CS-310 Scalable Software Architectures

Lecture 9:
SQL Database Scaling

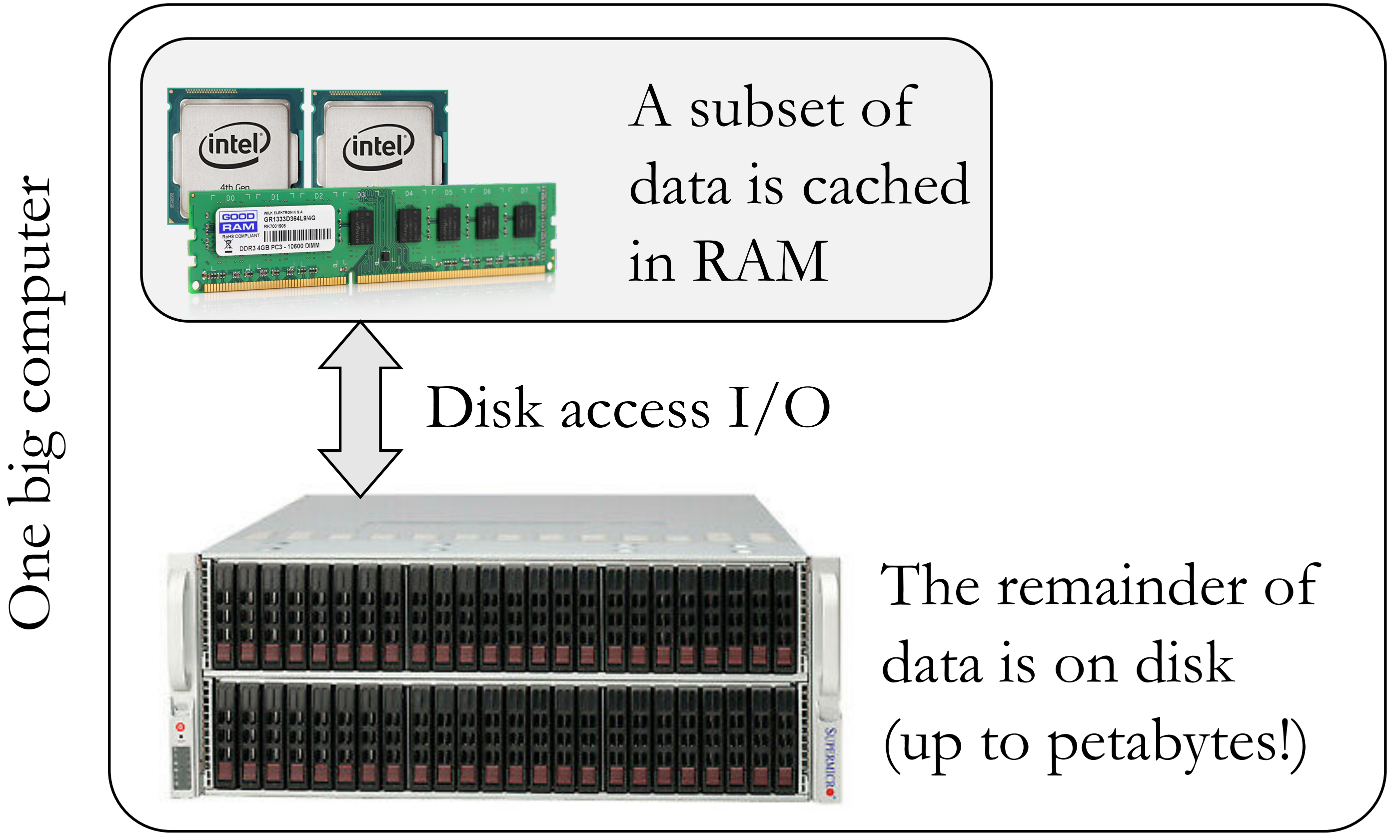
Steve Tarzia

Recap: Storage and Relational Databases

- **Persistent** storage requires special consideration due to slow performance and lack of language-level support.
  - **RAID** combines multiple disks for better capacity, storage, and fault tolerance.
- Databases solve lots of problems:
  - **scalability, persistence, indexing, concurrency,** etc.
  - Filesystems can solve some, but not all, of these problems.
- **Relational (SQL) databases** store data in tables.
- Developer defines the DB **schema** first (tables, columns, keys).
  - Rows are added during DB operation, and they must fit the schema.
- **Indexes** let us find rows quickly with value of one or more column.
- SQL query language lets us run analysis code "close to" data storage (filtering, aggregation – sum, count, min, max, avg, etc.).

Memory vs disk access in databases

- Remember that computers have a hierarchy of storage.
- RAM is 100,000x faster but ~100x smaller than disk.
- Database servers operate much faster when accessing data that is cached in RAM (memory).
  - RAM can be up to ~1TB.



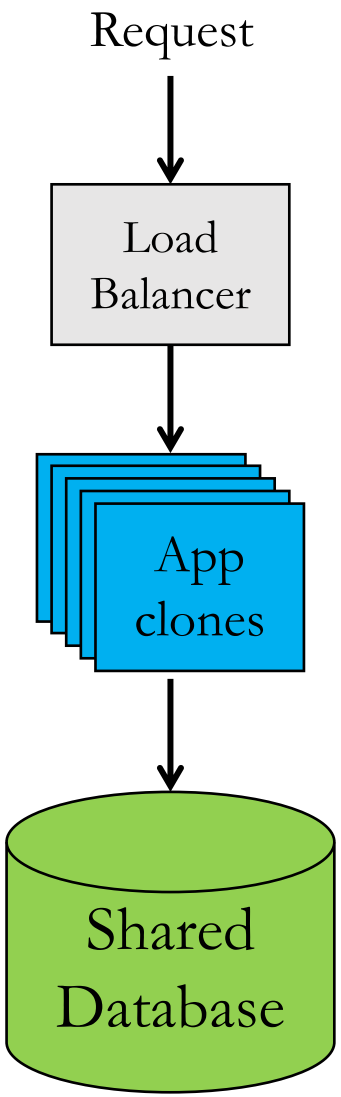
- **Goal**: fit entire **active data set** in RAM.
- Database/OS automatically cache most frequent data in RAM.

Databases are performance bottlenecks

- Why is load balancer not the bottleneck in this design?

  - Load balancer does much less work per request than the database.
- Why not create clones of the database?

  - Traditional scalable service design relies on a single shared database for **coordination**. App clones share state through the database.
  - However, we'll learn some tricks in this lecture.



Relational Database performance optimizations

- **Query planners** optimize order of table access and use of indexes:
  - SELECT * FROM user NATURAL JOIN post
    WHERE post.date > "2010-01-01" AND user.birth_year < 1920;

- RAM is used to store the most important data and indexes.
- Responses can be cached and replayed if data has not changed.

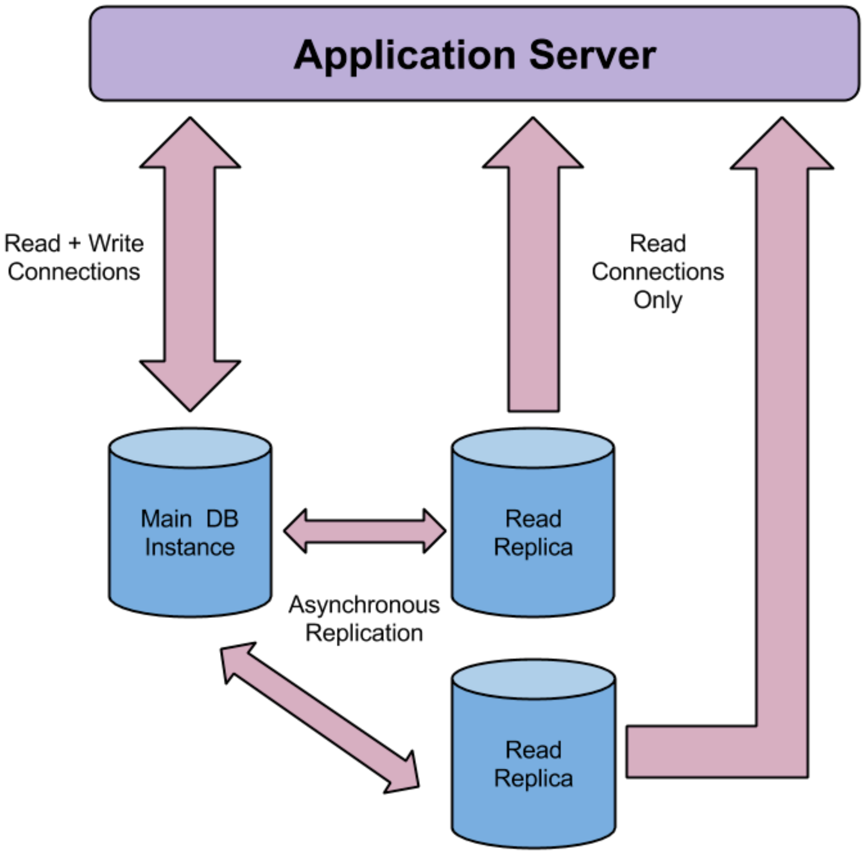
To avoid a database bottleneck:

- Avoid unnecessary queries (cache data in the frontend).
- Buy a really fast machine, with plenty of RAM for caching.
- Use the fastest possible disks (SSDs, RAID).

Vertical Scaling

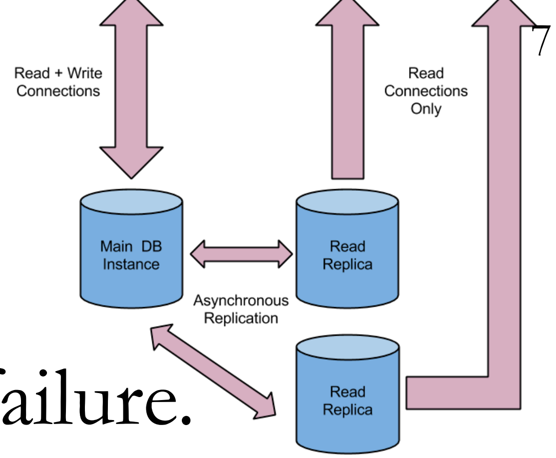
- Use **read replicas** or **sharding** – *Horizontal Scaling*

Read replicas



- Often, > 95% of DB traffic is **reads**.
- **Replica** servers each have a **full copy** of all the data, and they can handle read requests (SELECT).
- All writes (UPDATE, DELETE) must go to the **Primary** server (a.k.a. Main, Master)
- Data changes are pushed to read replicas.
- However, replicas may be slightly behind the primary, so read requests that are sensitive to consistency should use the primary.
- Too many replicas would make the data push process a bottleneck in the primary.

What limits the number of read replicas?



- This design is not infinitely scalable.
- The Primary is a central bottleneck and single point of failure.
- If there are N replicas, Primary must send N copies of each write.
- If there are R times as many reads as writes, and we want to equalize load on Primary and Replicas (to the max machine capacity), we get:

$$\text{primary\_load} = \text{repl\_load}$$

$$\text{primary\_reads} + \text{primary\_writes} + \text{data\_xfer} = \text{repl\_reads} + \text{repl\_writes} + \text{data\_xfer}$$

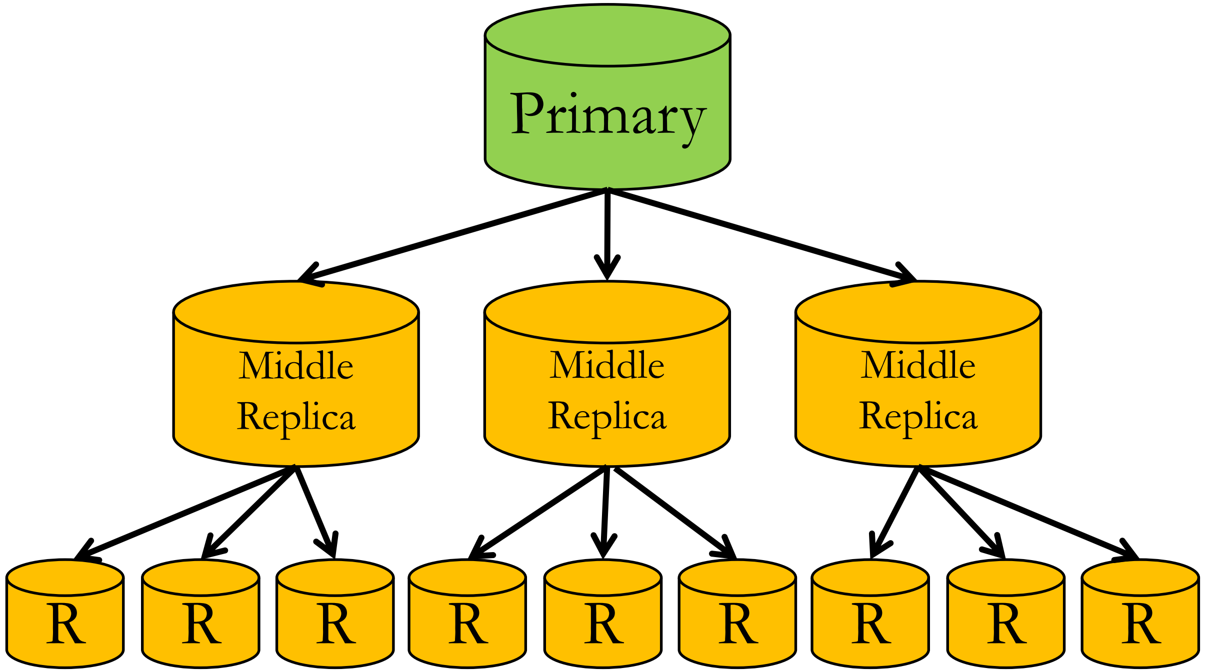
$$0 + 1 + N = R + 0 + 1$$

$$N = R$$

- Here, the optimal number of replicas is directly proportional to the ratio of reads to writes, perhaps about ten in a typical application.

Ideas for greater scaling of reads?

Multi-level replication can extend read-scalability



This is a kind of **horizontal scaling** for database reads.

Where do read requests go?

- To the bottom level replicas.
  (nine are shown in this diagram)

Why not read from middle replicas?

- Like the primary, they are busy pushing writes to their many children.

Where do write requests go?

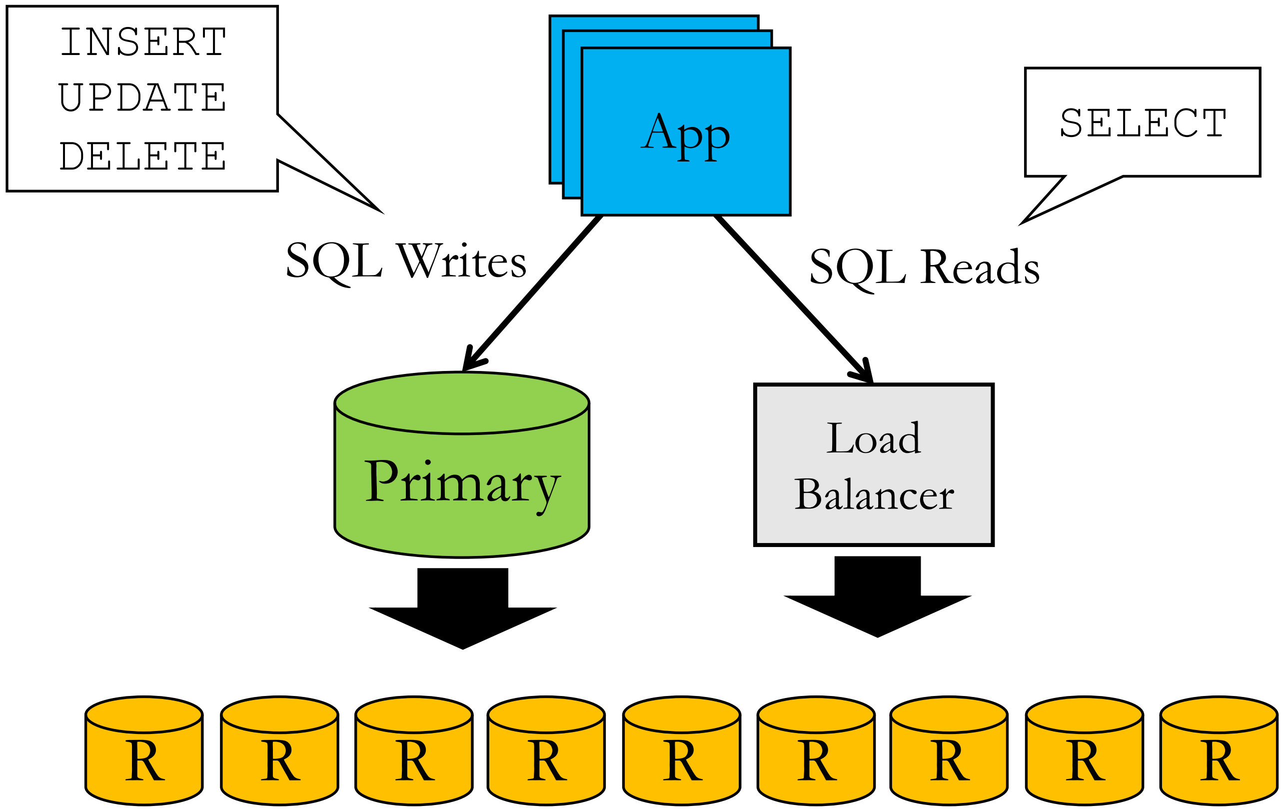
- To the one primary.

Can we add more replication levels (to achieve arbitrary *width*)?

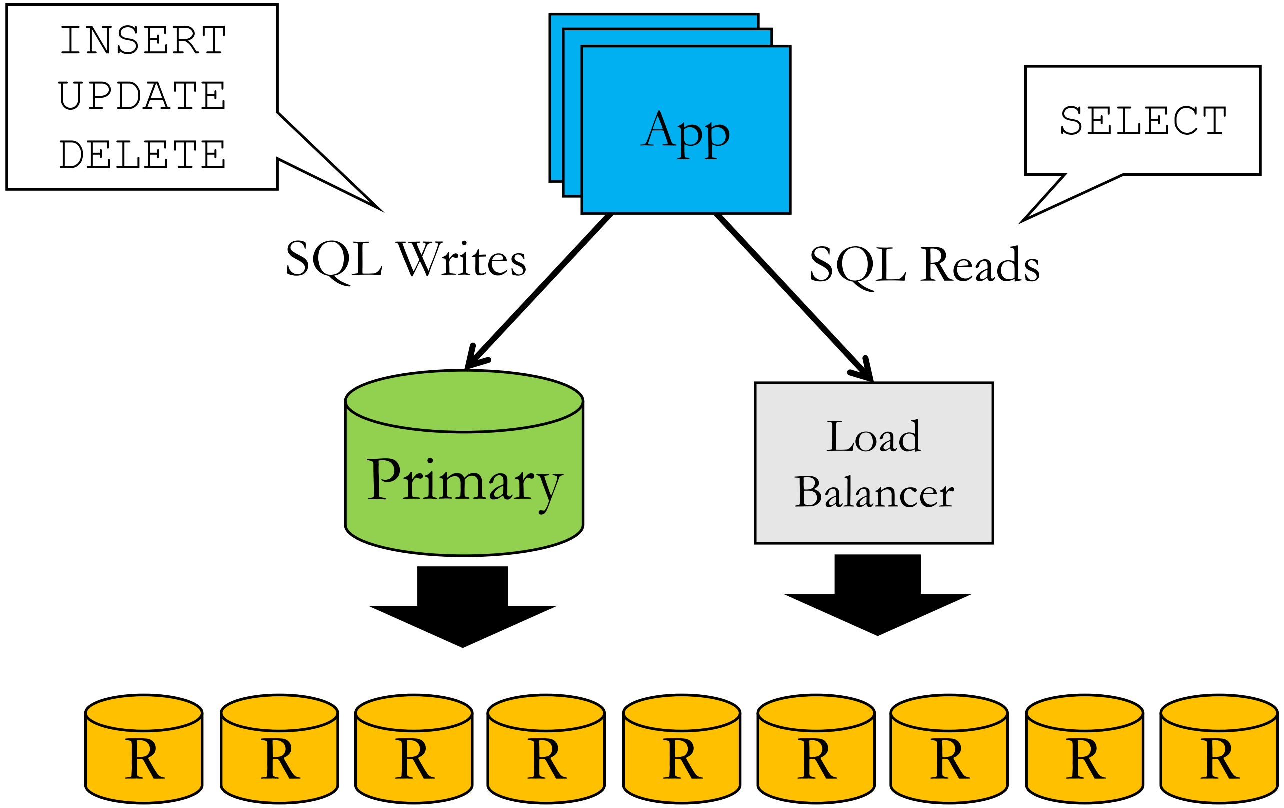
- Yes, but each level adds more **delay** between write at primary and data availability at read replicas.

How to use read-replicas?

- Put a load balancer in front of all the read replicas.
- This can be a NAT-type local LB or a simple software library. (eg.)



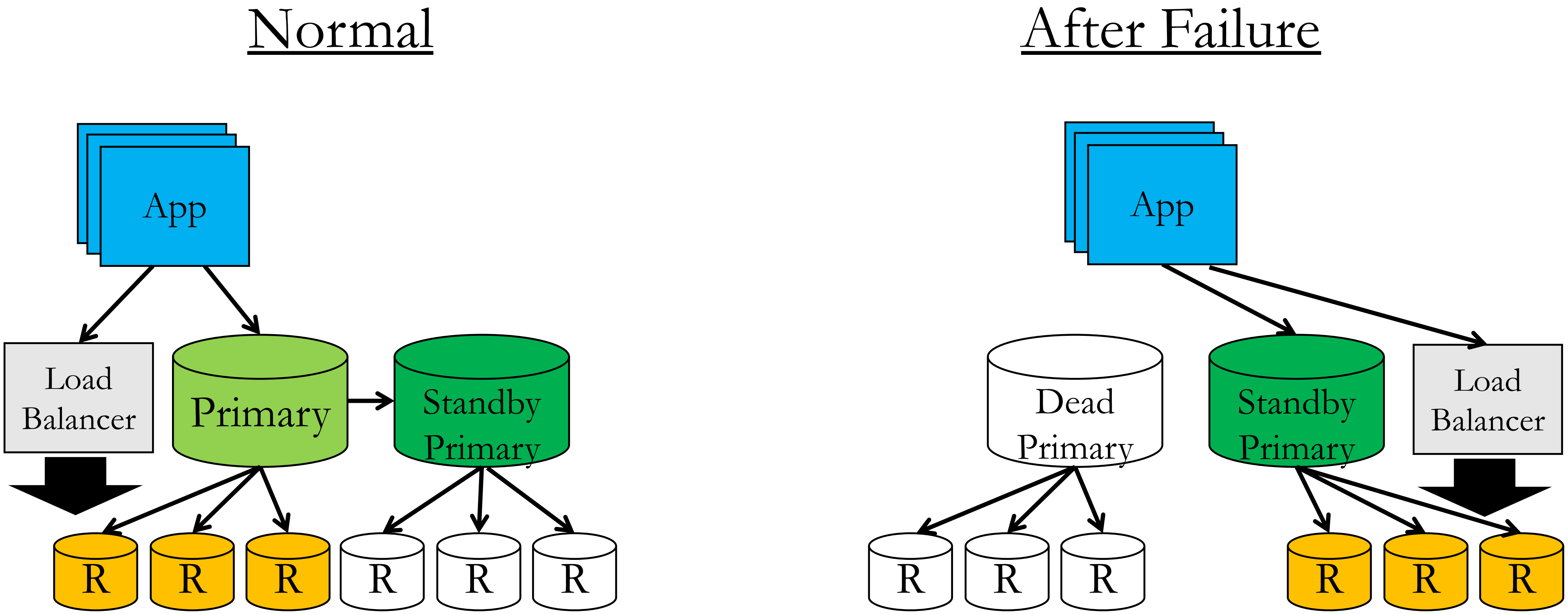
Replication shortcomings?



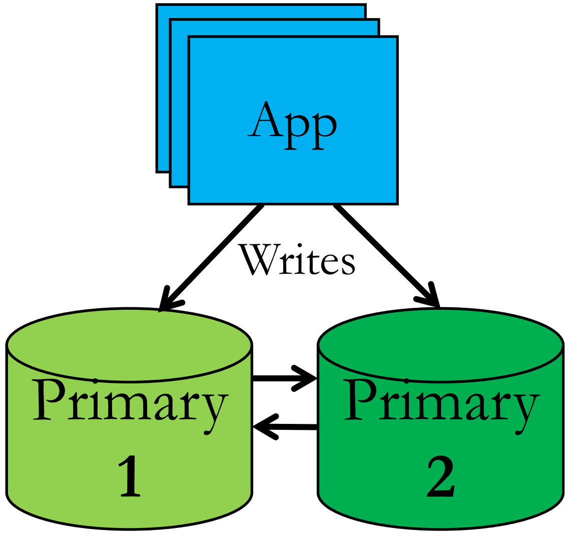
- **Writes** are not scalable. They are all handled by one DB machine.
- **Capacity** is not scalable. All the data must fit on each DB machine.
- Primary is a **single point of failure**.

Primary-primary failover for robustness

- Keep a "standby" primary ready to take over if the main primary fails.
- App will switch over to Standby if the main primary stops responding.



Why not allow writes to multiple primarys?



- Each Primary still must handle all the writes, though indirectly.
- Thus, the same performance bottleneck remains.

- Also, data can become **inconsistent** if operations happen concurrently.

How to scale **writes** and storage **capacity**?

- We already tried vertical scaling.
- How to implement **horizontal** scaling of a writes and capacity?

Some kind of **partitioning** is needed:

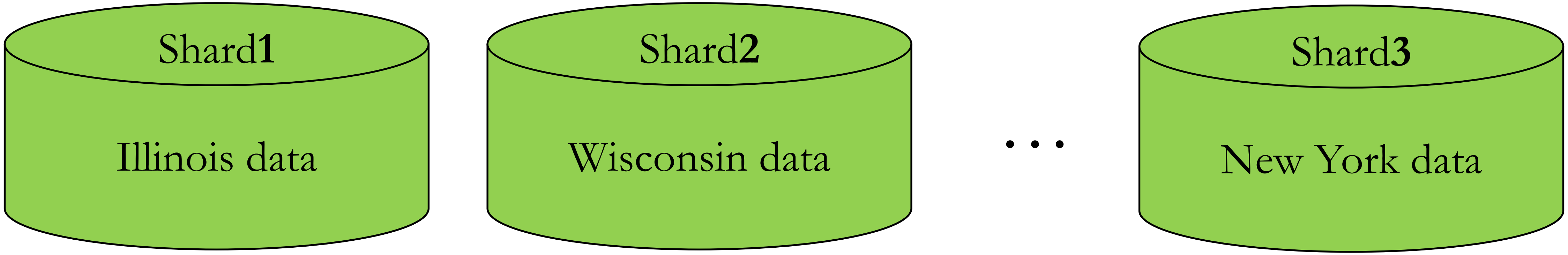
- **Functional partitioning**:
  - Create multiple databases storing different categories/types of data.
  - Eg.: three separate databases for: accounts, orders, and customers.
  - Cons:
    - Limits queries joining rows in tables in different DBs
    - Only a few functional partitions are possible. It's not highly scalable.
- **Data partitioning** is a more general approach…

Functional partitioning divides by **tables**

Data partitioning divides by **rows**

Sharding *(data partitioning)* relational databases

- Divide your data universe into disjoint subsets is called **shards**.
- For example: Consider parallelizing Facebook's database…
  - Maybe put Illinois users in one machine, Wisconsin in another, etc.
  - Each node stores rows for all tables, but only a subset of rows.



- **Sharding key** determines assignment of rows to shards.
- Relational databases usually don't support sharding natively, it must be somehow hacked at the application level.

Sharding example

Shard0

User

user	name
0	Steve
2	Yingyi
4	Alex

Post

user	date	text
0	04-25	Hi there…
0	04-27	Still tea…
2	03-12	Web scal…
2	04-25	Tips and…

Shard1

User

user	name
1	Guannan
3	Clarissa

Post

user	date	text
3	04-05	Box pl…
1	04-27	Sound…
1	03-12	Random…
3	04-27	Northw…

- In this example, `shard_id = user % 2`
- How to implement query for all posts by Steve?

All the data we need must be on Shard 0.

```
SELECT * FROM Post NATURAL JOIN User WHERE user=0?
```

Sharding example 2

Shard0

User

user	name
0	Steve
2	Yingyi
4	Alex

Post

user	date	text
0	04-25	Hi there…
0	04-27	Still tea…
2	03-12	Web scal…
2	04-25	Tips and…

Friend

user	friend
0	1
0	2
0	3
0	4
6	0

- How to implement query for latest 10 posts from Steve's friends?

```
SELECT * FROM User
 NATURAL JOIN Friend
 JOIN Post ON
    Post.user=
    Friend.friend
WHERE
User.name="Steve"
ORDER BY date DESC
LIMIT 10;
```

STOP and THINK

- Steve may be friends with users in all the shards; all shards must be queried.
- Query above will not work verbatim: user=0 row only exists in Shard0.
- Each shard can supply ten latest posts, app must manually merge them and choose the latest ten.

Sharding conclusions

Pros

Because each row is stored once:

✓**Capacity** scales.

✓Data is **consistent.**

If sharding key is chosen carefully:

✓Data will be **balanced**.

✓Many queries will involve only one or a few shards. There is no central bottleneck for these.

Cons

✗Cannot use plain SQL.

✗Queries must be manually adapted to match sharding.

✗If sharding key is chosen poorly, shard load will be imbalanced, either by capacity or traffic.

✗Some queries will involve all the shards. The capacity for handling such queries is limited by each single machine's speed.

Some Simple Scaling math

- **N** nodes
- **R** total request rate *(requests per second or another time frame)*
- Each node has the capacity to handle a maximum rate of requests **C**.

- If each request is sent to one node:
  - $R_{max} = NC$
- If each request is sent to a constant **k** number of nodes:
  - $R_{max} = NC/k = \mathcal{O}(NC)$

Scalable (increases with N)

- If each request is sent to all nodes:
  - $R_{max} = C$

Not Scalable

Summary

- **Read replicas** horizontally scale databases for reading.
  - Writes are done in one place and propagated to many replicas.
  - Data on a given replica may lag behind primary, but it's self-**consistent.**
  - Works well if writes are much less common than reads.
- Horizontal scaling of writes suggests **data partitioning**.
  - Each data row/element is assigned a single "home"
  - If not, consistency is very tricky (write race conditions for transactions).
- **Sharding** is data partitioning for SQL/relational DBs.
  - Works well for queries that can be handled within a single shard.
  - Sharding divides data along just one dimension, so inevitably some queries will involve all the nodes, and thus will not be scalable.
- Next time… NoSQL databases for more horizontal scaling!