

# Введение в базы данных

Конспект основан на лекциях Корнеева Георгия Александровича\*

## Содержание

<b>1 Теория</b>	<b>4</b>
1.1 Развитие баз данных . . . . .	5
1.1.1 Простые и структурированные файлы . . . . .	5
1.1.2 Файловая система . . . . .	5
1.1.3 Иерархическая модель данных . . . . .	6
1.1.4 Сетевая модель данных . . . . .	6
1.1.5 Реляционная модель данных . . . . .	7
1.1.6 Объектные базы данных . . . . .	7
1.1.7 NoSQL (Not only SQL) . . . . .	8
1.2 Архитектура РСУБД . . . . .	9
1.3 Современные РСУБД . . . . .	11
1.3.1 Корпоративные . . . . .	11
1.3.2 Свободные . . . . .	11
1.3.3 Встраиваемые . . . . .	12
1.4 Модель сущность-связь . . . . .	13
1.4.1 Сущность . . . . .	13
1.4.2 Связь . . . . .	13
1.4.3 Ассоциации . . . . .	14
1.4.4 Слабая сущность . . . . .	15
1.5 Реляционная алгебра. Предназначение и свойства . . . . .	16
1.6 Реляционная алгебра. Унарные и множественные операции . . . . .	18
1.6.1 Проекция . . . . .	18
1.6.2 Фильтрация . . . . .	19
1.6.3 Переименование . . . . .	21
1.6.4 Множественные операции . . . . .	21
1.7 Реляционная алгебра. Соединения . . . . .	24
1.7.1 Полное соединение . . . . .	24
1.7.2 Естественное соединение . . . . .	24
1.7.3 Внешнее соединение . . . . .	25

---

\*kgeorgiy.info

1.7.4	Левое и правое соединения . . . . .	26
1.7.5	Полусоединения . . . . .	26
1.7.6	Условные соединения . . . . .	27
1.8	Реляционная алгебра. Деление и операции над данными . . . . .	29
1.8.1	Деление . . . . .	29
1.8.2	Большое деление . . . . .	29
1.8.3	Расширение . . . . .	30
1.8.4	Агрегирование . . . . .	30
1.9	Исчисление кортежей и его реляционная полнота . . . . .	32
1.9.1	Реляционное исчисление . . . . .	32
1.9.2	Исчисление кортежей . . . . .	32
1.9.3	Связи реляционной алгебры и исчисления кортежей . . . . .	34
1.10	Исчисление доменов и его реляционная полнота . . . . .	37
1.10.1	Исчисление доменов . . . . .	37
1.10.2	Связи реляционной алгебры и исчисления доменов . . . . .	38
1.11	Datalog и рекурсия . . . . .	40
1.11.1	Datalog . . . . .	40
1.11.2	Рекурсивные запросы . . . . .	42
1.11.3	Связь реляционного исчисления и SQL . . . . .	43
1.11.4	Подзапросы . . . . .	43
1.11.5	Рекурсия . . . . .	45
1.12	Представления и их обновление . . . . .	46
1.12.1	Определение представлений . . . . .	46
1.12.2	Обновление представлений . . . . .	46
1.12.3	Материализованные представления . . . . .	47
1.13	Транзакции. Восстановление. Классический алгоритм . . . . .	49
1.13.1	Транзакции . . . . .	49
1.13.2	Восстановление . . . . .	50
1.13.3	Классический алгоритм восстановления . . . . .	52
1.13.4	Отказ оборудования . . . . .	53
1.14	Транзакции. Восстановление. Алгоритм ARIES . . . . .	55
1.14.1	Алгоритм восстановления ARIES . . . . .	55
1.14.2	Сравнение алгоритмов . . . . .	56
1.15	Транзакции. Параллельное исполнение. Блокировки. . . . .	57
1.15.1	Параллельное исполнение . . . . .	57
1.15.2	Блокировки . . . . .	58
1.15.3	Восстановление и параллелизм . . . . .	60
1.15.4	Гранулярность блокировок . . . . .	60
1.16	Транзакции. Параллельное исполнение. Уровни изоляции. . . . .	61
1.16.1	Упорядочиваемый . . . . .	61
1.16.2	“Слепок” . . . . .	61
1.16.3	Повторяемое чтение . . . . .	62
1.16.4	Чтение зафиксированных . . . . .	62
1.16.5	Чтение незафиксированных . . . . .	62
1.17	Секционирование . . . . .	63
1.17.1	Вертикальное секционирование . . . . .	63

1.17.2 Горизонтальное секционирование . . . . .	64
1.18 Репликация . . . . .	68
1.18.1 Реализация репликации . . . . .	68
1.18.2 Применения репликации . . . . .	69
1.19 Распределенные транзакции . . . . .	71
1.20 Распределенные базы данных. Цели и проблемы . . . . .	74
1.20.1 Цели распределения . . . . .	74
1.20.2 Проблемы распределения . . . . .	75
1.21 Трактовки null и операции с ним . . . . .	79
1.21.1 Трактовки null . . . . .	79
1.21.2 Операции с null . . . . .	79
1.21.3 Операции с null в SQL . . . . .	83
<b>2 Практика</b>	<b>84</b>

# 1 Теория

## 1.1 Развитие баз данных

### 1.1.1 Простые и структурированные файлы

Простые файлы состоят из:

- Заголовок – название столбцов.
- Данные – значения, разделённые запятой.

В структурированных файлах в заголовках написано не только название столбца, но и его тип и длина.

**Замечание.** В структурированной версии можно быстро искать запись по номеру, то есть прочитать заголовок и узнать сколько занимает одна запись, умножить на нужный номер и прочесть сразу нужную запись.

Достоинства:

- Простота чтения – написать код который будет читать такие данные просто.
- Сложность поиска – не реализовать эффективный поиск которому не нужно было бы загружать всё в память.
- Сложность обработки.
- Сложно хранить нетривиальные типы данных например даты - теряется информация на какой позиции месяц, на какой день.
- Нет проверки целостности (ограничений).

### 1.1.2 Файловая система

Устройство:

- Файл – одна запись.
- Иерархия записей – иерархия каталогов.

Достоинства:

- Простота реализации.
- Структурированные данные.

Недостатки:

- Сложно извлекать требуемые данные.
- Нет проверки целостности.
- Большое количество файлов.

### 1.1.3 Иерархическая модель данных

**Замечание.** Иерархия это хорошо, но использовать для этого файловую систему не эффективно.

**Деревья** Отношение родитель – ребёнок соответствует каталогу и его подкаталогам в файловой системе, но не будет выделяться по файлу для каждой записи, вместо этого записи с одинаковым типом будут группироваться (благодаря этому не нужно будет лишний раз обходить файловую систему).

Достоинства:

- Проверка целостности появляется благодаря структурированности (а именно связи родитель - ребёнок), например можно проверять что у человека нет двух оценок по одному предмету (хотя в файловой системе тоже можно было это делать).
- Последовательное расположение записей - ускорение выполнения запросов.

Недостатки:

- Представление только древовидных структур.
- Нет отношения многие ко многим, например у множества студентов есть множество оценок по разным предметам и родителем будет студент, а детьми оценки или наоборот, запросы к обоим этим множествам выполняться эффективно не могут.

### 1.1.4 Сетевая модель данных

Обобщение иерархических баз данных, нет единой строгой иерархии, есть базовая иерархия и есть дополнительные иерархии вида владелец – запись

Достоинства:

- Представление всех типов связей (в том числе многие-ко-многим).
- Возможность описания структуры.
- Эффективность реализации – эффективные запросы к обоим мн-вам из связи многие ко многим, но эффективность разная из-за последовательной записи, только записи базовой иерархии записаны последовательно.

Недостатки:

- Более сложная реализация.
- Жесткое ограничение структуры – если мы не подумали о каком то виде запросов заранее, то возможно для его исполнения придётся поднять все данные.

### 1.1.5 Реляционная модель данных

**Хранение** Данные хранятся в таблицах, также в таблицах хранится информация о связях, связи задаются в запросах.

Достоинства:

- Представление всех типов связей-
- Гибкая структура данных – можно задавать произвольные запросы.
- Математическая модель – позволяет говорить что некоторые запросы эквивалентны, то есть запрос не обязан исполняться как написан, мб исполнен любой эквивалентный запрос, выбирается самый эффективный из эквивалентных и получается тот же самый результат что и при исходном запросе потому что запросы эквивалентны.

Недостатки:

- Сложность реализации.
- Сложность представления иерархических данных.
- Сложность составления эффективных запросов.

### 1.1.6 Объектные базы данных

Цель – хранить граф объектов, который уже находится в памяти, в базе данных. Обычная реализация – слой трансляции в реляционную базу данных

Достоинства:

- Работа в терминах объектов а не записей.
- Логичное направление ссылок, например можем легко взять все оценки студента потому что есть соответствующее отображение из студента в оценки.

Недостатки:

- Сложность реализации.
- Сложность миграции схемы, например добавление поля объекту, в базе уже есть объекты без этого поля.
- Малая распространенность.

### 1.1.7 NoSQL (Not only SQL)

**Основная мысль** Реляционные базы данных умеют слишком много – они заточены чтобы работать одинаково эффективно в куче различных сценариев, а если у нас какой то один сценарий, то можно оптимизировать ровно для него и написать эффективней.

Различные типы:

- Документ-ориентированне – есть куча документов, важно что внутри них, главное уметь их быстро искать.
- Ключ-значение – всё что предоставляет движок - быстро по ключу достать значение.
- Табличные и столбчатые – хранить таблицы по столбцам, так если у нас множество запросов к конкретным двум столбцам, то мы сможем прочитать только их, читать придётся дважды (каждый столбец отдельно читается), но зато не нужно читать все столбцы как в табличном подходе.
- Графовые – хотим хранить графы.

Достоинства:

- Большой выбор – отказываемся почти от всего кроме одного, у чего получаем большую производительность.
- Гибкость – в момент разработки базы, не тогда когда уже есть база.
- Скорость работы.

Недостатки:

- Множество вещей делается в коде.
- Нет стандартных оптимизаторов.
- Легко ошибиться.



## 1.2 Архитектура РСУБД

Есть программа, есть данные, и программа обращается к данным.

**Протокол** Для взаимодействия нужен протокол, его реализуют драйвера, которые находятся и на стороне программы, и на стороне СУБД, которая уже будет обращаться в хранилище. Могут быть различные протоколы, традиционно у каждого СУБД есть свой протокол, по которому можно с ней общаться

**Замечание.** СУБД и хранилище находятся на одном компьютере, благодаря этому нет передачи данных по сети во время исполнения запроса (если хранилище представляет из себя несколько компьютеров то взаимодействие по сети всё же есть).

### Запрос

1. Стандартный SQL запрос требуется разобрать, для этого есть модуль *Разборщик запроса*, который представляет из себя парсер.
2. *Исполнитель запроса* исполняет запрос, но так как исполнять ровно тот запрос который написан не эффективно существует *Оптимизатор*.
3. *Посторитель плана исполнения* или *Оптимизатор* берёт разобранный запрос и решает как он будет исполняться: какие, откуда и в каком порядке будут загружаться данные, какие индексы будут использованы.
4. Управление памятью - это важно для исполнителя, потому что от того поместятся все данные в память или нет зависит эффективная реализация запроса.
5. Статистика. Например нам нужно прочитать данные о всех студентах конкретного пола, либо мальчиков, либо девочек, тогда имея статистику и том что их кол-во отличается на порядок оптимизатор может понять что всех мальчиков будет быстрее прочитать просто читая все данные подряд, а девочек, возможно при наличии способа быстро идентифицировать именно девочек, читая данные только о них.



Рис. 1: Полноценная схема

## 1.3 Современные РСУБД

### 1.3.1 Корпоративные

**Корпоративные СУБД** предназначены для продажи большим корпорациям, но у большинства таких РСУБД есть разработческие лицензии, которые позволяют использовать их в ограниченной среде (ограничение на количество ядер и размер памяти).

- Oracle (Oracle)
  - Высокая пропускная способность (throughput).
  - Невысокая скорость обновления (latency).

**Замечание.** Два утверждения выше не противоречат друг другу, хотя каждый запрос и исполняется медленно, пропускная способность получается большой за счёт того, что конкретная СУБД заточена на исполнение тысяч параллельных запросов, и суммарная пропускная способность всех этих запросов, а не отдельных запросов, будет высокой.

- DB2 (IBM)
  - Ориентация на «большие» машины, то есть с точки зрения IBM, СУБД это не приложение, которое крутится на сервере, а отдельное железо.
  - Мало распространена в России, так как развивалась в 60'е - 80'е годы предыдущего века.
  - Неполная совместимость с SQL.
- SQL Server (Microsoft)
  - Работа под Windows.
  - Масштабируемость (путём добавления новых процессоров).

### 1.3.2 Свободные

- MySQL
  - Поддерживаются различные форматы хранения БД.
  - Неполная поддержка SQL.
  - Есть enterprise и community версии.
- PostgreSQL
  - Непосредственная поддержка связей – СУБД достаточно стабильна, чтобы использовать в реальных проектах.
  - Объектные расширения – но в то же время эта СУБД – экспериментальный проект, в который добавляется куча различных возможностей, некоторые из которых не выходят из экспериментального статуса.

- Firebird
  - Была очень популярна когда делалась Borland'ом под Delphi.
  - Используется только в старых проектах, так как в БД, которые используют это СУБД, есть данные, которые нельзя потерять, а перенести их очень сложно.

### 1.3.3 Встраиваемые

- SQLite
  - Компактна, поэтому много используется на мобильных устройствах.
  - In-memory mode – все данные должны поместиться в память.
  - Ограниченная реализация SQL-92.
- Apache Derby
  - In-memory mode – умеет быть полностью in-memory, а также умеет работать с данными которые в память не поместились.
  - Хорошо совместим с DB2, так как был проектом IBM'а, и из-за этого же не очень хорошо совместим со всеми остальными.
  - Pure Java – встраивается в любое Java приложение.
- HyperSQLDB
  - Pure Java.
  - Не поддерживает транзакции.
  - In-memory mode.
  - В основном используется для тестирования.
- Access
  - Совмещение СУБД и RAD.
  - Встраиваемые приложения.

**Замечание.** In-memory базы данных хорошо подходят для тестирования, потому что каждый пользователь может легко поднять свой instance из-за того что база in-memory и это всё ещё SQL, и каждому из instance'ов не будут мешать тесты других пользователей, также нет проблем с тем что схема данных может быть старой.

## 1.4 Модель сущность-связь

### 1.4.1 Сущность

У сущности есть имя и атрибуты, атрибут представляет из себя имя, домен и свойства.

- Домен – информация о тех данных которые хранятся в атрибуте, конкретный физический тип будет выбран позже.
- Свойства – обязательный/не обязательный, основной ключ (Pk)/дополнительный ключ (Kn).

На рисунке 2 приведена иллюстрация сущности.



Рис. 2: Иллюстрация сущности

### 1.4.2 Связь

**Связь** Связывает несколько сущностей, имеет имя и тип. Тип связи обозначается на её концах.

На рисунке 3 приведены типы концов связей.



Рис. 3: Типы концов

**Замечание.** Тут кривой рисунок, 'Один' - должна быть просто прямая линия, без значка обязательности.

На рисунке 4 приведены типы связей.



Рис. 4: Примеры связей

### 1.4.3 Ассоциации

**Ассоциация** – это некоторое обобщение связи, два типа обобщения:

1. Нагрузить связь, задать ей дополнительные свойства.
2. Сделать многосторонней, больше чем два конца.

На рисунке 5 приведен пример ассоциации.



Рис. 5: Пример ассоциации

**Замечание.** У ассоциаций, в отличие от обычных связей, есть название и атрибуты, но нет ключей, иначе они стали бы сущностями.

**Замечание.** Из-за того что у ассоциаций нет ключей, у них нет связей между собой.

**Ограничения по Чену** (look-across, Chen-like), если зафиксировать все сущности кроме одной, то получим ограничения на оставшуюся.

**Ограничения по Мерис** (look-here Merise-like) – ограничение непосредственно на сущность.

На рисунке 6 изображена иллюстрация обобщения ограничений:



Рис. 6: Generic

#### 1.4.4 Слабая сущность

Сущность, на которую можно сослаться, но у неё недостаточно атрибутов для её идентификации. Поэтому для слабой сущности вводится понятие идентифицирующей связи (которые рисуются двойной линией).

## 1.5 Реляционная алгебра. Предназначение и свойства

Базы данных нужно уметь не только проектировать, но и использовать. Существует несколько способов формулировать запросы. Первый из рассматриваемых – *реляционная алгебра*.

**Мотивация** Действительно, в базах данных можно не только хранить данные, но и делать выборки, изменять их каким-либо образом. Для этого вводится понятие запроса. При первом рассмотрении, запросы нужны как минимум для выполнения следующих действий:

- Выборка данных: получить данные из базы, чтобы тем или иным способом обрабатывать их уже извне.
- Область действия обновлений: запросы позволят указывать область действия тех или иных операций, что крайне полезно. Например, к таким операциям относятся операции удаления или изменения данных: хочется указывать, на какие именно записи эти операции подействуют.
- Ограничения целостности: до сих пор было только два вида ограничений (ключи и внешние ключи). Некоторые базы данных позволяют создавать произвольные ограничения целостности, заданные на поддерживаемом языке. В рамках этих ограничений очень удобно пользоваться запросами.
- Ограничения доступа.

**Определение.** *Реляционная алгебра* – алгебра над множеством всех отношений.

Далее будут определены некоторые из операций (которые по определению должны быть замкнуты над носителем), и ограничения, которые им соответствуют. В целом, реляционная алгебра – императивный язык для работы с отношениями, который позволяет в явном виде, по действиям, описать, каким именно образом должен быть получен результат.

**Примеры** Рассмотрим несколько простых примеров операций в рамках реляционной алгебры.

- Проекция отношения на множество атрибутов:  $\pi_A(R)$ ;
- Естественное соединение  $R_1 \bowtie R_2$ .

**Замечание.** Как уже говорилось, все операции в рамках алгебры замкнуты по определению. Это означает, что их можно комбинировать произвольным образом (при сохранении условий на возможность исполнения операции). Например:  $\pi_A(R_1 \bowtie \pi_B(R_2)) \bowtie R_3$ .



**Операции** В текущем контексте полезно уточнить, что именно понимается под операцией над отношениями в рамках реляционной алгебры. А именно, для того, чтобы определить операцию, необходимо определить следующее:

- Правило построения заголовка по заданным отношениям;
- Правило построения тела по заданным отношениям;
- Условия, при которых операция выполнима, то есть ограничения на отношения, к которым она применяется.

## 1.6 Реляционная алгебра. Унарные и множественные операции

В этом разделе будут описаны унарные операции в рамках реляционной алгебры. В соответствии с определением, для определения каждой операции нужно указать способ построения заголовка, тела отношения, а также условия применимости, если такие есть.

### 1.6.1 Проекция

**Определение.** *Проекцией* отношения  $R$  на множество атрибутов  $A = \{a_1, a_2, \dots, a_n\}$  называется отношение, полученное из исходного путем удаления атрибутов не из  $A$ . Обозначается  $\pi_A(R)$ .

Данная операция может быть полезна для следующего:

- Привести отношение к виду, в котором над ним можно будет осуществить другую операцию (например, объединение);
- Выбрать из отношения только нужные данные (для выборки).

На рисунке 7 приведена иллюстрация к определению  $\pi_{A_2, A_4, A_5}(A)$ .

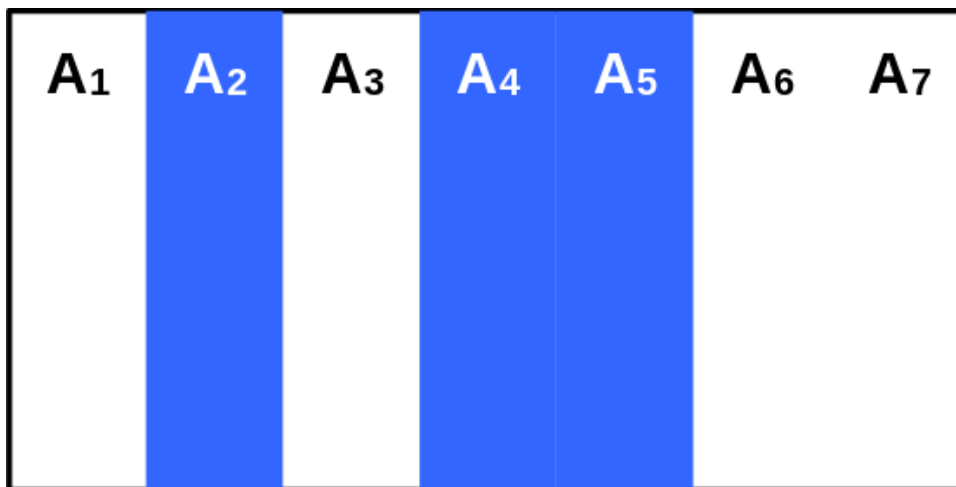


Рис. 7: Иллюстрация к определению проекции

Синим здесь обозначены столбцы, которые есть в результате операции. Остальные столбцы не используются, и результат никак не зависит от их содержимого.

**Примеры** Приведем несколько тривиальных примеров применения проекции.

- $\pi_{FirstName, LastName}$



Рис. 8: Проекция. Пример 1

- $\pi_{FirstName}$



Рис. 9: Проекция. Пример 2

### 1.6.2 Фильтрация

**Определение.** *Фильтрацией (селекцией, выборкой из)* отношения  $R$  называется отношение, чей заголовок полностью совпадает с заголовком  $R$ , но тело содержит только кортежи, удовлетворяющее условию  $s$ . Обозначение:  $\sigma_c(R)$ .

Операция часто используется для

- Ограничения области действия изменяющих запросов;
- Получения выборки данных, соответствующих определенному условию.

На рисунке 10 приведена иллюстрация к определению  $\sigma_c(R)$ .



Рис. 10: Иллюстрация к определению фильтрации

Синим здесь обозначены столбцы, которые есть в результате операции. Остальные столбцы не используются, и результат никак не зависит от их содержимого.

**Примеры** Приведем несколько тривиальных примеров применения фильтрации.

- $\sigma_{Id>2}$

Id	LastName	FirstName
1	Иванов	Иван
2	Петров	Петр
3	Сидоров	Сидор
4	Кулебякин	Иван

➔

Id	LastName	FirstName
3	Сидоров	Сидор
4	Кулебякин	Иван

Рис. 11: Фильтрация. Пример 1

- Можно писать более сложные условия.  $\sigma_{Id>2 \wedge FirstName=Иван}$

Id	LastName	FirstName
1	Иванов	Иван
2	Петров	Петр
3	Сидоров	Сидор
4	Кулебякин	Иван

➔

Id	LastName	FirstName
4	Кулебякин	Иван

Рис. 12: Фильтрация. Пример 2

- Можно использовать функции, доступные в используемой БД.

$\sigma_{\text{length}(FirstName)+2 \geq \text{length}(LastName)}$

Id	LastName	FirstName
1	Иванов	Иван
2	Петров	Петр
3	Сидоров	Сидор
4	Кулебякин	Иван



Id	LastName	FirstName
1	Иванов	Иван
2	Петров	Петр
3	Сидоров	Сидор

Рис. 13: Фильтрация. Пример 3

### 1.6.3 Переименование

**Определение.** *Переименованием* называется операция, при которой меняются названия атрибутов отношения. Тело при этом остается неизменным.

Операция часто применяется для того, чтобы отношение можно было использовать в рамках другой операции (например, при объединении с другим отношением).

**Примеры** Ниже приведен тривиальный пример-пояснение для операции переименования.

- $\rho_{Name=FirstName, Surname=LastName}$

Id	LastName	FirstName
1	Иванов	Иван
2	Петров	Петр
3	Сидоров	Сидор
4	Кулебякин	Иван



Id	Surname	Name
1	Иванов	Иван
2	Петров	Петр
3	Сидоров	Сидор
4	Кулебякин	Иван

Рис. 14: Переименование. Пример

### 1.6.4 Множественные операции

Из теории множеств в реляционную алгебру естественным образом переходят операции:

- $R_1 \cup R_2$  – объединение.
- $R_1 \cap R_2$  – пересечение.
- $R_1 \setminus R_2$  – разность.

Эти операции по определению применимы только к отношениям с одинаковыми заголовками. В результате получается отношение с таким же заголовком и телом, полученным в соответствии с множественной операцией. Иначе говоря, заголовок остается тем же, а над телами отношений производится соответствующая множественная операция (объединение, пересечение, вычитание и прочие).

## Примеры

- Объединение отношений:  $R_1 \cup R_2$

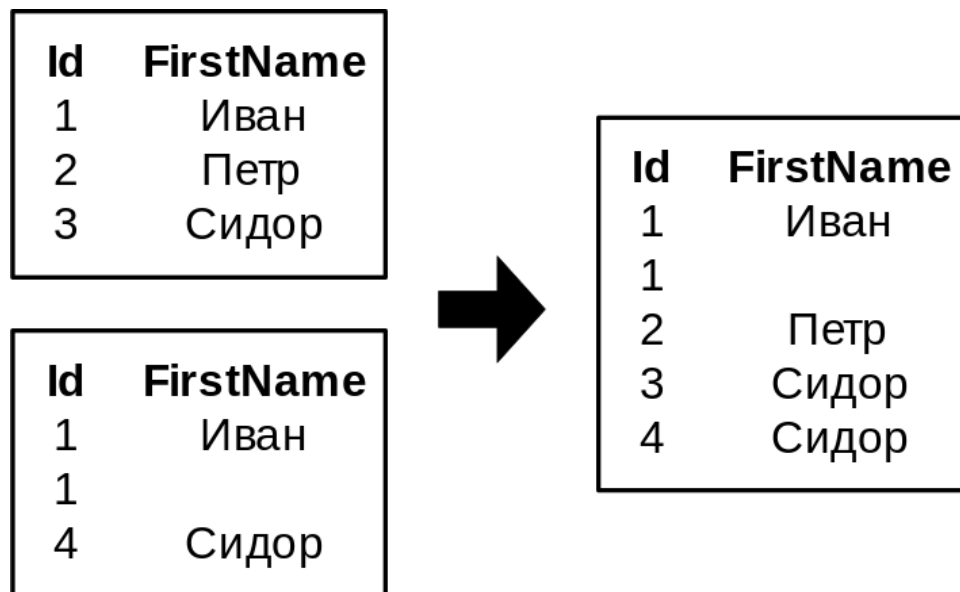


Рис. 15: Объединение отношений

- Пересечение отношений:  $R_1 \cap R_2$

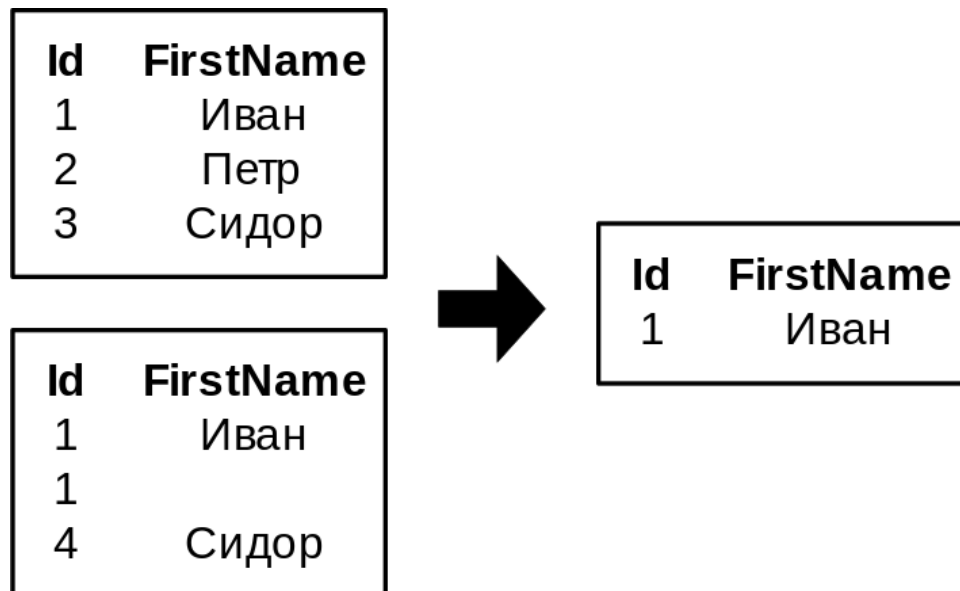


Рис. 16: Пересечение отношений

- Разность отношений:  $R_1 \setminus R_2$



Рис. 17: Разность отношений

Стоит отметить, что для объединения отношений с различающимися именами атрибутов, но при равном их количестве, можно воспользоваться переименованием для того, чтобы привести заголовки к одному виду.

## 1.7 Реляционная алгебра. Соединения

### 1.7.1 Полное соединение

**Определение.** *Полным соединением* называется декартово произведение двух отношений. Заголовком является объединение заголовков, а телом – декартово произведение тел отношений.

**Замечание.** Необходимым условием выполнимости этой операции является отсутствие общих имен атрибутов. При желании, этого можно добиться операцией переименования (см. 1.6.3).

На рисунке 18 приведен пример полного соединения.



Рис. 18: Пример полного соединения

### 1.7.2 Естественное соединение

**Определение.** *Естественным соединением* называется операция, при которой у двух отношений соединяются кортежи, имеющие равные значения атрибутов с одинаковыми именами. Обозначается  $R_1 \bowtie R_2$ . Заголовком является объединение заголовков.

**Замечание.** При отсутствии общих атрибутов в заголовках отношений, естественное соединение эквивалентно полному.

На рисунке 19 приведен пример естественного соединения.



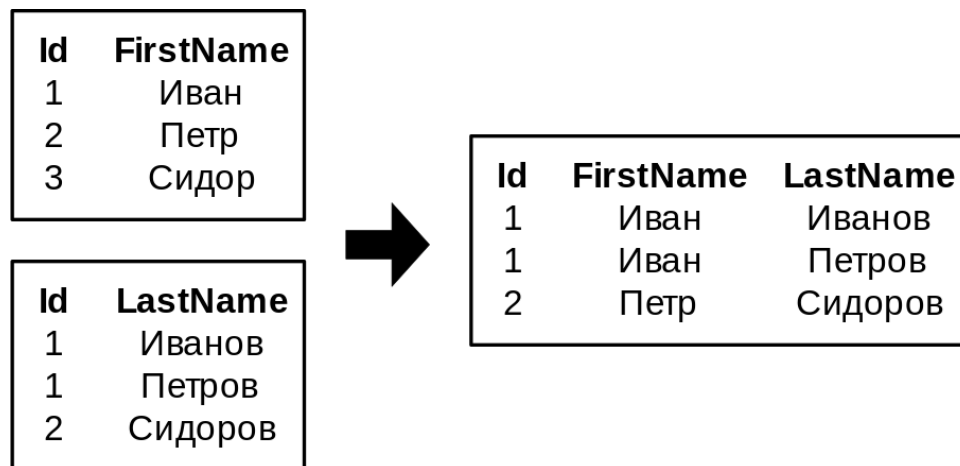


Рис. 19: Пример естественного соединения

**Размер естественного соединения** Нетрудно понять, какой минимальный и максимальный размер естественного соединения может получиться:

$$0 \leq |R_1 \bowtie R_2| \leq |R_1| \times |R_2|.$$

Нижняя оценка достигается при отсутствии равных атрибутов, верхняя – при отсутствии общих атрибутов в заголовке.

### 1.7.3 Внешнее соединение

**Определение.** *Внешнее соединение* похоже на естественное (см. 1.7.2), только оно дополнительно сохраняет те кортежи, для которых нет соответствующих во втором отношении. Вместо соответствующего берется пустой кортеж. Заголовком все так же является объединение заголовков. Обозначение:  $R_1 \Join R_2$ .

На рисунке 20 приведен пример внешнего соединения.

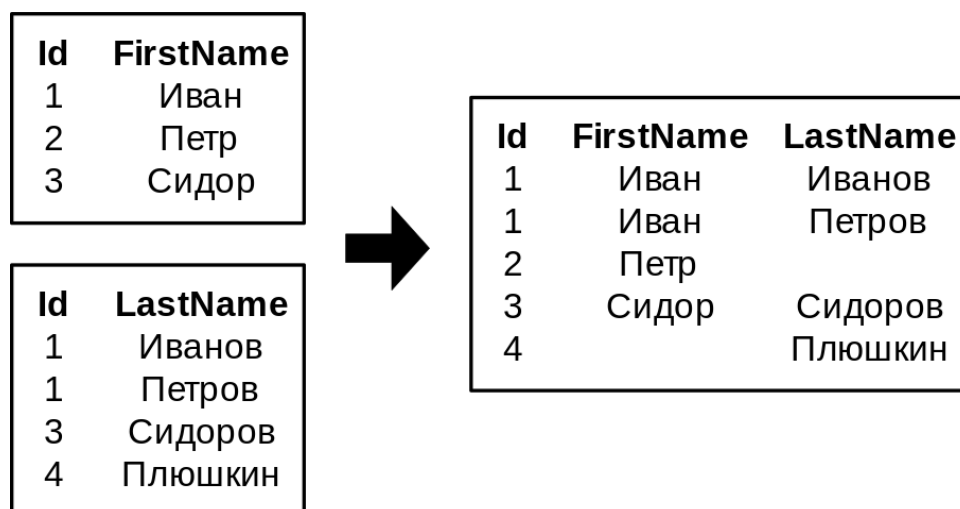


Рис. 20: Пример внешнего соединения

#### 1.7.4 Левое и правое соединения

**Определение.** *Левое соединение* похоже на естественное (см. 1.7.2), только оно дополнительно сохраняет те кортежи левого отношения, для которых нет соответствующих в правом отношении. Вместо соответствующего берется пустой кортеж. Заголовком все так же является объединение заголовков. Обозначение:  $R_1 \bowtie R_2$ . Правое определяется симметрично и обозначается  $R_1 \ltimes R_2$ .

**Замечание.**

$$R_1 \bowtie R_2 \equiv (R_1 \bowtie R_2) \cup (R_1 \setminus \pi_{R_1}(R_1 \bowtie R_2)),$$

$$R_1 \ltimes R_2 \equiv (R_1 \bowtie R_2) \cup (R_2 \setminus \pi_{R_2}(R_1 \bowtie R_2)).$$

Кроме того, через левое и правое соединения можно выразить внешнее:

$$R_1 \Join R_2 \equiv (R_1 \bowtie R_2) \cup (R_1 \ltimes R_2).$$

На рисунке 21 приведены примеры косых соединений  $R_1 \bowtie R_2$  и  $R_1 \ltimes R_2$ .



Рис. 21: Примеры косых соединений

#### 1.7.5 Полусоединения

**Определение.** *Левым (правым) полусоединением* называется следующее отношение:  $R_1 \ltimes R_2 \equiv \pi_{R_1}(R_1 \bowtie R_2)$  ( $R_1 \bowtie R_2 \equiv \pi_{R_2}(R_1 \bowtie R_2)$ ). Иначе говоря, это отношение, состоящее из строк левого (правого) отношения, для которых есть соответствующие строки в правом (левом).

**Замечание.**

- $R_1 \ltimes R_2 = (R_1 \bowtie R_2) \cup (R_1 \setminus R_1 \bowtie R_2)$ .
- $R_1 \bowtie R_2 = (R_1 \bowtie R_2) \cup (R_2 \setminus R_1 \bowtie R_2)$ .

На рисунке 22 приведены примеры полусоединений  $R_1 \bowtie R_2$  и  $R_1 \ltimes R_2$ .



Рис. 22: Примеры полусоединений

### 1.7.6 Условные соединения

**Определение.** Возьмем любое соединение и наложим на него дополнительное условие  $\theta$ . Получится соответствующее условное соединение. Имеется в виду, что нужно отфильтровать результат соединения. Однако, не всегда это возможно. Покажем на нескольких примерах, как это должно выглядеть.

- $R_1 \bowtie_{\theta} R_2 = \sigma_{\theta}(R_1 \bowtie R_2)$ .
- $R_1 \ltimes_{\theta} R_2 = J \cup (R_1 \setminus \pi_{R_1}(J))$ ,  $J = \sigma_{\theta}(R_1 \bowtie R_2)$ .
- $R_1 \ltimes_{\theta} R_2 = J \cup (R_2 \setminus \pi_{R_2}(J))$ ,  $J = \sigma_{\theta}(R_1 \bowtie R_2)$ .
- $R_1 \ltimes_{\theta} R_2 = (R_1 \ltimes_{\theta} R_2) \cup (R_1 \ltimes_{\theta} R_2)$ .

На рисунке 23 приведен пример левого условного соединения  $R_1 \ltimes_{|FirstName|+2 < |LastName|} R_2$ .

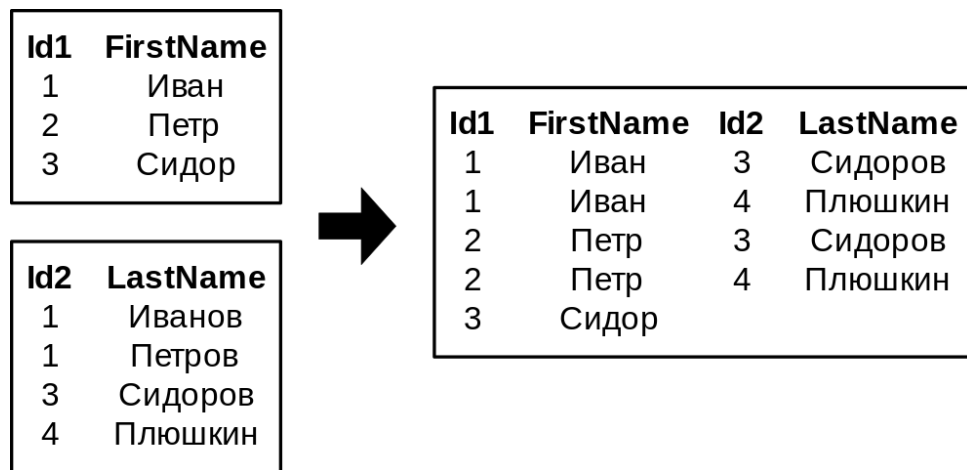


Рис. 23: Пример левого условного соединения

## 1.8 Реляционная алгебра. Деление и операции над данными

### 1.8.1 Деление

**Определение.** Делением называется операция, результат которой  $R(X) = Q(XY) \div S(Y)$  максимальный при условии  $R \times S \subseteq Q$ . Эту операцию можно записать по-другому:

- $Q \div S \equiv \{x \mid x \in \pi_X(Q), \{x\} \times S \subseteq Q\}$ .
- $Q \div S \equiv \pi_X(Q) \setminus \pi_X(\pi_X(Q) \times S \setminus Q)$ .

Заголовок результирующего отношения – X.  $S \subseteq Q$ .

**Замечание.** Интуитивно, эта операция – запрос всех X таких, что для всех Y найдется пара, равная по X:  $x \in \pi_X(Q): \forall y \in S: (x, y) \in Q$ .

На рисунке 24 представлен пример деления.



Рис. 24: Пример деления

### 1.8.2 Большое деление

**Определение.** Большим делением называется операция, которую можно представить следующим образом:

- $Q(XY) * S(YZ) \equiv \{(x, z) \mid \{x\} \times \pi_Y(\sigma_{Z=z}(S)) \subseteq Q\}$ .
- $Q(XY) * S(YZ) \equiv \pi_X(Q) \times \pi_Z(S) \setminus \pi_{XZ}(\pi_X(Q) \times S \setminus Q \bowtie S)$ .

Заголовок результирующего отношения – XZ.

**Замечание.** Интуитивно, большое деление – запрос ‘для всех связанных’, или деление для каждого z. Иначе говоря, для каждого z найти такие x, что для всех y, связанных с z, найдется соответствующий x:

$$(x, z) \in \pi_X(Q) \times \pi_Z(S): \forall y \in \pi_Y(\sigma_{Z=z}(S)) (x, y) \in Q.$$

На рисунке 25 представлен пример большого деления.



Рис. 25: Пример большого деления

### 1.8.3 Расширение

**Определение.** *Расширение* – операция над данными, добавляющая новый вычисляемый атрибут. Заголовком результата будет  $R \cup \{A\}$ . Обозначение:  $\varepsilon_{A=\text{expr}}(R)$ . К каждому кортежу тела  $R$  добавится вычисленное значение  $\text{expr}$ . Выражением может быть комбинация атрибутов  $R$ , а также различные функции и операции, доступные в БД.

На рисунке 26 изображен пример композиции расширений  $\varepsilon_{\text{Tax}=\text{tax10}(\text{Total})} \circ \varepsilon_{\text{Total}=\text{Price} \cdot \text{Items}}$ .



Рис. 26: Пример расширения

### 1.8.4 Агрегирование

**Определение.** *Агрегирование* – обработка набора значений. Обозначается  $\text{func}_{Q,A}(R)$ .

- Примеры  $\text{func}$ : count, sum, avg, max, min, all, any.
- $Q$  – агрегируемый атрибут.
- $A$  – сохраняемые атрибуты.
- $r \in \pi_A(R)$  расширяется атрибутом  $Q = \text{func}(\pi_Q\{q \in R \mid \pi_A(q) = r\})$

**Замечание.** Интуитивно, данные разбиваются по корзинам с одинаковыми  $A$ , после чего каким-то образом сворачиваются, то есть, на каждой корзине по отдельности считается заданная функция.

На рисунках 27, 28 изображены примеры агрегирования.

- $\text{sum}_{\text{Total},\{\text{Supplier}\}} \circ \varepsilon_{\text{Total}=\text{Price} \cdot \text{Items}}$

Supplier	Price	Items
1	1	4
1	2	5
2	3	6




Supplier	Total
1	14
2	18

Рис. 27: Пример агрегирования 1

- $\text{sum}_{\text{Total},\emptyset} \circ \varepsilon_{\text{Total}=\text{Price} \cdot \text{Items}}$

Price	Items
1	4
2	5
3	6



Total
32

Рис. 28: Пример агрегирования 2

## 1.9 Исчисление кортежей и его реляционная полнота

### 1.9.1 Реляционное исчисление

**Определение.** *Реляционное исчисление* – декларативный язык для работы с отношениями.

В отличие от реляционной алгебры, в реляционном исчислении описывается не то, как построить запрос, а свойства результата, который ожидаем получить.

Выделяют два вида реляционного исчисления: **исчисление кортежей** и **исчисление доменов**.

#### Части запроса

- Определение переменных,
- Определение свойств результата.

```
<variables definition>  
select <attributes list>  
from <variables>  
where <condition>
```

**Пример** Рассмотрим запрос, для получения идентификаторов студентов, обучающихся в группе М34371.

Запрос в реляционной алгебре:

$$\pi_{SId}(\sigma_{Name=M34371}(Students \bowtie Groups))$$

Запрос в реляционном исчислении:

```
select S.SId from S  
where  $\exists G (S.GId = G.GId \wedge G.Name = 'M34371')$ 
```

### 1.9.2 Исчисление кортежей

В качестве переменных выступают кортежи. Тип переменной определяется именем и типами атрибутов, а также набором значений. Заметим, что это есть отношение.

**Синтаксис** Указывается имя и *отношение*.

```
<Variable> :: <Relation>
```



## Операции с отношениями

- **Ограничение**

`<Relation> where <Condition>`

- **Объединение**

`<Relation 1>, <Relation 2>`

**Примеры** Рассмотренное выше отношение G4 можно выразить иначе:

```
G4 :: Groups where Name = 'M34351',  
      Groups where Name = 'M34371',  
      Groups where Name = 'M34391'
```

**Условия** Условия во многом аналогичны условиям из реляционной алгебры. Дополнительно вводятся кванторы.

- **Простые условия.**

- Сравнение атрибутов с константами

```
S.Name = 'John'  
S.Id < 5
```

- Сравнение атрибутов между собой

```
S.Id ≥ G.Id
```

- Сравнение с применением формул

```
length(S.FirstName) = length(S.LastName) + 3
```

- **Составные условия.** Вводятся логические связки:  $\vee$ ,  $\wedge$ ,  $\neg$ .

```
G where Name = 'M34371' ∨ Name = 'M34391'  
S where FirstName = 'John' ∧ LastName <> 'Smith'
```

- **Условия с кванторами.** Вводятся кванторы всеобщности  $\forall$  и существования  $\exists$ . Синтаксис: `<quantifier> <variable> (<condition>)`.

```
G where ∃ S (S.FirstName = 'John' ∧ S.GId = G.GId)  
G where ∀ S (S.FirstName = 'John' ∨ S.GId = G.GId)
```

**Примеры** Рассмотрим примеры введения переменных, запроса, который требует в реляционной алгебре применения деления, и работы с несколькими отношениями.

— *Variables declaration*

```
S :: Students; G :: Groups; C :: Courses; P :: Point;
G4 :: Groups where Name = 'M34351' ∨
      Name = 'M34371' ∨ Name = 'M34391'
```

— *Fully marked groups*

```
select G.GId from G where ∀ S (∀ C(∃ P
      (S.Sid = P.Sid ∧ C.CId = P.CId ∧ P.Points ≥ 60)))
```

— *Multiple relations*

```
select S.FirstName, S.LastName, G.Name
from S, G
where S.GId = G.GId
```

### 1.9.3 Связи реляционной алгебры и исчисления кортежей

**Алгебра через исчисление** Выразим реляционную алгебру через исчисление кортежей.

- **Проекция.**

- $\pi_{A_1, \dots, A_n}(R)$
  - `select A1, ..., An from R`

- **Фильтрация.**

- $\sigma_{\theta}(R)$
  - `from R where  $\theta$`

- **Создание нового столбца.**

- $\epsilon_{A=expr}(R)$
  - `select R.*, expr as A from R`

- **Объединение.**

- $R_1 \cup R_2$
  - `R :: R1, R2`

- **Разность.**

- $R_1 \setminus R_2$
  - `R :: R1 where  $\neg \exists R2 (R1 = R2)$`

- **Декартово произведение.**

- $R_1 \times R_2$
- `select R1.*, R2.* from R1, R2`

- **Объединение.**

- $R_1 \bowtie R_2$
- `select R1.*, R2.* from R1, R2  
where R1.<common_attrs> = R2.<common_attrs>`

Указанные выше операции образуют базис реляционной алгебры. Следовательно, выразительная мощность реляционной алгебры *не больше* выразительной мощности реляционного исчисления, и алгебру можно выразить через исчисление.

**Исчисление через алгебру** Выражения с кванторами можно привести к предваренной нормальной форме (сначала идут только кванторы, затем – условие). После этого можно преобразовать выражение в алгебру:

- Построение выражения для каждой переменной,
- Построение декартового произведения по всем затронутым отношениям,
- Фильтрация по условию,
- Применение кванторов:
  - Квантор существования: проекция, исключая атрибуты, порожденные переменной. Если существует хотя бы одно значение кортежной переменной, удовлетворяющее условию, то проекция будет непустой. Верно обратное.
  - Квантор всеобщности: как было замечено, логике этого квантора соответствует деление. Таким образом, достаточно делить на все столбцы кортежной переменной.

**Пример** Запрос получает идентификаторы групп, в которых есть хотя бы один студент, аттестованный по всем дисциплинам.

```
select G.GId where ∃ S(∀ C(∃ P
  (G.GId = S.GId ∧ S.SId = P.SId ∧
    C.CId = P.CId ∧ P.Points ≥ 60)))
```

Преобразуем его в реляционную алгебру.

$$T_0 = \sigma_{P.Points \geq 60}((G \bowtie S \times C) \bowtie P)$$

Самый внутренний квантор ( $\exists P$ ) – проекция на все внешние атрибуты (находящиеся левее в выражении):

$$T_1 = \pi_{G_*, S_*, C_*}(T_0)$$

Далее следует квантор всеобщности ( $\forall C$ ) – деление на переменную:

$$T_2 = T_1 \div C$$

Последний квантор существования ( $\exists S$ ) – проекция на единственный оставшийся внешний атрибут:

$$T_3 = \pi_{G_*}(T_2)$$

Получили выражение в терминах реляционной алгебры:

$$T = \pi_{G_*}(\pi_{G_*, S_*, C_*}(\sigma_{P.Points \geq 60}((G \bowtie S \times C) \bowtie P)) \div C)$$

**Утверждение 1.1.** Выразительная мощность исчисления кортежей равна выразительной мощности реляционной алгебры.

**Определение.** *Реляционно-полные языки* – языки, выразительная мощность которых не меньше выразительной мощности реляционной алгебры.

Следовательно, исчисление кортежей – реляционно-полный язык.

## 1.10 Исчисление доменов и его реляционная полнота

### 1.10.1 Исчисление доменов

В качестве переменных выступают домены (типы). Иными словами, работа ведется с атомарными переменными в отличие от исчисления кортежей, где работа ведется с отношениями.

**Синтаксис объявления переменных** Указывается имя и *домен (тип)*.

`<Variable> :: <Type>`

**Пример** Можно использовать типы из SQL.

```
SId :: Int
FirstName :: Varchar(100)
```

**Условие принадлежности** Для связи доменов с отношениями вводится предикат проверки принадлежности значения заданному кортежу.

**Синтаксис** Отсутствие какого-либо атрибута отношения означает, что его конкретное значение нас не интересует.

```
<Relation> {
    <Attribute1> = <Value1>,
    <Attribute2> = <Value2>,
    ...
}
```

**Пример** В последнем примере `SId = SId` означает, что атрибут `SId` (слева) равен значению переменной с именем `SId`. Не отличается по смыслу от предыдущего.

```
S{FirstName = 'John', LastName = 'Smith'}
S{SId = Id}
S{SId = SId}
```

### Примеры

- Идентификаторы всех студентов:

```
SId where S{SId = SId}
```

- Идентификаторы всех студентов определенной группы:

```
SId where S{SId = SId, GId = 'M34371'}
```

- Идентификаторы всех студентов двух определенных групп:

SId where S{SId = SId, GId = 'M34371'}  $\vee$   
 S{SId = SId, GId = 'M34391'}

- Идентификаторы всех студентов, не сдавших курс с определенным идентификатором.

SId where  $\neg \exists$  Points (Points  $\geq 60 \wedge$   
 P {SId = SId, Points = Points, CId = 10})

### 1.10.2 Связи реляционной алгебры и исчисления доменов

**Алгебра через исчисление** Выразим реляционную алгебру через исчисление доменов.

- **Проекция.**

–  $\pi_{A_1, \dots, A_n}(R)$   
 – A1, ..., An from R where R{A1 = A1, ..., An = An}

- **Фильтрация.**

–  $\sigma_{\theta}(R)$   
 – A1, ..., An from R where R{A1 = A1, ..., An = An}  $\wedge \theta$

- **Создание нового столбца.**

–  $\epsilon_{A=expr}(R)$   
 – expr as A from R where R{A1 = A1, ..., An = An}

- **Объединение.**

–  $R_1 \cup R_2$   
 – A1, ..., An where R1{..., Ai = Ai, ...}  $\vee$  R2{..., Ai = Ai, ...}

- **Разность.**

–  $R_1 \setminus R_2$   
 – A1, ..., An where R1{..., Ai = Ai, ...}  $\vee \neg$  R2{..., Ai = Ai, ...}

- **Декартово произведение.**

–  $R_1 \times R_2$   
 – A1, ..., An, B1, ..., Bm where R1{..., Ai = Ai, ...}  $\wedge$   
 R2{..., Bj = Bj, ...}

- **Объединение.**

–  $R_1 \bowtie R_2$

$- A_1, \dots, A_n, B_1, \dots, B_m, C_1, \dots, C_l$   
 where  $R_1\{\dots, A_i = A_i, \dots, B_j = B_j, \dots\} \wedge$   
 $R_2\{\dots, B_j = B_j, \dots, C_k = C_k, \dots\}$

**Утверждение 1.2.** Выразительная мощность исчисления доменов не меньше выразительной мощности реляционной алгебры, то есть исчисление доменов – реляционно-полный язык.

## 1.11 Datalog и рекурсия

### 1.11.1 Datalog

*Datalog* – язык запросов к БД. Разработан в 1978 году, имеет много схожего с языком Prolog. Повлиял на развитие SQL. Реализует исчисление доменов.

Программа на языке состоит из набора определений отношений. Результатом считается тело одного из отношений.

#### Конструкции языка

- **Реляционные атомы.** Реализуют предикат вхождения из исчисления доменов. Используется позиционная запись аргументов.

— Values belong to relation R

$R(x_1, x_2, \dots, x_n)$

— Values do not belong to relation R

$\neg R(x_1, x_2, \dots, x_n)$

- **Арифметические атомы.** Сравнение двух выражений.

**Синтаксис** Программа состоит из списка строк:

$\langle \text{Relation} \rangle (\langle \text{Attr } 1 \rangle, \langle \text{Attr } 2 \rangle, \dots, \langle \text{Attr } n \rangle) :- \langle \text{Target} \rangle$

Цель (target) – последовательность атомов. В отношение (Relation) входят кортежи, удовлетворяющие хотя бы одной цели (всем атомам хотя бы одной цели). Иначе говоря, запросы пишутся в дизъюнктивной нормальной форме.

#### Примеры

- Идентификаторы и фамилии всех студентов с заданным именем:

```
Johns(Id, LastName) :-  
    Students(Id, FirstName, LastName),  
    FirstName = 'John'.
```

- Имена всех студентов и преподавателей:

```
Names(Name) :- Students(_, Name, _).  
Names(Name) :- Lecturers(_, Name, _).
```

**Бесконечные отношения** Запросы могут породить бесконечные отношения. Например:

```
NotStudent(Id, Name) :-  $\neg$  Students(Id, Name, _).
```

Здесь нет ограничения на фамилию студента – ни по типу, ни по значению – поэтому это отношение бесконечно и содержит студентов с почти произвольными фамилиями.



**Утверждение 1.3.** Для запрета бесконечных отношения достаточно принудить каждую переменную входить как минимум в один неотрицательный реляционный атом.

**Реляционная алгебра через Datalog** Выразим реляционную алгебру через исчисление доменов.

- **Проекция.**

- $\pi_{A_1, \dots, A_n}(R)$
- $Q(A_1, \dots, A_n) :- R(A_1, \dots, A_n, \_, \dots, \_).$

- **Фильтрация.**

- $\sigma_\theta(R)$
- $Q(A_1, \dots, A_n) :- R(A_1, \dots, A_n, \_, \dots, \_), \theta.$

- **Объединение.**

- $R_1 \cup R_2$
- $Q(A_1, \dots, A_n) :- R_1(A_1, \dots, A_n).$
- $Q(A_1, \dots, A_n) :- R_2(A_1, \dots, A_n).$

- **Разность.**

- $R_1 \setminus R_2$
- $Q(A_1, \dots, A_n) :- R_1(A_1, \dots, A_n), \neg R_2(A_1, \dots, A_n).$
- **Замечание.** Запись корректна, поскольку атрибуты  $A_1, \dots, A_n$  входят в неотрицательный реляционный атом  $R_1$ .

- **Декартово произведение.**

- $R_1 \times R_2$
- $Q(A_1, \dots, A_n, B_1, \dots, B_m) :- R_1(A_1, \dots, A_n), R_2(B_1, \dots, B_m).$

- **Объединение.**

- $R_1 \bowtie R_2$
- $Q(A_1, \dots, A_n, B_1, \dots, B_m, C_1, \dots, C_l) :-$   
 $R_1(A_1, \dots, A_n, B_1, \dots, B_m), R_2(B_1, \dots, B_m, C_1, \dots, C_l).$

**Утверждение 1.4.** Datalog – реляционно-полный язык.

**Замечание.** В Datalog нет явных кванторов. Есть только неявный квантор существования – для свободных переменных справа в выражении.

**Пример** Рассмотрим таблицу:

Person(Id, Name, MotherId, FatherId)

Получить для каждого человека имена обоих родителей:

Parents(N, FN, MN) :- Person(\_, N, FId, MId),  
                          Person(FId, FN, \_, \_), Person(MId, MN, \_, \_).

Получить для каждого человека имя его родителя:

Parents(N, FN) :- Person(\_, N, FId, \_), Person(FId, FN, \_, \_).  
Parents(N, MN) :- Person(\_, N, MId, \_), Person(MId, MN, \_, \_).

### 1.11.2 Рекурсивные запросы

Datalog позволяет строить рекурсивные запросы, например, для транзитивного замыкания. Рассмотрим для примера таблицу:

Parent(Id, ParentId)

И запрос для получения всех предков (представляющий собой транзитивное замыкание):

Ancestor(Id, PId) :- Parent(Id, PId).  
Ancestor(Id, GId) :- Parent(Id, PId), Ancestor(PId, GId).

Такому запросу отвечает несколько множеств кортежей. Формально, помимо ожидаемого транзитивного замыкания, декартово произведение всех возможных идентификаторов будет удовлетворять запросу. При этом, такое множество кортежей будет *неподвижной точкой* – множеством, которое нельзя дополнить по правилам вывода. Но такой результат не будет иметь практического смысла

Во избежание неопределенности, в Datalog результатом рекурсивного запроса является *минимальная неподвижная точка* – минимальное по включению множество, такое что левая и правая части совпадают.

На практике рекурсивное правило применяется, пока не будет достигнута неподвижная точка. Очевидно, она будет минимальной по включению.

#### Поиск минимальной неподвижной точки

- Инициализация нерекурсивными данным,
- Пока есть следствия, данные пополняются ими.

**Замечание.** Отрицания в рекурсивных правилах могут приводить к парадоксам (например, на шаге  $n$  какие-то данные должны быть добавлены, на  $n + 1$  – нет). Поэтому в рекурсивных запросах явно запрещены циклы, содержащие отрицания.

**Замечание.** Как было замечено ранее, реляционная алгебра не позволяет построить рекурсивные запросы. Следовательно, выразительная мощность Datalog больше, чем у реляционной алгебры.

### 1.11.3 Связь реляционного исчисления и SQL

- **Заголовок запроса.** Определяет, какие данные будут получены. В реляционной алгебре соответствует самой внешней проекции. В реляционном исчислении – заголовку запроса.
- **Раздел from.** Определяет набор отношений, с которыми производится работа. В реляционной алгебре здесь указываются соединения. В реляционном исчислении – переменные отношений.
- **Раздел where.** Определяет набор отношений, с которыми производится работа. В реляционной алгебре соответствует самой внешней фильтрации. В реляционном исчислении – условию на кортежи.

**Замечание.** В языке SQL на практике большинство кванторов применяется для соединений. Однако, существует также явный квантор существования. Как было показано в Datalog, квантор всеобщности избыточен ( $\forall v(p(v)) \equiv \neg \exists v(\neg p(v))$ )

### 1.11.4 Подзапросы

Квантор существования предоставлен в SQL через подзапросы существования:

- Существования (exists),
- Вхождения (in),
- Условные (any, all),
- Скалярные.

**Подзапрос существования** Если подзапросу удовлетворяет хотя бы один кортеж, то оператор возвращает true, иначе false.

```
[not] exists (select ...)
```

**Пример** Группы, в которых нет студентов:

```
select G.Name where  
not exists (select * from S where S.Gid = G.Gid)
```

**Подзапрос вхождения** Если результаты всех выражений входят в подзапрос, то оператор возвращает true, иначе false.

```
(<list of expressions>) [not] in (select ...)
```

**Замечание.** Многие СУБД поддерживают подзапросы только с одним столбцом.

**Пример** Оценки по предметам 4 курса:

```
select * from P where P.CId in
      (select CId from C where Year = 4)
```

**Условный подзапрос** any возвращает true, если для одного из значений выполняется условие. Аналогично, all возвращает true, если для всех значений выполняется условие.

```
<expression> <condition> { any | all } (select ...)
```

**Пример** Лучшие студенты по предметам:

```
select SId, CId from P as PE where PE.Points >=
      all (select Points from P where P.CId = PE.CId)
```

**Скалярные подзапросы** Запросы, возвращающие не более одного значения.

```
(select ...)
```

**Замечание.** Результатом скалярного подзапроса, вернувшего пустой результат, является null.

**Пример** Студенты и название группы:

```
select
      SId,
      (select Name from G where G.GId = S.GId)
from S
```

### Кореллированные подзапросы

**Определение.** Подзапрос называется *кореллированным* (некореллированным), если в нем есть (отсутствуют) свободные переменные.

**Примеры** Лучшие студенты по предметам (кореллированный подзапрос, PE – свободная переменная):

```
select SId, CId from P as PE where PE.Points >=
      all (select Points from P where P.CId = PE.CId)
```

Оценки по предметам четвертого курса (некореллированный подзапрос):

```
select * from P where P.CId in
      (select CId from C where Year = 4)
```

**Замечание.** Преимущество некореллированных подзапросов в том, что их можно посчитать один раз и переиспользовать результат в дальнейшем. Поэтому на практике они предпочтительнее кореллированных.

### 1.11.5 Рекурсия

Поддержка рекурсии была добавлена в SQL3. Однако, многие СУБД не поддерживают ее.

**Синтаксис** Как и в Datalog, отрицание должно быть стратифицировано – оно запрещено в циклах рекурсии.

```
with recursive <relation>(<columns>) as ...
```

**Пример** Транзитивное замыкание предков:

```
with recursive Ancestor(Id, AId) as  
  select Id, PId from Parent  
  union  
  select P.Id, AId from Ancestor a  
    inner join Parent p on a.Id = p.PId  
select * from Ancestor
```

## 1.12 Представления и их обновление

### 1.12.1 Определение представлений

**Определение.** *Представление* – именованный запрос.

Представления применяются для:

- **Макросы,**
- **Соккрытие данных,**
- **Независимость от данных.** При обнаружении аномалий и декомпозиции таблицы использование представлений позволяет обратно соединять таблицы прозрачно для пользователя.

**Синтаксис**

```
create view <name> as <query>;  
drop view <name>;
```

**Замечание.** Представление – не зафиксированные данные на момент создания. При каждом его использовании производится запрос.

**Примеры.**

```
create view AveragePoints as  
select SId, avg(Points) from Points;  
  
create view StudentCourse as  
select s.FirstName, s.LastName, c.Name, p.Points  
from Students s natural join Points p natural join  
Courses c
```

### 1.12.2 Обновление представлений

Поддержка модифицирующих операций над представлениями является полезным инструментом и может применяться для обеспечения независимости от данных и их сокращения. Иными словами, ожидается, что обновления в представлениях должны прозрачно производиться в таблицах, из которых оно составлено. Важно заметить, что дополнительно требуется взаимная обратимость вставки и удаления.

**Замечание.** Обновление будет рассмотрено как последовательные удаление и вставка без промежуточной проверки целостности.

Рассмотрим, как должно распространяться обновление через сеть операций, составляющих представление:

- $\sigma_p(R)$ . Вставим только кортеж, удовлетворяющий условию фильтрации, иначе он не появится в представлении. Аналогично при удалении.

- $\pi_A(R)$ . Вставим кортеж с дополненными значениями атрибутов по умолчанию. При удалении данных из проекции необходимо удалить *все* соответствующие кортежи.
- $\rho_{a=b}(R)$ . Столбцы не добавляются и не удаляются. При вставке и удалении атрибуты переименовываются в соответствии с операцией.
- $R_1 \cup R_2$ . Назовем условия, при котором кортеж попадает в  $R_1$  или  $R_2$  *предикатами* соответствующих отношений. При вставке в представление будем проверять, удовлетворяет ли кортеж предикату  $R_1$ , и в положительном случае будем добавлять его в  $R_1$ . Аналогично, если кортеж удовлетворяет предикату  $R_2$ , добавим его и туда. Таким образом если кортеж удовлетворяет предикатам  $R_1$  и  $R_2$  одновременно, то кортеж следует добавить в оба отношения. При удалении кортеж следует удалить как из  $R_1$ , так и из  $R_2$ .
- $R_1 \cap R_2$ . Вставим кортеж одновременно в  $R_1$  и  $R_2$ . При удалении – удалим его одновременно из  $R_1$  и  $R_2$ .
- $R_1 \setminus R_2$ . Вставим или удалим кортеж из  $R_1$ . Заметим, что при удалении также корректно добавить кортеж в  $R_2$ , но это противоречит смыслу удаления.
- $R_1 \bowtie R_2$ . При вставке и удалении в  $R_1$  и  $R_2$  соответственно вставляются части кортежа, спроецированные на соответствующие отношения. При этом важно учитывать ограничения целостности между отношениями  $R_1$  и  $R_2$ :
  - *Один-к-одному*. Вставка и удаление производятся для каждого отношения.
  - *Один-ко-многим*. Обязательна вставка в отношение со стороны “многие”, но со стороны “один” – только если кортежа еще не было. Аналогично при удалении.
  - *Многие-ко-многим*. С точки зрения БД такое отношение преобразуется в join table с двумя отношениями *один-ко-многим*.

**Обновление в SQL** Стандарт утверждает, что унарные операции ( $\sigma_p(R)$ ,  $\pi_A(R)$ ,  $\rho_{a=b}(R)$ ) являются обновляемыми, множественные ( $R_1 \cup R_2$ ,  $R_1 \cap R_2$ ,  $R_1 \setminus R_2$ ) – не обновляемые. Соединения “один-к-одному” – обновляемые, “один-ко-многим” – обновляемые только со стороны “многие”. “Многие-ко-многим”, формально, не существует, но при поддержке считается, что они не обновляемые.

**Замечание.** Большинство СУБД не поддерживают обновление представлений.

### 1.12.3 Материализованные представления

**Определение.** Материализованное представление – “слепок” данных на определенный момент времени.

Материализованные представления обычно хранятся физически и не изменяются при изменении базовой версии данных. Преимуществами являются быстрота выборки (сложные запросы не будут пересчитываться) и возможность зафиксировать состояние БД на конкретный момент времени. Недостатки: необходимость обновления при устаревании данных и необходимость занимать место в памяти.

#### Синтаксис

```
create materialized view <name>
[
    refresh [{fast | complete}] [on {demand | commit}]
    [start with <time>] [next <time>]
]
as <query>;
refresh materialized view <name>;
```

При выборе типа обновления **fast** изменения применяются инкрементально, то есть на основе конкретных изменений данных, в то время как **complete** форсирует полный пересчет данных. Опции **on demand** или **on commit** определяют, что пересчет необходимо производить по требованию или при применении изменений соответственно.

#### Пример.

```
create materialized view AceragePoints
refresh next dateadd(day, now(), 1)
as select SId, avg(Points) from Points group by SId
```

**Замечание.** Синтаксис объявления материализованных представлений во многом специфичен для конкретных СУБД. Также некоторые СУБД не поддерживают этот функционал в принципе.



## 1.13 Транзакции. Восстановление. Классический алгоритм

### 1.13.1 Транзакции

**Определение.** *Транзакция* – минимальный объем работы, который можно зафиксировать в базе данных.

Каждый оператор заключен в неявную транзакцию, которая начинается непосредственно перед оператором и заканчивается после него. Не все действия в СУБД являются транзакционными. Например, во многих реализациях не поддерживается транзакционное изменение схемы данных.

#### Свойства транзакций (ACID)

**Определение.** *Атомарность (Atomicity)* – с точки зрения БД, транзакция либо выполняется целиком, либо полностью откатывается. Иначе говоря, никто со стороны не может увидеть промежуточное состояние выполнения транзакции.

**Определение.** *Согласованность (Consistency)* – после завершения транзакции БД остается в согласованном состоянии.

**Определение.** *Изоляция (Isolation)* – транзакции не могут взаимодействовать между собой. Это означает, что транзакции не могут пользоваться промежуточными результатами друг друга.

**Определение.** *Устойчивость (Durability)* – при успешном завершении транзакции результаты ее исполнения сохраняются в БД, при откате транзакции все внесенные ею изменения отменяются.

#### Корректность и согласованность

**Определение.** Состояние БД является *согласованным*, если оно удовлетворяет всем объявленным ограничениям. Это свойство автоматически проверяется СУБД.

**Определение.** Состояние БД является *корректным*, если оно соответствует реальному миру. Автоматически проверено быть не может.

Существуют условия корректности, которые нельзя проверить ограничениями. Например, после перевода денег в банке их общая сумма в системе не должна измениться. Однако, эта сумма заранее неизвестна, поэтому заранее задать ограничение невозможно.

**Минимизация транзакций** Транзакции требуют большие ресурсные затраты, поэтому должны быть минимальными. По возможности следует использовать неявные транзакции.

Однако, есть типичные ситуации, в которых использования неявных транзакций недостаточно:

- *Условное обновление* – проверку условия и обновление необходимо сделать в рамках одной транзакции, в противном случае в момент изменения условие может перестать выполняться;
- *Множественное обновление* – при обновлении данных, особенно в различных таблицах, использование транзакции необходимо для исключения несогласованности;
- *Промежуточная несовместимость* – некоторые действия требуют временного нарушения согласованности с последующим его восстановлением, использование транзакций позволяет откладывать проверку согласованности до завершения всех действий и исключают видимость несогласованного состояния другими пользователями.

Также следует отметить, что результат завершения транзакции **не должен зависеть от человека**. Человеческий фактор может привести к зависшей транзакции. Если требуется принятие решения от человека, транзакцию следует разбить на две: первая читает данные, а вторая их записывает, предварительно проверяя данные на соответствие результату первой. В таком случае от решения человека зависит применение второй транзакции.

### 1.13.2 Восстановление

Напомним, что свойство *устойчивости (durability)* (см. Свойства транзакций (ACID)) транзакции подразумевает сохранение результатов транзакции в БД даже при сбоях.

Хранение данных в оперативной памяти может приводить к потерям, например, при перезагрузке. Это считается нормальным, что приводит к необходимости хранения информации на дисках. Напомним, что с диска быстрее читать данные последовательно.

#### Типы сбоев

- **Локальный.** Сбой одной транзакции. Для восстановления достаточно откатить затронутую транзакцию.
- **Глобальный.** Сбой процесса СУБД, затрагивает все транзакции. Для восстановления достаточно откатить все незавершенные транзакции, а также заново применить все успешно завершённые транзакции.
- **Аппаратный.** Например, перезагрузка компьютера. С точки зрения СУБД, не существенно отличается от глобального сбоя.
- **Отказ оборудования.** СУБД не может восстановиться после этого типа сбоя. Однако, многие СУБД предоставляют *средства* для восстановления (например, запись данных на несколько дисков и синхронизация копий).

Свойство устойчивости не является абсолютным. Существуют сбои, при которых его нельзя поддержать.

**Восстановление после сбоя** Для восстановления БД достаточно сделать следующее:

- Успешные транзакции – зафиксировать;
- Откаченные транзакции – откатить;
- Незавершенные транзакции – откатить.

Существует несколько популярных подходов для отката:

**Shadow copy** Каждая транзакция пишет данные в новое место. При успешном завершении транзакции копия помечается успешной, пользователь уведомляется об успешной транзакции, и начинается запись из копии в БД. При сбое во время записи производится повторная запись. Проблема подхода заключается в частых чтениях и записях shadow copy, которые расположены в случайных местах на диске, что медленно.

**Transaction log** Данные пишутся сразу в БД, параллельно записывая изменения, примененные в рамках каждой транзакции, в журнал. Данный подход более популярен.

Журнал записывается в надежное хранилище изменений. Это означает, что его утрата есть невозстановимый сбой. Однако, записи ведутся последовательно, что делает данный подход быстрее предыдущего.

В журнал записываются: старые данные, новые данные, маркеры начала и завершения транзакции.

При завершении транзакции все изменения записываются в журнал, записывается маркер завершения транзакции, пользователь уведомляется о завершении транзакции.

### Реализация журнала

- **Постоянная запись на диск.** При записи каждого изменения в журнал существенно возрастает число операций, конкуренция за доступ к диску, а также накапливаются откаченные транзакции, которые в будущем не принесут пользы.
- **Запись при завершении.** В журнал при завершении транзакции записываются порожденные изменения. При больших изменениях это приводит к росту потребления памяти журналом.
- **Точки восстановления.** В журнал периодически записывается "слепок" состояния системы: текущие изменения, завершённые транзакции (не записанные ранее), откаченные транзакции (не записанные ранее), открытые транзакции. Создание точки восстановления требует приостановки изменений.

**Структура журнала** С использованием механизма точек восстановления, получаем следующую структуру журнала.

- **Точка восстановления;**
- **События:**
  - Идентификатор транзакции,
  - Указатель на *предыдущее* событие транзакции:
    - \* Начало транзакции,
    - \* Изменение,
    - \* Завершение транзакции,
    - \* Откат транзакции.

**Примеры** Рассмотрим пример сбоя и определим, что должно произойти с каждой из транзакций.

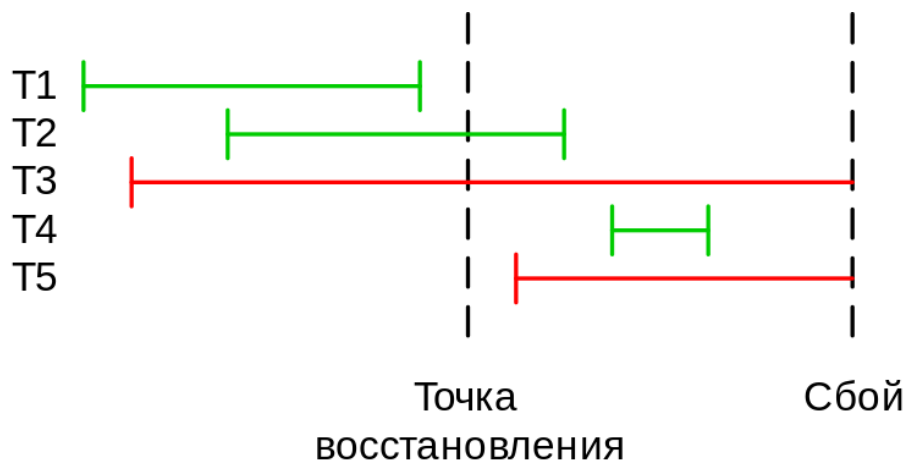


Рис. 29: Иллюстрация к определению проекции

Зеленым и красным цветом отмечены транзакции, которые должны быть завершены или откатены соответственно при восстановлении. Отметим, что эти решения однозначно определены гарантиями ACID транзакций.

### 1.13.3 Классический алгоритм восстановления

#### Фазы алгоритма

- **Разметка транзакций.** Каждая транзакция отмечается как *Redo* или *Undo*.
- **Откат транзакций.** Для помеченных как *Undo*.
- **Повтор транзакций.** Для помеченных как *Redo*.

### Фаза разметки транзакций

- Чтение журнала идет от последней точки восстановления до конца. Все открытые транзакции помещаются в *Undo*.
- При чтении маркера начала транзакция добавляется в *Undo*.
- При чтении маркера конца транзакция переносится из *Undo* в *Redo*.

### Фаза отката транзакций

- Чтение журнала идет от конца к началу, пока множество *Undo* не пусто.
- При чтении маркера начала транзакция удаляется из *Undo*.
- При чтении изменения оно откатывается, если транзакция в *Undo*.

### Фаза повторения транзакций

- Чтение журнала идет от последней точки восстановления до конца.
- При чтении маркера конца транзакция удаляется из *Redo*.
- При чтении изменения оно применяется, если транзакция в *Redo*.

**Утверждение 1.5.** После успешного выполнения всех фаз БД находится в корректном состоянии и гарантирует выполнения свойства устойчивости.

**Утверждение 1.6.** Рассмотрим все открытые транзакции. Для каждой из них найдем ближайшую точку восстановления из будущего. Все данные до самой ранней точки восстановления из рассматриваемых можно удалить, поскольку они не понадобятся при восстановлении.

#### 1.13.4 Отказ оборудования

До этого были рассмотрены методы восстановления при глобальном или аппаратном сбое. Рассмотрим методы борьбы с последствиями отказа оборудования.

**Репликация данных** Несколько БД, которые содержат одинаковые данные. Обычно разнесены географически. В процессе работы необходимо поддерживать синхронность данных на копиях, что приводит к необходимости использования распределенных транзакций. Реализация последних – технически сложная задача. При отказе достаточно назначить основной БД любую из оставшихся копий.

**Избыточность оборудования** Одна БД, которая работает параллельно с несколькими дисками или RAID. Запись происходит параллельно на каждое ПЗУ. При отказе диска достаточно его заменить и скопировать на него данные с другого диска или положиться на алгоритмы RAID при его использовании.

**Резервное копирование** Вся БД периодически копируется в отдельное хранилище. Для корректности копирования требуется приостановка обновлений данных. При отказе достаточно скопировать данные из резервной копии. Стоит отметить, что при этом данные будут актуальными на момент создания копии, поэтому при необходимости и возможности следует произвести повторное внесение данных.

## 1.14 Транзакции. Восстановление. Алгоритм ARIES

Про транзакции и восстановление БД после сбоев, можно прочитать в билете 1.13.

### 1.14.1 Алгоритм восстановления ARIES

#### Фазы алгоритма

- **Разметка транзакций.** Каждая транзакция отмечается как *Redo* или *Undo*.
- **Повторение истории.** Восстановление состояния системы на момент сбоя.
- **Откат транзакций.** Восстановление корректного состояния системы.

**Фаза разметки транзакций** Полностью эквивалентна классическому алгоритму. Может быть совмещена со следующей фазой.

#### Фаза повторения истории

- Чтение журнала идет от последней точки восстановления до конца.
- При чтении изменения оно применяется.

#### Фаза отката транзакций

- Чтение журнала идет по транзакциям из *Undo*, от конца к началу.
- При чтении изменения оно откатывается.

**Компенсационные записи** В текущей версии алгоритма нет прогресса восстановления при повторном сбое. Добиться этого можно путем введения **компенсационных записей**.

Будем производить следующие действия при необходимости отката изменений:

- **Откат изменения;**
- **Запись на диск;**
- **Внесение компенсационной записи.** Запись означает, что откатываемое изменение, а также все изменения, которые идут в логе позднее, были успешно откаты.

**Утверждение 1.7.** Компенсационные записи фиксируют прогресс восстановления БД и исключают повторный откат изменения при очередном восстановлении. Таким образом, повторные сбои не мешают завершению восстановления.

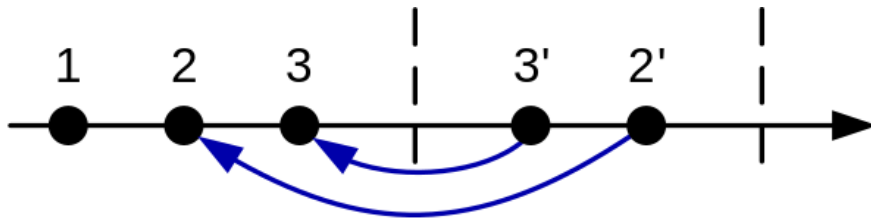


Рис. 30: Иллюстрация компенсирующих записей при повторных сбоях

### 1.14.2 Сравнение алгоритмов

#### Число проходов

- Классический алгоритм. 3
- Алгоритм ARIES. 2

#### Рост журнала транзакций

- Классический алгоритм. В журнал добавляются записи отмены.
- Алгоритм ARIES. В журнал добавляются компенсирующие записи.

#### Повторные сбои

- Классический алгоритм. Полный перезапуск процесса восстановления.
- Алгоритм ARIES. Постепенное завершение.



## 1.15 Транзакции. Параллельное исполнение. Блокировки.

### 1.15.1 Параллельное исполнение

Напомним, что свойство *изоляции* (*isolation*) транзакции (см. Свойства транзакций (ACID)) , что она должна исполняться так, как будто она в системе одна, а также корректные по отдельности транзакции должны быть корректными в совокупности. Следовательно, транзакции могут исполняться параллельно.

Обычно параллельные транзакции выполняются в разных потоках, что приводит к необходимости использования блокировок для синхронизации.

#### Примеры конфликтов

- **Потерянное обновление (1).** Обновление, сделанное транзакцией 1, потеряно.

Транзакция 1	Транзакция 2
retrieve T	
	retrieve T
update T	
	update T
commit	
	commit

- **Потерянное обновление (2).** Обновление, сделанное транзакцией 1, потеряно.

Транзакция 1	Транзакция 2
	update T
update T	
commit	
	rollback

- **Незафиксированное изменение.** Значение, полученное транзакцией 1, не было зафиксировано.

Транзакция 1	Транзакция 2
	update T
retrieve T	
	rollback
commit	

- **Несо согласованное состояние.** Значение, полученное транзакцией 1, не могло быть получено в согласованном состоянии.

Транзакция 1	Транзакция 2
retrieve T_1	update T_1 = T_1 - 10
retrieve T_2	
	update T_2 = T_2 + 10
commit	
	commit

## Типы конфликтов

- **Чтение-чтение.** Нет конфликтов.
- **Чтение-запись.** Некорректное состояние.
- **Запись-чтение.** Зависимость от незафиксированного изменения.
- **Запись-запись.** Потерянное обновление.

### 1.15.2 Блокировки

Нам потребуются многоуровневые блокировки на фрагменты данных: разделяемая (для чтения, *S*) и эксклюзивная (для записи, *X*). Отсутствие блокировки обозначим -.

**Определение.** *Протокол двухфазной блокировки.* Для чтения требуется получение *S*, для записи – *X*, при завершении или откате транзакции требуется **последовательно** освободить все блокировки. Важно, что в первую фазу количество блокировок только растет, а во вторую – только уменьшается.

**Определение.** *Строгий протокол двухфазной блокировки.* Аналогично предыдущему определению, но все блокировки во второй фазе необходимо отпускать **после завершения или отката** транзакции.

### Взаимная блокировка (ВБ) Пример взаимной блокировки

Транзакция 1	Транзакция 2
retrieve T_1	
	retrieve T_2
update T_2	
	update T_1

Транзакция 1 и транзакция 2 сначала берут *S* блокировки на чтение *T\_1* и *T\_2* соответственно. Затем транзакция 1 пытается взять *X* блокировку на запись в *T\_2*, что не удастся сделать, пока транзакция 2 владеет *S* блокировкой *T\_2*. Аналогично, транзакция 2 не может взять *X* блокировку на запись, поскольку так как транзакция 1 владеет *S* блокировкой на *T\_2*.

## Устранение ВБ

- **Построение графа ожиданий.** Наличие цикла в таком графе свидетельствует о ВБ. На практике графы слишком большие, поэтому данный подход менее популярен.
- **Выставление таймаутов.** Отсутствие прогресса на протяжении долгого времени вероятно свидетельствует о ВБ.

При обнаружении ВБ следует откатить одну из транзакций. Также, если СУБД владеет транзакцией, то есть может ее перезапустить, то это следует сделать.

**Предотвращение ВБ** Пусть транзакция  $A$  претендует на блокировку, конфликтующую с блокировками транзакции  $B$ . Возможны следующие стратегии.

- **Стратегия ожидание-отмена.**
  - $A$  началась раньше  $B$  –  $A$  ожидает;
  - $A$  началась позже  $B$  –  $A$  отменяется (и, по возможности, перезапускается);
- **Стратегия отмена-ожидание.**
  - $A$  началась раньше  $B$  –  $B$  отменяется (и, по возможности, перезапускается);
  - $A$  началась позже  $B$  –  $A$  ожидает;

ВБ в каждой стратегии исключается, поскольку в графе ожидания ребра идут только от старшей к младшей или от младшей к старшей транзакциям соответственно. Стоит отметить, что стратегии порождают много лишних откатов.

## Упорядочиваемость

**Определение.** *Упорядочиваемость (serializability)* – любая последовательность исполнения транзакций эквивалентна (равны состояния до начала и после окончания исполнения) некоторому последовательному исполнению.

**Утверждение 1.8.** Строгий протокол двухфазной блокировки гарантирует упорядочиваемость.

**Утверждение 1.9.** Протокол двухфазной блокировки гарантирует упорядочиваемость.

### 1.15.3 Восстановление и параллелизм

Рассмотрим следующий пример.

Транзакция 1	Транзакция 2
retrieve T commit	update T  rollback

Транзакция 1 фиксирует изменения, внесенные транзакцией 2. Однако, в будущем транзакция 2 может быть откатена, например, из-за сбоя. Таким образом, фиксируется изменение, зависящее от незафиксированного.

**Определение. Критерий восстанавливаемости.** Если транзакция *A* использует значения, обновленные транзакцией *B*, то *A* должна завершиться позже, чем *B*.

В случае противоречий возникают взаимные блокировки, способы борьбы с которыми были рассмотрены выше. Однако, это может привести к цепочкам форсированных откатов (откат транзакции *B* форсирует откат транзакции *A*).

**Утверждение 1.10.** При строгом протоколе двухфазной блокировки цепочки отката отсутствуют.

**Доказательство.** Транзакция *A* сможет получить блокировку на чтение только после отпущения эксклюзивной блокировки на запись транзакцией *B*. Последнее в строгом протоколе двухфазной блокировки произойдет только после завершения транзакции *B*. ■

### 1.15.4 Гранулярность блокировок

- **Блокировка поля записи.** Блокируются отдельные поля каждой записи. Не используется на практике.
- **Блокировка записей.** Блокируются отдельные записи. Дает высокий параллелизм и требует больших ресурсов.
- **Блокировка страниц.** Блокируется страница памяти, на которой расположена запись. Более практично по сравнению с блокировкой отдельных записей, поскольку на одной странице может быть расположено много записей.
- **Блокировка индексов.** Блокируется элемент (например, поддерево в В-дереве или корзина в хеш-индексе) или страница индекса. Запрещает добавление или удаление в рамках заблокированного индекса.
- **Блокировка таблиц.** Блокируется таблица целиком. Требует мало ресурсов и предоставляет низкий параллелизм.
- **Блокировка БД.** Используется для резервного копирования, изменения определения таблиц и представлений, изменения хранимых процедур и функций и изменения прав доступа.

Аномалия **фантомные записи** – при повторном чтении могут появиться новые записи. Возможна при гранулярности блокировки меньше, чем по таблицам.

## 1.16 Транзакции. Параллельное исполнение. Уровни изоляции.

**Уровни изоляции транзакций** Мы рассматриваем следующие уровни изоляции транзакций. Все, кроме “Слепок”, определены в стандарте SQL.

- Упорядочиваемый (serializable),
- “Слепок” (snapshot),
- Повторяемое чтение (repeatable read),
- Чтение зафиксированных (read committed),
- Чтение незафиксированных (read uncommitted).

### 1.16.1 Упорядочиваемый

Дает наиболее сильные гарантии с самой низкой скоростью исполнения. Детали реализации были рассмотрены в предыдущем билете.

### 1.16.2 “Слепок”

Каждая транзакция работает со своим “слепоком” БД. Вносит изменения в режиме сору-он-write. При реинтеграции изменений они фиксируются, при отсутствии конфликтов изменений.

Является аналогом упорядочиваемого уровня изоляции с меньшими гарантиями, используется в базах-“версионниках”, в которых синхронизация основана на версиях вместо блокировок.

**Аномалия “косая запись”** На данном уровне изоляции возможна аномалия “косая запись”. Она возникает при одновременном обновлении разных записей, которые вместе должны гарантировать некоторый инвариант.

**Пример.** Положим инвариант  $t\_1 + t\_2 \geq 0$ .

- Транзакция 1

```
if t_1 + t_2 >= DELTA begin
    t_1 = t_1 - DELTA
end if
```

- Транзакция 2

```
if t_1 + t_2 >= DELTA begin
    t_2 = t_2 - DELTA
end if
```

Реинтеграция изменений пройдет успешно, поскольку записи идут в разные переменные. Однако, при параллельном исполнении инвариант может быть нарушен.

### **1.16.3 Повторяемое чтение**

Уровень изоляции гарантирует, что при повторном чтении значения не будут меняться. Исключение – запись, произведенная самой транзакцией. Реализуется путем взятия блокировок записей или страниц на чтение.

**Аномалия “фантомная запись”** При повторном чтении могут появиться новые записи. Возможно при параллельном исполнении другой транзакции.

### **1.16.4 Чтение зафиксированных**

Уровень изоляции гарантирует, что читаемые значения зафиксированы другими транзакциями. Реализуется путем взятия *частичных* блокировок записей или страниц на чтение.

**Аномалия “неповторяемое чтение”** При повторном чтении могут значение записи может измениться. Возможно при параллельном исполнении другой транзакции.

### **1.16.5 Чтение незафиксированных**

На уровне изоляции не используются блокировки, что обеспечивает наивысшую скорость. По стандарту SQL разрешено только чтение. Используется для сбора статистики.

**Аномалия “грязное чтение”** Может быть прочитано некорректное значение.

## 1.17 Секционирование

**Определение.** *Секционирование* – разбиение таблицы на фрагменты, хранящиеся в разных местах (в случае разных компьютеров называется *шардинг*). Используется для увеличения скорости чтения за счет параллельного обращения.

Различают два вида секционирования: **вертикальное** и **горизонтальное**.

### 1.17.1 Вертикальное секционирование

Таблица разбивается по столбцам. Возможно при корректности соединения (5 НФ). Реализуется посредством проекции и соединения.

#### Преимущества

- Отделение данных, к которым часто обращаются, от тех, к которым обращаются редко.
- Защита информации.
- Поддерживается во многих СУБД для CLOB и BLOB.

#### Недостатки

- Нет специальной поддержки в СУБД. Считается, что проекции и соединения для указанных целей достаточно.
- Зависимость от представления (соединенных данных). Некоторые СУБД накладывают ограничения на представления, например, запрещают создавать внешние ключи на них.
- Необходимость обновляемых представлений также не гарантирована.

**Пример** Рассмотрим исходную таблицу.

```
Students(SId, GId, FirstName, LastName, PassSeries,
        PassNo, PassIssued, Photo)
```

Разобьем ее на секции.

```
StudentData(SId, GId, FirstName, LastName)
StudentPasses(SId, PassSeries, PassNo, PassIssued)
StudentPhotos(SId, Photo)
```

Обращение к фото (StudentPhotos) происходит значительно реже, чем к основным данным студента (StudentData). Также таблица с персональными данными (StudentPasses) требует повышенных прав доступа. Таким образом, были использованы все преимущества вертикального секционирования.

Создадим также представление для работы с исходной таблицей.

```
create view Students as StudentData
    natural join StudentPasses
    natural join StudentPhotos;
```

### 1.17.2 Горизонтальное секционирование

Таблица разделяется по строкам. Корректно, когда каждая строка попадает ровно в одну секцию. Реализуется посредством фильтрации и объединения.

#### Преимущества

- Отделение данных, к которым часто обращаются, от тех, к которым обращаются редко. Например, чаще всего старые данные нужны реже новых.
- При уменьшении размера секции уменьшается размер индекса.
- Требуется встроенная поддержка.
- Прозрачно для пользователя.

#### Недостатки

- В некоторых случаях может приводить к замедлению работы.

**Пример** Рассмотрим исходную таблицу.

`Points (SId , CId , Points , Date)`

Введем секционирование по Date:

- `Points2021-1` – оценки за весенний семестр 2021,
- `Points2021-2` – оценки за осенний семестр 2021,
- `Points2020-1` – оценки за весенний семестр 2020,
- ...

#### Методы секционирования

- **Простые.**
  - По диапазонам значений,
  - По значениям,
  - По хешу.
- **По выражению.** Поддерживаются реже.
- **Составные.**
  - По диапазонам и хешу,
  - ...



**Пример** Секционирование по диапазонам.

```
create table Points (...)  
partition by range(Date) (  
    partition pHist values less than '2021-02-01',  
    partition p2021s1 values less than '2021-07-01',  
    partition p2021s2 values less than '2022-02-01',  
    partition pFuture values less than maxvalue  
);
```

maxvalue – максимальное теоретическое значение.

**Пример** Секционирование по значениям.

Чаще используется для перечислений.

```
create table Points (...)  
partition by list(Term) (  
    partition pHist values in ('t2020-1', 't2020-2', ...),  
    partition p2021s1 values in ('t2021-1'),  
    partition p2021s2 values in ('t2021-2'),  
    partition pFuture values in ('t2022-1')  
);
```

При таком подходе секция может быть не определена при записи.

```
insert into Points (Term) values ('t2001-1');
```

При чтении из несуществующей секции будет получен пустой результат.

**Пример** Секционирование по хешу.

Хешируется по набору столбцов. Работает эффективно при хорошей и быстрой хеш-функции.

```
create table Points (...)  
partition by hash(Term)  
partitions 4;
```

**Пример** Секционирование по выражению.

Зависит от определенной в БД функции.

```
create table Points (...)  
partition by year(Date) (  
    partition pHist values less than 2021,  
    partition pCurrent values less than 2022,  
    partition pFuture values less than maxvalue  
);
```

**Пример** Составное секционирование.

Секции разбиваются на подсекции, возможно, по разным атрибутам.

```
create table Points (...)  
partition by year(Date)  
subpartition by hash(Term) (  
    partition History values less than 2021 (  
        subpartition History1, subpartition History2  
    ),  
    partition Current values less than 2022 (  
        subpartition Current1, subpartition Current2  
    )  
);
```

### Управление секциями

**Замечание.** Данные команды не входят в стандарт SQL, поэтому синтаксис может отличаться в зависимости от СУБД.

#### Удаление секции

```
alter table <table> drop partition <section>;
```

#### Разбиение секции

```
alter table <table> reorganize <section> into (...);
```

#### Перехеширование

— *Add count of partitions*

```
alter table <table> add partition <count>;
```

— *Delete count of partitions*

```
alter table <table> coalesce partition <count>;
```

**Утверждение 1.11.** Оптимизатор владеет информацией о секциях. В частности, где какие данные находятся.

```
select * from Points where Date = '2021-12-06'
```

При таком запросе выборка будет производиться только из секции 2021 года.

**Секционирование и индексы** Можно определить следующие индексы:

- **Локальный** – один на секцию. Ускорение при выборе секций.
- **Глобальный** – один на таблицу. Также ускорение при выборе секций.
- **Секционированный** – разбит на секции. Обеспечивает согласованное секционирование (могут помочь при склеивании секций).

**Секционирование и ключи** Лучше всего, если множество столбцов, по которым идет секционирование, образует подключ. Еще лучше – подключ всех ключей (например, если внешние ключи на таблицу ссылаются на разные ее ключи).

## 1.18 Репликация

**Определение.** Репликация – поддержание одинаковых данных на нескольких узлах.

### 1.18.1 Реализация репликации

Репликация бывает **синхронной** (с использованием распределенных транзакций) и **асинхронной** (информация до реплик доходит с задержками). С другой стороны, различают схемы репликации с **основной копией** и **симметричную**.

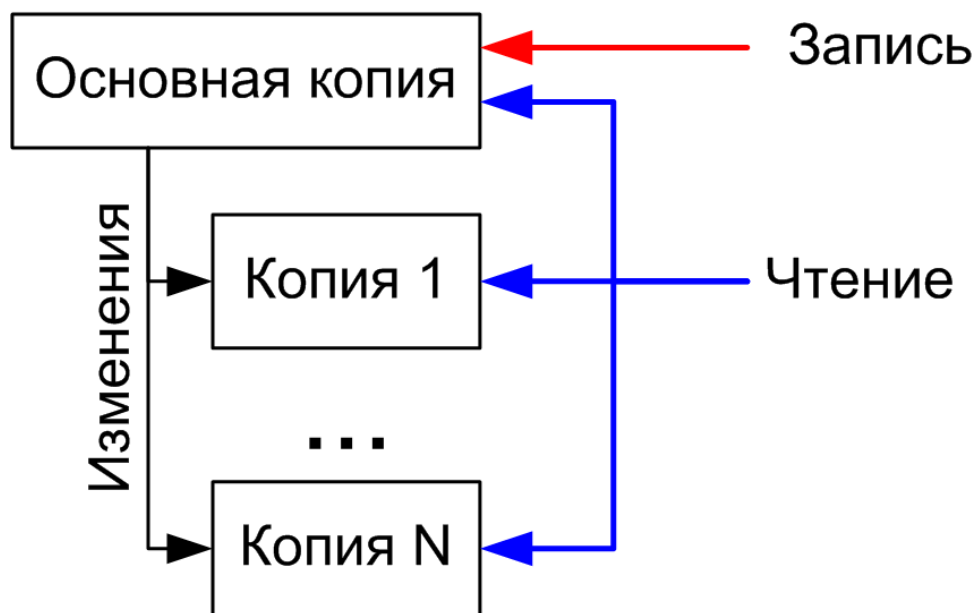


Рис. 31: Схема репликации с основной копией

**Репликация с основной копией** Чтение данных можно производить из любой копии БД, в то время как запись – только в основную. Согласованность всех копий обеспечивается за счет проверки при записи в основную копию. Данная схема подходит, если число записей сильно меньше числа чтений.

**Симметричная репликация** Чтение и запись производятся в каждую копию независимо, все копии равноправны и автономны. Для борьбы с противоречивыми изменениями в данной схеме требуется синхронность изменений.

**Реализация репликации** Данные об изменениях можно рассылать из журнала транзакций. Можно рассылать данные в различной гранулярности:

- **Репликация операторов.** При таком подходе используется меньше данных. Однако, каждый оператор должен быть детерминированным, а также необходимо учитывать взаимный порядок выполнения транзакций. Сложно для реализации.

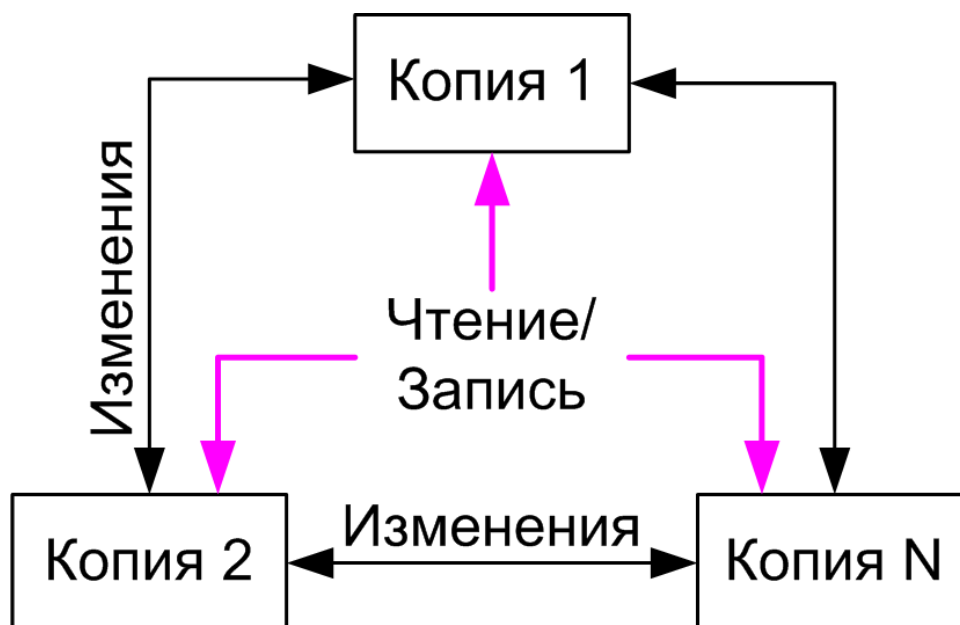


Рис. 32: Схема симметричной репликации

- **Репликация записей.** Рассылается информация об изменении отдельных записей. При таком подходе нет требования к детерминированности. Однако, крупные обновления данных приведут к рассылке больших сообщений.

### 1.18.2 Применения репликации

**Вертикальное масштабирование** При необходимости вертикального масштабирования (наращивания мощности системы) следует использовать **асимметричную схему**. Напомним, что ее использование целесообразно, когда количество изменений гораздо меньше количества чтений.

Применяется в ситуациях, когда допустима асинхронность. Например, в Web-серверах и ERP-системах.

**Горизонтальное масштабирование** В ситуациях, когда необходимо производить множество локальных операций, например, в разных географических областях, применяется **симметричная схема**. Каждая реплика отвечает за определенные данные в зависимости от запроса. Также этот подход полезен в случае непостоянной связи.

**Повышение доступности** Для повышения доступности данных следует использовать **асимметричную схему**, которая позволяет менять основную реплику при выходе из строя прошлой. В случае синхронной репликации потери данных отсутствуют, однако, в случае асинхронной вышедшая из строя основная реплика могла не успеть разослать другим репликам изменения.

Также полезно **резервное копирование БД**. В асимметричной схеме для этого достаточно создать реплику, с которой постоянно синхронизируется основная. Для

создания резервной копии достаточно отключить такую реплику, скопировать данные, включить обратно в систему и восполнить пропущенные обновления.

**Преобразование данных** Используется **асимметричная схема**, которая предоставляет отлаженный способ получения всех изменений данных. На основе них можно строить преобразование данных: изменение формата хранения, консолидация, унификация, подсчет статистики и так далее. Таким образом, преобразования происходят при репликации.

## 1.19 Распределенные транзакции

**Определение.** *Распределенные транзакции* – транзакции, в которых участвует несколько узлов. Каждая транзакция должна быть либо применена, либо откатена на всех узлах одновременно.

В рамках распределенных транзакций необходимо рассматривать свои как *участников*, так и *коммуникаций*.

**Протокол двухфазной транзакции** Классический подход реализации распределенных транзакций. Один из узлов выбирается координатором (как правило, узел, на который пришел запрос). Это означает, что при параллельном исполнении нескольких транзакций несколько узлов одновременно могут быть координаторами.

Далее каждая транзакция проходит через две фазы.

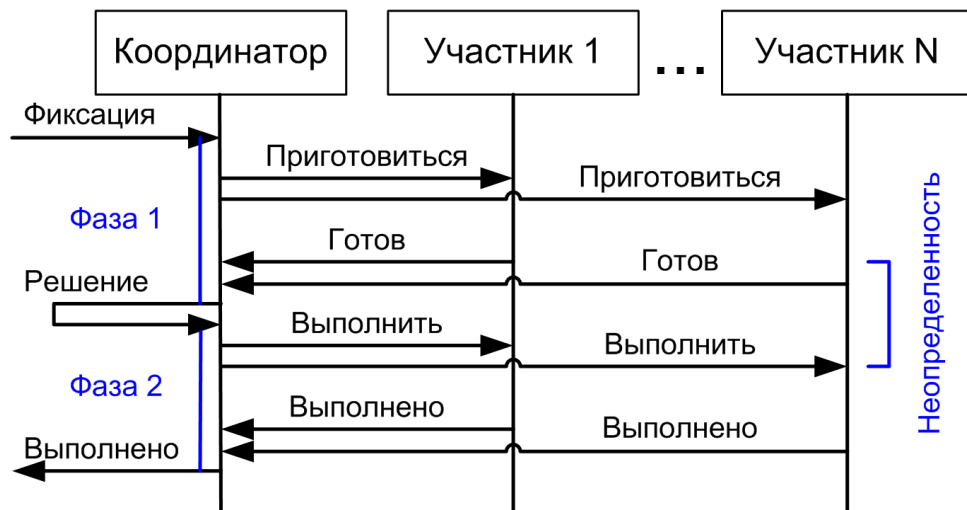


Рис. 33: Схема протокола двухфазной транзакции

- **Фаза подготовки.** Координатор рассылает информацию об изменении всем другим копиям. Далее эти копии должны ответить, что готовы зафиксировать изменение (нет противоречий, данные находятся в согласованном состоянии, нет иных ошибок).
- **Решение координатора.** После получения ответов от всех копий, координатор принимает решение, выполнять ли фиксацию данных. Решение положительное, если от все пришедшие ответы от других копий положительные.
- **Фаза выполнения.** При положительном решении координатор рассылает сообщение о том, что изменения должны быть зафиксированы, и ожидает подтверждения этого от всех копий.
- **Результат.** Если координатор получает подтверждения от всех копий, то транзакция считается выполненной.

Если *координатор* не получил хотя бы одно подтверждение на первой фазе, то транзакция автоматически откатывается. Как только принимается решение о фиксации, оно записывается в журнал транзакций координатора – так копии смогут заново воспроизвести решение при восстановлении. Однако, если координатор не получил хотя бы одно подтверждение на второй фазе, то это неразрешимая ситуация.

*Копии* при сбое до отправки подтверждения на первой фазе должны откатить транзакцию, поскольку координатор не мог принять решения о фиксации без подтверждения. При сбое до получения решения о фиксации копия должна запросить решение снова у координатора. При получении решения о фиксации копия записывает в свой журнал транзакций изменение для восстановления в будущем.

Однако, задача теоретически не разрешима. То есть существует сценарий, при котором ни координатор, ни реплика не смогут продвинуться (но эти случаи выродены). Это показывает отсутствие решения следующей задачи.

**Задача двух генералов** Генералам двух армий необходимо напасть на врага. Они достигнут успеха только если нападут одновременно. Им необходимо прийти к консенсусу: нападать или нет. Генералы могут посылать гонцов с сообщениями, которых могут убить в пути.

**Утверждение 1.12.** Не существует гарантированного решения задачи двух генералов.

*Доказательство.* Предположим противное: существует конечный протокол, при использовании которого можно прийти к консенсусу. Следовательно, существует минимальное по включению множество сообщений, которые необходимо отправить. Предположим, что последнее сообщение не было доставлено. Тогда невозможно принять решение. Противоречие. ■

**Протоколы предполагаемой фиксации и отката** Как было показано выше, копии могут запрашивать решение у координатора повторно. Таким образом, координатору необходимо сохранить информацию о решении до тех пор, пока не получит подтверждения от всех копий. Рассмотрим два подхода оптимизации объема памяти, занимаемого решениями координатора.

- **Протокол предполагаемой фиксации.** При решении о фиксации транзакции она “забывается” координатором. Если координатор получает запрос о решении по неизвестной ему транзакции, то он отвечает, что ее нужно зафиксировать. Данный подход позволяет сэкономить память, если число зафиксированных транзакций сильно превышает число откаченных.
- **Протокол предполагаемого отката.** При решении об откате транзакции она “забывается” координатором. Если координатор получает запрос о решении по неизвестной ему транзакции, то он отвечает, что ее нужно откатить. Данный подход позволяет сэкономить память, если число откаченных транзакций сильно превышает число зафиксированных.



**Замечание.** Протокол предполагаемой фиксации может привести к ложной фиксации. Например, если информация о решении была ошибочно утеряна. В таком случае всем другим участникам будет приходить ответ о фиксации, которой в действительности не было. Поэтому рекомендуется использовать протокол предполагаемого отката.

## 1.20 Распределенные базы данных. Цели и проблемы

### 1.20.1 Цели распределения

Определим ряд целей, которые стремятся достигнуть при разработке распределенных баз данных.

- **Децентрализация.**

- *Локальная независимость.* Узел должен продолжать функционировать даже в изоляции от других.
- *Отсутствие центрального узла.* При наличии такового отказ этого узла приведет к отказу системы целиком, что есть уязвимость.
- *Непрерывное функционирование.* Хранилище данных должно быть надежным и доступным.

- **Прозрачность.**

- *Независимость от расположения.* Логически программы должны работать на произвольном узле и обеспечивать удаленный доступ к данным. Иными словами, распределенная БД с точки зрения пользователя должна себя вести так же, как если бы она представляла собой одну большую. При этом нет требований к временным задержкам.
- *Независимость от фрагментации.* Программы не должны зависеть от информации об узле, на котором находятся данные. Доступ должен быть унифицированным.
- *Независимость от репликации.* Должна быть поддержана автоматически.

- **Распределенные транзакции.**

- *Поддержка распределенных запросов.* Получение данных с разных узлов в рамках одного запроса.
- *Поддержка распределенных записей.* Несколько узлов должны участвовать в единой транзакции. При этом необходима согласованность фиксации и отката.

- **Независимость от окружения.**

- *Независимость от аппаратуры.* Унифицированное представление данных.
- *Независимость от ОС.* Прозрачная поддержка различных ОС и конвертация данных. Один узел может быть запущен на Windows, а другой – на Ubuntu.
- *Независимость от сети.* Прозрачная поддержка различных сетей. Копии могут быть как расположены внутри одного датацентра, так и разнесены географически.

- *Независимость от типа СУБД*. Прозрачная поддержка различных СУБД и конвертация данных.

Из требований децентрализации следует равноправие узлов, что влечет за собой решение задач распределенного консенсуса, в частности, выбора лидера.

**Замечание.** Важно, что распределенные базы данных отличаются от репликации тем, что каждый узел реально владеет данными, расположенными на нем. То есть каждая запись лежит ровно на одном узле.

### 1.20.2 Проблемы распределения

**Теорема 1.13** (CAP-теорема). Одна система может обладать не более чем двумя свойствами из нижеуказанных одновременно:

- *Consistency (C)* – информация на разных узлах согласована;
- *Availability (A)* – система отвечает на запросы в любой момент времени;
- *Partition tolerance (P)* – система работает корректно при обрыве связи между узлами.

**Замечание.** Важно, что вышеуказанные свойства не бинарны, в то время как доказательство теоремы верно только для бинарных свойств. Например, если допускается, что согласованность может периодически нарушаться, то применение теоремы некорректно, и такую систему теоретически можно реализовать. Таким образом, систему со всеми свойствами реализовать возможно, но частично уступив в некоторых свойствах.

Обычно теорему рассматривают с точки зрения свойства *P*, поскольку при отсутствии *P* система не разделена.

- Система реализуема при частичном отказе от *A*. Следовательно, при обрыве связи в системе функционирует только одна ее часть.
- Система реализуема при частичном отказе от *C*. Следовательно, системе необходим механизм объединения состояний.

Поскольку свойства CAP-теоремы слишком сильные для одновременной поддержки, рассматривают подход с ослабленными свойствами.

**Определение.** *Свойства BASE.*

- *Basically Available (BA)* – сбой узла приводит к отказу только для части пользователей;
- *Soft-state (S)* – изменения в системе могут происходить не только по причине внешнего вмешательства (например, при восстановлении соединения);
- *Eventual consistency (E)* – несогласованность устраняется со временем.

**Восстановление согласованности** При восстановлении согласованности можно использовать следующие подходы: при чтении, при записи или асинхронно. Первые два варианта замедляют соответствующие операции, асинхронный подход требует создания специального процесса. Для разрешения используются механизмы меток времени и векторные часы.

**Оптимизация запросов** В распределенной системе задача оптимизации запросов дополнительно усложняется. Дополнительно необходимо минимизировать количество доступов к удаленным данным.

Добиться этого можно следующими средствами:

- **Выбор узлов получения и обработки данных.** Например, лучше выполнять запрос на узле, который владеет большей долей затрагиваемых данных локально для минимизации коммуникаций.
- **Полусоединения.** Позволяют запрашивать с других узлов только необходимые данные.
- **Применение репликации.** Для получения данных может быть достаточным обращение к реплике.

**Управление параллельностью** Базы данных, основанные на блокировках, требуют реализации механизмов распределенных блокировок и детектирования распределенных взаимных блокировок. Это усложняет параллелизацию запроса и изоляцию транзакций. Основные средства для решения: использование более сложных распределенных алгоритмов и схемы с основной копией.

**Управление каталогом** Каталог также может быть распределен. Однако, традиционно используются СУБД, в которых считается, что каталог один. Информация о каталоге полностью реплицируется на все узлы.

**Независимость от окружения** Для обеспечения независимости от реализации СУБД используется механизм шлюзов, которые решают задачи конвертации данных и перезаписи запросов. Например, Oracle предоставляет шлюз с MySQL. Со стороны шлюх выглядит как очередная сущность СУБД. Также ожидается, что он поддерживает распределенные транзакции и блокировки.

,C,

,C,,C,

## 1.21 Трактовки null и операции с ним

### 1.21.1 Трактовки null

SQL допускает использования в качестве значения null по умолчанию. Однако, это значение может трактоваться по-разному.

Рассмотрим для этого следующий пример: пусть в БД “Университет” идентификатор группы студента равен null, то есть значение отсутствует. Это может означать одно следующего:

- **Значение не известно.** Неизвестно, в какой группе обучается студент;
- **Значение не верно.** Студент учится в группе, имя которой неверно;
- **Значение еще не существует.** Студент зачислен, но не определен в группу;
- **Значение уже не существует.** Группа отчисленного студента;
- **Значение не имеет смысла.** Студент из другого университета;
- **Значение не доступно.** У пользователя нет права узнать группу.

Заметим, что отсутствующие значения можно реализовать без null – используя необязательную связь 1:1. Однако, null может появиться в результате множественных операций и внешних соединений. Поэтому полностью null исключить нельзя.

### 1.21.2 Операции с null

**Тернарная логика** С точки зрения SQL *результат* результат логического выражения может быть true, false или unknown. Однако, тип boolean может *принимать значения* true, false или null. С практической точки зрения, unknown и null не отличаются.

Рассмотрим основные логические операции.

AND	true	unknown	false
true	true	unknown	false
unknown	unknown	unknown	false
false	false	false	false

OR	true	unknown	false
true	true	true	true
unknown	true	unknown	unknown
false	true	unknown	false

NOT	true	unknown	false
	false	unknown	true

Интуитивно,  $\text{true} > \text{unknown} > \text{false}$ . **AND** возвращает меньшее из аргументов, **OR** – большее из аргументов, **NOT** – обратное по порядку. Для сравнения используются два оператора: бинарный **=** и унарный **is [not] (true | unknown | false)**.

=	true	unknown	false
true	true	unknown	false
unknown	unknown	unknown	unknown
false	false	unknown	true

is	true	unknown	false
is true	true	false	false
is unknown	false	true	false
is false	false	false	true
is not true	false	true	true
is not unknown	true	false	true
is not false	true	true	false

**Пример** Рассмотрим импликацию: через **OR** ( $A \rightarrow B = (\text{not } A) \text{ or } B$ ) и через **AND** ( $A \rightarrow B = \text{not } (A \text{ and not } B)$ ).

OR $\rightarrow$	true	unknown	false
true	true	unknown	false
unknown	true	unknown	unknown
false	true	unknown	true

AND $\rightarrow$	true	unknown	false
true	true	unknown	false
unknown	true	unknown	unknown
false	true	true	true

**Утверждение 1.14.** Законы Де Моргана неприменимы в тернарной логике.

**Операции с null** null заразен. Следующие операции при неизвестном значении хотя бы одного аргумента возвращают null:

- **Операции сравнения:**  $=, <>, <, <=, >, >=$ ;
- **Арифметические операции:**  $+, -, *, /$ ;
- **Конкатенация строк:**  $||$ ;
- **Подзапрос вхождения:** `in`:

– `in(true, false, null) == null`.

Для избавления от null используется операция `coalesce(v1, v2, ...)`, принимающая произвольное число аргументов и возвращающая первый из них, не равный null, или null при отсутствии таковых.



**Дубликаты** `null != null`, следовательно, одинаковые кортежи, содержащие `null`, не равны:

$$(1, \text{null}) \neq (1, \text{null}).$$

Пусть в отношении  $R$  содержится такой кортеж. Тогда:

$$R \cup R \neq R \quad (1.1)$$

$$R \cap R \neq R \quad (1.2)$$

$$R \bowtie R \neq R \quad (1.3)$$

При объединении такие кортежи будут продублированы, при пересечении и естественном соединении – удалены.

### Спецэффекты

- $x = x$ . Результат – `true` или `null`;
- $x <> x$ . Результат – `true` или `null`;
- $x \text{ or } x$ . Результат –  $x$
- $x \text{ or not } x$ . Результат – `true` или `null`;
- $x \text{ and not } x$ . Результат – `false` или `null`;
- $x \text{ or not } x$ . Результат – `true` или `null`;
- **Отсутствуют транзитивность и рефлексивность сравнения.**

### Ключи

- **Основной ключ.** Не могут содержать `null` по стандарту.
- **Дополнительный ключ.** Использование `null` разрешено.
- **Простой внешний ключ.** `null` разрешен – такое значение трактуется как отсутствующая ссылка, по ней не будет проходить соединение.
- **Составной внешний ключ.** Поддерживаются внешние ключи, отсутствующие целиком. Если отсутствует только часть ключа, то «ничем хорошим это не закончится».

**Интуитивность** Рассмотрим пример. Ожидается, что запрос возвращает информацию о студентах, не учащих в определенной группе:

```
select * from Students where GId <> 'M34391'
```

Пусть поле GId может отсутствовать. Корректность запроса зависит от смысла отсутствующего значения. Если смысл – “значение не известно”, то неизвестно, должен ли такой кортеж попасть в результат. Если же “значение неприменимо” (например, студент другого университета) – то такой кортеж точно не должен быть в ответе.

Рассмотрим другой пример. Ожидается, что запрос возвращает информацию о всех студентах:

```
select * from Students where GId <> 'M34391'
union
select * from Students where GId = 'M34391'
```

В действительности будет получена информация по всем студентам, у которых группа *не отсутствует*.

**Вывод** При работе с данными, значения которых могут отсутствовать, нужно как учитывать спецэффекты тернарной логики и операторов при работе с null, так и смысл, который вкладывается в отсутствующее значение.

### 1.21.3 Операции с null в SQL

#### Предикаты

- **DML.** В результате `where` и `having` остается только `true`. Все другие значения преобразовываются в `false`, включая `unknown`.
- **DDL.** В `check` не подходят только `false`. Все другие значения преобразовываются в `true`, включая `unknown`. То есть ограничение, вернувшее `unknown` считается удовлетворяющим.

**Различимость** Сравнение двух `null` возвращает `unknown`. При использовании его в `where` результат преобразуется в `false`. Следовательно, два `null` не равны. Кроме того, они не различимы.

Неразличимость не эквивалентна неравенству. Например, при использовании `distinct` из двух одинаковых кортежей с отсутствующими полями в результате будет один. Аналогично, `group by` породит только одну корзину.

**Столбцы** По умолчанию, столбцы допускают `null` в качестве значения:

```
birthday date
```

Для запрета отсутствующих значений это требуется явно указать:

```
birthday date not null
```

**Проверки значений** Для надежной проверки, чему равно значение из тернарной логики, используется следующий оператор:

```
<value> is [not] {null | true | false | unknown}
```

`is null` – способ проверить, равно ли что-либо `null`. `is unknown` применяется для булевских значений. Однако, для булевских значений `unknown` представляется как `null`. На практике, популярных СУБД эти два значения – одно и то же.

#### Прочие спецэффекты

- `exists`. Возвращает только `true` или `false`. Если все строки состоят из `null` – результат равен `false`.
- **Агрегирующие функции.** По умолчанию пропускают `null`. Если в результате остается пустой набор аргументов, то результат будет равен `null`. Единственное исключение – `count(*)` – считает число строк.
- `order by`. Можно указать `nulls first` или `nulls last` и определить, считать `null` меньше или больше других значений соответственно.

## 2 Практика