Partitioning

- separate from replication
- partitioning shards/splits a database based on some key, so that queries only need to look at the particular shard of interest
- each partition is in turn replicated!

The main reason for wanting to partition data is *scalability.*

- different partitions can be placed on different nodes in a shared nothing architecture
- so a large dataset can be distributed across many disks and the query load across many processors
- queries that touch a single partition can be executed on the corresponding node -- so query throughput can be scaled by adding more nodes
- large queries can potentially be parallelized as well though this is much harder

Partitioning schemes for key-value stores: how to decide data → partition?

- Partition by Key Range
  - all keys within a range are on one partition
  - every range does not the same length of boundary range
    - the partition boundaries need to adapt to the data
  - Within each partition, keys can be maintained in sorted order
    - perhaps using an SSTable
    - this makes range scans easy
  - Pick a good key/combination of keys so you prevernt hot spots
    - eg if you partition on (current) timestamp range, then at different points in the day all the write traffic will be focused on one partition ie the partition that maps to the current timestamp range
    - so you can perhaps use (sensor_name + timestamp) as a key
      - assuming there are many sensors, this will spread the write load more evenly across partitions
      - but when you want to retrieve all the data for a particular time range, you ened to perform a separate query per sensor
- Partition by Key Hash
  - A good hash() makes a skewed distribution uniform
    - so it can be useful to spread keys evenlyh among partitions
    - however, you lose the ability to run effective range queries since sort order is not preserved -- they now need to be sent to all partitions
  - Cassandra does both: compound primary key consisting of several columns = concatenated index
    - first key is used to hash to a partition
    - the other columns are used as a concatenated index to sort the data
    - so if a query specifies a fixed value for the first column and a range across

the others, this works well
        - this is typically a good model for 1-many relationships: eg (user_id, timestamp) for every user making many social media posts
        - partition on user, sort by timestamp
        - different users may be on different nodes, but for a given user, all the data is sorted and stored on one node
- Skewed Workload and Relieving Hot Spots
  - in the extreme case where you have all reads and writes for the same key, or keys on the same node, you have a hot node
  - approaches
    * split hot keys randomly: just a random 2 digit number would split the key evenly across 100 partitions
      - problem is all reads for the key have to pull data from 100 partitions and combine them
      - also, you need to track which keys are being split this way

Partitioning and Secondary indices

- A secondary index does not usually specify a record uniquely but is a way to of searching for occurrences of a particular value
- Eg: elasticsearch and solr, which are search servers that provide an index over the dataset
- the problem is that they dont map neatly to partitions
- two approaches
  - document based partitioning
    * each partition is completely separate -- each partition maintains its own secondary indexes covering only the documents in that partition = local index
    * but this typically means read queries on secondary indices need to be sent to all partitititions, and the application needs to combine the results that are returned
      - this approach is called scatter/gather, and it can make read queries on secondary indices expensive
  - term based partitioning
    * construct a global index that covers data in all partitions
    * however the index cant be on one node - else it would be a bottleneck - so it needs to be partitioned as well, but it can be partitioned differently from the primary key index
    * makes reads easier but writes are complicated, because a single db write can affect n partitions -- since each term in the write might be on a different partition
      - not all dbs support distributed transactions

Rebalancing partitions: how to decide partition → node?

The system may change

- query throughput increases, so we need more cpus to deal with the load
- dataset size increases, so we need ot add more disks or RAM
- a machine fails, and other machines need to take over the failed machine's responsibilities

All of these need data to be moved from one node in the cluster to another -- this is called rebalancing.

Strategies for rebalancing

- dont do hash(key) mod N because when N changes, the rebalancing operation is very expensive
  - have to move a lot of data around for every change to N
- Fixed number of parititions, many more than there are nodes, and assign several partitions to each node. Eg 1000 partitions for n=10 nodes
  - then if a new node is added, it can steal some partitions from every existing node until the number of partitions per node is roughly equal
  - only entire partitions are moved!
- dynamic partitioning
  - if a partition gets too big, split it into 2, both of which stay on the same node
  - the number of partitions adapts to the data volume
  - usually useful for key-range partitioned data
- make number of partitions proportional to number of nodes i.e. fixed number of partitoins per node

Request routing

- The fundamental question is: if you partition your dataset across multipl nodes -- when a client wants to make a request, how does it know which node to connect to?
  - as partitions are rebalanced, the answer changes as the assigment of partitions to nodes changes
  - which IP and port does the client need to connect to?
- This is an instance of the more general problem of *service discovery*
  - any piece of software that is accessible over a network has this problem, especially if it is aiming for high availability -- i.e. it is running in a redundant configuraion across multiple machines
- At a high level, there are a few approaches
  - allow clients to contact any node (e.g. via a round-robin load balancer). if that node coincidentally owns the partition to which the request applies, it can handle

        the request directly. Else if forwards the request to the appropriate node, receives the reply, and passes the response to the client
            * Q: how does the node know what the right partition is?
    - send all client requests to a routing tier first, which determines the node that should handle each request and forwards the request accordingly.
        * the routing tier does not handle any requests: it just acts as a partition-aware load balancer
    - require that the client be aware of the assignment of partitions to nodes, and that it can therefore connect to the required node without any intermediary
- In all cases, the question is: how does the component making the routing decision learn about changes in the assignment of partitions to nodes?
- This is challenging!
    - all the participants need to agree on the partition <> node mapping! or requests can get routed wrongly
- Many distributed data systems rely on a separate coordination service like Zoo Keeper to keep track of this cluster metadata
    - each node registers itself in zookeeper
    - zookeeper maintains the canonical mapping of partitions <> nodes
    - all actors subscribe to this information in zookeeper
    - when a partition changes ownership, Zookeeper notifies the routing tier so that it can keep its routing information up to date
- Another approach is to use a *gossip protocol* among the nodes to disseminate changes in cluster state

https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/
Salting hashes sounds like one of the steps of a hash browns recipe, but in cryptography, the expression refers to adding random data to the input of a hash function to guarantee a unique output, the hash, even when the inputs are the same. Consequently, the unique hash produced by adding the salt can protect us against different attack vectors, such as hash table attacks, while slowing down dictionary and brute-force offline attacks.