

# Sharded and Distributed Are Not the Same: What You Must Know When PostgreSQL Is Not Enough

Evgenii Ivanov,  
Principal Software Developer, YDB





# About myself

- YDB developer
- Amateur speaker
- Outside YDB I enjoy spending time with my family, aerial photography, and reading





# Rumors about YDB and YugabyteDB

- Many believe that YDB and YugabyteDB are the same thing
- Others say we once had a bar fight





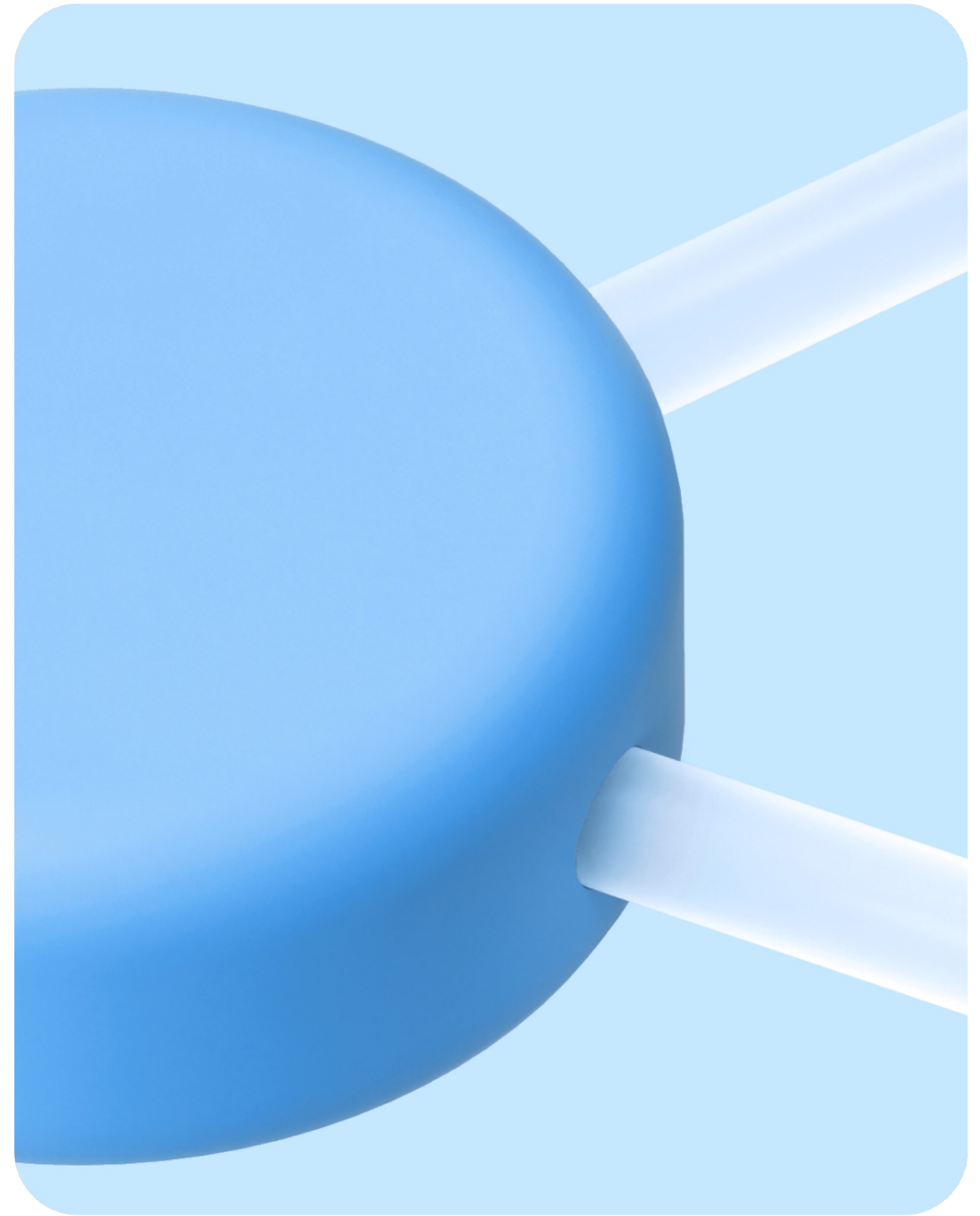
# The truth

- YDB and YugabyteDB are **different** distributed DBMSs
- We enjoy discussing topics related to benchmarking and distributed systems



# DBMS types and sharding a monolith

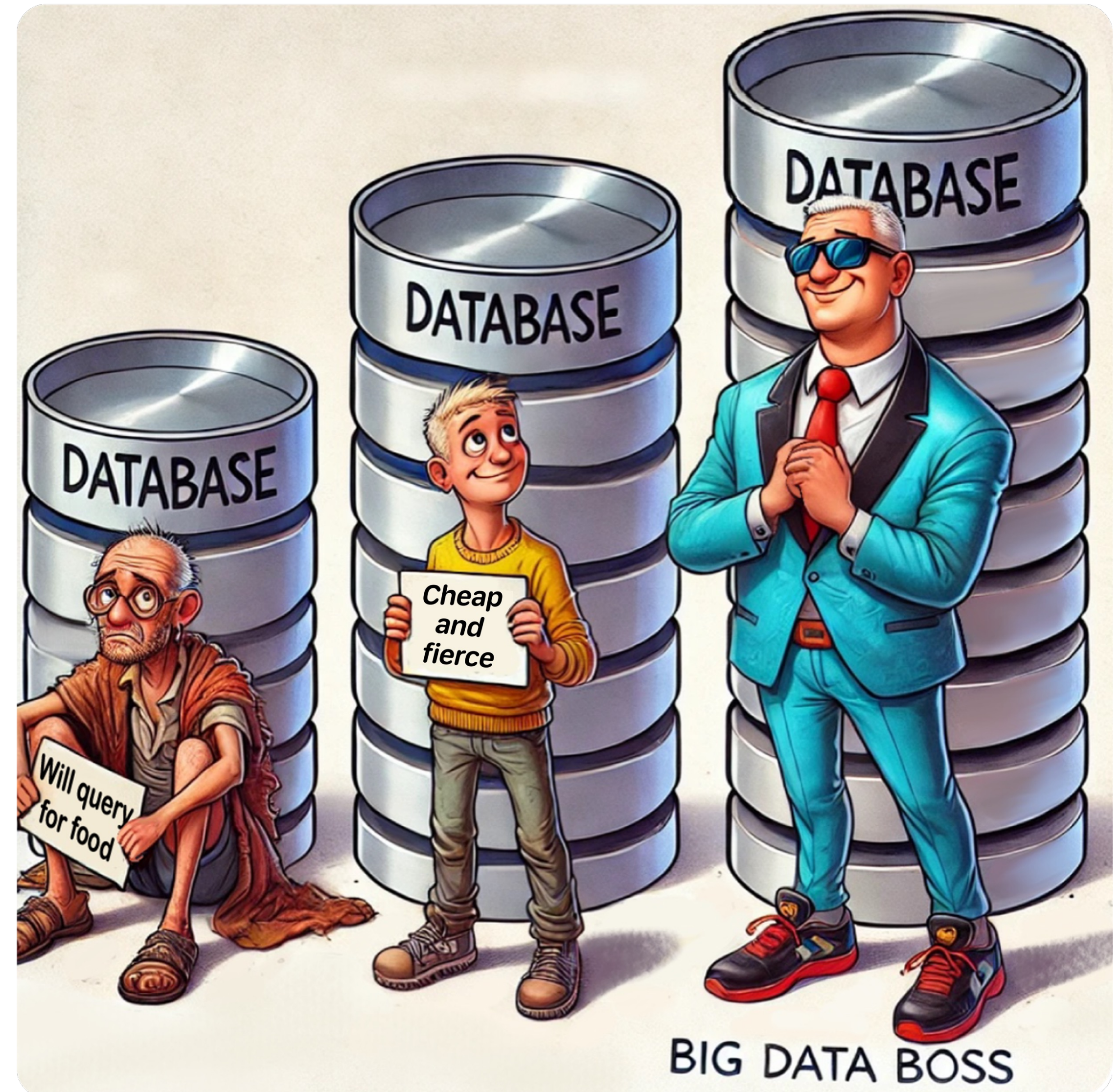
01





# DBMS usage evolution

1. **No synchronous replication:**  
it's OK to lose data
2. **Monolith DBMS like PostgreSQL:**  
scalability is limited
3. **Sharded or Distributed DBMS:** many users and large-scale project
4. **Distributed DBMS:** consistent global snapshot, on-the-fly scaling at any time





# It's not just about performance

- Availability
- Durability

**All of this implies replication**

And efficiency of resource utilization depends on whether we use replicas for query processing or not



# What we will talk about today

1

We will discuss myths related to sharding, wide/distributed transactions, and two-phase commit

3

Using TPC-C as an example, we will show that PostgreSQL is highly efficient, but synchronous replication might limit vertical scaling

2

In case of multi-shard transactions Citus-like solutions are not ACID and do not provide the same guarantees as PostgreSQL

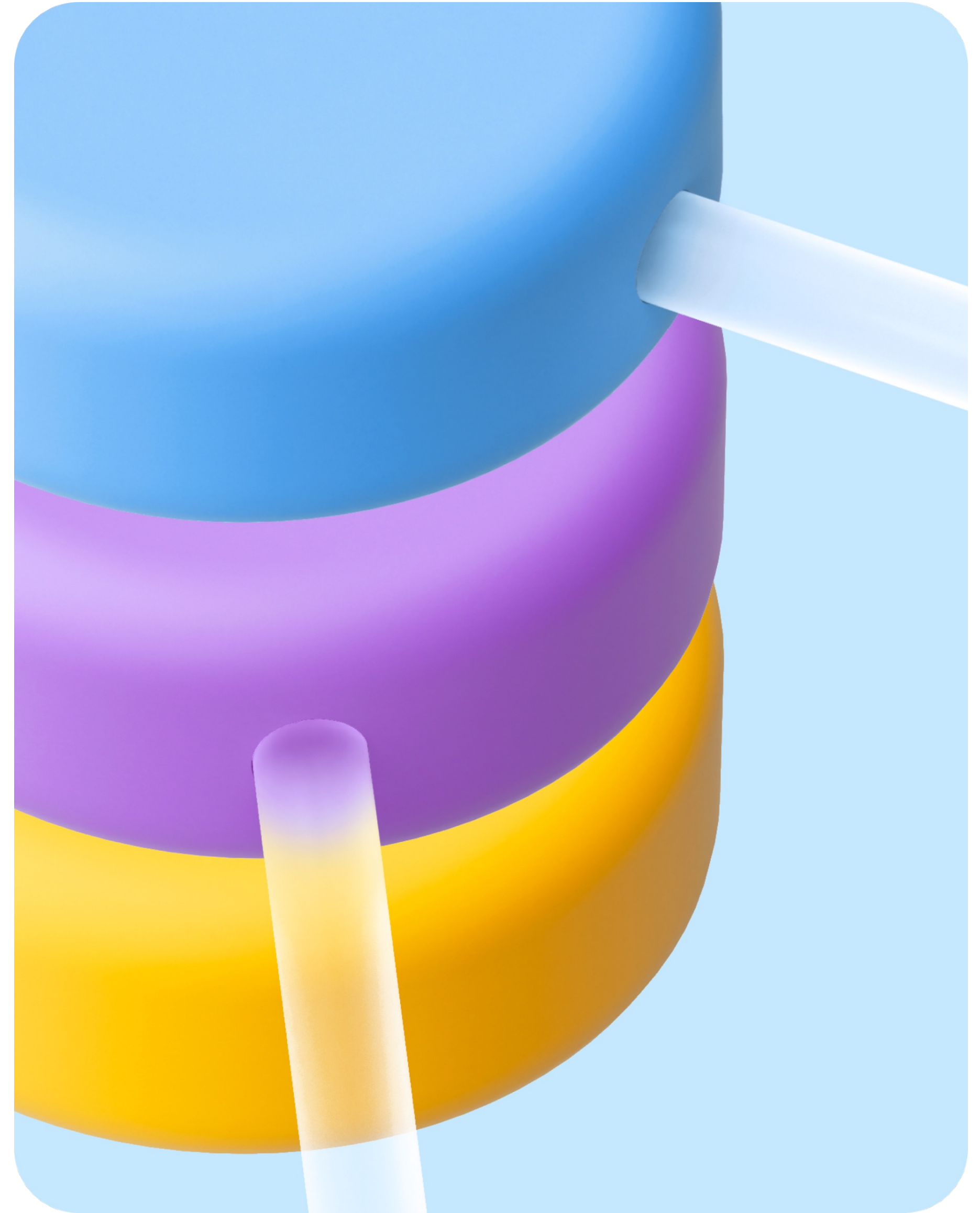
4

Distributed DBMSs are more efficient than commonly believed



# Myths and misconceptions

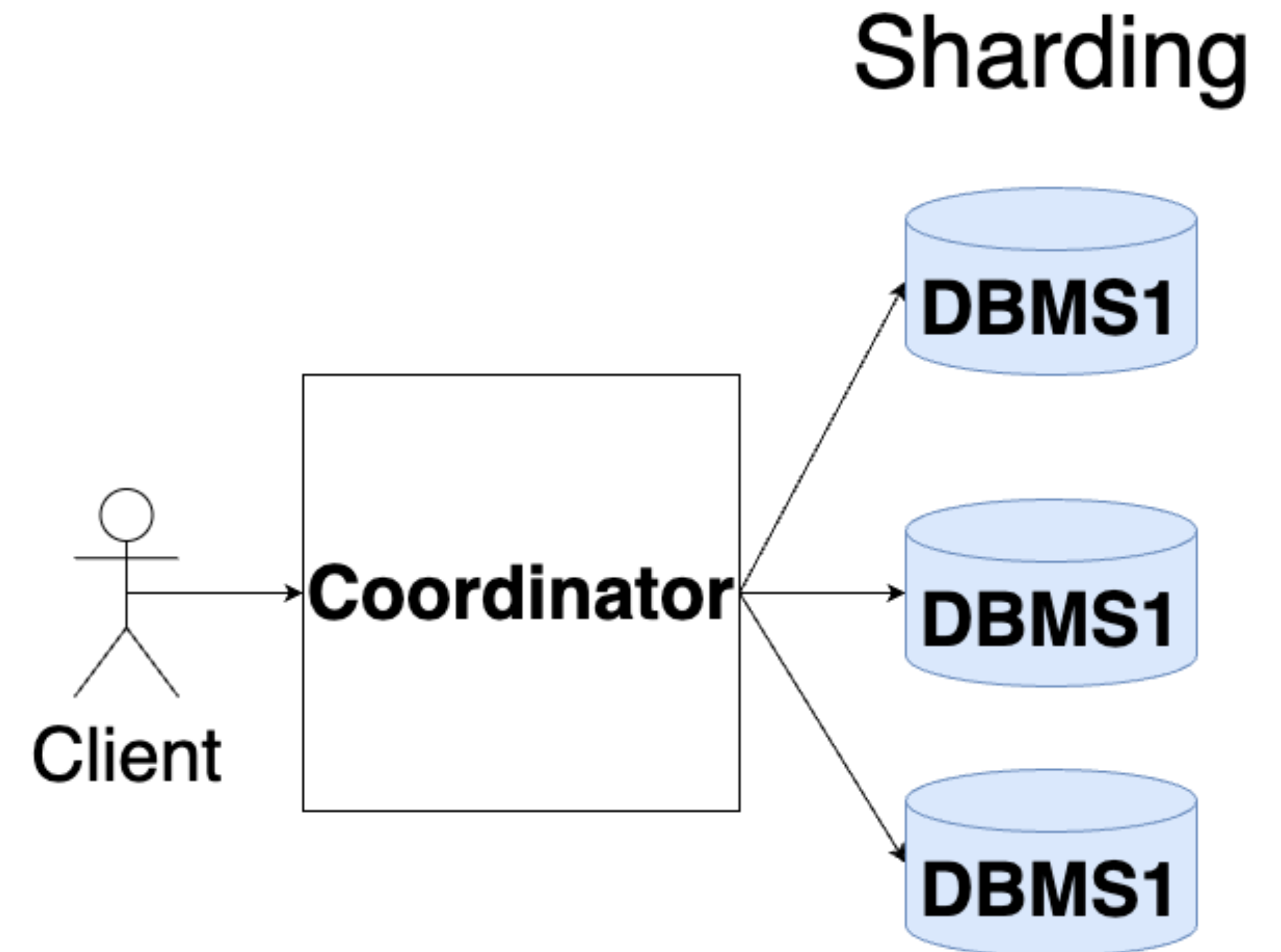
02





# Monolith sharding

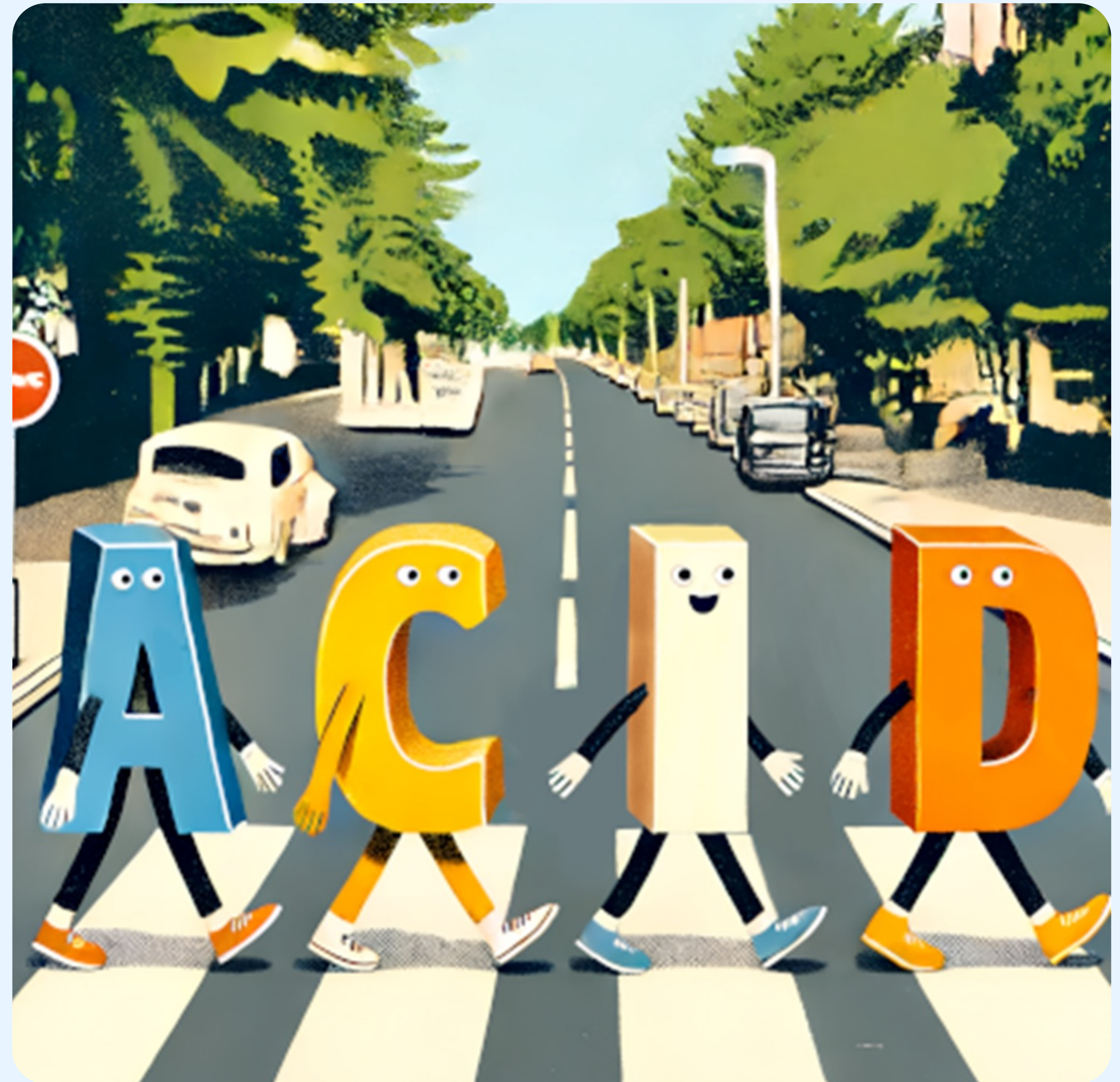
1. Instead of a single DBMS, we have N DBMSs, managed by a coordinator (routing layer).
2. Single-shard and multi-shard (wide) transactions.
3. Shards are visible to the user, as single-shard and multi-shard transactions have different guarantees.





# All your transactions need is ACID

- Atomicity
- Consistency
- Isolation
- Durability





# Isolation levels

**Serializable** — the default level in SQL standard, CockroachDB and YDB.  
Anomalies are impossible.

**Weaker isolation levels**  
(anomalies are possible [\[1\]](#)):

- repeatable read (snapshot isolation)
- read committed — the default in PostgreSQL
- read uncommitted



# Isolation levels: practical considerations

## Serializable

DBMS is the one who takes care about A-C-**I**-D.

## Weaker isolation levels

Application developer is responsible for transaction isolation.



# Isolation levels: Citus is not ACID

**Wide transactions in Citus are not isolated!\***

*“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

*\* however, not everybody needs it. It depends on your app.*





# When the balance is incorrect

-- 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;
```

-- Calc the total balance

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

```
SELECT SUM(balance)  
AS total_balance  
FROM accounts;
```

```
COMMIT;
```



# What about Atomicity?

1

**Atomic** commit does not provide atomic visibility. «Atomic» means «all or nothing»

2

Some suggest calling this property **Abortability** rather than Atomicity

3

**Two-phase commit (2PC)** achieves Abortability, but not atomic visibility

4

2PC does not implement distributed transactions [\[3\]](#)



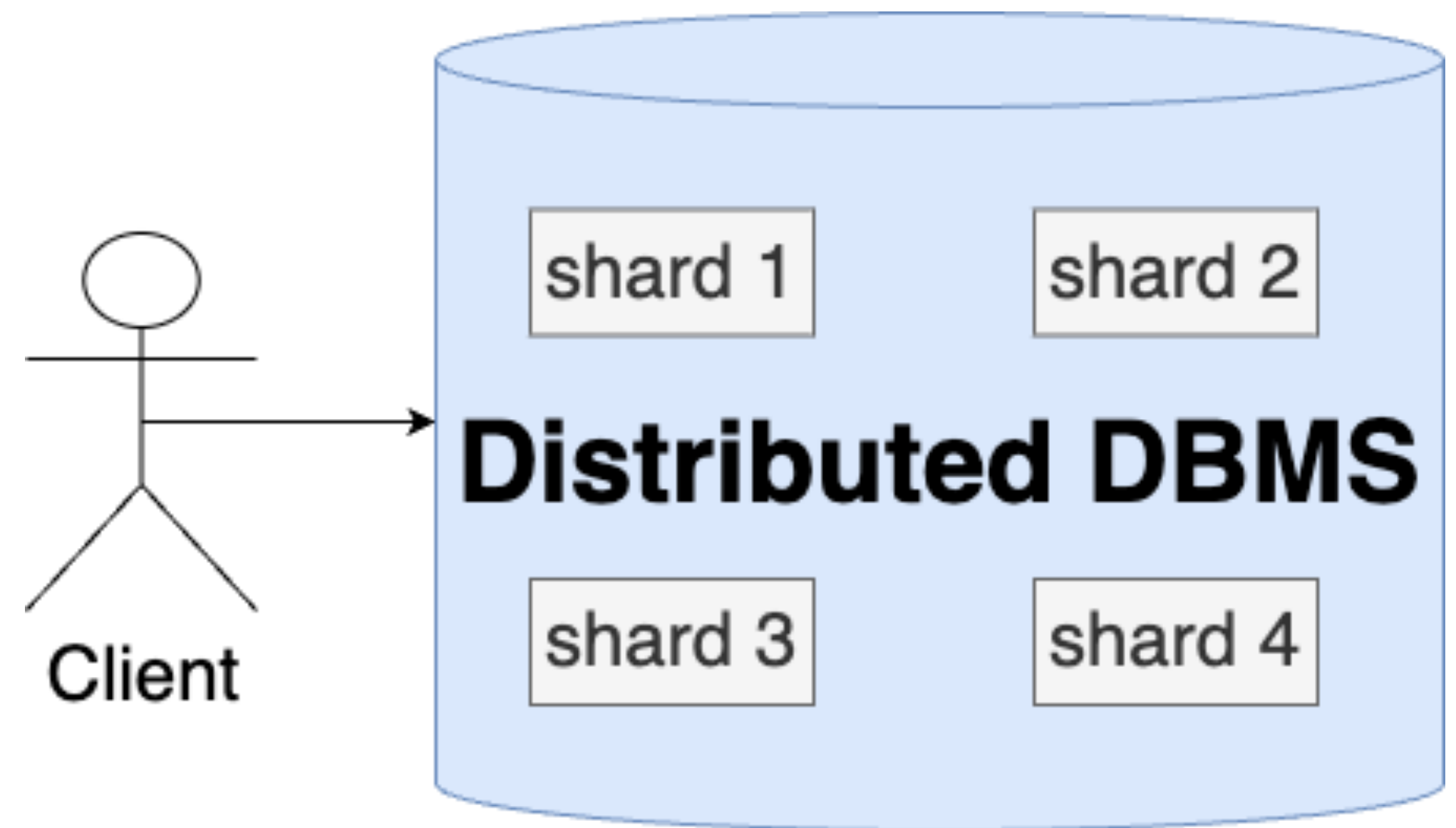
# Sharding in a distributed DBMS

1

Shard is just an implementation detail of a DBMS

2

For the user, there is no difference between a monolithic and a distributed DBMS: the same guarantees for any transactions





# Are wide transactions really that expensive? Theory.

## 1

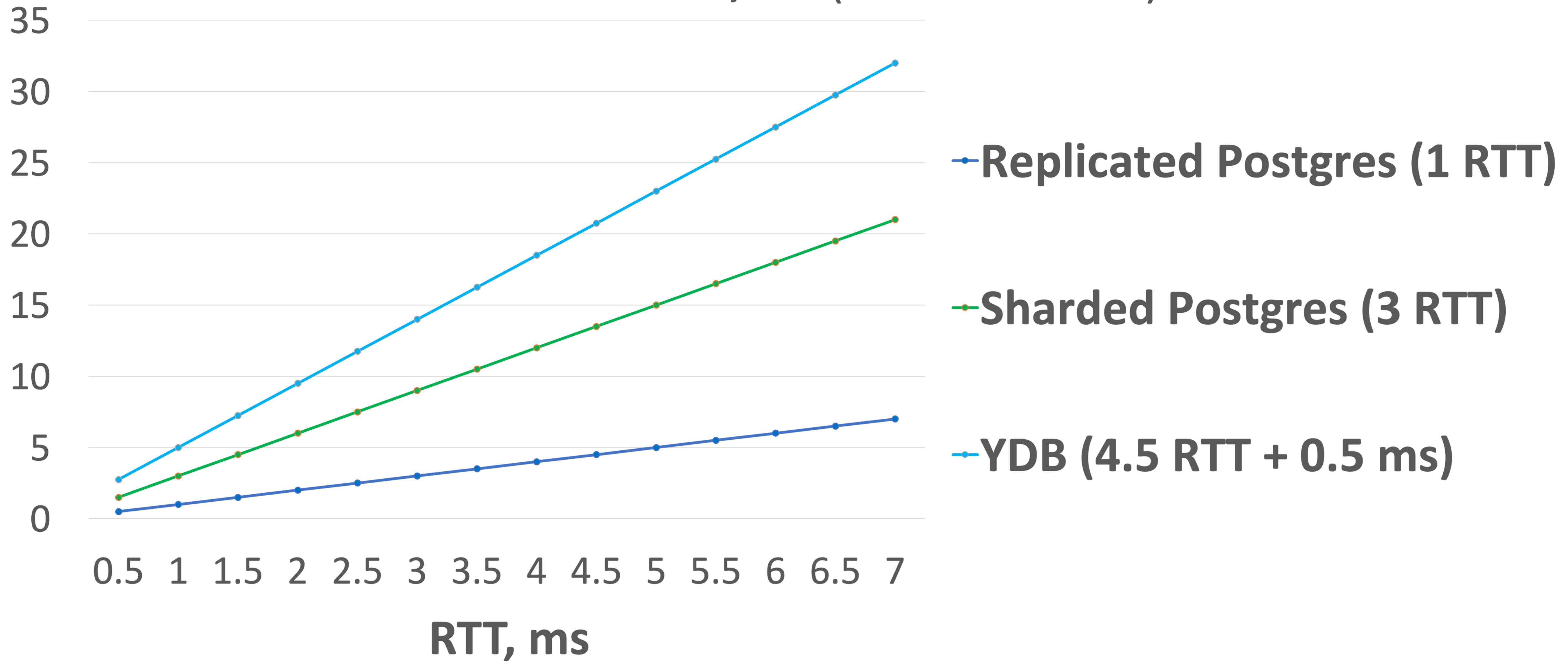
Transaction execution time is typically expressed in terms of the number of consecutive RTTs (Round Trip Time) and the number of I/O operations

## 2

NVMe disks — I/O can be neglected

- Postgres: 1 RTT (replication)
- Sharded Postgres: 3 RTT where 1 RTT (replication) + 2 RTT (2PC)
- YDB: 4.5 RTT + 0.5 ms plan/batch [\[4\]](#)

# Transaction time, ms (lower is better)





# Are wide transactions really that expensive? A practical perspective.

1

In a single availability zone installation, the difference is only a few milliseconds

2

In a multi-availability zone installation, the difference can be up to 10 ms

But distributed transactions are still below 50 ms

3

In a multi region cluster, the difference can be significant. In this case if your workload allows, pure sharding might be better

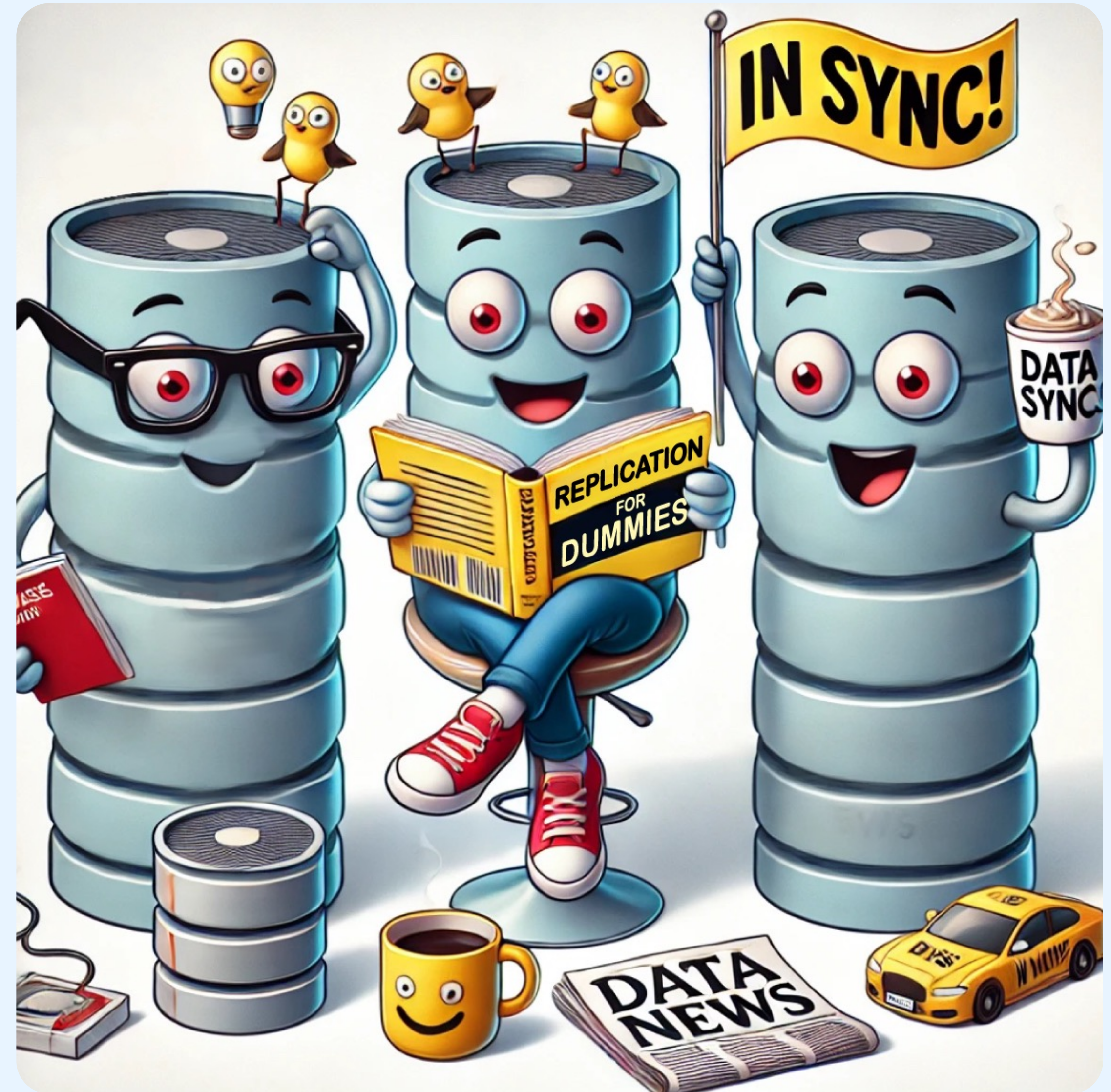
# Replication





# How Many Standby Replicas Are Enough?

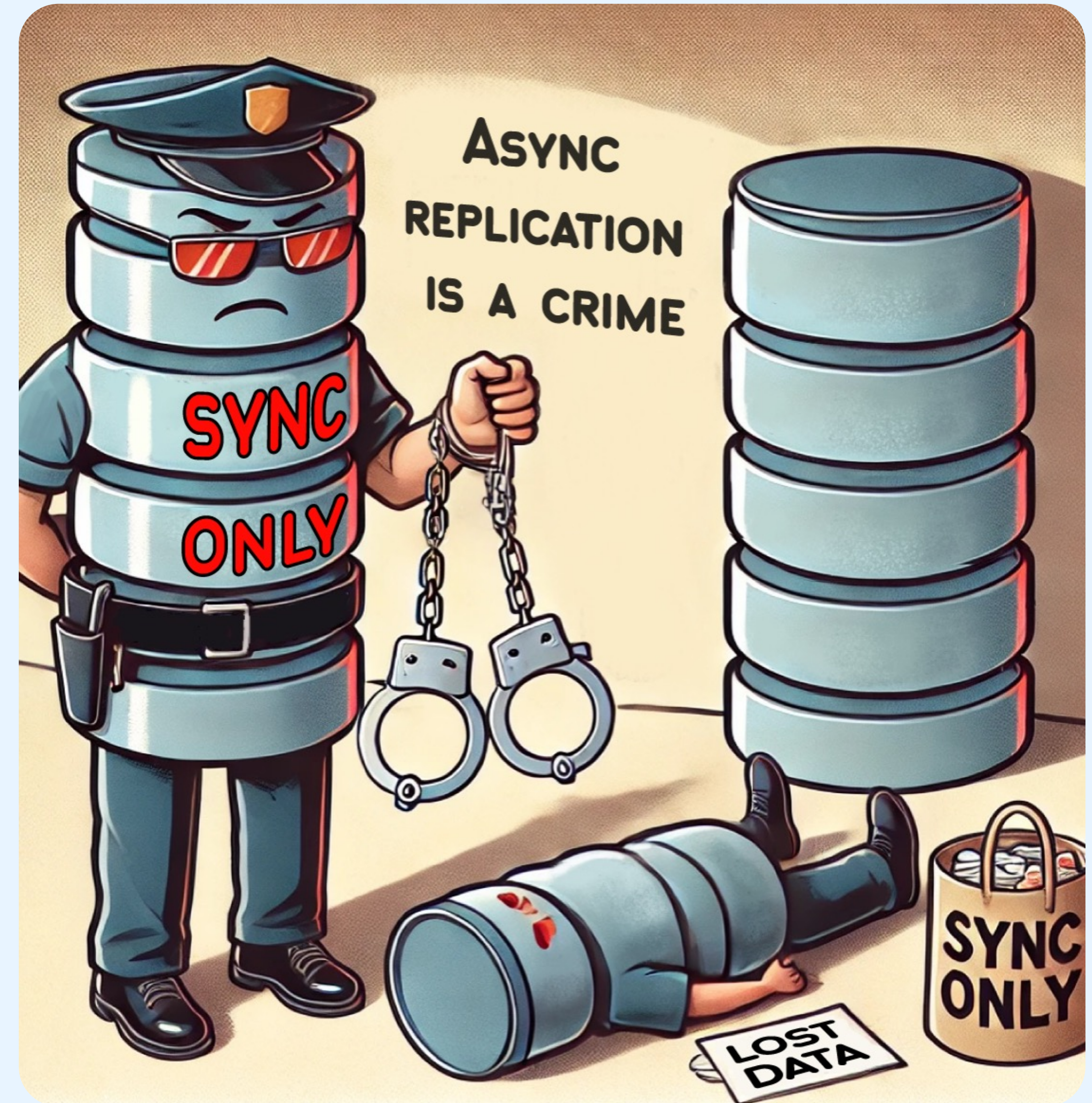
It depends on your fault tolerance model, but **three replicas** is a good minimum number (leader and two standby replicas)





# Async replication in the absence of Sync replication

- risk of data loss
- stale reads and anomalies
- combination of synchronous and asynchronous replication only with a larger number of replicas





# Replicas utilization in a monolith (1)

**1** The leader uses **X CPU cores** for processing, while there are three servers in the cluster, each with X cores and **3X cores in total**.

The replicas remain idle

**2** We want to tolerate the failure of one server. The original **X cores** load could be distributed between two servers left, using **X/2 cores** on each

**3** Also, if replicas are used, you could have 3 servers with **X/2 cores each** and **less RAM**

**4** This usually helps reduce latency

# Replicas utilization in a monolith (2)

**1** With two replicas, the 'idle time' is 66.6% — the same poor number as utilization at 99.9%

**2** If the server has only 16–32 cores, it's not that expensive

**3** But what if the server has 64-128 cores and many NVMe disks?





# Replication in both sharded and distributed DBMS's

1

Replicas and leaders are distributed across all hosts: **66.6%** hardware utilization VS. **33.3%** in a monolith DBMS.

2

Thanks to sharding, we have many small replication threads, which scale better

# Remember that

**1** Citus works great with single-shard transactions. In a multi-region installations it might outperform distributed DBMSs.

**2** Citus is not PostgreSQL: it provides different guarantees for single- and multishard transactions.

**3** Citus is not a distributed DBMS: isolation of multi-shard transactions is just read committed.

**4** Don't be afraid of YugabyteDB, CocrkoachDB and YDB: distributed transactions are not that expensive when you have a fast network.



# But when is PostgreSQL not enough?

1

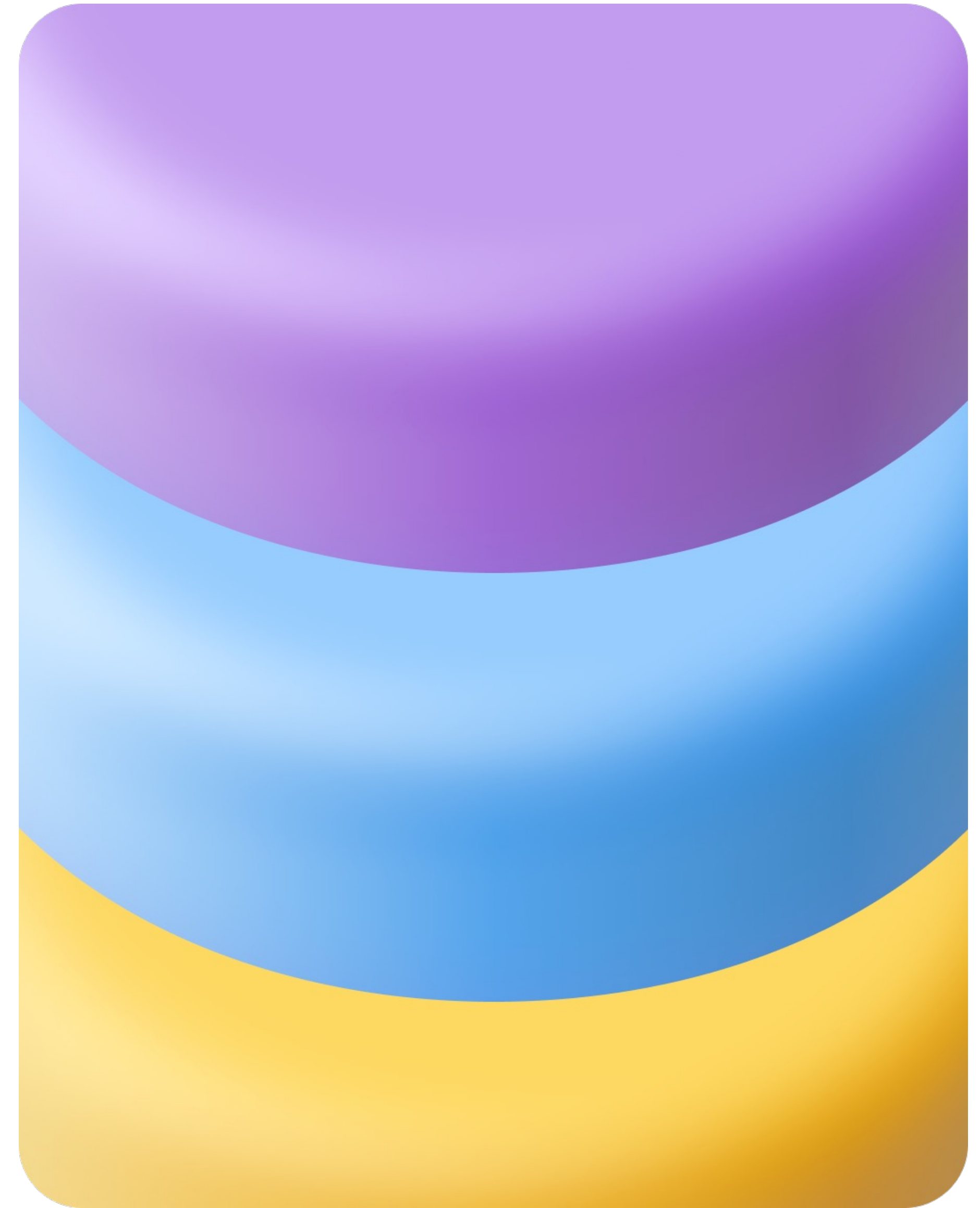
We took TPC-C – a very popular OLTP benchmark, 3 powerful servers, and found the limit when PostgreSQL fails to handle it

2

We evaluated the performance of distributed DBMSs compared to PostgreSQL in such a small installation

# TPC-C results

03





# TPC-C

Since 1992

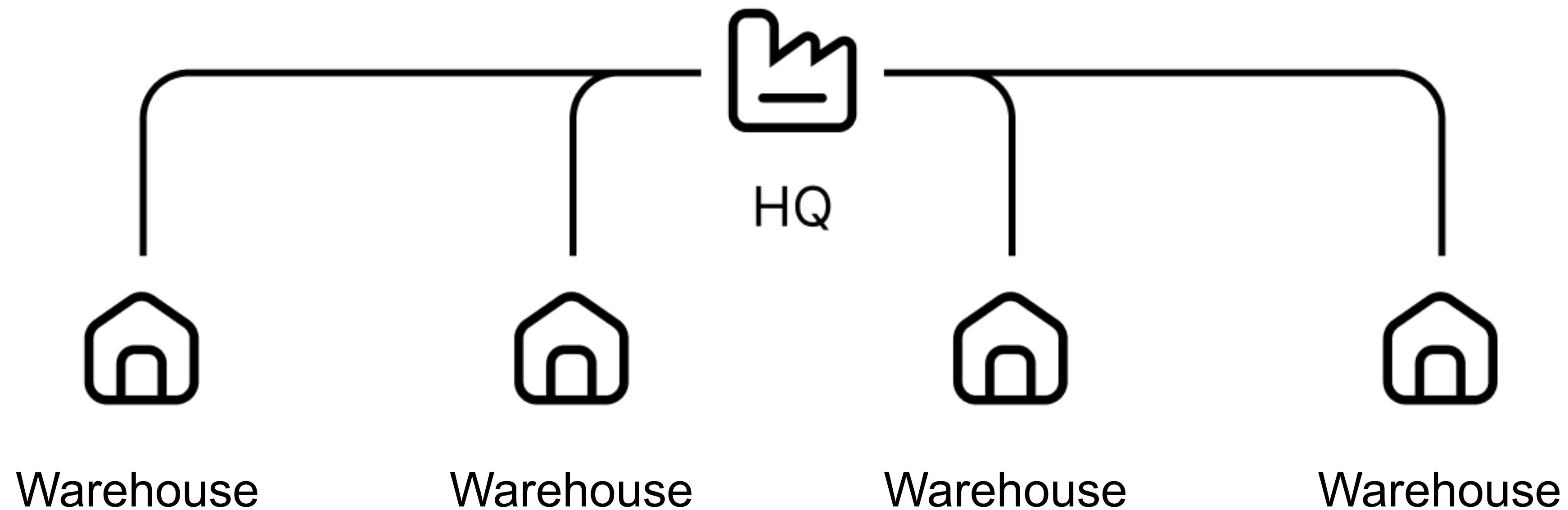
«The only objective comparison for evaluating OLTP performance» — CockroachDB

YugabyteDB and TiDB also stated that TPC-C is the most objective performance measurement of OLTP systems





# Simulates an e-commerce organization





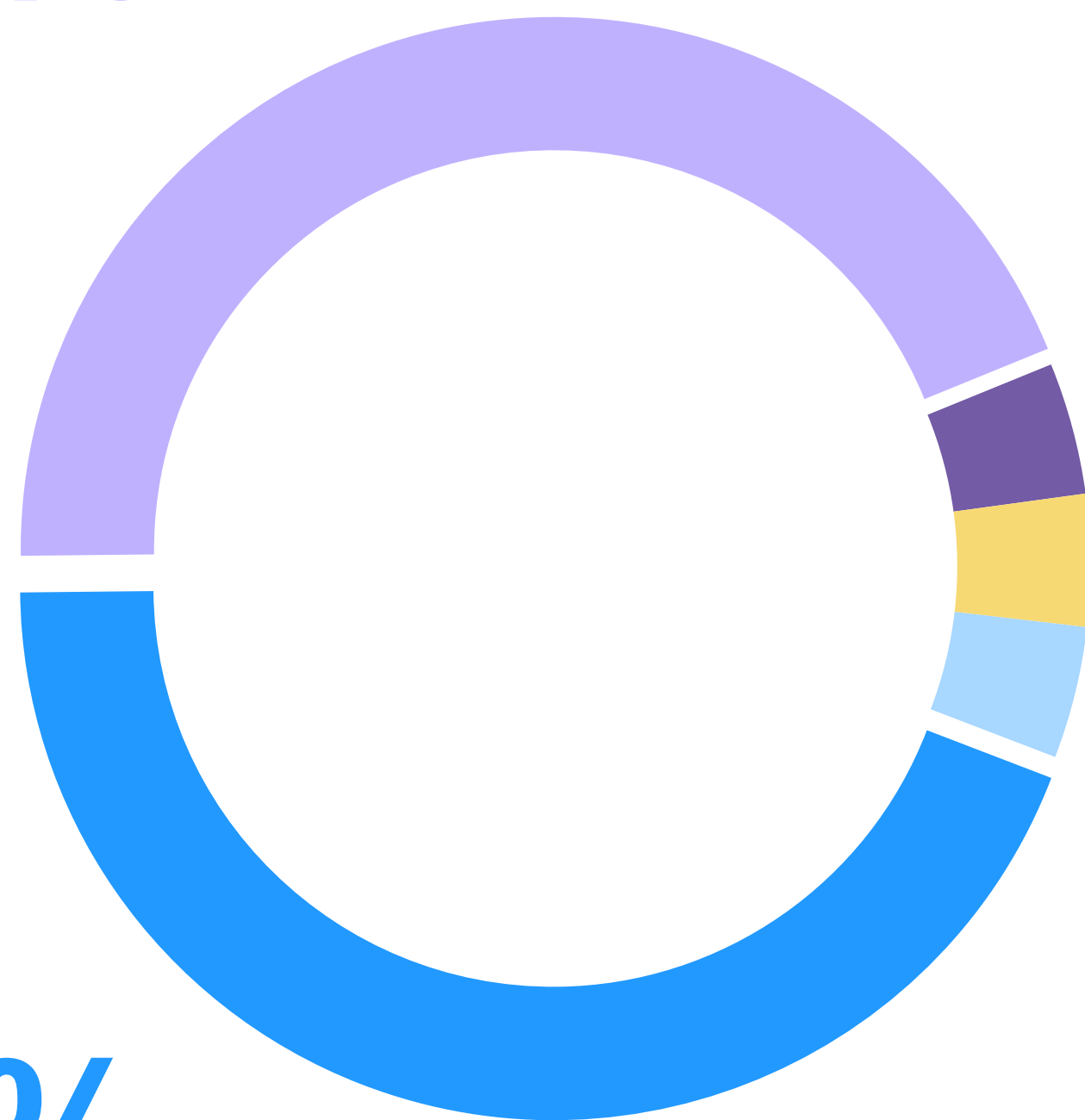
# TPC-C logic

- Number of warehouses is a parameter
- Each warehouse (around 100 MB of data) serves 10 districts
- Each district has a terminal
- Customers use a terminal for orders and payments
- Sometimes customers check the order status
- Delivery is handled by database as well
- Warehouses rarely make inventORIZATION

# TPC-C transactions

44%

44%



4%

4%

4%

NewOrder

Payment

OrderStatus

Delivery

StockLevel



# TPC-C transactions

**Serializable level of isolation**  
(repeatable read in Postgres is enough)

**1.9:1 read-to-write ratio**

**Multi-step (interactive)**

**tpmC integral metric:** benchmark measures the number of New Order transactions per minute

# CMU Benchbase



- Multi-DBMS SQL Benchmarking Framework via JDBC
- Developed by Carnegie Mellon under Andy Pavlo's supervision
- It's easy to add new DBMS and benchmarks

- The only well known TPC-C implementation
- YugabyteDB uses Benchbase fork
- We had to fork too (with a goal to upstream the YDB support)



# Client-side requirements for **15 000** warehouses

**150K**

**OS threads**

**600 GB**

**RAM**

To test YDB running on 3 servers, we used 5 servers to run the benchmark (each 128 cores and 512 GB RAM)



# Scaling out

- DBMS with 9, 15, 30, 60, 81 servers
- YDB, CockroachDB, YugabyteDB

**\$10,000**

**Single run in AWS**

**Multiple runs are usually required**





# Our fork and upstream

- [github.com/ydb-platform/tpcc](https://github.com/ydb-platform/tpcc) and [github.com/ydb-platform/tpcc-postgres](https://github.com/ydb-platform/tpcc-postgres)
- We plan to upstream the improvements
- We adapted TPC-C to Java virtual threads, which can lead to deadlocks in other benchmarks supported by Benchbase

[\[5\]](#) How we switched to Java 21 virtual threads and got a deadlock in TPC-C for PostgreSQL



# Tuning PostgreSQL



04



# Setup: 3 bare metal servers, single DC

## 128 logical CPU cores

Two Intel Xeon Gold 6338 CPU @ 2.00GHz,  
hyper-threading is turned on

---

Transparent hugepages  
(huge pages for PostgreSQL)

## 512 GB

RAM

---

Ubuntu 20.04.3 LTS

## 4 NVMe disks

RAID0 for PostgreSQL

# **DBMS should survive a single server failure**

**PostgreSQL has two sync replicas**

**CockroachDB and YDB use replication factor 3**



# In PostgreSQL, everything is configurable!

**1**

Write-ahead  
log

**2**

B-Tree

**3**

Execution  
engine

**4**

Replication

**5**

I/O

# Our approach to tuning

From fault-intolerant and extremely fast to slower, but fault-tolerant PostgreSQL

Three NVMe RAID0 — data, One NVMe — WAL:

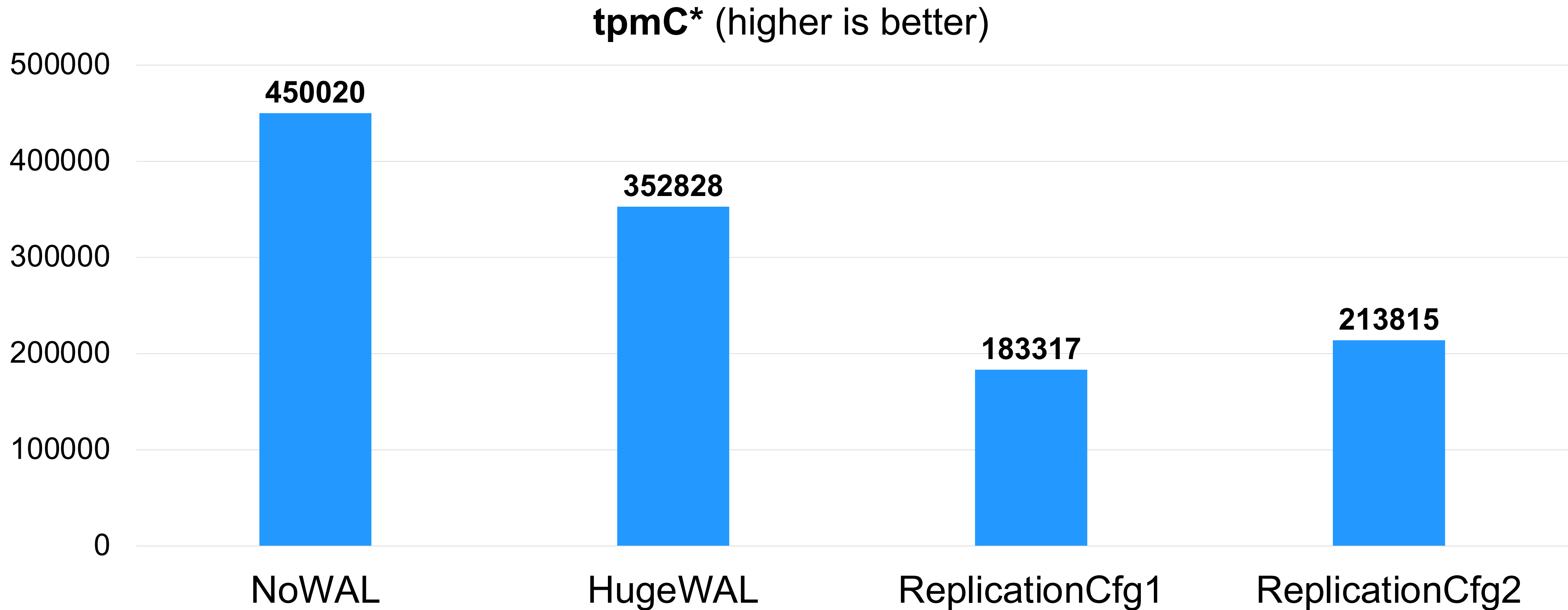
1. Unlogged tables with replication turned off: **NoWAL**
2. Huge WAL (Recovery time is tens of minutes) with ideal I/O distribution: **HugeWAL**
3. Two sync replicas: **ReplicationCfg1**

Two NVMe RAID0 — data, two NVMe RAID0 — WAL:

4. Two sync replicas with `synchronous_commit = apply`: **ReplicationCfg2**



# PostgreSQL configurations evaluation



*\* The results are not officially recognized TPC results and are not comparable with other TPC-C test results published on the TPC website.*

# Results summary

**1** Fault-intolerant **PostgreSQL** is **incredibly fast**

**2** With replication, the result is twice as slow, but still good

**3** PostgreSQL replicas use only one thread to apply the WAL

**4** Synchronous replication in PostgreSQL is a bottleneck and limits vertical scalability

[\[6\]](#) More details on configurations and results.



# Is 200K tpmC a lot?

**~8 000**

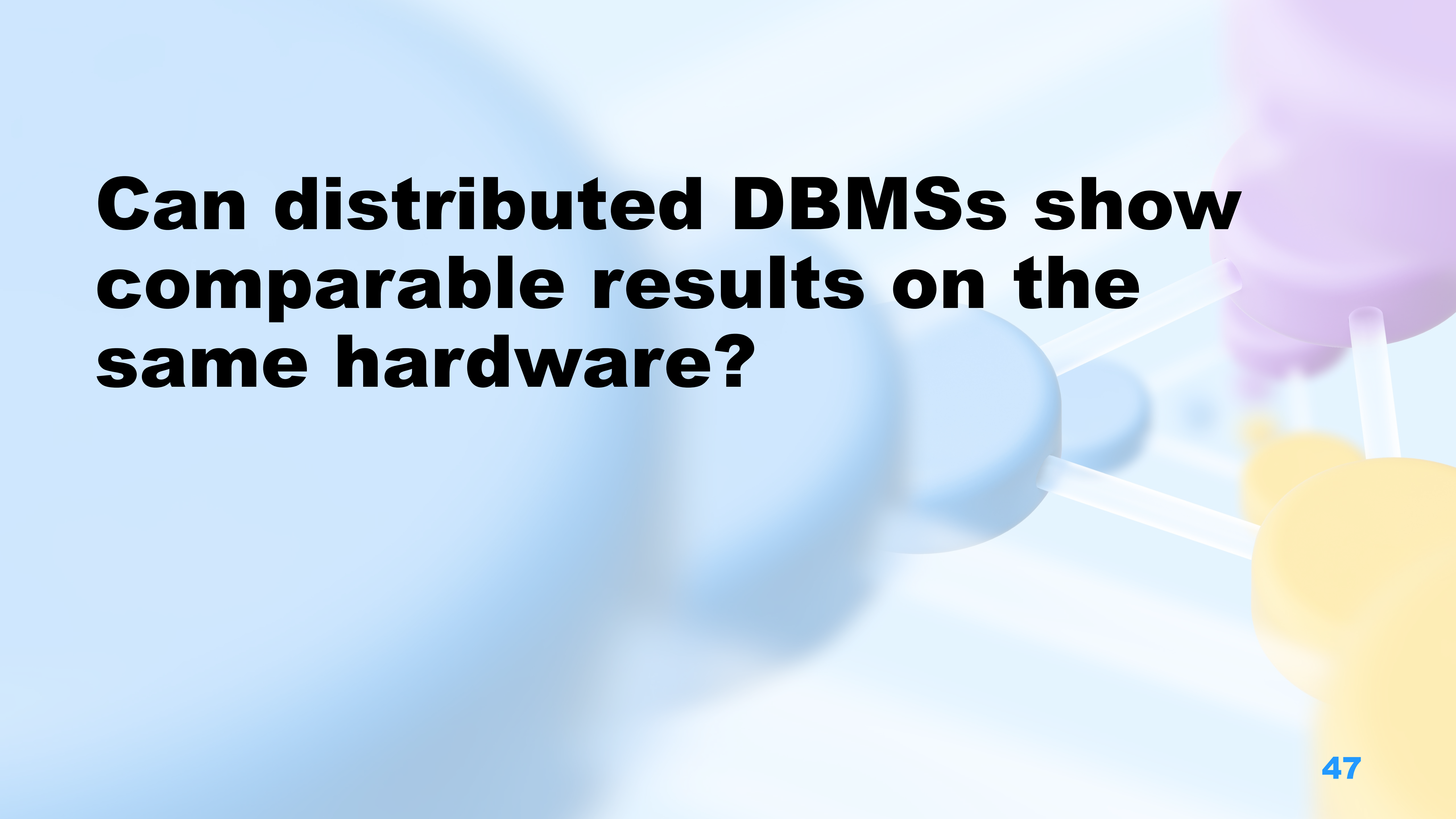
interactive transactions per second

**~130 000**

database requests (queries) per second

## Leader server:

- WAL write 400 MB/s,
- data write 600 MB/s
- read 700 MB/s
- network consumption 9 Gbit/s
- CPU usage: on average 20 cores (out of 128)



**Can distributed DBMSs show comparable results on the same hardware?**



# PostgreSQL vs. distributed DBMSs



*vs.*



# YDB

## Open-Source Distributed SQL Database

**1** Partial PostgreSQL compatibility [\[7\]](#).

OLTP, OLAP, Kafka-like topics

Transactions between topics and tables

**2** Strong consistency

**3** Clusters with thousands of servers

**4** Apache 2.0 license

**5** Star [ydb-platform](#) on GitHub



# CockroachDB

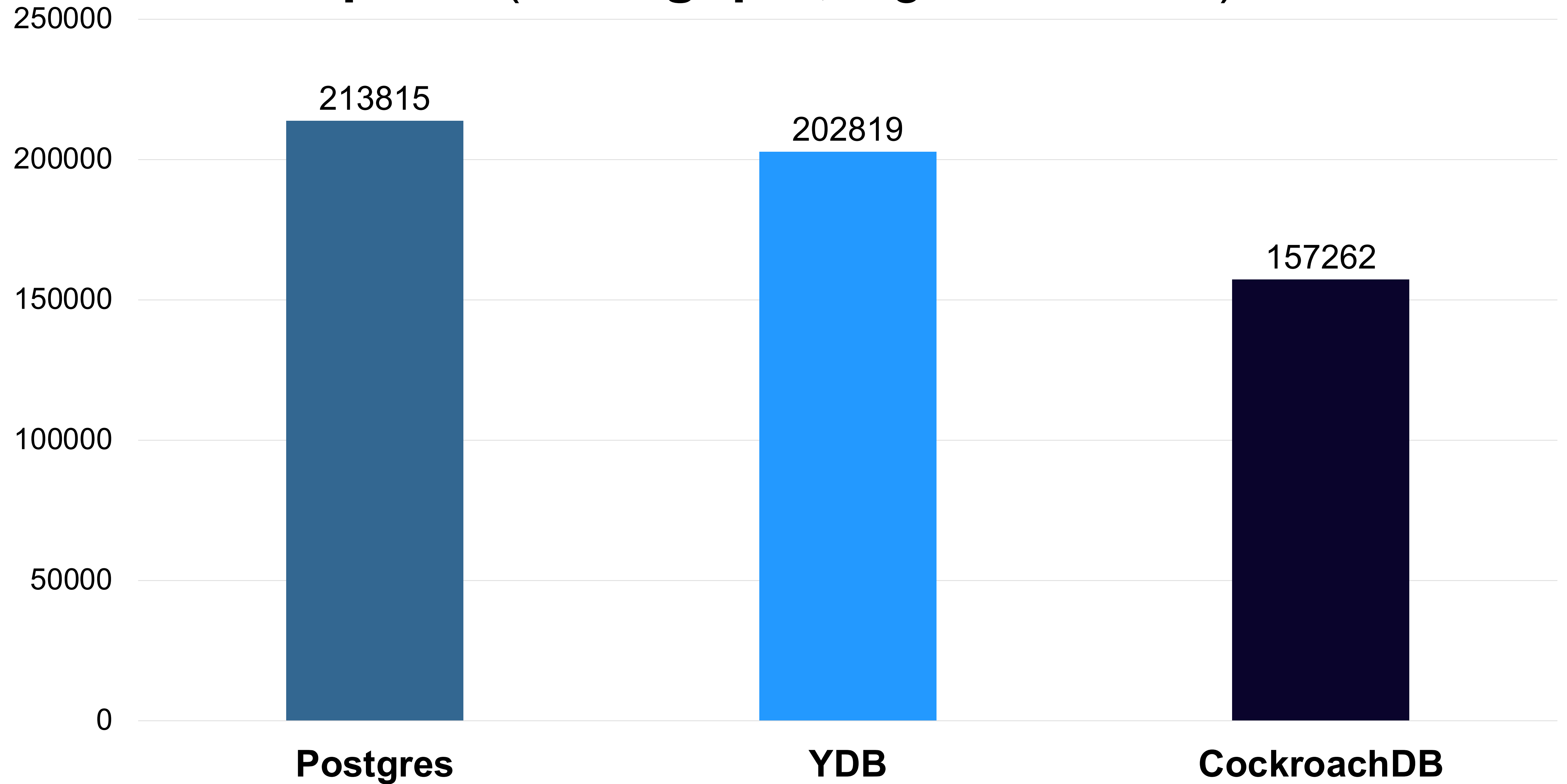
## ~~Open-Source~~ Distributed SQL Database

**1** Partial PostgreSQL  
compatibility

OLTP only

**2** Strong consistency

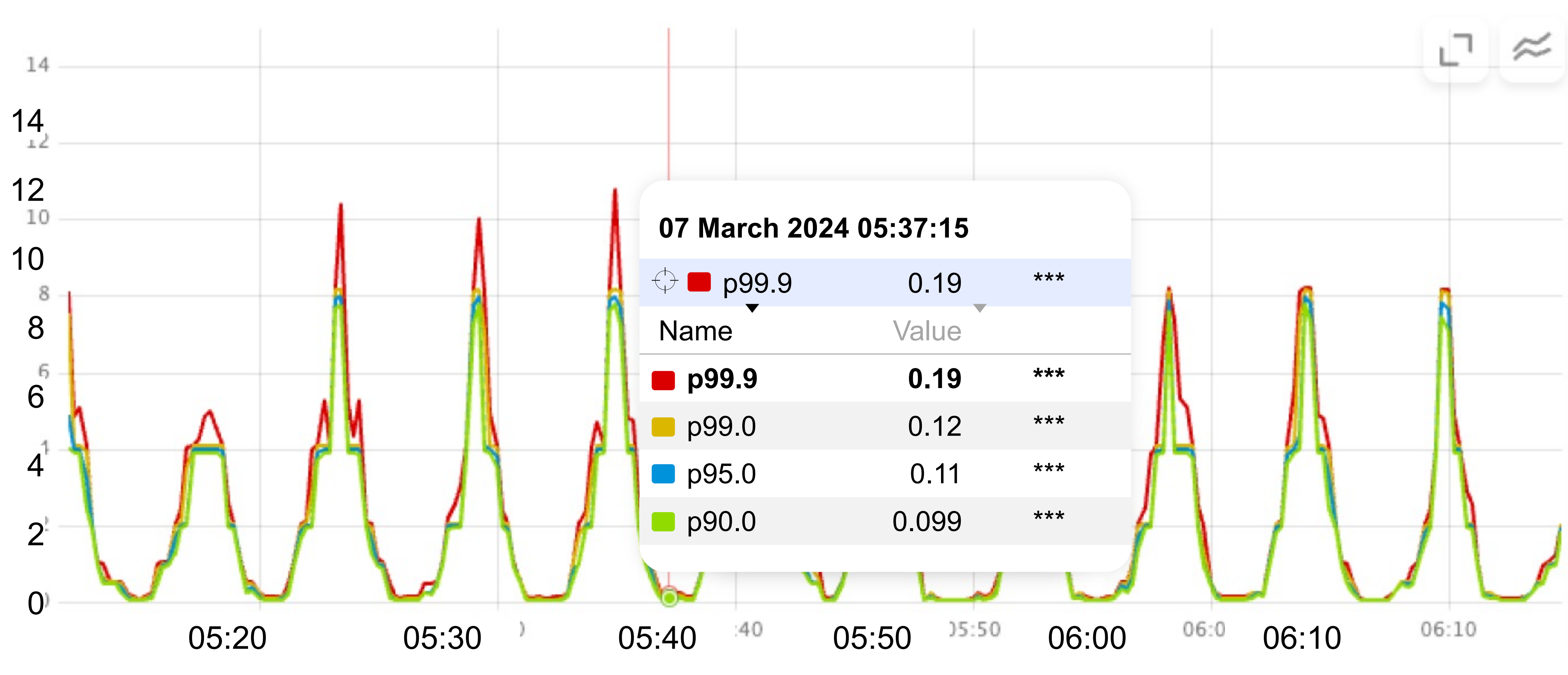
# tpmC\* (throughput, higher is better)



*\* The results are not officially recognized TPC results and are not comparable with other TPC-C test results published on the TPC website.*



# Postgres NewOrder Latencies\*, seconds (lower is better)



\* The results are not officially recognized TPC results and are not comparable with other TPC-C test results published on the TPC website.

# NewOrder latency in Postgres

Each peak corresponds to the start of a checkpoint

Sessions are 'hanging' waiting for IPC: SyncRep

This is an architectural issue (only 1 thread for receiving and applying WAL by replicas)



# Conclusions

- 1** PostgreSQL is highly efficient, but:
  1. It does not scale horizontally.
  2. Synchronous replication limits vertical scaling and it's not always possible to just add more cores and RAM.
- 2** Citus-like solutions are not ACID-compliant and do not provide the same guarantees as PostgreSQL in case of multi-shard (distributed) transactions.
- 3** When you need serializable distributed transactions, consider distributed DBMSs: they are more efficient than commonly believed.

# Questions?

**Evgenii Ivanov (twitter: @eivanov89),**  
Principal Software Developer, YDB



Slides and materials