DDIA Notes Chapter – Partitioning

As systems scale, a single machine is no longer feasible to process large datasets or high query throughput. Partitioning (sharding) breaks up the dataset into smaller subsets. Each partition becomes its own small database. Partitioning spreads the data and query load across multiple machines.

Typically, the database will assign multiple partitions to a node. This improves dynamic load balancing across nodes. If a leader-follower replication model is used, each partition's leader is assigned to one node and its followers are assigned to other nodes. Each node could be a leader to some partitions and a follower to other partitions.

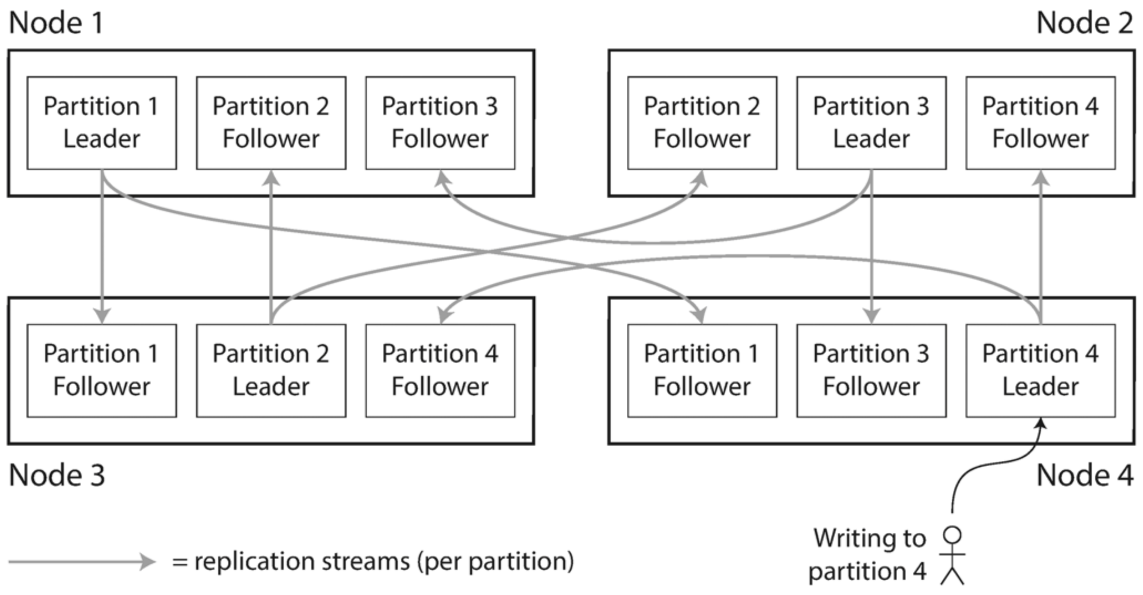

Figure 6-1. Combining replication and partitioning: each node acts as leader for some partitions and follower for other partitions.

Federation

Federation is where the data model is split up by function across different machines. For example, a monolithic data model can be split up to forums, users, and products. Federation alone is not effective if the data model has very large functions or tables. Also, joining data between two functions becomes complex. Once the data is federated appropriately, the following partitioning methods can be applied.

Key-Value Data Model

With key-value stores, there are two main approaches to partitioning: key range partitioning, and hash partitioning.

Key Range Partitioning

Key range partitioning is where keys are sorted and partitioned by a range of keys. This allows for efficient key range queries.

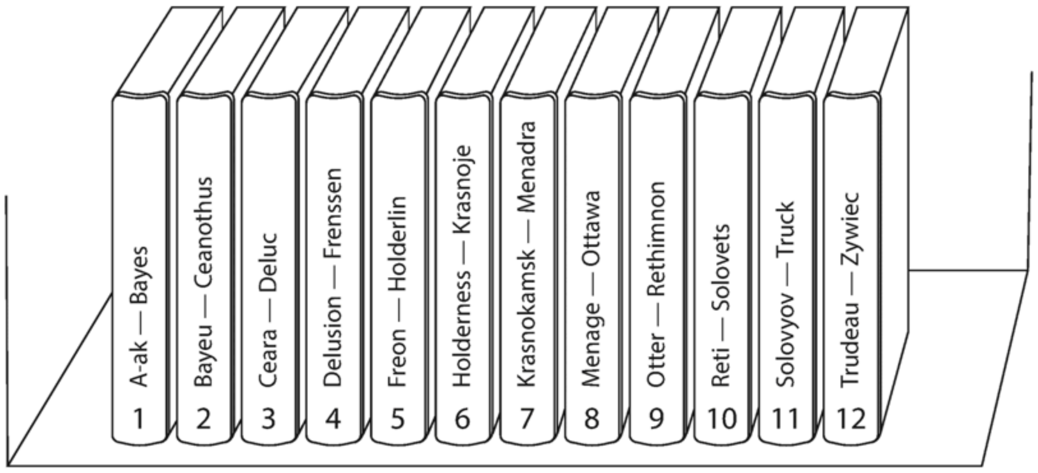

Figure 6-2. A print encyclopedia is partitioned by key range.

With this approach, a partition can have disproportionately high load (a hot spot). In some cases, all load could end up on one partition. This would happen when a specific range of keys become very popular. To mitigate this risk, partitions can be rebalanced by splitting a range into two subranges (1 partition divided into 2 partitions).

In another case, if the database is partitioned by timestamp, all writes would go to the same partition (the one for today). This is mitigated by prefixing the timestamp key with some attribute. For example, the key could be prefixed with a user or component name so that the load is distributed by this name instead of the timestamp.

Hash Partitioning

Hash partitioning is where a hash function is applied to each key and partitioned by a range of hashes (instead of range of keys).

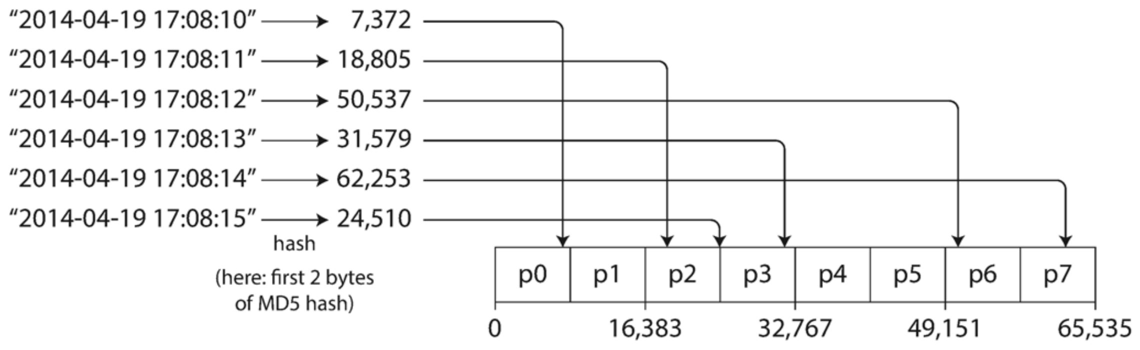

Figure 6-3. Partitioning by hash of key.

This approach is used to uniformly distribute data and minimize hot spots. Key range queries become inefficient as there is no ordering of keys. Hash partitioning can be implemented as consistent hashing or rendezvous hashing.

Even with hash partitioning, a single key can become very hot. Applications can use a simple technique to mitigate this: add a random number at the beginning (or end) of the key. A two-digit decimal random number will split a key evenly across 100 different keys. This will distribute writes to different partitions. Reads will have to do additional work as they will need to read data from 100 keys and compile the data.

Data Model with Secondary Indexes

Partitioning becomes more complicated when secondary indexes are involved. The secondary indexes need to be separately partitioned. With secondary indexes, there are two main approaches to partitioning: document-based partitioning (local indexes) and term-based partitioning (global indexes).

Document-Based Partitioning

Document-based partitioning (local indexes) is where the secondary indexes are stored in the same partition as the primary key and value. Each partition only maintains the secondary indexes of the datasets it controls.

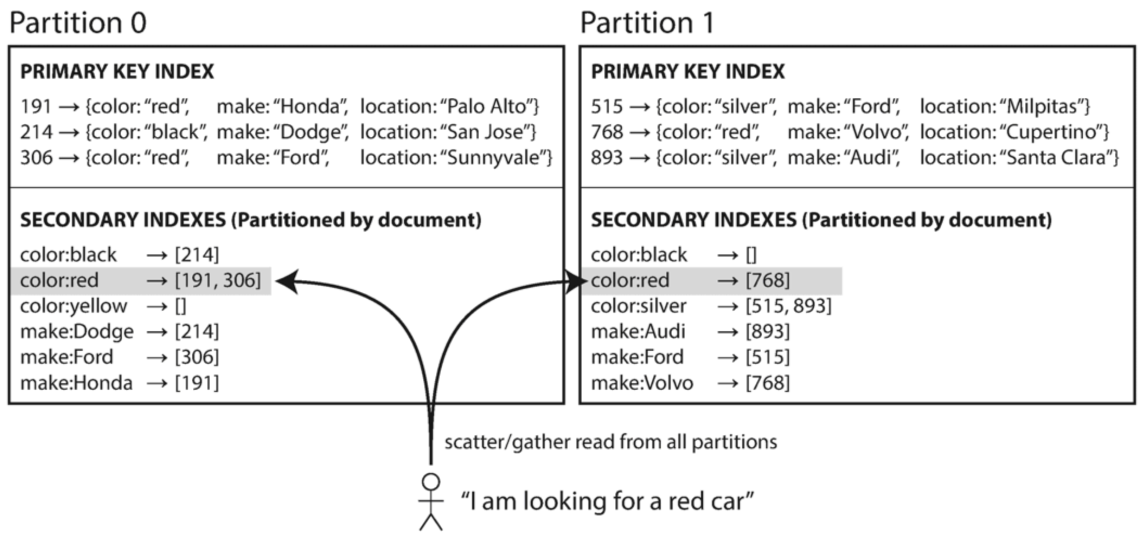
Partition 0
PRIMARY KEY INDEX
191 → {color: "red", make: "Honda", location: "Palo Alto"}
214 → {color: "black", make: "Dodge", location: "San Jose"}
306 → {color: "red", make: "Ford", location: "Sunnyvale"}
SECONDARY INDEXES (Partitioned by document)
color:black → [214]
color:red → [191, 306]
color:yellow → []
make:Dodge → [214]
make:Ford → [306]
make:Honda → [191]
Partition 1
PRIMARY KEY INDEX
515 → {color: "silver", make: "Ford", location: "Milpitas"}
768 → {color: "red", make: "Volvo", location: "Cupertino"}
893 → {color: "silver", make: "Audi", location: "Santa Clara"}
SECONDARY INDEXES (Partitioned by document)
color:black → []
color:red → [768]
color:silver → [515, 893]
make:Audi → [893]
make:Ford → [515]
make:Volvo → [768]
scatter/gather read from all partitions
"I am looking for a red car"

Figure 6-4. Partitioning secondary indexes by document.

On write, a single partition needs to be updated. On read, secondary indexes must be gathered across ALL partitions (scatter/gather). This makes read queries on secondary indexes very expensive.

Term-Based Partitioning

Term-based partitioning (global indexes) is where secondary indexes are partitioned separately from primary keys. On write, several partitions of the secondary index must be updated.

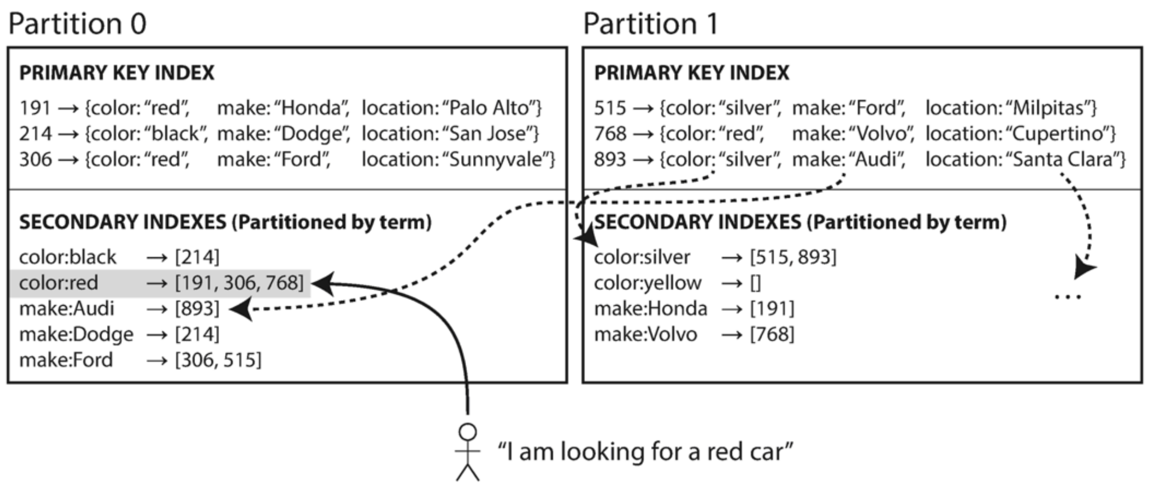
Partition 0
PRIMARY KEY INDEX
191 → {color: "red", make: "Honda", location: "Palo Alto"}
214 → {color: "black", make: "Dodge", location: "San Jose"}
306 → {color: "red", make: "Ford", location: "Sunnyvale"}
SECONDARY INDEXES (Partitioned by term)
color:black → [214]
color:red → [191, 306, 768]
make:Audi → [893]
make:Dodge → [214]
make:Ford → [306, 515]
Partition 1
PRIMARY KEY INDEX
515 → {color: "silver", make: "Ford", location: "Milpitas"}
768 → {color: "red", make: "Volvo", location: "Cupertino"}
893 → {color: "silver", make: "Audi", location: "Santa Clara"}
SECONDARY INDEXES (Partitioned by term)
color:silver → [515, 893]
color:yellow → []
make:Honda → [191]
make:Volvo → [768]
...
"I am looking for a red car"

Figure 6-5. Partitioning secondary indexes by term.

On read, data can be served from a single partition. This makes read queries on secondary indexes efficient. At the same time, writes to secondary indexes become slower and more complicated.

DynamoDB uses global indexes. Updates to secondary indexes are asynchronous and therefore, eventually consistent.