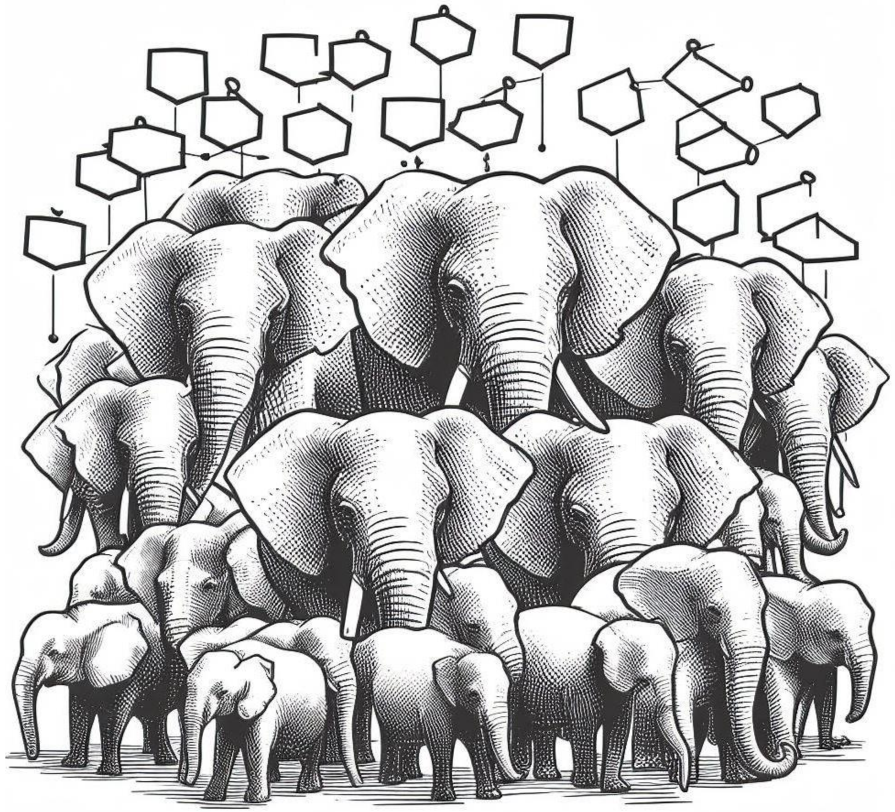
PostgreSQL Distributed

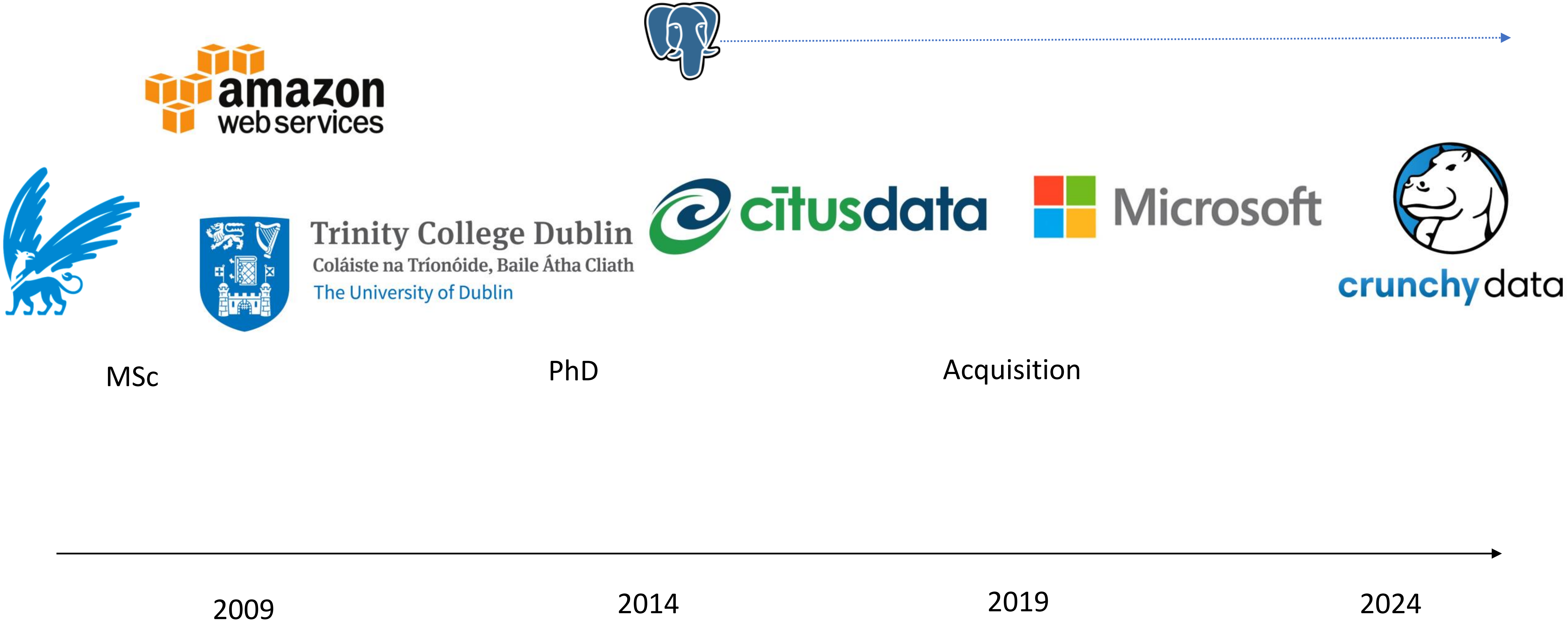
An overview of architectures and trade-offs



Marco Slot - marco.slot@gmail.com

Formerly: founding engineer at Citus Data, architect at Microsoft

My career timeline



MSc

PhD

Acquisition

2009

2014

2019

2024

My background in Distributed PostgreSQL

Developed Citus since 2014 https://github.com/citusdata/citus

Citus is a PostgreSQL *extension* that adds the ability to transparently distribute or replicate tables across a cluster of PostgreSQL servers.

See “Citus: Distributed PostgreSQL for Data-Intensive Applications”

- SIGMOD ’21

Also pg_cron, azure_storage, other PostgreSQL extensions.

Today’s talk on PostgreSQL Distributed

Many distributed database talks discuss algorithms for distributed query planning, transactions, etc.

In distributed systems, trade-offs are more important than algorithms.

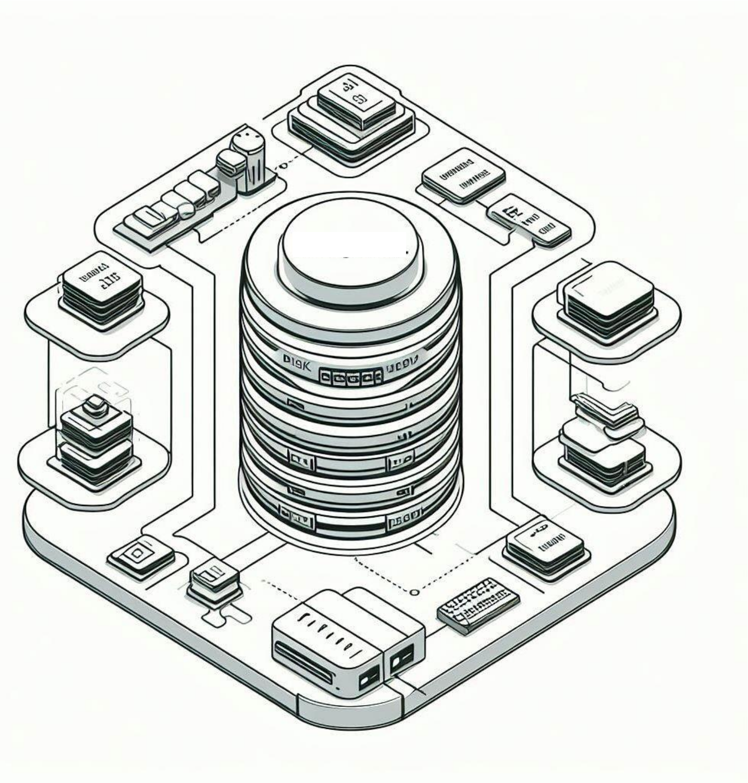
Vendors and even many papers rarely talk about trade-offs.

Moreover, many different PostgreSQL distributed system architectures with different trade-offs exist.

Experiment: Discuss PostgreSQL distributed systems architecture trade-offs by example.

Single machine PostgreSQL

PostgreSQL on a single machine can be incredibly fast



No network latency

Millions of IOPS

Microsecond disk latency

Low cost / fast hardware

Can co-locate application server

Single machine PostgreSQL?

PostgreSQL on a single machine comes with operational hazards

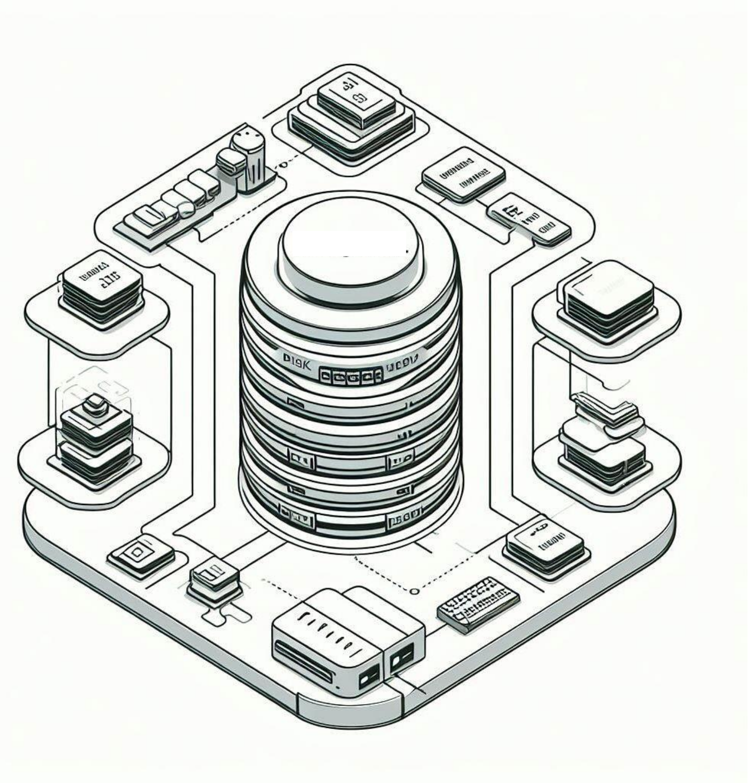
Machine/DC failure (downtime)

Disk failure (data loss)

System overload (difficult to scale)

Accidental data deletion (need backups)

Disk full (downtime)



PostgreSQL Distributed (in the cloud)

Fixing the operational hazards of single node PostgreSQL requires a distributed set up.

The cloud enables flexible distributed set ups, with resources shared between customers for high efficiency and resiliency.

Goals of distributed database architecture

Goal: Offer same functionality and transactional semantics as single node RDBMS, with superior properties

Mechanisms:

Replication	- Place copies of data on different machines
Distribution	- Place partitions of data on different machines
Decentralization	- Place different DBMS activities on different machines

Reality: Concessions in terms of performance, transactional semantics, functionality, and/or operational complexity

PostgreSQL Distributed Layers

Distributed architectures can hook in at different layers — many are orthogonal!

Client	Manual sharding, load balancing, write to multiple endpoints
Pooler	Load balancing and sharding (e.g. pgbouncer, pgcat)
Query engine	Transparent sharding (e.g. Citus, Aurora limitless)
Logical data layer	Active-active, federation (e.g. BDR, postgres_fdw)
Storage manager	DBMS-optimized cloud storage (e.g. Aurora, Neon)
Data files, WAL	Read replicas, hot standby
Disk	Cloud block storage (e.g. Amazon EBS, Azure Premium SSD)

Practical view of Distributed PostgreSQL

Today we will cover:

- Network-attached cloud storage
- Read replicas & hot standby
- DBMS-optimized cloud storage
- Transparent Sharding
- Active-active deployments
- Distributed key-value stores with SQL

Two questions:

1) What are the trade-offs?

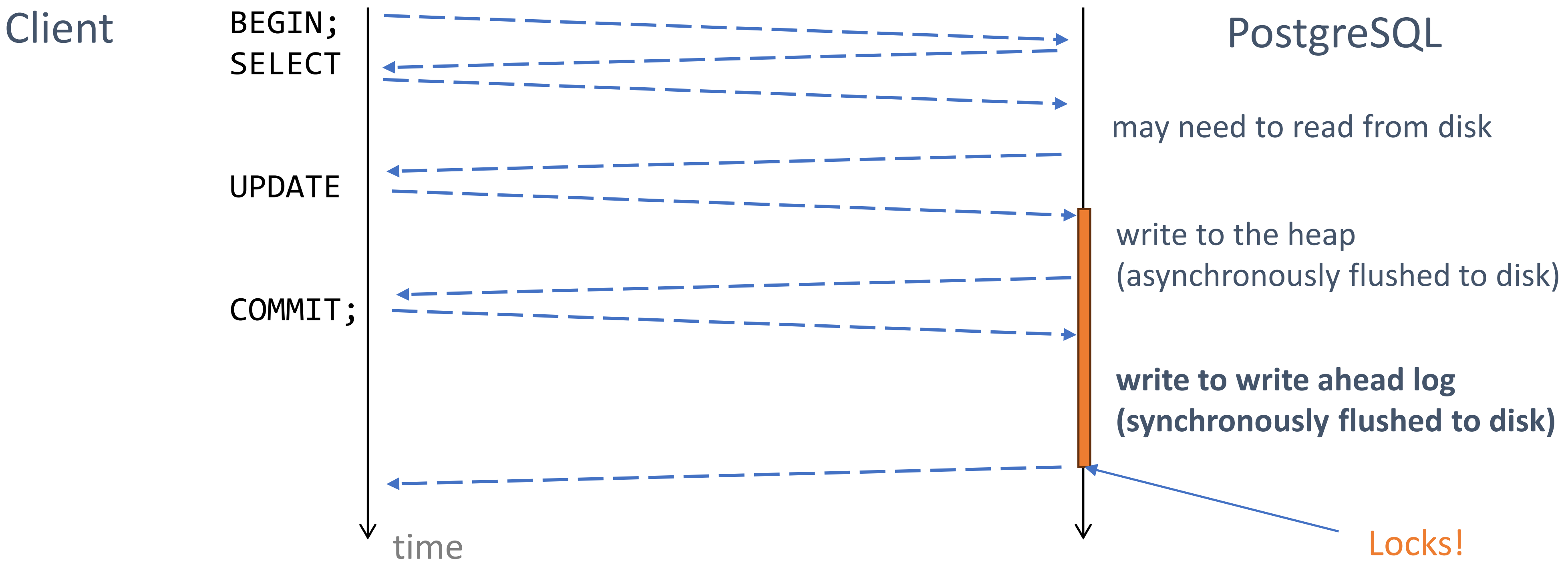
Latency, Efficiency, Cost, Scalability, Availability, Consistency, Complexity, …

2) For which workloads?

Lookups, analytical queries, small updates, large transforms, batch loads, …

The perils of latency: Synchronous protocol

Transactions are performed step-by-step on each session.



Client

BEGIN;
SELECT

UPDATE

COMMIT;

time

PostgreSQL

may need to read from disk

write to the heap
(asynchronously flushed to disk)

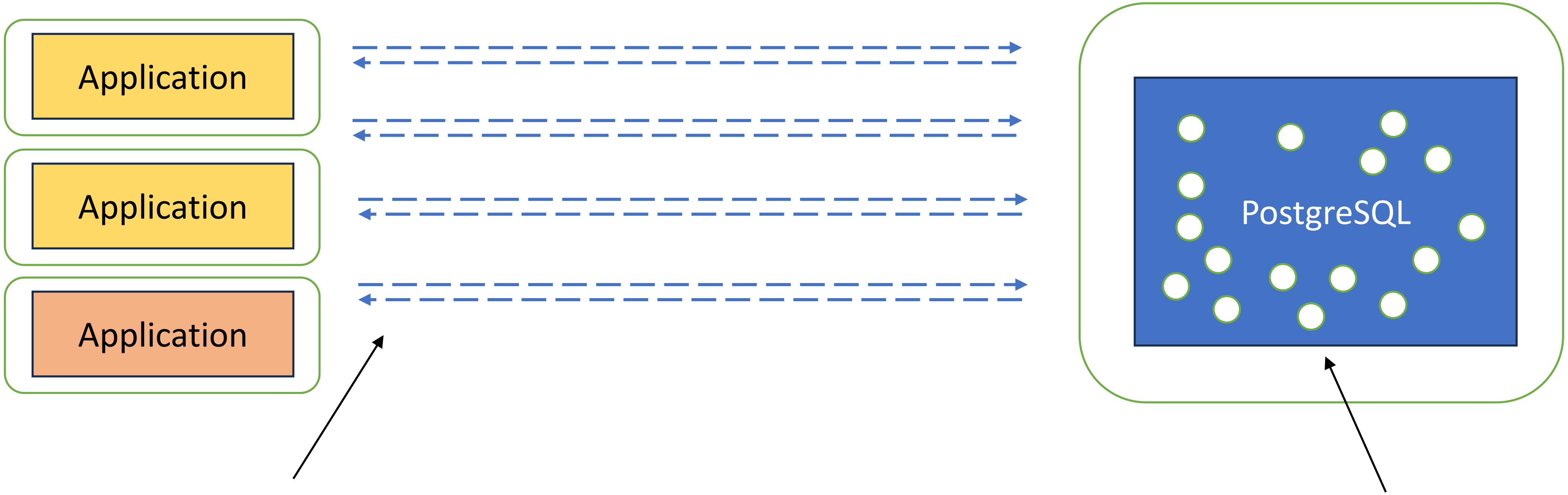
write to write ahead log
(synchronously flushed to disk)

Locks!

Max throughput per session = 1 / avg. response time

The perils of latency: Connection limits

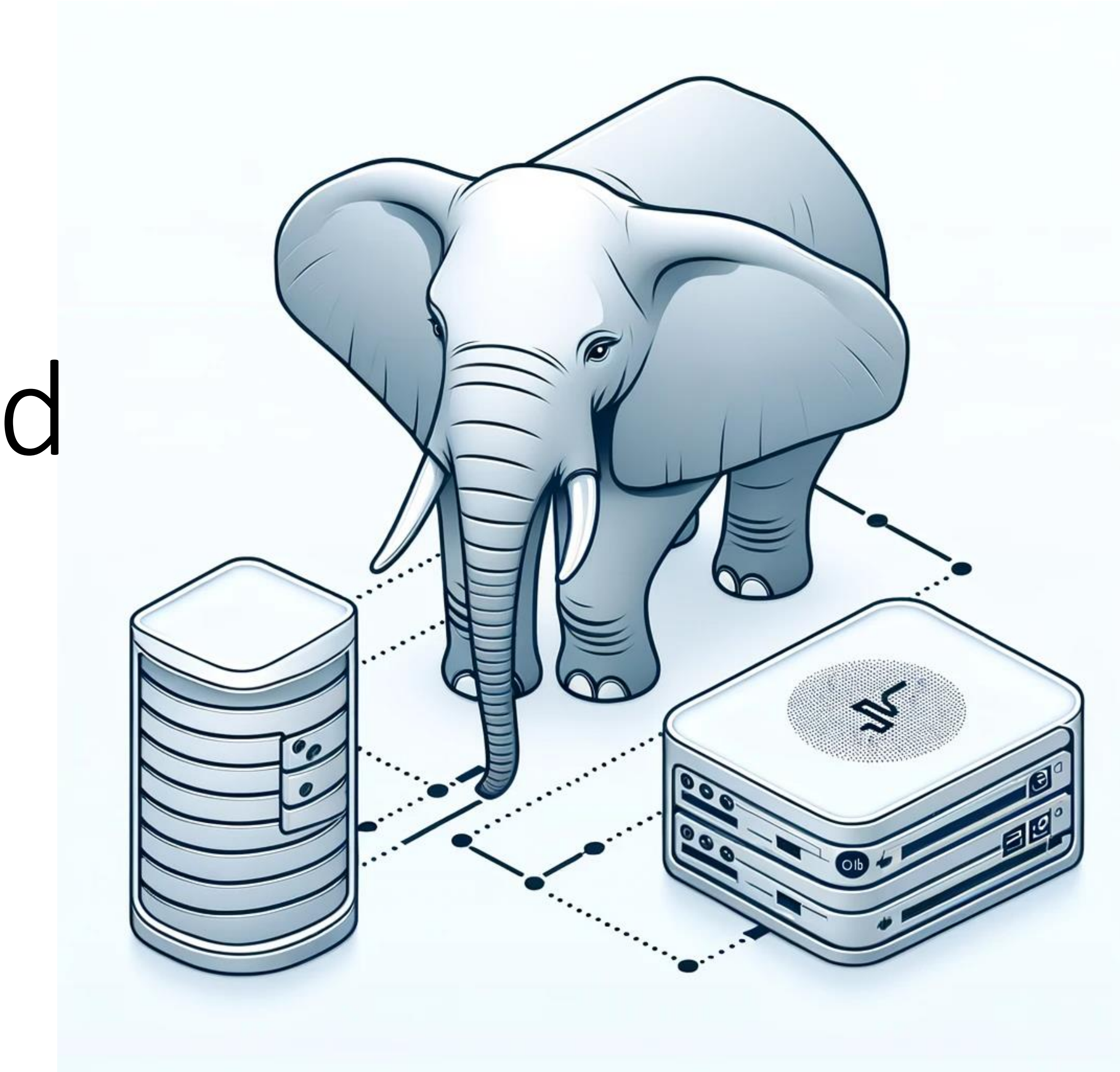
Max overall throughput: #sessions / avg.response time



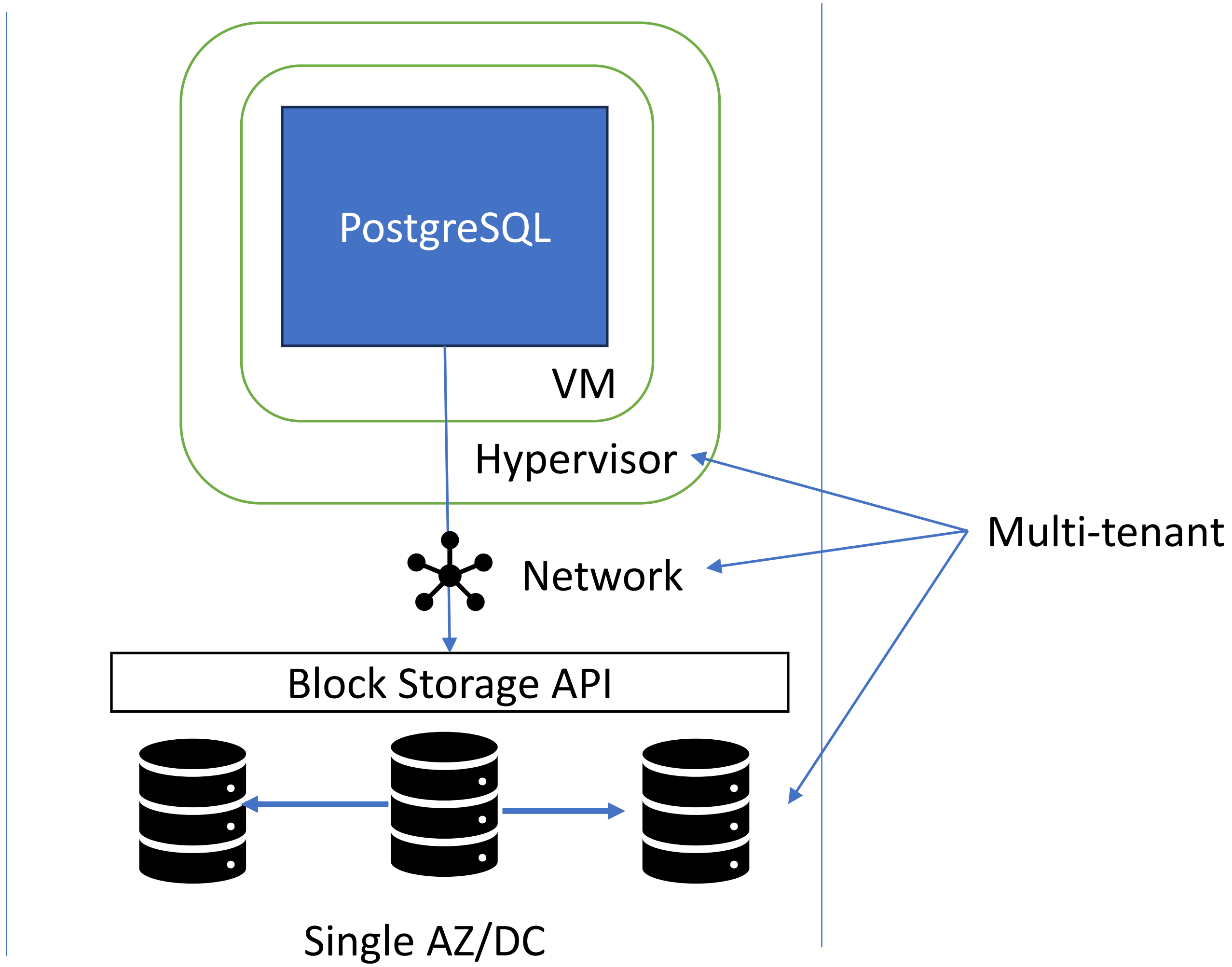
Number of connections limited by app architecture

Number of processes limited by memory, contention

Network-attached block storage



Network-attached block storage



Network-attached storage

Pros:

Higher durability (replication)

Higher uptime (replace VM, reattach)

Fast backups and replica creation (snapshots)

Disk is resizable

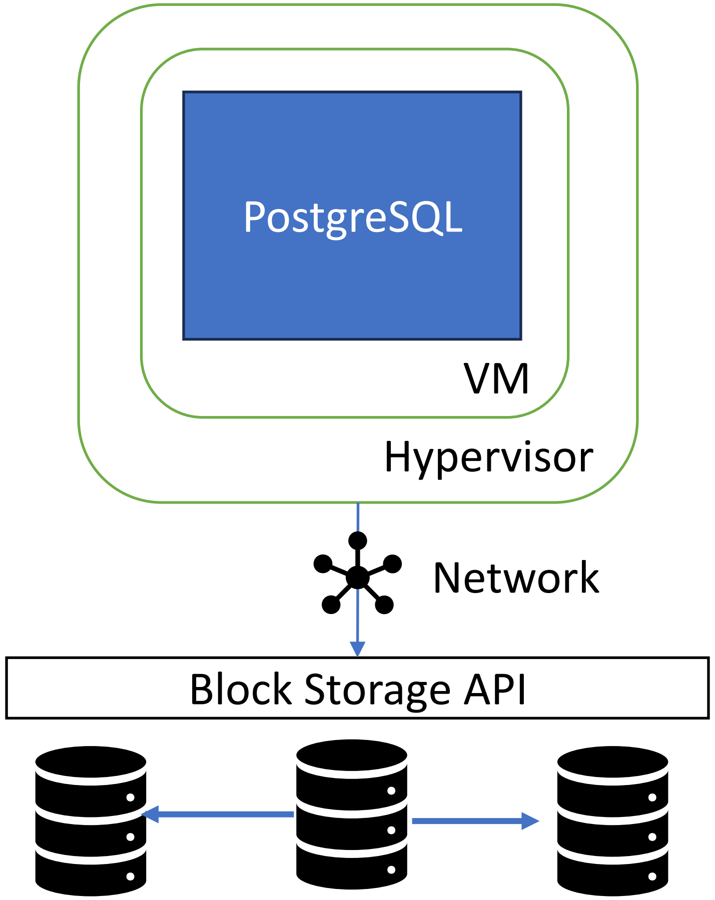
Cons:

Higher disk latency, esp. for writes (~10µs -> ~1000µs)

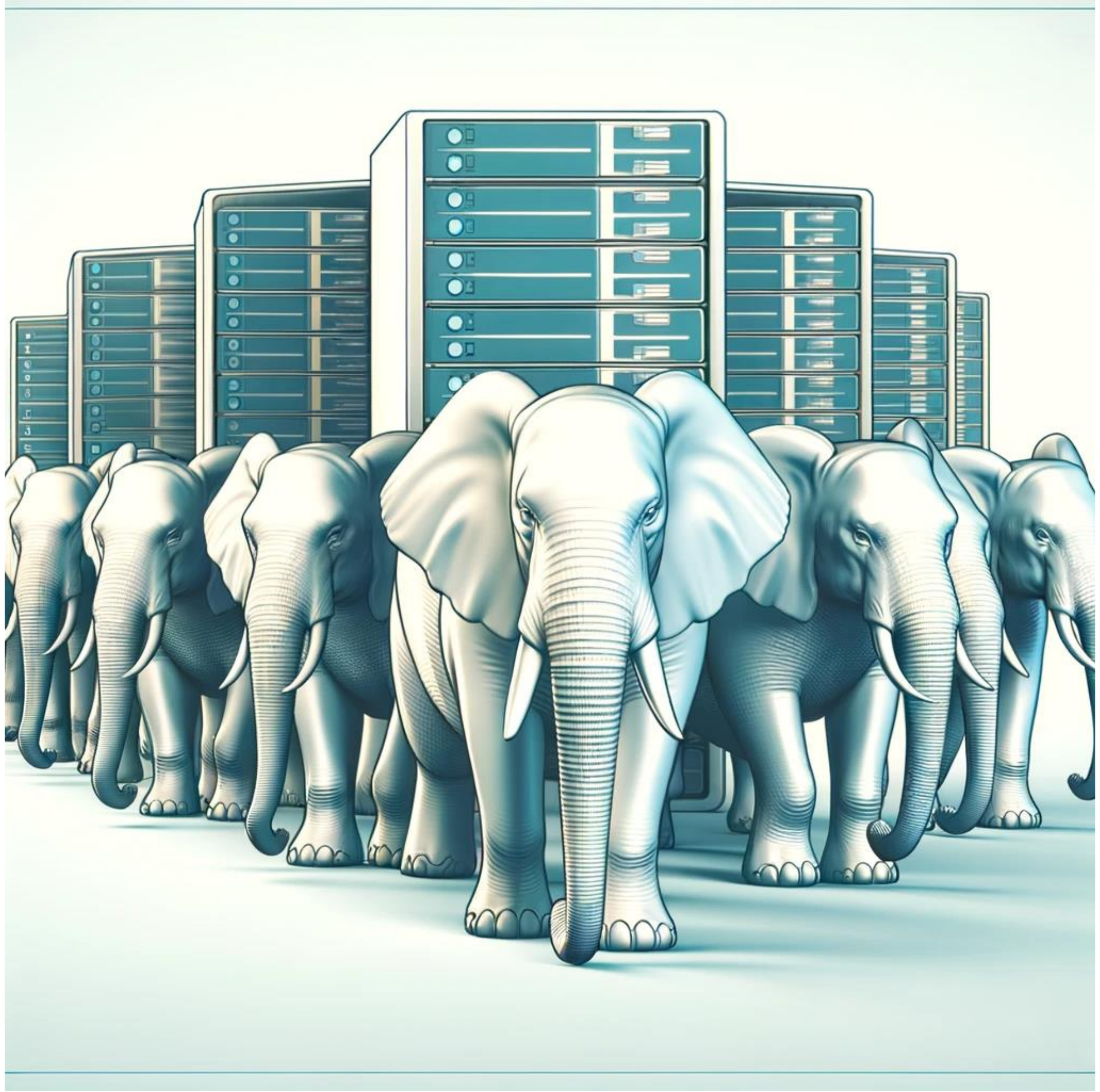
Lower IOPS (~1M -> ~10k IOPS)

Crash recovery on restart takes time

Cost can be high

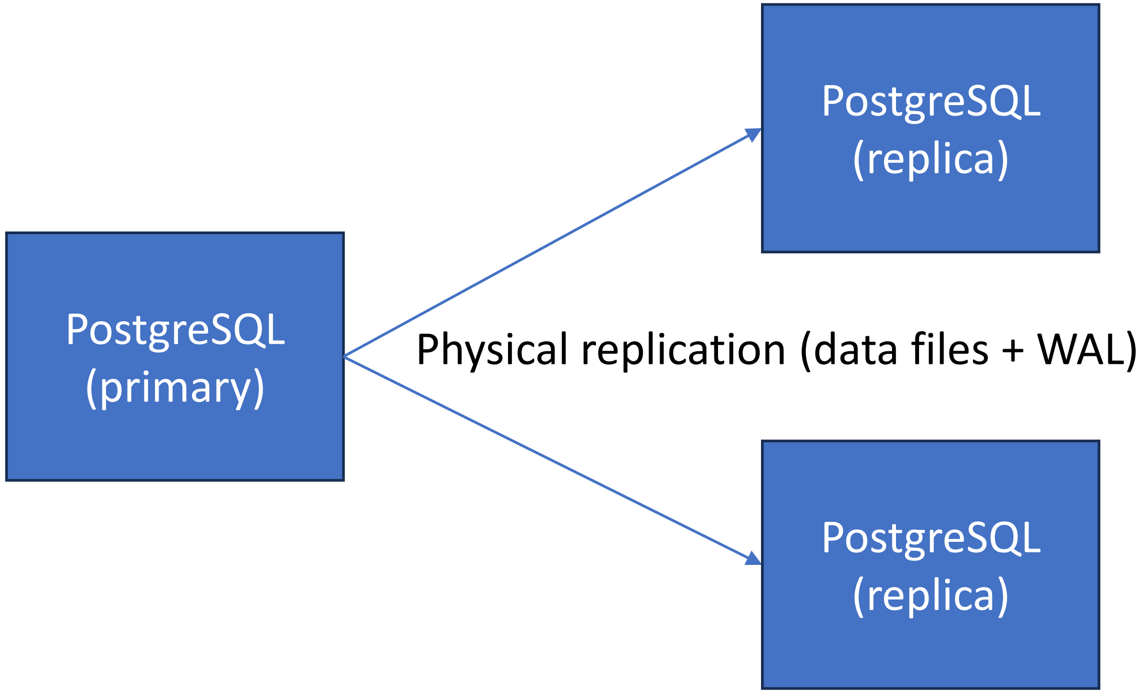


Read replicas



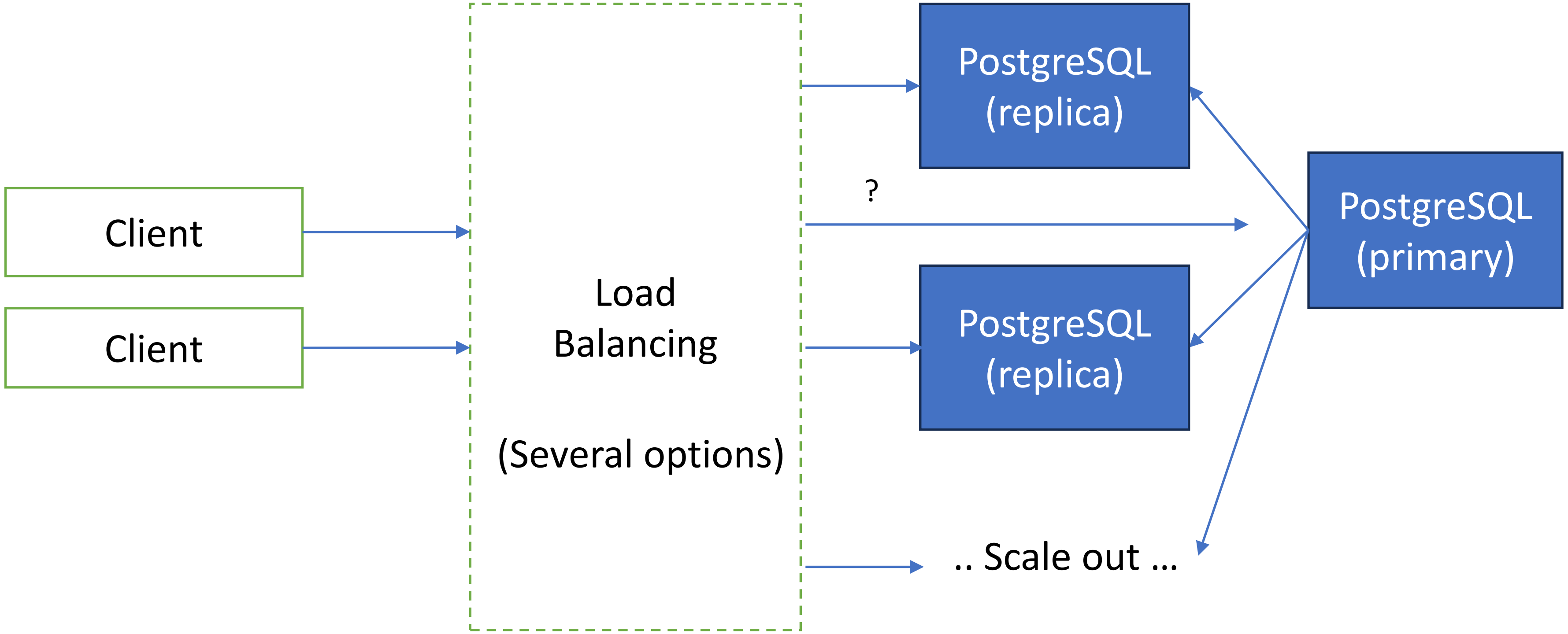
Read replicas

Readable replicas can help you scale read throughput, reduce latency through cross-region replication, improve availability through auto-failover.



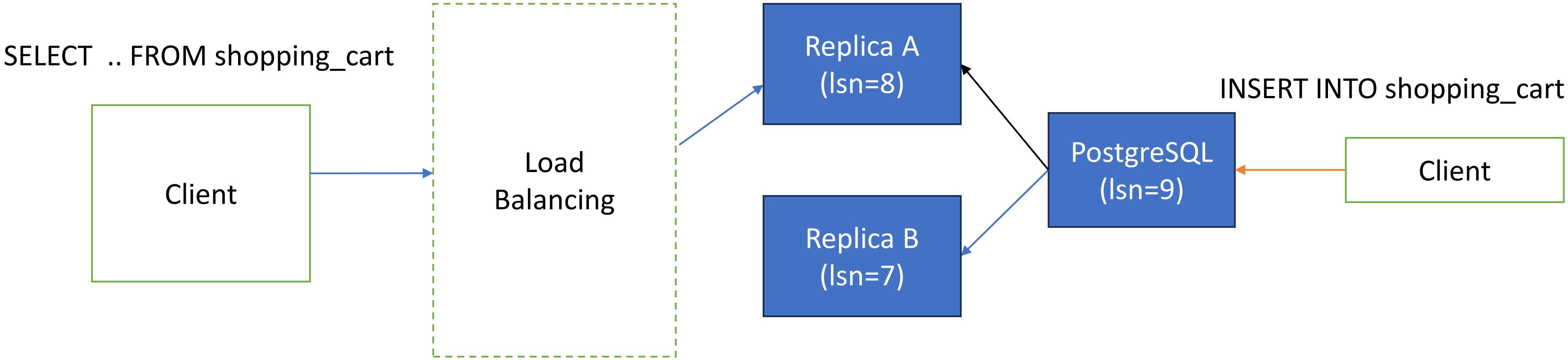
Scaling read throughput

Readable replicas can help you scale read throughput (when reads are CPU or I/O bottlenecked) by load balancing queries across replicas.



Eventual read-your-writes consistency

Read replicas can be behind on the primary, cannot always read your writes.



SELECT .. FROM shopping_cart

Client

Load Balancing

Replica A (lsn=8)

Replica B (lsn=7)

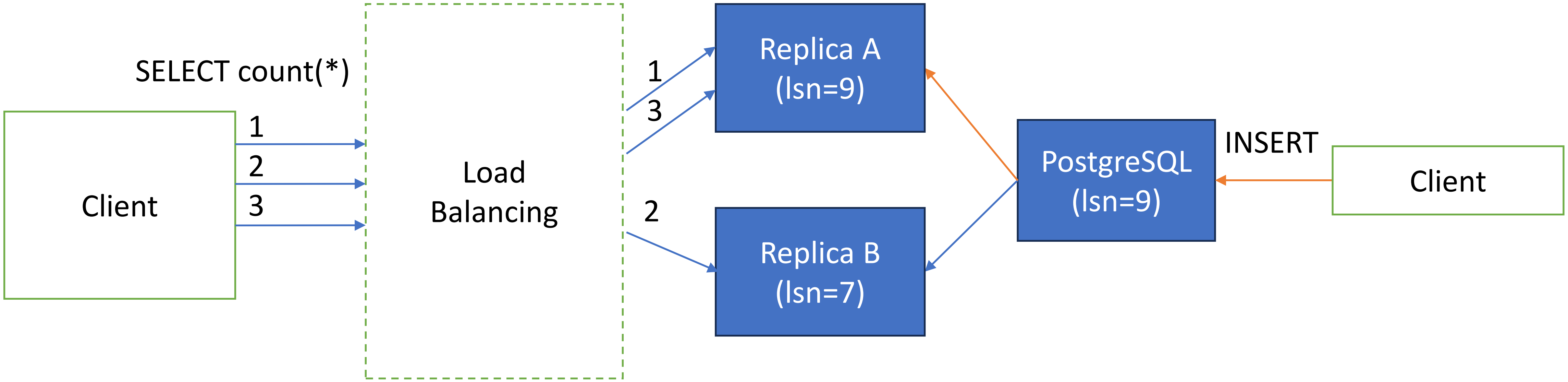
PostgreSQL (lsn=9)

INSERT INTO shopping_cart

Client

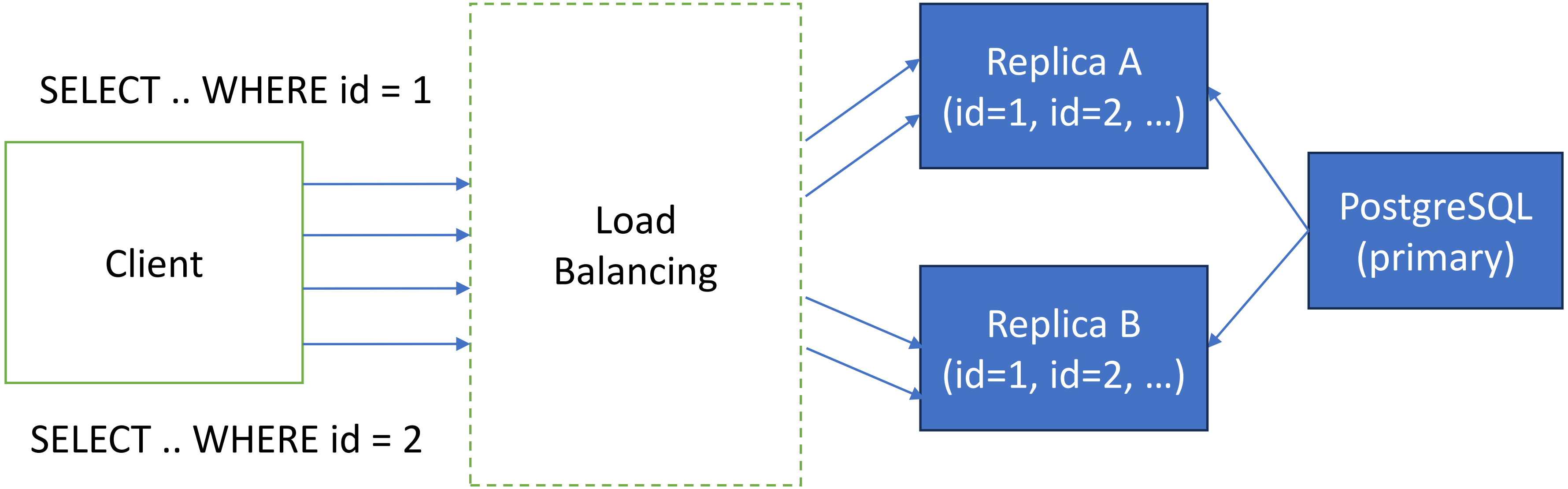
No monotonic read consistency

Load-balancing across read replicas will cause you to go back-and-forth in time.



Poor cache usage

If all replicas are equal, they all have the same stuff in cache



If working set >> memory, all replicas get bottlenecked on disk I/O.

Read scaling trade-offs

Pros:

Read throughput scales linearly

Low latency stale reads if read replica is closer than primary

Lower load on primary

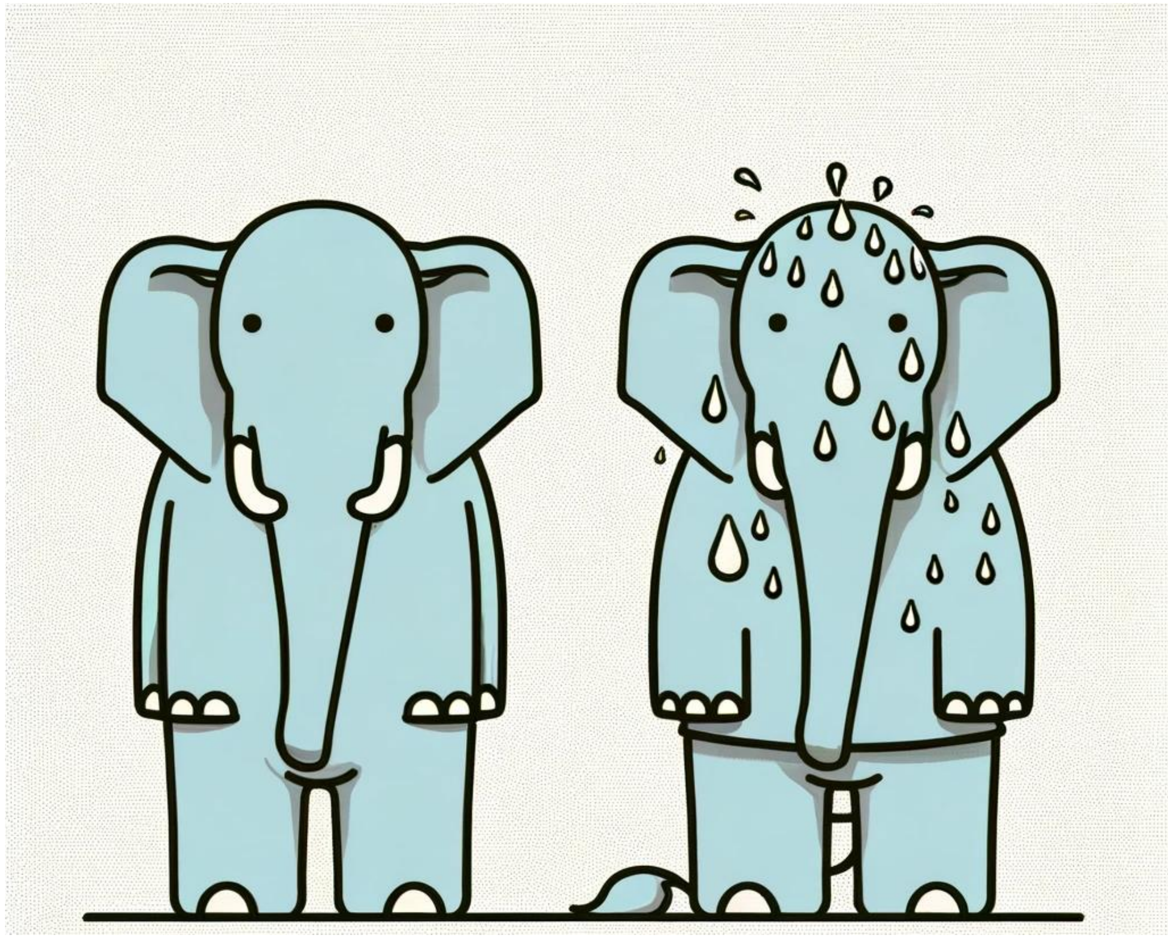
Cons:

Eventual read-your-writes consistency

No monotonic read consistency

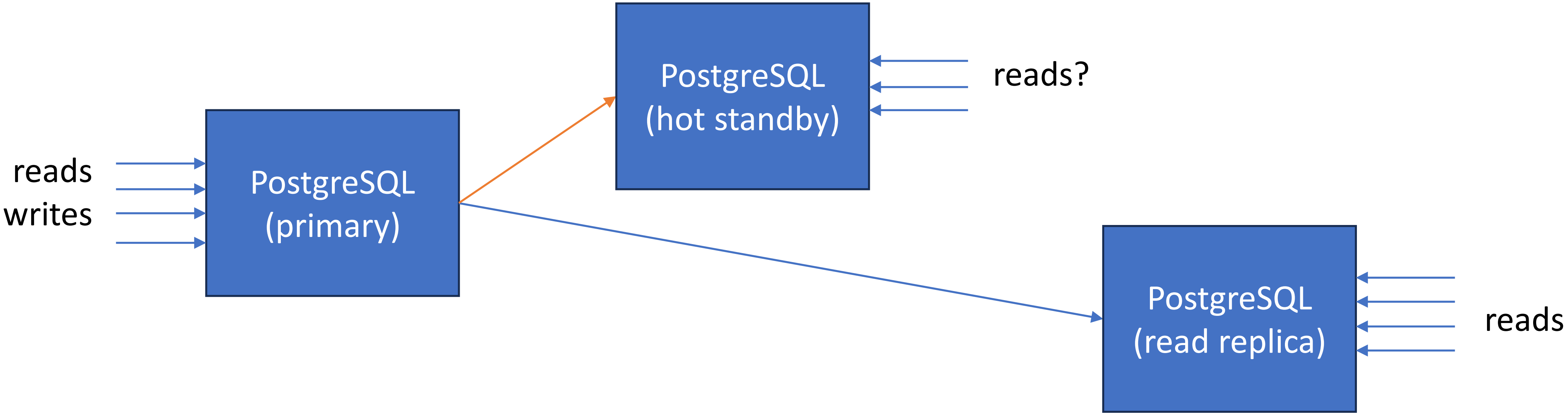
Poor cache usage

Hot standby



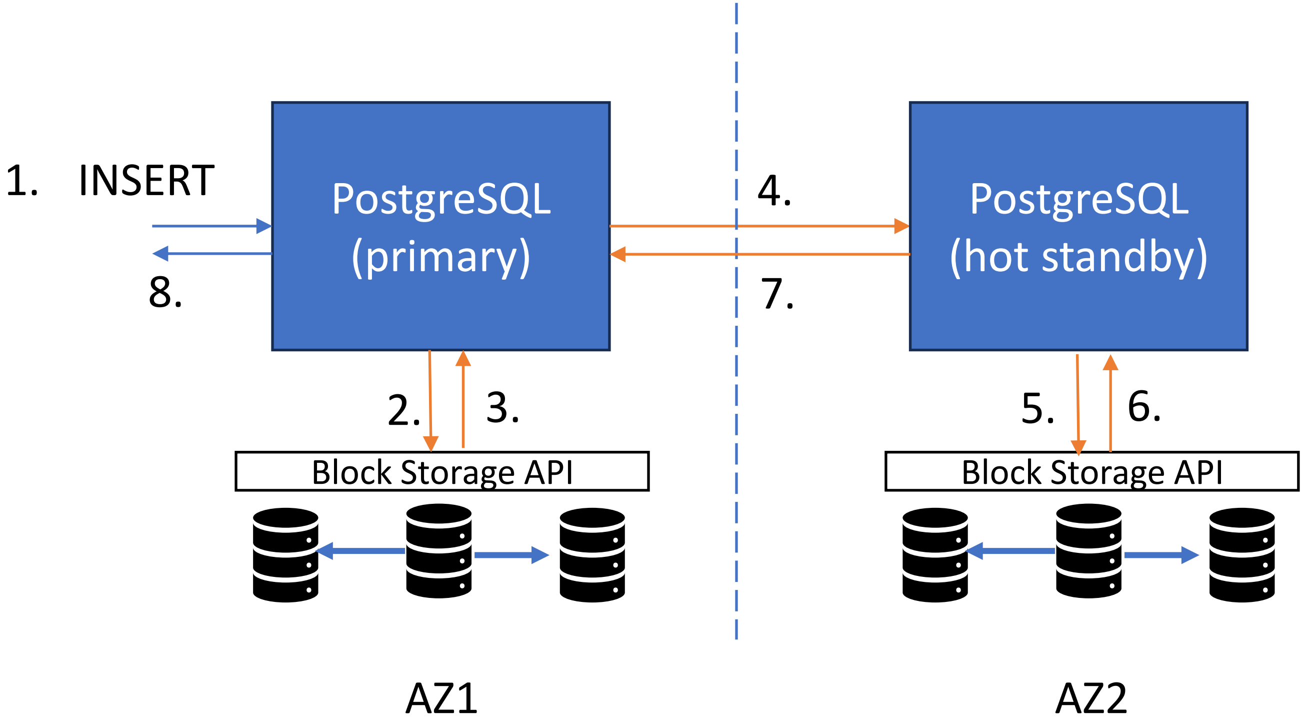
Read replica vs. hot standby

“Hot standby” refers to a read replica that is a candidate for auto-failover, can also serve reads.



Hot standby write latency

PostgreSQL first writes to primary WAL and then waits for replication to standby WAL.



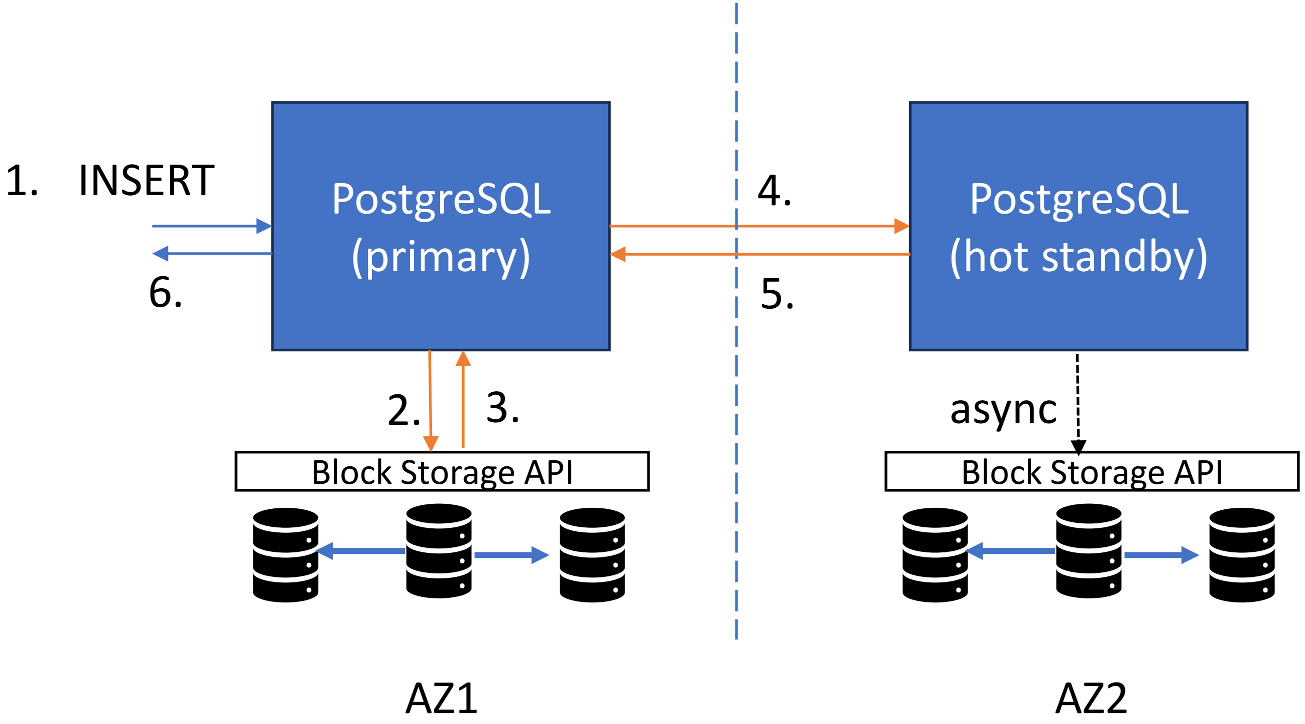
`synchronous_commit = on`

Batching writes is helpful!

Use COPY, multi-row inserts, etc.

Hot standby write latency

PostgreSQL first writes to primary WAL and then waits for replication to standby WAL.



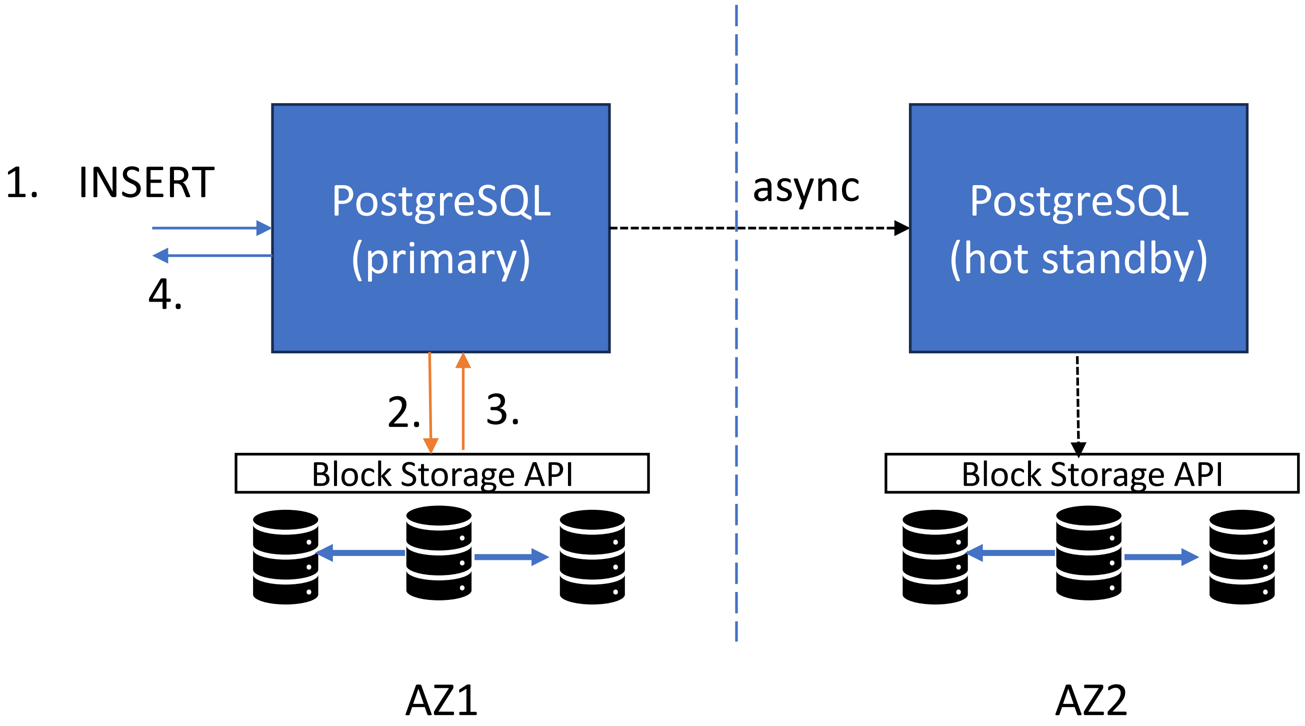
synchronous_commit = remote_write

Some data lost only when simultaneous failure of primary and hot standby.

`synchronous_commit = remote_write`

Hot standby write latency

PostgreSQL first writes to primary WAL and then waits for replication to standby WAL.



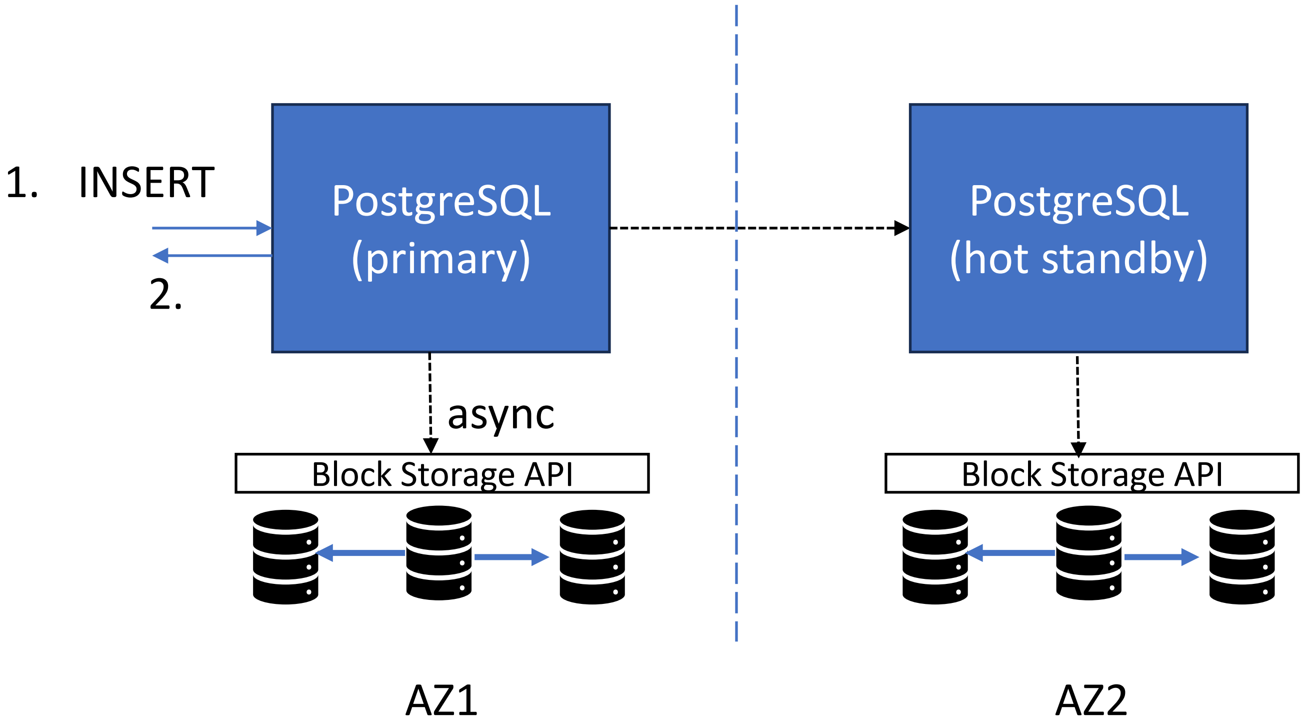
synchronous_commit = local

Some data lost when failover is triggered.

`synchronous_commit = local`

Hot standby write latency

PostgreSQL first writes to primary WAL and then waits for replication to standby WAL.



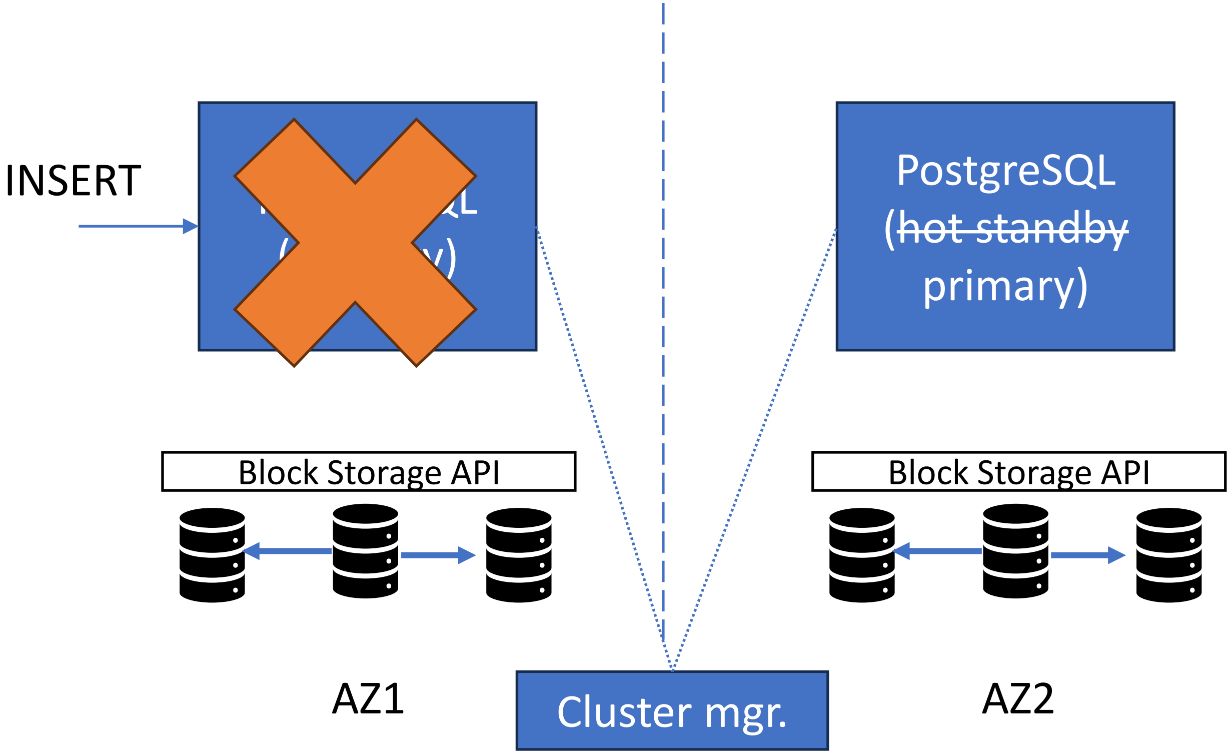
synchronous_commit = off

Some data lost when primary fails.

`synchronous_commit = local`

Primary failure

When a primary fails from the cluster manager point-of-view, it initiates failover.



In a few seconds to minutes:

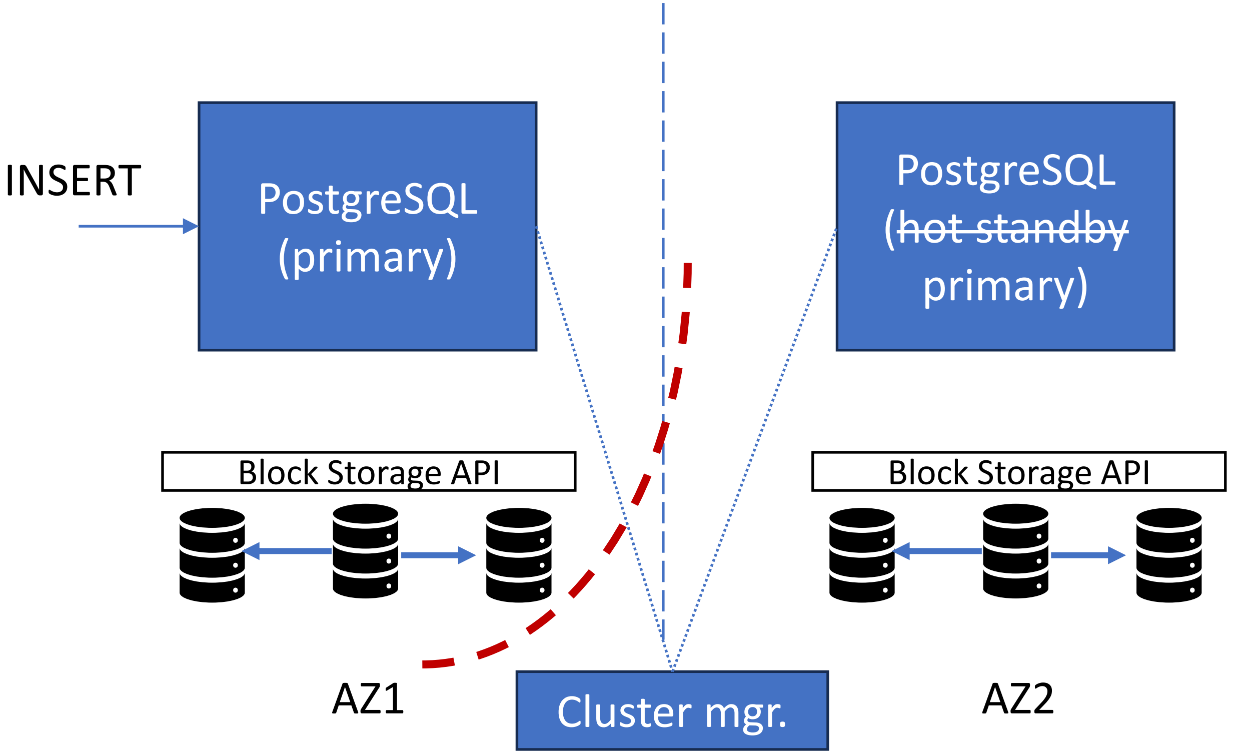
1) **Detect failure**
2) **Demote primary (?)**
3) **Promote standby**
4) **Updating routing mechanism:**
   - **Load balancer / VIP (requires infra to be up)**
   - **DNS (requires infra to be up, wait for TTLS)**
   - **Pooler (can fail, be stale)**
   - **Client-side (reliable, but clumsy)**

In a few minutes:

5) **Replace or rewind old primary**

Primary network partition

Primary might not actually be down…



In a few seconds to minutes:

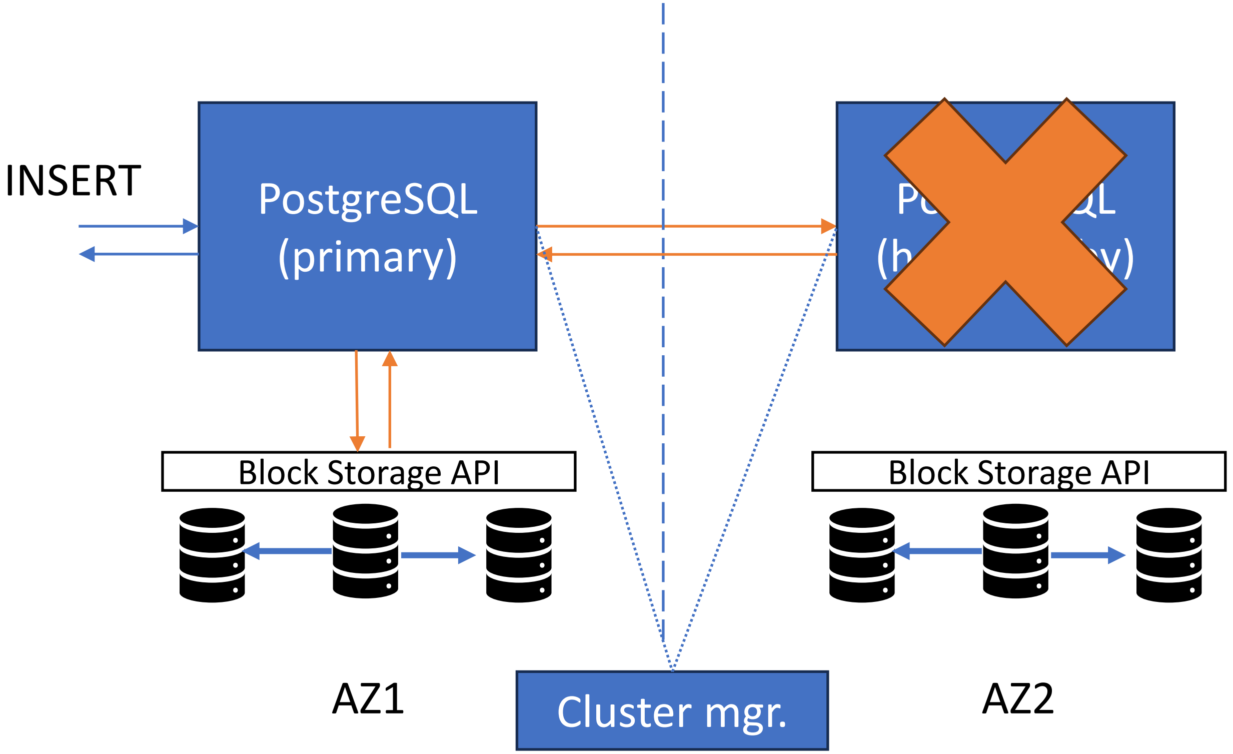
1) **Detect failure**
2) **Demote primary (?)**
3) **Promote standby**
4) **Updating routing mechanism:**
   - **Load balancer / VIP (requires infra to be up)**
   - **DNS (requires infra to be up, wait for TTLS)**
   - **Pooler (can fail, be stale)**
   - **Client-side (reliable, but clumsy)**

In a few minutes:

5) **Replace or rewind old primary**

Hot standby failure

Writes get blocked when hot standby fails, but control plane can quickly react and make the standby ineligible.



In a few seconds:

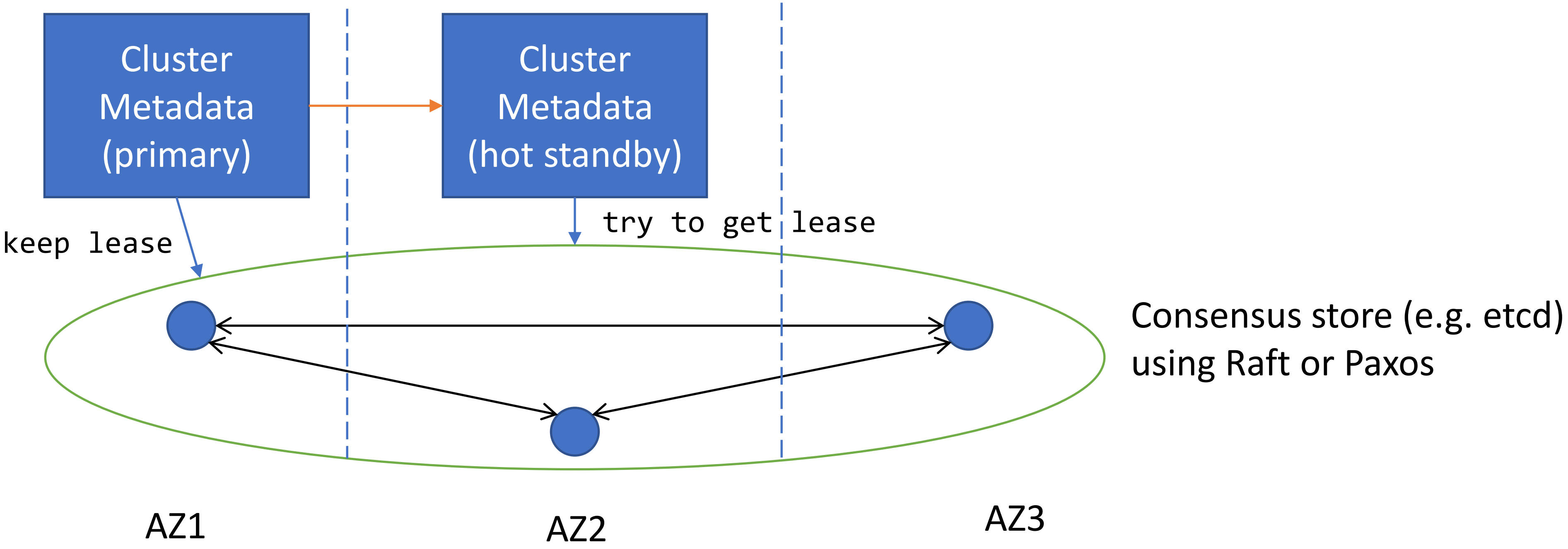
1) **Pause failover (back online)**
2) **Reduce synchronous commit**

In a few minutes:

2) Create new standby or wait for recovery
3) Resume failover

Cluster manager

Cluster managers use consensus to decide a primary metadata store.


Cluster Metadata (primary)
Cluster Metadata (hot standby)
keep lease
try to get lease
Consensus store (e.g. etcd) using Raft or Paxos
AZ1
AZ2
AZ3

Hot standby trade-offs

Pros:

Can survive node and availability zone failure

No crash recovery when resuming from failure

Cons:

Write latency can be very high

Cost is twice as high

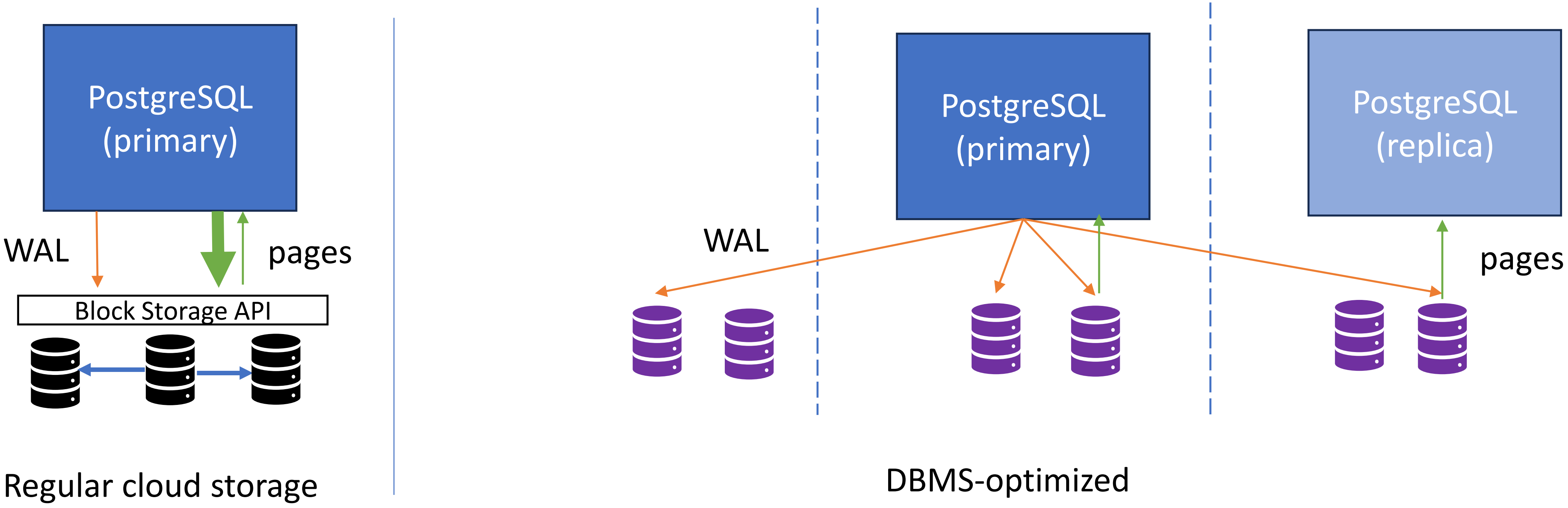
Rerouting on failover can have issues / take a while

DBMS-optimized storage

Like Aurora, Neon, AlloyDB

DBMS-optimized storage

Cloud storage that can perform background page writes autonomously, which saves on write I/O from primary. Also optimized for other DBMS needs (e.g. read replicas).



DBMS-optimized storage trade-offs

Pros:

Potential performance benefits by avoiding page writes from primary

No crash recovery

Replicas can reuse storage, incl. hot standby

Less rigid than network-attached storage implementations (faster reattach, branching, ...)

Cons:

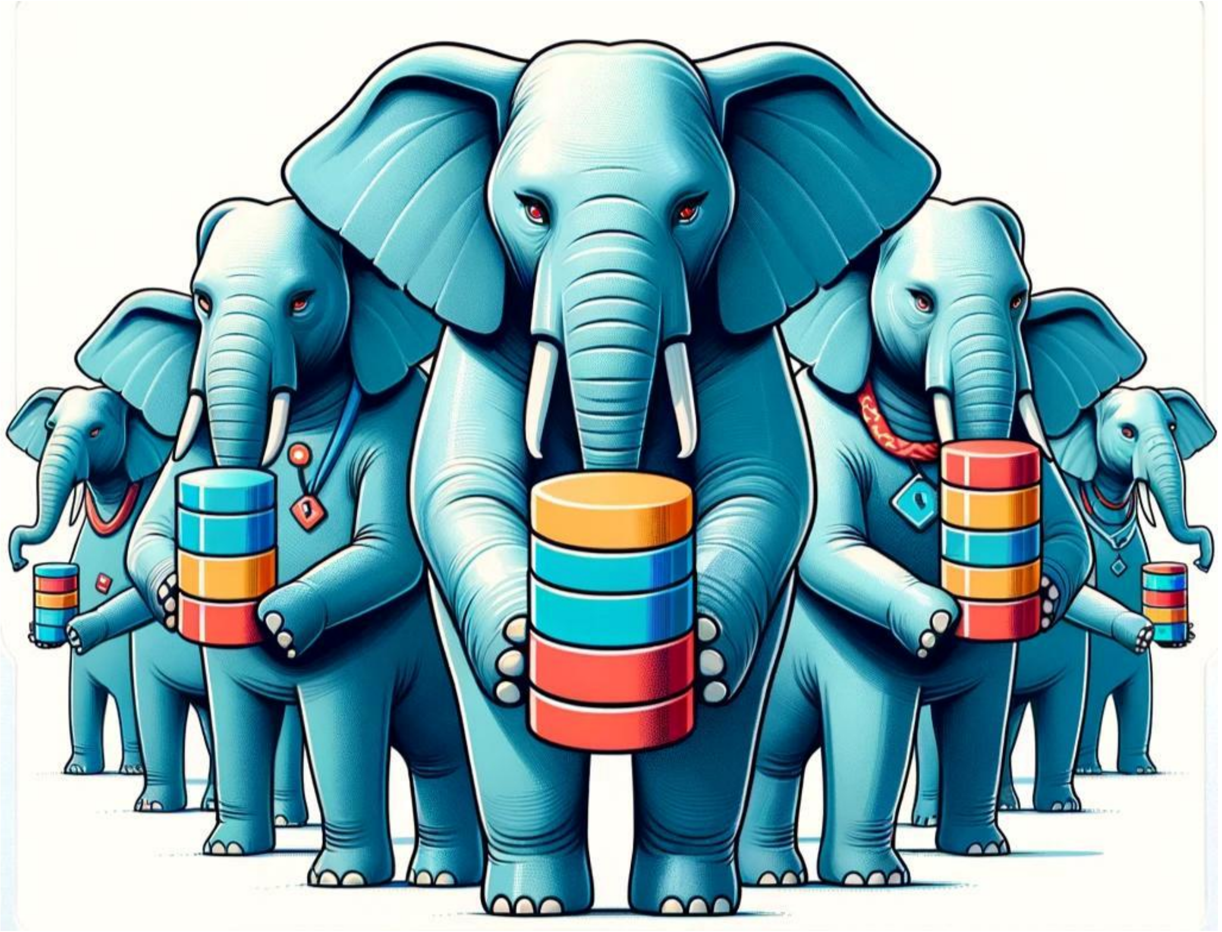
Write latency is high by default

High cost / pricing

PostgreSQL is not designed for it, can be slower than regular storage

Transparent sharding

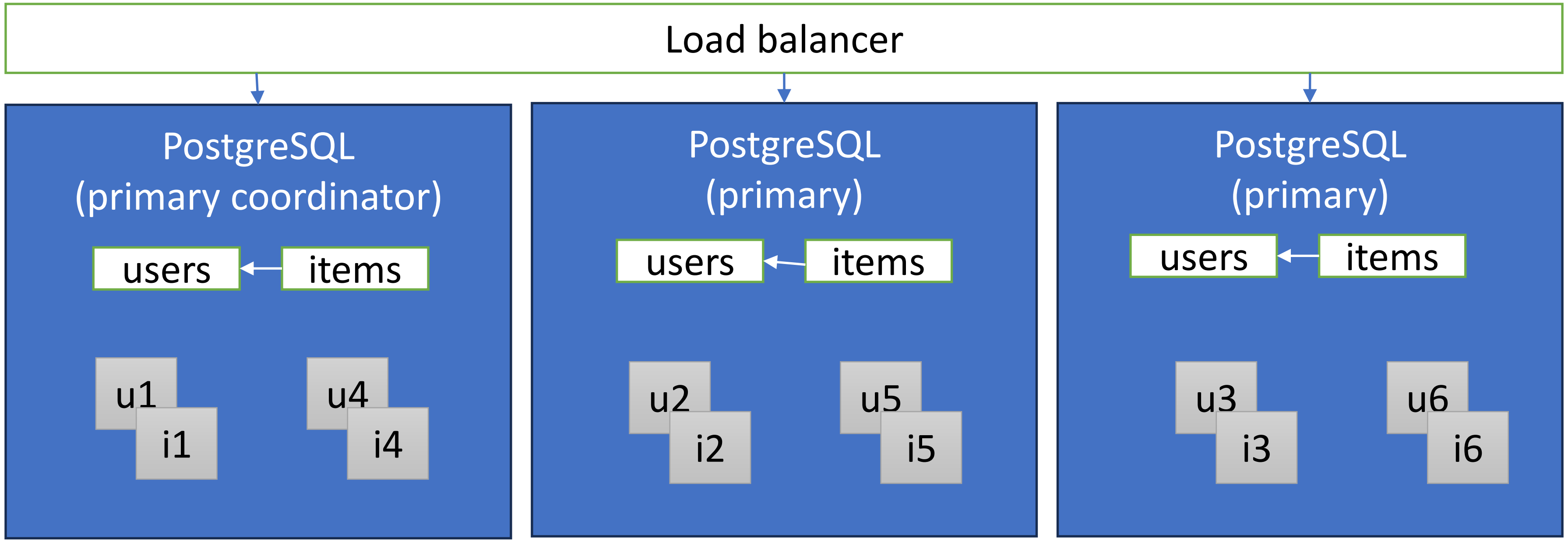
Like Citus



Transparent sharding

Distribute tables by a shard key and/or replicate tables across multiple (primary) nodes.

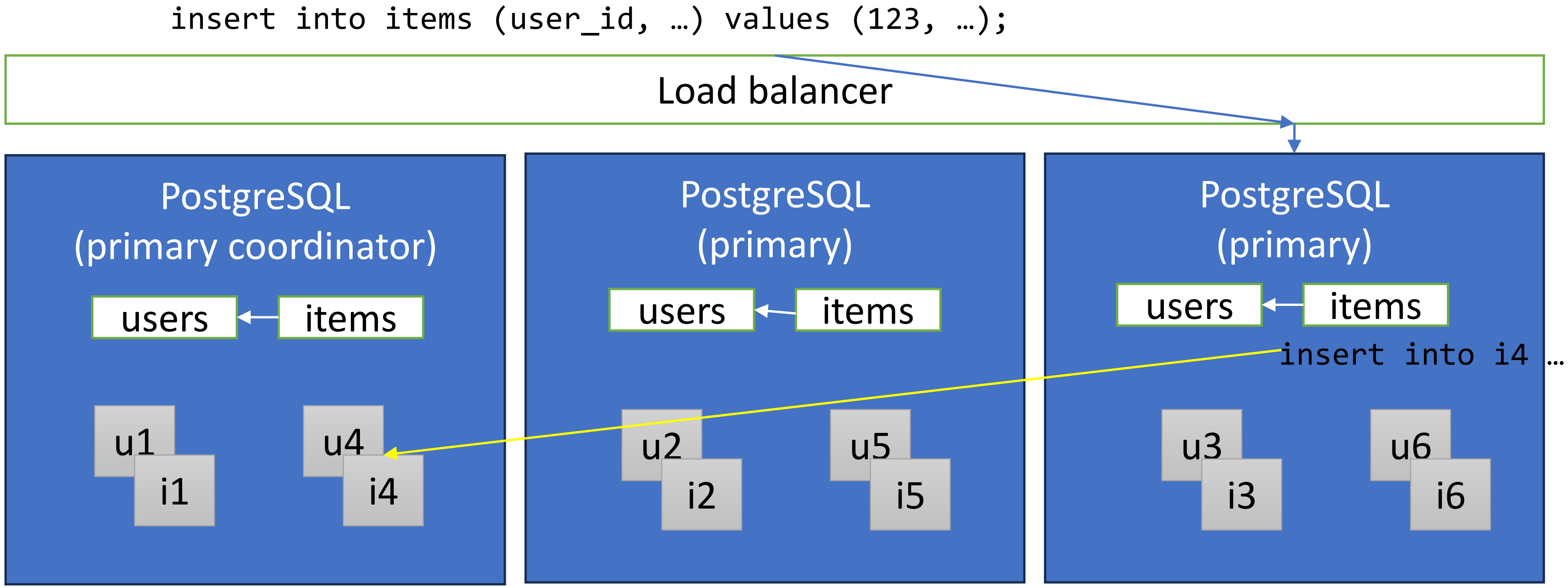
Queries & transactions are transparently routed / parallelized.



Tables can be co-located to enable local joins, foreign keys, etc. by the shard key.

Single shard queries for operational workloads

Scale capacity for handling a high rate of single shard key queries:



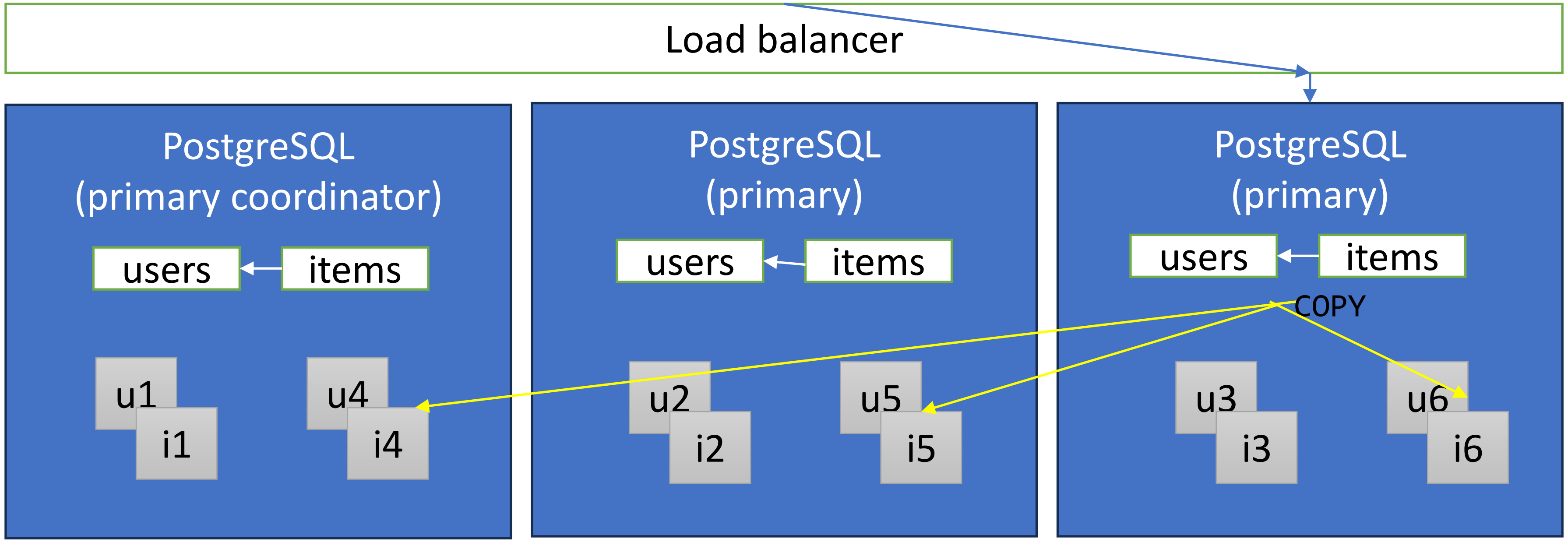
```
insert into items (user_id, …) values (123, …);
```

Per-statement latency can be a bottleneck!

Data loading in sharded system

Pipelining through COPY can make data loading a lot more efficient and scalable

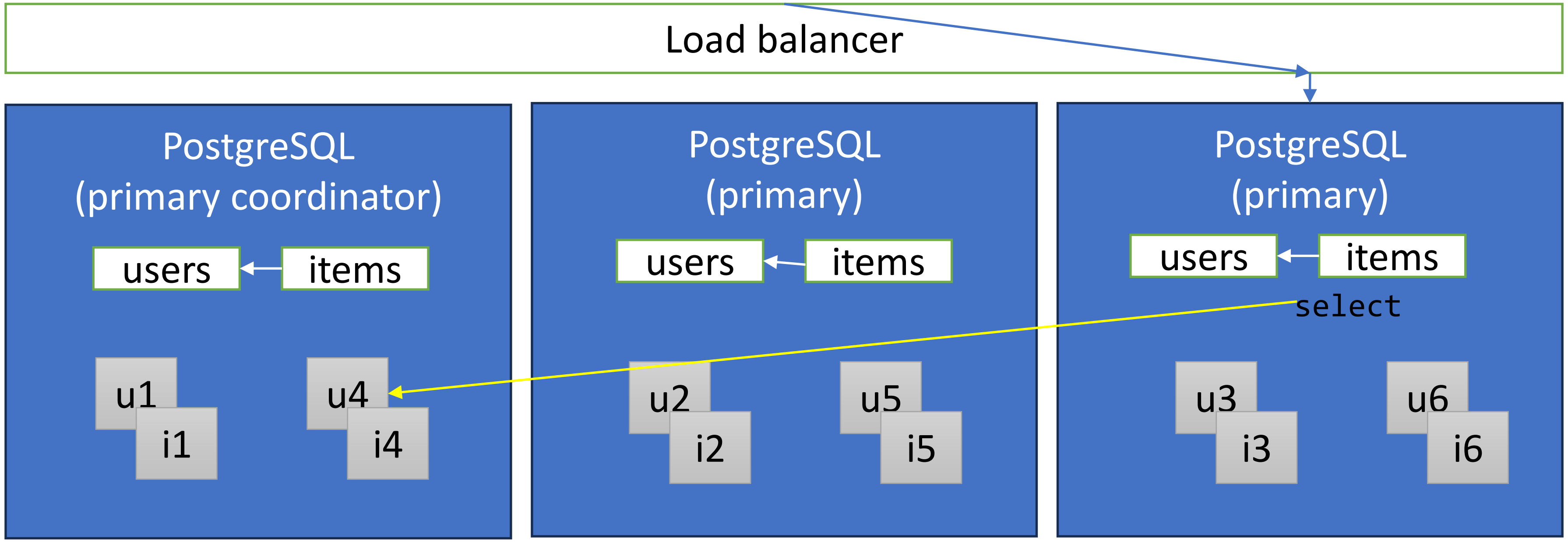
```
COPY items FROM STDIN WITH (format 'csv')
```



Compute-heavy queries

Compute-heavy queries (shard key joins, json, vector, …) get the most relative benefit

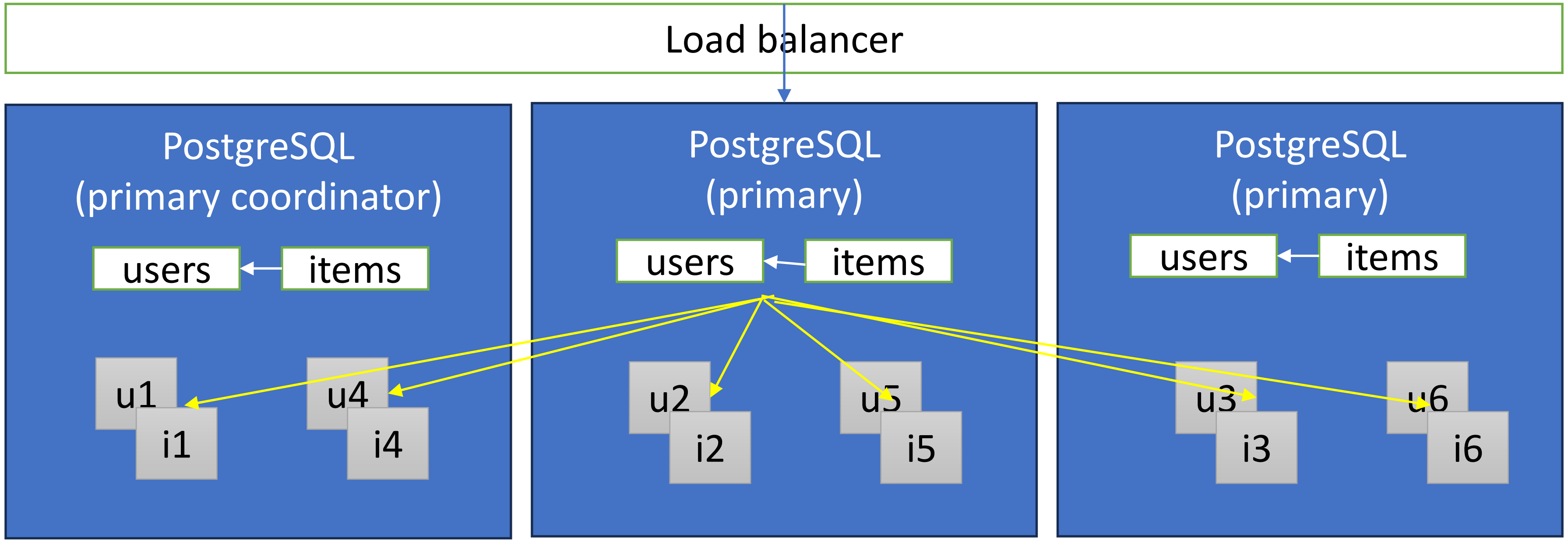
```
select compute_stuff(…) from users join items using (user_id) where user_id = 123 …
```



Multi-shard queries for analytical workloads

Parallel multi-shard queries can quickly answer analytical queries across shard keys:

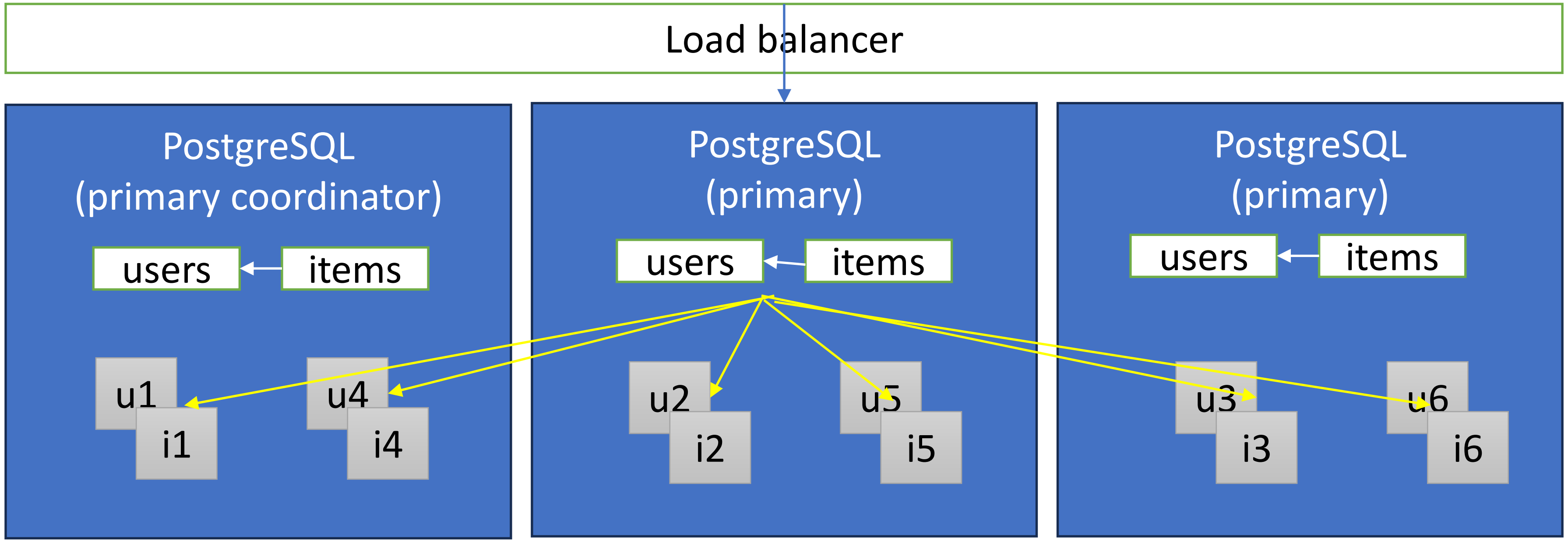
```
select country, count(*) from items, users where … group by 1 order by 2 desc limit 10;
```



Multi-shard queries for operational workloads

Multi-shard queries add significant overhead for simple non-shard-key queries

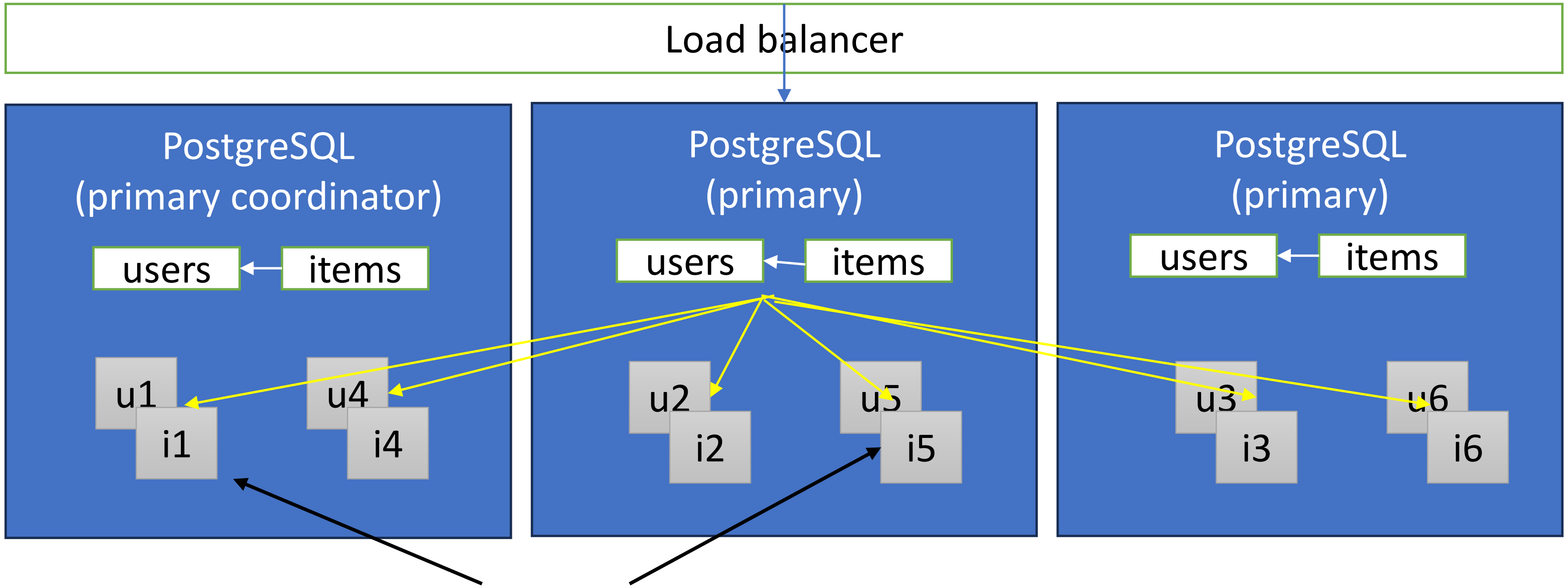
```
select * from items where item_id = 87;
```



Multi-shard queries for analytical workloads

Snapshot isolation is a challenge (involves trade-offs):

```
select country, count(*) from items, users where … group by 1 order by 2 desc limit 10;
```



```
↔ BEGIN;
← INSERT INTO items VALUES (123, …);
→ INSERT INTO items VALUES (456, …);
↔ COMMIT;
```

Sharding trade-offs

Pros:

Scale throughput for reads & writes (CPU & IOPS)

Scale memory for large working sets

Parallelize analytical queries, batch operations

Cons:

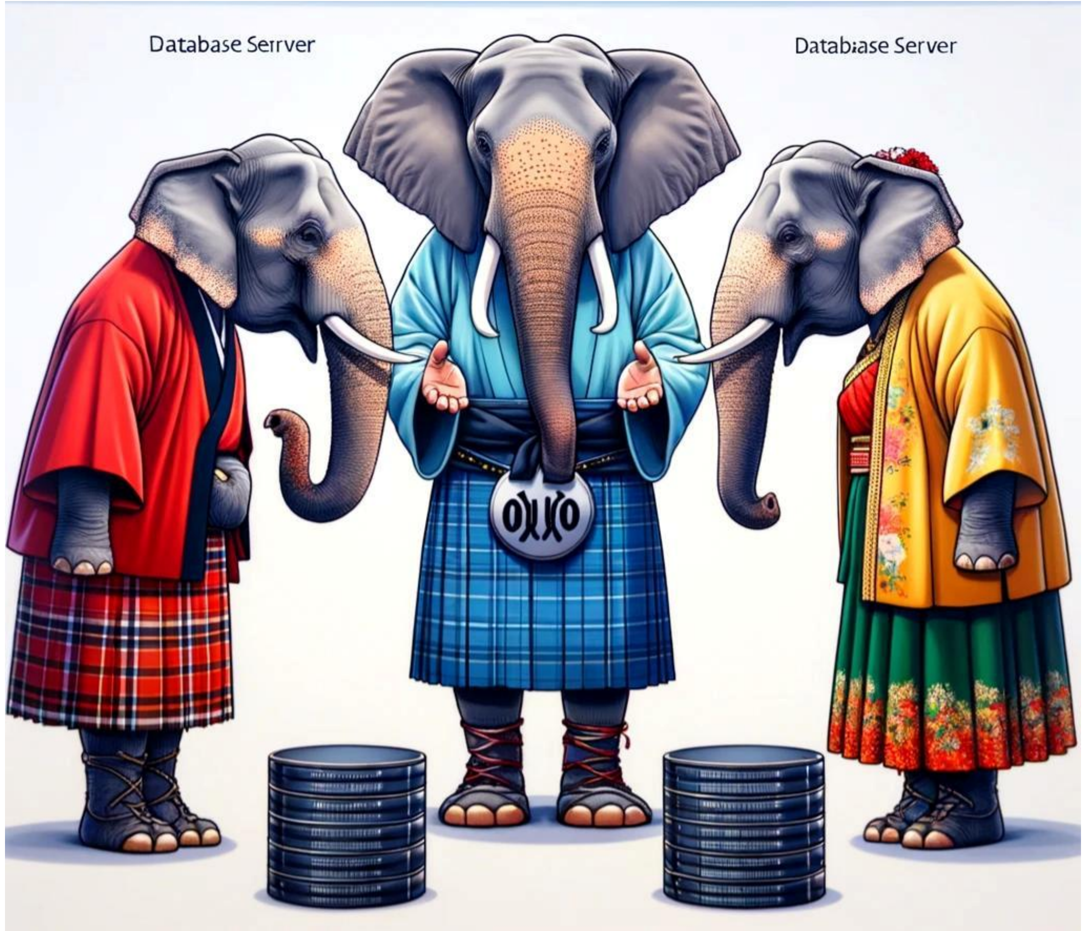
High read and write latency

Data model decisions have high impact on performance

Snapshot isolation concessions

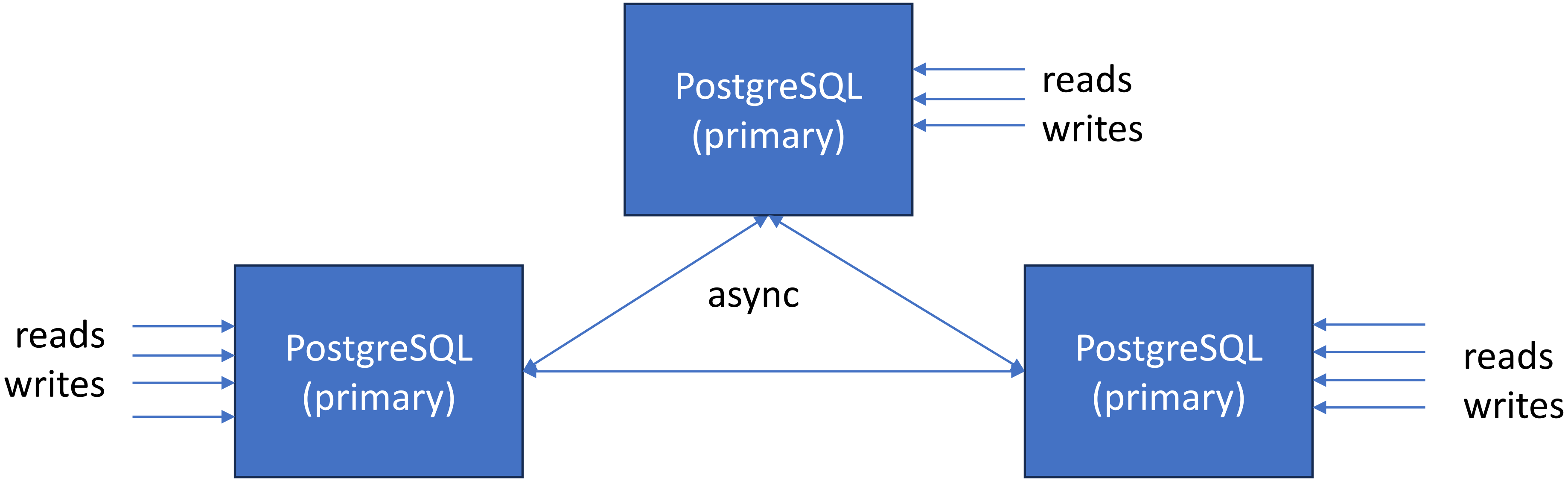
Active-active

Like BDR, pgactive, pgEdge, …



Active-active / n-way replication

Accept writes from any node, use logical replication to asynchronously exchange and consolidate writes.

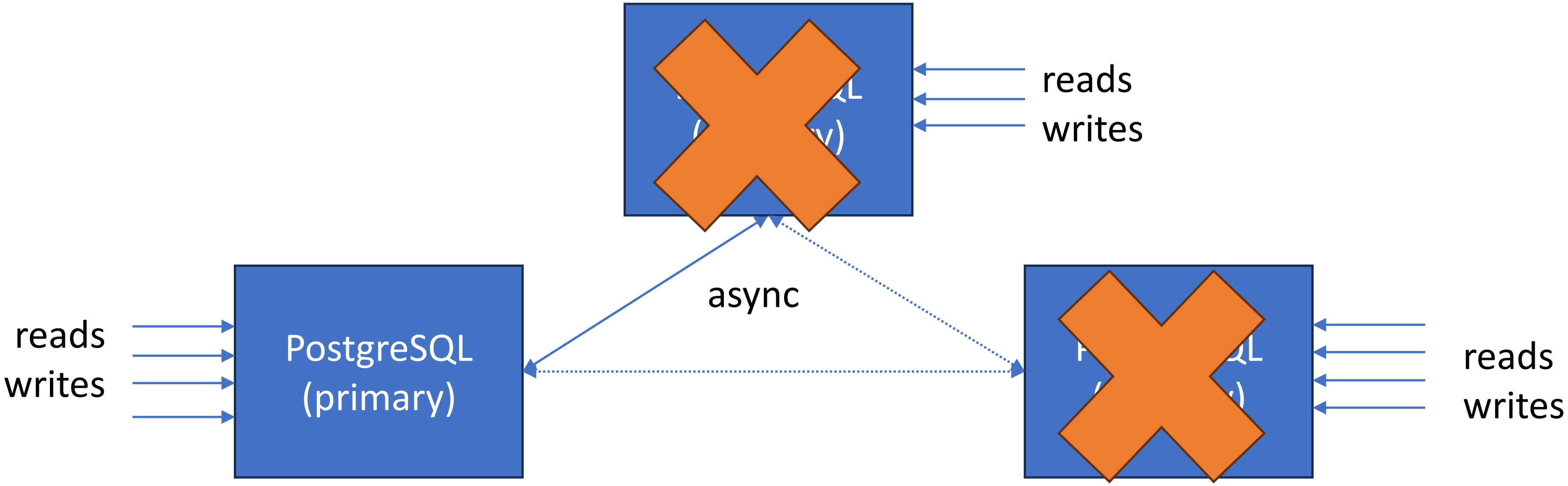


UPDATE counters SET val = val + 1

UPDATE counters SET val = val + 1

Active-active / n-way replication

All nodes can survive network partitions by accepting writes locally, but no linear history (CAP).



Active-active trade-offs

Pros:

Very high read and write availability

Low read and write latency

Read throughput scales linearly

Cons:

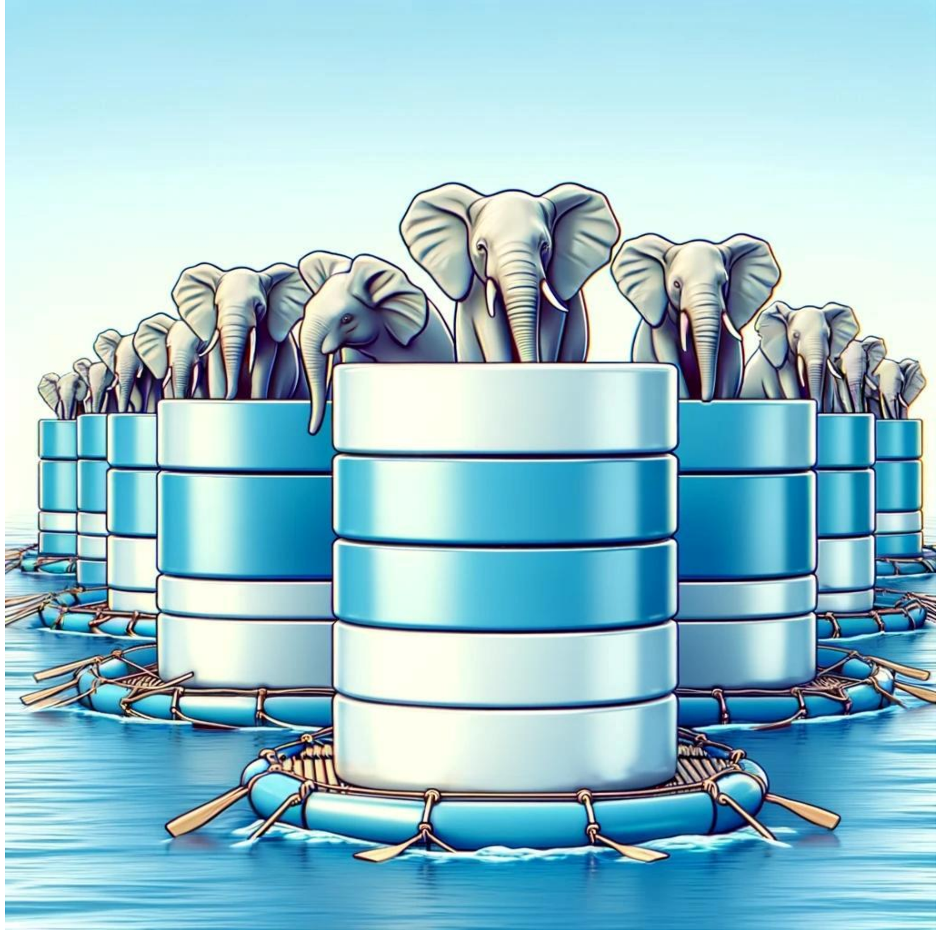
Eventual read-your-writes consistency

No monotonic read consistency

No linear history (updates might conflict after commit)

Distributed SQL

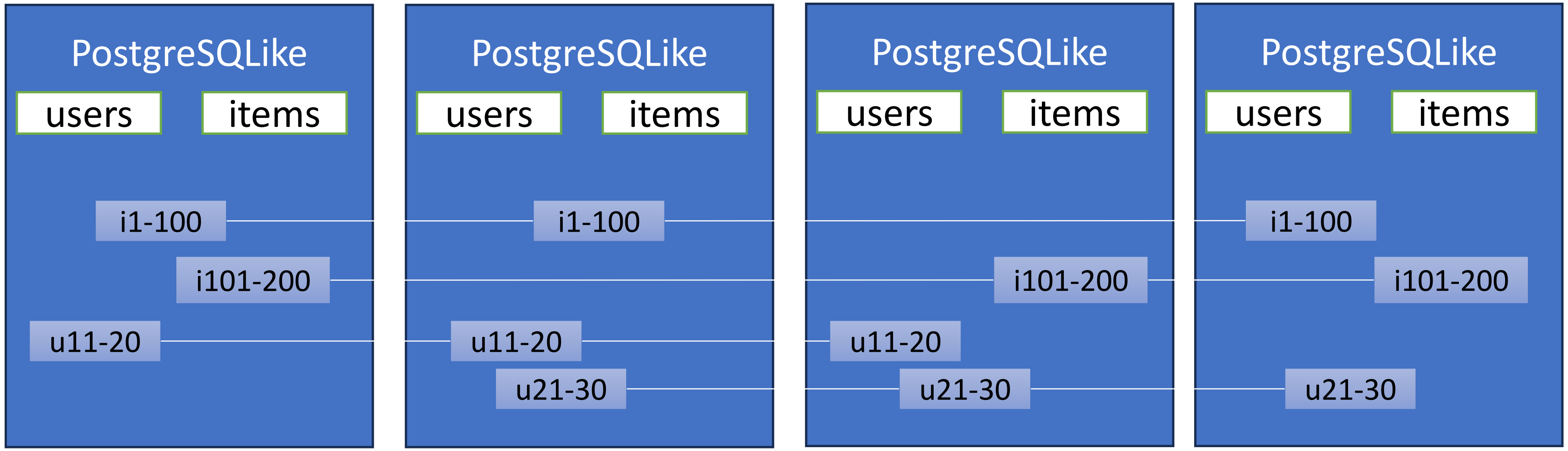
Like Yugabyte, CockroachDB, Spanner



Distributed key-value storage with SQL (DSQL)

Tables are stored on distributed key-value stores, shards replicated using Paxos/Raft.

Distributed transactions with snapshot isolation via global timestamps (HLC or TrueTime).



Distributed key-value storage trade-offs

Pros:

Good read and write availability (shard-level failover)

Single table, single key operations scale well

No additional data modelling steps or snapshot isolation concessions

Cons:

Many operations (index lookups, evaluating joins and foreign keys) incur high latency

No local joins (co-location, reference tables) in current implementations

Less mature and optimized than PostgreSQL

Conclusion

PostgreSQL can be distributed at different layers.

Each architecture can introduce severe trade-offs.

Almost nothing comes for free..

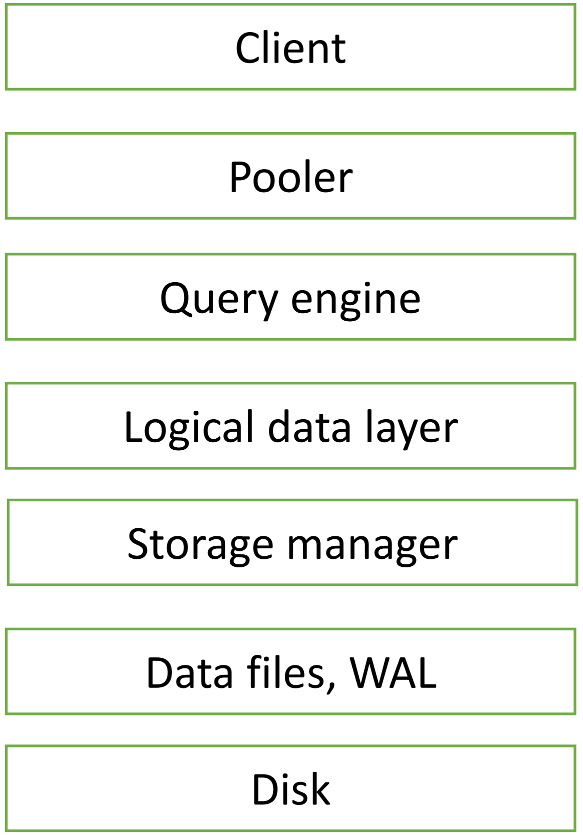
Keep asking:

What do I really want?

Which architecture achieves that?

What are the trade-offs?

What can my application tolerate? (can I change it?)



Client

Pooler

Query engine

Logical data layer

Storage manager

Data files, WAL

Disk

General guidelines

- **Network-attached storage** operational benefits usually outweigh the perf. downsides
- **Read replicas** scale reads linearly, but anomalies can be cumbersome in practice
- **Hot standby** is good for availability, but costly and need to batch your writes
- **DBMS-optimized storage** makes subtle trade-offs, mileage may vary
- **Sharding** makes crude trade-offs to achieve scalability, requires data model shift
- **Active-active** anomalies usually outweigh its benefits
- **DSQL** works well as a versatile key-value store, but just use PostgreSQL

Questions?

Marco.slot@gmail.com