Теория

1. Типы данных

- Определение: Атрибуты, определяющие характер данных, которые могут храниться в столбце таблицы (целые числа, дробные, строки, даты, бинарные данные и т.д.).
- Основные Категории:
    - Числовые: `INT`/`INTEGER`, `SMALLINT`, `BIGINT`, `DECIMAL(p,s)`/`NUMERIC(p,s)` (точные, `p` - общее число цифр, `s` - после запятой), `FLOAT`, `REAL`, `DOUBLE PRECISION` (приближенные).
    - Символьные/Строковые: `CHAR(n)` (фиксированная длина, дополняется пробелами), `VARCHAR(n)`/`CHARACTER VARYING(n)` (переменная длина, до `n` символов), `TEXT` (длинный неограниченный текст).
    - Дата и Время: `DATE` (только дата), `TIME` (только время), `TIMESTAMP` (дата и время), `TIMESTAMP WITH TIME ZONE`, `INTERVAL` (промежуток времени).
    - Логический: `BOOLEAN` (TRUE, FALSE, NULL).
    - Бинарные: `BINARY(n)`, `VARBINARY(n)`, `BLOB` (Binary Large OBject).
    - Прочие: `JSON`, `XML`, `UUID`, перечисления (`ENUM`), массивы (`ARRAY` - зависит от СУБД).
- Важность: Обеспечивают целостность данных, влияют на эффективность хранения и производительность запросов.

2. Модели и определения

- Реляционная модель данных (РМД): Теоретическая основа большинства СУБД. Данные представляются в виде:
    - Отношений (Relations): По сути, таблицы.
    - Кортежей (Tuples): Строки таблицы.
    - Атрибутов (Attributes): Столбцы таблицы.
    - Ключей (Keys): Первичный ключ (`PRIMARY KEY`) - уникально идентифицирует строку. Внешний ключ (`FOREIGN KEY`) - ссылается на первичный ключ другой таблицы, обеспечивая ссылочную целостность.

- СУБД (DBMS): Система управления базами данных (напр., PostgreSQL, MySQL, SQL Server, Oracle) - программное обеспечение для создания, управления и взаимодействия с БД.
- Схема (Schema): Контейнер внутри БД для организации объектов (таблиц, представлений, функций и т.д.), управление правами доступа.

3. Что такое USE (или SET SCHEMA)

- Назначение: Устанавливает *текущую рабочую базу данных* или *схему* для сессии.
- Синтаксис (Пример):
```
USE DatabaseName; -- T-SQL (SQL Server), MySQL
SET SCHEMA SchemaName; -- Некоторые СУБД (DB2, иногда PostgreSQL)
```
- Эффект: Все последующие команды (SELECT, CREATE TABLE и т.д.) по умолчанию будут выполняться в контексте указанной базы данных или схемы. Позволяет не указывать базу/схему в каждом запросе явно (`DatabaseName.TableName` или `SchemaName.TableName`).

4. DROP TABLE: что удаляется и что остается

- Удаляется:
    - Сама таблица (ее определение из системного каталога).
    - *Все* данные, содержавшиеся в таблице.
    - *Все* индексы, созданные на этой таблице.
    - *Все* триггеры, связанные с этой таблицей.
- Не удаляется (остается):
    - Представления (`VIEWS`), которые ссылались на удаленную таблицу. Они становятся невалидными ("broken"). При попытке их использования возникнет ошибка.
    - Хранимые процедуры (`PROCEDURES`) или функции (`FUNCTIONS`), которые использовали эту таблицу. Они останутся в базе, но при выполнении запроса внутри них, обращающегося к удаленной таблице, возникнет ошибка.
    - Ограничения внешнего ключа (`FOREIGN KEY`) *в других таблицах*, которые ссылались на удаленную таблицу. Они также становятся

невалидными и могут помешать удалению таблицы (если не используется `CASCADE`).

- `DROP TABLE ... CASCADE`: Явное указание удалить таблицу *и* все объекты, которые от нее зависят (представления, триггеры на ней, внешние ключи на нее из других таблиц). Опасно!

5. HAVING, GROUP BY, DISTINCT, COUNT(*)

- GROUP BY: Группирует строки результата по значениям одного или нескольких указанных столбцов. Обычно используется с агрегатными функциями (`SUM()`, `AVG()`, `MIN()`, `MAX()`, `COUNT()`), которые вычисляют одно значение для каждой группы.

```
SELECT department_id, COUNT(*) AS num_employees, AVG(salary)
FROM employees
GROUP BY department_id; -- Группировка по отделу, подсчет сотрудников и ср. зарплата в каждом
```

- HAVING: Фильтрует результаты *после* группировки (`GROUP BY`) по результатам агрегатных функций. Нельзя использовать для фильтрации отдельных строк до группировки (для этого используется `WHERE`).

```
SELECT department_id, AVG(salary) AS avg_sal
FROM employees
GROUP BY department_id
HAVING AVG(salary) > 50000; -- Фильтр по результату агрегации (ср. зп)
```

- DISTINCT: Удаляет дубликаты строк из результирующего набора *до* применения агрегации или вывода. Применяется ко всем столбцам в `SELECT`.

```
SELECT DISTINCT job_id FROM employees; -- Уникальные должности
```

- COUNT(*): Агрегатная функция, подсчитывающая *все* строки в группе (или в таблице, если нет `GROUP BY`), включая строки со значениями `NULL`.
- COUNT(column): Подсчитывает количество строк в группе, где значение указанного столбца *не является* `NULL`.

```
SELECT COUNT(*) AS total_rows, COUNT(manager_id) AS has_manager_count
FROM employees;
```

6. WHERE vs WHERE EXISTS

- WHERE: Фильтрует *отдельные строки* в основной таблице (или после `JOIN`) *до* группировки (`GROUP BY`) или применения агрегатных функций.

Использует условия сравнения (`=`, `>`, `<`, `LIKE`, `IN`, `BETWEEN` и т.д.) со столбцами таблиц.

```
SELECT * FROM employees WHERE salary > 70000;
```

- WHERE EXISTS: Фильтрует строки основной таблицы на основе *существования* хотя бы одной строки в подзапросе (коррелированном или нет). Возвращает `TRUE`, если подзапрос вернул хотя бы одну строку. Ключевое отличие: Работает с результатом *существования* строки из подзапроса, а не с конкретными значениями. Часто используется для проверки связей ("есть ли заказы у этого клиента?").

```
SELECT *
FROM departments d
WHERE EXISTS (
    SELECT 1
    FROM employees e
    WHERE e.department_id = d.department_id -- Корреляция
    AND e.salary > 100000
); -- Найти отделы, в которых есть хотя бы один сотрудник с зп > 100к
```

7. IS NULL

- Назначение: Оператор для проверки, является ли значение выражения `NULL`.
- Синтаксис: `expression IS [NOT] NULL`
- Важность: `NULL` означает "неизвестное", "отсутствующее" или "неприменимое" значение. Нельзя сравнивать с `NULL` с помощью обычных операторов сравнения (`=`, `<>`). `NULL = NULL` не `TRUE`, а `UNKNOWN` (что в логике SQL эквивалентно `FALSE` для условий `WHERE`).

```
SELECT * FROM employees WHERE manager_id IS NULL; -- Сотрудники без мен
еджера
SELECT * FROM employees WHERE manager_id = NULL; -- Этот запрос НЕ сраб
отает, вернет 0 строк!
```

8. LIKE и Агрегатные функции

- LIKE:
    - Оператор для сопоставления строки с шаблоном.
    - Использует подстановочные знаки:
        - `%`: Любая последовательность символов (включая пустую).
        - `_`: Ровно один любой символ.

    - Регистрозависимость: Зависит от параметров сортировки (collation) СУБД (часто не зависит в MySQL, зависит в PostgreSQL/SQL Server по умолчанию). Используйте `ILIKE` (PostgreSQL) или функции (`UPPER()`, `LOWER()`) для регистронезависимого поиска.

```
SELECT * FROM products WHERE name LIKE 'App%'; -- Начинается с "App"
SELECT * FROM products WHERE name LIKE 'C_mp%'; -- Второй символ любой
```

- Агрегатные функции:
    - Функции, выполняющие вычисление на наборе строк (часто в группе) и возвращающие одно значение.
    - Основные: `SUM(column)`, `AVG(column)`, `MIN(column)`, `MAX(column)`, `COUNT(expression)` (`COUNT(*)` или `COUNT(column_name)`).
    - Особенности: Обычно игнорируют значения `NULL` (кроме `COUNT(*)`).

9. CASE

- Назначение: Условное выражение, позволяющее выполнять ветвление логики прямо в SQL-запросе (аналог `if-then-else`).
- Формы:
    1. Простая (Simple CASE): Сравнивает выражение с набором значений.

```
SELECT employee_id,
       CASE department_id
           WHEN 10 THEN 'Accounting'
           WHEN 20 THEN 'Research'
           WHEN 30 THEN 'Sales'
           ELSE 'Other'
       END AS dept_name
FROM employees;
```

    2. Поисковая (Searched CASE): Проверяет набор логических условий.

```
SELECT employee_id, salary,
       CASE
           WHEN salary >= 100000 THEN 'High'
           WHEN salary >= 60000 THEN 'Medium'
           WHEN salary < 60000 THEN 'Low'
           ELSE 'Unknown'
       END AS salary_grade
FROM employees;
```

- Использование: В `SELECT`, `WHERE`, `ORDER BY`, `HAVING`, `UPDATE` (`SET`), `INSERT` (`VALUES`) и т.д. Возвращает значение первого

выполнившегося условия `WHEN`. `ELSE` необязателен, но рекомендуется. Если ни одно условие не истинно и нет `ELSE`, возвращает `NULL`.

10. Типы JOIN

- Соединения (JOIN): Комбинируют строки из двух или более таблиц на основе связанных столбцов.
- Виды:
    - CROSS JOIN: Декартово произведение. Каждая строка первой таблицы соединяется с каждой строкой второй таблицы. `N x M` строк. Редко используется напрямую.
      ```
      SELECT * FROM table1 CROSS JOIN table2;
      ```
    - [INNER] JOIN: Возвращает строки, где есть совпадение (`ON`/`USING`) в *обеих* таблицах. Самый распространенный тип.
      ```
      SELECT * FROM orders o
      INNER JOIN customers c ON o.customer_id = c.customer_id;
      ```
    - LEFT [OUTER] JOIN: Возвращает *все* строки из левой (первой) таблицы и соответствующие строки из правой таблицы. Если соответствия нет, для столбцов правой таблицы возвращаются `NULL`.
      ```
      SELECT c.customer_name, o.order_id
      FROM customers c
      LEFT JOIN orders o ON c.customer_id = o.customer_id; -- Все клиен
      ты, даже без заказов
      ```
    - RIGHT [OUTER] JOIN: Возвращает *все* строки из правой (второй) таблицы и соответствующие строки из левой таблицы. Если соответствия нет, для столбцов левой таблицы возвращаются `NULL`. (Менее распространен, часто заменяется на `LEFT JOIN` с перестановкой таблиц).
    - FULL [OUTER] JOIN: Возвращает все строки из обеих таблиц. Если строке из одной таблицы нет соответствия в другой, для столбцов недостающей таблицы возвращаются `NULL`. Объединение результатов `LEFT JOIN` и `RIGHT JOIN`.
      ```
      SELECT * FROM employees e
      FULL JOIN departments d ON e.department_id = d.department_id; --
      Все сотрудники и все отделы
      ```

11. Псевдонимы ("Aliases")

- Назначение: Временное имя, присваиваемое столбцу или таблице в запросе для удобства чтения, сокращения или устранения неоднозначности.
- Синтаксис столбца: `SELECT column_name AS alias_name ...` (`AS` часто можно опустить).
- Синтаксис таблицы: `FROM table_name alias_name ...` (`AS` обычно *нельзя* использовать для псевдонимов таблиц в большинстве СУБД).
- Применение:
    - Укорачивание длинных имен столбцов/таблиц.
    - Присвоение понятных имен результатам выражений или агрегатных функций.
    - Устранение неоднозначности при наличии одинаковых имен столбцов в разных таблицах (особенно в `JOIN`).
    - Ссылка на таблицу в коррелированных подзапросах.

```
SELECT e.last_name AS surname, d.department_name AS dept,
       (SELECT COUNT(*) FROM employees e2 WHERE e2.department_id = e.de
partment_id) AS emp_count
FROM employees e
INNER JOIN departments d ON e.department_id = d.department_id
WHERE e.salary > 100000;
```

12. UNION

- Назначение: Оператор для объединения результирующих наборов двух или более `SELECT`-запросов в один набор строк.
- Требования:
    - Количество столбцов в каждом `SELECT` должно быть одинаковым.
    - Типы данных соответствующих столбцов должны быть совместимыми (или приводимыми).
- Поведение: По умолчанию (`UNION`) удаляет дубликаты строк из общего результата. Для сохранения дубликатов используется `UNION ALL`.
- Порядок: Строки результирующего набора не упорядочены. Для упорядочивания используйте `ORDER BY` *в самом конце* (применяется ко всему объединенному результату).
- Пример:

```
SELECT customer_id, 'Active' AS status FROM customers WHERE active = 1
```

```
UNION
SELECT customer_id, 'Inactive' AS status FROM customers WHERE active =
0
ORDER BY customer_id;
```

13. Управляющие конструкции (IF-ELSE, GROUP BY, Локальные переменные)

- IF-ELSE: Не является частью стандартного SQL запроса `SELECT`! Это конструкция процедурных расширений (PL/pgSQL, T-SQL, PL/SQL) для управления потоком выполнения внутри хранимых процедур, функций или скриптов.

```
-- Пример T-SQL
DECLARE @Sal MONEY = 75000;
IF @Sal > 100000
    PRINT 'High Salary'
ELSE
    PRINT 'Not High Salary';
```

- GROUP BY: (См. п.5) Используется в `SELECT` для агрегации данных по группам строк.
- Локальные переменные (`@var` в T-SQL, `DECLARE var ...` в других): Не часть стандартного SQL запроса `SELECT`! Используются внутри хранимых процедур, функций, пакетов или скриптов для временного хранения значений. В T-SQL для переменных уровня сессии/пакета используется префикс `@`. Объявляются через `DECLARE`.

```
-- Пример T-SQL
DECLARE @MinSal MONEY;
SET @MinSal = (SELECT MIN(salary) FROM employees);
SELECT * FROM employees WHERE salary > @MinSal;
```

14. DECLARE (и DECIMAL)

- DECLARE: Ключевое слово процедурных расширений SQL. Используется для объявления:
    - Локальных переменных внутри хранимых процедур, функций, триггеров или пакетов.
    - Курсоров.
    - Обработчиков исключений.
    - Синтаксис (пример T-SQL): `DECLARE @VariableName DataType [ = InitialValue ];`

- DECIMAL(p,s): Тип данных (см. п.1). Используется для хранения точных числовых значений с фиксированной запятой.
    - `p` (precision) - Общее количество цифр (макс. обычно 38).
    - `s` (scale) - Количество цифр после десятичной запятой.
    - Пример: `DECIMAL(10,2)` - числа вида `12345678.99`. Критически важен для финансовых расчетов, где важна точность (в отличие от `FLOAT`).

15. WHILE, TRY-CATCH-THROW

- WHILE: Конструкция процедурных расширений. Цикл, выполняющий блок кода, пока условие истинно.

```
-- Пример T-SQL
DECLARE @Counter INT = 1;
WHILE @Counter <= 10
BEGIN
    PRINT 'Iteration: ' + CAST(@Counter AS VARCHAR);
    SET @Counter = @Counter + 1;
END;
```

- TRY-CATCH: Конструкция обработки ошибок в процедурных расширениях (T-SQL, PL/pgSQL и др.).
    - `BEGIN TRY ... END TRY`: Блок кода, в котором отслеживаются ошибки.
    - `BEGIN CATCH ... END CATCH`: Блок, выполняемый при возникновении ошибки в соответствующем блоке `TRY`. Содержит функции для получения информации об ошибке (`ERROR_MESSAGE()`, `ERROR_NUMBER()`, `ERROR_STATE()` и т.д.).

```
-- Пример T-SQL
BEGIN TRY
    -- Код, который может вызвать ошибку (напр., деление на 0)
    SELECT 1 / 0;
END TRY
BEGIN CATCH
    PRINT 'Error: ' + ERROR_MESSAGE();
END CATCH;
```

- THROW (или RAISE в PostgreSQL): Конструкция процедурных расширений. Используется для *генерации* (выбрасывания) исключения (ошибки).
    - В `CATCH`-блоке: для повторного выброса перехваченной ошибки наверх (`THROW;` в T-SQL) или генерации новой ошибки (`THROW`

`<номер>, <сообщение>, <состояние>;` в T-SQL, `RAISE EXCEPTION ...` в PL/pgSQL).

16. Хранимые процедуры (Stored Procedures) и Функции (Functions)

- Хранимая процедура:
    - Предварительно скомпилированный блок кода SQL и процедурных расширений, хранящийся на сервере БД.
    - Вызов: `EXECUTE`/`EXEC ProcedureName @Param1, ...;` или `CALL ProcedureName(...);` (зависит от СУБД).
    - Может:
        - Принимать входные параметры (`IN`).
        - Возвращать выходные параметры (`OUT`).
        - Возвращать несколько результирующих наборов (через `SELECT` внутри).
        - Выполнять DML (изменять данные), DDL (создавать/удалять объекты - осторожно!).
        - Содержать сложную бизнес-логику, транзакции, управляющие конструкции.
    - Не может: Использоваться напрямую внутри SQL-выражения (как часть `SELECT`, `WHERE`).
    - Возвращаемое значение: Обычно "успех/неудача" (через код возврата) или через `OUT`-параметры.
- Функция (User-Defined Function - UDF):
    - Предварительно скомпилированный блок кода SQL и процедурных расширений, хранящийся на сервере БД.
    - Вызов: Используется *внутри* SQL-выражений (`SELECT`, `WHERE`, `SET` и т.д.): `SELECT FunctionName(@Param1, ...) ...`.
    - Обязана: Возвращать *одно* скалярное значение (число, строку, дату) или *табличное* значение (набор строк - `RETURNS TABLE`).
    - Может: Принимать входные параметры (`IN`). Обычно *не может* иметь `OUT`-параметров (если не скалярная) и не должна изменять состояние БД (не выполнять DML/DDL - это требование "чистоты" функции для использования в запросах; есть исключения,

но не рекомендуется). Должна быть детерминированной (не всегда, но желательно).
    - Возвращаемое значение: Через оператор `RETURN` (скаляр) или `RETURN QUERY` (таблица).

17. Индексы

- Назначение: Структуры данных, ускоряющие поиск, сортировку (`ORDER BY`) и соединение (`JOIN`) данных в таблицах. Работают как оглавление книги.
- Как работают: Индекс создается на одном или нескольких столбцах таблицы. СУБД хранит отсортированные (или структурированные иным оптимальным способом) значения этих столбцов вместе с указателями на соответствующие строки в таблице.
- Типы:
    - Кластерный индекс (Clustered Index): Определяет физический порядок хранения данных в таблице. Таблица может иметь только один кластерный индекс. Часто создается на первичном ключе (`PRIMARY KEY`). Данные в таблице физически отсортированы по ключу кластерного индекса.
    - Некластерный индекс (Non-Clustered Index): Отдельная структура данных, хранящая ключ индекса и указатель (обычно на кластерный ключ или RID) на соответствующую строку в таблице. Таблица может иметь много некластерных индексов. Данные в таблице физически *не переупорядочиваются*.
    - Уникальный индекс (Unique Index): Гарантирует уникальность значений в индексируемых столбцах (может быть как кластерным, так и некластерным). Первичный ключ автоматически создает уникальный кластерный (или некластерный) индекс.
    - Составной индекс (Composite Index): Индекс, созданный на двух или более столбцах.
    - Другие (зависит от СУБД): Filtered/Partial (только часть строк), Columnstore, Full-text, Spatial, Hash и т.д.
- Плюсы: Ускорение запросов (`WHERE`, `JOIN`, `ORDER BY`), обеспечение уникальности (`UNIQUE`).

- Минусы: Занимают место на диске, замедляют операции `INSERT`/`UPDATE`/`DELETE` (т.к. нужно обновлять и индекс). Не индексируйте все подряд! Индексируйте столбцы, часто используемые в `WHERE`, `JOIN`, `ORDER BY`.

18. Представления (Views)

- Назначение: Виртуальная таблица, результат сохраненного `SELECT`-запроса. Не хранит данные физически (за исключением материализованных представлений).
- Синтаксис: `CREATE VIEW ViewName AS SELECT ... ;`
- Как используется: `SELECT * FROM ViewName;` (Можно использовать как обычную таблицу в `SELECT`, часто в `JOIN`).
- Преимущества:
    - Упрощение сложных запросов: Скрывают сложность базового запроса.
    - Безопасность: Ограничивают доступ пользователей только к определенным столбцам/строкам базовых таблиц (через `GRANT` на представление).
    - Согласованность: Предоставляют постоянный интерфейс к данным, даже если структура базовых таблиц меняется (если возможно).
    - Логическая независимость: Позволяют реструктурировать базовые таблицы, не меняя запросы приложений, работающих с представлением.
- Материализованные представления (Materialized Views): Физически хранят результат запроса. Требуют обновления (`REFRESH`). Используются для ускорения сложных агрегирующих запросов.

19. Триггер (Trigger)

- Определение (как в методичке): Хранимый программный модуль (кусок кода), автоматически выполняемый СУБД в ответ на наступление определенного события, связанного с таблицей или представлением.
- События: `INSERT`, `UPDATE`, `DELETE` (DML-триггеры) или `CREATE`, `ALTER`, `DROP` (DDL-триггеры, менее распространены).

- Момент срабатывания:
    - `BEFORE` / `INSTEAD OF`: Выполняется *до* выполнения самого оператора (или *вместо* него, для `INSTEAD OF`).
    - `AFTER`: Выполняется *после* успешного выполнения оператора (и после проверки ограничений).
- Уровень:
    - `FOR EACH ROW` (Row-level trigger): Выполняется *один раз для каждой строки*, затронутой оператором. Внутри триггера есть доступ к данным старой (`OLD`) и новой (`NEW`) строки (для `UPDATE` и `DELETE` - `OLD`, для `INSERT` и `UPDATE` - `NEW`).
    - `FOR EACH STATEMENT` (Statement-level trigger): Выполняется *один раз для всего оператора*, независимо от количества затронутых строк. Доступа к `OLD`/`NEW` *нет*.
- Назначение: Обеспечение сложной бизнес-логики, аудит изменений, поддержание согласованности данных, которые нельзя обеспечить через `CHECK`/`FOREIGN KEY`, реализация каскадных операций, проверка сложных условий.
- Пример (аудит):

```
CREATE TRIGGER trg_AuditEmployeeUpdate
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO employee_audit(emp_id, changed_field, old_value, new_va
lue, change_time)
    VALUES (OLD.employee_id, 'SALARY', OLD.salary, NEW.salary, CURRENT_
TIMESTAMP);
END;
```

20. DML vs DDL

- Язык манипулирования данными (Data Manipulation Language - DML):
    - Назначение: Работа с данными внутри существующих объектов БД (таблиц).
    - Основные команды:
        - `SELECT`: Извлечение данных.
        - `INSERT`: Добавление новых строк.

        - `UPDATE`: Изменение существующих строк.
        - `DELETE`: Удаление строк.
        - `MERGE` (`UPSERT`): Объединение вставки и обновления.
    - Особенность: Операции DML могут быть частью транзакций (`COMMIT`/`ROLLBACK`).
- Язык определения данных (Data Definition Language - DDL):
    - Назначение: Создание, изменение и удаление *структуры* объектов БД (схем, таблиц, индексов, представлений, процедур, триггеров и т.д.).
    - Основные команды:
        - `CREATE`: Создание объекта (базы, таблицы, индекса...).
        - `ALTER`: Изменение существующего объекта.
        - `DROP`: Удаление объекта.
        - `TRUNCATE TABLE`: Быстрое удаление *всех* строк из таблицы (сбрасывает идентификаторы, обычно не журнализируется построчно - осторожно!).
        - `RENAME`: Переименование объекта.
    - Особенность: Операции DDL неявно завершают текущую транзакцию (выполняют `COMMIT` перед собой и после себя в большинстве СУБД). Их нельзя откатить обычным `ROLLBACK` (хотя некоторые СУБД поддерживают DDL внутри транзакций с возможностью отката).

21. COMMIT TRANSACTION, ROLLBACK TRANSACTION, BEGIN TRANSACTION

- Транзакция: Логическая единица работы в СУБД. Последовательность операторов DML (и некоторых DDL, осторожно), которая должна быть выполнена как единое целое по принципу "всё или ничего".
- Свойства ACID:
    - A (Atomicity - Атомарность): Транзакция выполняется целиком или не выполняется вовсе. Гарантируется `COMMIT`/`ROLLBACK`.
    - C (Consistency - Согласованность): Транзакция переводит БД из одного согласованного состояния в другое (сохраняются все ограничения целостности - PK, FK, CHECK).

    - I (Isolation - Изолированность): Параллельно выполняющиеся транзакции не должны мешать друг другу. Уровни изоляции (`READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, `SERIALIZABLE`) определяют степень этой изоляции (и накладных расходов).
    - D (Durability - Долговечность): Если транзакция зафиксирована (`COMMIT`), ее результаты гарантированно сохраняются в БД, даже в случае сбоя системы.
- Ключевые команды:
    - `BEGIN TRANSACTION` (или `START TRANSACTION`): Явно начинает новую транзакцию. (Необязательно, если используется режим неявной транзакции).
    - `COMMIT TRANSACTION` (или `COMMIT`): Фиксирует транзакцию. Все изменения, сделанные в транзакции, становятся постоянными и видимыми другим пользователям. Освобождает ресурсы (блокировки).
    - `ROLLBACK TRANSACTION` (или `ROLLBACK`): Откатывает транзакцию. Все изменения, сделанные в текущей транзакции, отменяются. Освобождает ресурсы (блокировки). Может откатить к точке сохранения (`SAVEPOINT`).
    - `SAVEPOINT SavepointName`: Создает точку сохранения внутри транзакции.
    - `ROLLBACK TO SAVEPOINT SavepointName`: Откатывает транзакцию к указанной точке сохранения, не завершая всю транзакцию.
- Режимы: Явные транзакции (управляются командами `BEGIN`/`COMMIT`/`ROLLBACK`) и неявные транзакции (каждый оператор выполняется как отдельная транзакция, если не включен режим явных).

22. Транзакции (Общее)

- Жизненный цикл: Начало (`BEGIN`) -> Выполнение операторов (DML) -> [Создание точек сохранения (`SAVEPOINT`)] -> Фиксация (`COMMIT`) / Откат (`ROLLBACK` [TO `SAVEPOINT`]).

- Уровни изоляции (Isolation Levels): Определяют, какие "побочные эффекты" параллельных транзакций видит текущая транзакция. Чем выше уровень, тем строже изоляция (меньше аномалий), но ниже производительность из-за блокировок.
    - READ UNCOMMITTED: Самая низкая. Видны "грязные" чтения (`Dirty Reads` - данные из незафиксированных транзакций).
    - READ COMMITTED (По умолчанию в большинстве СУБД): Гарантирует, что читаются только зафиксированные данные. Возможны `Non-Repeatable Reads` (значение строки изменилось при повторном чтении) и `Phantom Reads` (появились новые строки при повторном чтении).
    - REPEATABLE READ: Гарантирует, что строки, прочитанные один раз, не изменятся при повторном чтении в той же транзакции. Возможны `Phantom Reads`.
    - SERIALIZABLE: Самая высокая. Гарантирует полную изоляцию, как будто транзакции выполняются строго последовательно. Исключает `Dirty Reads`, `Non-Repeatable Reads`, `Phantom Reads`. Наибольшие накладные расходы.
- Блокировки (Locks): Механизм, используемый СУБД для обеспечения изоляции и согласованности. Типы: разделяемые (`Shared` - для чтения), монопольные (`Exclusive` - для записи), намеренные (`Intent`). Управляются автоматически СУБД в зависимости от уровня изоляции и выполняемых операций. Могут приводить к блокировкам (`Deadlocks`), которые СУБД обнаруживает и разрешает (обычно откатом одной из транзакций).
- Точки сохранения (Savepoints): Позволяют откатить часть транзакции, не отменяя ее всю.
- Вложенные транзакции: В большинстве СУБД не поддерживаются "нативно". `BEGIN TRANSACTION` внутри другой транзакции часто просто увеличивает счетчик вложенности, а `COMMIT` уменьшает его. Фиксация происходит только по `COMMIT` самого внешнего уровня. `ROLLBACK` откатывает *всю* транзакцию, независимо от уровня вложенности.

23. Блокировка (Lock)

Что это?

Механизм, который **ограничивает доступ** к данным другим транзакциям, пока текущая транзакция работает с ними. Нужен для предотвращения конфликтов при параллельном доступе.

Какие бывают?

1. **По уровню доступа:**
    - **Разделяемая (Shared Lock):** Транзакция читает данные. Другие транзакции могут тоже ставить разделяемые блокировки, но не эксклюзивные.
    - **Эксклюзивная (Exclusive Lock):** Транзакция изменяет данные. Никто другой не может читать или писать заблокированные данные.
2. **По уровню granularity (детализации):**
    - **Строка (Row-level):** Блокировка только одной строки.
    - **Страница (Page-level):** Блокировка страницы памяти (например, 8 КБ данных).
    - **Таблица (Table-level):** Блокировка всей таблицы.

Пример конфликта блокировок:

- Транзакция 1 поставила эксклюзивную блокировку на строку → Транзакция 2 ждет, пока блокировка не снимется.

24. Взаимоблокировка (Deadlock)

Что это?

Ситуация, когда **две или более транзакций** ждут друг друга, чтобы завершиться. Ни одна не может продолжить работу.

Пример deadlock:

1. Транзакция 1 блокирует строку A и хочет заблокировать строку B.
2. Транзакция 2 блокирует строку B и хочет заблокировать строку A.
   → Обе транзакции ждут друг друга → **Deadlock!**

Как решается?

- СУБД автоматически обнаруживает deadlock через определенное время.
- **Одна из транзакций отменяется (ROLLBACK),** чтобы снять блокировку.

Как избежать deadlock:

- Работать с данными в **одинаковом порядке** (например, всегда блокировать A → B → C).
- Использовать короткие транзакции.
- Использовать уровень изоляции `READ COMMITTED`.

Чек-лист для запоминания ✅

- **Транзакция** = "все или ничего". Бывают явные, неявные.
- **Блокировки** = защита от конфликтов. Типы: разделяемые (на чтение), эксклюзивные (на запись).
- **Deadlock** = взаимная блокировка. Решение: откат одной транзакции.

Примеры кода по каждому из пунктов

1. Типы данных

```
CREATE TABLE Users (
    id INT PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    balance DECIMAL(10, 2),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    is_active BOOLEAN DEFAULT true
);
```

2. Модели и определения

```
-- Создание таблицы с PK и FK
CREATE TABLE Departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(100) NOT NULL
);

CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100) NOT NULL,
    dept_id INT REFERENCES Departments(dept_id)
);
```

3. USE

```
USE AdventureWorks; -- Выбор базы для работы
SELECT * FROM Products; -- Запрос выполняется в AdventureWorks
```

4. DROP TABLE

```
DROP TABLE TempData; -- Удаляет таблицу и данные
-- Представления, хранимые процедуры, ссылающиеся на TempData, останутся но с
танут невалидными
```

5. HAVING и GROUP BY

```
SELECT department_id, AVG(salary) AS avg_salary, COUNT(*) AS emp_count
FROM employees
GROUP BY department_id
HAVING AVG(salary) > 50000 AND COUNT(*) > 5;
```

6. WHERE vs WHERE EXISTS

```
-- WHERE
SELECT * FROM orders WHERE total_amount > 1000;
```

```sql
-- WHERE EXISTS
SELECT * FROM customers c
WHERE EXISTS (
    SELECT 1 FROM orders o
    WHERE o.customer_id = c.customer_id
    AND o.order_date > '2023-01-01'
);
```

7. IS NULL

```sql
SELECT * FROM employees
WHERE manager_id IS NULL; -- Сотрудники без менеджера

SELECT * FROM contacts
WHERE phone IS NOT NULL; -- Контакты с указанным телефоном
```

8. LIKE и агрегатные функции

```sql
-- LIKE
SELECT * FROM products
WHERE product_name LIKE 'Apple%'; -- Начинается с Apple

-- Агрегатные функции
SELECT category, MAX(price), MIN(price), AVG(price)
FROM products GROUP BY category;
```

9. CASE

```sql
SELECT product_name, price,
    CASE
        WHEN price > 1000 THEN 'Premium'
        WHEN price > 500 THEN 'Standard'
        ELSE 'Budget'
    END AS price_category
FROM products;
```

10. JOIN

```sql
-- INNER JOIN
SELECT o.order_id, c.name
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id;

-- LEFT JOIN
SELECT d.dept_name, e.emp_name
FROM departments d
LEFT JOIN employees e ON d.dept_id = e.dept_id;
```

```sql
-- FULL OUTER JOIN
SELECT *
FROM table1
FULL OUTER JOIN table2 ON table1.id = table2.id;
```

11. Псевдонимы

```sql
SELECT
    e.emp_name AS "Employee",
    d.dept_name AS "Department",
    (SELECT MAX(salary) FROM employees) AS max_sal
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id;
```

12. UNION

```sql
SELECT 'Active' AS status, COUNT(*) FROM users WHERE is_active = 1
UNION
SELECT 'Inactive', COUNT(*) FROM users WHERE is_active = 0;
```

13. IF-ELSE и переменные (T-SQL)

```sql
DECLARE @avg_salary DECIMAL;
SELECT @avg_salary = AVG(salary) FROM employees;

IF @avg_salary > 50000
    PRINT 'Above average'
ELSE
    PRINT 'Below average';
```

14. DECLARE и DECIMAL

```sql
DECLARE @tax_rate DECIMAL(4,2) = 0.15; -- 15% налог
SELECT product, price * @tax_rate AS tax
FROM products;
```

15. WHILE и TRY-CATCH (T-SQL)

```sql
DECLARE @counter INT = 1;
WHILE @counter <= 5
BEGIN
    PRINT 'Iteration: ' + CAST(@counter AS VARCHAR);
    SET @counter += 1;
END;

BEGIN TRY
    INSERT INTO orders VALUES (NULL, 100); -- Ошибка
END TRY
BEGIN CATCH
```

```
    PRINT 'Error: ' + ERROR_MESSAGE();
    THROW;
END CATCH;
```

16. Процедуры и функции

```
-- Процедура
CREATE PROCEDURE GetEmployee @emp_id INT
AS
SELECT * FROM employees WHERE emp_id = @emp_id;
GO
EXEC GetEmployee 123;

-- Функция
CREATE FUNCTION CalculateBonus(@salary DECIMAL)
RETURNS DECIMAL
AS
BEGIN
    RETURN @salary * 0.1;
END;
GO
SELECT emp_name, dbo.CalculateBonus(salary) FROM employees;
```

17. Индексы

```
-- Создание индекса
CREATE INDEX idx_employee_name ON employees(emp_name);

-- Кластерный индекс (создается автоматически для PRIMARY KEY)
```

18. Представления

```
CREATE VIEW ActiveUsers AS
SELECT user_id, username, email
FROM users
WHERE is_active = 1;
GO
SELECT * FROM ActiveUsers; -- Использование
```

19. Триггер

```
CREATE TRIGGER log_salary_changes
ON employees
AFTER UPDATE
AS
BEGIN
    IF UPDATE(salary)
        INSERT INTO salary_log(emp_id, old_salary, new_salary)
        SELECT d.emp_id, d.salary, i.salary
```

```
        FROM deleted d
        JOIN inserted i ON d.emp_id = i.emp_id;
END;
```

20. DML vs DDL

```
-- DML (работа с данными)
INSERT INTO employees VALUES (1, 'John Doe', 75000);
UPDATE employees SET salary = 80000 WHERE emp_id = 1;

-- DDL (работа со структурой)
CREATE TABLE projects (id INT, name VARCHAR(100));
ALTER TABLE employees ADD bonus DECIMAL(10,2);
```

21. Транзакции

```
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT TRANSACTION; -- Фиксация изменений

-- ROLLBACK TRANSACTION; -- Откат изменений
```

22. Команды транзакций

```
BEGIN TRANSACTION;
SAVE TRANSACTION Savepoint1;
INSERT INTO orders VALUES (1, 100);
SAVE TRANSACTION Savepoint2;
INSERT INTO order_details VALUES (1, 1, 5);
-- ROLLBACK TRANSACTION Savepoint2; -- Откат к Savepoint2
COMMIT TRANSACTION;
```

Каждый пример демонстрирует базовый синтаксис конструкции. Для более сложных сценариев можно комбинировать эти элементы. Удачи на экзамене!