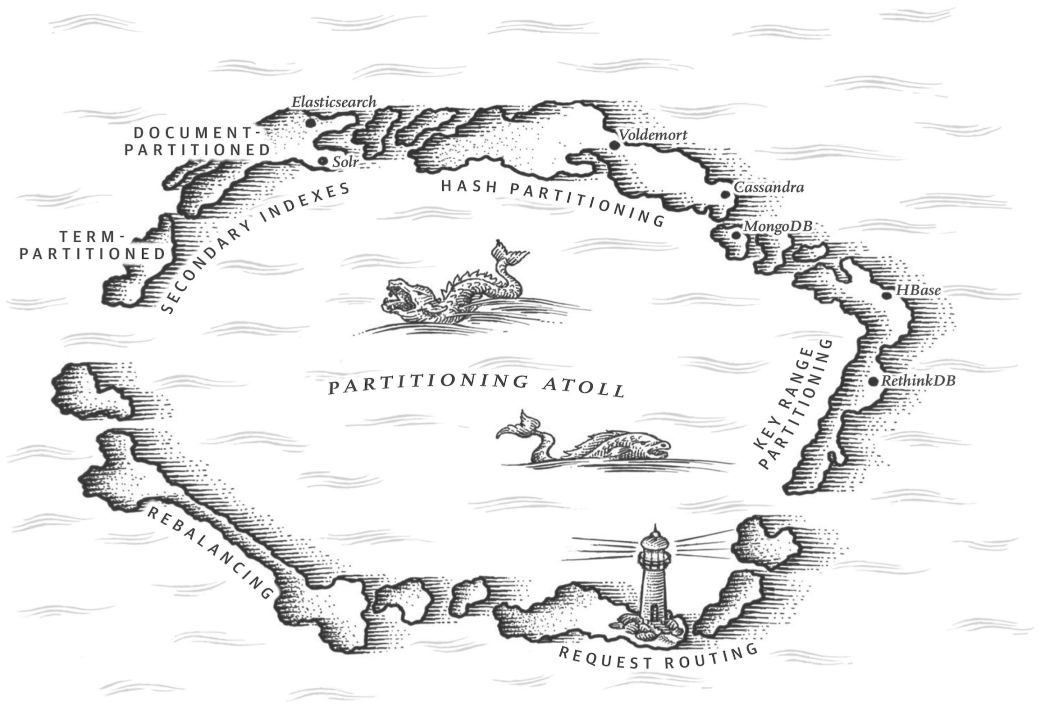
Designing Data-Intensive Applications

Chapter 6: Partitioning



Introduction

- Partitioning is breaking up the data so that each piece of data is part of one partition, and the partitions are spread across multiple servers
- The main motivation for partitioning is scalability
- Partitioning can be used with replication, and the partitioning strategy is largely independent of the replication methodology, so the two mostly can be considered separately

Partitioning of Key-Value Data

- Goal is to spread the data evenly across the partitions
- Partitioning by key range
  - Assigning continuous ranges of keys to partitions
  - The ranges won't be evenly spaced if the data isn't evenly distributed
  - Boundaries can be manual or chosen automatically
    - Automatic used with automated rebalancing
  - Data can be stored in sorted order, making range queries efficient
  - Beware that range partitioning can easily lead to hot spots
- Partitioning by hash of key
  - Hash functions make skewed data uniformly distributed, but they don't need to be cryptographically strong
  - Assign continuous ranges of hash values to partitions

    - Range queries are now difficult, and not supported on some products/queries
        - Cassandra allows compound primary keys, and while range queries can't be performed on the first column, they can be performed on the second column (and any additional columns)

Skewed Workloads and Relieving Hot Spots

- Hashing will generally even out requests across keys, but that doesn't guarantee no hot spots
- If you have an extreme number of requests to the same key, you still have a hot spot
- Most systems cannot automatically solve this type of situation
- Augmenting the key with a random value will force distribution across hashes, but this would add too much overhead to do it for all keys
    - It would still be up to the application to special-case high volume keys

Partitioning and Secondary Indexes

- Reminder that secondary indexes are usually not unique
- Some key-value stores don't support secondary indexes, but they are becoming more common
- Partitioning secondary indexes by document
    - This is a *local index*
    - Each partition maintains its own secondary indexes for keys on that partition
    - Writes are easy because all secondary indexes that need updating are on the same partition as the the key/document
    - Reads, however, must be sent to every partition, called *scatter/gather*
        - Because you must synchronously wait for every partition to respond, scatter/gather is prone to tail latency amplification
- Partitioning secondary indexes by term
    - This is a *global index*, but it still needs to be spread out to avoid hot spots
    - Typically term-partitioned secondary indexes use ranges of values (terms), not hashed values
    - Reads using a secondary index access usually only require one/few partitions
        - Note: if a secondary index returns many documents, they are likely to be spread across many partitions. Even if the complexity is hidden from the client, this still impacts performance.
    - Writes, however, are more complex because other partitions may need to be updated for each secondary index that exists
        - For this reason, updates to global secondary indexes are often asynchronous

Rebalancing Partitions

- Adding hardware or a node failure requires shifting requests from one node to another
- This process of moving workload between nodes is called rebalancing. Goals are:
    - While rebalancing, reads and writes still supported
    - Afterward, load is shared fairly between nodes

    - Try to minimize the amount of data moved during rebalancing

Strategies for Rebalancing

- How not to do it: hash mod N. This would cause massive data movement every time.
- Fixed number of partitions
    - Rather than one partition per node, create a number of partitions that is much larger than the number of nodes
    - All of the smaller partitions from one node can be redistributed among the others without impacting the data already on other nodes
    - The number of partitions and the assignment of keys to partitions are not affected by rebalancing
    - In theory, if you have uneven hardware, you can purposely give slower nodes fewer partitions so that response time is relatively consistent
    - Note that picking the right number of partitions can be hard if significant database growth is expected (multiple orders of magnitude)
- Dynamic partitioning
    - If a database with key range partitioning had fixed partitions, that could easily lead to uneven distribution of data, and manual changes would be a real pain
    - Dynamic partitioning splits a partition when it exceeds a certain size, and merges a partition with another when it drops below another size threshold
    - The number of partitions grows with the size of the data
        - You may want to initialize an empty database with one partition per node
- Partitioning proportionally to nodes
    - Instead of a fixed number of partitions or a fixed size range of partitions, have a fixed number of partitions that is proportional to the number of nodes (i.e. a fixed number of partitions per node).
    - Since workload often grows with database size, the number of nodes often ends up increasing with database size. This means that, in practice, the size of partitions winds up remaining somewhat stable.
    - When a node is added, it picks a number of existing partitions to split, and takes half of each
        - Randomly choosing partitions to split could, though unlikely, result in unfair splits. Cassandra 3.0 has an algorithm to ensure balance.

Operations: Automatic or manual rebalancing

- Rebalancing can be fully automatic, fully manual, or in between (e.g. suggesting a partition reassignment, but requiring an admin to approve it)
- One risk of fully automatic is that rebalancing moves a lot of data, and system performance could suffer if rebalancing is started at the wrong time
- Another issue is potential interactions between automatic failure detection. If the original failure is related to heavy load, then kicking off rebalancing may make make the load issue worse

Request Routing

- Since nodes can be moved and rebalanced, how does a client know where to send each request? This is a case of the *service discovery* problem. Three high level designs:
    - Client sends to any node, and the node behind the scenes forwards the request to the proper partition when needed
    - Client sends to an intermediate routing tier, and that tier forwards the request
    - Clients keep track of partitioning and go directly to correct node
- In all cases, there will be nodes/tier/clients that need to keep track of partition changes
    - Consensus in discussed in chapter 9, and can be hard
    - Many systems use an external service, such as Zookeeper, to keep track of cluster/partition metadata
    - Cassandra and Riak use a *gossip protocol* to communicate changes. Since they have nodes forward requests behind the scenes, nodes can handle the edge case where they haven't learned of an update yet without bothering the client

Parallel Query Execution

- For analytic use cases, often want large amounts of data (from multiple keys). Most NoSQL implementations only support simple reads and maybe scatter/gather.
- Some relational products support *massively parallel processing* (MPP) where queries with complex joins, grouping, aggregation, and filtering can be coordinated by multiple nodes in parallel. Typically an optimizer coordinates all of the activity. Details will be discussed in chapter 10.

Summary

- Two main approaches to partitioning are by key range and hash partitioning
- Secondary indexes can be document partitioned or term partitioned
- Request routing is a technical hurdle that must be dealt with
- Note that an operation that writes information to multiple partitions will have very complex failure modes. More to come (chapters 8 & 9).