Лекция 1

1. Введение в базы данных

Проблемы хранения данных в файлах

- Отсутствие гибкости при изменении структуры
- Проблемы с многопользовательским доступом
- Вынужденная избыточность данных

Решение: Базы данных (БД) — файлы с описанием хранимых данных под управлением СУБД

2. Классификация СУБД

По степени распределённости:

- Локальные (односерверные)
- Распределённые (многосерверные)

По способу доступа:

1. **Файл-серверные** (Access, dBase)
    - Данные на файл-сервере, СУБД на клиентах
2. **Клиент-серверные** (Oracle, PostgreSQL, SQL Server)
    - СУБД и данные на сервере
3. **Встраиваемые** (SQLite, BerkeleyDB)
    - Встроены в приложение

По модели данных:

- Сетевые (OrientDB)
- Объектно-ориентированные (InterSystems Cache)
- Реляционные и объектно-реляционные (PostgreSQL, Oracle)

3. Реляционные базы данных

Основные принципы (по Кодду):

1. Данные в таблицах с уникальными именами
2. Строки с одинаковой структурой в таблице
3. Одно значение на пересечении строки/столбца
4. Уникальность строк (хотя бы по одному полю)
5. Именованные столбцы с однородными данными
6. Отсутствие скрытых связей между таблицами
7. Независимость обработки от порядка строк/столбцов
8. Группировка таблиц в схемы

Терминология:

- Таблица (отношение)
- Строка (кортеж)
- Столбец (атрибут)
- Первичный ключ (уникальный идентификатор)
- Внешний ключ (ссылка на другую таблицу)

4. Целостность данных

Типы целостности:

1. **По сущностям**: первичный ключ не может быть NULL
2. **По ссылкам**: внешний ключ должен ссылаться на существующую запись или быть NULL
3. **Пользовательская**:
    - Уникальность значений
    - Диапазоны значений (например, оценки 2-5)
    - Принадлежность набору (например, пол "М"/"Ж")

5. Язык SQL

История:

- Разработан IBM в 1970-х
- Стандарты: SQL-86, SQL-89, SQL-92, SQL-99, SQL:2003 и др.

Категории операторов:

1. **DDL (Data Definition Language)**:
    - CREATE, ALTER, DROP — создание/изменение/удаление объектов
2. **DML (Data Manipulation Language)**:
    - SELECT, INSERT, UPDATE, DELETE — работа с данными
3. **DCL (Data Control Language)**:
    - GRANT, REVOKE — управление правами доступа

Оператор SELECT:

sql

Copy

Download

```sql
SELECT поля FROM таблицы
[WHERE условия]
[GROUP BY поля_группировки]
[HAVING условия_агрегации]
[ORDER BY поля_сортировки]
```

Агрегатные функции:

- COUNT(), SUM(), AVG(), MIN(), MAX()

GROUP BY и HAVING:

- GROUP BY — группировка строк для агрегации
- HAVING — фильтрация результатов группировки (аналог WHERE для агрегатов)

Пример:

sql

Copy

Download

```sql
SELECT department, AVG(salary)
FROM employees
GROUP BY department
HAVING AVG(salary) > 50000
```

Этот конспект охватывает все ключевые темы лекции и поможет вам подготовиться к экзамену. Рекомендую обратить особое внимание на принципы реляционных БД, типы целостности и синтаксис SQL-запросов.

Лекция 2

1. Процесс проектирования БД

Уровни проектирования:

1. **Пользовательский уровень**:
    - Представление предметной области (учебный процесс, магазин и т.п.)
    - Обычный пользователь не знает о физическом размещении данных
2. **Инфологический уровень**:
    - ER-диаграммы (модель "Сущность-Связь")
    - Независимость от физического хранилища
3. **Даталогический уровень**:
    - Учет модели данных (реляционная, объектная и др.)
    - Появление технических деталей (типы данных)
4. **Физический уровень**:
    - Реализация средствами конкретной СУБД (например, SQL-код)

2. ER-моделирование (инфологический уровень)

Основные элементы:

- **Сущность** (Student, Group) - класс объектов
- **Атрибут** (student_id, name) - свойства сущности
- **Связь** - ассоциация между сущностями

Типы связей:

1. **Один-к-одному** (1:1) - например, студент и его зачётная книжка
2. **Один-ко-многим** (1:M) - группа и студенты
3. **Многие-ко-многим** (M:M) - студенты и экзамены

Классификация сущностей:

- Стержневые (базовые независимые)
- Ассоциативные (для связей M:M)
- Характеристические (уточняющие другие сущности)

3. Создание реляционной БД (физический уровень)

SQL-объекты:

- Таблицы
- Представления
- Процедуры
- Триггеры

Категории SQL-операторов:

1. **DDL (Data Definition Language)**:
    - CREATE, ALTER, DROP - создание/изменение/удаление объектов
2. **DML (Data Manipulation Language)**:
    - SELECT, INSERT, UPDATE, DELETE - работа с данными
3. **DCL (Data Control Language)**:
    - GRANT, REVOKE - управление правами

4. Работа с таблицами

Создание таблиц:

sql

Copy

Download

```sql
CREATE TABLE students (
    student_id integer PRIMARY KEY,
    name text NOT NULL,
    surname text NOT NULL,
    group_id integer REFERENCES groups(group_id)
);
```

Типы данных в PostgreSQL:

- Числовые: integer, smallint, bigint, numeric
- Символьные: varchar(n), char(n), text
- Логические: boolean
- Дата/время: date, time, timestamp, interval

Ограничения целостности:

1. **NOT NULL** - запрет NULL-значений
2. **CHECK** - проверка условий (например, CHECK (grade >= 2 AND grade <= 5))
3. **UNIQUE** - уникальность значений
4. **PRIMARY KEY** - первичный ключ (UNIQUE + NOT NULL)
5. **FOREIGN KEY** - внешний ключ для связей между таблицами

5. Реализация связей в реляционной модели

Связь 1:M:

sql

Copy

Download

```sql
CREATE TABLE groups (
    group_id integer PRIMARY KEY,
    group_name text
);

CREATE TABLE students (
    student_id integer PRIMARY KEY,
    name text,
    group_id integer REFERENCES groups(group_id)
);
```

Связь M:M (через ассоциативную таблицу):

sql

Copy

Download

```sql
CREATE TABLE students (
    student_id integer PRIMARY KEY,
    name text
);

CREATE TABLE exams (
    exam_id integer PRIMARY KEY,
    exam_name text
);

CREATE TABLE student_exams (
    student_id integer REFERENCES students(student_id),
    exam_id integer REFERENCES exams(exam_id),
    PRIMARY KEY (student_id, exam_id)
);
```

6. Дополнительные возможности PostgreSQL

Последовательности:

sql

Copy

Download

```sql
CREATE SEQUENCE student_seq START 1 INCREMENT 1;

CREATE TABLE students (
    student_id integer DEFAULT nextval('student_seq') PRIMARY KEY,
    name text
);
```

Тип SERIAL (автоинкремент):

sql

Copy

Download

```sql
CREATE TABLE students (
    student_id serial PRIMARY KEY,
    name text
);
```

Создание таблиц на основе существующих:

sql

Copy

Download

```sql
CREATE TABLE new_students (LIKE students);
```

7. Особенности работы с NULL

- NULL ≠ NULL (сравнение всегда дает NULL)
- Игнорируются в агрегатных функциях (кроме COUNT(*))
- Особенности сортировки (по умолчанию идут последними)
- Проверка: `IS NULL`, `IS NOT NULL`

Пример:

sql

Copy

Download

```sql
SELECT COUNT(*) FROM students WHERE group_id IS NULL;
```

Этот конспект охватывает все ключевые аспекты лекции по созданию баз данных, от проектирования до физической реализации в PostgreSQL. Особое внимание следует уделить ER-моделированию, типам связей и их реализации через ограничения целостности.

Лекция 3

1. Реляционное представление данных

Основные понятия:

- **Отношение (relation)** - двумерная таблица особого вида
- **Атрибуты** - столбцы таблицы с уникальными именами
- **Кортежи** - строки таблицы
- **Домен** - множество допустимых значений для атрибута

Характеристики отношений:

- **Степень отношения** - количество атрибутов (унарное, бинарное, n-арное)
- **Кардинальное число** - количество кортежей в отношении

2. Реляционная алгебра

Основные операции:

1. **Выборка (σ)**:
   - `σφ(R)` - выбирает кортежи из R, удовлетворяющие условию φ
   - Пример SQL: `SELECT * FROM students WHERE group = '3100'`
2. **Проекция (π)**:
   - `πattr(R)` - выбирает указанные атрибуты из R
   - Пример SQL: `SELECT name, group FROM students`
3. **Соединение (⋈)**:
   - `R ⋈θ S` - объединяет отношения R и S по условию θ
   - Пример SQL: `SELECT * FROM students JOIN exams ON students.id = exams.stud_id`

Свойства операций:

- Коммутативность: `R ⋈θ S ≡ S ⋈θ R`
- Ассоциативность: `R ⋈θ (S ⋈φ T) ≡ (R ⋈θ S) ⋈φ T`

3. Нормализация баз данных

Проблемы ненормализованных данных:

- **Аномалии вставки**: необходимость вводить избыточные данные
- **Аномалии обновления**: необходимость изменять данные в нескольких местах
- **Аномалии удаления**: потеря информации при удалении записей

Функциональные зависимости:

- `A → B` - атрибут B функционально зависит от атрибута A
- **Полная функциональная зависимость**: нельзя удалить ни один атрибут из детерминанта без потери зависимости
- **Транзитивная зависимость**: `A → B → C` (C транзитивно зависит от A через B)

4. Нормальные формы

Первая нормальная форма (1НФ):

- На пересечении каждой строки и столбца - ровно одно значение
- Нет повторяющихся групп данных

Пример нарушения:

text

Copy

Download

```
| STUDENT | EXAMS                |
|---------|----------------------|
| 1       | Math, Physics        |
| 2       | Chemistry            |
```

Вторая нормальная форма (2НФ):

- Отношение в 1НФ

- Все неключевые атрибуты находятся в полной функциональной зависимости от первичного ключа

Пример нарушения:

text

Copy
Download

```
| STUDENT_EXAMS |
| student_id | exam_id | exam_name | ... |
```

(экзамен зависит только от exam_id, а не от составного ключа)

Третья нормальная форма (3НФ):

- Отношение в 2НФ
- Нет транзитивных зависимостей неключевых атрибутов от первичного ключа

Пример нарушения:

text

Copy
Download

```
| STUDENT_EXAMS |
| student_id | exam_id | professor | professor_phone |
```

(телефон профессора зависит от профессора, а не напрямую от ключа)

Нормальная форма Бойса-Кодда (НФБК):

- Усиленная версия 3НФ
- Все детерминанты являются потенциальными ключами

5. Процесс нормализации

1. Приведение к 1НФ:
    - Устранение повторяющихся групп
    - Создание отдельных строк для каждого значения
2. Приведение к 2НФ:
    - Выделение частичных зависимостей в отдельные таблицы
    - Пример: отделение данных о студентах от данных об экзаменах

3. Приведение к 3НФ:
    - Устранение транзитивных зависимостей
    - Пример: выделение данных о преподавателях в отдельную таблицу

6. Денормализация

- Намеренное отступление от нормальных форм для повышения производительности
- Причины:
    - Уменьшение количества соединений таблиц
    - Ускорение выполнения часто используемых запросов
- Недостатки:
    - Увеличение избыточности данных
    - Усложнение поддержания целостности данных

Пример нормализации:

Исходная таблица (нарушает 1НФ):

text

Copy

Download

```
| STUDENT | EXAMS                |
|---------|----------------------|
| 1       | Math, Physics        |
| 2       | Chemistry            |
```

После приведения к 1НФ:

text

Copy

Download

```
| STUDENT | EXAM    |
|---------|---------|
| 1       | Math    |
| 1       | Physics |
| 2       | Chemistry |
```

После полной нормализации (3НФ):

text

Copy

Download

```
STUDENTS:           EXAMS:            STUDENT_EXAMS:
| id | name |      | id | name |     | student_id | exam_id |
|----|------|      |----|------|     |------------|---------|
| 1  | A    |      | 1  | Math |     | 1          | 1       |
| 2  | B    |      | 2  | Physics|   | 1          | 2       |
                   | 3  | Chemistry| | 2          | 3       |
```

Этот конспект охватывает ключевые аспекты нормализации реляционных баз данных, от основных понятий реляционной алгебры до практических шагов приведения базы данных к нормальным формам. Особое внимание следует уделить пониманию функциональных зависимостей и правилам преобразования между нормальными формами.

Лекция 4

1. Основы SQL-запросов

Структура SELECT-запроса:

sql

Copy

Download

```sql
SELECT поля
FROM таблицы
[WHERE условия]
[GROUP BY группировка]
[HAVING условия_агрегации]
[ORDER BY сортировка]
```

Логические операторы:

- `AND`, `OR`, `NOT` - коммутативны
- Результат: `TRUE`, `FALSE`, `NULL`

Операторы сравнения:

- `>`, `<`, `>=`, `<=`, `=`, `<>`/`!=`
- С NULL всегда возвращают NULL

Специальные предикаты:

- `BETWEEN x AND y` / `NOT BETWEEN`
- `IS NULL` / `IS NOT NULL`
- `IS DISTINCT FROM` (учитывает NULL)
- `LIKE` / `ILIKE` (регистронезависимый) для поиска по шаблону:
    - `_` - один символ
    - `%` - любое количество символов
- `SIMILAR TO` - расширенный LIKE
- Регулярные выражения POSIX: `~`, `~*`, `!~`, `!~*`

2. Условные выражения

CASE:

sql

Copy

Download

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

Пример:

sql

Copy

Download

```sql
SELECT Result,
  CASE
    WHEN Result >= 91 THEN 'Отл.'
    WHEN Result >= 75 THEN 'Хор.'
    WHEN Result >= 60 THEN 'Удовл.'
    ELSE 'Неуд.'
  END AS Grade
FROM EXAM;
```

Преобразование типов:

- `CAST(expr AS type)`
- `expr::type` (PostgreSQL)
- `type(expr)` через функцию

3. Соединения таблиц (JOIN)

Декартово произведение:

- Объединение всех строк из первой таблицы со всеми строками из второй
- Получается через `CROSS JOIN` или перечисление таблиц через запятую

Виды соединений:

1. **INNER JOIN** - только совпадающие строки
2. **LEFT JOIN** - все строки левой таблицы + совпадения справа (NULL если нет)
3. **RIGHT JOIN** - все строки правой таблицы + совпадения слева
4. **FULL JOIN** - все строки обеих таблиц (NULL где нет совпадений)
5. **NATURAL JOIN** - автоматически по одинаковым именам столбцов

Синтаксис:

sql

Copy

Download

```sql
SELECT ...
FROM t1 JOIN t2 ON t1.col = t2.col  -- явное условие
FROM t1 JOIN t2 USING (col)          -- по общему столбцу
FROM t1 NATURAL JOIN t2              -- по всем общим столбцам
```

4. Вложенные подзапросы

Простые подзапросы:

- Выполняются один раз перед основным запросом
- Пример с IN:

sql

Copy

Download

```sql
SELECT Surname
FROM STUDENT
WHERE City IN (
  SELECT City FROM CITIES
  WHERE Country = 'Россия'
);
```

Коррелированные подзапросы:

- Зависят от внешнего запроса

- Выполняются для каждой строки внешнего запроса
- Пример с EXISTS:

sql

Copy

Download

```sql
SELECT Surname
FROM STUDENT
WHERE EXISTS (
  SELECT 1 FROM EXAM
  WHERE StID = STUDENT.StudentID
  AND Result < 60
);
```

5. Операторы для подзапросов

IN:

- `expr IN (подзапрос)` - true если есть совпадение
- Эквивалентно `= ANY`

EXISTS:

- `EXISTS (подзапрос)` - true если подзапрос вернул строки
- Часто используется с коррелированными подзапросами

ANY/SOME:

- `expr op ANY (подзапрос)` - true если хотя бы одно сравнение верно
- `op` - оператор сравнения (=, >, < и т.д.)

ALL:

- `expr op ALL (подзапрос)` - true если все сравнения верны
- Пример - найти студентов с баллами выше всех в группе:

sql

Copy

Download

```sql
SELECT * FROM STUDENT
WHERE Score > ALL (
  SELECT Score FROM STUDENT
  WHERE GroupID = 'P3100'
);
```

6. Особенности работы с NULL

- Логические операции: NULL AND TRUE = NULL, NULL OR FALSE = NULL
- Сравнения: любое сравнение с NULL дает NULL
- IN: если в списке есть NULL и нет совпадений - результат NULL
- EXISTS: всегда возвращает TRUE/FALSE, даже для NULL
- ANY/ALL: NULL в результатах подзапроса могут приводить к NULL

Этот конспект охватывает ключевые аспекты работы с SQL-запросами, включая соединения таблиц, вложенные подзапросы и работу с NULL-значениями. Особое внимание следует уделить различиям между видами соединений и особенностям выполнения подзапросов.

Лекция 5

1. Операции манипуляции данными (DML)

INSERT - добавление данных:

sql

Copy

Download

```sql
INSERT INTO student VALUES (123, 'Vasya', 345);
INSERT INTO student (stud_id, name, group_id)
VALUES (123, 'Vasya', 345);
INSERT INTO student
SELECT id, name, 345 FROM tmp_person WHERE exam > 70;
```

UPDATE - изменение данных:

sql

Copy

Download

```sql
UPDATE student SET group_id = 578;  -- всем
UPDATE student SET group_id = 34 WHERE name = 'Ivan';  -- с условием
```

DELETE - удаление данных:

sql

Copy

Download

```sql
DELETE FROM student WHERE group_id = 34;  -- выборочное
```

```sql
DELETE FROM student;  -- все данные
TRUNCATE TABLE student;  -- быстрая очистка большой таблицы
TRUNCATE TABLE student, group, exam;  -- очистка нескольких таблиц
```

2. Представления (Views)

Обычные представления:

sql

Copy

Download

```sql
CREATE VIEW p3100_students AS
SELECT * FROM student WHERE group_id IN (
  SELECT group_id FROM group WHERE group_name LIKE 'P3100%'
);
```

Материализованные представления:

sql

Copy

Download

```sql
CREATE MATERIALIZED VIEW p3100_students_m AS
SELECT student_id, surname FROM student WHERE group_id IN (
  SELECT group_id FROM group WHERE group_name LIKE 'P3100%'
);

REFRESH MATERIALIZED VIEW p3100_students_m;  -- обновление данных
```

3. PL/pgSQL - процедурное расширение SQL

Создание функции:

sql

Copy

Download

```sql
CREATE OR REPLACE FUNCTION update_student_group(stud_id int, new_group int)
RETURNS void AS $$
BEGIN
  UPDATE student SET group_id = new_group WHERE student_id = stud_id;
END;
$$ LANGUAGE plpgsql;
```

Особенности синтаксиса:

- Блочная структура: DECLARE, BEGIN, END
- Поддержка переменных и вложенных блоков

- Использование $$ для экранирования строк
- Анонимные блоки через DO

Пример функции с переменными:

sql

Copy

Download

```sql
CREATE FUNCTION count_students() RETURNS integer AS $$
DECLARE
  student_count integer;
BEGIN
  SELECT COUNT(*) INTO student_count FROM student;
  RETURN student_count;
END;
$$ LANGUAGE plpgsql;
```

4. Триггеры

Создание триггера:

sql

Copy

Download

```sql
CREATE TRIGGER audit_employee
AFTER INSERT ON employee
FOR EACH ROW EXECUTE PROCEDURE audit_employee_func();

CREATE OR REPLACE FUNCTION audit_employee_func()
RETURNS TRIGGER AS $$
BEGIN
  INSERT INTO audit(emp_id, entry_date)
  VALUES (NEW.id, current_timestamp);
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Типы триггеров:

1. **По времени срабатывания**:
    - BEFORE - перед операцией
    - AFTER - после операции
    - INSTEAD OF - вместо операции (только для представлений)
2. **По уровню**:

    - ROW - для каждой строки
    - STATEMENT - для оператора (один раз)
3. **По событию**:
    - DML (INSERT, UPDATE, DELETE)
    - DDL (CREATE, ALTER, DROP)
    - Триггеры замещения

Специальные переменные в триггерах:

- NEW - новая строка (для INSERT/UPDATE)
- OLD - старая строка (для UPDATE/DELETE)
- TG_OP - операция ('INSERT', 'UPDATE', 'DELETE')
- TG_NAME - имя триггера

5. Транзакции

Основные принципы (ACID):

- **Атомарность** - все или ничего
- **Согласованность** - переход между согласованными состояниями
- **Изолированность** - параллельные транзакции не влияют друг на друга
- **Долговечность** - сохранение результатов после завершения

Управление транзакциями:

sql

Copy

Download

```sql
BEGIN;  -- начало транзакции
UPDATE account SET balance = balance - 50 WHERE name = 'Alex';
UPDATE account SET balance = balance + 50 WHERE name = 'Ivan';
COMMIT;  -- подтверждение изменений

-- или
ROLLBACK;  -- отмена изменений
```

Точки сохранения (SAVEPOINT):

sql

Copy

Download

```sql
BEGIN;
```

```sql
UPDATE account SET balance = balance - 50 WHERE name = 'Alex';
SAVEPOINT save1;
UPDATE account SET balance = balance + 50 WHERE name = 'Ivan';
-- если что-то пошло не так:
ROLLBACK TO save1;
UPDATE account SET balance = balance + 50 WHERE name = 'Ivan2';
COMMIT;
```

Этот конспект охватывает ключевые аспекты работы с PL/pgSQL, триггерами и транзакциями в PostgreSQL. Особое внимание следует уделить отличиям между видами триггеров и принципам ACID при работе с транзакциями.

Лекция 6

1. Логические операторы и NULL

- **AND**: Результат зависит от значений A и B. Например:
  - TRUE AND NULL → NULL
  - FALSE AND NULL → FALSE
- **OR**: Результат также зависит от комбинаций значений. Например:
  - TRUE OR NULL → TRUE
  - FALSE OR NULL → NULL
- **NOT**: Инвертирует значение:
  - NOT TRUE → FALSE
  - NOT NULL → NULL

2. Индексы в SQL

- **Определение**: Индекс — это структура данных, которая упорядочивает значения в столбцах таблицы для ускорения поиска.
- **Пример**: Для таблицы `STUDENT` с полями `StudID`, `GroupID`, `Name` индекс по `StudID` позволяет быстро находить строки с заданным `StudID` без полного перебора.

3. Преимущества индексов

- Ускорение операций:

- Фильтрация (`WHERE`).
- Соединения (`JOIN`).
- Агрегатные функции (`MIN`, `MAX`).
- Сортировка и группировка.

4. Недостатки индексов

- **Затраты памяти**: Индексы занимают дополнительное пространство.
- **Замедление операций изменения данных**: При вставке, обновлении или удалении строк индексы требуют обновления.
- **Неэффективность для малых таблиц**: Для таблиц с небольшим количеством строк индексы могут не давать преимуществ.
- **Неэффективность при выборке больших объемов данных**: Если условие выбирает значительную часть таблицы, индекс может не использоваться.

5. Стратегии применения индексов

- Анализ частоты операций: запросы vs обновления.
- Выбор столбцов для индексации: часто используемые в условиях `WHERE` или `JOIN`.
- Учет специфики запросов.

6. Создание индексов

- **Синтаксис**:

  sql

  Copy

  Download

  ```sql
  CREATE INDEX index_name ON table_name (column_name);
  CREATE INDEX index_name ON table_name (column1, column2);
  ```

- **Типы индексов**:
  - **B-tree**: Подходит для операций сравнения (`=`, `>`, `<`, `BETWEEN`).
  - **Hash-index**: Эффективен для точных совпадений (`=`), но не поддерживает диапазонные запросы.

    - **GIST/GIN**: Специализированные индексы (например, для полнотекстового поиска).

7. B-tree индекс

- **Структура**: Сбалансированное дерево с отсортированными ключами.
- **Преимущества**:
    - Минимизация количества переходов при поиске.
    - Поддержка диапазонных запросов.
- **Пример**: Поиск `StudID = 85` в таблице `STUDENT` с использованием B-tree индекса.

8. Hash-index

- **Принцип работы**: Использует хэш-функцию для преобразования ключей в уникальные значения.
- **Применение**: Только для точных совпадений (`attr = value`).

9. Индексы в PostgreSQL

- **GIST**: Обобщенное дерево поиска.
- **GIN**: Обобщенный инвертированный индекс (для полнотекстового поиска).

10. Литература

- В.В. Кириллов, Г.Ю. Громов. "Введение в реляционные базы данных".
- Документация PostgreSQL.

Ключевые выводы:

1. Индексы ускоряют поиск, но требуют дополнительных ресурсов.
2. Выбор типа индекса зависит от типов запросов (B-tree для диапазонов, Hash для точных совпадений).
3. Индексы не всегда полезны: их эффективность зависит от размера таблицы и характера операций.

Лекция 7

1. Основные этапы выполнения запросов

1. **Разбор запроса (Parser)**:
    - Построение дерева запроса.
2. **Преобразование запроса (Rewritter)**:
    - Оптимизация структуры запроса.
3. **Планировщик и оптимизатор (Planner)**:
    - Генерация возможных планов выполнения.
    - Выбор наиболее эффективного плана на основе оценочной стоимости.
4. **Исполнение (Executor)**:
    - Выполнение выбранного плана.

2. План выполнения запроса

- **Цель**: Построить программу, которая эффективно выполнит SQL-запрос.
- **Критерии выбора плана**:
    - Число операций ввода-вывода.
    - Время выполнения.
    - Использование индексов.

3. Операции реляционной алгебры

- **Выборка (σ)**:
    - Пример: `σ_{GROUP='3100' ∧ ID≥150000}(STUDENTS)`.
- **Проекция (π)**:
    - Пример: `π_{name, group}(STUDENTS)`.
- **Соединение (⋈)**:
    - Пример: `STUDENTS ⋈_{ID=STUD_ID} EXAMS`.

4. Законы оптимизации запросов

1. **Коммутативность соединения**:
   `R ⋈_θ S ≡ S ⋈_θ R`.

2. **Ассоциативность соединения**:
   $R \bowtie_\theta (S \bowtie_\varphi T) \equiv (R \bowtie_\theta S) \bowtie_\varphi T$.
3. **Дистрибутивность выборки**:
   $\sigma_{\theta \wedge \varphi}(R) \equiv \sigma_\theta(\sigma_\varphi(R))$.
4. **Пуш-даун предикатов**:
   $\sigma_\varphi(R \bowtie_\theta S) \equiv \sigma_\varphi(R) \bowtie_\theta S$, если φ относится к атрибутам R.

5. Материализация и конвейерная обработка

- **Материализация**:
  - Сохранение промежуточных результатов.
  - Увеличивает время выполнения из-за операций записи/чтения.
- **Конвейерная обработка**:
  - Передача данных между операциями без материализации.
  - Эффективна для левосторонних деревьев планов.

6. Типы соединений (JOIN)

1. **Nested Loop Join**:
   - Для каждой строки внешней таблицы ищет совпадения во внутренней.
   - Эффективен, если внешняя таблица мала.
2. **Block Nested Loop Join**:
   - Обработка блоков строк для уменьшения числа чтений.
3. **Sort-Merge Join**:
   - Требует предварительной сортировки таблиц по ключам соединения.
   - Эффективен для больших таблиц.
4. **Hash Join**:
   - Использует хэш-таблицу для поиска совпадений.
   - Подходит для равенства ($=$).

7. Индексы в планах выполнения

- **Использование**:
  - Ускоряют выборку и соединения.
- **Недостатки**:

- Затраты на обновление.
- Неэффективны для больших объемов данных или малых таблиц.

8. Просмотр плана выполнения в PostgreSQL

- **EXPLAIN**:
  - Показывает план без выполнения запроса.
  - Пример:
    sql

    Copy

    Download

    ```sql
    EXPLAIN SELECT * FROM STUDENTS WHERE StudID = 942;
    ```
- **EXPLAIN ANALYZE**:
  - Выполняет запрос и показывает фактический план и затраты.
  - Пример:
    sql

    Copy

    Download

    ```sql
    EXPLAIN ANALYZE SELECT * FROM STUDENTS WHERE StudID = 942;
    ```

9. Советы по оптимизации

1. **Ранняя фильтрация**:
   - Применять условия `WHERE` как можно раньше.
2. **Проекция на ранних этапах**:
   - Уменьшает объем обрабатываемых данных.
3. **Использование конвейера**:
   - Избегать материализации промежуточных результатов.
4. **Правильное соединение таблиц**:
   - Выбирать метод соединения в зависимости от размера таблиц и наличия индексов.

10. Литература

- В.В. Кириллов, Г.Ю. Громов. "Введение в реляционные базы данных".
- Документация PostgreSQL.

Ключевые выводы:

1. Оптимизация запросов основана на выборе эффективного плана выполнения.
2. Использование индексов и правильных методов соединения значительно ускоряет обработку данных.
3. Конвейерная обработка минимизирует накладные расходы на материализацию.
4. Инструменты `EXPLAIN` и `EXPLAIN ANALYZE` помогают анализировать и улучшать производительность запросов.

Вопросы с лаб

1. Архитектура ANSI-SPARC
2. Модель "Сущность-Связь". Классификация сущностей. Виды связей. Ограничения целостности.
3. DDL
4. DML
5. SQL
6. Соединение таблиц
7. Подзапросы
8. Представления
9. Последовательности
10. Нормализация. Формы
11. Функциональные зависимости. Виды
12. Денормализация
13. Язык PL/pgSQL
14. Индексы
15. Оптимизация запросов
16. Выбор плана выполнения запросов

1. Архитектура ANSI-SPARC

ANSI-SPARC (American National Standards Institute – Standards Planning And Requirements Committee) — это трехуровневая архитектура СУБД, которая разделяет представление данных на:

1. **Внешний уровень (External Level)**
    - Пользовательские представления (Views).
    - Каждый пользователь видит только те данные, которые ему нужны.
    - Пример: `CREATE VIEW` в PostgreSQL.
2. **Концептуальный уровень (Conceptual Level)**
    - Логическая структура всей БД (таблицы, связи, ограничения).
    - Определяется схемой (`CREATE SCHEMA`).
    - Независим от физического хранения.
3. **Внутренний уровень (Internal Level)**
    - Физическое хранение данных (индексы, файлы, страницы).
    - Оптимизация запросов и управление памятью.

Преимущества:

- Независимость данных (логическая и физическая).
- Безопасность (разграничение прав).

---

2. Модель "Сущность-Связь" (ER-модель)

Используется на этапе проектирования БД.

Сущности (Entities)

- Объекты реального мира (например, `Студент`, `Группа`).
- **Классификация:**
    - Сильные (не зависят от других, имеют PK).
    - Слабые (зависят от сильных, нет своего PK).

Связи (Relationships)

- Отношения между сущностями (`Студент учится в Группе`).
- **Виды связей:**
  - `1:1` (Один к одному) – например, человек и паспорт.
  - `1:N` (Один ко многим) – группа и студенты.
  - `M:N` (Многие ко многим) – студенты и курсы (через связующую таблицу).

Ограничения целостности

- **Первичный ключ (PRIMARY KEY)** – уникальность + NOT NULL.
- **Внешний ключ (FOREIGN KEY)** – ссылочная целостность.
- **UNIQUE** – уникальность значений.
- **CHECK** – проверка условий (`CHECK (age > 0)`).

---

3. DDL (Data Definition Language)

Язык определения данных:

sql

Copy

Download

```sql
-- Создание таблицы
CREATE TABLE Students (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    group_id INT REFERENCES Groups(id)
);

-- Изменение таблицы
ALTER TABLE Students ADD COLUMN email VARCHAR(100);

-- Удаление таблицы
DROP TABLE Students;
```

Основные команды:

- `CREATE` (создание объектов).
- `ALTER` (изменение структуры).
- `DROP` (удаление объектов).

---

4. DML (Data Manipulation Language)

Язык манипулирования данными:

sql

Copy

Download

```sql
-- Вставка данных
INSERT INTO Students (name, group_id) VALUES ('Иван', 1);

-- Выборка данных
SELECT * FROM Students WHERE group_id = 1;

-- Обновление данных
UPDATE Students SET name = 'Петр' WHERE id = 1;

-- Удаление данных
DELETE FROM Students WHERE id = 1;
```

Основные команды:

- `SELECT` – выборка.
- `INSERT` – добавление.
- `UPDATE` – обновление.
- `DELETE` – удаление.

1. SQL (Structured Query Language)

SQL — язык структурированных запросов для работы с БД. Включает:

- **DDL** (определение данных, напр. `CREATE TABLE`)
- **DML** (манипуляция данными, напр. `SELECT`, `INSERT`)
- **DCL** (управление доступом, напр. `GRANT`, `REVOKE`)
- **TCL** (управление транзакциями, напр. `COMMIT`, `ROLLBACK`)

Примеры:

sql

Copy

Download

```sql
-- Создание таблицы (DDL)
CREATE TABLE users (id SERIAL PRIMARY KEY, name TEXT);
```

```sql
-- Вставка данных (DML)
INSERT INTO users (name) VALUES ('Alice'), ('Bob');

-- Выборка данных (DML)
SELECT * FROM users WHERE name LIKE 'A%';
```

---

2. Соединение таблиц (JOIN)

Используется для выборки данных из нескольких таблиц по условию связи.

Типы JOIN:

Тип JOIN	Описание
INNER JOIN	Возвращает только совпадающие строки (`A ∩ B`)
LEFT JOIN	Все строки из левой таблицы + совпадения справа (если нет — `NULL`)
RIGHT JOIN	Все строки из правой таблицы + совпадения слева (если нет — `NULL`)
FULL JOIN	Все строки из обеих таблиц (`A ∪ B`), отсутствующие — `NULL`
CROSS JOIN	Декартово произведение (все возможные комбинации)

Пример:

sql

Copy

Download

```sql
SELECT u.name, o.order_date
FROM users u
INNER JOIN orders o ON u.id = o.user_id;
```

---

3. Подзапросы (Subqueries)

Запрос внутри другого запроса (в `SELECT`, `FROM`, `WHERE` и др.).

Виды подзапросов:

1. **В WHERE:**

   sql

   Copy

   Download

   ```sql
   SELECT * FROM products
   WHERE price > (SELECT AVG(price) FROM products);
   ```

2. **В FROM (производные таблицы):**
   sql

   Copy

   Download

   ```sql
   SELECT avg_price.category, avg_price.avg
   FROM (SELECT category, AVG(price) AS avg FROM products GROUP BY categor
   y) AS avg_price;
   ```

3. **В SELECT (скалярные подзапросы):**
   sql

   Copy

   Download

   ```sql
   SELECT name, (SELECT COUNT(*) FROM orders WHERE user_id = u.id) AS orde
   r_count
   FROM users u;
   ```

Ключевые операторы: `IN`, `EXISTS`, `ANY`, `ALL`.

---

4. Представления (Views)

Виртуальные таблицы, основанные на результате SQL-запроса.

Преимущества:

- Упрощение сложных запросов.
- Ограничение доступа (можно давать права на `VIEW`, но не на таблицу).
- Абстракция данных.

Пример:

sql

Copy

Download

```sql
-- Создание представления
CREATE VIEW active_users AS
SELECT * FROM users WHERE last_login > CURRENT_DATE - INTERVAL '30 days';

-- Использование
SELECT * FROM active_users;
```

Материализованные представления (`MATERIALIZED VIEW`) — хранят результат на диске (обновляются по `REFRESH`).

---

5. Последовательности (Sequences)

Генератор уникальных чисел (часто используется для `SERIAL`/`IDENTITY`).

Основные команды:

sql

Copy

Download

```sql
-- Создание последовательности
CREATE SEQUENCE user_id_seq START WITH 1 INCREMENT BY 1;

-- Использование в таблице
CREATE TABLE users (
    id INT DEFAULT nextval('user_id_seq') PRIMARY KEY,
    name TEXT
);

-- Текущее значение
SELECT currval('user_id_seq');

-- Сброс последовательности
ALTER SEQUENCE user_id_seq RESTART WITH 1;
```

Аналог в PostgreSQL:

sql

Copy

Download

```sql
-- Автоинкремент через SERIAL
CREATE TABLE users (id SERIAL PRIMARY KEY, name TEXT);
-- (это скрытая последовательность + DEFAULT nextval)
```

1. Нормализация. Формы нормализации

Нормализация — процесс устранения избыточности данных и минимизации аномалий при вставке, обновлении и удалении.

Основные нормальные формы:

Форма	Описание	Пример нарушения	Исправление
1NF (Первая)	Все атрибуты атомарны (неделимы), нет повторяющихся групп.	Столбец `Телефоны: "123, 456"`	Разделить на отдельные строки или столбцы (`phone1`, `phone2`).
2NF (Вторая)	Выполнена 1NF + нет частичных зависимостей от составного ключа.	В таблице `Заказ(OrderID, ProductID, ProductName, Quantity)` `ProductName` зависит только от `ProductID`, а не от всего ключа `(OrderID, ProductID)`.	Вынести `ProductName` в отдельную таблицу `Products`.
3NF (Третья)	Выполнена 2NF + нет транзитивных зависимостей (неключевые атрибуты не зависят друг от друга).	В таблице `Студент(StudentID, GroupID, GroupName)` `GroupName` зависит от `GroupID`, а не напрямую от `StudentID`.	Вынести `GroupID`, `GroupName` в таблицу `Groups`.
BCNF (Бойса-Кодда)	Усиленная 3NF: каждая детерминанта (левая часть функциональной зависимости) — потенциальный ключ.	В таблице `Преподаватель-Курс(LecturerID, CourseID, Department)` предполагается, что преподаватель ведет курс в одном отделе, но если он может вести курс в разных отделах, это нарушает BCNF.	Разделить на две таблицы: `(LecturerID, CourseID)` и `(CourseID, Department)`.

Цель нормализации:

- Уменьшение дублирования данных.
- Устранение аномалий (вставки, обновления, удаления).

---

2. Функциональные зависимости. Виды

Функциональная зависимость (ФЗ) — отношение вида `X → Y`, где значение `X` однозначно определяет `Y`.

Виды функциональных зависимостей:

1. **Полная функциональная зависимость**
   - `Y` зависит от всего ключа `X`, а не от его части.
   - Пример: В `OrderDetails(OrderID, ProductID → Quantity)` `Quantity` зависит от всей пары `(OrderID, ProductID)`.
2. **Частичная зависимость** (нарушает 2NF)
   - `Y` зависит от части составного ключа.
   - Пример: В `Orders(OrderID, CustomerID → CustomerName)` `CustomerName` зависит только от `CustomerID`.
3. **Транзитивная зависимость** (нарушает 3NF)
   - `X → Y` и `Y → Z`, значит `X → Z`, но `Z` неключевой атрибут.
   - Пример: В `Employee(EmployeeID → DepartmentID → DepartmentName)` `DepartmentName` транзитивно зависит от `EmployeeID`.

---

3. Денормализация

Денормализация — намеренное нарушение нормальных форм для повышения производительности.

Когда применяется?

- Частые `JOIN` замедляют запросы.
- В отчетах и аналитических системах (OLAP).

- В кэшируемых данных (например, `total_orders` в таблице `users`).

Пример денормализации:

sql

Copy

Download

```sql
-- Нормализованная структура
CREATE TABLE orders (id INT, user_id INT, amount DECIMAL);
CREATE TABLE users (id INT, name TEXT);

-- Денормализованная (добавляем total_spent в users)
ALTER TABLE users ADD COLUMN total_spent DECIMAL;
UPDATE users u
SET total_spent = (SELECT SUM(amount) FROM orders WHERE user_id = u.id);
```

Плюсы:

- Ускорение `SELECT` (не нужен `JOIN`).

  Минусы:

- Аномалии при изменении (`UPDATE orders` требует пересчета `total_spent`).

---

4. Язык PL/pgSQL

PL/pgSQL — процедурный язык PostgreSQL для написания хранимых процедур, функций и триггеров.

Основные элементы:

1. **Структура функции:**

   sql

   Copy

   Download

   ```sql
   CREATE OR REPLACE FUNCTION get_user_orders(user_id INT)
   RETURNS TABLE (order_id INT, amount DECIMAL) AS $$
   BEGIN
       RETURN QUERY
       SELECT id, amount FROM orders WHERE user_id = user_id;
   ```

```
END;
$$ LANGUAGE plpgsql;
```

2. **Условные операторы (`IF`, `CASE`):**
   sql

   Copy

   Download

```
IF user_age >= 18 THEN
    RETURN 'Взрослый';
ELSE
    RETURN 'Ребенок';
END IF;
```

3. **Циклы (`LOOP`, `FOR`, `WHILE`):**
   sql

   Copy

   Download

```
FOR i IN 1..10 LOOP
    RAISE NOTICE 'i = %', i;
END LOOP;
```

4. **Обработка исключений (`EXCEPTION`):**
   sql

   Copy

   Download

```
BEGIN
    INSERT INTO users (id, name) VALUES (1, 'Alice');
EXCEPTION WHEN unique_violation THEN
    RAISE NOTICE 'Пользователь уже существует!';
END;
```

5. **Триггеры:**
   sql

   Copy

   Download

```
CREATE OR REPLACE FUNCTION update_total_spent()
RETURNS TRIGGER AS $$
BEGIN
```

```
    UPDATE users
    SET total_spent = (SELECT SUM(amount) FROM orders WHERE user_id = N
EW.user_id)
    WHERE id = NEW.user_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_total
AFTER INSERT OR UPDATE OR DELETE ON orders
FOR EACH ROW EXECUTE FUNCTION update_total_spent();
```

Когда использовать PL/pgSQL?

- Сложная бизнес-логика в БД.
- Триггеры для поддержания целостности.
- Оптимизация запросов (кеширование в функциях).

1. Индексы в PostgreSQL

Основные типы индексов

1. **B-дерево (B-tree)** - стандартный индекс, подходит для большинства операций:

   sql

   Copy

   Download

   ```
   CREATE INDEX idx_users_name ON users(name);
   ```

   - Оптимален для: `=`, `>`, `<`, `BETWEEN`, `ORDER BY`
   - Поддерживает уникальность (`CREATE UNIQUE INDEX`)
2. **Хеш-индекс (Hash)** - только для операций равенства:
   sql

   Copy

   Download

   ```
   CREATE INDEX idx_users_id_hash ON users USING HASH(id);
   ```

- Быстрее B-tree для `=`, но не поддерживает диапазонные запросы

3. **GiST (Generalized Search Tree)** - для сложных типов данных:
   sql

   Copy

   Download

   ```
   CREATE INDEX idx_geo ON locations USING GIST(geom);
   ```

   - Подходит для: геоданных, полнотекстового поиска, IP-адресов
4. **GIN (Generalized Inverted Index)** - для составных значений:
   sql

   Copy

   Download

   ```
   CREATE INDEX idx_tags ON articles USING GIN(tags);
   ```

   - Оптимален для: массивов, JSON, полнотекстового поиска
5. **BRIN (Block Range INdexes)** - для очень больших таблиц:
   sql

   Copy

   Download

   ```
   CREATE INDEX idx_sensor_data ON measurements USING BRIN(timestamp);
   ```

   - Эффективен для данных с естественной сортировкой (например, временные ряды)

Когда создавать индексы

- На столбцах, часто используемых в `WHERE`
- На столбцах соединения (`JOIN`)
- На столбцах в `ORDER BY`, `GROUP BY`
- На столбцах с ограничениями уникальности

Анализ использования индексов

sql

Copy

Download

```
-- Показать существующие индексы
SELECT * FROM pg_indexes WHERE tablename = 'users';

-- Проверить использование индекса
EXPLAIN ANALYZE SELECT * FROM users WHERE name = 'Иван';
```

2. Оптимизация запросов

Основные методы оптимизации

1. **Анализ запросов**:

   sql

   Copy

   Download

   ```
   EXPLAIN ANALYZE SELECT * FROM orders WHERE user_id = 100;
   ```

2. **Оптимизация JOIN**:
   - Использовать индексы для столбцов соединения
   - Ограничивать размер соединяемых таблиц с помощью WHERE
3. **Оптимизация подзапросов**:
   - Замена подзапросов на JOIN там, где это возможно
   - Использование CTE (Common Table Expressions) для сложных запросов
4. **Оптимизация агрегации**:

   sql

   Copy

   Download

   ```
   -- Неоптимально
   SELECT user_id, COUNT(*) FROM orders GROUP BY user_id HAVING COUNT(*) >
   5;

   -- Оптимальнее
   WITH user_orders AS (
     SELECT user_id, COUNT(*) as order_count
     FROM orders
     GROUP BY user_id
   )
   SELECT * FROM user_orders WHERE order_count > 5;
   ```

5. **Использование частичных индексов**:
   sql

   Copy

   Download

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

3. Выбор плана выполнения запросов

Как работает планировщик PostgreSQL

1. **Этапы планирования**:
   - Парсинг SQL-запроса
   - Трансформация запроса
   - Генерация возможных планов выполнения
   - Оценка стоимости каждого плана
   - Выбор оптимального плана
2. **Факторы, влияющие на выбор плана**:
   - Статистика таблиц (`ANALYZE table_name`)
   - Настройки стоимости в postgresql.conf
   - Доступные индексы
   - Размеры таблиц
3. **Анализ плана выполнения**:
   sql

   Copy

   Download

```sql
EXPLAIN SELECT * FROM users WHERE id = 100;

-- Пример вывода:
-- Index Scan using users_pkey on users  (cost=0.15..8.17 rows=1 width=72
)
--   Index Cond: (id = 100)
```

4. **Типы операций в плане выполнения**:
   - **Seq Scan** - полное сканирование таблицы
   - **Index Scan** - сканирование по индексу

- **Bitmap Heap Scan** - комбинация индекса и таблицы
- **Hash Join/Nested Loop/Merge Join** - методы соединения таблиц

5. **Управление планировщиком**:
   sql

   Copy

   Download

```sql
-- Принудительное отключение индекса
SET enable_indexscan = off;

-- Установка стоимости операций
SET random_page_cost = 1.1;
```

Практические советы по оптимизации

1. **Обновляйте статистику**:

   sql

   Copy

   Download

```sql
ANALYZE table_name;
```

2. **Используйте покрывающие индексы**:
   sql

   Copy

   Download

```sql
CREATE INDEX idx_covering ON orders(user_id, status) INCLUDE (amount);
```

3. **Оптимизируйте работу с большими таблицами**:
   - Разбивайте на партиции
   - Используйте BRIN-индексы
   - Рассмотрите материализованные представления
4. **Настройте параметры PostgreSQL**:
   - `work_mem` - для операций сортировки и хеширования
   - `maintenance_work_mem` - для операций обслуживания
   - `random_page_cost` - для SSD-дисков

Вариант 1

1. На основании предметной области из варианта продумать инфологическую модель (минимум **6 сущностей**, требуется **связь многие-ко-многим**). Построить **даталогическую модель**. Описать ограничения для обеспечения целостности БД / Create ER-model: **6 Entities, Many-To-Many** relationship required. **Draw datalogical model**, describe constraints.
2. На основе даталогической модели из п.1 напишите **SQL-запрос**, который использует сущности, соединенные связью многие-ко-многим через необходимые соединения (**INNER JOIN**) / Based on the datalogical model from task **1** write **SQL-query** that uses entities connected with Many-To-Many relationship using **INNER JOIN**.
3. Для запроса из п.2 изобразите эффективный **план выполнения запроса** (можно добавить индексы). Для использованных соединений (INNER JOIN) укажите **алгоритм**. **Поясните**, почему был выбран именно такой план / For query from task №2 draw **effective query** execution plan (required indexes can be added). Specify **algorithms** used for INNER JOIN-operations. **Describe** your solution.

1. Инфологическая модель (ER-диаграмма)

Сущности:

1. **Студент** (Student) - информация о проживающих студентах
2. **Комната** (Room) - информация о комнатах общежития
3. **Блок** (Block) - блок общежития (этаж/секция)
4. **Оборудование** (Equipment) - мебель и техника в комнатах
5. **Поселение** (CheckIn) - история заселений студентов (связь M:N между Студент и Комната)
6. **Ответственный** (Staff) - сотрудники общежития

Связи:

- Студент M:N Комната (через сущность Поселение)
- Комната 1:M Оборудование
- Блок 1:M Комната
- Ответственный 1:M Блок

2. Даталогическая модель (SQL DDL)

sql

Copy

Download

```sql
CREATE TABLE Block (
    block_id SERIAL PRIMARY KEY,
    block_number VARCHAR(10) NOT NULL,
    floor INTEGER NOT NULL,
    staff_id INTEGER NOT NULL
);

CREATE TABLE Staff (
    staff_id SERIAL PRIMARY KEY,
    full_name VARCHAR(100) NOT NULL,
    position VARCHAR(50) NOT NULL,
    phone VARCHAR(20)
);

CREATE TABLE Room (
    room_id SERIAL PRIMARY KEY,
    room_number VARCHAR(10) NOT NULL,
    capacity INTEGER NOT NULL,
    block_id INTEGER NOT NULL,
    FOREIGN KEY (block_id) REFERENCES Block(block_id)
);

CREATE TABLE Student (
    student_id SERIAL PRIMARY KEY,
    full_name VARCHAR(100) NOT NULL,
    faculty VARCHAR(50) NOT NULL,
    course INTEGER NOT NULL,
    phone VARCHAR(20)
);

CREATE TABLE Equipment (
    equipment_id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    inventory_number VARCHAR(20) UNIQUE,
    room_id INTEGER NOT NULL,
    FOREIGN KEY (room_id) REFERENCES Room(room_id)
);

CREATE TABLE CheckIn (
    checkin_id SERIAL PRIMARY KEY,
    student_id INTEGER NOT NULL,
    room_id INTEGER NOT NULL,
```

```
    checkin_date DATE NOT NULL,
    checkout_date DATE,
    FOREIGN KEY (student_id) REFERENCES Student(student_id),
    FOREIGN KEY (room_id) REFERENCES Room(room_id),
    CONSTRAINT valid_dates CHECK (checkout_date IS NULL OR checkout_date > ch
eckin_date)
);

-- Добавляем внешний ключ для Block после создания Staff
ALTER TABLE Block ADD CONSTRAINT fk_staff
    FOREIGN KEY (staff_id) REFERENCES Staff(staff_id);
```

3. SQL-запрос с соединением M:N

sql

Copy

Download

```
SELECT
    s.full_name AS student_name,
    s.faculty,
    r.room_number,
    b.block_number,
    b.floor,
    st.full_name AS staff_name
FROM
    Student s
INNER JOIN CheckIn ci ON s.student_id = ci.student_id
INNER JOIN Room r ON ci.room_id = r.room_id
INNER JOIN Block b ON r.block_id = b.block_id
INNER JOIN Staff st ON b.staff_id = st.staff_id
WHERE
    ci.checkout_date IS NULL  -- Только текущие проживающие
ORDER BY
    b.block_number, r.room_number;
```

4. План выполнения запроса и оптимизация

Оптимальный план выполнения:

text

Copy

Download

```
QUERY PLAN
Sort  (cost=287.15..289.65 rows=1000 width=144)
  Sort Key: b.block_number, r.room_number
  ->  Hash Join  (cost=101.50..212.50 rows=1000 width=144)
        Hash Cond: (b.block_id = r.block_id)
```

```
        ->  Hash Join  (cost=50.75..126.75 rows=1000 width=88)
              Hash Cond: (b.staff_id = st.staff_id)
              ->  Seq Scan on block b  (cost=0.00..25.00 rows=1000 width=44)
              ->  Hash  (cost=30.00..30.00 rows=1000 width=44)
                    ->  Seq Scan on staff st  (cost=0.00..30.00 rows=1000 wid
th=44)
        ->  Hash  (cost=45.75..45.75 rows=1000 width=60)
              ->  Hash Join  (cost=25.25..45.75 rows=1000 width=60)
                    Hash Cond: (r.room_id = ci.room_id)
                    ->  Seq Scan on room r  (cost=0.00..15.00 rows=1000 width
=36)
                    ->  Hash  (cost=20.25..20.25 rows=1000 width=24)
                          ->  Hash Join  (cost=10.00..20.25 rows=1000 width=2
4)
                                Hash Cond: (ci.student_id = s.student_id)
                                ->  Seq Scan on checkin ci  (cost=0.00..5.00
rows=1000 width=8)
                                      Filter: (checkout_date IS NULL)
                                ->  Hash  (cost=7.50..7.50 rows=1000 width=16
)
                                      ->  Seq Scan on student s  (cost=0.00..
7.50 rows=1000 width=16)
```

Рекомендуемые индексы для оптимизации:

sql

Copy

Download

```sql
-- Для соединений
CREATE INDEX idx_checkin_student ON CheckIn(student_id);
CREATE INDEX idx_checkin_room ON CheckIn(room_id);
CREATE INDEX idx_room_block ON Room(block_id);
CREATE INDEX idx_block_staff ON Block(staff_id);

-- Для условия WHERE
CREATE INDEX idx_checkin_dates ON CheckIn(checkout_date) WHERE checkout_date
IS NULL;

-- Для сортировки
CREATE INDEX idx_block_room ON Block(block_number, floor);
```

Алгоритмы соединений и обоснование:

1. **Hash Join** для соединения CheckIn-Student и CheckIn-Room:
    - Выбран, так как ожидается большое количество строк
    - Эффективен для равенства при отсутствии сортировки

    - PostgreSQL автоматически выбирает Hash Join при достаточном work_mem
2. **Hash Join** для соединения Room-Block и Block-Staff:
    - Аналогичное обоснование - эффективность при соединении по равенству
    - Позволяет обрабатывать данные в памяти без сортировки
3. **Sort** для финального ORDER BY:
    - Необходим для выполнения сортировки по block_number и room_number
    - Можно было бы использовать Index Scan, если бы индексы покрывали все поля

Обоснование выбора плана:

- План использует наиболее эффективные методы соединения для данного случая
- Hash Join предпочтительнее Nested Loop из-за ожидаемого размера таблиц
- Seq Scan выбран для небольших таблиц (Staff, Block) - полное сканирование быстрее
- Индексы ускоряют поиск по условиям и соединениям

Вариант 2

1. На основании предметной области из варианта продумать инфологическую модель (минимум **6 сущностей**, требуется **связь многие-ко-многим**). Построить **даталогическую модель**. Описать ограничения для обеспечения целостности БД / Create ER-model: **6 Entities, Many-To-Many** relationship required. **Draw datalogical model**, describe constraints.
2. Предложите **триггеры** для **разработанной модели**, опишите виды триггеров / Suggest triggers for the model from task 1, describe trigger types.
3. На основе даталогической модели из п.1 напишите **SQL-запрос**, который использует сущности, соединенные связью многие-ко-многим через необходимые соединения (**INNER JOIN**) / Based on the datalogical model from task **1** write **SQL-query** that uses entities connected with Many-To-Many relationship using **INNER JOIN**.

4. Для запроса из п.3 изобразите эффективный **план выполнения запроса** (можно добавить индексы). Для использованных соединений (INNER JOIN) укажите **алгоритм**. **Поясните**, почему был выбран именно такой план / For query from task №3 draw **effective query** execution plan (required indexes can be added). Specify **algorithms** used for INNER JOIN-operations. **Describe** your solution.

1. Инфологическая модель (ER-диаграмма)

Сущности:

1. **Клиент** (Client) - информация о заказчиках
2. **Изделие** (Product) - виды изделий (платье, костюм и т.д.)
3. **Материал** (Material) - ткани и фурнитура
4. **Заказ** (Order) - информация о заказах (связь M:N между Клиент и Изделие)
5. **Сотрудник** (Employee) - работники ателье
6. **Поставщик** (Supplier) - поставщики материалов

Связи:

- Клиент M:N Изделие (через сущность Заказ)
- Изделие M:N Материал (через дополнительную таблицу ProductMaterial)
- Сотрудник 1:M Заказ
- Поставщик 1:M Материал

2. Даталогическая модель (SQL DDL) с ограничениями целостности

sql

Copy

Download

```sql
CREATE TABLE Supplier (
    supplier_id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    contact_phone VARCHAR(20) NOT NULL,
    address TEXT
);

CREATE TABLE Material (
    material_id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
```

```sql
    type VARCHAR(30) NOT NULL,
    price_per_unit DECIMAL(10,2) NOT NULL CHECK (price_per_unit > 0),
    quantity_in_stock INTEGER NOT NULL CHECK (quantity_in_stock >= 0),
    supplier_id INTEGER NOT NULL,
    FOREIGN KEY (supplier_id) REFERENCES Supplier(supplier_id)
);

CREATE TABLE Employee (
    employee_id SERIAL PRIMARY KEY,
    full_name VARCHAR(100) NOT NULL,
    position VARCHAR(50) NOT NULL,
    hire_date DATE NOT NULL CHECK (hire_date <= CURRENT_DATE),
    salary DECIMAL(10,2) CHECK (salary > 0)
);

CREATE TABLE Client (
    client_id SERIAL PRIMARY KEY,
    full_name VARCHAR(100) NOT NULL,
    phone VARCHAR(20) NOT NULL UNIQUE,
    address TEXT,
    discount DECIMAL(5,2) DEFAULT 0 CHECK (discount >= 0 AND discount <= 50)
);

CREATE TABLE Product (
    product_id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    base_price DECIMAL(10,2) NOT NULL CHECK (base_price > 0),
    production_time_days INTEGER NOT NULL CHECK (production_time_days > 0)
);

-- Таблица для связи M:N между Product и Material
CREATE TABLE ProductMaterial (
    product_id INTEGER NOT NULL,
    material_id INTEGER NOT NULL,
    quantity DECIMAL(8,2) NOT NULL CHECK (quantity > 0),
    PRIMARY KEY (product_id, material_id),
    FOREIGN KEY (product_id) REFERENCES Product(product_id),
    FOREIGN KEY (material_id) REFERENCES Material(material_id)
);

CREATE TABLE "Order" (
    order_id SERIAL PRIMARY KEY,
    client_id INTEGER NOT NULL,
    product_id INTEGER NOT NULL,
    employee_id INTEGER NOT NULL,
    order_date DATE NOT NULL DEFAULT CURRENT_DATE,
    completion_date DATE CHECK (completion_date >= order_date),
```

```
    status VARCHAR(20) NOT NULL CHECK (status IN ('принят', 'в работе', 'гото
в', 'выдан')),
    total_price DECIMAL(10,2) NOT NULL CHECK (total_price > 0),
    FOREIGN KEY (client_id) REFERENCES Client(client_id),
    FOREIGN KEY (product_id) REFERENCES Product(product_id),
    FOREIGN KEY (employee_id) REFERENCES Employee(employee_id)
);
```

3. Триггеры для модели

1. Триггер для автоматического расчета стоимости заказа

sql

Copy

Download

```
CREATE OR REPLACE FUNCTION calculate_order_price()
RETURNS TRIGGER AS $$
BEGIN
    NEW.total_price := (SELECT base_price FROM Product WHERE product_id = NEW
.product_id) *
                       (1 - (SELECT discount FROM Client WHERE client_id = NEW
.client_id)/100);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_order_price
BEFORE INSERT OR UPDATE OF product_id, client_id ON "Order"
FOR EACH ROW EXECUTE FUNCTION calculate_order_price();
```

2. Триггер для контроля остатков материалов

sql

Copy

Download

```
CREATE OR REPLACE FUNCTION check_material_stock()
RETURNS TRIGGER AS $$
DECLARE
    required DECIMAL;
    available INTEGER;
BEGIN
    SELECT pm.quantity INTO required
    FROM ProductMaterial pm
    WHERE pm.product_id = NEW.product_id;

    SELECT m.quantity_in_stock INTO available
    FROM Material m
    JOIN ProductMaterial pm ON m.material_id = pm.material_id
```

```
    WHERE pm.product_id = NEW.product_id;

    IF available < required THEN
        RAISE EXCEPTION 'Недостаточно материалов на складе для изделия %',
                        (SELECT name FROM Product WHERE product_id = NEW.prod
uct_id);
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_check_materials
BEFORE INSERT ON "Order"
FOR EACH ROW EXECUTE FUNCTION check_material_stock();
```

3. Триггер для обновления статуса заказа

sql

Copy

Download

```
CREATE OR REPLACE FUNCTION update_order_status()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.completion_date IS NOT NULL AND NEW.status != 'выдан' THEN
        NEW.status := 'готов';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_update_status
BEFORE UPDATE OF completion_date ON "Order"
FOR EACH ROW EXECUTE FUNCTION update_order_status();
```

Виды триггеров:

1. **BEFORE** - выполняются перед операцией (полезны для валидации)
2. **AFTER** - выполняются после операции (полезны для аудита)
3. **INSTEAD OF** - заменяют операцию (для представлений)
4. **FOR EACH ROW** - построчная обработка
5. **FOR EACH STATEMENT** - обработка на уровне оператора

4. SQL-запрос с соединением M:N

sql

Copy
Download

```
SELECT
    c.full_name AS client_name,
    c.phone,
    p.name AS product_name,
    e.full_name AS employee_name,
    o.order_date,
    o.completion_date,
    o.total_price,
    STRING_AGG(m.name, ', ') AS materials_used
FROM
    "Order" o
INNER JOIN Client c ON o.client_id = c.client_id
INNER JOIN Product p ON o.product_id = p.product_id
INNER JOIN Employee e ON o.employee_id = e.employee_id
INNER JOIN ProductMaterial pm ON p.product_id = pm.product_id
INNER JOIN Material m ON pm.material_id = m.material_id
WHERE
    o.status = 'готов'
    AND o.completion_date BETWEEN CURRENT_DATE - INTERVAL '30 days' AND CURRE
NT_DATE
GROUP BY
    c.client_id, p.product_id, e.employee_id, o.order_id
ORDER BY
    o.completion_date DESC;
```

5. План выполнения запроса и оптимизация

Оптимальный план выполнения:

text

Copy
Download

```
QUERY PLAN
Sort  (cost=458.29..459.29 rows=400 width=172)
  Sort Key: o.completion_date DESC
  ->  GroupAggregate  (cost=358.17..442.17 rows=400 width=172)
        Group Key: c.client_id, p.product_id, e.employee_id, o.order_id
        ->  Sort  (cost=358.17..359.17 rows=400 width=140)
              Sort Key: c.client_id, p.product_id, e.employee_id, o.order_id
              ->  Nested Loop  (cost=20.15..340.17 rows=400 width=140)
                    ->  Hash Join  (cost=19.85..38.85 rows=100 width=108)
                          Hash Cond: (o.employee_id = e.employee_id)
                          ->  Nested Loop  (cost=10.25..27.25 rows=100 width=
44)
```

```
                                ->  Hash Join  (cost=9.85..16.35 rows=100 wid
th=24)
                                      Hash Cond: (o.client_id = c.client_id)
                                      ->  Seq Scan on "Order" o  (cost=0.00..
5.50 rows=100 width=16)
                                            Filter: ((status = 'готов'::text)
AND
                                                   (completion_date >= (CURRE
NT_DATE - 30)) AND
                                                   (completion_date <= CURREN
T_DATE))
                                      ->  Hash  (cost=6.50..6.50 rows=200 wid
th=8)
                                            ->  Seq Scan on Client c  (cost=0
.00..6.50 rows=200 width=8)
                                ->  Index Scan using product_pkey on Product
p  (cost=0.40..0.42 rows=1 width=20)
                                      Index Cond: (product_id = o.product_id)
                          ->  Hash  (cost=6.50..6.50 rows=200 width=64)
                                ->  Seq Scan on Employee e  (cost=0.00..6.50
rows=200 width=64)
                    ->  Materialize  (cost=0.30..3.30 rows=4 width=32)
                          ->  Nested Loop  (cost=0.30..3.10 rows=4 width=32)
                                ->  Index Scan using productmaterial_product_
idx on ProductMaterial pm  (cost=0.15..1.17 rows=4 width=8)
                                      Index Cond: (product_id = p.product_id)
                                ->  Index Scan using material_pkey on Materia
l m  (cost=0.15..0.47 rows=1 width=24)
                                      Index Cond: (material_id = pm.material_
id)
```

Рекомендуемые индексы для оптимизации:

sql

Copy

Download

```sql
-- Для соединений
CREATE INDEX idx_order_client ON "Order"(client_id);
CREATE INDEX idx_order_product ON "Order"(product_id);
CREATE INDEX idx_order_employee ON "Order"(employee_id);
CREATE INDEX idx_productmaterial_product ON ProductMaterial(product_id);
CREATE INDEX idx_productmaterial_material ON ProductMaterial(material_id);

-- Для условий WHERE
CREATE INDEX idx_order_status_date ON "Order"(status, completion_date);

-- Для сортировки
CREATE INDEX idx_order_completion_date ON "Order"(completion_date DESC);
```

Алгоритмы соединений и обоснование:

1. **Hash Join** для Order-Employee:
    - Эффективен при соединении большой таблицы заказов с таблицей сотрудников
    - PostgreSQL выбирает этот метод, когда одна таблица может быть полностью помещена в хеш-таблицу
2. **Nested Loop** для Order-Client и ProductMaterial-Material:
    - Используется для точечных соединений по первичным ключам
    - Эффективен при небольшом количестве строк (для ProductMaterial-Material)
3. **GroupAggregate** для агрегации материалов:
    - Оптимален для GROUP BY после предварительной сортировки
    - STRING_AGG требует полного набора данных перед агрегацией

Обоснование выбора плана:

- План минимизирует стоимость операций для данного запроса
- Nested Loop выбран для соединений с Product и Material из-за высокой селективности
- Hash Join эффективен для соединения с Employee (меньшая таблица)
- Индексы ускоряют поиск по статусу и дате выполнения заказов
- Сортировка по completion_date использует индекс при его наличии