Data Modeling for Relationships Handling and Data Distribution

Dr. Fabio Fumarola

Agenda

- How to deal with relationships
  - Graph Databases
  - Materialized Views
- Modeling for Data Access
- Distribution Models
  - Single server
  - Sharding
  - Master-Slave
  - Peer-to-Peer

How to Deal with Relationships

- Aggregates are useful to put together data that is commonly accessed jointly
- But we still have cases when we need to deal with relationships.
- Also in the previous examples we modeled some relation between orders and customers.
- In this cases we want to separate order and customer aggregates but with some kind of relationship.

How to Deal with Relationships

- The simplest way to provide such link is to embed the ID of the customer within the order's aggregate data.
- In this way if we need data from a customer record,
  - We read the customer ID and
  - We make another call to the database to read the customer data
- This can be applied several times but the database does not know about such relationship.

How to Deal with Relationships

- However, provide a way to make a relationship visible to the database can be useful.
- For example,
  - Document store make the content of an aggregate available to the database to form indexes.
  - Some key-value store allows us to put link information in metadata, supporting partial lookup (Riak).

Relationship, Aggregates and Updates

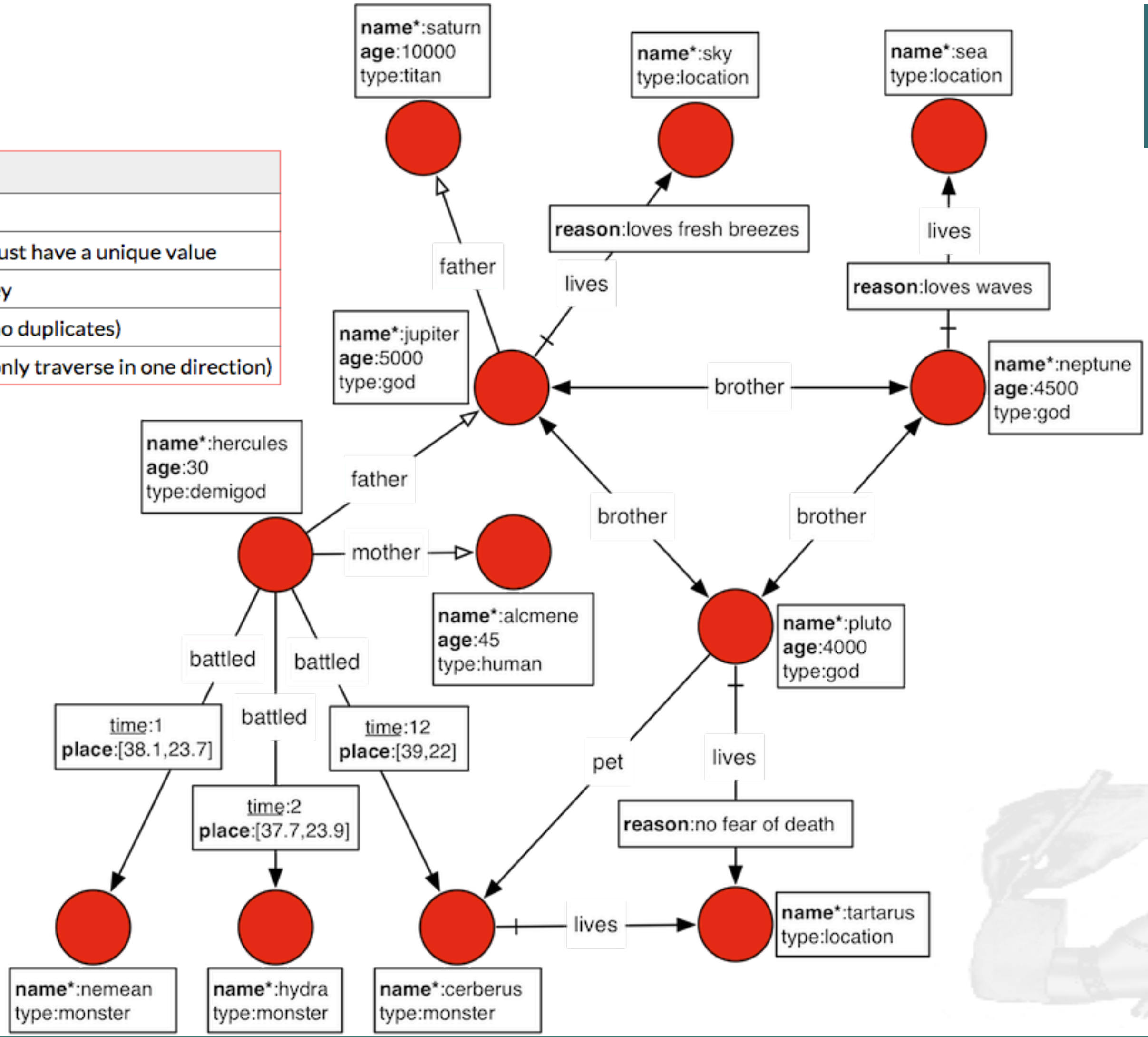
- In the end, aggregate-oriented databases treat aggregate as the unit of data-management.
- An atomicity is supported only at aggregate level.
- Thus, we need to model relation and relationships basing data access patterns
  – In the next slides we will explore how to deal with this.
- However, this problem make a good point to introduce another category of NoSql databases: Graph Databases

Graph Databases

- While other NoSql database are designed to run on a cluster, Graph databases are not.
- Graph databases are motivated by a different annoyance with relational databases.
- That is no SQL
- They have an opposite model, based on
  - Small records with complex interconnections

Example

visual symbol	meaning
bold key	a graph indexed key
bold key with star	a graph indexed key that must have a unique value
underlined key	a vertex-centric indexed key
hollow-head edge	a functional/unique edge (no duplicates)
tail-crossed edge	a unidirectional edge (can only traverse in one direction)



Titan:db by AURELIUS

Graph Databases

- In the previous figure we have an example of graph database known as *The Graph of the Gods*.
- Its nodes are very small, in number of attributes
- But, there is a rich structure of interconnections.
- Within this structure we can ask questions like:
  1. saturn.in('father').in('father').name
  2. hercules.out('father','mother').name

Graph Databases: features

- Their data models is very simple: nodes connected by edges (also called arcs)
  - FlockDB model only nodes and edges
  - Neo4J allows us to attach java object to nodes and edges
  - OrientDB stores Java Objects that are sub-classes of nodes and edges.
- They are ideal for modeling large scale data with complex relationship such as social networks, product preferences, etc.

Graph Databases: features

- It allows us to query the modeled network as nodes or edges based queries.
- Graph and relational databases are different for Relationships and Joins.
- In relational databases:
  - Relations are implemented using foreign keys.
  - The joins required to navigate around can get quite expensive (especially for strongly connected databases).

Graph Databases: features

- Graph databases:
  - Make the relationship using edges (direct and undirected)
  - Joins are done making traversal along the relationship (very cheap)
- The fast traversal is because most of the work from query time to insert time.
- This paying off in situations where querying performance is more important than insert speed.

Graph Databases: Query

- Another key difference is how query are done.
- In relational database you make a query by selecting some tuples using a where condition
- On a graph database you query starting from a node (saturn, hercules,…).
- However nodes can be indexed by an attributes such as ID.
- Thus, they expect most of the query work to be navigating realtionships

Graph Databases: Rationale

- The emphasis on relationships makes graph databases very different from aggregated-oriented
- However, these databases are:
  - More likely to run on a single server,
  - ACID transactions need to cover multiple nodes and edges
- The only think they have in common with aggregate-oriented databases is the rejection of the relational model.

Materialized views

- Wi aggregate-models we stressed their advantage of aggregation
  - If you want to access orders, it is useful to have orders stored in a single unit
- We know that aggregated orientation has the disadvantage over analytics queries.
- A first solution to reduce this problem is by building an index but we are still working against aggregates

Materialized Views: RDBMS

- Relational databases offer another mechanism to deal with analytics queries, that is **views**.
- A view is like A relational table but defined by computation over the base tables
- When a views is accessed the databases execute the computation required.
- Or with materialized views are computed in advance and cached on disk

Materialized Views: NoSQL

- NoSQL database do not have views, but they have pre-computed and cached queries.
- This is a central aspect for aggregate-oriented databases since some queries may not fit with the aggregate structure.
- Often, materialized views are created using map-reduce computation.

Materialized Views: Approaches

- There are two approaches: Eager and Lazy.
- Eager
  - the materialized view is updated when the base tables are updated.
  - This approach is good when we have more frequent reads than writes
- Lazy:
  - The updates are run via batch jobs at regular interval
  - It is good when the data updates are not business critical

Materialized Views: Approaches

- Moreover, we can create views outside the database.
- We can read the data, computing the view, and saving it back to the database (MapReduce).
- Often the databases support building materialized views themselves (Incremental MapReduce).
  - We provide the need computation and
  - The databases execute computation when needed

Materialized Views: Approaches

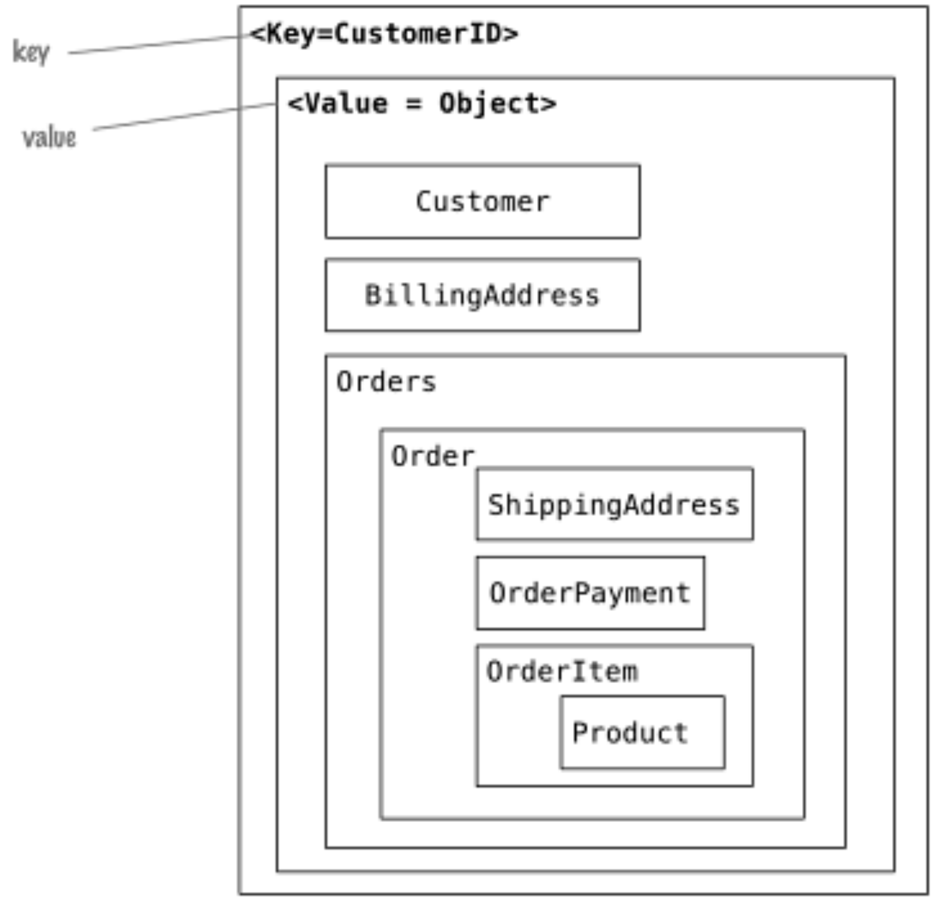
- They can be used within the same aggregate.
  - An order document might include an order summary element of the same order which can be used as view
- In column-oriented databases different column-families are used for materialized views.
- An advantage of doing this is that it allows you to update views with the same atomic operation.

Modeling for Data Access

- Now we can consider in detail how to model data with aggregate-orientation
- We need to consider:
  1. How the data is going to be read
  2. What are the side effects on data related to those aggregates

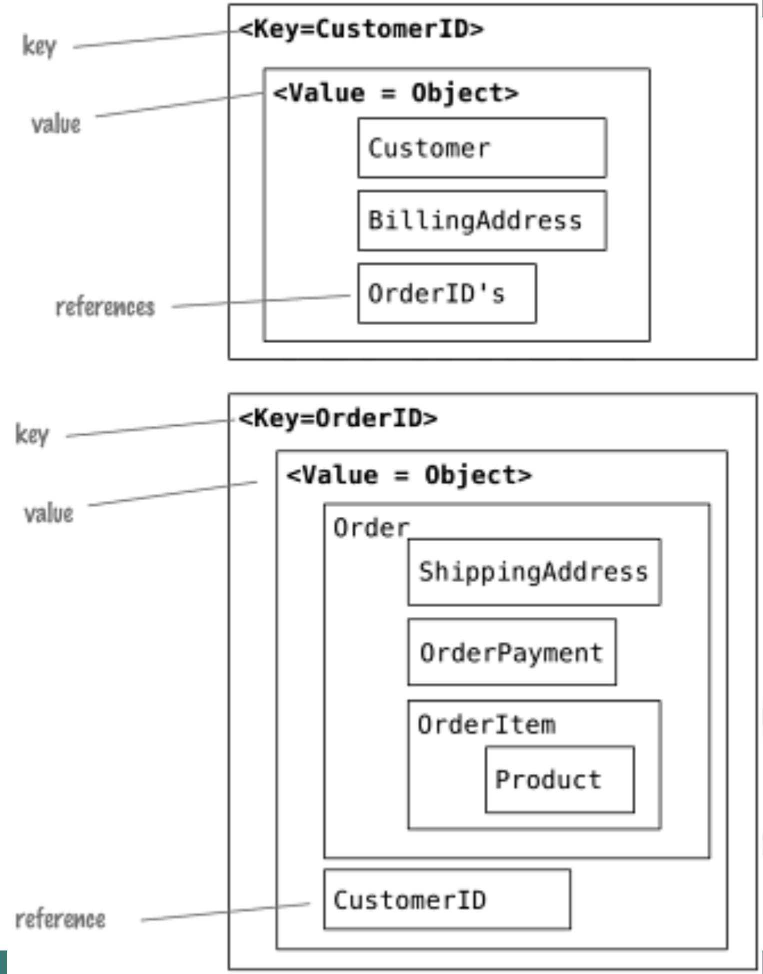
Modeling: key-value store

- In this scenario, the application can read the customer's information and all the related data by using the key.
- If we need to read the products sold in each order we need to read and parse all the object



Modeling: document

- When references are needed, we can switch to documents and query inside the documents
- With references we can find orders independently from the customer.
- With orderId reference we can find all Orders
- But we have to push Order reference into Customer

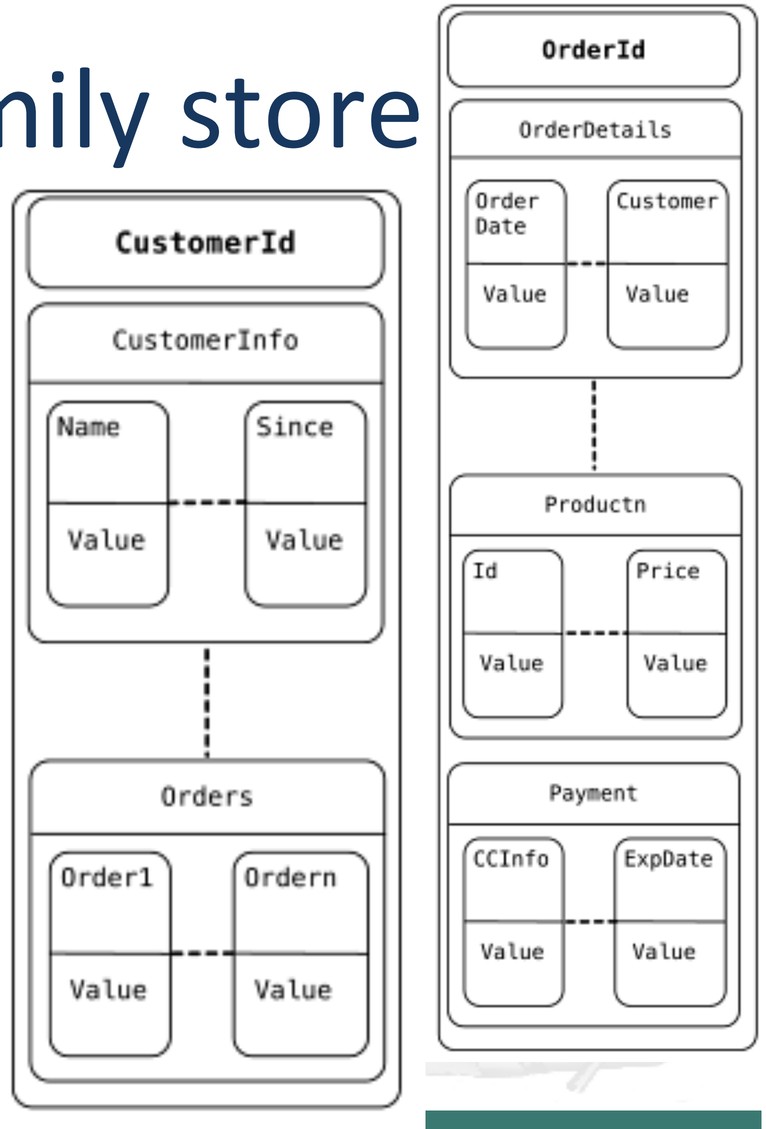


Modeling: column-family stores

- We have the benefit of the columns being ordered,
- It allows us to name consistently the columns that are accessed frequently so that they are fetched first.
- When using column-families to model data, remember to do it per query requirements and not for writing purpose.
- The general rule is to make it easy to query and denormalize data during write.

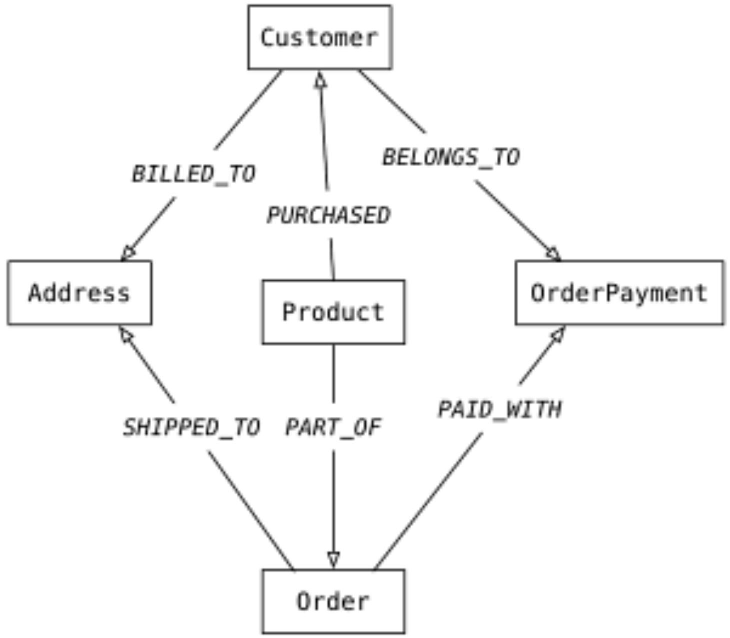
Modeling: column-family store

- There are multiple ways to model data.
- One way is to store the Customer and Order in different column families.
- Note that the reference to all the orders placed by customer are in the Customer column-family.
- Similarly other denormalizations are generally done so that query (read) performance is improved.



Modeling: Graph Databases

- We model all objects as nodes and relationship as edges.
- Relationship have types and direction significance.
- Relationship have names, that let we traverse the graph.
- Graph databases allows us to make query such as: list all the customers that purchased “akka in action”



DISTRIBUTION MODELS

Distribution Models

- We already discussed the advantages of scale up vs. scale out.
- Scale out is more appealing since we can run databases on a cluster of servers.
- Depending on the distribution model the data store can give us the ability:
  1. To handle large quantity of data,
  2. To process a greater read or write traffic
  3. To have more availability in the case of network slowdowns of breakages

Distribution Models

- These are important benefit, but they come at a cost.
- Running over a cluster introduces complexity.
- There are two path for distribution:
  - Replication and
  - Sharding


Distribution Model: Single Server

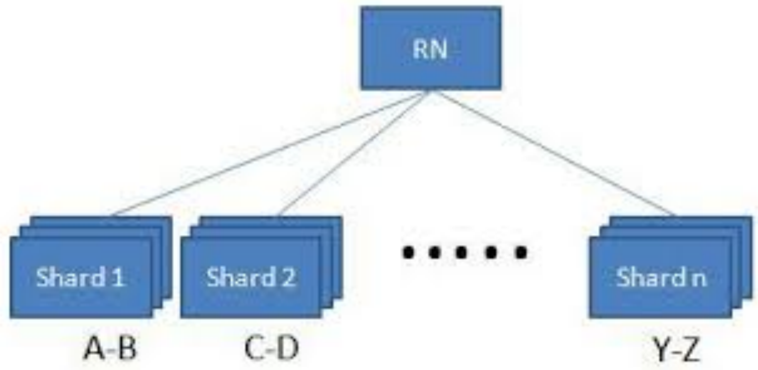
- It is the first and simplest distribution option.
- Also if NoSQL database are designed to run on a cluster they can be used in a single server application.
- This make sense if a NoSQL database is more suited for the application data model.
- Graph database are the more obvious
- If data usage is most about processing aggregates, than a key or a document store may be useful.

Sharding

- Often, a data store is busy because different people are accessing different part of the dataset.
- In this cases we can support **horizontal scalability** by putting different part of the data onto different servers (**Sharding**)
- The concept of sharding is not new as a part of application logic.
- It consists in put all the customer with surname A-D on one shard and E-G to another

Sharding

- This complicates the programming model as the application code needs to distributed the load across the shards
- In the ideal setting we have each user to talk one server and the load is balanced.
- Of course the ideal case is rare.



Sharding: Approaches

- In order to get the ideal case we have to guarantee that data accessed together are stored in the same node.
  - This is very simple using aggregates.
- When considering data distribution across nodes.
  - If access is based on physical location, we can place data close to where are accessed.

Sharding: Approaches

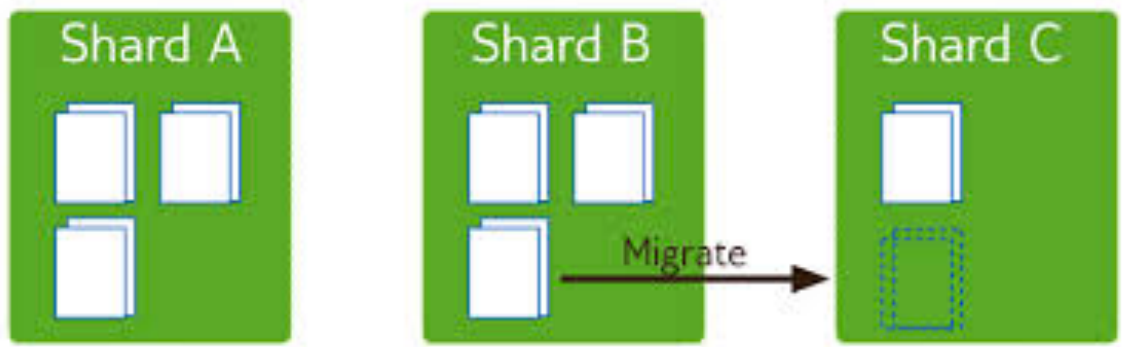
- Another factor is trying to keep data balanced.
- We should arrange aggregates so they are evenly distributed in order that each node receive the same amount of the load.
- Another approach is to put aggregate together if we think they may be read in sequence (BigTable).
- In BigTable as examples data on web addresses are stored in reverse domain names.

Sharding and NoSQL

- In general, many NoSQL databases offers **auto-sharding.**
- This can make much easier to use sharding in an application.
- Sharding is especially valuable for performance because it improves read and write performances.
- It scales read and writes on the different nodes of the same cluster.

Sharding and Resilience

- Sharding does little to improve resilience when used alone.
- Since data is on different nodes, a node failure makes shard's data unavailable.
- So in practice, sharding alone is likely to decrease resilience.



Sharding: right time

- Some databases are intended to be sharded at the beginning
- Some other let us start with a single node and then distribute and shard.
- However, sharding very late may create trouble
  - especially if done in production where the database became essentially unavailable during the moving of the data to the new shards.

Master-Slave Replication

- In this setting one node is designated as the master, or primary ad the other as slaves.
- The master is the authoritative source for the date and designed to process updates and send them to slaves.
- The slaves are used for read operations.
- This allows us to scale in data intensive dataset.

Master-Slave Replication

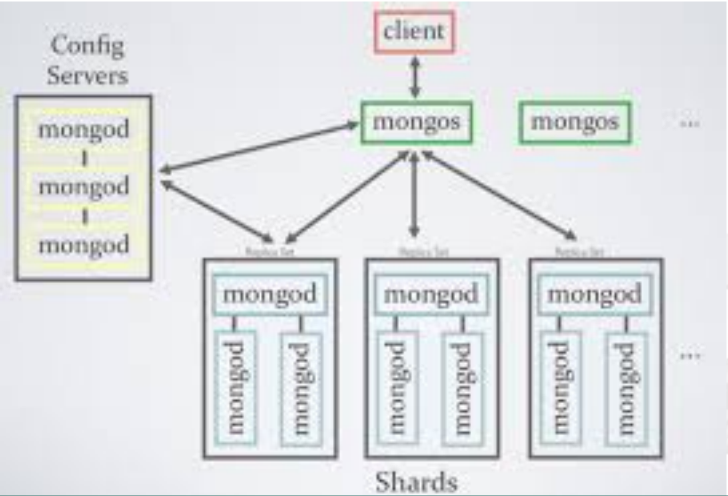
- We can scale horizontally by adding more slaves
- But, we are limited by the ability of the master in processing incoming data.
- An advantage is **read resilience**.
  - Also if the master fails the slaves can still handle read requests.
  - Anyway writes are not allowed until the master is not restored.

Master-Slave Replication

- Another characteristic is that a slave can be appointed as master.
- Masters can be appointed manually or automatically.
- In order to achieve resilience we need that read and write paths are different.
- This is normally done using separate database connections.

Master-Slave Replication

- Replication in master-slave have the analyzed advantages but it come with the problem of **inconsistency**.
- The readers reading from the slaves can read data not updated.

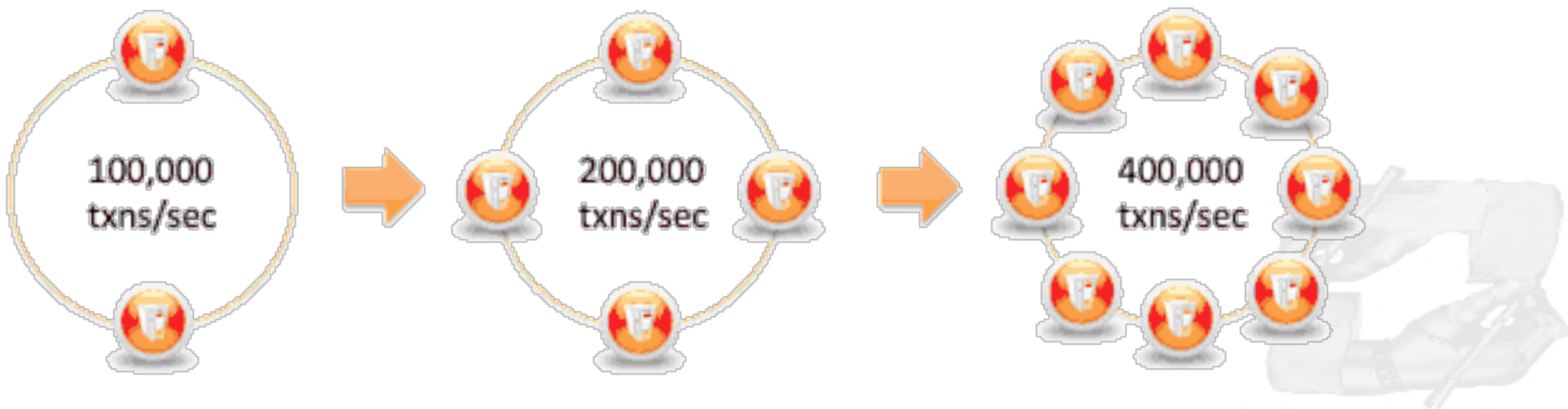


Peer-to-Peer Replication

- Master-Slave replication helps with read **scalability** but has problems on scalability of writes.
- Moreover, it provides **resilience** on read but not on writes.
- The master is still a single point of failure.
- Peer-to-Peer attacks these problems by not having a master.

Peer-to-Peer Replication

- All the replica are equal (accept writes and reads)
- With a Peer-to-Peer we can have node failures without lose write capability and losing data.



Peer-to-Peer Replication

- Furthermore we can easily add nodes for performances.
- The bigger compliance here is **consistency**.
- When we can write on different nodes, we increase the probability to have inconsistency on writes.
- However there is a way to deal with this problem.

Combining Sharing with Replication

- Master-slave and sharding: we have multiple masters, but each data has a single master.
  - Depending on the configuration we can decide the master for each group of data.
- Peer-to-Peer and sharding is a common strategy for column-family databases.
  - This is commonly composed using replication of the shards