Cloud-Based Data Processing

Distributed Data – Part 1

Jana Giceva

Distributed Data

- **Various reasons why we need to distribute a database/data across multiple machines:**
  - **Scalability**
    - If the data volume, read or write load grows bigger than what a single machine can handle, you can spread it across different machines.
  - **Fault tolerance / High availability**
    - If the application needs to work even if one machine (or the network or an entire DC) goes down, you can use redundancy. When one fails, another can take over.
  - **Latency**
    - If you have users across the world, you may want to have the data in a DC that is geographically close to the users. That can significantly reduce the response time.

Replication vs. Partitioning

- There are two common ways data is distributed across multiple nodes.

- **Replication**
  - Keeps a copy of the same data on different nodes (potentially different locations).
  - Provides redundancy – If some nodes are unavailable, others can continue serving requests.
  - Reduces latency especially for high load or wide distribution of users across the globe.

- **Partitioning**
  - Split the big dataset into smaller subsets called *partitions.*
  - Each partition placed on a separate node.
  - Reduces latency for analytical jobs
  - Can improve availability

- One can combine both replication and partitioning!

Replication

Replication

- **Replication – keeping a copy of the same data on multiple machines** that are connected via network.

- **Benefits** of replication:
  - Keep data geographically close to users – **reduce the access latency**
  - Ensure the system continues working even in case of failures – **increase availability**
  - Scale out the number of machines used that can serve read queries – **increase read throughput**

- **For read-only workload, data replication is easy and always beneficial.**

- The challenge **is how to handle data that changes in a replicated system**:
  - Should there be a **leader replica** and if yes, how many?
  - Should one use a **synchronous** or **asynchronous propagation** of the **updates** among the replicas?
  - How to **handle a failed replica** if it is the follower? What if the leader failed? How does a resurrected replica catch up with the leader?

- Replication is an old topic, that was extensively studied in the 1970s, but has been popularized recently.

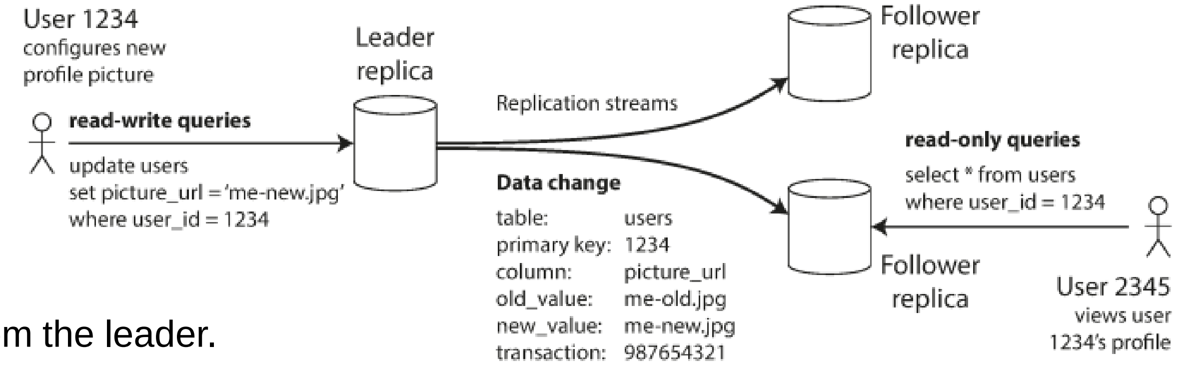
Leaders and Followers

- Each node that stores a copy of the dataset is called a **replica.**

- Every write needs to be processed by every replica; otherwise, the nodes will not hold the same data.
- The most common solution is **leader-based replication.**

- **Leader** – when clients write to the database, they must send their request to the leader.

- Other replicas are known as **followers,** which are updated by applying the **replication log** from the leader.

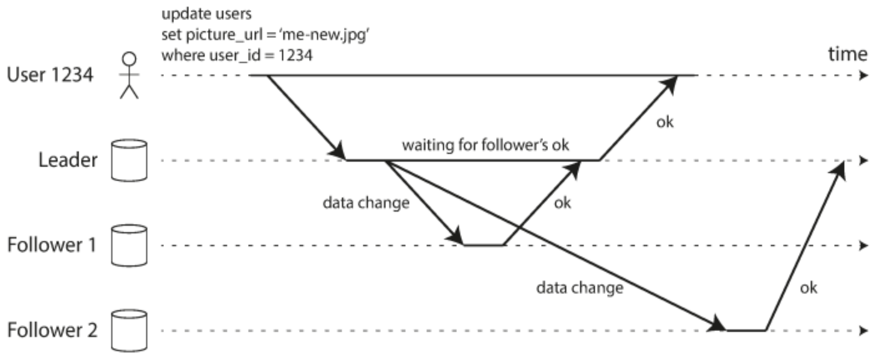


- A client can **read from anywhere** (the leader or any follower).

PostgreSQL (since v9.0), MySQL, Oracle Data Guard, SQL Server's AlwaysOn Availability Groups, MongoDB, RethinkDB, Espresso, Kafka, RabbitMQ, some networked FS and replicated block devices.

Synchronous vs. asynchronous Replication

- **Synchronous** if the leader waits until the follower(s) have confirmed that they applied the write before reporting success to the user.
  – e.g., the replication to follower 1 is synchronous.
- **Asynchronous** if the leader sends the update message to its follower(s), but does not wait for a response before answering success to the user.
  – e.g., the replication to follower 1 is asynchronous.



- Advantages of synchronous is that the follower is guaranteed to have an up-to-date version of the data.
- But, if a synchronous follower does not respond – the system will not be able to support writes.

- Fully-asynchronous replication trades availability at the cost of weakened durability

Setting up new Followers

- How to ensure that the **new follower** has an **accurate copy** of the leader's data **without downtime**?
  - Simply copying data files from one node to another is not sufficient as clients are constantly writing.

- The process needs a few steps:

1. **Take** a **consistent snapshot** of the leader's database (the same one used for back-up).
2. **Copy** the **snapshot to** the **follower** node.
3. The follower **gets** the **leader's replication log** since the snapshot.
4. Once the follower has processed the backlog, we say it has **caught up.**

Handling node outages/failures

- Any node in the system can go down.

- Goal is **high availability with leader-based replication**
  - i.e., how to reboot individual nodes without downtime.

- **Follower** failure → **catch-up recovery**
  - Keep a log of the data changes already applied on a local disk
  - After a reboot, apply the outstanding changes before re-connecting to the leader

- **Leader** failure → **failover**
  - Detect that the leader has failed.
  - Promote one of the followers as a new leader.
  - Reconfigure the system to use the new leader

Implementation of Replication Logs

- **Statement based replication**
  - The leader logs every write request (statement) that it executes

- **Write-ahead log (WAL) based replication – physical log**
  - Use WAL to build and maintain the followers.
  - But, the log is very low level and replication is coupled to the storage engine

- **Change data capture (CDC) based replication – logical log**
  - Sequence of records that describe the write to database tables at the granularity of rows.
  - Replicas can run on different versions or storage engines.
  - Easier to parse for external applications.

- **Trigger-based replication** (done in application layer)

Problems with Replication Lag

- In **Leader-based replication** all writes go to the leader, but read-only queries can go to any replica.
- This makes it **attractive** also **for scalability** and **latency**, in addition to fault-tolerance.

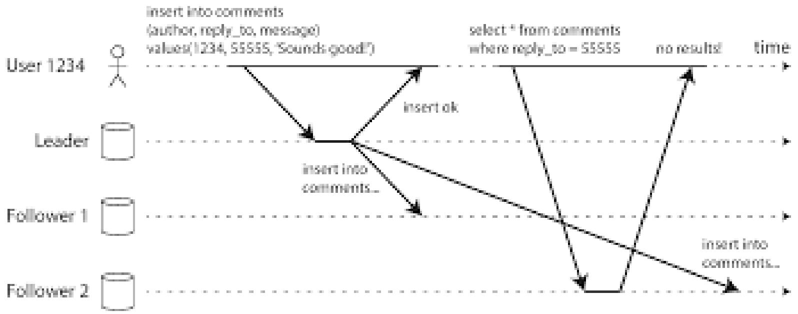
- **For read-mainly workloads: have many followers and distribute the reads across those followers.**
  - Removes load from the leader, allows read requests to be served by nearby replicas.
  - But, **only realistic for asynchronous replication** otherwise the system will not be available.

- If an application reads from an asynchronous follower, it may see outdated information.
  - Running the same query on the leader and a follower at the same time may get inconsistent results.
  - The effect is known as **eventual consistency.**

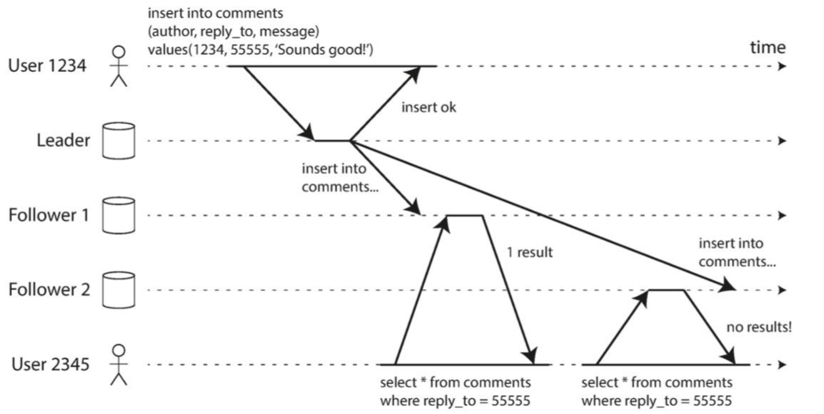
- The term **eventually** is deliberately vague – there is no limit how far a replica can fall behind.

Example problems with eventual consistency

- **Reading you own writes**
  - Requires read-after-write consistency
  - Makes no promises to other users
  - e.g., (always or for some time after a write) read from the leader, read based on timestamp
  - Cross-device read-after-write consistency



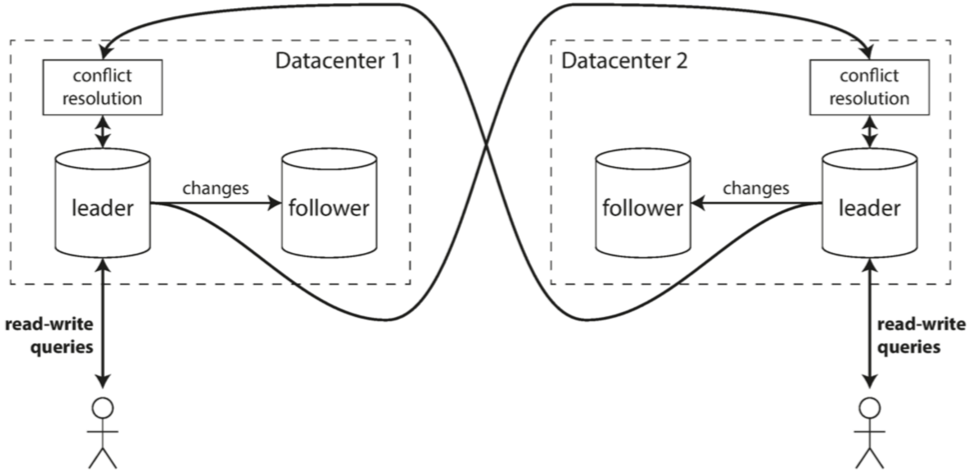
- **Monotonic reads**
  - Avoid a user to see things as moving back in time.
  - If one user makes several reads in sequence, they will not read older data after previously reading newer data.
  - e.g., by enforcing that each user always makes their reads from the same replica.



Multi-leader Replication

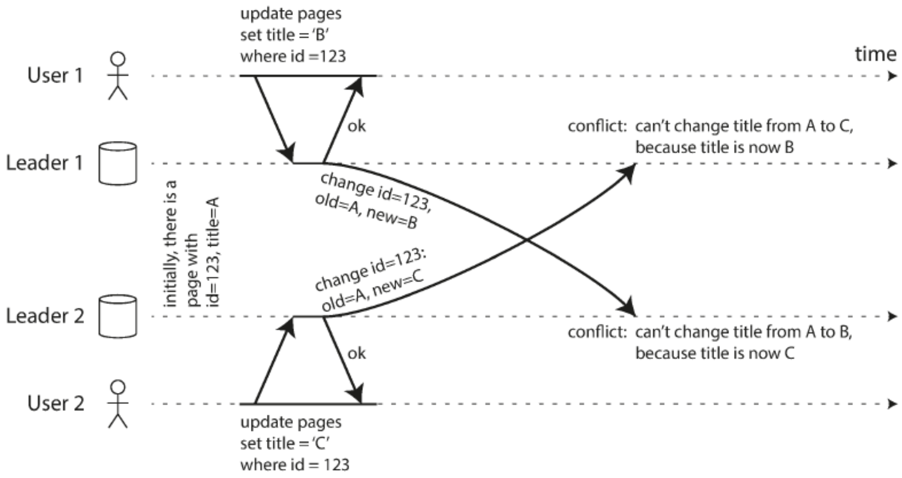
- **Leader-based replication has a single bottleneck – the leader.**
  - All writes must go through it. If there is a network interruption between the user and the leader, then no writes are allowed.

- **Alternative** approach **is multi-leader based replication.**
  - Multi-datacenter operation
    - advantages to single-leader replication for performance, tolerance to DC and network outages.
  - Clients with offline operations
    - Every device has a local database that acts as a leader.
  - Collaborative editing



Conflict resolution

- A problem with multi-leader replication is that **write conflicts** can **occur.**



- **Handling write conflicts:**
  - **Synchronous** vs **asynchronous**
    make the conflict detection synchronous.
  - **Conflict avoidance:** make all writes for a particular record go through the same leader
  - **Converging to a consistent state:**
    - last writer wins (LWW): each write has a unique ID, pick the write with the highest ID as the winner and throw away all other writes → prone to data loss.
    - let the application decide on the custom conflict resolution logic (on read or write).

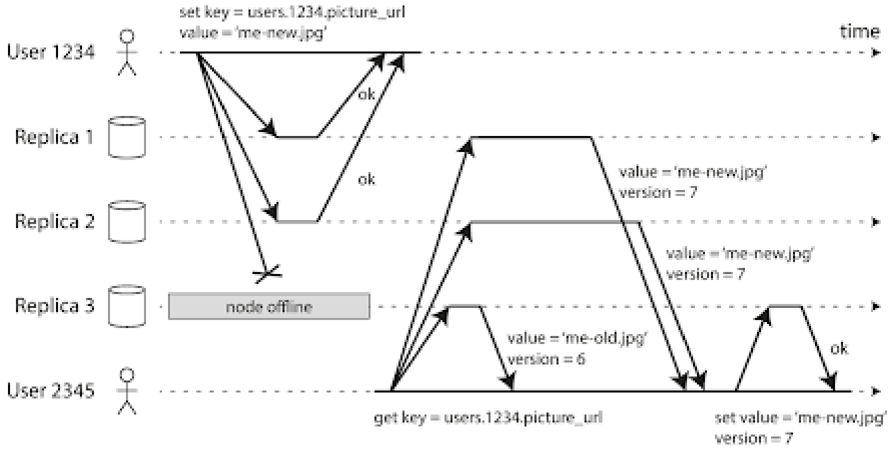
Leaderless Replication

- Abandon the concept of a leader, and **allow any replica to directly accept writes from clients**.

- Some of the earliest replicated data systems were leaderless (from the 1970s), but the idea was **resurrected by Amazon's Dynamo**.
  – Riak, Cassandra, and Voldemort are open source datastores with leaderless replication models, inspired by Dynamo. Also known as Dynamo-style replication.

- In a leaderless replication, there is no failover when a node fails.

- **A client both writes to and reads from multiple nodes in the system.**
  – **Version numbers** are used to determine which value is newer in case of different read values.

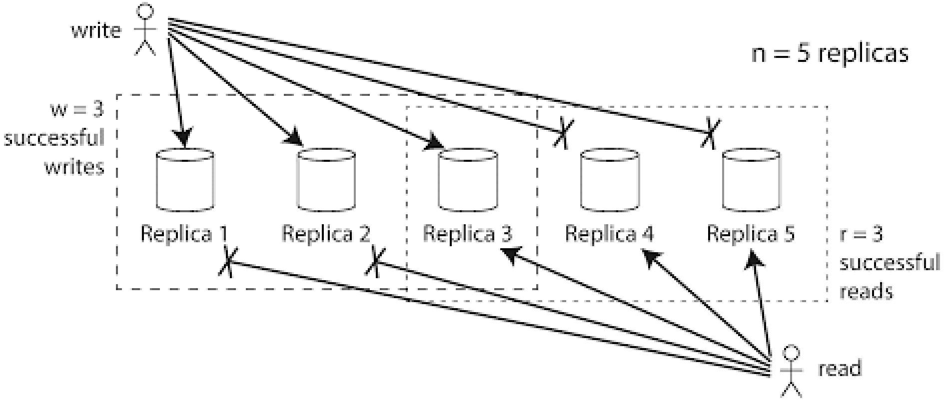


How does a node catch up the writes it missed

- **The replication system should ensure that eventually all the data is copied to every replica.**
- After an unavailable node comes back online, **how does it catch up on the writes it missed?**
- Two mechanisms are often used:
  - **Read repair**: the client can detect a stale response, and can write a newer value back to the replica.
    - Works well for values that are frequently read.
  - **Anti-entropy process**: a background process that checks for inconsistencies and fixes them.
    - Unlike the replication log in the leader-based replication mechanisms, here the order is not preserved.

Quorums for reading and writing

- **If there are n replicas, every write must be confirmed by w nodes to be considered successful, and we must query at least r nodes for each read.**



- As long as $w + r > n$, we expect to get an up-to-date value when reading.
- Reads and writes that obey these r and w values are called **quorum** reads and writes.
- The quorum conditions, allows the system to tolerate unavailable nodes as follows:
  - If w<n, we can still process writes if a node is unavailable.
  - If r<n, we can still process reads, if a node is unavailable.
  - With (3,2,2) we can tolerate 1 node failure, with (5,3,3) we can tolerate 2 nodes.
  - Normally, reads and writes are sent to all n replicas in parallel; w and r determine how many nodes we wait for before we consider the read or write to be successful.

Limitations of Quorum Consistency

- **Even with w+r > n, there are likely to be edge-cases where stale values are returned.**
  - e.g., with sloppy quorum; if two writes occur simultaneously; if a write happens at the same time as a read; if a write succeeded on some replicas but failed on others and overall succeeded on less than w nodes; if a node carrying a new value fails, and its data is restored from a replica carrying an old value.

- **Monitoring staleness and quantifying "eventual".**
  - There is no fixed order in which writes are applied – making monitoring of data staleness difficult.
  - It would be good to include staleness measurement in the standard set of metrics to quantify "eventual".

Sloppy quorum and hinted handoff

- Quorums are not as fault-tolerant as they could be.
  - A network interruption can cut off a client from a large number of database nodes.

- A **sloppy quorum**: used in case of network partitioning. The writes and reads still require w and r successful responses, but those may be by nodes that are not the designated “home” nodes for a value.
  - Once the network interruption is fixed, any writes that one node temporarily accepted on behalf of another node are sent to the appropriate “home” nodes. This is called the **hinted handoff**.

- Particularly good to increase write availability

- Note that it is not a quorum in the traditional sense, it is only an assurance of durability.

Summary

- **Replication is used for high availability, disconnected operation, latency and scalability.**
- **Three main approaches to replication**:
  - Single-leader replication
  - Multi-leader replication
  - Leaderless replication
- Replication can be **synchronous** or **asynchronous**. Follower replicas apply the **replication log.**
- Different ways to keep replicas in sync, or recover when a replica fails, etc.
- Replication **lag** can **lead** to **eventual consistency**. Some other consistency models that may be helpful:
  - Read-after-write consistency
  - Monotonic reads
  - Consistent prefix reads

Partitioning

Partitioning

- For very **large datasets**, or very **high throughput**, we need to break the data up into **partitions.**

- Clarifying terminology:
  - What we call a **partition** here is called a *shard* in MongoDB, Elasticsearch, and SolrCloud; *region* in Hbase, a *tablet* in BigTable, a *vnode* in Cassandra and Riak, and a *vBucket* in Couchbase.

- Partitions are defined in such a way that a piece of data belongs to **exactly one partition.**

- Partitioning is important for achieving better **scalability**, but it can also
  - Reduce **contention**
  - Improve **performance**
  - **Optimize** storage **costs**
  - Improve **security**

Why partition data?

- **Improve scalability**
  - Different partitions can be placed on different nodes in a shared nothing cluster
- **Improve performance**
  - Data operations on each partition work on **smaller** data **volume**
  - Operations that affect more than one partition can run in **parallel**
- **Improve security**
  - Can separate sensitive and non-sensitive data into different partitions and apply different security controls to the sensitive data
- **Improve availability**
  - Avoid a single point of failure. If one partition becomes unavailable, the others are still intact.
- **Allows better customization**

Designing partitions

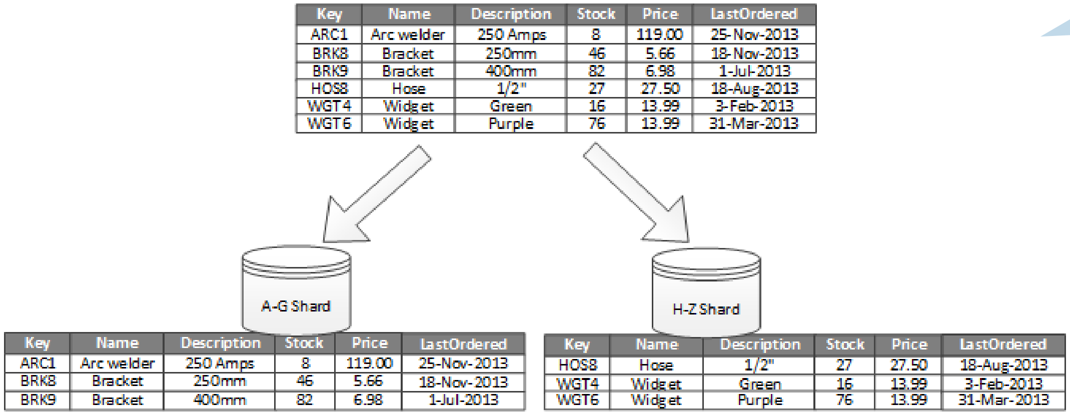
- **Horizontal partition (sharding):**
  - Each partition is a separate data store, but all partitions have the same schema
  - Each partition is known as a *shard* and holds a specific subset of the data
    - e.g., all the orders for a specific set of customers

- **Vertical partitioning:**
  - Each partition holds a *subset of the fields* for items in the data store
    - e.g., frequently accessed fields, may be placed in one vertical partition and less frequently accessed fields in another.

- **Functional partitioning:**
  - Data is aggregated according to how it is used by each bounded context in the system
    - e.g., An e-commerce might store *invoice data* in one partition and *product inventory data* in another

Horizontal partitioning (sharding)

- Example horizontal partitioning or sharding



Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

A-G Shard

Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013

H-Z Shard

Key	Name	Description	Stock	Price	LastOrdered
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Product inventory data is divided into shards based on the product key.

Each shard holds the data for a cont. range of shard keys (A-G and H-Z)

Spread the load over more nodes, to reduce contention and response time.

- The most important factor is the **choice of sharding key.**
  - **Goal** is to **spread** the **data** and query **load evenly** across the nodes.
  - If the partitioning is unfair, some partitions will have more data or queries, we call it **skewed.**
  - A partition with disproportionally high load is called **a hot spot.**

Horizontal Partitioning strategies

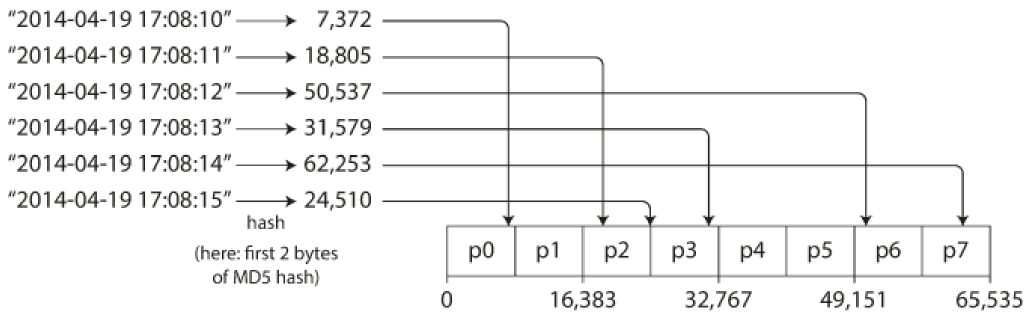
- **by Key Range**
  - Assign a **continuous range of keys** to each **partition**.
  - The range of keys are not necessarily evenly spaced, because your data may not be evenly distributed.
    BigTable, Hbase, RethinkDB, and MongoDB before v2.4

- **Advantage:**
  - Within each partition we can keep the keys in sorted order → range scans are fast and easy
  - Can fetch several related records in one query

- **Disadvantage:**
  - Certain access patterns can lead to hot spots

Horizontal Partitioning strategies II

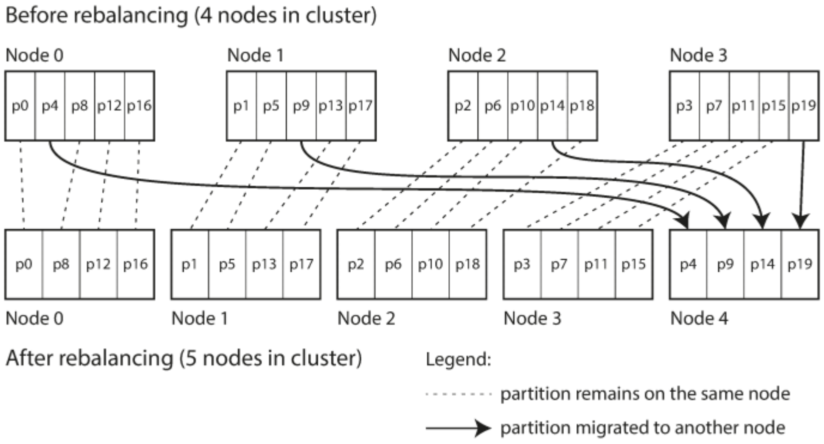
- **by Hash of Key**
  - **hash a key** to determine the partition
  - a **partition** for a **range of hashes**
  - if a key's hash value belongs to a partition's range then the key is placed in that partition.



- **Advantage:**
  - No problem with skew and hot spots (overstatement, we may still have issues, but they are rare)

- **Disadvantage:**
  - No longer easy to do efficient range queries.
  - e.g., range queries on the primary key are not supported by Riak, Couchbase or Voldemort.

Rebalancing partitions

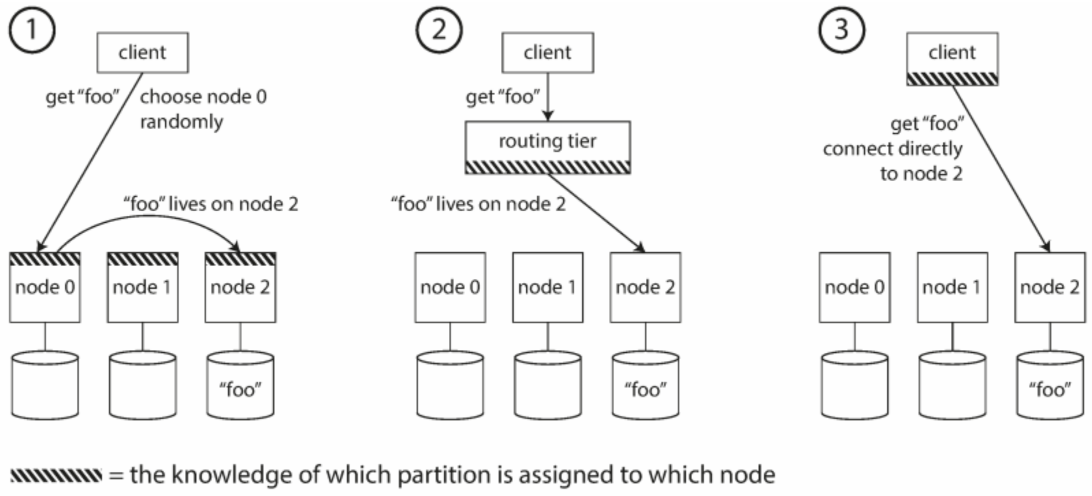


- **Rebalancing is often necessary**

- **Strategies of rebalancing:**
- **How not to do it? Hash mod N.**
  - If the number of nodes N changes, most of the keys will need to be moved from one node to another.
- **Fix the number of partitions P so that P >> N**
  - If a node is removed/added to the cluster, only a few (entire) partitions need to be moved.
  - The number of partitions remains the same, and the assignment of keys to partitions is not changed.
- **Dynamic partitioning**
  - Applicable with range and hash partitioning
  - When a partition grows to exceed a size, split it into two (like in a B-tree).
- **Partitioning proportional to nodes**
  - Have a fixed number of partitions per node.

Request routing

- **Open question: when a client wants to make a request, how does it know which node to ask?**
  - As partitions are rebalanced, the assignment of partitions to nodes changes
  - Someone needs to have the top-level overview.



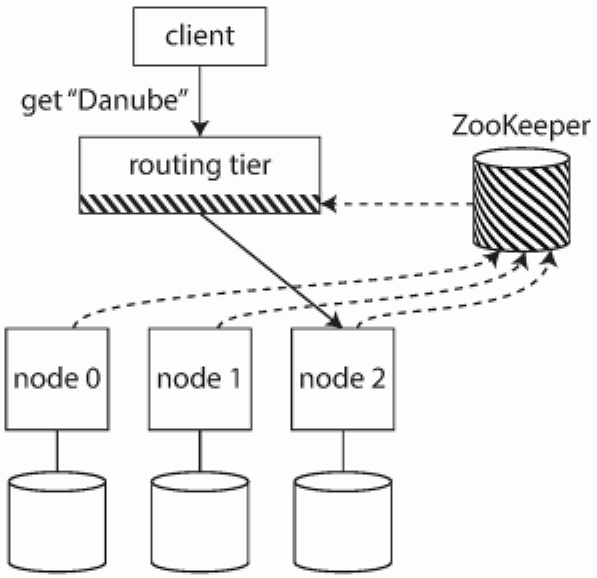
- **Three main options:**
  - The node layer
  - The routing tier (or third party)
  - The clients

  - It is a challenging problem as all participants need to agree → requires reaching a consensus.

- Many systems rely on a **coordination service** such as Zookeeper to keep track of cluster meta data.
- Others use alternatives like **gossip protocol** among the nodes to disseminate cluster state changes.

Example using ZooKeeper to keep track

- The routing tier can subscribe to this information from the ZooKeeper service



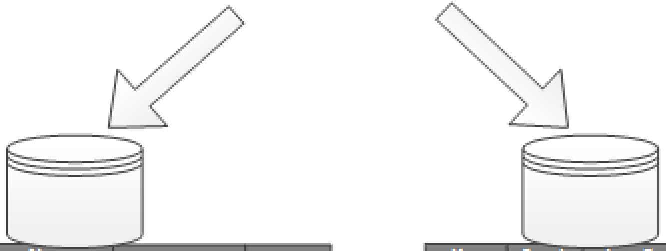
Key range	Partition	Node	IP address
A-ak — Bayes	partition 0	node 0	10.20.30.100
Bayeu — Ceanothus	partition 1	node 1	10.20.30.101
Ceara — Deluc	partition 2	node 2	10.20.30.102
Delusion — Frenssen	partition 3	node 0	10.20.30.100
Freon — Holderlin	partition 4	node 1	10.20.30.101
Holderness — Krasnoje	partition 5	node 2	10.20.30.102
Krasnokamsk — Menadra	partition 6	node 0	10.20.30.100
Menage — Ottawa	partition 7	node 1	10.20.30.101
Otter — Rethimnon	partition 8	node 2	10.20.30.102
Reti — Solovets	partition 9	node 0	10.20.30.100
Solovyov — Truck	partition 10	node 1	10.20.30.101
Trudeau — Zywiec	partition 11	node 2	10.20.30.102

= the knowledge of which partition is assigned to which node

Vertical partitioning

- Goal to **reduce the I/O and performance costs** when fetching items that are frequently accessed.

Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013



Key	Name	Description	Price
ARC1	Arc welder	250 Amps	119.00
BRK8	Bracket	250mm	5.66
BRK9	Bracket	400mm	6.98
HOS8	Hose	1/2"	27.50
WGT4	Widget	Green	13.99
WGT6	Widget	Purple	13.99

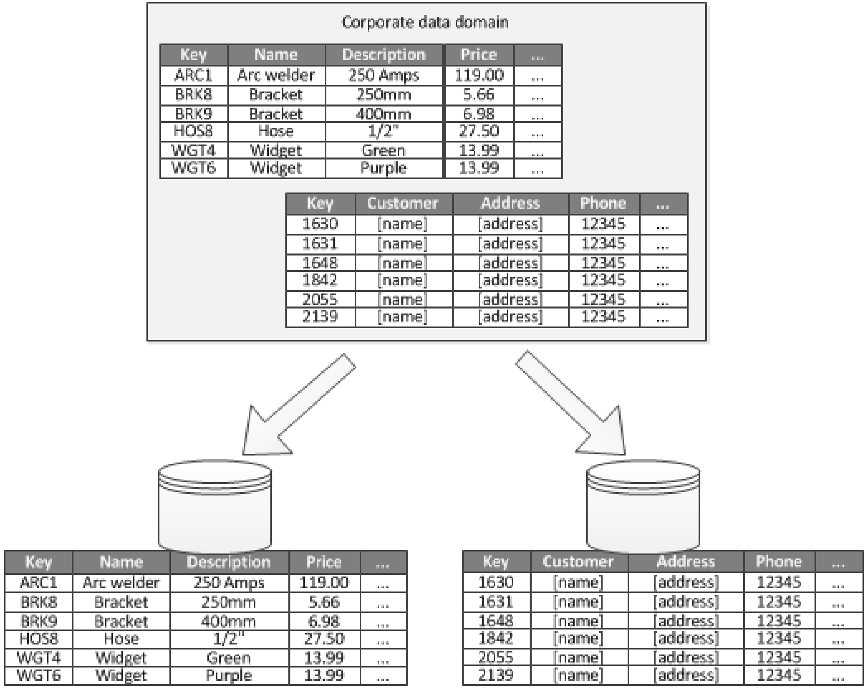
Key	Stock	LastOrdered
ARC1	8	25-Nov-2013
BRK8	46	18-Nov-2013
BRK9	82	1-Jul-2013
HOS8	27	18-Aug-2013
WGT4	16	3-Feb-2013
WGT6	76	31-Mar-2013

- Different properties of an item are stored in different partitions.
  - One partition holds data that is accessed more frequently: product name, description and price
  - Another holds inventory data: the stock count and the last ordered date.
- Application regularly gets the product name, desc. and price when displaying the product details.
- Stock count and last ordered data are commonly used together and are more frequently modified.

Vertical partitioning cont.

- **Other advantages:**
  - Relatively slow moving data can be separated from the more dynamic data
    - Slow moving data is a good candidate for an application to cache in memory
  - Sensitive data can be stored in a separate partition with additional security control.
- Ideally **suited** for **column-oriented data stores.**

Functional partitioning



Key	Name	Description	Price	...
ARC1	Arc welder	250 Amps	119.00	...
BRK8	Bracket	250mm	5.66	...
BRK9	Bracket	400mm	6.98	...
HOS8	Hose	1/2"	27.50	...
WGT4	Widget	Green	13.99	...
WGT6	Widget	Purple	13.99	...

Key	Customer	Address	Phone	...
1630	[name]	[address]	12345	...
1631	[name]	[address]	12345	...
1648	[name]	[address]	12345	...
1842	[name]	[address]	12345	...
2055	[name]	[address]	12345	...
2139	[name]	[address]	12345	...

- When possible to identify a **bounded context**, **functional partitioning** is a way to **improve isolation** and **data access performance**.
- Another common use is to separate read-write data from read-only data
- This strategy can help reduce data access contention across different parts of the system

Partitioning for scalability

- **Analyze the application to understand the data access patterns**:
  - Result set returned by each query
  - The frequency of access
  - The inherent latency
  - The server-side compute processing requirements.
- **Determine the current and future scalability targets, such as data size and workload**
  - Distribute the data across the partitions to meet the scalability target, choose the right shard key.
  - Make sure each node has enough resources to handle the requirements in terms of storage space, processing power or network bandwidth.
- **Monitor to verify that the data is distributed well and that the partitions can handle the load**
  - Actual usage does not always match what an analysis predicts
  - It may be required to rebalance the partitions

Partitioning for improved query performance

- Query performance can be boosted by using smaller data sets and by running parallel queries.
- Each partition should contain a small proportion of the entire data set.

- Follow these steps to improve the overall query performance of your system/application.

- Examine the application requirements and performance.
  - Identify the critical queries that must always perform quickly.
  - Monitor the system to detect any queries that perform slowly.
  - Find which queries are performed most frequently.

- Partition the data that causes slow performance.

- Consider running queries in parallel across partitions to improve response time.

Partitioning for better availability

- **Avoid** having the entire dataset does not constitute **a single point of failure.**

- **Consider the following factors that affect availability:**
  - **Identify critical data**
    - Consider storing critical data in highly available partitions with an appropriate back-up plan
    - Establish separate management and monitoring procedures for the different datasets
    - Place data that has the same level of criticality in the same partition
  - **Decide how to manage individual partitions**
    - If a partition fails, it can be recovered independently
    - Partition data by geographical area allows scheduled maintenance at off-peak hours
  - **Replicate critical data across partitions.**
    - This strategy can improve availability and performance, but
      can also introduce consistency issues related to replication lag.

We did not cover…

- **How to partition a secondary index**
  - **Document-partitioned index (local indexes)**, where the secondary index are stored in the same partition as the primary key and value.
    - Only a single partition needs to be updated on write, but a read requires scatter/gather across all.
  - **Term-partitioned index (global indexes)**, where the secondary indexes are partitioned separately, using the indexed values.
    - When a document is written, several partitions of the secondary index need to be updates; however a read can be served from a single partition.

- **Creating materialized views that summarize data to support fast query operations**.
  - Useful in a partitioned data store if the partitions that contain the data being summarized are distributed across multiple sites.

- **Parallel Query Execution in presence of partitions**
- **Distributed Transactions**

Summary

- **Partitioning is necessary when data and load volume exceeds a single machine's capacity.**
- The goal is to **spread the data and query load evenly across multiple machines**, avoiding hotspots.
- Need to be careful when choosing the partitioning scheme so that it is appropriate to the data and workload properties, and rebalance it when nodes are added/removed.
- **Three main types of partitioning**:
  - horizontal,
  - vertical and
  - functional.
- **Two main approaches for horizontal partitioning: key range and hash-based.**
- **Various techniques for rebalancing and routing.**

References

The material covered in this class is mainly based on:

- The book *"Designing Data-Intensive Applications – The Big Ideas Behind Reliable, Scalable, and Maintainable Systems"* by Martin Kleppmann (Chapters 5 and 6) (link)

Some information and images were based on material from:

- Microsoft's Azure Application Architecture Guide
  - Best practices for horizontal, vertical and functional data partitioning (link)
  - Data partitioning strategies in various Azure services (link)
  - Sharding pattern (link)