系统设计面试需要掌握的基础概念 – NOSQL Patterns

NOSQL Patterns

Over the last couple years, we see an emerging data storage mechanism for storing large scale of data. These storage solution differs quite significantly with the RDBMS model and is also known as the NOSQL. Some of the key players include ...

• GoogleBigTable, HBase, Hypertable

• AmazonDynamo, Voldemort, Cassendra, Riak

• Redis

• CouchDB, MongoDB

These solutions has a number of characteristics in common

• Key value store

• Run on large number of commodity machines

• Data are partitioned and replicated among these machines

• Relax the data consistency requirement. (because the CAP theorem proves that you cannot get Consistency, Availability and Partitioning at the the same time)

The aim of this blog is to extract the underlying technologies that these solutions have in common, and get a deeper understanding on the implication to your application's design. I am not intending to compare the features of these solutions, nor to suggest which one to use.

API model

The underlying data model can be considered as a large Hashtable (key/value store).

The basic form of API access is

• get(key) -- Extract the value given a key

• put(key, value) -- Create or Update the value given its key

• delete(key) -- Remove the key and its associated value

More advance form of API allows to execute user defined function in the server environment

• execute(key, operation, parameters) -- Invoke an operation to the value (given its key) which is a special data structure (e.g. List, Set, Map .... etc).

• mapreduce(keyList, mapFunc, reduceFunc) -- Invoke a map/reduce function across a key range.

Machines layout

The underlying infrastructure is composed of large number (hundreds or thousands) of cheap, commoditized, unreliable machines connected through a network. We call each machine a physical node (PN). Each PN has the same set of software configuration but may have varying hardware capacity in terms of CPU, memory and disk storage. Within each PN, there will be a variable number of virtual node (VN) running according to the available hardware capacity of the PN.

Data partitioning (Consistent Hashing)

Since the overall hashtable is distributed across many VNs, we need a way to map each key to the corresponding VN. One way is to use

partition = key mod (total_VNs)

The disadvantage of this scheme is when we alter the number of VNs, then the ownership of existing keys has changed dramatically, which requires full data redistribution. Most large scale store use a "consistent hashing" technique to minimize the amount of ownership changes.

In the consistent hashing scheme, the key space is finite and lie on the circumference of a ring. The virtual node id is also allocated from the same key space. For any key, its owner node is defined as the first encountered virtual node if walking clockwise from that key. If the owner node crashes, all the key it owns will be adopted by its clockwise neighbor. Therefore, key redistribution happens only within the neighbor of the crashed node, all other nodes retains the same set of keys.

Data replication

To provide high reiability from individually unreliable resource, we need to replicate the data partitions.

Replication not only improves the overall reliability of data, it also helps performance by spreading the workload across multiple replicas.

While read-only request can be dispatched to any replicas, update request is more challenging because we need to carefully co-ordinate the update which happens in these replicas.

Membership Changes

Notice that virtual nodes can join and leave the network at any time without impacting the operation of the ring.

When a new node joins the network

1. The joining node announce its presence and its id to some well known VNs or just broadcast)
2. All the neighbors (left and right side) will adjust the change of key ownership as well as the change of replica memberships. This is typically done synchronously.
3. The joining node starts to bulk copy data from its neighbor in parallel asynchronously.
4. The membership change is asynchronously propagate to the other nodes.

Notice that other nodes may not have their membership view updated yet so they may still forward the request to the old nodes. But since these old nodes (which is the neighbor of the new joined node) has been updated (in step 2), so they will forward the request to the new joined node.

On the other hand, the new joined node may still in the process of downloading the data and not ready to serve yet. We use the vector clock (described below) to determine whether the new joined node is ready to serve the request and if not, the client can contact another replica.

When an existing node leaves the network (e.g. crash)

1. The crashed node no longer respond to gossip message so its neighbors knows about it.
2. The neighbor will update the membership changes and copy data asynchronously

We haven't talked about how the virtual nodes is mapped into the physical nodes. Many schemes are possible with the main goal that Virtual Node replicas should not be sitting on the same physical node. One simple scheme is to assigned Virtual node to Physical node in a random manner but check to make sure that a physical node doesn't contain replicas of

the same key ranges.

Notice that since machine crashes happen at the physical node level, which has many virtual nodes runs on it. So when a single Physical node crashes, the workload (of its multiple virtual node) is scattered across many physical machines. Therefore the increased workload due to physical node crashes is evenly balanced.

Client Consistency

Once we have multiple copies of the same data, we need to worry about how to synchronize them such that the client can has a consistent view of the data.

There is a number of client consistency models

1. Strict Consistency (one copy serializability): This provides the semantics as if there is only one copy of data. Any update is observed instantaneously.

2. Read your write consistency: The allows the client to see his own update immediately (and the client can switch server between requests), but not the updates made by other clients

3. Session consistency: Provide the read-your-write consistency only when the client is issuing the request under the same session scope (which is usually bind to the same server)

4. Monotonic Read Consistency: This provide the time monotonicity guarantee that the client will only see more updated version of the data in future requests.

5. Eventual Consistency: This provides the weakness form of guarantee. The client can see an inconsistent view as the update are in progress. This model works when concurrent access of the same data is very unlikely, and the client need to wait for some time if he needs to see his previous update.

Depends on which consistency model to provide, 2 mechanisms need to be arranged ...

• How the client request is dispatched to a replica

• How the replicas propagate and apply the updates

There are various models how these 2 aspects can be done, with different tradeoffs.

Master Slave (or Single Master) Model

Under this model, each data partition has a single master and multiple slaves. In above model, B is the master of keyAB and C, D are the slaves. All update requests has to go to the master where update is applied and then asynchronously propagated to the slaves.

Notice that there is a time window of data lost if the master crashes before it propagate its update to any slaves, so some system will wait synchronously for the update to be propagated to at least one slave.

Read requests can go to any replicas if the client can tolerate some degree of data staleness. This is where the read workload is distributed among many replicas. If the client cannot tolerate staleness for certain data, it also need to go to the master.

Note that this model doesn't mean there is one particular physical node that plays the role as the master. The granularity of "mastership" happens at the virtual node level. Each physical node has some virtual nodes acts as master of some partitions while other virtual nodes acts as slaves of other partitions. Therefore, the write workload is also distributed across different physical node, although this is due to partitioning rather than replicas

When a physical node crashes, the masters of certain partitions will be lost. Usually, the most updated slave will be nominated to become the new master.

Master Slave model works very well in general when the application has a high read/write ratio. It also works very well when the update happens evenly in the key range. So it is the predominant model of data replication.

There are 2 ways how the master propagate updates to the slave; State transfer and Operation transfer. In State transfer, the master passes its latest state to the slave, which then replace its current state with the latest state. In operation transfer, the master propagate a sequence of operations to the slave which then apply the operations in its local state.

The state transfer model is more robust against message lost because as long as a latter more updated message arrives, the replica still be able to advance to the latest state.

Even in state transfer mode, we don't want to send the full object for updating other replicas because changes typically happens within a small portion of the object. In will be a waste of network bandwidth if we send the unchanged portion of the object, so we need a mechanism to detect and send just the delta (the portion that has been changed). One common approach is break the object into chunks and compute a hash tree of the object.

So the replica can just compare their hash tree to figure out which chunk of the object has been changed and only send those over. In operation transfer mode, usually much less data need to be send over the network. However, it requires a reliable message mechanism with delivery order guarantee.

Multi-Master (or No Master) Model

If there is hot spots in certain key range, and there is intensive write request, the master slave model will be unable to spread the workload evenly. Multi-master model allows updates to happen at any replica (I think call it "No-Master" is more accurate).

If any client can issue any update to any server, how do we synchronize the states such that we can retain client consistency and also eventually every replica will get to the same state? We describe a number of different approaches in following ...

Quorum Based 2PC

To provide "strict consistency", we can use a traditional 2PC protocol to bring all replicas to the same state at every update. Lets say there is N replicas for a data. When the data is update, there is a "prepare" phase where the coordinator ask every replica to confirm whether each of them is ready to perform the update. Each of the replica will then write the data to a log file and when success, respond to the coordinator.

After gathering all replicas responses positively, the coordinator will initiate the second "commit" phase and then ask every replicas to commit and each replica then write another log entry to confirm the update. Notice that there are some scalability issue as the coordinator need to "synchronously" wait for quite a lot of back and forth network round trip and disk I/O to complete.

On the other hand, if any one of the replica crashes, the update will be unsuccessful. As there are more replicas, chance of having one of them increases. Therefore, replication is hurting the availability rather than helping. This make traditional 2PC not a popular choice for high throughput transactional system.

A more efficient way is to use the quorum based 2PC (e.g. PAXOS). In this model, the coordinator only need to update W replicas (rather than all N replicas) synchronously. The coordinator still write to all the N replicas but only wait for positive acknowledgment for any W of the N to confirm. This is much more efficient from a probabilistic standpoint. However, since no all replicas are update, we need to be careful when reading the data to make sure the read can reach at least one replica that has been previously updated successful. When reading the data, we need to read R replicas and return the one with the latest timestamp.

For "strict consistency", the important condition is to make sure the read set and the write set overlap. ie: $W + R > N$

As you can see, the quorum based 2PC can be considered as a general 2PC protocol where the traditional 2PC is a special case where $W = N$ and $R = 1$. The general quorum-based model allow us to pick W and R according to our tradeoff decisions between read and write workload ratio.

If the user cannot afford to pick W, R large enough, ie: W + R <= N, then the client is relaxing its consistency model to a weaker one.

If the client can tolerate a more relax consistency model, we don't need to use the 2PC commit or quorum based protocol as above. Here we describe a Gossip model where updates are propagate asynchronous via gossip message exchanges and an auto-entropy protocol to apply the update such that every replica eventually get to the latest state.

Vector Clock

Vector Clock is a timestamp mechanism such that we can reason about causal relationship between updates. First of all, each replica keeps vector clock. Lets say replica i has its clock Vi. Vi[i] is the logical clock which if every replica follows certain rules to update its vector clock.

• Whenever an internal operation happens at replica i, it will advance its clock Vi[i]

• Whenever replica i send a message to replica j, it will first advance its clock Vi[i] and attach its vector clock Vi to the message

• Whenever replica j receive a message from replica i, it will first advance its clock Vj[j] and then merge its clock with the clock Vm attached in the message. ie: Vj[k] = max(Vj[k], Vm[k])

A partial order relationship can be defined such that Vi > Vj iff for all k, Vi[k] >= Vj[k]. We can use these partial ordering to derive causal relationship between updates. The reasoning behind is

• The effect of an internal operation will be seen immediately at the same node

• After receiving a message, the receiving node knows the situation of the sending node at the time when the message is send. The situation is not only including what is happening at the sending node, but also all the other nodes that the sending node knows about.

• In other words, Vi[i] reflects the time of the latest internal operation happens at node i. Vi[k] = 6 reflects replica i has known the situation of replica k up to its logical clock 6.

Notice that the term "situation" is used here in an abstract sense. Depends on what information is passed in the message, the situation can be different. This will affect how the vector clock will be advanced. In below, we describe the "state transfer model" and the "operation transfer model" which has different information passed in the message and the advancement of their vector clock will also be different.

Because state is always flow from the replica to the client but not the other way round, the client doesn't have an entry in the Vector clock. The vector clock contains only one entry for each replica. However, the client will also keep a vector clock from the last replica it contacts. This is important for support the client consistency model we describe above. For example, to support monotonic read, the replica will make sure the vector clock attached to the data is > the client's submitted vector clock in the request.

Gossip (State Transfer Model)

In a state transfer model, each replica maintain a vector clock as well as a state version tree where each state is neither > or < among each other (based on vector clock comparison). In other words, the state version tree contains all the conflicting updates.

At query time, the client will attach its vector clock and the replica will send back a subset of the state tree which precedes the client's vector clock (this will provide monotonic read consistency). The client will then advance its vector clock by merging all the versions. This means the client is responsible to resolve the conflict of all these versions because when the client sends the update later, its vector clock will precede all these versions.

At update, the client will send its vector clock and the replica will check whether the client state precedes any of its existing version, if so, it will throw away the client's update.

Replicas also gossip among each other in the background and try to merge their version tree together.

Gossip (Operation Transfer Model)

In an operation transfer approach, the sequence of applying the operations is very important. At the minimum causal order need to be maintained. Because of the ordering issue, each replica has to defer executing the operation until all the preceding operations has been executed. Therefore replicas save the operation request to a log file and exchange the log among each other and consolidate these operation logs to figure out the right sequence to apply the operations to their local store in an appropriate order.

"Causal order" means every replica will apply changes to the "causes" before apply changes to the "effect". "Total order" requires that every replica applies the operation in the same sequence.

In this model, each replica keeps a list of vector clock, Vi is the vector clock the replica itself and Vj is the vector clock when replica i receive replica j's gossip message. There is also a V-state that represent the vector clock of the last updated state.

When a query is submitted by the client, it will also send along its vector clock which reflect the client's view of the world. The replica will check if it has a view of the state that is later than the client's view.

When an update operation is received, the replica will buffer the update operation until it can be applied to the local state. Every submitted operation will be tag with 2 timestamp, V-client indicates the client's view when he is making the update request. V-@receive is the replica's view when it receives the submission.

This update operation request will be sitting in the queue until the replica has received all the other updates that this one depends on. This condition is reflected in the vector clock Vi when it is larger than V-client

On the background, different replicas exchange their log for the queued updates and update each other's vector clock. After the log exchange, each replica will check whether certain operation can be applied (when all the dependent operation has been received) and apply them accordingly. Notice that it is possible that multiple operations are ready for applying at the same time, the replica will sort these operation in causal order (by using the Vector clock comparison) and apply them in the right order.

The concurrent update problem at different replica can also happen. Which means there can be multiple valid sequences of operation. In order for different replica to apply concurrent update in the same order, we need a total ordering mechanism.

One approach is whoever do the update first acquire a monotonic sequence number and late comers follow the sequence. On the other hand, if the operation itself is commutative, then the order to apply the operations doesn't matter

After applying the update, the update operation cannot be immediately removed from the queue because the update may not be fully exchange to every replica yet. We continuously check the Vector clock of each replicas after log exchange and after we confirm than everyone has receive this update, then we'll remove it from the queue.

Map Reduce Execution

Notice that the distributed store architecture fits well into distributed processing as well.

For example, to process a Map/Reduce operation over an input key list.

The system will push the map and reduce function to all the nodes (ie: moving the processing logic towards the data). The map function of the input keys will be distributed among the replicas of owning those input, and then forward the map output to the reduce function, where the aggregation logic will be executed.

Handling Deletes

In a multi-master replication system, we use Vector clock timestamp to determine causal order, we need to handle "delete" very carefully such that we don't lost the associated timestamp information of the deleted object, otherwise we cannot even reason the order of when to apply the delete.

Therefore, we typically handle delete as a special update by marking the object as "deleted" but still keep its metadata / timestamp information around. Around a long enough time that we are confident that every replica has marked this object deleted, then we garbage collected the deleted object to reclaim its space.

Storage Implementaton

One strategy is to use make the storage implementation pluggable. e.g. A local MySQL DB,
Berkeley DB, Filesystem or even a in memory Hashtable can be used as a storage mechanism.
Another strategy is to implement the storage in a highly scalable way. Here are some techniques that I learn from CouchDB and Google BigTable.
CouchDB has a MVCC model that uses a copy-on-modified approach. Any update will cause a private copy being made which in turn cause the index also need to be modified and causing the a private copy of the index as well, all the way up to the root pointer.
Notice that the update happens in an append-only mode where the modified data is appended to the file and the old data becomes garbage. Periodic garbage collection is done to compact the data. Here is how the model is implemented in memory and disks
In Google BigTable model, the data is broken down into multiple generations and the memory is use to hold the newest generation. Any query will search the mem data as well as all the data sets on disks and merge all the return results. Fast detection of whether a generation contains a key can be done by checking a bloom filter.
When update happens, both the mem data and the commit log will be written so that if the machine crashes before the mem data flush to disk, it can be recovered from the commit log.