общий зал	Базовый	+7-722-342-18-16
Клюева Надежда Ивановна	Борисова (Йога)	16:00	Малый зал	Расширенный	+7-719-900-16-11
Новосельцев Анатолий Ефремович	Абрамов (Плавание)	14:00	Бассейн	Базовый	+7-782-412-18-01
Кислицына Антонина Николаевна	Васильев (Силовая)	17:00	Общий зал	Базовый	+7-732-325-55-45
Бузыкин Андрей Иванович	Абрамов (Функциональная)	17:00	Общий зал	Базовый	+7-722-342-18-16
Новосельцев Анатолий Ефремович	Васильев (Силовая)	18:00	Общий зал	Базовый	+7-782-412-18-01

Для начала нам необходимо заполнить таблицы данными.

ПРИМЕЧАНИЕ.

Готовые скрипты для создания таблиц и внесения в них данных можно найти в электронном приложении на сайте издательства.

Сперва добавим данные в независимые таблицы (те таблицы, которые не имеют внешних ссылок на другие таблицы):

```
-- Таблица membership_types (типы абонементов)
INSERT INTO membership_types (id, name) VALUES
(1, 'VIP'),
(2, 'Базовый'),
(3, 'Гостевой'),
(4, 'Расширенный');

-- Таблица places (места проведения)
INSERT INTO places (id, name) VALUES
(1, 'Общий зал'),
(2, 'Малый зал'),
(3, 'Бассейн');

-- Таблица coaches (тренеры)
INSERT INTO coaches (id, name) VALUES
(1, 'Васильев'),
(2, 'Борисова'),
(3, 'Абрамов');
```

Теперь заполним данные зависимых таблиц. Начнем с видов тренировок: силовые, функциональные и кардиотренировки проходят в общем зале (мы определили для общего зала ключ id = 1), танцы, пилатес и йога проводятся в малом зале (id = 2), ну и плавание — в бассейне (id = 3):

```
INSERT INTO workout_types (id, name, place_id) VALUES
(1, 'Силовая', 1),
(2, 'Пилатес', 2),
(3, 'Кардио', 1),
(4, 'Плавание', 3),
(5, 'Танцы', 2),
(6, 'Функциональная', 1),
(7, 'Йога', 2);
```

Таблица посетителей имеет простую структуру, однако ссылается на тип абонемента (membership_type:1— VIP, 2 — Базовый, 3 — Гостевой, 4 — Расширенный):

SKLADCHIK.ORG


```
INSERT INTO visitors (id, name, phone, membership_type) VALUES
(1, 'Калугина Людмила Петровна', '+7-717-882-13-12', 1),
(2, 'Кислицына Антонина Николаевна', '+7-732-325-55-45', 2),
(3, 'Лукашин Евгений Павлович', '+7-782-811-46-79', 3),
(4, 'Саахов Джабраил Сахмедович', '+7-717-442-13-12', 1),
(5, 'Новосельцев Анатолий Ефремович', '+7-782-412-18-01', 2),
(6, 'Горбунков Семен Семенович', '+7-719-322-86-12', 2),
(7, 'Бузыкин Андрей Иванович', '+7-722-342-18-16', 2),
(8, 'Клюева Надежда Ивановна', '+7-719-900-16-11', 4);
```

Таблица компетенций тренеров (**coach_skills**) обеспечивает связь вида *многие-ко-многим* между таблицами тренеров и тренировками, связывая факты о том, какие виды тренировок может проводить каждый из тренеров. Для этого мы указываем в строках пары значений (ид тренера, ид вида тренировок), например, для Васильева (id 1) это будут: силовая (id 1), кардио (id 3) и функциональная (id 6).

```
INSERT INTO coach_skills (coach_id, workout_id) VALUES
(1, 1), (1, 3), (1, 6), -- Васильев
(2, 2), (2, 5), (2, 7), -- Борисова: Пилатес, Танцы, Йога
(3, 4), (3, 6);         -- Абрамов: Плавание, Функциональная
```

Ну и, наконец, мы подошли, собственно, к расписанию, где все данные собираются воедино. В таблице сеансов мы указываем время, ид тренера, ид вида тренировки и ид посетителя (уникальный идентификатор сеанса имеет автонумерацию, поэтому мы его здесь не указываем):

```
INSERT INTO sessions
   (time, coach_id, workout_type_id, visitor_id)
VALUES
('11:00:00', 1, 1, 1),
('12:00:00', 2, 2, 2),
('16:00:00', 1, 3, 3),
('12:00:00', 3, 4, 4),
('14:00:00', 1, 3, 1),
('12:00:00', 1, 1, 5),
('14:00:00', 2, 5, 4),
('16:00:00', 3, 6, 6),
```

```
('15:00:00', 1, 6, 7),
('16:00:00', 2, 7, 8),
('14:00:00', 3, 4, 5),
('17:00:00', 1, 1, 2),
('17:00:00', 3, 6, 7),
('18:00:00', 1, 1, 5);
```

Примечание о вставке данных

В этом учебном примере мы используем ручной ввод данных через SQL-скрипты, чтобы наглядно продемонстрировать работу с таблицами и их связями. Однако в реальных проектах данные редко добавляются таким образом. Обычно информация поступает в базу через пользовательские интерфейсы веб-сайтов или программ (например, формы регистрации или бронирования), программные интерфейсы (API) мобильных приложений или внешних сервисов, импортируются из внешних источников данных (других баз данных, электронных таблиц или CSV-файлов) либо собираются автоматически с датчиков и прочих *smart*-устройств.

Ручной ввод используется здесь исключительно для обучения — он помогает понять структуру данных и проверить корректность связей между таблицами. На практике же важно проектировать системы так, чтобы минимизировать ручную работу и обеспечить надежное автоматическое заполнение данных. Умение писать такие скрипты пригодится вам для тестирования, миграции данных или восстановления информации в случае сбоев.

Мы потратили много времени, но чем это лучше той прекрасной таблицы, которая была вначале, где все видно сразу?

Давайте представим ситуацию, когда на работу заступил новый менеджер и хочет поставить тренера на тренировку, которую он не может проводить, например, тренера Борисову (id 2) поставит проводить плавание (id 4). Попробуем:

```
INSERT INTO sessions
   (time, coach_id, workout_type_id, visitor_id)
VALUES
('21:00:00', 2, 4, 1);
```

СУБД не даст нам внести подобную запись из-за ограничения на внешний ключ (тренер не может проводить тренировку данного типа).

Попробуем записать какого-нибудь посетителя к тренеру Васильеву на силовую тренировку в 16:00:

```
INSERT INTO sessions
   (time, coach_id, workout_type_id, visitor_id)
VALUES
('16:00:00', 1, 1, 4);
```

Мы снова не сможем это сделать из-за ограничения UNIQUE (coach_id, time) — время и ид тренера не должны повторяться, так как на это время у данного тренера уже назначена тренировка.

Если бы мы вводили данные в исходную таблицу, мы бы столкнулись с недовольными клиентами.

Теперь, когда у нас есть хорошая структура данных и сами данные, мы можем перейти, собственно, к тому, как объединять нормализованные данные через присоединение таблиц.

5.2. Простое объединение

5.2.1. Как присоединять таблицу

Представьте, что вам нужно получить полную информацию о клиенте: его личные данные, тип абонемента, историю посещений и даже имена тренеров. Для этого недостаточно просто выбрать записи из одной таблицы — необходимо **связать данные из нескольких источников** в единый результат. Именно здесь на помощь приходит оператор **JOIN** — ключевой инструмент для объединения таблиц в реляционных базах данных.

Для начала, чтобы приноровиться, давайте просто создадим простое расписание, где будут указаны время и имя посетителя, который на это время записался.

Учитывая, что время сеанса хранится в таблице **sessions**, интересующие нас данные можно было бы получить таким запросом:

```
SELECT time, visitor_id
FROM sessions
ORDER BY time;
```

Но вместо имен мы получим только идентификаторы посетителей:

time	visitor_id
11:00:00	1
12:00:00	2
12:00:00	4
...	...

Нам надо как-то ПРИСОЕДИНИТЬ (JOIN) таблицу посетителей и заменить внешние ключи на реальные значения, хранящиеся в таблице **visitors**, то есть в результате мы должны получить значение *time* из **sessions** и *name* из **visitors**. В запросе, помимо имени столбца, нам еще надо указать имя таблицы.

В SQL это делается при помощи разделителя — символа точки (.) (он еще называется ***оператором разрешения имени*** — *name resolution operator*) в формате:

```
<имя таблицы>.<имя столбца>
```

В нашем случае мы должны написать:

```
SELECT sessions.time, visitors.name
FROM sessions
```

Собственно, дальше мы должны сказать СУБД, что в данном запросе надо присоединить таблицу **visitors**:

```
JOIN visitors
```

Но этого мало, нам необходимо объяснить, как именно провести сопоставление данных. Нам нужно, чтобы сопоставление осуществлялось по совпадению visitor_id в сеансах с идентификатором в таблице посетителей:

```
ON sessions.visitor_id = visitors.id
```

Ну и дальше мы проводим сортировку по времени. Весь запрос выглядит так:

```
SELECT sessions.time AS Время, visitors.name As Посетитель
FROM sessions
JOIN visitors
ON sessions.visitor_id = visitors.id
ORDER BY sessions.time;
```

Выполнение запроса даст нам желаемый результат:

Время	**Посетитель**
11:00:00	Калугина Людмила Петровна
12:00:00	Кислицына Антонина Николаевна
12:00:00	Саахов Джабраил Сахмедович
...	...

СУБД на каждую найденную строку запроса к основной таблице (sessions) находит запись, которая удовлетворяет условию сопоставления, в нашем случае это проверка на равенство visitor_id в основной таблице и **id** в присоединенной таблице (условия указываются после ключевого слова ON). После того как сопоставление найдено, в столбец результатов подставляется значение из запрашиваемого столбца (в нашем случае это имя клиента, name).

Стоит сразу обратить внимание на то, что после ключевого слова ON может идти абсолютно любое выражение, которое может и не касаться данных из двух таблиц (впрочем, надобность в этом возникает очень редко в весьма нетривиальных ситуациях).

Например, чтобы проиллюстрировать этот факт, выполните следующий запрос:

```
SELECT sessions.time AS Время, visitors.name As Посетитель
FROM sessions
JOIN visitors
ON 1=1
ORDER BY sessions.time;
```

или такой:

```
SELECT sessions.time AS Время, visitors.name As Посетитель
FROM sessions
JOIN visitors
ON sessions.id % 2 = 0 AND visitors.id % 3 = 0
ORDER BY sessions.time;
```

Присоединение подчиненной таблицы происходит при выполнении указанных условий. В первом случае присоединение выполняется всегда (выражение 1=1 всегда истинно), таким образом, вы получите в результате запроса 112 строк, где каждое значение из сеансов (14 записей) сопоставлено со всеми 8 посетителями (14 * 8 = 112).

Второй запрос сложнее, в нем мы использовали оператор нахождения остатка от деления %. Сопоставление происходило при соблюдении двух условий:

1. **id** сеанса должен быть четным (остаток от деления на 2 равен 0).
2. **id** посетителя должен быть кратным 3.

В таблице сеансов четных идентификаторов всего 7, в чем можно удостовериться, выполнив запрос:

```
SELECT * FROM sessions WHERE id % 2 = 0;
```

Ну а в таблице посетителей только два клиента имеют **id** кратный 3:

```
SELECT * FROM visitors WHERE id % 3 = 0;
```

- Лукашин Евгений Павлович (ид 3).
- Горбунков Семен Семенович (ид 6).

Таким образом, во втором запросе каждый сеанс с четным **id** был сопоставлен и с Лукашиным, и с Горбунковым, что дало нам 7 * 2 = 14 строк в результате.

После ключевого слова ON может идти абсолютно любое выражение, которое только можно придумать, и в случае, если результат его вычисления

не будет иметь значение NULL или FALSE, СУБД сопоставит подходящие записи из присоединенной таблицы итогам.

Это может пригодиться, например, если необходимо создать такое же расписание, но только для посетителей, имеющих VIP статус (membership_type = 1). Мы просто добавим условие:

```
SELECT sessions.time AS Время, visitors.name As Посетитель
FROM sessions
JOIN visitors
ON sessions.visitor_id = visitors.id
   AND visitors.membership_type = 1
ORDER BY sessions.time;
```

Получим следующий результат:

Время	Посетитель
11:00:00	Калугина Людмила Петровна
12:00:00	Саахов Джабраил Сахмедович
14:00:00	Калугина Людмила Петровна
14:00:00	Саахов Джабраил Сахмедович
...	...

ПРИМЕЧАНИЕ.

Справедливости ради, стоит отметить, что последний запрос полностью эквивалентен следующему:

```
SELECT sessions.time AS Время, visitors.name As Посетитель
FROM sessions
JOIN visitors
ON sessions.visitor_id = visitors.id
WHERE visitors.membership_type = 1
ORDER BY sessions.time;
```

Мы можем производить фильтрацию как основной таблицы, так и присоединяемой. Оба запроса одинаково эффективны в данном случае (простой JOIN), однако будьте внимательны с указанием дополнительных условий в более сложных случаях, о которых речь пойдет дальше. Хотя оба варианта запроса (и с дополнительным условием ON, и с условием WHERE) приводят к одному и тому же результату, семантически правильнее использовать вариант с WHERE, так как мы фильтруем данные по статусу клиента, а не используем статус как правило для присоединения таблицы.

Для закрепления материала попробуйте самостоятельно построить запрос, выводящий все виды тренировок и соответствующие им места проведения (задача 5.1, пример решения приведен в конце главы).

5.2.2. Присоединение нескольких таблиц

Если необходимо объединить данные из двух или более таблиц, синтаксис запроса почти не меняется, нам надо сперва перечислить все столбцы, данные из которых мы хотим увидеть, затем перечислить присоединяемые таблицы. Например, надо построить список тренеров с видами тренировок, которые они могут проводить.

Здесь мы столкнулись с проблемой, что названия тренировок и тренеры связаны через третью таблицу **workout_types**, соответственно, чтобы вывести нужный нам список, надо к списку тренеров (coaches) присоединить не одну, а две таблицы (**coach_skills** и **workout_types**).

Чтобы составить такой запрос, мы должны просто перечислить таблицы, которые мы присоединяем одну за другой через несколько операторов JOIN:

```
SELECT
    coaches.name AS "Тренер",
    workout_types.name AS "Тренировка"
FROM
    coaches
JOIN
    coach_skills ON coaches.id = coach_skills.coach_id
JOIN
    workout_types ON coach_skills.workout_id = workout_types.id
ORDER BY
    coaches.name;
```

Мы делаем запрос к таблице тренеров, затем просим СУБД присоединить таблицу **coach_skills** через идентификатор тренера. Затем к получившемуся результату мы присоединяем через *workout_id* таблицу **workout_types**.

Результат должен получиться следующим:

Тренер	Тренировка
Абрамов	Плавание
Абрамов	Функциональная
Борисова	Пилатес
Борисова	Танцы
Борисова	Йога
Васильев	Силовая
Васильев	Кардио
Васильев	Функциональная

5.2.3. Использование псевдонимов таблиц (aliases)

При построении сложного запроса записи вида:

```
…ON coach_skills.workout_id = workout_types.id
```

могут быть очень громоздкими, так как мы с каждым столбцом должны повторять имя таблицы. Чтобы облегчить написание запроса, мы можем сказать СУБД, что будем использовать псевдоним (короткое обозначение таблицы):

```
SELECT
    c.name AS "Тренер",
    wt.name AS "Тренировка"
FROM
    coaches AS c
JOIN
    coach_skills AS cs ON c.id = cs.coach_id
JOIN
    workout_types AS wt ON cs.workout_id = wt.id
ORDER BY
    c.name;
```

Мы сообщили СУБД посредством ключевого слова AS, что вместо названия *coaches* будем использовать однобуквенный *псевдоним* (*alias*) *c*.

Вместо *coach_skills* — псевдоним *cs*, а вместо *workout_types* — *wt*.

Запись стала гораздо компактнее. В современных СУБД ключевое слово AS не обязательно, его тоже можно не писать, достаточно просто указать псевдоним после имени таблицы:

```
SELECT
    c.name "Тренер",
    wt.name "Тренировка"
FROM
    coaches c
JOIN
    coach_skills cs ON c.id = cs.coach_id
JOIN
    workout_types wt ON cs.workout_id = wt.id
ORDER BY
    c.name;
```

Самостоятельно попробуйте выполнить эти запросы:

- Составьте расписание тренировок для посетителя Калугиной (задача 5.2).
- Составьте расписание занятий (время и вид тренировки) для тренера Васильева (задача 5.3).

5.3. Виды объединений на примере системы регистрации в аэропорту

5.3.1. Внутреннее объединение INNER JOIN

До сих пор мы рассматривали объединения таблиц, где для каждой строки основной таблицы находилось сопоставление в присоединяемых данных.

SKLADCHIK.ORG

Основная (ЛЕВАЯ) таблица

PK	Данные	FK	Внешнее значение
1	...	2	B
2	...	3	C
3	...	1	A
4	...	5	
...	...	...	...

Присоединяемая (ПРАВАЯ) таблица

PK	Данные
1	A
2	B
3	C
4	D
...	...

Результат

FK	Внешнее значение
1	B
2	C
3	A

Рис. 5.1. *Внутреннее объединение*

Такое объединение еще называют внутренним (INNER). Поскольку такой тип объединения используется по умолчанию, ключевое слово **INNER** можно не указывать, чем мы и пользовались в предыдущем разделе.

5.3.2. Левое внешнее объединение LEFT OUTER JOIN

Однако далеко не всегда мы имеем дело с ситуацией, когда имеются кандидаты, отвечающие условиям сопоставления в присоединяемой таблице. Иногда требуется получать данные даже тогда, когда в одной таблице отсутствуют записи, совпадающие с другой. Для этого существуют другие виды объединений — LEFT JOIN, RIGHT JOIN и FULL OUTER JOIN.

Для примера давайте рассмотрим аэропорт и систему регистрации на рейсы.

В этом примере нам понадобятся две сущности: таблица авиарейсы (flights), содержащая сведения о номере рейса (flight_id), пункте отправления (departure_city) и пункте назначения (arrival_city), а также таблица

пассажиры (passengers), где указаны идентификаторы пассажира (passenger_id), имена (name) и номера рейсов, на которые они зарегистрированы.

Создадим базу данных **airport**:

```
CREATE DATABASE airport;
```

ПРИМЕЧАНИЕ.

Для MSSQL укажите колляцию в явном виде:

```
CREATE DATABASE airport
COLLATE Latin1_General_100_CI_AS_SC_UTF8;
```

Либо используйте тип данных NVARCHAR, подставляя префикс N к строкам данных в инструкции INSERT (например N'Москва').

Создадим структуру таблиц и добавим данные (скрипт можно найти в электронном приложении):

```
-- Рейсы
CREATE TABLE flights (
    flight_id INT PRIMARY KEY,
    departure_city VARCHAR(50),
    arrival_city VARCHAR(50)
);

INSERT INTO flights VALUES
(1, 'Москва', 'Санкт-Петербург'),
(2, 'Москва', 'Новосибирск'),
(3, 'Москва', 'Владивосток');

-- Пассажиры
CREATE TABLE passengers (
    passenger_id INT PRIMARY KEY,
    name VARCHAR(50),
    flight_id INT REFERENCES flights(flight_id)
```

```
);

INSERT INTO passengers VALUES
(1, 'Рязанов', 3),
(2, 'Шевелева', NULL),
(3, 'Лукашин', 1);
```

Мы видим, что пассажиры Рязанов и Лукашин уже зарегистрировались на рейсы, а пассажирка Шевелева еще не определилась.

Давайте для начала выведем информацию о пассажирах и рейсах, на которые они зарегистрированы:

```
SELECT p.name Пассажир, f.flight_id Рейс, f.arrival_city Куда
FROM passengers p
JOIN flights f ON f.flight_id = p.flight_id
ORDER BY p.name;
```

Результат получим такой:

Пассажир	**Рейс**	**Куда**
Лукашин	1	Санкт-Петербург
Рязанов	3	Владивосток

Как видите, пассажирка Шевелева в результате запроса не отобразилась, потому что значению рейса (NULL) в ее записи не нашлось сопоставления в таблице рейсов.

Чтобы исправить ситуацию, нам необходимо воспользоваться так называемым ЛЕВЫМ ВНЕШНИМ присоединением (LEFT OUTER JOIN).

Этим самым мы сообщаем СУБД, что мы хотим вывести **ВСЕ** записи из основной (*левой*) таблицы и по возможности (если найдется сопоставление) вывести значения из присоединяемой (*правой*) таблицы.

Основная (ЛЕВАЯ) таблица

PK	Данные	FK	Внешнее значение
1	...	2	B
2	...	3	C
3	...	1	A
4	...	5	
...	...	...	...

Присоединяемая (ПРАВАЯ) таблица

PK	Данные
1	A
2	B
3	C
4	D
...	...

Результат

FK	Внешнее значение
1	B
2	C
3	A
5	NULL

Рис. 5.2. *Левое внешнее объединение (LEFT OUTER JOIN)*

Изменим запрос следующим образом:

```
SELECT p.name Пассажир, f.flight_id Рейс, f.arrival_city Куда
FROM passengers p
LEFT JOIN flights f ON f.flight_id = p.flight_id
ORDER BY p.name;
```

Результат будет содержать уже всех пассажиров, а напротив Шевелевой будет стоять NULL, так как на рейс она еще не зарегистрировалась:

Пассажир	Рейс	Куда
Лукашин	1	Санкт-Петербург
Рязанов	3	Владивосток
Шевелева	NULL	NULL

В отличие от внутреннего (INNER), все другие объединения считаются внешними. Соответственно, в SQL предусмотрено ключевое слово OUTER, которое, впрочем, в большинстве современных СУБД можно не использовать.

5.3.3. Правое внешнее объединение RIGHT OUTER JOIN

Результат предыдущего запроса не содержит информации о рейсе в Новосибирск, так как на него не зарегистрировано ни одного пассажира. Если нам необходимо вывести ВСЕ записи из присоединяемой (правой) таблицы и по возможности — вывести данные, которые можно им сопоставить, из основной (левой) таблицы, нужно воспользоваться *правым объединением* (**RIGHT JOIN**). Заменим LEFT на RIGHT в предыдущем запросе:

```
SELECT p.name Пассажир, f.flight_id Рейс, f.arrival_city Куда
FROM passengers p
RIGHT JOIN flights f ON f.flight_id = p.flight_id
ORDER BY p.name;
```

Результат будет содержать все рейсы, однако, поскольку ни одного пассажира на рейс в Новосибирск не зарегистрировано, ему будет сопоставлено значение NULL:

Пассажир	Рейс	Куда
NULL	2	Новосибирск
Лукашин	1	Санкт-Петербург
Рязанов	3	Владивосток

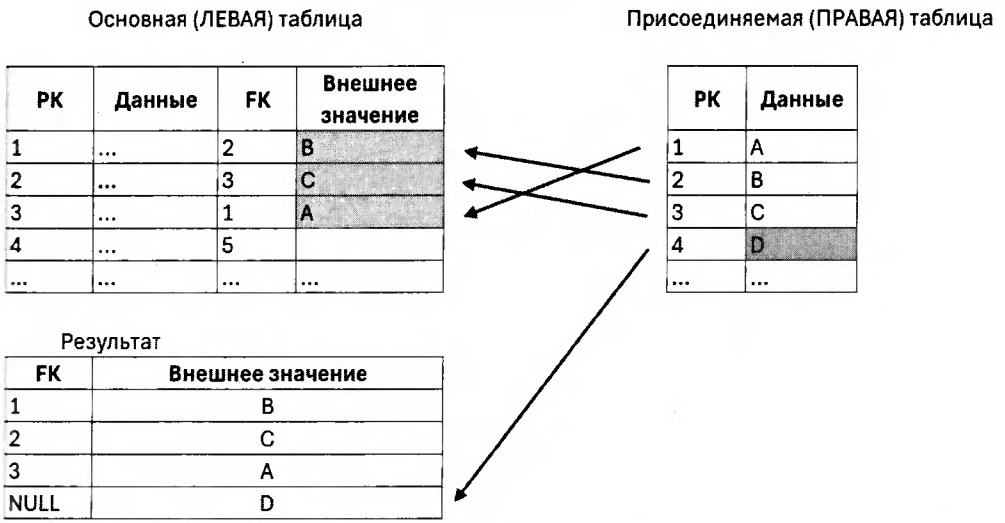


Рис. 5.3. *Правое внешнее объединение (RIGHT OUTER JOIN)*

ПРИМЕЧАНИЕ.

В простых случаях основную и присоединяемую таблицы можно поменять местами, изменив направление объединения.

5.3.4. Полное объединение (FULL OUTER JOIN)

Ну и, наконец, если мы хотим получить все данные из основной таблицы вместе со всеми данными из присоединяемой таблицы, надо применить полное внешнее объединение (FULL OUTER JOIN).

ПРИМЕЧАНИЕ.

Для пользователей MySQL данная СУБД не поддерживает на момент подготовки этого материала полное внешнее объединение, однако добиться того же результата все-таки можно с использованием оператора **UNION**, о котором речь пойдет чуть дальше.

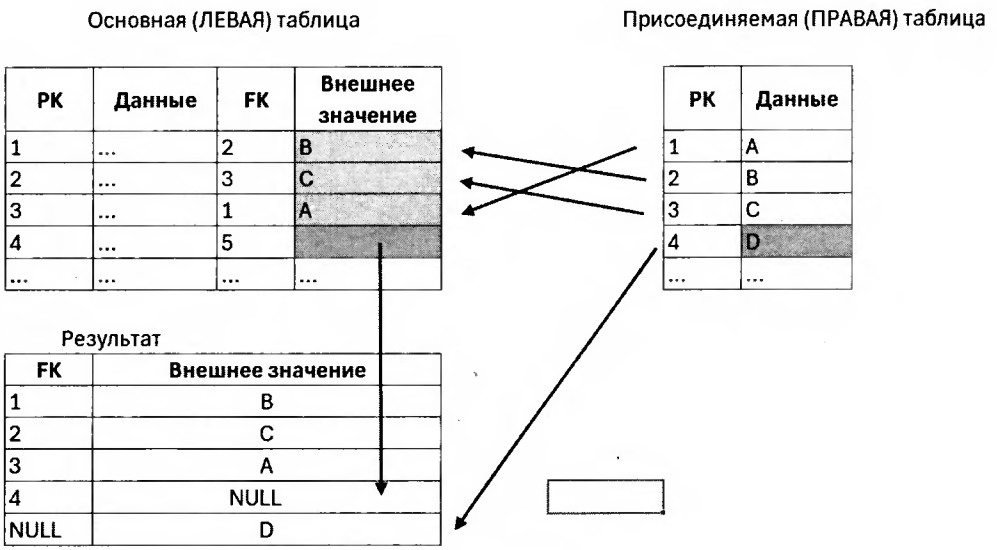


Рис. 5.4. *Полное внешнее объединение (FULL OUTER JOIN)*

Запрос на полное объединение для MSSQL и PostgreSQL выглядит следующим образом:

```
SELECT p.name Пассажир, f.flight_id Рейс, f.arrival_city Куда
FROM passengers p
FULL OUTER JOIN flights f ON f.flight_id = p.flight_id
ORDER BY p.name;
```

Пассажир	Рейс	Куда
NULL	2	Новосибирск
Лукашин	1	Санкт-Петербург
Рязанов	3	Владивосток
Шевелева	NULL	NULL

5.3.5. Перекрестное объединение CROSS JOIN

Данный вид объединения стоит особняком от остальных, так как в нем не производится условного сопоставления. При этом виде объединения к каждой записи основной таблицы присоединяются все записи присоединяемой. Общее количество строк в результате такого присоединения, таким образом, составляет произведение количества строк в обеих таблицах.

Несмотря на кажущуюся странность и избыточность результатов CROSS JOIN, существует ряд ситуаций, когда такая операция оказывается крайне полезна.

При помощи CROSS JOIN также удобно создавать разнообразные сетки данных (например, расписания занятий, где каждый учитель должен провести урок в каждом классе). Например, мы можем получить все рейсы, на которых мог улететь пассажир (предположим, он в розыске):

```
-- Перекрестное пересечение двух таблиц
SELECT f.arrival_city, p.name
FROM flights f
CROSS JOIN passengers p;
```

Результат:

arrival_city	name
Владивосток	Рязанов
Новосибирск	Рязанов
Санкт-Петербург	Рязанов
Владивосток	Шевелева
Новосибирск	Шевелева
Санкт-Петербург	Шевелева
Владивосток	Лукашин
Новосибирск	Лукашин
Санкт-Петербург	Лукашин

Довольно часто возникают ситуации, когда необходимо сгенерировать большие объемы данных, например, для нужд тестирования или формирования всех возможных комбинаций из нескольких наборов. Имея небольшой набор первичных данных, используя CROSS JOIN цепочкой, можно получить огромные наборы строк для тестирования.

Мы можем использовать и самопересечения, когда используем CROSS JOIN на той же самой таблице. Например, если у нас есть два соседних места в самолете, мы можем узнать все возможные комбинации пассажиров, которые могут оказаться рядом друг с другом:

```
-- Самопересечение
SELECT p1.name as "Место 2A", p2.name as "Место 2B"
FROM passengers p1
CROSS JOIN passengers p2
-- Исключаем невозможный сценарий
WHERE p1.passenger_id <> p2.passenger_id;
```

ПРИМЕЧАНИЕ.

В таких сценариях необходимо исключить совпадающие варианты (в нашем примере один пассажир не может сидеть одновременно на двух местах, поэтому мы используем фильтрующее условие в блоке WHERE).

Получился результат, в котором выводятся 6 возможных комбинаций:

Место 2A	Место 2B
Лукашин	Рязанов
Шевелева	Рязанов
Лукашин	Шевелева
Рязанов	Шевелева
Шевелева	Лукашин
Рязанов	Лукашин

5.3.6. Слияние (UNION и UNION ALL)

Иногда требуется совместить данные из двух таблиц без какого-либо сопоставления. В прошлом разделе мы упомянули, что в MySQL отсутствует поддержка FULL OUTER JOIN, поэтому это послужит хорошим примером, как можно достичь того же результата при помощи слияний. Пример можно использовать в любой СУБД. Мы просто объединим результаты двух запросов (с левым и правым объединением):

```
-- Эмуляция FULL OUTER JOIN через UNION
SELECT
   p.name AS Пассажир,
   f.flight_id AS Рейс,
   f.arrival_city AS Куда
FROM passengers p
LEFT JOIN flights f ON p.flight_id = f.flight_id

UNION

SELECT
   p.name AS Пассажир,
   f.flight_id AS Рейс,
   f.arrival_city AS Куда
FROM passengers p
RIGHT JOIN flights f ON p.flight_id = f.flight_id
ORDER BY Пассажир;
```

Это два запроса. Результатом первого запроса (с LEFT JOIN) будет:

Пассажир	Рейс	Куда
Лукашин	**1**	**Санкт-Петербург**
Рязанов	3	Владивосток
Шевелева	NULL	NULL

Результат второго (с RIGHT JOIN):

Пассажир	Рейс	Куда
Лукашин	**1**	**Санкт-Петербург**
NULL	2	Новосибирск
Рязанов	3	Владивосток

Слияние двух таблиц даст тот же результат, что и с FULL JOIN. Обратите внимание, что в результат не включены дублирующиеся записи (с пассажиром Лукашиным):

Пассажир	Рейс	Куда
NULL	2	Новосибирск
Лукашин	1	Санкт-Петербург
Рязанов	3	Владивосток
Шевелева	NULL	NULL

Если мы хотим, чтобы дублирующиеся записи были включены в результат, мы должны использовать конструкцию **UNION ALL**:

```
SELECT
p.name AS Пассажир,
f.flight_id AS Рейс,
f.arrival_city AS Куда
FROM passengers p
LEFT JOIN flights f ON p.flight_id = f.flight_id

UNION ALL

SELECT
```

```
p.name AS Пассажир,
f.flight_id AS Рейс,
f.arrival_city AS Куда
FROM passengers p
RIGHT JOIN flights f ON p.flight_id = f.flight_id
```

В данном случае результаты второго запроса попросту добавляются к первому, и фильтрация дубликатов не производится:

Пассажир	Рейс	Куда
Рязанов	3	Владивосток
Шевелева	NULL	NULL
Лукашин	1	Санкт-Петербург
Лукашин	1	Санкт-Петербург
NULL	2	Новосибирск
Рязанов	3	Владивосток

5.3.7. Полная карта объединений

Ниже приведено краткое описание операторов объединения:

Оператор	Описание
INNER JON	Возвращает только строки с совпадающими значениями в обеих таблицах
LEFT OUTER JOIN	Возвращает **все** строки из **основной** таблицы с совпадающими значениями из присоединяемой таблицы, заменяя отсутствующие совпадения NULL
RIGHT OUTER JOIN	Возвращает **все** строки из **присоединяемой** таблицы, подставляя NULL вместо значений, по которым отсутствуют совпадения в основной таблице
FULL OUTER JOIN	Возвращает **все** строки из обеих таблиц, заполняя NULL-значениями отсутствующие совпадения

UNION	Объединяет результаты двух запросов, удаляя дубликаты
UNION ALL	Объединяет результаты двух запросов, сохраняя все строки, включая дубликаты

Для улучшения понимания, как работает механизм объединения данных в SQL, вспоминайте эту схему:

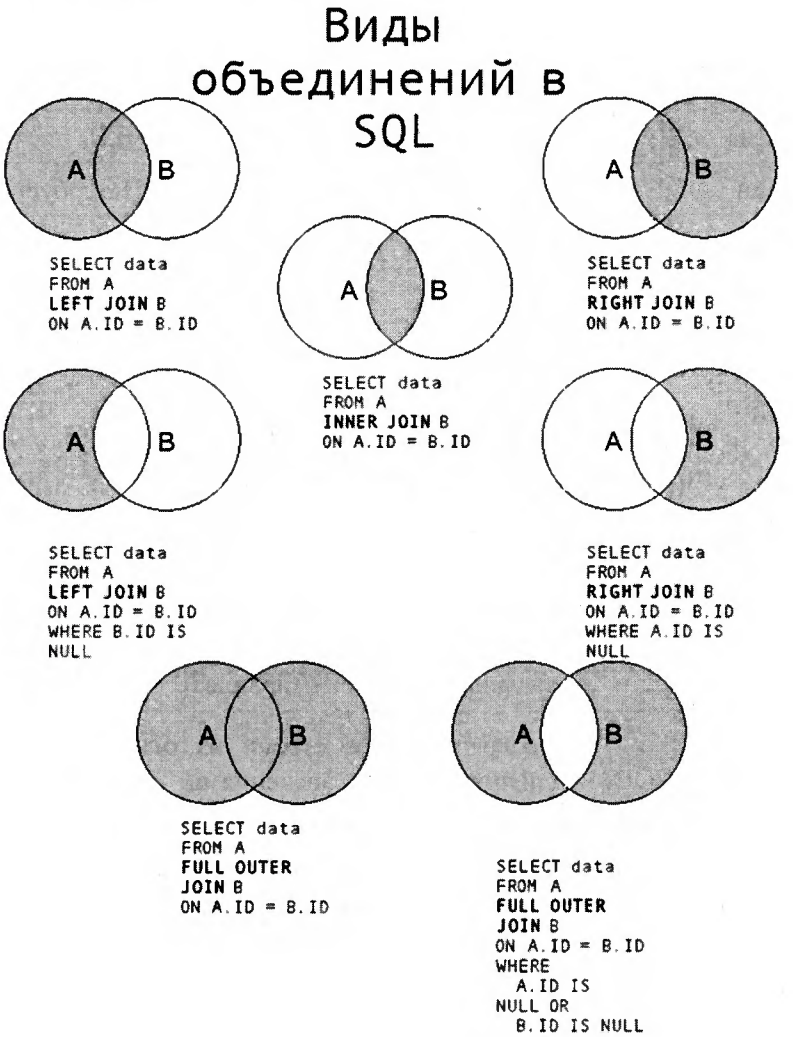


Рис. 5.5. *Виды объединений в SQL*

5.3.8. Порядок объединений, или как ничего не потерять

Порядок, в котором вы указываете таблицы при объединении, может существенно повлиять на результат и производительность запроса. Особенно важным это может быть в связях вида *многие-ко-многим*, таким как между тренерами и видами тренировок в нашей базе фитнес-клуба, которую мы создали ранее.

Добавим тренера Иванова, для которого в базу еще не внесены компетенции:

```
INSERT INTO coaches (id, name) values(4, 'Иванов');
```

Допустим, нам нужно вывести всех тренеров и виды тренировок, которые они могут проводить. Для этого мы будем использовать таблицы **coaches**, **coach_skills** и **workout_types**.

Вариант 1

Будем выбирать из coach_skills, затем присоединим workout_types, затем — coaches:

```
SELECT
    c.name AS "Тренер",
    wt.name AS "Тренировка"
FROM
    coach_skills cs
RIGHT JOIN
    workout_types wt ON cs.workout_id = wt.id
LEFT JOIN
    coaches c ON c.id = cs.coach_id
ORDER BY
    c.name;
```

Результат не включит в себя нового тренера без навыков:

Тренер	Тренировка
"Абрамов"	"Плавание"
"Абрамов"	"Функциональная"

"Борисова"	"Танцы"
"Борисова"	"Пилатес"
"Борисова"	"Йога"
"Васильев"	"Кардио"
"Васильев"	"Функциональная"
"Васильев"	"Силовая"

Вариант 2

Сначала объединяем **coaches** с **coach_skills**, затем с **workout_types**:

```
SELECT
    c.name AS "Тренер",
    wt.name AS "Тренировка"
FROM
    coaches c
LEFT JOIN
    coach_skills cs ON c.id = cs.coach_id
LEFT JOIN
    workout_types wt ON cs.workout_id = wt.id
ORDER BY
    c.name;
```

В результате этого запроса выводятся все тренеры:

Тренер	Тренировка
"Абрамов"	"Плавание"
"Абрамов"	"Функциональная"
"Борисова"	"Танцы"
"Борисова"	"Пилатес"
"Борисова"	"Йога"
"Васильев"	"Кардио"

"Васильев"	"Функциональная"
"Васильев"	"Силовая"
"Иванов"	NULL

Порядок JOIN определяет, какие данные считаются "главными". Это не ошибка, а инструмент для управления результатом на основе бизнес-логики.

Что узнали и чему мы научились?

Оператор JOIN — это фундаментальный инструмент работы с реляционными базами данных, который превращает разрозненные таблицы в единую систему, способную отвечать на сложные вопросы.

Нормализация данных требует разделения информации, но бизнес-логика часто требует ее объединения.

JOIN — это мост между структурой и практикой.

В этой главе мы рассмотрели, как:

- **Собирать данные из разных источников** в осмысленные отчеты.
- **Использовать типы JOIN** (INNER, LEFT, RIGHT, FULL) для решения задач, требующих гибкости.
- **Повышать эффективность запросов**, которая зависит не только от синтаксиса, но и от порядка объединения таблиц. Например, выбор между LEFT JOIN и RIGHT JOIN может кардинально изменить результат.

Практические советы:

- **Всегда задавайтесь вопросом**: "Какие данные должны быть в результате гарантированно?". Это поможет выбрать правильный тип JOIN.

- **Тестируйте запросы** на реалистичных данных. Даже незначительная ошибка в условии ON может привести к некорректным выводам.
- **Используйте псевдонимы** для упрощения запросов. Короткие имена таблиц делают код чище и понятнее.

Приложение к главе 5. Варианты решения заданий

Задача 5.1. Вывести все виды тренировок и соответствующие им места проведения.

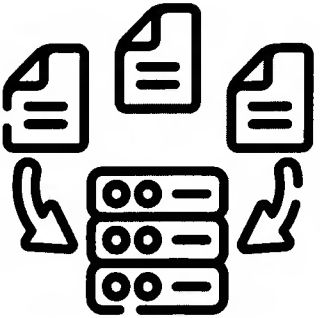
```
SELECT
    wt.name AS Тип,
    p.name AS Место
FROM
    workout_types wt
INNER JOIN
    places p ON wt.place_id = p.id
ORDER BY
    wt.name;
```

Задача 5.2. Расписание тренировок для посетительницы Калугиной.

```
SELECT
    s.time AS "Время",
    wt.name AS "Тип тренировки"
FROM
    sessions s
INNER JOIN
    visitors v ON s.visitor_id = v.id
INNER JOIN
    workout_types wt ON s.workout_type_id = wt.id
WHERE v.name LIKE 'Калугина%'
```

Задача 5.3. Расписание занятий для тренера Васильева.

```
SELECT
    s.time AS "Время",
    wt.name AS "Тренировка"
FROM
    sessions s
INNER JOIN
    coaches c ON s.coach_id = c.id
INNER JOIN
    workout_types wt ON s.workout_type_id = wt.id
WHERE
    c.name LIKE 'Васильев%'
ORDER BY
    s.time;
```



Глава 6.

Группировка данных и агрегатные функции

До сих пор мы извлекали данные из таблиц точно в таком же виде, как они были записаны, но данные редко полезны сами по себе — их ценность раскрывается в аналитике, статистике и закономерностях:

- Телекоммуникационная компания хочет выяснить, из каких регионов больше всего совершается звонков в выходные дни.
- Радиостанция хочет составить карту музыкальных предпочтений по регионам.
- Автодилер хочет знать частоту поломок автомобилей в зависимости от года выпуска.

SQL был создан для ответов на множество подобных вопросов. Группировка и агрегатные функции — это основа аналитики в SQL. Они превращают хаотичные данные в структурированные отчеты, которые помогают принимать взвешенные решения.

6.1. Основы аналитики на примере рыболовецкой компании

Чтобы представить, как работают группировка и агрегатные функции, создадим рыболовецкую компанию:

```
CREATE DATABASE fishing;
```

SKLADCHIK.ORG

У компании есть несколько кораблей разных типов:

```
-- Сведения о кораблях
CREATE TABLE ships (
    ship_id INT PRIMARY KEY,
    ship_name VARCHAR(50) NOT NULL,
    type_ship VARCHAR(50) NOT NULL
);
-- Данные кораблей
INSERT INTO ships (ship_id, ship_name, type_ship) VALUES
(1, 'Фортуна', 'Траулер'),
(2, 'Прометей', 'Шхуна'),
(3, 'Альбатрос', 'Судно-холодильник'),
(4, 'Академик В.В. Рыбкин', 'Траулер');
```

Компания ведет свою деятельность в Баренцевом море, где имеется несколько рыболовецких зон:

```
-- Сведения о зонах ловли
CREATE TABLE zones (
    zone_id INT PRIMARY KEY,
    region_name VARCHAR(100) NOT NULL
);
INSERT INTO zones (zone_id, region_name) VALUES
(1, 'Печорский залив'),
(2, 'Кольский залив'),
(3, 'Русская банка'),
(4, 'Центральная акватория'),
(5, 'Шпицбергенская акватория');
```

Осуществляется промысел нескольких видов рыбы:

```
-- Сведения о рыбе
CREATE TABLE fish (
    fish_id INT PRIMARY KEY,
    species VARCHAR(50) NOT NULL,
    price_per_kg DECIMAL(10, 2) NOT NULL
);
```

```
-- Данные о рыбе
INSERT INTO fish (fish_id, species, price_per_kg) VALUES
(1, 'Лосось', 150.00),
(2, 'Кета', 120.00),
(3, 'Минтай', 60.00),
(4, 'Сельдь', 40.00),
(5, 'Треска', 75.00)
```

В компании ведется учет по уловам в разрезе зон, вида рыбы и судна:

```
-- Сведения об уловах
CREATE TABLE catches (
    catch_id INT PRIMARY KEY,
    ship_id INT NOT NULL,
    fish_id INT NOT NULL,
    zone_id INT NOT NULL,
    quantity_kg INTEGER NOT NULL CHECK (quantity_kg >= 0),
    date_catch DATE NOT NULL,
    FOREIGN KEY (ship_id) REFERENCES ships(ship_id),
    FOREIGN KEY (fish_id) REFERENCES fish(fish_id),
    FOREIGN KEY (zone_id) REFERENCES zones(zone_id)
);

-- Данные по уловам
INSERT INTO catches (
   catch_id,
   ship_id,
   fish_id,
   zone_id,
   quantity_kg,
   date_catch)
VALUES
-- Уловы корабля 1 (Фортуна)
(1, 1, 1, 1, 500, '2025-01-15'),  -- Лосось в Печорском заливе
(2, 1, 2, 2, 300, '2025-01-20'),  -- Кета в Кольском заливе
(3, 1, 1, 1, 450, '2025-02-10'),  -- Лосось в Печорском заливе
(4, 1, 5, 3, 200, '2025-02-25'),  -- Треска в Русской банке
```

```
-- Уловы корабля 2 (Прометей)
(5, 2, 3, 3, 700, '2025-02-10'),  -- Минтай в Русской банке
(6, 2, 4, 4, 400, '2025-03-05'),  -- Сельдь в Центральной акв.
(7, 2, 3, 3, 650, '2025-03-20'),  -- Минтай в Русской банке

-- Уловы корабля 3 (Альбатрос)
(8, 3, 4, 4, 1000, '2025-03-05'), -- Сельдь в Центральной акв.
(9, 3, 5, 5, 800, '2025-04-01'),  -- Треска в Шпицберг. акв

-- Уловы корабля 4 (Академик В.В. Рыбкин)
(10, 4, 5, 5, 600, '2025-04-15'), -- Треска в Шпицберг. акв
(11, 4, 1, 1, 350, '2025-05-01'), -- Лосось в Печорском заливе
(12, 4, 2, 2, 250, '2025-05-10'); -- Кета в Кольском заливе
```

Листинг можно найти в электронном приложении.

6.2. Агрегатные функции

Имея дело с данными, очень часто мы сталкиваемся с необходимостью посчитать сумму, среднее значение, найти минимальное или максимальное значение. В SQL в данных подсчетах нам помогают **агрегатные функции**.

6.2.1. DISTINCT — вывод уникальных записей

Одним из ключевых инструментов, тесно связанным с агрегатными функциями, является оператор **DISTINCT**, позволяющий исключить повторяющиеся значения из результирующего набора данных.

Сравните результаты двух разных запросов:

```
SELECT date_catch AS dates FROM catches;  -- все записи (12)
```

и

```
SELECT DISTINCT(date_catch) AS dates FROM catches;
```

В первом запросе дважды будут повторяться даты 2025-02-10 и 2025-03-05, во втором будут выбраны только уникальные значения дат, поэтому повторяющиеся даты будут исключены.

В скобках указывается столбец, по которому необходимо собрать уникальные значения.

6.2.2. COUNT — подсчет количества записей

Подсчитывает количество значений в таблице, удовлетворяющих определенным условиям.

```
-- Всего количество уловов:
SELECT COUNT(*) FROM catches; -- 12
-- Количество уловов трески (fish_id = 5):
SELECT COUNT(*) FROM catches WHERE fish_id = 5; -- 3
```

В качестве аргумента (значение в скобках) функция count может принимать следующие значения:

- **COUNT(*)** — подсчет общего количества записей, удовлетворяющих условиям запроса.
- **COUNT(*имя_столбца*)** — подсчет количества значений в результате, где значение столбца не NULL: `SELECT COUNT(fish_id) FROM catches`.
- **COUNT(1)** — аналог **COUNT(*)**, где 1 — константа, не связанная с данными таблицы.
- **COUNT (DISTINCT *имя_столбца*)** — подсчет количества **уникальных** значений для заданного столбца: `SELECT COUNT(DISTINCT(fish_id)) FROM catches; -- 5`.

Когда вам нужно просто подсчитать общее количество записей, используйте **COUNT(*)**. Большинство СУБД оптимизированы под запросы вида **COUNT(*)** с точки зрения производительности, это самый быстрый вариант.

COUNT(1) чуть медленнее, но тоже довольно быстр, а **COUNT(*имя столбца*)** — самый медленный вариант, так как в случае его использования анализируются еще и значения.

6.2.3. SUM — сумма

Чтобы просуммировать значения из результата, используется функция SUM. Например, для получения общей массы всех уловов мы можем воспользоваться таким запросом:

```
SELECT SUM(quantity_kg) "Общий улов" FROM catches; -- 6200 кг
```

В скобках мы указываем имя столбца, по которому необходимо просуммировать данные.

Общий улов трески (fish_id = 5):

```
SELECT SUM(quantity_kg) "Общий улов трески"
FROM catches
WHERE fish_id = 5; -- 1600 кг
```

Ключевые особенности:

- Игнорирование NULL: SUM автоматически пропускает значения NULL.
- SUM работает только с числовыми типами данных (FLOAT, INT, DECIMAL...). Попытка вызвать функцию с нечисловым типом (например, VARCHAR) вызовет ошибку.
- Если строк в таблице нет, SUM вернет NULL, а не 0:

```
SELECT SUM(quantity_kg) "Общий улов"
FROM catches
WHERE ship_id = 10; -- Несуществующее судно, результат NULL
```

6.2.4. MIN/MAX — минимальное и максимальное значения

Функции для нахождения минимального и максимального значения в выборке:

- Самая ранняя дата улова:

```
SELECT MIN(date_catch) FROM catches; --  2025-01-15
```

- Самый большой улов:

```
SELECT MAX(quantity_kg) FROM catches; -- 1000 кг
```

Аналогично SUM, функции MIN и MAX игнорируют значения NULL при подсчете и возвращают NULL, если нет записей, удовлетворяющих условиям запроса, или все значения имеют значение NULL.

В то же время, в отличие от SUM, функции MIN и MAX также могут работать с датой и временем и со строковыми данными (учитывается порядок в алфавитной сортировке с учетом текущей колляции, **а не длина строки**!):

```
SELECT MAX(ship_name) FROM ships; -- Фортуна
```

6.2.5. AVG — среднее значение

Функция AVG вычисляет среднее арифметическое значений в столбце:

```
SELECT AVG(quantity_kg) FROM catches;
```

Функция работает только с числовыми значениями, не работает с датами, строками, булевыми типами. Как и другие функции, игнорирует значения NULL и возвращает NULL, если других значений нет.

В зависимости от СУБД, тип возвращаемого значения может различаться — особенно важно это для MS SQL, где для целых типов возвращаемое значение тоже будет целым числом (дробная часть отбрасывается, а не округляется по правилам математического округления):

<table>
<tr><th>Исходный тип данных</th><th>MySQL</th><th>PostgreSQL</th><th>MS SQL</th></tr>
<tr><td>TINYINT, SMALLINT, INT</td><td rowspan="4">DECIMAL(65,4)</td><td rowspan="3">NUMERIC</td><td>INT</td></tr>
<tr><td>BIGINT</td><td>BIGINT</td></tr>
<tr><td>DECIMAL(p, s)</td><td>DECIMAL(38, MAX(s,6))</td></tr>
<tr><td>FLOAT/REAL</td><td>DOUBLE</td><td>FLOAT</td></tr>
</table>

Соответственно, результат предыдущего запроса будет отличаться в разных СУБД:

```
MySQL: 516.6667
PostgreSQL: 516.666666666666667
MSSQL: 516
```

6.2.6. COALESCE — замещение NULL

Эту функцию часто считают агрегатной функцией, хотя технически это скалярная функция, которая возвращает первое ненулевое выражение из списка аргументов.

Данная функция очень полезна в сочетании с другими агрегатными функциями для обработки данных, где может встретиться NULL.

Предположим, мы хотим посчитать средний улов по несуществующему кораблю:

```
SELECT AVG(quantity_kg) FROM catches WHERE ship_id = 999;
```

Результат будет NULL, что не очень удобно, если он будет использован в дальнейших расчетах. Здесь на помощь приходит функция COALESCE:

```
SELECT
COALESCE(AVG(quantity_kg), 0)
FROM catches WHERE ship_id = 999;
```

Мы говорим СУБД, чтобы она вывела первое значение, которое не будет являться NULL, из следующих двух аргументов: AVG(quantity_kg) или 0.

В случае если функция AVG получит в результате NULL, COALESCE вместо этого заменит его значением 0.

COALESCE работает с любым типом данных, однако список аргументов должен быть одного типа. Например, следующий запрос вернет ошибку:

```
-- Ошибка:
SELECT COALESCE(AVG(quantity_kg), 'НЕТ ДАННЫХ') FROM catches
WHERE ship_id = 555;
```

6.3. Группировка результатов GROUP BY

Агрегатные функции полезны сами по себе, однако полная их мощность становится очевидна с использованием **группировок**.

Вам может понадобиться сегментировать данные, разделить их на группы и рассчитать показатели для каждой группы индивидуально. Именно для этого в SQL предусмотрен специализированный оператор **GROUP BY**, позволяющий разбивать данные на сегменты по одному или нескольким признакам.

Чтобы лучше понимать, как работают группировки, начните с вопроса: в каком разрезе вы хотите получить данные? Например, вы хотите узнать средний улов по зонам ловли, соответственно, сгруппировать данные вам необходимо по столбцу *zone_id*:

```
SELECT
   z.region_name Акватория,
   ROUND(AVG(c.quantity_kg), 2) "Средний улов"
FROM catches c
JOIN zones z ON z.zone_id = c.zone_id
GROUP BY z.region_name;
```

В результате СУБД сгруппирует данные по зонам вылова (мы указали столбец, по которому нужно производить группировку следом за GROUP BY) и по каждой категории подсчитает среднее значение (примечание: функция ROUND округляет аргумент до указанного количества знаков после запятой — в нашем случае 2).

Акватория	**Средний улов**
Кольский залив	275.00
Центральная акватория	700.00
Шпицбергенская акватория	700.00
Русская банка	516.67
Печорский залив	433.33

Если вы хотите получить суммарный улов по каждому из кораблей, укажите соответствующее поле после оператора GROUP BY:

```
SELECT
s.ship_name Судно,
SUM(c.quantity_kg) "Общий улов"
```

```
FROM catches c
JOIN ships s ON s.ship_id = c.ship_id
GROUP BY s.ship_name;
```

Аналогичным образом запрос соединяет таблицы **ships** и **catches**, затем делит строки на группы по полю *ship_name* и считает общую массу уловов при помощи функции SUM:

Судно	Общий улов
Академик В. В. Рыбкин	1200
Альбатрос	1800
Прометей	1750
Фортуна	1450

В SQL также возможно задавать несколько параметров для группировки. Допустим, мы хотим узнать суммарный улов по каждому виду рыбы по каждой зоне ловли. Запрос будет выглядеть следующим образом:

```
SELECT
f.species Вид, z.region_name Акватория, SUM(c.quantity_kg) Улов
FROM catches c
JOIN zones z ON z.zone_id = c.zone_id
JOIN fish f ON f.fish_id = c.fish_id
GROUP BY f.species, z.region_name
```

Вид	Акватория	Улов
Кета	Кольский залив	550
Лосось	Печорский залив	1300
Минтай	Русская банка	1350
Треска	Русская банка	200
Сельдь	Центральная акватория	1400
Треска	Шпицбергенская акватория	1400

Запрос выполняется поэтапно:

1. Объединение таблиц (JOIN).
2. Формирование групп (GROUP BY).
3. Агрегатные функции (SUM, AVG, COUNT и т.д.).

Общее правило: если после SELECT вы указали имя столбца без использования агрегатной функции, это же имя вы должны включить в группировку GROUP BY, в противном случае возникнет ошибка:

```
-- ОШИБКА - f.species не включена в группировку:
SELECT
f.species Вид, z.region_name Акватория, SUM(c.quantity_kg) Улов
FROM catches c
JOIN zones z ON z.zone_id = c.zone_id
JOIN fish f ON f.fish_id = c.fish_id
GROUP BY z.region_name
```

В MySQL это требование не обязательное, однако это противоречит стандарту SQL и может приводить к неочевидным и непредсказуемым результатам, так как MySQL возвращает первое встреченное значение для неагрегированных столбцов.

6.4. Фильтрация результатов по группам — HAVING

После группировки данных часто требуется отобрать только те группы, которые удовлетворяют определенным условиям. Например, компания может захотеть узнать, в каких зонах ловли средний улов превышает 500 кг или какие корабли выловили более 1000 кг рыбы. Для такой фильтрации используется оператор **HAVING**.

Представьте, что вы сгруппировали данные по кораблям и посчитали их общий улов. Теперь нужно выбрать только корабли с уловом больше 1500 кг. Если попробовать использовать WHERE SUM(quantity_kg) > 1500, возникнет ошибка, так как WHERE работает с отдельными строками и фильтрует их ДО группировки, а не с агрегированными результатами. Как только вам

SKLADCHIK.ORG

понадобится отфильтровать результаты по значениям агрегированных функций, вам понадобится оператор **HAVING**.

> **WHERE** фильтрует по данным столбцов, **HAVING** фильтрует по значениям агрегатных функций.

Давайте найдем зоны, где суммарный объем выловленной трески превышает 1 тыс. килограмм:

```
SELECT
    z.region_name AS "Акватория",
    SUM(c.quantity_kg) AS "Улов трески"
FROM catches c
JOIN zones z ON c.zone_id = z.zone_id
JOIN fish f ON c.fish_id = f.fish_id
WHERE f.species = 'Треска'
GROUP BY z.region_name
HAVING SUM(c.quantity_kg) > 1000;
```

Полученный результат:

Акватория	Улов трески
Шпицбергенская акватория	1400

На русской банке было выловлено всего 200 кг трески, поэтому в результат данного запроса эти данные не попали. Попытка использовать WHERE SUM (…) вызовет ошибку, так как на этапе фильтрации данных агрегатные суммы еще не вычислены.

Запрос выполняется в следующем порядке:

Порядок выполнения запроса

1. FROM/JOIN — загрузка данных из таблиц.
2. WHERE — фильтрация строк (например, только треска).
3. GROUP BY — группировка по зонам.

4. HAVING — фильтрация групп (сумма > 1000).
5. SELECT — вывод результатов.

ВАЖНО!

Может сложиться впечатление, что раз функция SUM в запросе была указана дважды, она и вычисляться будет дважды. Агрегатные функции в HAVING не вычисляются дважды — СУБД использует промежуточные результаты группировки.

ПРИМЕЧАНИЕ.

MySQL позволяет использовать псевдонимы в части HAVING, при этом обязательно использовать символ обратного апострофа для псевдонима, содержащего пробелы:

```
SELECT
    z.region_name AS 'Акватория',
    SUM(c.quantity_kg) AS 'Улов трески'
FROM catches c
JOIN zones z ON c.zone_id = z.zone_id
JOIN fish f ON c.fish_id = f.fish_id
WHERE f.species = 'Треска'
GROUP BY z.region_name
HAVING 'Улов трески' > 1000;
```

6.5. Некоторые нестандартные агрегатные функции в разных СУБД

СУБД, с которой вы работаете, скорее всего, имеет множество нестандартных агрегатных функций, которые могут вам помочь в определенных ситуациях. Перечень, приведенный ниже, — далеко не исчерпывающий, прочитайте соответствующий раздел документации по вашей СУБД, чтобы ознакомиться с полным перечнем агрегатных функций.

6.5.1. GROUP_CONCAT (MySQL) / STRING_AGG (PostgreSQL/ MySQL)

Объединяет все строки через указанный разделитель. Например, с помощью этой функции можно вывести акватории и списки кораблей, которые вели в них рыбную ловлю.

В функции указывается имя столбца для группировки и далее — строка-разделитель, которая будет использоваться при объединении.

В PostgreSQL и MS SQL Server эта функция называется STRING_AGG (агрегация строк):

```
SELECT
    z.region_name AS "Акватория",
    STRING_AGG(s.ship_name, ', ') AS "Суда"
FROM catches c
JOIN zones z ON c.zone_id = z.zone_id
JOIN ships s ON s.ship_id = c.ship_id
GROUP BY z.region_name;
```

В MySQL используется функция GROUP_CONCAT (групповая конкатенация):

```
SELECT
    z.region_name AS "Акватория",
    GROUP_CONCAT(s.ship_name, ', ') AS "Суда"
FROM catches c
JOIN zones z ON c.zone_id = z.zone_id
JOIN ships s ON s.ship_id = c.ship_id
GROUP BY z.region_name;
```

Результат:

Акватория	Суда
Кольский залив	Фортуна, Академик В. В. Рыбкин
Печорский залив	Фортуна, Фортуна, Академик В. В. Рыбкин
Русская банка	Фортуна, Прометей, Прометей

Акватория	Суда
Центральная акватория	Прометей, Альбатрос
Шпицбергенская акватория	Альбатрос, Академик В. В. Рыбкин

6.5.2. STDDEV_POP / STDEVP

Рассчитывает стандартное отклонение для всей совокупности данных (population standard deviation).

Пример:

```
-- Стандартное отклонение уловов по всем кораблям
SELECT STDDEV_POP(quantity_kg) FROM catches;
```

Результат: для данных из таблицы catches ≈ 230.33 кг.

Поддерживаются MySQL, PostgreSQL, аналог для MSSQL: **STDEVP()**.

6.5.3. STDEV_SAMP / STDEV

Рассчитывает стандартное отклонение для выборки (sample standard deviation).

```
-- Стандартное отклонение уловов по выборке (например, для двух
кораблей)
SELECT STDDEV_SAMP(quantity_kg)
FROM catches WHERE ship_id = 1 or ship_id = 2
```

Результат: ≈ 178.95 кг

Поддерживаются MySQL, PostgreSQL, аналог для MSSQL: **STDEV()**.

6.5.4. VAR_SAMP / VAR

Рассчитывает дисперсию для выборки (sample variance).

```
-- Дисперсия уловов по Русской банке (zone_id = 3)
SELECT VAR_SAMP(quantity_kg) FROM catches WHERE zone_id = 3;
```

Результат: ≈ 75833.33

Поддерживаются MySQL, PostgreSQL, аналог для MSSQL: **VAR()**.

6.5.5. PostgreSQL: WITHIN GROUP и PERCENTILE_CONT / PERCENTILE_DISC

Некоторые функции, например эта, требуют упорядочивания результатов перед применением. В отличие от ORDER BY, который сможет отсортировать итоговый результат запроса, конструкция WITHIN GROUP сортирует результаты до применения функции, такой как PERCENTILE_CONT (возвращает процентиль на основе непрерывного распределения) или PERCENTILE_DISC (процентиль на основе дискретного распределения).

Пример:

```
-- Вычисление медианы уловов (50-й процентиль)
SELECT PERCENTILE_CONT(0.5)
WITHIN GROUP (ORDER BY quantity_kg) AS median
FROM catches;
```

Результат (PostgreSQL): 475.

ПРИМЕЧАНИЕ.

Конструкция WITHIN GROUP (в группе) снижает переносимость кода. В общем случае ее стоит избегать, так как имеется более дружественное стандарту SQL решение. По этой причине в MySQL и MS SQL Server данный функционал не поддерживается напрямую, требует использования оконных функций, которые мы рассмотрим позднее в этой книге.

Что узнали и чему мы научились?

В этой главе вы научились:

- Применять агрегатные функции (SUM, AVG).
- Группировать эти данные с помощью GROUP BY.
- Фильтровать группы через оператор HAVING.

Эти навыки помогут вам создавать детальные сводные отчеты, однако сложные запросы часто требуют многоуровневой обработки данных.

Например, как сравнить улов каждого корабля со средним по флоту? Или как организовать временные результаты для повторного использования? Решать подобные задачи и оптимизировать сложные вычисления, работая с временными наборами данных, мы научимся в следующей главе.

Глава 7.

Подзапросы и обобщённые табличные выражения (CTE)

7.1. Данные на основании других данных. Практический пример компьютерной игры

Мы продвинулись уже достаточно далеко в изучении SQL и можем объединять и группировать данные, вычленяя нужные нам факты. Однако одной из распространенных проблем остается необходимость работы с многоуровневыми данными, когда требуется не только извлекать информацию, но и обрабатывать ее в контексте других данных. Привычные методы могут просто не справиться с такой задачей и потребовать сложных манипуляций, которые могут привести к неэффективности и усложнению кода.

Начнем, как всегда, с примера. Предположим, мы устроились в компанию, которая производит компьютерные игры, и как раз сейчас от нас требуется создать метаданные для нее. В игре игроки покоряют галактику, исследуя различные технологии. Каждая новая технология зависит от предыдущей, образуя сложное дерево.

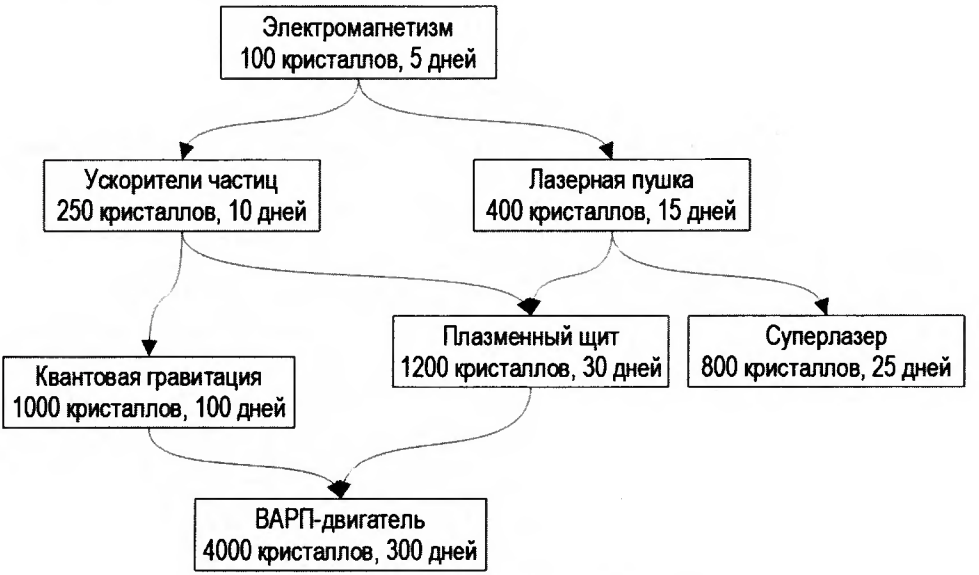


Рис. 7.1. *Дерево технологий для компьютерной игры*

Как узнать, сколько ресурсов нужно, чтобы открыть нужную технологию? Как сообщить игроку информацию, какие технологии нужно открыть, чтобы добраться до нужной?

Создадим базу данных:

```
CREATE DATABASE gamedev;
```

Сущность технологии имеет свойства: название, стоимость, длительность исследования.

Создадим таблицу:

```
-- Таблица технологий
CREATE TABLE technologies (
    tech_id INT PRIMARY KEY,
    tech_name VARCHAR(100) NOT NULL,
    crystals_cost INT NOT NULL,
    days_required INT NOT NULL
);
```

Кроме того, каждая технология имеет зависимость от других технологий, что делает необходимым создание другой сущности, отвечающей за хранение связей типа *многие-ко-многим:*

```
-- Таблица зависимостей технологий
CREATE TABLE tech_dependencies (
    tech_id INT NOT NULL,
    required_tech_id INT NOT NULL,
    FOREIGN KEY (tech_id) REFERENCES technologies(tech_id),
    FOREIGN KEY (required_tech_id)
      REFERENCES technologies(tech_id),
    PRIMARY KEY (tech_id, required_tech_id)
);
```

В этой таблице мы имеем два внешних ключа на основную таблицу: технологию (tech_id) и требуемую технологию для открытия (required_tech_id). Для обеспечения уникальности используется составной первичный ключ из этих двух комбинаций.

Каждая запись в этой таблице будет представлять стрелку на схеме дерева технологий.

Наполним таблицу данными (листинг доступен в электронном приложении):

```
-- Вставка данных о технологиях
INSERT INTO technologies
(tech_id, tech_name, crystals_cost, days_required)
VALUES
(1, 'Электромагнетизм', 100, 5),
(2, 'Ускорители частиц', 250, 10),
(3, 'Лазерная пушка', 400, 15),
(4, 'Квантовая гравитация', 1000, 100),
(5, 'Плазменный щит', 1200, 30),
(6, 'Суперлазер', 800, 25),
(7, 'ВАРП-Двигатель', 4000, 300);

-- Вставка зависимостей
INSERT INTO tech_dependencies
(tech_id, required_tech_id)
VALUES
-- Ускорители частиц зависят от Электромагнетизма
(2, 1),
-- Лазерная пушка зависит от Электромагнетизма
(3, 1),
-- Квантовая гравитация зависит от Ускорителей частиц
(4, 2),
-- Плазменный щит зависит от Ускорителей частиц
-- и Лазерной пушки
(5, 2),
(5, 3),
-- Суперлазер зависит от Лазерной пушки
(6, 3),
-- ВАРП-Двигатель зависит от Квантовой гравитации
-- и Плазменного щита
(7, 4),
(7, 5);
```

SKLADCHIK.ORG

7.2. Подзапросы

Подзапрос — это SQL-запрос, встроенный в другой запрос. Он позволяет разбивать сложные задачи на части.

7.2.1. Подзапрос, вложенный в SELECT

Предположим, надо вывести технологии и их стоимость в общем объеме исследований:

```
SELECT
    tech_name,
    crystals_cost,
    (crystals_cost * 100 / (SELECT SUM(crystals_cost) FROM
technologies)) AS cost_percent
FROM technologies;
```

Прямо в основном SELECT в качестве одного из столбцов результата мы запрашиваем отношение стоимости данной технологии к суммарной стоимости всех технологий, вместе взятых, присвоив данному столбцу псевдоним *cost_percent*. Правила синтаксиса требуют, чтобы подзапрос был взят в круглые скобки ().

Результат:

tech_name	crystals_cost	cost_percent
Электромагнетизм	100	1
Ускорители частиц	250	3
Лазерная пушка	400	5
Квантовая гравитация	1000	12
Плазменный щит	1200	15
Суперлазер	800	10
ВАРП-Двигатель	4000	51

7.2.2. Подзапрос, вложенный в WHERE

В данном случае мы можем проверять условия на основании подзапроса. Например, вывести технологии, которые дешевле средней стоимости:

```
SELECT tech_name
FROM technologies
WHERE crystals_cost <
   (SELECT AVG(crystals_cost) FROM technologies);
```

Не пугайтесь, подобный подзапрос не будет выполняться для каждой строки. СУБД сначала выполнит подзапрос и вычислит средний показатель стоимости технологии, а уже полученное число будет применяться ко всем строкам основной таблицы.

tech_name
Электромагнетизм
Ускорители частиц
Лазерная пушка
Квантовая гравитация
Суперлазер

Для подобных вычислений существуют более эффективные способы. Речь о них пойдет речь чуть дальше в этой главе.

7.2.3. Подзапрос, вложенный в FROM

Используется, когда требуется выборка данных из другой выборки:

```
-- Все технологии со временем исследования больше 20 дней
SELECT *
FROM (
   SELECT * FROM technologies WHERE days_required > 20
   ) AS long_research;
```

7.2.4. Коррелируемый подзапрос

Давайте используем почти все, что мы изучили ранее, и построим сложный запрос:

```
-- Все технологии с полным временем разработки,
-- включая предшественников и их количество.
SELECT main.tech_name,
COALESCE(dependencies.num_techs, 0) required_count,
COALESCE(dependencies.sum_days, 0) total_days
FROM technologies AS main
LEFT JOIN (
    SELECT
          td.tech_id,
          COUNT(1) AS num_techs,
          SUM(t.days_required) AS sum_days
    FROM tech_dependencies AS td
    JOIN technologies AS t ON td.required_tech_id = t.tech_id
    GROUP BY td.tech_id
) AS dependencies ON main.tech_id = dependencies.tech_id;
```

Разберемся, что здесь происходит.

Начинается выполнение с подзапроса, который высчитывает количество зависимостей (num_techs) и суммирует время, требуемое на исследование (sum_days), группируя результаты по каждой из технологий.

Затем полученная таблица присоединяется к основной таблице технологий при помощи левого внешнего объединения по совпадению идентификатора tech_id (напомним, что такой вид объединения оставляет все записи исходной таблицы, подставляя NULL в качестве значений присоединяемой таблицы, в случае если не нашлось сопоставления).

В конце такие NULL заменяются значениями 0 при помощи функции COALESCE, что приводит в финале вот к такому результату:

tech_name	required_count	total_days
Электромагнетизм	0	0
Ускорители частиц	1	5

tech_name	required_count	total_days
Лазерная пушка	1	5
Квантовая гравитация	1	10
Плазменный щит	2	25
Суперлазер	1	15
ВАРП-Двигатель	2	130

7.2.5. Проверка на наличие в наборе значений IN

Чтобы проверить, входит ли искомое значение в заданный список или результат подзапроса, используется оператор **IN.** Мы можем просто указать его аргументами интересующий нас список:

```
SELECT tech_name
FROM technologies
WHERE tech_id IN (1, 3, 6);
```

Выбрать технологию, чей идентификатор находится **В** списке (1, 3, 6).

Данный запрос эквивалентен использованию оператора OR, однако синтаксис будет более громоздким и сложно читаемым:

```
SELECT tech_name
FROM technologies
WHERE tech_id = 1 OR tech_id = 3 OR tech_id = 6;
```

Кроме того, список можно задавать не только явным образом, но и в виде подзапроса:

```
-- Выбрать технологии, от которых зависит
-- ВАРП-Двигатель (ИД 7)
SELECT tech_name
FROM technologies
WHERE tech_id IN (
    SELECT required_tech_id
    FROM tech_dependencies
    WHERE tech_id = 7 -- ИД ВАРП-Двигателя
);
```

tech_name
Квантовая гравитация
Плазменный щит

Если скомбинировать оператор IN с унитарным оператором NOT IN, можно проверять отсутствие искомого значения в заданном списке:

```
-- Выбор финальных технологий,
-- от которых уже ничего не зависит
SELECT tech_name
FROM technologies
WHERE tech_id NOT IN (
    SELECT required_tech_id
    FROM tech_dependencies
    WHERE required_tech_id IS NOT NULL
);
```

tech_name
Суперлазер
ВАРП-Двигатель

ВНИМАНИЕ!

Если подзапрос в NOT IN возвращает NULL, результат может быть неожиданным:

```
-- Если подзапрос вернет NULL, NOT IN не сработает!
SELECT tech_name
FROM technologies
WHERE tech_id NOT IN (1, NULL); -- Пустой результат
```

Вместо этого используйте NOT EXISTS (см. след. раздел) или фильтруйте результат (IS NOT NULL).

7.2.6. Проверка на условие с наборами значений ANY / ALL

Если оператор IN требует точного совпадения с одним из элементов списка, то операторы ANY и ALL производят сравнение отдельного значения

с любым (ANY) элементом набора данных с использованием какого-либо другого оператора сравнения (например, =, <, >) либо со всеми (ALL) ними.

```
-- Найти технологии, исследование которых
-- занимает меньше времени, чем любая
-- другая технология стоимостью до 800 кристаллов
SELECT tech_name
FROM technologies
WHERE days_required <= ANY
 (SELECT days_required
  FROM technologies
  WHERE crystals_cost < 800)
```

Сначала выбираются длительности открытия технологий, стоимость которых меньше 800 кристаллов, получается список 5, 10 и 15 дней. Затем производится запрос по общему списку и отбираются только те, длительность которых меньше либо равна ЛЮБОМУ (ANY) из перечисленных значений. В результате получим Электромагнетизм, Ускорители частиц и Лазерную пушку.

Соответственно, в случае если мы используем ALL вместо ANY, то получим только электромагнетизм, так как только у данной технологии длительность исследования меньше либо равна ВСЕМ (ALL) найденным в подзапросе значениям.

7.2.7. Быстрая проверка наличия WHERE EXISTS

Иногда нам достаточно просто удостовериться, а возвращает ли подзапрос хоть что-нибудь, не важно что. В этом случае можно использовать оператор проверки наличия записей EXISTS (существует ли?). В силу того, что поиск останавливается сразу, как только находится хоть одна запись, удовлетворяющая условиям запроса, это позволяет существенно ускорить быстродействие. Читаемость кода тоже улучшается.

Синтаксис:

```
SELECT *
FROM таблица1
WHERE EXISTS (
```

```
    SELECT 1
    FROM таблица2
    WHERE таблица2.столбец = таблица1.столбец
);
```

Например, чтобы получить список технологий без зависимостей, мы можем использовать уже известным нам JOIN:

```
SELECT DISTINCT t.tech_name
FROM technologies t
JOIN tech_dependencies td ON t.tech_id = td.tech_id;
```

Или воспользоваться WHERE EXISTS:

```
SELECT tech_name
FROM technologies t
WHERE EXISTS (
    SELECT 1
    FROM tech_dependencies td
    WHERE td.tech_id = t.tech_id
);
```

Оба запроса выведут один и тот же результат, однако, поскольку нам не требуются столбцы из связанной таблицы, WHERE EXISTS будет выполняться гораздо быстрее из-за того, что проверка условий присоединения прекращается сразу же, как только будет найден хоть один удовлетворяющий условиям результат (td.tech_id = t.tech_id), а в случае с JOIN поиск может продолжаться, пока не будет просканирована вся связанная таблица.

ПРИМЕЧАНИЕ.

Если вы используете связанные данные только для проверки условий или проверки наличия связи, при этом вам не требуются сами данные из связанной таблицы, используйте WHERE EXISTS вместо JOIN.

```
-- Проверка по условию
-- Найти технологии, которые требуют Плазменного щита
SELECT tech_name
FROM technologies t
WHERE EXISTS (
    SELECT 1
    FROM tech_dependencies td
    WHERE td.tech_id = t.tech_id
        AND td.required_tech_id = 5 -- ID Плазменного щита
);
```

Результатом будет ВАРП-двигатель.

Чтобы, наоборот, выполнить проверку на отсутствие связи, скомбинируйте WHERE EXISTS с унитарным оператором NOT:

```
-- Технологии без зависимостей
SELECT tech_name
FROM technologies t
WHERE NOT EXISTS (
    SELECT 1
    FROM tech_dependencies td
    WHERE td.tech_id = t.tech_id
); -- Электромагнетизм
```

7.2.8. Когда использовать подзапросы

Подзапросы могут быть мощным инструментом, однако их использование может серьезно снижать производительность системы, поэтому придерживайтесь следующих правил при их использовании:

- В вашем запросе только один подзапрос.
- Используйте их в простых условиях с WHERE или SELECT.
- Нужно получить значение для каждой строки (коррелированные запросы).
- Нужно быстро проверить наличие или отсутствие записей (EXISTS).

Язык SQL очень часто позволяет добиться одних и тех же результатов разными путями и подходами, поэтому очень полезно знать не один, а несколько путей реализации поставленной задачи. Используйте советы, приведенные здесь, с известной долей сомнений, так как каждая конкретная задача в каждой конкретной СУБД может потребовать разных путей для оптимизации. Позднее мы еще вернемся к теме оптимизации запросов.

7.3. Проблемы подзапросов

7.3.1. Сложность для восприятия

В сложных ситуациях количество подзапросов может серьезно затруднять читаемость кода, превращая его в лабиринт.

```
SELECT …
FROM (
    SELECT …
    FROM …
    WHERE … (
        SELECT … FROM …
        WHERE … IN (SELECT … FROM…)
    )
);
```

7.3.2. Повторяющаяся логика

Если подзапрос используется несколько раз, СУБД будет выполнять его каждый раз заново:

```
SELECT
 tech_name,
 (
  SELECT AVG(crystals_cost)
  FROM technologies
 ) avg_cost,
 crystals_cost -
```

```
 (
    SELECT AVG(crystals_cost)
    FROM technologies
 ) AS diff
FROM technologies;
```

В этом запросе дважды будет выполнен один и тот же подзапрос:

```
SELECT AVG(crystals_cost) FROM technologies
```

7.3.3. Производительность

Коррелированные подзапросы могут выполняться для каждой строки, что медленно на больших данных:

```
SELECT tech_name,
    (SELECT COUNT(*) FROM tech_dependencies td
     WHERE td.tech_id = t.tech_id)
FROM technologies t;
```

7.4. Обобщенные табличные выражения

7.4.1. Решение неэффективности подзапросов

CTE (*Common Table Expressions — обобщенные табличные выражения*) представляют собой временный набор данных, который создается в рамках одного SQL-запроса и существует лишь на протяжении его выполнения. Они позволяют разбивать сложный запрос на отдельные части, делая код проще для понимания и обслуживания.

Обычные подзапросы часто приводят к повторному выполнению одних и тех же операций, особенно если одна и та же логика встречается многократно. Например, когда один и тот же подзапрос используется дважды или трижды, СУБД вынуждена каждый раз заново его обрабатывать. Это увеличивает нагрузку на систему и замедляет выполнение запроса.

SKLADCHIK.ORG

CTE решает эту проблему, создавая временную таблицу с результатами вычислений, которую можно повторно использовать в запросе. Таким образом, сложные вычисления выполняются всего один раз, а полученные данные применяются далее без повторного расчета.

Кроме того, использование CTE улучшает читаемость и структурированность запросов, позволяя разделить сложную логику на понятные этапы. Такой подход облегчает поддержку и развитие кода, уменьшает вероятность ошибок и упрощает процесс тестирования отдельных компонентов запроса.

7.4.2. Как это работает

При выполнении запроса с CTE база данных создает временную виртуальную таблицу с указанным именем и структурой, определяемой инструкциями внутри CTE. Эта временная таблица доступна только в пределах текущего запроса и исчезает сразу после завершения его исполнения.

Основной принцип работы заключается в следующем:

- Определенная CTE вычисляется единожды.
- Результаты сохраняются временно и используются в дальнейшем.
- Другие компоненты запроса обращаются к данным из CTE, минуя необходимость многократного повторения одних и тех же вычислений.

Предположим, у нас такой запрос:

```
SELECT tech_name
FROM technologies
WHERE crystals_cost >
  (SELECT AVG(crystals_cost) FROM technologies);
```

Заменим подзапрос на CTE:

```
WITH AverageCost AS (
    SELECT AVG(crystals_cost) AS avg_cost
    FROM technologies
)
```

```
SELECT tech_name
FROM technologies, AverageCost
WHERE crystals_cost > avg_cost;
```

Мы используем ключевое слово WITH, указывающее на то, что далее мы хотим описать CTE, после чего указываем название нашей временной таблицы (AverageCost). После в скобках мы указываем сам текст нашего запроса, результаты которого мы будем использовать в дальнейшем.

Далее мы просто оперируем данными из AverageCost, будто бы они содержались в отдельной таблице.

Если в первом варианте подзапрос будет снова и снова выполняться для каждой строки, то значение в таблице AverageCost будет посчитано один раз и далее, сколько бы раз мы к ней не обращались, никаких дополнительных вычислений СУБД уже производить не будет.

Читать можно так: "С [использованием данных] AverageCost (описание) ВЫБЕРИ … ИЗ AverageCost…".

Код становится гораздо яснее и легче для восприятия. Логика расчета средней стоимости вынесена в отдельную секцию (AverageCost), что отделяет ее от основного тела запроса. Теперь будет проще разобраться, что происходит в каждом блоке, а также поддерживать и изменять запрос в будущем — мы можем изменить логику CTE, не меняя логики основного запроса, и наоборот.

7.4.3. Простые CTE

Мы уже использовали скалярное значение средней стоимости технологий AverageCost в качестве CTE, но, разумеется, мы можем использовать и более сложные конструкции, например, присоединяя данные из CTE к основному запросу:

```
WITH TechDependencyCount AS (
    SELECT required_tech_id, COUNT(*) AS dependency_count
    FROM tech_dependencies
    GROUP BY required_tech_id
)
SELECT
```

```
  t.tech_name Технология,
  COALESCE(d.dependency_count, 0) "Кол-во зависимостей"
FROM technologies t
LEFT JOIN TechDependencyCount d
       ON t.tech_id = d.required_tech_id;
```

Мы высчитали количество предшествующих технологий по каждой и запомнили временный результат TechDependencyCost. Затем мы присоединяем полученные результаты к основной таблице.

ОБРАТИТЕ ВНИМАНИЕ:

для адресации столбцов в CTE необходимо использовать псевдонимы. В данном случае мы используем псевдоним dependency_count в качестве названия столбца и ссылаемся на него из основного запроса.

7.4.4. Рекурсия в CTE

По-научному наше дерево технологий называется графом, и в SQL придуманы методики его обхода. В условиях реальной разработки подобной игры узлов в таком графе могут быть сотни и даже тысячи.

Там, где мы имеем дело с иерархическими данными, мы можем использовать технику **рекурсии**.

В общем понимании **рекурсия** — это самоподобный процесс, в программировании — прием, когда функция вызывает саму себя, в контексте SQL рекурсивный запрос — это специальный вид запроса, позволяющий обращаться к собственным результатам в процессе выполнения.

Подобные запросы состоят из двух частей:

1. Базового случая, или **якоря** (*anchor member*) — начального запроса, который возвращает корневой элемент иерархии (графа);

2. Рекурсивного вызова (recursive member) — запроса, который объединяется с текущим результатом CTE, который будет вызываться до тех пор, пока не будут обработаны все элементы.

Обе части объединяются при помощи оператора UNION ALL:

```
-- Построение списка технологий с зависимостями:
WITH RECURSIVE TechTree AS (
    -- Якорь: начальные условия
    SELECT 1 as tech_id, 0 as required_tech_id
    UNION ALL
    -- Рекурсивный член: связь с предыдущим шагом
    SELECT t.tech_id, t.required_tech_id
    FROM tech_dependencies t
    JOIN TechTree tt ON t.required_tech_id = tt.tech_id
)
SELECT * FROM TechTree;
```

ПРИМЕЧАНИЕ.

В MS SQL Server ключевое слово RECURSIVE не используется, просто удалите его.

Поскольку в таблице **tech_dependencies** у нас отсутствует корневая технология, базовый случай мы задаем явным образом:

```
SELECT 1 as tech_id, 0 as required_tech_id;
```

Базовая часть будет выглядеть следующим образом:

tech_id	required_tech_id
1	0

Первичное значение якоря (tech_id = 1) очень важно. Если при выполнении последующего рекурсивного запроса данному значению не будет найдено сопоставление, ничего, кроме базы, выведено не будет (можете в этом убедиться, заменив 1 на любой ид, не существующий в таблице технологий, например 100).

Теперь, когда у нас есть *якорь* (anchor), мы "нанизываем" на этот результат следующий, вызвав рекурсивный запрос:

```
SELECT t.tech_id, t.required_tech_id
    FROM tech_dependencies t
    JOIN TechTree tt ON t.required_tech_id = tt.tech_id
```

Будет просканирована таблица зависимостей tech_dependencies с целью сопоставления ее данных по условию совпадения значения *tech_id* в столбце *required_tech*.

Таких технологий найдется две (с идентификаторами 2 — Ускорители частиц и 3 — Лазерная пушка). К результату добавятся две строки:

tech_id	required_tech_id
1	0
2	1
3	1

Далее СУБД выполнит следующий шаг рекурсии, сопоставляя данные из таблицы **tech_dependencies** с уже имеющимися значениями 1, 2 или 3, дописав в таблицу результатов новые строки:

tech_id	required_tech_id
1	0
2	1
3	1
4	2
5	2
5	3
6	3

Следующий шаг будет искать технологии, зависимые от технологий с tech_id 4, 5 или 6:

tech_id	required_tech_id
1	0
2	1
3	1
4	2
5	2
5	3
6	3
7	4
7	5
7	5

На этом шаге рекурсия остановится, так как в таблице **tech_dependencies** не найдется технологий, зависимых от ВАРП-Двигателя.

ОБРАТИТЕ ВНИМАНИЕ:

в итоговом результате строка 7, 5 выведена дважды, так как этой технологии ведет два "маршрута". Мы можем избавиться от этого поведения, если вместо UNION ALL (Объединять все) напишем просто UNION (Исключить дубликаты), однако подобный прием существенно замедлит выполнение запроса, поэтому по возможности всегда используйте UNION ALL и удаляйте дубликаты на последующих этапах (DISTINCT) (в MSSQL использование UNION ALL обязательно).

Рекурсивный вызов можно производить в любом направлении, например с конца: посчитаем общее количество кристаллов, необходимое, чтобы открыть ВАРП-Двигатель.

Начнем с якоря (базы) — самого ВАРП-Двигателя и будем идти к началу, пока не достигнем корня дерева.

Нам потребуется 800 + 400 + 100 = 1300 кристаллов, что и покажет результат этого запроса:

```
-- Сколько кристаллов всего надо для суперлазера?
-- Уберите ключевое слово RECURSIVE для MSSQL
WITH RECURSIVE CostCalculator AS (
    SELECT t.tech_id, t.crystals_cost
    FROM technologies t
    WHERE t.tech_id = 6 -- Суперлазер
    UNION
    SELECT t.tech_id, t.crystals_cost
    FROM technologies t
    JOIN tech_dependencies td
        ON t.tech_id = td.required_tech_id
    JOIN CostCalculator cc ON td.tech_id = cc.tech_id
)
SELECT SUM(crystals_cost) AS total_crystals
FROM CostCalculator; -- 1300
```

7.4.5. Глубина рекурсии и опасность петель

Рекурсивные вызовы очень помогают при работе с иерархиями, однако в сложных графах существует опасность возникновения замкнутых петель, обнаружить которые на больших объемах данных не так-то легко.

Вот искусственный пример, как организовать петлю в рекурсивном запросе:

```
WITH RECURSIVE Loop AS (
    SELECT 1 AS num
    UNION ALL
    SELECT num + 1 FROM Loop -- Бесконечный цикл!
)
SELECT * FROM Loop;
```

С каждой итерацией мы просто прибавляем к значению num единицу и не останавливаемся.

Такая же ситуация может возникнуть и в реальных данных либо из-за ошибки, либо наоборот — как вполне штатная ситуация.

В качестве иллюстрации давайте замкнем наше дерево и создадим петлю, поставив корневой элемент (Электромагнетизм с tech_id = 1) в зависимость от верхнего "листика" (ВАРП-Двигатель с tech_id = 7):

```
INSERT INTO tech_dependencies(tech_id , required_tech_id)
VALUES (1, 7);
```

После чего запустите ранее рассмотренный запрос на построение таблицы зависимостей:

```
-- Построение списка технологий с зависимостями:
WITH RECURSIVE TechTree AS (
    -- Якорь: начальные условия
    SELECT 1 as tech_id, 0 as required_tech_id
    UNION ALL
    -- Рекурсивный член: связь с предыдущим шагом
    SELECT t.tech_id, t.required_tech_id
    FROM tech_dependencies t
    JOIN TechTree tt ON t.required_tech_id = tt.tech_id
)
SELECT * FROM TechTree;
```

В зависимости от настроек вашей СУБД ваш запрос либо уйдет в бесконечный цикл, либо остановится с ошибкой, когда количество итераций превысит максимально допустимое.

В этом плане самой защищенной от подобных ситуаций является MS SQL, которая по умолчанию ограничивает количество итераций 100 циклами. MySQL, наоборот, требует ручного контроля подобных ситуаций, а PostgreSQL хотя и предоставляет средства для контроля глубины рекурсии, но их, как и в MySQL, необходимо применять явно.

Для начала рассмотрим универсальный способ, который годится для всех СУБД: применение контрольного значения внутри запроса для контроля глубины рекурсии.

```
-- Ограничение глубины рекурсии
WITH RECURSIVE TechTree AS (
    SELECT
        tech_id,
        required_tech_id,
        1 AS depth  -- Начальный уровень
    FROM tech_dependencies
    WHERE tech_id = 1  -- Стартовая технология
    UNION ALL
    SELECT
        t.tech_id,
        t.required_tech_id,
        tt.depth + 1  -- Увеличиваем уровень
    FROM tech_dependencies t
    JOIN TechTree tt ON t.required_tech_id = tt.tech_id
    WHERE tt.depth < 10  -- Максимальная глубина
)
SELECT * FROM TechTree;
```

Здесь в базовый запрос мы добавляем 1 в качестве первоначального значения глубины рекурсии.

Каждую последующую итерацию мы прибавляем к этому значению 1, предусмотрев в рекурсивном запросе условие, что глубина не может превышать 10.

7.4.6. Техники обнаружения петель в разных СУБД

Еще одним относительно универсальным способом является построение пути обхода для обнаружения петель.

```
-- Синтаксис MYSQL
WITH RECURSIVE TechTree AS (
    SELECT
        tech_id,
        required_tech_id,
        CAST(tech_id AS CHAR(255)) AS path  -- Трекер пути
```

```
    FROM tech_dependencies
    WHERE tech_id = 1
    UNION ALL
    SELECT
        t.tech_id,
        t.required_tech_id,
        CONCAT(tt.path, '->', t.tech_id)  -- Обновляем путь
    FROM tech_dependencies t
    JOIN TechTree tt ON t.required_tech_id = tt.tech_id
    WHERE tt.path NOT LIKE CONCAT('%->', t.tech_id, '%')  -- Избегаем
циклов
)
SELECT * FROM TechTree;
```

В этом запросе используется встроенная функция **CAST**, которая преобразует аргумент к заданному типу CHAR(200), а также функция **CONCAT**, которая объединяет несколько строк в одну, тем самым формируя текущий путь обхода:

tech_id	required_tech_id	path
1	7	1
2	1	1->2
...		
6	3	1->2->5->7->1->3->6
4	2	1->3->5->7->1->2->4

ПРИМЕЧАНИЕ.

В силу особенностей реализации в PostgreSQL необходимо заменить

```
CAST(tech_id AS CHAR(255)) AS path
```

на

```
tech_id::TEXT AS path  -- Трекер пути
```

SKLADCHIK.ORG

Оператор :: в PostgreSQL позволяет приводить один тип к другому, в нашем случае тип INT, который имеет столбец tech_id, к типу TEXT (нестандартному типу данных PostgreSQL, аналогичному VARCHAR с произвольной длиной).

Листинги запросов для всех трех СУБД приведены в электронном приложении. Различия в них касаются только способов приведения одного типа данных к другому. Суть же заключается в проверке NOT LIKE, где по шаблону %->N% находится "петля" и не дает запросу зациклиться.

При использовании PostgreSQL очень удобным вместо медленных операций приведения типов и конкатенации становится использование типа данных ARRAY (массив). Вот как это можно реализовать при помощи массивов:

```
WITH RECURSIVE DependencyTree AS (
  -- Якорь: начальная технология (ВАРП-Двигатель)
  SELECT
    td.tech_id,
    td.required_tech_id,
    ARRAY[td.tech_id] AS path, -- Массив для отслеживания пути
    false AS is_cycle -- BOOLEAN признак зацикливания
    FROM tech_dependencies td
    WHERE td.tech_id = 7  -- ID ВАРП-Двигателя

    UNION ALL

  -- Рекурсия: поднимаемся к родителям
  SELECT
    td.tech_id,
    td.required_tech_id,
    dt.path || td.tech_id AS path,  -- Добавляем tech_id в путь
    td.tech_id = ANY(dt.path) AS is_cycle -- tech_id в пути?
    FROM tech_dependencies td
    JOIN DependencyTree dt ON td.tech_id = dt.required_tech_id
    WHERE NOT dt.is_cycle  -- Стоп, если признак цикла TRUE
)
-- Выводим результат, исключая циклы
SELECT * FROM DependencyTree WHERE NOT is_cycle;
```

В данном случае мы пользуемся тем, что в PostgreSQL имеется очень удобный тип данных ARRAY, который в якоре инициализируется первым элементом `ARRAY[td.tech_id] AS path`, а затем рекурсивным вызовом в конец данного массива добавляются новые элементы при помощи оператора || (конкатенация массивов).

Наконец, при помощи оператора ANY проверяется, имеется ли уже в массиве **path** текущая *tech_id*:

tech_id	required_tech_id	path	is_cycle
7	4	{7}	false
...			
1	7	{7,4,2,1}	false
1	7	{7,5,2,1}	false

Подход с использованием трекинга обхода в MS SQL Server идентичен подходу для MySQL (листинг приведен в электронном приложении). Однако в MSSQL имеется встроенный по умолчанию счетчик рекурсивных вызовов. Если количество рекурсивных запросов превысит предельное максимальное значение (по умолчанию 100), СУБД автоматически прервет выполнение запроса.

```
-- Встроенное ограничение в MS SQL
WITH Loop AS (
    SELECT 1 AS num
    UNION ALL
    SELECT num + 1 FROM Loop
)
SELECT * FROM Loop; -- ПРЕРВЕТСЯ НА 100 ИТЕРАЦИЯХ
```

В случае с большими графами количество итераций может превышать данный порог, поэтому в MS SQL предусмотрена дополнительная опция MAXRECURSION, которой можно задать индивидуальное количество рекурсивных вызовов. Просто добавьте в конец запроса OPTION(MAXRECURSION N), где N — новое количество итераций:

```
-- Встроенное ограничение в MS SQL
WITH Loop AS (
```

```
    SELECT 1 AS num
    UNION ALL
    SELECT num + 1 FROM Loop
)
SELECT * FROM Loop
OPTION (MAXRECURSION 200); -- ПРЕРВЕТСЯ НА 200 ИТЕРАЦИЯХ
```

7.4.7. Где рекурсия может быть полезна, а где — вредна

Рекурсия в SQL — это мощный инструмент, который может значительно упростить многие задачи, связанные с иерархическими данными и комплексными связанными структурами.

Типовые ситуации, где рекурсия может оказаться вам полезной:

- **Организационные структуры организации**. Рекурсия может быть использована для извлечения данных из таблицы сотрудников (отделов), чтобы определить, кто кому подчиняется.
- **Структуры продуктовых категорий и подкатегорий**. В интернет-магазине рекурсивные запросы помогают извлекать товары и их подкатегории, позволяя вывести все уровни вложенности.
- **Деревья комментариев на интернет-ресурсах**. При построении структуры комментариев на сайте, где комментарий может содержать ответ на другой комментарий, рекурсия позволяет легко извлекать всю иерархию целиком.
- **Навигация по файловым системам**. Запросы, извлекающие дерево файлов и папок в файловых системах, также могут быть эффективно реализованы с помощью CTE.
- **Исторические данные и версии**. В хранилищах данных рекурсия может помочь проследить изменения в различных версиях записей, выявляя, как данные менялись во времени.
- **Производственные процессы и их связи**. Рекурсивные запросы могут использоваться для моделирования и анализа цепочек поставок, где каждая операция может включать подоперации.

Несмотря на многообразие этих применений, следует помнить, что рекурсия может быть ресурсоемкой операцией. При наличии больших объемов данных и глубокой иерархии неосторожное использование рекурсии может привести к увеличению времени выполнения запроса или даже к исчерпанию ресурсов — например, восстановление и отображение значительного количества уровней иерархии может занять много времени.

При использовании рекурсии существует риск **бесконечного цикла**, если в данных есть ошибки или неучтенные условия. Например, если в иерархии есть циклические ссылки (петли).

В таких случаях лучше использовать другие подходы. Например, если требуется извлечение всех подкатегорий для определенной категории товаров, можно воспользоваться классическим методом с использованием объединения таблиц (JOIN). Это эффективно, особенно при наличии индексов на местах объединения (индексы мы рассмотрим позднее, когда будем учиться оптимизировать запросы). Например, вместо использования рекурсии для поиска всех дочерних категорий можно создать итоговую таблицу с полным списком всех подкатегорий на момент запроса и просто фильтровать ее по нужному уровню.

При этом приложение для извлечения данных может работать более эффективно и иметь меньшую вероятность столкновения с проблемами производительности и бесконечных циклов.

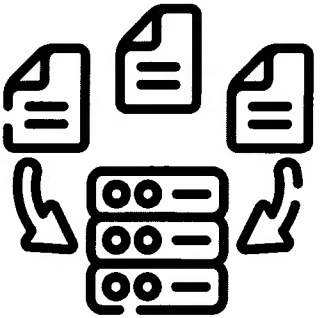
Что узнали и чему мы научились?

Подзапросы, агрегатные функции и CTE являются важными инструментами в арсенале разработчиков, позволяя организовывать данные и осуществлять сложные операции над ними:

- **Подзапросы** позволяют динамически извлекать необходимые сорта данных, что дает пользователям возможность более гибко подходить к построению запросов.

- **Агрегатные функции** обрабатывают наборы данных и предоставляют возможность быстрого анализа, например подсчета сумм или средних значений, обеспечивая потоки информации для принятия основных бизнес-решений.
- **CTE**, в свою очередь, не только упрощает написание и восприятие сложных запросов, но и позволяет выполнять рекурсивные операции. Они могут использоваться для более чистого и понятного кодирования при работе с иерархическими структурами. Однако важно помнить о потенциальных рисках производительности и понимать, когда и как эффективно использовать возможности рекурсии.

Помните, что SQL — декларативный язык, в котором вы пишете, что нужно получить, а не то, каким образом извлечь необходимые данные. Очень часто он позволяет прийти к одним и тем же результатам разными путями, поэтому фокусируйтесь на результате, а не на технике его получения.



Глава 8.

Оконные функции. Анализируем данные

8.1. Данные в контексте "окна". Пример мониторинга состояния пациента в больнице

В предыдущих главах мы научились извлекать и структурировать данные, объединять таблицы, группировать результаты и даже работать с иерархиями. Однако часто возникает необходимость анализировать данные не в статике, а в динамике — сравнивать текущие значения с предыдущими, вычислять скользящие средние, ранжировать записи или определять вклад каждой строки в общий результат. В таких случаях классические агрегатные функции и подзапросы могут оказаться неудобными или неэффективными, да и вовсе бесполезными.

Вот лишь некоторые из примеров подобных задач:

- Ранжирование результатов (создание любых рейтингов и все, что с ними связано: ABC-анализ и пр.).
- Сравнение текущих значений с предыдущими или следующими (используется для выявления аномалий и резких изменений в данных в любых периодических показателях).
- Расчет накопительных или скользящих итогов (любые задачи, связанные с потреблением ресурсов, в экономическом анализе, на производстве, в логистике, энергетике, медицине).
- Определение доли, удельного веса в общей сумме по категории (финансовый анализ, маркетинг, продажи и пр.).

- Анализ событий в контексте времени (сеансы, простои, циклы).

Все эти задачи объединяет одно — необходимость провести какой-то анализ не на общем массиве данных целиком, а на какой-то его части, в контекстном "окне", определенном по некоторым критериям.

Для погружения в контекст представим себе систему мониторинга состояния пациентов в больнице.

```
CREATE DATABASE hospital;
```

Задача системы — выявлять аномалии в показателях жизнедеятельности пациентов и вовремя оповещать медицинский персонал.

Для этих целей у пациентов отслеживается температура, пульс и давление пациента. Сущность, с которой мы будем работать, хранит ид пациента, время замера и три этих показателя:

```
CREATE TABLE measurements (
  measurement_id INT PRIMARY KEY,
  patient_id INT,              -- ИД пациента
  -- MSSQL:
  -- measure_time DATETIME,
  measure_time TIMESTAMP, -- время замера
  heart_rate INT,      -- пульс
  systolic INT,        -- систолическое давление
  diastolic INT,       -- диастолическое давление
  temperature DECIMAL(3,1)-- температура
);
```

Тестовый набор данных для замеров в данной таблице (доступен в электронном приложении):

```
INSERT INTO measurements VALUES
(1, 1, '2025-10-01 09:00', 75, 120, 80, 36.6),
(2, 1, '2025-10-01 15:00', 85, 130, 85, 37.1),
(3, 1, '2025-10-02 08:00', 90, 140, 90, 38.5),
(4, 2, '2025-10-02 16:00', 65, 110, 70, 36.8),
(5, 2, '2025-10-03 09:00', 72, 115, 75, 37.0),
(6, 1, '2025-10-03 12:00', 88, 135, 85, 37.3),
```

```
(7, 3, '2025-10-03 14:00', 102, 145, 95, 38.9),
(8, 3, '2025-10-04 09:00', 95, 140, 90, 37.8),
(9, 4, '2025-10-05 10:30', 70, 120, 80, 36.9),
(10, 4, '2025-10-05 16:00', 73, 125, 82, 37.2);
```

8.2. Что такое оконные функции

8.2.1. Синтаксис

Оконные функции в SQL представляют собой инструмент для анализа, позволяющий выполнять вычисления по наборам строк без объединения строк в группы, как это делает GROUP BY.

В SQL "окно" — это набор строк, определенный по определенным критериям. При использовании оконных функций в запросе мы должны сообщить СУБД, по каким критериям мы определяем окно и далее — что мы хотим посчитать в рамках данного окна.

Общее для всех оконных функций свойство — данные не сворачиваются, как в случае с GROUP BY, каждая строка сохраняет свои индивидуальные значения.

Синтаксис оконной функции в общем виде выглядит так:

```
Имя_функции(аргументы, если есть) OVER(описание "окна");
```

Можно прочитать как: *вычислить функцию [НАД] ("окном" данных).*

Синтаксис описания окна содержит ключевое слово `PARTITION BY` (разделить [данные] по…), за которым следует описание правил, по которым наш набор данных разделяется по окнам:

```
PARTITION BY столбцы
ORDER BY -- упорядочивает данные внутри окна
ROWS/RANGE - определяет границы фрейма
```

SKLADCHIK.ORG

Фрейм (рамка) — часть окна, которую функция учитывает в текущий момент.

Например, "последние 3 записи" или "все записи до текущей даты".

- **ROWS** (строки) — оперирует физическими строками (порядок определяется ORDER BY). Учитывает конкретное количество строк до/после текущей, например *текущая строка и две предыдущие*.
- **RANGE** (диапазон) — работает с логическими интервалами на основе значений столбца из ORDER BY. Учитывает строки, попадающие в диапазон значений текущей строки, например *все строки, где значение ≤ текущему*.

ВАЖНОЕ ПРИМЕЧАНИЕ:

в MS SQL Server спецификаторы RANGE поддерживают только ограничители типа UNBOUNDED PRECEDING, CURRENT ROW и аналогичные (без возможности указания фиксированного числа строк). Это связано с тем, что логически рамка в режиме RANGE подразумевает объединение строк с одинаковыми значениями, а не точное физическое расстояние. В то же время СУБД MySQL и PostgreSQL допускают задание конкретного числа строк для режима RANGE, хотя формально это противоречит стандарту ANSI SQL.

Для указания границ фрейма используется конструкция BETWEEN… AND… (между… и …):

```
BETWEEN начало_фрейма AND конец_фрейма;
```

Соответственно, определяя границы фрейма, мы можем указывать:

- **UNBOUNDED PRECEEDING** — без ограничений предыдущих строк (все строки, начиная с самого начала окна);

- **UNBOUNDED FOLLOWING** — без ограничений последующих строк (все строки до конца окна);
- **CURRENT ROW** — текущая строка;
- ***N* PRECEEDING** — N предшествующих строк;
- ***N* FOLLOWING** — N последующих строк.

Примеры определения границ фреймов:

- Строки [внутри окна] между 2 предшествующими [от текущей] и 1 последующей:

```
ROWS BETWEEN 2 PRECEDING AND 1 FOLLOWING
```

- Все строки, где значение из ORDER BY находится в диапазоне [текущее -50; текущее +50]:

```
RANGE BETWEEN 50 PRECEDING AND 50 FOLLOWING
```

Синтаксис позволяет указать только одну границу, если вторая автоматически принимается как текущая строка (CURRENT ROW). Например:

```
-- Текущая строка и две предыдущие:
ROWS 2 PRECEDING
-- Эквивалентно
ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
```

Таким образом, некоторые границы фреймов можно задать в упрощенном виде:

```
-- Все строки до текущей:
ROWS UNBOUNDED PRECEDING
-- Эквивалентная полная запись:
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

-- 3 предыдущие строки и текущая:
ROWS 3 PRECEDING
-- Эквивалентная полная запись:
ROWS BETWEEN 3 PRECEDING AND CURRENT ROW
```

Если вам необходимо указать как нижнюю, так и верхнюю границу фрейма, необходимо использовать полную форму записи.

8.2.2. Агрегатные функции: SUM, AVG, COUNT, MAX, MIN

Мы уже знакомы с агрегатными функциями. Их оконные эквиваленты работают схожим образом (SUM считает сумму, AVG — среднее значение, а COUNT — количество строк, MAX и MIN — максимальное и минимальное значение, соответственно), только значения рассчитываются не по группе, а по диапазону из указанной границы фрейма.

Задача: рассчитать скользящее среднее температуры для каждого пациента за каждые 2 последовательных измерения:

```
SELECT
  patient_id, measure_time, temperature,
  AVG(temperature) OVER (
    PARTITION BY patient_id
    ORDER BY measure_time
    ROWS BETWEEN 1 PRECEDING AND CURRENT ROW
  ) AS moving_avg
FROM measurements;
```

Мы указываем, что нам нужно среднее значение по температуре: AVG(temperature) по окнам, где данные разделяются по каждому пациенту (PARTITION BY patient_id). Внутри окна данные упорядочиваются (ORDER BY) по времени измерений.

Далее мы указываем, что фрейм (набор данных, по которым необходимо выполнить вычисление средней) определяется предшествующей строкой и текущей.

Запрос обрабатывается так:

1. Данные таблицы разбиваются по пациентам (PARTITION BY patient_id).
2. Внутри каждого окна производится сортировка данных по времени измерения (ORDER BY measure_time).

Сортировка внутри окна очень важна. Поскольку мы задаем границы фрейма, указывая предшествующие и последующие строки, их порядок должен быть неизменным!

3. Для пациента 1 в итоге получается 4 замера:

measure_time	temperature
2025-10-01 09:00:00.000	36.6
2025-10-01 15:00:00.000	37.1
2025-10-02 08:00:00.000	38.5
2025-10-03 12:00:00.000	37.3

4. Запрос начинает последовательно обрабатывать строки внутри полученного окна. Поскольку первое значение не имеет предшествующего, средняя вычисляется только по одному значению 36,6.

   Следующая строка с показателем 37,1 уже имеет 1 предшествующую (PRECEEDING 1), соответственно, функция AVG посчитает (36,6 + 37,1) / 2 = 36,85.

   Следующая строка с показателем 38,5 уже не будет учитывать первую, фрейм определен как 1 предшествующая строка и текущая. Соответственно, средняя температура будет (37,1 + 38,5) / 2 = 37,8.

   И так далее…

Полный результат запроса приведен ниже. Обратите внимание, что индивидуальные данные каждой строки не потеряны, в отличие от варианта с группировкой мы по-прежнему имеем доступ как к результату агрегатной функции, так и к текущему значению температуры:

patient_id	measure_time	temperature	moving_avg
1	2025-10-01 09:00:00.000	36.6	36.600000
1	2025-10-01 15:00:00.000	37.1	36.850000
1	2025-10-02 08:00:00.000	38.5	37.800000
1	2025-10-03 12:00:00.000	37.3	37.900000
2	2025-10-02 16:00:00.000	36.8	36.800000

patient_id	measure_time	temperature	moving_avg
2	2025-10-03 09:00:00.000	37.0	36.900000
3	2025-10-03 14:00:00.000	38.9	38.900000
3	2025-10-04 09:00:00.000	37.8	38.350000
4	2025-10-05 10:30:00.000	36.9	36.900000
4	2025-10-05 16:00:00.000	37.2	37.050000

С использованием средней можно наглядно рассмотреть разницу между спецификаторами RANGE и ROWS (пример работает только в PostgreSQL и MySQL). Функция ROUND используется для округления результата до 2 знаков после запятой:

```
-- Разница между ROWS и RANGE:
SELECT
    measure_time,
    heart_rate,
    ROUND(AVG(heart_rate) OVER (PARTITION BY patient_id ORDER BY
heart_rate ROWS BETWEEN 2 PRECEDING AND CURRENT ROW), 2) AS avg_rows,
    ROUND(AVG(heart_rate) OVER (PARTITION BY patient_id ORDER BY
heart_rate RANGE BETWEEN 2 PRECEDING AND CURRENT ROW), 2) AS avg_range
FROM
    measurements
WHERE
   patient_id = 1
```

Результат запроса:

measure_time	heart_rate	avg_rows	avg_range
2025-10-01 09:00:00	75	75.00	75.00
2025-10-01 15:00:00	85	80.00	85.00
2025-10-03 12:00:00	88	82.67	88.00
2025-10-02 08:00:00	90	87.67	89.00

Если используется спецификатор ROWS, то в каждой строке считается среднее арифметическое между текущим значением и 2 предыдущими:

1. (75 + 0 + 0) / 1 = 75 (1, потому что двух других значений нет).
2. (85 + 75 + 0) / 2 = 80 (есть только 2 значения).
3. (88 + 85 + 75) / 3 = 82,67.
4. (90 + 88 + 85) / 3 = 87,67.

ОБРАТИТЕ ВНИМАНИЕ:

несмотря на то, что для спецификатора **ROWS** мы указали границы рамки в 3 строки (2 предыдущие и текущая), в первых двух строках среднее арифметическое высчитывается делением не на 3, а на фактически найденное количество значений (1 и 2 соответственно).

Со спецификатором **RANGE** ситуация интереснее. Мы указываем СУБД, что надо посчитать среднее между значением текущей строки минус 2 и текущим значением. Здесь 2 — не количество строк, а разница в пульсе (ударов в минуту), то есть нам надо найти предыдущие значения в пределах от 73 до 75. Поскольку таких не находится, первый результат:

1. 75/1 = 75
   Во второй и третьей строке нам нужно найти среднее значение между всеми предыдущими значениями, попадающими в диапазон от 85 – 2 = 83 до 85 и от 88 – 2 = 86 до 88 соответственно. Таковых тоже не находится, поэтому второй и третий результат совпадают с самим показателем:
2. 85 / 1 = 85
3. 88 / 1 = 88
   Зато в четвертой строке, где надо найти значение в пределах от 90 – 2 = 88 до 90, таких значений находится 2: 88 и 90, соответственно, в результат запроса выводится среднее значение.
4. (88 + 90) / 2 = 89

8.2.3. Ранжирующие функции: ROW_NUMBER, RANK, DENSE_RANK

Ранжирующие функции позволяют присваивать порядковые номера строкам в результате запроса на основе заданной сортировки:

- Выявления топ-N записей (например, самые высокие показатели).
- Определения позиции элемента в рейтинге.
- Анализа распределения данных (процентили, ранги).

В SQL предусмотрены три основные функции для работы с рангами: ROW_NUMBER, RANK и DENSE_RANK. Лучше всего проиллюстрировать разницу между ними на примере:

```
-- Разница между RANK, DENSE_RANK, ROW_NUMBER
SELECT
  measure_time,
  diastolic,
  RANK() OVER (
    PARTITION BY patient_id
    ORDER BY diastolic
  ) AS "Rank",
  DENSE_RANK() OVER (
    PARTITION BY patient_id
    ORDER BY diastolic
  ) AS "Dense Rank",
  ROW_NUMBER() OVER (
    PARTITION BY patient_id
    ORDER BY diastolic
  ) AS "Row Number"
FROM measurements
WHERE patient_id = 1;
```

По окну первого пациента мы сортируем данные по показателю диастолического давления и выводим значения всех трех ранжирующих функций для каждого показателя:

measure_time	diastolic	Rank	Dense Rank	Row Number
2025-10-01 09:00:00.000	80	1	1	1
2025-10-01 15:00:00.000	85	2	2	2
2025-10-03 12:00:00.000	85	2	2	3
2025-10-02 08:00:00.000	90	4	3	4

Все три функции, как вы видите, ранжируют показатели внутри окна, но каждая делает это по-разному.

- Функция **RANK** (ранг) помещает совпадающие показатели на одно и то же "место" в рейтинге (в двух разных измерениях давление совпадает, поэтому им обоим присвоен один и тот же ранг 2), зато показателю 90, поскольку он четвертый по списку, присвоен ранг 4.
- Функция **DENSE_RANK** (уплотненный ранг) делает все то же самое, что и RANK, только присваивает ранг не по месту в изначальном списке, а по порядку. Таким образом, если RANK пропустила 3 и сразу присвоила последнему показателю ранг 4, DENSE_RANK использовала следующий порядковый номер 3 и присвоила его последнему значению в окне.
- Функция **ROW_NUMBER** (номер строки) просто нумерует отсортированный результат в окне, присвоив каждому измерению порядковый номер. В этом варианте два совпадающих значения 85 будут иметь разные номера по порядку.

Когда какую функцию использовать?

- **RANK** — для ранжирования с пропусками (например, если в спортивных соревнованиях имеются два серебряных призера, бронзу получит участник, пришедший к финишу четвертым).
- **DENSE_RANK** — для непрерывной нумерации (например, распределение абитуриентов по баллам: два первых места, следующий студент займет второе место).
- **ROW_NUMBER** — для уникальной идентификации строк (например, надо пронумеровать измерения по порядку).

Задача для самостоятельной работы (варианты ответов доступны в электронном приложении): выявить моменты, когда температура пациента впервые превысила 38°C. Подсказка: используйте CTE с оконной функцией внутри.

8.2.4. Функции распределения / относительного ранга: NTILE, PERCENT_RANK, CUME_DIST

Функции распределения (относительного ранга) помогают анализировать данные в контексте всего набора, а не отдельных строк. Иногда нам важен не столько ранг, сколько разбиение результатов на группы.

Функции из этой группы позволяют отвечать на вопросы:

- В каком процентиле находится значение (PERCENT_RANK)?
- Какая доля строк имеет значения ≤ текущему (CUME_DIST)?
- Как разбить данные на равные группы (NTILE)?

Эти функции полезны для статистического анализа, сегментации данных и выявления аномалий.

PERCENT_RANK

Рассчитывает *процентиль* текущего значения внутри окна.

Является эквивалентом вычисления RANK() — 1 / кол-во строк в окне. Результат функции всегда находится в диапазоне от 0 до 1.

```
SELECT
  patient_id,
  measure_time,
  systolic,
  PERCENT_RANK() OVER (
    PARTITION BY patient_id
    ORDER BY systolic
  ) AS percentile_rank
FROM measurements
WHERE patient_id = 1;
```

measure_time	systolic	percentile_rank
2025-10-01 09:00:00.000	120	0
2025-10-01 15:00:00.000	130	0,333333333333333
2025-10-03 12:00:00.000	135	0,666666666666667
2025-10-02 08:00:00.000	140	1

Для минимального значения (120) процентиль = 0.

Для максимального (140) процентиль = 1.

Значение 130 находится на 33.3% от минимума к максимуму.

Данная функция сильно облегчает работу для нахождения значений, входящих в какой-то процент от выборки (например, топ 5%).

CUME_DIST

Часто бывает полезно разделить набор данных на равные по численности группы, основываясь на значении определенного столбца. Например, если нужно сегментировать покупателей по сумме покупок таким образом, чтобы каждая группа имела примерно одинаковое количество клиентов.

Функция **CUME_DIST** позволяет автоматически рассчитать границы этих сегментов (показывает долю строк, значения которых меньше или равны текущему).

Рассчитывает **накопительное** (кумулятивное) **распределение** (cumulative distribution) значения внутри других значений окна по формуле: количество строк меньше или равных значению текущей строки / общее количество строк в окне.

```
SELECT
  measure_time,
  diastolic,
  CUME_DIST() OVER (
    PARTITION BY patient_id
```

SKLADCHIK.ORG


```
    ORDER BY diastolic
  ) AS cumulative_dist
FROM measurements
WHERE patient_id = 1;
```

Результат:

measure_time	diastolic	cumulative_dist
2025-10-01 09:00:00.000	80	0,25
2025-10-01 15:00:00.000	85	0,75
2025-10-03 12:00:00.000	85	0,75
2025-10-02 08:00:00.000	90	1

Объяснение:

- 25% значений ≤ 80 (только первая строка).
- 75% значений ≤ 85 (три строки из четырех).

В плане практического применения функция CUME_DIST поможет с легкостью выяснить, например, долю посетителей веб-сайта, совершивших целевое действие; с ее помощью можно определить долю работников или учащихся, достигших высоких показателей, или наоборот, выделить группу нуждающихся в дополнительном обучении или мотивации. Данная функция пригодится в любой задаче, где требуется выделить из данных некую долю по одному или совокупности нескольких показателей.

NTILE(n)

При помощи **NTILE(n)** можно легко разбить данные на n примерно равных групп.

Например, надо разделить все замеры пульса на три примерно равные группы:

```
SELECT
  patient_id,
```

```
  measure_time,
  heart_rate,
  NTILE(3) OVER (
    ORDER BY heart_rate
  ) AS ntile_group
FROM measurements;
```

Результат:

patient_id	measure_time	heart_rate	ntile_group
2	2025-10-02 16:00:00.000	65	1
4	2025-10-05 10:30:00.000	70	1
2	2025-10-03 09:00:00.000	72	1
4	2025-10-05 16:00:00.000	73	1
1	2025-10-01 09:00:00.000	75	2
1	2025-10-01 15:00:00.000	85	2
1	2025-10-03 12:00:00.000	88	2
1	2025-10-02 08:00:00.000	90	3
3	2025-10-04 09:00:00.000	95	3
3	2025-10-03 14:00:00.000	102	3

Каждый из замеров отнесен, соответственно, к группе 1, 2 или 3. Размеры каждой группы примерно равны (4, 3 и 3 строки соответственно).

8.2.5. Аналитические функции: LAG, LEAD, FIRST_VALUE, LAST_VALUE

Аналитические функции (еще их называют ***функции смещения***) позволяют анализировать данные в контексте временных или логических последовательностей. Эти функции незаменимы для сравнений текущих значений с предыдущими или следующими; выявления трендов (например, рост/падение показателей) или определения базовых и финальных значений в динамике.

LAG — доступ к предыдущим данным

Возвращает значение из строки, находящейся на указанное количество шагов назад относительно текущей.

Синтаксис:

```
LAG(столбцы, смещение, значение по умолчанию)
```

Смещение можно не указывать, по умолчанию будет принято значение 1 (предыдущее значение — отстоящее на 1 строку назад).

Значение по умолчанию тоже можно не указывать, по умолчанию будет NULL. Полезно бывает заменить на 0, вместо того чтобы потом пользоваться COALESCE.

ПРИМЕР: определить, на сколько изменился пульс пациента с момента предыдущего замера.

В запросе, помимо пациента и времени замера, выводится текущее показание пульса, предыдущее показание и разница между ними:

```
SELECT
  patient_id,
  measure_time,
  heart_rate,
  LAG(heart_rate, 1, 0) OVER (
    PARTITION BY patient_id
    ORDER BY measure_time
  ) AS prev_heart_rate,
  heart_rate - LAG(heart_rate, 1, 0) OVER (
    PARTITION BY patient_id
    ORDER BY measure_time
  ) AS hr_change
FROM measurements;
```

Для краткости приведен результат только по пациенту 1:

patient_id	measure_time	heart_rate	prev_heart_rate	hr_change
1	2025-10-01 09:00:00.000	75	0	75
1	2025-10-01 15:00:00.000	85	75	10
1	2025-10-02 08:00:00.000	90	85	5
1	2025-10-03 12:00:00.000	88	90	-2

ОБРАТИТЕ ВНИМАНИЕ:

в запросе можно было бы просто написать LAG(heart_rate), но тогда в результат вывелись бы значения NULL там, где предыдущее значение отсутствует, что не всегда удобно.

LEAD — доступ к последующим данным

Дополняет функцию LAG аналогичным поведением в отношении последующих N строк. Может применяться для выявления тенденций роста или спада, анализа поступления данных в будущих периодах, контроля последовательности событий.

ПРИМЕР: выяснить, через какое время после текущего замера произошел следующий.

```
SELECT
  patient_id,
  measure_time,
  LEAD(measure_time) OVER (
    PARTITION BY patient_id
    ORDER BY measure_time
  ) AS next_measure_time,
  LEAD(measure_time) OVER (
    PARTITION BY patient_id
    ORDER BY measure_time
  ) - measure_time AS time_interval
FROM measurements;
```

Обратите внимание, поскольку мы не использовали необязательные аргументы для указания значения по умолчанию, как в примере с LAG, данные по последнему замеру для каждого пациента имеют значения NULL (для краткости приведены только данные по первому пациенту).

patient_id	measure_time	next_measure_time	time_interval
1	2025-10-01 09:00:00.000	2025-10-01 15:00:00.000	1900-01-01 06:00:00.000
1	2025-10-01 15:00:00.000	2025-10-02 08:00:00.000	1900-01-01 17:00:00.000
1	2025-10-02 08:00:00.000	2025-10-03 12:00:00.000	1900-01-02 04:00:00.000
1	2025-10-03 12:00:00.000	NULL	NULL

FIRST_VALUE — первое значение окна

Возвращает значение из первой строки окна. Очень полезно для оценки текущей динамики, например, в финансах, где необходимо проанализировать доходность актива относительно его первоначальной стоимости. В нашем примере с больницей можно анализировать текущие показатели по сравнению с показателями пациента при поступлении:

```
WITH MeasurementsWithInitial AS (
  SELECT
    measure_time,
    systolic,
    FIRST_VALUE(systolic) OVER (
      PARTITION BY patient_id
      ORDER BY measure_time
    ) AS initial_systolic
  FROM measurements
  WHERE patient_id = 1
)
SELECT
  measure_time,
  systolic,
  initial_systolic,
  systolic - initial_systolic AS systolic_delta
FROM MeasurementsWithInitial;
```

Поскольку нам необходимо дважды использовать в запросе значение для первоначального показателя *initiali_systolic*, его вычисление вынесено в CTE MeasurementsWithInitial.

LAST_VALUE — последнее значение окна

Дополняет FIRST_VALUE и возвращает значение из последней строки окна. Может оказаться полезным при анализе телеметрии при анализе последних значений за сеанс связи. В нашем примере поможет найти последние зафиксированные измерения до настоящего момента:

```
SELECT
  patient_id,
  LAST_VALUE(temperature) OVER (
    PARTITION BY patient_id
    ORDER BY measure_time
  ) AS last_temperature
FROM measurements;
```

8.3. Вопросы производительности и антипаттерны

8.3.1. Вложенные оконные функции

Вы можете комбинировать оконные функции, создавая многоуровневые вычисления. Это полезно, когда требуется провести анализ на основе результатов другого оконного выражения. Однако такие конструкции требуют осторожности из-за потенциального влияния на производительность. Старайтесь избегать подобных запросов там, где важна производительность.

ПРИМЕР: определение стабильности давления. Для каждого пациента найти замеры, где систолическое давление выше среднего по всем его замерам, затем ранжировать результаты:

```
WITH patient_stats AS (
  SELECT
    patient_id,
    systolic,
    AVG(systolic) OVER (PARTITION BY patient_id) AS avg_systolic,
    RANK() OVER (PARTITION BY patient_id ORDER BY systolic DESC)
    AS pressure_rank
  FROM measurements
)
SELECT
  patient_id,
  systolic,
  avg_systolic,
  pressure_rank
FROM patient_stats
WHERE systolic > avg_systolic;
```

Сначала внутри обобщенного табличного выражения (CTE) patient_stats вычисляется среднее давление по пациенту при помощи функции **AVG**.

Затем внутри того же запроса используется RANK для ранжирования замеров. Внешний запрос фильтрует строки, где давление выше среднего.

Результат:

patient_id	systolic	avg_systolic	pressure_rank
1	140	131	1
1	135	131	2
2	115	112	1
3	145	142	1
4	125	122	1

8.3.2. ROWS вместо RANGE

Старайтесь по возможности использовать ROWS. Производительность оконных функций со спецификаторами ROWS и RANGE заметно различается по причине внутренней логики их работы:

- **ROWS** осуществляет выборку строк, основываясь на их физическом расположении в наборе данных. Это прямой перебор строк сверху вниз, начиная с заданной точки (например, текущей строки и двух предыдущих). Это не требует сложного анализа или дополнительной сортировки.
- **RANGE** использует логику диапазонов значений. Например, при вычислении средней температуры за период функция учитывает не физические строки, а все строки с температурой, соответствующей заданному диапазону. Это сложнее, так как требует постоянной проверки диапазона значений и потенциально большего объема данных для анализа.

8.3.3. Фильтрация до оконных функций

Если возможно, используйте WHERE до применения оконной функции, чтобы уменьшить объем обрабатываемых данных.

```
SELECT *
FROM (
  SELECT
    patient_id,
    measure_time,
    heart_rate,
    AVG(heart_rate) OVER (...) AS avg_hr
  FROM measurements
  WHERE patient_id = 1  -- Фильтрация на раннем этапе
) AS filtered_data;
```

8.3.4. Индексы по столбцам PARTITION BY и ORDER BY

Детально вопросами производительности и индексами мы будем заниматься в следующих главах, однако здесь, применительно к работе с оконными функциями, стоит отметить, что наличие индексов по столбцам, которые используются для разбиения данных на окна (PARTITION BY), а также для внутренней сортировки данных внутри окон (ORDER BY), весьма существенно ускорит ваши запросы с использованием оконных функций.

8.3.5. Антипаттерн: избыточное секционирование

Используйте только необходимые столбцы для определения окон. Слишком мелкие столбцы увеличивают нагрузку:

```
RANK() OVER (PARTITION BY patient_id, DATE(measure_time) ORDER BY ...)
```

8.3.6. Антипаттерн: оконные функции вместо простых агрегатов

Используйте оконные функции только в случае, если вам действительно нужны детализированные данные. Если имеется возможность выбора, всегда используйте простые агрегаты, так как они работают быстрее.

Для классического примера из анекдота про среднюю температуру данный показатель можно получить двумя способами.

Сложным:

```
SELECT
    patient_id,
    AVG(temperature) OVER (PARTITION BY patient_id) AS avg_
temperature
FROM measurements;
```

или простым, более быстрым:

```
SELECT
    patient_id,
    AVG(temperature) AS avg_temperature
FROM measurements
GROUP BY patient_id;
```

Что узнали и чему мы научились?

Мы изучили оконные функции как средство, позволяющее проводить анализ данных на основании "соседей", и рассмотрели, чем такие функции отличаются от обычных агрегатных функций с группировками GROUP BY.

Мы научились синтаксису, при помощи которого мы можем определять окна и границы рамок для данных внутри этих окон при помощи ключевых слов OVER, PARTITION BY, ORDER BY, ROWS/RANGE.

Мы охватили разные виды оконных функций, включая агрегатные, ранжирующие, функции смещения и распределения, а также рассмотрели некоторые вопросы производительности запросов, содержащих оконные функции.

SKLADCHIK.ORG

Глава 9.

Изменение и удаление данных

9.1. Манипуляция данными на примере таксопарка

Базы данных — это не просто хранилища информации, а динамичные системы, требующие активного взаимодействия. Помимо извлечения данных (SELECT), не менее важны операции для их изменения: добавление, обновление, удаление.

Эти действия формируют основу **Data Manipulation Language** (**DML** — язык для манипуляции данных) — части SQL, отвечающей за работу с содержимым таблиц.

DML позволяет не только влиять на отдельные записи, но и выполнять массовые операции, разрешать конфликты целостности, возвращать результаты модификаций и даже объединять действия в единые транзакции.

К сожалению, синтаксические особенности DML в части добавления, обновления и удаления данных не настолько хорошо стандартизированы, как это сделано для выборки данных: в каждой из рассматриваемых нами СУБД имеются свои нюансы, которые могут потребовать дополнительного изучения документации.

В этой главе вы узнаете, как эффективно управлять данными, используя возможности INSERT, UPDATE, DELETE, а также комбинированных техник вроде UPSERT. Мы разберем как стандартные подходы SQL, так и специфичные для популярных СУБД решения. Если вас не интересуют подробности для СУБД, с которой вы не планируете работать, можете пропустить соответствующие подразделы.

Чтобы нам было с чем работать в этой главе, рассмотрим работу таксопарка в качестве примера и создадим для него базу данных.

```
CREATE DATABASE taxi;
```

Добавим в нее две таблицы с мастер-данными: **водители** (master_drivers) и **автомобили** (cars). Листинги доступны в электронном приложении. Поскольку мы будем учиться изменять данные в ходе изучения материала к этой главе, мастер-данные мы трогать не будем, а будем работать с их копиями.

```
CREATE TABLE master_cars (
  -- Раскомментируйте для MS SQL Server
  -- car_id INT PRIMARY KEY IDENTITY(1,1),
  -- Для MySQL:
  -- car_id INT PRIMARY KEY AUTO_INCREMENT,
  -- Для PostgreSQL:
  -- car_id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  license_plate VARCHAR(15) UNIQUE, -- рег. номер
  manufacturer VARCHAR(30),   -- производитель
  model VARCHAR(30),    -- модель
  manufacture_year INT,     -- год производства
  mileage INT,        -- пробег
  last_service DATE,    -- дата ТО
  status VARCHAR(20)
        -- статус (активна/на ТО/в ремонте)
        CHECK
  (status IN ('active', 'maintenance', 'in repairs'))
  DEFAULT('active')
);


CREATE TABLE master_drivers (
  driver_id INT PRIMARY KEY,  -- ид водителя
  full_name VARCHAR(50) NOT NULL   -- имя
);

-- Автомобили
```

```
INSERT INTO master_cars (
  license_plate,
  manufacturer,
  model,
  manufacture_year,
  mileage,
  last_service,
  status)
VALUES
('ABC123', 'Toyota', 'Camry', 2020, 184200, '2024-02-14', 'maintenance'),
('XYZ789', 'Honda', 'Accord', 2023, 67800, '2025-01-20', 'active'),
('DEF456', 'Ford', 'Transit', 2022, 96300, '2024-11-07', 'active'),
('GHI789', 'Chevrolet', 'Malibu', 2018, 324200, '2024-12-24', 'in repairs'),
('JKL012', 'Hyundai', 'Sonata', 2023, 56400, '2024-04-12', 'active'),
('MNO345', 'Kia', 'Optima', 2021, 77500, '2024-06-14', 'active'),
('PQR678', 'Volkswagen', 'Passat', 2020, 102400, '2023-12-10', 'active'),
('STU901', 'Nissan', 'Altima', 2021, 83800, '2024-10-05', 'in repairs');

-- Водители
 INSERT INTO master_drivers (driver_id, full_name) VALUES
(101, 'Деточкин Юрий'),
(102, 'Шлыков Иван'),
(103, 'Морозов Александр'),
(104, 'Хачикян Рубик'),
(105, 'Громов Виктор'),
(106, 'Шумилов Григорий'),
(107, 'Низовцев Андрей');
```

9.2. Еще немного о вставке данных

9.2.1. Вставка данных из подзапроса

Ранее мы уже рассматривали инструкцию INSERT, которая была нам необходима, чтобы добавлять данные в таблицы для примеров. С помощью нее мы научились добавлять данные в таблицы напрямую, но INSERT может быть мощнее, чем кажется. Помимо явного перечисления добавляемых зна-

чений, она позволяет загружать данные напрямую из результатов запроса, включая фильтрацию, агрегацию и соединение таблиц.

Главная сила INSERT в таком сценарии — гибкость. Команда может использовать результат любого подзапроса, если он соответствует структуре целевой таблицы: количество и типы столбцов должны совпадать. Однако есть исключения:

- Автоматически генерируемые значения (например, поля с автонумерацией) не требуют указания в подзапросе — СУБД заполнит их сама.
- Столбцы со значением по умолчанию или без ограничения NOT NULL тоже можно пропустить.

Таким образом, подзапрос дает возможность не только копировать данные "как есть", но и трансформировать их на лету: фильтровать ненужные строки, объединять таблицы или даже агрегировать информацию перед вставкой.

Создайте таблицу **cars**, идентичную по структуре мастер-данным:

```
CREATE TABLE cars (
  -- Раскомментируйте для MS SQL Server
  -- car_id INT PRIMARY KEY IDENTITY(1,1),
  -- Для MySQL:
  -- car_id INT PRIMARY KEY AUTO_INCREMENT,
  -- Для PostgreSQL:
  -- car_id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  license_plate VARCHAR(15) UNIQUE, -- рег. номер
  manufacturer VARCHAR(30),   -- производитель
  model VARCHAR(30),     -- модель
  manufacture_year INT,      -- год производства
  mileage INT,         -- пробег
  last_service DATE,     -- дата ТО
  status VARCHAR(20)
    -- статус (активна/на ТО/в ремонте)
    CHECK
    (status IN ('active', 'maintenance', 'in repairs'))
  DEFAULT('active')
);
```

Теперь давайте попробуем скопировать в нее только автомобили, которые имеют статус `'active'`.

Начало в запросе INSERT такое же, как если бы мы делали обычную вставку. Мы указываем только те столбцы, которые будем копировать. Столбец *car_id* заполняется автоматически, то же касается статуса — это поле имеет значение по умолчанию *active*, то есть, даже если мы не укажем статус, поскольку хотим скопировать только активные автомобили, соответствующий статус проставится автоматически:

```
INSERT INTO cars
(manufacturer, model, license_plate, mileage, last_service)
SELECT manufacturer, model, license_plate, mileage, last_service
FROM master_cars
WHERE status = 'active';
```

Мы видим подзапрос SELECT, возвращающий требуемую структуру данных для команды INSERT. Если мы добавляем данные из результатов запроса, мы не указываем ключевое слово VALUES, а просто начинаем писать запрос SELECT.

Обратите внимание: в примере намеренно изменен порядок столбцов, чтобы проиллюстрировать тот факт, что соблюдать его в том же виде, в каком он был указан при создании таблицы, не обязательно — мы должны соблюдать порядок указания столбцов между первой (INSERT INTO) и второй (SELECT) частью запроса.

Столбец с годом выпуска автомобиля пропущен специально, его мы заполним чуть позднее. Кроме того, мы пропустили столбец *car_id* и статус, так как СУБД их заполнит автоматически.

Чтобы удостовериться, что копирование произведено, запросите все данные из таблицы **cars**:

```
SELECT * FROM cars;
```

Вы можете убедиться, что статусы для автомобилей заполнены значениями по умолчанию (active), как указано в ограничении DEFAULT.

В качестве запроса для вставки можно использовать любой запрос SELECT, возвращающий данные требуемой последовательности столбцов и типов данных.

9.2.2. Возврат данных при добавлении

Очень часто при добавлении данных в таблицы, которые генерируют идентификаторы автоматически, может возникнуть ситуация, когда нам требуется получить этот идентификатор сразу в момент проведения запроса.

Каждая СУБД решает эту задачу по-своему.

Microsoft SQL Server (OUTPUT)

В данной СУБД, чтобы возвратить значение для столбца из только что добавленной записи, используется ключевое слово OUTPUT (вывод), вслед за которым идет ссылка на вставленный столбец INSERTED, после которого указывается имя столбца, значение которого необходимо получить (обычно используется имя первичного ключа).

Добавим новый автомобиль:

```
INSERT INTO cars (license_plate, manufacturer, model)
OUTPUT INSERTED.car_id
VALUES ('ABC111', 'Ford', 'Mustang');
```

После выполнения данного запроса вам будет выведен идентификатор добавленной записи.

ПРИМЕЧАНИЕ.

В сети вы также можете столкнуться с советами для MS SQL Server использовать для этих нужд функции SCOPE_IDENTITY() или @@IDENTITY, однако их использование сопряжено с определенными рисками и не гарантируют требуемого результата.

SCOPE_IDENTITY() возвращает значение последнего идентификатора, созданное только в текущей области видимости (например, в рамках хранимой процедуры или текущего пакета запросов). Это защищает от побочных эффектов, если в таблице есть триггеры, которые сами выполняют вставки в другие таблицы с автонумеруемыми столбцами.

Это менее гибкий подход, так как требует выполнения двух запросов подряд:

```
INSERT INTO … ;
SELECT SCOPE_IDENTITY();
```

Функция @@IDENTITY:

```
SELECT @@IDENTITY;
```

Возвращает последний сгенерированный идентификатор в любой области видимости, включая те, что сгенерированы триггерами (триггеры будут рассматриваться позднее), так что результат может быть непредсказуемым, и вам вернется ID из другой таблицы.

Эти встроенные функции имеют очень узкие границы применения, и, если вы не уверены, используйте конструкцию OUTPUT INSERTED.

PostgreSQL (RETURNING)

В PostgreSQL существует аналогичный функционал, позволяющий получить идентификатор добавленного столбца сразу в момент вставки. Для этого используется ключевое слово RETURNING, которое указывается в конце основного запроса.

Вы можете указать любое имя столбца из добавляемых данных:

```
INSERT INTO cars (license_plate, manufacturer, model)
VALUES
('ABC112', 'Lamborgini', 'Diablo')
RETURNING car_id
```

Как и с аналогом SCOPE_IDENTITY(), в PostgreSQL имеется аналогичная функция LASTVAL(). Однако LASTVAL() зависит от текущей последовательности в сессии, что может привести к ошибкам, если между вставкой и вызовом функции выполнялись другие операции с последовательностями. Использовать данную функцию не рекомендуется.

MySQL

В MySQL, к сожалению, на момент написания этого материала, отсутствует возможность получения идентификатора сразу в результате запроса вставки. Для этих целей потребуется выполнить дополнительный запрос с вызовом специально предназначенной для этого функции LAST_INSERT_ID().

MySQL для получения последнего сгенерированного идентификатора используется функция LAST_INSERT_ID(). Используйте вызов этой функции сразу после добавления:

```
INSERT INTO cars (license_plate, manufacturer, model)
VALUES ('ABC111', 'Ford', 'Mustang');
SELECT LAST_INSERT_ID();
```

Функция возвращает значение, сгенерированное для текущего сеанса, что гарантирует корректность даже при параллельных запросах от других клиентов. Важно отметить, что LAST_INSERT_ID() работает только в рамках одного соединения с БД. Если нужно получить несколько идентификаторов или другие данные из добавленной строки, придется выполнять дополнительные запросы.

9.3. Изменение данных

В прошлом разделе при копировании данных для таблицы cars мы забыли указать год выпуска автомобиля в запросе, и поэтому сейчас в качестве года выпуска везде указано значение NULL. Давайте добавим недостающие данные в таблицу. Мы можем это сделать несколькими способами.

9.3.1. Простое изменение

Мы можем просто указать, значение из какой таблицы, какого столбца в какой строке мы хотим изменить. Для изменения данных в SQL используется ключевое слово UPDATE (обновить).

Общий синтаксис команды на обновление данных в таблице выглядит так:

```
UPDATE <имя_таблицы>
SET
<столбец1> = <значение1>,
<столбец2> = <значение2>,
…
WHERE <условие>
```

В нашем случае мы хотим обновить (*update*) таблицу **cars** и установить значение *manufacture_year* равным 2023 для Honda Accord с регистрационным знаком XYZ789.

Запишем запрос:

```
UPDATE cars SET
manufacture_year = 2023
WHERE license_plate = 'XYZ789';
```

Мы просим СУБД установить значение 2023 в любой строке таблицы, где значение *license_plate* равно 'XYZ789'. Результатом этого запроса должно стать число 1, а именно количество строк, которые были затронуты нашим запросом на изменение.

Подобный способ обновления данных используется, когда необходимо точечно исправить или добавить недостающие данные (например, при изменении или исправлении конкретных анкетных данных в конкретной записи).

Мы можем также модифицировать и несколько записей одновременно, если это необходимо.

Например, все автомобили марки Ford встают на техническое обслуживание с 1 июня 2025 года, и мы хотим отразить это в данных:

SKLADCHIK.ORG


```
UPDATE cars
SET
   last_service = '2025-06-01',
   status = 'maintenance'
WHERE manufacturer = 'Ford';
```

Как видите, мы можем использовать произвольные условия и несколько столбцов для модификации.

9.3.2. О важности WHERE

При модификации данных и при их удалении SQL не предусматривает вопросов и подтверждений наших намерений. В случае если мы отправляем запрос, он исполняется так, как мы написали. За исключением случаев, когда запрос используется как часть транзакции, которые мы будем рассматривать позднее, единственной возможностью исправить результат изменяющего запроса будет являться восстановление данных из резервной копии или из копии мастер-данных, если они, конечно, имеются.

В предыдущем примере мы исправляли год выпуска автомобиля для известного нам номерного знака. Зная, что номерной знак уникален (и у нас для этих целей имеется ограничение UNIQUE), мы уверены, что в таблице не более одной записи, удовлетворяющих заданному условию license_plate = 'XYZ789', соответственно, изменена будет только 1 запись (или ни одной, если автомобиля с таким номерным знаком не окажется в таблице).

Теперь представьте, мы исполним предыдущий запрос в следующем виде, забыв указать WHERE:

```
UPDATE cars SET
manufacture_year = 2023;
```

В его результате изменятся все записи в таблице, даже те, где значение года выпуска было указано верно. Это настолько опасная ситуация, что некоторые СУБД даже ограничивают по умолчанию применение подобных запросов без условия WHERE, однако не стоит полагаться в этом на СУБД.

При использовании изменяющих запросов следуйте этим советам:

- ***Всегда делайте резервную копию данных.***
- ***Начинайте написание запроса с ключевого слова WHERE.***
- ***Сначала сформулируйте условие фильтрации, затем пишите основной запрос.***
- ***Проверьте условие фильтрации, выполнив SELECT с вашими условиями в WHERE.***
- ***Используйте транзакции, если это возможно.***

9.3.3. Обновление данных из подзапроса/CTE

Новые значения, которые требуется записать в WHERE, могут быть получены любым способом, включая подзапросы или табличные выражения.

Добавим для иллюстрации таблицу водителей и скопируем в нее данные:

```
CREATE TABLE drivers (
  driver_id INT PRIMARY KEY,
  full_name VARCHAR(50) NOT NULL,
  assigned_car INT,
  CONSTRAINT fk_drivers_cars
    FOREIGN KEY (assigned_car)
    REFERENCES cars(car_id)
);
```

В отличие от мастер-данных, мы добавили ограничение **fk_drivers_cars** по внешнему ключу на закрепленную за водителем машину (имя данного ограничения нам понадобится позднее).

Скопируем мастер-данные:

```
INSERT INTO drivers(driver_id, full_name)
SELECT * FROM master_drivers; -- Плохая практика!
```

Если мы уверены, что в результате запроса SELECT будут возвращены нужные нам столбцы в нужном порядке, можно использовать SELECT *, однако это является плохой практикой. В будущем таблицы и столбцы в них могут измениться, поэтому лучше явно указывать, что мы хотим скопировать:

```
INSERT INTO drivers(driver_id, full_name)
SELECT driver_id, full_name FROM master_drivers;
```

Теперь давайте назначим водителям автомобили.

> Юрий Деточкин (driver_id 101) получает автомобиль Hyundai с номером JKL012, а Рубик Хачикян (driver_id 104) получает Kia с номером MNO345.

Наша проблема заключается в том, что мы не знаем, какие идентификаторы для автомобилей присвоила наша СУБД, потому что мы использовали столбец с автонумерацией. На помощь придут подзапросы.

```
WITH car_ids AS
(
  SELECT license_plate, car_id
  FROM cars
)
UPDATE drivers
SET assigned_car =
  CASE driver_id
  WHEN 101 THEN (SELECT car_id FROM car_ids WHERE license_plate =
'JKL012')
  WHEN 104 THEN (SELECT car_id FROM car_ids WHERE license_plate =
'MNO345')
  END;
```

Мы можем назначить автомобили обоим водителям одним запросом при помощи CASE… WHEN (*в случае если*), где для каждого случая мы выполним подзапрос к табличному выражению *car_ids*, содержащему номера автомобилей и их идентификаторы, и найдем нужный идентификатор по номеру.

Без конструкции CASE… WHEN нам пришлось бы делать два запроса на обновление для каждого из водителей.

ПРИМЕЧАНИЕ.

CTE с номерами автомобилей в данном случае несколько избыточно, так как того же результата можно было бы добиться, напрямую обратившись к таблице **cars**. Такой подход может быть все же оправдан, если бы нам приходилось выбирать не из всех автомобилей, а, например, только из автомобилей Kia с пробегом менее 100 тыс. км.

9.3.4. Обновление данных через присоединение JOIN

Мы оставили таблицу **cars** без заполненных сведений о годе выпуска автомобиля, однако эти сведения присутствуют в мастер-данных (таблице **master_cars**). Мы можем обновить сразу все значения *manufacture_year*, если будем использовать присоединение JOIN.

Microsoft SQL Server

Для MS SQL Server синтаксис для обновления таблицы через присоединение выглядит следующим образом:

```
UPDATE c
SET c.manufacture_year = mc.manufacture_year
FROM cars c
INNER JOIN master_cars mc
  ON c.license_plate = mc.license_plate
WHERE c.manufacture_year IS NULL;
```

Мы сообщаем СУБД, что необходимо выполнить обновление таблицы cars по данным таблицы **cars**, к которой присоединена таблица **master_cars** по совпадению номерного знака (помните, что идентификаторы *car_id* в

этих двух таблицах могут быть разными) там, где номерной знак имеет значение NULL.

Принцип присоединения таблиц здесь точно такой же, как в случае присоединения с использованием оператора SELECT.

MySQL

В MySQL используется следующий синтаксис для решения данной задачи:

```
UPDATE cars
INNER JOIN master_cars
  ON cars.license_plate = master_cars.license_plate
SET cars.manufacture_year = master_cars.manufacture_year
WHERE cars.manufacture_year IS NULL;
```

За исключением порядка следования операторов и отсутствия FROM CARS, синтаксис очень похож на тот, что используется для MS SQL.

ПРИМЕЧАНИЕ.

Если вы используете MySQL Workbench, вам, возможно, потребуется отключить настройку безопасности (меню Edit/ Preferences, вкладка SQL Editor, флажок *Safe Updates...*). Данная настройка не позволяет выполнить запросы UPDATE или DELETE без указания WHERE со ссылкой на первичный ключ.

PostgreSQL

Для PostgreSQL синтаксис несколько отличается от MS SQL и MySQL тем, что в данном запросе не используется ключевое слово JOIN, а только подразумевается:

```
UPDATE cars
SET manufacture_year = master_cars.manufacture_year
FROM master_cars
WHERE cars.license_plate = master_cars.license_plate
AND cars. manufacture_year IS NULL;
```

Несмотря на отсутствие JOIN, данный запрос наиболее интуитивно понятен: обновить автомобили, установить год выпуска по значению из master_cars ИЗ [данных таблицы] master_cars ГДЕ … (номера совпадают), впрочем, человеку, который мигрирует из другой СУБД, данный синтаксис может показаться странным.

9.4. Разрешение конфликтов с ограничениями

9.4.1. Техника UPSERT

Попробуйте внести новый автомобиль в таблицу CARS, но при этом укажите уже имеющийся в таблице регистрационный номер, например 'XYZ789':

```
INSERT INTO cars (license_plate, manufacturer, model)
VALUES ('XYZ789', 'Lamborgini', 'Diablo');
```

Вы получите сообщение об ошибке. Точный текст зависит от СУБД, которую вы используете, суть же сообщения будет в том, что мы нарушаем ограничение уникальности (UNIQUE), предусмотренное для столбца *license_plate*.

Ошибка также возникнет, если мы попробуем нарушить ограничение CHECK для столбца *status*:

```
INSERT INTO cars (license_plate, manufacturer, model, status)
VALUES ('AAA111', 'Lamborgini', 'Diablo', 'неизвестно');
```

При нарушении ограничений (UNIQUE, CHECK, PRIMARY KEY и др.) СУБД предотвращает некорректную вставку данных, но часто требуется осознанно обработать конфликт.

Средствами SQL мы можем либо не производить вставку в случае конфликта, либо обновить существующую запись (ту, с которой возник конфликт). Такая техника называется UPSERT — комбинация слов UPDATE и INSERT.

Рассмотрим техники подробнее для каждой из СУБД.

Microsoft SQL Server

Реализовать оба варианта можно при помощи предварительной проверки, существует ли запись с конфликтующим значением:

```
IF NOT EXISTS (SELECT 1 FROM cars WHERE license_plate = 'XYZ789')
INSERT INTO cars (license_plate) VALUES ('XYZ789')
```

Конструкция IF NOT EXISTS (*если не существует*) вычисляет аргумент в скобках и продолжает выполнение запроса, только если результат вычисления истинен.

В предложенном выше варианте вставка будет выполнена, только если автомобиля с таким регистрационным знаком не существует.

Если требуется выполнить UPSERT, можно дополнить выражение конструкцией ELSE (иначе), которая будет выполнена, если результат вычисления выражения в скобках окажется ложным.

```
IF NOT EXISTS (SELECT 1 FROM cars WHERE license_plate = 'XYZ789')
INSERT INTO cars (license_plate) VALUES ('XYZ789')
ELSE UPDATE cars SET mileage = 68000 WHERE license_plate = 'XYZ789';
```

Можно обратить условие проверки, чтобы проверить не отсутствие, а наличие записи, удалив из запроса унитарный оператор NOT и поменяв местами запросы на вставку и удаление.

```
IF EXISTS (SELECT 1 FROM cars WHERE license_plate = 'XYZ789')
UPDATE cars SET mileage = 68000 WHERE license_plate = 'XYZ789'
ELSE INSERT INTO cars (license_plate) VALUES ('XYZ789');
```

Оба запроса эквивалентны.

MySQL

Данная СУБД имеет несколько способов для безопасной вставки данных.

1. Во-первых, можно просто игнорировать ошибку, используя ключевое слово IGNORE:

```
INSERT IGNORE INTO cars (license_plate, manufacturer, model)
VALUES ('XYZ789', 'Lamborgini', 'Diablo');
```

Получить информацию о том, была ли проведена вставка, можно по возвращаемому значению (1 — запись добавлена, 0 — запись не добавлена, произошла ошибка).

ПРИМЕЧАНИЕ.

Ключевое слово IGNORE проигнорирует любое предупреждение, а не только конфликт, вернув 0 в случае, если что-то пойдет не так.

2. Во-вторых, можно провести замену существующей записи или вставку, если ее нет, при помощи команды REPLACE (заменить):

```
REPLACE INTO cars (license_plate, manufacturer, model)
VALUES ('XYZ789', 'Lamborgini', 'Diablo');
```

В случае если запись существует, значения указанных столбцов будут заменены на новые (если имя столбца не указано в запросе, его значение останется неизменным). Если записи нет, логика будет идентичной команде INSERT.

Еще одним способом провести UPSERT в MySQL является конструкция ON DUPLICATE KEY … в составе команды INSERT (приведен новый синтаксис с использованием псевдонима):

```
INSERT INTO cars (license_plate, manufacturer, model)
VALUES ('XYZ789', 'Lada', '21063')
AS new_values
ON DUPLICATE KEY UPDATE
  manufacturer = new_values.manufacturer,
  model = new_values.model;
```

ПРИМЕЧАНИЕ.

Даже в случае существования записи счетчик автонумератора первичного ключа все равно увеличится, принимайте это во внимание и используйте альтернативы, если такое поведение СУБД нежелательно.

В сети вы можете встретить альтернативный синтаксис с использованием функции VALUES(), однако поддержка данной функции в будущих версиях MySQL будет отключена, поэтому используйте вариант с применением псевдонима (*new_values* в нашем примере).

PostgreSQL

В PostgreSQL для обработки конфликтов используется конструкция ON CONFLICT (*при конфликте*). Для пропуска значения необходимо добавить DO NOTHING (*ничего не делать*):

```
INSERT INTO cars (license_plate, manufacturer, model)
VALUES ('XYZ789', 'Lada', 'Granta')
ON CONFLICT (license_plate)
DO NOTHING; -- Пропустить вставку
```

Если же требуется замена существующих значений, необходимо указать DO UPDATE SET (*провести обновление, установить*), затем указать, какие значения нужно обновить (обновлять можно не все поля):

```
INSERT INTO cars (license_plate, manufacturer, model)
VALUES ('XYZ789', 'Lada', 'Granta')
ON CONFLICT (license_plate)
DO UPDATE SET
  manufacturer = EXCLUDED.manufacturer,
  model = EXCLUDED.model;
```

Сгенерированная временная таблица с псевдонимом EXCLUDED будет содержать исключенные из вставки значения.

9.4.2. Объединение таблиц (MERGE)

Иногда требуется обновить информацию в одной таблице по данным другой, следуя определенной логике при разрешении конфликтов. Для этих целей используется команда MERGE (*слияние*). Эта команда позволяет синхронизировать данные между двумя таблицами, выполняя обновление, вставку или удаление записей в целевой таблице на основе сравнения с исходными данными. Это особенно полезно при работе с мастер-данными, журналами изменений или пакетными обновлениями.

Ключевые элементы синтаксиса:

```
MERGE INTO <целевая_таблица>
```

Указывает целевую таблицу, в которую будут вноситься изменения (*провести слияние таблицы...*):

```
USING <таблица_источник>
```

Определяет таблицу (это может быть и подзапрос), которая является источником данных для сравнения (*...используя таблицу-источник...*):

```
ON <условие>
```

Эта конструкция задает условие, по которому записи из целевой и исходной таблиц считаются совпадающими. Принципы сопоставления условий здесь такие же, какие используются при объединении JOIN.

```
WHEN MATCHED THEN <действие>
```

"*Когда совпадет, тогда...*" — здесь указывается действие для записей, которые удовлетворяют условиям сопоставления, указанным в конструкции ON. Обычно в этом разделе указывается действие по обновлению таблицы, например:

```
UPDATE SET целевая_таблица.столбец = источник.столбец
```

Далее идут необязательные части:

```
WHEN NOT MATCHED THEN <действие>
```

SKLADCHIK.ORG

"*Когда НЕ совпадет, тогда...*" — здесь указывается действие для записей, для которых не будут найдены сопоставления, указанные в конструкции ON. В данном месте обычно указывается команда на вставке:

```
INSERT INTO <целевая_таблица> (столбец) VALUES (источник.столбец)
```

Данное действие выполняется, когда в источнике есть записи, отсутствующие в целевой таблице.

Наконец, для записей, которые имеются в целевой таблице, но отсутствуют в источнике, предусмотрен раздел:

```
WHEN NOT MATCHED BY SOURCE THEN <действие>
```

"*Когда нет совпадений в источнике, тогда...*" — обычно в данном месте указывается команда на удаление (DELETE).

Вот как можно решить задачу по обновлению данных из мастер-данных по автомобилям (листинг доступен в электронном приложении):

```
MERGE INTO cars AS target
USING master_cars AS source
ON target.license_plate = source.license_plate
WHEN MATCHED THEN
 UPDATE SET
  manufacturer = source.manufacturer,
  model = source.model,
  manufacture_year = source.manufacture_year,
  mileage = source.mileage,
  last_service = source.last_service,
  status = source.status
WHEN NOT MATCHED THEN
 INSERT (
  license_plate,
  manufacturer,
  model,
  manufacture_year,
  last_service,
  status)
```

```
 VALUES (
  source.license_plate,
  source.manufacturer,
  source.model,
  source.manufacture_year,
  source.last_service,
 source.status)
WHEN NOT MATCHED BY SOURCE THEN DELETE;
```

Сначала мы определяем целевую таблицу (cars) и источник данных (master_cars), обратите внимание, для удобства мы присваиваем обеим таблицам псевдонимы: *source* — источник и *target* — таблица назначения. Затем определяется правило сопоставления (ON), где регистрационные знаки в обеих таблицах должны совпадать.

В случае совпадения (WHEN MATCHED THEN) содержимое полей совпадающей записи из таблицы-источника копируется в сопоставленную запись таблицы-назначения при помощи команды UPDATE.

В противном случае (WHEN NOT MATCHED THEN) выполняется команда INSERT по данным таблицы-источника.

И наконец, если в таблице cars присутствуют какие-либо дополнительные записи (WHEN NOT MATCHED BY SOURCE), отсутствующие в мастер-данных, они удаляются (DELETE).

ПРИМЕЧАНИЕ.

Если указано несколько условий WHEN MATCHED, их порядок влияет на результат. Используйте AND для уточнения:

```
WHEN MATCHED AND source.status = 'in repairs' THEN …
```

Всегда проверяйте, что условие ON однозначно идентифицирует записи. Если в источнике есть дубликаты, запрос завершится ошибкой.

Для больших объемов данных используйте индексы на колонках, участвующих в условии ON (индексы мы будем подробно рассматривать позднее).

ПРИМЕЧАНИЕ.

Для пользователей **MySQL** функционал MERGE в данной СУБД появился только в **версии 3.23.25** и на момент написания данного материала является тестируемой возможностью, имеющей ограниченную поддержку. До момента, пока данный функционал не станет поддерживаться официально, рекомендуется пользоваться техникой ON DUPLICATE KEY UPDATE, рассмотренной ранее.

9.5. Удаление данных

9.5.1. Простое удаление DELETE

Для простого удаления данных из таблицы необходимо воспользоваться командой DELETE:

```
DELETE FROM <имя_таблицы> WHERE <условие>;
```

ВАЖНО!

- Всегда начинайте написание запроса DELETE с написания условий в WHERE.
- Всегда проверяйте условие WHERE.
- Используйте транзакции (будем рассматривать позднее).
- Старайтесь иметь свежую и проверенную резервную копию данных, если возможно.

Простое удаление данных из таблицы **cars** по номеру:

```
DELETE FROM cars WHERE license_plate = 'XYZ789';
```

ПРИМЕЧАНИЕ.

Строго говоря, конструкция WHERE необязательна, однако, если ее не указать, из таблицы будут удалены все записи.

9.5.2. Удаление по данным подзапроса или CTE

Мы можем удалять данные из таблицы, по данным поздапроса. Обычно данный прием используется, когда необходимо получить идентификаторы для удаления из другой таблицы.

Например, чтобы удалить данные из таблицы **cars** на основании статуса из мастер-данных, можно выполнить следующий запрос:

```
-- Удалить автомобили со статусом "в ремонте"
DELETE FROM cars WHERE license_plate IN
(SELECT license_plate FROM master_cars WHERE status = 'in repairs');
```

Мы используем подзапрос, выбирающий номера автомобилей из мастер-данных со статусом "in repairs" (*в ремонте*), и удаляем из таблицы **cars** те записи, чьи номера найдены в результатах подзапроса (используется оператор IN).

Мы также можем использовать обобщенные табличные выражения (CTE) для получения данных о том, что нужно удалить.

Использование CTE в следующем примере излишне, так как того же эффекта можно добиться при помощи обычного подзапроса, пример приводится в качестве иллюстрации концепции:

```
-- Удалить автомобили с пробегом более 100000 км
WITH old_cars AS
(SELECT license_plate FROM master_cars WHERE mileage > 100000)
DELETE FROM cars
WHERE license_plate IN (SELECT license_plate FROM old_cars);
```

9.5.3. Удаление через присоединение

Мы также можем использовать присоединение (JOIN), чтобы удалить только те данные, которые удовлетворяют условиям, указанным в ON. Это полезно, когда необходимо использовать данные из связанной через внешний ключ таблицы:

```
-- Удалить водителей, чьи автомобили в ремонте
DELETE d
FROM drivers d
JOIN cars c ON d.assigned_car = c.car_id
WHERE c.status = 'in repairs';
```

Удаление происходит по результатам сопоставления двух таблиц по внешнему ключу. Мы фильтруем автомобили по статусу "in repairs" (в ремонте), затем удаляем водителей из таблицы **drivers**, чьи автомобили имеют такой статус.

ПРИМЕЧАНИЕ.

В PostgreSQL удаление через присоединение не поддерживается. Аналогичный результат достигается использованием подзапроса:

```
-- PostgreSQL аналог без JOIN
DELETE FROM drivers
WHERE assigned_car IN (
  SELECT car_id
  FROM cars
  WHERE status = 'in repairs'
);
```

9.6. Контроль целостности при удалении данных

9.6.1. Действия ON DELETE

При удалении данных из таблиц может возникнуть ситуация, когда на удаляемые данные имеется ссылка из другой таблицы через внешний ключ. Поведение СУБД в таких ситуациях будут зависеть от действий ON DELETE, которые были указаны при определении внешнего ключа для столбца.

Чтобы проиллюстрировать пример, попробуем выполнить удаление из таблицы cars для тех записей, на которые имеется внешняя ссылка (если вы экспериментировали ранее, вам, возможно, потребуется восстановить записи в таблицах из мастер-данных и назначить автомобили нескольким водителям):

```
DELETE FROM cars
WHERE car_id IN
(SELECT assigned_car FROM drivers WHERE assigned_car IS NOT NULL);
```

Поскольку мы не указали никаких действий на случай удаления (ON DELETE) для внешнего ключа *assigned_car* при создании таблицы **drivers**, СУБД применяет действие по умолчанию (запрет) и не разрешает выполнение запроса.

Давайте изменим таблицу и добавим в нее действие ON DELETE SET NULL.

Вспомним, что изменения для структуры таблиц производятся через команду ALTER (Изменить):

```
-- Удаляем существующее ограничение
ALTER TABLE drivers
DROP CONSTRAINT fk_drivers_cars;

-- Создаем новое ограничение
ALTER TABLE drivers
ADD CONSTRAINT fk_drivers_cars
FOREIGN KEY (assigned_car)
REFERENCES cars(car_id)
ON DELETE SET NULL;
```

Мы удалили имеющееся ограничение *fk_drivers_cars* и создали новое, где указано действие SET NULL.

Теперь, как вы можете сами убедиться, при выполнении того же запроса автомобили, закрепленные за водителями, успешно удалились, а в таблице **drivers** значения столбца *assigned_car* будут установлены в NULL.

Восстановите мастер-данные в таблицах и закрепление автомобилей за водителями по примерам, приведенным ранее, затем измените действие ON DELETE CASCADE для *fk_drivers_cars*:

```
-- Удаляем существующее ограничение
ALTER TABLE drivers
DROP CONSTRAINT fk_drivers_cars;

-- Создаем новое ограничение
ALTER TABLE drivers
ADD CONSTRAINT fk_drivers_cars
FOREIGN KEY (assigned_car)
REFERENCES cars(car_id)
ON DELETE CASCADE;
```

Удостоверьтесь, что в таблице **drivers** имеются закрепленные за водителями автомобили:

```
SELECT d.driver_id, d.full_name, c.license_plate
FROM drivers d
LEFT JOIN cars c
ON d.assigned_car = c.car_id;
```

Если вы выполняли запросы из примеров, результат данного запроса должен выглядеть следующим образом:

driver_id	full_name	license_plate
101	Деточкин Юрий	JKL012
102	Шлыков Иван	NULL
103	Морозов Александр	NULL

104	Хачикян Рубик	MNO345
105	Громов Виктор	NULL
106	Шумилов Григорий	NULL
107	Низовцев Андрей	NULL

Теперь удалим из таблицы cars автомобиль MNO345:

```
DELETE FROM cars WHERE license_plate = 'MNO345';
```

Затем снова проверьте содержимое таблицы **drivers**. Вы можете удостовериться, что водитель 104 удален вместе с автомобилем:

driver_id	full_name	license_plate
101	Деточкин Юрий	JKL012
102	Шлыков Иван	NULL
103	Морозов Александр	NULL
105	Громов Виктор	NULL
106	Шумилов Григорий	NULL
107	Низовцев Андрей	NULL

Стоит напомнить, что действие NO ACTION аналогично RESTRICT, но действует немного позднее (после завершения транзакции).

9.6.2. Таблица сравнения действий

Ниже приведена в качестве напоминания таблица сравнения возможных действий для внешних связей:

Действие	Описание	Типичное применение	Производительность
RESTRICT	Запрещает удаление записи в родительской таблице, если в дочерней таблице есть связанные данные. Удаление будет отклонено до тех пор, пока зависимые записи не будут удалены или изменены вручную	Когда дочерние данные **обязательно должны существовать только вместе с родителем**. Например, нельзя удалить клиента, если у этого клиента есть заказы	Очень высокая, так как проверяется только само наличие зависимых записей

Действие	Описание	Типичное применение	Производительность
SET NULL	Устанавливает в зависимой значение NULL. Столбец внешнего ключа должен поддерживать значения NULL	Когда связь между таблицами не обязательна и дочерние данные могут существовать без родителя.	Быстрее, чем CASCADE, так как не требует удаления данных. Может приводить к фрагментации индексов, если NULL-значений становится много
SET DEFAULT	Устанавливает значение по умолчанию. Столбец внешнего ключа должен иметь значение по умолчанию (DEFAULT)	Когда нужно сохранить **базовое состояние** дочерних записей, например, товар "без категорий", если была удалена категория для товара	Чуть медленнее SET NULL, так как требует проверки значений по умолчанию
CASCADE	При удалении родительской записи автоматически удаляются все связанные записи в дочерней таблице	Когда дочерние данные не имеют смысла без родителя и удаление таких данных допустимо	Может быть медленной при большом объеме дочерних данных, так как требует удаления множества записей. Также существует риск блокировок при массовых операциях. Может вызвать рекурсивное удаление, если на дочерние данные также имеются внешние ссылки

Рекомендации:

- Если на текущий момент нет четкого понимания, как обрабатывать зависимости, используйте действие по умолчанию (**RESTRICT**).
- Если данные могут существовать независимо друг от друга (например, водитель без автомобиля), используйте **SET NULL**.

- Если вы имеете дело со столбцом, имеющим ограничение NOT NULL, укажите для него значение по умолчанию и используйте действие **SET DEFAULT**.
- Если вы имеете дело с жестко связанными данными и дочерние данные не имеют смысла без родительских, используйте **CASCADE** (может серьезно замедлить работу, особенно если операция удаления используется часто).

9.7. Подходы к хранению невостребованных данных

Хранение невостребованных данных — задача, требующая баланса между производительностью, затратами на хранение и юридическими или бизнес-требованиями.

Существует несколько стратегий по хранению и удалению данных. Ниже представлены некоторые из них.

9.7.1. Немедленное удаление (IMMEDIATE DELETE)

При реализации этой стратегии данные полностью удаляются из базы, как только становятся ненужными. Подходит для временных данных (кеш, логи с коротким сроком актуальности) при условии, что объем таких данных небольшой и не имеет зависимостей, так как операция немедленного удаления может отвлечь ресурсы СУБД от более важных операций.

Плюсы:

- Экономия места на диске.
- Упрощение запросов (нет необходимости фильтровать "удаленные" записи).

Минусы:

- Невозможность восстановления данных или аудита.

SKLADCHIK.ORG

- Риск потери критической информации (например, истории действий пользователей).

9.7.2. "Мягкое" удаление (SOFT DELETE)

Данные вовсе не обязательно удалять сразу. Можно просто пометить их как "удаленные", снабдив таблицу столбцом *is_deleted*. Подходит для пользовательских данных (аккаунты, заказы), где важен контроль изменений.

Плюсы:

- Сохранение истории для аудита и анализа.
- Возможность восстановления данных ("отмена удаления").

Минусы:

- Усложнение запросов: во всех выборках нужно учитывать условие `WHERE is_deleted = false`.
- Рост объема таблиц, что замедляет работу и тратит дисковое пространство.

9.7.3. Отсроченное удаление (DELAYED DELETE)

Данные помечаются как устаревшие и удаляются через заданный интервал, например через 90 дней. Обычно реализуется путем добавления столбца с DATETIME, где указывается дата предполагаемого удаления. В будущем все данные с датой удаления ранее определенной даты можно удалить одним запросом. Применяется в случаях, например, где данные удаляются после определенного срока неактивности пользователя или где имеются регуляторные требования по хранению (например, налоговая отчетность, документы по которой необходимо хранить определенное количество лет).

Плюсы:

- Гибкость при определении срока хранения данных.
- Баланс между объемом хранения и возможностью восстановления данных.
- Удаление можно проводить в удобное время (например, в периоды минимальной нагрузки).

Минусы:

- Сложность реализации: требуется механизм очистки (скрипты, планировщики).
- Повышенный расход дискового пространства.

9.7.4. Постоянное хранение (NO DELETE)

Не подразумевает удаление в принципе. Данные не удаляются никогда, даже если утратили актуальность. Используется, когда существуют требования по соответствию юридическим нормам (например, финансовые транзакции).

Плюсы:

- Полная история доступна всегда для полного анализа.

Минусы:

- Неограниченный рост занимаемого объема.
- Возможно низкое быстродействие.
- Высокие затраты.

9.8. Производительность запросов *UPDATE* и *DELETE*

Операции UPDATE и DELETE могут стать узким местом в работе с данными, особенно при обработке больших объемов. Если вы сталкиваетесь с низкой производительностью при обновлении или удалении данных, вопросы, которые рассматриваются в данном разделе, могут оказаться вам полезны.

9.8.1. Временное отключение проверок и триггеров

Действия ON UPDATE/ON DELETE (CASCADE, SET NULL и др.) автоматически обновляют или удаляют связанные данные. Это удобно, но может замедлить операции.

Проверяйте, какие таблицы зависят от данных, которые вы удаляете. Удостоверьтесь, что вам действительно необходимо поддерживать эти связи, особенно это касается возможности рекурсивного обновления или удаления, когда в цепочке из нескольких связанных таблиц включено действие CASCADE.

Кроме того, в базах данных имеется механизм, при котором по определенному событию будут выполняться некоторые инструкции SQL, которые называются **триггерами** (их мы будем рассматривать позднее). Например, **триггер** может проводить запись в журнал каждого удаления из таблицы, что дополнительно замедлит запрос на удаление.

Если это допустимо в вашем случае, рассмотрите возможность отключать проверки и триггеры перед проведением подобных операций, однако это несет в себе риски утери целостности данных.

Отключение и повторное включение проверок в **Microsoft SQL Server** выполняется следующими запросами (отключает любые проверки, не только ограничения по внешнему ключу):

```
-- Отключить проверки ограничений для таблицы drivers:
ALTER TABLE drivers NOCHECK CONSTRAINT ALL;
-- Повторно включить проверки:
ALTER TABLE drivers CHECK CONSTRAINT ALL;
```

Для **MySQL** проверки по внешнему ключу выключаются и включаются при помощи следующих запросов (настройка действует для текущего сеанса):

```
-- Отключить проверки внешнего ключа
SET FOREIGN_KEY_CHECKS = 0;
-- Повторно включить
SET FOREIGN_KEY_CHECKS = 1;
```

В **PostgreSQL** для отключения проверок придется удалять ограничения, а потом заново их включать.

9.8.2. Дробление операций на части

Массовые операции лучше разбивать на пакеты, чтобы избежать блокировок таблиц и снизить нагрузку. Ниже приведены реализации данного подхода для всех рассматриваемых СУБД. Разумеется, при искусственном ограничении количества обрабатываемых строк за один запрос необходимо будет повторять эти запросы до тех пор, пока они не будут возвращать 0 (количество затронутых запросом строк), либо иным образом контролировать, остались ли еще в таблице записи, подлежащие изменению или удалению.

Microsoft SQL Server

Используется функция TOP(n), ограничивающая исполнение запроса n строками:

```
-- Удаление автомобилей, произведенных до 2000 года
DELETE TOP (1000) FROM cars
WHERE manufacture_year < 2000;
```

MySQL

Используем уже известную нам по запросам SELECT инструкцию LIMIT (ограничить):

```
-- Удаление автомобилей, произведенных до 2000 года
DELETE FROM cars
WHERE manufacture_year < 2000
LIMIT 1000; -- Удаляет по 1000 строк за раз
```

PostgreSQL

Данная СУБД, к сожалению, не поддерживает ограничение LIMIT в запросах на удаление, поэтому необходимо использовать CTE:

```
WITH batch AS (
  SELECT car_id
```

```
  FROM cars
  WHERE manufacture_year < 2000
  LIMIT 1000 -- ограничиться 1000 строками
)
DELETE FROM cars
WHERE car_id IN (SELECT car_id FROM batch);
```

9.8.3. Дополнительные рекомендации по производительности

При построении запросов на удаление помните, что поиск по первичному ключу очень хорошо оптимизирован во всех СУБД, поэтому всегда по возможности используйте фильтрацию по первичному ключу.

Иногда, если требуется несколько раз обрабатывать определенный набор данных, который содержится в большой таблице, есть смысл перенести эти данные во временную таблицу, обработать их там, затем перенести уже законченный результат обратно.

Что узнали и чему мы научились?

В этой главе вы освоили ключевые аспекты работы с данными в SQL, выходящие за рамки простых запросов на выборку. Вы научились применять дополнительные методы копирования данных, использовать для этого CTE и подзапросы, научились получать значение поля с автонумерацией при выполнении вставки.

Вы также освоили как методы модификации и удаления единичных строк, так и приемы массового обновления на основе данных из других таблиц, научились применять CASE для условного обновления.

Вы познакомились с принципами и инструментами, лежащими в основе управления целостностью данных, и научились управлять этими инструментами.

Теперь, когда вы научились эффективно изменять данные, наступает время задуматься о производительности. Даже самые продуманные запросы могут замедляться из-за отсутствия индексов, неоптимальных планов выполнения или блокировок.

Глава 10.

Индексы и оптимизация. Улучшаем производительность

10.1. Что такое индекс и зачем он нужен?

Представьте себе бумажную книгу на несколько сотен или даже тысяч страниц, в которой содержится информация о людях, фактах, событиях. Если бы вам потребовалось найти в книге все упоминания какой-либо определенной фамилии, вам бы потребовалось перелистывать каждую страницу и просматривать текст в поисках нужного слова, что заняло бы очень много времени.

Но если бы в конце книги был алфавитный указатель со всеми встречающимися в книге фамилиями и указанием номеров страницы, где они упоминаются, вам было бы достаточно просто открыть книгу на указанной странице, и ваш поиск был завершен.

Индексы в СУБД играют точно такую же роль. При помощи индекса СУБД может найти искомую запись очень быстро, не просматривая всю таблицу целиком.

Представьте, что вы ищете всех клиентов из города Санкт-Петербург (WHERE city = 'Санкт-Петербург'). Без индекса база данных (как человек с книгой без указателя) проверит каждую запись в таблице клиентов, чтобы понять, в каком городе живет каждый. Это медленно, особенно если записей миллионы. Если же по столбцу city создан индекс, база данных сразу знает, где искать Санкт-Петербург (как вы знаете страницы из указателя), и находит нужные записи в разы быстрее. То же самое относится к соединению таблиц (JOIN) — индекс помогает быстро найти совпадающие записи. Если вам нужно отсортировать результат по фамилии (ORDER BY last_name), а по last_name есть индекс (который уже хранит фамилии по порядку), база данных может просто "прочитать" данные в порядке индекса, избегая медленной сортировки всей таблицы. Индексы — главные помощники для быстрого поиска и сортировки данных.

Однако, как и у книжного указателя, у индексов есть своя цена. Представьте, что в вашу книгу добавили новую главу или изменили имя персонажа на нескольких страницах.

Чтобы указатель оставался полезным, библиотекарю (базе данных) придется обновить и сам указатель — добавить новые слова или изменить номера страниц для измененных. Точно так же, когда вы добавляете новую запись в таблицу (INSERT), изменяете существующую (UPDATE) или удаляете (DELETE), база данных должна обновить все индексы, связанные с этой таблицей. Это добавляет лишней работы и может замедлить эти операции. Кроме того, сами индексы занимают место на диске, как дополнительные страницы в конце книги. Чем больше индексов вы создаете, тем больше места они требуют и тем сильнее могут замедлить изменение данных. Поэтому индексы создают не на все столбцы подряд, а выборочно, на те, которые часто используются для поиска и сортировки.

При использовании индексов всегда приходится идти на компромисс — скорость чтения против скорости добавления/обновления/удаления и места на диске.

10.2. Виды индексов

В разных СУБД существуют разные виды индексов, но два самых основных — это **кластеризованный** и **некластеризованный**. Представьте вашу таблицу как папку с бумагами.

Кластеризованный (clustered) индекс определяет физический порядок хранения этих бумаг в папке. Например, если вы создаете кластеризованный индекс по дате (order_date), то все записи в таблице будут фактически храниться на диске в порядке возрастания даты, как если бы вы подшили бумаги в хронологическом порядке.

Таблица может иметь только один кластеризованный индекс (ведь бумаги нельзя физически отсортировать одновременно по дате и по имени!).

Некластеризованный (non clustered) индекс — это отдельный список, как тот самый алфавитный указатель в конце книги. Он хранит копии значений из индексируемых столбцов (например, last_name) и ссылки (как номера страниц) на соответствующие строки в основной таблице.

Таких "отдельных списков" (некластеризованных индексов) для одной таблицы может быть много. Кластеризованный индекс упорядочивает сами данные, а некластеризованный создает отдельную "карту" для быстрого поиска.

10.3. Идентификаторы строк в СУБД

У некоторых читателей может сложиться впечатление, что данные внутри таблиц всегда хранятся в отсортированном по первичному ключу виде, однако это не так. Каждая СУБД по-своему оптимизирует внутреннее хранение, однако общий принцип остается одним и тем же: для каждой строки таблицы СУБД хранит внутренний идентификатор строки, который иногда называют ROWID (*row identifier* — идентификатор строки), — это и есть тот самый внутренний уникальный адрес каждой строки данных в таблице на физическом уровне хранения.

Идентификатор строки — прямое и самое быстрое указание местоположения конкретной строки в таблице. Когда индекс (наш "алфавитный указатель") находит нужную запись, он обычно получает именно ROWID этой записи.

Используя ROWID, СУБД может почти мгновенно (теоретически это O(1) — константная временная сложность, вне зависимости от размера данных) перейти к физическому месту хранения строки и прочитать или изменить ее данные, минуя полный перебор таблицы.

Каждая СУБД реализует собственные, довольно сложные и изощренные механизмы по управлению идентификаторами строк при добавлении, из-

SKLADCHIK.ORG

менении и удалении данных, и ROWID обычно недоступны пользователю, однако нужно понимать, что именно с ROWID в большинстве случаев ссылаются индексы.

ВАЖНОЕ ПРИМЕЧАНИЕ!

Не храните ROWID (CTID) — это временный физический адрес, он очень вероятно изменится, даже если вы явно не выполняете каких-либо изменяющих запросов к таблице, так как сервер СУБД может проводить какие-либо оптимизации с данными таблиц даже без участия пользователя.

ИД строки

Кластеризованный индекс ▼

ROWID	PK	name	surname	department	username
1	4	Игорь	Абрамов	Продажи	i.abramov
2	3	Светлана	Борисова	ИТ	s.borisova
3	1	Василий	Виноградов	ИТ	v.vinogradov
4	5	Марина	Георгиева	Бухгалтерия	m.georgieva
5	2	Сергей	Дмитриев	Продажи	s.dmitriev

Кластеризованный индекс по фамилии

surname	ROWID
Абрамов	1
Борисова	2
Виноградов	3
Георгиева	4
Дмитриев	5

Некластеризованный индекс по PK

PK	ROWID
1	3
2	5
3	2
4	1
5	4

Некластеризованный индекс по отделу

department	ROWID
Бухгалтерия	4
ИТ	2
ИТ	3
Продажи	1
Продажи	5

Некластеризованный индекс по имени пользователя с включённым (INCLUDE) названием отдела

username	department	ROWID
i.abramov	Продажи	3
m.georgieva	Бухгалтерия	5
s.borisova	ИТ	2
s.dmitriev	Продажи	1
v.vinogradov	ИТ	4

Рис. 10.1. *Данные, ROWID и некоторые типы индексов*

На этом примере видно, что физически данные хранятся в отсортированном виде согласно кластеризованному индексу по фамилии пользователя, данные первичного ключа обычно имеют свой некластеризованный индекс.

10.4. Типы индексов в разных СУБД

Все современные СУБД умеют работать с кластеризованными и некластеризованными индексами более-менее стандартным образом, однако у каждой из них имеются свои "фишки", наличие которых, возможно, позволит принять осознанное решение при выборе конкретной СУБД для предстоящего проекта. Ниже собраны сжатые сведения о некоторых особенностях, которые отличают одну СУБД от другой.

В большинстве ситуаций для поиска нужного значения в таблице применяется алгоритм B-Tree (двоичное дерево), обеспечивающий сложность поиска O(log *N*), где *N* — общее количество строк в таблице.

Физически указатели от индекса к данным располагаются в т. н. "страницах", это физические блоки памяти, в которых хранятся элементы индекса. Каждая страница содержит набор записей (ключей индекса) и ссылки на соседние страницы, образуя иерархическую структуру, называемую деревом индекса (обычно это дерево B+).

Каждый элемент страницы индекса состоит из:

- Ключа индекса (например, значение столбца, по которому построен индекс).
- Смещения или ссылки на фактические данные (в кластерных индексах сами данные хранятся в листьях, а некластерный индекс ссылается на эти данные.

В отдельных случаях применяется очень быстрый тип индекса, основанный на хеш-таблице (hash table), который работает быстрее (в идеале — O(1), на практике из-за возможных коллизий может быть чуть больше), однако такой тип индекса пригоден только для поиска строгого соответствия и бесполезен, если нужно найти значения больше или меньше определенного.

```
SELECT * FROM USERS WHERE ID = 448
```

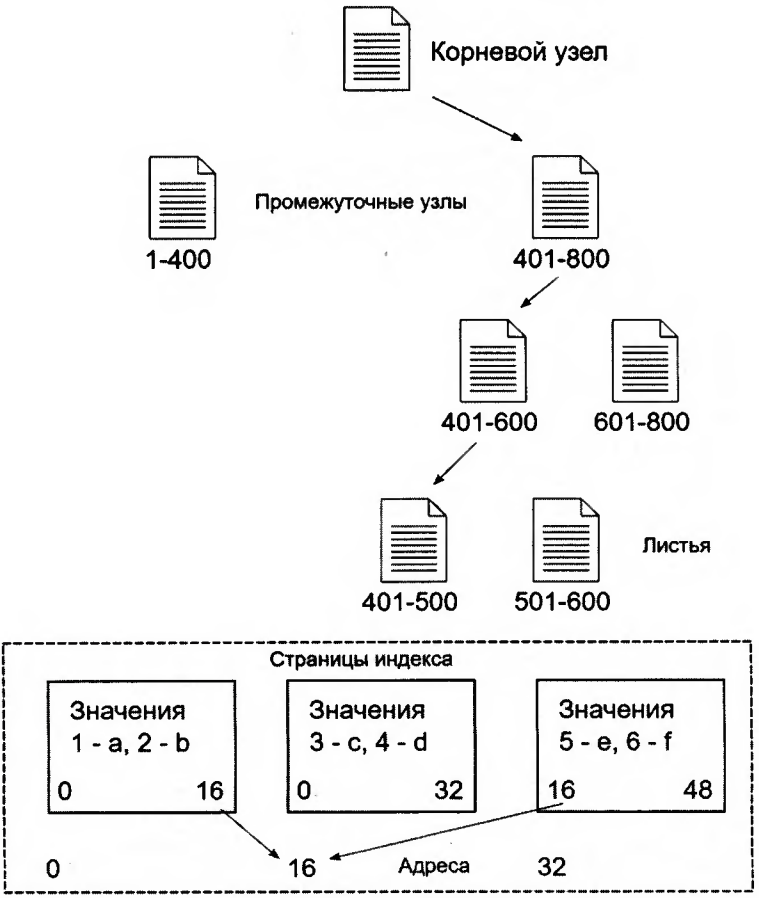


Рис. 10.2. *Дерево индекса*

10.4.1. MS SQL Server

Среди возможностей работы с индексами в MS SQL Server можно выделить три основные особенности:

- **Фильтрованные индексы**. Представьте, у вас есть таблица заказов и вы очень часто ищете только отмененные заказы (WHERE status = 'Отменен'). Фильтрованный индекс создаст указатель только для этих отмененных заказов. Он занимает меньше места и работает быстрее для

таких конкретных запросов, но бесполезен для поиска заказов с другим статусом.

- **Индексы столбцового хранения** (**COLUMNSTORE**). Представьте огромную таблицу с миллионами строк для отчетов и анализа (например, продажи за 10 лет). Обычные индексы тут могут быть неэффективны. Columnstore-индекс хранит данные не по строкам (одна запись за раз), а по **столбцам** (все значения одного столбца вместе), как будто для отчета собрали все цены вместе, все даты вместе и т.д. Это фантастически ускоряет запросы, которые суммируют (SUM), усредняют (AVG) или сканируют большие объемы данных по нескольким столбцам, но не подходит для случаев, где данные часто меняются или когда нужно находить отдельные строки в таблице.

- **Инклюзивные индексы**. Это не эксклюзивная особенность для MS SQL, но реализована лучше, чем в других СУБД. Обычно некластеризованный индекс хранит только индексируемые столбцы и ссылку на строку. Но если ваш запрос часто использует данные из других столбцов (например, SELECT name, email WHERE city='Москва'), базе приходится "ходить" по ссылкам в основную таблицу за **name** и **email**. Включенные столбцы (**INCLUDE** (name, email)) позволяют добавить эти данные в сам индекс. Теперь индекс по **city** содержит и **name** с **email** прямо в "указателе". Если продолжить аналогию с книжным указателем, это похоже на то, что рядом с фамилией и страницей сразу печаталась бы и должность человека. Запрос выполняется быстрее, но индекс занимает чуть больше места.

Кроме того, MS SQL Server имеет впечатляющие возможности по работе с XML-данными и данными о расположении объектов в пространстве (SPATIAL DATA — точки, линии, полигоны и т. д.).

Для XML-документов (структурированный текст) эта СУБД может создать специальный индекс, чтобы быстро искать теги или атрибуты внутри XML.

Для пространственных данных создаются особые индексы, позволяющие очень быстро отвечать на вопросы типа "Найди все магазины в радиусе 5 км от меня" или "Какие участки граничат с этим лесопарком?". Это узкоспециализированные инструменты для своих задач.

10.4.2. MySQL (InnoDB)

Главная особенность индексов в MySQL — кластеризованный индекс **всегда** создается по первичному Ключу (PK). Это основа хранения всей таблицы.

Все вторичные (некластеризованные) индексы в InnoDB (это современный "движок" базы данных для MySQL, пришедший на смену устаревшему MyISAM) автоматически включают в себя значения столбцов Первичного Ключа (PK). Зачем это нужно: когда вторичный индекс (например, по email) находит нужную запись, ему нужен PK, чтобы по нему найти полную строку в кластеризованном индексе. Храня PK прямо во вторичном индексе, InnoDB избегает лишнего шага поиска. Поэтому **при работе с MySQL старайтесь выбирать простой и компактный первичный ключ (например, целое число), это повысит эффективность всех используемых индексов в MySQL!**

Еще одной отличительной особенностью для MySQL является довольно удобное и простое в использовании средство индексирования для полнотекстового (**FULLTEXT**) поиска, когда в больших текстах надо искать текст не по точному соответствию, а по смыслу. Например, надо найти все статьи, где упоминаются 'базы данных' и 'оптимизация'. Обычный индекс по текстовому столбцу тут плохо поможет. FULLTEXT индекс разбивает текст на слова, учитывает их формы (ищет "бежать", если спросили "бежал") и позволяет искать по релевантности. Средства для полнотекстового поиска имеются и в других СУБД, но по простоте использования MySQL лидирует.

Составные индексы — не уникальны только для MySQL, но очень важны при использовании данной СУБД. Индекс можно создать не по одному столбцу, а по нескольким сразу (last_name, first_name). Такой индекс полезен, когда ваши запросы используют условия или сортировку по этим столбцам вместе (например, WHERE last_name='Иванов' AND first_name='Петр' или ORDER BY last_name, first_name). **Порядок столбцов в таком индексе критически важен!**

Одним из возможных недостатков для MySQL при решении некоторого класса задач является ограниченная поддержка Hash индексов, которые доступны только для специального вида таблиц (in-memory), которые хранятся в оперативной памяти.

10.4.3. PostgreSQL

Поскольку PostgreSQL предлагает самый богатый набор разнообразных видов индексов, здесь следует рассмотреть каждый из видов подробнее. Помимо уже известных B-Tree и Hash, PostgreSQL предлагает:

- **GiST** (**G**eneralized **S**earch **T**ree — *Обобщенное дерево поиска*) — это "швейцарский нож" для индексации сложных данных. GiST позволяет создавать индексы для самых разных типов данных и операций: геоданных (полигоны, расстояния), диапазонов чисел или дат, полнотекстового поиска, работы с графами (поиск путей). Он очень гибкий, но может быть медленнее специализированных индексов для конкретных задач.

- **SP-GiST** (**S**pace-**P**artitioned **GiST**) — улучшенная версия GiST, особенно хорошо подходящая для индексации данных, которые можно естественным образом разделить на непересекающиеся области: точки на плоскости, деревья (иерархии), IP-адреса. Может быть эффективнее обычного GiST для таких структур.

- **GIN** (**G**eneralized **I**nverted **I**ndex — *обобщенный инвертированный индекс*) — специалист по работе с составными значениями, где нужно искать вхождения элементов. Идеален для массивов, JSON-данных и хорошо подходит для полнотекстового поиска.

- **BRIN** (**B**lock **R**ange **IN**dex — *блочный диапазонный индекс*) — предназначен для очень больших таблиц, где данные физически хранятся на диске в определенном порядке (например, по дате создания). BRIN не запоминает каждую строку, а хранит сводную информацию (мин./макс. значения) для блоков данных. Если вы ищете данные за конкретный день, BRIN быстро скажет: "Смотри в блоках 100–120, там лежат записи с 2025-01-01 по 2025-01-05, остальные блоки пропускай". Он занимает очень мало места и отлично ускоряет запросы по упорядоченным столбцам в огромных таблицах, но менее точен, чем B-Tree.

PostgreSQL также поддерживает частичные индексы (WHERE-условия), которые являются полным аналогом фильтрованных индексов в MS SQL Server. Создает индекс только для подмножества строк, удовлетворяющих условию (CREATE INDEX ... WHERE status = 'active'). Экономит место и ускоряет запросы именно к этому подмножеству.

10.4.4. Обобщенная информация по индексам для разных СУБД

Выбор СУБД редко зависит только от индексов. Однако понимание их сильных сторон помогает предсказать, какая система лучше справится с вашими конкретными типами запросов и структурой данных.

В таблице собраны основные ключевые моменты по работе с индексами для разных СУБД:

Возможности	MS SQL Server	MySQL	PostgreSQL	Примечания
Кластеризация	На любом столбце (1 на таблицу)	Всегда на первичном ключе	На любом столбце (1 на таблицу)	Нет выраженного лидера. MySQL жестко связывает структуру данных с PK. Выбор компактного PK в MySQL критически важен для всех индексов
Вторичные индексы	Стандартные	Автоматически включают столбцы PK	Стандартные	Нет выраженного лидера В MySQL PK "вшит" во все вторичные индексы для быстрого перехода к данным
Включение других столбцов в индекс (INCLUDE)	Есть	Нет (частично заменяется вшитым PK)	Нет (есть концепция покрывающих индексов, но без спец. синтаксиса INCLUDE)	**MS SQL Server** лидирует: позволяет добавлять в индекс неключевые столбцы для покрытия запроса, экономя доступ к основной таблице

Возможности	MS SQL Server	MySQL	PostgreSQL	Примечания
Фильтрованные (Частичные)	Есть	Нет	Есть	Выбор за MS SQL Server или PostgreSQL
Аналитика (OLAP)	Индексы COLUMNSTORE	Нет (или ограниченно через движки типа ColumnStore)	Ограниченно через BRIN (для больших упорядоченных данных)	**SQL Server** лидирует: COLUMNSTORE фантастически ускоряет аналитические запросы (SUM, AVG, сканирования) по огромным таблицам
Работа с JSON/ массивами	Специальные XML-индексы	Обычные B-tree по извлеченным значениям	GIN/GiST-индексы	**PostgreSQL** — лидер: специализированные индексы (GIN) невероятно эффективны для поиска внутри JSON, массивов, полнотекстового поиска
Геоданные (Spatial)	Спец. (SPATIAL) индекс	Спец. (SPATIAL) индекс	GiST/SP-GiST-индексы	PostgreSQL слегка выделяется богатством операторов и производительностью на общем фоне, хотя поддержку имеют все
Полнотекстовый поиск	Встроенный полнотекстовый индекс	FULLTEXT индекс (в InnoDB)	GIN/GiST-индексы	**MySQL** лидирует, PostgreSQL имеют хорошую встроенную поддержку. MS SQL Server часто требует отдельной конфигурации

Возможности	MS SQL Server	MySQL	PostgreSQL	Примечания
Hash-индексы	Нет (внутренняя оптимизация)	Только для MEMORY-таблиц	Есть, для точного совпадения	**PostgreSQL** позволяет использовать хеш для быстрого поиска по точному совпадению на обычных таблицах. MySQL — только для временных таблиц в памяти
Спец-индексы	Нет	Нет	BRIN (для очень больших таблиц), SP-GiST	**PostgreSQL** — лидер по разнообразию

10.5. Как индексы ускоряют запросы

10.5.1. Генерация больших объемов данных

До сих пор мы использовали малые объемы данных для примеров, однако работа и польза индексов тем нагляднее, чем больше объем данных, по которым производится запрос, поэтому в этот раз нам необходимо сгенерировать внушительный объем данных (несколько миллионов строк). Листинги для генерации приведены в электронном приложении к книге.

Представим, нам надо проанализировать журналы посещений сайта крупного маркетплейса:

```
CREATE DATABASE marketplace;
```

На портал приходят разные пользователи, некоторые из них зарегистрированы как постоянные покупатели, а некоторые — нет, для учета регистрированных используем таблицу **users**:

```
-- Таблица пользователей (зарегистрированные)
CREATE TABLE users (
```

```
  user_id INT PRIMARY KEY,
  email VARCHAR(255) NOT NULL
);
```

Пользователи могут приходить по ссылкам от рефералов, у каждого из которых может быть предусмотрена скидка до 30%:

```
-- Таблица реферальных ссылок
CREATE TABLE referrals (
  referral_id INT PRIMARY KEY,
  discount DECIMAL(5,2) CHECK (discount BETWEEN 0 AND 30) --
Скидка 0-30%
);
```

Система фиксирует каждый сеанс (визит) пользователя на портал, где фиксирует пользователя, IP-адрес, реферала, если он есть, и дату начала сеанса:

```
-- Таблица сессий пользователей
CREATE TABLE sessions (
  session_id BIGINT PRIMARY KEY,
  user_id INT NULL, -- NULL для анонимных
  referral_id INT NULL, -- Ид реферала (если есть)
  ip_address VARCHAR(15) NOT NULL, -- IP-адрес
  -- заменить на TIMESTAMP для PostgreSQL:
  start_time DATETIME NOT NULL, -- Время подключения
  FOREIGN KEY (user_id) REFERENCES users(user_id),
  FOREIGN KEY (referral_id) REFERENCES referrals(referral_id)
);
```

Теперь попробуем на практике использовать те знания, которые мы получили до этого, чтобы заполнить таблицы. Нам понадобится около 100 рефералов, около 50 тыс. пользователей и порядка 10 млн сеансов).

Для работы с этим примером вам потребуется порядка 3 Гб дискового пространства под данные таблиц и индексы, которые будут созданы в процессе.

SKLADCHIK.ORG

Вот несколько полезных советов, которые могут вам пригодиться, если вы хотите поэкспериментировать с созданием тестовых данных самостоятельно.

1. Мы можем использовать CROSS JOIN для генерации больших объемов с использованием малых исходных данных.
2. Для генерации случайных чисел в MySQL используется функция RAND(), в PostgreSQL — функция RANDOM(). Результатом их вызова будет число в диапазоне от 0 до 1. Чтобы получить случайное число, например до 20, необходимо умножить результат функции на 20.
3. В MS SQL Server для генерации случайного числа используйте комбинацию ABS(CHECKSUM(NEWID())), а не RAND(), потому что RAND() в рамках одного запроса генерирует число только один раз. NEWID() при каждом вызове создает новый 128-битный идентификатор, который сложно использовать напрямую, поэтому мы вычисляем из него контрольную сумму CRC при помощи функции CHECKSUM, а затем функцией ABS берем получившееся число по модулю, так как CHESKSUM может вернуть и отрицательное значение. Поскольку результатом выполнения данной комбинации функций может быть любое целое число, для получения случайного в пределах от 0 до 20 надо использовать оператор получения остатка от деления (%) на 20.
4. Для конкатенации (склеивания строк) в MySQL и MSSQL можно использовать функцию CONCAT, в PostgreSQL можно также пользоваться оператором конкатенации ||.
5. Округление значений можно производить функциями FLOOR — для округления вниз, CEIL/CEILING — для округления вверх и функцией ROUND — для округления до нужного количества знаков до или после запятой.
6. В PostgreSQL для генерации последовательности из n чисел можно использовать функцию GENERATE_SEQUENCE().

Для заполнения таблиц данными сперва создадим временную таблицу **digits**, в которой содержатся числа от 0 до 9. Эту таблицу мы будем использовать для генерации любого количества записей, используя самопересечения. Один оператор CROSS JOIN даст нам 10*10 = 100 значений, два — 1000 и т.д.

```
CREATE TABLE numbers(n INT);

INSERT INTO numbers
SELECT 0 AS n
  UNION SELECT 1 UNION SELECT 2 UNION SELECT 3
  UNION SELECT 4 UNION SELECT 5 UNION SELECT 6
  UNION SELECT 7 UNION SELECT 8 UNION SELECT 9;
```

Используем самопересечение CROSS JOIN, чтобы создать 100 записей в таблице рефералов:

```
-- 100 рефералов
-- Раскомментируйте нужную строку для вашей СУБД
INSERT INTO referrals
SELECT n1.n + n2.n * 10 id,
  -- MS SQL Server
  -- CASE WHEN n2.n % 2 = 0 THEN (ABS(CHECKSUM(NEWID())) % 30)
  -- MySQL
  -- CASE WHEN n2.n % 2 = 0 THEN ROUND(RAND() * 30)
  -- PostgreSQL
  -- CASE WHEN n2.n % 2 = 0 THEN ROUND(RANDOM() * 30)
  ELSE NULL
END
FROM numbers n1
CROSS JOIN numbers n2
```

Следующим этапом нам надо будет сгенерировать 50 тыс. зарегистрированных пользователей. Здесь нам предстоит сгенерировать случайные адреса электронной почты. Для этих целей сгенерируем набор из 1000 случайных символов в диапазоне от *a* до *z* (коды символов с 97 по 122 включительно). В СУБД, с которыми мы работаем, имеется функция CHAR (в PostgreSQL она называется CHR), которая возвращает символ по его коду. В MySQL потребуется дополнительно приводить результат к типу CHAR.

Для создания 1000-символьной строки можно воспользоваться функцией STRING_AGG (доступна в MS SQL и PostgreSQL, в MySQL имеется эквивалент GROUP_CONCAT). Функция используется для агрегации строк в одну строку с возможным указанием разделителя между ними.

Для получения 1000 строк мы используем троекратное самопересечение вспомогательной таблицы **numbers**, содержащей числа от 0 до 9.

Из сгенерированного набора мы будем выбирать случайные подстроки длиной до 6 символов для того, чтобы создать случайные имена пользователей и домены для адресов электронной почты. Для выбора подстроки во всех трех СУБД имеется функция SUBSTRING (подстрока), которая принимает 3 аргумента: строку, из которой надо извлечь часть, позицию, начиная с которой надо начать извлечение, и длину подстроки, которую надо вернуть:

```
SELECT SUBSTRING('abcdefg', 2, 3); -- bcd
```

Полный запрос по формированию таблицы пользователей выглядит следующим образом (в комментариях указаны варианты строк для конкретной СУБД):

```
-- 50 тыс. пользователей
-- 1000 номеров
WITH thousandnumbers AS
(
  SELECT n1.n + n2.n * 10 + n3.n * 100 as n
  FROM
  -- троекратное самопересечение
  numbers n1 CROSS JOIN numbers n2 CROSS JOIN numbers n3
),
-- 1000 случайных символов
RandomSequence AS (
  SELECT
  -- MS SQL Server:
  -- STRING_AGG(CHAR(97 + ABS(CHECKSUM(NEWID())) % 26), '')
  -- MySQL:
  -- CAST(GROUP_CONCAT(CHAR(97 + RAND() * 26) SEPARATOR '') AS CHAR)
  -- PostgreSQL:
  -- STRING_AGG(CHR(97 + FLOOR(RANDOM() * 26)::INT), '')
  AS random_letters
  FROM thousandnumbers
)
INSERT INTO users
```

```
SELECT
  -- идентификатор
  n1.n + n2.n * 10 + n3.n * 100 + n4.n * 1000 + n5.n * 10000,
  -- email
  CONCAT(
  -- случайная подстрока из random_letters до 6 символов длиной
  -- MS SQL Server:
  -- SUBSTRING(rs.random_letters,
  --   ABS(CHECKSUM(NEWID())) % 1000, 2 + ABS(CHECKSUM(NEWID())) % 6),
  -- MySQL:
  -- SUBSTRING(rs.random_letters, RAND() * 1000, 2 + RAND() * 7),
  -- PostgreSQL:
  -- SUBSTRING(rs.random_letters, FLOOR(RANDOM() * 1000)::INT,
  --   2 + FLOOR(RANDOM() * 7)::INT),
  '@',
  -- MS SQL Server:
  -- SUBSTRING(rs.random_letters,
  --   ABS(CHECKSUM(NEWID())) % 1000, 2 + ABS(CHECKSUM(NEWID())) % 6),
  -- MySQL:
  -- SUBSTRING(rs.random_letters, RAND() * 1000, 2 + RAND() * 7),
  -- PostgreSQL:
  -- SUBSTRING(rs.random_letters, FLOOR(RANDOM() * 1000)::INT,
  --   2 + FLOOR(RANDOM() * 7)::INT),
  '.ru')
-- 50000 записей (5 самопересечений):
FROM RandomSequence rs
CROSS JOIN numbers n1 CROSS JOIN numbers n2 CROSS JOIN numbers n3
CROSS JOIN numbers n4 CROSS JOIN numbers n5
WHERE n5.n < 5 -- Нужно только 50 тыс.
```

ВНИМАНИЕ!

Следующий запрос в зависимости от вашей машины может занять длительное время (до 10 минут). Если вы используете графическую оболочку, вам, возможно, потребуется изменить время тайм-аута для запросов, чтобы не терять соединение с СУБД.

Нам предстоит добавить 10 млн записей с информацией о сеансах покупателей (время подключения, IP-адрес, информацию о реферальной ссылке, сведения о регистрации пользователя и время начала сеанса).

Если запросы на вашей машине работают чрезмерно медленно (более 10 минут), попробуйте уменьшить количество записей, сократив количество самопересечений в CTE tenmillion:

Microsoft SQL Server

```
-- 10 млн сеансов
WITH tenmillion AS
(
  -- число от 0 до 9 999 999
  SELECT n1.n + n2.n * 10 + n3.n * 100 +
         n4.n * 1000 + n5.n * 10000 +
         n6.n * 100000 + n7.n * 1000000
  AS n
  FROM numbers n1
  CROSS JOIN numbers n2 -- 10
  CROSS JOIN numbers n3 -- 100
  CROSS JOIN numbers n4 -- 1000
  CROSS JOIN numbers n5 -- 10000
  CROSS JOIN numbers n6 -- 100000
  CROSS JOIN numbers n7 -- 1000000
)
INSERT INTO sessions
SELECT
  -- Идентификатор сеанса
  tm.n,
  -- Идентфикатор пользователя
  -- Каждый 4-й сеанс от зарегистрированного пользователя
  CASE WHEN CHECKSUM(NEWID()) % 4 = 0
     THEN ABS(CHECKSUM(NEWID())) % 50000 ELSE NULL END,
  -- Каждый 100-й сеанс - через реферальную ссылку
  CASE WHEN CHECKSUM(NEWID()) % 100 = 0
     THEN ABS(CHECKSUM(NEWID())) % 100 ELSE NULL END,
  -- IP-адрес
```

```
  CONCAT(
  ABS(CHECKSUM(NEWID())) % 256, '.', ABS(CHECKSUM(NEWID())) % 256, '.',
  ABS(CHECKSUM(NEWID())) % 256, '.', ABS(CHECKSUM(NEWID())) % 256),
  -- Время начала сеанса
  DATEADD(SECOND, tm.n * 31536000.0 / 10000000.0, '2024-01-01')
FROM tenmillion tm
```

Здесь используются те же приемы, что мы уже пробовали раньше: используя 7 самопересечений с таблицей **numbers**, мы довели количество записей до 10 млн. Число 31536000 соответствует количеству секунд в одном году (365 * 24 * 60 * 60), используя его, мы просто равномерно распределили все сеансы на весь 2024 год.

Тексты запросов для MySQL и PostgreSQL отличаются лишь особенностями синтаксиса и принципиально не отличаются от подхода, который был использован ранее.

MySQL

```
INSERT INTO sessions
WITH tenmillion AS
(
  -- число от 0 до 9 999 999
  SELECT
    n1.n + n2.n * 10 + n3.n * 100 +
    n4.n * 1000 + n5.n * 10000 + n6.n * 100000 + n7.n * 1000000
  AS n
  FROM numbers n1
  CROSS JOIN numbers n2 -- 10
  CROSS JOIN numbers n3 -- 100
  CROSS JOIN numbers n4 -- 1000
  CROSS JOIN numbers n5 -- 10000
  CROSS JOIN numbers n6 -- 100000
  CROSS JOIN numbers n7 -- 1000000
)
SELECT
  -- Идентификатор сеанса
  tm.n,
```

```
  -- Идентфикатор пользователя
  -- Каждый 4-й сеанс от зарегистрированного пользователя
  CASE WHEN FLOOR(RAND() * 4) = 0 THEN FLOOR(RAND() * 50000) ELSE NULL END,
  -- Каждый 100-й сеанс - через реферальную ссылку
  CASE WHEN FLOOR(RAND() * 100) = 0 THEN FLOOR(RAND() * 100) ELSE NULL END,
  -- IP-адрес
  CONCAT(
  FLOOR(RAND() * 256), '.', FLOOR(RAND() * 256), '.',
  FLOOR(RAND() * 256), '.', FLOOR(RAND() * 256)),
  -- Время начала сеанса
  DATE_ADD('2024-01-01', INTERVAL tm.n * 31536000.0 / 10000000.0 SECOND)
FROM tenmillion tm
```

PostgreSQL

```
INSERT INTO sessions
WITH tenmillion AS
(
  -- число от 0 до 9 999 999
  SELECT n1.n + n2.n * 10 + n3.n * 100 + n4.n * 1000 + n5.n * 10000 +
n6.n * 100000 + n7.n * 1000000
  AS n
  FROM numbers n1
  CROSS JOIN numbers n2 -- 10
  CROSS JOIN numbers n3 -- 100
  CROSS JOIN numbers n4 -- 1000
  CROSS JOIN numbers n5 -- 10000
  CROSS JOIN numbers n6 -- 100000
  CROSS JOIN numbers n7 -- 1000000
)
SELECT
  -- Идентификатор сеанса
  tm.n,
  -- Идентфикатор пользователя
  -- Каждый 4-й сеанс от зарегистрированного пользователя
  CASE WHEN FLOOR(RANDOM() * 4) = 0 THEN FLOOR(RANDOM() * 50000) ELSE
NULL END,
  -- Каждый 100-й сеанс - через реферальную ссылку
```

```
  CASE WHEN FLOOR(RANDOM() * 100) = 0 THEN FLOOR(RANDOM() * 100) ELSE
NULL END,
  -- IP-адрес
  CONCAT(
  FLOOR(RANDOM() * 256), '.', FLOOR(RANDOM() * 256), '.',
  FLOOR(RANDOM() * 256), '.', FLOOR(RANDOM() * 256)),
  -- Время начала сеанса
  '2024-01-01'::TIMESTAMP + (tm.n * 31536000.0 / 10000000.0) * INTERVAL
'1 second'
FROM tenmillion tm;
```

Теперь, когда мы создали значительный объем данных, давайте посмотрим, с какой скоростью СУБД справляется с запросами. Мы построим несколько сложных (в части производительности) запросов и затем попробуем оптимизировать базу данных при помощи индексов.

10.5.2. Медленные запросы

Запрос 1. Количество посещений и средняя скидка для регулярных пользователей

Предположим, нам необходимо узнать количество визитов и среднюю скидку от рефералов за все время для пользователей, которые заходили чаще всех (чье количество визитов превышает 80% от лидера по посещениям).

Воспользуемся агрегатной функцией и отберем те значения количества посещений, которые превышают порог в 80% от максимального (HAVING с подзапросом).

```
-- Количество посещений и средняя скидка
-- для наиболее частых пользователей,
-- количество визитов которых больше 80%
-- от максимального
SELECT
 u.user_id,
 u.email,
 COUNT(s.session_id) AS sessions_count,
 AVG(r.discount) AS avg_referral_discount
FROM sessions s
```

```
JOIN users u ON s.user_id = u.user_id
LEFT JOIN referrals r ON s.referral_id = r.referral_id
GROUP BY u.user_id, u.email
HAVING COUNT(s.session_id) >= (
 SELECT MAX(sessions_count) * 0.8
 FROM (
  SELECT COUNT(*) AS sessions_count
  FROM sessions
  WHERE user_id IS NOT NULL
  GROUP BY user_id
 ) t
)
ORDER BY sessions_count DESC;
```

Для иллюстрации предлагается статистика времени исполнения данного запроса на локальной машине автора (среднее время за 10 попыток):

Запрос 1 (пользователи, посещения и средняя скидка):

- MS SQL: 4,753 сек.
- MySQL: 26,644 сек.
- PostgreSQL: 2,760 сек.

Эта статистика ни в коем случае не иллюстрирует преимущества или недостатки той или иной СУБД: конфигурации у каждого из установленных на локальную машину серверов баз данных слишком разные, чтобы являться полноценным стендом для объективного сравнения производительности разных СУБД. Время исполнения нам понадобится, чтобы оценить прирост быстродействия с использованием индексов.

ПРИМЕЧАНИЕ.

Чтобы MS SQL SERVER выводила время выполнения запроса, включите настройку:

```
SET STATISTICS TIME ON;
```

Запрос 2. Разные пользователи с одинаковым IP

Специалистам по рекламе важно знать каждую мелочь для повышения конверсии. К сожалению, пользователи не всегда спешат делиться какой-либо информацией о себе, но мы можем узнавать некоторые вещи и по косвенным признакам. Например, если два разных пользователя заходили на сайт с одного и того же IP-адреса, это может свидетельствовать о том, что они являются родственниками и проживают в одном месте. И наоборот — выявить злоупотребления (если, к примеру, за регистрацию дается какой-нибудь подарок).

Воспользуемся CTE, где посчитаем количество зарегистрированных пользователей для каждого IP-адреса, после чего дополнительно выведем в запрос IP-адрес, e-mail посетителя, количество посещений, а также время первого и последнего посещения. Посещения незарегистрированных пользователей игнорируются.

```
-- Поиск пользователей с одинаковым IP
WITH shared_ips AS (
 SELECT
  ip_address,
  COUNT(DISTINCT user_id) AS unique_users_count
 FROM sessions
 WHERE user_id IS NOT NULL
 GROUP BY ip_address
 HAVING COUNT(DISTINCT user_id) > 1
)
SELECT
 s.ip_address,
 u.user_id,
 u.email,
 COUNT(s.session_id) AS sessions_count,
 MIN(s.start_time) AS first_activity,
 MAX(s.start_time) AS last_activity
FROM sessions s
JOIN users u ON s.user_id = u.user_id
JOIN shared_ips si ON s.ip_address = si.ip_address
GROUP BY s.ip_address, u.user_id, u.email
ORDER BY s.ip_address, sessions_count DESC;
```

SKLADCHIK.ORG

Запрос 2 (пользователи с одинаковым IP):

- MS SQL: 7,675 сек.
- MySQL: 146,765 сек.
- PostgreSQL: 38,398 сек.

Анализ рефералов по группам

Рефералы могут генерировать много входящего трафика. Поскольку у каждого из них своя скидка, разобьем на 4 группы (без скидок, скидка до 10%, скидка от 20 до 30% и свыше 20%) и посмотрим, сколько каждая группа привлекла посетителей всего в абсолютном (количество посещений) и относительном (доля в общем количестве) выражении, а также количество зарегистрированных пользователей среди всех привлеченных посетителей за август.

Сначала отберем из основной таблицы сеансов все посещения за август, пришедшие с реферальных ссылок. Затем используем конструкцию CASE WHEN… и разобьем рефералов по группам:

```
CASE
 WHEN r.discount > 20 THEN 1
 WHEN r.discount BETWEEN 10 AND 20 THEN 2
 WHEN r.discount < 10 THEN 3
 ELSE 4
END AS discount_group
```

После этого подготовим табличное выражение, где назначим группу каждому рефералу:

```
WITH august_sessions AS (
  SELECT
    s.referral_id,
    s.user_id,
    r.discount,
    [CASE...] AS discount_group
  FROM sessions s
```

```
  LEFT JOIN referrals r ON s.referral_id = r.referral_id
  WHERE [условия периода и наличие реферальной ссылки]
)
```

В еще одном CTE считаем метрики по группам и выводим результат. Готовый запрос выглядит следующим образом:

```
-- Статистика по реферальным группам
WITH august_sessions AS (
 SELECT
  s.referral_id,
  s.user_id,
  r.discount,
  CASE
   WHEN r.discount > 20 THEN 1
   WHEN r.discount BETWEEN 10 AND 20 THEN 2
   WHEN r.discount < 10 THEN 3
   ELSE 4
  END AS discount_group
 FROM sessions s
 LEFT JOIN referrals r ON s.referral_id = r.referral_id
 WHERE s.start_time BETWEEN '2024-08-01' AND '2024-08-31'
  AND s.referral_id IS NOT NULL
),
group_stats AS (
 SELECT
  discount_group,
  CASE discount_group
   WHEN 1 THEN 'Высокие скидки'
   WHEN 2 THEN 'Средние скидки'
   WHEN 3 THEN 'Низкие скидки'
   ELSE 'Без скидок'
  END AS discount_group_name,
  COUNT(*) AS total_visits,
  COUNT(user_id) AS registered_visits,
  SUM(COUNT(*)) OVER() AS total_all_visits,
  AVG(CASE WHEN discount_group <> 4 THEN discount END) AS avg_discount
 FROM august_sessions
```

```
 GROUP BY discount_group
)
SELECT
 discount_group_name,
 total_visits,
 registered_visits,
 ROUND(total_visits * 100.0 / total_all_visits, 2) AS percentage_of_
total,
 COALESCE(ROUND(avg_discount, 2), 0) AS avg_discount
FROM group_stats
ORDER BY discount_group;
```

3 (статистика посещений по реферальным группам):

- MS SQL: 0,478 сек.
- MySQL: 11,165 сек.
- PostgreSQL: 0,558 сек.

10.5.3. Добавляем индексы

Статистика для всех трех запросов из примера ужасная. Мы имеем дело с относительно небольшим объемом данных на 10 млн строк, в то время как в современном мире приходится иметь дело с объемами данных, которые на несколько порядков превышают объем нашего примера. Некоторым может показаться, что запрос, выполненный менее чем за секунду, — это быстро, но, если представить, что такой запрос выполняется каждый раз, когда кто-то заходит на сайт, а таких посещений может быть несколько тысяч или даже десятков тысяч в секунду, это будет готовым рецептом к потере работоспособности.

Запрос 1 (количество посещений и средняя скидка)

В запросе № 1 производится примерно с такими "трудозатратами":

- Группировка по *user_id* (полное сканирование 10 млн строк).
- Подзапрос в HAVING (вложенное сканирование той же таблицы дважды).

- Вложенные циклы в сравнении между *sessions* и *referrals* (10 млн * 100 = 1 млрд операций сравнения).

Нас интересуют имена пользователя, поэтому, данный столбец – наилучший кандидат на создание индекса. А поскольку у нас еще есть присоединение таблицы рефералов, логично добавить в индекс еще и *referral_id*. Кроме того, мы производим агрегацию по *session_id*, его тоже можно добавить в индекс.

Создание индекса в общем случае выглядит так:

```
CREATE INDEX <имя индекса> ON <таблица> (<столбцы>);
```

Создадим такой индекс:

```
CREATE INDEX idx_sessions_optimizer ON sessions(
user_id,
referral_id,
session_id);
```

Создание индекса может занимать продолжительное время, так как СУБД создает, по сути, еще одну таблицу, в которой будет храниться требуемая информация.

Дождавшись создания индекса, можно сравнить результаты:

Запрос 1	MS SQL	MySQL	PostgreSQL
Было	4,753	26,644	2,760
Стало	1,556	7,144	1,223
Ускорение	**67,25%**	**73,19%**	**55,71%**

В результате добавления индексов размер базы увеличился на 15–20%. Это плата за скорость обработки запросов.

Запрос 2 (общие IP-адреса для разных пользователей)

В табличном выражении *shared_ips* блока видим, что производится сканирование таблицы **sessions** с проверкой условия на столбец *user_id*. Чтобы ускорить этот этап, создадим индекс по двум важным полям: *ip_address* и

user_id. В этом CTE мы также используем агрегатные функции со столбцами *start_time* и *session_id*. Поскольку поиск по этим столбцам не осуществляется (не фигурируют в выражении WHERE), их можно просто добавить (INCLUDE) к индексу. К сожалению, такой прием невозможен для MySQL, и там придется создавать индекс по всем столбцам.

Кроме того, поскольку в дальнейшем мы еще группируем запросы по **e-mail**, мы также можем добавить индекс и на таблицу users.

```
-- Для MySQL:
CREATE INDEX idx_covering_all ON sessions(
      ip_address, user_id, start_time, session_id);

-- Для PostgreSQL и MS SQL Server:
CREATE INDEX idx_covering_all ON sessions(ip_address, user_id)
INCLUDE (start_time, session_id);

-- Индекс для email
CREATE INDEX idx_emails ON users(email);
```

Повторим запрос и сравним статистику до и после.

Запрос 2	**MS SQL**	**MySQL**	**PostgreSQL**
Было	7,675	146,766	38,398
Стало	4,931	38,125	4,990
Ускорение	**35,75%**	**74,02%**	**87,01%**

Запрос 3 (статистика посещений по реферальным группам)

По сравнению с двумя предыдущими запросами этот — относительно быстрый, тем не менее, и здесь есть резервы для оптимизации.

В запросе выполняется фильтрация данных:

- Временной диапазон: `s.start_time BETWEEN …`
- Только реферальные сессии: `s.referral_id IS NOT NULL`.

Затем группировка с агрегацией:

- Категоризация скидок: 1 (>20%), 2 (10-20%), 3 (<10%), 4 (без скидки).
- Агрегация: `COUNT(*), COUNT(user_id), AVG(discount)`.

И вычисления:

- Процент от общего трафика.
- Оконная функция: `SUM(COUNT(*)) OVER()`.

Поскольку применяется фильтрация по временному интервалу и рефералу, данные два столбца идеально подходят для индексирования.

Кроме того, раз в CTE выбирается еще и имя пользователя, этот столбец также можно добавить к индексу (в PostgreSQL и MS SQL достаточно просто его включить в индекс при помощи ключевого слова INCLUDE, а в MySQL данный столбец придется добавить в сам индекс).

Стоит напомнить, что, когда столбец включен в сам индекс (как *start_time*), его значение будет использоваться при поиске и фильтрации. Если столбец просто добавлен (INCLUDE), то индекс не поможет при поиске добавленного значения, но, если потребуется его извлечь для найденной по индексу строки, значение будет "рядом".

```
-- MS SQL Server и PostgreSQL:
CREATE INDEX idx_ref_optimizer ON sessions(start_time, referral_id)
INCLUDE (user_id);

-- MySQL:
CREATE INDEX idx_ref_optimizer ON sessions
  (start_time, referral_id, user_id);
```

Статистика выполнения запроса улучшилась почти десятикратно:

Запрос 3	MS SQL	MySQL	PostgreSQL
Было	0,478	11,166	0,558
Стало	0,047	0,587	0,064
Ускорение	**90,25%**	**94,74%**	**88,57%**

Как можно убедиться, добавление индекса позволяет кратно улучшить производительность запроса.

10.5.4. Особенности синтаксиса

Инструкция CREATE INDEX может быть дополнительно снабжена следующими уточнениями:

- Индекс может быть кластеризованным (CLUSTERED) или нет (NONCLUSTERED):

```
-- Кластеризованный только 1 на таблицу
CREATE CLUSTERED INDEX…
CREATE NONCLUSTERED INDEX…
```

ПРИМЕЧАНИЕ.

В MySQL единственный кластеризованный индекс — это **индекс первичного ключа**. В других СУБД кластеризованный индекс можно создать по любому столбцу, но только один.

- Индекс может быть уникальным (UNIQUE):

```
CREATE UNIQUE INDEX …
```

Мы уже сталкивались с ключевым словом UNIQUE, когда речь шла об ограничениях. Разница между ограничением (CONSTRAINT) и уникальным индексом заключается в том, что если ограничение просто ограничивает вставку дубликатов, то уникальный индекс еще и ускоряет поиск (хотя многие СУБД "под капотом" при создании UNIQUE CONSTRAINT также создают и индекс). Индекс более гибок и в некоторых случаях поддерживает частичную уникальность, например:

```
-- PostgreSQL
-- требовать уникальности электронной почты
-- только для активных пользователей:
CREATE UNIQUE INDEX uq_active_email ON users(email) WHERE is_active;
```

- Индекс может включать другие столбцы (используйте, если в запросе после поиска нужной строки необходимо получить и другие значения):

```
CREATE INDEX … INCLUDE (<столбцы>);
```

ПРИМЕЧАНИЕ.

В MySQL работает с ограничениями только в поздних версиях.

- Индекс может быть фильтрованным, то есть включать только определенные значения (полезно, если делаются частые запросы именно с этим условием):

```
CREATE INDEX … WHERE <условие>;
```

ПРИМЕЧАНИЕ.

В MySQL напрямую не поддерживается.

Можно эмулировать аналогичное поведение через вычислимый столбец (например, указать вычислимое выражение для столбца, которое принимает значение, которое надо индексировать).

- Можно указывать сортировку данных в индексе (по возрастанию ASC / по убыванию DESC):

```
-- Сортировка по убыванию
CREATE INDEX idx_users ON users (id DESC)
```

MS SQL Server — управление заполняемостью страниц

MS SQL позволяет пользователю дополнительно контролировать параметры заполнения страниц индекса FILLFACTOR и PAD-INDEX.

FILLFACTOR — это процент свободного пространства, оставляемый на страницах индекса при его первоначальном создании или перестроении. Значение варьируется от 0 до 100%. При 100% страницы заполняются полностью, свободное пространство отсутствует. Параметр менее 100% оставляет некоторое свободное пространство на странице индекса, позволяя новым строкам вписываться туда, предотвращая фрагментацию и частые расщепления страниц.

Уменьшение FILLFACTOR улучшает производительность для таблиц с высокой нагрузкой на обновление (INSERT, UPDATE) и помогает предотвратить страничное расщепление, возникающее при внесении новых записей или изменении существующих данных. Когда страница полностью занята, новая запись вызывает раскол страницы, что негативно сказывается на производительности и увеличивает фрагментацию.

PAD_INDEX — это дополнительный параметр, который дополняет эффект FILLFACTOR и применяется ко внутренним уровням индекса (то есть страницам верхнего уровня индекса). Аналогично FILLFACTOR, PAD_INDEX уменьшает вероятность расщеплений страниц в верхней части индекса, уменьшая фрагментированность индекса и поддерживая высокую производительность. Особенно полезен для больших индексов с большим объемом модификаций данных.

Выбор значений зависит от частоты внесения изменений в таблицу:

- **Высокая частота изменений** (много INSERT, UPDATE) — используйте FILLFACTOR около 70–80% и PAD_INDEX = ON.
- **Редко меняющаяся таблица** — оставьте FILLFACTOR равным 100%.
- **Большие объемы данных и высокие темпы роста** — поэкспериментируйте со снижением FILLFACTOR ниже стандартных рекомендаций (до 60%).

Важно помнить, что чрезмерно низкое значение FILLFACTOR приведет к увеличению размера индекса и снижению производительности для операций чтения.

Синтаксис:

```
CREATE INDEX IX_OrderByCustomer ON Orders(CustomerID)
WITH (FILLFACTOR = 80, PAD_INDEX = ON);
```

MySQL — префиксные индексы

При работе с текстовыми данными стандартные индексы работают идеально, когда индексируются небольшие числовые или короткие текстовые поля. Однако при попытке индексировать большие объемы текста возникают две основные проблемы:

1. **Рост объема индекса**. Индекс большого текстового столбца займет много места на диске и в оперативной памяти.
2. **Медленная обработка запросов**. Из-за большего размера индекс медленнее обрабатывает запросы, увеличиваются накладные расходы на поддержание индекса.

Допустим, у вас есть таблица **books**, в которой столбец *title* имеет тип VARCHAR(255) и хранит названия книг длиной до 255 символов. Индексирование полного заголовка может оказаться избыточным, если многие заголовки начинаются одинаково. Вместо этого можно создать префиксный индекс, охватывающий только первую часть строки:

```
-- Индексируем первые 10 символов:
CREATE INDEX idx_books_title_prefix ON books(title(10));
```

Префиксные индексы имеют ряд ограничений:

- **Невозможность точного сопоставления всей строки**. Запросы, зависящие от полной длины строки, могут не использовать префиксный индекс.

SKLADCHIK.ORG

- **Потеря точности поиска**. Иногда бывают ситуации, когда одинаковые начальные символы встречаются у разных строк, и точность поиска снижается.

К выбору длины префикса следует подходить осторожно, учитывая требования конкретного проекта.

MySQL, PostgreSQL — выбор типа индекса

При помощи ключевого слова USING можно указать тип индекса, отличный от стандартного B-Tree.

MySQL в виде альтернативы предлагает только HASH, в то время как PostgreSQL предлагает гораздо больший выбор (HASH, GIN, GIST, BRIN).

Синтаксис:

```
CREATE INDEX …. USING HASH;
```

Построение индекса без блокировки таблицы в PostgreSQL

При построении индекса большинство СУБД блокируют таблицу, соответственно, все запросы к этой таблицы от всех пользователей должны будут ожидать завершения операции или прекращаться по истечении времени ожидания.

Если таблица активно используется приложением и остановка работы недопустима, необходимо минимизировать негативное воздействие на производительность при создании индекса. В PostgreSQL при создании индекса можно использовать ключевое слово CONCURRENTLY (параллельно, асинхронно):

```
CREATE INDEX CONCURRENTLY…
```

Процесс создания индекса таким способом занимает больше времени, чем обычное синхронное создание, и, несмотря на минимальные блокировки, возможна небольшая деградация производительности в периоды пиковой активности.

Альтернативы для других СУБД

Если создание индекса не терпит задержек, но использование CONCURRENTLY нежелательно или невозможно (если не используется PostgreSQL), можно сделать копию таблицы, построить индекс на копии, а затем заменить старую таблицу новой.

10.5.5. Фрагментация индексов и их перестройка

При частых операциях INSERT/UPDATE/DELETE в таблице физический порядок хранения данных в страницах индекса нарушается. При удалении или обновлении строк освобождается место — в индексируемых данных появляются пропуски. Новые данные будут записываться на свободные места, что не гарантирует их последовательное расположение. Такое явление называется **фрагментацией** (дроблением).

Чтение фрагментированного индекса требует больше операций ввода-вывода из-за необходимости обращаться по разным адресам физического хранилища, что замедляет работу СУБД, кроме того, фрагментированный индекс из-за того, что его страницы заполнены не полностью, занимает больше места.

Оценка фрагментации индекса является важным шагом в поддержании производительности базы данных. И каждая из систем управления базами данных предлагает свои способы мониторинга и анализа фрагментации.

РЕКОМЕНДАЦИЯ.

Не боритесь с фрагментацией везде. Анализируйте наиболее часто используемые (особенно в диапазонных сканированиях) и крупные индексы.

Как посмотреть, насколько индекс фрагментирован?

MS SQL Server

В Microsoft SQL Server оценка фрагментации индексов осуществляется с помощью функции **sys.dm_db_index_physical_stats**, которая позволяет получить информацию о состоянии индекса. Эта процедура возвращает детальную статистику физической структуры индекса.

Ключевыми показателями в результате, который возвращает данная функция, являются:

- **avg_fragmentation_in_percent** — главный показатель, процент логической фрагментации (нарушение порядка страниц в связанном списке индекса). Именно она критична для производительности сканирования диапазонов.
- **avg_page_space_used_in_percent** — процент заполненности страниц (фрагментация внутри страниц). Низкое значение указывает на пустое место из-за удалений/обновлений.
- **page_count, fragment_count** — общее число страниц и фрагментов (физически несмежных групп страниц) индекса. Большое количество фрагментов при малом количестве страниц — признак сильной фрагментации.

Пример запроса может выглядеть следующим образом:

```
SELECT
  OBJECT_NAME(ips.object_id) AS TableName,
  i.name AS IndexName,
  ips.index_type_desc AS IndexType,
  ips.avg_fragmentation_in_percent AS Fragmentation,
  ips.page_count AS PageCount,
  ips.fragment_count AS FragmentCount
FROM
  -- Вместо 'DETAILED' можно
  -- указать'SAMPLED' или 'LIMITED' для производительности
  sys.dm_db_index_physical_stats
   (DB_ID(), NULL, NULL, NULL, 'DETAILED') ips
JOIN
  sys.indexes i ON ips.object_id = i.object_id
```

```
  AND ips.index_id = i.index_id
WHERE
  -- Вместо 0 можно указать требуемый порог,
  -- например > 5% или > 30%
  ips.avg_fragmentation_in_percent > 0
  -- Игнорировать очень маленькие индексы (страницы)
  AND ips.page_count > 8
  -- Игнорировать системные объекты
  AND OBJECT_NAME(ips.object_id) IS NOT NULL;
```

При интерпретации результатов данного запроса следуйте следующей линии рассуждений.

Если логическая фрагментация (avg_fragmentation_in_percent) в пределах 0% до 5%, то фрагментация незначительна и никаких действий обычно не требуется.

Если данный показатель находится в пределах от 5% до 30%, то рекомендуется реорганизация индекса:

```
ALTER INDEX <имя_индекса> ON <таблица> REORGANIZE;
```

При высокой фрагментации (≥ 30%) рекомендуется перестроить индекс (более длительная и ресурсоемкая операция):

```
ALTER INDEX <имя_индекса> ON <таблица> REBUILD;
```

Высокая фрагментация страниц (avg_page_space_used_in_percent) указывает на неэффективное использование пространства. Для решения этой проблемы также рекомендуется выполнить перестройку (REBUILD).

MySQL

В MySQL, к сожалению, отсутствуют детальные метрики, явно показывающие степень фрагментации индекса.

Приблизительно такую оценку можно получить:

```
SELECT
  DATA_LENGTH AS Data_Length,
  INDEX_LENGTH AS Index_Length,
  DATA_FREE AS Data_Free,
  ROUND((DATA_FREE / (DATA_LENGTH + INDEX_LENGTH)) * 100, 2)
    AS Est_Fragmentation_Percent
FROM information_schema.TABLES
WHERE TABLE_NAME = 'имя_таблицы' AND TABLE_SCHEMA = DATABASE();
```

Здесь производится запрос к внутренней системной таблице information_schema, где мы по заданной таблице получаем отношение свободного места, выделенного под данную таблицу, но пока не использованного, к общему размеру данных, выделенного под хранение данной таблицы.

Если показатель ожидаемого уровня фрагментации (Est_Fragmentation_Percent в данном примере) превышает 20–30%, это может свидетельствовать о высоком уровне фрагментации индексов.

К сожалению, это очень грубая оценка, так как высокий размер свободного места не обязательно свидетельствует о высокой фрагментации. Вместо этого пользователям MySQL рекомендуется на регулярной основе выполнять запрос:

```
OPTIMIZE TABLE имя_таблицы;
```

ВНИМАНИЕ!

Данная команда блокирует таблицу на время выполнения.

PostgreSQL

В обычной конфигурации PosrgreSQL также не располагает средствами мониторинга метрик в разрезе отдельного индекса, однако при использовании расширения **pgstattuple** можно получить расширенную статистику по каждой из таблиц.

Чтобы воспользоваться расширением, его необходимо сначала установить при помощи следующего запроса:

```
CREATE EXTENSION IF NOT EXISTS pgstattuple;
```

После этого мы можем получить расширенную статистику по таблице при помощи следующей команды:

```
SELECT * FROM pgstattuple('имя_таблицы');
```

Результаты данного запроса интерпретируются следующим образом:

- **free_percent** — процент свободного пространства на страницах (аналог внутренней фрагментации страниц).
- **tuple_percent** — процент пространства, занятого актуальными данными.
- **dead_tuple_percent** — процент пространства, занятого "мертвыми" (удаленными или устаревшими) строками. Высокий показатель — явный сигнал к оптимизации.
- **table_len** — общий размер таблицы.
- **tuple_count** — количество "живых" строк.

PostgreSQL также позволяет получить детальную статистику по каждому индексу при помощи команды:

```
SELECT * FROM pgstatindex('имя_индекса');
```

В контексте оценки степени фрагментации у результата этого запроса важны два показателя:

- **avg_leaf_density** — средняя плотность заполнения листовых страниц (в %). Низкое значение (< 60–70%) указывает на сильную внутреннюю фрагментацию страниц индекса.
- **leaf_fragmentation** — процент фрагментации листовых страниц (логическая фрагментация). Показывает, насколько физический порядок страниц индекса отклоняется от идеального логического порядка. Если это значение больше 0%, последовательность нарушена.

Как и в других СУБД, в PostgreSQL имеется средство для обслуживания "здоровья" индексов и переиндексации. Для оптимизации индексов и сокращении занимаемого места используется команда VACUUM (пылесос).

```
VACUUM [VERBOSE] [ANALYZE] <имя_таблицы>;
```

Опция ANALYZE (ее указывать не обязательно), помимо собственно чистки обновит статистику для планировщика запросов, а опция VERBOSE выведет расширенную информацию при выполнении команды.

Данную операцию рекомендуется выполнять на регулярной основе в профилактических целях.

При сильной фрагментации (когда pgstattuple показывает >30% мертвых кортежей) или перед резервным копированием для уменьшения размера базы рекомендуется выполнить операцию полной перезаписи таблицы:

```
VACUUM FULL [VERBOSE] <имя_таблицы>;
```

ВНИМАНИЕ!

Данная операция блокирует таблицу!

Помимо VACUUM, в PostgreSQL также имеется команда REINDEX (переиндексирование), ее прелесть заключается в том, что ее можно применить как к отдельному индексу, так и к таблице или базе целиком:

```
-- Переиндексация для индекса
REINDEX INDEX имя_индекса;
-- для таблицы
REINDEX TABLE имя_таблицы;
-- для базы данных:
REINDEX DATABASE имя_базы_данных;
```

10.5.6. Удаление и модификация индексов

Для операций по удалению индексов для таблицы используется команда DROP INDEX…, а для модификации — ALTER INDEX…:

```
-- Удаление индексов
-- MS SQL и MySQL
DROP INDEX <имя_индекса> ON <имя_таблицы>;

-- PostgreSQL
DROP INDEX <имя_индекса>;
```

Для PostgreSQL и MS SQL возможна также опция DROP INDEX IF EXISTS... (Удалить индекс, если существует...). У PostgreSQL также можно указать опцию CONCURRENTLY (Выполнить параллельно, без блокировок).

При этом возможности для изменения индексов сильно ограничены. В большинстве случаев при необходимости поменять индекс его нужно пересоздать (удалить существующий и создать новый с новыми параметрами).

При этом в разных СУБД все же можно использовать ALTER INDEX в ряде случаев.

Так, для MS SQL Server существует возможность изменить параметры хранения (FILLFACTOR, PAD_INDEX), а для PostgreSQL — переименовать:

```
ALTER INDEX index_name RENAME TO new_name;
```

В MySQL индекс требует пересоздания во всех случаях. Кроме того, в этой СУБД индексы автоматически создаются для внешних ключей. Это делается для поддержания целостности ссылок между таблицами. Когда внешний ключ создается, MySQL требует соответствующего индекса, чтобы обеспечить быструю проверку условий целостности при вставке, обновлении или удалении данных. Таким образом, если вы создаете внешний ключ, MySQL автоматически создает индекс на соответствующем поле, которое ссылается на первичный ключ или уникальный индекс другой таблицы.

10.6. Анализ планов выполнения

10.6.1. Планировщик запросов

В предыдущих разделах этой главы мы анализировали запрос, который написали, и "на глаз" прикидывали план его выполнения. Хотя в отдельных простых случаях этого бывает достаточно, при анализе сложного запроса и тем более — при его оптимизации — нашей интуиции оказывается недостаточно.

Прежде чем выполнить запрос, любая СУБД задействует планировщик запросов (оптимизатор запросов). Этот компонент СУБД отвечает за то, как

именно будет выполняться запрос. Сначала он анализирует структуру запроса и статистику данных (размер таблиц, селективность столбцов, наличие индексов) и генерирует план выполнения — последовательность операций для извлечения результата с минимальными затратами ресурсов. Этот план определяет порядок чтения таблиц, типы их соединений (JOIN), какие индексы будут использованы, оценивается также необходимость сортировки или агрегации.

Оптимизатор оценивает "стоимость" (**cost**) каждого варианта (в условных единицах) и выбирает оптимальный. Каждый производитель СУБД реализует свою собственную концепцию оценки стоимости. Несмотря на значительные различия в реализациях, большинство современных реляционных баз данных придерживаются похожих принципов. Стоимость отражает ресурсные затраты на исполнение разных частей плана, такие как:

- Время выполнения.
- Потребляемая оперативная память.
- Число операций ввода-вывода (чтение и запись).
- Затраты на соединение таблиц и сортировку данных.

Основная задача — подобрать план, который обеспечит наименьшую стоимость выполнения.

10.6.2. Получение плана выполнения

Для каждой из СУБД предусмотрен свой способ анализировать план выполнения запроса.

MS SQL Server

В MS SQL Server наиболее удобным средством для анализа плана выполнения является его графическое представление в виде дерева операций, которое можно получить в оболочке SQL Server Management Studio (SSMS). Для его отображения достаточно в окне ввода запроса нажать CTRL + L (либо правым щелчком мыши активировать вплывающее меню и выбрать пункт **Показать ожидаемый план выполнения** (Show Estimated Execution

Plan). Все операции, которые должна будет выполнить СУБД для обработки такого запроса, будут отображены в виде графа.

Отрисовка этого графа производится по сгенерированным планировщиком XML данным, в каждом узле которых указывается операция для каждого из этапов выполнения запроса.

Вывод этих данных можно запросить вручную, если предварительно включить настройку SHOWPLAN_XML:

```
SET SHOWPLAN_XML ON;
GO
```

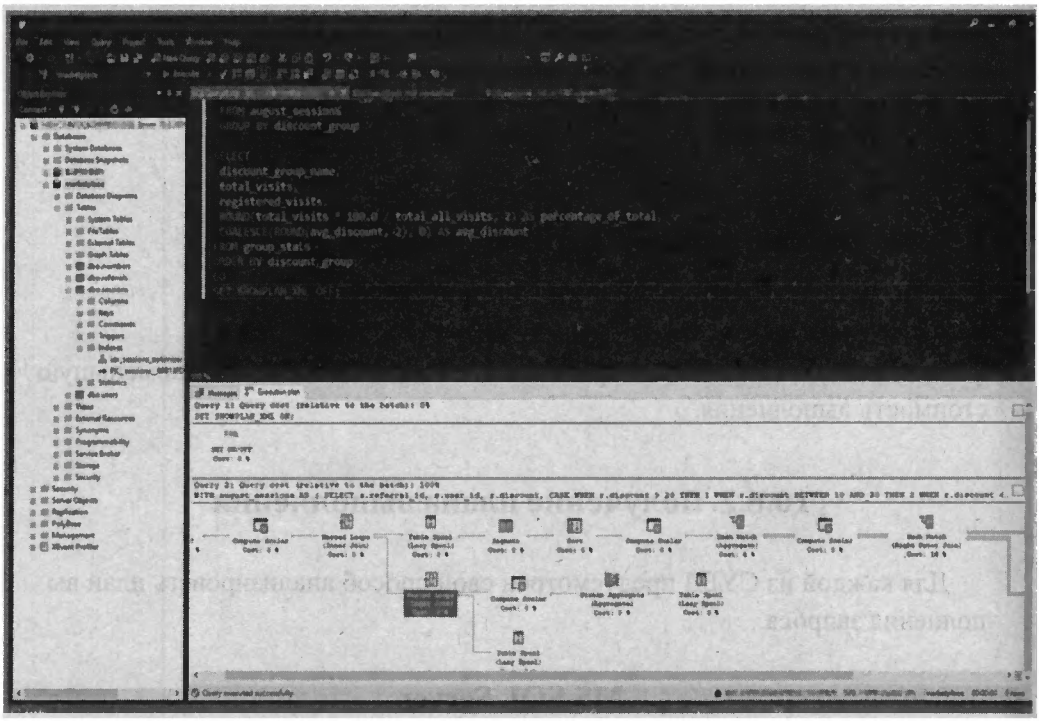


Рис. 10.3. *Графический план выполнения запроса в SSMS*

Если вы не пользуетесь по каким-то причинам SSMS (что, на субъективный взгляд автора, является лучшей графической оболочкой для SQL Server), можно использовать сторонние утилиты либо попросить СУБД построить текстовую версию. Для этого необходимо указать соответствующую настройку:

SKLADCHIK.ORG


```
SET SHOWPLAN_TEXT ON;
GO
-- Текст вашего запроса
SELECT …
GO
-- Выключение вывода плана выполнения
SET SHOWPLAN_TEXT OFF;
```

При необходимости можно запросить расширенную информацию:

```
SET SHOWPLAN_ALL ON;
-- Текст вашего запроса
GO
SET SHOWPLAN_ALL OFF;
```

Эта команда выводит детальную информацию о плане выполнения, включая оценки стоимости операций, количества обработанных строк и многое другое.

Для сбора фактических метрик выполнения запроса в MS SQL можно включать и отключать настройки для вывода разнообразной статистики:

```
SET STATISTICS <метрика> ON - включение
GO
-- Ваш запрос
SET STATISTICS <метрика> ON -- выключение
GO
```

Доступны следующие виды метрик (все они по умолчанию выключены и требуют явного включения (ON) перед выполнением запроса:

```
SET STATISTICS TIME ON
```

Включает вывод сведений о затраченном процессорном времени и общем времени выполнения запроса. Мы уже пользовались данной метрикой для оценки времени выполнения запроса.

```
SET STATISTICS IO ON
```

Управляет выводом сведений о вводе-выводе, связанных с запросом. Из-за сравнительной "дороговизны" операций ввода-вывода позволяет оценить возможные проблемы в производительности.

```
SET STATISTICS PROFILE ON
```

Выводит фактический план выполнения запроса с дополнительными деталями, такими как количество затронутых строк, стоимость каждого из этапов и методы доступа к данным.

```
SET STATISTICS XML ON
```

Предоставляет подробный XML-файл с фактическим планом выполнения запроса.

```
SET STATISTICS ROWCOUNT ON
```

Контролирует вывод количества строк, возвращенных каждым запросом или инструкцией.

```
SET STATISTICS SHOWPLAN_ALL
```

Аналогичен команде SET STATISTICS PROFILE, но предоставляет дополнительный уровень детализации и показывает полную внутреннюю работу оптимизатора запросов.

MySQL

В MySQL для получения ожидаемого плана выполнения запроса используется команда EXPLAIN (синоним DESCRIBE):

```
EXPLAIN
-- Текст запроса
```

В результате получится таблица со следующими полями:

Название столбца	Пояснение
id	Порядок выполнения операции
select_type	Тип выборки по таблице: • PRIMARY — главный запрос. • DERIVED — временная таблица, созданная из подзапроса (в FROM) Другие возможные значения: SIMPLE, SUBQUERY, UNION и др.
table	Название или псевдоним таблицы
type	**Критически важный столбец!** Показывает тип доступа к данным (от лучшего к худшему): eq_ref — уникальный поиск по PK/FK. range — диапазонный поиск по индексу. ALL — полное сканирование таблицы
possible_keys	Индексы, которые могли бы быть использованы при запросе
key	Фактически выбранный оптимизатором индекс
ref	Что сравнивается с индексом (NULL – нет сравнения)
rows	Предполагаемое количество проверяемых строк
filtered	Процент строк, прошедших фильтрацию после индекса
extra	Дополнительная информация (**тоже важный столбец!**): • **Using index** — данные берутся только из индекса ("покрывающий индекс"). • **Using where** — фильтрация после индекса. • **Using temporary** — создана временная таблица (возможно, есть место для оптимизации). • **Using filesort** — дополнительная сортировка (требует оптимизации)

В качестве альтернативы данной таблице можно воспользоваться форматом вывода древовидной структуры:

```
EXPLAIN FORMAT=TREE <текст_запроса>;
```

или в виде машиночитаемой JSON-структуры для последующей визуализации:

```
EXPLAIN FORMAT=JSON <текст_запроса>;
```

Чтобы получить фактический план исполнения, необходимо воспользоваться командой EXPLAIN ANALYZE:

```
EXPLAIN ANALYZE <текст_запроса>
```

Вместо простого плана выполнения ANALYZE исполняет сам запрос, замеряя точное поведение системы на практике. Это значит, что, помимо предполагаемых показателей, вы получите точные данные о реальном исполнении, такие как:

- время выполнения каждого шага;
- точные объемы прочитанных данных;
- фактическое количество строк, полученных на каждом этапе;
- подробности использования индексов и блокировки;
- уровень параллельной обработки (если применимо).

PostgreSQL

Как и в MySQL, в PostgresSQL для получения плана выполнения запроса используется команда EXPLAIN, а для фактического анализа выполнения запроса — EXPLAIN ANALYZE.

По умолчанию, в отличие от MySQL, для вывода используется древовидная структура при выводе плана исполнения.

10.6.3. Как читать план выполнения?

Несмотря на кажущуюся сложность, план выполнения запроса предоставляет пользователю очень мощный аналитический инструмент, который полезен при работе с хоть сколько-нибудь большим объемом данных и критически важен при работе на высоконагруженных системах, обслуживающих сотни или даже тысячи запросов в секунду. Умение читать план позволяет лучше понимать, почему же "тормозит" запрос и что нужно сделать, чтобы

это исправить (возможные действия не ограничиваются только созданием нужного индекса, возможно, анализ выявит проблемы с самой структурой таблиц и связей, которую, возможно, будет эффективнее реорганизовать).

> **План выполнения** — это всего лишь последовательность операций, которые СУБД будет выполнять при обработке запроса. Результат выполнения одной операции становится исходными данными для выполнения следующей. Современные СУБД используют адаптивный подход к выполнению запроса и могут изменять план выполнения "на лету", поэтому очень важно тестировать именно фактическое выполнение запроса.

Также стоит отметить, что MS SQL и особенно PostgreSQL активно используют параллелизм при исполнении, поэтому выполнение одной операции может разбиваться на несколько параллельных потоков.

Операции разделяются на несколько основных категорий. Ниже приведены описания к основным видам.

Операции доступа к данным

Если требуется получить данные из таблицы, СУБД определяет, как именно будет осуществляться доступ:

- **Table scan / Seq Scan** — полное последовательное чтение всех строк таблицы (MySQL: ALL в столбце *type* результата EXPLAIN). Самый "дорогостоящий" сценарий и почти всегда требует создания индекса по столбцам, участвующим в запросе.
- **Index Scan** — полное чтение всех записей индекса. В большинстве случаев — быстрее, чем предыдущий вариант, из-за меньшего объема сканируемых данных, хранения в оптимизированном формате, физической упорядоченности страниц индекса и кеширования. Кроме того, индексы очень часто отсортированы. Индекс считается **покрывающим**, если он включает в себя все столбцы запроса и обращение к таблице не требуется вовсе.
- **Index Seek** — точечный доступ к данным через индекс (для условий =, >, < и т.д.). СУБД находит искомые строки в индексе очень быстро, так как пропускает лишние части индекса и концентрируется только на нужном

диапазоне данных. Чем точнее заданы критерии поиска, тем эффективнее работает данная операция.

- **Key Lookup / CTID** — дополнительный поиск в таблице после индекса (если индекс не покрывающий).

Операции соединения (JOIN)

Ввиду того, что сложность (стоимость) таких операций может колебаться в очень широких пределах, при анализе плана выполнения стоит в первую очередь обращать внимание именно на них. Так, при соединении двух таблиц, в одной из которых *n* строк, а в другой — *m*, СУБД может выполнить m × n операций сравнений, а в случае оптимизации — только *n* (или только m), что дает ощутимую экономию во времени и ресурсах при должной оптимизации.

- **Nested loop** (*присоединение вложенным циклом*). Для каждой строки **таблицы А** ищется совпадение в **таблице Б**. На больших таблицах А и Б подобная операция может очень существенно замедлить запрос. Ситуация, когда применяется присоединение вложенным циклом, в принципе, допустима на таблицах малого размера либо при низком уровне селективности (при малом количестве совпадающих строк).

ПРИМЕЧАНИЕ.

В MySQL данный вид операции в EXPLAIN соответствует значению ALL в столбце *type*.

- **Hash Join / Hash match** (*хеш-присоединение*). Сначала для таблицы меньшего размера в оперативной памяти строится хеш-таблица. Затем начинается перебор значений из второй таблицы, для каждой строки которой также вычисляется хеш, после чего СУБД проверяет, имеется ли такое же значение в созданной ранее хеш-таблице. Сложность проверки наличия значения в хеш-таблице стремится к O(1) (лучше уже и быть не может), к тому же современные СУБД выполняют данную операцию параллельно в несколько потоков, поэтому такой вид присоединения проводится гораздо быстрее (даже несмотря на "накладные расходы" в виде необходимости рассчитывать хеш-значения). Тем не менее, если объем данных достаточно большой и не позволяет эффективно разместить хеш-

таблицу в памяти, такой вид присоединения может быть медленнее. Всегда тестируйте фактический план выполнения.

- **Merge Join** (*присоединение слиянием*). Данный метод работает очень быстро, выполняя один проход по обеим таблицам сразу, но применим, только если обе таблицы предварительно отсортированы по ключу объединения. В некоторых случаях стоит рассмотреть сортировку (ORDER BY) перед JOIN и проверить, будет ли в плане выполнения выбран именно этот вид ("стоимость" сортировки может оказаться слишком высокой, и оптимизатор запросов выберет иной тип).

ПРИМЕЧАНИЕ.

В MySQL — ref/range.

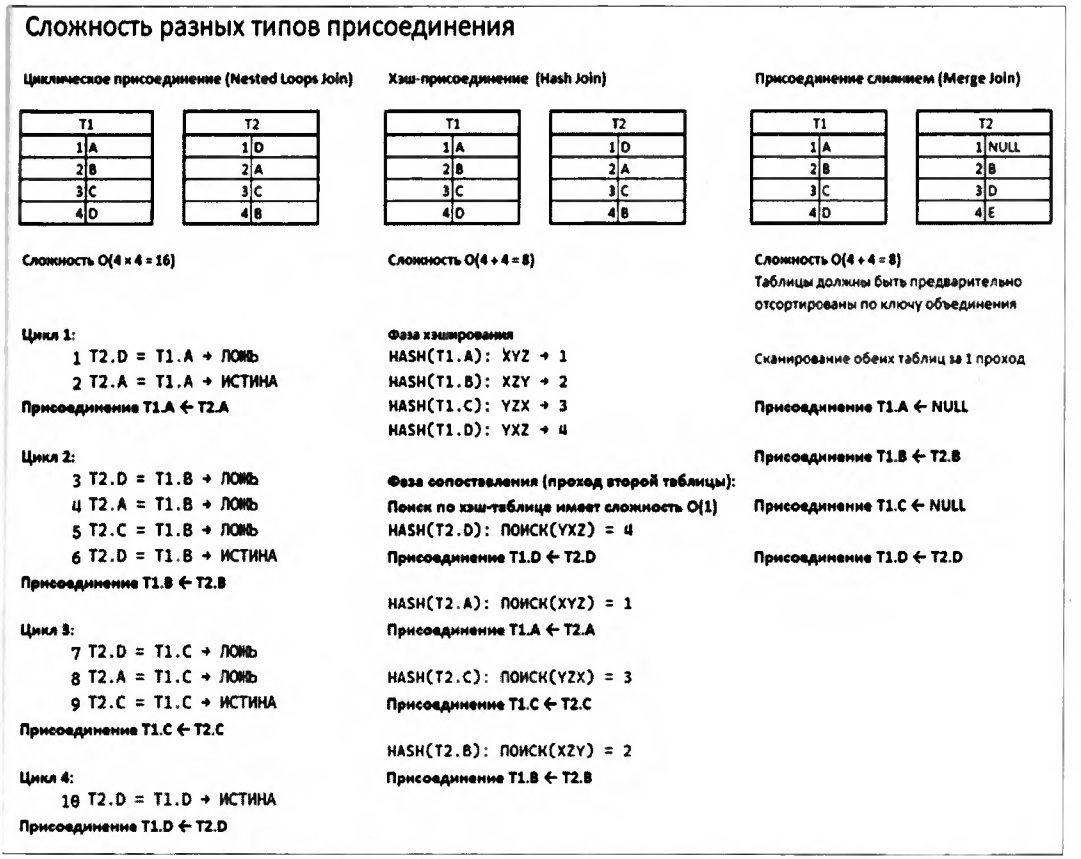


Рис. 10.4. *Сложность разных видов присоединений*

Операции сортировки и агрегации

Данные операции выделяют следующие основные типы:

- **Sort** (*сортировка*). Во многих случаях на один запрос не должно быть более одной сортировки (исключение составляют оконные функции или намеренная сортировка для обеспечения Merge Join).

ПРИМЕЧАНИЕ.

В MySQL этому типу операции соответствует значение **using filesort** в столбце *extra*.

- **Hash Aggregate** (*хеш-агрегация*). Агрегация данных с использованием хеш-таблицы (GROUP BY, DISTINCT). В MySQL соответствует значению **using temporary** в столбце *extra*.
- **Stream Aggregate / GroupAggregate** (*поточная агрегация*). Агрегация с предварительной сортировкой данных.

10.7. Оптимизация запросов с помощью индексов на практике

Для закрепления материала давайте попробуем прочитать реальный план исполнения для запроса, ищущего пользователей с одинаковым IP, который мы разбирали ранее. Ниже приведен его текст для удобства:

```
-- Поиск пользователей с одинаковым IP
WITH shared_ips AS (
 SELECT
  ip_address,
  COUNT(DISTINCT user_id) AS unique_users_count
 FROM sessions
 WHERE user_id IS NOT NULL
 GROUP BY ip_address
```

```
 HAVING COUNT(DISTINCT user_id) > 1
)
SELECT
 s.ip_address,
 u.user_id,
 u.email,
 COUNT(s.session_id) AS sessions_count,
 MIN(s.start_time) AS first_activity,
 MAX(s.start_time) AS last_activity
FROM sessions s
JOIN users u ON s.user_id = u.user_id
JOIN shared_ips si ON s.ip_address = si.ip_address
GROUP BY s.ip_address, u.user_id, u.email
ORDER BY s.ip_address, sessions_count DESC;
```

Для начала удалите существующие индексы для таблицы **sessions** при помощи инструкции DROP INDEX…, чтобы просмотреть план исполнения до и после создания индекса.

Получите план исполнения данного запроса без индекса (в зависимости от вашей конфигурации и версии СУБД данные будут отличаться от приведенных здесь).

10.7.1. MS SQL Server

ПРИМЕЧАНИЕ.

Оболочка SSMS, помимо того, что визуализирует план исполнения в виде графа, еще и самостоятельно предлагает создать недостающий индекс (Missing index). Данная информация выводится в верхней части окна с планом исполнения.

Итак, время выполнения запроса — 9219 мс.

Анализ плана исполнения ведется от конца к началу:

```
|--Filter(WHERE:([Expr1004]>(1)))
 |--Compute Scalar(…)
  |--Hash Match(Aggregate, HASH:(…[sessions].[ip_address]),…( COUNT(*))
   |--Hash Match(Aggregate,
      HASH:(…[sessions].[user_id],….[sessions].[ip_address]), ...)
    |--Clustered Index Scan(…[sessions].[PK…]),
          WHERE:([user_id] IS NOT NULL))
```

1-й этап. Подготовка CTE

- **Clustered Index Scan** (полное сканирование таблицы sessions).

  Считывает все строки таблицы (**дорого**!). Фильтр user_id IS NOT NULL применяется во время сканирования.

- Двойная **Hash Match (Aggregate)**:
  - » Первый этап: группирует по (user_id, ip_address) для получения уникальных пар (эмулирует COUNT(DISTINCT user_id)).
  - » Второй этап: группирует по ip_address и считает количество уникальных user_id.
- **Возможные проблемы**: хеш-агрегация требует много памяти и CPU. При больших данных возможно, что будет задействован дисковый накопитель (spill — *пролив*). Таких случаях надо смотреть фактический план, оценить количество операций ввода-вывода (I/O).
- **Compute scalar** — дешевая операция, ею можно пренебречь.
- **Filter** — оставляет только IP с >1 пользователем (Expr1004 > 1).

2-й этап. Соединение с основной таблицей

```
--Hash Match(Inner Join, HASH:([sessions].[ip_address])=([s].[ip_
address]))
     |--[...]  <!-- Результат CTE shared_ips -->
     |--Clustered Index Scan(OBJECT:([sessions] AS [s]))
```

Опять читаем снизу вверх:

SKLADCHIK.ORG

- **Clustered Index Scan** таблицы **sessions** (**второе полное сканирование!**) — повторное чтение всех данных таблицы для соединения по ip_address.
- **Hash Match Join** — строит хеш-таблицу по результату CTE и соединяет с отсканированной **sessions**. Ресурсоемко при большом количестве IP.

3-й этап. Соединение с таблицей *users*

```
|--Hash Match(Inner Join, HASH:([u].[user_id])=([s].[user_id]), ...)
     |--Clustered Index Scan(OBJECT:([users].[PK__users__...] AS [u]))
```

- **Clustered Index Scan** таблицы users — полное сканирование (почти всегда проблема, если таблица большая).
- **Hash Match Join** — соединяет по user_id. Эффективно, если **users** небольшая.

Финальный этап. Агрегация и сортировка

```
|--Sort(ORDER BY:([s].[user_id] ASC, [s].[ip_address] ASC)
     |--Stream Aggregate(GROUP BY:([u].[user_id], [s].[ip_address])
DEFINE:([Expr1015]=Count(*), ...)
          |--Compute Scalar(...)
               |--Sort(ORDER BY:([s].[ip_address] ASC, [Expr1005] DESC)
```

- **Stream Aggregate** — группирует по user_id, ip_address, считает COUNT(session_id), MIN/MAX(start_time). Требует сортировки данных перед агрегацией.

  Здесь мы имеем две сортировки (Sort): первая — для группировки, вторая — для финального ORDER BY.

Итого в плане имеются следующие проблемы:

1. Два полных сканирования **sessions** (учитывая, что в ней 10 млн записей — это основная проблема).
2. Ресурсоемкая агрегация для CTE — два этапа Hash Match.
3. Полное сканирование таблицы **users**.

4. Сортировки перед группировкой из-за отсутствия требуемого порядка данных.

Для операций поиска в таблице **sessions** используются столбцы *ip_address*, *user_id*, кроме того, нам требуются сведения *start_time* и *session_id* (но эти два поля не используются при поиске).

Оптимальным покрывающим индексом для таблицы **sessions** станет:

```
CREATE INDEX idx_covering_index
ON sessions(ip_address, user_id)
INCLUDE (start_time, session_id);
```

Поскольку в плане присутствует еще и полное сканирование таблицы **users**, есть смысл добавить индекс и на нее:

```
CREATE INDEX idx_emails ON users(email);
```

После добавления индексов время выполнения стало 4948 мс (было 9219 мс).

CTE

```
|--Filter(WHERE:([Expr1004]>(1)))
|     |--Compute Scalar(…
|        |--Stream Aggregate(…Count(*))
|              |--Stream Aggregate(GROUP BY..)
|                    |--Index Scan(…[idx_covering_index])…) ORDERED
FORWARD)
```

Было: Hash Match, стало: Stream Aggregate (агрегация в потоке).

Было: Clustered Scan, стало: Index Scan (чтение только индекса).

ORDERED FORWARD (данные сразу в нужном порядке).

Присоединение CTE

```
|--Merge Join(Inner Join, MERGE:([ip_address]))
     |--[...]  <!-- CTE -->
     |--Index Scan(idx_covering_index AS s) ORDERED FORWARD
```

Было: Hash Join, стало: Merge Join (более эффективный алгоритм, данные уже отсортированы).

Было: Clustered Scan, стало: Index Scan (чтение только индекса).

ORDERED FORWARD (данные уже отсортированы по ip_address).

Присоединение *users*

```
|--Hash Match(Join)
     |--Index Scan(idx_emails)
```

Было: Clustered Scan, стало: Index Scan (чтение только индекса).

Финальный этап

Сортировка на выходе осталась, то теперь сортируется меньше данных (результат Merge Join вместо полной таблицы).

10.7.2. MySQL

До формирования индексов время выполнения запроса составляло 109359 мс.

Этап 1. Обработка CTE

Проблема: отсутствие индекса для группировки и фильтрации.

```
-> Filter: (sessions.user_id is not null) (actual time=0.354..4468
rows=2.5e+6 loops=1)
    -> Table scan on sessions (cost=1e+6 rows=9.72e+6) (actual
time=0.35..4008 rows=10e+6 loops=1)
-> Sort: sessions.ip_address (actual time=6941..7099 rows=2.5e+6
loops=1)
-> Group aggregate: count(distinct sessions.user_id) (actual
time=6941..8050 rows=1.52e+6 loops=1)
-> Filter: (count(distinct sessions.user_id) > 1) (actual
time=6941..8167 rows=638968 loops=1)
-> Materialize CTE shared_ips (actual time=10202..10202 rows=638968
loops=1)
```

- Полное сканирование таблицы **sessions** (10 млн строк).
- Сортировка 2.5M строк по ip_address (заняла 3.1 сек.).
- Ресурсоемкая агрегация с COUNT(DISTINCT user_id).
- Высокое время обработки: 8.1 сек. для 1.52 млн групп.

Этап 2. Присоединение CTE

Проблема: отсутствие индекса для соединения по ip_address.

```
-> Nested loop inner join (cost=4.41e+6 rows=9.72e+6) (actual
time=1.63..13200 rows=2.5e+6 loops=1)
    -> Filter: (s.user_id is not null) (actual time=1.61..6935
rows=2.5e+6 loops=1)
        -> Table scan on s (actual time=1.6..6313 rows=10e+6
loops=1)
    -> Single-row index lookup on u using PRIMARY (actual
time=0.00228..0.00231 rows=1 loops=2.5e+6)
-> Nested loop inner join (cost=3.06e+9 rows=30.3e+9) (actual
time=10205..38352 rows=1.61e+6 loops=1)
    -> Index lookup on si using <auto_key0> (ip_address=s.ip_
address) (actual time=0.00945..0.00979 rows=0.645 loops=2.5e+6)
```

- Второе полное сканирование **sessions** (10 млн строк).
- Неэффективные соединение (Nested Loop).

Этап 3. Агрегация и сортировка

```
-> Aggregate using temporary table (actual time=115467..115467 rows=1.
61e+6 loops=1)
-> Table scan on <temporary> (actual time=115467..118821 rows=1.61e+6
loops=1)
-> Sort: s.ip_address, sessions_count DESC (actual time=120878..
121131 rows=1.61e+6 loops=1)
```

- Временная таблица для 1.61 млн строк.
- Медленная сортировка 1.61 млн строк (2.6 сек.).
- Общее время агрегации: 115 сек.

- Причина: большой объем данных для агрегации.

Всего за запрос:

- 20 млн строк (2 сканирования sessions) — главная проблема.
- 2.5 млн обращений к CTE.
- 1.61 млн строк во временной таблице.

Чтобы получить при сканировании таблицы **sessions** все требуемые значения (*ip_address*, *user_id*, *start_time*, *session_id*) без дополнительного обращения к данным таблицы, создадим индекс:

```
CREATE INDEX idx_sessions_ip_user ON sessions
(ip_address, user_id, start_time, session_id);
```

Чтобы сократить время получения адреса электронной почты из таблицы **users**, дополнительно можно создать такой индекс:

```
CREATE INDEX idx_emails ON users(email);
```

После создания индексов время выполнения запроса сократилось до 44672 мс (было 109359 мс).

На этапе обработки CTE в плане выполнения теперь значится следующее:

```
-> Covering index skip scan using idx_sessions_ip_user (actual
time=0.756..9696 rows=4.92e+6 loops=1)
```

Было: долгое сканирование + сортировка.

Стало:

- сканирование индекса вместо таблицы (в 2 раза меньше строк);
- устранена сортировка (данные сразу в порядке ip_address).

На этапе присоединения:

```
-> Covering index lookup on s using idx_sessions_ip_user (actual
time=0.0128..0.017 rows=6.28 loops=638968)
-> Single-row index lookup on u (actual time=0.00181..0.00184
rows=1 loops=1.61e+6)
```

Было:

- Два полных сканирования **sessions** (20 млн строк).
- Медленное соединение: 24,5 сек. (2,5 млн × 0,00979 с).
- Nested Loop с высокой стоимостью.

Стало:

- Покрывающий индекс для **sessions**.
- Быстрое соединение: 10,8 сек. (638 тыс. × 0.017 с).
- Уменьшение циклов с 2.5 млн до 638 тыс.

Агрегация и временная таблица:

```
-> Aggregate using temporary table (actual time=41979..41979 rows=1.
61e+6 loops=1)
```

Время выполнения до индекса: 115 сек., после: 41,9 сек.

Сортировка:

```
-> Sort: s.ip_address, sessions_count DESC (actual time=43922.. 44101
rows=1.61e+6 loops=1)
```

Время выполнения до индекса: 253 мс (121131–120878), после индекса: 179 мс (44101–43922).

10.7.3. PostgreSQL

До применения индекса запрос выполняется 40,201 сек.

Как и в других СУБД, основная проблема: двойное полное сканирование таблицы **sessions**:

1. Первое сканирование (2472333 строк) для CTE shared_ips:

```
->  Seq Scan on sessions  (cost=0.00..176080.00 rows=2472333
width=18) (actual time=0.028..1828.317 rows=2499867 loops=3)
```

ПРИМЕЧАНИЕ.

В PostgreSQL операция полного сканирования таблицы называется Seq Scan.

Кроме того, фильтр (user_id IS NOT NULL) убирает 75% строк из основного запроса (`Rows Removed by Filter: 7500133`).

- Второе сканирование:

```
Parallel Seq Scan on sessions s  (cost=0.00..117746.67 rows=4166667
width=34) (actual time=0.576..647.526 rows=3333333 loops=3)
```

В PostgreSQL имеется возможность параллельного (Parallel) сканирования таблиц.

Ресурсоемкая сортировка

Сортировка 2.4M строк для CTE на диске (external merge, 72 Мб):

```
Sort  (cost=540019.14..546199.97 rows=2472333 width=18)
(actual time=32340.446..35615.613 rows=2499867 loops=3)"
Sort Key: sessions.ip_address, sessions.user_id"
Sort Method: external merge  Disk: 72840kB"
  Worker 0:  Sort Method: external merge  Disk: 72832kB"
  Worker 1:  Sort Method: external merge  Disk: 72840kB"
```

Неэффективные соединения:

```
->  Hash Join  (cost=613468.30..812183.86 rows=343380
width=48) (actual time=36793.208..39253.164 rows=483 loops=3)"
  Hash Cond: (s.user_id = u.user_id)"
  ->  Hash Join  (cost=612032.30..809846.46 rows=343380
width=34) (actual time=36765.601..39224.723 rows=483 loops=3)"
  Hash Cond: ((s.ip_address)::text = (si.ip_address)::text)"
```

Решение проблем:

1. Необходим индекс для *ip_address* и *user_id*, чтобы оптимизировать сканирование по таблице **sessions**.

2. Поскольку PostgreSQL поддерживает условия в индексах, а нас интересуют только зарегистрированные пользователи, необходимо добавить в индекс:

```
… WHERE user_id IS NOT NULL;
```

3. Запрос использует столбец *start_time*, функции MIN и MAX, что требует быстрого доступа к экстремальным значениям, но прямо не используется в группировке.

> Есть **два варианта индексирования** для этого столбца: включение в ключевую часть либо добавление через ключевое слово INCLUDE.

В первом варианте СУБД сможет мгновенно находить min/max значения через переход к первому/последнему элементу в группе. Кроме того, записи будут физически упорядочены: для каждой группы (ip_address, user_id) записи отсортированы по *start_time*.

Во втором варианте порядок данных внутри групп не будет обеспечен, и придется перебирать все значения, зато индекс будет занимать физически меньше места.

При незначительном (как в нашем случае) размере групп применять можно оба варианта, второй будет более эффективен в случае, когда в таблицу будут добавляться новые значения.

Остановимся на первом варианте, но добавим в него *session_id*:

```
CREATE INDEX idx_covering_index
ON sessions(ip_address, user_id, start_time)
INCLUDE (session_id) WHERE user_id IS NOT NULL;
```

Чтобы ускорить доступ к *e-mail* в таблице **users**, добавим индекс:

```
CREATE INDEX idx_emails ON users(email);
```

После создания индексов время выполнения запроса составило 3,763 секунды (было 40,201 секунд).

Результаты:

- Устранено полное сканирование таблицы, устранена необходимость фильтрации (условие user_id IS NOT NULL уже включено в индекс).

Было:

```
->  Parallel Seq Scan on sessions s (cost=0.00..117746.67 rows=4166667
width=34) (actual time=0.576..647.526 rows=3333333 loops=3)
```

Стало:

```
->  Index Only Scan using idx_covering_index on sessions s
(cost=0.43..107185.43 rows=10000000 width=34) (actual time=0.509..622.678
rows=2493761 loops=1)
  Heap Fetches: 0
```

- Устранена необходимость фильтрации user_id IS NOT NULL (условие включено в индекс). Данные полностью удовлетворяются индексом (Heap Fetches: 0).

Устранение дорогостоящей сортировки на диске, данные поступают из индекса в нужном порядке (ip_address, user_id). Сортировка производится в памяти для 724 строк вместо 2.5 млн.

Было:

```
->  Sort (cost=540019.14..546199.97 rows=2472333 width=18)
(actual time=32340.446..35615.613 rows=2499867 loops=3)
      Sort Method: external merge  Disk: 72840kB
```

Стало:

```
->  GroupAggregate (cost=0.43..150451.25 rows=824111 width=22)
(actual time=4.233..1791.925 rows=724 loops=1)
      ->  Index Only Scan using idx_covering_index on sessions
(cost=0.43..107185.43 rows=2472333 width=18)
```

Оптимизация соединений (из Hash Join в Merge Join), устранено создание хеш-таблицы. Оба источника данных отсортированы по *ip_address* (благодаря индексу).

Было:

```
->  Hash Join (cost=612032.30..809846.46 rows=343380 width=34)
(actual time=36765.601..39224.723 rows=483 loops=3)
      Hash Cond: ((s.ip_address)::text = (si.ip_address)::text)
```

Стало:

```
->  Merge Join (cost=0.86..301179.17 rows=824111 width=34) (actual
time=9.250..3667.936 rows=1448 loops=1)
      Merge Cond: ((s.ip_address)::text = (sessions.ip_address)::text)
```

Общий итог: ускорение запроса в 10 раз, устранены операции с диском (external merge), использование памяти снизилось с 72 Мб до 27 Кб.

10.8. Диагностика проблем производительности

Производительность запросов — краеугольный камень работы с базами данных. Когда запросы начинают выполняться неприемлемо долго, первым подозреваемым часто становятся индексы, а точнее их отсутствие или неэффективное использование.

Диагностика проблем требует внимательного изучения двух ключевых аспектов: состояния самих индексов и актуальности статистики по данным.

10.8.1. Неэффективные индексы

Одна из самых распространенных проблем — наличие индексов, которые созданы, но фактически не используются оптимизатором запросов. Такие "мертвые" индексы не только не приносят пользы, но и замедляют операции записи (INSERT, UPDATE, DELETE), так как их структуру приходится поддерживать в актуальном состоянии.

SKLADCHIK.ORG

Для выявления таких индексов вы можете обратиться к специализированным системным представлениям, предоставляемым СУБД.

В Microsoft SQL Server ценную информацию об использовании индексов можно почерпнуть из динамического административного представления **sys.dm_db_index_usage_stats**, которое отслеживает операции чтения и записи по каждому индексу.

При помощи этого запроса можно выяснить даты и время последних операций по индексу в текущей базе данных:

```
SELECT
    s.database_id,
    s.object_id,
    i.name AS index_name, -- Имя индекса
    s.user_seeks,
    s.user_scans,
    s.user_lookups,
    s.user_updates,
    s.last_user_seek, -- Последнее обращение
    s.last_user_scan, -- Последний обход
    s.last_user_lookup, -- Последний поиск
    s.last_user_update - Последнее обновление
FROM
    sys.dm_db_index_usage_stats s
INNER JOIN
    sys.indexes i ON s.object_id = i.object_id AND s.index_id
= i.index_id
WHERE
    s.database_id = DB_ID();
```

Тем, кто работает с MySQL, аналогичные данные можно найти в таблице **table_io_waits_summary_by_index_usage** схемы *performance_schema*, где отслеживается статистика по объектам.

```
SELECT * FROM performance_schema.table_io_waits_summary_by_
index_usage
ORDER BY COUNT_READ DESC;
```

Для PostgreSQL ключевыми индикаторами являются столбцы *idx_scan* и *last_idx_scan* в представлении **pg_stat_user_indexes**, который показывает, сколько раз конкретный индекс был использован для сканирования и когда это происходило в последний раз:

```
SELECT * FROM pg_stat_user_indexes;
```

10.8.2. Статистика и ее роль

Правильная работа оптимизатора запросов критически зависит от актуальности статистики по данным. Эта статистика хранит информацию о распределении значений в столбцах таблиц и индексов, помогая оптимизатору оценить селективность условий и выбрать самый эффективный план выполнения — например, решить, использовать ли индекс или выполнить полное сканирование таблицы.

Если статистика устаревает (например, после массовой вставки, удаления или обновления значительной части данных), оптимизатор начинает строить планы на основе неверных представлений о данных. Это может привести к катастрофическому падению производительности — выбору сканирования огромной таблицы вместо эффективного поиска по индексу или наоборот.

Чтобы предотвратить это, необходимо регулярно обновлять статистику:

- В SQL Server для этого используется команда **UPDATE STATISTICS**.
- В MySQL запускают **ANALYSE TABLE** для пересчета статистики по конкретной таблице.
- В PostgreSQL обновить статистику можно командой **ANALYSE**, которая может быть выполнена как для всей базы, так и для отдельных таблиц.

При работе с базами данных, работающими в производственной среде с реальными данными, эти операции должны стать частью рутинного обслуживания.

10.9. Лучшие практики и антипаттерны

Создание индексов — это не разовая акция, а постоянный процесс балансировки между ускорением чтения и замедлением записи.

10.9.1. Рекомендации

Фундаментальный принцип звучит просто: индексировать следует по потребности. Прежде чем создавать индекс, стоит проанализировать реальные рабочие нагрузки. Какие столбцы чаще всего фигурируют в условиях WHERE для фильтрации? По каким полям часто выполняется сортировка (ORDER BY) или объединение таблиц (JOIN)? Именно эти столбцы — первичные кандидаты на индексацию.

Не менее важно избегать избыточности. Наличие нескольких индексов с одинаковым или почти одинаковым набором столбцов (например, (A) и (A, B)) обычно бесполезно — составной индекс (A, B) уже может эффективно обслуживать запросы, фильтрующие только по A.

Регулярно проводите аудит, чтобы выявлять и удалять дубликаты и ненужные индексы.

10.9.2. Чего избегать

Некоторые подходы к индексации чаще приносят больше вреда, чем пользы. Создание индексов на столбцах, которые очень часто обновляются (например, счетчики или статусы в высоконагруженных транзакционных системах), может значительно снизить пропускную способность при записи, так как каждое изменение требует перестройки индекса.

На маленьких таблицах (например, справочниках с десятками записей) индексы часто не дают заметного прироста скорости чтения (оптимизатор может предпочесть быстрое полное сканирование), но при этом замедляют операции модификации данных.

Индексация низкоселективных столбцов, таких как gender (пол), где всего 2 возможных значения, или столбцов, часто содержащих NULL, также редко бывает эффективной.

Оптимизатор, понимая, что индекс отфильтрует лишь малую часть данных (например, половину таблицы по значению gender = 'M'), скорее всего,

проигнорирует его и выберет сканирование таблицы, сделав индекс бесполезной нагрузкой на систему. Следует тщательно оценивать потенциальную выгоду от индекса в таких сценариях.

Что узнали и чему мы научились?

Индексы позволяют превратить медленный запрос в молниеносный, но их применение требует глубокого понимания и дисциплины.

Ключевые принципы, которые мы усвоили:

- Индексы — компромисс между скоростью чтения и записи, они ускоряют поиск, сортировку и соединение, но замедляют вставку, обновление и удаление.
- Занимают дисковое пространство, требуют обслуживания (перестройка, статистика).
- План выполнения — ключ к пониманию путей к оптимизации.
- Индексы требуют регулярного аудита и обслуживания.
- Статистика — "глаза" оптимизатора.
- Индексы создаются по потребности, а не заблаговременно.

Глава 11.

Представления (Views)

11.1. Зачем нужны представления?

Очень часто при эксплуатации базы данных регулярно приходится выполнять набор одних и тех же запросов. Запросы могут быть довольно сложными, и писать их каждый раз — задача трудоемкая и чреватая ошибками. Можно, разумеется, сохранять текст запроса в файле и каждый раз копировать и вставлять, но в этом нет необходимости, так как СУБД сама предоставляет возможность для сохранения нужных запросов. Такие сохраненные запросы называются представлениями (**VIEW**).

Представления, как и таблицы, имеют уникальные имена в рамках одной базы данных (например, SalesReport или ActiveCustomers), и запросы к ним можно строить так же, как и к таблицам, при помощи команды SELECT. Когда вы запрашиваете данные из представления, база данных сперва выполняет тот самый сохраненный сложный запрос и возвращает вам готовый результат. Это сильно упрощает работу, особенно для пользователей, которым не нужно знать все тонкости структуры базы. Пользователь или приложения видят только нужный им набор столбцов, им не требуется знать, откуда эти данные берутся. Так достигается одна из функций представлений — **абстракция**.

Но упрощение доступа к сложным запросам — это только одна из функций представлений. Не менее важную роль представления играют в обеспечении **безопасности**. Представления позволяют контролировать, какие данные видят пользователи.

К примеру, имеется таблица **employees**, в которой содержатся персональные данные, и конфиденциальные поля, вроде зарплаты. Создав представление только с требуемыми для работы столбцами (например, предоставив руководителям отделов представление MyTeam), можно ограничить распространение чувствительных данных. В одной из следующих глав мы рассмотрим на практике, как осуществлять разграничение доступа более подробно.

В больших и сложных проектах одна и та же бизнес-логика (например, расчет скидки, определение статуса заказа, фильтр "активные клиенты") используется во многих местах. Одним из кошмаров разработчика при работе с базами данных, которые эксплуатируются и эволюционируют много лет или даже уже десятков лет, является несколько разных блоков, выполняющих одну и ту же функцию, созданные в разное время разными разработчиками. Избежать подобного можно при помощи представления, которое будет служить единственным "источником истины", обеспечивая **согласованность логики**.

Все, кто использует представление, будут получать данные, рассчитанные по одним и тем же правилам. И что еще важнее — если правила нужно изменить (например, поменять формулу скидки), достаточно обновить одно представление, и все запросы, которые на него ссылаются, автоматически начнут использовать новую логику. Это избавляет от необходимости искать и править одну и ту же формулу в десятках разных отчетов или скриптов.

Структура баз данных со временем может меняться: таблицы переименовываются, добавляются или удаляются столбцы. Представления могут помочь с защитой от подобных изменений. Если приложения напрямую обращаются к этим таблицам, любое изменение может их сломать. Представление может служить стабильным "буфером". Например, для получения информации о клиентах вместо таблиц приложение продолжает запрашивать данные по клиентам из представления CustomerData, даже если внутри него изменились названия таблиц или порядок столбцов.

Чтобы обеспечить **совместимость** с новой структурой базы, достаточно будет лишь обновить представление, чтобы оно соответствовало новой структуре, а код приложения останется прежним.

Представления помогают организовать данные для разных задач или групп пользователей. На одних и тех же базовых таблицах можно создать разные представления: одно для отдела продаж (с акцентом на клиентов и

заказы), другое для склада (с акцентом на остатки и поставки), третье для бухгалтерии (с финансовыми показателями).

Каждая группа работает с "своим" удобным срезом данных. Так представления помогают обеспечивать **логическое разделение данных**.

Таким образом, представления — это не просто способ сохранить запрос. Это инструмент для упрощения разработки, усиления безопасности, обеспечения согласованности данных, повышения стабильности приложений и лучшей организации работы с информацией в базе.

11.2. Создание тестовых данных. Пример распределения гонораров и вознаграждений

Предположим, мы проводим конференцию, на которой будут высокопоставленные докладчики (speakers), каждому из которых предполагается выплата определенного гонорара, размер которого не разглашается, посетители (attendees) с разными категориями пригласительных билетов (VIP, пресса, стандарт).

Создадим базу данных:

```
CREATE DATABASE conference;
```

Создадим таблицы и заполним тестовыми данными (листинги доступны в электронном приложении):

```
CREATE TABLE speakers -- Докладчики
(
  speaker_id INT PRIMARY KEY,
  name VARCHAR(80),
  phone VARCHAR(25),
  fee DECIMAL(10,2), -- Гонорар (конфиденциальная информация)
  account_no VARCHAR(30), -- Номер счета для перечисления
);
```

```
CREATE TABLE talks -- Доклады
(
  talk_id INT PRIMARY KEY,
  title VARCHAR(200),
  start_time TIME,
  room VARCHAR(50) CHECK(room IN('Зал 1', 'Зал 2')),
  speaker_id INT,
  FOREIGN KEY (speaker_id) REFERENCES speakers(speaker_id)
);

CREATE TABLE attendees -- Слушатели
(
  attendee_id INT PRIMARY KEY,
  name VARCHAR(80),
  ticket_type VARCHAR(20)
   CHECK(ticket_type IN ('standard', 'vip', 'press'))
);


INSERT INTO speakers VALUES
(1, 'Афанасий Борщов', '+7-665-132-32-32', 50000, '4010280001'),
(2, 'Андрей Бузыкин', '+7-665-123-23-23', 80000, '4010280003');

INSERT INTO talks VALUES
(1, 'Управление ожиданиями стейкхолдеров в сфере ЖКХ',
 '10:00', 'Зал 1', 1),
(2, 'Коммуникативные паттерны в условиях взаимоисключающих
обязательств', '12:00', 'Зал 2', 2);

INSERT INTO attendees VALUES
(1, 'Екатерина Снегирева', 'standard'),
(2, 'Василий Харитонов', 'vip');
```

11.3. Работа с представлениями

11.3.1. Синтаксис создания и абстракция сложности

Чтобы создать представление на основании вашего запроса, достаточно написать:

```
CREATE VIEW <имя_представления> AS
-- Далее текст запроса SELECT
```

Чтобы получить публичное расписание докладов, например, создадим представление **v_talks_schedule**:

```
CREATE VIEW v_talks_schedule AS
SELECT t.title, t.start_time, t.room, s.name
FROM talks t
JOIN Speakers s ON t.speaker_id = s.speaker_id;
```

Теперь, чтобы получить расписание, не нужно писать сложный запрос с JOIN, нам достаточно написать:

```
SELECT * FROM v_talks_schedule;
```

чтобы получить тот же результат:

title	start_time	room	name
Управление ожиданиями стейкхолдеров в сфере ЖКХ	10:00	Зал 1	Афанасий Борщов
Коммуникативные паттерны в условиях взаимоисключающих обязательств	12:00	Зал 2	Андрей Бузыкин

Теперь пользователи смогут обращаться к представлению **v_talks_schedule** вне зависимости от того, как именно реализован запрос, формирующий расписание. Создав это представление, мы добились абстракции сложности и создали механизм для обеспечения совместимости — нам достаточно поддерживать одно представление, чтобы не сломать ничего на стороне клиентов базы данных.

SKLADCHIK.ORG

11.3.2. Безопасность и логическое разделение данных

Контактный телефон, указанный в таблице **speakers**, является общедоступной информацией для посетителей, пользоваться которой может, к примеру, приложение для смартфона, использующее нашу базу данных. В то же время размер выплачиваемого гонорара является конфиденциальной информацией. Поскольку и то и другое хранится в одной таблице, то, имея доступ к этой таблице через приложение, злоумышленнику ничего не стоит получить и конфиденциальную информацию. Чтобы избежать подобных рисков, для приложения необходимо создать представление, которое выводит только общедоступную информацию:

```
CREATE VIEW v_speakers_phones AS
SELECT speaker_id, name, phone
FROM speakers;
```

Теперь приложение получает только общедоступную информацию:

speaker_id	name	phone
1	Афанасий Борщов	+7-665-132-32-32
2	Андрей Бузыкин	+7-665-123-23-23

В то же время бухгалтерии необходимо знать размеры вознаграждений и номера счетов докладчиков. При этом бухгалтерии вовсе не обязательно знать телефон и имя докладчика.

Мы можем предоставить только эти данные в отдельном представлении:

```
CREATE VIEW v_payments AS
SELECT account_no, fee
FROM speakers;
```

Теперь бухгалтерское приложение сможет получить требуемую информацию запросом:

```
SELECT * FROM v_payments;
```

Результат:

account_no	fee
4010280001	50000.00
4010280003	80000.00

В одной из следующих глав мы подробнее рассмотрим возможности разных СУБД по разграничению доступов к таблицам и представлениям.

11.3.3. Обновляемые представления

Роль представлений не ограничена только запросами SELECT, они могут прийти на помощь и при добавлении.

Предположим, на кассе работает приложение по регистрации участников, которые могут приобрести стандартное (standard) или vip-приглашение. Однако в таблице **attendees** предусмотрено также и приглашение для прессы (press). Аккредитация представителя СМИ требует отдельной процедуры, поэтому мы можем сделать невозможным для кассира регистрировать представителей прессы, создав соответствующее представление:

```
CREATE VIEW v_registration AS
SELECT attendee_id, full_name, ticket_type
FROM Attendees
WHERE ticket_type IN ('standard', 'vip')
WITH CHECK OPTION;
```

Конструкция WITH CHECK OPTION в представлениях гарантирует, что все операции изменения данных (INSERT, UPDATE) через представление будут соответствовать условию WHERE этого представления.

Теперь можно добавлять пользователей через представление:

```
INSERT INTO v_registration VALUES
(3, 'Алла Ермакова', 'standard');
```

Однако попытка добавить представителя прессы завершится ошибкой:

```
-- Вызовет ошибку
INSERT INTO v_registration VALUES
(4, 'Билл Хансен', 'press');
```

ПРИМЕЧАНИЕ.

Родительская таблица все еще позволит добавлять любой тип посетителей, в отличие от представления.

Важный момент: обновляемые представления имеют ряд ограничений:

- Запрос должен содержать уникальный идентификатор записи: для обновления представления каждая строка должна однозначно определяться. Обычно это значит, что столбцы первичного ключа базовой таблицы должны присутствовать в результате запроса.
- В запросе не должно быть объединений (JOIN). Если в запросе присутствуют объединения нескольких таблиц, то обновление становится невозможным.
- Запрос не должен содержать агрегатных функций.
- Нельзя использовать DISTINCT, GROUP BY, HAVING, оконные функции.
- Запрос должен быть только из одной таблицы.
- Условия WHERE не должны включать внешние зависимости от других представлений или функций.
- Таблица, лежащая в основе представления, должна содержать первичный ключ.
- Запрос не должен содержать LIMIT/TOP или OFFSET.

11.3.4. Материализованные представления

Некоторые запросы, особенно те, которые используют присоединения, сложные формулы расчетов, выполняются очень медленно, даже если таблицы снабжены покрывающими индексами.

В этом случае бывает полезно иметь предварительно рассчитанные и физически сохраненные результаты такого запроса.

Для этих целей используется механизм "материализации" (физического сохранения результатов запроса).

Материализованные представления решают ключевую проблему производительности: они предварительно вычисляют и физически сохраняют результаты сложных запросов.

Такая стратегия помогает в случае работы с тяжелыми аналитическими запросами (агрегации по миллионам строк, многократные JOIN), системами, где частые запросы к "сырым" данным перегружают СУБД, и со сценариями с приемлемой задержкой данных (ежедневные отчеты, исторический анализ).

В **PostgreSQL**, чтобы создать материализованное представление, необходимо добавить ключевое слово MATERIALIZED в инструкцию создания:

```
CREATE MATERIALIZED VIEW <имя_представления> AS
SELECT … -- текст запроса;
```

Создав материализованное представление, можно добиться существенного ускорения запроса, так как данные хранятся уже в готовом виде с уже выполненными расчетами, агрегацией, сортировкой и присоединениями. Более того, материализованные представления поддерживают индексы, что может еще больше сократить время получения результата, однако есть и минус — данные в представлении не обновляются автоматически, они актуальны только на момент последнего обновления, которое необходимо производить отдельно, при помощи команды REFRESH:

```
REFRESH MATERIALIZED VIEW [CONCURRENTLY] <имя_представления>;
```

Ключевое слово CONCURRENTLY не является обязательным и используется для того, чтобы провести обновление без блокировок таблиц. С такой опцией обновление будет проходить медленнее.

В **MS SQL Server** прямой поддержки материализованных представлений нет, тем не менее, похожий эффект достигается с помощью **индексированных представлений** (indexed views). Для этого необходимо создать само представление с обязательным дополнением WITH SCHEMABINDING, а затем — индекс к этому представлению:

```
CREATE VIEW <имя_представления> WITH SCHEMABINDING AS
SELECT … -- Текст запроса;
CREATE CLUSTERED INDEX <имя_индекса> ON <имя_представления> …
```

Конструкция WITH SCHEMABINDING очень важна — она фиксирует схему представления, предотвращая изменения базовых таблиц и столбцов, участвующих в определении представления. Это делается для того, чтобы сохранить целостность данных и гарантировать стабильность работы индексированного представления.

При создании индексированного представления WITH SCHEMABINDING SQL Server накладывает следующие ограничения:

1. После создания индексированного представления нельзя изменить структуру базовых таблиц (добавлять или удалять столбцы, менять типы данных колонок), пока существует данное представление. Любые попытки модификации будут завершаться ошибкой.
2. Система следит за изменениями схем таблиц, которые используются в запросе представления. Например, если изменяется одна из таблиц, используемых в представлении, система откажется применить изменение, пока представление не удалено или не перестало зависеть от данной таблицы.
3. Столбцы, участвующие в запросе, должны иметь уникальные имена среди выбранных таблиц. То есть, если одно и то же имя встречается дважды, надо явно переименовать один из столбцов с помощью псевдонимов.
4. Вы не сможете удалить базовые таблицы или изменить их структуры, пока активен запрос, использующий эти таблицы.

Наличие данных ограничений имеет и плюсы — индексированное представление не требуется обновлять вручную, данные в нем будут всегда актуальны, в отличие от материализованных представлений в PostgreSQL.

ВНИМАНИЕ!

Частые обновления таблицы могут приводить к повышению "стоимости" поддержки индекса.

В **MySQL** поддержки материализованных представлений нет. Для получения похожих результатов необходимо создавать временные таблицы и регулярно обновлять в них данные.

ПРЕДОСТЕРЕЖЕНИЕ.

Высокая стоимость поддержки индекса при частых изменениях базовых данных. Необходимость тщательного анализа производительности.

11.4. Когда представления могут мешать?

Наиболее распространенная ошибка при работе с представлениями — **глубокая вложенность представлений** (представление поверх другого представления). Оптимизатор запросов не всегда способен выбрать оптимальный план выполнения, и многоуровневые представления приводят к значительным потерям производительности. Лучше избегать чрезмерной глубины вложений.

По мере увеличения числа уровней вложенности представлений задача выявления причины низкой производительности или некорректных результатов заметно осложняется. Важно помнить, что каждое представление добавляет уровень непрозрачности и замедляет диагностику.

Часто представления создаются для разового использования, забывая впоследствии удалить. Такие объекты увеличивают сложность инфраструктуры и снижают общую управляемость системой.

Некоторые виды представлений, например те, которые содержат агрегационные функции или объединение нескольких таблиц, не поддерживают

прямые операции UPDATE, DELETE или INSERT. Приходится прибегать к дополнительным ухищрениям, что увеличивает риски возникновения ошибок.

Представления формируют жесткую зависимость от структур базовых таблиц. Модификация таблицы может нарушить работоспособность представления. В MS SQL Server имеется механизм SCHEMABINDING, который частично решает проблему, фиксируя схему, но одновременно лишает пользователя свободы внесения изменений.

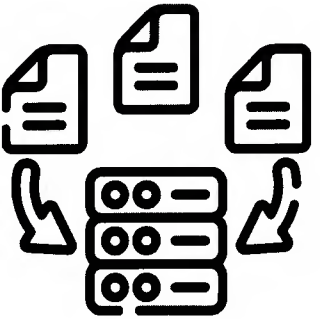
Что узнали и чему мы научились?

Представления — еще один инструмент в вашем арсенале. Их правильное использование упрощает разработку приложений, облегчает поддержку инфраструктуры, обеспечивает стабильный интерфейс доступа к данным, централизует бизнес-логику и способствует повышению уровня безопасности.

В то же время чрезмерное увлечение представлениями способно привести к проблемам с производительностью, усложнению архитектуры и трудностям в поддержке.

Советы и рекомендации:

- Используйте представления умеренно и продуманно. Избегайте глубоких уровней вложенности, старайтесь минимизировать количество промежуточных слоев.
- Анализируйте планы выполнения запросов через представления. Проверьте, насколько эффективно оптимизатор обрабатывает ваш запрос.
- Прежде чем внедрять сложное представление, убедитесь, что базовый запрос оптимален и эффективен.
- Применяйте материализованные представления аккуратно, учитывая затраты на хранение и ресурсы на обновление.
- Помните, что представления создают зависимости между объектами БД, будьте внимательны при внесении изменений в базу данных.



Глава 12.

Транзакции, изоляция и блокировки

12.1. Принципы ACID

Представьте себе ситуацию с денежным переводом в банке: со счета отправителя списывается некая сумма, затем происходит сбой системы, и данная сумма просто исчезает — на счет получателя она так и не поступила. Целостность данных катастрофически нарушена.

Представьте другую ситуацию: при заказе в интернет-магазине проверяется остаток товара на складе: в случае если остаток имеется, формируется заказ, а остаток уменьшается. В какой-то момент времени один покупатель заказывает товар, проверка наличия проходит успешно (на складе 1 шт.), после чего инициируется процедура формирования заказа, но в этот же момент другой покупатель размещает заказ на этот же товар. Поскольку первый заказ еще не сформирован до конца и запись об остатке на складе не уменьшилась, проверка по второму заказу точно так же проходит успешно. В результате на товар, который физически присутствует на складе в 1 экземпляре, формируются два заказа, а количество в базе данных составляет 1 шт.

Ответом на такие сценарии стала концепция транзакции — неделимого набора операций, которые выполняются вместе и воспринимаются как единое целое. Хотя сама концепция обработки групп операций как единого целого существовала и раньше, транзакция как формализованная концепция была описана в 1981 году в работе американского исследователя в области вычислительных систем Джима Грея. Еще через два года исследователями Тео Хердером и Андреасом Рейтером в их статье "Principles of Transaction-Oriented Database Recovery" был создан акроним ACID для описания единого набора требований к транзакциям, на которых работают все современные СУБД.

SKLADCHIK.ORG

ACID — сокращение от 4 принципов: Atomicity, Consistency, Isolation и Durability (Атомарность, Согласованность, Изоляция и Долговечность).

Atomicity (атомарность)

Транзакция неделима ("атомон" по-гречески *неделимый*). Транзакция подчиняется принципу "Все или ничего". Либо выполняются ВСЕ операции внутри транзакции, либо не выполняется НИ ОДНА из них. Принцип атомарности решает проблему частичных обновлений (деньги списали, но не зачислили).

Consistency (Согласованность)

Транзакция переводит базу данных из одного корректного состояния в другое корректное состояние. Принцип согласованности решает проблему появления в базе невозможных данных (отрицательное количество, бронирование одного места двумя пассажирами, заказ без покупателя и пр.).

Isolation (Изолированность)

Две параллельные транзакции не должны мешать друг другу. Результат одновременного выполнения нескольких транзакций должен быть эквивалентен результату их последовательного выполнения (хотя на практике могут встречаться компромиссы, о которых речь пойдет чуть позже).

Durability (Долговечность)

Если транзакция успешно **зафиксирована** (действие называется COMMIT), то ее изменения становятся постоянными и должны сохраниться в базе данных в случае любого сбоя (отказ питания, крах ОС, аппаратный сбой диска).

Принцип исключает потерю критически важных данных после того, как система сообщила пользователю об успехе операции (например, деньги

списались, пользователь увидел "Успешно!", но из-за сбоя информация о зачислении пропала).

12.2. Процесс выполнения транзакции

Физическая реализация принципов ACID транзакций в современных реляционных СУБД основана на концепции журнала транзакций, механизма отката (rollback) и механизма точек восстановления (checkpoints). Тонкости реализации в разных СУБД могут отличаться, об этом можно почитать в документации, фундаментально же принцип проведения транзакции следующий:

1. **Подготовка транзакции**. СУБД инициирует транзакцию, резервирует необходимые ресурсы, устанавливает точку отката (восстановления) для транзакции в журнале транзакций.
2. **Выполнение транзакции**. Последовательно выполняются SQL-запросы, Любое изменение данных (INSERT, UPDATE, DELETE) обязательно предваряется записью соответствующей информации в журнале. Такая запись содержит:
   - Идентификатор транзакции.
   - Тип операции.
   - Изменяемый объект (таблицу).
   - Старые значения данных.
   - Новые значения данных.

   Все изменения сначала применяются в оперативной памяти (пока позволяют ресурсы) для производительности.
3. **Подготовка к фиксации**. После отработки последнего запроса внутри транзакции начинается подготовка к ее фиксации. СУБД удостоверяется, что все записи из журнала транзакций сброшены на физическое хранилище (диск), чтобы даже в случае сбоя изменения данные можно было восстановить из журнала. На этом этапе также проверяются все ограничения (constraints), уникальность и т.д. При их нарушении транзакция будет прервана.

4. **Фиксация** (COMMIT). Как только гарантирована запись журнала на диск, СУБД делает в журнал транзакций специальную запись **COMMIT** с идентификатором транзакции. Сброс этой записи на диск является моментом фиксации. После этого транзакция считается успешно завершенной, и ее изменения становятся видимыми для других транзакций (в соответствии с уровнем изоляции).
5. **Выход из транзакции.** СУБД освобождает блокировки и ресурсы, связанные с этой транзакцией.

Откат транзакции (ROLLBACK) — операция отмены всех изменений, произведенных транзакцией. Откат может вызываться явно по указанию пользователя либо производиться автоматически, если происходит ошибка или сбой во время выполнения транзакции (до фиксации).

Используя информацию, сохраненную в журнале транзакций, СУБД восстанавливает старые значения, записанные на этапе создания точки восстановления, после чего, в зависимости от СУБД, либо делает в журнал транзакций запись **ROLLBACK**, либо просто помечает транзакцию как отмененную.

После отката освобождаются блокировки и ресурсы.

12.3. Уровни изоляции транзакций

Опыт параллельной работы нескольких пользователей с одной базой данных позволил выявить три основных типа аномалий, которые возникают в процессе такой работы.

Грязное чтение (Dirty Read) — *транзакция А* видит незафиксированные изменения *транзакции В*.

Если В откатится, А будет основываться на данных, которых никогда официально не существовало.

ПРИМЕР: кассир в кассе бронирования авиабилетов видит, что коллега только что "освободил" место 15А (начал отмену брони, но еще не подтвердил). Кассир продает место 15А новому клиенту. Потом выясняется, что коллега передумал и не отменил бронь. Результат: два билета на одно место.

Неповторяющееся чтение (Non-repeatable Read) — *транзакция А* читает одну и ту же строку дважды и видит разные значения, потому что между чтениями *транзакция В* изменила и зафиксировала эту строку.

ПРИМЕР: пользователь в интернет-банке дважды проверяет баланс счета в рамках одной сессии (транзакции). Первый раз — 1000 руб. Пока он формирует платеж на 900 рублей, происходит автоматическое списание 200 рублей за какую-нибудь подписку. Попытка списания 900 рублей заканчивается неудачей — на счете не хватает денег.

Фантомное чтение (Phantom Read) — *транзакция А* повторяет запрос с некоторым условием (например, WHERE flight_id = 123) и получает другой набор строк, потому что транзакция В зафиксировала вставку или удаление строк, удовлетворяющих этому условию.

ПРИМЕР: менеджер авиакомпании запускает отчет "Свободные места на рейс 123" (видит 10 мест). Пока он анализирует данные, кассир продает билет на одно место. Если менеджер обновит отчет, он увидит 9 мест. Если его транзакция должна была работать с фиксированной картиной на момент начала, это станет проблемой.

Стандарт SQL (ANSI/ISO) определяет уровни изоляции, характеризуя их через виды аномалий, которые они должны предотвращать:

1. **READ UNCOMMITTED** — разрешает все три аномалии. Это самый быстрый, но и самый "грязный" уровень. Подходит только для грубой аналитики, где точность не важна.

2. **READ COMMITTED** — гарантирует, что читаются только зафиксированные данные. Предотвращает грязное чтение, но разрешает неповторяющееся и фантомное чтение. Самый распространенный уровень по умолчанию.

3. **REPEATABLE READ** — гарантирует, что строки, прочитанные один раз, не изменятся при повторном чтении в рамках той же транзакции. Предотвращает грязное и неповторяющееся чтение, но (по стандарту) разрешает фантомное чтение.

4. **SERIALIZABLE** — самый строгий уровень. Гарантирует, что результат параллельного выполнения транзакций эквивалентен их последовательному выполнению. Предотвращает все три аномалии. Достигается либо через жесткие блокировки (снижающие параллелизм), либо через сложные механизмы контроля версий с откатами при конфликтах.

Выбор уровня изоляции зависит от требований приложения:

- Для перевода денег между счетами нужна максимальная строгость (SERIALIZABLE), чтобы исключить любые аномалии.
- Для генерации аналитического отчета, где абсолютная точность на миллисекунду не критична, подойдет более слабый уровень (READ COMMITTED), чтобы не тормозить оперативные операции.
- Для отображения количества лайков под постом в соцсети допустим даже READ UNCOMMITTED, так как временная неточность не смертельна.

12.4. Реализация уровней изоляции в разных СУБД. Пример авиакассы в час пик

Вернемся к нашей авиакассе в час пик: десять операторов одновременно бронируют места на рейс.

Если система позволит всем видеть абсолютно идентичную картину свободных мест в реальном времени, это гарантирует, что никто не продаст одно место дважды. Но какой ценой? Каждому оператору придется ждать, пока предыдущий завершит бронирование. Очередь клиентов будет двигаться мучительно медленно.

Другая крайность: операторы видят данные без ограничений, первый, кто нажмет "Купить", получит место. Система будет молниеносной. Но что если два оператора одновременно увидят свободное место и оба попытаются его забронировать? Высока вероятность, что оба получат подтверждение, а пассажиры потом будут скандалить в аэропорту.

Необходимость в разных уровнях изоляции прямо вытекает из описанной ситуации: чем строже изоляция транзакций (и чем ближе результат к их последовательному выполнению), тем выше целостность данных, но тем ниже скорость работы из-за блокировок и ожиданий. И наоборот: более слабые уровни изоляции повышают производительность и параллелизм, но увеличивают риск аномалий, когда транзакция может увидеть несогласованные данные из-за действий других параллельных транзакций.

Для изоляции транзакций современные СУБД прибегают к двум базовым механизмам:

1. **Блокировки (Locking)** — традиционный метод. При чтении или изменении данных транзакция устанавливает блокировки — механизм защиты данных от несогласованной параллельной обработки несколькими пользователями или приложениями. Когда одна транзакция обращается к ресурсу (таблица, строка, страница данных), СУБД устанавливает блокировку на этот ресурс. Другие транзакции, пытающиеся обратиться к заблокированным данным, ожидают освобождения блокировки первой транзакцией.

2. **Управление версиями данных (MVCC — Multi-Version Concurrency Control)** — более современный и часто более производительный подход. Каждое изменение создает новую версию строки. Транзакция работает со снимком (snapshot) данных, актуальным на момент ее начала (для REPEATABLE READ/SERIALIZABLE) или на момент начала каждого оператора (для READ COMMITTED). Старые версии сохраняются до тех пор, пока они нужны для работы активных транзакций. Читающие тран-

закции не блокируют пишущие, и наоборот. MVCC — основа реализации в PostgreSQL и MySQL InnoDB.

Хотя все СУБД стремятся соответствовать стандарту, детали реализации и поведение на одном и том же уровне могут различаться.

12.4.1. Microsoft SQL Server

В MS SQL Server по умолчанию установлен уровень изоляции READ COMMITTED, который реализован при помощи механизма блокировок.

Чтение устанавливает разделяемую блокировку на строку, которая снимается сразу после прочтения. Предотвращает грязное чтение. Разрешает неповторяющееся и фантомное чтение. В качестве альтернативы можно включить опцию базы данных READ_COMMITTED_SNAPSHOT, которая использует MVCC (снимок на момент начала оператора), что снимает блокировки при чтении.

```
ALTER DATABASE <имя_бд> SET READ_COMMITTED_SNAPSHOT ON;
```

- Уровень REPEATABLE READ реализуется через блокировки. Удерживает разделяемые блокировки на всех прочитанных строках до конца транзакции. Предотвращает грязное и неповторяющееся чтение. Разрешает фантомное чтение (по стандарту), так как не блокирует диапазоны.
- Так же MS SQL поддерживает уровень изоляции SNAPSHOT (нестандартный, но важный), является аналогом REPEATABLE READ в MVCC-стиле. Использует снимок данных на момент начала транзакции, что предотвращает грязное, неповторяющееся и фантомное чтение в рамках снимка. При попытке изменить строку, измененную другой транзакцией после начала снимка, произойдет ошибка.
- Уровень SERIALIZABLE использует блокировки диапазонов (range locks) для предотвращения фантомов. Удерживает блокировки на всех прочитанных строках и диапазонах до конца транзакции. Высокий риск взаимных блокировок.

- READ UNCOMMITTED — позволяет грязное чтение (часто через подсказку NOLOCK).

```
SELECT * FROM <имя_таблицы> WITH(NOLOCK);
```

12.4.2. MySQL (InnoDB)

Основной уровень изоляции по умолчанию — REPEATABLE READ. Использует снимок на момент начала первого чтения. Гарантирует стабильность прочитанных строк. Ключевой особенностью движка InnoDB является предотвращение фантомного чтения уже на этом уровне (вопреки стандарту) с помощью **блокировок следующего ключа** (next-key locks), которые блокируют не только существующие строки, но и **промежутки** (gaps) в индексах, предотвращая вставки в прочитанные диапазоны. Это делает уровень очень надежным для большинства задач.

- READ COMMITTED — каждый SELECT видит снимок на момент своего начала. Предотвращает грязное чтение. Для UPDATE/DELETE InnoDB смотрит на последнюю зафиксированную версию строки. Разрешает неповторяющееся и фантомное чтение. Требует явной настройки, как и остальные уровни.
- READ UNCOMMITTED — позволяет грязное чтение (видит незафиксированные данные других транзакций). Используется редко.
- SERIALIZABLE — работает как REPEATABLE READ, но автоматически добавляет блокировку FOR SHARE ко всем простым SELECT, превращая их в блокирующие чтения. Сильно снижает параллелизм.

12.4.3. PostgreSQL

По умолчанию установлен уровень READ COMMITTED (По умолчанию) — каждый SELECT видит снимок данных на момент своего начала. Предотвращает грязное чтение.

Обновления (UPDATE/DELETE) видят последние зафиксированные данные на момент начала оператора и могут перепроверять условия при конфликтах. Разрешает неповторяющееся и фантомное чтение.

- REPEATABLE READ — весьма строгий. Использует единый снимок данных на момент начала первой команды в транзакции. Гарантирует, что все последующие чтения увидят именно этот снимок. Важное отличие PostgreSQL: на этом уровне предотвращается фантомное чтение (хотя стандарт его разрешает), блокируя потенциальные вставки в прочитанных диапазонах. Попытка изменения данных, измененных другой транзакцией, вызовет ошибку сериализации.
- SERIALIZABLE — использует сложные и изощренные алгоритмы (предикатные блокировки, сериализация порядка) для гарантии строгой изоляции. При обнаружении потенциального нарушения сериализуемости транзакция прерывается с ошибкой *serialization_failure*, и приложение должно ее перезапустить. Самый безопасный, но требующий особой обработки ошибок.
- READ UNCOMMITTED — в PostgreSQL фактически ведет себя как READ COMMITTED (грязное чтение как таковое не реализовано).

12.5. Практическая работа с транзакциями. Пример упрощеной банковской системы

Чтобы на практике рассмотреть работу с транзакциями, создадим упрощенную банковскую учетную систему:

```
CREATE TABLE accounts
(
  account INT PRIMARY KEY,
  balance DECIMAL(10,2) NOT NULL DEFAULT 0.0
);
INSERT INTO accounts VALUES (101, 1000.00); -- Начальный баланс
```

В отличие от всех прошлых примеров, в данном случае нам понадобятся два экземпляра (по тексту они будут обозначены "Клиент 1" и "Клиент 2") одновременно запущенных клиентов СУБД (запустите два экземпляра клиентской оболочки и подключитесь из обеих к этой базе данных).

Чтобы начать транзакцию, необходимо ввести команду:

```
BEGIN TRANSACTION;
```

Теперь все команды, которые вы будете вводить, будут считаться частью одной транзакции вплоть до команды фиксации (COMMIT).

ПРИМЕЧАНИЕ.

В MySQL иной синтаксис, можно написать либо START TRANSACTION, либо просто BEGIN, обе команды являются синонимами.

После команды о начале транзакции, если мы хотим установить отличный от принятого по умолчанию уровень изоляции, нужно воспользоваться командой:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

ПРИМЕЧАНИЕ.

В MySQL данную настройку необходимо сделать ДО начала транзакции.

В данной практической работе мы будем его изменять.

Выполните запрос:

```
SELECT balance FROM accounts WHERE account = 101;
```

Теперь переключитесь в *Клиент 2* и выполните следующий запрос:

```
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance - 200.00 WHERE account = 101;
COMMIT; -- Фиксация списания!
```

Вернитесь в *Клиент 1*, повторите запрос и завершите транзакцию:

```
SELECT balance FROM accounts WHERE account = 101;
COMMIT;
```

SKLADCHIK.ORG

Второй запрос вернет результат 800.

Пользователь 1 в рамках одной транзакции увидел два разных значения баланса. Его отчет может быть внутренне противоречивым ("На момент начала операции было 1000, но, пока я смотрел, стало 800?"). На данном уровне изоляции такое поведение системы является нормальным.

Теперь проделайте то же самое, изменив уровень изоляции:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Сколько бы вы ни запрашивали баланс в первом клиенте, ответ будет неизменным.

Если теперь завершить транзакцию в первом клиенте и повторить запрос баланса, сумма изменится на актуальную.

Транзакция *Пользователя 1* работает с снимком данных (snapshot), зафиксированным на момент ее первого оператора чтения (или начала транзакции, в зависимости от СУБД). Она не видит изменения, зафиксированные *Пользователем 2* после этого момента. Данные повторяются при повторном чтении.

ВАЖНО! (особенно в PostgreSQL)

Если *транзакция 1* попытается изменить баланс на основе своего устаревшего снимка (например, UPDATE accounte SET balance = 1000 WHERE account = 101), она получит ошибку конфликта, так как данные под ней уже изменились. Это защищает от логических ошибок на основе неактуальных данных.

Верните баланс к первоначальной сумме 1000 и поменяйте уровень изоляции на SERIALIZABLE:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Для PostgreSQL и MySQL поведение будет аналогичным REPEATABLE READ в плане чтения (увидит 1000.00 оба раза).

В MSSQL Server система может предотвратить саму возможность параллельного выполнения этих транзакций столь "вредным" образом (например,

с помощью предикатных блокировок), или гарантировать, что их результат будет эквивалентен последовательному выполнению (сначала А, потом Б, или сначала Б, потом А). На практике одна из транзакций может быть прервана с ошибкой сериализации, требующей перезапуска, либо оболочка уйдет в бесконечное ожидание окончания запроса из-за блокировки.

Теперь модифицируйте пример 2 следующим образом:

```
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance - 200 WHERE account = 101;
SELECT * FROM accounts WHERE account = 101;
```

В рамках данной транзакции вы увидите, что баланс уменьшился.

Теперь вместо COMMIT выполните команду ROLLBACK и снова запросите баланс:

```
-- Откат транзакции
ROLLBACK;
SELECT * FROM accounts WHERE account = 101;
```

Действие предыдущего UPDATE будет отменено, а баланс вернется к первоначальному значению.

Задания для самостоятельной работы

Попробуйте поэкспериментировать с **фантомным чтением:** добавьте операцию INSERT в *Транзакцию 2* (например, добавление новой записи о транзакции в связанную таблицу **transactions**). Покажите, как на READ COMMITTED / REPEATABLE READ (в стандартной реализации) эта новая запись "появится" при повторном SELECT ... в *Транзакции 1*.

Грязное чтение (Dirty Read): в *Транзакции 2* сделайте UPDATE, но НЕ ДЕЛАЙТЕ COMMIT. На уровне READ UNCOMMITTED в *Транзакции 1* будет видно это незафиксированное изменение. Затем сделайте ROLLBACK в *Транзакции 2* и покажите, что *Транзакция 1* основывалась на несуществующих данных.

12.6. Блокировки

Для предотвращения ситуации, когда разные транзакции пытаются одновременно изменять одну и ту же область данных, приводя к возникновению конфликтов и появлению несогласованных состояний данных, СУБД может использовать механизм блокировок.

Для реализации этого механизма в ядре СУБД работает специальный компонент — **Менеджер блокировок** (Lock Manager). Основная его задача — работа с таблицей блокировок (Lock Table), которая хранит информацию о всех текущих блокировках: какой объект заблокирован, какой транзакцией, тип блокировки и режим совместимости.

Когда транзакция запрашивает блокировку, Lock Manager проверяет, совместима ли запрошенная блокировка с уже существующими на этом объекте. Если она совместима — блокировка выдается, и информация заносится в таблицу, если нет — транзакция ставится в очередь ожидания.

Когда транзакция фиксируется (COMMIT) или откатывается (ROLLBACK), она освобождает все блокировки, выданные для данной транзакции. Lock Manager удаляет записи о них и проверяет, могут ли теперь быть выданы блокировки транзакциям, ожидающим в очереди.

Периодически менеджер проверяет существующие блокировки на наличие "петель" — взаимных блокировок (**deadlock**), то есть ситуаций, когда две или более транзакций бесконечно ожидают освобождения ресурсов (блокировок), захваченных друг другом. Ни одна не может продолжить работу. Взаимную блокировку можно проиллюстрировать на следующем примере:

- *Транзакция А* заблокировала *строку 1* (например, для обновления).
- *Транзакция Б* заблокировала *строку 2*.
- *Транзакция А* пытается заблокировать *строку 2* (которая у Б) и встает в ожидание.

- *Транзакция Б* пытается заблокировать *строку 1* (которая у А) и тоже встает в ожидание.
- Результат: А ждет Б, Б ждет А. Вечный тупик.

Для обнаружения взаимных блокировок менеджер периодически на основании данных таблицы блокировок строит граф (какой ресурс от какого зависит). Если в графе обнаруживается петля (*Транзакция А* ждет ресурс, удерживаемый Б, а Б ждет ресурс, удерживаемый А, или более длинная цепочка, замыкающаяся в петлю), значит, есть взаимоблокировка.

Для разрешения ситуации СУБД приступает к разрешению: выбору "жертвы" взаимной блокировки (*victim selection*). Критерии выбора "жертвы" могут основываться на таких факторах, как меньший объем работы, меньший тайм-аут, простота отката и пр.

После выбора жертвы производится ее откат — выбранная транзакция принудительно откатывается (выполняется ROLLBACK), освобождая ВСЕ свои блокировки. Остальные транзакции в цикле могут продолжить работу. Клиент при этом получит получает ошибку с сообщением о взаимной блокировке.

12.6.1. Виды блокировок

Блокировки классифицируются по типу доступа, по гранулярности (по видам заблокированных объектов) и по способу установки.

Классификация по виду доступа:

Тип блокировки	Описание
Разделяемые (Shared Locks, S-Locks)	Ставятся при чтении данных. Несколько транзакций могут одновременно удерживать S-блокировки на одних и тех же данных. S-Lock конфликтует только с X-Lock (другие S-Lock разрешены)

Тип блокировки	Описание
Эксклюзивные (Exclusive Locks, X-Locks)	Ставятся при изменении данных (INSERT, UPDATE, DELETE). Только одна транзакция может удерживать X-блокировку на конкретном элементе данных. X-Lock конфликтует со ВСЕМИ другими блокировками (S и X) на том же элементе
Обновляемые (Update Locks, U-Locks)	Промежуточный тип. Ставится, когда транзакция планирует изменить данные (сначала читает, потом решит обновить). U-Lock совместим с S-Lock, но конфликтует с другими U-Lock и X-Lock. Позволяет избежать взаимоблокировок в сценариях "чтение → обновление". При реальном обновлении U-Lock преобразуется в X-Lock

По гранулярности блокировки разделяют по следующим видам:

Тип блокировки	Описание
Блокировки строк (Row-Level Locks)	Наиболее распространенные и желательные. Блокируют только одну строку таблицы, минимизируя конфликты. Основа высокой параллельности
Блокировки страниц (Page-Level Locks)	Блокируют физическую страницу данных (группу строк). Менее желательны, так как могут блокировать нерелевантные строки
Блокировки таблиц (Table-Level Locks)	Блокируют всю таблицу целиком. Сильно снижают параллельность. Используются редко, обычно при операциях DDL (ALTER TABLE) или при массовых обновлениях, где блокировка строк неэффективна

Тип блокировки	Описание
Блокировки предикатов/ диапазонов (Predicate/Key-Range Locks)	Используются на уровнях изоляции SERIALIZABLE или в REPEATABLE READ (в MySQL/PostgreSQL для предотвращения фантомов). Блокируют не существующие строки, а диапазон значений или условие, предотвращая вставку "фантомных" строк

По способу установки различают:

- **Неявные (Implicit Locks)** — автоматически устанавливаются СУБД при выполнении операторов SQL (SELECT ... FOR UPDATE, UPDATE, DELETE и т.д.) в зависимости от уровня изоляции.
- **Явные (Explicit Locks)** — устанавливаются пользователем с помощью специальных команд (например, SELECT ... FOR UPDATE, SELECT ... FOR SHARE в PostgreSQL; SELECT ... WITH (UPDLOCK, HOLDLOCK) в SQL Server; LOCK TABLE ...).

12.6.2. Как избегать взаимных блокировок?

Профилактика лучше лечения, поэтому при разработке базы и при построении запросов к ней рекомендуется руководствоваться следующими соображениями:

1. **Сохраняйте единый порядок доступа**. Самый эффективный способ. Всегда блокируйте ресурсы (например, строки таблиц) в одном и том же строгом порядке во всем приложении. В примере выше: если бы и А, и Б сначала блокировали *строку 1*, а потом *строку 2*, тупика бы не возникло. Одна транзакция получила бы обе блокировки, другая ждала бы освобождения первой строки.
2. **Минимизируйте время удержания** блокировок. Старайтесь выполнять работу внутри транзакции (особенно получение пользовательского ввода или вызовы внешних сервисов) ДО получения блокировок или

ПОСЛЕ их освобождения. Чем короче транзакция, тем меньше окно для конфликта.

3. **Минимизируйте размер транзакций**. Разбивайте большие транзакции на более мелкие атомарные операции, где это логически допустимо.

4. Используйте **менее строгие уровни изоляции**. Если бизнес-логика позволяет, используйте READ COMMITTED вместо REPEATABLE READ или SERIALIZABLE, чтобы уменьшить количество и длительность удерживаемых блокировок.

5. Используйте **блокировки обновления** (**U-Locks**). При реализации подхода "прочитал данные, затем решил их изменить" включите U-Lock на этапе чтения. Это может предотвратить классический сценарий взаимоблокировки при одновременном чтении с последующим обновлением одних и тех же данных.

12.6.3. Явное управление блокировками

В некоторых случаях при выполнении операции выборки (SELECT) бывает полезно указать СУБД, с какой именно целью производится чтение. Если мы используем алгоритм чтение-проверка-обновление, то можно явно создать блокировку на запрошенные данные (создается **X-Lock**):

```
-- Синтаксис для MS SQL:
SELECT … FROM … WITH (UPDLOCK, ROWLOCK)
-- Синтаксис для MySQL и PostgreSQL
SELECT … FROM … FOR UPDATE;
```

Если вы хотите получить гарантию, что они не изменятся до завершения вашей транзакции (например, формирование отчета на основе согласованного снимка), вы можете создать **S-Lock** следующим образом:

```
-- Синтаксис для MS SQL:
SELECT … FROM … WITH (HOLDLOCK, ROWLOCK)
-- Синтаксис для MySQL и PostgreSQL
SELECT … FROM … FOR SHARE;
```

Такой вид блокирует выбранные строки для чтения. Разрешает другим транзакциям ставить FOR SHARE, но запрещает FOR UPDATE и эксклюзивные изменения.

Когда требуемые данные оказываются заблокированными другой транзакцией, СУБД обычно ожидает их освобождения какое-то время. В определенных случаях такое ожидание является нежелательным (например, не стоит заставлять пользователя ждать, лучше сразу вывести сообщение "Попробуйте позднее"). Чтобы сообщить СУБД, что вы не желаете ожидать и хотите получить ошибку сразу, если окажется, что требуемые ресурсы заблокированы, можно использовать конструкцию NOWAIT (не ожидать):

```
-- Синтаксис для MS SQL:
SELECT … FROM … WITH (NOWAIT)
-- Синтаксис для MySQL и PostgreSQL
SELECT … FROM … FOR UPDATE NOWAIT;
```

Если у вас обновляется большой блок данных, вы можете указать СУБД пропускать заблокированные строки. Это особенно полезно при создании обработчиков очередей, когда несколько клиентов обрабатывают общую очередь поступивших заданий:

```
-- Синтаксис для MS SQL:
SELECT … FROM … WITH (READPAST)
-- Синтаксис для MySQL и PostgreSQL
SELECT … FROM … FOR UPDATE SKIP LOCKED;
```

PostgreSQL предоставляет дополнительную возможность блокировать строку от изменений, но при этом разрешать создание и обновление дочерних записей по внешнему ключу. Например, хотя данные внешнего ключа (например, номер счета) используется в транзакции с переводом денег, можно разрешить обновление имени владельца, которое хранится во внешней таблице.

```
-- Только PostgreSQL:
SELECT … FROM … FOR NO KEY UPDATE;
```

Наконец, если вам предстоит массовая загрузка данных, которая потребует большого количества построчных блокировок, либо изменение схемы

таблицы (добавление, модификация или удаление столбцов), разумно заблокировать таблицу целиком, чтобы снизить нагрузку на СУБД.

Для **MS SQL** для этих целей используются подсказки в конструкции WITH:

```
BEGIN TRANSACTION;
SELECT * FROM <имя_таблицы> WITH (TABLOCKX); -- Эксклюзивная
-- или
SELECT * FROM <имя_таблицы> WITH (TABLOCK);   -- Разделяемая
COMMIT; -- Освобождает блокировку
```

- TABLLOCKX — запрещает любые параллельные операции.
- TABLOCK — разрешено параллельное чтение.

Для **MYSQL** используется такой синтаксис:

```
LOCK TABLES
  <имя_таблицы1> [AS <псевдоним>] READ [LOCAL] | WRITE,
  <имя_таблицы2> READ,
  ...
[WAIT n | NOWAIT];
```

- READ разрешает чтение всем, блокирует запись для других.
- READ LOCAL разрешает INSERT другим транзакциям (только для движка MyISAM, в InnoDb не поддерживается!).
- WRITE: эксклюзивная блокировка (запрет всех операций другим).
- WAIT n устанавливает максимальное время ожидание n секунд.
- NOWAIT — указывает, что ожидать освобождения ресурсов не требуется.

PostgreSQL радует богатым набором опций:

```
LOCK TABLE <имя_таблицы> [AS <псевдоним>]
IN <режим_блокировки> MODE [NOWAIT];
```

Ниже перечислены некоторые распространенные режимы блокировки:

- ACCESS SHARE (самый слабый) — разрешает любые запросы, кроме ALTER TABLE и DROP TABLE.
- ROW EXCLUSIVE — разрешает чтение, блокирует изменения структуры. Автоматически (по умолчанию) устанавливается при INSERT/UPDATE/DELETE.
- SHARE UPDATE EXCLUSIVE — блокирует VACUUM, ALTER TABLE, создание индексов.
- SHARE — разрешает чтение, блокирует все изменения данных и структуры.
- EXCLUSIVE — разрешает только чтение без блокировок, блокирует все изменения и SELECT ... FOR UPDATE/SHARE.
- ACCESS EXCLUSIVE (самый строгий) — полная блокировка (DDL, VACUUM, все запросы).

12.6.4. Отключение блокировок (READ UNCOMMITTED)

Для очень больших и долгих отчетов, где абсолютная точность на текущий момент не критична, а скорость выполнения важна, можно применять режим чтения без блокировок READ UNCOMMITTED (или подсказка NOLOCK в MS SQL). Такой запрос выполняется, не мешая работе оперативных транзакций, и выполнится быстрее. Ценой данного выигрыша является риск чтения "грязных" (незафиксированных) данных, неповторяющихся чтений, фантомов. Данные могут быть внутренне несогласованными.

Этот режим можно также использовать при чтении системных таблиц и представлений СУБД для чтения метаданных, где конфликты маловероятны или некритичны.

Также в случае очень высокой конкуренции на "горячих" данных подобный режим может использоваться в качестве отчаянной меры, когда пессимистические блокировки приводят к невыносимым задержкам или тупикам, а потеря точности приемлема.

Использование READ UNCOMMITTED / NOLOCK должно быть сильно обосновано и тщательно протестировано, так как оно нарушает базовые принципы ACID и может привести к трудноуловимым ошибкам и несогла-

SKLADCHIK.ORG

сованности данных в приложении. Всегда сначала ищите другие пути оптимизации (индексы, нормализация, MVCC, кеширование).

12.6.5. Оптимистичная блокировка

Описанные выше уровни изоляции и методы блокировки являются пессимистичными, то есть предполагают, что во время выполнения транзакции будет наблюдаться конкуренция (другие транзакции будут конкурировать за изменяемые данные), поэтому доступ к ресурсам контролируется и блокируется заблаговременно.

> **Оптимистичная блокировка** не является, строго говоря, встроенным механизмом СУБД, скорее техникой обеспечения согласованности данных, который позволяет нескольким транзакциям работать с одними и теми же ресурсами одновременно, при этом надеясь, что конфликты редки и возникнут крайне маловероятно (оптимистичный подход).

Исходя из этого предположения, блокировки заранее не ставятся, и клиенты спокойно выполняют свои операции. Лишь в самом конце, при фиксации изменений, проверяется, не изменилось ли состояние данных с момента последнего чтения. Если данные успели поменяться (другая транзакция успела внести изменения), оптимистичная блокировка инициирует повтор операции или уведомляет пользователя о возникновении конфликта.

Этот подход ориентирован на минимальные блокировки и максимизирует пропускную способность системы, избегая долгого ожидания при конкурентном доступе к данным.

Добавим в таблицу **accounts** еще один столбец:

```
ALTER TABLE accounts ADD version INT NOT NULL DEFAULT 0;
-- Возвращение первоначальной суммы
UPDATE accounts SET balance = 1000.0 WHERE account = 101;
```

Мы добавили столбец с версией данных, который будем контролировать вручную следующим образом: при успешном изменении версия увеличивается на единицу, в случае ошибки — остается с прежним значением.

Для запоминания версии можно вместо временной таблицы использовать переменные, но об этом мы поговорим в следующей главе.

```
-- Оптимистичная блокировка
-- Для MySQL BEGIN или START TRANSACTION
BEGIN TRANSACTION;

-- Для MS SQL уберите ключевое слово TEMPORARY
CREATE TEMPORARY TABLE existing_data(version INT);

-- Читаем данные и запоминаем версию
INSERT INTO existing_data
SELECT version
FROM accounts
WHERE account = 101;


UPDATE accounts
SET balance = balance - 500, version = version + 1
WHERE account = 101 AND
-- Ключевая проверка версия должна соответствовать сохраненному значению
-- Для MS SQL Server пользуйтесь функцией TOP(1)
version = (SELECT version FROM existing_data LIMIT 1);

DROP TABLE existing_data;

COMMIT;
```

Мы надеемся, что данные в ходе выполнения транзакции не изменились. Мы запомнили версию данных во временной таблице и затем использовали условие WHERE, где уточняем, что данные нужно обновить только в случае, если их версия осталась прежней.

ПРИМЕЧАНИЕ.

Всегда явно удаляйте временные таблицы при помощи вызова DROP TABLE.

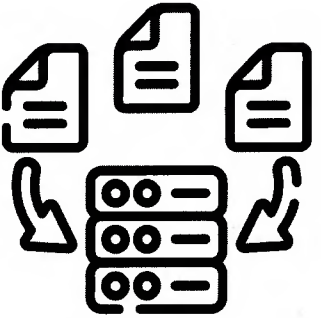
Что узнали и чему мы научились?

Мы ознакомились с фундаментальными механизмами обеспечения целостности данных в многопользовательских средах, с принципами ACID, лежащими в основе транзакционной модели, с жизненным циклом транзакции от BEGIN до COMMIT/ROLLBACK.

Мы рассмотрели компромиссы между согласованностью и производительностью через уровни изоляции и познакомились со стратегиями управления блокировками для предотвращения конфликтов.

Транзакции — один из механизмов, отвечающих за бизнес-логику всей системы, однако теперь давайте зададим себе важный вопрос: **как систематизировать и упростить внедрение бизнес-правил и логики в наших проектах, построенных на SQL?**

Ведь известно, что хорошо спроектированная архитектура системы включает четкий слой бизнес-логики, отделенный от прочих слоев (вроде слоя данных или слоя пользовательских представлений). Именно здесь на помощь приходят **хранимые процедуры** и **функции**, позволяющие перенести большую часть вычислительной логики на сторону базы данных.



Глава 13.

Бизнес-логика. Хранимые процедуры и функции

В предыдущих главах мы рассмотрели SQL как средство непосредственного доступа и манипуляций с данными, изучили концепцию транзакций, которая позволила связывать несколько запросов в единую группу операций. Однако простое выполнение одиночных запросов очень часто бывает недостаточным для реализации полноценной бизнес-логики.

Часто приходится учитывать разнообразные условия, осуществлять многократные итерации и обеспечивать надежную обработку возможных исключительных ситуаций и ошибок.

Именно здесь на сцену выходят **хранимые процедуры и функции**, позволяющие организовывать и контролировать исполнение сложных последовательностей запросов, включая условия, циклы и обработку ошибок. Они превращают базу данных в активный элемент информационной системы, способный решать практические задачи без участия внешнего приложения.

Благодаря хранимым процедурам и функциям можно существенно упростить взаимодействие с данными, уменьшить объем передаваемого трафика и повысить производительность приложения.

Более того, размещение логики на стороне базы данных позволяет унифицировать поведение и избавиться от необходимости повторять однотипные операции в различных частях клиентских приложений.

13.1. Синтаксис создания

Если совсем упростить, **хранимая процедура** — это просто набор команд, которые выполняются друг за другом. С этой концепцией мы уже познакомились, когда изучали транзакции.

К сожалению, каждая из СУБД имеет свои синтаксические нюансы, однако общей для всех остается команда CREATE PROCEDURE.

MS SQL Server

```
CREATE PROCEDURE hello_sql
AS
BEGIN
  CREATE TABLE newtable (message VARCHAR(20));
  INSERT INTO newtable VALUES ('Hello, SQL!');
END;
GO
```

Между ключевыми BEGIN и END располагается последовательность команд, включенных в процедуру. В нашем случае создается таблица **newtable**, к которую добавляется сообщение *Hello, SQL!*.

GO в MS SQL Server — это специальная служебная команда, которая разделяет вводимые вами команды на отдельные пакеты отправки к серверу. Эта команда не является частью SQL, а служит сигналом программе-клиенту (например, SSMS), что отправленная ей серия команд закончена и нужно отправить их серверу как отдельный пакет. Благодаря такому подходу мы можем не экранировать символы точки с запятой внутри тела процедуры даже через программу-клиент с текстовым интерфейсом (например **sqlcmd**).

Чтобы вызвать процедуру, используйте команду вызова:

```
EXEC hello_sql;
```

Внешне ничего не произошло, но если вы выполните команду:

```
SELECT * FROM newtable;
```

то получите результат с содержимым вновь созданной таблицы:

message
Hello, SQL!

MySQL

```
DELIMITER $$
CREATE PROCEDURE hello_sql()
BEGIN
  CREATE TABLE newtable (message VARCHAR(20));
  INSERT INTO newtable VALUES ('Hello, SQL!');
END$$
DELIMITER ;
```

Ключевым отличием в данном случае служит использование команды DELIMITER, после которой указаны символы $$. Этой командой вы говорите СУБД, что в качестве разделителя команд при непосредственном вводе будет использоваться комбинация символов $$. В принципе, можно использовать любую комбинацию, лишь бы она не совпадала с зарезервированными словами. Такой прием нужен для того, чтобы разделять окончание команды CREATE PROCEDURE и символы точки с запятой, разделяющие команды внутри самой процедуры.

Также обратите внимание, что за именем процедуры идут открывающие и закрывающие скобки (), в них указываются параметры, передаваемые в процедуру. Поскольку в данном случае параметров нет, внутри скобок тоже пусто.

Для вызова процедуры используйте команду вызова:

```
CALL hello_sql();
```

Чтобы удостовериться, что процедура сработала как нужно, выполните команду:

```
SELECT * from newtable;
```

PostgreSQL

Для PostgreSQL синтаксис чуть сложнее:

```
CREATE PROCEDURE hello_sql()
LANGUAGE plpgsql
AS $$
BEGIN
  CREATE TABLE newtable (message VARCHAR(20));
  INSERT INTO newtable VALUES ('Hello, SQL!');
END;
$$;
```

Для вызова используется команда CALL:

```
CALL hello_sql();
```

Основное существенное отличие — наличие конструкции с ключевым словом LANGUAGE (язык), за которым следует указание, собственно, языка **plpgsql**.

Все дело в том, что процедуры и функции в Postgres можно писать не только командами SQL, но и на других языках программирования:

1. **plpgsql** (Procedural Language for PostgreSQL) — самый популярный язык процедурного программирования, встроенный в PostgreSQL. Используется для создания хранимых процедур, функций и триггеров.
2. **plpythonu/plpython3u** — позволяет использовать Python для написания процедур и функций. Удобен для сложных расчетов и анализа данных.
3. **pltcl** — язык Tcl, который может пригодиться для интеграции существующих TCL-кодов.

4. **plperl** — интеграция Perl для написания функций и процедур.
5. **plv8** — JavaScript-поддерживаемый язык, полезный для фронтенд-разработчиков, которым удобнее мыслить в терминах JS.
6. **plsh** — bash-скрипты могут быть использованы для простых административных задач и быстрой обработки данных.

ПРИМЕЧАНИЕ.

Поскольку данная книга посвящена языку SQL, все дальнейшие примеры будут приводиться с использованием **plpgsql**.

Символы $$ служат для экранирования спецсимволов внутри текста процедуры. Указание $$ сообщают серверу PostgreSQL: все, что находится между этими знаками, является телом процедуры или функции.

Для удобства можно использовать и другие комбинации, например:

```
...
AS $HELLO$
BEGIN
...-- Тело процедуры или функции
END;
$HELLO$;
```

Важно, чтобы маркеры начала и конца были одинаковыми.

Конструкцию с указанием языка LANGUAGE можно указывать как до тела процедуры, так и после:

```
CREATE PROCEDURE hello_sql()
AS $$
BEGIN
...
END;
$$ LANGUAGE plpgsql;
```

SKLADCHIK.ORG

13.2. Переменные

Переменные — именованные ячейки в памяти, которые используются для хранения данных на время выполнения хранимых процедур и функций. Они позволяют хранить промежуточные результаты, проводить вычисления и затем использовать эти данные в дальнейшем коде.

Переменные отличаются от **столбцов** таблиц тем, что их жизнь ограничена рамками одной сессии выполнения процедуры или функции, что позволяет избежать лишнего использования ресурсов для хранения временных значений.

Переменные в SQL имеют несколько значительных преимуществ по сравнению с хранением данных в таблицах.

Во-первых, они существенно упрощают процесс манипуляций с данными. При помощи переменных можно легко сохранять промежуточные результаты, выполнять операции и затем использовать их, что делает код более читаемым и понятным.

Во-вторых, использование переменных позволяет избежать избыточности данных. Если результат, необходимый для дальнейших вычислений, можно сформировать с помощью переменной, то нет необходимости создавать временные таблицы, что значительно облегчает управление данными и может повысить производительность.

Переменные особенно полезны при работе с большими объемами данных — с их помощью можно существенно сократить количество операций ввода-вывода с базой данных. Это также позволяет избежать блокировок и удержания ресурсов, что, в свою очередь, делает выполнение запросов более быстрым и эффективным.

Разные СУБД по-разному регламентируют использование переменных. Если в MS SQL Server переменные можно объявлять и использовать вне про-

цедур и функций, то в MySQL их можно использовать только внутри тела процедуры.

PostgreSQL разрешает использование переменных либо так же — внутри процедур и функций, либо внутри особых блоков DO:

```
DO $$
BEGIN
   -- код PL/PgSQL
END;
$$ language plpgsql;
```

Блоки DO бывают полезны, когда требуется сделать небольшую операцию единовременно, но при этом нет необходимости создавать постоянную процедуру или функцию, особенно в интерактивных сессиях командной строки *psql* или утилиты администрирования базы данных (pgAdmin).

Прежде чем использовать переменную, ее надо объявить — указать СУБД ее имя и тип данных. Мы можем объявлять переменные на 2 уровнях — уровне процедуры и уровне сеанса.

В **MS SQL Server** уровень видимости переменной зависит от места, где она была объявлена. Объявление переменной производится при помощи команды DECLARE:

```
DECLARE @number INT;
```

В MSSQL использование знака @ перед именем переменной обязательно. Так синтаксически СУБД отличает переменные от имен других объектов.

Переменная после объявления будет иметь значение NULL. Мы можем присвоить значение переменной командой SET:

```
SET @number = 10;
```

Теперь мы можем использовать переменную в расчетах:

```
SELECT @number + 2 AS Result; -- 12
```

Мы можем присваивать значения переменной сразу при инициализации:

```
DECLARE @number INT = 10;
```

Если переменная объявлена внутри тела хранимой процедуры, к ней нельзя обратиться извне. Если же переменная была объявлена вне процедуры, доступ к ней будут иметь любые запросы в рамках текущего сеанса (если отключиться и подключиться заново, переменная исчезнет, вместе со своим значением).

В **MySQL** правила объявления переменных следующие:

```
CREATE PROCEDURE MyProcedure()
BEGIN
  DECLARE myVariable INT DEFAULT 0; -- Объявление
  SET myVariable = myVariable + 10;  -- Использование
  SELECT myVariable; -- Вывод 10
END;
```

Значение по умолчанию указывается при помощи ключевого слова DEFAULT.

Переменные, объявленные таким образом, будут видны только внутри процедуры. В то же время, если требуется объявить переменную уровня сеанса, нужно использовать символ @.

Для объявления переменных сеанса не требуется использовать ключевое слово DECLARE:

```
SET @mySecondVariable = 100; -- объявление и инициализация
SELECT @mySecondVariable AS Value;
```

В **PostgreSQL** переменные могут существовать либо в пределах процедуры или функции, где она объявлена, либо внутри блока DO.

```
DO $$
DECLARE
  myVariable INT := 100; -- Объявление и инициализация
BEGIN
  RAISE INFO 'Value: %', myVariable; -- Вывод значения
END $$;
```

ПРИМЕЧАНИЕ.

В PostgreSQL содержимое переменной при помощи SELECT вывести нельзя. Вместо этого используется инструкция RAISE, при помощи которой можно в консоль вывести отладочные сообщения, как в этом примере.

Хотя переменных уровня сеанса в PostgreSQL как таковых нет, существует их альтернатива:

```
-- Объявить и инициализировать
SET my.SessionVar TO 'Hello';

-- Получить значение
-- SHOW my.SessionVar;
-- Альтернативно
-- SELECT current_setting('my.SessionVar')

-- Сброс значений
-- RESET;
```

Строго говоря, механизм настроек не является реализацией переменных уровня сеанса. Вы просто имеете доступ к словарю (набору записей вида ключ-значение), содержащему только строковые данные (даже если вы попытаетесь сохранить число таким образом, оно все равно будет преобразовано к строке).

Для отображения значения можно воспользоваться инструкцией *SHOW*, если же требуется выполнить со значением какие-либо манипуляции, воспользуйтесь встроенной функцией *current_setting*.

Дополнительно обратите внимание, что перед названием переменной мы указали "my." — это так называемое *пространство имен* (**namespace**). PostgreSQL строго относится к использованию нестандартных переменных.

По умолчанию любые имена переменных уровня сеанса воспринимаются как параметры конфигурации PostgreSQL. К примеру, если вы попробуете задать переменную без префикса пространства имен:

```
SET SessionVar TO 'Hello';
```

система попытается интерпретировать это как попытку изменить настройку внутреннего параметра, которого, скорее всего, не существует. В результате получите ошибку. Вы можете использовать любое имя для обозначения пространства имен, кроме, разумеется, тех, что уже зарезервированы.

13.3. Параметры

Создадим простую таблицу с ценами на товары:

```
CREATE TABLE products
(
  product_id INT NOT NULL PRIMARY KEY,
  base_price DECIMAL(12,2) DEFAULT 0.0,
  current_price DECIMAL(12,2) DEFAULT 0.0
);

INSERT INTO products VALUES
 (1, 100, 100),
 (2, 120, 120),
 (3, 150, 150);
```

Предположим, мы хотим сделать скидку в 5% на все товары. Мы очень легко можем это сделать при помощи запроса:

```
UPDATE products SET current_price = base_price * 0.95;
```

Но как быть, если мы не знаем, какая скидка будет установлена завтра или послезавтра. Сегодня скидка будет 5%, завтра на 7%, а послезавтра — 2%.

Мы можем создать процедуру *set_discount* и передать ей нужный множитель. Параметры передаются в скобках с указанием типа данных. Внутри

процедуры к значению переданного параметра можно обратиться, указав его имя.

MS SQL Server

```
CREATE PROCEDURE set_discount
  (@discount_percent INT)
AS
BEGIN
  UPDATE products
  SET current_price = base_price * (100.0 - @discount_percent) / 100.0;
END;

-- Вызов процедуры
EXEC set_discount 5;
```

MySQL

```
DELIMITER //
CREATE PROCEDURE set_discount
  (discount_percent INT)
BEGIN
  UPDATE products
  SET current_price = base_price * (100.0 - discount_percent) / 100.0;
END//
DELIMITER ;

-- Вызов процедуры
CALL set_discount(5);
```

PostgreSQL

```
CREATE OR REPLACE PROCEDURE set_discount(discount_percent INT)
LANGUAGE plpgsql
AS $$
BEGIN
  UPDATE products
  SET current_price = base_price * (100.0 - discount_percent) / 100.0;
```

```
END;
$$;

-- Вызов процедуры
CALL set_ discount(5);
```

Обратите внимание на замечательную особенность PostgreSQL — мы можем указать CREATE OR REPLACE (создай или замени) вместо просто CREATE и таким образом избавимся от необходимости удалять процедуру перед ее изменением (DROP PROCEDURE...).

Мы передаем в процедуру параметр discount_percent и далее используем переданное значение для расчета скидки.

Внутри процедуры параметр в плане использования ничем не отличается от переменной.

13.4. Типы параметров, функции

13.4.1. Входные (IN), выходные (OUT) и смешанные (INOUT) параметры

Мы научились передавать в процедуры параметры, но что если мы хотим вернуть что-либо из процедуры? Все рассматриваемые нами СУБД умеют это делать. В предыдущем разделе мы использовали **входной** параметр (значение которого входит в процедуру). Существуют и **выходные** параметры — те, значения которых изменяются процедурами и могут быть использованы вне самой процедуры.

Предположим, у нас есть расписание приема у врача, где хранится время начала и ид пациента:

```
CREATE TABLE appointments(
  id INT PRIMARY KEY,
  start_hour INT NOT NULL,
  patient_id INT);
```

```
INSERT INTO appointments VALUES
(1, 10, 1),(2, 12, 3),(3, 16, 2),(4, 17, 4);
```

Нам необходимо записать нового пациента (id = 5) на прием на любое свободное время.

Мы можем найти свободное время для приема при помощи запроса:

```
-- Все доступные часы
WITH all_hours AS (
  SELECT 9 AS hour
  UNION ALL SELECT 10 UNION ALL SELECT 11 UNION ALL SELECT 12
  UNION ALL SELECT 13 UNION ALL SELECT 14 UNION ALL SELECT 15
  UNION ALL SELECT 16 UNION ALL SELECT 17 UNION ALL SELECT 18
  )
SELECT MIN(all_hours.hour) AS first_free_slot
FROM all_hours
LEFT JOIN appointments ON all_hours.hour = appointments.start_
hour
WHERE appointments.start_hour IS NULL;
```

Теперь давайте создадим процедуру на основании этого запроса.

Синтаксис **MS SQL**:

```
CREATE PROCEDURE book_appointment (
  @p_patient_id INT,
  @p_start_hour INT OUTPUT
)
AS
BEGIN
  -- Генерируем список часов с 9 до 18
  WITH all_hours AS (
    SELECT 9 AS hour
    UNION ALL SELECT 10 UNION ALL SELECT 11 UNION ALL SELECT 12
    UNION ALL SELECT 13 UNION ALL SELECT 14 UNION ALL SELECT 15
    UNION ALL SELECT 16 UNION ALL SELECT 17 UNION ALL SELECT 18
  )
  SELECT @p_start_hour = MIN(all_hours.hour)
```

```
  FROM all_hours
  LEFT JOIN appointments ON all_hours.hour = appointments.start_hour
  WHERE appointments.start_hour IS NULL;
  -- Создаем новый уникальный ID
  DECLARE @new_id INT;
  SET @new_id = (SELECT COALESCE(MAX(id), 0) + 1 FROM appointments);
  INSERT INTO appointments (id, start_hour, patient_id)
    VALUES (@new_id, @p_start_hour, @p_patient_id);
END;
```

Первое, что мы видим в коде создания процедуры, — параметр @p_start_hour с ключевым словом **OUTPUT**. Это говорит СУБД, что процедура должна присвоить значение указанному параметру по ходу своего исполнения.

Еще одним заслуживающим внимания моментом, отличающимся от других СУБД, служит синтаксис присвоения значения переменной:

```
SELECT @p_start_hour = MIN(all_hours.hour)…
```

В инструкции SELECT, таким образом, не обязательно использовать инструкцию SET для присваивания значения. Вот как выглядит назначение пациента с использованием данной процедуры:

```
-- Подготовить переменную для передачи в процедуру
DECLARE @appointed_time INT;
EXEC book_appointment 5, @appointed_time OUTPUT;
SELECT @appointed_time;
```

Мы объявляем переменную, которая получит значение назначенного времени, затем вызываем процедуру назначения, указав ид нового пациента 5, дополнительно передавая параметр @appointed_time со спецификатором OUTPUT (это обязательно, иначе СУБД не присвоит найденное значение).

Если мы теперь проверим содержимое таблицы, то увидим, что к ней добавилась новая запись:

id	start_hour	patient_id
5	9	5

Проделаем то же самое в **MySQL**.

С учетом синтаксических особенностей, наша процедура приобретет следующий вид:

```
DELIMITER //
CREATE PROCEDURE book_appointment (
  IN p_patient_id INT,
  OUT p_start_hour INT
)
BEGIN
  -- Объявляем нужную переменную
  DECLARE new_id INT;
  -- Генерируем список часов с 9 до 18
  WITH all_hours AS (
    SELECT 9 AS hour
    UNION ALL SELECT 10 UNION ALL SELECT 11 UNION ALL SELECT 12
    UNION ALL SELECT 13 UNION ALL SELECT 14 UNION ALL SELECT 15
    UNION ALL SELECT 16 UNION ALL SELECT 17 UNION ALL SELECT 18
  )
  SELECT MIN(all_hours.hour) INTO p_start_hour
  FROM all_hours
  LEFT JOIN appointments ON all_hours.hour = appointments.start_hour
  WHERE appointments.start_hour IS NULL;
  -- Создаем новый уникальный ID
  SET new_id = (SELECT COALESCE(MAX(id), 0) + 1 FROM appointments);
  INSERT INTO appointments (id, start_hour, patient_id)
    VALUES (new_id, p_start_hour, p_patient_id);
END//
DELIMITER ;
```

Чтобы проверить работу процедуры, как и в MS SQL, создадим переменную, в которую будет записано значение назначенного времени:

```
SET @appointed_time = 0;
CALL book_appointment(5, @appointed_time);
SELECT @appointed_time; -- 9
```

SKLADCHIK.ORG

Обратите внимание, что параметры в нашей процедуры промаркированы:

```
IN p_patient_id INT,
OUT p_start_hour INT
```

- **IN** (можно не указывать, используется по умолчанию) — говорит СУБД, что параметр будет передан извне.
- **OUT** — говорит СУБД, что процедура должна присвоить значение передаваемому параметру для последующего использования вне процедуры.

Также существует тип **INOUT**, который комбинирует обе настройки — такие параметры можно использовать внутри процедуры и присваивать им значения для использования за ее пределами.

ПРИМЕЧАНИЕ.

В MSSQL тип INOUT отсутствует.

Переменная *new_id* должна быть задекларирована в начале процедуры, сразу после BEGIN. Присваивание ей значения происходит через конструкцию INTO:

```
SELECT MIN(all_hours.hour) INTO p_start_hour…
```

Если запросим данные таблицы, получим идентичный результат:

```
SELECT * FROM appointments WHERE patient_id = 5;
```

id	start_hour	patient_id
5	9	5

Создадим такую же процедуру в **PostgreSQL**:

```
CREATE OR REPLACE PROCEDURE book_appointment (
  IN p_patient_id INT,
  OUT p_start_hour INT
)
AS $$
```

```
DECLARE
  -- Объявляем нужную переменную
  new_id INT;
BEGIN
  -- Генерируем список часов с 9 до 18
  WITH all_hours AS (
    SELECT 9 AS hour
    UNION ALL SELECT 10 UNION ALL SELECT 11 UNION ALL SELECT 12
    UNION ALL SELECT 13 UNION ALL SELECT 14 UNION ALL SELECT 15
    UNION ALL SELECT 16 UNION ALL SELECT 17 UNION ALL SELECT 18
  )
  SELECT MIN(all_hours.hour) INTO p_start_hour
  FROM all_hours
  LEFT JOIN appointments ON all_hours.hour = appointments.start_hour
  WHERE appointments.start_hour IS NULL;

  -- Создаем новый уникальный ID
  new_id := (SELECT COALESCE(MAX(id), 0) + 1 FROM appointments);
  INSERT INTO appointments (id, start_hour, patient_id)
    VALUES (new_id, p_start_hour, p_patient_id);
END;
$$ LANGUAGE plpgsql
```

В данном коде следует обратить внимание на следующие синтаксические различия:

1. Блок с объявлениями переменных должен быть указан до начала процедуры (BEGIN).

2. Присвоение значения переменной может осуществляться через конструкцию SELECT … INTO:

```
  SELECT MIN(all_hours.hour) INTO p_start_hour
```

3. Альтернативно можно использовать оператор присваивания (:=).

```
  new_id := (SELECT …)
```

Выражение SELECT должно быть заключено в скобки.

Чтобы воспользоваться процедурой, необходимо создать блок DO, внутри которого объявить переменную, которая будет использоваться в качестве OUT-параметра для процедуры. Обратите внимание, что блок объявления переменных здесь так же должен быть указан до начала блока с кодом, то есть до ключевого слова BEGIN.

```
DO $$
DECLARE appointed_time INT;
BEGIN
  CALL book_appointment(5, appointed_time);
  RAISE INFO 'Назначенное время: %', appointed_time;
END
$$;
```

Перегрузка в PostgreSQL

В PostgreSQL допускается **перегрузка** для процедур и функций (функции рассмотрены в следующем подразделе).

Перегрузка — это возможность вызвать процедуру или функцию с разным набором параметров.

Рассмотрим пример:

```
-- Первая версия функции принимает строку
CREATE FUNCTION greetings(name TEXT)
RETURNS TEXT AS $$
BEGIN
  RETURN 'Привет, ' || name;
END;
$$ LANGUAGE plpgsql;

-- Вторая версия функции принимает числовой аргумент
CREATE FUNCTION greetings(number INTEGER)
RETURNS TEXT AS $$
BEGIN
  RETURN 'Ваш номер: ' || number;
```

```
END;
$$ LANGUAGE plpgsql;
```

Одна и та же процедура *greetings* использует разные наборы параметров. В PostgreSQL практика перегрузки активно поощряется и приветствуется.

13.4.2. Функции

Мы использовали процедуру для возврата значения. Однако данный результат нельзя использовать напрямую в другом SQL-выражении. Мы сначала должны объявлять переменную вне процедуры, затем вызвать процедуру, а уже затем пользоваться полученным результатом. Если все, что нам нужно, — это результат, мы можем использовать **функции**. Если совсем упростить, функции — это процедуры, которые сами возвращают значение. Если чуть углубиться, то вот список различий между функциями и процедурами:

Характеристика	Функции	Процедуры
Основная цель	Возврат значения	Выполнение действий
Использование в SQL	Непосредственно в выражении	Отдельным вызовом (CALL/EXEC)
Возвращаемое значение	Обязательно	Не обязательно через OUT-параметры
Побочные эффекты	Не рекомендуются	Ожидаемы
Транзакции	Работают в рамках вызвавшей транзакции	Могут сами управлять транзакциями

Функции идеальны для детерминированных операций (расчеты, преобразования данных), где не требуется изменение состояния БД. От функций не ожидаются какие-либо "побочные эффекты" (изменение состояния таблиц, структуры данных), до такой степени, что внутри в MS SQL Server функции не могут выполнять DML-операции (INSERT/UPDATE/DELETE), что гаран-

тирует их безопасность в запросах. В то же время внутри процедур такие изменения вполне ожидаемы.

> Главное правило: используйте функции для **вычислений**, процедуры для **операций**. Если нужно и то, и другое — лучше разделить функционал на функцию для вычислений и процедуру, которая ее использует, вместе с операциями изменения данных.

Попробуем подсчитать, сколько пациентов посетит врача в заданном временном интервале. Мы просто вызовем COUNT(*) между двумя переданными в качестве параметров часами.

Для создания функции используется команда CREATE FUNCTION. Ее синтаксис практически не отличается от синтаксиса создания процедуры.

Основным отличием служат обозначения типа возвращаемого значения (RETURNS) и обязательного наличия ключевого слова RETURN внутри тела функции — команды на возврат значения и выход из функции.

MS SQL Server

Синтаксис для **MS SQL Server**:

```
CREATE FUNCTION dbo.count_patients_in_range (
  @from_hour INT,
  @to_hour INT)
RETURNS INT
AS
BEGIN
 RETURN (
  SELECT COUNT(*)
  FROM appointments
  WHERE start_hour BETWEEN @from_hour AND @to_hour
 );
END;
```

Функции, как и процедуры, принимают внешние параметры. В нашем случае *from_hour* и *to_hour* — час начала и час окончания искомого интерва-

ла времени. Мы также указываем обязательное для функции условие — тип возвращаемого значения (RETURNS INT).

В теле функции мы просто возвращаем (RETURN) результат запроса, высчитывающего количество пациентов между часом начала и часом окончания.

MySQL

```
DELIMITER //
CREATE FUNCTION count_patients_in_range(from_hour INT, to_hour INT)
RETURNS INT READS SQL DATA
BEGIN
  RETURN (
   SELECT COUNT(*)
   FROM appointments
   WHERE start_hour BETWEEN from_hour AND to_hour
 );
END //
DELIMITER ;
```

По содержимому функция идентична. Однако в MySQL для функций существует дополнительная необязательная настройка (READS SQL DATA), на которой следует остановиться поподробнее. Данная настройка очень важна для работы оптимизатора запросов MySQL. Хотя она не является обязательной, все же ее рекомендуется применять явным образом. Это позволит оптимизатору строить более эффективные планы выполнения запросов с использованием данной функции.

Мы указали READS SQL DATA. Использование этого атрибута позволит оптимизатору запросов и движку MySQL корректно оценивать зависимость функции от базы данных.

Например, если вы планируете строить индексы на вычисляемых столбцах или использовать функцию в выражениях индекса, это позволит MySQL принять правильные меры предосторожности.

При объявлении функции вы можете указать один из следующих атрибутов:

- **DETERMINISTIC** — указывает на свойство детерминированности функции. Это значит, что при одних и тех же входных данных функция всегда возвращает одинаковое значение. Другими словами, если дважды вызвать функцию с одними и теми же аргументами, результат будет тот же. Например, функция возведения в квадрат — в любое время один и тот же аргумент даст один и тот же результат.
- **NOT DETERMINISTIC** — обратное значение DETERMINISTIC. Означает, что функция может выдавать разные результаты при одних и тех же входных данных. Часто используется, если функция обращается к внешним источникам данных, зависит от времени или использует генератор случайных чисел.
- **READS SQL DATA** — наш случай. Показывает, что функция считывает данные из базы, но не изменяет их. Это важно для MySQL, чтобы понимать, нуждается ли функция в доступе к данным, и влиять на планировщик запросов.
- **MODIFIES SQL DATA** — указывает, что функция изменяет данные в базе (обновляет, удаляет или вставляет записи). Это запрещает использование таких функций в выражениях индексов и ограничивает их использование в операциях с участием индексов.
- **NO SQL** — атрибут, говорящий о том, что функция не выполняет никаких SQL-запросов. Подходит для простых функций, производящих исключительно математические или строковые манипуляции (например, форматирование).
- **CONTAINS SQL** — данный атрибут используется редко и означает, что функция сама не выполняет SQL-запросы, но потенциально может вызвать другую функцию или процедуру, которая выполняет SQL.

ПРИМЕЧАНИЕ.

Поскольку в MS SQL Server операции DML запрещены внутри функций, аналогов данному атрибуту нет. Оптимизатор полагается на внутреннюю классификацию функций и процессов, а разработчики должны сами соблюдать правила и требования для повышения производительности и стабильности.

PostgreSQL

```
CREATE OR REPLACE FUNCTION count_patients_in_range(
    from_hour INT,
    to_hour INT)
RETURNS INT AS $$
BEGIN
  RETURN (
    SELECT COUNT(*)
    FROM appointments
    WHERE start_hour BETWEEN from_hour AND to_hour
  );
END;
$$ LANGUAGE plpgsql STABLE;
```

Аналогично примеру с MySQL, в PostgreSQL имеются атрибуты для функций (в нашем случае указано значение STABLE), которые рекомендуется указывать явно, чтобы облегчить и сделать более эффективной работу оптимизатора запросов.

Существует три вида значений:

1. **IMMUTABLE** — используйте этот атрибут, если ваша функция вычисляется исключительно на основании своих аргументов и не зависит от состояния базы данных или других факторов окружающей среды. Аналог DETERMINISTIC в MySQL.
2. **STABLE** — этот атрибут выбирается, если функция считывает данные из базы данных, но результат ее выполнения остается постоянным в рамках одной транзакции. Например, функция CURRENT_DATE, возвращающая текущую дату, которая остается стабильной в рамках одной транзакции, но меняется от транзакции к транзакции.
3. **VOLATILE** — используйте этот атрибут, если функция изменяет данные или структуру данных либо использует внешние источники данных.

Использование функций для всех трех СУБД одинаково — вы просто указываете эту функцию в запросе:

```
-- Количество пациентов с 10-го по 15-й час
SELECT count_patients_in_range(10, 15);
```

ПРИМЕЧАНИЕ.

Для MS SQL Server, возможно, потребуется явно указать схему данных (dbo) в силу особенностей реализации:

```
SELECT dbo.count_patients_in_range(10, 15);
```

13.4.3. Скалярные и табличные результаты

До сих пор мы использовали так называемые **скалярные** (scalar) результаты, представляющие собой какое-то одно значение (число, строку, дату и пр.). Функции и процедуры могут возвращать и целые наборы данных (табличные результаты).

Создадим функцию, возвращающую время и код пациента из таблицы **appointments**.

MS SQL Server

Чтобы указать, что функция или процедура возвращает таблицу, необходимо явно это указать в объявлении (RETURNS TABLE).

```
-- Табличный результат
CREATE FUNCTION all_appointments ()
RETURNS TABLE
AS
  RETURN (
  SELECT start_hour, patient_id
  FROM appointments
  WHERE patient_id IS NOT NULL
  );
-- Использование результатов
-- как любой таблицы или представления
SELECT * FROM dbo.all_appointments();
```

MySQL

В **MySQL** функции не могут возвращать табличные результаты, для этих целей необходимо использовать процедуру, внутри которой достаточно просто вызвать SELECT внутри тела процедуры:

```
DELIMITER \\
CREATE PROCEDURE all_appointments()
BEGIN
  SELECT start_hour, patient_id
  FROM appointments
  WHERE patient_id IS NOT NULL;
END\\
DELIMITER ;
```

Недостатком **MySQL** в данном случае является то, что результат такой процедуры нельзя использовать напрямую в SQL-запросе. Вместо этого приходится использовать промежуточную временную таблицу:

```
-- Шаг 1: Создать временную таблицу для результатов
CREATE TEMPORARY TABLE temp_appointments (
  start_hour DATETIME,
  patient_id INT
);

-- Шаг 2: Вызвать процедуру и вставить данные
INSERT INTO temp_appointments
CALL all_appointments();

-- Теперь можно использовать temp_appointments в запросах
SELECT * FROM temp_appointments;
```

PostgreSQL

В **PostgreSQL** аналогом описанных ранее функций будет следующий вариант:

```
CREATE OR REPLACE FUNCTION all_appointments()
RETURNS TABLE (start_hour INT, patient_id INT)
AS $$
```

SKLADCHIK.ORG


```
  SELECT start_hour, patient_id
  FROM appointments
  WHERE patient_id IS NOT NULL
$$ LANGUAGE SQL;
```

Основным отличием от MS SQL Server служит то, что в PostgreSQL нужно явно указывать структуру возвращаемой таблицы: RETURNS TABLE (start_hour INT, patient_id INT).

В то же время PostgreSQL предоставляет альтернативу в виде SETOF.

```
CREATE TYPE appointment_result AS (
  start_hour INT,
  patient_id INT
);

CREATE OR REPLACE FUNCTION all_appointments()
RETURNS SETOF appointment_result AS $$
  SELECT start_hour, patient_id
  FROM appointments
  WHERE patient_id IS NOT NULL;
$$ LANGUAGE SQL;
```

Вместо объявления специального типа *appointment_results* может быть структура данных другой таблицы.

TABLE — используется для возврата таблицы с фиксированным набором столбцов. Каждый столбец указывается явно при объявлении функции.

Преимущества:

- Удобно определять и документировать структуру возвращаемых данных.
- Легко интегрируется с обычными SQL-запросами, так как возвращается четкая таблица.

Недостатки:

- Менее гибкий, если структура данных заранее неизвестна или может изменяться.

Используйте, когда вы знаете точное количество и названия столбцов, которые будут возвращены.

SETOF — позволяет возвращать коллекцию строк (обычно набор записей) произвольной длины. Структура возвращаемого набора похожа на обычную таблицу, но более гибкая.

Преимущества:

- Более гибкое использование: не нужно заранее объявлять количество и имена столбцов.
- Хорошо сочетается с функциями, которые обрабатывают данные потоковым методом.

Используйте, когда структура данных заранее неизвестна или может варьироваться.

13.5. Проверка условий, ветвления

Проверка каких-либо условий и выполнение разных наборов инструкций в зависимости от результатов проверки (ветвление алгоритма), наверное, основная причина, ради которой люди придумали хранимые процедуры.

Синтаксис реализации выглядит так:

Для **MS SQL Server**:

```
-- Если …
IF <условие>
BEGIN
-- код, если условие выполняется
END
```

```
ELSE - иначе
BEGIN
-- код, если условие не выполняется
END;
```

Конструкции BEGIN…END используются, если внутри блока IF или ELSE находится более одной инструкции. Если инструкция одна, BEGIN и END можно не указывать.

Конструкция ELSE (иначе) тоже является не обязательной.

Для **MySQL** и **PostgreSQL** синтаксис идентичен:

```
IF <условие> THEN
    -- Набор инструкций, если условие выполнено
ELSE
    -- Набор инструкций, если условие не выполнено
END IF;
```

Блок ELSE также является необязательным.

Доработаем нашу процедуру назначения у врача. Если все интервалы для приема заняты, вызов процедуры **book_appointment** вызовет ошибку, потому что результат этого запроса вернет NULL:

```
SELECT MIN(all_hours.hour)
  FROM all_hours
  LEFT JOIN appointments ON all_hours.hour = appointments.start_hour
  WHERE appointments.start_hour IS NULL;
```

Соответственно, поскольку столбец *start_hour* объявлен с ограничением NOT NULL, последующая операция вставки вызовет ошибку.

Соответственно, нам нет смысла вызвать INSERT, если свободного времени уже нет.

Добавим в код процедуры проверку на случай, если в результате первого запроса значение переменной *p_start_hour* останется NULL (свободного времени не найдено).

MS SQL Server

```
-- Версия 2 процедуры (проверка)
ALTER PROCEDURE [dbo].[book_appointment] (
  @p_patient_id INT,
  @p_start_hour INT OUTPUT
)
AS
BEGIN
  -- Генерируем список часов с 9 до 18
  WITH all_hours AS (
    SELECT 9 AS hour
    UNION ALL SELECT 10 UNION ALL SELECT 11 UNION ALL SELECT 12
    UNION ALL SELECT 13 UNION ALL SELECT 14 UNION ALL SELECT 15
    UNION ALL SELECT 16 UNION ALL SELECT 17 UNION ALL SELECT 18
  )
  SELECT @p_start_hour = MIN(all_hours.hour)
  FROM all_hours
  LEFT JOIN appointments ON all_hours.hour = appointments.start_hour
  WHERE appointments.start_hour IS NULL;
  -- Проверяем, доступны ли свободные слоты для записи
  IF @p_start_hour IS NOT NULL
  BEGIN
    DECLARE @new_id INT;
    SET @new_id = (SELECT COALESCE(MAX(id), 0) + 1 FROM appointments);
    INSERT INTO appointments (id, start_hour, patient_id)
    VALUES (@new_id, @p_start_hour, @p_patient_id);
  END
END;
```

Проверка простая: если после выполнения результата первого запроса переменная @p_start_hour будет NULL (свободного интервала не найдено), код с добавлением записи в таблицу **appointments** не выполняется.

ОБРАТИТЕ ВНИМАНИЕ:

вместо удаления (DROP) и создания новой процедуры в MS SQL Server можно воспользоваться инструкцией **ALTER PROCEDURE** (изменить процедуру).

MySQL

```
-- Версия 2-й процедуры (проверка)
DROP PROCEDURE IF EXISTS book_appointment;
DELIMITER //
CREATE PROCEDURE book_appointment (
  IN p_patient_id INT,
  OUT p_start_hour INT
)
BEGIN
  -- Объявляем нужную переменную
  DECLARE new_id INT;
  -- Генерируем список часов с 9 до 18
  WITH all_hours AS (
    SELECT 9 AS hour
    UNION ALL SELECT 10 UNION ALL SELECT 11 UNION ALL SELECT 12
    UNION ALL SELECT 13 UNION ALL SELECT 14 UNION ALL SELECT 15
    UNION ALL SELECT 16 UNION ALL SELECT 17 UNION ALL SELECT 18
  )
  SELECT MIN(all_hours.hour) INTO p_start_hour
  FROM all_hours
  LEFT JOIN appointments ON all_hours.hour = appointments.start_hour
  WHERE appointments.start_hour IS NULL;
  -- Проверка, найден ли свободный временной слот
  IF p_start_hour IS NOT NULL THEN
    SET new_id = (SELECT COALESCE(MAX(id), 0) + 1 FROM appointments);
    INSERT INTO appointments (id, start_hour, patient_id)
      VALUES (new_id, p_start_hour, p_patient_id);
  END IF;
END//
DELIMITER ;
```

В MySQL отсутствует возможность изменить процедуру при помощи команды ALTER, поэтому мы ее удаляем (DROP) и создаем новую.

Проверка реализована следующим образом:

```
IF p_start_hour IS NOT NULL THEN...
-- код, который выполняется, если p_start_hour не NULL
END IF;
```

PostgreSQL

В **PostgreSQL** синтаксис проверки идентичен:

```
-- Версия 2-й процедуры (проверка)
CREATE OR REPLACE PROCEDURE public.book_appointment(
   IN p_patient_id integer,
   OUT p_start_hour integer)
LANGUAGE 'plpgsql'
AS $$
DECLARE
  new_id INT;
BEGIN
  -- Объявляем нужную переменную
  -- Генерируем список часов с 9 до 18
  WITH all_hours AS (
    SELECT 9 AS hour
    UNION ALL SELECT 10 UNION ALL SELECT 11 UNION ALL SELECT 12
    UNION ALL SELECT 13 UNION ALL SELECT 14 UNION ALL SELECT 15
    UNION ALL SELECT 16 UNION ALL SELECT 17 UNION ALL SELECT 18
  )
  SELECT MIN(all_hours.hour) INTO p_start_hour
  FROM all_hours
  LEFT JOIN appointments ON all_hours.hour = appointments.start_hour
  WHERE appointments.start_hour IS NULL;
  -- Проверяем, есть ли свободные временные слоты
  IF p_start_hour IS NOT NULL THEN
    new_id := (SELECT COALESCE(MAX(id), 0) + 1 FROM appointments);
    INSERT INTO appointments (id, start_hour, patient_id)
      VALUES (new_id, p_start_hour, p_patient_id);
  END IF;
END;
$$;
```

13.6. Циклы

13.6.1. Организация циклов в разных СУБД

До сих пор мы использовали UNION для генерации списка всех возможных часов для назначения врача, а в главе, посвященной индексам, использовали весьма изощренные способы генерации больших объемов данных. Хотя SQL в первую очередь — декларативный язык (говорит, что надо сделать, а не как), в некоторых случаях полезнее императивный подход (как надо делать). Генерация последовательностей — как раз такой случай.

Иногда требуется повторить набор одних и тех же инструкций несколько раз. Иногда количество известно, а иногда требуется повторение до тех пор, пока не будет выполнено какое-либо условие.

Циклом в программировании вообще и в SQL в частности называется блок кода, управляющий повтором одной или нескольких инструкций с контролем условия прекращения. Сам блок повторяющегося кода называется телом цикла.

Простой цикл для **MS SQL Server**:

```
DECLARE @i INT = 1;

WHILE @i <= 10
BEGIN
  PRINT 'Iteration number: ' + CAST(@i AS VARCHAR(10));
  SET @i += 1;
END;
```

В данном случае используется конструкция WHILE (пока). Пока значение переменной *i* меньше, либо равно 10, выполняется тело цикла (код между BEGIN и END). Внутри тела цикла в каждой итерации к текущему значению @i прибавляется единица. Запись @i += 1 является сокращенным аналогом записи @i = @i + 1, который допускается в MS SQL.

Данная строка очень важна, если бы ее не было, СУБД ушла бы в бесконечный цикл, так как значение @i никогда бы не превысило 10.

В **MySQL** примерным соответствием будет такой вариант:

```
DELIMITER //
CREATE PROCEDURE print_iterations()
BEGIN
  DECLARE i INT DEFAULT 1;
  WHILE i <= 10 DO
    SELECT CONCAT('Iteration number: ', i);
    SET i = i + 1;
  END WHILE;
END //
DELIMITER ;

-- Вызов процедуры
CALL print_iterations();
```

За исключением синтаксических особенностей, он идентичен предыдущему примеру.

В MySQL существует несколько синтаксических альтернатив для организации циклов. Одна из них позволяет реализовать цикл "До тех пор, пока...":

```
DROP PROCEDURE IF EXISTS print_iterations;
DELIMITER //
CREATE PROCEDURE print_iterations()
BEGIN
  DECLARE i INT DEFAULT 1;
  REPEAT
    SELECT CONCAT('Iteration number: ', i) AS message;
    SET i = i + 1;
  UNTIL i > 10 END REPEAT;
END //
DELIMITER ;
```

В отличие от цикла WHILE, который может вообще ни разу не выполниться, если проверка условия вернет FALSE, при использовании конструкции REPEAT... UNTIL тело цикла выполнится хотя бы один раз, потому что проверка цикла происходит в конце.

Еще одним вариантом организации цикла в MySQL является конструкция LOOP… END LOOP:

```
DROP PROCEDURE IF EXISTS print_iterations;
DELIMITER //
CREATE PROCEDURE print_iterations()
BEGIN
  DECLARE i INT DEFAULT 1;
myloop: LOOP
  IF i > 10 THEN
    LEAVE myloop;
  END IF;
    SELECT CONCAT('Iteration number: ', i);
    SET i = i + 1;
  END LOOP;
END //
DELIMITER ;
```

Тело цикла между LOOP и END LOOP будет повторяться бесконечно, поэтому обязательно нужно встраивать проверку условия на выход из цикла. Собственно, сам выход осуществляется при помощи команды LEAVE. При использовании данного синтаксиса обязательным является использование метки (myloop). Метка — просто именованная (имя вы присваиваете сами, какое хотите) область кода, на которую можно ссылаться по этому имени. Именно это происходит здесь при использовании команды LEAVE, вслед за которой вы должны указать имя метки, которая отмечает начало цикла.

В **PostgreSQL** для организации циклов предусмотрен синтаксис LOOP… END LOOP, однако использование несколько отличается от MySQL. Существует несколько вариантов использования этой конструкции, одна из них — с ключевым словом WHILE:

```
DO $$
DECLARE i INT = 0;
BEGIN
WHILE i <= 10 LOOP
  RAISE INFO 'Iteration number: %', i;
  i := i + 1;
```

```
END LOOP;
END
$$;
```

Альтернативно в PostgreSQL можно использовать конструкцию FOR:

```
DO $$
DECLARE
 i INT;
BEGIN
 FOR i IN 0..10 LOOP
  RAISE INFO 'Iteration number: %', i;
 END LOOP;
END
$$;
```

Такая конструкция безопаснее, так как не требует явного приращения счетчика, конструкция FOR делает это "под капотом". Диапазон чисел, определяющих границы диапазона итерации, осуществляется при помощи задания диапазона 0…10.

13.6.2. Досрочный выход из цикла, пропуск итерации

Иногда требуется завершить цикл раньше срока, например, в случае наступления какой-либо исключительной ситуации (ошибки) или просто если того требует бизнес-логика. Кроме того, иногда нужно пропустить какую-либо итерацию, либо для повышения производительности, либо по иным причинам.

Для реализации досрочного выхода из цикла в разных СУБД используются следующие команды:

Операция	MS SQL	MySQL	PostgreSQL
Досрочный выход из цикла	BREAK	LEAVE <метка>	EXIT [<метка>]
Пропуск итерации	CONTINUE	ITERATE <метка>	CONTINUE [<метка>]

SKLADCHIK.ORG

ПРИМЕЧАНИЕ.

Использование меток в PostgreSQL не обязательно, однако они могут оказаться очень полезны, если у вас есть два или более вложенных цикла — тогда при помощи меток вы можете контролировать, к какому именно циклу относится ваша команда.

Для иллюстрации данной концепции организуем внешне бесконечный цикл, но внутри ограничим его 10 итерациями. При этом будем выводить информацию, только если значение счетчика четное.

MS SQL Server

```
DECLARE @i INT = 0;
WHILE 1=1
BEGIN
  -- Увеличиваем счетчик
  -- Это надо делать до всех проверок
  SET @i += 1;
  -- Выход после 10 итераций
  IF @i > 10 BREAK;
  -- Если счетчик нечетный - пропускаем
  IF @i % 2 = 1 CONTINUE;
  PRINT 'Iteration number: ' + CAST(@i AS VARCHAR(10));
END;
```

Поскольку в MS SQL Server нет булева типа данных, обычно для эквивалента TRUE применяется условие 1 = 1, которое всегда выполняется.

MySQL

```
DROP PROCEDURE IF EXISTS print_iterations;
DELIMITER //
CREATE PROCEDURE print_iterations()
BEGIN
```

```
  DECLARE i INT DEFAULT 1;
myloop: LOOP
    -- Увеличивайте счетчик до проверки на четность
    SET i = i + 1;
    IF i > 10 THEN LEAVE myloop; END IF;
    IF i % 2 = 1 THEN ITERATE myloop; END IF;
    SELECT CONCAT('Iteration number: ', i);
  END LOOP;
END //
DELIMITER ;
```

PostgreSQL

```
DO $$
DECLARE
 -- Не забудьте инициализировать переменную,
 -- потому что NULL + 1 = NULL
 i INT = 0;
BEGIN
 WHILE 1 LOOP
  i := i + 1;
  IF i > 10 THEN EXIT; END IF;
  IF i % 2 = 1 THEN CONTINUE; END IF;
  RAISE INFO 'Iteration number: %', i;
 END LOOP;
END
$$;
```

13.6.3. Модификация процедуры записи на прием

Теперь, когда мы умеем организовывать циклы, давайте перепишем процедуру записи пациента на прием без использования CTE с UNION при помощи цикла.

ПРИМЕЧАНИЕ.

На малых диапазонах CTE с UNION окажется эффективнее, поэтому не торопитесь применять циклы. Помните, что SQL — язык декларативный и циклы — чуждая концепция.

MS SQL Server

```
-- Версия 3-й процедуры
ALTER PROCEDURE [dbo].[book_appointment] (
  @p_patient_id INT,
  @p_start_hour INT OUTPUT
)
AS
BEGIN
 SET @p_start_hour = NULL; -- Инициализация выходной переменной
 DECLARE @current_hour INT = 9; -- Начало диапазона

 -- Перебор часов с 9 до 18
 WHILE @current_hour <= 18
 BEGIN
  -- Проверка доступности часа
  IF NOT EXISTS (
   SELECT 1
   FROM appointments
   WHERE start_hour = @current_hour
  )
  BEGIN
   SET @p_start_hour = @current_hour; -- Найден свободный час
   -- Генерация нового ID
   DECLARE @new_id INT;
   SELECT @new_id = COALESCE(MAX(id), 0) + 1 FROM appointments;
   -- Вставка записи
   INSERT INTO appointments (id, start_hour, patient_id)
   VALUES (@new_id, @current_hour, @p_patient_id);
   BREAK; -- Прерываем цикл после вставки
```

```
  END

  SET @current_hour = @current_hour + 1; -- Следующий час
 END
END;
```

MySQL

```
-- Версия 3-й обработки (цикл)
DROP PROCEDURE IF EXISTS book_appointment;
DELIMITER //
CREATE PROCEDURE book_appointment (
  IN p_patient_id INT,
  OUT p_start_hour INT
)
BEGIN
 DECLARE new_id INT;
 DECLARE hour INT;
 -- Настройка начальной позиции часа
 SET hour = 9;
 -- Цикл по часам от 9 до 18
 find_slot: LOOP
  -- Проверяем, занят ли этот час
  IF NOT EXISTS (SELECT 1 FROM appointments WHERE start_hour =
hour) THEN
   -- Нашли свободный слот
   SET p_start_hour = hour;
   -- Создаем новый уникальный ID
   SET new_id = (SELECT COALESCE(MAX(id), 0) + 1 FROM appointments);
   -- Вставляем запись в таблицу
   INSERT INTO appointments (id, start_hour, patient_id)
     VALUES (new_id, p_start_hour, p_patient_id);
  -- Прерываем цикл
   LEAVE find_slot;
  END IF;
  -- Переходим к следующему часу
```

```
  SET hour = hour + 1;
  -- Если просмотрели все часы, заканчиваем цикл
  IF hour > 18 THEN
   LEAVE find_slot;
  END IF;
 END LOOP find_slot;
END//
DELIMITER ;
```

PostgreSQL

```
CREATE OR REPLACE PROCEDURE book_appointment(
 IN p_patient_id integer,
 OUT p_start_hour integer)
LANGUAGE plpgsql
AS $$
DECLARE
 new_id INT;
 current_hour INT := 9; -- Начальное значение часа
BEGIN
 p_start_hour := NULL; -- Инициализация выходного параметра
 -- Перебор часов с 9 до 18
 WHILE current_hour <= 18 LOOP
  -- Проверяем, свободен ли текущий час
  IF NOT EXISTS (
   SELECT 1
   FROM appointments
   WHERE start_hour = current_hour
  ) THEN
   p_start_hour := current_hour; -- Найден свободный слот
   -- Генерируем новый ID
   SELECT COALESCE(MAX(id), 0) + 1 INTO new_id FROM appointments;
   -- Вставляем запись
   INSERT INTO appointments (id, start_hour, patient_id)
   VALUES (new_id, current_hour, p_patient_id);
   EXIT; -- Выходим из цикла после успешной записи
  END IF;
```

```
  current_hour := current_hour + 1; -- Переходим к следующему часу
 END LOOP;
END;
$$;
```

13.6.4. Курсоры. Построчная обработка запроса

Курсоры — это еще один способ изменить декларативную суть SQL. Это инструмент для последовательной обработки строк в результатах запроса. Представьте, вы ведете указкой по каждой записи в списке — каждую запись вы можете прочитать и обработать индивидуально, возможно, по-разному.

Конечно, в подавляющем большинстве случаев можно обойтись без курсоров, используя обычные SQL-запросы, однако есть ситуации, когда использование курсоров действительно обоснованно и удобно:

- **Работа с индивидуальными действиями для каждой строки**.

  Иногда требуется провести операцию, которая может выполняться только построчно. Например, массовое обновление счетов клиентов на основе сложной бизнес-логики, когда каждое действие требует индивидуальных расчетов. (Например, начислить бонус каждому покупателю пропорционально сумме покупок, но сумма бонусов рассчитывается индивидуально для каждого покупателя, основываясь на данных предыдущих заказов и персональных скидок.)

- **Обработка сложноорганизованных данных**.

  Иногда структура данных настолько сложна, что удобнее манипулировать отдельными строками, проходя по набору данных в определенном порядке. (Например: необходимо построить дерево категорий товаров, где родительские категории имеют дочерние элементы. Чтобы построить иерархию, можно пройти по категориям, выстраивая дерево "снизу вверх".)

- **Работа с внешней бизнес-логикой**.

  Если результат расчетов зависит от внешней бизнес-логики (другой базы данных, приложения, внешнего API) и результат каждой операции влияет на результат следующей (например, отправка ордера на отгрузку товара во внешнюю складскую систему и получение текущего остатка в реальном времени).

13.6.5. Пример поднятия цен в книжном магазине

В качестве примера рассмотрим книжный магазин. Нам необходимо для всех книг с ценой ниже 150 рублей поднять цену на 10%. Если даже после этой операции цена все еще будет ниже 150 рублей, необходимо установить пометку *special_offer* в таблицу.

```
CREATE TABLE books (
  id INT PRIMARY KEY,
  title VARCHAR(100),
  price DECIMAL(10,2),
  special_offer BOOL
);

INSERT INTO books VALUES
(1, 'Основы SQL', 100.00, FALSE),
(2, 'SQL для профессионалов', 250.00, FALSE),
(3, 'Проектирование баз данных', 140.0, FALSE),
(4, 'Оптимизация запросов', 99.99, FALSE);
```

НАПОМИНАНИЕ:

для **MS SQL** используйте тип BIT вместо BOOL и укажите значение 0 вместо FALSE (код запросов доступен в электронном приложении).

Цикл жизни курсора начинается с его объявления — он объявляется как переменная с типом CURSOR, для которой указывается источник данных (таблица, запрос, представление), по которому необходима построчная обработка.

После объявления курсор необходимо открыть (OPEN) — после выполнения этой команды физически производится запрос данных, для которых создавался курсор.

Затем у пользователя появляется возможность считывать данные из набора при помощи команды FETCH. Причем, в зависимости от режима работы, указанного в объявлении курсора, перемещаться можно не только вперед, но и назад, на произвольное количество строк (в **MySQL**, к сожалению, такой режим отсутствует — в этой СУБД реализован только один режим — последовательного доступа к данным строка за строкой). Доступ к следующей записи осуществляется так:

```
FETCH NEXT … FROM <имя_курсора> INTO …
```

Команда FETCH (в MS SQL и PostgreSQL), помимо операции NEXT (следующая запись), позволяет очень гибко перемещаться по набору данных:

Операция	Описание
ABSOLUTE	Переход по абсолютному номеру записи FETCH ABSOLUTE 2 … — переход ко второй строке
RELATIVE	Переход по относительному смещению: FETCH RELATIVE -2 … — переход на две строки назад FETCH RELATIVE 3 … — переход на три строки вперед
FIRST	Переход к первой строке
LAST	Переход к последней строке
NEXT	Переход к следующей строке
PRIOR	Переход к предыдущей строке
CURRENT	Переход к текущей строке

ПРИМЕЧАНИЕ.

Для двунаправленного перемещения, курсор должен быть создан в режиме прокрутки (с ключевым словом SCROLL).

По завершении работы курсор необходимо закрыть (CLOSE), так как, пока курсор открыт, транзакция, открывшая курсор, удерживает активные блокировки на ресурсах, участвующих в курсоре. Это может приводить к взаимоблокировке или длительной задержке выполнения других запросов, если какая-то другая транзакция ожидает освобождения заблокированных ресурсов. Кроме того, открытый курсор занимает оперативную память и другие ресурсы базы данных. Если вы забудете закрыть курсор, он продолжит занимать ресурсы, пока не завершится транзакция или не истечет таймаут ожидания. Большое количество открытых курсоров может существенно снизить общую производительность системы.

В **MS SQL** курсор требует явного освобождения выделенных ресурсов при помощи команды **DEALLOCATE**. Если этого не делать, то со временем накопившиеся неактивные курсоры начинают занимать значительную долю доступной памяти и ухудшают производительность системы. Это явление называют "утечкой курсоров". Если ваш код интенсивно использует курсоры и вы забываете очищать их после закрытия, это повлияет на способность системы обслуживать параллельные запросы и транзакции.

MS SQL Server

С учетом сказанного, наша процедура, позволяющая обработать таблицу с книгами в **MS SQL Server**, выглядит следующим образом:

```
CREATE PROCEDURE update_books AS
BEGIN
 DECLARE @book_id INT, @book_price DECIMAL(10,2);
 -- Создаем курсор
 DECLARE cur CURSOR LOCAL FOR
  SELECT id, price FROM books WHERE price < 150;
 OPEN cur;
 FETCH NEXT FROM cur INTO @book_id, @book_price;
 WHILE @@FETCH_STATUS = 0
 BEGIN
  SET @book_price = @book_price * 1.10;
  UPDATE books SET
    price = @book_price,
    special_offer = IIF(@book_price < 150, 1, 0)
```

```
    WHERE id = @book_id;
    FETCH NEXT FROM cur INTO @book_id, @book_price;
  END
  CLOSE cur;
  DEALLOCATE cur;
END
```

В SQL Server встроенная функция @@FETCH_STATUS используется для проверки состояния последнего вызова команды FETCH, связанной с курсором. Он возвращает специальное значение, показывающее успешность операции выборки строки из курсора. Функция возвращает 0, если очередная строка данных была успешно прочитана, -1, если больше нет данных (все сроки курсора исчерпаны), и -2, если произошла какая-либо ошибка (курсор закрыт или уничтожен).

Дополнительную гибкость в MYSQL также обеспечивают различные режимы работы для курсоров:

- **LOCAL** (по умолчанию) — видимость ограничивается областью видимости процедуры, в которой он был объявлен.
- **GLOBAL** — курсор доступен для текущего сеанса. Глобальные курсоры сохраняют свое состояние даже после завершения процедуры. Очень важно явно закрывать (**CLOSE**) и освобождать (**DEALLOCATE**) курсоры, объявленные таким образом, чтобы избежать риска блокировок и утечек памяти. Используйте, только если вы точно уверены, что по-другому вашу задачу никак не реализовать.
- **STATIC** — создает статический снимок данных, основанный на запросе. Данные фиксируются в момент открытия курсора и не отражают последующие изменения в таблице. Это хороший выбор, если нужно защитить данные от изменений во время работы с курсором.
- **DYNAMIC** — динамический курсор отражает все изменения, происходящие в базовой таблице. Если строка изменилась, удалена или добавлена, динамические курсоры немедленно показывают эти изменения. Очень ресурсоемки и медленны, но обеспечивают максимальную согласованность данных.

SKLADCHIK.ORG

- **KEYSET** — компромисс между статическими и динамическими курсорами. Идентификаторы (ключи) строк остаются зафиксированными, но сами данные могут изменяться. Например, если цена товара изменилась, динамическое изменение будет видно, но если товар удален или добавлен, это не отразится в результатах курсора.
- **FAST_FORWARD** — в этом режиме можно быстро перемещаться вперед по набору данных, но без возможности двигаться назад (forward-only).
- **SCROLL_LOCKS** — позволяет устанавливать блокировки на все строки, извлеченные курсором, предотвращая их изменение другими пользователями.
- **OPTIMISTIC** — работает по принципу оптимистической блокировки, предполагая, что конфликты редки. Проверяет изменения при попытке модификации строки.

MySQL

В **MySQL** аналогичная обработка таблицы **books** реализуется следующим образом:

```
-- MySQL
DELIMITER //
CREATE PROCEDURE update_books()
BEGIN
 DECLARE done BOOLEAN DEFAULT FALSE;
 DECLARE book_id INT;
 DECLARE book_price DECIMAL(10,2);
 -- Объявление курсора
 DECLARE cur CURSOR FOR
  SELECT id, price FROM books WHERE price < 150;
 -- Обработчик конца данных
 DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
 OPEN cur;
 read_loop: LOOP
  FETCH cur INTO book_id, book_price;
  IF done THEN LEAVE read_loop; END IF;
  -- Повышение цены
  SET book_price = book_price * 1.10;
  -- Обновление записи
```

```
  UPDATE books
  SET
     price = book_price,
     special_offer = book_price < 150
  WHERE id = book_id;
 END LOOP;
 CLOSE cur;
END //
DELIMITER ;
```

В этом варианте заслуживает внимания механизм контроля условий для выхода из режима чтения курсора. Если курсор не сможет вернуть ни одной записи, MySQL воспримет это как исключительную ситуацию (ошибку). Мы используем обработчик таких ситуаций, чтобы грамотно завершить процедуру:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
```

- Объявить (DECLARE) … обработчик (HANDLER) — стандартный механизм обработки исключительных ситуаций в MySQL.
- CONTINUE (Продолжить) — после обработки исключения выполнение процедуры продолжается с места, где возникла ошибка. Альтернативой будет команда EXIT (немедленный выход из процедуры).
- FOR NOT FOUND — условие срабатывания. В данном случае обработчик получит управление, если курсор не сможет вернуть следующую строку (окончание чтения). Также сработает, если команда SELECT не вернет ни одной строки, либо если UPDATE/DELETE не затронут ни одну запись.
- SET done = TRUE — код, который выполняется в случае срабатывания исключения. В данном случае служебной переменной done присвоится значение TRUE, данное значение впоследствии будет использовано в условии для выхода из цикла чтения.

PostgreSQL

PostgreSQL предлагает два уровня использования курсоров: **явный** и **неявный**. В явном виде работа с курсором очень похожа на другие СУБД:

```
CREATE OR REPLACE PROCEDURE update_books()
LANGUAGE plpgsql AS $$
DECLARE
 book_cursor CURSOR FOR
  SELECT id, price, special_offer FROM books WHERE price < 150;
 book_record RECORD;
BEGIN
 -- Открыть курсор
 OPEN book_cursor;
 -- Цикл по данным курсора
 LOOP
  FETCH book_cursor INTO book_record;
  EXIT WHEN NOT FOUND; -- Выход из цикла, если данные закончились
  -- Логика обработки
  book_record.price := book_record.price * 1.10;
  -- Обновляем книгу
  UPDATE books
  SET price = book_record.price, special_offer = book_record.
price < 150
  WHERE id = book_record.id;
  END LOOP;
  -- Закрыть курсор
  CLOSE book_cursor;
END;
$$;
```

В данном листинге нужно обратить внимание на конструкцию:

```
EXIT WHEN NOT FOUND;
```

И ее расположение — сразу после команды FETCH. Команда EXIT уже знакома, но к ней добавляется спецификация WHEN (когда) и ситуация-исключение (NOT FOUND).

Также здесь использован уникальный для PostgreSQL тип данных **record**. Он позволяет хранить всю строку данных (запись) целиком. Обращение к каждому из полей происходит через оператор доступа "точка"(.) (например, book_record.price).

PostgreSQL также может использовать курсор и в неявном виде. Данный подход имеет меньшую гибкость, так как не предоставляет прямого доступа к курсору, но в то же время делает код более читаемым:

```
CREATE OR REPLACE PROCEDURE update_books()
LANGUAGE plpgsql AS $$
DECLARE
 book_record RECORD;
BEGIN
 FOR book_record IN
   SELECT id, price FROM books WHERE price < 150
 LOOP
 book_record.price := book_record.price * 1.10;
 UPDATE books
   SET price = book_record.price, special_offer = book_record.
price < 150
 WHERE id = book_record.id;
 END LOOP;
END;
$$;
```

Эта процедура идентична предыдущей. Создание курсора осуществляется вместе с созданием цикла:

```
FOR book_record IN
   SELECT id, price FROM books WHERE price < 150
LOOP
...
END LOOP
```

13.7. Обработка ошибок

В идеальном мире каждый SQL-запрос выполняется успешно. Однако реальность такова, что операции с базой данных подвержены множеству сбоев. Так, довольно распространенная ситуация — нарушение ограничений (CHECK, UNIQUE и пр.).

Также ошибки могут возникнуть при попытке преобразования данных ('abc' → INT), при попытке деления на 0, при исчерпании доступной памяти, при достижении лимита соединений, при отсутствии прав доступа к данным и во многих других ситуациях.

Ошибки в данном контексте принято именовать исключениями (EXCEPTION) (исключительными ситуациями). Если не реагировать на ошибки, поведение системы становится плохо предсказуемым, именно для этого каждая из СУБД предлагает собственный механизм обработки исключений.

13.7.1. MS SQL Server

SQL Server использует интуитивно понятную конструкцию TRY ... CATCH, похожую на аналоги в императивных языках программирования. При ошибке в блоке TRY управление немедленно передается в CATCH. Весь последующий код в TRY пропускается:

```
BEGIN TRY
  -- Код, который может вызвать ошибку
END TRY
BEGIN CATCH
  -- Код, который неободимо выполнить в случае ошибки
  -- Получение информации об ошибке с использованием
  -- встроенных функций
  SELECT
    ERROR_NUMBER() AS ErrorNumber,
    ERROR_SEVERITY() AS ErrorSeverity,
    ERROR_STATE() AS ErrorState,
    ERROR_PROCEDURE() AS ErrorProcedure,
    ERROR_LINE() AS ErrorLine,
    ERROR_MESSAGE() AS ErrorMessage;
-- Можно повторно сгенерировать ошибку,
-- чтобы ее смог обработать вызывающий код
-- TROW;
END CATCH;
```

Конструкция TRY ... CATCH не управляет транзакциями автоматически. **Критически важно** явно проверять значение встроенной функции

@@TRANCOUNT и выполнять ROLLBACK в CATCH при необходимости, чтобы избежать "висящих" транзакций и блокировок.

Например:

```
BEGIN TRY
  PRINT 5/0;
  PRINT 'Не судьба'; -- Не будет выполнено
END TRY
BEGIN CATCH
  PRINT 'Деление на 0';
END CATCH
```

Если в ходе выполнения могут произойти ошибки разных типов, используйте встроенную функцию ERROR_NUMBER(), чтобы получить уникальный код ошибки конкретно для текущей ситуации.

13.7.2. MySQL

Для **MySQL** мы уже использовали обработчик исключений, когда рассматривали курсоры. Обработчик объявляется в конструкции DECLARE … HANDLER FOR:

```
DELIMITER //

CREATE PROCEDURE myProc()
BEGIN
  -- Для получения информации о возможной ошибке
  DECLARE exit_code INT DEFAULT 0;
  DECLARE error_msg VARCHAR(255) DEFAULT '';
  -- Объявление обработчика
  DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING, NOT FOUND
  BEGIN
    -- Код, который будет выполнен при срабатывании обработчика
    -- Получение кода и текста ошибки:
    GET DIAGNOSTICS CONDITION 1
      exit_code = MYSQL_ERRNO,
      error_msg = MESSAGE_TEXT;
```

```
    -- Действия при ошибке (Откат, логирование, установка флагов)
    ROLLBACK; -- Явный откат, если транзакция активна
  END;
  -- Начало транзакции (рекомендуется для атомарности)
  START TRANSACTION;
  -- Код, который может вызвать ошибку
  COMMIT;
END //

DELIMITER ;
```

Типы исключений:

- **DECLARE ... HANDLER FOR SQLEXCEPTION** — перехватывает большинство ошибок выполнения (кроме NOT FOUND).
- **DECLARE ... HANDLER FOR SQLWARNING** — перехватывает предупреждения (например, при выполнении DROP… IF EXISTS, если объекта не существовало, будет сгенерировано предупреждение).
- **DECLARE ... HANDLER FOR NOT FOUND** — перехватывает ситуацию, когда строка не найдена в результате SELECT либо когда в результате UPDATE/DELETE не было затронуто ни одной строки.

Вместо DECLARE EXIT можно использовать DECLARE CONTINUE (если требуется продолжить выполнение).

13.7.3. PostgreSQL

В PostgreSQL для контроля ошибок используется конструкция EXCEPTION:

```
CREATE OR REPLACE FUNCTION myProc() RETURNS void AS $$
BEGIN
  -- Код, который может вызвать ошибку
EXCEPTION  -- Секция перехвата исключений
  -- Обработка конкретного типа ошибки (нарушение уникальности)
  WHEN unique_violation THEN
```

```
    -- Действия в случае ошибки, например
    RAISE NOTICE 'Дубликат ключа! Действие отменено.';
  -- Обработка другого типа ошибки (нарушение CHECK)
  WHEN check_violation THEN
    RAISE NOTICE 'Нарушение проверочного ограничения!';
  -- Все прочие ошибки
  WHEN OTHERS THEN
    RAISE NOTICE 'Произошла непредвиденная ошибка: %', SQLERRM;
    -- RAISE; -- Повторное возбуждение исходной ошибки
END;
$$ LANGUAGE plpgsql;
```

Существует несколько типов исключений для обработки:

- **NOT FOUND** — обрабатывает ситуацию, когда строк больше нет.
- **UNIQUE_VIOLATION** — возникает, если нарушается уникальное ограничение (например, попытка вставить дубликат ключа).
- **FOREIGN_KEY_VIOLATION** — возникает, если нарушается ограничение внешнего ключа (например, ссылка на несуществующий ключ).
- **DATA_EXCEPTION** — обрабатываются общие ошибки данных, такие как превышение размера поля, недопустимый формат данных и т.д.
- **INTEGRITY_CONSTRAINT_VIOLATION** — включает нарушения любых ограничений целостности (включая уникальные ключи, внешние ключи и другие ограничения).
- **OTHERS** — ловит все остальные исключения, которые не попали в предыдущие группы.

Для более детального анализа ошибки можно использовать следующую аналитику:

- SQLSTATE — код состояния SQL-ошибки (строка из 5 символов).
- SQLERRM — текст сообщения об ошибке.
- GET STACKED DIAGNOSTICS — более мощная функция для получения различных атрибутов ошибки.

13.8. Триггеры

Довольно часто при работе со сложными базами данных возникает ситуация, когда необходимо выполнить определенную процедуру при наступлении какого-либо события. Иногда бывает сложно реализовать комплексные проверки через стандартные ограничения SQL (check, unique и пр.), и очень часто возникает необходимость вести аудит изменений в критически важных данных.

Для подобных целей в СУБД предусмотрены т. н. **триггеры** (triggers) — это специальные хранимые процедуры, которые автоматически выполняются при наступлении заданных событий в базе данных (INSERT, UPDATE, DELETE). Их ключевая задача — обеспечить автоматизацию сложной бизнес-логики на уровне СУБД, гарантируя целостность данных и снижая риски ошибок.

В качестве практического применения триггеров можно рассмотреть следующие примеры:

1. **Аудит изменений** — фиксация истории правок (кто, когда и что изменил).
2. **Сложные ограничения целостности** — например, запрет на снижение цены товара более чем на 10%.
3. **Автоматический пересчет данных** — обновление итоговых значений (например, суммы заказа при добавлении товара).
4. **Синхронизация таблиц** — копирование данных в архивные таблицы при удалении.
5. **Валидация связанных данных** — проверка согласованности информации между таблицами.

Триггеры разделяют по типам событий, которые их вызывают:

- INSERT — срабатывает при добавлении строк в таблицу;
- UPDATE — при обновлении строк в таблице;
- DELETE — при удалении строк из таблицы.

Или по времени выполнения:

- BEFORE — триггер срабатывает до проведения операции, которая его вызвала;
- AFTER — срабатывает после проведения операции;
- INSTEAD OF — (заменяет оригинальную логику).

По уровню срабатывания:

- FOR EACH ROW — операция, определенная в триггере, срабатывает для каждой изменяемой строки;
- FOR EACH STATEMENT — единожды на инструкцию.

13.8.1. Создание триггера

Общий синтаксис для создания триггера:

```
CREATE TRIGGER <имя_триггера>
[BEFORE|AFTER|INSTEAD OF] [INSERT|UPDATE|DELETE]
ON <имя_таблицы>
-- Уровень срабатывания (по строке или оператору)
[FOR EACH ROW|FOR EACH STATEMENT]
AS
BEGIN
  -- Логика триггера (SQL-код)
END;
```

Создадим простой пример с аудитом изменений. Первая таблица — электронный журнал успеваемости учеников, где указываются дата, предмет и оценка:

```
-- Основная таблица
CREATE TABLE student_marks
(
  id INT PRIMARY KEY,
```

SKLADCHIK.ORG


```
  student_id INT,
  discipline VARCHAR(30),
  date DATE,
  mark INT CHECK (mark > 0 AND mark <= 5)
);
```

Во вторую таблицу будет записываться информация о каждой операции изменений в первой таблице (если кто-то захочет изменить уже поставленную при помощи операции INSERT оценку при помощи операции UPDATE, эта информация будет зафиксирована):

```
CREATE TABLE marks_audit
(
 -- для MS SQL используйте DATETIME
 timestamp TIMESTAMP PRIMARY KEY,
 record_id INT,
 username VARCHAR(100),
 old_mark INT,
 new_mark INT,
 FOREIGN KEY (record_id) REFERENCES student_marks(id)
);
```

Теперь создадим триггер на обновление основной таблицы.

MS SQL Server

```
CREATE TRIGGER check_mark_on_delete_or_update
ON student_marks
AFTER UPDATE
AS
BEGIN
 -- Проверяем, изменялась ли оценка (столбец mark)
 IF UPDATE(mark)
 BEGIN
  -- Вставка в таблицу аудита
  INSERT INTO marks_audit
  SELECT
```

```
    GETDATE(),
    inserted.id,
    SUSER_SNAME(), -- Имя текущего пользователя
    deleted.mark, -- Старая оценка
    inserted.mark  -- Новая оценка
   FROM
    inserted
   INNER JOIN
    deleted ON inserted.id = deleted.id
   WHERE
    -- Проверяем, что оценка действительно изменилась
    inserted.mark != deleted.mark;
  END
END;
```

Внутри триггера нам доступна запись *deleted* и запись *inserted*, в которой будут храниться значения, соответственно, для старой и новой версии данных в случае операции UPDATE. Если триггер создается на операцию INSERT, то значения *deleted* будут недоступны, и наоборот — если триггер на операцию DELETE, недоступными станут значения *inserted*.

Для генерации текущего времени используется встроенная функция GETDATE(), а для получения текущего имени пользователя — SUSER_NAME().

MySQL

```
DELIMITER //
CREATE TRIGGER check_mark_on_delete_or_update
AFTER UPDATE ON student_marks
FOR EACH ROW
BEGIN
  IF (OLD.mark <> NEW.mark) THEN
    INSERT INTO marks_audit
      VALUES (NOW(), OLD.id, CURRENT_USER(), OLD.mark, NEW.mark);
  END IF;
END //
DELIMITER ;
```

Внутри триггера присутствуют ссылки на копии изменяемой строки. Функция NEW вернет ссылку на новую версию данных (после обновления), а функция OLD вернет ссылку на версию данных до изменений. В случае операции DELETE ссылка на удаляемые данные будет доступна через функцию OLD, а функция NEW вернет NULL.

> Для генерации текущей даты и времени используется функция NOW(), для получения имени текущего пользователя — функция CURRENT_USER().

PostgreSQL

В PostgreSQL триггеры также пишутся с использованием псевдострок OLD и NEW, но синтаксис и подход несколько отличаются: триггеры в PostgreSQL реализуются при помощи функций, которые возвращают тип **trigger**:

```
CREATE OR REPLACE FUNCTION check_mark_on_update_function()
RETURNS TRIGGER AS $$
BEGIN
  -- Проверяем, изменилась ли оценка
  IF OLD.mark <> NEW.mark THEN
    -- Вставляем запись в таблицу аудита
    INSERT INTO marks_audit
    VALUES (NOW(), OLD.id, CURRENT_USER, OLD.mark, NEW.mark);
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

После создания функции необходимо создать триггер, который будет вызывать эту функцию в нужный момент:

```
-- Создание триггера
CREATE TRIGGER check_mark_on_delete_or_update
```

```
AFTER UPDATE OF mark ON student_marks
FOR EACH ROW
EXECUTE FUNCTION check_mark_on_update_function();
```

Добавим оценки ученику:

```
INSERT INTO student_marks VALUES
(1, 1, 'Информатика', '2025-06-02', 4),
(2, 1, 'Химия', '2025-06-02', 4);
```

Теперь попробуем изменить оценку:

```
UPDATE student_marks SET mark = 5 WHERE id = 1;
```

Если теперь вы запросите данные из таблицы **marks_audit**:

```
SELECT * FROM marks_audit;
```

вы получите результат — время изменения, имя пользователя (старую и новую отметку).

13.8.2. Изменение триггеров

Во всех СУБД прямое изменение триггеров не поддерживается. Чтобы изменить триггер, необходимо его удалить (DROP), затем создать заново.

```
-- Для SQL Server, MySQL, PostgreSQL:
DROP TRIGGER [IF EXISTS] <имя_триггера>;
```

13.8.3. Вопросы производительности

В зависимости от сложности логики, которая будет выполняться внутри триггера, производительность системы может серьезно страдать.

Когда триггеры могут затормаживать работу системы?

- При операциях массового изменения данных. Код триггера будет вызываться каждый раз для каждой изменяемой строки.

- Особую опасность могут представлять рекурсивные триггеры — когда триггер изменяет таблицу, изменение которой вызывает этот же триггер.
- Длительные операции в триггере могут мешать другим транзакциям из-за блокировок.

В некоторых случаях (например, перед массовой вставкой, изменением или удалением данных) рекомендуется временно отключать триггеры.

MS SQL Server

```
-- Отключить один триггер
DISABLE TRIGGER <имя триггера> ON <имя_таблицы>;
-- Отключить все триггеры для таблицы
DISABLE TRIGGER ALL ON <имя_таблицы>;

-- Включить один триггер
ENABLE TRIGGER <имя триггера> ON <имя_таблицы>;
-- Включить все триггеры для таблицы
ENABLE TRIGGER ALL ON <имя_таблицы>;
```

MySQL

В MySQL нет прямого аналога отключения и включения триггеров, поэтому применяют обходные решения:

1. Удаление и повторное создание триггеров.
2. Переименование таблицы при помощи команды RENAME TABLE.

PostgreSQL

```
-- Отключить один триггер
ALTER TABLE <имя триггера>DISABLE TRIGGER <имя_таблицы>;
-- Отключить все триггеры на таблице
```

```
ALTER TABLE <имя триггера> DISABLE TRIGGER ALL;
-- Включить один триггер:
ALTER TABLE <имя триггера> ENABLE TRIGGER <имя_таблицы>;
-- Отключить все триггеры на таблице
ALTER TABLE <имя триггера> ENABLE TRIGGER ALL;
```

Триггеры — мощный инструмент для поддержания целостности данных и автоматизации, но их следует использовать осторожно.

13.8.4. Советы

- Тестируйте влияние триггеров на производительность.
- Документируйте логику триггеров.
- Избегайте рекурсивных сценариев.

13.9. Плюсы и минусы переноса бизнес-логики в СУБД

Разработка архитектуры приложения — это постоянный поиск оптимального распределения ответственности. Перенос части бизнес-логики (валидация, сложные расчеты, workflow, обеспечение целостности) на уровень СУБД с использованием хранимых процедур, функций, триггеров и ограничений во всех случаях требует обдумывания и взвешенного подхода.

13.9.1. Что вы получаете, реализуя бизнес-логику в СУБД

1. **Гарантированную целостность данных** (принципы ACID в полную силу): триггеры и ограничения обеспечивают соблюдение сложных бизнес-правил независимо от источника запроса (основное приложение,

API, хоть прямой SQL-запрос через консоль). Вам не требуется проводить валидацию на множестве точек входа. Все операции атомарны — использование транзакций внутри хранимых процедур исключает недействительные состояния данных.

2. **Высокое быстродействие** для операций обработки больших объемов данных и значительное снижение сетевого трафика: вместо схемы работы — получить данные, обработать, сохранить обратно, все операции выполняются "рядом" с данными. Чем больший объем данных нужно обрабатывать, тем выгоднее реализовывать встроенную в СУБД бизнес-логику. Выполняя фильтрацию и агрегацию на месте, клиентское приложение возвращает только необходимый объем данных, не требующий последующей обработки. Оптимизатор запросов может работать эффективнее с логикой, выраженной на его "родном" языке.

3. **Единую "точку истины"** — критически важная логика реализуется единожды в БД и используется всеми клиентами (веб, мобильные, десктоп, другие сервисы). Исключает риск рассинхронизации в логике реализаций в разных клиентских приложениях. Нет необходимости дублировать одну и ту же логику в разных приложениях, клиент просто вызывает одну процедуру вместо проведения предварительных проверок и сложных запросов.

4. **Безопасность** через контролируемый доступ — клиентам можно давать право только на выполнение процедур (GRANT EXECUTE), скрывая детали структуры таблиц и прямого DML-доступа. Инструменты разграничения доступа — материал следующей главы.

СУБД может являться удобной точкой интеграции и рефакторинга легаси-систем, позволяя постепенно забирать на себя функционал старых приложений.

13.9.2. Что плохого вы получите "в нагрузку"

1. **Сложность отладки и тестирования** — по сравнению со средами для разработки и отладки для традиционных языков программирования (Java, C#, Python), в силу декларативной сути SQL отладка может быть гораздо более трудоемкой и требовать больше времени. То же касается и тестиро-

вания — организация модульных, интеграционных и нагрузочных тестов для хранимого кода сложнее, а сам процесс менее стандартизирован, чем для кода приложения.

2. **Сложность масштабирования** — реализация сложной логики внутри СУБД закрепляет значительную часть вычислительной нагрузки за серверами базы данных. Если обычное приложение можно масштабировать относительно легко, просто добавляя однотипные серверы, то горизонтальное масштабирование вычислений внутри СУБД (особенно связанных с записью данных или требующих глобального обзора) является значительно более сложной, дорогой и ограниченной задачей. Это может стать серьезным архитектурным ограничением для высоконагруженных систем, требующих горизонтального роста.

3. **Жесткую привязку к одному производителю** — хранимый код (PL/pgSQL, T-SQL, PL/SQL) сильно привязан к конкретной СУБД (PostgreSQL, SQL Server). Миграция на другую СУБД становится весьма болезненной и дорогой задачей.

4. **Сложность сопровождения** (maintainability) — логика системы распределяется среди множества таблиц, хранимых процедур и функций. Понимание системы требует знания и кода приложения, и кода в БД. Очень велик риск появления в проекте "незаменимых" людей — носителей знаний, и усложняется процесс привлечения новых разработчиков.

5. **Риск антипаттерна "Big Ball of Mud"** (большой комок грязи) в БД — создания гигантских, запутанных и недокументированных хранимых процедур, которые становятся кошмаром для поддержки и сопровождения.

6. **Сложные математические вычисления**, работа со строками и некоторые алгоритмы могут выполняться медленнее в СУБД по сравнению с оптимизированным кодом на компилируемых языках (C++, Go, Java).

13.9.3. Переносить или не переносить?

Природа логики:

- **ДА**: целостность данных критична для системы (сложные проверки, валидация), многошаговые операции, операции с большими объемами данных (агрегация, фильтрация), логика, единая для многих клиентов.
- **НЕТ** (или осторожно): в системе используется сложная вычислительная логика, логика, сильно зависящая от внешних сервисов, часто меняющаяся логика из-за требований бизнеса.

Требования к производительности и масштабируемости:

- **ДА**: задачи, связанные с обработкой больших массивов данных, в которых сетевые издержки на транспорт этих данных могут стать "бутылочным горлышком" для производительности системы.
- **НЕТ**: системы, требующие экстремального горизонтального масштабирования вычислений, задачи, требующие высокой нагрузки на процессор.

Архитектура и Технологический Стек:

- **ДА**: монолитные приложения, системы с одним основным клиентом, "легаси"-системы.
- **НЕТ** (или минимум): микросервисная архитектура (риск создания распределенного монолита БД), проекты с высокой вероятностью смены СУБД, использование NoSQL БД без развитых хранимых процедур.

Навыки Команды и Процессы:

- **ДА**: при наличии сильных SQL-разработчиков и администраторов баз данных, налаженных процессах тестирования и версионирования БД-кода.
- **НЕТ**: если команда не имеет опыта или ресурсов для поддержки сложной SQL-логики.

Решение о размещении бизнес-логики — это не выбор между двумя крайностями "все в БД" или "ничего в БД". Это поиск оптимального баланса, основанного на сильных и слабых сторонах каждого подхода в контексте

вашего проекта. Хранимые процедуры, функции и триггеры позволяют разработчику создать централизованную единую систему критичных правил для валидации данных и обеспечения их целостности. Логика должна быть реализована там, где ее можно будет легко обслуживать. Понимая границы применения для инструментов, описанных в этой главе, вы сможете проектировать более надежные и эффективные системы.

Основой надежности любой системы, однако, служит тщательно продуманная и спроектированная система безопасности и защиты данных. В следующей главе мы рассмотрим основные инструменты по разграничению доступов к объектам БД.

SKLADCHIK.ORG

Глава 14.

Безопасность данных и управление пользователями

Мы научились обращаться с данными — выбирать, фильтровать, объединять, изменять, автоматизировать их обработку, однако, редко когда бывает, чтобы с базой данных работал только один человек. В большинстве случаев к тем или иным данным в базе имеют доступ много людей (сотни, тысячи, порой — миллионы). Умение правильно разграничивать доступы, кому можно только читать, а кому — только записывать данные, кто может запускать ту или иную процедуру, а кто не может, является прерогативой не только администратора баз данных (DBA), но и обязательным навыком для разработчика.

Утечки данных всегда неприятны, утечки чувствительных или персональных данных могут стать причиной финансовых потерь, а утечки критических данных могут стать причиной катастрофы. Разработчик лучше всех осведомлен о внутренней структуре БД, поэтому его участие в организации безопасности очень часто является критически важным.

Безопасность в базах данных строится на уровнях аутентификации, авторизации и шифрования данных. Механизм аутентификации позволяет проверить подлинность пользователя через проверку пары логин/пароль, пользовательского сертификата и пр.). В этой главе мы не будем углубляться в настройку методов аутентификации и шифрования данных, так как эти аспекты безопасности выходят за рамки чистого SQL и требуют действий DBA или системного администратора.

Авторизация (authorization) предоставляет ответ на вопрос "Что тебе разрешено?". Управление авторизацией в СУБД производится средствами SQL, поэтому этот аспект безопасности мы рассмотрим подробнее.

14.1. Ключевые концепции

14.1.1. Принцип наименьших привилегий

Принцип наименьших привилегий — основа безопасности данных, на которой строится все остальное: каждому пользователю, роли или приложению должны быть предоставлены ровно те права или привилегии, которые абсолютно необходимы для выполнения их текущих задач. Ни больше, ни меньше. В случае если злоумышленник получит доступ под учетной записью с минимальными правами, его возможности по краже или уничтожению данных будут ограничены лишь правами доступа скомпрометированной учетной записи.

Запуск запроса UPDATE или DELETE без WHERE по ошибке — ночной кошмар любого разработчика. Принцип наименьших привилегий позволяет минимизировать ущерб от ошибок своих же пользователей.

Чем точнее определены права, тем проще понять, кто за что отвечает, и проверить соответствие политикам безопасности. Назначение прав пользователям должно начинаться с нуля (все запрещено), затем добавляться только самое необходимое.

14.1.2. Субъекты безопасности

Управление доступом в SQL вращается вокруг взаимодействия нескольких ключевых сущностей:

- **Пользователи (Users).**

  Индивидуальные учетные записи, которые используются для подключения к СУБД. Каждая учетная запись уникальна, как пропуск на охраняемую территорию.

- **Роли (Roles).**

  Это именованные группы привилегий. Роли создаются для объединения набора прав, необходимого для выполнения определенной функции (например, роль оператора, роль аналитика, роль бизнес-пользователя, роль бухгалтера, роль администратора и т. д.)

Пользователям назначаются роли.

Хотя технически ничто не препятствует назначать индивидуальные полномочия каждому пользователю по отдельности, гораздо эффективнее назначать права по ролям, так как количество пользователей может измеряться сотнями и тысячами, и проводить аудит прав всех прав станет очень трудоемкой задачей, особенно если учесть, что права пользователей могут изменяться со временем (например, если сотрудник фирмы получит повышение или перевод в другое подразделение). Даже если сейчас у вас всего несколько пользователей в базе, лучше сразу применить ролевую модель разграничения доступа, возможно, это позволит вам сэкономить много времени впоследствии.

Изменение прав роли автоматически применяется ко всем пользователям, назначенным на эту роль. К примеру, чтобы предоставить всем аналитикам доступ к новой таблице, достаточно будет обновить одну роль.

ПРИМЕЧАНИЕ.

В PostgreSQL концепция "роль" охватывает обе идеи — и **пользователя**, и **группы пользователей**. Роль может представлять как индивидуального пользователя, так и группу пользователей. Такой подход, согласно идеологии разработчиков, позволяет гибко организовывать права доступа и избегать избыточности в управлении группами и пользователями.

14.1.3. Схемы (Schemas)

Схемы в PostgreSQL и MS SQL являются **логическими контейнерами внутри базы данных**, внутри которых объекты (таблицы, процедуры, представления) объединяются по принципу или принадлежности (например, *sales* для отдела продаж, *hr* — для отдела кадров и т. д.).

Очень часто права выдаются не на отдельные таблицы, а на целую схему. Если провести аналогию — сотруднику выдается право на доступ в определенное помещение, а не к каждому шкафу или папке с документами внутри этого помещения.

ПРИМЕЧАНИЕ.

В MySQL схемы не реализованы. Изначально проект MySQL ориентировался на потребности веб-приложений и небольших проектов, где необходимость многоуровневой организации была минимальной. Впоследствии этот подход стал частью философии продукта, направленной на легкость эксплуатации и низкую стоимость владения.

Когда вы создаете новую базу данных, в ней уже предопределена схема по умолчанию (**dbo** в MS SQL Server или **public** в PostgreSQL). Вы можете создавать свои схемы при помощи запроса:

```
CREATE SCHEMA <имя_схемы>;
```

Чтобы создать таблицу (или другой объект) для этой схемы, необходимо явно указать имя схемы при создании:

```
CREATE SCHEMA custom;
CREATE TABLE custom.newtable (id INT);
```

ВАЖНО!

В разных схемах могут существовать объекты с одинаковым именем. Хотя такой подход к именованию объектов является нежелательным, так как может являться причиной путаницы и ошибок, иногда подобное случается, поэтому при работе с несколькими схемами тщательно проверяйте, какая в вашем сеансе выбрана схема по умолчанию, а лучше — явно указывайте схему, чтобы быть уверенным, что вы работаете именно с нужным вам объектом.

Например:

```
CREATE SCHEMA spb; -- Схема для подразделения из Санкт-
Петербурга
-- GO - Для MS SQL
CREATE TABLE spb.projects (id INT);
-- GO - Для MS SQL
CREATE SCHEMA mos; --Схема для подразделения из Москвы
-- GO - Для MS SQL
CREATE TABLE mos.projects (id INT);
```

Теперь, в зависимости от того, какая схема сейчас является схемой по умолчанию в открытом сеансе, запрос:

```
SELECT * FROM projects;
```

Может привести к совершенно разным результатам. Надежнее указывать схему данных явным образом:

```
SELECT * FROM mos.projects;
```

14.2. Предопределенные (системные) роли

Каждая СУБД поставляется с набором встроенных ролей, обладающих широкими полномочиями. Обращение с ними требует особой осторожности!

В MS SQL Server, например, имеется роль **public** — особая роль, на которую автоматически назначен КАЖДЫЙ пользователь в базе. Крайне важно ограничивать привилегии, данные роли **public**, так как ее автоматически получают все пользователи. Часто это первый вектор атаки.

Sysadmin (SQL Server) / **root** (MySQL) / **postgres** (PostgreSQL) — учетные записи с неограниченными правами во всей СУБД. Обычным пользователям и приложениям должен быть категорически запрещен доступ к этим учетным записям. Они должны использоваться только для нужд администрирования.

ВАЖНОЕ ПРАВИЛО:

никогда не используйте предопределенные системные роли (их довольно много в каждой СУБД) для работы приложений или рядовых пользователей. Всегда создавайте собственные роли с наборами минимально необходимых прав.

14.3. Типы привилегий. Что можно разрешить

Привилегии (права) определяют, какие действия разрешено выполнять над объектами.

Их можно разделить на две большие категории:

- **Системные привилегии (Server-Level / Administrative Privileges)** — управляют операциями на уровне всей СУБД либо операциями создания или изменения структур (CREATE/ALTER/BACKUP/DROP DATABASE, ALTER LOGIN, SHUTDOWN и пр.).

Эти привилегии обычно требуются только администраторам баз данных (DBA). Разработчикам и приложениям они крайне редко нужны и представляют огромный риск при неконтролируемом назначении.

- **Объектные привилегии (Object-Level Privileges)** — управляют операциями над конкретными объектами внутри базы данных: таблицами, представлениями, хранимыми процедурами, функциями, схемами. Это те привилегии, с которыми разработчик взаимодействует чаще всего при настройке доступа приложения или пользовательских ролей.

  К таким привилегиям относятся операции по выборке, вставке, обновлению и удалению данных, а также по выполнению хранимых процедур или функций. К этим же привилегиям относятся операции по созданию внешних ключей, изменению (ALTER) или удалению (DROP) объектов.

Существует также шаблон для разрешения всех операций сразу (ALL / ALL PRIVILEGES), при использовании которого также необходимо соблюдать осторожность.

14.4. Управление пользователями и паролями

Команды для создания, модификации и удаления пользователей и ролей в разных СУБД богаты возможностями и имеют множество дополнительных необязательных ключей, опций и настроек. Ниже будут перечислены лишь самые необходимые для понимания материала команды, полную же информацию по тонкостям создания учетных записей рекомендуется получать из официальной документации.

14.4.1. Основные команды

Создание и последующее использование учетной записи пользователя "user1" с паролем "password123":

MS SQL Server

```
CREATE LOGIN user1 WITH PASSWORD = 'password123';
GO
```

В SQL Server следует различать учетную запись (LOGIN) и пользователя (USER). После создания пользователя эту учетную запись необходимо дополнительно привязать к требуемой базе данных (например, test_db):

```
USE test_db;
CREATE USER user1 FOR LOGIN user1;
GO
```

Чтобы удалить пользователя, надо проделать эти операции в обратном порядке:

```
-- Сначала удаляем привязку пользователя к базе данных
USE test_db;
DROP USER user1;
GO

-- Затем удаляем сам логин
DROP LOGIN user1;
GO
```

MySQL

Создание:

```
CREATE USER 'user1' IDENTIFIED BY 'password123';
```

Удаление:

```
-- Удаляем пользователя
DROP USER 'user1';
FLUSH PRIVILEGES;
```

Команда FLUSH PRIVILEGES требует от сервера немедленно перечитать все таблицы предоставления привилегий, чтобы изменения вступили в силу немедленно.

В силу своей ориентации на веб, в MySQL существует определенный синтаксис для указания пользователей, который включает в себя как имя пользователя, так и хост, с которого этот пользователь имеет право подключаться к базе данных. Вы можете использовать просто user1, но это будет означать, что пользователю user1 разрешен доступ с любого хоста. Это может быть небезопасно, если ваш MySQL-сервер доступен из сети, так как любой пользователь с правильным паролем сможет подключиться к базе данных с любого IP-адреса.

Альтернативно можно воспользоваться формой: 'user1'@'localhost'. В этом случае пользователь user1 сможет подключаться только с локальной машины (то есть с той же машины, на которой остановлен сервер MySQL).

Если ваша база данных находится в сети и пользователи подключаются удаленно через Интернет, вы можете указать конкретный IP-адрес или диапазон адресов, с которых разрешен доступ. Например, если ваш удаленный пользователь будет подключаться с IP-адреса 192.168.1.100, вы можете создать пользователя следующим образом: 'user1'@'192.168.1.100'. Это ограничит доступ к базе данных именно этому IP. Если потребуется разрешить доступ сразу нескольким IP-адресам, можно настроить пользователя с использованием шаблонов или подстановочных знаков, например, 'user1'@'%.example.com', что позволит всем пользователям с домена example.com подключаться под учетной записью user1.

Более подробную информацию вы найдете в документации к MySQL.

PostgreSQL

```
CREATE ROLE user1 LOGIN PASSWORD 'password123';
```

В данном случае ключевое слово LOGIN говорит о том, что мы создаем роль пользователя с правом входа в СУБД, в отличие от NOLOGIN — роли, при помощи которой войти нельзя, но можно присоединять к ней другие роли. Помните, что в PostgreSQL концепции роли и пользователя объединены.

SKLADCHIK.ORG

Удаление пользователя:

```
DROP ROLE user1;
```

14.4.2. Смена пароля

Администратор или сам пользователь (при условии, что пользователь имеет соответствующее разрешение на обновление собственного пароля) может изменять пароль для учетной записи. В примерах ниже для пользователя user1 устанавливается пароль "new_password".

MS SQL Server

```
ALTER LOGIN user1 WITH PASSWORD='new_password'
```

MySQL

```
ALTER USER 'user1' IDENTIFIED BY 'new_password';
```

ПРИМЕЧАНИЕ.

В некоторых материалах вы можете встретить синтаксис:

```
SET PASSWORD FOR 'user1' = PASSWORD('new_password');
```

Однако данный метод устарел, и начиная с версии 8 рекомендуется использовать первый вариант (IDENTIFIED BY).

PostgreSQL

```
ALTER ROLE user1 WITH PASSWORD 'new_password';
```

14.5. Управление ролями

Роли создаются во всех трех СУБД одинаково:

```
CREATE ROLE <название_роли>;
```

Чтобы назначить роль пользователю (например, роль *analyst* пользователю user1), в MS SQL Server используйте команду:

```
ALTER ROLE analyst ADD MEMBER user1;
```

В MySQL / PostgreSQL:

```
GRANT analyst to user1;
```

Снимаются роли у пользователей следующими командами:

```
-- MS SQL Server
ALTER ROLE analyst DROP MEMBER user1;
-- MySQL/PostgreSQL:
REVOKE analyst FROM user1;
```

Удаление роли производится при помощи команды

```
DROP ROLE <название_роли>;
```

ПРИМЕЧАНИЕ.

В MS SQL Server и MySQL удалить роль нельзя, пока из нее не будут исключены все пользователи. В PostgreSQL можно воспользоваться модификатором CASCADE:

```
DROP ROLE <название_роли> CASCADE;
```

При выполнении этой команды в PostgreSQL роль отзывается у всех пользователей, а все зависимые объекты (если роль является их владельцем) удаляются или передаются другому владельцу. **Используйте с осторожностью!**

14.6. Предоставление и отзыв привилегий

Ниже представлен общий синтаксис предоставления (GRANT) и отзыва (REVOKE) прав для пользователя или роли:

```
GRANT <права> ON <объект> TO <пользователь/роль>;
REVOKE <права> ON <объект> FROM <пользователь/роль>;
```

Права можно перечислять через запятую, в качестве значения можно указывать некоторые команды: DDL, SELECT, INSERT, UPDATE, DELETE — что разрешает или запрещает выполнение запросов соответствующего типа. Помимо этого, допускается указание EXECUTE (вызов хранимых процедур или функций) и ALL (в случае с MySQL: ALL PRIVILEGES) — для полного доступа к объекту.

В качестве объекта допускается указывать имя базы данных, схемы или иных объектов СУБД, которым предоставляются (GRANT) или у которых отзываются (REVOKE) права. Синтаксис для GRANT и для REVOKE идентичен, за исключением TO и FROM.

14.6.1. Правила указания объектов

В разных СУБД правила указания объектов различаются.

MS SQL Server

Тип объекта	Синтаксис	Пример
Конкретная БД		`GRANT CREATE TABLE TO user;` Позволяет пользователю *user* создавать таблицы в базе данных
Схема	`SCHEMA::имя_схемы`	`GRANT SELECT ON SCHEMA::sales TO user;` Позволяет пользователю *user* выполнять запросы типа SELECT в схеме **sales**
Конкретная таблица	`OBJECT::имя_таблицы`	`GRANT UPDATE ON OBJECT::orders TO user;` Позволяет пользователю *user* обновлять таблицу **orders**

Тип объекта	Синтаксис	Пример

Столбцы таблицы	Поддержка через представления	`CREATE VIEW restricted_view AS SELECT col1, col2 FROM table_name;`
		`GRANT SELECT ON restricted_view TO user;`
		Создает ограниченное двумя столбцами представление таблицы **table_name**, затем предоставляет пользователю *user* выбирать данные из этого представления

Хранимые процедуры или функции	`OBJECT::имя_процедуры`	`GRANT EXECUTE ON OBJECT::proc TO user;` Позволяет пользователю *user* запускать процедуру **proc**

ПРИМЕЧАНИЕ.

Префикс OBJECT:: или SCHEMA:: можно не использовать, если не возникает неоднозначности при разрешении имени объекта. Например, если у вас в базе присутствует схема **sales** и еще есть таблица **sales**, использование префиксов будет необходимо.

Права уровня БД (например, CREATE TABLE) не требуют указания объекта.

MySQL

Тип объекта	Синтаксис	Пример
СУБД целиком	`*.*`	`GRANT SELECT ON *.* TO user;` Позволяет пользователю *user* выполнять запросы SELECT по всем объектам базы
Конкретная БД	`имя_базы.*`	`GRANT INSERT ON shop.* TO user;` Позволяет пользователю *user* выполнять вставку данных для базы данных **shop**

Тип объекта	Синтаксис	Пример
Конкретная таблица	`имя_базы.имя_таблицы`	`GRANT DELETE ON shop.orders TO user;` Позволяет пользователю *user* выполнять удаление в таблице **orders** базы **shop**
Столбцы таблицы	`(столбец1, столбец2...)`	`GRANT SELECT (id, name) ON shop.products TO user;` Позволяет пользователю *user* выполнять выборку столбцов *id* и name из таблицы **products** базы **shop**
Хранимые процедуры	`PROCEDURE процедура`	`GRANT EXECUTE ON PROCEDURE get_data TO user;` Позволяет пользователю *user* запускать процедуру **get_data**
Функции	`FUNCTION функция`	`GRANT EXECUTE ON FUNCTION calc TO user;` Позволяет пользователю *user* вызывать функцию **calc**

PostgreSQL

Тип объекта	Синтаксис	Пример
Конкретная БД	`DATABASE имя_базы`	`GRANT CONNECT ON DATABASE shop TO user;` Позволяет пользователю *user* подключаться к базе данных **shop**
Схема	`SCHEMA имя_схемы`	`GRANT CREATE ON SCHEMA public TO user;` Позволяет пользователю *user* создавать объекты в схеме **public**

Тип объекта	Синтаксис	Пример
Конкретная таблица	`TABLE имя_таблицы`	`GRANT SELECT ON TABLE orders TO user;` Позволяет пользователю *user* выбирать данные из таблицы **orders**
Представление	`VIEW имя_представления`	`GRANT SELECT ON VIEW report TO user;` Позволяет пользователю *user* выбирать данные из представления **report**
Столбцы таблицы	`(столбец1, столбец2...)`	`GRANT UPDATE (price) ON TABLE products TO user;` Позволяет пользователю *user* выполнять обновление столбца *price* в таблице **products**
Хранимые процедуры	`PROCEDURE процедура (аргументы)`	`GRANT EXECUTE ON PROCEDURE update_salary(integer, numeric) TO user;` Позволяет пользователю user запускать процедуру **update_salary** с двумя аргументами *integer* и *numeric*
Функции	`FUNCTION функция (аргументы)`	`GRANT EXECUTE ON FUNCTION calc(int) TO user;` Позволяет пользователю *user* вызывать функцию **calc** с параметром типа INT
Все таблицы в схеме	`ALL TABLES IN SCHEMA имя_схемы`	`GRANT SELECT ON ALL TABLES IN SCHEMA public TO user;` Предоставляет пользователю *user* права на выборку данных из всех таблиц в схеме **public**

ПРИМЕЧАНИЕ.

Необходимость указания сигнатур (количества и типов параметров) для процедур и функций обусловлено поддержкой перегрузки в PostgreSQL, соответственно, права могут быть назначены индивидуально на каждую перегрузку отдельно.

14.6.2. Наследование ролей в PostgreSQL

В PostgreSQL роли могут наследовать привилегии от других ролей. Этот механизм необходим в силу объединения ролей и пользователей в одну сущность.

При создании роли можно дополнительно указывать ключевые слова INHERIT (поведение по умолчанию — роль будет наследовать привилегии) или NOINHERIT (требует явного указания полномочий).

```
CREATE ROLE parent_role; -- Роль "родителя"
CREATE ROLE child_role1; -- Автоматическое наследование
CREATE ROLE child_role2 NOINHERIT; -- Роль не наследует привилегии

-- Добавляем child_role1 в родительскую роль parent_role
GRANT parent_role TO child_role1;

-- Добавляем child_role2 в родительскую роль parent_role
GRANT parent_role TO child_role2;

-- Назначение привилегий родительской роли
GRANT SELECT ON table1 TO parent_role;
```

Роль **child_role1** автоматически наследует права от **parent_role**, так как она была создана без ключа NOINHERIT. Значит, она получает права на SELECT на таблицу table1. В то же время роль **child_role2**, несмотря на свое членство в **parent_role**, **не наследует** права, так как была создана с опцией NOINHERIT. Следовательно, она **не имеет** прав SELECT на таблицу table1.

14.6.3. Антипаттерны

GRANT ALL ... предоставляет неограниченный доступ к объектам. Как бы ни было соблазнительно использовать эту инструкцию "для собственных нужд", старайтесь вообще не пользоваться данной командой.

Роль **public** в PostgreSQL не должна использоваться для прав по умолчанию. Любой пользователь автоматически получает эти права, что упрощает атаки.

Например, в PostgreSQL отзовите все привилегии с **public**:

```
REVOKE ALL ON DATABASE mydb FROM public;
```

WITH GRANT OPTION... позволяет пользователям переделегировать права, создавая риски неконтролируемого распространения доступа. Контролируйте его через регулярный аудит.

14.7. Аудит и профилактика

Базы данных могут эксплуатироваться годами и даже десятилетиями. За такое время в них могут смениться десятки, сотни или даже тысячи пользователей, включая администраторов и пользователей с чрезмерными наборами привилегий. Старые учетные записи и избыточные роли создают угрозы для безопасности. Регулярный аудит позволяет выявить уязвимости до их эксплуатации злоумышленниками и минимизировать поверхность атаки.

Все листинги доступны в электронном приложении на сайте издательства.

14.7.1. SQL Server

Для анализа привилегий на уровне базы данных можно пользоваться системными представлениями:

```
sys.database_permissions
```

Это представление используется для получения сведений о правах доступа, установленных в текущей базе данных. Оно показывает, кто имеет

доступ к объектам (таблицам, процедурам, представлениям и другим ресурсам) и какие именно права предоставлены каждому объекту.

Основные поля:

- `major_id` — идентификатор защищаемого объекта (например, таблицы).
- `class_desc` — класс защищаемого объекта (TABLE, VIEW, PROCEDURE и др.).
- `permission_name` — наименование разрешения (например, SELECT, INSERT, EXECUTE).
- `state_desc` — состояние разрешения (GRANT, DENY, REVOKE).
- `grantee_principal_id` — ID пользователя/роли, которому выдано разрешение.
- `grantor_principal_id` — ID пользователя/роли, выдавшего разрешение.

sys.database_role_members

sys.database_role_members представляет собой справочник связей между ролями и пользователями в текущей базе данных. Используется для выяснения, кто входит в какую роль и какой набор прав получает автоматически от данной роли.

Основные поля:

- `role_principal_id` — идентификатор роли.
- `member_principal_id` — идентификатор пользователя или другой роли, входящей в данную роль.
- `database_id` — идентификатор базы данных, к которой относятся связи ролей и пользователей.

sys.database_principals

sys.database_principals используется для получения списка всех объектов безопасности (пользователей, ролей, приложений и других сущностей), существующих в текущей базе данных. Позволяет увидеть полный перечень именованных субъектов и определить, какие типы объектов представлены.

Основные поля:

- `principal_id` — уникальный идентификатор сущности.
- `name` — имя субъекта (пользователь, роль и т.п.).
- `type_desc` — тип сущности (USER, ROLE, APPLICATION_ROLE и др.).
- `default_schema_name` — схема по умолчанию для пользователя.

Кроме указанных представлений для детализации по ролям, можно воспользоваться вызовом:

```
-- Детализация по ролям
EXEC sp_helprotect;
```

Например, информацию о членстве в ролях можно получить через следующий запрос:

```
-- Пользователи и роли
SELECT dp.name AS Username,
       rp.name AS Rolename
FROM sys.database_role_members drm
JOIN sys.database_principals dp
  ON drm.member_principal_id = dp.principal_id
JOIN sys.database_principals rp
  ON drm.role_principal_id = rp.principal_id;
```

Поиск "опасных" привилегий (участников роли db_owner, например) можно осуществить в представлении **sys.database_principals** следующим образом:

SKLADCHIK.ORG


```
SELECT name AS username FROM sys.database_principals
WHERE IS_ROLEMEMBER('db_owner', name) = 1;
```

Также рекомендуется проводить аудит на разрешение ALTER ANY DATABASE, CONTROL SERVER, EXECUTE на системных процедурах.

Для проверки прав конкретного пользователя (user1) можно воспользоваться запросом:

```
-- Выданные полномочия конкретному пользователю (user1):
SELECT perm.permission_name, obj.name AS object_name
FROM sys.database_permissions perm
JOIN sys.objects obj ON perm.major_id = obj.object_id
WHERE perm.grantee_principal_id = USER_ID('user1');
```

14.7.2. MySQL

MySQL сохраняет информацию о привилегиях в специальных таблицах системной базы **mysql**. Для проверки текущих настроек рекомендуется проанализировать следующие таблицы:

- **mysql.user** — таблица, содержащая глобальные привилегии каждого пользователя.
- **mysql.db** — привилегии на уровне отдельных баз данных.
- **mysql.tables_priv**, **mysql.columns_priv**, **mysql.procs_priv** — специфические привилегии на объекты уровня таблиц, столбцов и хранимых процедур соответственно.
- **mysql.roles_mapping** — отображает связь между пользователями и ролями.

Ниже приведен ряд полезных запросов для аудита ролей и привилегий (в электронном приложении доступно еще несколько):

```
-- Активные роли пользователей
SELECT FROM_USER AS Role, TO_USER AS Username
FROM mysql.role_edges;
```

```
-- Пользователи с глобальными правами SUPER/ALL
SELECT User, Host
FROM mysql.user
WHERE Super_priv = 'Y'
   OR (Create_user_priv = 'Y' AND Shutdown_priv = 'Y');

-- Пользователи с полным доступом к конкретной БД
SELECT User, Host
FROM mysql.db
WHERE Db = 'your_db'
  AND Create_priv = 'Y'
  AND Drop_priv = 'Y'
  AND Alter_priv = 'Y'
  AND Grant_priv = 'Y';

-- Привилегии конкретного пользователя (user1)
-- Через встроенную команду
SHOW GRANTS FOR 'user1';
```

14.7.3. PostgreSQL

В PostgreSQL информация о ролях и привилегиях находится в системных представлениях схемы **information_schema**.

```
-- Все привилегии на объекты базы данных
SELECT
    grantee AS username,
    table_schema AS schema_name,
    table_name AS object_name,
    privilege_type AS permission,
    is_grantable
FROM information_schema.table_privileges

UNION ALL
-- Привилегии на функции
SELECT
    grantee,
```

```
    specific_schema,
    routine_name,
    privilege_type,
    is_grantable
FROM information_schema.routine_privileges;

-- Все роли и пользователи
SELECT
    oid AS principal_id,
    rolname AS name,
    CASE
        WHEN rolcanlogin THEN 'USER'
        ELSE 'ROLE'
    END AS type_desc
FROM pg_roles;

-- Членство в ролях
SELECT
    r.rolname AS role_name,
    m.rolname AS member_name
FROM pg_roles r
JOIN pg_auth_members am ON r.oid = am.roleid
JOIN pg_roles m ON am.member = m.oid;

-- Проверка конкретного права пользователя user1
-- выбирать данные (SELECT) из таблицы table
SELECT has_table_privilege('user1', 'table', 'SELECT');
```

Что узнали и чему мы научились?

Вопросы безопасности данных — неотъемлемая часть проектирования, разработки и эксплуатации баз данных. Ответственность за защиту информации лежит не только на администраторах (DBA), но и на разработчиках, которые лучше всех знают структуру БД и точки потенциального риска.

Мы рассмотрели следующие аспекты:

- **Принцип наименьших привилегий** — основа безопасности.
- **Ролевая модель** — эффективный инструмент управления, который значительно упрощает управление доступом для большого числа пользователей, делает права прозрачными и облегчает аудит.
- **Схемы** — логические контейнеры безопасности. Организация объектов БД в схемы позволяет управлять доступом на уровне функциональных областей (например, sales, hr), предоставляя права целым группам объектов.
- **Предопределенные роли** (public, sysadmin, root, postgres) обладают чрезмерными привилегиями. Их использование для обычных пользователей или приложений категорически недопустимо и создает огромные риски. Всегда создавайте собственные роли с минимально необходимыми правами.
- **Команды GRANT** и **REVOKE** — основной инструмент управления доступом. Важно точно указывать объект (база, схема, таблица, столбец, процедура) и действие (SELECT, INSERT, UPDATE, DELETE, EXECUTE). Избегайте слепого использования ALL PRIVILEGES и WITH GRANT OPTION без крайней необходимости.
- **Регулярный аудит** — обязательная практика. Безопасность — непрерывный процесс.

Эффективная безопасность данных достигается только совместными усилиями разработчиков (понимающих структуру и логику данных), администраторов БД (настраивающих аутентификацию, шифрование, резервное копирование) и системных администраторов (обеспечивающих безопасность ОС и сети).

Заключение

Данная книга предлагает системный подход изучения SQL «с нуля» до весьма продвинутых концепций. Освоив представленный материал, читатель, при должном усердии и некоторой практике, может претендовать на весьма продвинутый уровень владения языка SQL.

Полученные теоретические знания, однако, ничто без практики. При подготовке материала этой книги для каждой главы был разобран детальный пример с созданием отдельной базы данных в любой из рассматриваемых нами СУБД с приложением листингов для экспериментов. Старайтесь не ограничиваться прочитанным материалом, экспериментируйте, создавайте таблицы, модифицируйте их, экспериментируйте с запросами, не оставляйте тему, пока вы не почувствуете, что не пришло понимание материала на интуитивном уровне.

Несмотря на то, что в книге обобщены и систематизированы все основные аспекты языка SQL, безусловно, многое осталось без внимания, потому что охватить полностью все нюансы работы целых трех СУБД в одной книге — задача непосильная.

Куда двигаться дальше?

Углубляйте знания конкретной СУБД! Примените свои знания на практике, и продолжите более глубокое изучение всех нюансов какой-либо конкретной СУБД по вашему выбору. Каждая из них обладает уникальными расширениями языка SQL, специфическими типами данных, имеет особенности в работе оптимизатора, предлагает собственные инструменты для администрирования и мониторинга. Регулярно обновляйте свои знания, изучая официальную документацию. Знания, полученные в этой книге, позволят вам лучше ориентироваться в огромном количестве материала.

Больше практики! Придумайте небольшую задачу (личный бюджет, каталог книг/фильмов, трекер тренировок) и реализуйте ее от проектирования базы до сложных отчетов. Используйте ресурсы вроде LeetCode, HackerRank, SQLZoo, Mode Analytics Tutorials или Codewars. Они предлагают множество задач разного уровня сложности на реальных (или приближенных к реальным) данных. Найдите интересные открытые наборы данных и импортируйте их в свою СУБД. Пробуйте отвечать на нетривиальные вопросы с помощью запросов. Участвуйте в обсуждении реальных практических задач на площадках вроде Habr QnA или StackOverflow.

Уделяйте больше внимания вопросам производительности и оптимизации. Научитесь читать планы выполнения и анализировать узкие места, профилировать запросы, создавать эффективные индексы. Именно знание этих аспектов будет отличать вас как профессионала на фоне любителей.

Начните постигать искусство **администрирования баз данных (DBA)**. Это целый отдельный мир, заслуживающий отдельной книги. Даже если вы не планируете становиться профессионалом в этой области, понимание основ администрирования поможет вам повысить свой уровень в разработке или аналитике. В этой книге оставлены без внимания такие аспекты, как резервное копирование и восстановление, мониторинг производительности и ресурсов сервера, управление политиками безопасности и многие другие вещи из мира администрирования баз данных.

Смежные технологии и направления

Освойте **смежные технологии и направления**:

- **ORM (Object-Relational Mapping)**. Поймите, как фреймворки (Hibernate, Entity Framework, SQLAlchemy) взаимодействуют с БД, какие запросы генерируют и как это влияет на производительность. Знание SQL позволит вам эффективно использовать ORM и оптимизировать их работу.
- **ETL/ELT-процессы**. Инструменты извлечения, преобразования и загрузки данных (Apache Airflow, Talend, Pentaho, dbt) широко используют SQL. Ваши навыки здесь будут неоценимы.
- **Аналитика и Business Intelligence (BI)**. Углубляйтесь в построение сложных отчетов, дашбордов (Power BI, Tableau, Superset, Metabase), где SQL — ключевой инструмент для подготовки данных.

- **Big Data и NoSQL.** Изучение распределенных систем (Hadoop, Spark SQL) и NoSQL баз данных (MongoDB, Cassandra, Redis) расширит ваш кругозор и поможет выбрать правильный инструмент для задачи, основанный на понимании их отличия от реляционных СУБД.
- **Облачные базы данных (Cloud SQL, Azure SQL Database, Amazon RDS/Aurora).** Понимание особенностей работы и управления БД в облачных средах становится стандартным требованием.

SQL — не просто язык запросов; это фундаментальный навык для работы с данными, который остается актуальным десятилетиями. Данная книга дала вам карту и компас. Теперь ваш путь лежит через **реальный опыт**. Не бойтесь совершать ошибки, задавать вопросы в различных сообществах вроде Stack Overflow и пр., читать блоги экспертов и изучать исходный код сложных запросов.

Постоянно ставьте перед собой новые задачи, требующие нетривиальных решений на SQL. Помните: мастерство приходит с повторением, анализом и готовностью разбираться в деталях.

Удачи вам на этом увлекательном пути превращения знаний в настоящее мастерство работы с данными!

Глоссарий терминов

A

- **ACID** – Акроним, описывающий ключевые свойства транзакций: **А**томарность (Atomicity), **С**огласованность (Consistency), **И**золированность (Isolation), **Д**олговечность (Durability).

- **Абстракция данных (Data Abstraction)** – Одна из целей использования представлений. Сокрытие сложности структуры базы данных и сложных запросов за простым интерфейсом (именем представления).

- **Автонумерация (Auto-increment)** – Механизм, позволяющий автоматически генерировать уникальные последовательные числовые значения для столбца (обычно первичного ключа) при вставке новых записей. Реализуется с помощью ключевых слов `AUTO_INCREMENT` (MySQL), `SERIAL` или `IDENTITY` (PostgreSQL), `IDENTITY` (MS SQL Server).

- **Авторизация (Authorization)** – Процесс предоставления уже аутентифицированному пользователю или роли прав на выполнение определенных действий в базе данных.

- **Агрегатные функции** – Функции SQL (например, `COUNT`, `SUM`, `AVG`), которые выполняют вычисление на наборе строк и возвращают одно результирующее значение.

SKLADCHIK.ORG

- **Аномалия параллельного доступа (Concurrency Anomaly)** – Проблема, возникающая при одновременном выполнении транзакций без должной изоляции. Основные виды: грязное, неповторяющееся и фантомное чтение.

- **Аудит (Audit)** – Процесс регулярной проверки и анализа назначенных прав доступа, действий пользователей и конфигурации безопасности с целью выявления аномалий, нарушений политик безопасности и избыточных привилегий.

- **Аутентификация** – Процесс проверки подлинности пользователя или системы при подключении к СУБД (например, с помощью логина и пароля или встроенных средств Windows).

Б

- **Беззнаковый тип (UNSIGNED)** – Модификатор целочисленных типов (в MySQL), указывающий, что переменная может хранить только неотрицательные значения (0 и выше), тем самым увеличивая верхнюю границу диапазона.

- **Безопасность данных (Data Security)** – Одна из ключевых функций представлений. Ограничение доступа пользователей только к тем данным (столбцам и строкам), которые им необходимы, путем скрытия конфиденциальной информации в базовых таблицах.

- **Бизнес-аналитик** – Специалист, который использует данные для анализа процессов и принятия решений в бизнесе. Знание SQL значительно расширяет его возможности по извлечению и анализу данных.

- **Бизнес-логика (Business Logic)** – Набор правил и алгоритмов, определяющих работу приложения: как данные могут создаваться, отображаться, изменяться и взаимодействовать друг с другом. Может быть реализована

как на стороне клиентского приложения, так и на стороне СУБД с использованием хранимых процедур, функций и триггеров.

- **Блокировка (Lock)** – Механизм защиты данных от несогласованной параллельной обработки несколькими пользователями или приложениями.

- **Блокировка диапазонов (Range Lock)** – Тип блокировки, который предотвращает фантомное чтение, блокируя не только существующие строки, но и диапазон значений, куда могли бы быть вставлены новые строки.

- **Блокировка обновления (Update Lock, U-Lock)** – Промежуточный тип блокировки, используемый, когда транзакция планирует изменить данные. Совместима с S-Lock, но конфликтует с другими U-Lock и X-Lock. Нужна для избегания взаимоблокировок в сценариях «чтение – обновление».

В

- **Ветвление (Branching)** – Конструкция в хранимых процедурах и функциях, позволяющая изменять порядок выполнения команд в зависимости от выполнения определенных условий (например, с помощью оператора IF... ELSE).

- **Взаимная блокировка (Deadlock)** – Ситуация, когда две или более транзакций бесконечно ожидают освобождения ресурсов (блокировок), захваченных друг другом.

- **Внешний ключ (Foreign Key)** – Ограничение, которое обеспечивает ссылочную целостность данных, связывая поле одной таблицы с первичным ключом другой таблицы.

- **Вставка данных (INSERT)** – Операция SQL для добавления новых записей (строк) в таблицу.

- **Вычислимый столбец (Computed Column)** – Столбец в таблице, значение которого не хранится напрямую, а вычисляется на основе других столбцов этой же таблицы с помощью заданного выражения. Может быть `VIRTUAL` (вычисляется при чтении) или `PERSISTED/STORED` (значение вычисляется и сохраняется при вставке/обновлении).

Г

- **Генератор последовательностей** – Функция или объект базы данных (например, `GENERATE_SEQUENCE()` в PostgreSQL), предназначенный для создания набора последовательных чисел. Часто используется для генерации тестовых данных или заполнения столбцов с порядковыми номерами.

- **Глубокая вложенность представлений (Deep View Nesting)** – Антипаттерн проектирования, при котором представления строятся на основе других представлений, создавая многоуровневые зависимости. Это может серьезно затруднить понимание запроса, анализ производительности и сопровождение системы.

- **Графическая оболочка (GUI Tool)** – Программа с графическим интерфейсом для удобной работы с базой данных (написания запросов, просмотра данных, управления структурой). Примеры: SSMS, MySQL Workbench, pgAdmin, DBeaver.

- **Группировка (GROUP BY)** – операция в SQL, используемая\ для группировки строк, имеющих одинаковые значения в указанных столбцах, с целью применения агрегатных функций к каждой группе отдельно.

- **Грязное чтение (Dirty Read)** – Аномалия, при которой транзакция видит незафиксированные изменения другой транзакции. Если та откатится, первая транзакция будет основываться на данных, которых никогда официально не существовало.

Д

- **DDL (Data Definition Language)** – Подмножество SQL, содержащее команды для определения и изменения структуры базы данных и её объектов (схем, таблиц, индексов, представлений). Основные команды: `CREATE`, `ALTER`, `DROP`.

- **DML (Data Manipulation Language)** – Подмножество SQL, содержащее команды для работы с данными внутри таблиц: выборка (SELECT), вставка (INSERT), обновление (UPDATE), удаление (DELETE). В отличие от DDL, который работает со структурой, DML работает с содержимым.

- **Данные** – Информация, представленная в формализованном виде, пригодном для обработки компьютером.

- **Даталогическое проектирование** – Процесс создания структуры базы данных: таблиц, столбцов, типов данных, ключей и отношений между таблицами.

- **Декларативный подход** – Подход в программировании и запросах, при котором описывается **что** нужно получить, а не **как** это сделать. SQL является декларативным языком.

- **Дерево индекса (Index Tree)** – Иерархическая структура данных (обычно B-дерево или его вариации), используемая для организации индекса. Состоит из корневого, промежуточных и листовых страниц (узлов), что позволяет эффективно находить данные по ключу.

- **Детерминированная функция (Deterministic Function)** – Функция, которая всегда возвращает один и тот же результат для одних и тех же входных параметров и состояния базы данных. Важно для создания индексов на основе вычисляемых столбцов или функций.

- **Диалект SQL** – Вариант (диалект) языка SQL, реализованный в конкретной СУБД. Включает уникальные расширения, типы данных или особенности синтаксиса поверх стандарта SQL (например, T-SQL для MS SQL, PL/pgSQL для PostgreSQL).

- **Действительные числа (REAL numbers)** – Числа, имеющие дробную часть. В SQL представлены типами `FLOAT`, `REAL`, `DOUBLE PRECISION` (приближённые) и `NUMERIC/DECIMAL` (точные).

Ж

- **Журнал транзакций (Transaction Log)** – Специальный служебный файл СУБД, в который записывается информация обо всех изменениях данных перед их внесением в основные таблицы. Используется для отката транзакций и восстановления после сбоев.

З

- **Знаковый тип (SIGNED)** – Модификатор целочисленных типов по умолчанию, позволяющий хранить как положительные, так и отрицательные значения.

И

- **Идентификатор (ID)** – Уникальный признак объекта, позволяющий отличить его от других объектов. В базах данных часто является первичным ключом.

- **Изменяющий запрос** – Запрос, который модифицирует данные (INSERT, UPDATE, DELETE, MERGE) в отличие от запроса на выборку (SELECT).

- **Изоляция транзакций (Isolation)** – Свойство ACID, гарантирующее, что параллельные транзакции не мешают друг другу.

- **Императивное программирование (Imperative Programming)** – Подход к программированию, при котором явно описывается **как** необходимо выполнить задачу, шаг за шагом (с использованием циклов, условий, переменных). Противопоставляется декларативному подходу. Хранимые процедуры и функции используют императивный стиль внутри декларативного языка SQL.

- **Индекс** – Специализированный объект СУБД, который ускоряет операции извлечения данных из таблицы по определенному столбцу, но замедляет операции вставки и обновления.

- **Индексированное представление (Indexed View)** – Особый тип представления в MS SQL Server, для которого создается кластеризованный индекс. Такое представление физически хранит данные и автоматически поддерживает их актуальность, но накладывает строгие ограничения на схему базовых таблиц (требует `WITH SCHEMABINDING`).

- **Инклюзивный столбец индекса (INCLUDE column)** – Столбец, добавленный в некластеризованный индекс (в MS SQL Server) с помощью ключевого слова `INCLUDE`. Его значение хранится в листовом уровне индекса, но сам столбец не используется для поиска и сортировки в индексе. Позволяет создать покрывающий индекс без увеличения размера ключевой части индекса.

- **Инстанс (Экземпляр сервера)** – Запущенная копия сервера СУБД. На одном физическом сервере может быть запущено несколько инстансов разных или одной и той же СУБД.

- **Искусственный ключ (Surrogate Key)** – Первичный ключ, который не имеет смыслового (бизнес-) значения и добавляется в таблицу исключительно для целей уникальной идентификации записей (например, автоинкрементный ID). Противопоставляется натуральному ключу.

К

- **Каскадное удаление/обновление (CASCADE)** – Действие, определяемое для внешнего ключа, при котором удаление или обновление записи в родительской таблице автоматически вызывает удаление или обновление связанных записей в дочерней таблице. Задаётся с помощью конструкций `ON DELETE CASCADE` и `ON UPDATE CASCADE`.

- **Кластеризованный индекс (Clustered Index)** – Тип индекса, который определяет физический порядок хранения данных в таблице. Таблица может иметь только один кластеризованный индекс. Данные в таблице физически упорядочены по ключу этого индекса.

- **Компромисс индекса (Index Trade-off)** – Баланс, который необходимо соблюдать при использовании индексов: ускорение операций чтения (SELECT) достигается за счет замедления операций записи (INSERT, UPDATE, DELETE) и увеличения занимаемого дискового пространства.

- **Кодировка (Character Encoding)** – Набор правил для преобразования символов алфавита в числовые коды для хранения в компьютере. Например, ASCII, UTF-8, Windows-1251.

- **Колляция (Collation)** – Набор правил, определяющих порядок сортировки и сравнения символьных строк с учётом регистра, акцентов и особенностей алфавита конкретного языка.

- **Конфликт ограничений** – Ситуация, при которой операция с данными (чаще всего вставка или обновление) нарушает одно из **ограничений** (Constraint) целостности, установленных для таблицы (`UNIQUE`, `CHECK`, `PRIMARY KEY`, `FOREIGN KEY`).

- **Команда GO** – Специальная команда в утилите **sqlcmd** для MS SQL Server, которая указывает на выполнение всех введённых до неё команд.

- **Командная строка (CLI)** – Текстовый интерфейс для взаимодействия с операционной системой или программами (например, СУБД) путем ввода команд.

- **Корпоративная среда** – Информационная инфраструктура крупных компаний, где часто используются коммерческие СУБД, такие как Microsoft SQL Server или Oracle.

- **Коррелируемый подзапрос (Correlated Subquery)** – Подзапрос, который выполняется для каждой строки внешнего запроса, так как ссылается на значения из этой строки. Часто менее производителен, чем некоррелируемые подзапросы или JOIN.

- **Курсор (Cursor)** – Механизм в СУБД, позволяющий последовательно обрабатывать результат SQL-запроса построчно. Позволяет применять сложную логику к каждой строке набора данных, но часто менее эффективен.

Л

- **Логическое разделение данных (Logical Data Separation)** – Использование представлений для предоставления разным пользователям или приложениям разных «срезов» одних и тех же данных в соответствии с их потребностями и правами доступа.

М

- **Масштаб (Scale)** – В типе данных `NUMERIC/DECIMAL` – количество десятичных разрядов (цифр) после запятой.

- **Материализованное представление (Materialized View)** – Представление, которое физически хранит результат своего запроса на диске. Данные в нем необходимо периодически обновлять (REFRESH), но оно обеспечивает высокую производительность для сложных и редко меняющихся запросов.

- **Менеджер блокировок (Lock Manager)** – Компонент СУБД, отвечающий за выдачу, проверку совместимости и освобождение блокировок.

- **Метаданные** – Данные о данных. В контексте СУБД – это информация о структуре базы данных: имена таблиц, столбцов, типы данных и т.д.

Н

- **Натуральный ключ (Natural Key)** – Первичный ключ, который имеет смысловое (бизнес-) значение и используется для идентификации записи в реальном мире (например, артикул товара, номер паспорта). Часто менее предпочтителен, чем искусственный ключ, из-за возможной изменчивости и сложности обеспечения уникальности.

- **Национальные типы данных (NCHAR, NVARCHAR)** – Типы данных для хранения символьных строк в кодировке Unicode, что позволяет хранить текст на разных языках мира. Буква 'N' означает 'National'.

- **Недетерминированная функция (Non-deterministic Function)** – Функция, которая может возвращать разные результаты для одних и тех же входных параметров из-за зависимости от внешних факторов (например, текущего времени `GETDATE()`).

- **Некластеризованный индекс (Non-Clustered Index)** – Тип индекса, который создает отдельную структуру данных от самой таблицы. Эта структура содержит ключи индекса и указатели (например, `ROWID` или ключ кластеризованного индекса) на соответствующие строки в таблице. На одной таблице может быть создано несколько некластеризованных индексов.

- **Неповторяющееся чтение (Non-repeatable Read)** – Аномалия, при которой транзакция читает одну и ту же строку

дважды и видит разные значения, потому что между чтениями другая транзакция изменила и зафиксировала эту строку.

- **Нормализация** – Процесс организации данных в базе данных для уменьшения избыточности и улучшения целостности данных. Включает разделение данных по таблицам и установление связей между ними.

- **NULL** – Специальное значение (маркер), обозначающее отсутствие данных или неизвестное значение. Не является нулём, пустой строкой или пробелом.

О

- **Обновляемое представление (Updatable View)** – Представление, через которое можно выполнять операции модификации данных (INSERT, UPDATE, DELETE). Имеет строгие ограничения: обычно должно быть основано на одной таблице, содержать ее первичный ключ и не содержать агрегаций, соединений (JOIN) или других сложных конструкций.

- **Обобщённое табличное выражение (CTE, Common Table Expression)** – Временный именованный набор результатов, определённый в рамках выполнения одного оператора SELECT, INSERT, UPDATE или DELETE. Улучшает читаемость и структурированность сложных запросов, позволяя повторно использовать результат подзапроса.

- **Окно (Window)** – Набор строк, над которым работает оконная функция. Определяется с помощью предложения `OVER()`.

- **Оконная функция (Window Function)** – Функция, которая выполняет вычисления *across a set of table rows*, каким-то образом связанных с текущей строкой. В отличие от агрегатных функций с `GROUP BY`, оконные функции не сворачивают строки в группы, а возвращают значение для каждой строки исходного набора.

SKLADCHIK.ORG

- **Ограничение (Constraint)** – Правило, поддержания целостности базы данных. Примеры: `PRIMARY KEY`, `FOREIGN KEY`, `NOT NULL`, `UNIQUE`.

- **Оператор SELECT** – Команда SQL для выборки данных из одной или нескольких таблиц. Основной инструмент для запросов.

- **Оптимизатор запросов (Query Optimizer)** – Компонент СУБД, который анализирует SQL-запрос и на основе статистики, структуры данных и доступных индексов выбирает наиболее эффективный план его выполнения.

- **Оптимистичная блокировка (Optimistic Locking)** – Техника, при которой блокировки не устанавливаются заранее, а конфликт параллельных изменений разрешается на этапе фиксации транзакции путем проверки версии или временной метки данных.

- **Откат транзакции (Rollback)** – Операция отмены всех изменений, произведённых транзакцией.

- **Отношение (Relation)** – В реляционной модели данных – таблица, содержащая строки (кортежи) и столбцы (атрибуты).

- **Оценка стоимости (Cost Estimation)** – Процесс, выполняемый оптимизатором запросов, при котором каждому возможному плану выполнения присваивается условная «стоимость» (в единицах, основанных на прогнозируемых затратах CPU, I/O и памяти). Выбирается план с наименьшей стоимостью.

П

- **Параметр (Parameter)** – Переменная, передаваемая в хранимую процедуру или функцию для управления её поведением. Может быть входным (IN), выходным (OUT) или входно-выходным (INOUT).

- **Первичный ключ (Primary Key)** – Столбец (или набор столбцов), однозначно идентифицирующий каждую строку в таблице. Не может содержать NULL значений.

- **Перегрузка (Overloading)** – Возможность создания нескольких процедур или функций с одним и тем же именем, но с разным набором параметров. Поддерживается, например, в PostgreSQL.

- **Пессимистичная блокировка (Pessimistic Locking)** – Подход к управлению параллельным доступом, при котором СУБД заранее устанавливает блокировки на данные, предполагая высокую вероятность конфликта между транзакциями.

- **План выполнения (Execution Plan)** – Детализированный пошаговый план, *generated* оптимизатором запросов, который описывает, как именно СУБД будет выполнять запрос: порядок доступа к таблицам, используемые индексы, методы соединений, сортировок и агрегаций.

- **Платформа** – Совокупность аппаратного и программного обеспечения, на котором работает система. В контексте СУБД – это операционная система (Windows, Linux, macOS) и сама СУБД.

- **Подзапрос (Subquery)** – Запрос SQL, вложенный внутрь другого запроса (в его `SELECT`, `FROM`, `WHERE` и т.д.). Может быть коррелируемым (ссылается на столбцы внешнего запроса) или некоррелируемым.

- **Пользователь (User)** – Учётная запись в СУБД, которая определяет права доступа к базам данным, таблицам и другим объектам.

- **Покрывающий индекс (Covering Index)** – Индекс, который содержит все столбцы, необходимые для выполнения определенного запроса. Позволяет удовлетворить запрос, обращаясь только к индексу, без дорогостоящего обращения к самой таблице (также называется «index-only scan»).

- **Порядок сортировки** – Последовательность, в которой данные возвращаются в результате запроса. Задаётся с помощью оператора `ORDER BY`.

- **Постраничный вывод (Pagination)** – Метод разделения большого результата запроса на меньшие части (страницы) для последовательного отображения.

- **Правила сопоставления** – См. **Колляция (Collation)**.

- **Представление (View)** – Виртуальная таблица, содержимое которой определяется запросом (SELECT). Не хранит данные физически, а предоставляет удобный и безопасный способ доступа к данным из одной или нескольких базовых таблиц через именованный запрос.

- **Префиксный индекс (Prefix Index)** – Индекс (в MySQL), который индексирует только первые N символов строкового столбца, а не всю строку. Используется для экономии дискового пространства при индексации длинных текстовых полей.

- **Привилегии (Privileges)** – Права, предоставленные пользователю или роли на выполнение определенных действий с объектами базы данных (например, SELECT, INSERT, UPDATE, DELETE).

- **Принцип наименьших привилегий (Principle of Least Privilege, PoLP)** – Фундаментальный принцип безопасности, согласно которому пользователю, роли или процессу должны предоставляться минимальные права доступа, необходимые для выполнения их законных задач.

- **Присоединение/Соединение) JOIN** – см. **Соединение**.

- **Процедура хранимая** – Набор SQL-инструкций, сохранённых в базе данных и выполняемых как единое целое.

- **Псевдоним (Alias)** – Краткое, часто условное имя, назначаемое объекту в запросе SQL с помощью ключевого слова `AS` (или без него) для упрощения написания запроса.

Р

- **Рекурсивное CTE (Recursive CTE)** – Специальный тип **обобщённых табличных выражений (CTE)**, который ссылается на собственное определение. Используется для работы с иерархическими или древовидными структурами данных.

- **Рекурсивный член (Recursive Member)** – В **рекурсивном CTE**: часть запроса, которая ссылается на CTE по имени и объединяется с **якорным членом** с помощью `UNION ALL`, пока не будут возвращены все строки.

- **Реляционная база данных (RDBMS)** – База данных, основанная на реляционной модели данных, которая организует данные в таблицы (отношения), связанные между собой.

- **Репликация** – Процесс копирования и поддержания данных из одной базы данных (источника) на другой (реплике) для повышения доступности и отказоустойчивости.

- **Роль (Role)** – Сущность в базе данных, группирующая набор привилегий. Пользователям назначаются роли, чтобы упростить управление правами доступа.

С

- **Сервер базы данных** – Компьютерная программа (процесс), которая предоставляет услуги по управлению базами данных и обработке запросов к ним. Также может обозначать компьютер, на котором запущена эта программа.

- **Системная привилегия (Server-Level / Administrative Privilege)** – Привилегия, управляющая операциями на уровне всей СУБД (например, `CREATE DATABASE`, `SHUTDOWN`, `ALTER LOGIN`). Обычно требуются только администраторам баз данных (DBA).

- **Скалярный запрос** – Запрос, который возвращает одно значение.

- **Скалярная функция (Scalar Function)** – Функция, которая возвращает одно значение определенного типа данных (например, число, строку, дату).

- **Соединение (Join)** – Операция SQL, которая позволяет комбинировать строки из двух или более таблиц на основе значения связанного столбца между ними.

- **Составной первичный ключ (Composite Primary Key)** – Первичный ключ, состоящий из двух или более столбцов, комбинация значений которых уникально идентифицирует каждую строку в таблице.

- **Статистика (Statistics)** – Метаданные, собираемые СУБД о распределении данных в таблицах и индексах (например, гистограммы распределения значений). Используются оптимизатором запросов для точной оценки селективности условий и выбора оптимального плана выполнения. Требуют регулярного обновления (`UPDATE STATISTICS`, `ANALYZE`)

- **СУБД (Система управления базами данных)** – Программное обеспечение для создания, управления и взаимодействия с базами данных. Примеры: MySQL, PostgreSQL, Microsoft SQL Server.

- **Субъект безопасности (Security Principal)** – Сущность (пользователь, роль, группа), которой могут быть назначены права доступа к объектам в системе.

- **Схема (Schema)** – Контейнер внутри базы данных, который группирует объекты (таблицы, представления, процедуры) в

логические пространства имён, часто для управления правами доступа.

Т

- **Таблица** – Основная структура для хранения данных в реляционной базе данных. Состоит из строк (записей) и столбцов (полей).

- **Табличная функция (Table-Valued Function)** – Функция, которая возвращает результат в виде таблицы, с которой можно работать как с обычной таблицей или представлением в SQL-запросах.

- **Тип данных** – Атрибут, который определяет, какие данные могут храниться в столбце таблицы, а также какие операции можно с этими данными *perform*. Примеры: `INT`, `VARCHAR`, `DATE`.

- **Точность (Precision)** – В типе данных `NUMERIC/DECIMAL` – общее количество значащих цифр (как до, так и после запятой).

- **Транзакция** – Последовательность операций с базой данных, которая выполняется как одно целое. Транзакция либо выполняется полностью, либо не выполняется вовсе (принцип атомарности).

- **Триггер (Trigger)** – Специальный тип хранимой процедуры, который автоматически выполняется при наступлении определенного события в таблице (INSERT, UPDATE, DELETE). Часто используется для аудита, обеспечения сложной целостности данных и поддержания денормализованных данных в актуальном состоянии.

У

- **Уникальность (Uniqueness)** – Ограничение, гарантирующее, что все значения в столбце (или комбинации столбцов) таблицы различаются.

- **Управление доступом** – Механизм защиты данных, который определяет, кто и какие операции может выполнять с объектами базы данных.

Ф

- **Фантомное чтение (Phantom Read)** – Аномалия, при которой транзакция повторяет запрос с некоторым условием и получает другой набор строк, потому что другая транзакция зафиксировала вставку или удаление строк, удовлетворяющих этому условию.

- **Фиксация транзакции (Commit)** – Операция успешного завершения транзакции, после которой все её изменения становятся постоянными и видимыми другим транзакциям (в соответствии с уровнем изоляции).

- **Фильтрация** – Процесс отбора записей в результирующем наборе по определённому условию с помощью оператора `WHERE`.

- **Фильтрованный / частичный индекс (Filtered / Partial Index)** – Индекс, построенный не для всей таблицы, а только для подмножества строк, удовлетворяющих определенному условию (WHERE). Экономит место и повышает производительность для запросов, использующих это условие.

- **Фрагментация индекса (Index Fragmentation)** – Состояние, при котором логический порядок страниц индекса не соответствует их физическому порядку на диске (внешняя фрагментация) или страницы индекса заполнены не полностью (внутренняя фрагментация). Возникает из-за частых операций DML (INSERT, UPDATE, DELETE) и **leads** к снижению производительности чтения. Устраняется операциями `REORGANIZE/REBUILD` (MS SQL), `OPTIMIZE TABLE` (MySQL), `REINDEX/VACUUM` (PostgreSQL).

- **Фрейм (Frame)** – Подмножество строк внутри **окна**, конкретно определяемое для вычисления функции. Задаётся с помощью `ROWS` или `RANGE`.

Х

- **Хранилище данных** – Специализированная база данных, предназначенная для хранения больших объёмов исторических данных и их анализа, а не для операционной обработки транзакций (OLTP).

Ц

- **Целостность данных (Data Integrity)** – Обеспечение точности, непротиворечивости и достоверности данных на протяжении их жизненного цикла. Поддерживается **ограничениями (constraints)**, транзакциями и другими механизмами СУБД.
- **Цикл (Loop)** – Конструкция в хранимых процедурах и функциях, позволяющая повторять выполнение блока команд несколько раз. Основные виды: `WHILE`, `LOOP`, `REPEAT` (зависит от СУБД).

Э

- **Эксклюзивная блокировка (Exclusive Lock, X-Lock)** – Блокировка, устанавливаемая при изменении данных (INSERT, UPDATE, DELETE). Только одна транзакция может удерживать X-блокировку на конкретном элементе данных. Конфликтует со всеми другими блокировками.

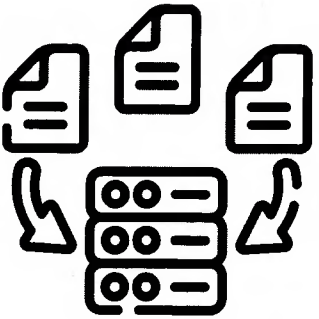
Ю

- **Юникод (Unicode)** – Стандарт кодирования символов, который предназначен для представления текста на большинстве письменных языков мира. Позволяет хранить в одном документе текст на разных языках. Основные кодировки: UTF-8, UTF-16.

Я

- **Явная блокировка (Explicit Locking)** – Ситуация, когда операция (например, создание индекса стандартным способом, `OPTIMIZE TABLE`, `VACUUM FULL`) полностью блокирует таблицу для других операций на время своего выполнения. Для избегания блокировок в PostgreSQL можно использовать `CREATE INDEX CONCURRENTLY` и `REINDEX CONCURRENTLY`.

- **Якорный член (Anchor Member) – В рекурсивном CTE:** начальная часть запроса, которая возвращает базовый набор строк.



SKLADCHIK.ORG

Список использованных источников информации

Список использованных источников информации:

- https://learn.microsoft.com/en-us/sql/t-sql/language-reference?view=sql-server-ver17
- https://dev.mysql.com/doc/refman/8.4/en/
- https://www.postgresql.org/docs/18/index.htmlSteven S. Skiena — The Algorithm Design Manual
- https://www.flaticon.com/ru/free-icon/data-collection_7440330

Издательство «Наука и Техника» выпускает книги более 25 лет!

Уважаемые авторы!

Приглашаем к сотрудничеству по созданию книг
по **IT-технологиям, электронике, электротехнике, медицине, педагогике.**

Наши преимущества:

- являемся одним из ведущих технических издательств страны;
- выпускаем книги большими тиражами, что положительно влияет на гонорар авторов;
- регулярно переиздаем тиражи, автоматически выплачивая гонорар за *каждый* тираж;
- применяем индивидуальный подход в работе с каждым автором;
- работаем профессионально: от корректуры до авторских дизайн-проектов;
- проводим политику доступной цены;
- имеем собственные каналы сбыта: от федеральных сетей, крупнейших книжных магазинов РФ, ведущих маркетплейсов ОЗОН, Wildberries, Яндекс-Маркет и др. до ведущих библиотек вузов, ссузов.

Ждем Ваши предложения:

- **тел. (812) 412-70-26**
- **эл. почта: nitmail@nit.com.ru**

Будем рады сотрудничеству!

Для заказа книг:

➢ **интернет-магазин: nit.com.ru**
- более 3000 пунктов выдачи на территории РФ, доставка 3–5 дней
- более 300 пунктов выдачи в Санкт-Петербурге и Москве, доставка 1–2 дня
- тел. (812) 412-70-26
- эл. почта nitmail@nit.com.ru

➢ **магазин издательства:** г. Санкт-Петербург, пр. Обуховской обороны, д. 107
- метро Елизаровская, 200 м за ДК им. Крупской
- ежедневно с 10.00 до 18.00
- справки и заказ: тел. (812) 412-70-26

➢ **книжные сети и магазины**
- «Читай-город» - сеть магазинов тел. +7 (495) 424-84-44
- «Буквоед» - сеть магазинов тел. +7 (812) 601-0-601
- Московский дом книги – сеть магазинов тел. +7 (495) 789-35-91
- ТД «БиблиоГлобус» тел. +7 (495) 781-19-12
- «Амиталь» — сеть магазинов тел. +7 (473) 223-00-02
- Дом книги, г. Екатеринбург тел. +7 (343) 289-40-45
- Дом книги, г. Нижний Новгород тел. +7 (831) 246-22-92
- Приморский торговый Дом книги тел. +7 (423) 263-10-54

➢ **маркетплейсы ОЗОН, Wildberries, Яндекс-Маркет, Myshop и др.**

Аклимов Р. Р.

Базы данных на SQL, MySQL, MS SQL и Postgre/PL/pgSQL

Проектирование и практическая реализация

+ виртуальный диск с примерами

Группа подготовки издания:

Зав. редакцией компьютерной литературы: *Е.В. Финков*

Редактор: *О.С. Петрунич, Н.В. Жерлов*

Корректор: *А.В. Громова*

Изображение на обложке использовано с ресурсов freepik.com и vecteezy.com

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Издательство не несет ответственности за возможный ущерб, причиненный в ходе использования материалов данной книги, а также за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на Интернет-ресурсы были действующими. Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

12+

ООО "Издательство Наука и Техника"

ОГРН 1217800116247, ИНН 7811763020, КПП 781101001

192029, г. Санкт-Петербург, пр. Обуховской обороны, д. 107, лит. Б, пом. 1-Н

Подписано в печать 17.09.2025. Формат 70x100 1/16.

Бумага газетная. Печать офсетная. Объем 32 п.л.

Тираж 2500. Заказ 15449.

Отпечатано с готового оригинал-макета

ООО «Принт-М», 142300, М.О., г.Чехов, ул. Полиграфистов, д.1

Аклимов Рамиль Ринатович

Автор — опытный специалист с многолетней практикой в области проектирования, разработки и интеграции программного обеспечения, успешно создавший десятки крупных информационных систем и веб-приложений. Накопленные знания и практический опыт в проектировании и управлении базами данных, подкрепленные обширным опытом работы с различными СУБД, легли в основу этой книги.

У вас в руках — исчерпывающий гид по языку SQL, призванный помочь не просто освоить синтаксис, а глубоко понять принципы работы с данными и стать востребованным специалистом. Вы начнёте с абсолютного нуля, и каждый шаг — от установки СУБД до сложных запросов, — будет подкреплён практическими примерами (интернет-магазин, фитнес-клуб, банковская система, компьютерная игра, таксопарк, система регистрации в аэропорту, мониторинг состояния пациента в больнице и т.п.) и детальными объяснениями.

Понимая, что в мире не существует идеальной СУБД, книга разбирает каждую концепцию на примере трёх самых популярных систем: MySQL, PostgreSQL и Microsoft SQL Server. Такой подход позволит вам на практике увидеть их сильные и слабые стороны и избавиться от страха перед «незнакомой» средой, став по-настоящему гибким специалистом.

Особое внимание уделено «больным» темам, которые часто остаются за кадром в учебниках, но критически важны в работе: тонкости типов данных, кодировок и колляций, механизмы обеспечения целостности, транзакции, уровни изоляции и блокировки. Вы поймёте не только «как это работает», но и «почему это работает именно так».

Также в книге мы выйдем за рамки чистого SQL и рассмотрим архитектурные вопросы: стратегии «мягкого» удаления, плюсы и минусы переноса бизнес-логики в хранимые процедуры, а также принципы безопасности и управления доступом.

Отдельный фокус сделан на проектировании таблиц, индексах и анализе планов выполнения запросов. Вы научитесь писать не только рабочие, но и эффективные запросы. Именно это умение — ключевой навык в реальной работе и главное отличие любителя от профессионала.

издательство НАУКА и ТЕХНИКА Санкт-Петербург

Наши книги — ваши инвестиции в будущее

ISBN 978-5-907592-91-9
9 785907 592919 >

«Издательство Наука и Техника» г. Санкт-Петербург
Для заказа книг: т. (812) 412-70-26
E-mail: nitmail@nit.com.ru
Сайт: nit.com.ru

НиТ издательство

nit.com.ru