



SmartData
2024

Шардированный не значит распределенный: что важно знать, когда PostgreSQL мало

Евгений Иванов,
Ведущий разработчик YDB, Яндекс

Олег Бондарь,
СРО YDB, Яндекс

Олег Бондарь

Директор продукта
YDB, Яндекс



Евгений Иванов

Ведущий разработчик,
YDB, Яндекс



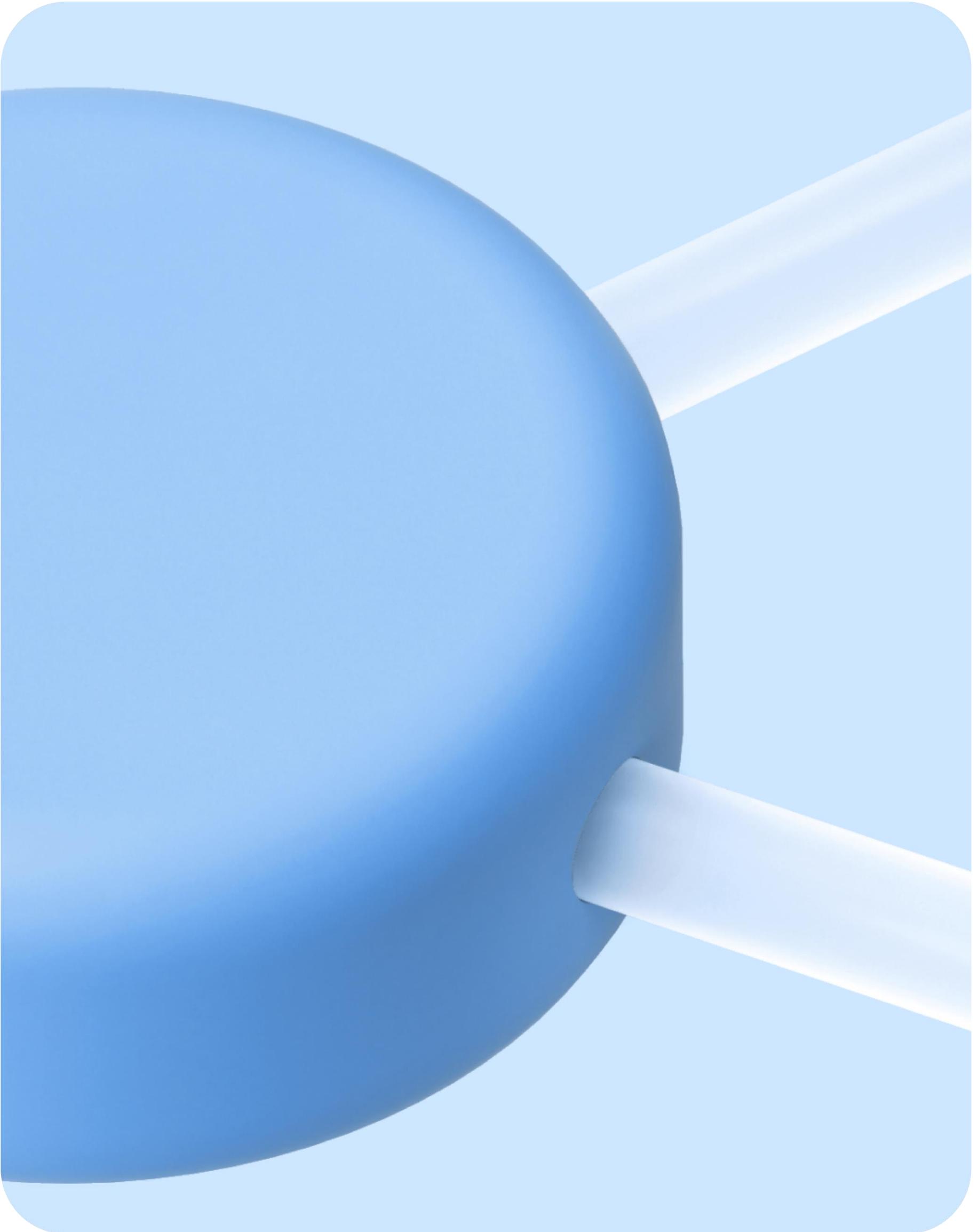
Евгений Ефимкин

Руководитель группы,
Managed PostgreSQL, Яндекс



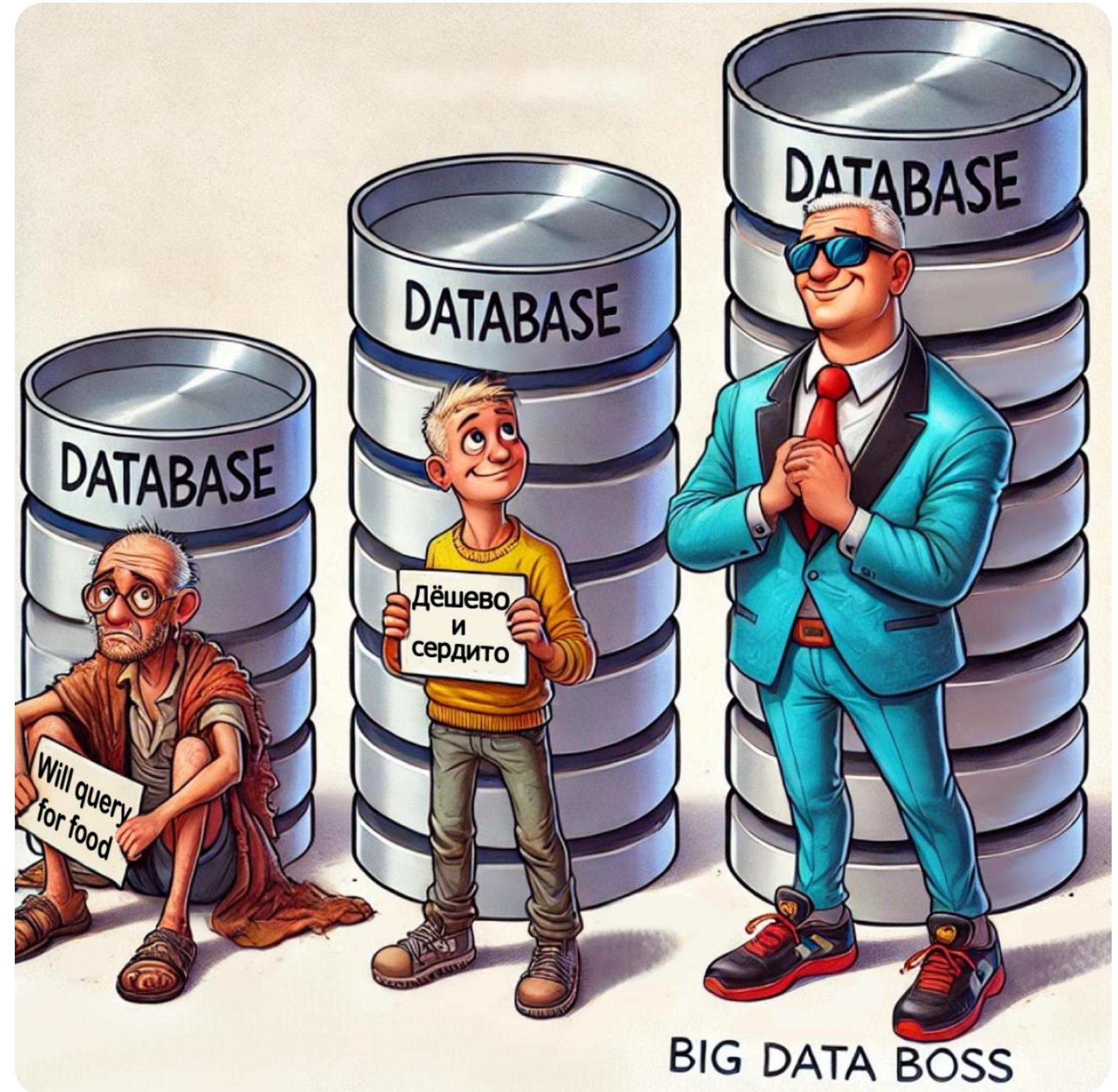
Типы субд и шардирование монолита

01



Эволюция внедрения СУБД

- 1. Нет синхронной репликации:**
данные можно терять.
- 2. Монолитная СУБД (Postgres):**
масштабируемость ограничена.
- 3. Шардированная или распределенная СУБД:** много пользователей и масштабный проект.
- 4. Распределенная СУБД:** консистентный глобальный снепшот, масштабирование на лету в любой момент.



Учитываем не только performance

- Доступность (Availability) и отсутствие даунтайма на обслуживание (замена железа, обновление ПО);
- сохранность данных (Durability).

Всё это подразумевает репликацию:
Эффективность утилизации вычислительных ресурсов: используем ли мы реплики для обработки запросов или нет.

О чём мы расскажем сегодня

1

Поговорим
о мифах, связанных
с шардированием,
широкими транзакциями
и двухфазным коммитом

2

Citus-подобные решения
не ACID и не дают те же
гарантии, что PostgreSQL

3

На примере ТРС-С покажем,
что PostgreSQL крайне
эффективен, но:

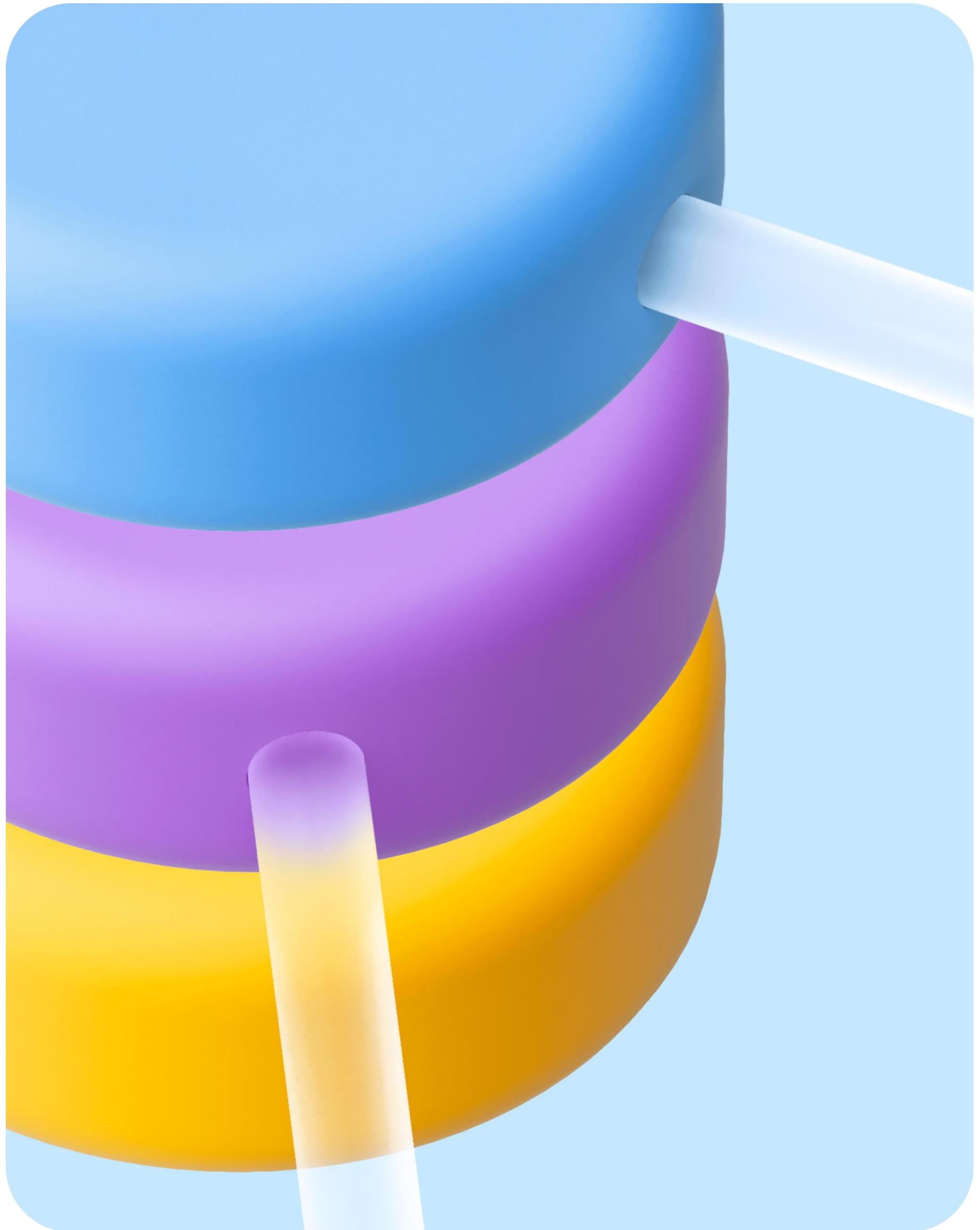
- Не масштабируется
горизонтально
- Синхронная репликация
ограничивает вертикальное
масштабирование

4

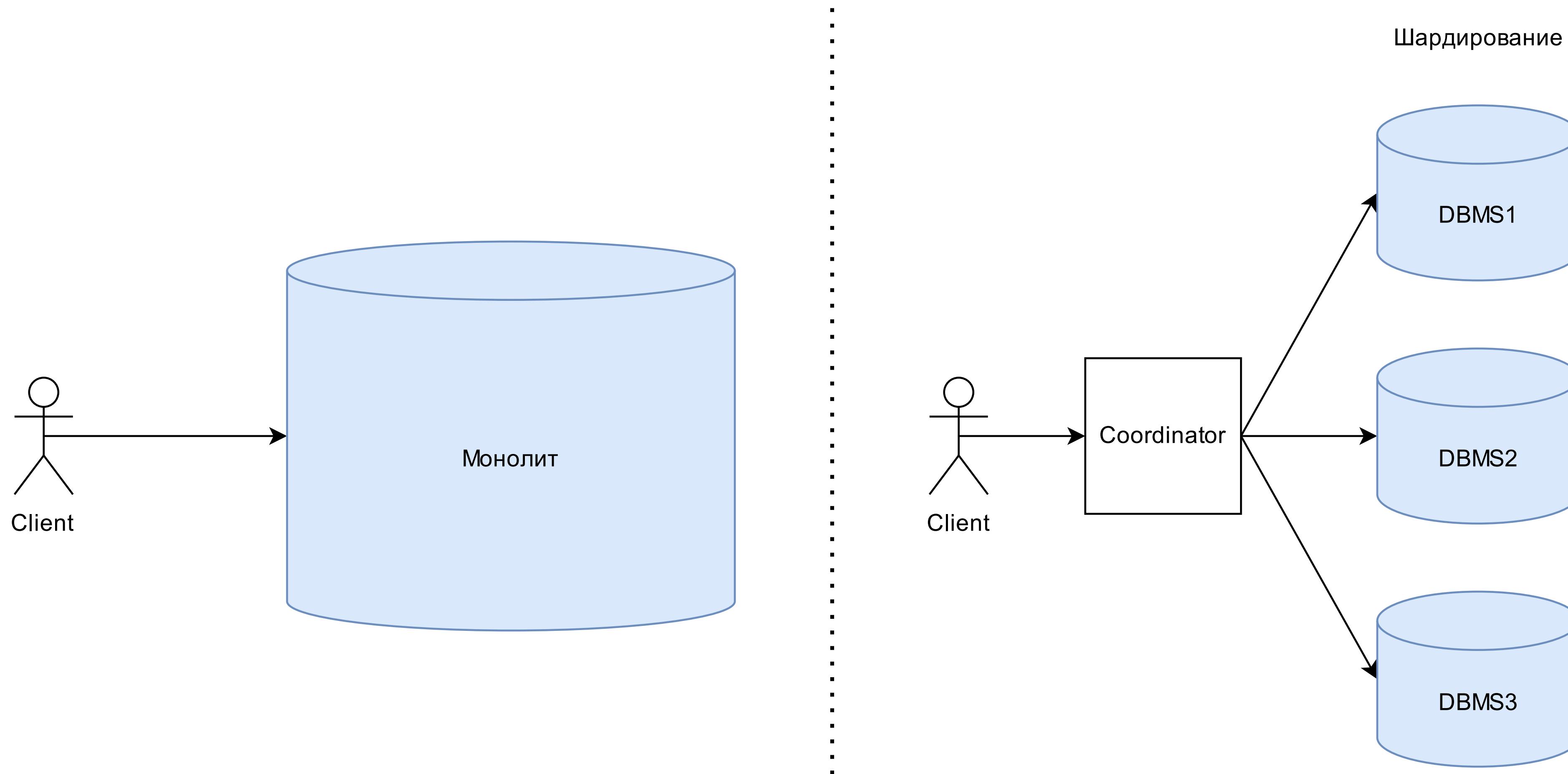
Распределенные СУБД
более эффективны,
чем принято считать

Мифы и заблуждения

02

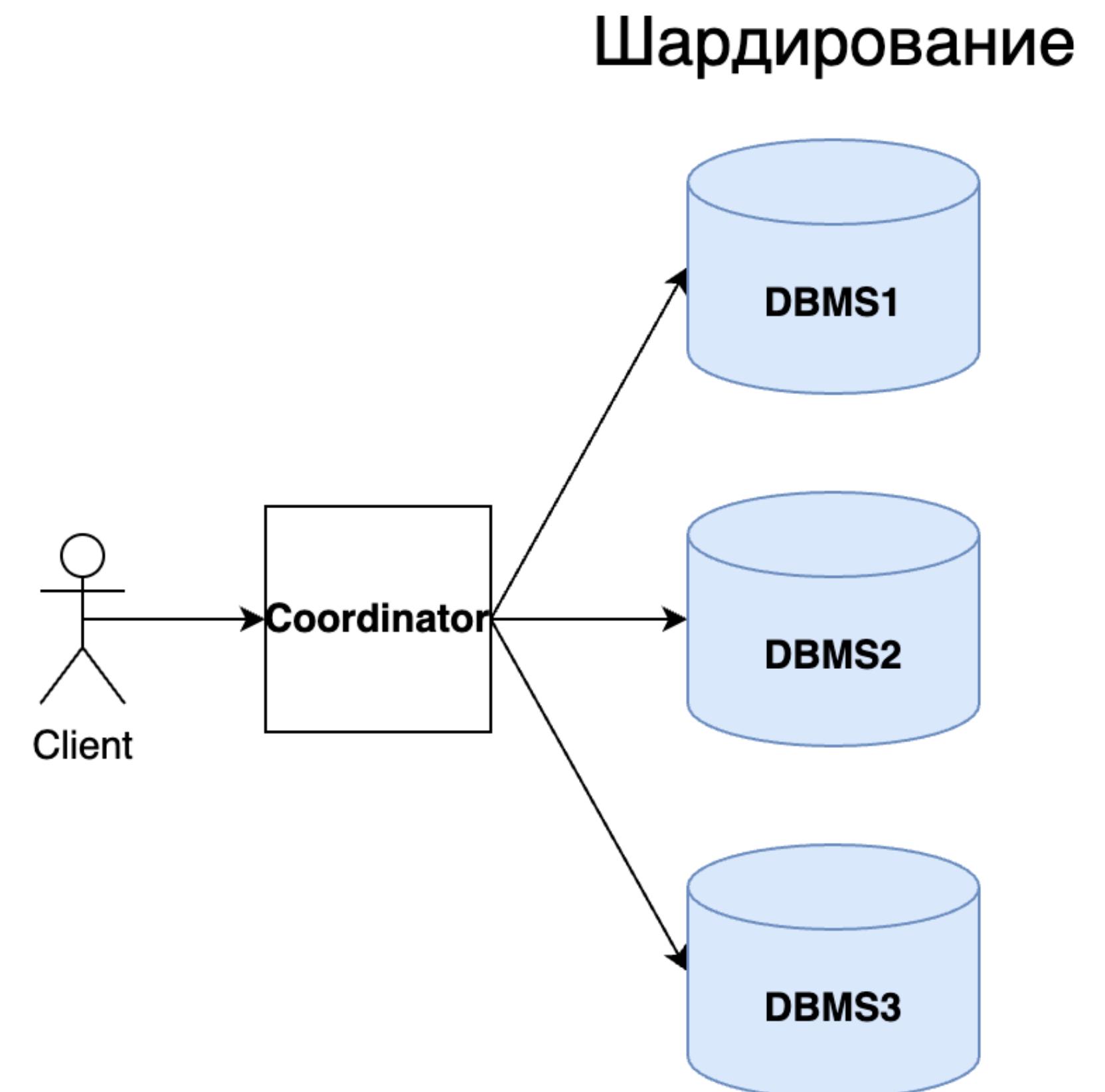


Шардирование монолита



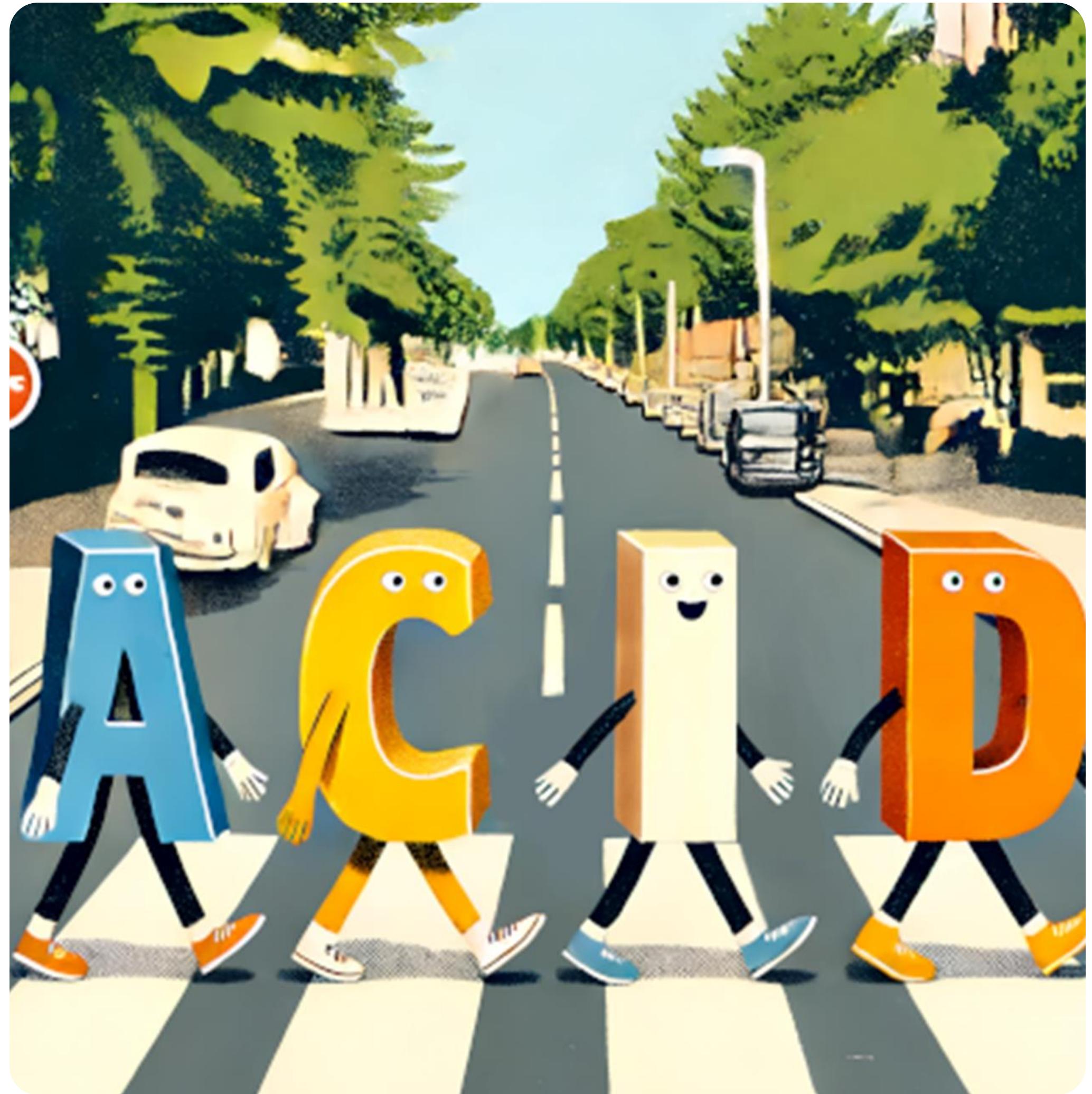
Шардирование монолита

1. Вместо одной СУБД у нас N СУБД, которыми управляет координатор (слой маршрутизации).
2. Одношардовые и многошардовые (широкие/распределенные) транзакции.
3. Шарды видны пользователю, т.к. у транзакций разного типа разные гарантии.



«All your transactions need is ACID» (C)

- Atomicity (Атомарность)
- Consistency (Согласованность)
- Isolation (Изоляция)
- Durability (Устойчивость)



Уровни изоляции

Serializable — уровень по умолчанию в стандарте SQL, CockroachDB и YDB. Аномалии невозможны.

Слабые уровни изоляции (возможны аномалии [\[1\]](#)):

- repeatable read (snapshot isolation);
- read committed — по умолчанию в PostgreSQL;
- read uncommitted.

Уровни изоляции: практический смысл

Serializable: СУБД берет на себя ответственность за обеспечение А-С-| - Д.

Слабые уровни изоляции: разработчик сам отвечает за изоляцию транзакций друг от друга.

Уровни изоляции: Citus не ACID

Широкие транзакции в Citus не изолированы друг от друга!*

«Multi-node transactions in Citus provide atomicity, consistency, and durability guarantees, but do not provide distributed snapshot isolation guarantees. A concurrent multi-node query could obtain a local MVCC snapshot before commit on one node, and after commit on another»

[2] Citus: Distributed PostgreSQL for Data-Intensive Applications

* но они не всем нужны



Когда баланс не сходится

```
--transfer 100 from Alice to Bob
```

```
BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
UPDATE accounts
```

```
SET balance = balance - 100
```

```
WHERE name = 'Alice';
```

```
UPDATE accounts
```

```
SET balance = balance + 100
```

```
WHERE name = 'Bob';
```

```
COMMIT;
```

```
-- Sum all balances
```

```
BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
SELECT SUM(balance) AS total_balance
```

```
FROM accounts;
```

```
COMMIT;
```

А как же Atomicity?

1

Атомарный коммит не обеспечивает атомарную видимость. «Атомарный» означает «всё или ничего».

2

Некоторые предлагают называть это свойство не **Atomicity**, а **Abortability**.

3

Двухфазный коммит (2PC) позволяет добиться **Abortability** — не атомарной видимости.

4

2PC не реализует распределенные транзакции [3].

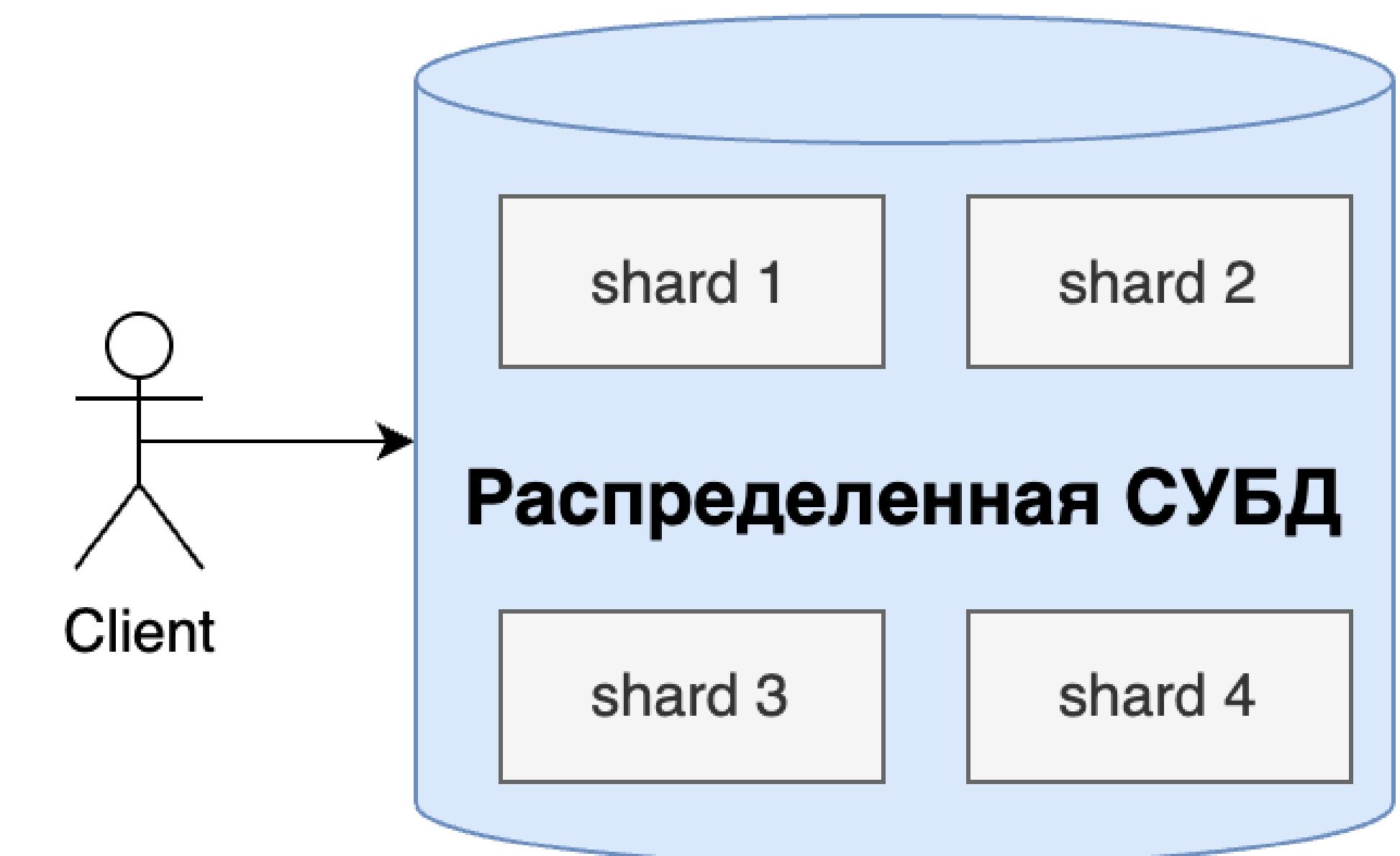
Шардирование в распределенной СУБД

1

Шард является деталью реализации.

2

Для пользователя нет разницы между монолитом и распределенной СУБД: одинаковые гарантии на любые транзакции.



Так ли дороги широкие транзакции?

Теория

1

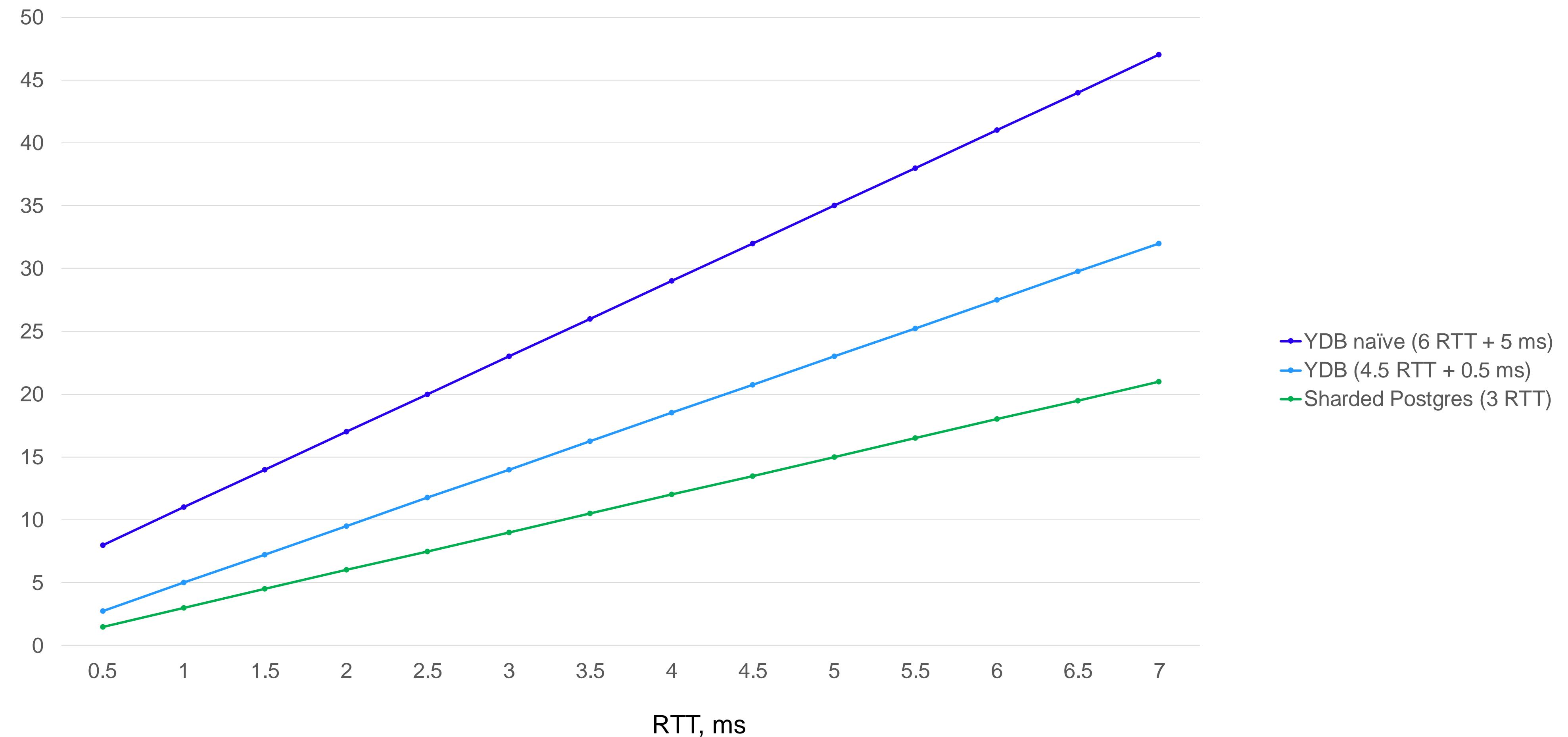
Время выполнения транзакции
принято выражать в количестве
последовательных RTT (Round Trip
Time) и числе операций I/O.

2

NVMe диски — можно пренебречь I/O:

- Postgres: 1 RTT (репликация)
- Sharded Postgres: 1 RTT (репликация)
+ 2 RTT (двуухфазный коммит) —
итого 3 RTT
- YDB naïve: 6 RTT + 5 ms plan/batch [\[4\]](#)
- YDB: 4.5 RTT + 0.5 ms plan / batch [\[5\]](#)

Рассчитанное время выполнения широкой транзакции, ms (меньше лучше)



Так ли дороги широкие транзакции?

Практика

1

В однодатацентровой инсталляции разница — всего единицы миллисекунд.

2

В инсталляции с несколькими зонами доступности разница может составлять до 10 мс.
Но даже наивная реализация широких транзакций в пределах 50 мс.

3

В геораспределённом кластере разница может быть значительной.

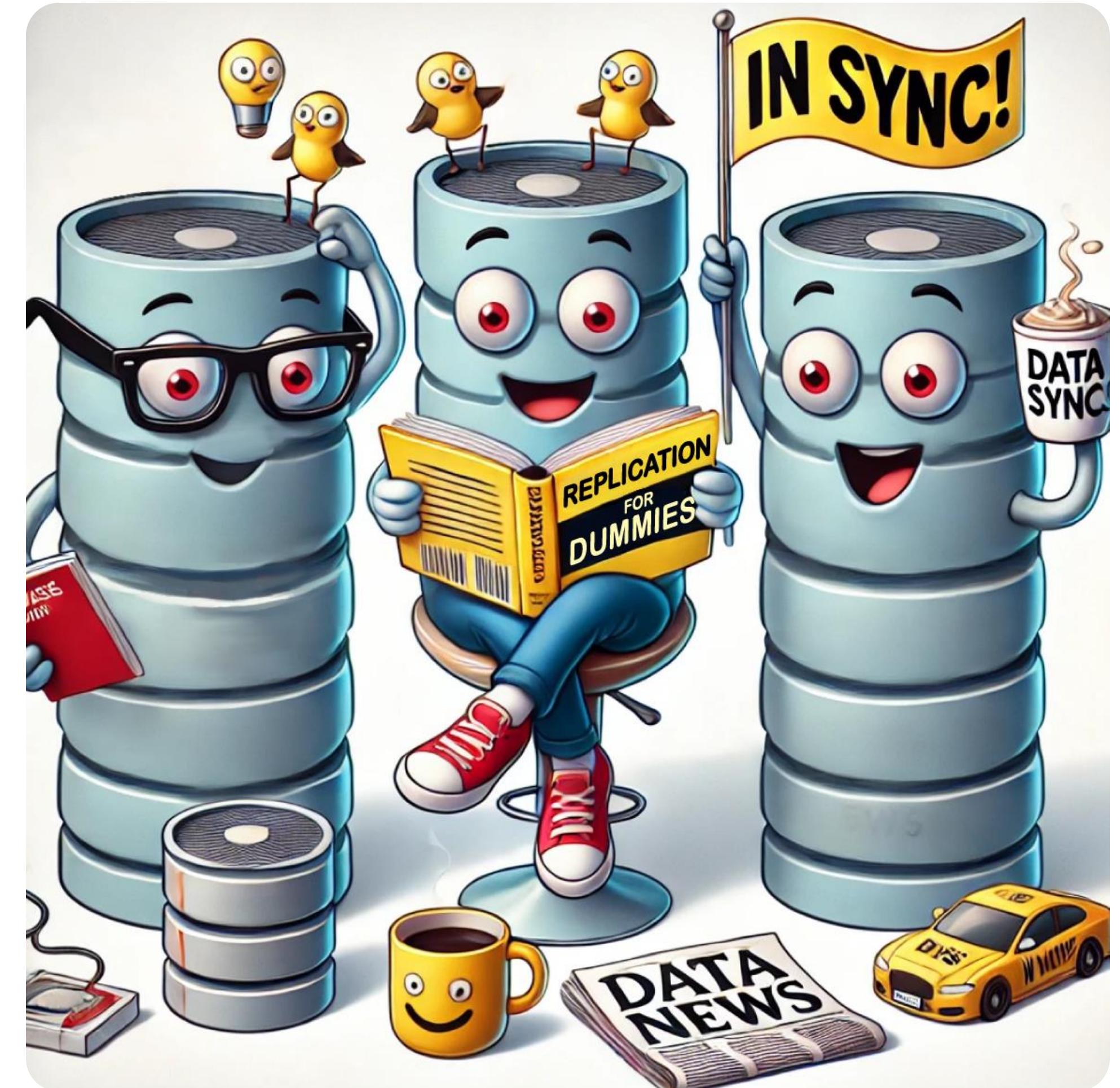
Репликация



Сколько реплик нужно для счастья?

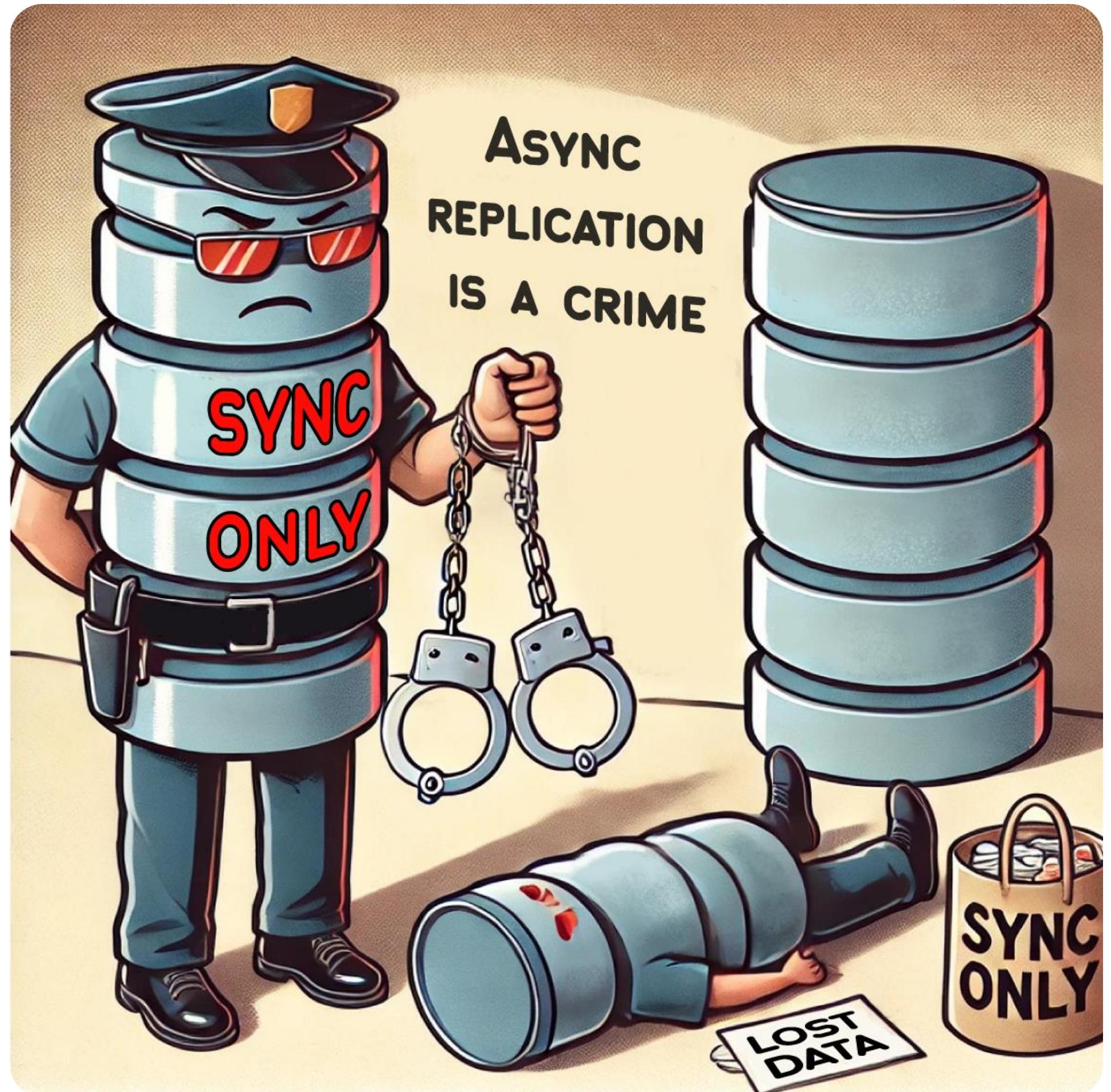
Зависит от вашей модели отказов, но

три реплики — хорошее
минимальное число.



Асинхронная репликация

- Риск потери данных;
- stale reads и аномалии;
- комбинация синхронной и асинхронной репликации только при большем числе реплик.



Утилизация реплик в монолите 1

- 1** Лидер тратит на обработку все X-ядер CPU, в кластере три сервера по X-ядер. Реплики простаивают.
- 2** Мы хотим переживать отказ одного сервера.
- 3** Можно было бы распределить нагрузку между двумя серверами и тратить на них по $X/2$ ядер.
- 4** Если использовать реплики, то можно иметь 3 сервера по $X/2$ ядер и с меньшим количеством RAM.
- 5** Обычно это позволяет уменьшить latency.

Утилизация реплик в монолите 2

- 1** При двух репликах «простой» составляет 66.6% — такое же плохое число, как и утилизация на 99.9%.
- 2** Если сервер всего 16–32 ядра — не так уж и дорого.
- 3** А если сервер 32-64 ядра и с полкой NVMe?



Репликация в шардированных и распределенных системах

1

Реплики и лидеры распределены по всем хостам: отличная утилизация ресурсов.

2

За счет шардирования у нас много маленьких потоков репликации, что лучше масштабируется.

Важно помнить

1

Citus не Postgres: нет консистентного распределенного снепшота и нет изоляции многошардовых транзакций.

2

Citus отлично работает с одношардовыми транзакциями.

3

Citus не является распределенной СУБД.

4

Многошардовые транзакции не так уж и дороги, когда у вас быстрая сеть и NVMe-диски.
Не надо их бояться, когда они нужны.

Но когда же именно PostgreSQL становится мало?

1

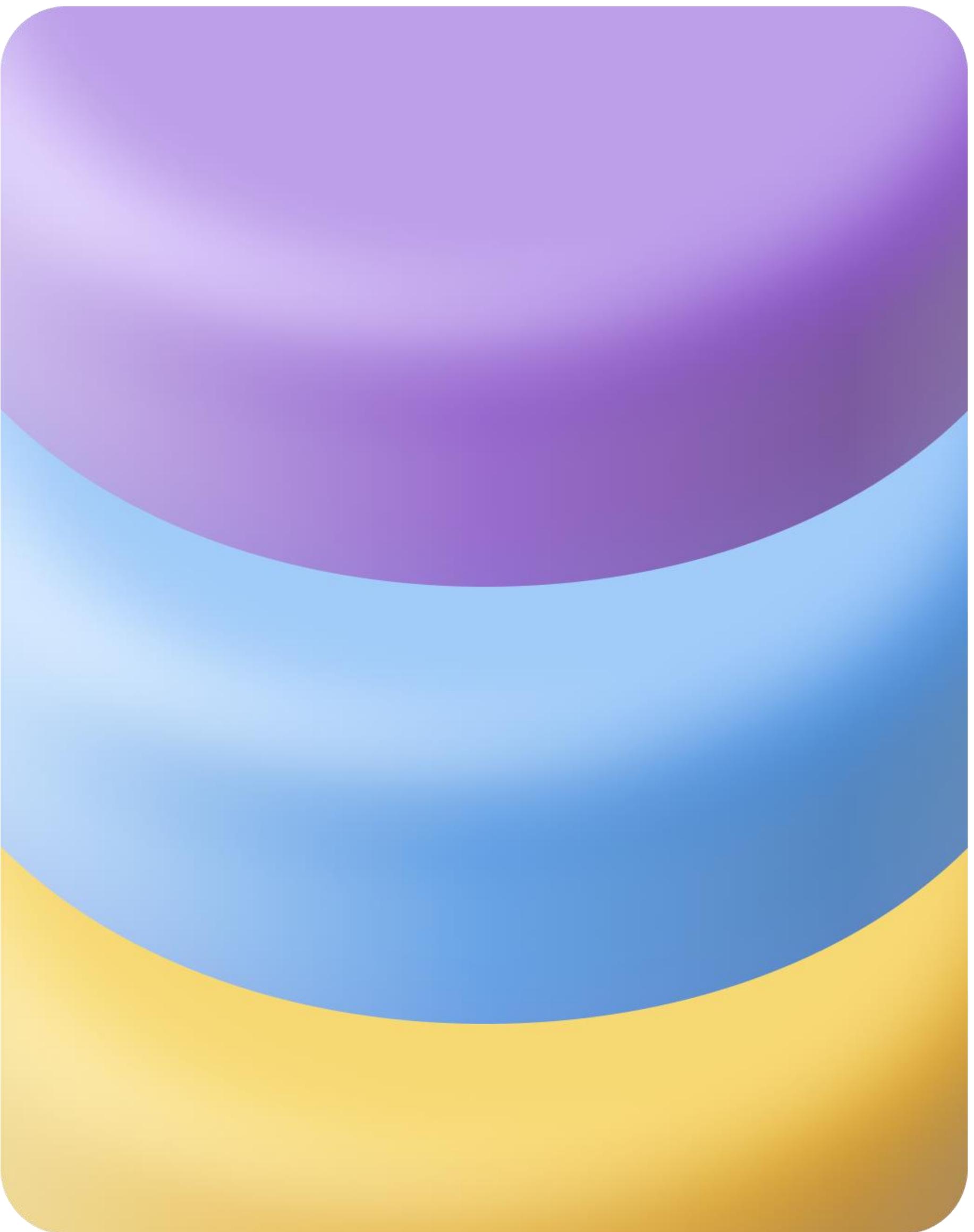
Мы взяли популярный OLTP бенчмарк TPC-C, 3 мощных сервера и нашли границу, когда PostgreSQL не справляется.

2

Оценили эффективность распределенных СУБД по сравнению с PostgreSQL в такой небольшой инсталляции.

ТРС-С и оценка производительности

03



TPC-C

Стандарт появился в 1992 году

«Единственная объективная методика оценки производительности OLTP»,

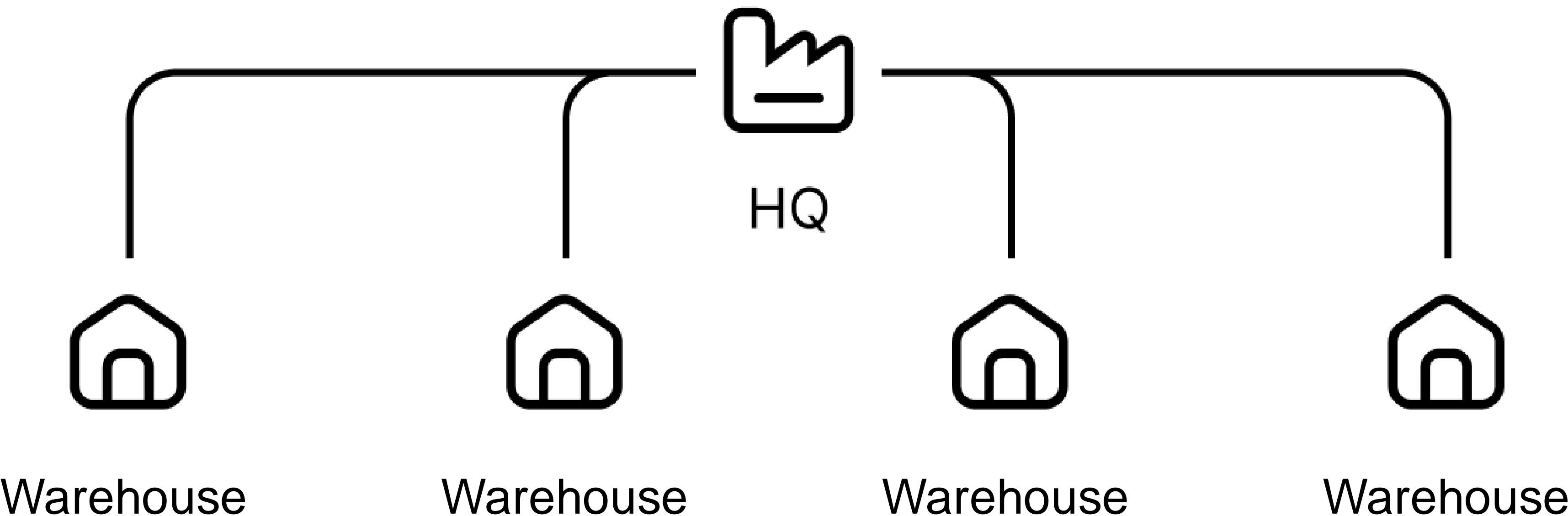
— CockroachDB

«Наиболее реалистичное и объективное измерение производительности OLTP-систем»,

— YugabyteDB



TPC-C – эмуляция e-commerce организаций

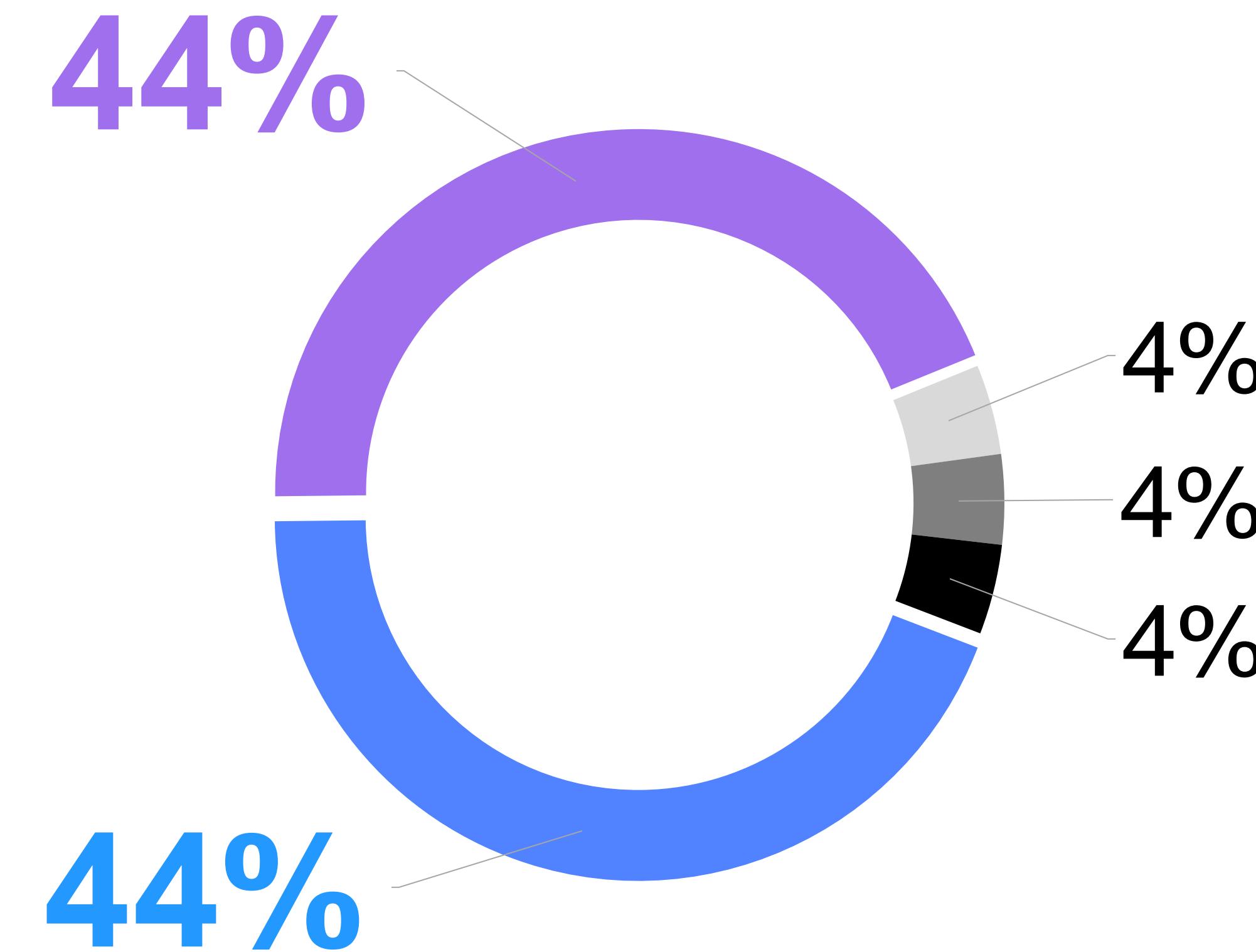


Логика ТРС-С

- Число складов задаётся при запуске;
- каждый склад обслуживает 10 районов, примерно 100 МБ данных;
- в каждом районе есть терминал;
- пользователи делают заказы и оплачивают их;
- иногда проверяется статус заказа;
- выполняется доставка;
- на складах проводится инвентаризация.

Транзакции ТРС-С

NewOrder Payment OrderStatus Delivery StockLevel



Транзакции ТРС-С

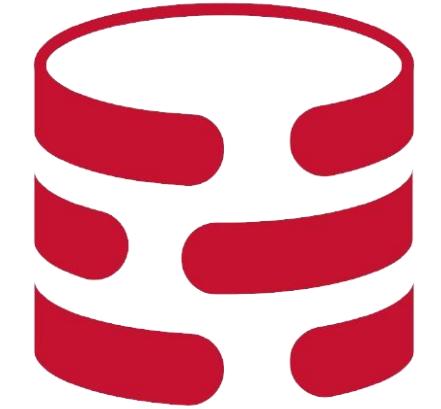
Требуется
уровень
`Serializable`.

Интерактивные
(многошаговые).

Соотношение
чтений
и записи — 2:1.

Измеряем только
число заказов
в минуту — `tpmC`.

Проект СМУ Benchbase



- Multi-DBMS SQL Benchmarking Framework на основе JDBC.
- Написан в Carnegie Mellon. под руководством проф. Энди Павло.
- Легко добавлять новые СУБД и бенчмарки.
- Единственная широко известная реализация ТРС-С.
- YugabyteDB использует форк Benchbase.
- Мы пошли тем же путём.

Требования к клиенту для запуска 15 000 складов

В оригинальном Benchbase:

- 150 000 threads;
- 600 GB RAM.

Запускали ТРС-С на 5 серверах
(каждый 128 ядер и 512 GB RAM)!

Масштабируемся

Хотим пропустить СУБД, в которой
9, 15, 30, 60, 81 серверов

YDB, CockroachDB, YugabyteDB

\$10,000

стоит один такой
эксперимент в AWS

И одним экспериментом
не обойтись



Наш форк и апстрим

- github.com/ydb-platform/tpcc и github.com/ydb-platform/tpcc-postgres;
- планируем постепенно затянуть наши улучшения в апстрим;
- мы переделали ТРС-С на виртуальные потоки Java, а они могут приводить к дедлокам в других бенчмарках, поддерживаемых Benchbase;

[6] Как мы начали использовать виртуальные потоки Java 21 и на раз-два получили дедлок в ТРС-С для PostgreSQL.

Postgres на прокачку

04



Наш сетап: 3 сервера в одном ДЦ

128 логических ядра

Два Intel Xeon Gold 6338 CPU @ 2.00GHz,
hyper-threading включен

Включены Transparent hugepages
(huge pages в случае PostgreSQL)

512 GB

RAM

Ubuntu 20.04.3 LTS

4 NVMe диска

RAID0 для PostgreSQL

База должна переживать отказ одного сервера

PostgreSQL имеет две синхронные реплики.

CockroachDB и YDB:
фактор репликации 3.

В PostgreSQL настраивается всё!

1

Write-ahead
log

2

B-Tree

3

Execution
engine

4

Replication

I/O

Наш подход к настройкам

От отказоустойчивого, но безумно быстрого, к менее быстрому, но отказоустойчивому PostgreSQL

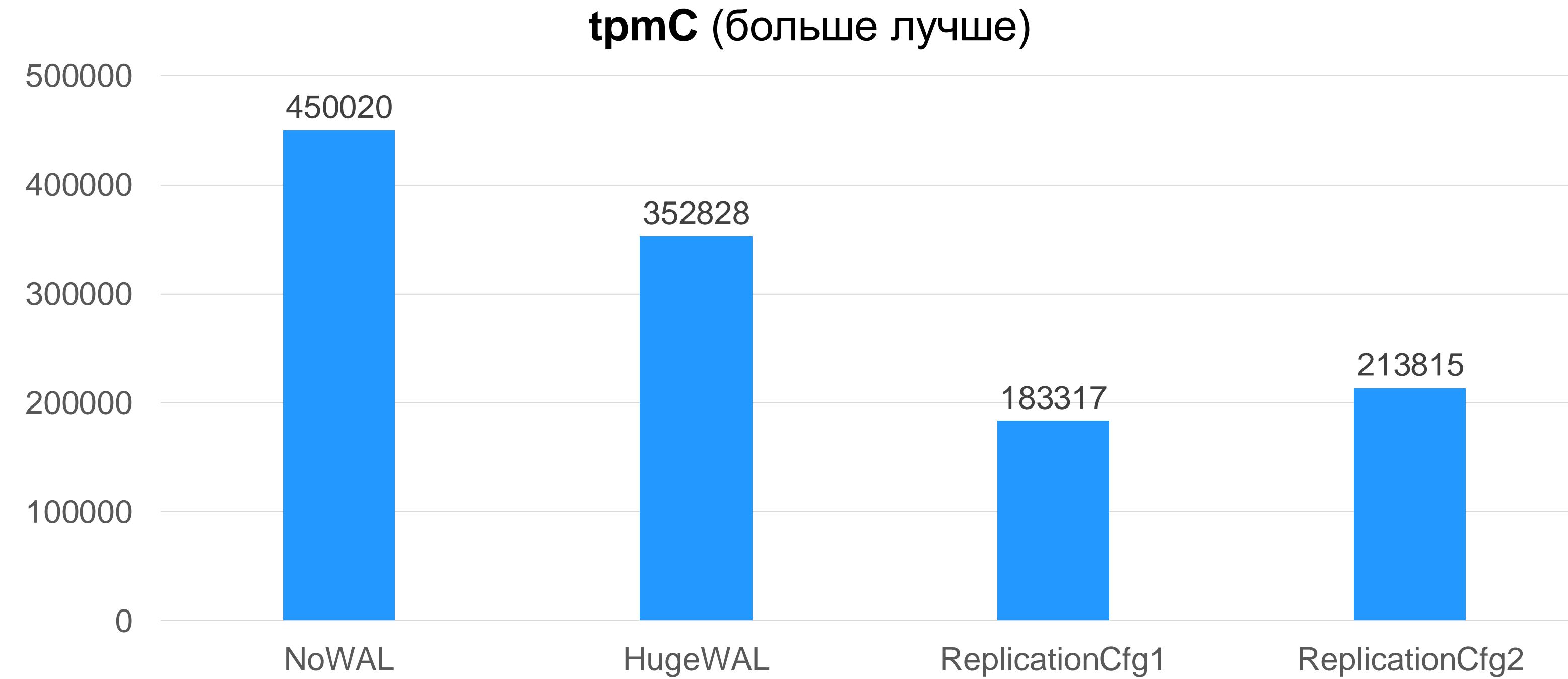
Три NVMe RAID0 — данные, один NVMe — WAL:

1. Unlogged таблицы с выключенной репликацией: [NoWAL](#)
2. Огромный WAL (время восстановления десятки минут), но зато идеальное распределение I/O: [HugeWAL](#)
3. Две синхронные реплики: [ReplicationCfg1](#)

Два NVMe RAID0 — данные, два NVMe RAID0 — WAL:

4. Две синхронные реплики + `synchronous_commit = apply`: [ReplicationCfg2](#)

Отказо**не**устойчивый и очень быстрый PostgreSQL



* Результаты не являются официально принятыми TPC результатами и несопоставимы с другими результатами теста TPC-C, опубликованными на сайте TPC

Влияние настроек на результат

1 Отказоустойчивый Postgres фантастически быстрый.

3 У PostgreSQL всего 1 поток на применение WAL репликами.

2 С репликацией результат в два раза хуже.

4 Синхронная репликация в PostgreSQL – узкое место и ограничивает возможности вертикального масштабирования.

[7] Подробнее о конфигах и результатах

Много ли 200К ттмс?

~8 000

интерактивных транзакций в секунду

~130 000

запросов в базу в секунду

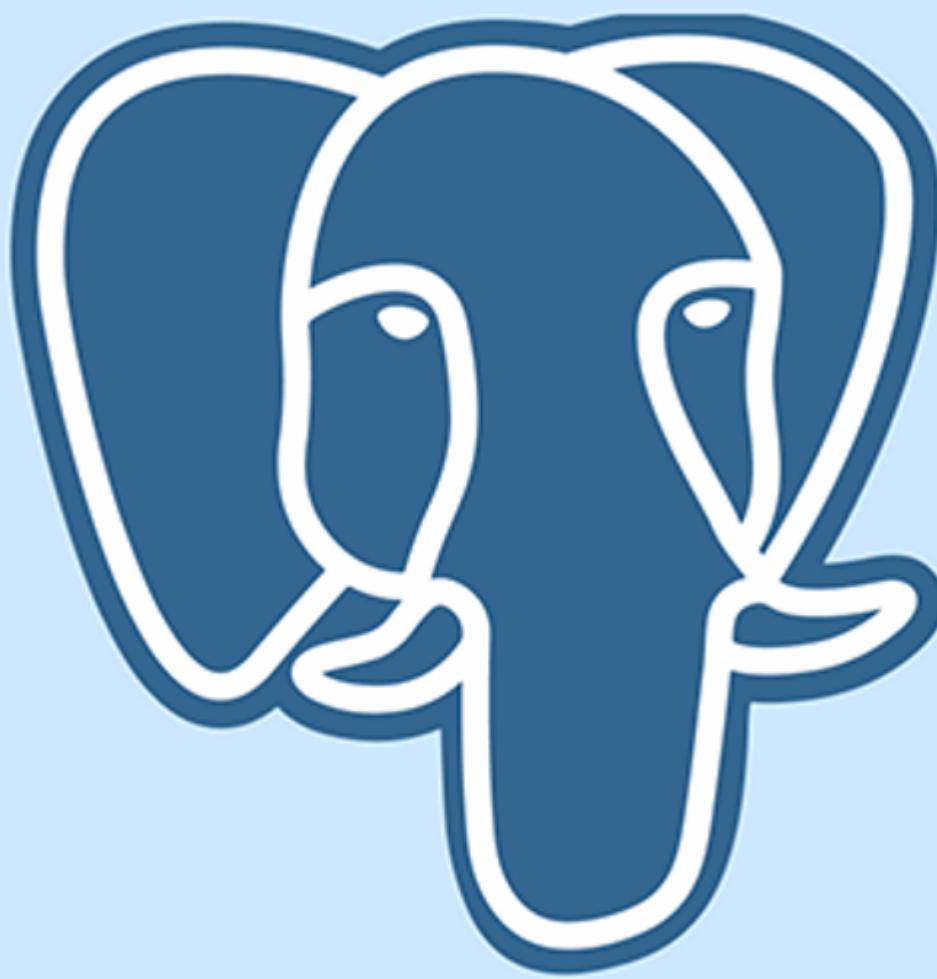
На мастере:

- запись WAL 400 MB/s,
запись данных 600 MB/s
- чтение 700 MB/s
- потребление сети 9 Gbit/s
- CPU в среднем 20 ядер (из 128)

**Могут ли распределенные
СУБД показать сопоставимый
результат на том же железе?**

PostgreSQL vs. распределенные СУБД

05



vs.



YDB

Open-Source Distributed SQL Database

1

**Реляционная
СУБД: OLTP
и OLAP**

2

**Кластеры с
тысячами
серверов**

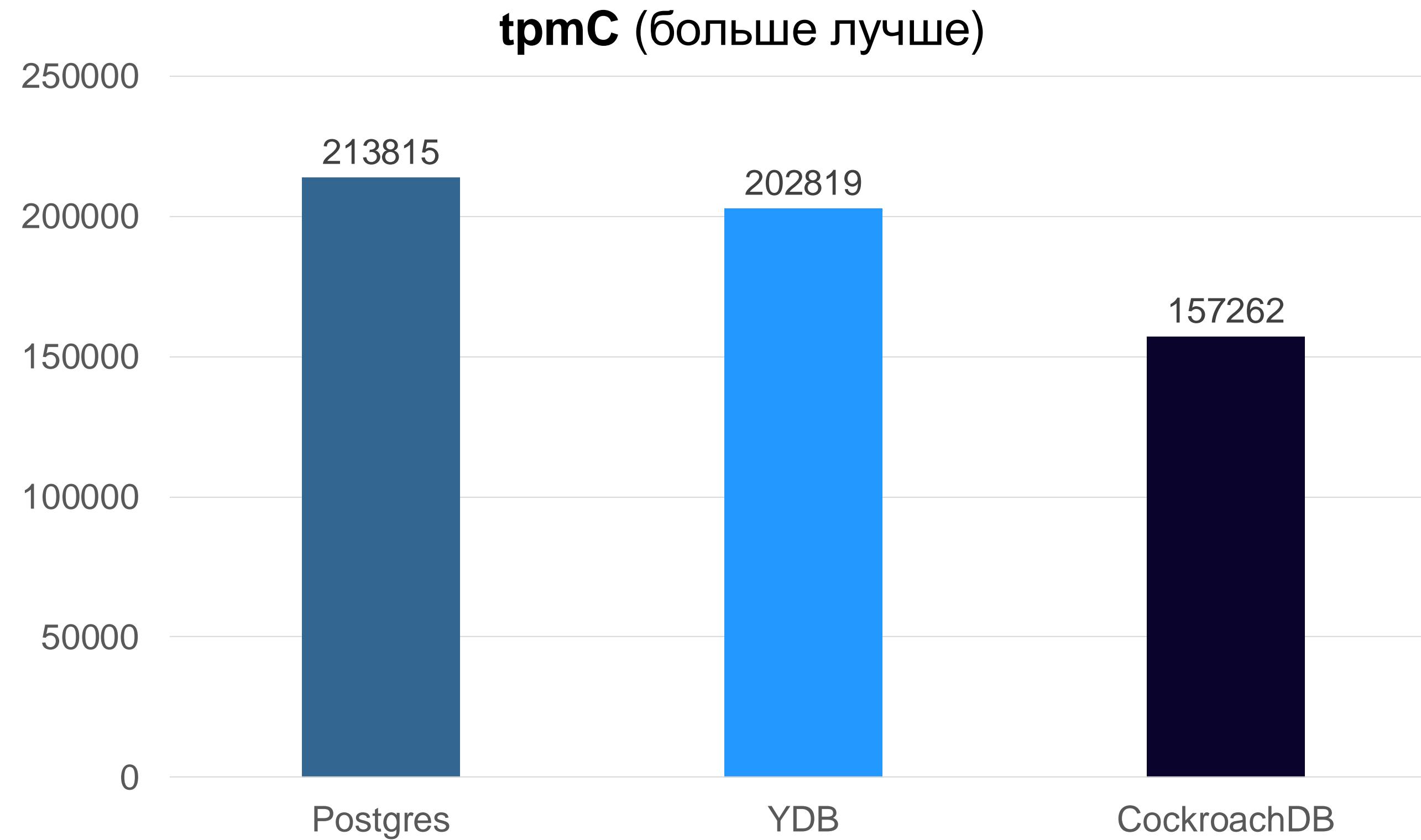
3

**Строгая
консистентность**

4

**Лицензия
Apache 2.0**

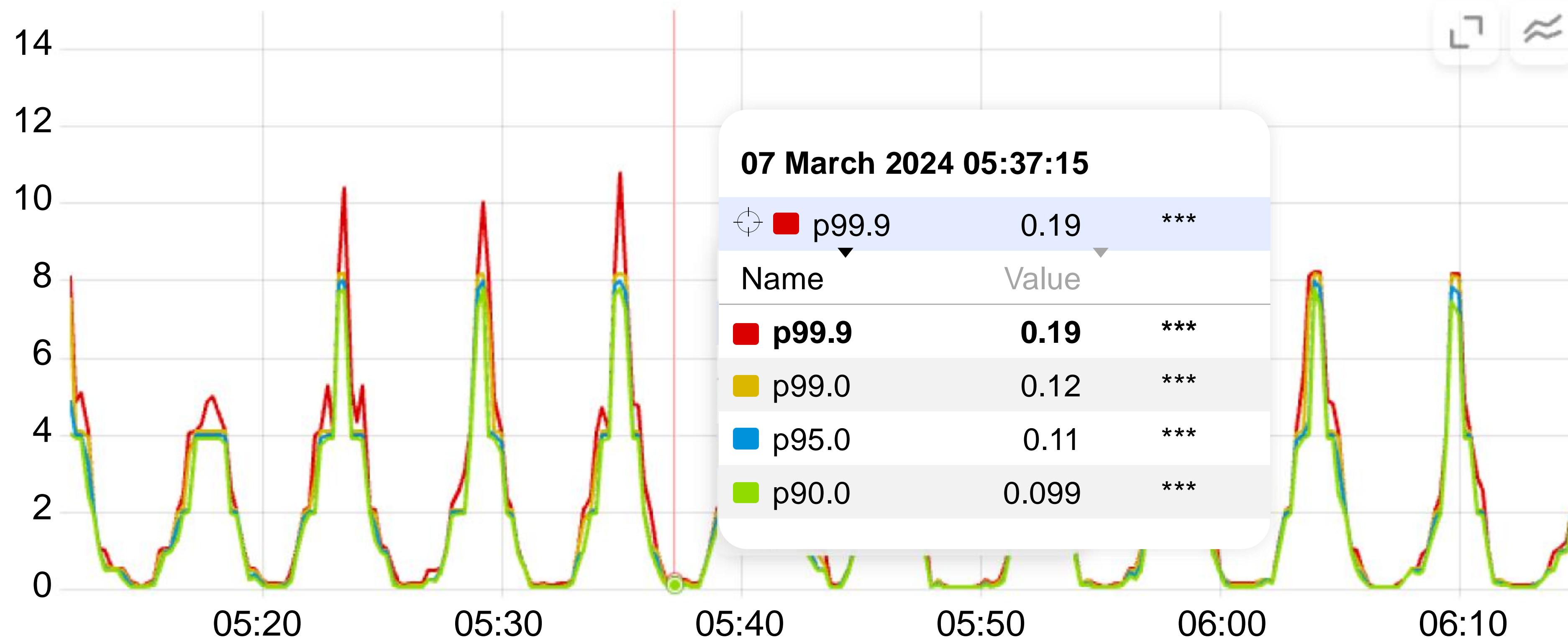
tpmC* (throughput)



* Результаты не являются официально принятыми TPC результатами и несопоставимы с другими результатами теста TPC-C, опубликованными на сайте TPC

NewOrder latency* в Postgres

Postgres NewOrder Latencies, секунды (меньше лучше)



* Результаты не являются официально принятыми TPC результатами и несопоставимы с другими результатами теста TPC-C, опубликованными на сайте TPC

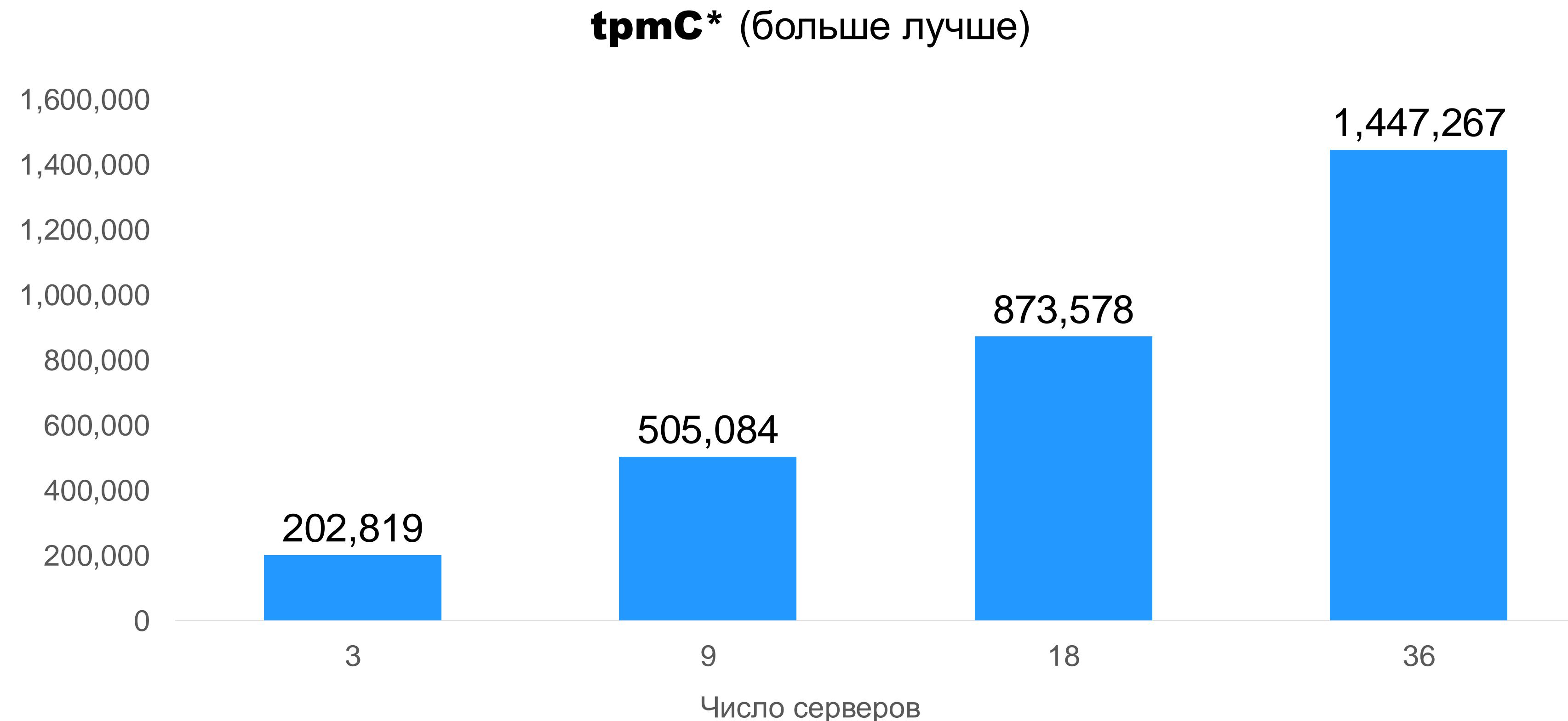
NewOrder latency в Postgres

Каждый пик совпадает с началом записи чекпоинта.

Сессии «висят» в ожидании IPC: SyncRep.

Это архитектурная проблема (всего 1 поток на получение и применение WAL репликами).

Масштабируемость YDB (один ДЦ, без роста latency)



* Результаты не являются официально принятыми TPC результатами и несопоставимы с другими результатами теста TPC-C, опубликованными на сайте TPC

Результаты ТРС-С

1

PostgreSQL
набрал на 5%
больше тpmC,
чем YDB.

2

Значительно
более высокие
latency
в PostgreSQL.

3

YDB обошла
CockroachDB
на 29% тpmC.

4

Распределенные
СУБД легко
масштабируются
горизонтально.

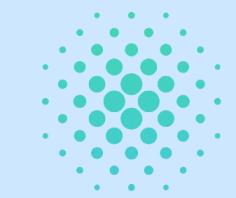
Заключение

1 PostgreSQL крайне эффективен, но:

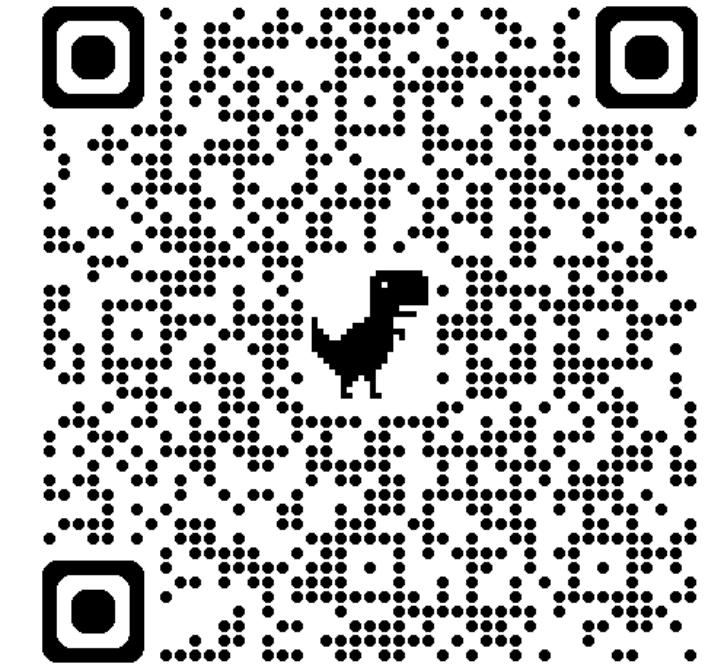
1. Не масштабируется горизонтально.
2. Синхронная репликация ограничивает вертикальное масштабирование.
3. Не всегда можно просто докинуть ядер и RAM.

2 Citus-подобные решения не ACID и не дают тех же гарантий, что PostgreSQL.

3 Распределенные СУБД более эффективны, чем принято считать, — обратите на них внимание, когда PostgreSQL вам станет мало.



SmartData
2024



Слайды и материалы

Вопросы?

Евгений Иванов,
Ведущий разработчик YDB, Яндекс

Олег Бондарь,
СРО YDB, Яндекс