Partitioning

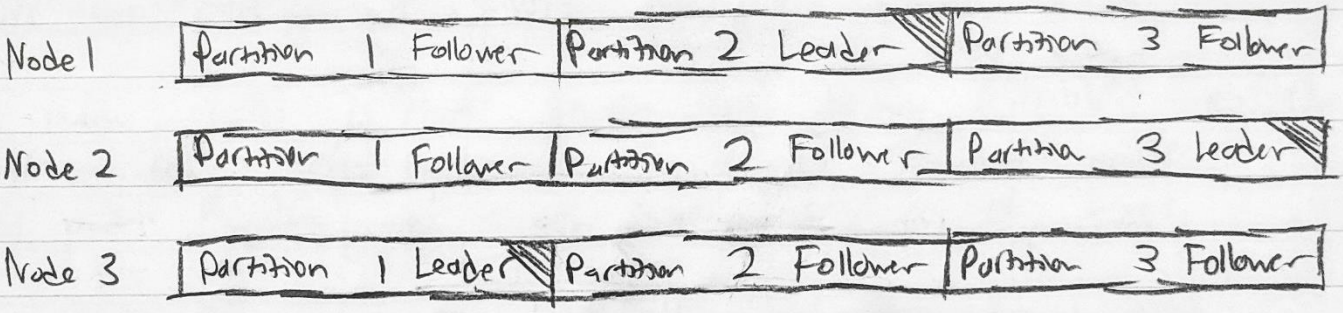
October 2nd 2019

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)



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.

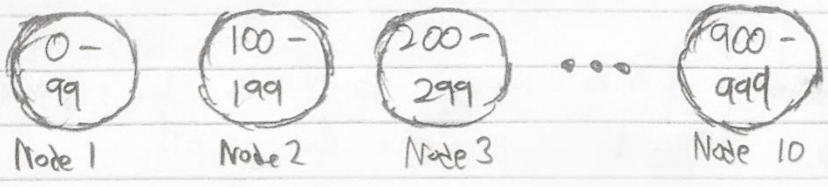
Instead, we can leverage key-value's primary indexing to also act as a look up. This means it's a giant distributed K-V table.

Partitioning by Key Range

October 2nd 2019

90

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

iriw

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]

October 6th 2019

Reads became more expensive due to queries now requiring all partitions to be searched, as the data is only searchable locally.

Widely used: MongoDB, Riak, Cassandra, Elasticsearch, SolrCloud, and VoltDB

A strategy to reduce read costs is to try to organize data so that queries only require data from one partition, but this isn't always possible.

Partitioning Secondary Indexes by Term (Global Indexing)

Instead of indexing each document locally, you could index them globally through certain terms or traits.

Since indexing it through all a single computer is limiting, the different terms that are indexed could itself be partitioned.

The same data from above would be indexed like this:

Partition 1

Primary Key Index

a95 : {"uni": "waterloo", "year": 3}

c45 : {"uni": "toronto", "year": 2}

Secondary Indexes

"uni": "waterloo" [a95, 91g]

"year": 2 [c45]

Partition 2

Primary Key Index

91g : {"uni": "waterloo" "year": 3}

xi2 : {"uni": "ottana" "year": 3}

Secondary Indexes

"uni": "toronto" [c45]

"uni": "ottana" [xi2]

"year": 3 [a95, 91g, xi2]

October 6th 2019

The terms themselves can be partitioned by value or their hash. Values allow for range scans (like ranges of numbers), while hashes distribute it more evenly.

Operation complexity is flipped: reads are cheap as only one partition is required to search something

But writes are more difficult as modifying a single document may change multiple properties, which can be stored on different partitions. This effectively means a distributed transaction is required, to implement it synchronously.

But usually they're not synchronous, as index updates are applied at some point later in time.

Partition Rebalancing

Scaling up or down requires load to be redistributed, which is known as partition rebalancing.

When rebalancing, the service should still be up, and should minimize data movement.

Rebalancing can occur either automatically or manually. When fully automatic, partitions are moved without any administrator interaction

If automatic, it can be unpredictable. It needs to ensure that it's done only when necessary, because the network and nodes can become overloaded

October 11th 2019

Problematic Strategy: Hash % N

When just modding by N, nodes that hold data for the higher numbered partitions will require excessive amounts of data to be moved.

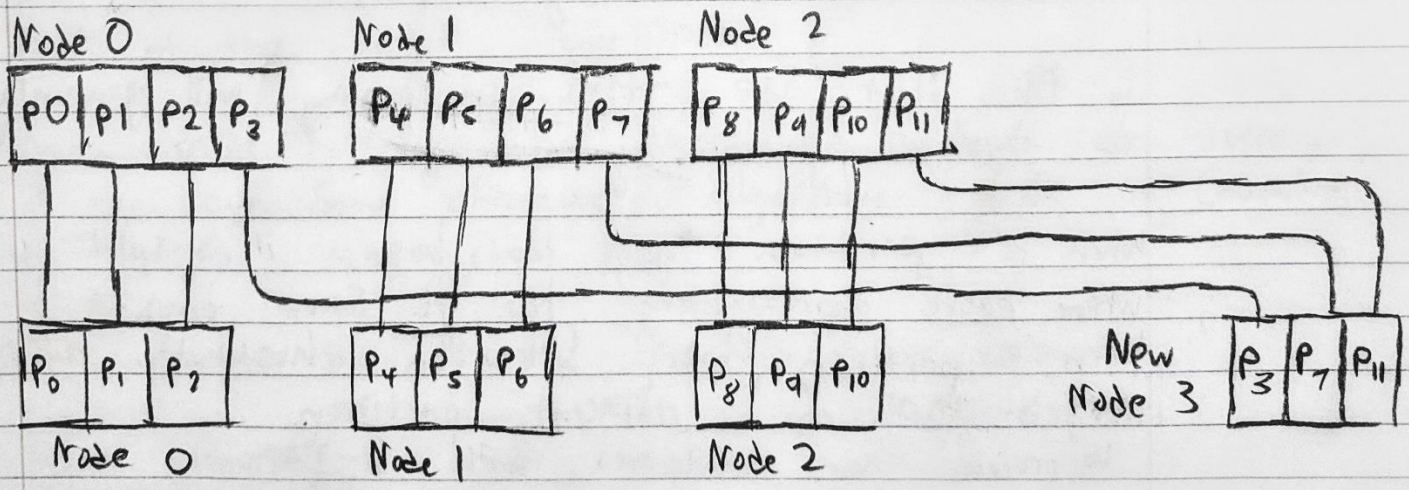
As N gets very large, the amount of data that actually needs to be moved can easily be over 50% on some nodes.

This is why assigning key ranges is preferable, as the ranges may be flexible

Partitioning Strategy: Fixed Partition Count

The idea is to have far more partitions than nodes, and then spread them out. Having >100 partitions per node is not unheard of.

Then, scaling up or down will mean taking or giving a partition from/to each other node.



Since partitions are sent whole, and ranges aren't themselves modified, the old nodes can service requests until the new node is up.

October 11th 20

Since partitions are fixed, the amount of partitions picked at the start shouldn't be too high or low

If partitions are too small and numerous:
- High management overhead, since partition count is higher

If partitions are too big and few:
- Rebalancing is hard on the nodes and network
- Node recovery is also more expensive
- Unable to accomodate future growth.

This is done in: Riak, ElasticSearch, Couchbase, and Voldemort.

Rebalancing Strategy: Dynamic Partitioning

When partitioning by key ranges (not hash ranges), fixed partitions are disadvantageous as the data distribution can be really skewed.

So DBs that use this partitioning will generally do dynamic partitioning.

When a partition gets too large, it is split into 2 where each approximately has the same amount of data. When a partition goes below a threshhold, it is merged with an adjacent partition.
↳ process shares similarities with B-trees

Since the amount and size of partitions always changes, scaling issues are not a problem.

October 11th 2019

If the DB starts with only one partition, then only one node can be used, leaving the rest idle. So HBase and MongoDB allow pre-splitting

Used in HBase and RethinkDB. In the case of MongoDB, which is hash partitioned, it still works well.

Rebalancing Strategy: Proportional Partitioning

Here, we decide to fix the number of partitions per node. This means the size of each partition grows proportionally when the number of nodes are stable, and the addition of new nodes will decrease the size of the partitions.

This contrasts to the other solutions where the number of partitions is actually independent of the number of nodes.

When scaling up, a random number of existing partitions are split. Then, the new node takes ownership of half of the split partitions.

Some splits may be unbalanced, but on average, it evens out. Some alternative algorithms exist in Cassandra to avoid unbalanced splits.

This approach is used in Cassandra and Ketama.

Request Routing

Since data is now partitioned, requests need to hit the node that has control over that partition.

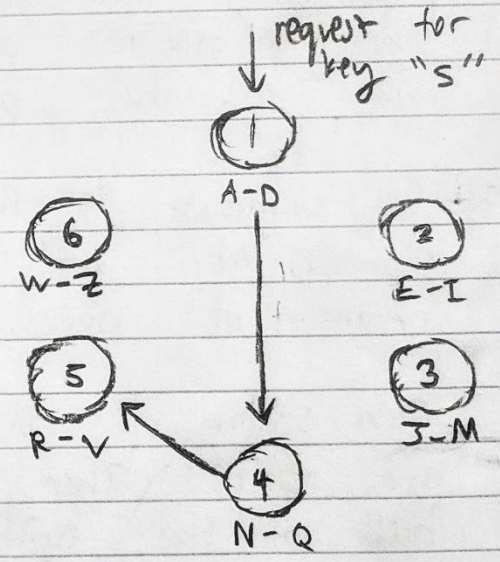
October 13th 2019

This effectively translates to mapping keys to Ip addresses. It's the same general problem as a DNS.

Request Forwarding: One way to tackle the issue is to allow all nodes to accept the request, but send it to the node that has control over the key.

It doesn't need to directly need to be forwarded to the key immediately - the request may be handed off

In this case, the knowledge of where the partitions are is kept within the node.



In the example, there are 2 forwards, as node 1 does not know of every partition, (ie. it only has partial knowledge).

However, when scaling up or down, the nodes need to be aware of the partitions rebalanced. This requires new and exiting nodes to update the others. This is typically done through gossip broadcasting.

This strategy is used in Cassandra and Riak.

Dedicated Router: Another strategy is to have a partition manager that keeps track of what node owns which partition.