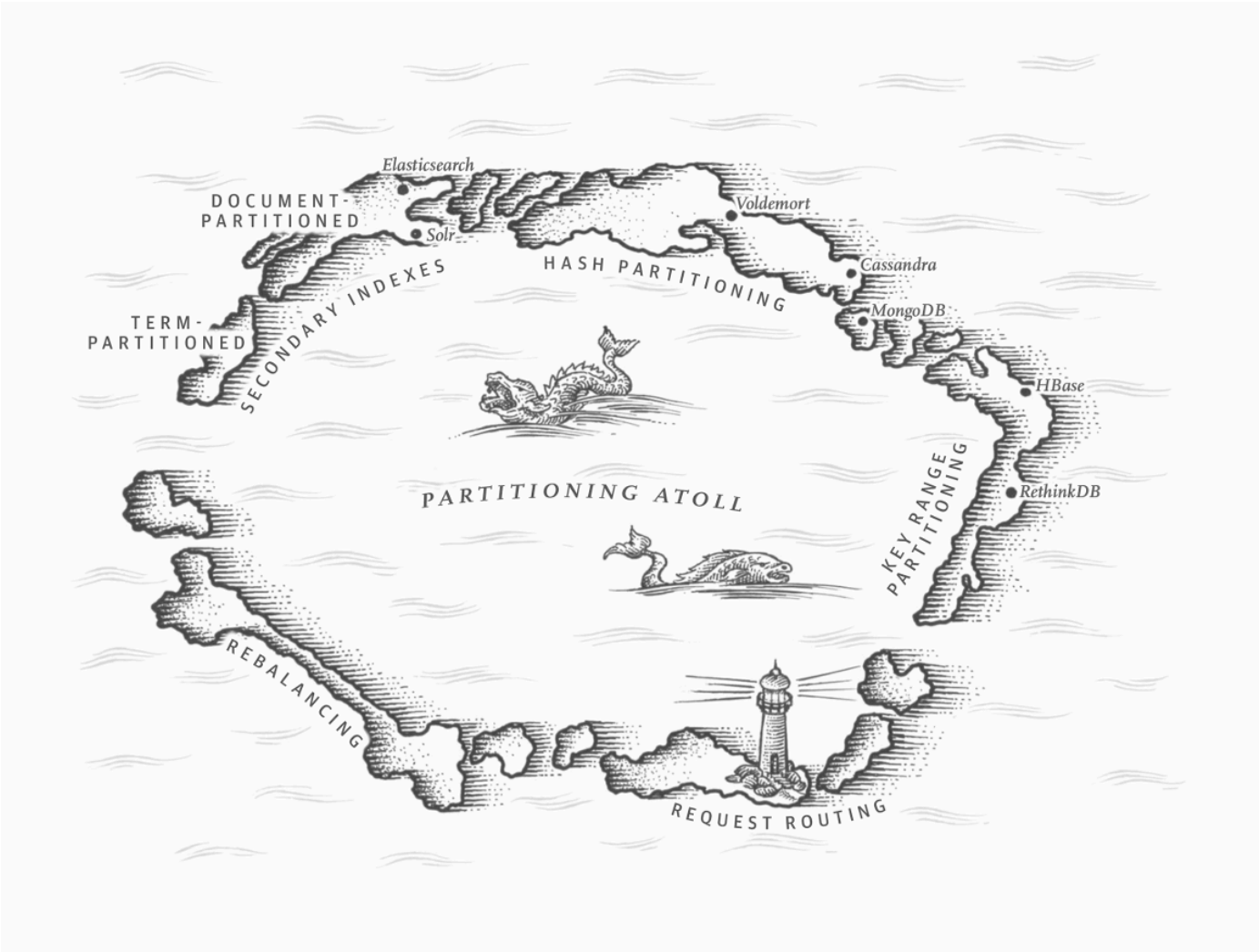


Chapter 6 Partitioning

Partition:
Shard - MongoDB, Elasticsearch, SolrCloud
Region - HBase
Tablet - Bigtable
Vnode - Cassandra, Riak
vBucket - Couchbase

Main Reason: Scalability

Partitioning and Replication

Combination of partitioning and replication in Leader-follower replication model:

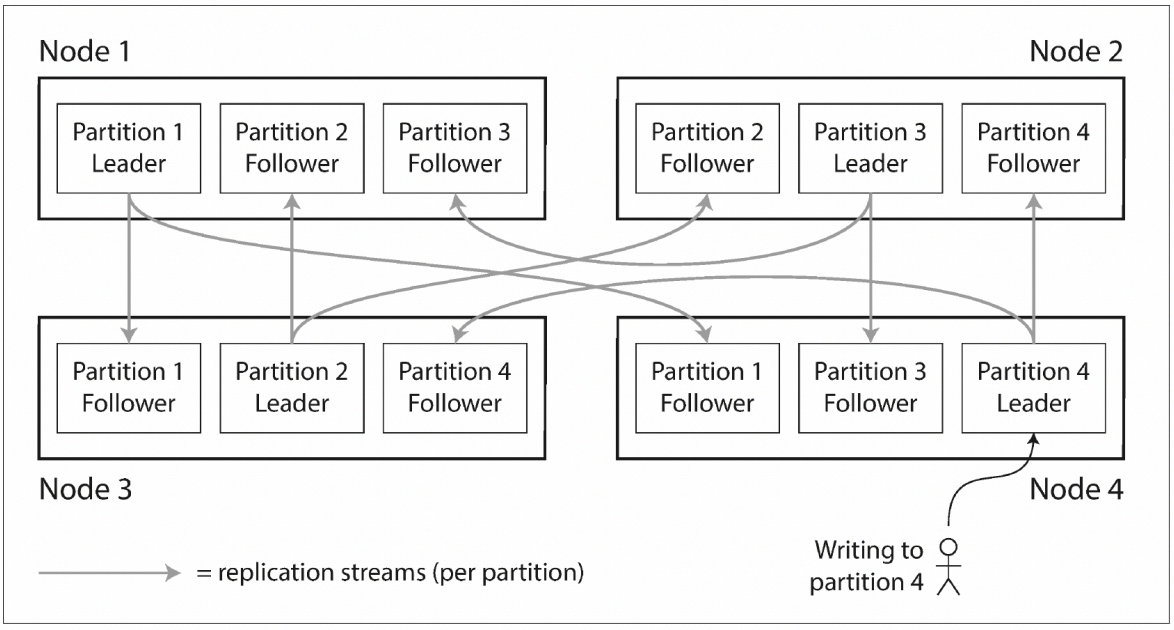


Figure 6-1. Combining replication and partitioning: each node acts as leader for some partitions and follower for other partitions.

Partitioning of Key-Value Data

Skew: Some partitions have more data or queries than others. This makes partitioning less effective.
Hot spot: A partition with disproportionately high load. To avoid hot spots, assign records to nodes randomly.

Partitioning by Key Range

- One way of partitioning is to assign a continuous range of keys (from some minimum to some maximum) to each partition, like the volumes of a paper encyclopedia.
- The ranges of keys are not necessarily evenly spaced, because your data may not be
- evenly distributed.
- Within each partition, we can keep keys in sorted order.
- The **downside** of key range partitioning: certain access patterns cause hot spots.
- To avoid hot spots. If you use timestamp as key, you can prefix timestamp with sensor name and then by time. Write load will be more evenly spread across the partitions.

When fetching values for multiple sensors within a time range, perform a range query for each sensor name.

Partition by Hash of Key

- To avoid skew risk and hot spots, use a hash function to determine the partition of a given key.
- One gets a suitable hash function for keys, assigning each partition a range of hashes(rather than a range of keys), every key whose hash falls within a partition's range stores in that partition.
- The partition boundaries can be evenly spaced, or they can be chosen pseudorandomly (consistent hashing).
- Consistent hashing is a way of evenly distributing load across an internet-wide system of caches such as CDN(content delivery network). It uses randomly chosen partition boundaries to avoid the need for central control or distributed consensus. Consistent describes a particular approach to rebalancing.
- **Disadvantage**: Use hash of key for partitioning, lose the ability for range queries.

Cassandra achieves a compromise between the two partitioning strategies. A table in Cassandra can be declared with a compound primary key consisting of several columns. Only the first part of that key is hashed to determine the partition, but the other columns are used as a concatenated index for sorting the data in Cassandra's SSTables. A query therefore cannot search for a range of values within the first column of a compound key, but if it specifies a fixed value for the first column, it can perform an efficient range scan over the other columns of the key.
The concatenated index approach enables an elegant data model for one-to-many relationships. For example, on a social media site, one user may post many updates. If the primary key for updates is chosen to be (user_id, update_timestamp), then you can efficiently retrieve all updates made by a particular user within some time interval, sorted by timestamp. Different users may be stored on different partitions, but within each user, the updates are stored ordered by timestamp on a single partition.

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 system 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

- A secondary index usually **doesn't** identify a record **uniquely** but rather is a way of searching for occurrences of a particular value. E.g., find all actions by a user.
- Secondary indexes are common in relational databases and document databases.
- They are the raison d'etre of search servers such as Solr and Elasticsearch.
- Problem: they don't map neatly to partitions.
- Two main approaches to partitioning a database with secondary indexes: document-based partitioning and term-based partitioning.

Partitioning Secondary Indexes by Document

- Use case: a website for selling used cars.
  Each listing has a unique ID(document ID), the database is partitioned by the document ID.
  Let users search for cars, filter by color and by make, need a secondary index on color and make(in a document database these are fields; in a relational database they are columns).
- If you have declared the index, the database can perform the indexing automatically. E.g., whenever a red car is added, the database partition automatically adds it to the list of document IDs for the index entry color:red.
- Document-partitioned index is **local index**(as opposed to a global index). Each partition maintains its own secondary indexes, covering only the documents in that partition.
- If you want to search for red cars, you need to send the query to all partitions, and combine all results. This approach to querying a partitioned database is **scatter/gather**, making **read queries on secondary indexes quite expensive**. Even worse if you use multiple secondary indexes in a single query(color and make at same time).
- MongoDB, Riak, **Cassandra**, Elasticsearch, SolrCloud, VoltDB

Partitioning Secondary Indexes by Term

- **Global index** or Term-partitioned index
- Construct a Global index that covers data in all partitions.
- Store that index on more than one node.
- A global index must also be **partitioned**, and can be partitioned differently from the primary key index.
- Call this kind of index term-partitioned. A term would be color:red. The name term comes from full-text indexes(a particular kind of secondary index), where the terms are all the words that occur in a document.
- Partitioning by term itself can be useful for range scans, whereas partitioning on a hash of the term gives a more even distribution of load.
- **Advantage**: reads fast.
- **Disadvantage**: writes are slower and more complicated
- In practice, updates to global secondary indexes are often **asynchronous**

- **Amazon DynamoDB**, Riak, Oracle data warehouse.

Rebalancing Partitions

Rebalancing: The process of moving data load from one node to another

Reasons for rebalancing:
- Query throughput increases
- Dataset size increases
- Machine fails

Minimum requirements for Rebalancing:
- After rebalancing, the load should be shared fairly between nodes
- While rebalancing, the database should continue accepting reads and writes
- No more data than necessary should be moved between nodes, to make it fast and to minimize the network and disk I/O.

Strategies for Rebalancing

Ways to assign partitions to nodes.

Hash mod N

- When partitioning by hash of a key, best to divide hashes into ranges and assign to a partition.
- Problem with mode N: if N changes, most of the keys need to be moved from one node to another.

Fixed number of partitions

- Create many more partitions than nodes, assign several partitions to each node.
- If a node is added to the cluster, the new node steals a few partitions from every existing node until partitions are fairly distributed. If a node is removed, in reverse.
- **Problem**: it takes time to transfer a large amount of data over network
- Riak, Elasticsearch, Couchbase, Voldemort
- Dataset growth might be an issue.
- If partitions are very large, rebalancing and recovery from node failures are expensive
- If partitions are too small,they incur too much overhead

Dynamic partitioning

- For key range partitioned databases, a fixed number of partitions with fixed boundaries are inconvenient. Thus key range-partitioned databases such as HBase and RethinkDB create partitions dynamically.

- When a partition grows to exceed a configured size (on HBase, the default is 10 GB), it is **split** into two partitions so that approximately half of the data ends up on each side of the split. Conversely, if lots of data is deleted and a partition shrinks below some threshold, it can be merged with an adjacent partition. This process is similar to what happens at the top level of a B-tree.
- Each partition is assigned to one node, and each node can handle multiple partitions, like in the case of a fixed number of partitions. After a large partition has been split, one of its two halves can be transferred to another node in order to balance the load. In the case of HBase, the transfer of partition files happens through HDFS, the underlying distributed filesystem.
- **Advantage** of dynamic partitioning is that **the number of partitions adapts to the total data volume.**
- Not only key range-partitioned data, but also hash-partitioned data. MongoDB supports both key-range and hash partitioning, and it splits partitions dynamically in either case.

Partitioning proportionally to nodes

- Make the number of partitions proportional to the number of nodes, have a **fixed number of partitions per node**.
- The size of each partition grows proportionally to the dataset size while the number of nodes remains unchanged, but when you increase the number of nodes, the partitions become smaller again. Since a larger data volume generally requires a larger number of nodes to store, **this approach also keeps the size of each partition fairly stable**.
- **Hash-based partition**(boundaries can be known by hash function), **Consistent Hashing**
- **Cassandra**, Ketama

Operations: Automatic or Manual Rebalancing

- There is a gradient between fully automatic rebalancing and fully manual.
- Fully automated rebalancing can be convenient. However, it can be unpredictable. It can be a good thing to have a human in the loop for rebalancing. It's slower than a fully automatic process, but it can help prevent operational surprises.

Request Routing

Service discovery: When a client wants to make a request, how does it know which node to connect to?

Approaches:

1. Allow clients to contact any node(e.g., via round-robin load balancer)
2. Send all requests from clients to a routing tier first, which acts as a partition-aware load balancer
3. Require that clients be aware of the partitioning and assignment of partitions to nodes.

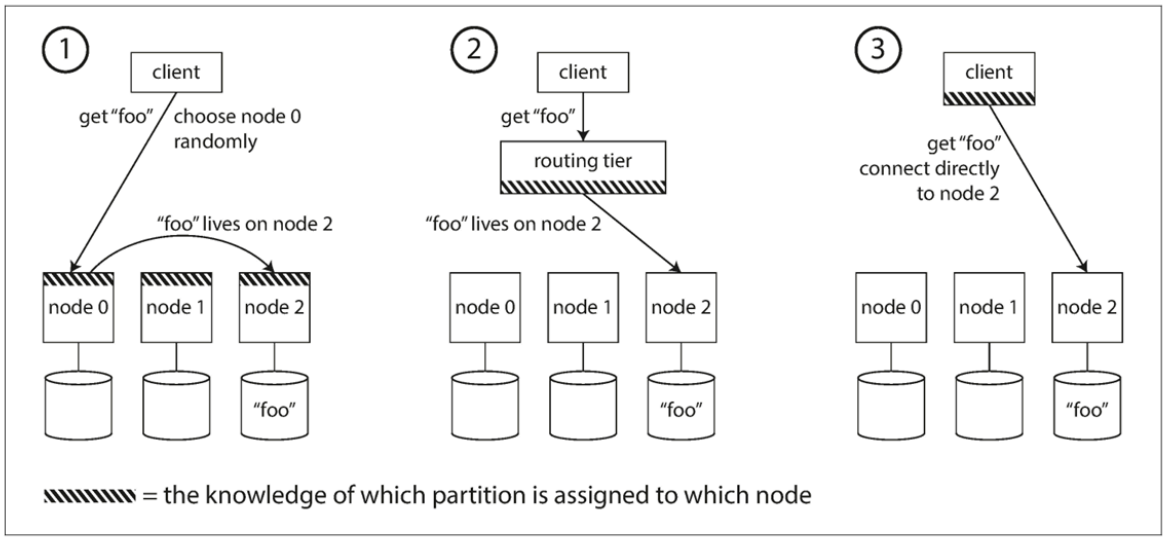
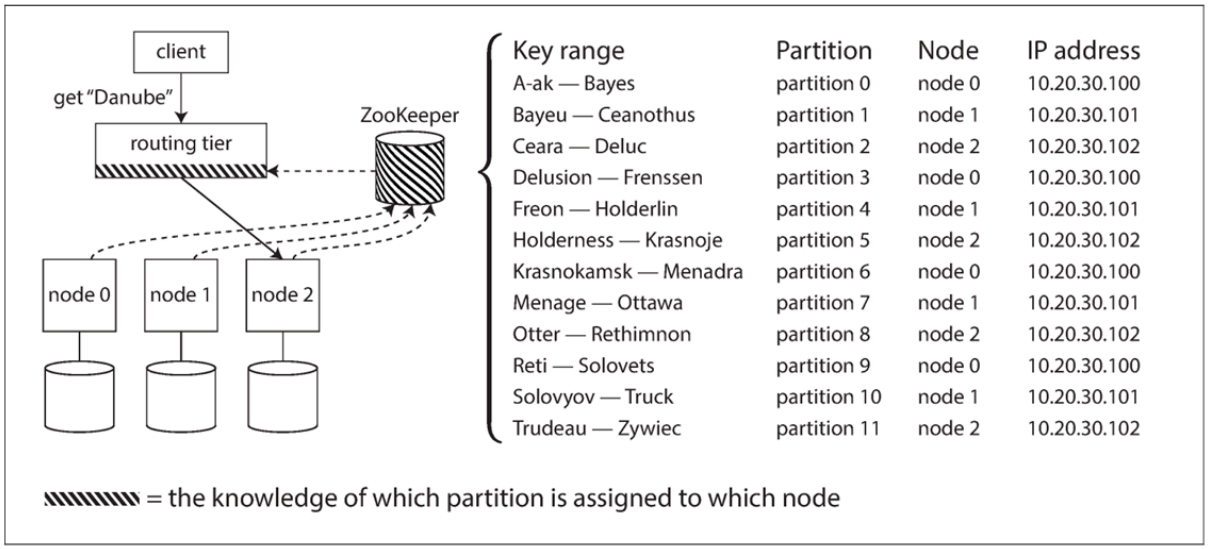


Figure 6-7. Three different ways of routing a request to the right node.

How does the component making the routing decision learn about changes in the assignment of partitions to nodes. There are protocols for achieving **consensus** in a distributed system, but they are hard to implement correctly.

- Many distributed data systems rely on a separate **coordination service** such as **ZooKeeper** to keep track of this cluster metadata.
- Each node registers itself in ZooKeeper.
- ZooKeeper maintains the mapping of partitions to nodes.
- **Routing tier** or the partitioning-aware client subscribes to this information in ZooKeeper. Whenever a partition changes ownership, or a node is added or removed, ZooKeeper notifies the routing tier to keep its routing information up to date.



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

Figure 6-8. Using ZooKeeper to keep track of assignment of partitions to nodes.

- LinkedIn's Espresso uses Helix for cluster management(ZooKeeper), and has a routing tier as above.
- HBase, SolrCloud, Kafka use ZooKeeper to track partition assignments.
- MongoDB has similar architecture, but has its own config server and mongos daemons as routing tier.
- **Cassandra** and Riak take a different approach: they use **gossip protocol** among nodes to disseminate changes in cluster state. Requests can be sent to any node, which forwards them to the appropriate node (Figure 6-7, approach 1). This puts more complexity in the database nodes, but **avoids dependency** on external coordination services like ZooKeeper.
- Clients need to find the IP addresses to connect to. Using DNS is sufficient.

Parallel Query Execution

- Simple queries, NoSQL distributed datastores support this level of access.
- **Massively parallel processing(MPP)** relational database products for **analytics**, **data warehouse**, support more sophisticated queries.
- The MPP query optimizer breaks this complex query into a number of execution stages and partitions, many of which can be executed in parallel on different nodes of the database cluster. Queries that involve scanning over large parts of the dataset particularly benefit from such parallel execution.