# C# Developer. Professional

## «Базы данных: организация работы с потоками данных»

otus.ru


**Проверить, идет ли запись**

# Меня хорошо видно && слышно?

Ставим “+”, если все хорошо
“-”, если есть проблемы

Тема вебинара

# «Базы данных: организация работы с потоками данных»


## Елена Сычева

**Team Lead Full-Stack Developer**

**Об опыте:**

Более 15 лет опыта работы разработчиком (C#, Angular, .Net, React, NodeJs)

**Телефон / эл. почта / соц. сети:**
https://t.me/lentsych

# Правила вебинара

Активно
участвуем

Задаем вопрос
в чат или голосом

Вопросы вижу в чате,
могу ответить не сразу

## Условные обозначения

Индивидуально

Время, необходимое
на активность

Пишем в чат

Говорим голосом

Документ

Ответьте себе или
задайте вопрос

# Маршрут вебинара

- Что это? Зачем это?
- Базы данных SQL: структура и принципы ACID
- Базы данных NoSQL: структура и варианты использования
- Шардинг, секционирование и репликация
- Объектно-реляционное сопоставление (ORM) в C#

# Цели вебинара

## К концу занятия вы сможете

1. Понимать какой тип базы где лучше применять
2. Отличать SQL, noSQL базы

# Смысл

## Зачем вам это уметь

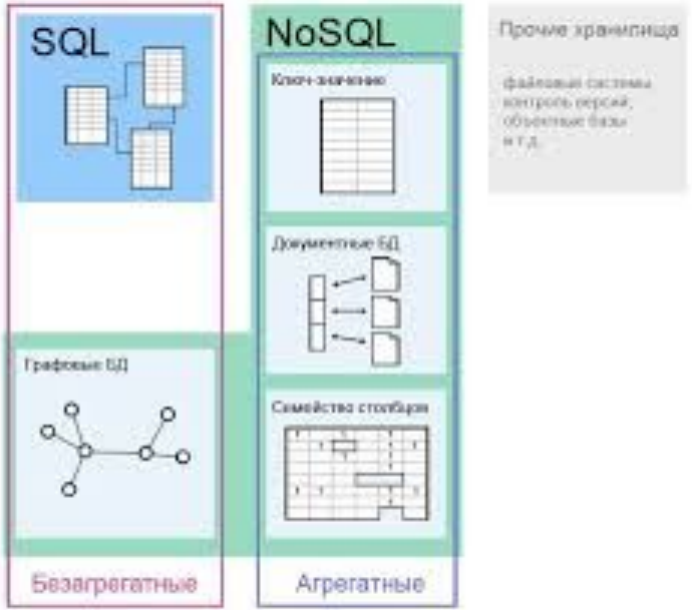
1. Работа с БД присутствует практически в любом приложении
2. Работа с аналитическими приложениями и с высоконагруженными системами требует разных подходов к построению БД

# Типы БД

1. Какие базы данных Вы уже знаете?

# Что такое база данных?

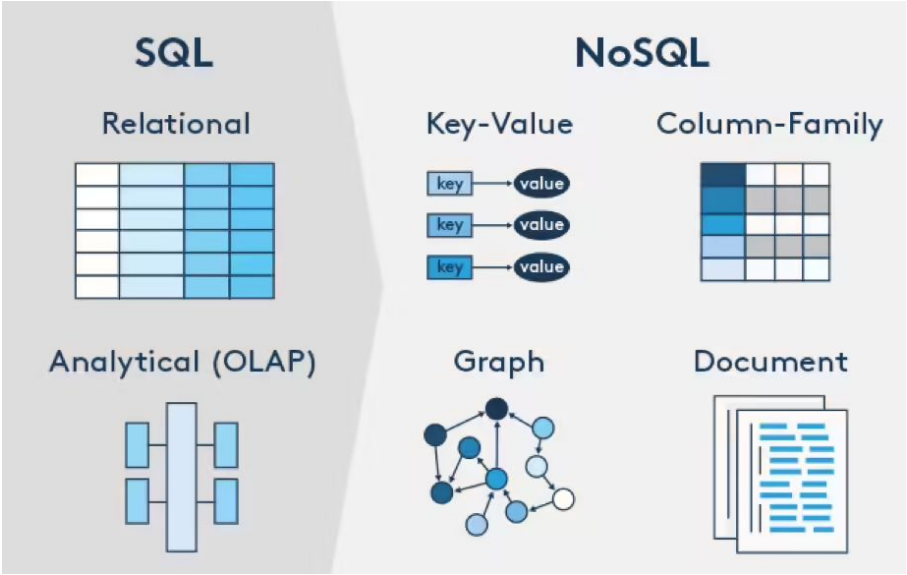
- Структурированный способ хранения и извлечения данных.
- Позволяет эффективно управлять большими наборами данных.
- Основная часть почти каждого приложения.



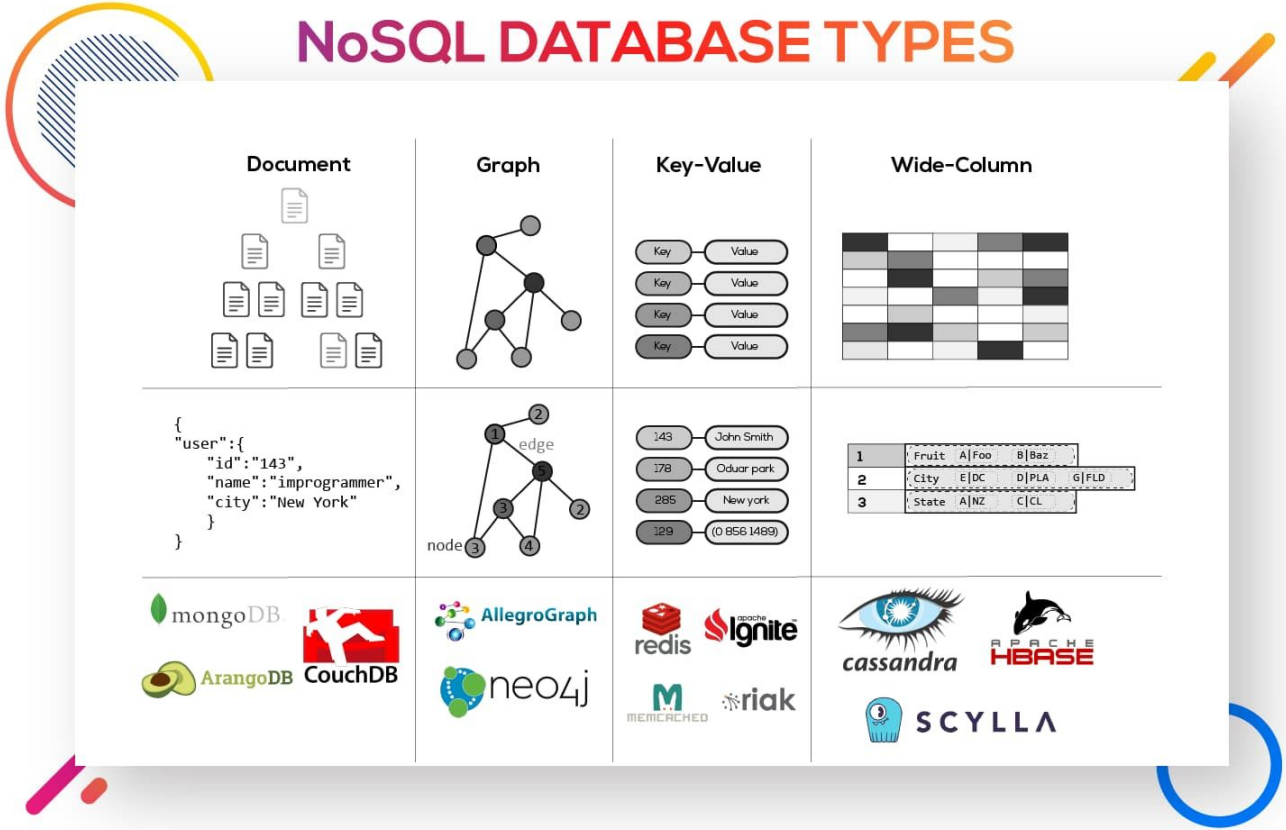
# Типы баз данных

Базы данных SQL: Структурированные, реляционные базы данных (например, MySQL, PostgreSQL).

Базы данных NoSQL: Нереляционные, гибкие схемы (например, MongoDB, Cassandra).



# Типы баз данных



# Ключевые различия между SQL и NoSQL

- **Схема:**

  SQL: фиксированная схема с таблицами и связями.

  NoSQL: гибкая схема, более адаптивная.

- **Целостность данных:**

  SQL: строгое соблюдение связей и целостности данных.

  NoSQL: отдает приоритет гибкости, а не строгим связям.

- **Масштабируемость:**

  SQL: вертикальное масштабирование (добавление большей мощности серверам).

  NoSQL: горизонтальное масштабирование (распределение по нескольким серверам).

# SQL или NoSQL – что выбрать?

- Нет решения на все случаи жизни
- Как правило, в средних и больших продуктах используются одновременно как SQL базы данных, так и NoSQL, для разных целей

**Когда предпочитаем SQL:**

- Структура данных неизменна

- Нужны гарантии ACID

**Когда предпочитаем NoSQL:**

- Большие объемы неструктурированных или слабоструктурированных данных

- Планируем задействовать облачные технологии

- Требуется быстрая разработка (стартап, proof of concept, MVP)

- Используемые структуры данных больше подходят под NoSQL

# Примеры использования SQL и NoSQL

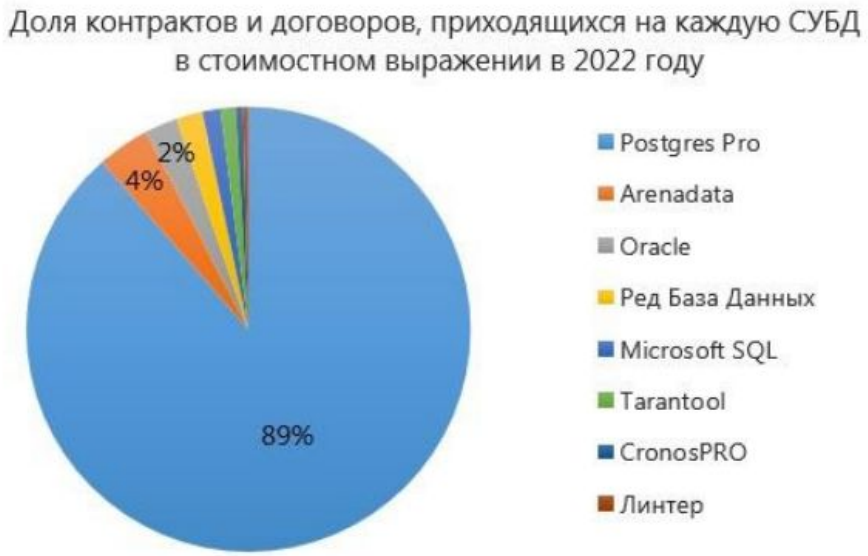
SQL:

- Платформы электронной коммерции для инвентаризации, данных клиентов.
- Банковские приложения для транзакций и информации о клиентах.

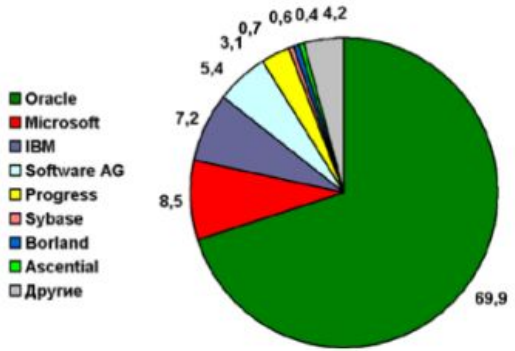
NoSQL:

- Социальные сети для взаимодействия с пользователем, сообщений.
- Данные IoT для аналитики в реальном времени и мониторинга устройств.

# Положение на рынке




2022




Доли рынка различных разработчиков СУБД в России

2011

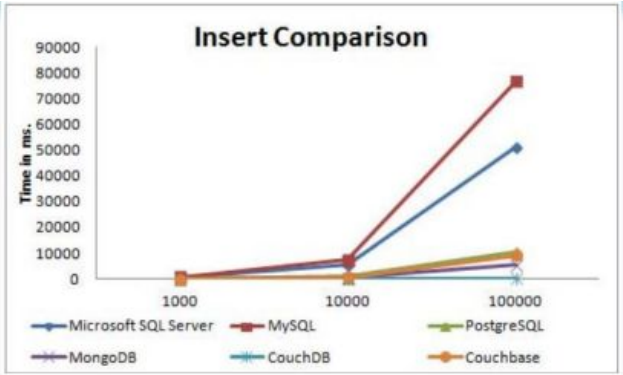
**The most popular database management systems**

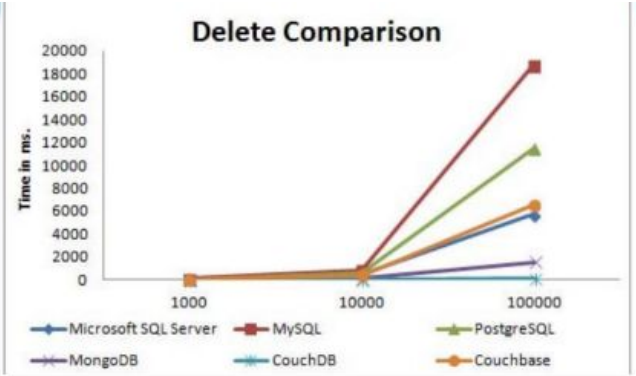
| August 2023 | | Score |
|---|---|---|
| 1. | Oracle | 1242 |
| 2. | MySQL | 1130 |
| 3. | Microsoft SQL Server | 921 |
| 4. | PostgreSQL | 620 |
| 5. | MongoDB | 434 |

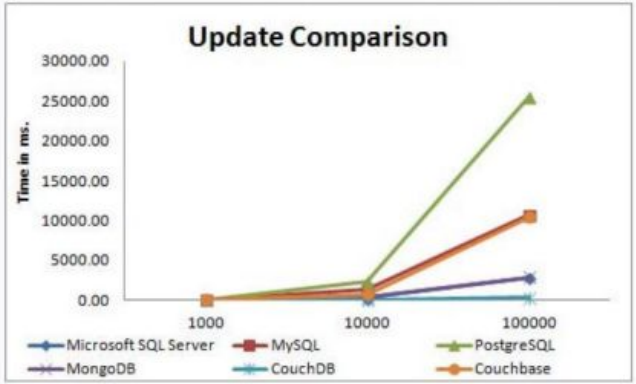
» more

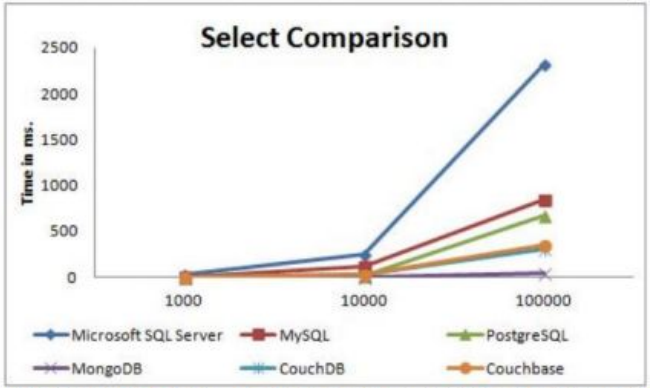
https://www.tadviser.ru/index.php/Статья:СУБД_%28рынок_России%29

# Бенчмарки CRUD-операций



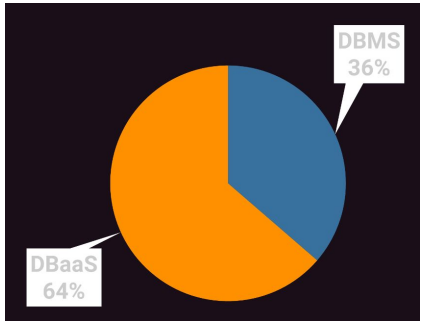



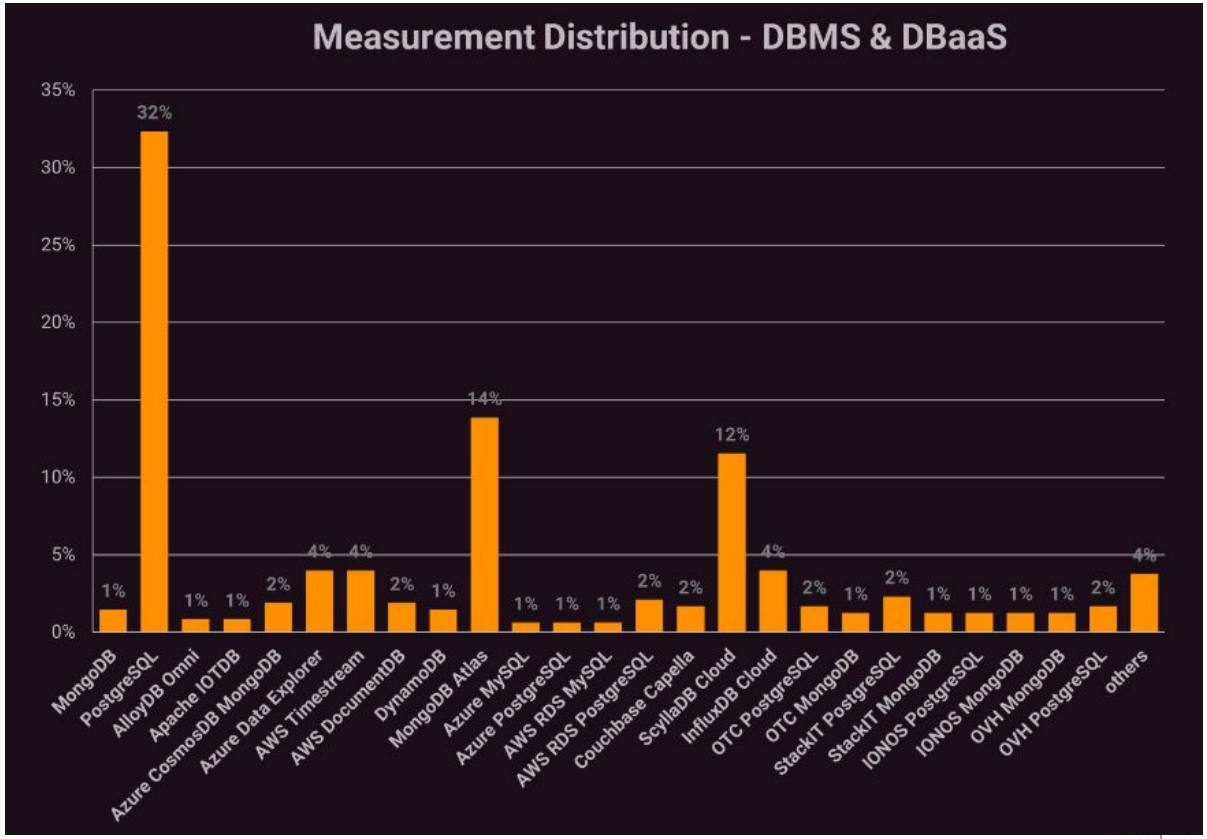



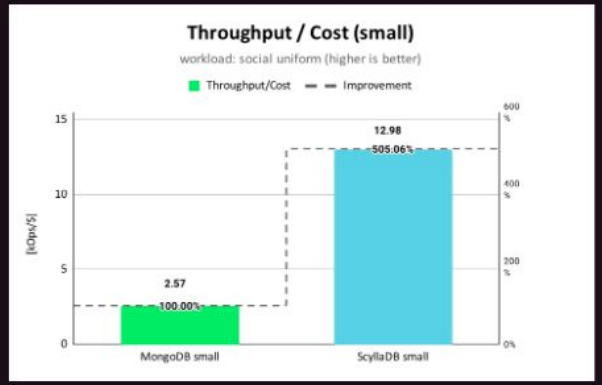

Truică, Ciprian-Octavian & Rădulescu, Florin & Boicea, Alexandru & Bucur, Ion. (2015). Performance Evaluation for CRUD Operations in Asynchronously Replicated Document Oriented Database. 10.1109/CSCS.2015.32.

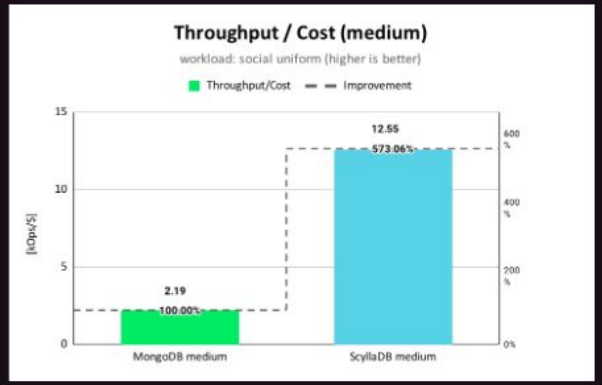
# В мире на 2023 год

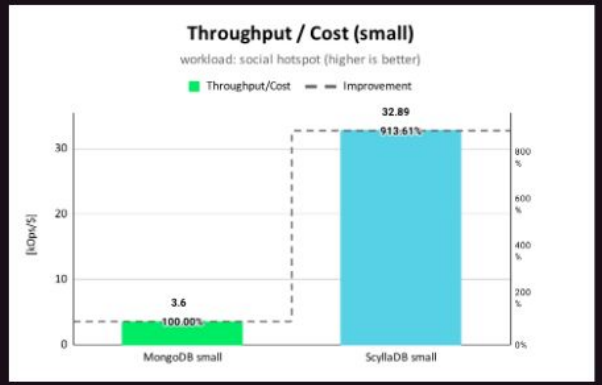


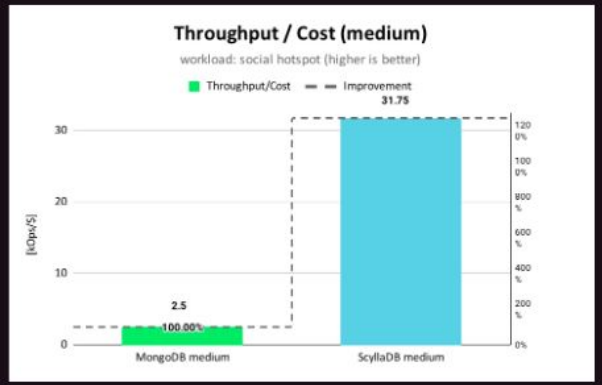



# Сравнение MongDb Scylla DB



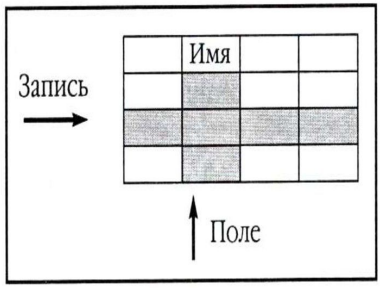
# Базы данных SQL – структура и принципы ACID

1. Что нужно для обеспечения целостности данных?

# Что такое базы данных SQL?

Основные моменты:

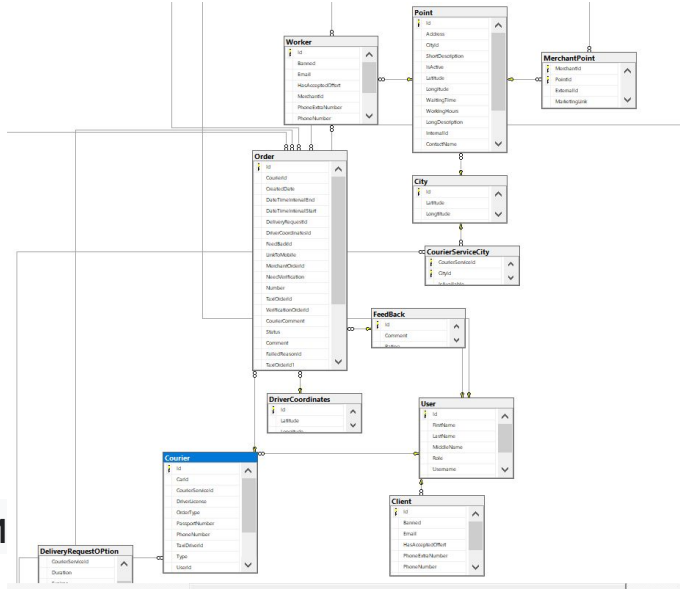
- Системы управления реляционными базами данных (СУБД).
- Используйте структурированные таблицы для хранения данных.
- Поддержка сложных запросов с использованием SQL (язык структурированных запросов).
- Распространенные примеры: MySQL, PostgreSQL, SQL Server.



# Структура баз данных SQL

Таблицы: Основная единица хранения данных, состоящая из строк (записей) и столбцов (полей).
Связи:

- Один к одному
- Один ко многим
- Многие ко многим




Первичные и внешние ключи: Обеспечивают связи и целостность данных.
Соединения (JOIN ): Объединяют данные из нескольких таблиц.

# Пример

1. Кто знает что такое ACID?

# Что такое ACID?

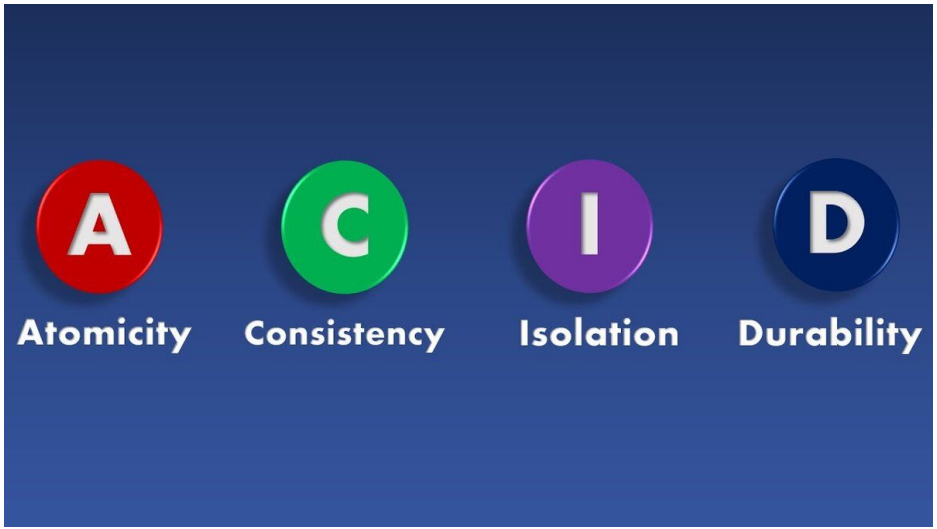
Набор свойств, обеспечивающих надежность транзакций.

Атомарность: Транзакции — это все или ничего.

Согласованность: Транзакции приводят к допустимому состоянию базы данных.

Изоляция: Транзакции не мешают друг другу.

Долговечность: Зафиксированные транзакции постоянны.



# ACID: Atomicity

Атомарность

Вся транзакция выполняется либо полностью, либо не выполняется совсем

```
BEGIN TRAN T1;
    UPDATE table1 ...;
    UPDATE table2 ...;
    SELECT * from table1;
    UPDATE table3 ...;
COMMIT TRAN T1;
```

# ACID: Consistency

Согласованность

Данные в БД всегда должны быть согласованными.

Добавлено в ACID в основном для красоты аббревиатуры.

Любая СУБД дает лишь незначительные гарантии согласованности (ForeignKey, Constraints)

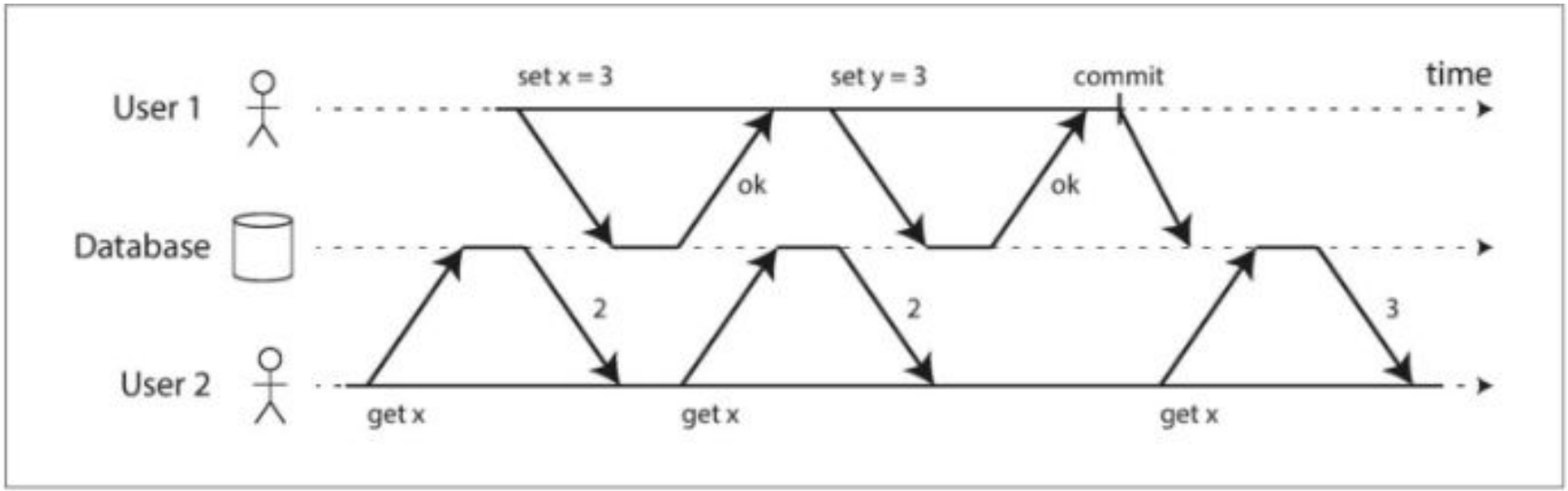
В основном согласованность зависит от кода приложения

# Миграции

- Изменение структуры существующей БД без потери консистентности данных varchar code -> int code
- В некоторых случаях возможен не только апгрейд, но и даунгрейд

# ACID: Isolation

Конкурентные транзакции



# Зачем нужно блокировать данные

Блокировка — это метод ограничения доступа к данным для обеспечения корректной обработки транзакций. Серверы баз данных используют блокировки, чтобы управлять одновременным доступом к данным, чтобы пока одна транзакция работает с данными, другие транзакции не могли их изменять.

# «Грязное чтение»

**«Грязное» чтение» (dirty read)** — это такое чтение, при котором могут быть считаны добавленные или изменённые данные из другой транзакции, которая впоследствии откатится;

| Транзакция 1 | Транзакция 2 |
| --- | --- |
| `SELECT age FROM users WHERE id = 1;` | |
| | `UPDATE users SET age = 21 WHERE id = 1;` |
| `SELECT age FROM users WHERE id = 1;` | |
| | `ROLLBACK` |

# Неповторяющееся чтение

**Неповторяющееся чтение (non-repeatable read)** — проявляется, когда при повторном чтении в рамках одной транзакции, ранее прочитанные данные, оказываются изменёнными;

| Транзакция 1 | Транзакция 2 |
| --- | --- |
| `SELECT * FROM users WHERE id = 1;` | |
| | `UPDATE users SET age = 21 WHERE id = 1; COMMIT;` |
| `SELECT age FROM users WHERE id = 1;` | |

# Фантомное чтение

можно наблюдать, когда одна транзакция в ходе своего выполнения несколько раз выбирает множество строк по одним и тем же критериям. При этом другая транзакция в интервалах между этими выборками добавляет или удаляет строки, или изменяет столбцы некоторых строк, используемых в критериях выборки первой транзакции, и успешно заканчивается. В результате получится, что одни и те же выборки в первой транзакции дают разные множества строк.

| Транзакция 1 | Транзакция 2 |
| --- | --- |
| `SELECT * FROM users`<br>`WHERE age > 10;` | |
| | `INSERT INTO users (id, name, age)`<br>`VALUES (1, 'Alice', 18);` |
| `SELECT * FROM users`<br>`WHERE age > 10;` | |

# Аномалия сериализации

**Аномалия сериализации (serialization anomaly)** — результат совместного выполнения нескольких транзакций не сводится к выполнению этих транзакций по одной ни в каком порядке;

```
INSERT INTO users (id, name, age) VALUES (1, 'Alice', 18);
INSERT INTO users (id, name, age) VALUES (2, 'Bob',   9);
```

| Транзакция 1 | Транзакция 2 |
| --- | --- |
| `INSERT INTO users (id, name, age)`<br>`SELECT 3, 'Craig', sum(age)`<br>`FROM users` | |
| | `INSERT INTO users (id, name, age)`<br>`SELECT 4, 'David', sum(age)`<br>`FROM users` |
| `COMMIT;` | |
| | `COMMIT;` |

# Потерянное обновление (lost update)

Потерянное обновление (lost update) — две транзакции выполняют одновременно UPDATE для одной и той же строки, и изменения, сделанные одной транзакцией, затираются другой;

# Блокировки

Пессимистические:
-На уровне таблицы LOCK TABLE films IN SHARE MODE;
-На уровне строки SELECT * FROM person WHERE name = 'John’ AND money = 1 FOR UPDATE;

Оптимистические:
-Столбец с версией данных UPDATE accounts SET balance = 200, version = 2 WHERE userid = 1 AND version = 1;

# Уровни изоляции транзакций

| Уровень изоляции | Грязное чтение | Неповторяющееся чтение | Фантомное чтение | Аномалия сериализации |
|---|---|---|---|---|
| Read uncommitted | Возможно, но не в PostgreSQL | Возможно | Возможно | Возможно |
| Read committed | Невозможно | Возможно | Возможно | Возможно |
| Repeatable read | Невозможно | Невозможно | Возможно, но не в PostgreSQL | Возможно |
| Serializable | Невозможно | Невозможно | Невозможно | Невозможно |

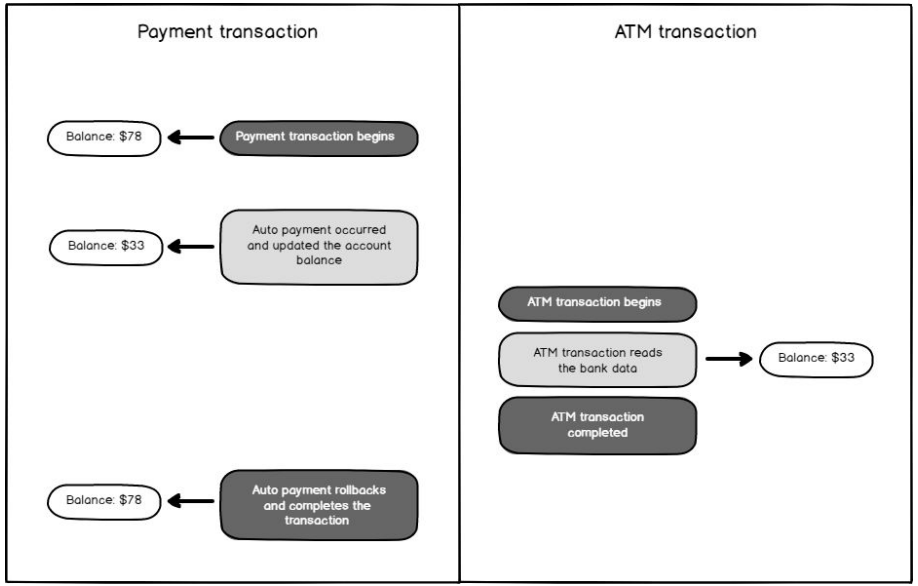
# ACID: Isolation

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION;
SELECT * FROM HumanResources.EmployeePayHistory;
SELECT * FROM HumanResources.Department;
COMMIT TRANSACTION;
```

# Read Uncommitted

Уровень изоляции READ UNCOMMITTED предоставляет самую простую форму изоляции между транзакциями, поскольку он вообще не изолирует операции чтения других транзакций.

Когда транзакция выбирает строку при этом уровне изоляции, она не задает никаких блокировок и не признает никаких существующих блокировок.
Считываемые такой транзакцией данные могут быть несогласованными. В таком случае транзакция читает данные, которые были обновлены какой-либо другой активной транзакцией. А если для этой другой транзакции позже выполняется откат, то значит, что первая транзакция прочитала данные, которые никогда по-настоящему не существовали.



# Read Uncommitted

Из четырех проблем одновременного конкурентного доступа к данным уровень изоляции READ UNCOMMITTED допускает три:

- грязное чтение,
- неповторяемое чтение
- фантомы.

Применение уровня изоляции READ UNCOMMITTED обычно крайне нежелательно, и его следует применять только в тех случаях, когда точность данных не представляет важности, или когда данные редко подвергаются изменениям.

В PostgreSQL внутренне не отличается от READ COMMITTED.

# Read Committed

Транзакция, которая читает строку и использует уровень изоляции READ COMMITTED, выполнит проверку только на наличие монопольной блокировки для данной строки. Если такая блокировка отсутствует, транзакция извлекает строку.

Это выполняется с использованием разделяемой блокировки.

Таким образом предотвращается чтение транзакцией данных, которые не были подтверждены и которые могут быть позже отменены.
После того, как данные были прочитаны, их можно изменять другими транзакциями.

Применяемые этим уровнем изоляции разделяемые блокировки отменяются сразу же после обработки данных (обычно все блокировки отменяются в конце транзакции).
Это улучшает параллельный одновременный конкурентный доступ к данным, но возможность неповторяемого чтения и фантомов продолжает существовать.

Уровень изоляции READ COMMITTED является уровнем изоляции по умолчанию.

# Repeatable Read

В отличие от уровня изоляции READ COMMITTED, уровень REPEATABLE READ устанавливает разделяемые блокировки на все считываемые данные и удерживает эти блокировки до тех пор, пока транзакция не будет подтверждена или отменена.
Поэтому в этом случае многократное выполнение запроса внутри транзакции всегда будет возвращать один и тот же результат.

Недостатком этого уровня изоляции является дальнейшее ухудшение одновременного конкурентного доступа, поскольку период времени, в течение которого другие транзакции не могут обновлять те же самые данные, значительно дольше, чем в случае уровня READ COMMITTED.

Этот уровень изоляции не препятствует другим инструкциям вставлять новые строки, которые включаются в последующие операции чтения, вследствие чего могут появляться фантомы.

# Serializable

Уровень изоляции SERIALIZABLE является самым строгим, потому что он не допускает возникновения всех четырех проблем параллельного одновременного конкурентного доступа, перечисленных ранее. Этот уровень устанавливает блокировку на всю область данных, считываемых соответствующей транзакцией. Поэтому этот уровень изоляции также предотвращает вставку новых строк другой транзакцией до тех пор, пока первая транзакция не будет подтверждена или отменена.

Уровень изоляции SERIALIZABLE реализуется, используя метод блокировки диапазона ключа. Суть этого метода заключается в блокировке отдельных строк включительно со всем диапазоном строк между ними. Блокировка диапазона ключа блокирует элементы индексов, а не определенные страницы или всю таблицу. В этом случае любые операции модификации другой транзакцией невозможны, вследствие невозможности выполнения требуемых изменений элементов индекса

# Явные блокировки в PostgreSQL

Помимо уровней изоляции транзакций, есть дополнительные настройки конкурентности.
Например:
SELECT … FOR UPDATE — выбранные записи блокируются так же, как и для изменения;
SELECT … FOR UPDATE SKIP LOCKED — то же самое, плюс записи, заблокированные кем-то ещё, пропускаются — актуально для уровней изоляции repeatable read и serializable;
LOCK TABLE … IN ACCESS EXCLUSIVE MODE — полная блокировка таблицы, никакая другая транзакция не сможет обратиться к указанной таблице каким-либо способом;
INSERT … ON CONFLICT DO UPDATE — если запись с таким же уникальным ключом уже существует, то обновить её;
INSERT … ON CONFLICT DO NOTHING — если запись с таким же уникальным ключом уже существует, то пропустить действие;

Полный список можно почитать в документации –
https://www.postgresql.org/docs/current/explicit-locking.html

# Рекомендую почитать

- Описание проблем, с которыми можно столкнуться при разработке высоконагруженных проектов
- Их решения
- Без привязки к языку программирования
- Очень хорошо читается

# DeadLock(взаимоблокировка)

**Взаимоблокировка (deadlock)** — это ситуация, при которой одному процессу для продолжения работы требуется ресурс, захваченный вторым процессом, а второму процессу требуется ресурс, захваченный первым процессом. В такой ситуации оба процесса оказываются в заблокированном состоянии и не могут продолжать работу.

*Также блокировку может вызвать набор больше чем из двух транзакций.*

# DeadLock пример

1. Транзакция 1 читает запись 1.
2. Транзакция 2 читает запись 2.
3. Транзакция 1 пытается изменить запись 2 и ждет, когда транзакция 2 закончится и отпустит свою блокировку.
4. Транзакция 2 пытается изменить запись 1 и ждет, когда транзакция 1 закончится и отпустит свою блокировку

— 1-я транзакция

```
BEGIN TRANSACTION;
SELECT * FROM public.Students WHERE Id = 1 FOR UPDATE;
SELECT pg_sleep(0.5);
UPDATE public.Students SET [Name] = 'test2' WHERE Id = 2;
COMMIT;
```

— 2-я транзакция

```
BEGIN TRANSACTION;
SELECT * FROM public.Students WHERE Id = 2 FOR UPDATE;
SELECT pg_sleep(0.5);
UPDATE public.Students SET [Name] = 'test1' WHERE Id = 1;
COMMIT;
```

# DeadLock пример

**Решение:**

PostgreSQL с помощью встроенного менеджера блокировок определяет взаимные блокировки и разрешает их. Разрешает очень просто — жертвует одной из транзакций, т. е. попросту откатывает ее и возвращает ошибку. Остальные транзакции продолжают выполняться. Какая транзакция будет выбрана в качестве жертвы — определяет сам PostgreSQL.

см. https://www.postgresql.org/docs/current/explicit-locking.html#LOCKING-DEADLOCKS

*В MS SQL дополнительно есть возможность указания DEADLOCK_PRIORITY.*

# ACID: Durability

Долговечность, стойкость

- Надежность даже в случае сбоев БД и/или аппаратного обеспечения
- Примеры решений: журнал упреждающей записи, репликации
- WAL (Write-Ahead Log) – WAL ведет учет транзакций и в случае сбоя данные лога перечитываются для восполнения консистентности данных

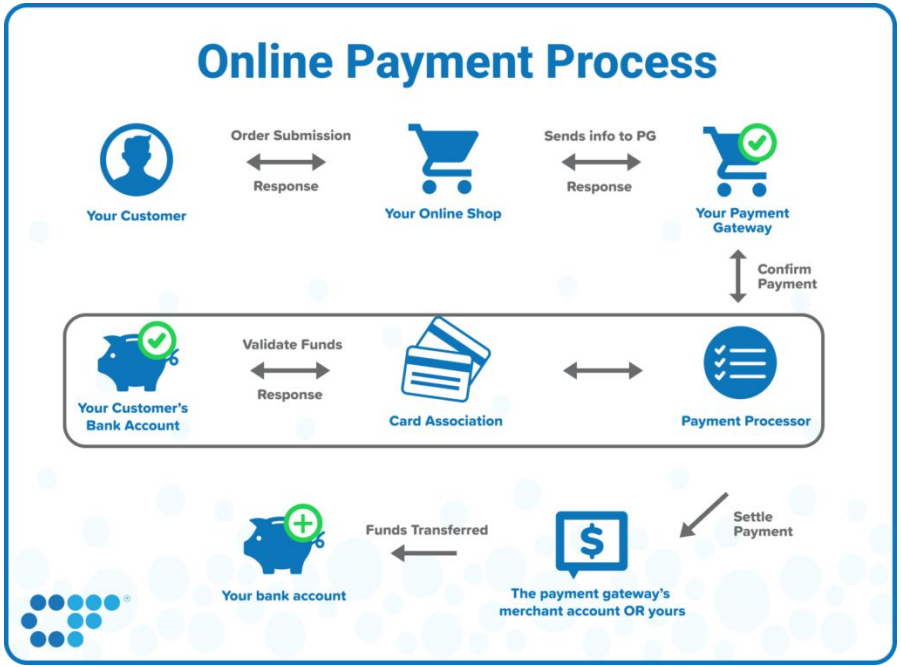
# ACID в действии — пример

Атомарность: обновление статуса заказа и платежа клиента в одной транзакции.

Последовательность: обеспечение соответствия суммы платежа общей сумме заказа.

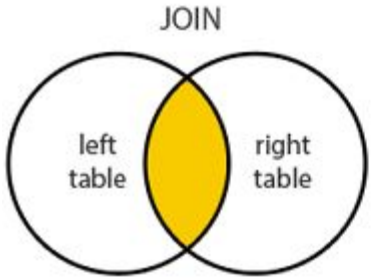
Изоляция: два клиента, размещающие заказы одновременно, не мешают друг другу.

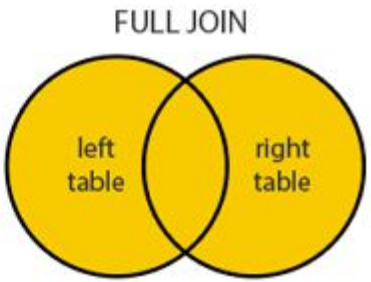
Долговечность: заказ остается обновленным даже после сбоя системы.

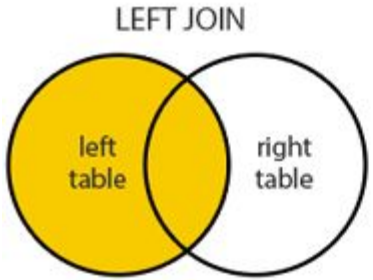


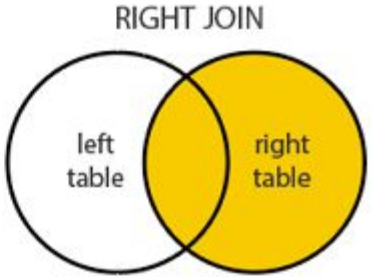
# Базы данных SQL — основные выводы

- Базы данных SQL используют структурированные таблицы для хранения данных.
- Реляционные модели обеспечивают прочные связи данных.
- Соединения позволяют сложным запросам объединять данные из нескольких таблиц.
- Принципы ACID гарантируют надежность и последовательность транзакций.



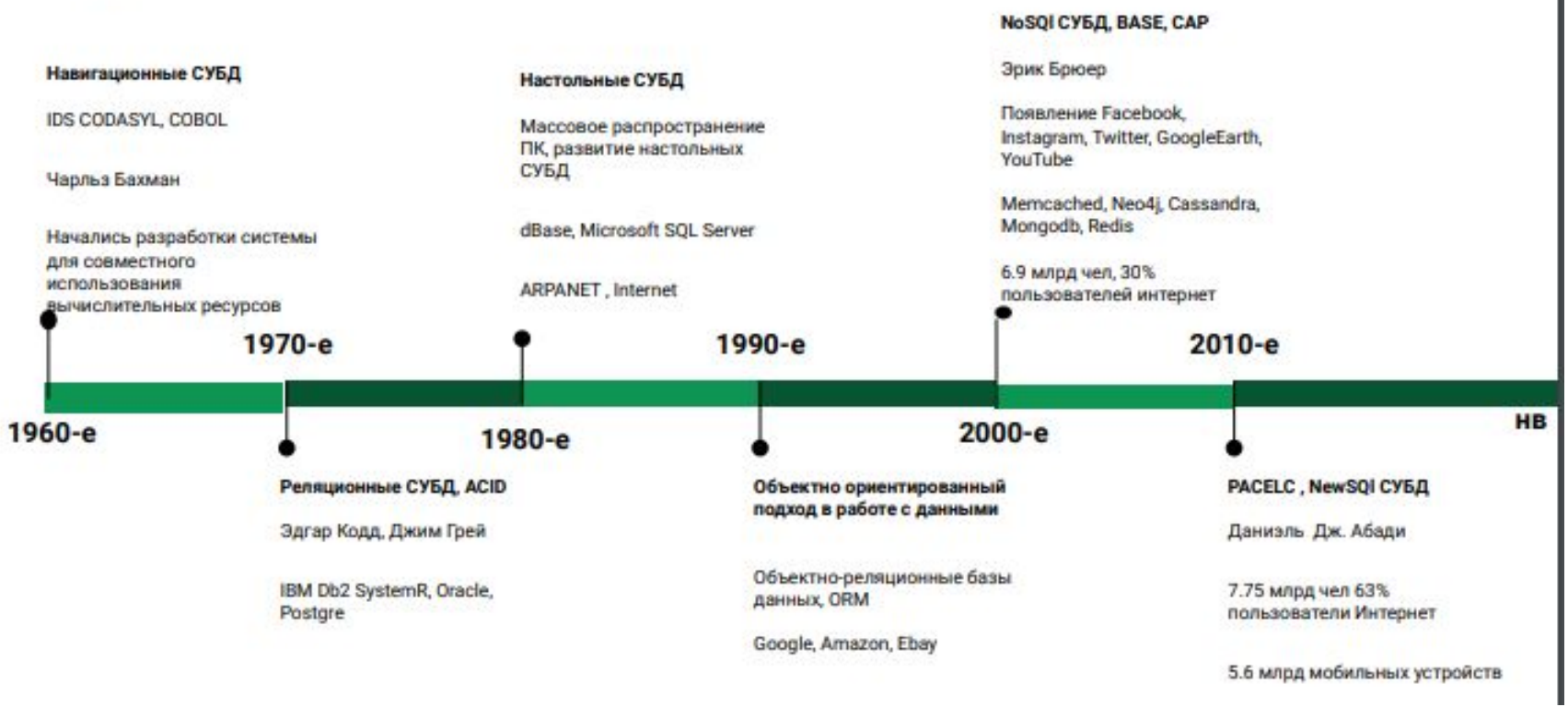



# Базы данных NoSQL – структура и варианты использования

# От реляционных СУБД к NoSQL СУБД

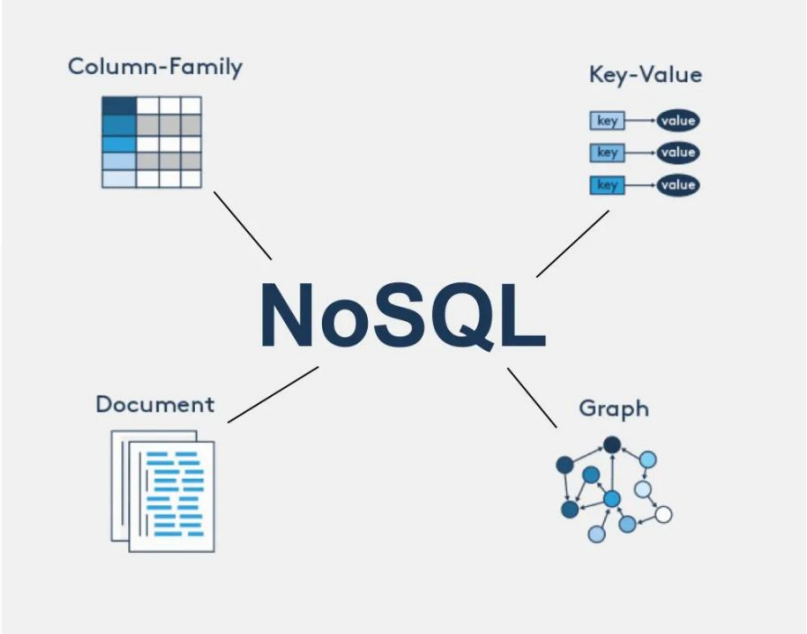


# Что такое базы данных NoSQL?

1. Предназначены для неструктурированных или полуструктурированных данных.
2. Без схемы: Нет фиксированной структуры таблиц.
3. Основное внимание уделяется горизонтальной масштабируемости.
4. Распространенные примеры: MongoDB, Cassandra, Redis.

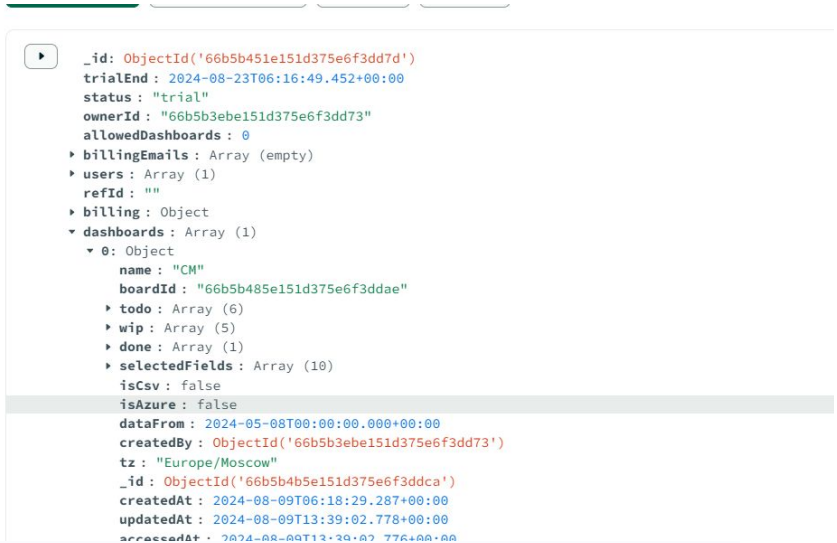
# Типы баз данных NoSQL

- Хранилища документов (например, MongoDB): хранят данные в виде документов, подобных JSON.
- Хранилища «ключ-значение» (например, Redis): простые пары «ключ-значение» для быстрого поиска.
- Хранилища семейств столбцов (например, Cassandra): данные хранятся в столбцах, а не в строках.
- Графовые базы данных (например, Neo4j): фокусируются на связях между точками данных.



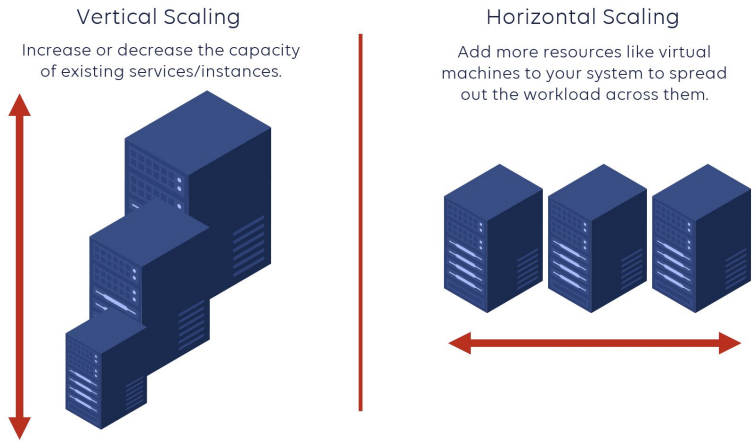
# Структура баз данных NoSQL

- Отсутствие строгой схемы: Гибкость в формате данных.
- Данные организованы на основе определенной модели NoSQL (документ, ключ-значение и т. д.).
- Горизонтальное масштабирование путем распределения данных по нескольким узлам.
- Денормализация для более быстрого доступа (хранение связанных данных вместе).



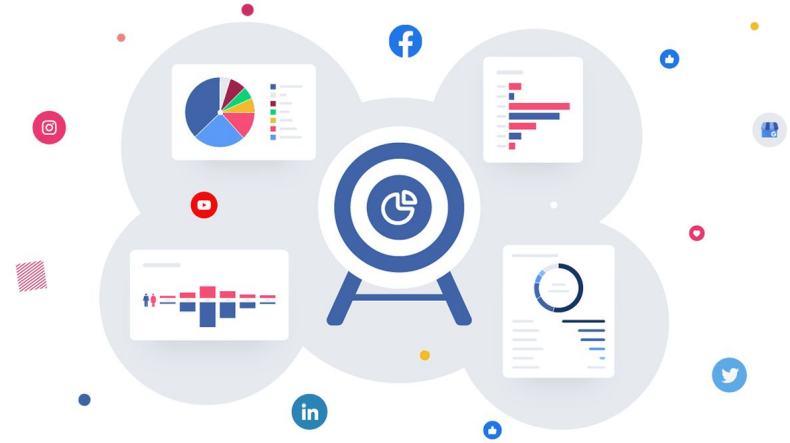
# Зачем использовать NoSQL?

- Гибкость: Может обрабатывать неструктурированные, динамические или изменяющиеся модели данных.
- Масштабируемость: Горизонтальное масштабирование делает его идеальным для больших наборов данных и распределенных систем.
- Высокая производительность: Отлично подходит для приложений с высокой пропускной способностью (например, аналитика в реальном времени, кэширование).
- Проектирование без схем: Снижает сложность в развивающихся приложениях.



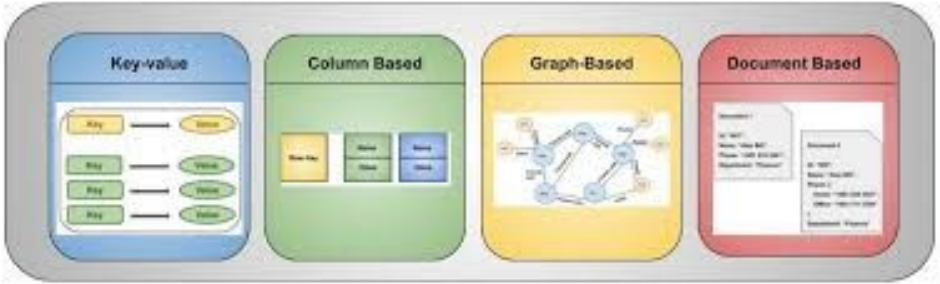
# Распространенные варианты использования NoSQL

- Аналитика больших данных в реальном времени (например, журналы, метрики): Cassandra.
- Системы управления контентом (например, блоги, медиаплатформы): MongoDB.
- Кэширование и хранение сеансов: Redis.
- Социальные сети и запросы на основе графов: Neo4j.



# Базы данных NoSQL — основные выводы

- Базы данных NoSQL обеспечивают гибкость и масштабируемость для неструктурированных данных.
- Типы включают хранилища документов, хранилища ключей и значений, хранилища семейств столбцов и графовые базы данных.
- Идеально подходят для приложений реального времени, аналитики больших данных и социальных сетей.
- Безсхемная конструкция делает NoSQL подходящим для развивающихся моделей данных.



# Сравнение SQL и NoSQL СУБД

| | SQL | NoSQL |
|---|---|---|
| СУБД | реляционная | не-реляционная или распределенная |
| Схема данных | Зафиксированная или статическая предопределенная | динамическая |
| Хранилище данных | БД не ориентирована на иерархическое, распределенное хранилище данных | БД ориентированы на иерархическое, распределенное хранилище данных |
| Масштабируемость | вертикальная | горизонтальная |
| Принципы построения | ACID | BASE |

# Шардинг, секционировани е и репликация

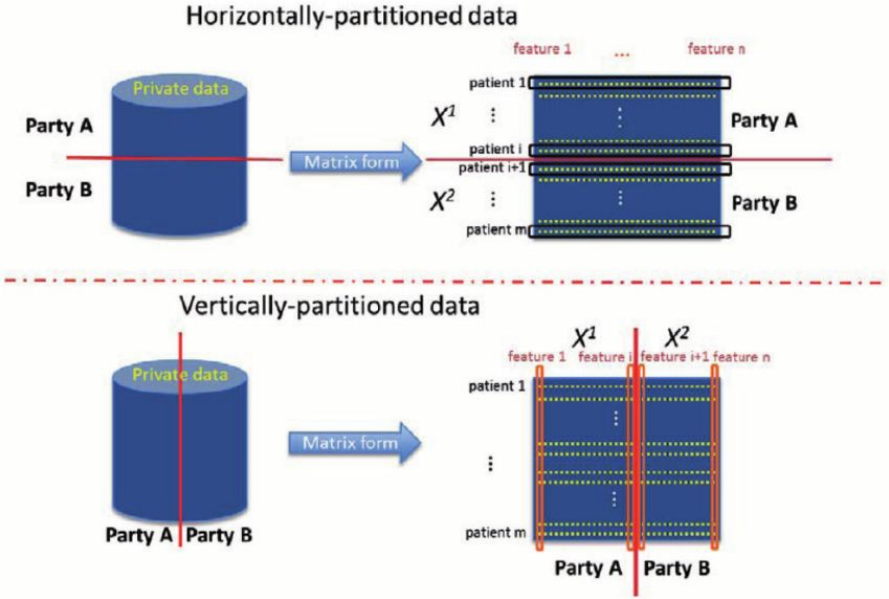
# Зачем масштабировать базы данных?

- Растущие наборы данных требуют больше ресурсов хранения и вычислительной мощности.
- Ограничения одного сервера (ЦП, память, диск).
- Распределенные базы данных обеспечивают высокую доступность и надежность.
- Стратегии масштабирования сокращают задержку и повышают производительность.

Используя такие методы, как сегментирование, разбиение на разделы и репликация, базы данных могут распределять рабочую нагрузку, гарантируя, что данные будут доступны даже во время сбоев сервера, одновременно сокращая задержку и повышая производительность запросов.

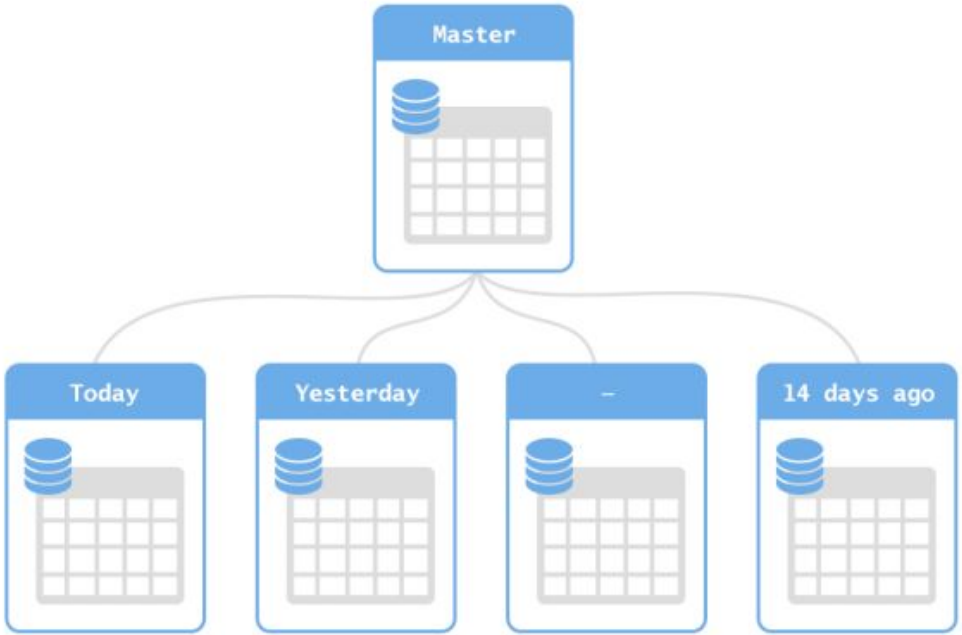
# Что такое секционирование?(Partitioning)

- Разбиение большого набора данных на более мелкие, управляемые части (партиции).
- Логическое или физическое разделение данных.
- Горизонтальное разделение: Разделение строк по таблицам.
- Вертикальное разделение: Разделение столбцов по таблицам.
- Улучшает производительность за счет изоляции часто используемых данных.




Разделение полезно для повышения производительности запросов за счет изоляции часто используемых данных и распределения их по разным серверам или местоположениям, что снижает нагрузку на любой отдельный узел.

# Что такое секционирование?(Partitioning)

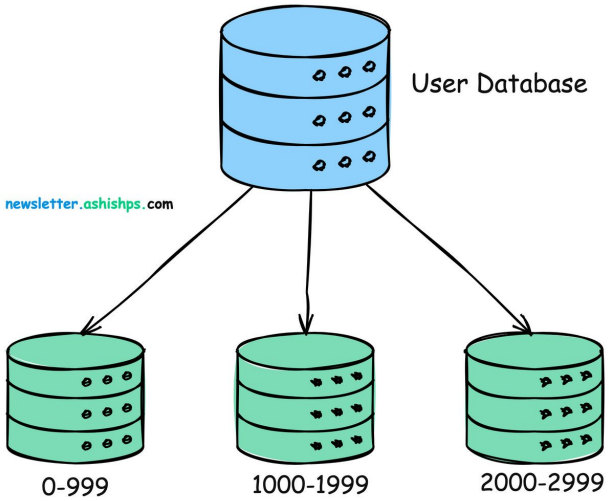



```
$ create table users (
    id              serial primary key,
    username        text not null unique,
    password        text,
    created_on      timestamptz not null,
    last_logged_on timestamptz not null
);
```

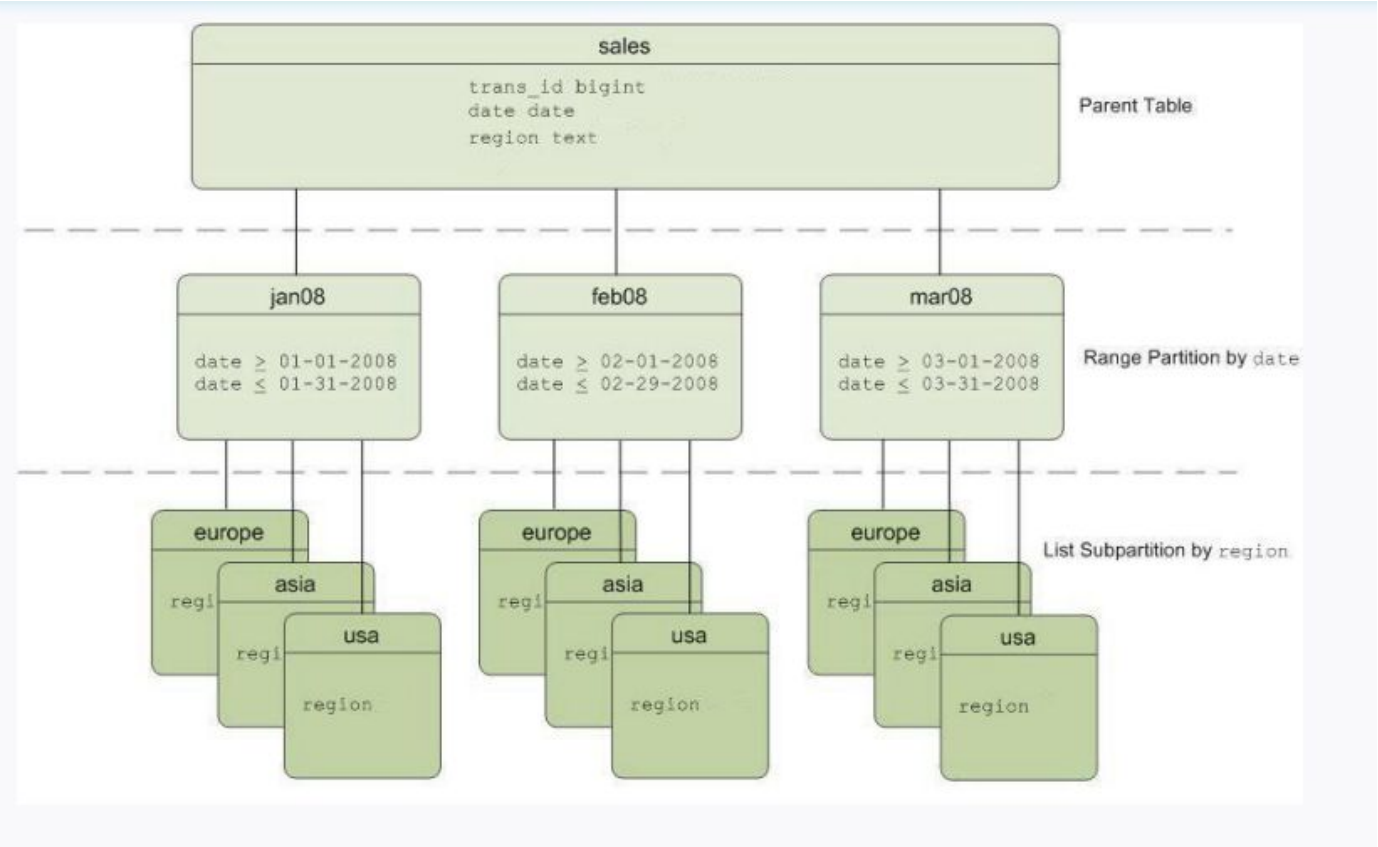
```
$ create table users_2 ( like users including all );
$ alter table users_2 inherit users;
...
$ create table users_10 ( like users including all );
$ alter table users_10 inherit users;
```

# Что такое шардинг?

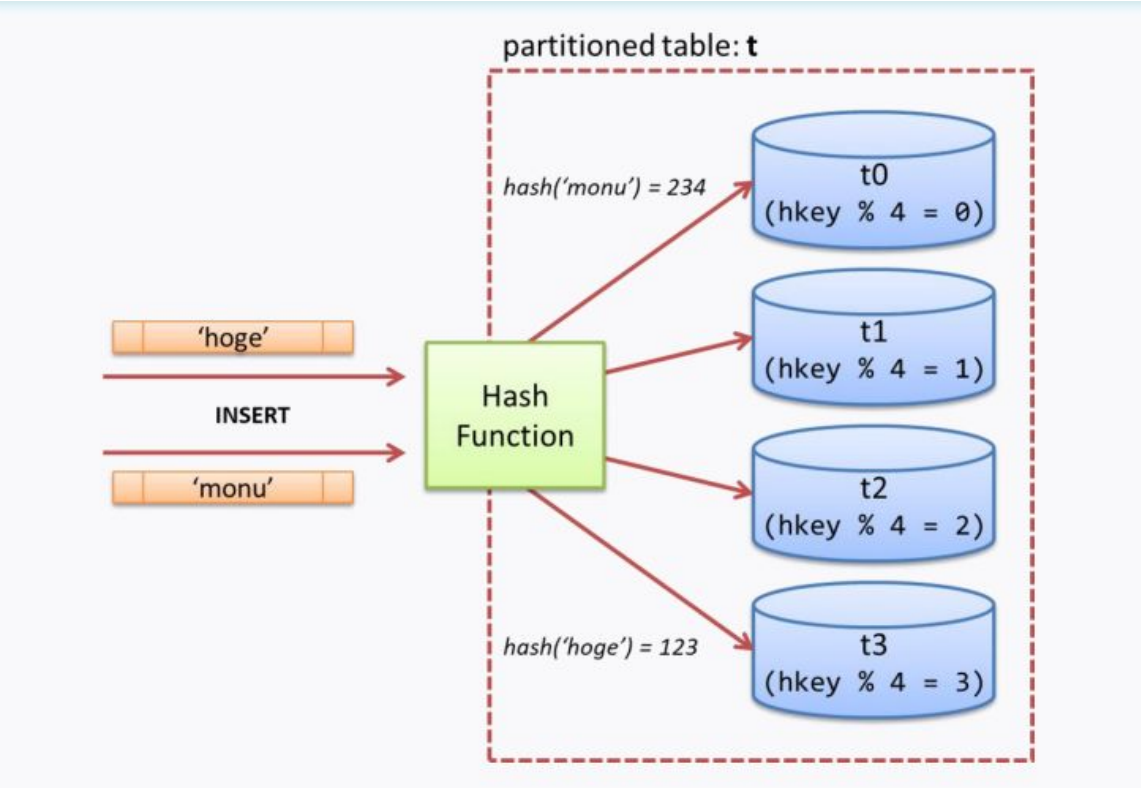
1. Особая форма горизонтального разделения.
2. Данные разделены на несколько шардов (серверов или баз данных).
3. Каждый шард содержит подмножество всех данных.
4. Часто используется в базах данных NoSQL, таких как MongoDB и Cassandra.
5. Шардинг обеспечивает масштабное горизонтальное масштабирование.



# Range и List Partitioning



# Hash Partitioning



# Процесс шардинга

Ключ шарда: Определяет, как данные разделяются по шардам.

Ключи шарда могут быть основаны на хэше или на диапазоне.

Данные распределяются на основе значения ключа шарда.

Запросы направляются в соответствующий шард на основе ключа.

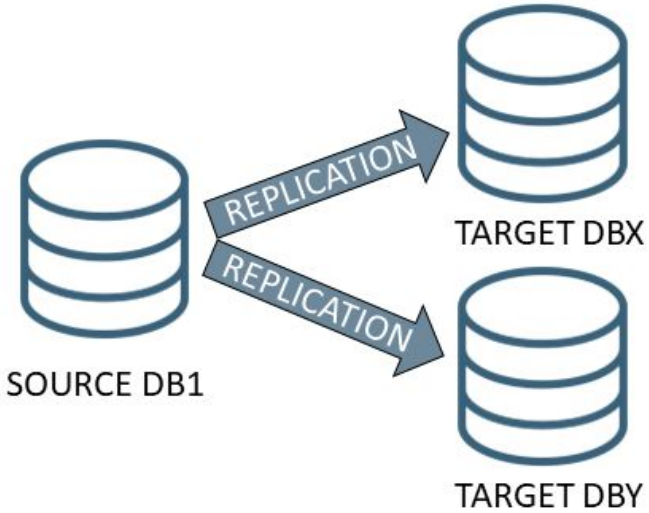
Локальность данных обеспечивает эффективное извлечение данных.

# Что такое репликация?

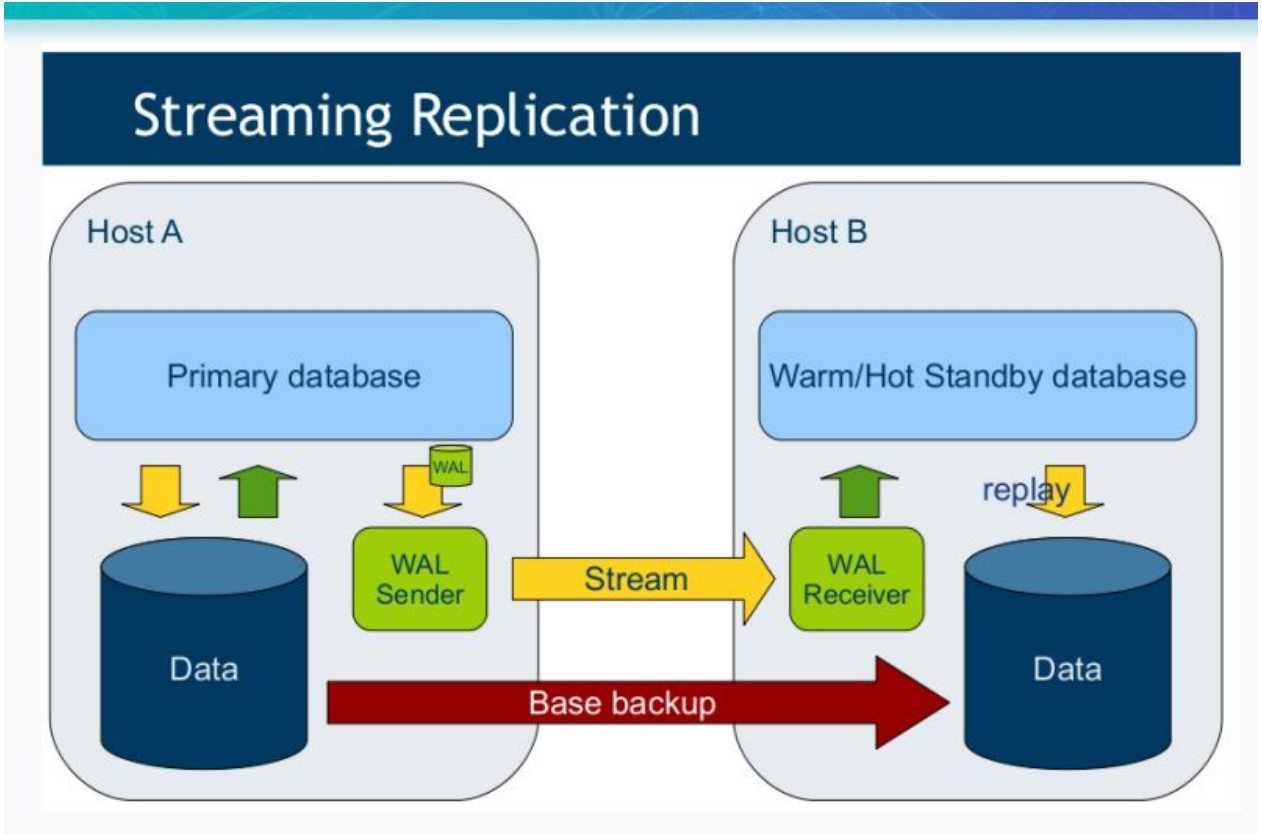
Дублирование данных на нескольких узлах.

Обеспечивает высокую доступность и отказоустойчивость.
Репликация Master-Slave: запись отправляется на главный узел, чтение — с реплик.
Репликация Multi-Master: запись может выполняться несколькими узлами.

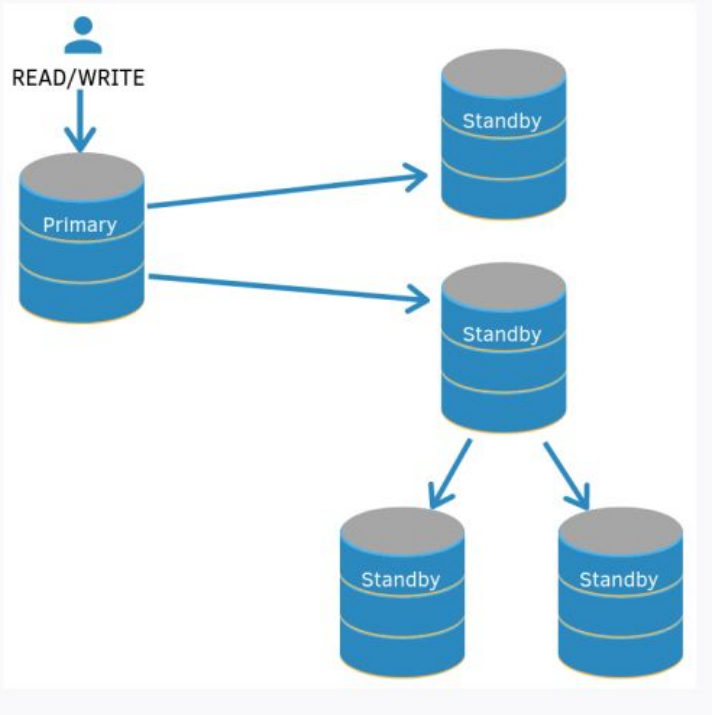
Повышает надежность данных в случае сбоев узлов.



# Ведущий-ведомый



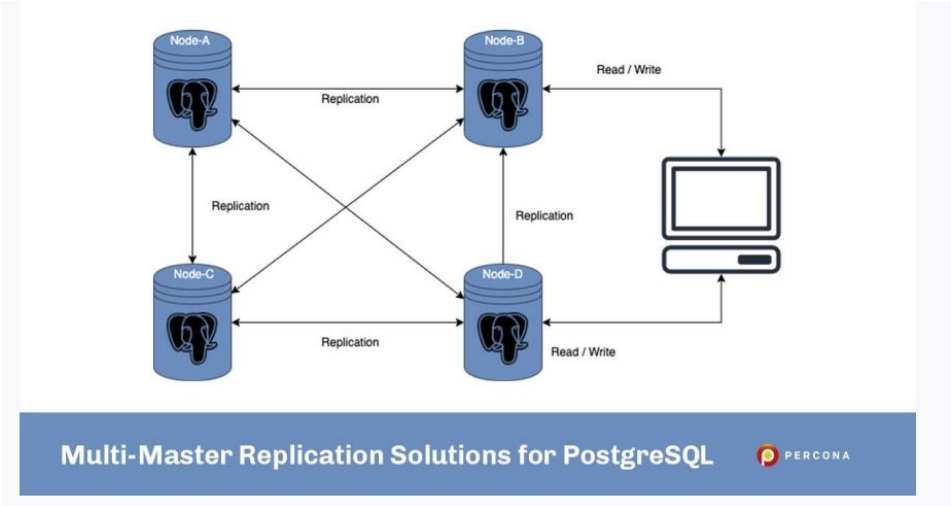
# Каскадная репликация



# Логическая Репликация

Поставщик публикует свои изменения, а подписчик получает и применяет эти изменения у себя.

Особенности:
• оба сервера могут быть и поставщиком и подписчиком, но на разные объекты;
• репликация возможна между разными ОС;
• возможна выборочная репликация отдельных объектов кластера.



# Примеры использования для шардинга, секционирования и репликации

- Шардинг: крупномасштабные распределенные системы, базы данных NoSQL (например, MongoDB, Cassandra).
- Разделение: базы данных SQL с большими таблицами или географическим распределением (например, MySQL, PostgreSQL).
- Репликация: системы высокой доступности, критически важные приложения (например, банковское дело, электронная коммерция).

# Ключевые выводы

- Шардинг предназначен для горизонтального масштабирования и распределения больших наборов данных по нескольким серверам.
- Разбиение разделяет данные на логические/физические части для повышения производительности.
- Репликация обеспечивает высокую доступность и отказоустойчивость за счет дублирования данных.
- Баланс масштабируемости, производительности и согласованности имеет важное значение при выборе стратегии.

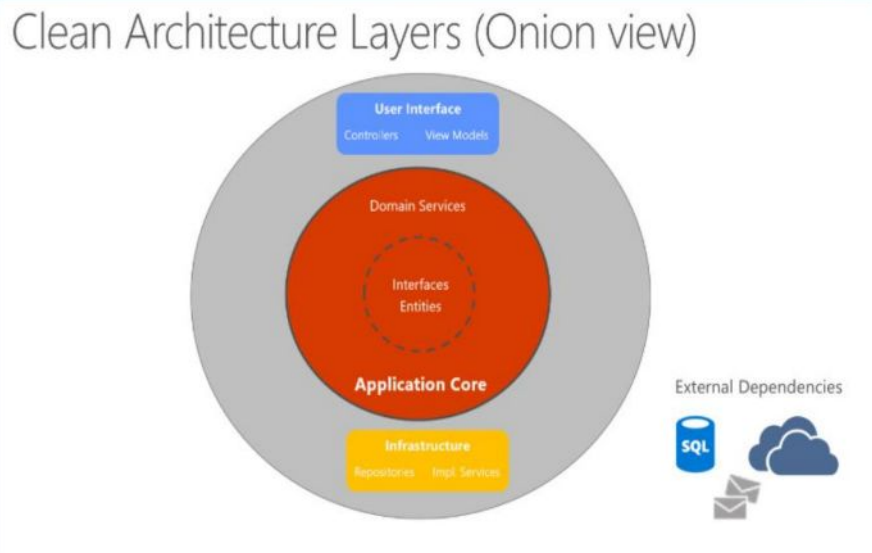
# ORM.
# Что это?

# ORM

ORM (Object-Relational Mapping): объектно-реляционное отображение — технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных».

ORM объединяет по своей сути различные парадигмы объектно-ориентированного программирования, где сущности представлены в виде классов и объектов, и реляционных баз данных, где данные хранятся в таблицах со строками и столбцами

# Какие модели маппит ORM

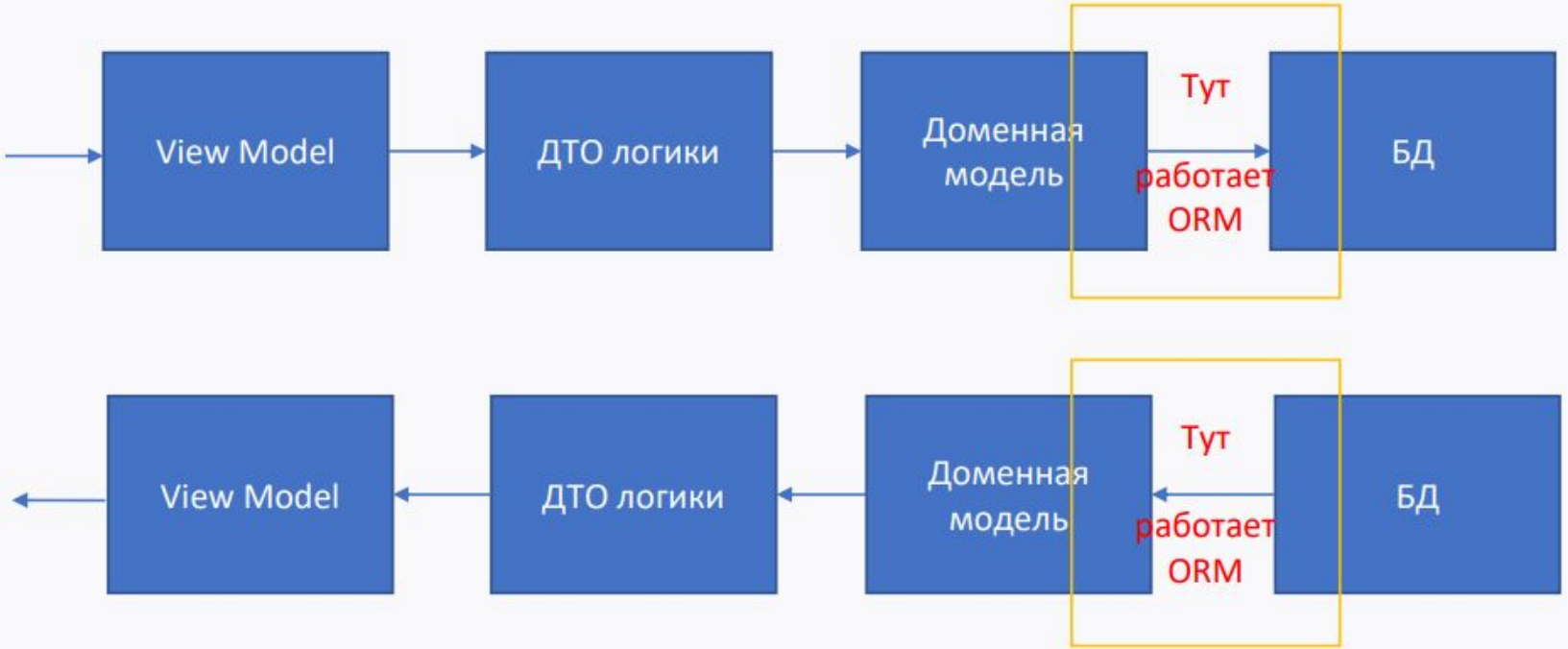



**Виды моделей**

1. Доменные модели (Entities), находятся в ядре
2. Модели логики (DTO), находятся в Domain Services
3. Модели для работы с клиентом (View Models), находятся в API или слое пользовательского интерфейса

**К БД мапим Entities**

# Какие модели маппит ORM



# Преимущества и недостатки ORM

## Преимущества

- **Простота**: ORM предоставляет простой интерфейс для работы с базой данных, который может быть понятным любому программисту. ORM скрывает сложности SQL-запросов, позволяя работать с данными на более высоком уровне абстракции.
- **Переносимость**: ORM может работать с различными СУДБ, что делает его более переносимым, чем SQL. Это позволяет разработчикам легко переносить свое приложение на другую СУБД без изменения кода.
- **Сопровождаемость**: ORM может значительно упростить сопровождение приложения, так как изменения в структуре базы данных могут быть внесены непосредственно в код ORM, а не в каждый SQL-запрос.
- **Безопасность**: ORM может предотвратить SQL-инъекции, поскольку ORM автоматически экранирует данные, которые передаются в базу данных.

# Преимущества и недостатки ORM

Недостатки

- **Сложность**: ORM может быть сложным для понимания, особенно для новых разработчиков. ORM требует определенных знаний и опыта, чтобы использовать его эффективно.
- **Производительность**: ORM может быть менее эффективным, чем работа с SQL напрямую. ORM должен обрабатывать запросы и преобразовывать их в SQL, что может замедлить производительность.
- **Ограничения**: ORM может иметь ограничения в отношении того, какие запросы могут быть выполнены. В случае, когда нужно выполнить сложный запрос или использовать специфичные функции базы данных, может потребоваться написание SQL-запроса напрямую.

# Вопросы?

Ставим “+”,
если вопросы есть

Ставим “–”,
если вопросов нет

# Рефлексия

# Рефлексия

С какими впечатлениями уходите с вебинара?

Как будете применять на практике то,
что узнали на вебинаре?