Оглавление

[Ответы на вопросы 1](#_Toc201168194)

[1. **Архитектура ANSI-SPARC** 1](#_Toc201168195)

[Уровни архитектуры ANSI-SPARC: 1](#_Toc201168196)

[Преимущества архитектуры ANSI-SPARC: 2](#_Toc201168197)

[2. **Модель "Сущность-Связь" (ER-модель)** 2](#_Toc201168198)

[Основные компоненты ER-модели: 2](#_Toc201168199)

[Классификация сущностей: 2](#_Toc201168200)

[Виды связей: 2](#_Toc201168201)

[3. **Ограничения целостности** 3](#_Toc201168202)

[4. **DDL (Data Definition Language)** 3](#_Toc201168203)

[5. **DML (Data Manipulation Language)** 4](#_Toc201168204)

# Ответы на вопросы

## 1. **Архитектура ANSI-SPARC**

Архитектура ANSI-SPARC (American National Standards Institute - Standards Planning and Requirements Committee) — это концептуальная модель, предложенная для стандартизации архитектуры систем управления базами данных (СУБД). Она состоит из трех уровней:

### Уровни архитектуры ANSI-SPARC:

1. **Внешний уровень (External Level)**:
   * Описывает представление данных для конкретных пользователей или приложений.
   * Каждый пользователь или приложение видит только те данные, которые ему нужны, в удобном формате.
   * Пример: Отчет для бухгалтерии, где выводятся только финансовые данные.
2. **Концептуальный уровень (Conceptual Level)**:
   * Описывает логическую структуру всей базы данных, независимо от физического хранения.
   * Включает все сущности, атрибуты, связи и ограничения.
   * Пример: ER-диаграмма, которая описывает все таблицы и связи между ними.
3. **Внутренний уровень (Internal Level)**:
   * Описывает физическое хранение данных на диске.
   * Включает структуры хранения, индексы, методы доступа и т.д.
   * Пример: Файлы базы данных, индексы B-дерева.

### Преимущества архитектуры ANSI-SPARC:

* **Независимость данных**: Изменения на одном уровне не влияют на другие уровни.
* **Гибкость**: Позволяет адаптировать базу данных под разные требования пользователей.
* **Безопасность**: Пользователи видят только те данные, которые им разрешены.

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

Модель "Сущность-Связь" (Entity-Relationship Model) — это концептуальный инструмент для проектирования баз данных. Она используется для визуализации структуры данных и связей между ними.

### Основные компоненты ER-модели:

1. **Сущность (Entity)**:
   * Объект, который существует в реальном мире и может быть идентифицирован.
   * Пример: Студент, Курс, Преподаватель.
   * Сущности представляются в виде прямоугольников.
2. **Атрибут (Attribute)**:
   * Свойство сущности.
   * Пример: Для сущности "Студент" атрибутами могут быть ID, Имя, Фамилия.
   * Атрибуты представляются в виде овалов.
3. **Связь (Relationship)**:
   * Отношение между сущностями.
   * Пример: Студент "записан" на Курс.
   * Связи представляются в виде ромбов.

### Классификация сущностей:

* **Сильные сущности**: Сущности, которые могут существовать независимо (например, "Студент").
* **Слабые сущности**: Сущности, которые зависят от других сущностей (например, "Оценка" зависит от "Студента" и "Курса").

### Виды связей:

1. **Один к одному (1:1)**:
   * Одна сущность связана с одной сущностью.
   * Пример: Человек и Паспорт.
2. **Один ко многим (1:N)**:
   * Одна сущность связана с несколькими сущностями.
   * Пример: Преподаватель и Курсы.
3. **Многие ко многим (N:M)**:
   * Несколько сущностей связаны с несколькими сущностями.
   * Пример: Студенты и Курсы (студент может записаться на несколько курсов, и на курс может записаться несколько студентов).

## 3. **Ограничения целостности**

Ограничения целостности — это правила, которые обеспечивают корректность данных в базе данных. Они делятся на несколько типов:

1. **Ограничения домена (Domain Constraints)**:
   * Определяют допустимые значения для атрибутов.
   * Пример: Возраст не может быть отрицательным.
2. **Ограничения сущности (Entity Constraints)**:
   * Каждая сущность должна иметь уникальный идентификатор (первичный ключ).
   * Пример: У каждого студента должен быть уникальный ID.
3. **Ограничения ссылочной целостности (Referential Integrity Constraints)**:
   * Обеспечивают корректность связей между таблицами.
   * Пример: Если студент записан на курс, то курс должен существовать.
4. **Пользовательские ограничения (User-Defined Constraints)**:
   * Определяются разработчиком для конкретных бизнес-правил.
   * Пример: Зарплата сотрудника не может превышать зарплату его руководителя.

## 4. **DDL (Data Definition Language)**

DDL — это язык определения данных, используемый для создания и изменения структуры базы данных. Основные команды DDL:

1. **CREATE**:
   * Создает новые объекты (таблицы, индексы, базы данных).
   * Пример:

sql

Copy

CREATE TABLE Students (

ID INT PRIMARY KEY,

Name VARCHAR(50),

Age INT

);

1. **ALTER**:
   * Изменяет структуру существующих объектов.
   * Пример:

sql

Copy

ALTER TABLE Students ADD COLUMN Email VARCHAR(100);

1. **DROP**:
   * Удаляет объекты из базы данных.
   * Пример:

sql

Copy

DROP TABLE Students;

1. **TRUNCATE**:
   * Удаляет все данные из таблицы, но сохраняет её структуру.
   * Пример:

sql

Copy

TRUNCATE TABLE Students;

## 5. **DML (Data Manipulation Language)**

DML — это язык манипуляции данными, используемый для работы с данными в таблицах. Основные команды DML:

1. **SELECT**:
   * Извлекает данные из таблицы.
   * Пример:

sql

Copy

SELECT \* FROM Students;

1. **INSERT**:
   * Добавляет новые строки в таблицу.
   * Пример:

sql

Copy

INSERT INTO Students (ID, Name, Age) VALUES (1, 'Alice', 20);

1. **UPDATE**:
   * Обновляет существующие данные в таблице.
   * Пример:

sql

Copy

UPDATE Students SET Age = 21 WHERE ID = 1;

1. **DELETE**:
   * Удаляет строки из таблицы.
   * Пример:

sql

Copy

DELETE FROM Students WHERE ID = 1;

1. **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.

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;

SELECT nextval('sequence\_1);

### **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.

## **Типы соединения**

### Nested Loop

Описание: Вложенный цикл - базовый метод соединения.

Как работает:

Для каждой строки внешней таблицы:

Для каждой строки внутренней таблицы:

Если условие соединения выполняется -> добавить в результат

Преимущества:

Эффективен для маленьких таблиц

Не требует сортировки или хеширования

Хорош, когда есть индекс для внутренней таблицы

Недостатки:

O(n\*m) сложность для больших таблиц

### Hash Join

Описание: Использует хеш-таблицу для соединения.

Как работает:

Строит хеш-таблицу для меньшей таблицы

Сканирует большую таблицу и ищет совпадения в хеш-таблице

Преимущества:

Эффективен для больших таблиц без индексов

Хорош для равенств (=)

Недостатки:

Требует памяти для хеш-таблицы

Не подходит для неравенств (>, <)

### Merge Join

Описание: Использует сортированные данные для соединения.

Как работает:

Сортирует обе таблицы по ключу соединения

Проходит по обеим таблицам параллельно, находя совпадения

Преимущества:

Эффективен для больших отсортированных данных

Подходит для неравенств (>, <, BETWEEN)

Недостатки:

Требует предварительной сортировки (если данные не отсортированы)

Использует больше памяти

### Parallel Hash Join (Параллельное хеш-соединение)

Как работает:

Рабочие процессы (workers) совместно строят общую хеш-таблицу из одной из таблиц

Затем параллельно сканируют вторую таблицу и ищут совпадения в хеш-таблице

Лидер-процесс собирает результаты от всех рабочих процессов

Преимущества:

Значительное ускорение для больших таблиц (в 4-8 раз на многоядерных системах)

Эффективное использование ресурсов сервера

Хорошо масштабируется с увеличением числа ядер

Недостатки:

Требует больше памяти (каждый worker использует свою копию хеш-таблицы)

Накладные расходы на координацию между процессами

Не эффективен для маленьких таблиц

Когда используется:

Для соединения больших таблиц (обычно > 100,000 строк)

Когда max\_parallel\_workers\_per\_gather > 0 и work\_mem достаточно

Для равенств в условиях JOIN (=)

### Parallel Nested Loop (Параллельные вложенные циклы)

Как работает:

Внешняя таблица разделяется между рабочими процессами

Каждый worker выполняет свой фрагмент Nested Loop

Результаты объединяются лидер-процессом

Преимущества:

Ускорение для CPU-интенсивных операций

Хорошо работает, когда внутренний цикл использует индекс

Эффективен для маленьких/средних таблиц с хорошей селективностью

Недостатки:

Ограниченная масштабируемость (обычно 2-4 workers)

Не эффективен, если внутренний цикл требует полного сканирования

Когда используется:

Для соединения таблиц, где внешняя таблица среднего размера

Когда есть индексы для внутреннего цикла

Для сложных условий, которые нельзя хешировать

### Bitmap Heap Scan + Bitmap Index Scan

Как работает:

Bitmap Index Scan создает битовую карту совпадений по индексу

Bitmap Heap Scan использует эту карту для эффективного чтения нужных страниц таблицы

Битовая карта (bitmap) в PostgreSQL — это компактная структура данных, которая представляет соответствие между индексными записями и физическим расположением данных в таблице с помощью битов (0 и 1).

Применяются дополнительные фильтры (если есть)

Преимущества:

Эффективно комбинирует несколько индексов

Минимизирует случайные чтения с диска

Хорош для сложных условий с AND/OR

Недостатки:

Требует построения битовой карты в памяти

Не всегда оптимален для простых условий

Когда используется:

Для сложных условий фильтрации с несколькими индексами

Когда нужно объединить результаты нескольких индексов

Для частичного совпадения по индексу с дополнительными фильтрами

### Semi Join и Anti Join

Как работает:

Semi Join: Возвращает строки из первой таблицы, где есть хотя бы одно совпадение во второй (без дубликатов)

Anti Join: Возвращает строки из первой таблицы, где нет совпадений во второй

Преимущества:

Оптимизированная работа для EXISTS/NOT EXISTS

Не обрабатывает все совпадения, только первое/отсутствие

Может использовать специальные алгоритмы (Hash/Merge/Nested)

Недостатки:

Ограниченная применимость (только для подзапросов)

Не всегда выбирается оптимальный план

Когда используется:

Для подзапросов с EXISTS/NOT EXISTS

Для IN/NOT IN с большими подзапросами

Когда нужно проверить наличие/отсутствие без полного соединения

### Materialized Join (Материализованное соединение)

Как работает:

Одна из таблиц полностью материализуется во временной структуре

Затем выполняется соединение с материализованной версией

Может комбинироваться с другими методами (Hash, Nested Loop)

Преимущества:

Уменьшает повторные чтения одной таблицы

Полезно для сложных подзапросов

Может ускорить повторяющиеся соединения

Недостатки:

Требует дополнительной памяти

Накладные расходы на материализацию

Когда используется:

Для сложных подзапросов в JOIN

Когда одна таблица используется несколько раз

Для оптимизации общих табличных выражений (WITH)

### LATERAL Join (Латеральное соединение)

Как работает:

Позволяет подзапросу справа ссылаться на столбцы из таблиц слева

Выполняется для каждой строки левой таблицы

Может использовать разные методы соединения внутри

Преимущества:

Гибкость в сложных запросах

Возможность параметризации подзапросов

Поддержка LIMIT внутри соединения

Недостатки:

Может быть неэффективным без правильных индексов

Сложнее для оптимизатора

Когда используется:

Для TOP-N запросов в соединениях

Когда нужно передавать параметры в подзапрос

Для сложных иерархических запросов

### Index Only Join (Только индексное соединение)

Как работает:

Все необходимые данные берутся только из индексов

Полностью избегает чтения табличных данных (heap)

Требует покрывающих индексов (INCLUDE в PostgreSQL 11+)

Преимущества:

Максимальная скорость (только индексные чтения)

Минимальные блокировки

Эффективное использование кэша

Недостатки:

Требует специальных индексов

Ограниченный набор поддерживаемых операций

Когда используется:

Для запросов, которые можно полностью покрыть индексами

В высоконагруженных OLTP-системах

Когда важна скорость чтения

### TID Scan Join (Соединение по идентификаторам кортежей)

Как работает:

Использует физические адреса строк (TID - Tuple ID)

Прямой доступ к данным по физическому расположению

Редко используется в явном виде

Преимущества:

Максимально быстрое обращение к конкретным строкам

Полезно для внутренних операций

Недостатки:

Очень специфическое применение

Хрупкость (TID меняется при VACUUM FULL)

Когда используется:

Для внутренних системных запросов

При работе с курсорами

В некоторых типах блокировок

## **Методы доступа**

### Index Only Scan

Как работает:

СУБД читает данные только из индекса, не обращаясь к таблице (heap)

Возможен когда в SELECT только столбцы, входящие в индекс

Преимущества:

Максимальная скорость (нет обращений к таблице)

Минимальная нагрузка на I/O

### Index Scan

Описание: использует индекс для поиска, но затем обращается к таблице за дополнительными данными.

Как работает:

Находит записи в индексе

По указателям из индекса получает полные строки из таблицы

Преимущества:

Быстрее, чем полное сканирование таблицы

Позволяет эффективно находить конкретные строки

Недостатки:

Требует дополнительных обращений к таблице (heap fetches)

### Bitmap Index Scan + Bitmap Heap Scan

Описание: Двухэтапный процесс для эффективного поиска по нескольким условиям.

Как работает:

Bitmap Index Scan: создает битовую карту совпадений

Bitmap Heap Scan: использует битовую карту для доступа к таблице

Преимущества:

Эффективен для комбинации условий

Минимизирует случайные чтения из таблицы

### Sequential Scan (Seq Scan)

Описание: Полное сканирование таблицы.

Когда используется:

Нет подходящего индекса

Запрашивается большая часть таблицы

Для очень маленьких таблиц

Недостатки:

Медленно для больших таблиц

Высокая нагрузка на I/O

### Tid Scan (Сканирование по физическому адресу строки)

Описание:

Прямой доступ к строке по ее физическому идентификатору (TID — Tuple ID).

Преимущества:

✔ Максимально быстрое чтение конкретной строки.

Недостатки:

✖ Крайне редко используется в обычных запросах.

✖ TID меняется после VACUUM FULL.

Когда используется:

Внутренние операции PostgreSQL (например, обновление строк).

Запросы с ctid (физический адрес строки).

Пример:

sql

EXPLAIN SELECT \* FROM users WHERE ctid = '(123, 5)';

### Parallel Scan (Параллельное сканирование)

Описание:

Распределяет сканирование таблицы или индекса между несколькими worker-процессами.

Преимущества:

✔ Ускорение для больших таблиц (в 2-8 раз на многоядерных CPU).

Недостатки:

✖ Накладные расходы на координацию процессов.

✖ Требует настройки (max\_parallel\_workers\_per\_gather).

Когда используется:

Большие Seq Scan/Index Scan (обычно >100K строк).

Аналитические запросы (COUNT, SUM, GROUP BY).

## **Типы индексов**

### 1. B-Tree (Балансированное дерево)

Как работает:

Хранит данные в сбалансированной древовидной структуре

Поддерживает сортировку значений

Обеспечивает поиск за O(log n)

Для чего подходит:

Стандартные сравнения: =, >, <, >=, <=, BETWEEN

LIKE 'prefix%' (но не LIKE '%suffix')

IS NULL, IS NOT NULL

Композитные индексы (многоколоночные)

Особенности:

sql

CREATE INDEX idx\_name ON table\_name (column1, column2);

Поддерживает уникальность (UNIQUE CONSTRAINT)

Оптимален для OLTP-нагрузки

### 2. Hash (Хеш-индекс)

Как работает:

Создает хеш-таблицу значений

Быстрый поиск точного совпадения (O(1) в идеальном случае)

Для чего подходит:

Только операторы равенства (=)

Точечные запросы без диапазонов

Особенности:

sql

CREATE INDEX idx\_name ON table\_name USING HASH (column);

Не поддерживает сортировку

Не поддерживает уникальность до PostgreSQL 10

Требует перестройки при больших изменениях данных

### 3. GiST (Generalized Search Tree)

Как работает:

Обобщенная структура для различных типов данных

Позволяет реализовать собственные стратегии поиска

Для чего подходит:

Геометрические данные (близость, пересечение)

Полнотекстовый поиск

Диапазоны (range types)

Иерархические данные (ltree)

Пример:

sql

CREATE INDEX idx\_gist ON table USING GiST (geom\_column);

### 4. SP-GiST (Space-Partitioned GiST)

Как работает:

Разбивает пространство на непересекающиеся области

Альтернатива GiST для определенных типов данных

Для чего подходит:

IP-адреса (inet, cidr)

Геометрические данные (особые виды разбиения)

Текст (различные стратегии разбиения)

Пример:

sql

CREATE INDEX idx\_spgist ON table USING SPGiST (ip\_column);

### 5. GIN (Generalized Inverted Index)

Как работает:

Инвертированный индекс (хранит mapping: значение → строки)

Оптимизирован для составных значений

Для чего подходит:

JSON/JSONB данные

Полнотекстовый поиск

Массивы (contains, overlap)

hStore (ключ-значение)

Пример:

sql

CREATE INDEX idx\_gin ON table USING GIN (jsonb\_column);

### 6. BRIN (Block Range INdex)

Как работает:

Хранит метаданные о диапазонах физических блоков

Крайне компактен

Для чего подходит:

Очень большие таблицы с коррелированной физической и логической сортировкой

Временные ряды (timestamp, последовательные ID)

Данные, которые редко изменяются

Пример:

sql

CREATE INDEX idx\_brin ON table USING BRIN (timestamp\_column);

### 7. Bloom (Фильтр Блума)

Как работает:

Вероятностная структура данных

Быстро проверяет "возможное наличие" элемента

Для чего подходит:

Многоколоночные запросы с большим количеством комбинаций

Когда важнее скорость, чем точность

Особенности:

sql

CREATE EXTENSION bloom;

CREATE INDEX idx\_bloom ON table USING bloom (col1, col2, col3);

Могут быть ложные срабатывания

Компактнее, чем B-Tree для многих столбцов

### 8. RUM (расширение GIN)

Как работает:

Улучшенная версия GIN

Хранит дополнительную информацию о позициях токенов

Для чего подходит:

Фразовый поиск

Ранжирование в полнотекстовом поиске

Более сложные операции с массивами

Пример:

sql

CREATE EXTENSION rum;

CREATE INDEX idx\_rum ON table USING rum (text\_column rum\_tsvector\_ops);

### 9. Partial (Частичный индекс)

Как работает:

Индексирует только подмножество строк

Содержит условие WHERE

Для чего подходит:

Когда интересна только часть данных

Уменьшает размер индекса

Ускоряет запросы к определенным подмножествам

Пример:

sql

CREATE INDEX idx\_partial ON table (column) WHERE status = 'active';

### 10. Covering (Покрывающий индекс)

Как работает:

Включает в индекс дополнительные столбцы

Позволяет Index Only Scan

Для чего подходит:

Частые запросы, выбирающие определенные столбцы

Избегание обращений к таблице

Пример:

sql

CREATE INDEX idx\_covering ON table (id) INCLUDE (name, email);

### 11. Functional (Функциональный индекс)

Как работает:

Индексирует результат функции

Вычисляемое значение хранится в индексе

Для чего подходит:

Запросы с функциями в условиях

Нормализация данных перед индексацией

Пример:

sql

CREATE INDEX idx\_func ON table (lower(email));

## **Прочие вопросы**

### Heap Blocks

Ключевые характеристики:

Размер по умолчанию: 8KB (8192 байта)

Содержит:

Заголовок блока (24 байта)

Массив указателей на строки (ItemId array)

Фактические данные строк (кортежи)

Свободное пространство

Как PostgreSQL использует Heap Blocks?

1. Структура блока

+-----------------------+

| Заголовок | (24 байта)

+-----------------------+

|ItemId array | (массив указателей на строки)

+-----------------------+

|Свободное место|

+-----------------------+

| Кортежи | (фактические данные строк)

### Buckets (Корзины/Бакеты)

Buckets - это метод организации данных путем их распределения в "корзины" на основе хеш-функции или диапазонов значений.

Применение Buckets:

Хеш-партиционирование:

sql

CREATE TABLE sales (

id serial,

sale\_date date,

amount numeric

) PARTITION BY HASH (id);

Ускорение JOIN операций:

Данные с одинаковым хешем попадают в одну корзину

Соединение происходит только между соответствующими корзинами

Группировка в аналитических запросах:

sql

SELECT width\_bucket(amount, 0, 1000, 10) AS bucket, COUNT(\*)

FROM sales

GROUP BY bucket;

Преимущества Buckets:

Уменьшение объема обрабатываемых данных

Параллельная обработка разных корзин

Равномерное распределение данных

### Batches (Пакетная обработка)

Что такое Batches?

Batches - это группировка операций в пакеты для более эффективной обработки.

Применение Batches:

Пакетные вставки:

sql

INSERT INTO table VALUES (1), (2), (3); -- вместо 3 отдельных INSERT

Пакетные обновления:

sql

UPDATE table SET col = new\_val WHERE id IN (1, 2, 3);

Чтение данных порциями:

python

# Пример на Python с psycopg2

cursor.execute("SELECT \* FROM large\_table")

while True:

batch = cursor.fetchmany(1000)

if not batch:

break

process\_batch(batch)

Преимущества Batches:

Снижение накладных расходов на коммуникацию

Лучшее использование кэша CPU

Уменьшение количества транзакций

### Когда индексы неэффективны или невыгодны?

3.1. Частые операции записи

Проблема:

Каждое INSERT/UPDATE/DELETE требует обновления индексов

При высокой частоте изменений индексы становятся "узким местом"

Решение:

Уменьшить количество индексов для часто изменяемых таблиц

Использовать индексы только для часто читаемых колонок

3.2. Большие таблицы с низкой селективностью

Проблема:

Если условие выбирает >15-20% строк таблицы

Seq Scan может быть быстрее Index Scan + Heap Fetches

Пример:

sql

-- Неэффективно для индекса, если статусов немного:

SELECT \* FROM orders WHERE status IN ('new', 'processing', 'completed');

3.3. Очень маленькие таблицы

Проблема:

Накладные расходы на использование индекса могут превысить выгоду

PostgreSQL часто выбирает Seq Scan для таблиц < 10-20 страниц

### Что такое Recheck Cond?

Recheck Cond (повторная проверка условия) — это этап выполнения запроса в PostgreSQL, когда система после первоначального отбора строк по индексу должна дополнительно проверить условие непосредственно на данных таблицы (heap).

Как это работает?

Первичная проверка: PostgreSQL использует индекс (обычно Bitmap Index Scan) для быстрого нахождения потенциально подходящих строк.

Построение битовой карты: Создается карта страниц (blocks), где могут находиться нужные строки.

Recheck Cond: При чтении данных из этих страниц PostgreSQL перепроверяет условие, так как:

Индекс может дать ложноположительные срабатывания

Условие может содержать часть, не покрытую индексом

## **Режимы кэширования (Cache Modes)**

### 1.1. Write-Through (Сквозная запись)

Как работает:

Данные записываются одновременно в кэш и основное хранилище

Обеспечивает согласованность данных

Преимущества:

Высокая надежность данных

Простота реализации

Недостатки:

Более медленные операции записи

Увеличенная нагрузка на основное хранилище

Использование:

Системы, где критична сохранность данных

Финансовые транзакции

1.2. Write-Back (Отложенная запись)

Как работает:

Данные сначала записываются в кэш

Запись в основное хранилище происходит позже (асинхронно)

Преимущества:

Быстрые операции записи

Снижение нагрузки на основное хранилище

Недостатки:

Риск потери данных при сбое

Сложнее обеспечить согласованность

Использование:

Высоконагруженные системы

Когда производительность важнее мгновенной согласованности

1.3. Write-Around (Обход кэша)

Как работает:

Данные записываются напрямую в хранилище, минуя кэш

Кэшируются только при чтении

Преимущества:

Не засоряет кэш одноразовыми записями

Эффективен для данных с однократной записью и редким чтением

Недостатки:

Первое чтение будет медленным

Использование:

Логирование

Архивные данные

1.4. Read-Through (Чтение через кэш)

Как работает:

При промахе кэша система автоматически загружает данные из хранилища

Прозрачно для приложения

Преимущества:

Упрощает логику приложения

Автоматическое заполнение кэша

Недостатки:

Первое обращение к данным может быть медленным

1.5. Cache-Aside (Lazy Loading)

Как работает:

Приложение явно управляет кэшем

При промахе приложение само загружает данные в кэш

Преимущества:

Гибкость управления

Можно реализовать сложную логику кэширования

Недостатки:

Более сложная реализация

Возможность кэширования устаревших данных

## Алгоритмы вытеснения данных их кэша (Eviction Policies)

### 2.1. LRU (Least Recently Used)

Принцип работы:

Вытесняет редко используемые элементы

Поддерживает очередь по времени последнего доступа

Плюсы:

Эффективен для большинства шаблонов доступа

Простая реализация

Минусы:

Не учитывает частоту использования

Уязвим к сканированию (scan resistance)

### 2.2. LFU (Least Frequently Used)

Принцип работы:

Вытесняет наименее часто используемые элементы

Подсчитывает количество обращений к каждому элементу

Плюсы:

Хорош для часто используемых "горячих" данных

Минусы:

Может задерживать устаревшие данные

Дополнительные накладные расходы

### 2.3. FIFO (First-In, First-Out)

Принцип работы:

Вытесняет самые старые элементы (по времени добавления)

Плюсы:

Очень простая реализация

Низкие накладные расходы

Минусы:

Низкая эффективность для большинства сценариев

### 2.4. LIRS (Low Inter-reference Recency Set)

Принцип работы:

Комбинирует LRU с учетом "холодных" элементов

Более устойчив к сканированию, чем LRU

Плюсы:

Лучшая производительность для некоторых workload'ов

Хорошая scan resistance

Минусы:

Более сложная реализация

### 2.5. ARC (Adaptive Replacement Cache)

Принцип работы:

Адаптивно балансирует между LRU и LFU

Поддерживает два списка: недавние и частые элементы

Плюсы:

Автоматическая адаптация к шаблонам доступа

Высокая эффективность

Минусы:

Высокие накладные расходы

Сложная реализация

## Переполнение кэша (Cache Overflows)

3.1. Причины переполнения:

Недостаточный размер кэша

Неоптимальная политика вытеснения

Изменение паттернов доступа к данным

Утечки памяти в реализации

3.2. Методы обработки переполнения:

Динамическое масштабирование:

Автоматическое увеличение размера кэша

Используется в распределенных системах

Сегментирование кэша:

Разделение кэша на независимые части

Разные политики для разных сегментов

Гибридные стратегии:

Комбинация нескольких алгоритмов вытеснения

Адаптивное переключение между политиками

Интеллектуальное предварительное вытеснение:

Прогнозирование будущих обращений

Заблаговременное освобождение места