**Chapter 6**

Design a key-value store

Key-value store – a non-relational database

* Keys
  + Must be unique
  + Can be plain text or hashed values
* Values
  + Accessed through keys
  + Can be strings, lists, objects
  + Usually treated as an opaque object
* 2 essential operations
  + Insert new key-value pairs – put(key, value)
  + Retrieve values based on keys – get(key)

**Design**

*Single server key-value store*

* Easy to manage but lack scalability
* A hash table stores all key-value pairs
  + Requires memory
  + As data scales, space constraints for memory becomes problematic
* Two optimizations
  + Data compression – reduce memory intake per key-value pair
  + Only store frequently used data in memory and the rest in disk

*Distributed key-value store*

Distributed hash table – distributing key-value pair stores across many servers

CAP theorem – Important for designing distributed systems

*CAP Theorem*

* Stands for Consistency, Availability, Partition Tolerance
* Impossible for a distributed system to simultaneously provide more than two:
  + Consistency – client should see the most up-to-date data
  + Availability – requests will receive a response even if some nodes are down
  + Partition Tolerance – system should continue operating despite having partitions – a communication break between two or more nodes
* Key-value stores can only maintain 2 of 3 CAP characteristics:
  + CA – system is always consistent and available
    - The ideal system but not realistic
    - Network failure is unavoidable, therefore, distributed systems must always tolerate partitions
  + CP – supports consistency
    - Blocks all write operations during a network partition
    - Avoid data inconsistency until system is fully availability
    - Ex) bank systems must be consistent since it deals with money
  + AP – supports availability
    - Accepts all operations even though the data may be stale
    - Data is synced when network partition is resolved
    - Ex) newsfeed or social media wants to maximize the engagement time of their users and therefore, value availability over consistency

**System Components**

*Data partition*

* For large apps, it is infeasible to fit complete data sets in a single server
* Simplest solution – split data into smaller partitions storing them in multiple servers
* Challenges for partitioning data:
  + Distributing data evenly across servers
  + Minimize data movement when nodes are added or removed
* Advantages to consistent hashing (see last chapter) to partition data:
  + Automatic scaling – servers can be added or removed automatically depending on the load
  + Heterogeneity – number of virtual nodes for a server is proportional to the server capacity to balance the distribution

*Data replication*

* Data must be replicated asynchronously over a configurable number of servers
* Nodes in the same data centers often fail at the same time due to power outages, network issues, natural disasters, etc.
  + For better reliability, replicas should be shared in distinct data centers

*Consistency*

Quorum Consensus – a strict concurrency control protocol that can guarantee consistency for both read and write operations

|  |  |
| --- | --- |
| **Variable** | **Definition** |
| N | number of replicas |
| W | a write quorum of size W. For a write operation to be considered as successful, write operation must be acknowledged from W replicas |
| R | a read quorum of size R. For a read operation to be considered successful, read operation must wait for responses from at least R replicas |

* A coordinator acts as a proxy between the client and nodes
* Configuration of W, R and N is a tradeoff between latency and consistency
  + When W or R = 1, the operation is quick because a coordinator only needs to wait for one replica to respond
  + Increasing W or R offers better consistency but less latency
* Some use cases for configurations
  + R = 1 and W = N – system is optimized for a fast read
  + W = 1 and R = N – system is optimized for a fast write
  + W + R > N – strong consistency is guaranteed

*Consistency models*

* Strong consistency – a client will never see out-of-date data
  + Often blocks all operations until every replica has agreed on current write
  + This approach is not ideal for high available systems
* Weak consistency – read operations may not always show the most updated value
* Eventual consistency – a specific form of weak consistency. Given enough time, all updates are propagated, and all replicas are consistent
  + Adopted by Dynamo and Cassandra
  + Allows inconsistent values to enter the system and force the client to read the values to reconcile

*Inconsistency Resolution: versioning*

Versioning – treat each data modification as a new immutable version of data

* Versioning and vector clocks are used to solve inconsistency problems
* When two different servers both receive write operations, vector clock is a common technique to solve this problem

Vector clock – [server, version] pair used to check If one version precedes, succeeds, or in conflict with others

* Represented by…

Where D is a data item

V1 is a version counter

S1 is a server number

* If data item D is written to server S, the system must perform one of these tasks:
  + Increment v if [Si, Vi] exists
  + Otherwise, create a new entry [Si, 1]
* Makes it easy to know which versions are ancestors or siblings of each other
* Disadvantages
  + Adds complexity to client and requires conflict resolution logic
  + A threshold is needed for the length of data items in vector clocks
    - Solution – can replace the oldest pairs when threshold is reached
    - This may can lead to inefficiencies in reconciliation
    - But Amazon Dynamo has not encountered this problem

**Handling failures**

*Failure detection*

* In distributed systems, it is insufficient to believe that a system is down because another says so
* Solutions
  + All-to-all multicasting – straightforward solution but ineffective when there are many servers in the system
  + Decentralized failure detection – gossip protocol

Gossip Protocol – a peer-to-peer communication mechanism in which nodes periodically exchange state information of themselves and other nodes they know about

* Communication is based on the way epidemics spread
* Concept
  + Each node contains a node membership list – contains member IDs and heartbeat counters
  + Each node periodically increments its heartbeat counter
  + Each node periodically sends heartbeats to a set of random nodes, which in turn propagate to another set of nodes
  + When heartbeat is received, membership list is updated
  + If heartbeat has not increased for more than a predefined period, that member is considered offline

*Handling temporary failures*

* Solutions
  + Quorum consensus can block read and write operations until nodes are back online, but this may be too strict
  + Sloppy quorum – instead of enforcing quorum requirements, the system chooses the first W healthy servers for writes and first R healthy servers for reads on the hash ring. Offline servers are ignored
  + Hinted handoff – when a server is down, another server will process requests temporarily. When the down server is up, changes will be pushed to consistency

*Handling permanent failures*

* Implement an anti-entropy protocol
* Anti-entropy – comparing each piece of data on replicas and updating each replica to the newest version

Merkle Tree – a tree where every leaf node is labelled with cryptographic hash of a data block and every node that is not a leaf (branch or inode) is labelled with the cryptographic hash of the labels of its child nodes

* Used for inconsistency detection and minimizing data transfer

|  |
| --- |
| *Building a Merkle tree*   1. Divide key space into buckets.    * A bucket is used as the root level node to maintain a limited depth 2. Hash each key in a bucket using a uniform hashing method 3. Create a single hash node per bucket 4. Build the tree upwards till root by calculating hashes of children |

* Compare two Merkle trees
  + Start with root hashes
  + If root hashes match, both servers have the same data
  + If hashes disagree, traverse down the tree comparing child hashes until which bucket in disagreement is found
* Using Merkle trees, the amount of data needed to be synchronized is proportional to the difference between the two replicas, and not the amount of data they contain

*Handling data center outage*

* Data center outage causes all servers in its region to go down
* It is important to replicate data across multiple data centers

**System architecture diagram**

* Main architectural features
  + Clients communicate with the key-value store through APIs
  + A coordinator is a node that acts a proxy between the client and store
  + Nodes are distributed on a ring using consistent hashing
  + Decentralized system so adding and moving nodes can be automatic
  + Data is replicated at multiple nodes
  + No single point of failure
  + Every node has the same set of responsibilities

**Architecture of Cassandra 8**

*Write Path*

* 1. Write request is persisted on a commit log file
  2. Data is saved in memory cache
  3. When memory cache is full or reaches a predefined threshold, data is flushed to SSTable on disk
* SSTable - sorted-string table of key value pairs

A diagram of a server

Description automatically generated

*Read Path (Memory Cache)*

* Read requests will first check if the data is in the memory cache

A computer screen shot of a server

Description automatically generated

*Read Path (Memory Cache)*

* When not in memory, it will retrieve from disk
* Bloom filter – an efficient way to find out which SSTable contains the key

1. Checks memory cache, skip subsequent steps if not found
2. Checks bloom filter
3. Bloom filter processes – figuring out which SSTables contains the key
4. SSTables return the result of the data set
5. Result is returned to the client

A computer screen shot of a server

Description automatically generated

**Summary**

|  |  |
| --- | --- |
| **Goals/Problems** | **Technique** |
| Storing large amount of data | Consistent hashing to distribute data evenly |
| High availability reads | Data replication  Multi-data center setup |
| High availability writes | Versioning and conflict solutions with vector clocks |
| Dataset partition | Consistent hashing |
| Incremental scalability | Consistent hashing |
| Heterogeneity | Consistent hashing |
| Tunable consistency | Quorum consensus |
| Handling temporary failures | Sloppy quorum & hinted handoff |
| Handling permanent failures | Merkle tree to identify failure |
| Handling data center outage | Cross-data center replication |