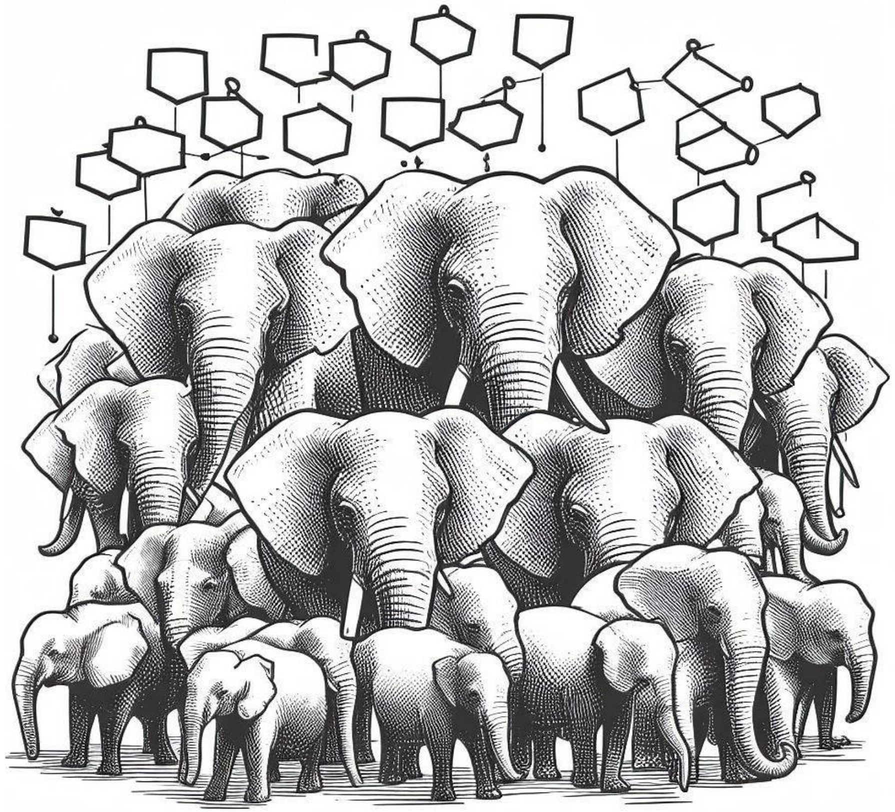
PostgreSQL Distributed

Architectures & Best Practices



Marco Slot - marco.slot@gmail.com

Formerly: founding engineer at Citus Data, architect at Microsoft

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.

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

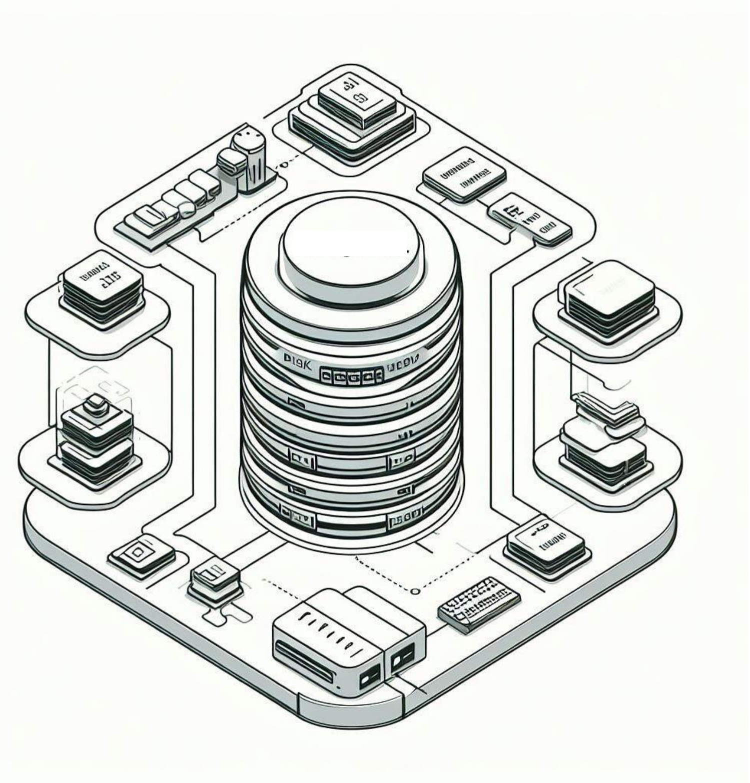
No network latency

Millions of IOPS

Microsecond disk latency

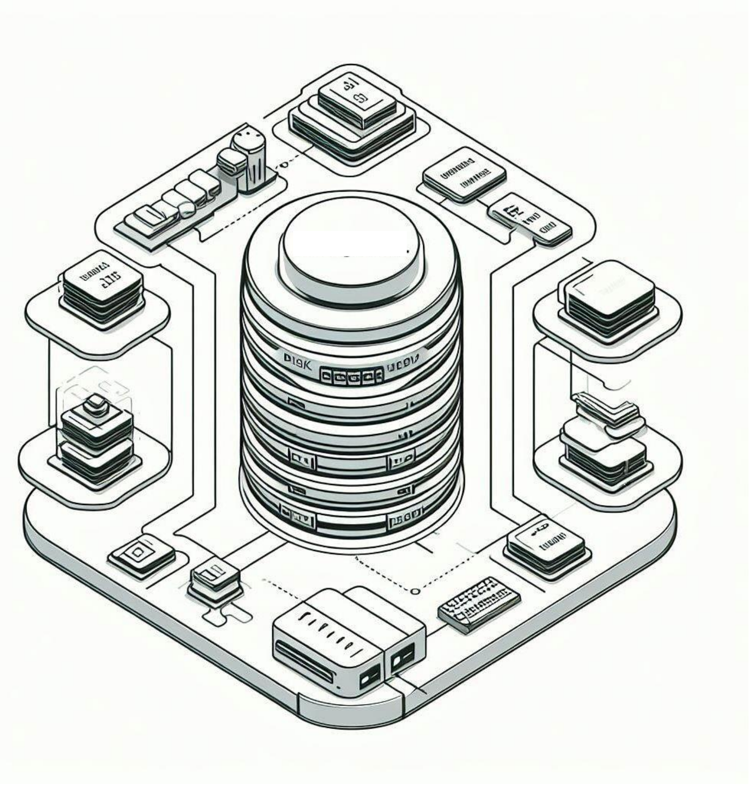
Low cost / fast hardware

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)

Disk full (downtime)

PostgreSQL Distributed (in the cloud)

Fixing the operational hazards of single machine 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 <u>performance</u>, 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), DSQL
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 block storage
- Read replicas
- DBMS-optimized cloud storage
- Transparent Sharding
- Active-active deployments
- Distributed key-value stores with SQL

Two questions:

1) What are the trade-offs?

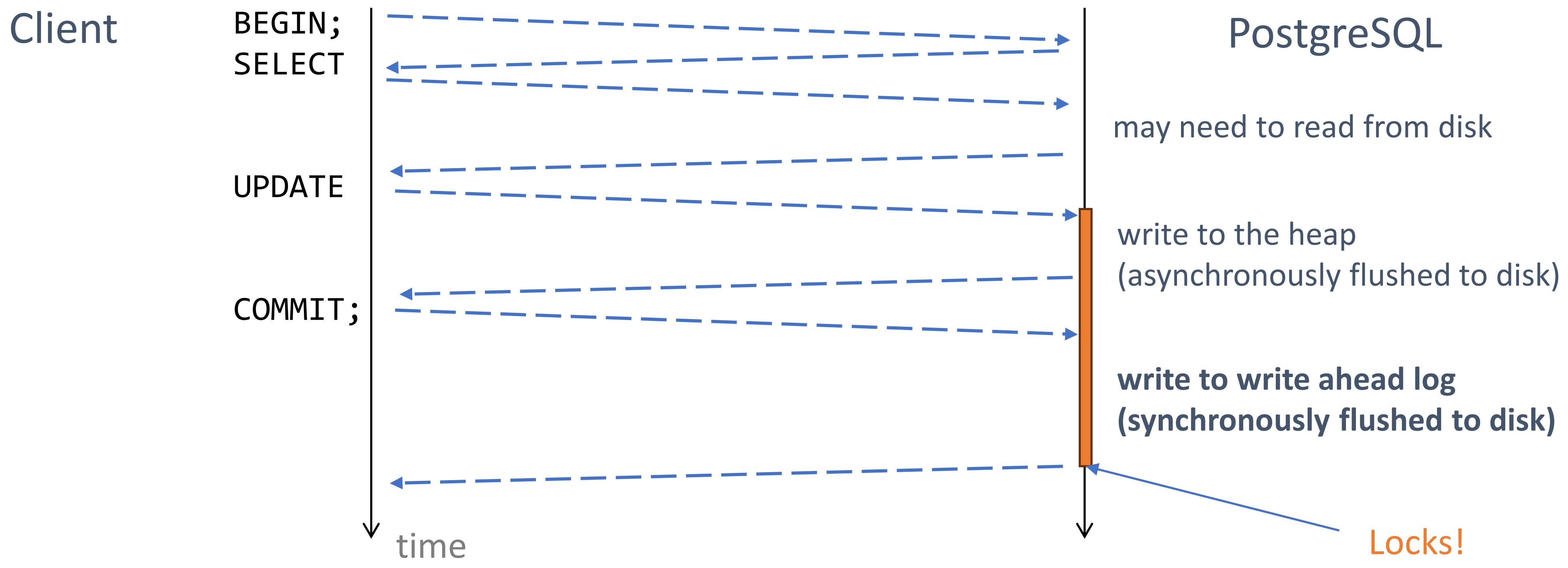
Latency, Efficiency, Cost, Scalability, Availability, Consistency, Complexity, …

2) For which workloads?

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

The perils of latency: Synchronous protocol

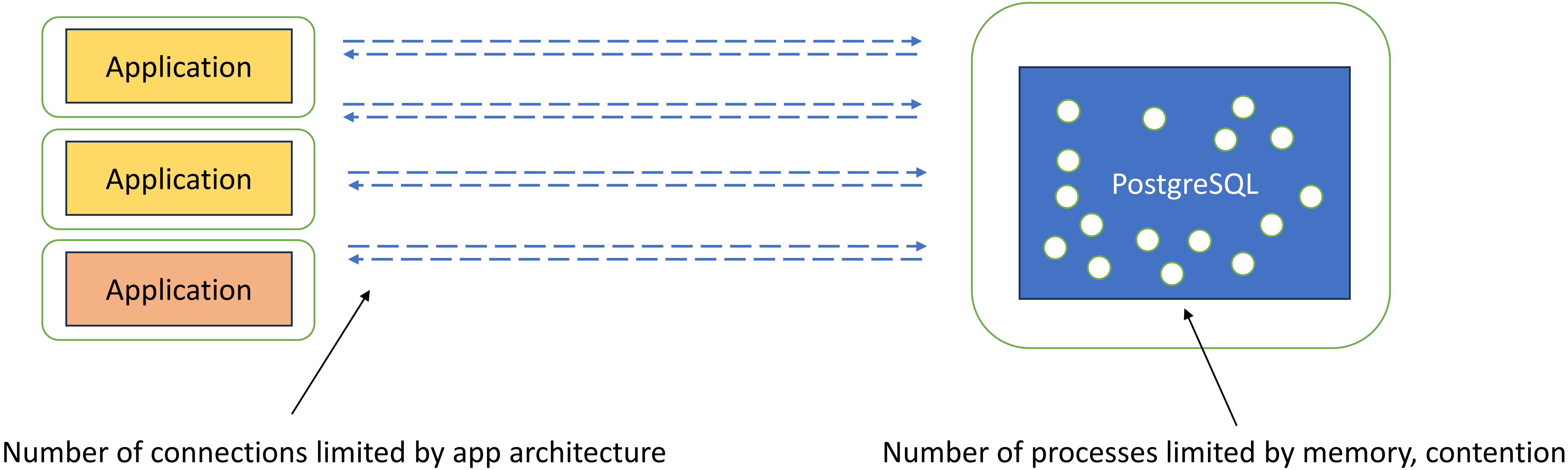
Transactions are performed step-by-step on each session.



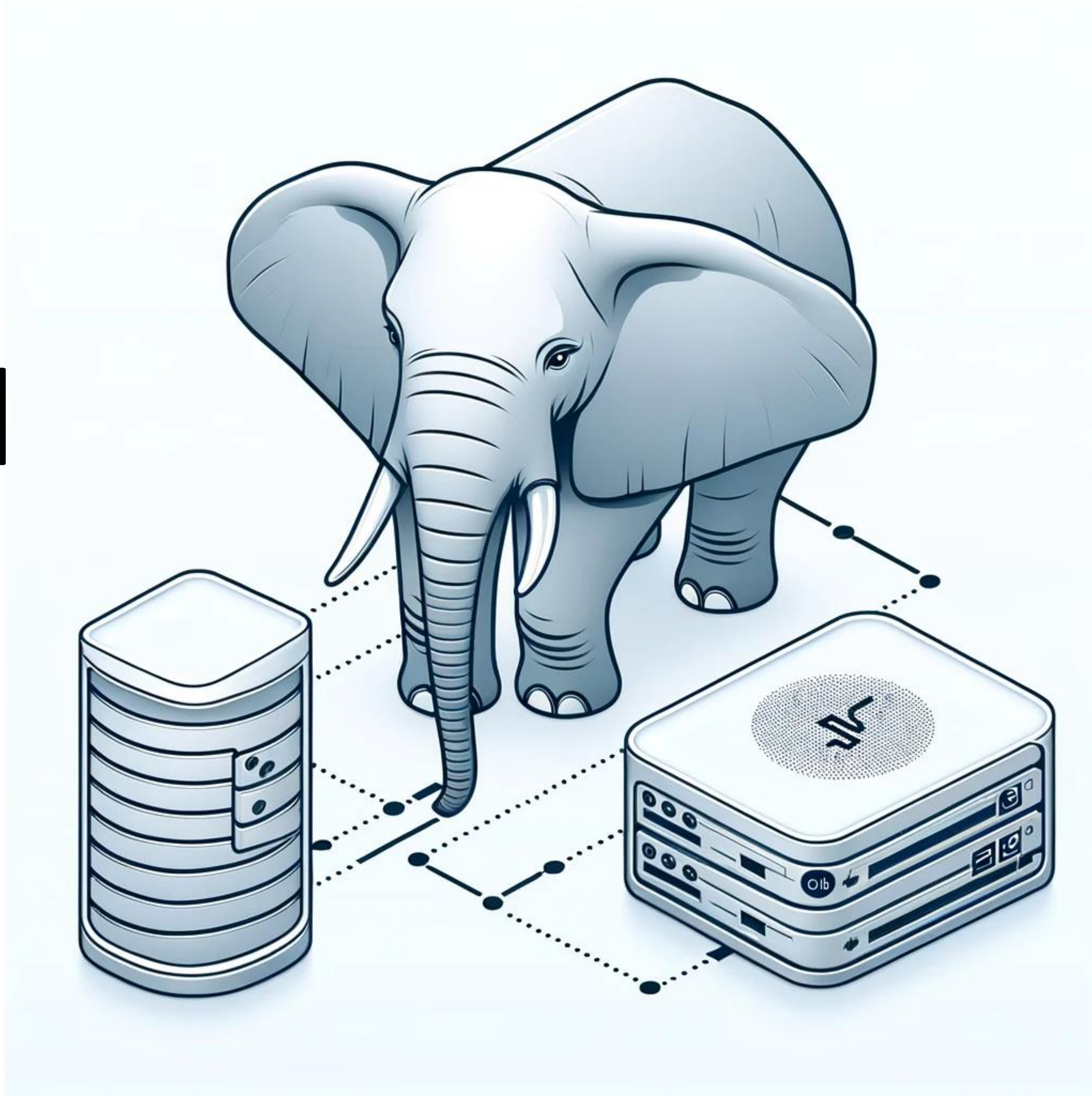
Max throughput per session = 1 / avg. response time

The perils of latency: Connection limits

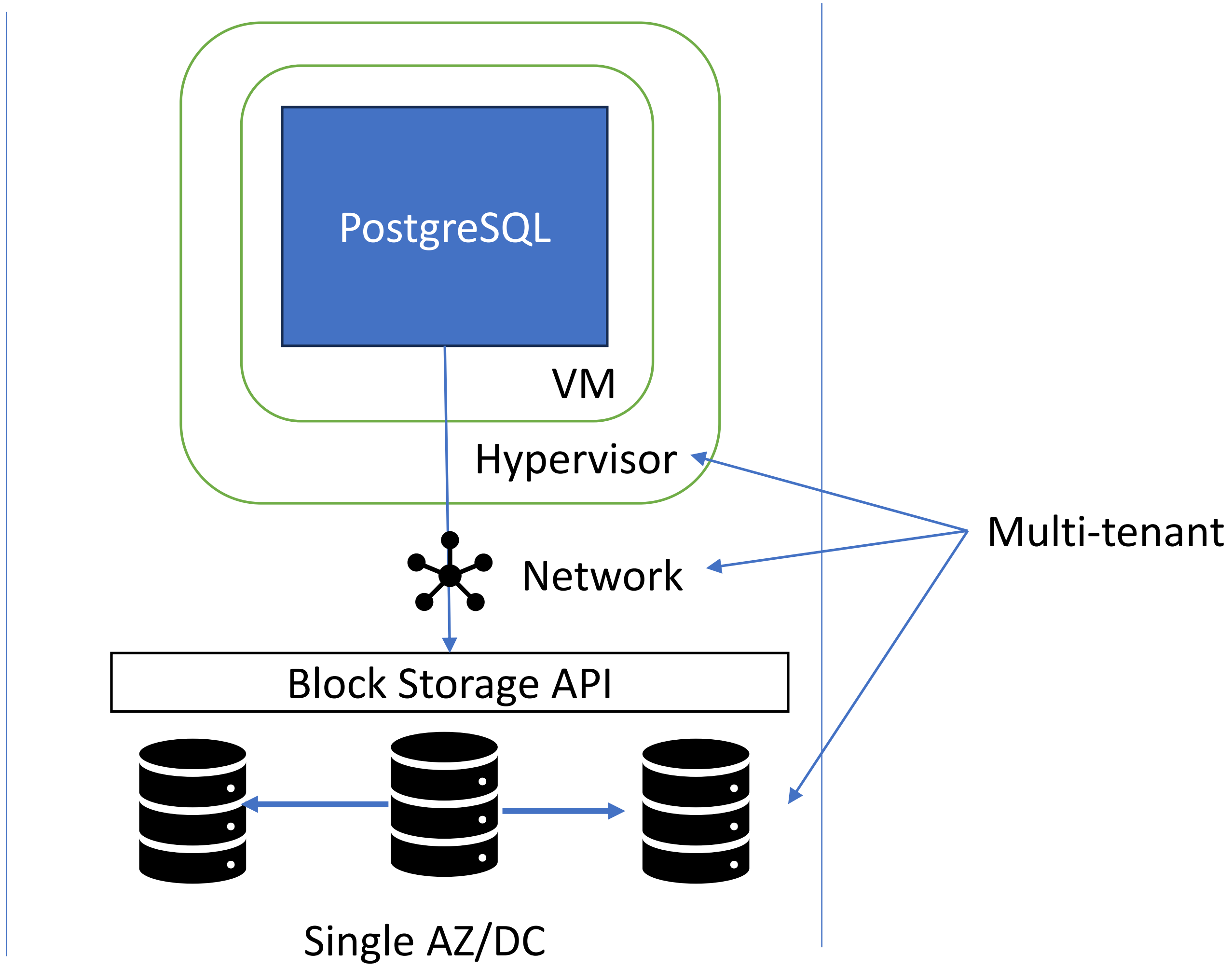
Max overall throughput: #sessions / avg.response time



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 (~20µs -> ~1000µs)

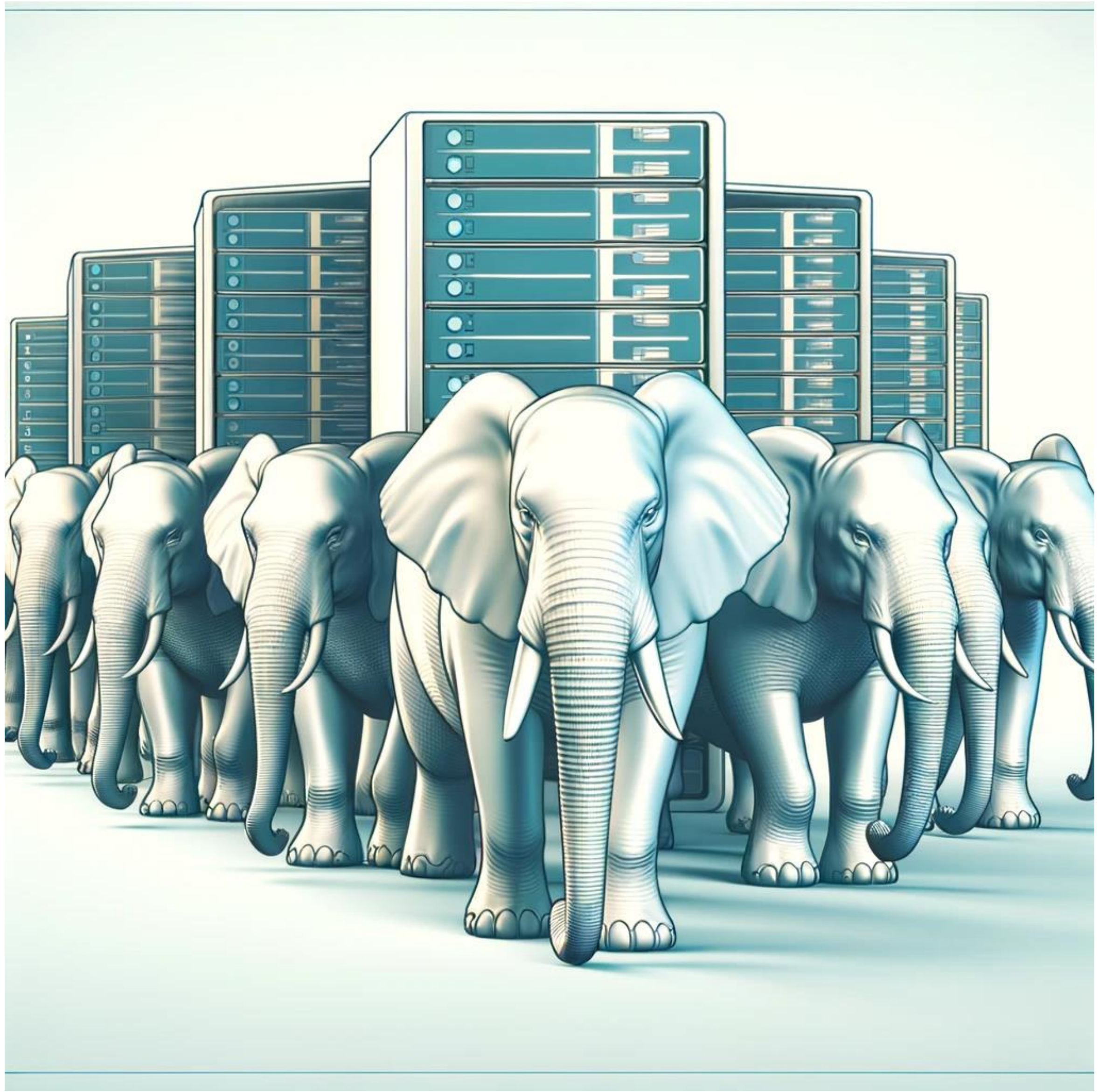
Lower IOPS (~1M -> ~10k IOPS)

Crash recovery on restart takes time

Cost can be high

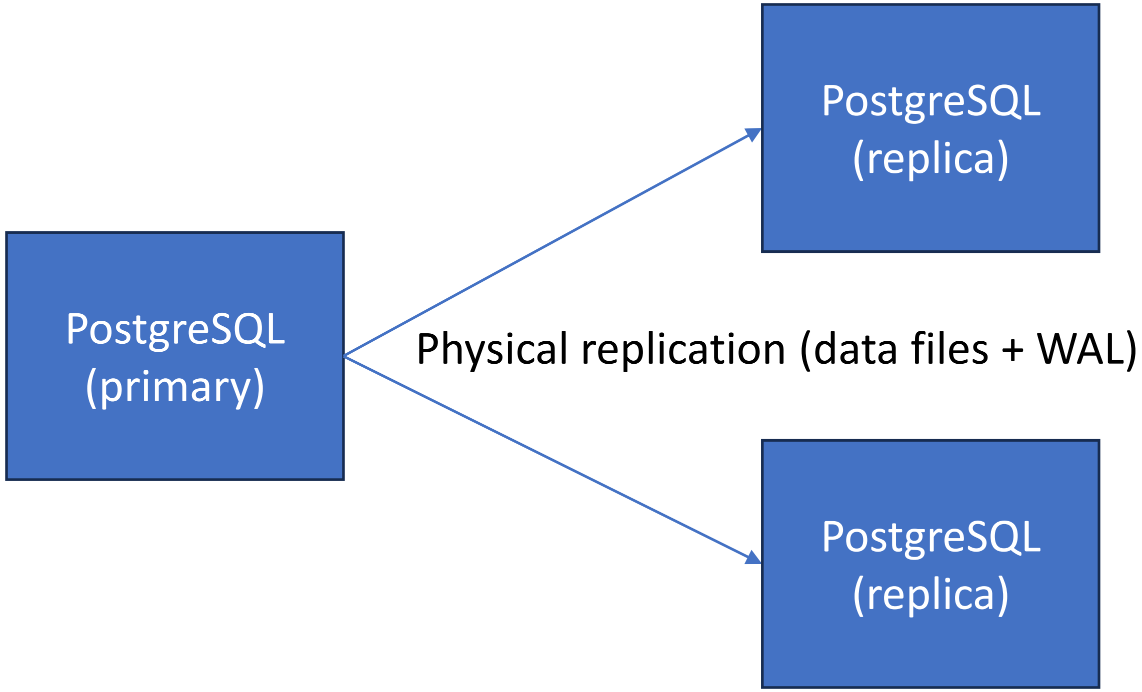
General guideline:
Always use, durability & availability are more important than performance.

Read replicas



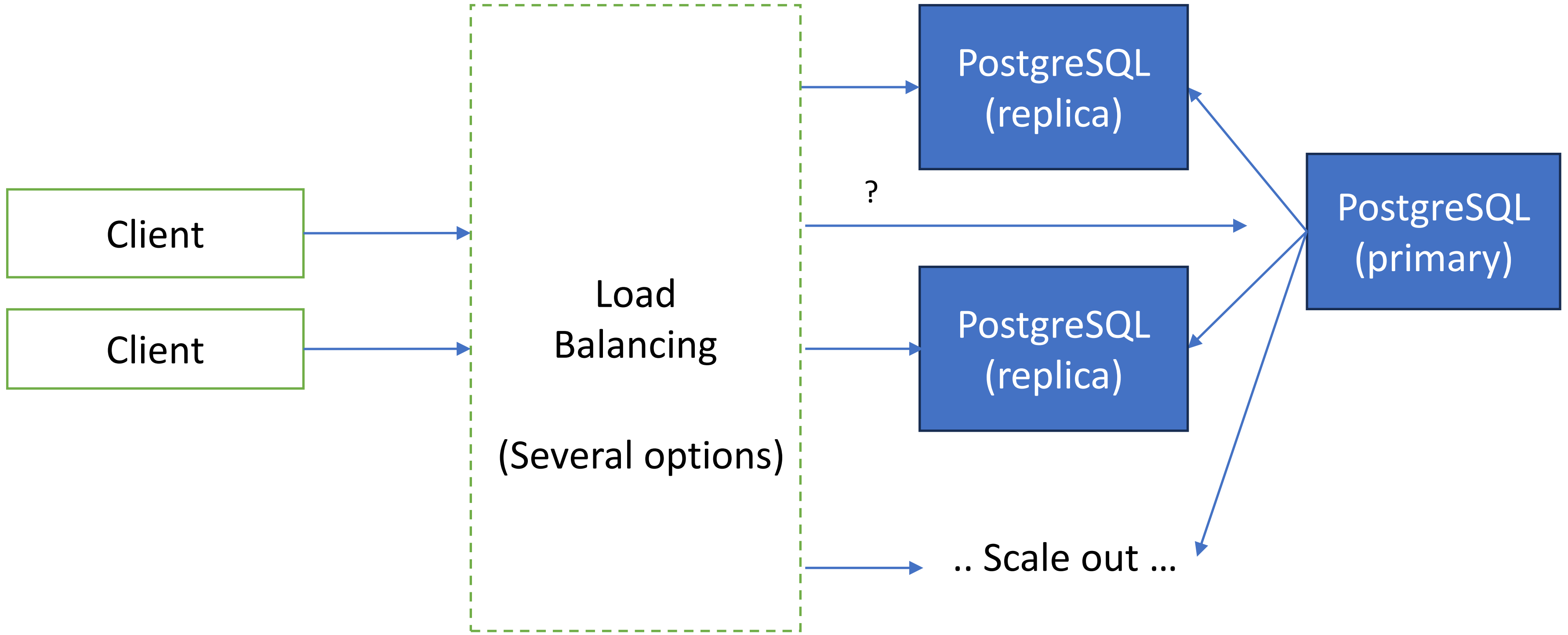
Read replicas

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



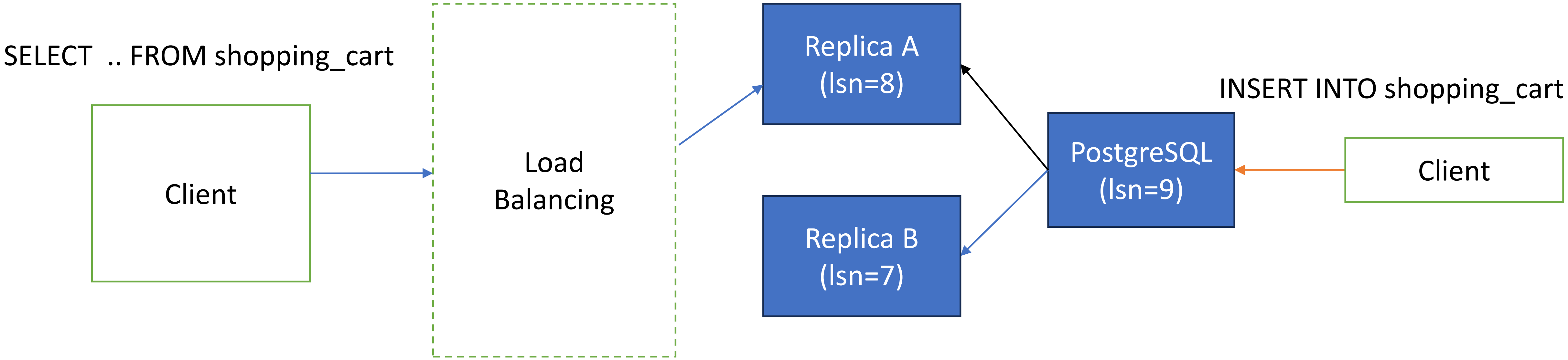
Scaling read throughput

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



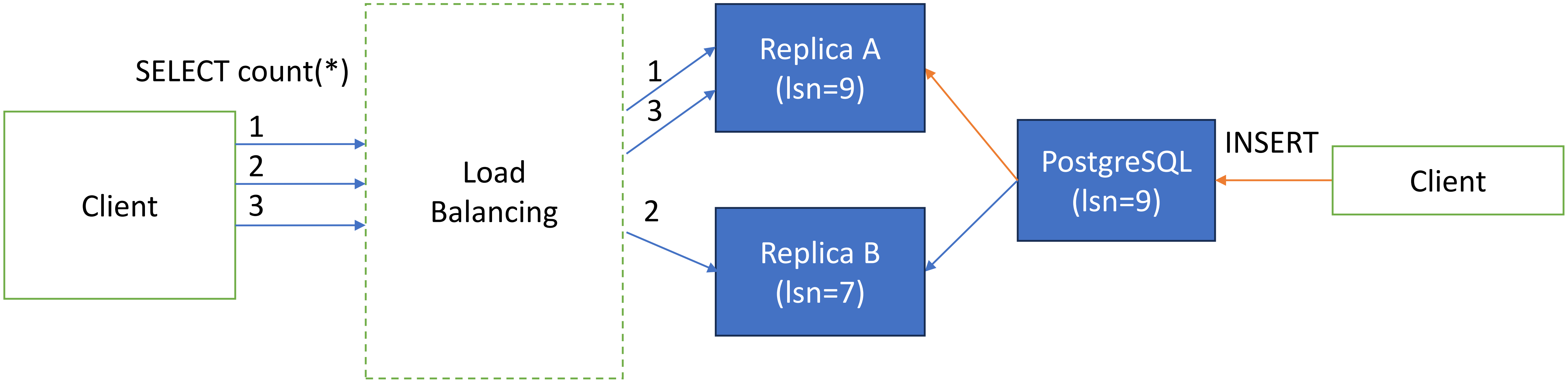
Eventual read-your-writes consistency

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



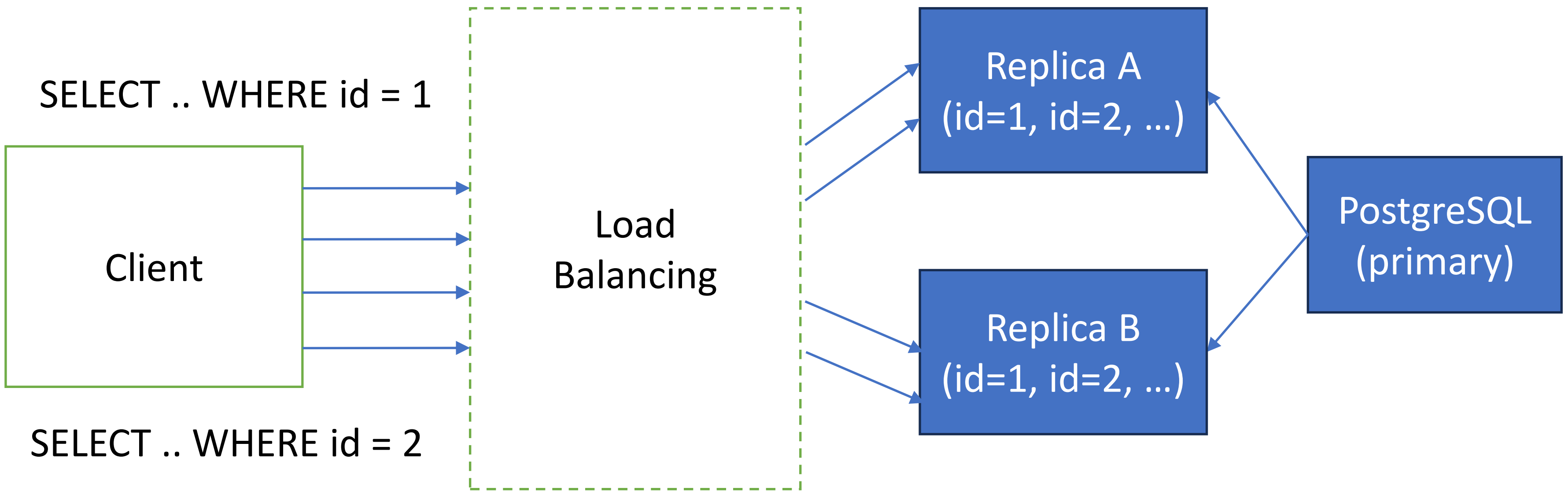
No monotonic read consistency

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

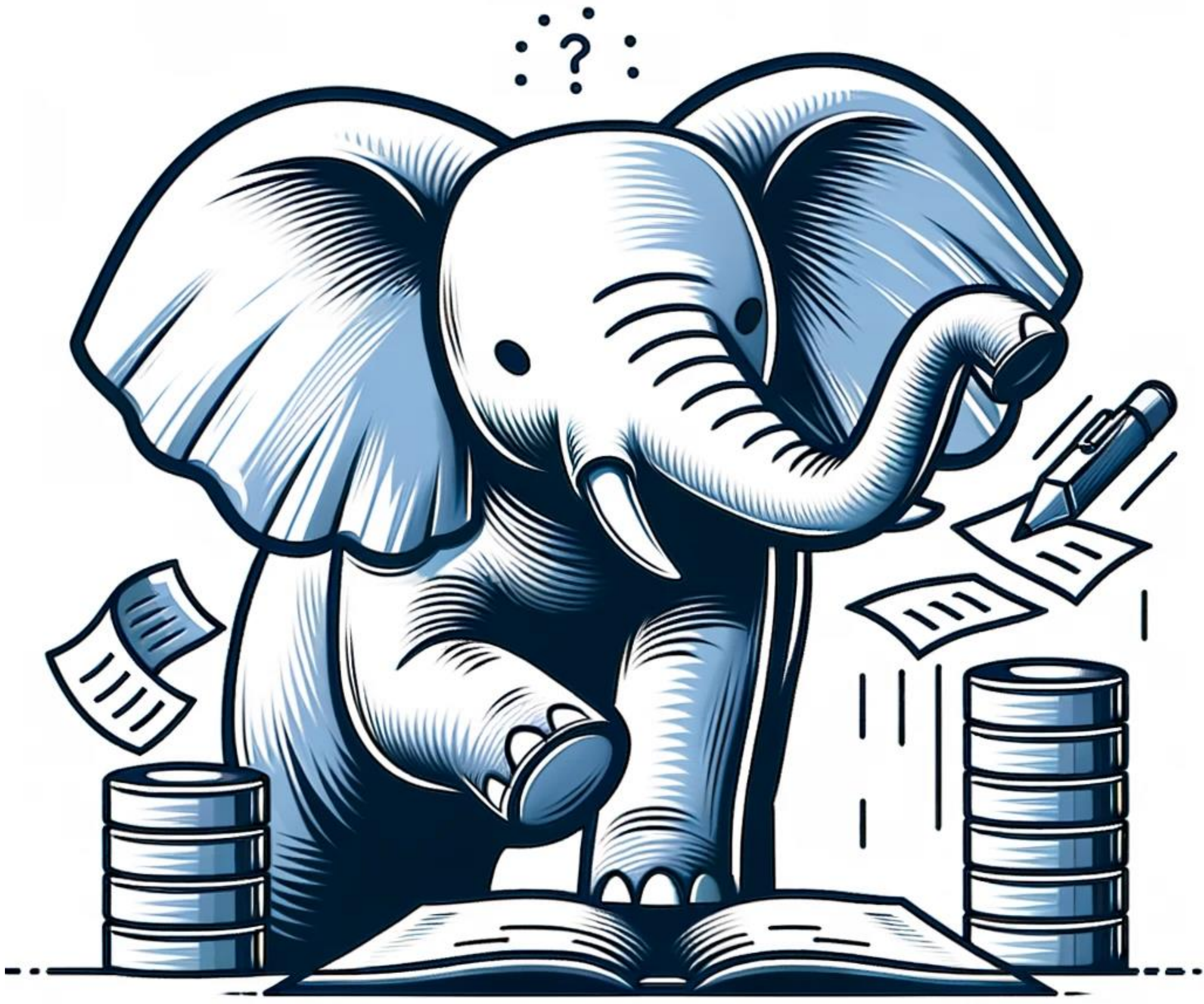
No monotonic read consistency

Poor cache usage

General guideline:
Consider at >100k reads/sec or heavy CPU bottleneck, but avoid for dependent transactions and large working sets.

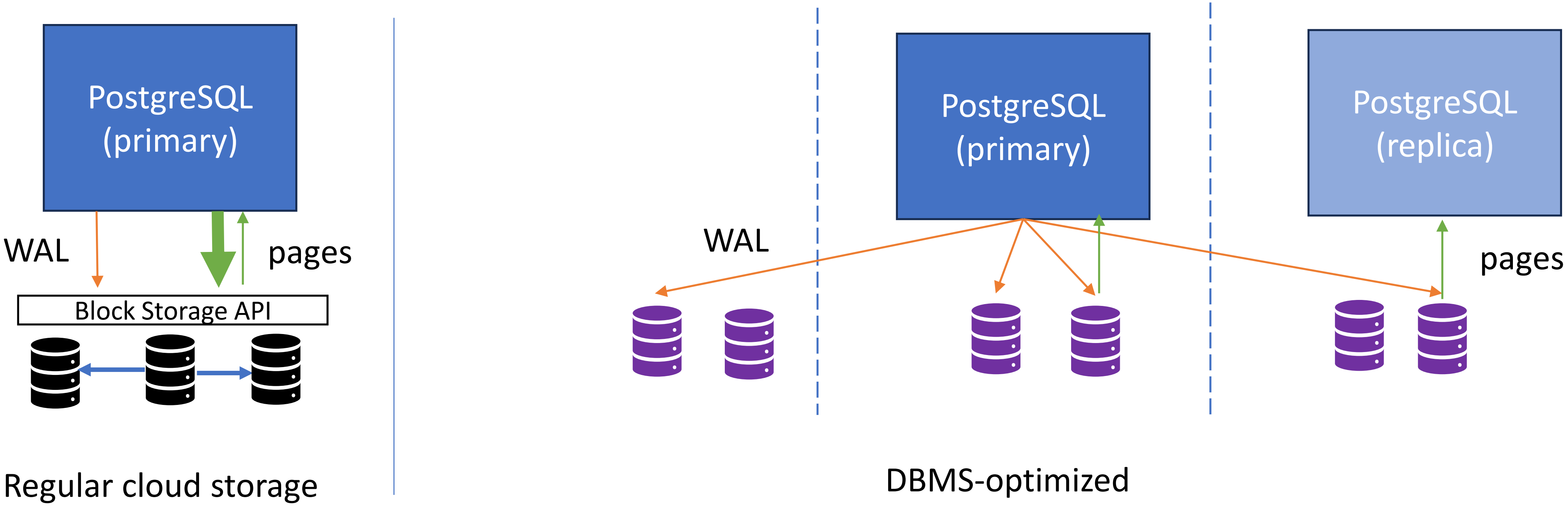
DBMS-optimized storage

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 long 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

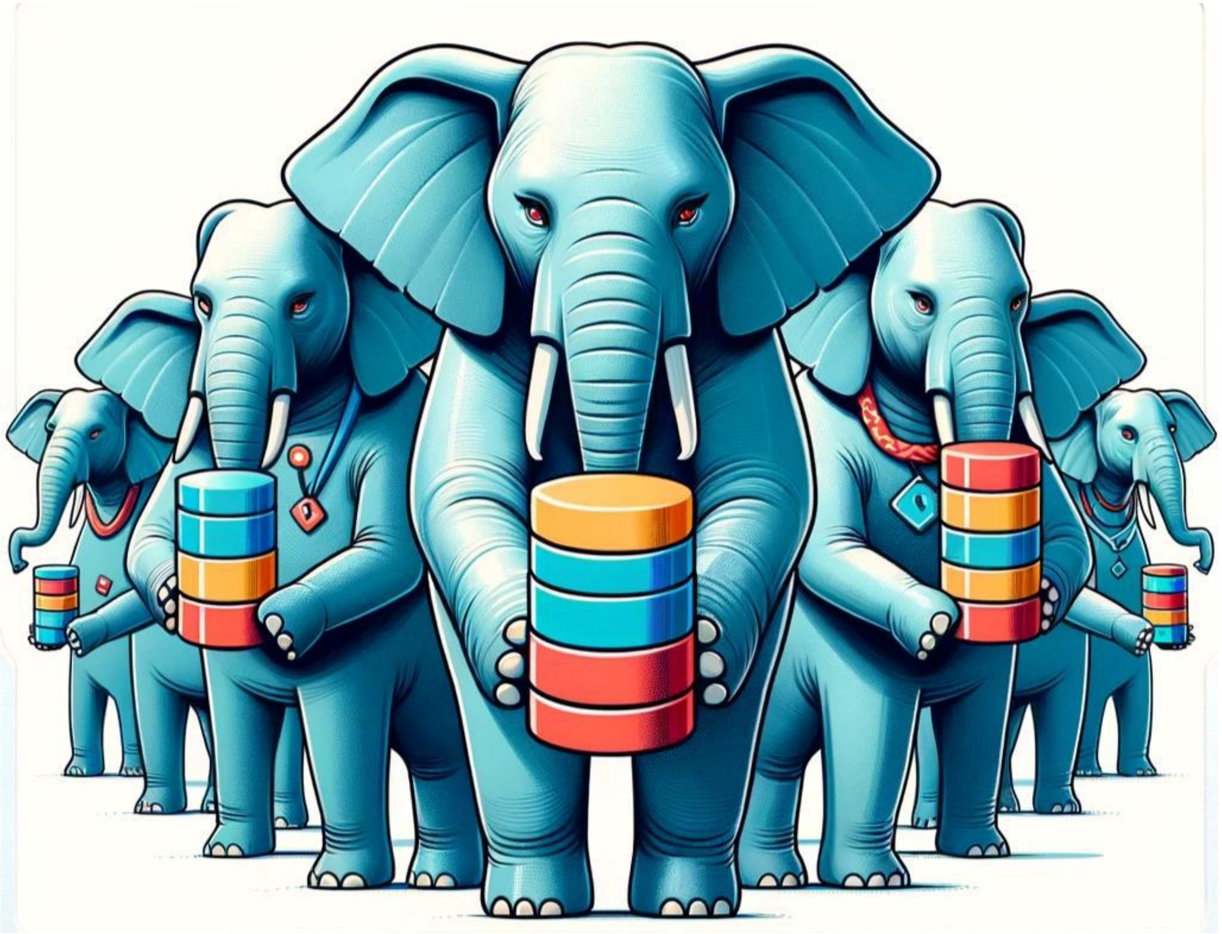
High cost / pricing

PostgreSQL is not designed for it

General guideline:
Consider using for complex workloads, but measure whether price-performance under load is better than a bigger machine.

Transparent sharding

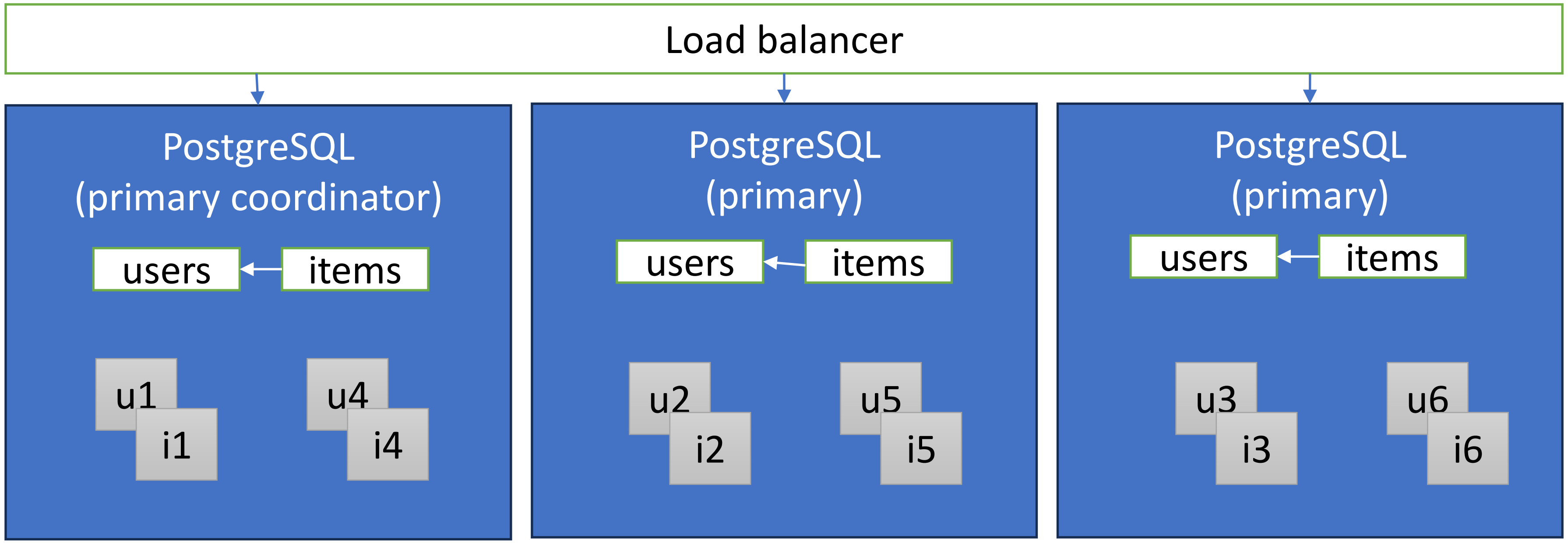
Like Citus



Transparent sharding

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

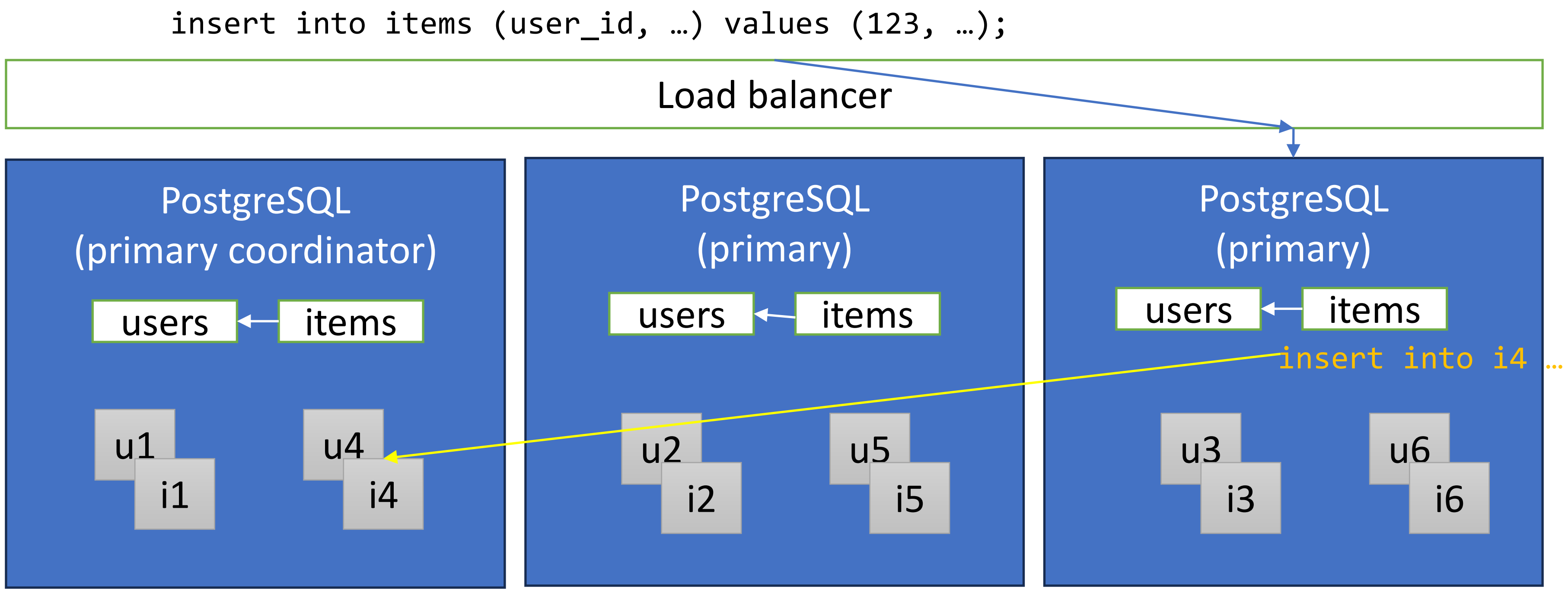
Queries & transactions are transparently routed / parallelized.



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

Single shard queries for operational workloads

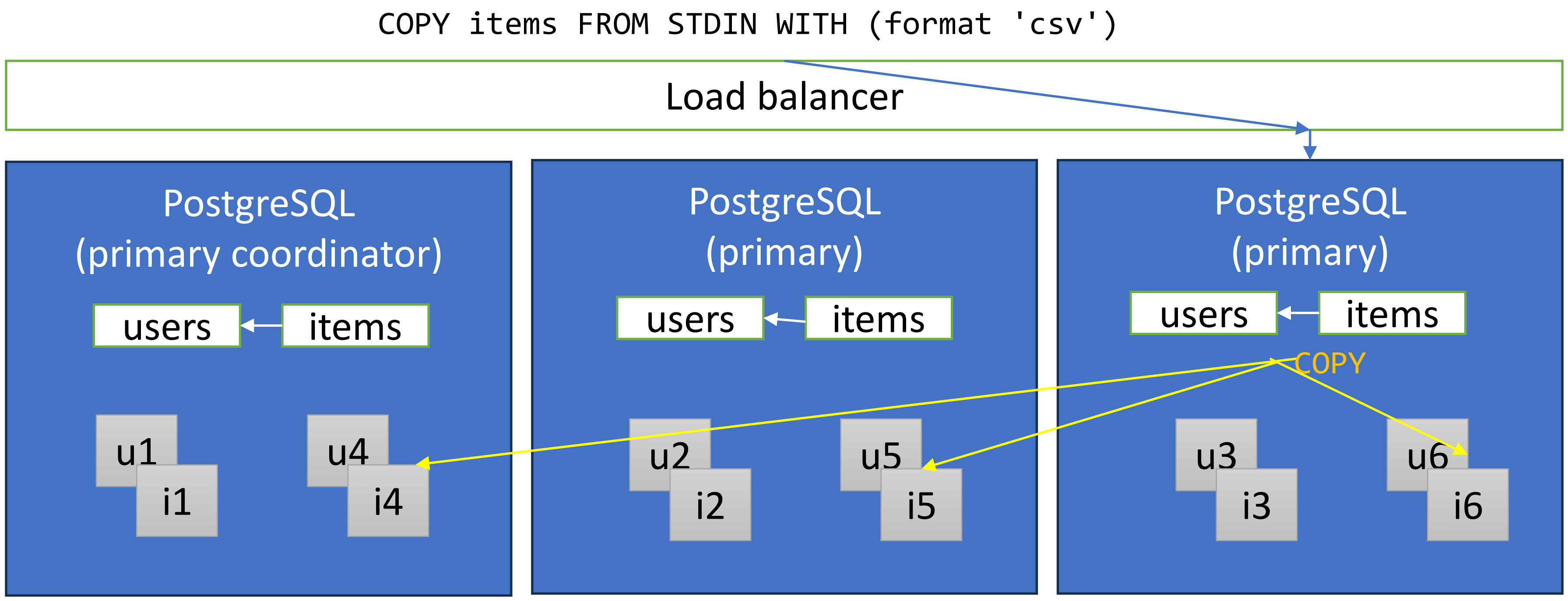
Scale capacity for handling a high rate of single shard key queries:



Per-statement latency can be a bottleneck!

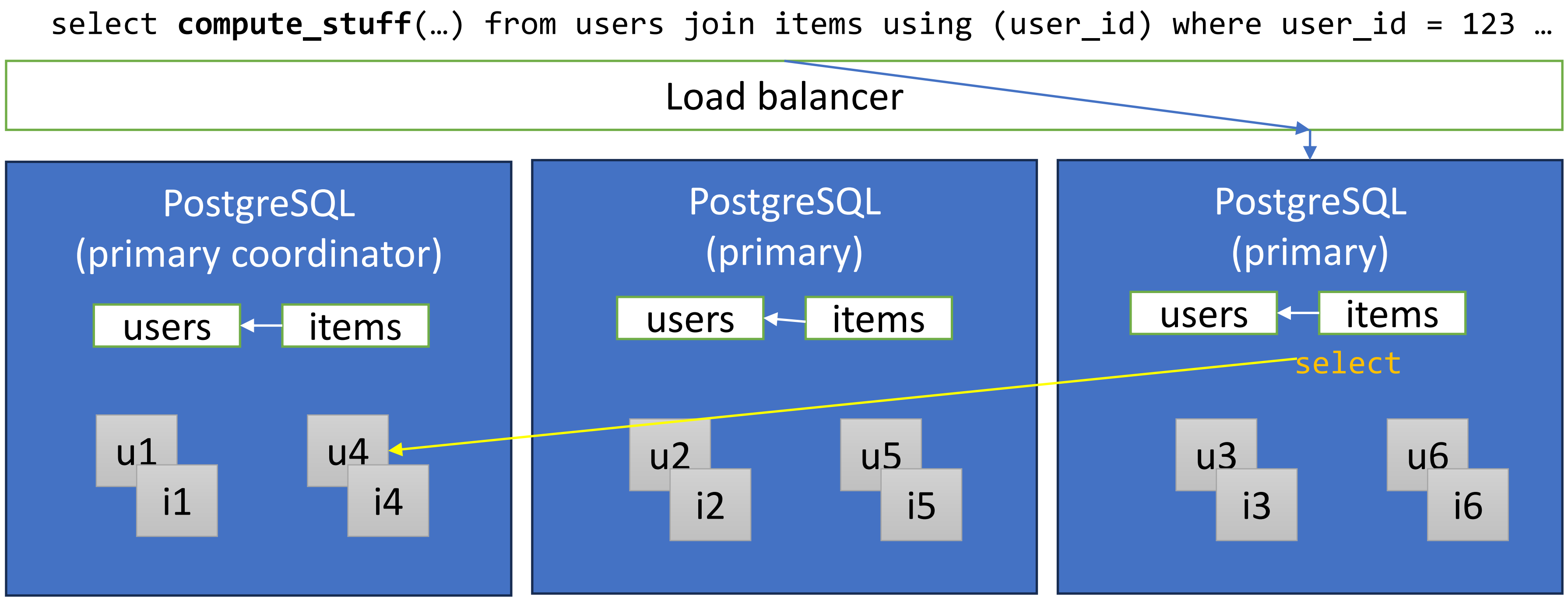
Data loading in sharded system

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



Compute-heavy queries

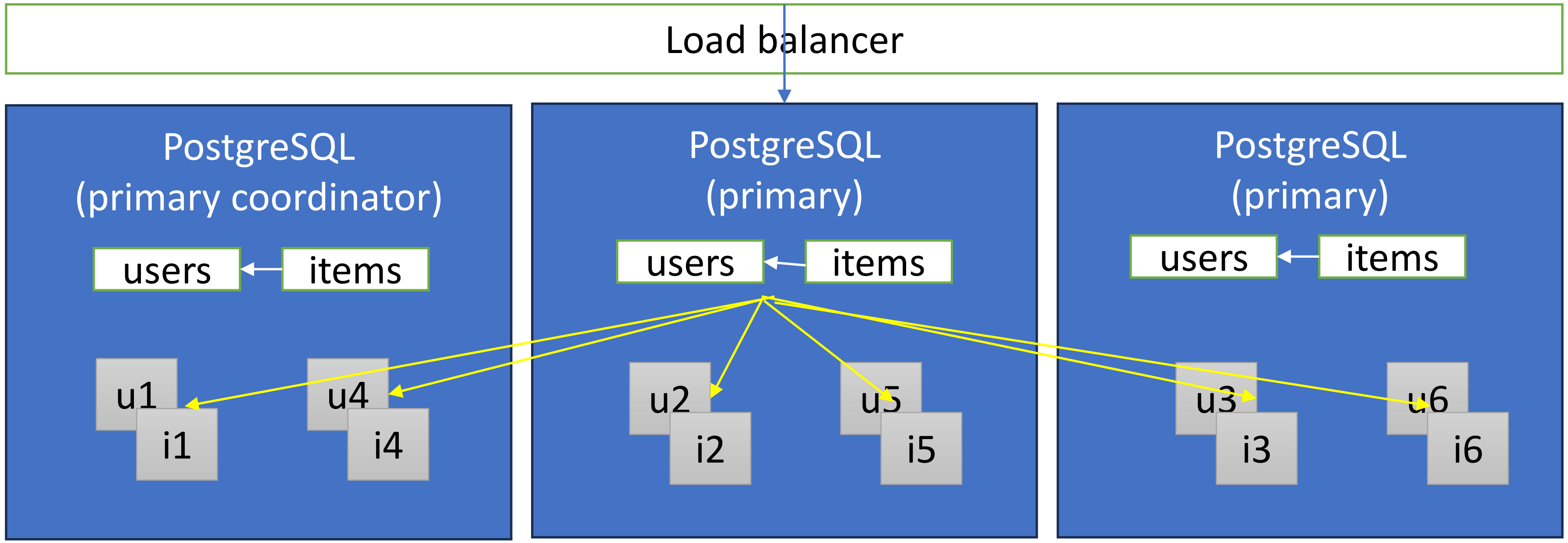
Compute-heavy queries (shard key joins, json, vector, …) get the most relative benefit



Multi-shard queries for analytical workloads

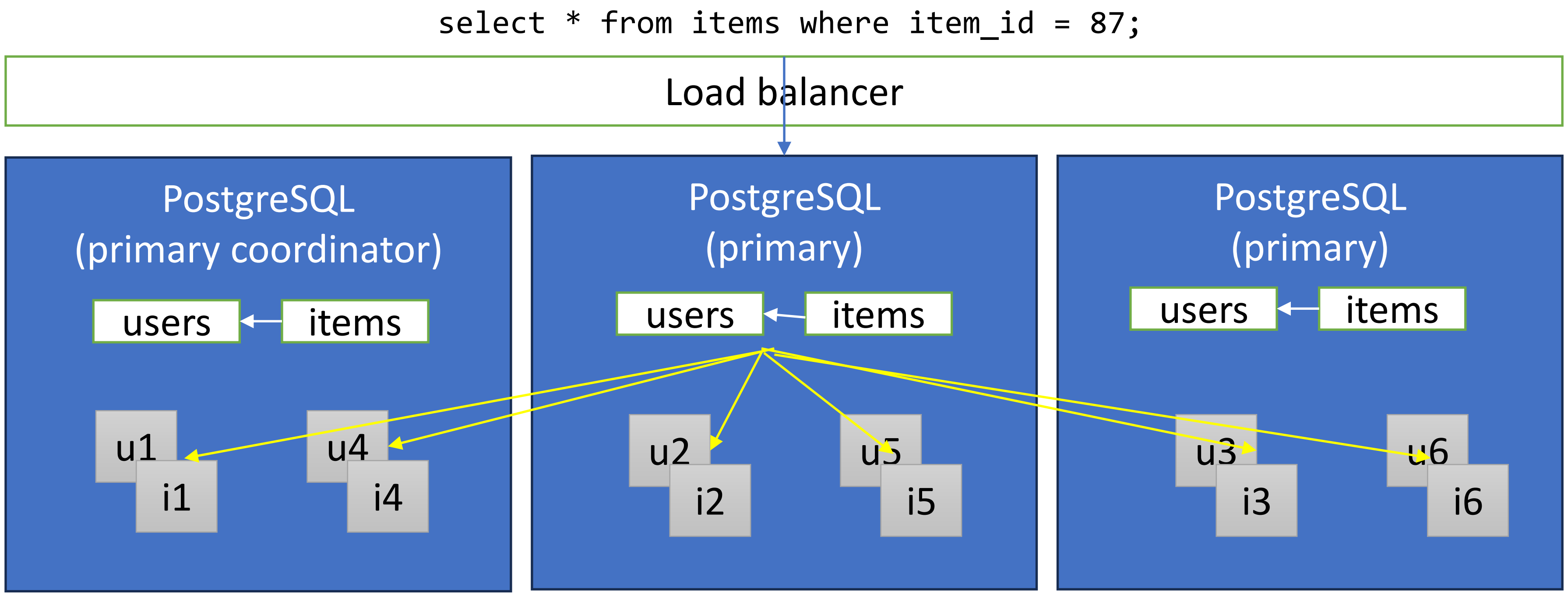
Parallel multi-shard queries can quickly answer analytical queries across shard keys:

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



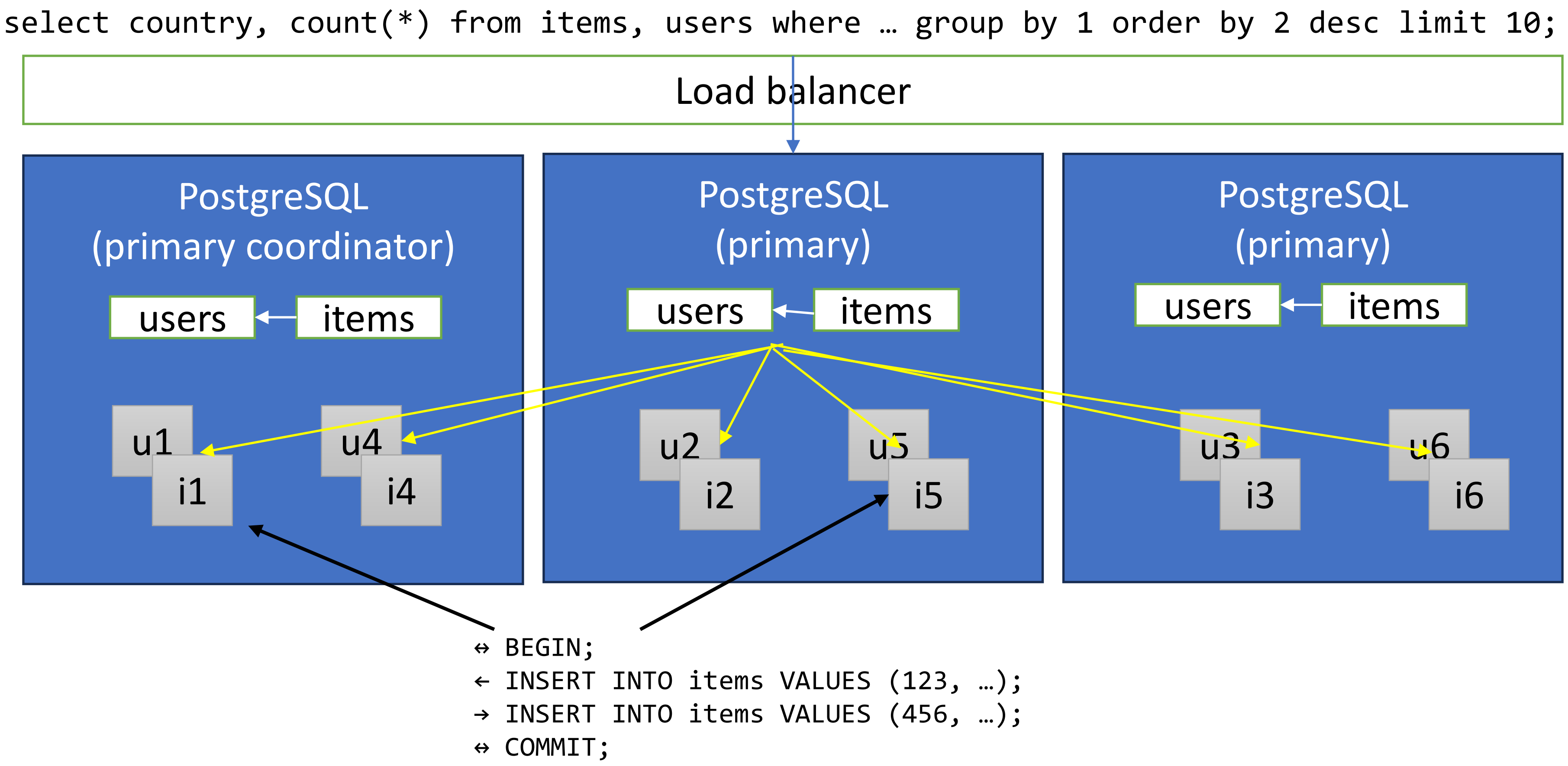
Multi-shard queries for operational workloads

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



Multi-shard queries for analytical workloads

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



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

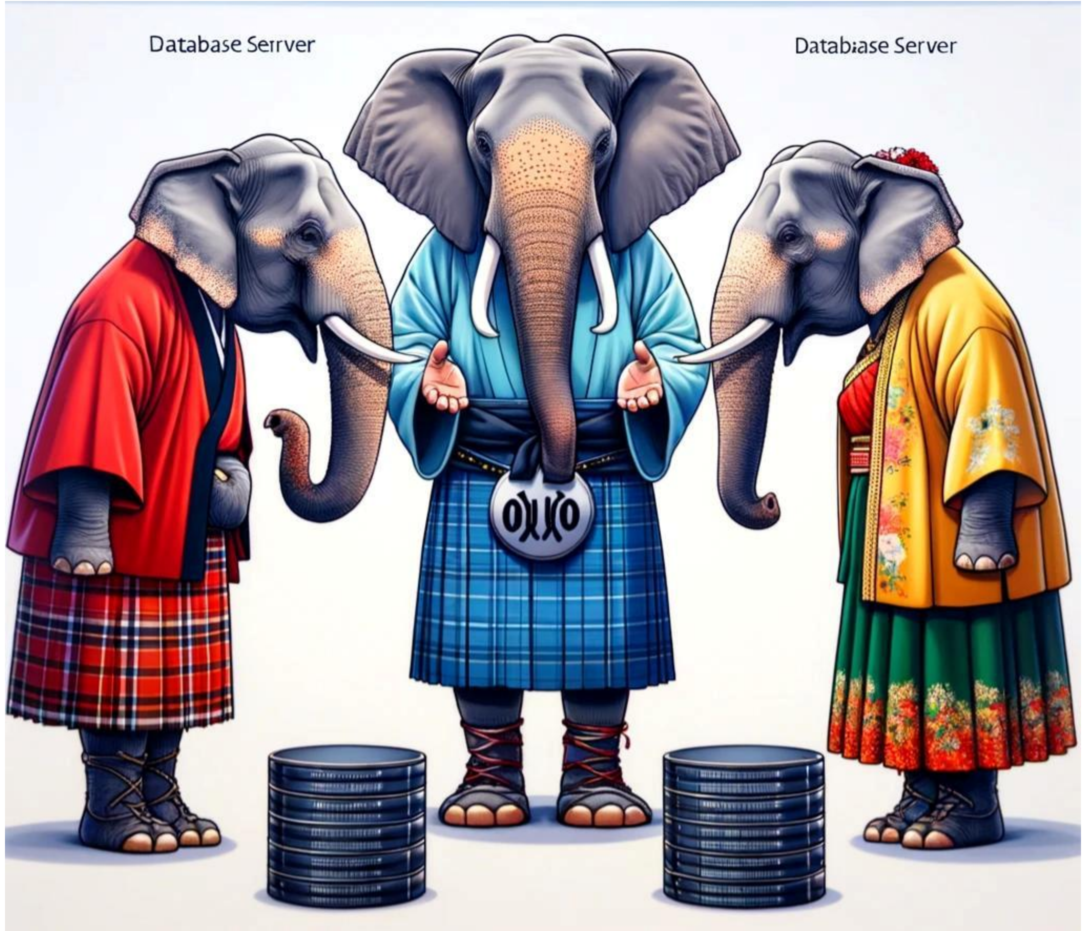
Data model decisions have high impact on performance

Snapshot isolation concessions

General guideline:
Use for multi-tenant apps, otherwise use for large working set (>100GB) or compute heavy queries.

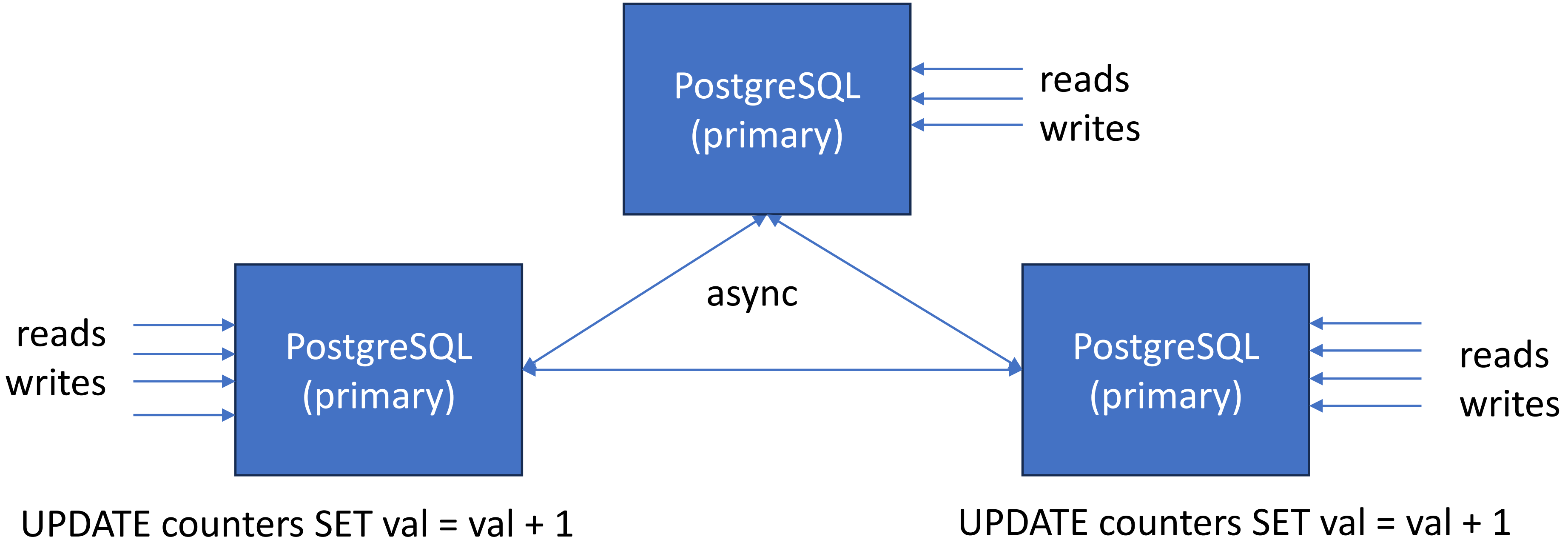
Active-active

Like BDR, pgactive, pgEdge, …



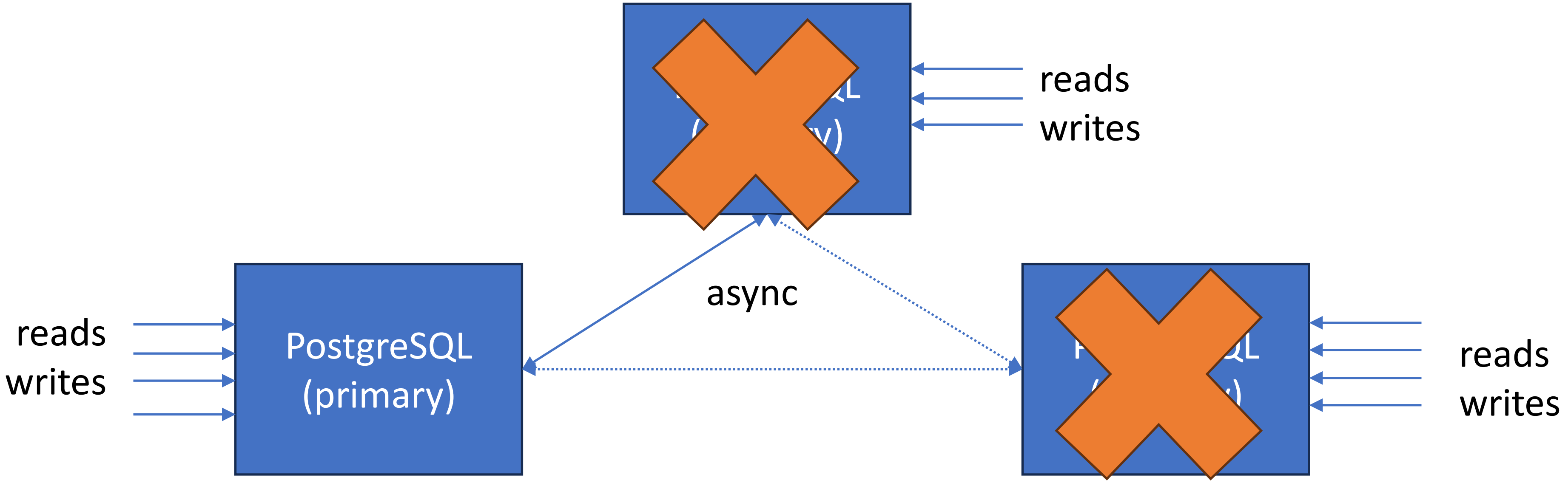
Active-active / n-way replication

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



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

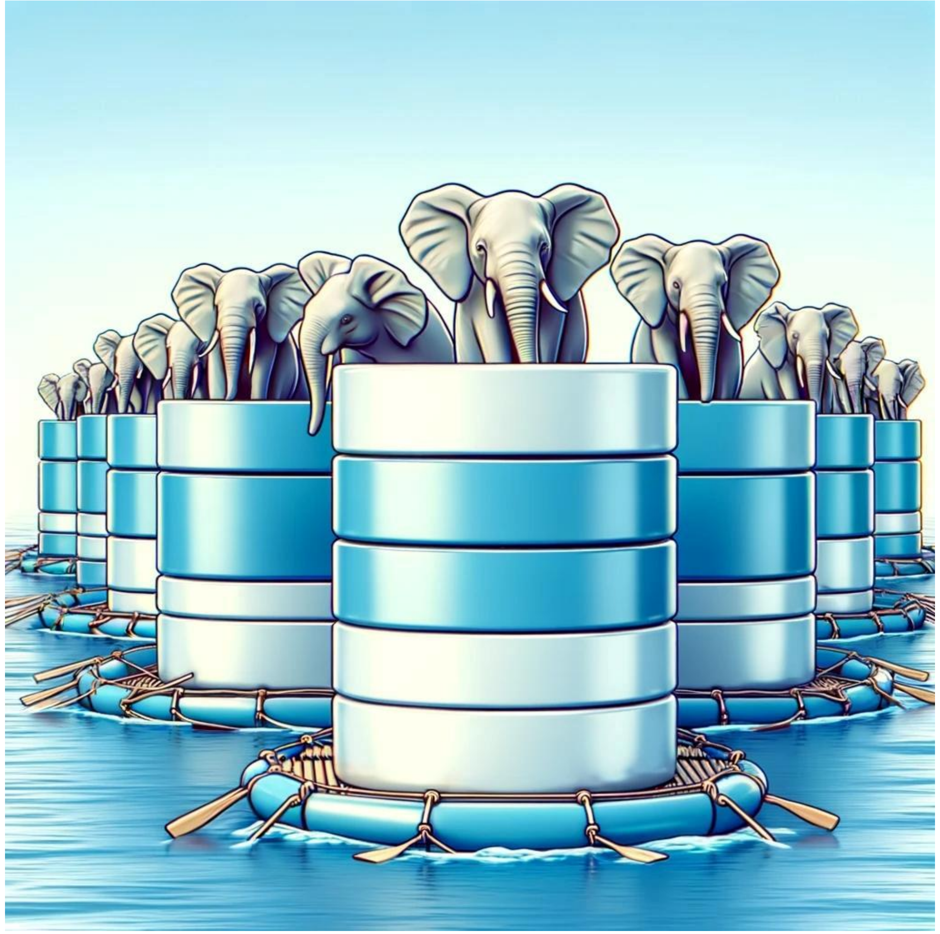
No monotonic read consistency

No linear history (updates might conflict after commit)

General guideline:
Consider only for simple data models (e.g. queues) and only if you really need the benefits.

Distributed SQL

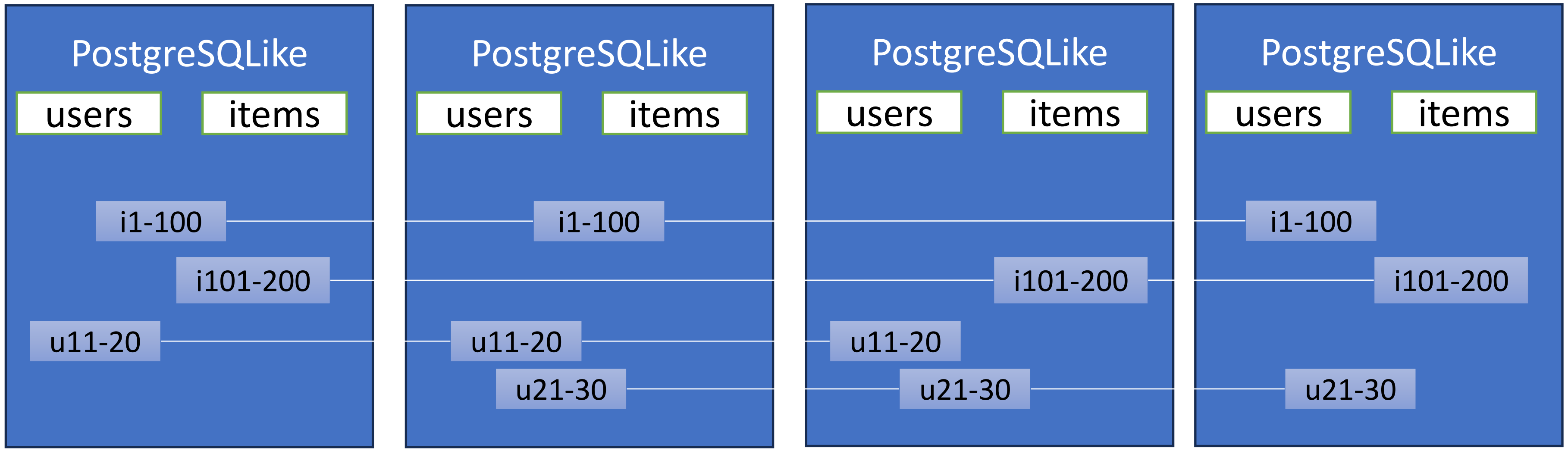
Like Yugabyte, CockroachDB, Spanner



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 internal operations incur high latency

No local joins in current implementations

Less mature and optimized than PostgreSQL

General guideline:
Just use PostgreSQL ;)

but for simple apps, the availability benefits can be useful

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

Questions?

Marco.slot@gmail.com