LOVE FOR PROGRAMMING

Distributed Systems Part-3: Managing Anti-Entropy using Merkle Trees

Pawan Bhadauria

In this post, we will look at how to effectively manage anti-entropy in your distributed system. Though a little oblique, this is an important topic. We will take forward, the system developed in last two series, fixing problems which might be lingering in production. I strongly recommend, that you read Distributed Systems Part-1: A peek into consistent hashing! & Distributed Systems Part-2: Consistency versus Availability, A Pragmatic Example! of this series. So here we go.

As we saw in Part-2, distributed systems provide great scalability, fault tolerance & efficiency. With these benefits, they also bring in different set of challenges. Recall, how we had to decide between 'availability' and 'consistency' in last part (CAP Theorem). While choosing availability, we settled for "eventual consistency", implementing an asynchronous mechanism to replicate data in background. After updating data on local storage, node will hand it over to the background process which will synchronize the data across replicas. The node is free to return a successful response to the user without worrying about replication. Using this, your load balanced & fault tolerant distributed system provided eventual consistency with stunning speed.

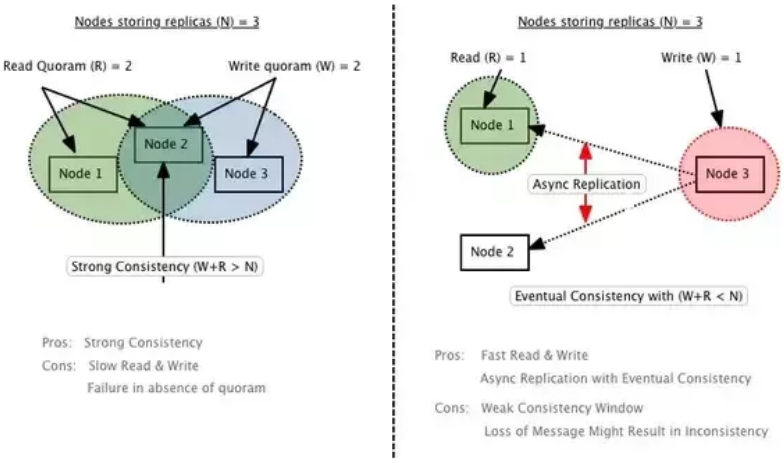
Your manager is quite happy with you. Your site has been growing like crazy and has reached to 50 millions active user now. Everyone is impressed by the distributed backend system you have put in place. You feel both humbled and excited. You have developed a great camaraderie with your manager and there is always "peace" now. One morning, you are having a conversation with your manager and one issue catches your attention. Your manager tells you that he has received few complaints where in some customer received less item then they had originally ordered. He fears that this might be related to that inconsistent window before system eventually becomes consistent. Since he had made a decision to live with it in rare case, its ok. Customers were sent missing item(s) along with extra compensation and issue was resolved. There were very few cases which came across and thus he didn't even bring up this issue. He is too happy to point finger at you. You feel that there is something fishy here. So rather than using it as a tradeoff for eventual consistency, you venture on to investigate the issue.

Digging through the logs, you find something peculiar. You see that after writing data to local storage, Node '3' handed the data to background process for replicating to Node '2' and Node '1'. It turns out, that Node '2' & '1' however, didn't receive the data. Meanwhile, Node '3' went down. When order delivery system invoked the shopping cart API, the request was served using data on Node '2' which had stale shopping cart information with one less item. The order delivery system, dispatched the order to customer with one less item. Though rare, but this looks like a serious issue. You have to fix it.

You start thinking about it. How can this possibly go wrong? Well, it seems like the background replication process, lets call it "Harvester" for the lack of better term, cannot guarantee data replication in certain exceptional cases. That's right but conversely it has worked for 99% of use cases as well. Since the issue is not

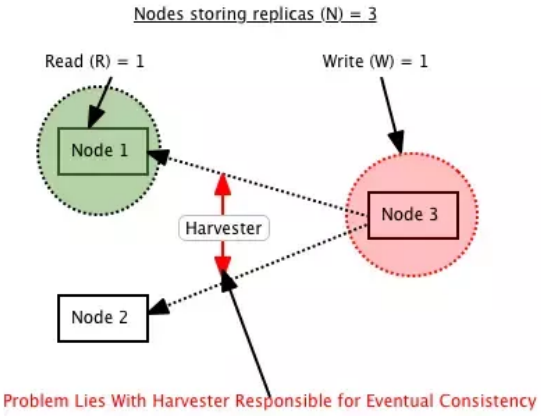
exceptional scenarios, there are a few more use cases to think about. Time to get a paper and pen.

N = Number of replicas which store data
W = Number of node which need to acknowledge update
R = Number of node contacted for read



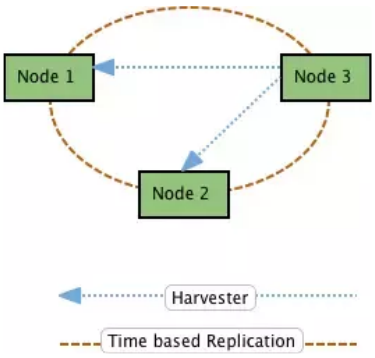
You look at the above diagram and are tempted to use the first strong consistency model. But you realize that, quorum write had in fact, created a problem for you earlier in Distributed Systems Part-2: Consistency versus Availability, A Pragmatic Example!. The second part of the diagram represents your current model. You look at it carefully. Your asynchronous replication process or 'Harvester', doesn't do a retry if it fails to write data to replicas. So due to communication or message failure, the replicas will not converge to identical copies, which might result in inconsistency.

One idea pops up. You can probably read data from all three replicas & return the most recent one. This in turn will help you identify nodes which contain inconsistent data. Subsequently, you can then do a 'read repair' on nodes which contain inconsistent data to help converge all replicas to identical copies. This looks like a good idea & can be done on the fly. But frankly speaking, your 'Harvester' is actually meant to do exactly the same thing in background. You think that rather then reducing the efficiency of reads for stronger consistency, you should instead fix the issues to guarantee fast replication. Also you see another problem with 'read repair'. If the data is not read for long time, it might eventually lead to phenomenon called Bit rot. You decide not to venture in that direction for now. Lets concentrate on fixing 'Harvester'.



Back to the problem. In your case, the problem doesn't appear as much about the time period of inconsistent window but more about guarantee of replication itself. While you are pondering about this problem, there is another issue which pops up. What if

now, you were explicitly remapping all the keys, which the node is supposed to have, but that is both error prone and very inefficient. Looks like you need two separate processes for replicating data. One which will replicate data on nodes during write, 'write sync' [contrast to read repair] or 'Harvester'. The other should be capable of comparing and synchronizing entire data set between two nodes. It can take care of efficiently synchronizing new node or an old node which came back online after a brief downtime. The latter can also work as a fallback in case your 'Harvester' fails to replicate data.



There is only one issue. Comparing data on two nodes could be a costly affair, if you explicitly compare each key value pair. Most of the time, there will be few differences between node data sets & comparing each key value is definitely an overkill. You remember what your nerdy friend, who works at a fast growing internet consumer company, had told you. He had talked about something called 'Merkle Trees'. You do some googling. As it turns out, Merkle Trees are exactly what you are looking for.

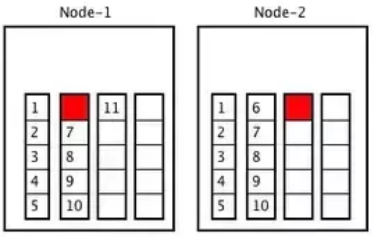
*"A Hash Tree or Merkle Tree is a tree in which every non-leaf node is labelled with the hash of the labels of its children nodes. Hash trees are useful because they allow efficient and secure verification of the contents of larger data structures. Hash trees can be used to verify any kind of data stored, handled and transferred in and between computers." [**Wikipedia**]*

Merkle Trees are amazing. Lets take a closer look at it, with a simple example. Lets say there are two nodes containing following data set:

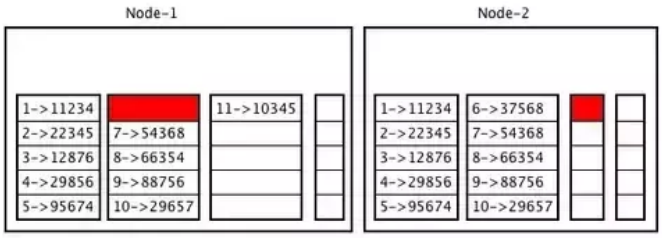
Node-1: 1,2,3,4,5,7,8,9,10,11
Node-2: 1,2,3,4,5,6,7,8,9,10

Now you want to compare content on these two nodes using Merkle tree. Also, if some modifications happen, you need a simple way to find what changed & where. For simplicity, lets say your key space is from 1-20. You divide the entire key space into four segments (five keys each) and distribute keys to these segments. Once the segment or key bucket is created, you hash each key in the segment using a uniform hashing scheme. Now using these key hashes, you create a single hash per segment. Then you traverse upwards till root, calculating hashes for a node group. Below diagram illustrates this step by step. [The box with red color indicates the missing key]

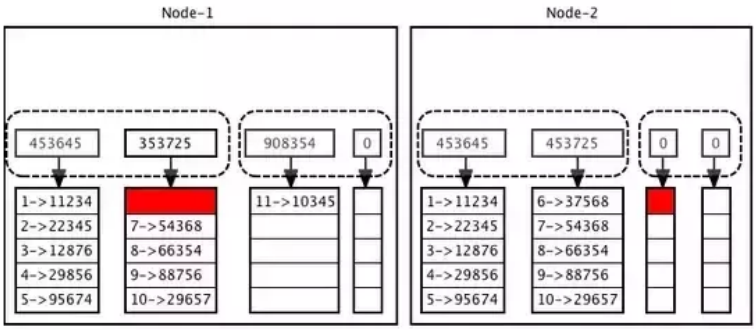
Step-1: Create segments for storing a set of keys



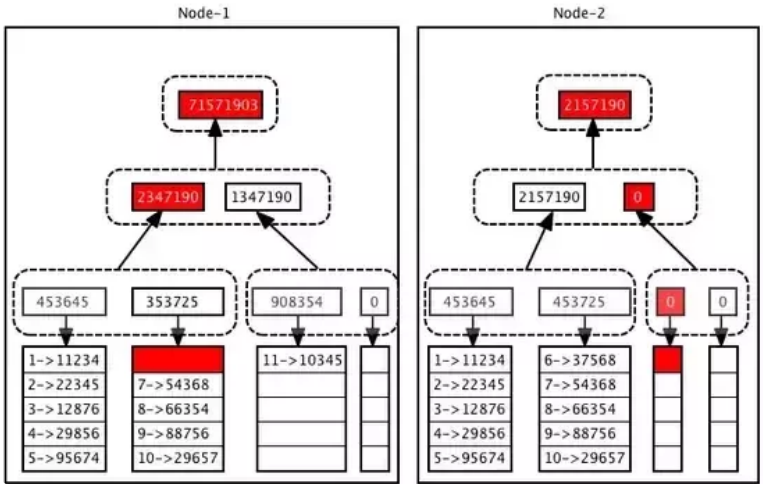
Step-2: Hash keys with a uniform hashing scheme



Step-2: Create hash for each segment



Step-2: Create a hash for a group of nodes and repeat the process till root. Here we have two nodes constituting a group.

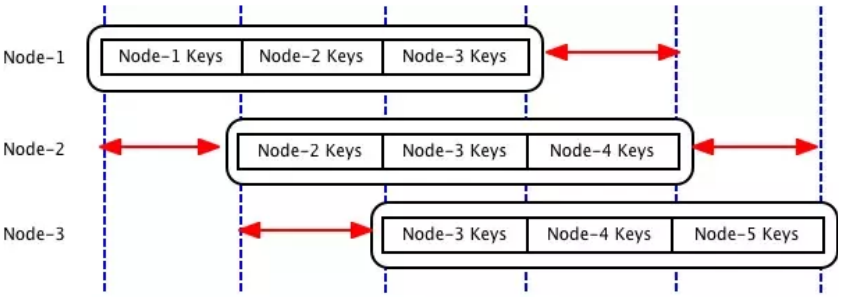


As you can clearly see in the above diagram, if you have a Merkle Trees for two nodes, you can find if the data on these two nodes is same or not, by just comparing root node. If the root hash is same, both nodes are guaranteed to have same content. If not, then you can traverse down and repeat the process of comparing child node hashes to find which parts of tree are different. This helps you reach to a particular segment(s), which is/are

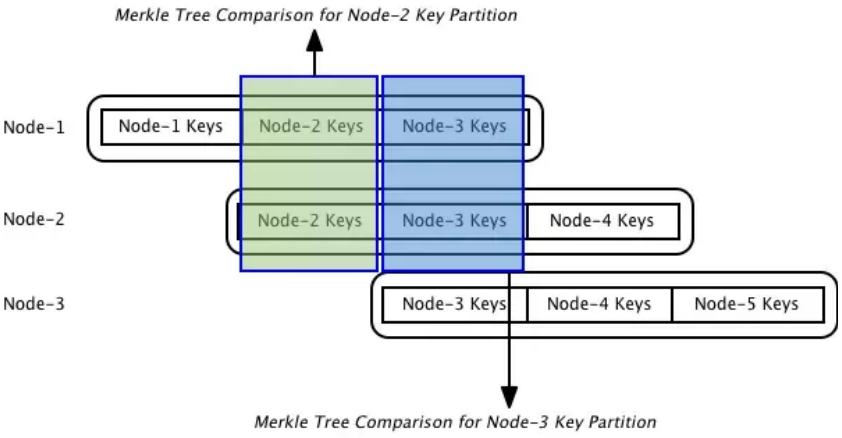
synchronization will be proportional to the differences between two replicas, not the amount of data that they contain. *This should provide amazing efficiency while synchronizing two nodes.*

This looks really interesting to you so you get your hands dirty & implement first version of Merkle tree. You divide each node's key space into few thousand segments and create a Merkle Tree based on its content. You create an anti entropy manager which runs every minute & synchronizes the data between nodes that store replicas. This is in addition to replication already done on predecessor nodes while updating data (by 'Harvester'). Together, they provide great eventual consistency & fault tolerance to your distributed system. This looks really awesome!!

You are excited and discuss the problem and potential solution with your manager. Your manager agrees with most part but points a fault in implementation. Since you have a replication factor of 3, each node in consistent hashing ring will contain data for two other nodes. So if you create a Merkle tree for entire data set & compare two nodes, they will always be different. Below diagram illustrates this. The red arrows indicate the difference between each node.



You look at the diagram and praise him for catching this. You feel that its wise to create and compare Merkle Trees specific to node key partitions only. So looks like you need to create a separate Merkle Tree for each node key partition. Once done, you can than compare Node-1 and Node-2 by only comparing Merkle Trees for partitions containing Node-2 keys and Node-3 keys as shown below. Similar process can be done for other nodes as well.



You change your implementation and create separate Merkle Trees for every node key partition on each node. This helps you separately compare node key partition data which is common on any two nodes. Since your key space per node is not very large, you keep these trees in memory on each node to boost performance. If this becomes a bottleneck, you can think about persisting it to disk later. You anti-entropy manager acts as a facilitator in synchronizing data between nodes using Merkle Trees. This looks amazing!!

using Merkle Trees via anti-entropy manager. *Since it only deals with diffs now, the synchronization is super fast*. A new node also uses this same mechanism but has to remap all keys to start with but thats expected. You have really come a long way.

Now you have a distributed system which is scalable, load balanced and truly fault tolerant. It also provides amazing eventual consistency by using dual mode replication; one through instant 'write sync' and other periodic synchronization using Merkle Trees. You test your implementation and push it to production. Once the system goes in production, the dual replication makes sure, that system provides guaranteed eventual consistency with a small inconsistent time window. The client issues related to stale orders have stopped popping up. You believe in philosophy that "customer is god" & are satisfied with the outcome which results in better user experience.

Let celebrate our new improved distributed system. Party time...

[As discussed in Distributed Systems Part-2: Consistency versus Availability, A Pragmatic Example!, you feel that conflict resolution is something which might come back to haunt you but since customer have not complained about manually resolving conflicts (if they have seen it at all), it should be fine. In any case, you still have vector clocks lingering in the back of your mind.]

Note: Amazon DynamoDB, AntiEntropy - Cassandra Wiki & Riak – Basho Technologies, all implement Merkle Trees for efficiently synchronizing node states. While The Apache Cassandra Project implements in-memory Merkle Trees, Riak persists Merkle Trees which is much more robust, scalable & fault tolerant. Riak uses a million segment in its implementation of Merkle trees. So lets say, if there are a billion keys, each segment bucket will contain 1000 keys, which is pretty nice & small, for transferring data to synchronize segments across nodes.