

# DB LEC 03

isagila

ablearthy

@dashaaaaaa\_ aaaaaa

@i9kin

Собрано 24.11.2023 в 14:45



# Содержание

<b>1. Лекции</b>	<b>3</b>
1.1. Лекция 23.09.01.	3
1.2. Лекция 23.09.08.	3
1.3. Лекция 23.09.15.	4
1.4. Лекция 23.09.22.	5
1.5. Лекция 23.09.29.	5
1.6. Лекция 23.10.06.	7
1.7. Лекция 23.10.13.	8
1.8. Лекция 23.10.20.	9
1.9. Лекция 23.10.27.	10
1.10. Лекция 23.11.03.	12
1.11. Лекция 23.11.10.	12
1.12. Лекция 23.11.17.	13
1.13. Лекция 23.11.24.	13

# 1. Лекции

## 1.1. Лекция 23.09.01.

Информация бывает трех видов:

1. Сигнал.

Например, для человека, который не знает какой-либо язык, текст на этом языке будет сигналом: т.е. можно понять, что какая-то информация была передана, но принять (понять) эту информацию нельзя.

2. Знание.

Студент, слушающий лектора и пишущий конспект, получает знания.

3. Данные.

Данные — это формализованные знания. В отличие от знаний данные не искажаются в процессе передачи. Если лектор передаст студентам конспект лекции, который он написал сам, то он передаст именно данные.

**Def 1.1.1.** *Данные* — это поддающиеся многократной интерпретации представления информации в формализованном виде, пригодном для передачи, интерпретации и обработки.

Все данные проходят путь вида:

Сбор → Обработка → Передача → Хранение → Представление

## 1.2. Лекция 23.09.08.

Уровни архитектуры данных:

1. Внешний.

Решаем проблему представления данных пользователю (как агрегировать и детализировать данные?).

2. Концептуальный.

Занимаемся выделением сущностей, решаем проблемы безопасности (разрешения + ограничения). Определяем семантику (например, договариваемся, что кредитный рейтинг будет храниться в виде числа от 1 до 5 и сопоставляем каждому числу некоторое словесное описание, которое записано в документации, но не в самой базе данных).

3. Внутренний.

Решаем проблемы хранения данных на физическом уровне.

Для простоты выделяют три модели данных, которые связаны с некоторыми описанными выше уровнями архитектуры данных.

1. Модель Сущность–Связь (внешний + концептуальный уровни).

2. Логическая (даталогическая модель) модель (все уровни).

3. Физическая модель (концептуальный + внутренний уровни).

### Модель Сущность–Связь

**Def 1.2.1.** *Сущность* — это множество экземпляров (реальных или абстрактных) однотипных объектов предметной области.

**Def 1.2.2.** Сущность называется *сильной*, если её экземпляры могут существовать независимо. *Слабые* сущности могут существовать только при наличии одного или нескольких экземпляров сильной сущности.

*Пример 1.2.3.* Пусть у нас есть две сущности: Студент и Группа. Студент будет сильной сущностью, т.к. он может существовать и без группы, а Группа будет слабой сущностью, т.к. она не может существовать без студентов.

Этот пример очень условный: деление сущностей на слабые и сильные зависит от обстоятельств. В некоторых случаях Группа вполне может быть сильной сущностью.

Каждая сущность обладает одним или несколькими атрибутами. Атрибуты делятся на

1. Простые (например дата рождения)

2. Составные (например адрес, если хранить город, улицу, дом и т.п. отдельно)

*Замечание 1.2.4.* Один и тот же атрибут (например, ФИО) может в разных случаях быть как простым (если его хранить как одну строчку), так и составным (если отдельно хранить фамилию, отдельно имя и отдельно отчество).

Также атрибуты можно делить по другому принципу:

1. Обязательные (например email в некоторых случаях может быть обязательным атрибутом).

2. Необязательные (например отчество может быть необязательным атрибутом).

Помимо этого, атрибуты делятся на

1. Однозначные (например дата рождения, она у каждого ровно одна).

2. Многозначные (например номер телефона, у кого-то может быть несколько номеров телефона).

Для каждого атрибута мы определяем *домен*, т.е. множество допустимых значений. Доменом может быть перечисление, множество всех натуральных чисел, регулярное выражение и т.д.

Чтобы показать отношения между сущностями, используются связи. На уровне Сущность–Связь они обычно несут семантическую нагрузку, например Студент **принадлежит** Группе. Существует три вида связи

1. Один-к-одному (1:1).

Обычно все связи этого вида это искусственно выделенные атрибуты. Зачем же тогда нужна эта связь? Рассмотрим на примере. Допустим, у нас есть две сущности: Студент и Паспорт. Логично, что у каждого студента один паспорт и у каждого паспорта один студент, т.е. связь вида 1:1. Почему же нельзя добавить паспорт в атрибуты сущности Студент? Так можно сделать, но связь вида 1:1 может быть удобна в целях безопасности: в таком случае все паспорта можно будет хранить отдельно от студентов и (например) дополнительно шифровать (шифровка же всей сущности Студент будет мало того, что излишне затратной, так еще и не очень нужной).

2. Один-ко-многим (1:N).

3. Многие-ко-многим (N:N).

### 1.3. Лекция 23.09.15.

**Def 1.3.1** (по Коннолли и Бегг). *База данных* — это совместно используемый набор логически связанных данных и описание этих данных, предназначенный для удовлетворения информационных потребностей предприятия.

**Def 1.3.2** (по Дейту). *База данных* — это набор постоянно хранимых данных, используемых прикладными системами предприятия.

**Def 1.3.3** (по Хомоненко). *База данных* — это совокупность специальным образом организованных данных, хранимых в памяти вычислительной системы и отображающих состояние объектов и их взаимосвязей в рассматриваемой предметной области.

Рассмотрим некоторые модели данных.

#### Иерархическая модель данных

Эта модель удобная для восприятия человеком, т.к. он часто сталкивается с разного рода иерархиями в реальном мире. У нас есть некоторые поля данных, которые являются неделимыми атомами, а некоторые объединение этих полей называется *сегментом данных*. Каждая запись в БД — это экземпляр сегмента данных.

К минусам этой модели можно отнести дублирование данных, вследствие чего возникает некоторая сложность поддержки их целостности.

#### Сетевая модель данных

Если иерархическая модель данных по сути являлась деревом, то сетевая модель представляет собой граф, в котором вершины представляют собой некоторые экземпляры объектов.

#### Реляционная модель данных

В отличие от иерархической и сетевой модели элементы модели сразу двумерные (т.е. таблицы) и при этом мы не храним связь явно — мы выделяем некоторый атрибут, который наделяем семантикой связи. Это может привести к некоторым проблемам — семантика может быть неправильна трактована, в результате чего запрос к БД либо не выполнится, либо выполнится некорректно. Проблему поддержки целостности данных будем решать нормализацией (об этом будет рассказано в последующих лекциях). Одним из плюсов этой модели является то, что мы можем оценить время работы запроса к БД.

#### Постреляционная модель данных

Берем реляционную модель и снимаем запрет на неделимость поля, т.е. в качестве атрибута теперь можно хранить массив строк через запятую или сразу JSON с какими-либо данными.

Эта модель была сделана для того, чтобы решить проблему очень большой персонификации данных. Допустим, есть таблица с деталями и у некоторых деталей есть какие-то атрибуты, которых в принципе не может быть у других. Если добавить все возможные атрибуты каждой детали, то получим очень разряженную таблицу почти полностью состоящую из null-значений  $\Rightarrow$  проблемы с памятью.

С другой стороны мы усложняем себе поиск по такой базе (сложно проверять детали, которые имеют разный набор атрибутов) и поддержку целостности — поле JSON-данными при желании можно записать некорректное значение (либо нужно жестко валидировать все записываемое в каждую такую ячейку  $\Rightarrow$  проблемы со скоростью работы).

#### Многомерная модель данных

Пусть есть таблица со столбцами «Продавец», «Товар», «Регион», «Квартал» и «Количество». Мы хотим уметь быстро строить сводки по продавцу, товару, региону, кварталу (или по комбинации параметром), находить наибольшее и наименьшее значения и т.п.

Можно представить наши данные в 4-ёх мерном пространстве, где первые четыре колонки будут задавать координаты, а пятая (количество товара) — значение. Теперь, чтобы получать различные метрики, мы будем рассекать полученный гиперкуб гиперплоскостями, искать в них минимум/максимум и т.п.

Данная модель имеет большие затраты по памяти, но зато с её помощью можно быстро выполнять запросы. Её можно использовать в следующем ключе: храним данные в реляционной модели. Ночью строим по ней многомерную модель, а днем пользуемся ей не внося никаких изменений (будем считать, что продажи товаров за день не сильно влияют на глобальную ситуацию  $\Rightarrow$  ими можно пренебречь). Следующей ночью заново строим многомерную модель по уже новым данным и так далее. Таким образом в течение дня мы получаем возможность быстро делать разные запросы и узнавать разные метрики.

### Объектно-ориентированная модель данных

Похожа на постреляционную модель: мы сериализуем все объекты в системе и храним пары вида `id: [serialized]`. Это усложняет поиск, т.к. нужно десериализовывать данные.

## 1.4. Лекция 23.09.22.

**Def 1.4.1.** Схема отношения — это строка заголовков.

**Def 1.4.2.** Одна строка называется кортежесем.

**Def 1.4.3.** Один столбец называется атрибутом. Заголовок столбца определяет его имя.

**Def 1.4.4.** Домен — это множество допустимых значений атрибута.

**Def 1.4.5.** Степень отношения — это количество его атрибутов.

**Def 1.4.6.** Кардинальность отношения — это количество его кортежей.

**Def 1.4.7.** Отношение — это множество упорядоченных кортежей, где каждое значение берется из соответствующего домена.

$$R = \{d_1, \dots, d_n\} \quad d_i \in D_i$$

Свойства отношений (по Кодду):

1. Каждая ячейка содержит одно неделимое значение.
2. Каждый кортеж уникален.
3. Уникальность имени отношения в реляционной схеме.
4. Уникальность имени атрибута в пределах отношения.
5. Значения атрибута берутся из одного и того же домена.
6. Порядок следования атрибутов и кортежей не имеет значения.

**Def 1.4.8.** Суперключ — это атрибут (множество атрибутов), который единственным образом идентифицирует кортеж.

**Замечание 1.4.9.** Схема отношения всегда является суперключом (см. второе свойство отношений).

**Def 1.4.10.** Потенциальный ключ — это суперключ, который не содержит подмножества, также являющегося суперключом этого отношения.

**Def 1.4.11.** Первичный ключ — это потенциальный ключ, который выбран для идентификации кортежей внутри отношения.

**Замечание 1.4.12.** Иногда используют технический первичный ключ (например поле `id`), который не несет никакой семантики, а служит лишь для определения уникальности кортежа.

**Def 1.4.13.** Внешний ключ — это атрибут (множество атрибутов) внутри отношения, который соответствует потенциальному ключу некоторого (возможно того же самого) отношения.

**Def 1.4.14.** Целостность сущностей означает, что в отношении первичный ключ не может содержать `NULL` значения.

**Def 1.4.15.** Ссылочная целостность означает, что если в отношении существует внешний ключ, то его значение либо соответствует значениям потенциального ключа, либо полностью состоит из `NULL` значений.

## 1.5. Лекция 23.09.29.

**Def 1.5.1.** Реляционная алгебра — это теоретический язык операций, позволяющий на основе одного или нескольких отношений создавать другие отношения без изменения самих исходных отношений.

Реляционная алгебра замкнута. В ней существуют следующие операции:

## 1. Проекция $\Pi_{a_1 \dots a_n}(R)$

Результатом проекции является новое отношение, содержащее вертикальное подмножество исходного отношения, создаваемое посредством извлечения указанных атрибутов и исключения из результата атрибутов-дубликатов.

## 2. Выборка $\sigma_{\text{предикат}}(R)$

Результатом выборки является отношение, которое содержит только те кортежи из исходного отношения, которые удовлетворяют заданному условию (предикату).

## 3. Объединение $R \cup S$

Объединение двух отношений  $R$  и  $S$  определяет новое отношение, которое включает все кортежи, содержащиеся только в  $R$ , все кортежи, содержащиеся только в  $S$  и кортежи, содержащиеся и в  $R$ , и в  $S$  с исключением дубликатов.

Объединять можно не все отношения, а только совместные по объединению отношения. Отношения совместны по объединению, когда они состоят из одинакового количества атрибутов и каждая соответствующая пара атрибутов имеет одинаковый домен.

## 4. Разность $R - S$

Разность состоит из кортежей, которые есть в  $R$ , но отсутствуют в  $S$ . Разность двух отношений определена только если они совместны по объединению.

## 5. Пересечение $R \cap S$

Операция пересечения определяет отношение, содержащее кортежи, находящиеся как в  $R$ , так и в  $S$ . Пересечение двух отношений определено только если они совместны по объединению.

## 6. Декартово произведение $R \times S$

Декартово произведение определяет новое отношение, которое является результатом конкатенации каждого кортежа из отношения  $R$  с каждым кортежем из отношения  $S$ .

## 7. Тета-соединение $R \bowtie_F S$

Определяет отношение содержащее кортежи из декартового произведения  $R \times S$ , удовлетворяющие предикату  $F = R_{a_i} \Theta S_{b_i}$ , где  $\Theta$  это одна из операций сравнения  $\{>, <, =, \dots\}$

## 8. Экви-соединение

Это тета-соединение, где  $\Theta$  это  $\ll=$ .

## 9. Естественное соединение $R \bowtie S$

Соединение по эквивалентности двух отношений, выполненное по всем общим атрибутам, из результатов которого исключили по одному экземпляру каждого атрибута.

## 10. Левое внешнее соединение $R \Join S$

Это естественное соединение, при котором в результирующее отношение включаются также кортежи отношения  $R$ , не имеющие совпадающих значений в общих атрибутах отношения  $S$ .

## 11. Полусоединение $R \triangleright_F S$

Это отношение, содержащее кортежи  $R$ , которые входят в тета-соединение  $R$  и  $S$ .

## 12. Деление.

Эту операцию мы рассматривать не будем.

Таким образом общий вид `SELECT` запроса будет следующим

```
SELECT [DISTINCT | ALL] { * | [ColumnExpr [AS NewName]] [, ...]}
FROM Table [AS NewName]
[ { INNER | LEFT OUTER | FULL } JOIN Table [AS NewName] [, ...]]
[WHERE condition]
[GROUP BY ColumnList [HAVING condition]]
[ORDER BY ColumnName [ASC | DESC]]
```

Порядок выполнения SQL-запроса будет таким:

1. FROM ... ON ... JOIN

2. WHERE

3. GROUP BY

4. HAVING

5. SELECT

6. DISTINCT

7. ORDER BY

## 1.6. Лекция 23.10.06.

Соединение двух таблиц можно сделать через два вложенных цикла (обычно медленно), либо через сортировку двух таблиц и проход по ним с помощью двух указателей (как в сортировке слиянием). Второй способ обычно быстрее, кроме случаев, когда в одной из таблиц очень мало записей, а в другой — очень много.

**Def 1.6.1.** *Нормализация* — это приведение отношения к нормальной форме. Она нужна, чтобы убрать избыточность и аномалии.

Пусть есть «плохое» отношение, в котором есть три атрибута: ФИО, Группа, Образовательная программа (ОП). Сразу видно дублирование: связь группы и образовательной программы хранится несколько раз, также есть избыточность: нам достаточно знать группу, чтобы определить ОП.

**Def 1.6.2.** *Аномалия модификации*: изменение значения одной записи повлечет за собой просмотр всей таблицы и изменение некоторых других записей

*Пример 1.6.3.* Нужно сменить ОП для группы? Придется пройти по всем записям. Это долго, а также можно что-то пропустить.

**Def 1.6.4.** *Аномалия удаления*: при удалении записи может пропасть и другая информация.

*Пример 1.6.5.* Удаляем всех студентов из группы и теряем связь группы с ОП.

**Def 1.6.6.** *Аномалия добавления*: информацию в таблицу нельзя поместить пока она не полная или требуется дополнительный просмотр таблицу.

*Пример 1.6.7.* Добавляем студента, который не знает ОП, а знает только группу.

**Def 1.6.8.** В отношении  $R$  атрибут  $y$  функционально зависит от атрибута  $x \iff$  каждому  $x$  соответствует в точности один  $y$ . В этом случае  $x$  называется детерминантой, а  $y$  — зависимой частью.

**Def 1.6.9.** *Частичная функциональная зависимость* — это зависимость неключевого атрибута от части составного потенциального ключа.

*Пример 1.6.10.* ОП частично функционально зависит от составного ключа ФИО-Группа.

**Def 1.6.11.** *Полная функциональная зависимость* — это зависимость неключевого атрибута от составного потенциального ключа.

*Пример 1.6.12.* Если добавить в рассматриваем пример атрибут Форма обучения, то он будет полно функционально зависеть от составного ключа ФИО-Группа

**Def 1.6.13.** *Транзитивная функциональная зависимость*: два атрибута находятся в транзитивной функциональной зависимости, если существует атрибут (множество атрибутов) такой, что второй атрибут находится в функциональной зависимости от этого атрибута, а сам этот атрибут функционально зависит от первого атрибута.

Иными словами, если даны три множества  $A, B, C$  и

1.  $B$  зависит от  $A$ ,

2.  $A$  не зависит от  $B$ ,

3.  $C$  зависит от  $B$ ,

то зависимость  $A \rightarrow C$  называется транзитивной.

**TODO:** в тетради нет четкого определения

## Нормальные формы

Нормальные формы «вложены» друг в друга: для того, чтобы отношение находилось в определенной нормальной форме, оно должно находиться в предыдущей нормальной форме и должно выполняться некоторое дополнительное условие. Названия нормальных форм будем сокращать как (например) 1НФ — первая нормальная форма.

**Def 1.6.14 (1НФ).** *Первая нормальная форма*: все атрибуты отношения являются *простыми* (не пытаемся использовать атрибуты как составные).

**Def 1.6.15 (2НФ).** *Вторая нормальная форма*: 1НФ и каждый неключевой атрибут *функционально полно* зависит от первичного ключа (то есть нет *частичных* функциональных зависимостей).

**Def 1.6.16 (3НФ).** *Третья нормальная форма*: 2НФ и все неключевые атрибуты взаимнонезависимы и полностью зависят от первичного ключа.

ЗНФ можно определить иначе:

**Def 1.6.17 (ЗНФ).** 2НФ и ни один неключевой атрибут не находится в *транзитивной* функциональной зависимости от потенциального ключа.

*Замечание 1.6.18.* (На это необязательно смотреть!)

Также есть другое эквивалентное определение, данное Карлом Заниоло: таблица находится в ЗНФ тогда и только тогда, когда для каждой функциональной зависимости  $X \rightarrow Y$  выполнено одно из следующих условий

1.  $Y \subseteq X$  — тривиальная зависимость

2.  $X$  является суперключом

3. каждый атрибут  $Y \setminus X$  является ключевым, то есть входит в состав какого-то потенциального ключа.

Данная формулировка удобна тем, что переход к БКНФ осуществляется удалением последнего требования.

**Def 1.6.19 (БКНФ).** *Нормальная форма Бойса-Кодда (БКНФ):* ЗНФ и детерминанты всех зависимостей являются потенциальными ключами.

*Пример 1.6.20.* Пусть есть отношение с атрибутами ИСУ, Паспорт, ID Проекта и Роль. ИСУ + ID Проекта и Паспорт + ID Проекта это первичные ключи. Если человек решит сменить паспорт, то он может сменить его относительно одного проекта, а относительно другого — забыть. Для решения этой проблемы следует разбить это отношение на два: ИСУ, ID Проекта, Роль и ИСУ, Паспорт.

**Def 1.6.21 (4НФ).** *Четвертая нормальная форма:* БКНФ и отношение не содержит нетривиальных *многозначных* зависимостей.

*Пример 1.6.22.* Пусть есть отношение ID Дисциплины, ID Лектора, ID Практика. Если лектор решил уволится, то нужно везде заменить его, но можно ошибиться и поменять не везде. Можно попытаться разбить это отношение на два: ID Дисциплины, ID Лектора и ID Дисциплины, ID Практика, но тогда потеряется связь лектора и практика (а что если некоторые лекторы могут работать только с определенными практиками и наоборот?).

*Замечание 1.6.23.* Если в отношении два столбца, то оно находится во всех нормальных формах, т.к. его более нельзя разбить

*Замечание 1.6.24.* Существуют также 5-я и 6-я нормальные формы, но они редко используются на практике. На практике же, наоборот, иногда прибегают к денормализации для повышения скорости работы.

## 1.7. Лекция 23.10.13.

### Индексы

Можно попробовать отсортировать записи по какому-нибудь атрибуту и использовать бинпоиск. Однако запись в БД нельзя прочитать точечно, поэтому приходится читать блоками, из-за этого бинпоиск не очень эффективен. Также возникают проблемы для строковых атрибутов (сравнение строк не очень быстрое). Помимо этого не очень ясно, как поддерживать отсортированность при добавлении/удалении элементов.

**Def 1.7.1.** *Индекс* — это структура данных типа ключ-значение, где ключ — это результат некой функции от совокупности атрибутов, а значение — адрес хранимой записи.

Пусть индексом будет первичный ключ, и хранить мы его будем в виде сбалансированного дерева. Тогда получаем:

**Def 1.7.2.** *Первичный индекс* — это индекс, построенный по первичному ключу при условии, что исходный файл отсортирован по нему же.

Первичный индекс является *плотным*, т.е. охватывает все уникальные записи (количество кортежей в отношении равно количеству вершин в дереве индекса).

**Def 1.7.3.** *Индекс кластеризации* — это индекс, построенный по неключевому полю при условии, что исходный файл отсортирован по нему же.

*Пример 1.7.4.* В качестве индекса кластеризации будем брать первые 5 символов ФИО и для каждого индекса будем хранить ссылку на первый кортеж в кластере. Теперь по ФИО можно быстро найти кластер, где лежат ФИО с одинаковыми первыми 5-ю символами, а дальше уже найти требуемый кортеж.

*Замечание 1.7.5.* При использовании индекса кластеризации мы ищем баланс между размером и плотностью индекса: либо индекс большой и плотный (нужно больше памяти), либо индекс маленький и разреженный (нужно меньше памяти).

**Def 1.7.6.** *Вторичный индекс* — это индекс, построенный по неключевому полю при условии, что исходный файл не отсортирован.

*Пример 1.7.7.* Почту можно развернуть и кластеризовать.

*Замечание 1.7.8.* Дерево индекса не обязательно всегда держать сбалансированным, можно считать метрику «разбалансированности» и иногда делать перебалансировку.

## Представления

**Def 1.7.9.** *Представление* — это динамически сформированный результат одной или нескольких реляционных операций, сохраненный в виде нового отношения.

Представления делятся на:

1. Материализованные.

На самом деле хранятся в памяти и обновляются раз в некоторое время.

2. Представления замены.

В памяти хранится только запрос для получения этого представления. При получении запроса извне этот сохраненный запрос подставляется во внешний запрос и полученный таким образом запрос выполняется. Такой способ обеспечивает безопасность, но не дает выигрыша в скорости (скорее даже снижает скорость).

Представления бывают обновляемыми (через них можно что-то поменять в исходных отношениях) и нет.

## Преимущества представлений

1. Независимость от данных.
2. Повышение защищенности данных.
3. Снижение сложности работы с данными.

## Недостатки представлений

1. Ограниченност в возможностях обновлений.
2. Структурные ограничения (вынужденность подстраиваться под заданный интерфейс представления).
3. Снижение производительности.

## 1.8. Лекция 23.10.20.

### Надежность

Данные должны быть целостны и доступны на всем протяжении эксплуатации системы. Существует физическая и логическая сохранность данных. Проблему физической сохранности данных можно решить на уровне оборудования. Например, есть RAID массивы. Самый простой RAID массив (RAID-1, mirror, зеркало) представляет собой два диска, один из которых содержит копию данных. Такой подход не очень эффективен, т.к. сильно возрастают затраты на память, а также могут проблемы с синхронной записью данных сразу на два диска.

Есть RAID-5: этот массив представляет собой три диска, причем на двух дисках лежат данные, а на третьем — побитовый XOR этих данных. В таком случае требуется меньше дополнительной памяти. Также можно для хранения XOR-а использовать не один диск, а каждый из трех дисков по-очереди. Например, разбить данные на блоки, после чего положить первые два блока на первые два диска, а их XOR — на третий. Третий и четвертый блоки можно положить на второй и третий диски, а их XOR — на первый. И так далее по кругу. Таким образом проблему физического хранения данных можно решить увеличив затраты на оборудование.

**Def 1.8.1.** *Транзакция* — это последовательность действий с базой данных, в которой либо все действия выполняются успешно, либо не выполняется ни одно из них.

Результатом транзакции может быть либо `commit`, либо `rollback`.

Свойства транзакции:

A *Атомарность* (неделимость). В транзакции выполняются либо все действия, либо ни одного.

C *Согласованность*. Транзакция переводит одно согласованное состояние в другое согласованное состояние без обязательной поддержки согласованности в промежуточных точках.

I *Изоляция*. Если запущено несколько конкурирующих транзакций, то любое обновление состояния базы данных, выполненное одной транзакцией скрыто от других транзакций до ее завершения.

D *Долговечность*. Когда транзакция завершена, ее результат обновления сохраняется, даже если в следующий момент произойдет сбой.

4 проблемы конкурирующих транзакций:

1. Проблема потерянного обновления. Несколько транзакций меняют один и тот же кортеж. В результате сохраняется результат внесения изменений только последней транзакции.

Пример: у человека  $n$  на счету 200 тугриков. Он переводит человеку  $k$  100 тугриков. В это же время человек  $m$  переводит человеку  $n$  100 тугриков. Две транзакции пытаются изменить счет человека  $n$ , в итоге у него окажется либо 100, либо 300 тугриков (в зависимости от порядка завершения транзакций).

2. Проблема грязного чтения. Возникает при чтении одной транзакцией кортежа, которых уже изменен, но еще не сохранен другой незавершенной транзакцией, которая потом будет отменена.

Пример: у человека  $n$  на счету 200 тугриков. Он переводит человеку  $k$  100 тугриков. В это же время по всей таблице идет транзакция, которая считает количество денег на счету и начисляет проценты. Она считает, что у  $n$  осталось 100 тугриков, однако при этом может возникнуть сбой, перевод тугриков не пройдет и 100 тугриков вернутся на счет  $n$ . В результате ему будут начислены неправильные проценты.

3. Проблема неповторяемого чтения. При повторном чтении данных, уже считанных ранее, транзакция обнаруживает модификации, вызванные другой завершенной транзакцией.

Пример: транзакция идет по таблице и считает сумму покупки, чтобы потом вычислить скидку. В это время другая транзакция меняет цены уже прочитанного товара. В итоге скидка может быть посчитана неверно.

4. Проблема фантомного чтения. При повторном чтении данных транзакция обнаруживает новые кортежи, добавленные другой транзакцией, завершенной после предыдущего чтения данных.

Пример: транзакция идет по таблице и считает сумму покупки, чтобы определить цену доставки. В это же время другая транзакция добавляет в эту таблицу новый кортеж. В итоге цена доставки может быть посчитана неверно. Блокировки бывают явные (прописываются сразу в тексте транзакции) и неявные (задан набор правил, по которым блокировка устанавливается автоматически). Блокировать можно один кортеж, одну таблицу или всю базу данных. Также блокировки делятся на монопольные (блокируют все виды доступа) и коллективные (блокируют только внесение изменений).

4 уровня изоляции (каждый уровень решает следующую проблему конкурирующих транзакций):

1. Незавершенное чтение. Изменять данные должна только одна транзакция.

2. Завершенное чтение. Если транзакция собирается менять данные, то она блокирует их монопольно.

3. Воспроизведимое чтение. Если транзакция планирует читать данные, то она блокирует их монопольно.

4. Сериализуемость. Если транзакция обращается к данным, то никакая другая транзакция не может добавить новые или изменить существующие строки, которые могут быть считаны при выполнении этой транзакции.

## 1.9. Лекция 23.10.27.

### Безопасность

Когда мы говорили о надежности, то мы говорили в первую очередь о предотвращении случайных ошибок, непреднамеренных поломок и т.д. Когда мы говорим о безопасности, то здесь речь идет в первую очередь об защите от намеренных негативных действий.

**Def 1.9.1.** Система будет *безопасной*, если она посредством специальных механизмов защиты контролирует доступ к информации таким образом, что только имеющие соответствующие полномочия лица или процессы могут получить доступ на чтение, изменение, создание, удаление информации.

### Классы безопасности

#### Класс D

Сюда относятся системы, которые не удовлетворяют требованиям других классов.

#### Класс C. Дискреционный доступ

Система относится к классу C, если она имеет систему аутентификации/идентификации и дискреционное управление доступом.

**Def 1.9.2. Идентификация** — это фиксация в системе идентификатора пользователя.

**Def 1.9.3. Аутентификация** — это механизм проверки идентификатора (сопоставление пользователя с его идентификатором).

**Def 1.9.4. Авторизация** — это процесс предоставления прав аутентифицированному пользователю.

Идентификаторы бывают трех видов:

1. То, что пользователь знает. Например, пароль, кодовое слово и т.п.

2. То, чем пользователь владеет. Например, телефон, ключ–карта и т.п.
3. То, чем является неотъемлемой частью пользователя (чаще всего это биометрия). Например, отпечаток пальца и т.п.

Далее о дискреционном доступе. Пусть есть объекты (файлы, таблицы, базы данных, в целом все, что мы хотим защитить) и субъекты (пользователи, группы пользователей). Составим табличку, где каждому субъекту будет соответствовать строка, а каждому объекту — строка. В каждой ячейке запишем уровень доступа (только чтение, чтение + запись, чтение + запись + удаление и т.п.), который мы хотим предоставить данному субъекту с данному объекту.

К созданию и редактированию получившейся таблицы есть три подхода:

1. Супер–админ. Есть только одна точка изменения (у этого есть как плюсы, так и минусы).
2. Для каждого объекта определяем ровно одного субъекта, который будет его владельцем. Владелец имеет право раздавать права на свой объект по своему усмотрению (зависит от ограничений и дизайна системы).
3. Создаем отдельное право «давать права» и наделяем этим правом некоторых субъектов (ответ на вопрос о рекурсивной выдаче прав зависит от дизайна системы).

*Замечание 1.9.5.* Уровень С делится на подуровни С1 и С2. Для подуровня С1 субъект может быть группой пользователей, в то время как для подуровня С2 необходима гранулированность субъектов до конкретного пользователя.

## Класс В. Мандатное управление доступом

Пусть есть следующая матрица доступа:

	01	02
S1	R	RW
S2		R

Пусть данные 01 — секретные. Однако субъект S1 может прочитать их и записать в 02, откуда они станут доступны для субъекта S2, который не должен иметь к ним никакого доступа. Для того, чтобы избежать подобных проблем, существует мандатное управление доступом.

**Def 1.9.6.** В случае *мандатного* управления доступом каждому объекту и субъекту сопоставляется некоторая метка секретности из заранее определенного ранжированного списка.

Субъекту разрешается читать данные своего уровня секретности и ниже. Субъекту разрешается писать данные на свой уровень секретности и **выше** (именно это и позволяет избежать утечки данных).

Уровень В делится на подуровни:

1. В1. Только мандатное управление доступом.
2. В2. Дополнительно к уровню В1 метками снабжаются не только все объекты в базе данных, но и все объекты, доступные системе.
3. В3. Добавляется журналирование всех событий

## Класс А. Проверенный дизайн

Помимо безопасности уровня В3 также гарантируем контроль безопасности не только конечной системы, но и всего процесса разработки.

### Ролевая модель доступа

	01	02		S1	S2
R1	R	RW	R1	R	RW
R2	R		R2	R	

Теперь у каждого субъекта есть несколько ролей, причем каждая из ролей дает свой уровень доступа к разным объектам. В таком варианте проще отслеживать и изменять доступ пользователя.

*Замечание 1.9.7. Аудит* — это анализ результатов логирования и журналирования. Например, можно настроить алERTы и автоматическую реакцию на подозрительные действия пользователя (частая смена пароля, например).

Шифрование:

1. Прозрачное шифрование. Выполняется средствами базы данных при записи на диск, при этом для всей базы одинаковый ключ шифрования.
2. Колоночное шифрование. Разные столбцы шифруются разными ключами.
3. Шифрование на уровне файловой системы.
4. Шифрование на уровне приложения.

## 1.10. Лекция 23.11.03.

Вместо лекции была рубежка.

## 1.11. Лекция 23.11.10.

**Def 1.11.1.** Распределенная база данных — набор логически связанных между собой разделяемых данных и их описаний, которые физически распределены по нескольким вычислительным узлам.

Фрагментация бывает

1. Горизонтальной (партиционирование, шардирование).
2. Вертикальной.  
В этом случае сложнее поддерживать целостность.
3. Смешанной.

**Def 1.11.2.** Реплика это множество различных физических копий одного объекта базы данных, для которых обеспечивается синхронизация между собой.

Есть 3 стратегии размещения:

1. Раздельное (фрагментированное) размещение.  
Каждый фрагмент существует в единственном экземпляре и лежит на отдельном узле.
2. Размещение с полной фрагментацией.  
На каждом узле храним полную копию базы данных.
3. Размещение с выборочной фрагментацией.  
Для каждого фрагмента определяем нужное количество реплик.

**Def 1.11.3.** Распределенная СУБД — это комплекс программ, предназначенных для управления распределенной базой данных и позволяющий сделать распределенность данных прозрачной для конечного пользователя.

4 уровня прозрачности:

1. Прозрачность фрагментации.
2. Прозрачность расположения фрагмента.
3. Прозрачность количества реплик.
4. Прозрачность контроля доступа.

Распределенные базы данных делятся на гомогенные (используется одна и та же СУБД) и гетерогенные (используются разные СУБД и, возможно, даже разные модели данных).

12 правил Дейта для распределенной СУБД:

1. Правило локальной автономности.  
Локальные данные принадлежат локальным владельцам и сопровождаются локально.
2. Отсутствие опоры на центральный узел.  
В системе не должно быть ни одного узла, без которого она не сможет функционировать.
3. Непрерывное функционирование.
4. Независимость от расположения.
5. Независимость от фрагментации.
6. Независимость от репликации.
7. Поддержка обработки распределенных запросов.
8. Поддержка обработки распределенных транзакций.
9. Независимость от типа оборудования.
10. Независимость от сетевой архитектуры.
11. Независимость от операционной системы.
12. Независимость от типа СУБД.

Для поддержки распределенных запросов и транзакций необходимо знать:

1. К какому фрагменту нужно обратиться?
2. Какую копию фрагмента использовать (если это репликация)?
3. Какое из местоположений нужно использовать для построения временных структур? Какие данные как хранить и как их перемещать?

## 1.12. Лекция 23.11.17.

Причины появления NoSQL решений:

1. Big Data
2. Взаимосвязь данных
3. Слабо структурированные данные
4. Архитектура информационных систем

Свойства NoSQL хранилища:

1. Не используем язык SQL для запросов
2. Schemaless
3. Aggregates (сразу храним частично агрегированные данные, несмотря на то, что это может привести к проблемам)
4. Weak ACID → BASE
  - (a) Basic availability: каждый запрос гарантированно завершается.
  - (b) Soft state: база находится в гибком состоянии.
  - (c) Eventual consistency: данные станут консистентными в конечном случае.

## 1.13. Лекция 23.11.24.

**Def 1.13.1.** Согласованность (consistency) — во всех узлах в один момент времени данные не противоречат друг другу.

**Def 1.13.2.** Доступность (availability) — любой запрос к распределенной системе завершается корректным откликом в пределах заданного интервала времени, однако без гарантии, что ответы всех узлов совпадут.

*Замечание 1.13.3.* В данном случае *корректно* не значит *правильно*. Под корректностью подразумевается то, что система не ответит «не знаю».

**Def 1.13.4.** Partition tolerance (на русский переводится примерно как «устойчивость к разбиению») — расщепление распределенной системы на несколько изолированных секций не приводит к некорректности отклика какой-либо из секций.

**Теорема 1.13.5. (CAP-теорема)** В распределенной системе возможно обеспечить не более двух из трех перечисленных свойств: consistency, availability, partition tolerance.

□ Т.к. формальное доказательство CAP-теоремы весьма длинное, то вместо него построим конструктивный пример, иллюстрирующий противоречие, заложенное этой теоремой. Допустим, мы хотим создать удаленный сервис, в который пользователи смогут звонить по телефону. У каждого пользователя будет две опции: запланировать событие на некоторую дату и получить список событий, запланированных им на текущий день.

При таком подходе получаем классическую систему, состоящую из одного узла. Ее основной проблемой является доступность. В нашем примере может произойти следующее:

1. Оператор, отвечающий на звонки клиентов, заболел  $\Rightarrow$  весь сервис теперь не может работать.
2. Количество клиентов сильно возросло  $\Rightarrow$  один оператор не успевает обслуживать их всех. Клиентам приходится подолгу ждать своей очереди.

Видя эти проблемы, мы решаем нанять себе несколько помощников. Теперь каждый звонок клиента сначала попадает на балансировщик, который определяет кому помочнику отправить этого клиента. Сейчас мы не будем вдаваться в то, как работает этот балансировщик, лишь предположим самый очевидный вариант: он выбирает помощника с наименьшей загруженностью.

Таким образом, мы решили проблему с доступностью: нам нужно нанять достаточно количество помощников, чтобы они успевали справляться со всеми запросами. Теперь выход из строя одного из помощников не сильно влияет на работу всего сервиса, более того, при необходимости можно варьировать количество помощников в зависимости от нагрузки на сервис.

Полученное решение называется распределенной системой с независимыми узлами. У него есть очевидная проблема с согласованностью данных: т.к. узлы никак не синхронизируются между собой, то пользователь может сначала попасть на один узел, оставить там запись о своих запланированных делах, а когда решит узнать о том, что он запланировал

на сегодня, балансировщик отправит его на другой узел. На другом узле нет информации о планах этого пользователя, в итоге мы обманываем пользователя.

Для решения этой проблемы мы решаем делать следующим образом: теперь каждый узел при получении запроса на запись, не «кладя трубу» с клиентом, отправляет запросы всем остальным узлам, чтобы те зафиксировали у себя новые запланированные дела этого клиента.

Это решение называется распределенной системой с транзакционной синхронизацией (с транзакционной репликацией). У этого решения есть проблемы:

1. Мы проигрываем в производительности: при каждом запросе на запись необходимо отправить данные для синхронизации всем узлам, причем некоторые узлы могут быть заняты в это время  $\Rightarrow$  нам придется ждать, пока они освободятся. В итоге клиенту, который нам позвонил, придется очень долго «висеть» на линии, ожидая того, пока все узлы получат необходимые данные и **сообщат исходному узлу об успешном получении**.
2. Если один из узлов вышел из строя, то мы можем неопределенно долго ждать его восстановления  $\Rightarrow$  теряем доступность данных. Не ждать мы не можем, т.к. в таком случае получим неконсистентные данные на разных узлах (когда отвалившийся узел вернется в строй).

Попытаемся решить вторую проблему. У каждого узла мы сделаем дополнительный буфер («почтовый ящик»). Теперь все данные для синхронизации будут приходить в этот буфер, и перед тем, как войти в строй, узел должен будет прочитать все «входящие сообщения», которые другие узлы ему прислали, и выполнить необходимые операции для восстановления консистентности данных.

Полученный подход называется распределенной системой с гарантированной репликацией. Его проблема заключается в том, что узлы обычно общаются по сети, а в сети могут быть помехи (нельзя гарантировать доставку данных). Таким образом, если узел нам не отвечает, то мы не можем гарантированно сказать почему это происходит. Проблема может быть как в узле (если он вышел из строя), так и в том, что наше соединение с этим узлом не устойчивое и данные просто не могут до него дойти. Чтобы решить эту проблему нужно все узлы расположить физически близко друг к другу, чтобы можно было передавать данные условно напрямую, но тогда нарушается третье свойство (partition tolerance) — мы не можем разбить такую систему на изолированные части.

В процессе организации работы нашего вымышленного сервиса мы всегда балансируем между согласованностью, доступностью и возможностью разбиения: ради чего-то одного приходилось отказываться от другого и никогда не получалось собрать все три свойства одновременно. ■

Для распределенной системы наиболее важно третье свойство (возможность разбиения на части), поэтому все распределенные системы это CP или AP системы. CA система не может быть распределенной.

Виды NoSQL баз данных:

1. Базы данных вида ключ-значение (обычно AP)
2. Документоориентированные базы данных (обычно CP)
3. Колоночные хранилища (обычно CA)

Храним набор пар вида *〈первичный ключ, значение колонки〉*. Данные храним колонками, т.е. сначала идут данные из первой колонки, потом из второй и так далее. Из плюсов: можно не хранить NULL значения, т.к. если для данного первичного ключа у нас нет пары текущего атрибута, то мы можем просто никак это не хранить.

4. Графовые базы данных

О них речь пойдет на следующей лекции.