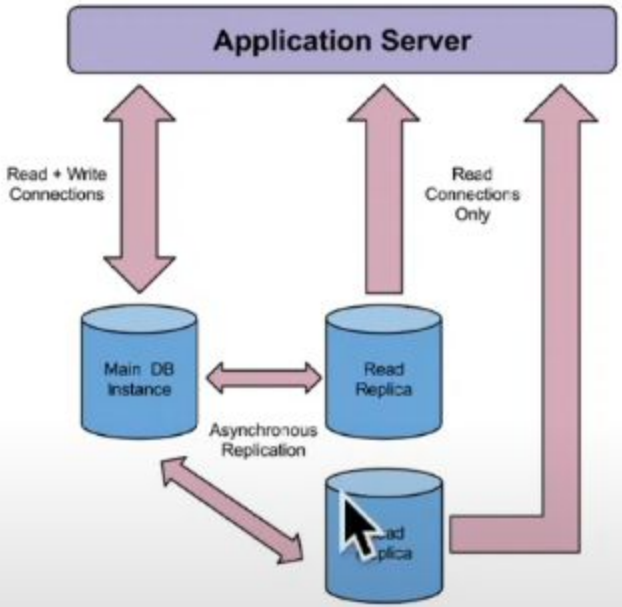
Summary

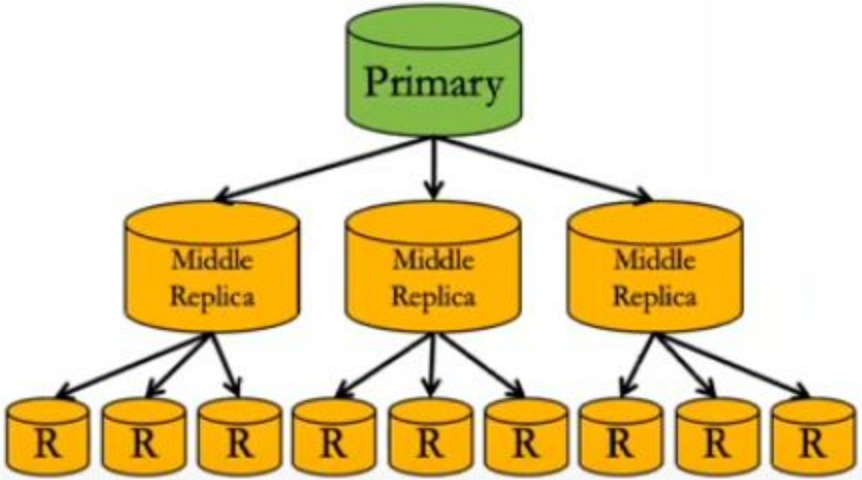
- **Read replicas** horizontally scale databases for reading.
  - Writes are done in one place and propagated to many replicas.
  - Data on a given replica may lag behind primary, but it's self-**consistent.**
  - Works well if writes are much less common than reads.
- Horizontal scaling of writes suggests **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!

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.

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?

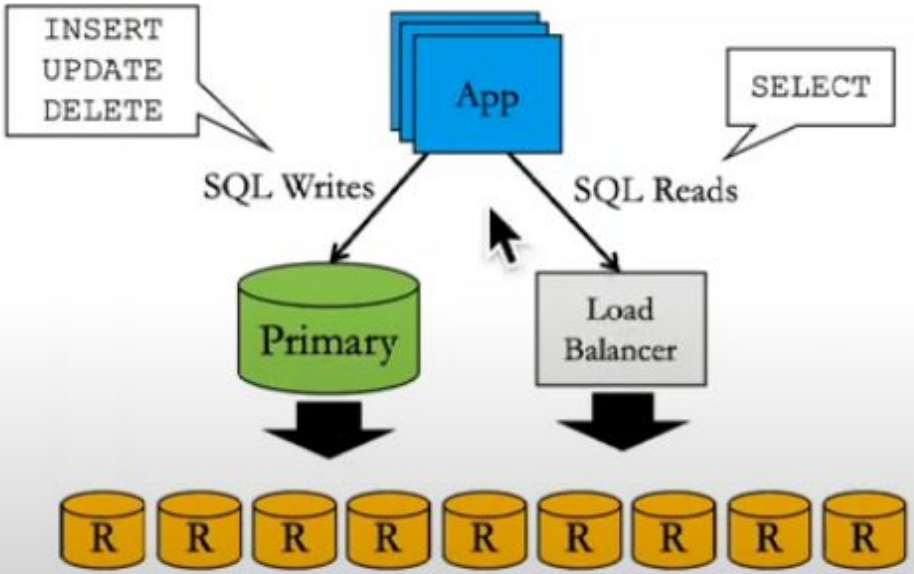
- To the one primary.

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.)



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.

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:

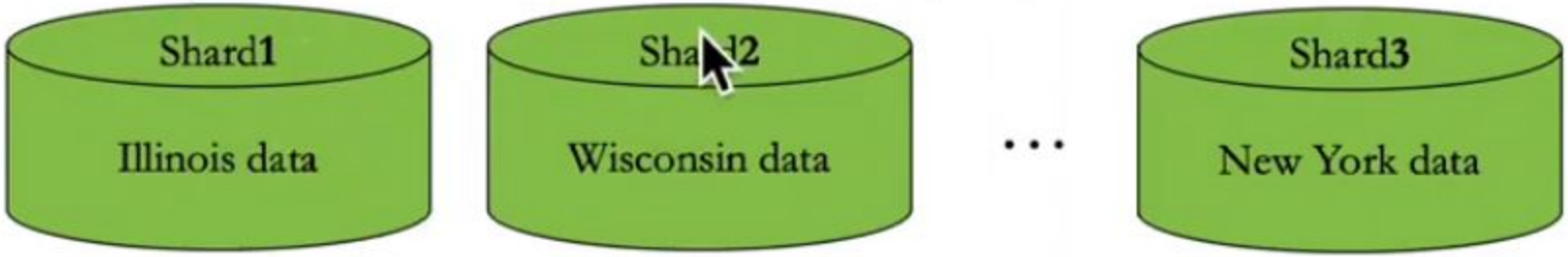
- **Functional partitioning**:
  - Create multiple databases storing different categories/types of data.
  - Eg.: three separate databases for: accounts, orders, and customers.
  - Cons:
    - Limits queries joining rows in tables in different DBs
    - Only a few functional partitions are possible. 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.

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!