# Базы данных, V семестр

**Осень 2016, лектор: Барашев Дмитрий**

**Автор: Дима Лапшин**

Собрано: 18 января 2017 г. 21:36

---

## Оглавление

# Глава 1

# Введение

## 1.1. Организационное

Отчётность — дифференцированный зачёт по теории, письменный. Если по практикам всё плохо — минус балл. В начале лекции — разминка, небольшой тест на небольших бумажке. В конце лекции — контрольные вопросы на Stepik-е, поощерение за выполнение теста на месте быстро и правильно — печеньки.

Про практику — две контрольные работы: по структуре БД и по запросам к БД. Только они решают, прошли вы практику или нет. Про структуру практик — подробнее на практиках.

Практики будут проходить в командах по 3, максимум 4, участника. Остальные волей священного `/dev/random` будут объеденины в команды. У нас будут четыре Ajile-забега:

**1 неделя:** новый материал. В основном постоянное повествование, с какими-то простыми вставками заданий.

**2 неделя:** чтение кода. Надо будет читать много безобразия, найти ошибки. Будет немного соревновательной части.

**3 неделя:** показ решения задания. Оно даётся на две недели (10 дней, ещё дня 4 на показ оппонентам, чтобы успели прочитать). Будут оппоненты (другая команда).

На практиках будет использоваться `postgresql`. Кому не хватает мощностей или хочет на команду решение — можно попросить у преподавателя облако (доступ по `SSH` с `publickey`), ещё есть облачный `sqool.papeeria.com`, но там нет администраторского доступа.

На лекциях будут следующие темы:

1. Модели данных
2. Физическая реализация
3. Транзакции, оптимизация запросов.

Контакты: `dbms@barashev.net`, по поводу курса пишите именно сюда, и никуда более. Есть вебсайтик (может, даже, работает) `http://dbms.barashev.net`, там могут появляться материалы.

## 1.2. Введение

Всем очевидно, что надо хранить какие-то данные. Хранить *персистентно*, то есть чтобы данные сохранялись. Вот вы программист, вам надо хранить данные пользователя приложения. Как вы можете их хранить?

**Файлик:** Ваше приложение «Контакты» просто хранит на телефоне текстовый файлик `contacts.txt`, в котором на каждой строке записан один контакт — имя, фамилия, телефон,..

В будущем, предположим, у вас появляется вторая программа «Почта», которой хочется работать с теми же данными. Пока, чтение и там, и там, происходит чтение так:

```
class ContactsReader {
    //...
}
```

Вам скажут: копипаста кода, плохо!

**Библиотека:** Через сколько-то усилий появляется библиотека `read_contacts`, которая и работает с файликом.

И тут вы получаете race condition на чтение-запись в файл. Поплакав над убытками, вы добавляете блокировку на ваш файл.

Ваш бизнес растёт, вы уже обслуживаете IKEA, и ваши программы уже крутятся на многих компьютерах во многих экземплярах. Они все хотят читать-писать наш файлик, который расположен на сетевом диске, и даже если средства синхронизации работают, но из-за задержек, количества пользователей, невозможности заблокировать кусок файла, всё работает уже совсем так себе. Что делать?

**Сервис:** Теперь у нас уже отдельный сервис `ContactsManager`, управляющий данными, а остальные с ним общаются. Общение уже не на языке Java, кстати — по сети вызовы методов не идут.

И как только эта задача будет решеша, вы, кстати, и написали свою *систему управления базами данных (СУБД)* — програмный комплекс, предоставляющий другим сервисам доступ к базам данных. Часто ещё говорят, что программа подключается или обращается к базе данных, на самом деле имея в виду обращения к СУБД. Сама же база данных у нас — ровно те файлики, в которых в каком-то формате хранят данные. Но теперь к вам приходят и предлагают создать аналогичную базу данных для продажи авиабилетов. Но наша СУБД завязана внутри на мебель! В каждом файлике же хранятся, без пометок, только значения:

| | |
|---|---|
| 1 | 20 |
| 2 | 130 |
| 3 | 200 |
| 5 | 32 |
| ... | ... |

**Настоящая СУБД:** Когда вы обобщите сервис от конкретной структуры данных, а сами базы данных внутри также будут описывать, что они хранят, вы получите настоящую, мощную базу данных.

**Def 1.2.1.** База данных — струтурированное, персистентное (сохраняющее данные между запусками), самодокументированное (описывающее свою структуру), интегрированное (позволяющее связывать состояния многих блоков данных) хранилище данных.

Строгого общего определения нет, у всех что-то подобное этому.

Как видно, наш текстовый файлик очень слабо подходит под это, нужно на него достаточно всего наворачивать.

Какие же функции у нашей СУБД?

1. Физически хранит данные.

2. Язык запросов.
3. Контроль доступа.
4. Конкурентный доступ.
5. Предоставление гарантий на случай сбоя.
6. Поддержка целостности и согласованности данных.

Современные бызы данных появились в 1970-х годах, благодаря идеям IBM. К 1990 уже трудно представить предприятие без БД от Microsoft, Sybase, Sun или кто ещё тогда был.

Сейчас, из самых популярных проприеритарных:

- Oracle
- IBM DB2
- Microsoft SQL Server

а из открытых

- MySQL / MariaDB
- PostgreSQL

## 1.3. Архитектура информационных систем

### 1.3.1. Встроенные БД

Работает прямо в приложении; как правильно, один пользователь; нет конкурентного доступа и контроля доступа (зачем?).

### 1.3.2. Классическая клиент-серверная архитектура

Стала крайне распространена в 1990-х, довольно широко используется и сейчас. Есть отдельный процесс СУБД (чаще всего, даже на отдельном хосте); есть набор клиентских приложений с интерфейсом и бизнес-логикой, обращающихся к СУБД для получения и сохранения данных.

Проблема — если надо обновить бизнес-логику, надо обновить код *на всех клиентах*, тысячах их. Одновременно. Удачи!

### 1.3.3. Многоуровневая архитектура

В современности делают так — клиентское приложение общается с сервером приложений, находящаяся рядом с базой данных. Сервер получает запросы (например, по HTTP), преобразует их по акутальной бизнес-логике в настоящие запросы к БД (если вообще не откидывает, не кеширует, чёрт его знает). В клиентской части остаётся только интерфейс. В современных СУБД есть несколько уровней абстракции, позволяющих оптимизировать работу. Оптимизировать мы можем не только непосредственно работу с данными, но и программиста. Внесение изменений могут быть вызваны:

1. Изменением требований;
2. Изменение «железа»;

3. Изменение нагрузки.

4. ...

Мы, как в прошлый раз разбирались, не хотели, чтобы, если пришлось менять механизм хранения, пришлось менять все клиентские приложения. Традиционно, для решения проблемы используется *трёхуровневая архитектура*:

**Внешний уровень (уровень представления):** уровень общения конечного приложения с СУБД. Для различных клиентов может быть разным (бугхалтеру не нужна информация о складе, начальнику склада — расчёт зарплат). Тут могут быть, например, протоколы вида HTTP, или чистые SQL-запросы.

**Концептуальный (логический) уровень:** логическая модель данных. Описывает, что хранится:

```
class Employee {
    int Salary;
    //...
}

class Warehouse {
    Employee boss;
    //...
}
//...
```

Надо понимать, что крайне редко модель данных неизменна.

**Физический уровень:** Сами данные могут храниться много где. Могут на одном диске, могут на сетевом кластере. Какие-то поля (фотография работника) могут эффективнее храниться отдельно от остальных данных. Естественно, мы не хотим, чтобы изменения на этом уровне вылезали выше.

Между соседними слоями существуют отображения. Отображения между внешним и концептуальным поддерживаются прикладными программистами, между концептуальным и физическим — администраторами и разработчиками СУБД.

# Глава 2

# Модели данных

## 2.1. Модель данных

**Def 2.1.1.** Модель данных — некоторый набор концепций, позволяющий описать структуру и поведение наших данных.

В разработке также под моделью данных могут понимать уже конкретную выстроенную реализацию (называя концепции метамоделью). Сами же данные могут называть экземпляром модели.

1. Структурный аспект (например, вершины дерева)
2. Ограничения (в дереве у вершины не более одного родителя)
3. Операции с данными (обход дерева, поиск элемента)

История моделей примерно такая: сначала была одна иерархическая модель (есть узел, можно спрашивать, что внутри). Позже возникла и стала крайне распространённая релационная модель (таблицы с отношениями между ними). В 1990х пришла объектная модель, в 2000х — слабоструктурированная. Кроме них, ещё есть модели вида «ключ–значение», смешанные виды.

Мы будем заниматься чистой реляционной моделью данных.

## 2.2. Реляционная модель

### 2.2.1. Структура

Гланвым структурным отношением в релационной модели является *отношение* (*relation*), а также *домéн* (*domain*).

**Def 2.2.1.** Домен — множество допустимых значений.

Их множество: булевский, число, строка, время года; какие захотите. У каждого домена есть свои правила операций и действий, в том числе сравнение, сложение, и так далее.

У отношения два определения. Они чем-то похожи, чем-то отличаются, важно понимать.

**Def 2.2.2.** Пусть есть некоторый список доменов $[D_1, D_2, ..., D_n]$. Рассмотрим функцию, переводящую элементы декартова произведения домнов в булевский тип:

$$f \colon D_1 \times D_2 \times ... \times D_n \to \{true, false\}$$

Такая функция и называется отображением.

Пока совсем не ясно, а где тут таблицы?..

**Def 2.2.3.** Отношение — пара из схемы и тела.

Схема отношения — множество атрибутов $\{A_i\}$, представляющих собой пары из имени атрибута, его индетифицирующего, и его домена. Атрибуты равны, если равны их имена.

Тело отношения — множество кортежей $\{t_i\}$, содержащий пары из названия атрибута и значения из соотвествующего домена. Каждый атрибут встречается ровно один раз. Кортежи равны, если для каждого атрибута его значения равны.

*Замечание* 2.2.1. В теории, можно разрешить дубликаты кортежей в теле.

Самым естественным видом такой конструкции является двумерная таблица. Столбцам соотвествуют атрибуты: в заголовке каждый столбец подписан его именем, указан его домен. Порядок атрибутов по определению не зафиксирован. Собственно, заголовок и является схемой. Тело представлено строками таблицы, в каждой строке в соотвествующем столбце записаны значения атрибутов. Порядок строк, опять-таки, не зафиксирован.

| $a_1 : D_1$ | $a_2 : D_2$ | $a_3 : D_3$ | $a_4 : D_4$ |
|---|---|---|---|
| 1 | 2 | ''foo'' | true |
| 2 | 4 | ''bar'' | false |
| 2 | 5 | ''baz'' | false |

В чём же связь? В первом определении можно сказать, что функция выдаёт значение, есть ли такой кортеж в теле; и наоборот, тело может задавать определение функции. Важное отличие: в первом определении фиксирован порядок атрибутов, но не заданы имена атрибутов (их просто нет). Мы будем пользоваться вторым определением, как более близкому к практической стороне.

### 2.2.2. Ограничения

Какие есть отношения ограничения в этой модели?

**Def 2.2.4.** Потенциальный ключ (candidate key) отношения $R$ — набор атрибутов из схемы отношения $R$ ($K \subset S_R$), обладающий свойствами:

**Уникальность:** Если взять два кортежа из отношения $t_i$ и $t_j$, если они равны по всем атрибутам ключа, то они равны целиком:

$$V_K(t_i) = V_K(t_j) \Rightarrow t_i = t_j$$

**Неизбыточность:** Любое собсвтенное подмножество атрибутов ключа не обладает свойством уникальности.

**Def 2.2.5.** Суперключ — надмножество потенциального ключа (и, следовательно, обладает уникальности).

Понятно, что потенциальных ключей может быть несколько. Традиционно, один из них выбирается *первичным ключом* (*primary key*). Ничем от других, кроме выбора, он не отличается. Это ограничение — одно из самых важных. По первичному ключу можно находить и индетифицировать записи.

### 2.2.3. Операции

К ключам мы ещё вернёмся, давайте теперь об операциях с данными. Есть несколько описывающих теорий, мы займёмся реляционной алгеброй. Те операции, что мы рассмотрим, так или иначе реализованы в современных системах.

Операции:

$$\cup \quad \cap \quad \setminus \quad \times \quad \bowtie \quad \sigma \quad \pi \quad \rightarrow$$

### Проекция $\pi_A$

Проекция отображения $R$ по подмножеству атрибутов $A$ — это отношение $\pi_A(R)$ на атрибутах $A$, что

$$t \in \pi_A(R) \Leftrightarrow \exists t' \in R \colon V_A(t') = V_A(t)$$

Короче говоря, отобрали только нужное подмножество атрибутов. В SQL это соотвествует перечислению атрибутов после слова `SELECT`.

### Селекция $\sigma_\theta$

Селекция отображения $R$ по предикату $\theta$ — отношение, в котором из тела остались только те кортежи, которые принял предикат.

$$\sigma_\theta = \{t \in R | \theta(t)\}$$

В SQL это соотвествует `WHERE`.

### Базовые операции над множествами

Операции объединения, пересечения и разности отношений делают соотвествующие операции на телах отношений, но есть соотвестующее требование — схемы отношений должны быть равны (как по именам атрибутов, так и по их доменов!).

На практике ограничения слабже, за счёт неявной следующей операции:

### Переименование $\xrightarrow{A \to A'}$

Операция переименования $R \xrightarrow{A \to A'} R'$ позволяет переименовать одно подмножество атрибутов, сменяя их имена, но не меняя домены.

### Декартого произведение $\times$

Декартого произведение чем-то похоже на обычное на множествах. Мы требуем, чтобы у отношений не было пересекающихся атрибутов. Тогда результатом будет отношение, схема которого — объединение отношений, а кортежи — склееные пары кортежей обоих отношений.

$$Scheme(R \times S) = Scheme(R) \sqcup Scheme(S)$$
$$t \in R \times S \Leftrightarrow \exists r \in R, s \in S \colon V_R(r) = V_R(t) \wedge V_S(s) = V_S(t)$$

На практике можно, чтобы атрибуты пересекались, их просто переименовывают. В SQL это написать через запятую несколько таблиц в `FROM`, например

```
SELECT * FROM R, S;
```

### Соединение $\bowtie$

Пусть есть два отношения на пересекающихся атрибутах: $R(A, B)$ и $S(B, C)$. Далее, соединение — это отношение на объединении атрибутов, а в тело войдут только те пары кортежей, у которых на пересекающимся множестве атрибутов значения равны.

$$Scheme(R \bowtie S) = Scheme(R) \cup Scheme(S)$$
$$t \in R \bowtie S \Leftrightarrow \exists r \in R, s \in S \colon V_R(r) = V_R(t) \wedge V_S(s) = V_S(t) \wedge V_B(r) = V_B(S)$$

В SQL это `NATURAL JOIN`. На практике накладывается $\theta$-соединение, в котором последнее условие на равенство на полях $B$ заменяется на предикат. По сути, мы построили выборку по декартову произведению $\sigma_\theta(R \times S)$, только не раздваивали атрибуты. Например, вот:

```sql
-- неявно
SELECT *
    FROM R, S
    WHERE R.id = S.id;
-- явно
SELECT *
    FROM R
    JOIN S ON R.id = S.id;
```

Тут в результат уйдут два атрибута равных значений из-за переименования.

Эта операция крайне распространённая на практике, вряд ли можно изучать базы данных дольше, чем два дня, и не написать это.

### 2.2.4. Внешние ключи

Пусть у нас есть таблица студентов (номер, ФИО), и таблица стипендий (номер студента, месяц, деньги).

| id | name |
|---|---|
| 1 | John |
| 9 | Jack |
| 2 | James |

| id | month | money |
|---|---|---|
| 1 | $1000 | Sep |
| 2 | $1000 | Sep |
| 8 | $8000 | Sep |

Как видно, $8000 достались никому. А как так-то? Можно проверять, что выражение истинно:

$$\pi_{StudentId}(Scholarship) \subset \pi_{StudentId}(Student)$$

**Def 2.2.6.** Внешним ключом (foreign key) в отношении $R$ на отношение $S$, имеющее первичный ключ $K$, называется множество атрибутов $R$, совпадающих по имени и домену с ключом $S$, что если в $R$ эти атрибуты принимают такие значения, то есть кортеж в $S$ с таким ключом:

$$\pi_K(S) \supset \pi_{FK}(R)$$

СУБД, соотвесвтенно, может предоставлять гарантию, что заданное ограничение выполняется. Программисту требуется задать ключ и внешний ключ. Для гарантии СУБД может делать следующее:

- При добавлении строчки в $R$ с неверным значением внешнего ключа отвергнуть его.
- При пропаже или изменении соотвествующей строчки в $S$ можно удалить/обновить всю зависимую информацию (каскадно), переписать значение в особое/сбросить в `NULL` или отвергнуть изменение (по умолчанию).

До недавнего времени MySQL имел ещё один вариант действия: забить. На практике совпадений имён не сильно требуют, нужно просто пары указывать или неявное переименование.

### 2.2.5. Разминка 29.09.2016

Задача: пусть есть отношения $R(a, b, c)$, $S(b, d)$, $T(c, e)$. Они имеют размеры $|R|$, $|S|$, $|T|$. Также мы знаем, что $S.b$ — это внешний ключ для $R.b$, а $T.c$ — это внешний ключ для $R.c$. Мы хотим узнать размер отношения $|(R \bowtie S) \bowtie T|$.

Ответ: что-то не большее $\min(|S|, |T|)$.

### 2.2.6. Функциональные зависимости

**Def 2.2.7.** Пусть есть отношение $R(A, B, C)$. Тогда мы говорим, что $B$ (*зависимая часть*) textit-функционально зависит от $A$ (*детерминант*), если для любого экземпляра $R$ (любого набора кортежей) верно следующее:

$$\forall t, t' \in R\colon V_A(t) = V_A(t') \Rightarrow V_B(t) = V_B(t')$$

То есть если две строки имеют одинаковые значения атрибутов $A$, то они гарантировано имеют одинаковые значения атрибутов $B$.

*Замечание* 2.2.2. Обозначается $A \to B$; *$A$ функционально определяет $B$.*

*Замечание* 2.2.3. Функциональная зависимость — это ограничение на уровне схемы, а не свойство конкретного экземпляра.

*Пример* 2.2.1. Пусть есть такой экземпляр $R$:

| $A$ | $B$ | $C$ |
|---|---|---|
| 2 | 4 | 5 |
| 2 | 4 | 6 |
| 3 | 8 | 1 |
| 3 | 8 | 2 |

Тут мы не можем сказать, есть ли функциональная зависимость между $A$ и $B$, потому что мы не знаем ничего про остальные возможные экземпляры $R$, нам дали только один.

*Пример* 2.2.2. Ключ всегда является детерминантом для всех остальных атрибутов.

*Пример* 2.2.3. Пусть есть таблица со следующими атрибутами, описывающая космические перелёты: планета, политический строй, дата, корабль, капитан, рейтинг капитана.

Тут есть ключ (планета, дата), если мы считаем, что в день не больше одного рейса на планету.

Помимо ключа, есть ещё два детерминанта:

1. Планета функционально определяет политический строй на планете.
2. Капитан функционально определяет его рейтинг.

Видим, что храним много избыточной информации.

Давайте теперь предположим, что у нас нет ключа (планета, дата), но зато мы знаем, что от (планета, дата) есть функциональная зависимость к капитану. А от капитана — к кораблю.

А теперь видим, что у нас возникает некоторое количество неявных функциональных зависимостей: по транзитивности и взятию подмножество можно сделать вывод, что (планета, дата) функционально определяет всё остальное. Тогда полный набор зависимостей называется *замыканием*. Соответствующие формальные правила называются аксиомами Армстронга:

1. Самоопределение: $A \to A$.
2. Рефлексивность: $AB \to A$.
3. Транзитивность: если $A \to B$ и $B \to C$, то $A \to C$.
4. Дополнение: если $A \to B$, и $A \to C$, то $A \to BC$.
5. Если $A \to BC$, то $A \to B$ и $A \to C$.
6. Если $A \to B$ и $C \to D$, то $AC \to BD$.

Если бы какая-нибудь БД желала бы проверять функциональные зависимостей, то ей бы эти правила очень помогли для вывода зависимостей. Но этим никто не занимается, кроме проверки уникальности ключа.

**Def 2.2.8.** Два множества зависимостей $S$ и $T$ называются эквивалентными ($S \sim T$), если совпадают множества их замыканий: $S^+ = T^+$.

В целом СУБД могла бы найти эквивалентное множество зависимостей, но этого никто не делает. Считается, что это забота программиста.

### 2.2.7. Нормальные формы

Давайте вспомним наш пример с космическими полётами. Там было много функциональных зависимостей и возникали следующие практические проблемы:

1. Если у капитана поменялся рейтинг, то его надо обновить во всех строчках. Мы считаем, что нас прошлый рейтинг не интересует, иначе мы скатываемся в так называемые «темпоральные базы данных», которые являются отдельной банкой с червями.

2. Если была открыта новая планета, то мы не сможем сохранить о ней информацию, пока не полетит хотя бы один рейс. Например, потому что первичным ключом является (планета, дата).

3. Если увольняем капитана, то надо удалить все его полёты и, возможно, даже некоторые планеты.

Нормальная форма — это некоторый набор правил/условий. Каждая более высокая НФ содержит все правила более низких НФ.

**Def 2.2.9.** *I нормальная форма*: любое корректное отношение находится в I НФ.

*Замечание* 2.2.4. Тут важно то, что в каждом атрибуте каждого кортежа находится ровно одно значение.

*Замечание* 2.2.5. На практике довольно часто нарушается. Например, если вы в таблицу с космическими полётами добавите атрибут «имена пассажиров через запятую», то вы нарушите I нормальную форму. Формально нарушения нет (одна же строчка в атрибуте хранится), но идеологически нарушение есть.

На самом деле даже если БД поддерживает тип данных «массив», то она уже нас провоцирует нарушать первую нормальную форму подобным образом.

**Def 2.2.10.** Функциональная зависимость $A \to b$ *неприводима*, если из левой части ничего нельзя выкинуть.

**Def 2.2.11.** *II нормальная форма* для отношения с ровно одним потенциальным ключом (не суперключом): любой неключевой атрибут (который не является частью ключа) функционально неприводимо зависит от этого ключа.

*Пример* 2.2.4. Вспоминаем таблицу с полётами и ключом (планета, дата). Она не находится во II нормальной форме, так как атрибут «строй» зависит только от атрибута «планета», то есть функциональная зависимость не неприводима.

*Замечание* 2.2.6. Если в отношении ключ не составной, то II нормальная форма автоматически выполняется.

*Замечание* 2.2.7. Тут, опять же, можно формально хитрить, заведя суррогатный ключ (SERIAL и иже с ним, значения которых генерируюются автоматически). Но это неинтересно: стоит смотреть на «реальные» ключи.

Что же делать чтобы соблюдать 2 нормальную форму? Стандартный приём — декомпозиция. Декомпозиция — разложить отношнение на проекции, собирать обратно склейкой. Понятно, что при этом у проекций должен быть общие атрибуты, но кажется, это не полное требование. Мы можем разбить нашу сложную строчку про корабли, выделив планеты, но общим атрибутом не надо оставлять строй. Поэтому надо добавить требование, чтобы соединение проекций было равно исходному отношению. Такая декомпозиция называется безопасной.

Нужен критерий безопасности декомпозиции.

**Теорема 2.2.1** (Хита (Heath))**.** Если в отношении $R(a, b, c)$ есть функциональная зависимоть $a \to c$, то $R$ можно безопасно декомпозировать в проекции $(a, b)$ и $(a, c)$.

Например, у нас в таблице про корабли имеет смысл выделить проекцию Планета— Строй, оставив у рейса аттрибут Планета. Это кажется довольно очевидным, но тем не менее важно и полезно.

**Def 2.2.12.** *III нормальная форма* — любой неключевой атрибут неприводимо и **нетранзитивно** зависит от потенциального ключа.

Например, если вспомнить нашу таблицу с планетами: дата, планета, капитан, корабль, капитан, рейтинг. В прошлый раз, ради 2 нормальной формы мы вынесли строй на планете в справочник. Тут мы наблюдаем, что рейтинг капитана зависит в том числе от капитана, что нарушает третью нормальную форму. Починить есть два способа декомпозиции:

1. Дата, планета, корабль, капитан; капитан, рейтинг.

2. Дата, планета, рейтинг; дата, планета, корабль, капитан.

Второй способ хуже, мы теряем функциональную зависимость между капитаном и его рейтингом.

**Def 2.2.13.** *Нормальная форма Бойса-Кодда* — В любой нетривиальной зависимоти детерминантом является потенциальный ключ.

Это надмножество 3 нормальной формы. Рассмотрим отношение — дата, корабль, капитан. Пусть капитан зависит от даты и корабля (это очевидно), а капитан всегда управляет только одним кораблём (и корабль зависит от капитана). Итого, есть два потенциальных ключа: дата— корабль, и дата— капитан. Каждый атрибут — часть потенциального ключа, 3 нормальная форма выполнена, но — не нормальная форма Бойса-Кодда, так как в зависимости капитан→корабль капитан не является потенциальным ключом.

Как же решить такой пример? Это довольно нетривиально. При декомпозиции теряются какие-то зависимости. Вообще, не все отношения можно привести в эту нормальную форму.

Есть ещё нормальные формы, но они довольно домороченные. Если вы уже дойдёте до НФБК, у вас всё довольно хорошо.

# Глава 3

# Физическая реализация

А где вообще может храниться информация?

| | Объём | RandAccess | SeqAccess | Зависит | Цена |
|---|---|---|---|---|---|
| Регистры | <1 Кб | 0 | N/A | нет | $\infty$ |
| L1-L3 кеши | 10 Кб-N Мб | 1 нс 5 нс 10 нс | N/A | нет | N/A |
| Оперативная память | до 1ТБ | 100 нс | N-NN Гб/с | нет[1] | 100$ / N Гб |
| HDD | N Тб | N мс | NN Мб/с | да | 50$ / 1 Тб |
| SSD | NNN Гб | 0.1мс | 100Мб/с | да | 500$ / Тб |
| Магнитная лента | $\infty$ | N/A | быстро | да | ? |

Что такое жёсткий диск (Hard Disk Drive, HDD)? Это набор блинов на единой вращающейся оси. На этих блинах есть набор концентрических дорожек. Также есть набор головок, парящих над этими блинами, находящихся на единой оси, и способными двигаться между этими дорожками. Дорожки поделены на сектора (хотя правильно их называть сегментами). Чтобы прочитать иди записать данные, надо передвинуть головку на дорожку, и прочитать-записать целиком сектор, когда поверхность прокрутится и сектор окажется под головкой.

Что такое твердотельные накопители (Solid State Drive, SSD)? Это целиком электронная схема, нет движущихся частей, гораздо быстрее работает, время записи при этом в несколько раз больше чтения. Но: дорого, ограничена перезапись одной ячейки.

*Замечание* 3.0.8. Разминка 13 октября. Есть отношение: Капитан, Корабль, Дата, Планета, Расстояние. В один день и капитан, и корабль не могут совершать больше одного полёта. Расстояние до планеты зависит от даты. В одну дату до заданной планеты происходит

1. много
2. не более одного

рейса. В какой нормальной форме находится отношение?

Решаем. Капитан, Дата — ключ. Корабль, Дата — тоже ключ. Планета, Дата — определяют расстояние. Ещё, возможно, Планета с Датой является ключом.

Как видно, в варианте 2 в левой части во всех детерминантах стоят ключи, так что НФБК; а в варианте 1 расстояние зависит от неключа, нет НФБК. При этом расстояние — единственный неключевой аттрибут — зависит от всех ключей, но не нетранзитивно (например, Капитан, Дата — Планета; Планета, Дата — расстояние). Поэтому только вторая.

Как мы видели, доступ к оперативной памяти заметно дешевле диска. Явно хочется не работать с диском, а работать без оперативной памяти не получается. Поэтому мерой эффективности будем считать количество обращений к диску.

Хочется взять все данные, загрузить в оперативную памяти и жить. И всё неплохо, но её может не хватить. Понятно, что в какой-то момент хочется прочитанные данные обратно сбросить. Этим

занимается buffer manager (по-русски не будем называть): он обрабатываем запросы "Дай страницу с диска отвечает или из памяти, или из диска, при этом, возможно, убирая что-то на диск или выкидывая. Какие могут быть политики:

**Время последнего обращения (Least recently used, LRU):** Выкидываем страницу, которую просили раньше всех других. Нужно при этом хранить структуру данных, отслеживающую времена обращений.

**Очередь (FIFO):** Простая структура.

**Количество обращений (Least frequently used, LFU):** К кому меньше всего обращаются, того и тапки. Но, проблема: если к кому-то очень много раз обратились и забыли, то «тонуть» оно будет долго.

**Часы:** Пусть есть кольцевой буфер. У каждый странице есть бит — было ли обращение. Ещё есть стрелка. Каждый раз, когда надо вытеснить страницу — мы двигаем стрелку до первого нуля, при этом обнуляя счётик всем, кого посетили. Таким оразом, если за оборот стрелки страница не нужна — её выкидывают.

Можем расширить счётчик численными значениями, чтобы можно задавать вес. По-сути, комбинируем сюда предыдущий подход.

Окей, а что же со страницами? Мы читаем с диска какими-то блоками. Удобно их и брать. В страницах мы храним сколько-то кортежей, каждый кортеж обычно хранится как просто набор байт подряд. Внутри страницы хранятся подряд несколько кортежей, и вряд ли очень хочется хранить кортеж по частям в разных страницах.

Теперь ищем запись в странице. Если мы знаем, что они фиксированного размера, то просто по сдвигу. Можно хранить в начале страницы директорию, где какая запись.

Как правило, отношение на странице не поместится, нужно по многим страницам. Можно устроить связный список страниц, для каждого отношения хранить первую страницу. Нужны указатели страниц на диске. Можем нумеровать блоки, а как мы помним, у диска есть поверхности, на них дорожки (цилиндры), на них сектора. Зачем нам нужна вся эта информация? Мы бы хотели, чтобы страницы шли подряд и быстро читались, для этого их хочется двигать. При этом рушатся физические адреса.

Хотим логические адреса на диске. С пейджингом и скоростью доступа. Это реализует файловая система, ОС или даже контроллер диска, хотя некоторые СУБД работают с диском напрямую, решая этот вопрос сами. Рассмотрим алгоритм Multiway Mergesort. Его используют очень многие СУБД. Пусть у нас есть $M$ блоков в буфере. Через них мы будем сортировать $N$ блоков на диске.

1. Читаем группами по $M$ блоков, сортируем их в RAM, сохраняем на диск. Получили $\lceil N/M \rceil$ отсортированных отрезков на диске.

2. Хотим сливать блоки. Берём, и из всех отсортированных отрезков загружаем первый блок, оставив один блок на вывод (если поместятся, иначе ниже). Выбираем минимум, его выталкиваем в блок вывода, переходим к следующему элементу в соотвествующем блоке. Если в блоке кончились элементы — догружаем следующий блок из его списка. Если в выводе место кончилось — на диск его!

Всего операций I/O: каждая запись пишется и читается по 2 раза, то есть $4N/M$. Работает, оно, конечно, когда $N \leqslant M^2$. Например, если у нас 16Гб RAM, то мы можем сортировать до $2^60$ байт.

Если не хватает — то во много слоёв. Обращений к диску линейно зависит от количества слоёв.

Можно, кстати, вывод делать не на диск, а на дальнейшую обработку по конвееру. Например, при обработке запроса

```sql
SELECT name, COUNT(*)
    FROM T
    GROUP BY name;
```

можно отсортировать по имени, и дальше передавать это в оконную функцию.

```sql
SELECT *
    FROM T
    WHERE Year = 1990;
```

Как можно это выполнить? Ну не знаю.

**Full Scan:** Читать всёёёёёё!

**Index:** Если есть уверенность, что данные в целом разные, то можно по какой-то структуре понять, где лежат все записи с таким значением, и это быстрее полного чтения.

**Отсортированность:** Если предыдущий элемент конвеера дал отсортированные по году данные, быстро пропускать данные до, дождаться конца нужных и убить оставшиеся. Или можно воспользоваться тем, что данные целиком отсортированы, и сделать двоичный поиск.

Что посложнее:

```sql
SELECT *
    FROM T
    NATURAL JOIN T2;
```

Тут помогут **Nested loops**. У нас есть массивы, хотим по ним построить пары. Как можно это сделать:

```
for i in a:
    for j in b:
        if i.id = j.id:
            output(i, j)
```

Куда уж проще?

Пусть у нас есть мало (можно уместить в оперативную память) записей одного типа и много других. Тех, которых мало, можно загрузить в память и держать там. Потом из большого отношения читаем все записи (поблочно, нам больше одной записи за раз не нужно), выписываем им все парные, идём дальше. Чтений получается суммарный объём данных.

А если оба не помезаются? Можно одно отношение читать большим окном, и с этим окном пробежаться по другому. Если читаем окном отношение $T$, а пробегаемся по $S$, то получим $B(T) + \lceil \frac{B(T)}{M} \rceil B(S)$ чтений.

Ещё есть всякие **SORT JOIN**, **HASH JOIN**.

### 3.0.8. Разминка 03.11.2016

Пусть есть буффер в памяти из $M = 100$ блоков, а также есть отношение $R$, состоящее из $B(R) = 10\,090$ блоков. Вопрос: за сколько операций ввода-вывода мы сможем его отсортировать?

Решение: если $B(R) \leqslant M^2$, то мы могли бы отсортировать за $3B(R)$ чтений и ещё $B(R)$ записей, итого $4B(R)$. Но у нас на 90 блоков больше, есть несколько вариантов:

- Можно сначала отсортировать первые $10\,000$ блоков, потом отсортировать хвост за 180 операций, потом слить два результата за $10\,000 + 90$ операций, получаем чуть больше $5 \cdot 10^4$ операций ввода-вывода.

- Давайте посмотрим повнимательнее, что происходит в алгоритме сортировки 10 000 блоков. А конкретнее — на ситуацию, когда у нас уже есть 100 списков из 100 отсортированных блоков в каждом (перед сливанием). В этот момент мы сделали 20 000 операций.

  Давайте сольём 91 список (из ста) и запишем на диск. Потом давайте прочитаем хвост из 90 блоков в память, отсортируем. Осталось 10 слотов в памяти. Из них 9 слотов отведём под головы оставшихся неслитых девяти списков, а один слот — под голову списка-результата.

  Получим порядка 48 000 операций.

- Третий подход: сначала будем сливать не 91 список, а 50. Потом сольём 90 блоков из хвоста. Итого получили 52 списка, которые уже можно слить за один подход по диску.

  Результат — порядка $40,270$ операций.

- А если мы сольём не 50 списков, а 10 (хвост тоже сольём), то получим 92 списка, их тоже можно слить за один подход.

  Получим порядка 32 270 операций.

- А если сольём ещё меньше списков, то получится 30 670.

# 3.1. Операции соединения

Пусть есть отношения $R$ и $S$, которые мы хотим соединить. У нас есть алгоритм соединения nested-loop-join с его стоимостью работы $B(R) + \frac{B(R) \cdot B(S)}{M}$ (в I/O операциях).

## 3.1.1. Sort join

Давайте попробуем улучшить результат при помощи так называемого sort join. Сначала предположим, что памяти у нас довольно много ($B(R), B(S) \leqslant M^2$). Тогда отсортируем $R$ и $S$ по общему атрибуту за $4(B(R) + B(S))$, и будем их объединять методом двух указателей за ещё $B(R) + B(S)$, на это потратим ещё $B(R) + B(S)$ чтений (записывать результат обратно на диск не будем). Итого $5B(R) + B(S)$, если памяти хватает.

Но есть проблема: у нас соединение может каждому элементу сопоставлять несколько. Например, если общий аттрибут в $R$ везде равен единице, то нам нужно будет вывести для каждого элемента $S$ вывести все $R$. Если $R$ целиком поместился в буфер, то проблем нет, а иначе нам придётся читать с диска старые блоки $R$.

Более строго: если у нас для любого значения общего аттрибута либо все такие строки из $R$ помещаются в память, либо из $S$, то проблем не возникает — читаем то, чего меньше, а потом выводим результат (конечно, надо угадать, кто поместится в память; либо попробовать прочитать каждое и честно посчитать, либо воспользоваться статистикой и угадать). А вот если и там, и там строк много, то у нас получится большое декартово произведение. И придётся сделать вложенный цикл с чтением блоков.

## 3.1.2. Улучшение sort join

Пока что sort join проигрывает nested-loop-join на небольших отношениях. Например, если $M = 100$, $B(R) = 1000$, $B(S) = 500$, то NLJ отработает за:

$$B(R) + \frac{B(R) \cdot B(S)}{M} = 1000 + \frac{1000 \cdot 500}{100} = 5500$$

А вот sort join будет читать с диска намного больше:

$$5(B(R) + B(S)) = 5 \cdot (1000 + 500) = 7500$$

Что делать? Можно, конечно, просто применять NLJ для небольших отношений. Но можно и соптимизировать sort join. Давайте посмотрим на сортировку в тот момент, когда она уже собрала списки, но ещё не начала их сливать вместе. В отношении $R$ она получила 10 списков, в отношении $S$ — 5 списков. Можно загрузить в память одновременно головы всех списков и выполнять merge списков одновременно с соединением, тогда нам не надо записывать на диск промежуточные результаты. Итого получим на каждый блок из каждого отношения две операции для получения списков (чтение+запись), и ещё одну операцию чтения после получения списков:

$$3(B(R) + B(S)) = 3 \cdot (1000 + 500) = 4500$$

Мы выкинули промежуточную запись-чтение между вторым этапом multiway merge sort и соединением отношений. Это всё работает только в том случае, если $B(R) + B(S) \leqslant M^2$ и если нам не надо возвращаться читать уже выкинутые блоки при помощи вложенных циклов. Это покрывает большое количество случаев; обычно соединения идут по ключу и внешнему ключу, а со стороны ключа каждая строчка уникальна, поэтому во вложенные циклы с повторным чтением блоков мы не свалимся.

### 3.1.3. Hash join

Основная идея: сгруппируем кортежи с одинаковым хэшом по корзинам хэш-таблицы, обработаем каждую корзину отдельно.

1. Выбираем количество корзин $K$, заведём столько корзин на диске. Каждая корзина — сколько-то блоков, которые мы будем заполнять кортежами.
2. Для каждого кортежа $r$ из $R$ вычисляем номер корзины и кладём его туда:
$$i = hash(V_a(r)) \bmod K$$
3. Аналогично для всех кортежей из $S$.
4. Теперь внутри каждой корзины лежит сколько-то кортежей из $R$ и сколько-то кортежей из $S$.
5. Теперь обработаем каждую корзину:
   - Если корзина маленькая, то мы можем целиком прочитать её в память.
   - Иначе можно написать NLJ. Если хотя бы одно отношение из корзины целиком поместится в память, то чтений всё ещё будет линейно.
   - А если в корзине каждого отношения больше памяти, то всё плохо.

Обычно корзины получаются маленькие. Итоговое количество чтений:

$$\underbrace{2(B(R) + B(S))}_{\textit{хэширование}} + \underbrace{(B(R) + B(S))}_{\textit{обработкакорзин}}$$

Тонкости: на первом этапе (хэширование) нам требуется, чтобы мы могли хранить последний (недозаполненный) блок от каждой корзины. Если $K > M$, то нам в худшем случае при добавлении кортежа в корзину придётся лезть на диск, и мы получим вообще $|R|$ операций ввода-вывода вместо $B(R)$, что очень плохо. Значит, точно требуется $K \leqslant M$.

А также надо, чтобы после хэширования размер каждой корзины получился не более $M$. Так как в сумме блоков в корзинах получается $B(R) + B(S)$, то мы получаем следующее ограничение применимости:

$$B(R) + B(S) \leqslant M^2$$

Тут, конечно, не учитывается то, что кортежи могут распределяться между корзинами неравномерно.

**Def 3.1.1.** Индекс — персистентная избыточная структура данных для ускорения некоторых операций. Обычно создаётся для одного атрибута, и позволяет быстро находить записи с заданными значениями атрибутами.

Например, мы храним в кортежи отсортированно по аттрибуту, и помним, где какой. Так мы при поиске по имени можем экономить чтения с диска, так как читаем только имена и позиции записей.

У одного отношения может быть много индексов по разным атрибутам. Индексы бывают разные. Например, разреженный индекс (Sparce index) содержит указания не на все аттрибуты. Плотный индекс содержит все записи. Иногда даже полезно создать индекс для индекса.

Как для атрибута хранить его записи? Если все кортежи для заданного значения аттрибута лежат плотно, то есть на диске кучно, почти заполняя блок, то индекс называется кластеризующим. Если же котрежи лежат неплотно, то индекс — некластеризующий.

Плюс: операции поиска становятся шустрее. Минус: индекс надо поддерживать в корректном состоянии, ещё и на диске место занимает. При больших обновлениях их легче перестраивать с нуля, очень дорого.

Есть ещё B-деревья. Это дерево поиска, обычно глубины 2-3. Она уже построенна на дисковых блоках. В нижнем уровне хранятся упорядоченные значения. В промежуточных хранятся какие-то значения ($n$), указатели на нижние уровни слева и справа от него (всего $n + 1$).

Как ищем? Встали в корень. Выбрали диапазон, в который попало значение. Перешли на следующий уровень, там то же самое. И так далее до нижнего уровня, где просто идём слева направо до нашего значения.

Цель B-дерева состоит в том, чтобы глубина дерева была постоянной. При модификации надо поддерживать это. Пусть мы хотим добавить значение. В лучшем случае в подходящем листе есть место. Иначе — расщипляем лист на два, и добавим диапазон в родителе. Там тоже может расщипиться (только так, чтобы слева и справа в поддеревьях оказалось поровну). Хотим хеш-таблицу для индекса. Все помнят, что такое хеш-таблица?

Например, мы хешируем значение, которое хотим сохранить, смотрим в соотвествующую ячейку и смотрим список значений в ней. При хорошей хеш-фукнции (т.е. наиболее близкой к случайной) и правильном размере хеш-таблицы доступ будет за амортизированную O(1). Когда элементов становится многовато (то есть какой-то список стал великоват), надо перестроить хеш-таблицу на размере побольше (в 2 раза).

На диске это живёт так: есть блок(и) с указателями на списки, сами списки хранятся на блоках, и больше, чем один блок, мы их не позволяем. Когда размер кончится, рехеширование будет долго жужжать диском.

Применим Extensible Hashing. Пусть размер хеш-таблицы всегда будет только степень двойки $2^k$. Каждой корзине сопоставим $b_i$ — *количество значищих бит* для этой корзины. Что это такое — чуть дальше. Возьмём и нестандартную хешфукнцию — которая берёт первые $k$ бит от хеша.

Поиск: взяли хеш, смотрим, куда, радуемся. Изменение размера: пусть какая-то корзина стала большая. Увеличили размер хеш-таблицы в два раза. Большую корзину расщепляем на две: так как из хеша мы теперь берём на бит больше, значения расползутся, увеличиваем у них число значищих бит $b_i$ на 1. Для всех старых блоков мы записываем указатель на них дважды, для каждого префикса, но у них количество значищих бит осталось прежним.

Теперь, если какой-то ещё блок станет слишком большим, то мы увидим, что у него $b_i \neq k$, и это означает, что не надо переделывать саму хештаблицу, надо просто разбить блок на два и переставить указатели.

По-сути, мы теперь каждый блок разрезаем лениво, помня про него, насколько много бит хеша он использует.

У хеш-таблички есть проблемка: искать значение оно умеет, а вот

```
SELECT * FROM T WHERE id < 1000;
```

— до свидания. Зато `JOIN` очень весело идёт. Взяли каждую строчку, нашли в хеш-табличке парные, радуемся.

$$B(R) + T(R) \cdot ?$$

Сколько же записей сопосатвится каждой? Если индекс по $S.b$ кластеризующий, то надо прочитать

$$? = B(S)/V(S,b)$$

$V$ — как там много значений такого аттрибута. Если некластеризующий, то куда больше, совсем грустно.

$$? = T(S)/V(S,b)$$

В целом, алгоритм выглядит так себе эффективным, если объединять всё со всем, но если на какую-то таблицу есть сильные ограничения или сделана фильтрация, то всё будет куда лучше.

Ещё есть Zig-Zag Join, который не читает строчки таблицы, которые нужны, например, если из объединения есть только индекс.

## 3.2. Оптимизация

Итак, у нас есть диск с данными, есть менеджер буферов, есть обработчик запросов. С другой стороны, есть парсер. Осталось разорбать тот компонент, который для данного запроса построит отпимальный план исполнения.

Как только мы пропарсим, мы получим какой-то план, в лоб. Наша задача — построить логический план, который эквивалентен исходному и оптимален.

Нам нужны свойства операций. У нас все операции коммутативны. Есть дистрибутивность:

$$(R \cup S) \cup T = R \cup (S \cup T)$$

если имеет смысл, то

$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

Далее, есть ещё такая дистрибутивность:

$$\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie \sigma_C(S)$$
$$\sigma_{C_R}(R \bowtie S) = \sigma_{C_R}(R) \bowtie S$$
$$\sigma_{C_R \wedge C_S}(R \bowtie S) = \sigma_{C_R}(R) \bowtie \sigma_{C_S}(S)$$
$$\sigma_{C_1 \vee C_2}(R) = \sigma_{C_1} \cup \sigma_{C_2}$$

Очень часто предикаты проталкиваются вниз, поэтому не надо в запросах писать скобки вокруг соединений с предикатами внутри. Наверх их тоже иногда поднимают.

Оптимизатор может делать ещё всякие весёлые вещи, заменяя бинарные операции на большей арности, например, дерево соединений заменить на одно, и пусть оптимизатор потом решает, в каком порядке их уже соединять.

Вообще, полезно оценивать стоимость операций. Оно основывается на времени чтении отношений, индексов, сортировок, буферов конвееров. От плана не меняется результат, а вот число чтений

отношений и размеры промежуточных отношений зависят. Промежуточные отношения имеют следующие свойства: они не хранятся персистентно, всегда можно сделать кластеризованным, всегда можно сортировать (если хочется), нет индексов.

Тут уже полезно перейти от количества операций чтения к размерам промежуточных отношений. Понятно, что все факторы учесть невозможно, и нас не интересуют точные значения; нам только планы сравнивать. Самое дорогое при выполнении — запись промежуточных записей, поэтому примерные размеры промежуточных данных нам вполне подходят.

Как же их считать? Ну... чёрная магия и эмпирические наблюдения. Размеры результатов операций:

$\sigma_{\alpha=\$1}(R)$ Как мы знаем, размер результата равен

$$\frac{T(R)}{V(R,\alpha)}$$

если значения распределены равномерно. В реальности, есть распределение Зипфа, которое говорит, что если упорядочить значения атрибута по колисеству записей, то эти числа образуют последовательность $1/n$.

Сейчас мы будем строить всякие модели для подсчёта размера соединений. Это очень важно. Нужно, чтобы

- Считалось быстро.
- Результаты были похожи на истину
- Были верные в том числе в граничных случаях
- Не зависили от способа и порядка выполнения

Предположения:

**Вложенность множеств значений атрибутов:** если есть отношения $R(a,b)$ и $S(a,c)$, и есть ординальность $V(R,a) < V(S,a)$, то $\sigma_a(R) \subset \sigma_a(S)$. Применим:

$$|R \bowtie S| = \frac{T(R)T(S)}{\max\{V(R,a),V(S,a)\}}$$

Если мы строим декартого произведение ($V = 1$), то работает. Если мы соединяем по внешнему ключу, то тоже работает.

**Сохранения множества значений атрибутов:** $V(S,a) = V(R \bowtie S, a)$. Оно работает, если каждому значению есть пара, иначе чуть грустнее. Попрменяем:

$$|T(c,d) \bowtie (R \bowtie S))| = \frac{T(T) \cdot \frac{T(R)T(S)}{\max\{V(R,a),V(S,a)\}}}{\max\{V(T,c),V(R \bowtie S,c)\}} = \frac{T(T)T(S)T(R)}{\max\{V(R,a),V(S,a)\}\max\{V(T,c),V(R \bowtie S,c)\}}$$

Вообще, при соединениях количество видов планов растёт очень быстро. Ведь соединения образуют деревья, и вариантов деревьев с заданным числом листьев растёт очень и очень быстро[2].

Давайте думать про сложность соединений. Пока мы соединяем два, у нас нет промежуточных отношений, и сложность есть 0. Если уже три, то понятно, что стоит взять минимальную пару в качестве промежуточного результата. Если уже хотя бы 4...

[2] Числа Каталана

| | R(a, b) | S(b, c) | T(c, d) | U(a, d) |
|---|---|---|---|---|
| T() | 1k | 2k | 5k | 10k |
| V(b) | 200 | 2k | | |
| V(c) | | 500 | 1k | |
| V(d) | | | 2k | 5k |
| V(a) | 1k | | | 100 |

Пробовать будем только леворекурсивные деревья, потому что алгоритмам обычно нужно какую-то определённую сторону держать в памяти, а держать кучу временных результатов дорого.

| | RS | RT | RU | ST | SU | TU |
|---|---|---|---|---|---|---|
| Size | 1k | 5M | 10k | 10k | 20M | 10k |
| Cost | | | | 0 | | |

| | (RS)U | R(TU) | (ST)U | (RS)T |
|---|---|---|---|---|
| Size | 10k | 10k | 20k | 5k |
| Cost | 1k | 10k | 10k | 1k |

Вы знаете динамическое программирование, мы им тут пользуемся. Выбираем, что добавляем к результату, и выбираем оптимальный. Ответ, кстати, оказался стоимостью в 6k.

Такой подход всё равно требует какой-то неприличный размер расчётов. Можно брать жадный алгоритм: каждый раз присоединям чтобы результат был поменьше. Он, конечно, смотрит только на малое число вариантов, и иногда, кажется, может лажать, но хорош.

# Глава 4

# Транзакции. Конкурентный доступ

Откуда вообще параллельный досутп? Многопроцессорность, минимизация ожидание начала операции, ожидание диска. Как же с ней жить?

Вспомним, что ещё есть менеджер буферов: страницы, которые прочитали с диска или недавно изменили, хранятся в памяти. Если мы храним каждую страницу не более чем в одном экземпляре, то одна транзакция может увидеть „грязные“ промежуточные данные другой транзакции, и они отработают не так, как хотелось бы. В идеале, конечно же, мы хотим, чтобы транзакции имели такой же эффект, как если бы они выполнялись последовательно.

С точки зрения программиста, транзакция — атомарная операция, переводящая БД из одного согласованного состояния (выполнены все ограничения) в другое согласованное. Эта атомарная операция состоит из более чем одного запроса, и в конце БД или принимает результат этих запросов, или откатывает целиком.

Какие бывают свойства у транзакций? ACID:

**Atomic, Атомарность:** Программист выставляет границы транзакции, СУБД гаранирует атомарность исполнения.

**Compatability, Согласованность:** Программист за транзакцию не нарушает согласованность, СУБД гарантирует сохранение согласованности.

**Isolation, Изолированность:** Это ниже.

**Durability, Долговечность:** СУБД гарантирует, что результаты транзакции, помеченных как выполненые, уже персистентно хранятся.

Изолированность даёт политики, по которым транзакция работает, как будто она одна; мы можем управлять тем, когда результаты транзакций становятся доступны другим.

Рассмотрим расписание выполнения транзакций (schedule). Доступ к диску должен быть как-то упорядочен. Упрощённо, транзакции читают страницы, пишут страницы и выполняют какую-то арифметику. Расписанием для транзакций будет упорядоченная последовательность действий, которые выполнется в БД, что каждая транзакция имеет естественный порядок своих действий, а между транзакций — ничего не знаем.

Формализуем операции:

$R_i(x)$**:** чтение данных $x$ в память транзакции.

$W_i(x)$**:** запись данных $x$ из памяти транзакции.

$A_i(x)$**:** манипулирование данных в памяти транзакции.

Можно составить сериализуемое расписание, эквивалентное некоторому последовательному исполнению транзакций. Пусть есть расписание:

$$
\begin{array}{ll}
R_1(x) & \\
A_1(x) & \\
W_1(x) & \\
 & R_2(x) \\
 & A_2(x) \\
 & W_2(x) \\
R_1(y) & \\
A_1(y) & \\
W_1(y) & \\
 & R_2(y) \\
 & A_2(y) \\
 & W_2(y)
\end{array}
$$

Это сериализуемое расписание, но не последовательное. Тут легко можно переставить пару $W_1$ - $R_2$ местами, и оно вообще перестанет быть сериализуемым.

Не стоит рассчитывать, что изменения пользователя не меняют данные, слишком это плохо предсказывается. Это конечно увеличивает множество сериализуемых расписаний, но опираться на это не надо; надо строить расписания, сериализуемые независимо от действий.

В СуБД есть компонент ядра, управляющий транзакциями, и планировщик. Планировщик взаимодействует с обработчиком запросов, узнавая, что за действия хочет выполнить транзакция. Он должен разрешить конфликтующие дейсвтия — соседние действия, порядок которых влияет результат.

Действия внутри транзакции по определению конфликтуют. Действия между транзакциями...

| | $R_2(X)$ | $R_2(Y)$ | $W_2(X)$ | $W_2(Y)$ |
|---|---|---|---|---|
| $R_1(X)$ | $+$ | $+$ | $-$ | $+$ |
| $R_1(Y)$ | $+$ | $+$ | $+$ | $-$ |
| $W_1(X)$ | $-$ | $+$ | $-$ | $+$ |
| $W_1(Y)$ | $+$ | $-$ | $+$ | $-$ |

Неконфликтующие соседние действия коммутируют, можем менять местами. Если расписание таково, что такими перестановками можно получить последовательное расписание, сериализуемое по конфликтам.

Важно понимать, что бывают сериализуемые расписания, не сериализуемые по конфликтам. Например, если изменения данных их не меняют, то конфликт есть, а менять местами что-то можно.

Как понять сериализуемость по конфликтам расписания? Может помочь граф: вершины — транзакции, рёбра — пары транзакций, между действиями которых есть конфликты, причём ребро идёт в порядке выполнения действий в расписании.

Сам критерий... В следующей серии!

Критрерий прост: если нет циклов, то сериализуемо.

Покажем индукцией по количеству вершин. Для одной вершины циклов нет. Для нескольких вершин — для графа без циклов есть вершина-исток (нет входящих рёбер), её можно отсоеденить и применить предположение индукции.

По-сути: делаем topsort.

Как же теперь построить планировщик? Нам помогут замки, или блокировки. Хочешь что-то сделать с каким-нибудь объектом? — Возьми блокировку ($L_i(X)$). Закончил что-то делать? Отпусти ($U_i(X)$). Двух замков на одной записи быть не может.

Пока эти правила нам не очень помогают — мы можем каждую операцию с элементом обернуть в захват-освобождение, и никакой сериализации по конфликтам не получается. Нужны ещё правила.

Все захваты должны быть до всех освобождений. Если нужны какие-то элементы, хватай блокировки заранее, и снимай в конце.

Такой набор правил называется двухфазовым протоколом (2PL). Он уже гарантирует сериализуемость по конфликтам. Почему? Возьмём первую транзакцию, которая будет снимать блокировки. Вытянем её в начало, продолжим по индукции. Получится ли это? Если возник конфликт, то есть нарушение блокировки, значит в оригинале был не 2PL.

Можно думать, что транзакция уже подтверждена, когда мы начинаем освобождать блокировки.

Ещё есть проблема с дедлоками. Хотелось бы гарантировать порядок взятий... Но так сложно. Обычно поддерживают граф ожидания, и отслеживают возникновения циклов. Если возник — кто-то нехороший, и надо убить.

Ещё блокировки — иногда слишком строго. Разрешим иногда разделяемые блокировки. Их можно одновременно брать на один и тот же ресурс, но нельзя одновременно брать на этот же ресурс обычную блокировку. Это позволяет оптимизировать чтения.

# 4.1. Сбои

**Сбой носителя.** Такое всё-таки случается: на работающем диске головка царапает поверхность, данные повреждаются. Способы борьбы: дублировать!

**Системный сбой** Что-то пошло не так, программа упала или что-то такое, но данные на диске целые, хотя и не всё записанно. Перезапустились, восстановили.

**Катастрофический сбой.** Вот был ваш датацентр, иии... сгорел. Очень жаль.

**Сбой региона.** Вот были ваши датаценты в Америке, а там... взорвался Йеллоустоун. И нет ваших датацентров... И Америки нет. Тут можно много такого придумывать. Кажется, в этом сценарии может быть немножко не до ваших данных.

Системный сбой — программа записала данные в буфер, а потом мы перезагрузились... и данные пропали, мы запишем на диск ересь, или вообще ничего. Можно делать сразу гарантируемую запись на диск, долго — случайный доступ очень уж медленный.

Можно пытаться устроить последовательную запись. Построим опережающий журнал (WAL, write-ahead log): в него на диске будем сразу писать информацию о всех действиях записи, потом только запускать запись на основной диск, а потом помечать в журнале об успешности. Такой журнал имеет последовательную запись, это быстрее.

Зачем он нужен? Мы так можем после сбоя прочитать журнал и узнать, что успели сделать, что ещё надо попробовать записать. Какие вообще записи можно разместить в журнале?

## 4.1.1. Undo-logging

Надо точно записывать начала и завершения (успешные и неуспешные) транзакций. Ещё будем записывать, что в такой-то транзакции такой-то элемент хотят записать, вместе со старым значением. Зачем? Это делает эту запись Undo, то есть отменяемой.

Если мы будем следовать правилам:

- Перед записью данных пишем в журнал, что хотим записать.
- Пишем подтверждение транзакции после успешного завершения записи всех данных.

то сможем легко восстановить данные при сбое:

- Найти в журнале все неподтверждённые транзакции.
- С конца отменять их действия.

Работает хорошо! Проблемы: надо читать весь журнал, `COMMIT` ждёт запись на диск.

Как не ждать весь журнал? Checkpoint — контрольная точка в журнале. Записываем в журнал, что началась контрольная точка, ждём окончания всех транзакций, только потом закрываем точку и продолжаем. Или, точнее, выкидываем весь журнал — всё на диске. Это крайне наивный подход — не хочется, чтобы транзакции ждали конца других, долго.

Лучше так: при начале контрольной точки выписываем все текущие транзакции. Новые мы принимаем, но как только все старые завершились — завершаем контрольную точку. При чтении журнала с конца читать дальше завершённой контрольной точки не надо.

### 4.1.2. Redo-logging

Пишем начала транзакции, концы транзакции, и те же записи о записях на диск. Но правила другие: мы ничего не пишем на диск, пока не подтвердится транзакция. Тогда все неподтверждённые транзакции ещё ничего не записали, а подтверждённые, если чего-то не записали, то имеют запись о успехе. Для восстановления надо найти все подтверждённые транзакции и читая журнал с начала записывать все их операции.

Проблема: очень уж много изменений в памяти может накопиться. Тут тоже можно сделать контрольные точки: в ходе неё подтверждённые транзакции берём и таки записываем. Можно воспользоваться битом изменения страниц памяти: берём и пишем все грязные страницы подтверждённых транзакций. Итого: надо восстанавливать только с момента начала последней контрольной точки.

Этот журнал хорош для реплицирования: можно другой реплике посылать журнал и он сам восстановится.

### 4.1.3. Undo-redo-logging

Пишем всё так же, правило одно: WAL (пишем в журнал до записи на диск). Восстановление: найти подтвердившиеся, откатить неподтвердившиеся, применить подтвердившиеся.

## 4.2. Timestamp-based scheduler

MVCC — Multi-Version Concurring Control.

Каждая транзакция получает время старта. Каждая операция проверяется: может ли в момент начала транзакции быть применена эта операция. Для этого хранятся моменты времени, когда каждый элемент последний раз был прочитан и записан; а ещё флаг подтверждения для каждых данных.

Например, две транзакции: первая началась, вторая началась, первая записала X, вторая прочитала его же. Тогда вторая увидит в момент чтения, что в момент начала своей транзакции она хочет прочитать значение, изменённое транзакцией, начавшейся до него, и всё, что она сделает — подождёт, пока та подтвердит запись в элемент.

Если же порядок будет другой, и транзакция пытается прочитать значение, записанное транзакцией, начавшейся после её начала, её откатывают.

Запись — если последняя операция с элементом была транзакцией, начавшейся до нас, тупо пишем. Если читал кто-то новее — дохнем. Если писал кто-то новее — если подтвердил, то забиваем на нашу запись, иначе — ждём подтверждения, проверяем.

Проблема — может откатывать даже чтения.

# Глава 5

# Список литературы

Крис Дейт. „Введение в системы баз данных“. Там неплохо про первую часть курса.

Jeffrey Ullman, et at. Database system implementation. Есть рефакторинг: „Базы данных: полный курс“.

Борис Новиков. „Настройка производительности баз данных“.