Санкт-Петербургский Национальный Исследовательский Университет ИТМО

Факультет Программной Инженерии и Компьютерной Техники



Вариант № 1902

Лабораторная работа №3

По дисциплине

Базы Данных

Выполнил студент группы P3119:

Ануфриев Андрей Сергеевич

Преподаватель:

Бойко Владислав Алексеевич

Санкт-Петербург 2025 г.

Оглавление

[Текст задания 3](#_Toc197252226)

[Изначальная бд 4](#_Toc197252227)

[Функциональные зависимости 5](#_Toc197252228)

[Нормальные формы 5](#_Toc197252229)

[Проверка на BCNF 6](#_Toc197252230)

[Денормализация 6](#_Toc197252231)

[Триггер и функция на PL/pgSQL 6](#_Toc197252232)

[Вывод 6](#_Toc197252233)

[Ответы на вопросы 7](#_Toc197252234)

# Текст задания

Для отношений, полученных при построении предметной области из лабораторной работы №1, выполните следующие действия:

* Опишите функциональные зависимости для отношений полученной схемы (минимальное множество);
* Приведите отношения в 3NF (как минимум). Постройте схему на основеNF (как минимум).
* Опишите изменения в функциональных зависимостях, произошедшие после преобразования в 3NF (как минимум). Постройте схему на основеNF;
* Преобразуйте отношения в BCNF. Докажите, что полученные отношения представлены в BCNF. Если ваша схема находится уже в BCNF, докажите это;
* Какие денормализации будут полезны для вашей схемы? Приведите подробное описание.

Придумайте триггер и связанную с ним функцию, относящиеся к вашей предметной области, согласуйте их с преподавателем и реализуйте на языке PL/pgSQL.

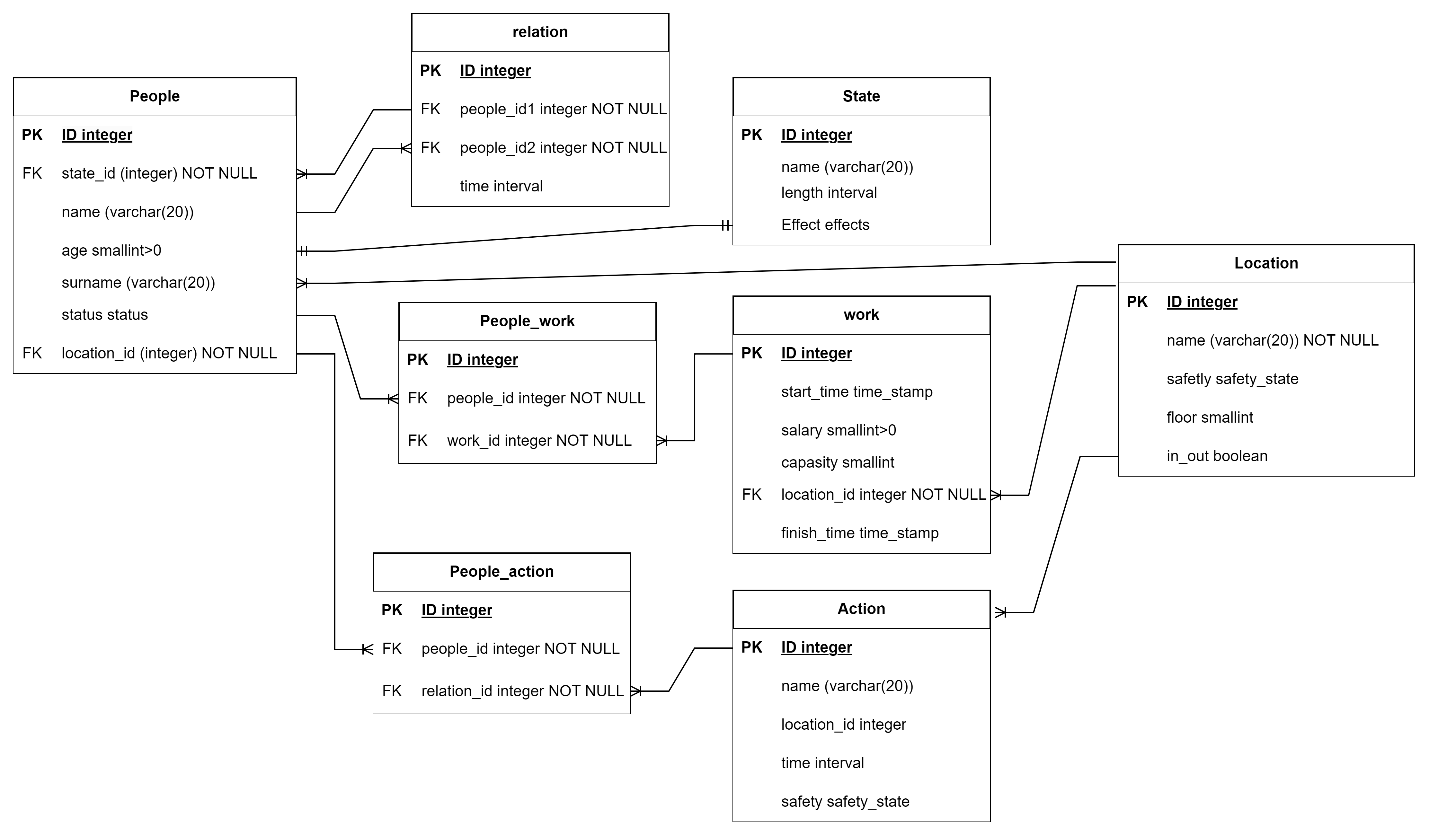
Отчёт по лабораторной работе должен содержать:

1. ~~Текст задания.~~
2. Исходная, нормализованная и денормализованная модели.
3. Ответы на вопросы, представленные в задании.
4. Функция и триггер на языке PL/pgSQL
5. Выводы по работе.

Темы для подготовки к защите лабораторной работы:

1. Нормализация. Формы
2. Функциональные зависимости. Виды
3. Денормализация
4. Язык PL/pgSQL

Изначальная бд





CREATE TYPE safety\_state AS ENUM('безопасный', 'очень опасный','опасный');  
CREATE TYPE effects AS ENUM('бояться', 'радоваться','болеть', 'сидеть', 'стоять', 'лежать', 'бежать');  
  
CREATE TABLE location(  
 id int PRIMARY KEY,  
 name varchar(20) NOT NULL DEFAULT 'безымянный',  
 safety safety\_state,  
 floor smallint,  
 in\_out BOOLEAN DEFAULT 'true'  
);  
CREATE TABLE work(  
 id int PRIMARY KEY,  
 start\_time timestamp,  
 finish\_time timestamp,  
 salary smallint check ( salary>0 ),  
 capacity smallint check ( capacity>0 ),  
 location\_id int NOT NULL references location(id)  
  
);  
CREATE TABLE action(  
 id int PRIMARY KEY,  
 name varchar(20),  
 location\_id int NOT NULL references location(id),  
 time interval,  
 safety safety\_state  
);  
CREATE TABLE state(  
 id int PRIMARY KEY,  
 name varchar(20) NOT NULL DEFAULT 'безымянный',  
 length interval,  
 effect effects  
);  
CREATE TABLE people(  
 id int PRIMARY KEY,  
 name varchar(20) NOT NULL DEFAULT 'безымянный',  
 surname varchar(20),  
 age smallint check ( age>0 ),  
 state\_id int NOT NULL references state(id),  
 location\_id int NOT NULL references location(id)  
);  
CREATE TABLE relation(  
 id int PRIMARY KEY,  
 people\_id1 int NOT NULL references people(id),  
 people\_id2 int NOT NULL references people(id),  
 time interval  
);  
  
CREATE TABLE people\_action(  
 id int PRIMARY KEY,  
 people\_id int NOT NULL references people(id),  
 action\_id int NOT NULL references action(id)  
);  
CREATE TABLE people\_work(  
 id int PRIMARY KEY,  
 people\_id int NOT NULL references people(id),  
 work\_id int NOT NULL references work(id)  
);

# Функциональные зависимости

Таблица location:

id → name, safety, floor, in\_out

Таблица work:

id → start\_time, finish\_time, salary, capacity, location\_id

Таблица action:

id → name, location\_id, time, safety

Таблица state:

id → name, length, effect

Таблица people:

id → name, surname, age, state\_id, location\_id

Таблица relation:

id → people\_id1, people\_id2, time

Таблица people\_action:

(people\_action.people\_id, people\_action.action\_id) -> people\_id, action\_id

Таблица people\_work:

(people\_work.people\_id, people\_ work. work \_id) -> people\_id, work \_id

# Нормальные формы

**1NF**: Отношение находится в 1NF, если все его атрибуты содержат только атомарные значения. У меня так и есть, ни в какой ячейки нет 2 значений.

**2NF**: Отношение находится в «NF, если оно находится в 1NF и все его неключевые атрибуты полностью функционально зависят от первичного ключа. У меня так и есть: Все атрибуты зависят от ключа и только от него.

**3NF**: Отношение находится в 3NF, если оно находится во 2NF и не содержит транзитивных зависимостей. У меня так и есть: все неключевые атрибуты зависят только от первичных ключей, и не содержат транзитивных зависимостей.

# Проверка на BCNF

**BCNF:** Отношение находится в BCNF,

если для каждой функциональной зависимости X ‎‎‎- Y,

X является суперключом. У меня так и есть: для всех функциональных зависимостей X является суперключом.

# Денормализация

* **Добавить вычисляемые поля:**

В таблицу people добавить количество действий (action\_count) и количество работ (work\_count) для быстрого подсчета без JOIN

* **Дублировать часто используемые данные:**

В таблицу people\_action добавить название действия (action\_name) из таблицы action для ускорения отчетов

В таблицу people\_work добавить название работы (work\_name) из таблицы work

* **Объединить редко используемые таблицы:**

Можно объединить people\_action и action, если действия редко используются отдельно от людей

# Триггер и функция на PL/pgSQL

Цель: если человека перемещают в локацию с safety = 'опасный', автоматически изменить его статус на 'боязнь'.

CREATE OR REPLACE FUNCTION *update\_status\_on\_danger*()  
RETURNS TRIGGER AS $$  
BEGIN  
 *-- Проверяем, что новая локация опасная* IF (SELECT safety FROM location WHERE id = NEW.location\_id) = 'опасный' THEN  
 *-- Находим ID статуса "боязнь" и обновляем state\_id у человека* NEW.state\_id := (SELECT id FROM state WHERE name = 'боязнь' LIMIT 1);  
 END IF;  
 RETURN NEW; *-- Возвращаем измененную запись*END;  
$$ LANGUAGE plpgsql;  
  
*-- Триггер срабатывает ПЕРЕД обновлением location\_id в таблице people*CREATE TRIGGER trg\_update\_status\_on\_danger  
BEFORE UPDATE OF location\_id ON people  
FOR EACH ROW  
EXECUTE FUNCTION *update\_status\_on\_danger*();

Только при изменении location\_id в таблице people.

До фактического обновления (BEFORE UPDATE), чтобы модифицировать запись перед сохранением.

Пример

-- Допустим, локация с ID=2 имеет safety='опасный'

UPDATE people SET location\_id = 2 WHERE id = 1;

Результат:

Помимо смены локации, у человека с id=1 автоматически обновится state\_id на статус 'боязнь'.

# Вывод

В ходе работы я узнал, что такое нормальные формы и зачем они нужны. Поизучал PL/pgSQL — процедурный язык SQL. Попробовал написать функции на нём. Также узнал, что такое триггеры, как и где они применяются и использовал это в работе.

-- Создаем функцию для обновления счетчиков

CREATE OR REPLACE FUNCTION *update\_people\_counters*()  
RETURNS TRIGGER AS $$  
BEGIN  
 *-- Проверяем, какая операция вызвала триггер* IF (TG\_OP = 'DELETE') THEN  
 *-- Если удаление, уменьшаем счетчики у связанного человека* UPDATE people   
 SET   
 action\_count = action\_count - CASE WHEN TG\_TABLE\_NAME = 'people\_action' THEN 1 ELSE 0 END,  
 work\_count = work\_count - CASE WHEN TG\_TABLE\_NAME = 'people\_work' THEN 1 ELSE 0 END  
 WHERE id = OLD.people\_id;  
   
 ELSIF (TG\_OP = 'INSERT') THEN  
 *-- Если вставка, увеличиваем счетчики у связанного человека* UPDATE people   
 SET   
 action\_count = action\_count + CASE WHEN TG\_TABLE\_NAME = 'people\_action' THEN 1 ELSE 0 END,  
 work\_count = work\_count + CASE WHEN TG\_TABLE\_NAME = 'people\_work' THEN 1 ELSE 0 END  
 WHERE id = NEW.people\_id;  
   
 ELSIF (TG\_OP = 'UPDATE') THEN  
 *-- Если обновление, меняем счетчики у старого и нового человека (если изменился people\_id)* IF (OLD.people\_id <> NEW.people\_id) THEN  
 *-- Уменьшаем у старого человека* UPDATE people   
 SET   
 action\_count = action\_count - CASE WHEN TG\_TABLE\_NAME = 'people\_action' THEN 1 ELSE 0 END,  
 work\_count = work\_count - CASE WHEN TG\_TABLE\_NAME = 'people\_work' THEN 1 ELSE 0 END  
 WHERE id = OLD.people\_id;  
   
 *-- Увеличиваем у нового человека* UPDATE people   
 SET   
 action\_count = action\_count + CASE WHEN TG\_TABLE\_NAME = 'people\_action' THEN 1 ELSE 0 END,  
 work\_count = work\_count + CASE WHEN TG\_TABLE\_NAME = 'people\_work' THEN 1 ELSE 0 END  
 WHERE id = NEW.people\_id;  
 END IF;  
 END IF;  
   
 *-- Для AFTER триггера возвращаемое значение не важно* RETURN NULL;  
END;  
$$ LANGUAGE plpgsql;  
  
*-- Создаем триггеры для таблицы people\_action*CREATE TRIGGER trg\_people\_action\_count  
AFTER INSERT OR UPDATE OR DELETE ON people\_action  
FOR EACH ROW  
EXECUTE FUNCTION *update\_people\_counters*();  
  
*-- Создаем триггеры для таблицы people\_work*CREATE TRIGGER trg\_people\_work\_count  
AFTER INSERT OR UPDATE OR DELETE ON people\_work  
FOR EACH ROW  
EXECUTE FUNCTION *update\_people\_counters*();

При DELETE: уменьшаем соответствующий счетчик у человека

При INSERT: увеличиваем соответствующий счетчик у человека

При UPDATE: если изменился people\_id, уменьшаем счетчик у старого человека и увеличиваем у нового

Пример работы:

При добавлении новой связи между человеком и действием:

sql

INSERT INTO people\_action (id, people\_id, action\_id) VALUES (5, 1, 3);

Триггер увеличит action\_count у человека с id=1

При удалении связи:

sql

DELETE FROM people\_action WHERE id = 5;

Триггер уменьшит action\_count у человека с id=1

При изменении связи:

sql

UPDATE people\_action SET people\_id = 2 WHERE id = 5;

Триггер уменьшит action\_count у человека 1 и увеличит у человека 2

# Ответы на вопросы

ANY\SOME

* Подзапрос должен возвращать ровно один столбец
* Значение выражения сравнивается со значением в каждой

строке результата подзапроса с помощью заданного

оператора, который должен возвращать логическое

значение.

* Результатом ANY будет «true», если хотя бы для одной

строки условие истинно, и «false» в противном случае (в

том числе, если подзапрос не возвращает строк).

* Оператор IN аналогичен = ANY (= SOME)

1. **SQL**

**WHERE** — для определения, какие строки должны быть выбраны или включены в GROUP BY.

**GROUP BY** — для объединения строк с общими значениями в элементы меньшего набора строк.

**HAVING** — для определения, какие строки после GROUP BY должны быть выбраны.

**ORDER BY** — сортировка результирующего набора данных

**DISTINCT -** Указывает, что для вычислений применяются только уникальные значения столбца.

**AVG**()- среднее значение, игнорирует NULL

**COUNT**() — подсчитывает количество строк в наборе результатов или количество ненулевых значений в столбце.

**CAST** **(expression AS target\_type)** для преобразования значения одного типа данных в другой. Можно и через :: SELECT '123'::INTEGER;

**Агрегатные функции это MIN, MAX, SUM, AVG, COUNT, COUNT\_BIG**

1. **Соединение таблиц**

**OUTER** - это внешнее соединение. Оно возвращает не только строгое пересечение между двумя таблицами, но и отдельные элементы, которые принадлежат только одному из множеств.

**USING** — сокращённая запись условия, когда с обеих сторон соединения столбцы имеют одинаковые имена.

**NATURAL** — упрощённая форма USING: образует список USING из всех имён столбцов, существующих в обеих входных таблицах. Если столбцов с одинаковыми именами нет, NATURAL работает как CROSS JOIN

**1. INNER JOIN (Внутреннее соединение)**

Возвращает только строки, где есть совпадения в обеих таблицах.

SELECT a.\*, b.\*

FROM table\_a a

INNER JOIN table\_b b ON a.id = b.a\_id;

**2. LEFT JOIN (LEFT OUTER JOIN)**

Возвращает все строки из левой таблицы и соответствующие строки из правой. Если соответствия нет, возвращает NULL для правой таблицы.

SELECT a.\*, b.\*

FROM table\_a a

LEFT JOIN table\_b b ON a.id = b.a\_id;

**3. NATURAL JOIN**

Соединяет таблицы по столбцам с одинаковыми именами.

**4. FULL JOIN (FULL OUTER JOIN)**

Возвращает все строки из обеих таблиц, с NULL значениями там, где нет соответствий.

SELECT a.\*, b.\*

FROM table\_a a

FULL JOIN table\_b b ON a.id = b.a\_id;

**5. CROSS JOIN**

Декартово произведение - каждая строка из первой таблицы соединяется с каждой строкой из второй.

1. **Подзапросы**

Вложенный подзапрос (subquery) — предложение SELECT, которое заключено в круглые скобки и вложено в WHERE/HAVING часть другого SQL предложения

Коррелированные вложенные подзапросы

Вложенный подзапрос не может быть выполнен до обработки внешнего запроса: SELECT Surname FROM STUDENT WHERE EXISTS ( SELECT 1 FROM STUDENT\_OLYMPIAD WHERE StID = STUDENT.StudentID );

1. Представления

Представление — именованный запрос.

CREATE VIEW VIEW\_NAME

[ ( ColumnName [, ...] ) ]

AS подзапрос

Материализованные представления

CREATE MATERIALIZED VIEW PICTStudents3 AS

(PICTId, pSurname) AS

SELECT StudentID, Surname FROM STUDENT

WHERE GroupID IN (

SELECT GroupID FROM GROUP

WHERE GroupName LIKE 'P3%' );

Результат запроса сохраняется в базе данных.

Для обновления данных:

REFRESH MATERIALIZED VIEW PICTStudents3;

1. Последовательности

Последовательности это – это определяемые пользователем объекты базы данных, предназначенные для генерации ряда числовых значений. В отличие от столбцов идентификаторов, которые тесно связаны с конкретными таблицами, последовательности являются независимыми объектами и могут использоваться в нескольких таблицах. Они позволяют приложениям получать следующее число в последовательности по мере необходимости, предлагая простой и эффективный способ генерации уникальных чисел по запросу.

Автоматическая генерация первичного ключа: последовательности автоматически генерируют уникальные значения, которые можно использовать в качестве первичных или уникальных ключей в таблицах баз данных.

По возрастанию или по убыванию: последовательности можно настроить на генерацию чисел в порядке возрастания или убывания.

Использование нескольких таблиц: одна последовательность может использоваться для генерации значений для нескольких таблиц, что делает ее гибкой и пригодной для повторного использования.

Независимые от таблиц: в отличие от столбцов идентификаторов, последовательности являются независимыми и могут использоваться в разных таблицах.

CREATE SEQUENCE sequence\_1  
start with 1  
increment by 1  
minvalue 0  
maxvalue 100  
cycle;

### **Примеры операций реляционной алгебры в PostgreSQL**

Реляционная алгебра — это теоретическая основа SQL, и PostgreSQL поддерживает все её основные операции. Рассмотрим, как каждая операция реализуется в PostgreSQL.

## **🔹 1. Теоретико-множественные операции**

### **1.1. Объединение (UNION)**

Возвращает уникальные строки из двух таблиц.

**Пример:**

sql

Copy

Download

-- Студенты из двух групп (без дубликатов)

SELECT \* FROM group1

UNION

SELECT \* FROM group2;

### **1.2. Пересечение (INTERSECT)**

Возвращает только общие строки.

**Пример:**

sql

Copy

Download

-- Студенты, которые есть в обеих группах

SELECT \* FROM group1

INTERSECT

SELECT \* FROM group2;

### **1.3. Разность (EXCEPT или MINUS)**

Возвращает строки из первой таблицы, которых нет во второй.

**Пример:**

sql

Copy

Download

-- Студенты из group1, которых нет в group2

SELECT \* FROM group1

EXCEPT

SELECT \* FROM group2;

### **1.4. Декартово произведение (CROSS JOIN)**

Комбинация всех строк из двух таблиц.

**Пример:**

sql

Copy

Download

-- Все возможные пары "студент × предмет"

SELECT \* FROM students

CROSS JOIN courses;

## **🔹 2. Специальные реляционные операции**

### **2.1. Выборка (WHERE)**

Аналог операции **σ (sigma)**.

**Пример:**

sql

Copy

Download

-- Студенты старше 20 лет (σ(age > 20)(Students))

SELECT \* FROM students WHERE age > 20;

### **2.2. Проекция (SELECT)**

Аналог операции **π (pi)**.

**Пример:**

sql

Copy

Download

-- Имена и возраст студентов (π(name, age)(Students))

SELECT name, age FROM students;

### **2.3. Соединение (JOIN)**

Аналог операции **⋈ (Join)**.

**Примеры:**

#### **Естественное соединение (по общему полю)**

sql

Copy

Download

-- Студенты и их оценки (Students ⋈ Grades)

SELECT \* FROM students

JOIN grades ON students.id = grades.student\_id;

#### **Тета-соединение (по условию)**

sql

Copy

Download

-- Студенты и курсы, где возраст студента > минимального возраста курса

SELECT \* FROM students

JOIN courses ON students.age > courses.min\_age;

#### **Внешнее соединение (LEFT/RIGHT/FULL JOIN)**

sql

Copy

Download

-- Все студенты, включая тех, у кого нет оценок (LEFT JOIN)

SELECT \* FROM students

LEFT JOIN grades ON students.id = grades.student\_id;

### **2.4. Переименование (AS)**

Аналог операции **ρ (rho)**.

**Пример:**

sql

Copy

Download

-- Переименование столбца

SELECT name AS student\_name FROM students;

-- Переименование таблицы в запросе

SELECT s.name FROM students AS s;

### **2.5. Деление (эмулируется через NOT EXISTS)**

Операция **÷ (Division)** в SQL не имеет прямого аналога, но её можно эмулировать.

**Пример:**

sql

Copy

Download

-- Найти студентов, записанных на ВСЕ курсы (R ÷ S)

SELECT s.name

FROM students s

WHERE NOT EXISTS (

SELECT c.id FROM courses c

WHERE NOT EXISTS (

SELECT 1 FROM enrollments e

WHERE e.student\_id = s.id AND e.course\_id = c.id

)

);

## **🔹 3. Комбинированные примеры (как в реляционной алгебре)**

### **Пример 1: Студенты, изучающие математику**

**Реляционная алгебра:**

Copy

Download

π(name)(σ(course = 'Math')(Students ⋈ Enrollments))

**PostgreSQL:**

sql

Copy

Download

SELECT s.name

FROM students s

JOIN enrollments e ON s.id = e.student\_id

WHERE e.course = 'Math';

### **Пример 2: Курсы, на которые записан студент "Иван"**

**Реляционная алгебра:**

Copy

Download

π(course)(σ(name = 'Иван')(Students) ⋈ Enrollments)

**PostgreSQL:**

sql

Copy

Download

SELECT e.course

FROM students s

JOIN enrollments e ON s.id = e.student\_id

WHERE s.name = 'Иван';

### **Пример 3: Студенты, не записанные ни на один курс**

**Реляционная алгебра:**

Copy

Download

π(name)(Students) − π(name)(Students ⋈ Enrollments)

**PostgreSQL:**

sql

Copy

Download

SELECT s.name

FROM students s

LEFT JOIN enrollments e ON s.id = e.student\_id

WHERE e.student\_id IS NULL;

## **🔹 Вывод**

PostgreSQL полностью поддерживает все операции реляционной алгебры, хотя некоторые (например, деление) требуют сложных подзапросов. Понимание реляционной алгебры помогает писать **более эффективные SQL-запросы** и оптимизировать работу с базами данных.

Если нужно углубиться в конкретную операцию — спрашивайте! 🚀

Аксиомы, правила и формулы в реляционной алгебре

### **Аксиомы, правила и формулы реляционной алгебры**

Реляционная алгебра — это формальная система, основанная на математических принципах. Она включает **аксиомы** (базовые утверждения), **правила эквивалентности** (преобразования запросов) и **формулы** (выражения операций).

## **🔹 1. Аксиомы реляционной алгебры**

Аксиомы определяют свойства операций. Основные из них:

### **1.1. Аксиомы объединения (∪) и пересечения (∩)**

* **Коммутативность**:  
  R∪S=S∪R*R*∪*S*=*S*∪*R*  
  R∩S=S∩R*R*∩*S*=*S*∩*R*
* **Ассоциативность**:  
  (R∪S)∪T=R∪(S∪T)(*R*∪*S*)∪*T*=*R*∪(*S*∪*T*)  
  (R∩S)∩T=R∩(S∩T)(*R*∩*S*)∩*T*=*R*∩(*S*∩*T*)
* **Дистрибутивность**:  
  R∪(S∩T)=(R∪S)∩(R∪T)*R*∪(*S*∩*T*)=(*R*∪*S*)∩(*R*∪*T*)  
  R∩(S∪T)=(R∩S)∪(R∩T)*R*∩(*S*∪*T*)=(*R*∩*S*)∪(*R*∩*T*)

### **1.2. Аксиомы разности (−)**

* R−S=R−(R∩S)*R*−*S*=*R*−(*R*∩*S*)
* R−(S∪T)=(R−S)∩(R−T)*R*−(*S*∪*T*)=(*R*−*S*)∩(*R*−*T*)

### **1.3. Аксиомы декартова произведения (×)**

* R×(S×T)=(R×S)×T*R*×(*S*×*T*)=(*R*×*S*)×*T*
* Если R*R* и S*S* не имеют общих атрибутов:  
  σcond(R×S)=R⋈condS*σcond*​(*R*×*S*)=*R*⋈*cond*​*S*

## **🔹 2. Правила эквивалентности**

Позволяют оптимизировать запросы, преобразуя их без изменения результата.

### **2.1. Коммутативность и ассоциативность соединений**

* R⋈S=S⋈R*R*⋈*S*=*S*⋈*R*
* (R⋈S)⋈T=R⋈(S⋈T)(*R*⋈*S*)⋈*T*=*R*⋈(*S*⋈*T*)

### **2.2. Дистрибутивность выборки (σ)**

* σcond1(σcond2(R))=σcond1∧cond2(R)*σcond*1​(*σcond*2​(*R*))=*σcond*1∧*cond*2​(*R*)
* σcond(R∪S)=σcond(R)∪σcond(S)*σcond*​(*R*∪*S*)=*σcond*​(*R*)∪*σcond*​(*S*)

### **2.3. Перестановка проекции (π) и выборки (σ)**

Если условие cond зависит только от атрибутов в A:

* πA(σcond(R))=σcond(πA(R))*πA*​(*σcond*​(*R*))=*σcond*​(*πA*​(*R*))

### **2.4. Разложение соединения**

* R⋈(S⋈T)=(R⋈S)⋈T*R*⋈(*S*⋈*T*)=(*R*⋈*S*)⋈*T*

## **🔹 3. Основные формулы реляционной алгебры**

Формальные выражения операций.

### **3.1. Выборка (σ)**

σcond(R)={t∣t∈R, cond(t)}*σcond*​(*R*)={*t*∣*t*∈*R*,*cond*(*t*)}

**Пример:**  
σage>20(Students)*σage*>20​(*Students*) — студенты старше 20 лет.

### **3.2. Проекция (π)**

πA1,A2,…,Ak(R)={t[A1,Aё2,…,Ak]∣t∈R}*πA*1​,*A*2​,…,*Ak*​​(*R*)={*t*[*A*1​,*A*2​,…,*Ak*​]∣*t*∈*R*}

ё**Пример:**  
πname,age(Students)*πname*,*age*​(*Students*) — имена и возраст студентов.

### **3.3. Соединение (⋈)**

* **Естественное соединение**:  
  R⋈S={t∪s∣t∈R, s∈S, t[общие атрибуты]=s[общие атрибуты]}*R*⋈*S*={*t*∪*s*∣*t*∈*R*,*s*∈*S*,*t*[общие атрибуты]=*s*[общие атрибуты]}
* **Тета-соединение**:  
  R⋈condS=σcond(R×S)*R*⋈*cond*​*S*=*σcond*​(*R*×*S*)

### **3.4. Деление (÷)**

R÷S={t∣∀s∈S, (t∪s)∈R}*R*÷*S*={*t*∣∀*s*∈*S*,(*t*∪*s*)∈*R*}

**Пример:**  
Students÷CoursesStudents÷Courses — студенты, записанные на все курсы.

## **🔹 4. Примеры преобразований запросов**

### **4.1. Перенос выборки перед соединением**

Исходный запрос:  
πname(σage>20(Students⋈Enrollments))*πname*​(*σage*>20​(*Students*⋈*Enrollments*))

Оптимизированный:  
πname(σage>20(Students)⋈Enrollments)*πname*​(*σage*>20​(*Students*)⋈*Enrollments*)

### **4.2. Разложение сложного условия**

σage>20∧course=′Math′(Students⋈Enrollments)=σcourse=′Math′(σage>20(Students)⋈Enrollments)*σage*>20∧*course*=′*Math*′​(*Students*⋈*Enrollments*)=*σcourse*=′*Math*′​(*σage*>20​(*Students*)⋈*Enrollments*)

## **🔹 5. Связь с реляционным исчислением**

Реляционная алгебра эквивалентна **безопасному реляционному исчислению** (Codd’s Theorem). Например:

* πA(R)*πA*​(*R*) соответствует {t.A∣R(t)}{*t*.*A*∣*R*(*t*)}.
* σcond(R)*σcond*​(*R*) соответствует {t∣R(t)∧cond(t)}{*t*∣*R*(*t*)∧*cond*(*t*)}.

**3 Аксиомы амстронга**

### **Аксиомы Армстронга в теории баз данных**

Аксиомы Армстронга — это набор фундаментальных правил, используемых для вывода **функциональных зависимостей (ФЗ)** в реляционных базах данных. Они были предложены Уильямом Армстронгом в 1974 году и лежат в основе нормализации БД.

## **1. Формальное определение функциональной зависимости**

Функциональная зависимость X→Y*X*→*Y* означает, что если два кортежа совпадают по атрибутам X*X*, то они обязательно совпадают и по атрибутам Y*Y*.

**Пример:**  
В таблице Students(StudentID, Name, Age) справедлива ФЗ:  
StudentID→Name, AgeStudentID→Name, Age  
(если StudentID одинаковый, то Name и Age тоже одинаковые).

## **2. Три** функциональные зависимости из имеющихся.

### **2.1. Рефлексивность (Reflexivity)**

**Если Y⊆X*Y*⊆*X*, то X→Y*X*→*Y*.**

**Интерпретация:**

* Любое множество атрибутов функционально определяет своё подмножество.
* Тривиальная зависимость (например, StudentID, Name→NameStudentID, Name→Name).

### **2.2. Пополнение (Augmentation)**

**Если X→Y*X*→*Y*, то XZ→YZ*XZ*→*YZ* для любого Z*Z*.**

**Интерпретация:**

* Добавление атрибутов в левую и правую часть не нарушает зависимость.
* Пример:  
  Дано: StudentID→NameStudentID→Name  
  Следует: StudentID, Age→Name, AgeStudentID, Age→Name, Age.

### **2.3. Транзитивность (Transitivity)**

**Если X→Y*X*→*Y* и Y→Z*Y*→*Z*, то X→Z*X*→*Z*.**

**Интерпретация:**

* Цепочка зависимостей сокращается.
* Пример:  
  Дано:  
  StudentID→DepartmentIDStudentID→DepartmentID  
  DepartmentID→DeanDepartmentID→Dean  
  Следует:  
  StudentID→DeanStudentID→Dean.

## **3. Дополнительные правила (выводимые из аксиом)**

Из трёх аксиом Армстронга можно вывести дополнительные полезные правила.

### **3.1. Объединение (Union)**

**Если X→Y*X*→*Y* и X→Z*X*→*Z*, то X→YZ*X*→*YZ*.**

**Пример:**  
StudentID→NameStudentID→Name  
StudentID→AgeStudentID→Age  
⇒ StudentID→Name, AgeStudentID→Name, Age.

### **3.2. Декомпозиция (Decomposition)**

**Если X→YZ*X*→*YZ*, то X→Y*X*→*Y* и X→Z*X*→*Z*.**

**Пример:**  
StudentID→Name, AgeStudentID→Name, Age  
⇒ StudentID→NameStudentID→Name  
и StudentID→AgeStudentID→Age.

### **3.3. Псевдотранзитивность (Pseudotransitivity)**

**Если X→Y*X*→*Y* и YW→Z*YW*→*Z*, то XW→Z*XW*→*Z*.**

**Пример:**  
StudentID→DepartmentIDStudentID→DepartmentID  
DepartmentID, Year→BudgetDepartmentID, Year→Budget  
⇒ StudentID, Year→BudgetStudentID, Year→Budget.

## **4. Пример использования аксиом**

**Дано:**

1. A→B*A*→*B*
2. B→C*B*→*C*
3. C→D*C*→*D*

**Вывод:**

1. По транзитивности (1 и 2): A→C*A*→*C*.
2. По транзитивности (результат и 3): A→D*A*→*D*.

**Итог:** A*A* определяет B,C,D*B*,*C*,*D*.

## **5. Применение в нормализации БД**

Аксиомы Армстронга помогают:

1. Находить **минимальный ключ** (набор атрибутов, от которых зависят все остальные).
2. Проверять, находится ли таблица в **нормальной форме (3NF, BCNF)**.
3. Выявлять **избыточные зависимости** для декомпозиции таблиц.

**Пример:**  
Если StudentID→DepartmentStudentID→Department и Department→BuildingDepartment→Building, то таблицу нужно разбить, чтобы избежать транзитивной зависимости.

## **6. Заключение**

* **Аксиомы Армстронга** — основа для анализа функциональных зависимостей.
* **Рефлексивность, пополнение, транзитивность** — три ключевых правила.
* Дополнительные правила (**объединение, декомпозиция, псевдотранзитивность**) упрощают работу с ФЗ.

Эти принципы критически важны для проектирования эффективных и нормализованных баз данных.

Виды аномалий при денормалиазции

### **Виды аномалий при денормализации**

Денормализация базы данных — это намеренное нарушение нормальных форм (1NF, 2NF, 3NF, BCNF) для повышения производительности за счёт избыточности данных. Однако это приводит к **аномалиям** — проблемам при изменении данных.

## **1. Аномалии вставки (Insert Anomalies)**

**Проблема:** Невозможность добавить данные без наличия связанных данных.

**Пример:**  
Таблица Orders\_Products (денормализованная, хранит заказы и информацию о товарах):

| order\_id | customer | product\_id | product\_name | price |
| --- | --- | --- | --- | --- |
| 1001 | Иванов | P50 | Клавиатура | 2000 |

**Аномалия:**

* Нельзя добавить новый товар (product\_name, price) без привязки к заказу (order\_id).
* Придётся либо создавать "пустой" заказ, либо хранить товары в отдельной таблице (нормализованный подход).

## **2. Аномалии обновления (Update Anomalies)**

**Проблема:** Необходимость обновлять одни и те же данные в нескольких местах, что может привести к противоречиям.

**Пример:**  
Таблица Employees\_Departments (сотрудники + отделы):

| emp\_id | emp\_name | dept\_id | dept\_name | dept\_location |
| --- | --- | --- | --- | --- |
| E101 | Петров | D10 | IT | Москва |
| E102 | Сидоров | D10 | IT | Москва |

**Аномалия:**

* Если отдел IT переезжает в СПб, нужно обновить dept\_location у всех сотрудников отдела D10.
* При пропуске хотя бы одной строки данные станут противоречивыми.

## **3. Аномалии удаления (Delete Anomalies)**

**Проблема:** Потеря важных данных при удалении связанных записей.

**Пример:**  
Та же таблица Employees\_Departments:

**Аномалия:**

* Если удалить сотрудника Сидорова (emp\_id = E102), то информация об отделе D10 (IT, Москва) останется.
* Но если удалить последнего сотрудника отдела (Петрова), данные об отделе **исчезнут полностью**, хотя отдел может существовать.

## **4. Проблемы согласованности (Consistency Issues)**

**Проблема:** Избыточность данных ведёт к риску нарушения их целостности.

**Пример:**  
Таблица Book\_Author (денормализованная):

| book\_id | title | author\_id | author\_name |
| --- | --- | --- | --- |
| B001 | Война и мир | A100 | Толстой Л.Н. |
| B002 | Анна Каренина | A100 | Толстой Л.Н. |

**Аномалия:**

* Если изменить author\_name в одной строке (на "Толстой Л."), но забыть в другой — возникнет рассогласование.

## **5. Когда денормализация оправдана?**

Несмотря на аномалии, денормализация применяется для:

1. **Ускорения запросов** (меньше JOIN'ов).
2. **Упрощения отчётов** (данные уже агрегированы).
3. **Кэширования** часто используемых данных.

**Примеры:**

* Ленты новостей (данные пользователя + посты).
* Аналитические базы (OLAP), где важнее скорость, чем актуальность.

## **6. Как минимизировать аномалии?**

1. **Частичная денормализация:** дублировать только критичные для производительности поля.
2. **Триггеры и процедуры:** автоматическое обновление избыточных данных.
3. **Периодическая синхронизация:** например, ночной пересчёт агрегатов.

**Пример:**  
Вместо хранения order\_count в таблице Customers, можно использовать триггер:

sql

Copy

Download

CREATE TRIGGER update\_order\_count

AFTER INSERT ON Orders

FOR EACH ROW

UPDATE Customers SET order\_count = order\_count + 1

WHERE customer\_id = NEW.customer\_id;

## **Вывод**

* **Аномалии вставки, обновления, удаления** — главные недостатки денормализации.
* **Целостность данных** страдает, но производительность растёт.
* **Решение:** баланс между нормализацией и денормализацией + контроль целостности.