Основные сущности

- **DATABASE:** Контейнер верхнего уровня для объектов (таблицы, индексы, представления и т.д.). Изолирует данные.
- **TABLE:** Базовая структура хранения. Состоит из столбцов (атрибуты с типами) и строк (записи).

DDL (Data Definition Language): Работа со структурой

- **CREATE:**
  - `CREATE DATABASE db_name;`
  - `CREATE TABLE tbl_name (col1 type1, col2 type2, ...);` (с опциональными ограничениями: PK, FK, NOT NULL и т.д.)
- **ALTER (Изменение существующей структуры):**
  - `ALTER TABLE tbl_name ADD col_name type;` (Добавить столбец)
  - `ALTER TABLE tbl_name DROP COLUMN col_name;` (Удалить столбец)
  - `ALTER TABLE tbl_name ALTER COLUMN col_name new_type;` (Изменить тип столбца - синтаксис СУБД-зависим)
  - `ALTER TABLE tbl_name ADD CONSTRAINT ...;` (Добавить PK, FK, UNIQUE и т.д.)
- **DROP:**
  - `DROP TABLE tbl_name;` (Удалить таблицу и данные)
  - `DROP DATABASE db_name;` (Удалить БД и все содержимое)

DML (Data Manipulation Language): Работа с данными (строками)

- **SELECT:** Извлечение данных.
  - `SELECT col1, col2 FROM tbl_name WHERE condition ORDER BY col;`
  - Основные части: `SELECT` (столбцы/выражения), `FROM` (таблица(ы)), `WHERE` (фильтрация), `ORDER BY` (сортировка).
- **INSERT:** Добавление новых строк.
  - `INSERT INTO tbl_name (col1, col2) VALUES (val1, val2);` (Явные столбцы)

    - `INSERT INTO tbl_name VALUES (val1, val2, ...);` (Все столбцы, порядок важен)
    - `INSERT INTO tbl_name SELECT ... FROM ...;` (Вставка результата запроса)
- **UPDATE:** Модификация существующих строк.
    - `UPDATE tbl_name SET col1 = val1, col2 = val2 WHERE condition;`
    - `WHERE` критичен для выбора изменяемых строк.
- **DELETE:** Удаление строк.
    - `DELETE FROM tbl_name WHERE condition;`
    - `WHERE` критичен для выбора удаляемых строк. Удаляет данные, структура таблицы сохраняется.

Резюме:

- **Сущности:** `DATABASE` (контейнер), `TABLE` (данные: столбцы + строки).
- **DDL (Структура):** `CREATE` (новый объект), `ALTER` (модифицировать объект), `DROP` (удалить объект).
- **DML (Данные):** `SELECT` (читать), `INSERT` (добавить строки), `UPDATE` (изменить строки), `DELETE` (удалить строки).

Специфичные операторы

1. DISTINCT

- **Назначение:** Устранение дубликатов строк в результирующем наборе `SELECT`.
- **Применение:** В предложении `SELECT`.
- **Синтаксис:** `SELECT DISTINCT column1, column2 FROM table_name;`
- **Особенности:** Работает на уровне всей строки результата. Часто используется внутри агрегатных функций (`COUNT(DISTINCT column)`).

2. UNIQUE

- **Назначение (Ограничение):** Гарантирует уникальность значений в столбце или группе столбцов таблицы.
- **Применение:** В `CREATE TABLE` или `ALTER TABLE`.

- **Синтаксис:**

```
CREATE TABLE table_name (
    column1 type,
    column2 type,
    UNIQUE (column1) -- или UNIQUE (column1, column2)
);
```

- **Отличие от PRIMARY KEY:** Допускает множество значений `NULL` (если тип столбца это позволяет). Может быть несколько на таблицу.
- **Примечание:** `UNIQUE` в контексте `SELECT` является устаревшим синонимом `DISTINCT` (не рекомендуется).

3. PRIMARY KEY (Первичный ключ)

- **Назначение:** Уникальная идентификация каждой строки в таблице. Главное ограничение целостности.
- **Применение:** В `CREATE TABLE` или `ALTER TABLE`.
- **Синтаксис:**

```
CREATE TABLE table_name (
    id INT PRIMARY KEY, -- Вариант 1: для одного столбца
    column2 type,
    ...
    -- Вариант 2: для составного ключа
    PRIMARY KEY (col1, col2)
);
```

- **Характеристики:**
    - Значения **уникальны** и **не могут быть NULL**.
    - В таблице может быть только **один** PRIMARY KEY.
    - Автоматически создает **уникальный индекс** (обычно кластеризованный).
    - База для связей `FOREIGN KEY`.

4. FOREIGN KEY (Внешний ключ)

- **Назначение:** Обеспечение ссылочной целостности. Связывает столбец(цы) в одной таблице (`дочерней`) с PRIMARY KEY или UNIQUE в другой таблице (`родительской`).
- **Применение:** В `CREATE TABLE` или `ALTER TABLE`.
- **Синтаксис:**

```
CREATE TABLE child_table (
    id INT PRIMARY KEY,
    parent_id INT,
    FOREIGN KEY (parent_id) REFERENCES parent_table(parent_id)
    [ON DELETE {NO ACTION | CASCADE | SET NULL | SET DEFAULT}]
    [ON UPDATE {NO ACTION | CASCADE | SET NULL | SET DEFAULT}]
);
```

- **Ключевые аспекты:**
  - Значения в столбце `FOREIGN KEY` должны существовать в связанном столбце родительской таблицы (`REFERENCES`).
  - `ON DELETE/UPDATE`: Определяет поведение при удалении/изменении родительской строки (`CASCADE`, `SET NULL`, `SET DEFAULT`, `RESTRICT/NO ACTION`).

5. DEFAULT

- **Назначение:** Задание значения по умолчанию для столбца при вставке новой строки, если значение не указано явно.
- **Применение:** В `CREATE TABLE` или `ALTER TABLE`.
- **Синтаксис:**

```
CREATE TABLE table_name (
    column1 type DEFAULT default_value, -- Литерал (число, строка)
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP -- Системная функция
);
```

- **Типы значений:** Литералы (`10`, `'Pending'`, `TRUE`), Системные функции (`CURRENT_DATE`, `NOW()`), `NULL` (если столбец допускает `NULL`).

6. CHECK

- **Назначение:** Ограничение домена столбца. Проверяет, что значение в столбце удовлетворяет заданному условию (булевому выражению) перед вставкой или обновлением.
- **Применение:** В `CREATE TABLE` или `ALTER TABLE` (на уровне столбца или таблицы).
- **Синтаксис:**

```
CREATE TABLE table_name (
    age INT CHECK (age >= 18), -- Ограничение уровня столбца
    price DECIMAL(10,2),
    discount DECIMAL(10,2),
    CHECK (price > discount) -- Ограничение уровня таблицы);
```

- **Выражение:** Может использовать другие столбцы таблицы, операторы сравнения, логические операторы, функции.

Резюме для быстрого повторения:

- `DISTINCT` – Уникальные строки в `SELECT`.
- `UNIQUE` – Ограничение: уникальные значения (допускает NULL).
- `PRIMARY KEY` – Ограничение: уникальный идентификатор строки (NOT NULL, один на таблицу).
- `FOREIGN KEY` – Ограничение: ссылка на PK/UNIQUE другой таблицы + правила (`ON DELETE/UPDATE`).
- `DEFAULT` – Значение столбца при `INSERT`, если не указано.
- `CHECK` – Ограничение: условие для значения(й) столбца(ов).

1. CASE: Условная логика

Назначение: Аналог `if-else` в SQL для преобразования значений.

Синтаксис:

```
SELECT
  name,
  CASE
    WHEN score >= 90 THEN 'A'
    WHEN score >= 80 THEN 'B'
    ELSE 'C'
  END AS grade
FROM students;
```

Варианты:

- Простой CASE:

  ```
  CASE department_id
    WHEN 1 THEN 'HR'
    WHEN 2 THEN 'IT'
  END
  ```

---

2. LIKE: Поиск по шаблону

Назначение: Фильтрация строк с использованием масок.

Синтаксис:

```
SELECT * FROM users
WHERE name LIKE 'J%';  -- Начинается на J
```

Спецсимволы:

- `%` - любое количество символов
- `_` - один символ
- `ESCAPE` для экранирования:
  ```
  WHERE comment LIKE '%30!%%' ESCAPE '!';  -- Ищет "30%"
  ```

Регистрозависимость:

- Зависит от СУБД: `ILIKE` в PostgreSQL для регистронезависимого поиска

---

3. WHERE EXISTS: Проверка существования

Назначение: Фильтрация по результату подзапроса.

Синтаксис:

```
SELECT * FROM departments d
WHERE EXISTS (
  SELECT 1 FROM employees e
  WHERE e.department_id = d.id
  AND e.salary > 100000
);
```

Особенности:

- Возвращает `TRUE`, если подзапрос вернул ≥1 строки
- Оптимальнее `IN` для больших таблиц
- Коррелированные подзапросы - ссылаются на внешнюю таблицу

---

4. UNION: Объединение результатов

Назначение: Комбинирование результатов нескольких запросов.

Синтаксис:

```
SELECT city FROM suppliers
UNION [ALL]
SELECT city FROM customers;
```

Типы:

Оператор	Дубликаты	Сортировка	Производительность
UNION	Удаляет	Да	Медленнее
UNION ALL	Сохраняет	Нет	Быстрее

Правила:

1. Количество столбцов должно совпадать
2. Типы данных должны быть совместимы
3. Имена столбцов берутся из первого запроса

Пример с сортировкой:

```
(SELECT name, salary FROM employees ORDER BY salary DESC LIMIT 3)
UNION ALL
(SELECT name, salary FROM managers ORDER BY salary DESC LIMIT 2)
ORDER BY salary DESC;
```

Процедурные конструкции и функции

1. Управляющие конструкции

IF-ELSE: Условное выполнение

```
-- T-SQL (SQL Server)
IF EXISTS(SELECT * FROM users WHERE age < 18)
BEGIN
    PRINT 'Found minors';
    -- Действия при истинном условии
END
ELSE IF (SELECT COUNT(*) FROM users) = 0
BEGIN
    PRINT 'No users';
END
ELSE
```

```
BEGIN
    PRINT 'All users are adults';
END
```

WHILE: Циклы

```
-- PostgreSQL
DECLARE counter INT := 0;
WHILE counter < 10 LOOP
    INSERT INTO logs (message) VALUES ('Iteration: ' || counter);
    counter := counter + 1;
END LOOP;
```

---

2. Обработка ошибок

TRY-CATCH-THROW (T-SQL):

```
BEGIN TRY
    BEGIN TRANSACTION;
    UPDATE accounts SET balance = balance - 100 WHERE id = 1;
    UPDATE accounts SET balance = balance + 100 WHERE id = 2;
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    THROW;  -- Проброс ошибки
    -- Или кастомная ошибка:
    -- THROW 51000, 'Transfer failed', 1;
END CATCH
```

PostgreSQL (EXCEPTION):

```
BEGIN
    UPDATE accounts SET balance = balance - 100 WHERE id = 1;
    -- ...
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Caught division by zero';
    WHEN OTHERS THEN
        ROLLBACK;
        RAISE EXCEPTION 'Unexpected error: %', SQLERRM;
END;
```

---

3. Функции

Отличия от процедур:

- Возвращают значение
- Могут использоваться в SELECT
- Часто без побочных эффектов

Типы функций:

Тип	Описание	Пример использования
Скалярные	Возвращают одно значение	`SELECT dbo.CalculateTax(price)`
Табличные	Возвращают таблицу	`SELECT * FROM GetUsersByRole('admin')`
Агрегатные	Пользовательские агрегаты	`SELECT AVG_CUSTOM(score) FROM tests`

Создание скалярной функции (PostgreSQL):

```
CREATE OR REPLACE FUNCTION GetTax(price NUMERIC)
RETURNS NUMERIC AS $$
BEGIN
    RETURN price * 0.20; -- 20% tax
END;
$$ LANGUAGE plpgsql;
```

Табличная функция (SQL Server):

```
CREATE FUNCTION GetEmployeeProjects(@emp_id INT)
RETURNS TABLE AS
RETURN (
    SELECT p.name, p.deadline
    FROM projects p
    JOIN assignments a ON p.id = a.project_id
    WHERE a.employee_id = @emp_id
);
```

Резюме для экзамена:

```
-- IF-ELSE
IF condition THEN ... ELSE ... END IF;

-- WHILE
WHILE condition LOOP ... END LOOP;
```

```
-- TRY-CATCH
BEGIN TRY ... END TRY BEGIN CATCH ... END CATCH

-- Функция
CREATE FUNCTION name(...) RETURNS type AS $$ ... $$;
```

Группировка данных

1. **GROUP BY**

- **Назначение:** Группировка строк с одинаковыми значениями в указанных столбцах для применения агрегатных функций.
- **Синтаксис:**
  ```
  SELECT column1, aggregate_function(column2)
  FROM table
  GROUP BY column1;
  ```
- **Ключевое:**
  - Все неагрегированные столбцы в `SELECT` должны быть в `GROUP BY`
  - Работает с агрегатными функциями: `COUNT()`, `SUM()`, `AVG()`, `MIN()`, `MAX()`

2. **HAVING**

- **Назначение:** Фильтрация результатов группировки (аналог `WHERE` для групп).
- **Синтаксис:**

  ```
  SELECT column1, COUNT(*)
  FROM table
  GROUP BY column1
  HAVING COUNT(*) > 5;
  ```
- **Отличие от WHERE:**
  - `WHERE` фильтрует строки **до** группировки
  - `HAVING` фильтрует группы **после** группировки
  - Может использовать агрегатные функции в условиях

Операторы диапазона

3. **BETWEEN**

- **Назначение:** Проверка вхождения значения в диапазон (включительно).
- **Синтаксис:**

```
SELECT * FROM table
WHERE column BETWEEN value1 AND value2;
```

- **Эквивалент:** `column >= value1 AND column <= value2`
- **Работает с:** Числами, датами, строками (лексикографическое сравнение)

Важное дополнение (отсутствовало в запросе):

Агрегатные функции (необходимы для GROUP BY):

- `COUNT()` - количество строк
- `SUM()` - сумма значений
- `AVG()` - среднее значение
- `MAX()`/`MIN()` - максимальное/минимальное значение

Пример полного использования:

```
SELECT department_id,
       AVG(salary) AS avg_salary,
       COUNT(*) AS employees
FROM employees
WHERE hire_date BETWEEN '2020-01-01' AND '2023-12-31'
GROUP BY department_id
HAVING AVG(salary) > 70000;
```

Резюме:

- `GROUP BY` - группировка строк для агрегации
- `HAVING` - фильтрация результатов группировки
- `BETWEEN` - проверка вхождения в диапазон (включительно)
- Агрегатные функции (`COUNT`, `SUM`, `AVG` и др.) - основа для работы с группировками

JOIN-операции

Назначение: Комбинирование строк из двух и более таблиц на основе логической связи между ними.

Основные типы JOIN:

1. **INNER JOIN**
    - Возвращает только совпадающие строки из обеих таблиц
    - Синтаксис:
      ```
      SELECT *
      FROM TableA a
      INNER JOIN TableB b ON a.key = b.key;
      ```
    - Эквивалент: `JOIN` (без указания типа)
2. **LEFT JOIN (OUTER)**
    - Все строки из левой таблицы + совпадения из правой
    - При отсутствии совпадения: `NULL` в колонках правой таблицы
    - Синтаксис:
      ```
      SELECT *
      FROM TableA a
      LEFT JOIN TableB b ON a.key = b.key;
      ```
3. **RIGHT JOIN (OUTER)**
    - Все строки из правой таблицы + совпадения из левой
    - Зеркальный аналог LEFT JOIN
    - На практике используется реже
4. **FULL JOIN (OUTER)**
    - Все строки из обеих таблиц
    - `NULL` в отсутствующих совпадениях
    - Синтаксис:
      ```
      SELECT *
      FROM TableA a
      FULL JOIN TableB b ON a.key = b.key;
      ```

5. **CROSS JOIN**
    - Декартово произведение (все комбинации строк)
    - Синтаксис:
      ```
      SELECT *
      FROM TableA
      CROSS JOIN TableB;
      ```

Специальные случаи:

- **SELF JOIN:**

  Соединение таблицы с самой собой (требует алиасов)

  ```
  SELECT e.name, m.name AS manager
  FROM employees e
  LEFT JOIN employees m ON e.manager_id = m.id;
  ```

- **NATURAL JOIN:**

  Автоматическое соединение по колонкам с одинаковыми именами (не рекомендуется – риск ошибок)

  ```
  SELECT * FROM TableA NATURAL JOIN TableB;
  ```

Критические аспекты:

1. **Условия соединения:**
   - Всегда явно указывать `ON` (избегать неявных соединений через `WHERE`)
   - Для сложных связей: `ON a.key1 = b.key1 AND a.key2 = b.key2`
2. **Производительность:**
   - Индексы на join-колонках обязательны
   - Порядок таблиц влияет на план выполнения (особенно для OUTER JOIN)
   - Фильтрацию в `WHERE` применять до JOIN где возможно
3. **Обработка NULL:**
   - `WHERE b.key IS NULL` в LEFT JOIN для поиска отсутствующих связей
   - `COALESCE()` для подстановки значений вместо NULL

Валидные альтернативы (но устаревшие):

```
-- Implicit JOIN (не рекомендуется)
SELECT *
FROM TableA, TableB WHERE a.key = b.key;
```

Резюме:

- `INNER JOIN` – базовое соединение по совпадениям
- `LEFT JOIN` – основной инструмент для анализа связей
- `FULL JOIN` – редкий сценайрий полного объединения
- Ключевое: всегда использовать явный `ON`, контролировать `NULL`, индексировать ключи

Constraint`ы (ограничения)

Добавление ограничений через `ADD CONSTRAINT`

Назначение: Явное создание именованных ограничений на существующую таблицу через `ALTER TABLE`.
Ключевое преимущество: Контроль имен ограничений для удобства управления (удаление, отключение).

Синтаксис:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name constraint_type (column_list | condition);
```

Основные типы ограничений:

1. **Первичный ключ (PRIMARY KEY)**

   ```
   ALTER TABLE employees

   ADD CONSTRAINT pk_emp_id PRIMARY KEY (emp_id);
   ```

   - Требует уникальности + `NOT NULL` во всех колонках
   - Автоматически создает кластеризованный индекс
2. **Внешний ключ (FOREIGN KEY)**

   ```
   ALTER TABLE orders
   ADD CONSTRAINT fk_orders_customers
   FOREIGN KEY (cust_id) REFERENCES customers(cust_id)
   ON DELETE CASCADE;
   ```

   - Обязательная часть: `REFERENCES parent_table(column)`
   - Опционально: `ON DELETE`/`ON UPDATE` действия (`CASCADE`, `SET NULL`, `NO ACTION`)
3. **Уникальность (UNIQUE)**

   ```
   ALTER TABLE users
   ADD CONSTRAINT uq_user_email UNIQUE (email);
   ```

- Допускает множественные `NULL` (в большинстве СУБД)
- Создает некластеризованный индекс

4. **Проверочное условие (CHECK)**

```
ALTER TABLE products
ADD CONSTRAINT chk_price_positive CHECK (price > 0);
```

   - Может включать несколько колонок:

```
ADD CONSTRAINT chk_valid_discount CHECK (discount < price);
```

5. **Ограничение DEFAULT (через изменение столбца)**

```
ALTER TABLE orders
ADD CONSTRAINT df_order_date
DEFAULT GETDATE() FOR order_date;
```

   - **Важно:** Синтаксис отличается от других ограничений!

Критические особенности:

1. **Проверка существующих данных:**

   При добавлении ограничения СУБД **проверит все текущие данные** в таблице. При нарушении правил операция завершится ошибкой.

```
-- Пример ошибки:
-- The ALTER TABLE statement conflicted with the CHECK constraint "chk_
price_positive"
```

2. **Управление ограничениями:**

   - Удаление:

```
ALTER TABLE products
DROP CONSTRAINT chk_price_positive;
```

   - Временное отключение (не все СУБД):

```
ALTER TABLE orders NOCHECK CONSTRAINT fk_orders_customers;
```

3. **Составные ограничения:**

```
ALTER TABLE shipments
ADD CONSTRAINT pk_shipments
PRIMARY KEY (order_id, product_id);
```

4. **Именование (best practices):**

   - `pk_<table>_<column>` – первичный ключ
   - `fk_<child_table>_<parent_table>` – внешний ключ
   - `chk_<table>_<condition>` – проверочное ограничение

- `uq_<table>_<column>` – уникальность

Ошибки при добавлении:

1. **Дубликаты значений** при создании `PRIMARY KEY`/`UNIQUE`
2. **Неподдерживаемые типы данных** (например, `TEXT` для PK)
3. **Отсутствие индекса** в родительской таблице для `FOREIGN KEY`
4. **Нарушение ссылочной целостности** для `FOREIGN KEY`

Резюме для экзамена:

```
-- Общий шаблон
ALTER TABLE <table>
ADD CONSTRAINT <name> <type> <details>;

-- Особые случаи:
PRIMARY KEY (col1, col2)        -- Составной ключ
FOREIGN KEY (col) REFERENCES ... ON DELETE CASCADE  -- Каскадное удаление
CHECK (col > 0)                 -- Условие валидации
DEFAULT <value> FOR <column>    -- Синтаксис для DEFAULT
```

Процедуры

Хранимые процедуры (Stored Procedures)

Назначение: Предкомпилированные блоки SQL-кода, хранящиеся в БД для выполнения сложных операций. Оптимизируют производительность, безопасность и управление бизнес-логикой.

Ключевые характеристики:

1. **Инкапсуляция логики:** Объединение DML, транзакций, условий в одном объекте.

2. **Повторное использование:** Вызов из приложений/триггеров/других процедур.
3. **Безопасность:** Выполнение с правами владельца (`SECURITY DEFINER`).
4. **Производительность:** Кэширование плана выполнения.

Синтаксис (общий вид):

```
CREATE [OR REPLACE] PROCEDURE proc_name (
    [IN | OUT | INOUT] param1 data_type [,
    ...]
)
LANGUAGE SQL | plpgsql | ...
AS $$
DECLARE
    -- Локальные переменные
BEGIN
    -- Тело процедуры (SQL + логика)
    [RETURN | RETURNING]
END;
$$;
```

Основные компоненты:

1. **Параметры:**
    - `IN` (входные, по умолчанию)
    - `OUT` (выходные)
    - `INOUT` (входо-выходные)

    ```
    CREATE PROCEDURE add_user(
      IN name VARCHAR(50),
      OUT new_id INT
    ) ...
    ```
2. **Язык выполнения:**
    - Стандартный SQL (простой)
    - Процедурные расширения: `PL/pgSQL` (PostgreSQL), `PL/SQL` (Oracle), `T-SQL` (SQL Server)

3. **Логика управления:**
    - Условия: `IF ... THEN ... ELSEIF ... END IF;`
    - Циклы: `FOR`, `WHILE`, `LOOP`
    - Обработка ошибок: `EXCEPTION WHEN ... THEN`

Пример (PostgreSQL):

```
CREATE OR REPLACE PROCEDURE transfer_funds(
    IN from_account INT,
    IN to_account INT,
    IN amount DECIMAL(10,2)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Начало транзакции
    UPDATE accounts SET balance = balance - amount WHERE id = from_account;
    UPDATE accounts SET balance = balance + amount WHERE id = to_account;

    -- Фиксация при успехе
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        RAISE;
END;
$$;
```

Управление процедурами:

1. **Вызов:**
   ```
   CALL proc_name(arg1, arg2);
   -- Для OUT-параметров:
   CALL proc_name('input', result);
   ```
2. **Просмотр:**
   ```
   SELECT * FROM information_schema.routines
   WHERE routine_type = 'PROCEDURE';
   ```
3. **Удаление:**
   ```
   DROP PROCEDURE proc_name [CASCADE];
   ```

Критические аспекты:

1. **Транзакции:**
    - Явное управление (`COMMIT`/`ROLLBACK`) внутри процедуры
    - Автокоммит по умолчанию в некоторых СУБД
2. **Производительность:**
    - Прекомпиляция → быстрый повторный запуск
    - Риск "протекания абстракции" при сложной логике
3. **Отладка:**
    - Сложность трассировки ошибок
    - Инструменты: `RAISE NOTICE` (логирование), `GET DIAGNOSTICS`
4. **Версионирование:**
    - `OR REPLACE` для обновления
    - Миграции через системы контроля версий (Liquibase, Flyway)

Отличие от функций:

Процедуры	Функции
Нет возвращаемого значения	Обязательно `RETURNS`
`CALL` для выполнения	Вызов в `SELECT`
Разрешены DDL/DCL	Обычно только DML
Могут менять состояние БД	Часто детерминированы

Резюме для экзамена:

```
-- Создание
CREATE PROCEDURE name(IN param type)
AS $$ ... $$;
```

```
-- Вызов
CALL name(args);

-- Удаление
DROP PROCEDURE name;
```

Индексы

Индексы (Indexes)

Назначение: Ускоряют поиск данных за счет предварительно отсортированных структур (B-деревья, хеши и др.). Торговля между скоростью чтения и стоимостью записи.

Ключевые типы индексов:

1. **B-Tree (Стандартный)**
   - Оптимален для диапазонов (`BETWEEN`, `>`, `<`) и сортировки
   - Поддерживает `=`, `>`, `<`, `LIKE 'prefix%'`

   ```
   CREATE INDEX idx_users_email ON users(email);
   ```

2. **Hash**
   - Только точное совпадение (`=`)
   - Быстрее B-Tree для `WHERE key = value`
   - Не поддерживает сортировку/диапазоны

   ```
   CREATE INDEX idx_orders_id_hash ON orders USING HASH (id);
   ```

3. **Составной (Composite)**
   - Индекс по нескольким столбцам
   - Порядок столбцов критичен!

   ```
   CREATE INDEX idx_orders_status_date ON orders(status, created_date);
   -- Эффективен для:
   --   WHERE status = 'shipped'
   --   WHERE status = 'shipped' AND created_date > '2023-01-01'
   ```

4. **Частичный (Partial)**
    - Индексирует подмножество строк (WHERE-фильтр)

```
CREATE INDEX idx_active_users ON users(email) WHERE is_active = true;
```

5. **UNIQUE**
    - Обеспечивает уникальность + ускорение поиска

```
CREATE UNIQUE INDEX uq_employee_passport ON employees(passport_number);
```

Критерии применения:

- **Добавлять:**
    - Частые `WHERE`/`JOIN` по колонке
    - Критичные по скорости запросы
    - Высокая кардинальность данных
- **Не добавлять:**
    - Частые массовые вставки (`INSERT ... SELECT`)
    - Таблицы < 10k строк (часто full scan быстрее)
    - Колонки с NULL > 90%

Эксплуатация:

1. **Анализ запросов:**

```
EXPLAIN ANALYZE SELECT * FROM orders WHERE status = 'processing';
```

2. **Перестроение:**

```
REINDEX INDEX idx_orders_status;  -- PostgreSQL
ALTER INDEX idx_orders_status REBUILD;  -- SQL Server
```

3. **Мониторинг:**
    - `pg_stat_user_indexes` (PostgreSQL)
    - `sys.dm_db_index_usage_stats` (SQL Server)

Специальные индексы (СУБД-специфичные):

Тип	СУБД	Использование
GIN	PostgreSQL	JSONB, полнотекстовый поиск
BRIN	PostgreSQL	Большие таблицы с сортировкой
Columnstore	SQL Server	OLAP-аналитика
Spatial	Все	Геоданные (PostGIS, etc.)
Full-Text	Все	`CONTAINS()`, `@@`

Антипаттерны:

1. **Индекс-подушка:**
   ```
   -- Плохо:
   CREATE INDEX idx_everything ON table(a, b, c, d, e);
   ```
2. **Индексы на часто изменяемые данные:**
   Высокие накладные расходы на `UPDATE`/`DELETE`.
3. **Дублирование:**
   `INDEX(a,b)` и `INDEX(a)` → второй индекс избыточен.

Резюме:

```
-- Создать индекс
CREATE [UNIQUE] INDEX index_name ON table (column [ASC/DESC], ...)
[WHERE condition];  -- Для частичных индексов

-- Удалить индекс
DROP INDEX index_name;
```

Принципы:

- **B-Tree** — универсальный выбор для большинства сценариев
- **Составные индексы** — порядок столбцов = порядку фильтрации
- **Мониторинг** — удаляйте неиспользуемые индексы

Представления

Назначение: Виртуальные таблицы, определяемые SQL-запросом. Не хранят данные, а предоставляют "окно" к результату запроса.

Ключевые характеристики:

1. **Абстракция данных:**
    - Скрывают сложность запросов (джойны, агрегации)
    - Предоставляют упрощённый интерфейс доступа
2. **Безопасность:**
    - Ограничивают доступ к столбцам/строкам исходных таблиц

```
CREATE VIEW hr_employee_view AS
SELECT id, name, department FROM employees;
```

3. **Логический слой:**
    - Независимость от изменений схемы (если структура VIEW сохраняется)

Синтаксис:

Создание:

```
CREATE [OR REPLACE] VIEW view_name [(column_list)]
AS
SELECT ...
[WITH CHECK OPTION];  -- Для updatable views
```

Удаление:

```
DROP VIEW view_name [CASCADE];
```

Типы представлений:

1. **Простые (Updatable):**
   - На основе одной таблицы
   - Поддерживают `INSERT`/`UPDATE`/`DELETE` (с ограничениями)

```
CREATE VIEW active_users AS
SELECT * FROM users WHERE is_active = true
WITH CHECK OPTION;  -- Запрещает INSERT строк, не удовлетворяющих WHERE
```

2. **Сложные (Read-only):**
   - Содержат джойны, группировки, агрегатные функции
   - Только для чтения (`SELECT`)

```
CREATE VIEW order_summary AS
SELECT o.id, c.name, SUM(oi.price) AS total
FROM orders o
JOIN customers c ON o.customer_id = c.id
JOIN order_items oi ON o.id = oi.order_id
GROUP BY o.id, c.name;
```

3. **Материализованные (Materialized Views):**
   - **Физически хранят результат запроса** (как snapshot)
   - Требуют ручного/автообновления (`REFRESH`)
   - Используются для тяжёлых отчётов (OLAP)

```
-- PostgreSQL:
CREATE MATERIALIZED VIEW mv_sales_report AS
SELECT region, SUM(sales) FROM orders GROUP BY region;

REFRESH MATERIALIZED VIEW mv_sales_report;
```

Преимущества:

- **Упрощение запросов:**

```
-- Вместо:
SELECT * FROM (сложный 50-строчный запрос)
-- Используем:
SELECT * FROM report_view;
```

- **Контроль доступа:**

```
GRANT SELECT ON finance_view TO analyst_role;
```

- **Совместимость:** Прозрачная замена таблиц в legacy-коде.

Ограничения:

1. **Производительность:**
    - Нет собственной оптимизации (выполняется базовый запрос)
    - Риск снижения скорости при вложенных VIEW
2. **Обновление данных:**
    - `UPDATE` разрешён только для простых VIEW (без DISTINCT, GROUP BY, агрегатов)
3. **Зависимости:**
    - Изменение структуры исходных таблиц может сломать VIEW
    - `CREATE OR REPLACE VIEW` для обновления

Best Practices:

1. **Именование:**
    - Суффикс `_v` для обычных VIEW (`users_v`)
    - Суффикс `_mv` для материализованных (`sales_mv`)
2. **Документация:**
    - Комментарии к VIEW и столбцам:

```
COMMENT ON VIEW order_summary IS 'Агрегированные данные по заказам';
```

3. **Оптимизация:**
    - Избегать многоуровневых вложенных VIEW
    - Использовать `WITH CHECK OPTION` для защиты целостности

Резюме:

```
-- Создание
CREATE VIEW view_name AS SELECT ...;

-- Материализованное (кэшированное)
CREATE MATERIALIZED VIEW mv_name AS SELECT ...;

-- Обновление
REFRESH MATERIALIZED VIEW mv_name;  -- Для материализованных
CREATE OR REPLACE VIEW view_name AS ...;  -- Для обычных
```

Когда использовать:

- Сложные часто используемые запросы
- Ограничение доступа к чувствительным данным
- Кэширование ресурсоёмких отчётов (materialized)

Транзакции

Назначение: Группа операций, выполняемая как атомарная единица работы. Гарантирует целостность данных при сбоях и параллельном доступе.

Ключевые свойства (ACID):

1. **Atomicity (Атомарность)**
    - Все операции выполняются либо целиком, либо ни одна
    - Инструменты: `COMMIT` (фиксация), `ROLLBACK` (откат)
2. **Consistency (Согласованность)**
    - Транзакция переводит БД из одного валидного состояния в другое
    - Обеспечивается ограничениями (PK, FK, CHECK)
3. **Isolation (Изолированность)**
    - Параллельные транзакции не влияют друг на друга
    - Регулируется уровнями изоляции
4. **Durability (Долговечность)**
    - Результаты зафиксированной транзакции устойчивы к сбоям

Управление транзакциями:

```
START TRANSACTION;  -- или BEGIN (PostgreSQL), BEGIN TRANSACTION (SQL Server)

-- Операции:
INSERT INTO orders ...;
UPDATE inventory ...;
-- Логика приложения

COMMIT;  -- Фиксация изменений

-- При ошибке:
ROLLBACK;  -- Отмена всех изменений в транзакции
```

Уровни изоляции (ANSI):

Уровень	Грязное чтение	Неповторяемое чтение	Фантомы	Производительность
READ UNCOMMITTED	✓	✓	✓	Высокая
READ COMMITTED	✗	✓	✓	Средняя
REPEATABLE READ	✗	✗	✓	Низкая
SERIALIZABLE	✗	✗	✗	Очень низкая

Установка уровня:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

Практические паттерны:

1. **Точки сохранения (Savepoints):**

   ```
   SAVEPOINT sp1;
   -- Операции...
   ROLLBACK TO sp1;  -- Частичный откат
   ```

2. **Автокоммит:**
```
SET autocommit = 0;  -- Отключение для сессии (MySQL)
```
3. **Обработка ошибок:**
```
BEGIN
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    ROLLBACK;
    RESIGNAL;
  END;

  START TRANSACTION;
  -- Операции...
  COMMIT;
END
```

Критические аспекты:

1. **Блокировки:**
   - Эксклюзивные (X), разделяемые (S), намеренные (IX, IS)
   - Риск взаимоблокировок (deadlocks):
     ```
     SHOW ENGINE INNODB STATUS;  -- Анализ deadlock (MySQL)
     ```
2. **Длинные транзакции:**
   - Блокируют ресурсы → снижают параллелизм
   - Решение: разбиение на мелкие транзакции
3. **Nested Transactions:**
   - Не поддерживаются в большинстве СУБД
   - Эмуляция через savepoints

СУБД-специфика:

- **PostgreSQL:**
```
BEGIN ISOLATION LEVEL REPEATABLE READ;
-- ...
COMMIT;
```

- **SQL Server:**
  ```
  SET IMPLICIT_TRANSACTIONS ON;  -- Неявные транзакции
  ```
- **Oracle:**
  ```
  SET TRANSACTION READ WRITE;  -- Явное управление
  ```

Резюме для экзамена:

```
-- Стандартный поток
START TRANSACTION;
[Операции DML]
COMMIT / ROLLBACK;

-- Управление изоляцией
SET TRANSACTION ISOLATION LEVEL ...;

-- Точки сохранения
SAVEPOINT name;
ROLLBACK TO name;
```

Правила применения:

1. Минимизируйте время транзакции
2. Избегайте DDL внутри транзакций
3. Используйте `READ COMMITTED` как баланс целостности/производительности
4. Всегда обрабатывайте ошибки с `ROLLBACK`

Готов к следующим темам!

Триггеры

Триггеры (Triggers)

Назначение: Автоматически выполняемый код в ответ на события DML (INSERT/UPDATE/DELETE) или DDL. Работают в контексте транзакции, где произошло событие.

Ключевые характеристики:

1. **Реагируют на события:**
   - `BEFORE` или `AFTER` операций
   - `INSTEAD OF` для замены операции (часто для представлений)
2. **Доступ к данным:**
   - `OLD` - значения до изменения (для UPDATE/DELETE)
   - `NEW` - новые значения (для INSERT/UPDATE)

Типы триггеров:

1. **DML Triggers**
   - Срабатывают при изменении данных

```
-- Пример (PostgreSQL)
CREATE TRIGGER log_salary_change
BEFORE UPDATE ON employees
FOR EACH ROW
EXECUTE FUNCTION log_salary_update();
```

2. **DDL Triggers**
   - Реагируют на изменения схемы (`CREATE`, `ALTER`, `DROP`)

```
-- Пример (SQL Server)
CREATE TRIGGER prevent_drop_table
ON DATABASE
FOR DROP_TABLE
AS ROLLBACK;
```

3. **INSTEAD OF Triggers**
   - Заменяют оригинальную операцию

```
CREATE TRIGGER update_complex_view
INSTEAD OF UPDATE ON sales_summary
FOR EACH ROW
EXECUTE FUNCTION redistribute_sales();
```

Структура создания (DML):

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF} {INSERT | UPDATE | DELETE}
ON table_name
[FOR EACH {ROW | STATEMENT}]
[WHEN (condition)]
EXECUTE {FUNCTION | PROCEDURE} function_name();
```

Основные сценарии использования:

1. **Аудит изменений:**

   ```
   CREATE FUNCTION audit_employee() RETURNS TRIGGER AS $$
   BEGIN
     INSERT INTO audit_log (action, table_name, record_id)
     VALUES (TG_OP, 'employees', NEW.id);
     RETURN NEW;
   END;
   $$ LANGUAGE plpgsql;
   ```

2. **Проверка данных:**

   ```
   CREATE FUNCTION check_salary() RETURNS TRIGGER AS $$
   BEGIN
     IF NEW.salary < 0 THEN
        RAISE EXCEPTION 'Salary cannot be negative';
     END IF;
     RETURN NEW;
   END;
   $$ LANGUAGE plpgsql;
   ```

3. **Поддержание денормализованных данных:**

   ```
   CREATE FUNCTION update_order_total() RETURNS TRIGGER AS $$
   BEGIN
     UPDATE orders
     SET total = total + NEW.amount
     WHERE id = NEW.order_id;
     RETURN NEW;
   END;
   $$ LANGUAGE plpgsql;
   ```

Критические аспекты:

1. **Производительность:**
    - Срабатывают на каждую строку → замедляют массовые операции
    - Решение: оптимизация логики, переход на пакетную обработку
2. **Рекурсия:**
    - Триггер может вызвать другой триггер → бесконечный цикл

```
ALTER TABLE table_name DISABLE TRIGGER trigger_name;
```

3. **Порядок выполнения:**
    - Несколько триггеров на одно событие → порядок не гарантирован
    - Решение: консолидация логики в один триггер
4. **Транзакционность:**
    - Ошибка в триггере → откат всей транзакции

Best Practices:

1. **Именование:**
    - `trg_{table}_{action}_{time}` (пример: `trg_employees_before_update`)
2. **Логирование:**
    - Добавлять обработку исключений в тело триггера
3. **Тестирование:**
    - Учитывать NULL в `OLD`/`NEW`
    - Проверять массовые операции (`UPDATE table SET ...`)

СУБД-специфика:

Функция	PostgreSQL	SQL Server
Язык	PL/pgSQL	T-SQL

Функция	PostgreSQL	SQL Server
Возвращаемое значение	`RETURN NEW/OLD`	Не требуется
Управление	`CREATE TRIGGER`	`CREATE TRIGGER`
Отключение	`ALTER TABLE ... DISABLE TRIGGER`	`DISABLE TRIGGER`

Резюме для экзамена:

```
-- Создание
CREATE TRIGGER name BEFORE INSERT ON table
FOR EACH ROW EXECUTE FUNCTION fn();

-- Удаление
DROP TRIGGER name ON table;
```

Ключевые правила:

- Избегать сложной бизнес-логики в триггерах
- Документировать все триггеры в системе
- Минимизировать количество операций в теле

Блокировки

Блокировки (Locks)

Назначение: Контроль параллельного доступа к данным для обеспечения согласованности. Предотвращают конфликты при одновременном изменении данных.

Ключевые типы блокировок:

1. **По уровню гранулярности:**

- **Строковые (Row-level):**

  ```
  SELECT * FROM accounts WHERE id = 1 FOR UPDATE; -- X-блокировка строки
  ```

- **Страничные (Page-level):** Блокировка группы строк
- **Табличные (Table-level):**

  ```
  LOCK TABLE orders IN EXCLUSIVE MODE; -- PostgreSQL
  ```

- **Базы данных (Database-level)**

2. **По режиму доступа:**

Тип	Чтение	Запись	Совместимость
S (Shared)	✓	✗	Совместима с S, не с X
X (Exclusive)	✗	✓	Не совместима ни с чем
IS/IX (Intent)	-	-	Сигнал о намерении блокировки

Основные сценарии:

1. **Автоматические блокировки:**
   - `UPDATE`/`DELETE` → X-блокировки строк
   - `SELECT` → S-блокировки (зависит от изоляции)
2. **Явные блокировки:**

```
-- Блокировка строки для изменения
SELECT * FROM inventory
WHERE product_id = 100
FOR UPDATE;  -- X-блокировка
```

Проблемы параллелизма:

1. **Взаимоблокировка (Deadlock):**

```
graph LR
T1[Транзакция 1] -->|Блокирует строку A| A
T2[Транзакция 2] -->|Блокирует строку B| B
T1 -->|Ждет строку B| B
T2 -->|Ждет строку A| A
```

Решение: Детектор deadlock автоматически отменяет одну транзакцию

2. **Блокировки ожидания:**
   - `lock_timeout`: Автоотмена при долгом ожидании
     ```
     SET lock_timeout = '5s'; -- PostgreSQL
     ```

Управление блокировками:

1. **Мониторинг:**
   ```
   -- PostgreSQL
   SELECT * FROM pg_locks;

   -- MySQL
   SHOW ENGINE INNODB STATUS;

   -- SQL Server
   EXEC sp_lock;
   ```
2. **Изоляция транзакций:**

Уровень	Разрешено
`READ UNCOMMITTED`	Dirty reads
`READ COMMITTED` (default)	Non-repeatable reads
`REPEATABLE READ`	Phantom reads
`SERIALIZABLE`	Полная изоляция

3. **Оптимизация:**
   - `NOWAIT`: Ошибка вместо ожидания
     ```
     SELECT * FROM table FOR UPDATE NOWAIT;
     ```
   - `SKIP LOCKED`: Пропуск заблокированных строк
     ```
     SELECT * FROM tasks FOR UPDATE SKIP LOCKED;
     ```

СУБД-специфика:

1. **PostgreSQL:**

- MVCC (Multiversion Concurrency Control)
- Блокировки версий строк вместо данных

2. **SQL Server:**
   - Подсказки блокировок:
     ```
     SELECT * FROM table WITH (UPDLOCK, HOLDLOCK)
     ```
3. **Oracle:**
   - `SELECT FOR UPDATE WAIT [n]`

Best Practices:

1. **Короткие транзакции:** Минимизируйте время блокировки
2. **Одинаковый порядок:** Всегда блокируйте ресурсы в одном порядке
3. **Индексы:** Ускоряют поиск строк → сокращают время блокировки
4. **Избегайте:**
   - Долгих операций в транзакциях
   - `SELECT *` без ограничений

Резюме для экзамена:

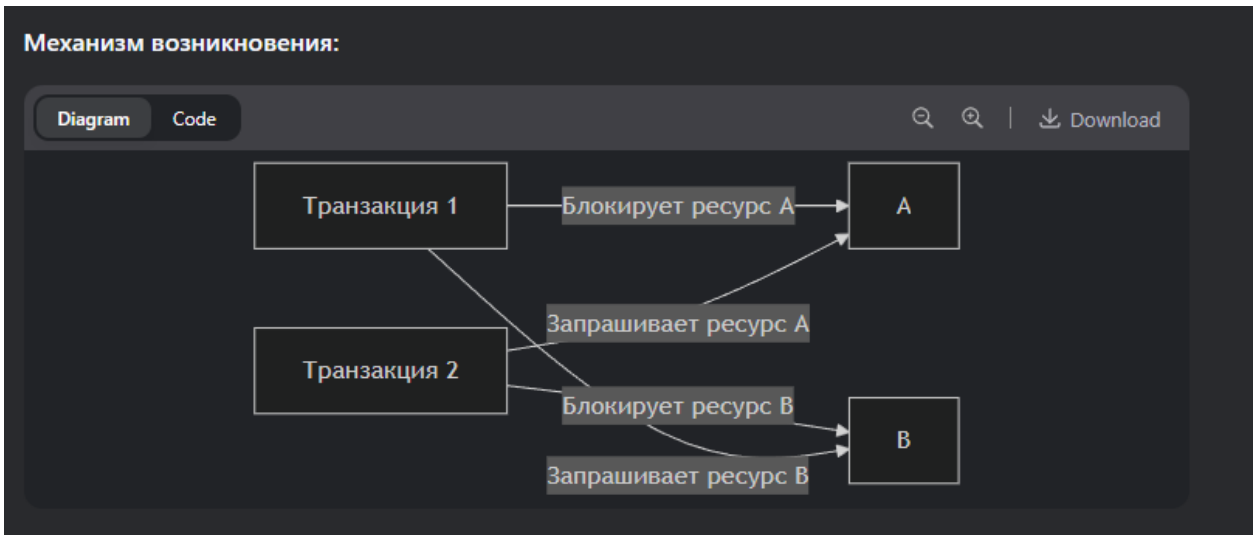
```
-- Явная блокировка строк
SELECT ... FOR UPDATE [NOWAIT|SKIP LOCKED];

-- Управление таймаутом
SET lock_timeout = '5s';

-- Анализ блокировок
SHOW LOCKS;  -- (СУБД-специфично)
```

Deadlock (Взаимоблокировка)

Определение: Ситуация, когда две или более транзакций взаимно блокируют друг друга, ожидая ресурсы.



Условия deadlock:

1. **Mutual exclusion** - ресурс монополен
2. **Hold and wait** - удержание ресурса + ожидание другого
3. **No preemption** - нельзя отобрать ресурс
4. **Circular wait** - круговая зависимость

Диагностика:

Симптомы:

- Зависание приложений
- Ошибки `1213` (MySQL), `1205` (SQL Server)
- Рост числа процессов в состоянии `waiting`

Инструменты:

```
-- MySQL
SHOW ENGINE INNODB STATUS;

-- SQL Server
SELECT * FROM sys.dm_tran_locks;
EXEC sp_who2;

-- PostgreSQL
SELECT * FROM pg_locks;
```

Пример воспроизведения:

Сессия 1:

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1; -- Блокировка строки 1
-- Ждет завершения сессии 2...
UPDATE accounts SET balance = balance + 100 WHERE id = 2; -- Запрос блокировки строки 2
COMMIT;
```

Сессия 2:

```
BEGIN;
UPDATE accounts SET balance = balance - 50 WHERE id = 2; -- Блокировка строки 2
UPDATE accounts SET balance = balance + 50 WHERE id = 1; -- Запрос блокировки строки 1 → DEADLOCK!
COMMIT;
```

Стратегии предотвращения:

1. **Единый порядок блокировки:**

   Всегда блокировать ресурсы в одинаковой последовательности

   ```
   - Транзакция 1: A → B
   - Транзакция 2: B → A
   + Транзакция 1: A → B
   + Транзакция 2: A → B
   ```

2. **Короткие транзакции:**

   Минимизируйте время удержания блокировок

3. **Оптимальные запросы:**

   - Фильтруйте по индексированным полям
   - Избегайте `SELECT FOR UPDATE` без `WHERE`

4. **Настройки СУБД:**

   ```
   -- MySQL
   SET innodb_lock_wait_timeout = 30; -- Таймаут ожидания
   ```

```
-- SQL Server
SET DEADLOCK_PRIORITY LOW; -- Приоритет при разрешении
```

Техники обработки:

1. **Автоматическое разрешение:**
    - Детектор deadlock убивает "жертву" (менее затратную транзакцию)
    - Ошибка `ERROR 1213 (40001): Deadlock found`