Data Partitioning

- Why Partition?
  - At some point, single node based DB isn't going to cut it: concurrent read/write traffic will go up as you scale, so latency & throughput are affected
  - A no-SQL like solution could be better, but historical codebase & close cohesion w/traditional DB make this expensive
  - Data Partitioning: enables us to use multiple nodes where each node manages some part of the whole data

- Sharding (Partitioning)
  - To divide load among multiple nodes, we need to partition the data using sharding
  - The partitions or "shards" should be even in size approximately
  - If the partitions are unbalanced, most queries will go to a few nodes
  - Heavily loaded shards will form a bottleneck: hotspots
  - Two ways to shard: vertical/horizontal Partitioning

- Vertical Partitioning
  - used to increase speed of data retrieval from a table consisting of columns w/very wide text or a binary large object (blob)
    ↳ In this case column w/ large text is split into diff table

- Horizontal Sharding
  - Splits tables row-wise
    - ↳ each partition of the original table distributed over DB servers is called a shard
  - Two types: key-range based sharding & hash based
  - Key-Range based
    - ↳ each partition assigned a continuous range of keys
    - ↳ sometimes, a DB consists of multiple tables bound by foreign key relationships. In such a case, the horizontal partition is made using the same key on all tables in a relation
      - tables (or subtables) that belong to the same partition key are distributed to one DB shard
- Basic Design Techniques
  - partition key in the customer mapping table
    - ↳ this table resides on each shard, stores partition keys used in the shard
    - ↳ Applications create a mapping logic b/t the partition keys & DB shards by reading this table from all shards to make the mapping efficient
  - Partition key column is replicated in all other tables as a data isolation point
    - ↳ It has a tradeoff: ↑ storage ↓ speed to find desired shard.
  - Primary keys are unique across all DB shards
    - ↳ avoids key collision during data migration among shards & merging of data in online analytical processing environments

- Hash Based Sharding
  - uses hash like function on an attribute
  - use hash function on the key to get a hash value & then mod by # partitions

- when we've found an appropriate function for keys, we may give each partition a range of hashes rather than a range of keys. Any key whose hash occurs inside that range will be kept in that partition

- Consistent Hashing
  - assigns each server or item in a distributed hash table to a place on an abstract circle, called a ring, irrespective of the num of servers in the table
    ↳ permits servers & objects to scale w/out compromising system's performance

Pros:

- Easy to scale horizontally
- Increases throughput & improves latency

Cons:

- Randomly assigning nodes in the ring may cause non-uniform distribution

- Rebalance the Partitions
  - Query load can be imbalanced across nodes b/c:
    ↳ data distribution of the data isn't equal
    ↳ Too much load on a single shard
    ↳ There's an ↑ in query traffic, we need to add more nodes to keep up

Solutions:

- Avoid Hash mod n
- Fixed num of partitions
  ↳ # partitions created is fixed @ time we setup DB
  ↳ create more partitions than nodes & assign them to nodes

- Dynamic Partitioning
  - ↳ when the size of the partition reaches the threshold, it's split evenly into 2 partitions
  - ↳ one of the 2 splits is assigned to one node & the other one to another node. In this way, the load is ÷ equally
  - ↳ Cons: Difficult to apply dynamic rebalancing while serving reads/writes
- Partitioning & Secondary Indexes
  - ↳ How do we partition if we have access to records thru secondary indices?
    - Partition secondary indexes by document
      - ↳ each partition has its secondary indices covering just the docs in that partition
      - ↳ This can be expensive