October 2nd 2019

Partitioning

When the total amount of unique data exceeds one computer's storage capacity, or query processing capability, the data needs to be partitioned.

Larger, more complex queries may need to be parallelized across multiple nodes, which is hard to do well.

Partitions can be replicated, and the replicas can be stored on a whole bunch of other nodes that may be the leaders of other partitions: (if using leader-follower)

Node 1	Partition 1 Follower	Partition 2 Leader ◣	Partition 3 Follower
Node 2	Partition 1 Follower	Partition 2 Follower	Partition 3 Leader ◣
Node 3	Partition 1 Leader ◣	Partition 2 Follower	Partition 3 Follower

Partitioning of Key-Value Data

If the data is perfectly distributed, then the throughput of the whole system scales linearly. But if the partitioning is bad, then you can end up with hot spots.

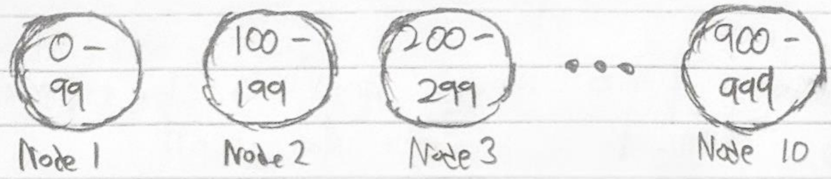
Random assignment is bad as reads can't determine which node it's on, and so you have to query all nodes.

Instead, we can leverage key-value's primary indexing to also act as a look up. This means it's a giant distributed K-V table.

Partitioning by Key Range

October 2nd 2019

If the keys span a continuous range, each partition can contain a sub-range of the total:



To distribute it evenly, the spacing should be equal.

Partition boundaries can be determined manually, or automatically (which requires rebalancing).

This is used by BigTable, HBase, RethinkDB, and MongoDB before V2.4

Data between ranges can be kept in sorted order, meaning range searches are easy.

A downside is that it doesn't account for the amount of load a particular partition may receive. This means that the key has to be chosen carefully.

Partitioning by Key Hashing

Hash functions can be used to partition, and doesn't need to be cryptographically strong. As long as it's uniform, it's good.

Be aware that some language built-in hash functions like Object.hashCode() in Java may have different hashes for different processes, so it's unsuitable for partitioning.

Now, instead of separating boundaries by key, you can separate it by hash ranges.

October 2nd 2019

Consistent hashing is when the hash boundaries are randomly chosen, and is rarely used in databases because it doesn't work well in practice. (So it's usually used in other applications). The term is sometimes misused in place of "hash partitioning".

A downside is that sorting by key is not possible anymore, so range queries are harder. Most DBs don't allow for range queries by primary key, or they do by sending it to all partitions (as seen in MongoDB).

Workload Balancing

The main issue is that despite keys being distributed perfectly, it's perfectly possible (albeit unlikely) that all the reads and writes go to a small amount of keys.

For example, a trendy reddit link can be bombarded heavily in a small time frame. Since it's the same entry, all the queries go to the same partition(s) anyway.

Most DBs don't automatically handle skewed workloads, so the application needs to be responsible.

One technique is to divide the key into subkeys: you could append a random number to the front/back, and if done from 1-100, throughputs for writes increases by 100

The downside is reads need to query 100 times more keys, so this is only worth doing if write throughput is extremely high.

Partitioning and Secondary Indexes

October 2nd 2019

r/w

Without any secondary indexes, then the primary key can be solely used to determine where to go for queries.

Secondary indexes allows for efficient (i.e. practical) searching for particular values or ranges. They're complex to implement in K-V databases, so most don't (HBase, Voldemort, etc.) but some do (Riak).

Some databases focus on secondary indexes a lot, like search DBs (Elasticsearch, Solr).

Partitioning Secondary Indexes by Document

The idea is to treat each partition completely separately. Any secondary indexes will only index documents within a specific partition.

The index's locality means that any write operations will only involve the partition that involves that document ID.

Partition 1

Primary Key Index
a9f : { "uni": "waterloo", "year": 3 }
c45 : { "uni": "toronto" "year" : 2 }

Secondary Indexes	
"uni": "waterloo"	[a9f]
"uni": "toronto"	[c45]
"uni": "ottawa"	[]
"year": 2	[c45]
"year": 3	[a9f]

Partition 2

Primary Key Index
91g : { "uni": "waterloo", "year": 3 }
xi2 : { "uni": "ottawa", "year": 3 }

Secondary Indexes	
"uni": "waterloo"	[91g]
"uni": "toronto"	[]
"uni": "ottawa"	[xi2]
"year": 2	[]
"year": 3	[91g, xi2]