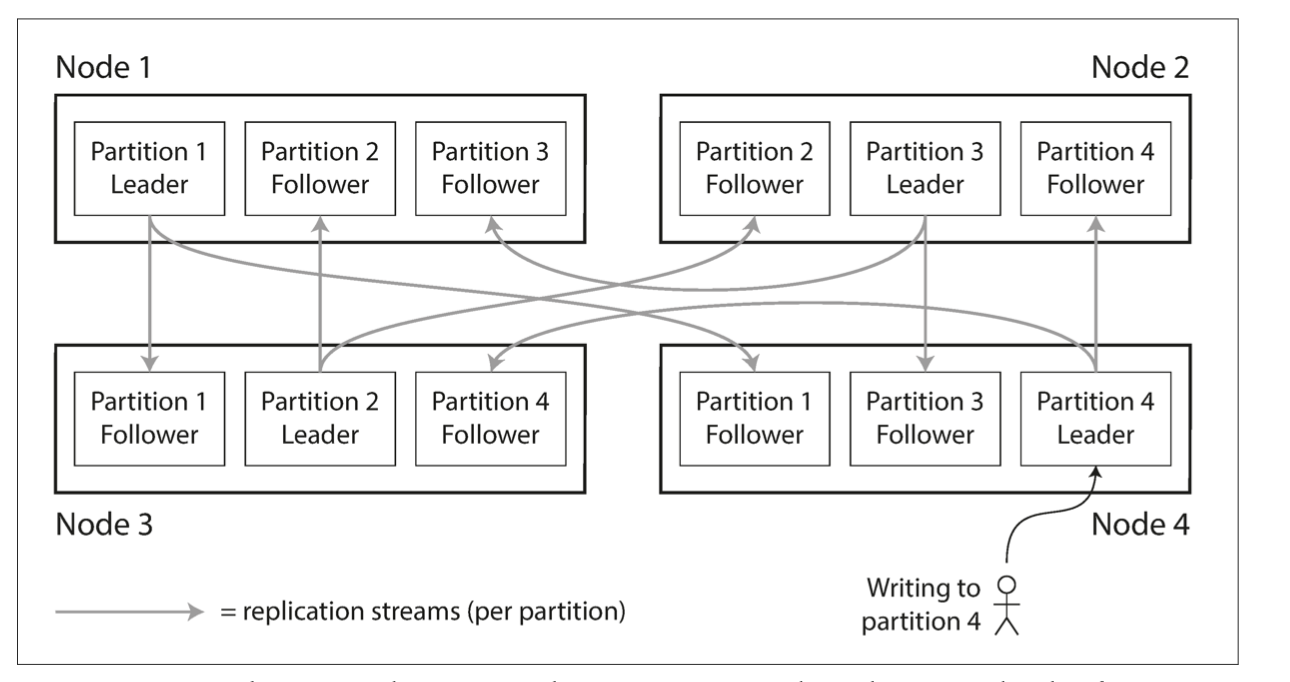
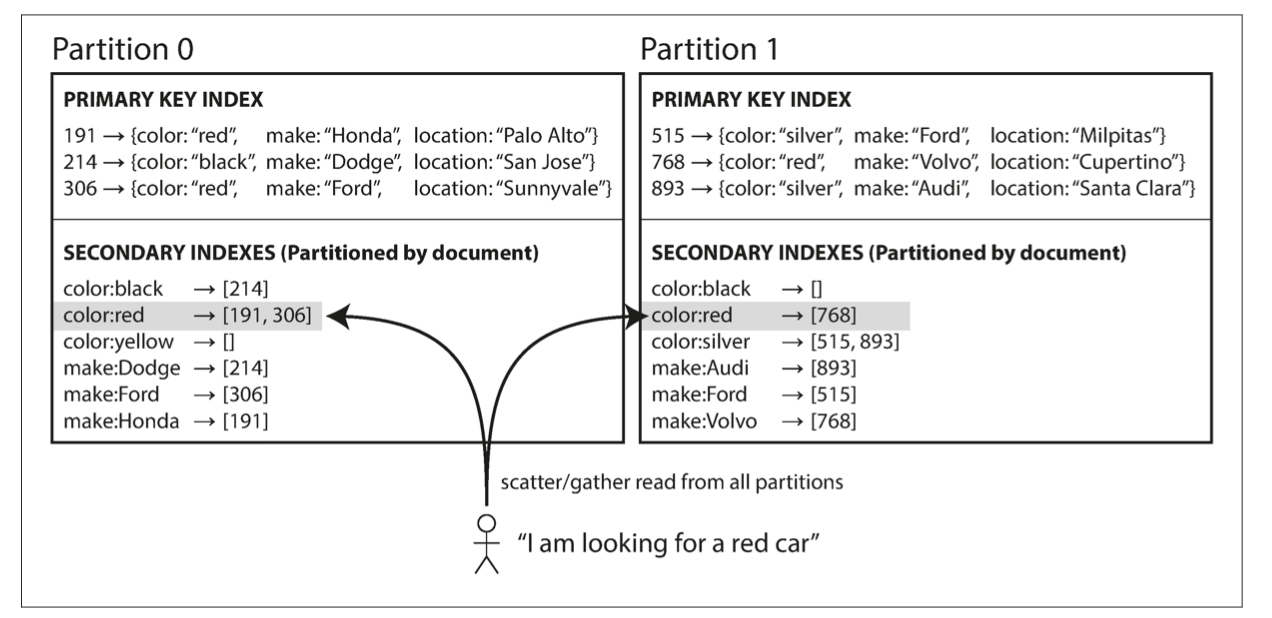
**Databases related notes**

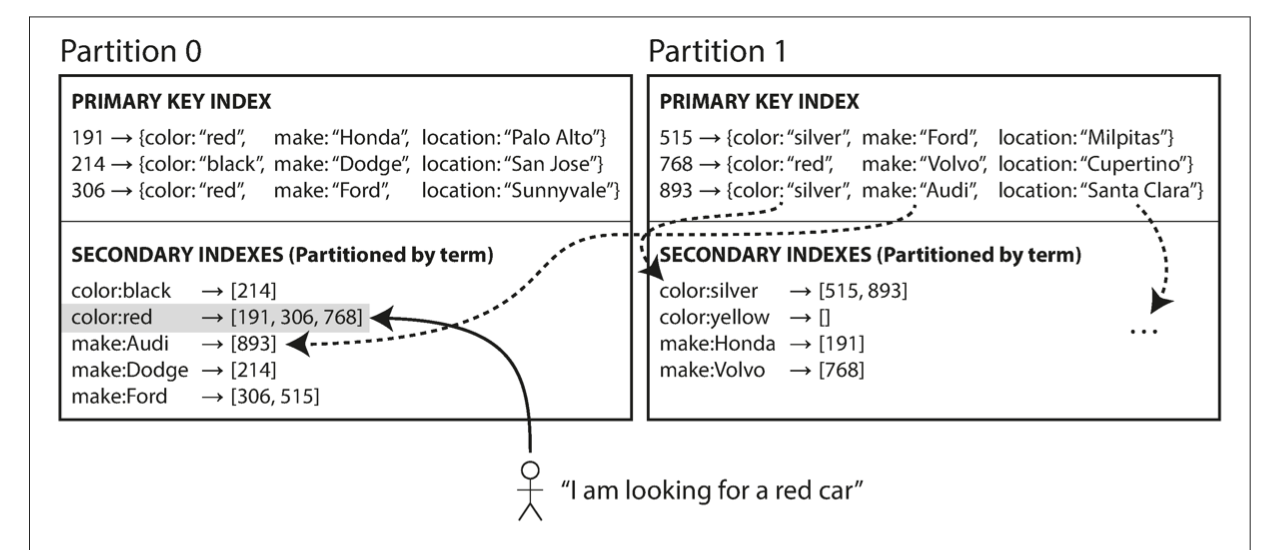
1. **Problems with scalability of Relational Databases:**
   1. If you horizontally scale across a set of machines, joins result in network request. Since SQL allows you to write complex queries, you never know where exactly data is going to be fetched from till the read time. This is not a problem with NoSQL DBs because a single request is handled by a single machine and no joins are required (Data is stored in sort of a Pre Join format so that network calls are not necessary). Since all the data that is needed by a query is guaranteed to be stored on a single node, you can split it up using partition keys.
   2. What purpose are joins serving in relational DB – In relational DB’s, data is stored in normalized form and foreign keys come into play. This has a few benefits – **storage efficiency, data integrity** (let’s say we have orders and customers information. Orders table refer to customers table with a foreign key customer\_id. If the data is not stored in the normalized form and customer’s name needs to change, all the order rows for that customer needs to change. This is not necessary in normalized form, only one row for in the customer table needs to change. Normalization means data is scattered all over the place – this is useful if data access patterns is not known in advance. You can write joins later for your data access needs. **– Flexible data access**. But these advantages become a bottleneck with scalability because joins are expensive – CPU and memory. For NoSQL DBs, Flexbile data access is not a need, you need to know in advance before designing the data model. “Pre Join” you data in a way it will be read. So no joins are needed later. Tradeoff. Data integrity is now an application level concern, let the application take care of copying the data. Another tradeoff. NoSQL DBs are less storage efficient, but not a concern because compute is more expensive than storage now
2. **WAL (Write ahead log):** More details <https://www.interdb.jp/pg/pgsql09.html>. Advantages: Less IO, instead of storing data pages to disk on every commit, just one fsync of WAL file can do. You can even batch transactions as fsync of the WAL file can save multiple transactions on the WAL log. Faster IO as well because of sequential appending to the log. Useful in recovery after OS crash. WAL is replayed from the last checkpoint.
3. **Partitioning:** Done for scalability – Data distribution across many disks and load distribution across many processes. Partitioning is done in combination with replication and looks like this:

****

* 1. **Partitioning by key range:** Assign a continuous range of keys to each partition. Uneven load distribution. BigTable and Hbase. Within a partition, keep keys in sorted order so that range scans are easy. Also leads to hotspots. For example, if the key is a timestamp, all writes end up going to the same partition for the day.
  2. **Partitioning by Hash:** Cassandra and Mongodb. Assign each partition a range of hashes. No range based scans now as adjacent keys are scattered across partitions.
  3. **Partitioning and Secondary Indexes:** Useful for searching occurrences of a particular word, for example Very important in ElasticSearch DBs. How do you partition DBs with secondary indices:
     1. **Document based partitioning:** Each partition maintains it’s own list of secondary indices. 

Reading is trouble because you have to read from all the partitions and then aggregate. Scatter gather. MongoDB, Riak, Cassandra.

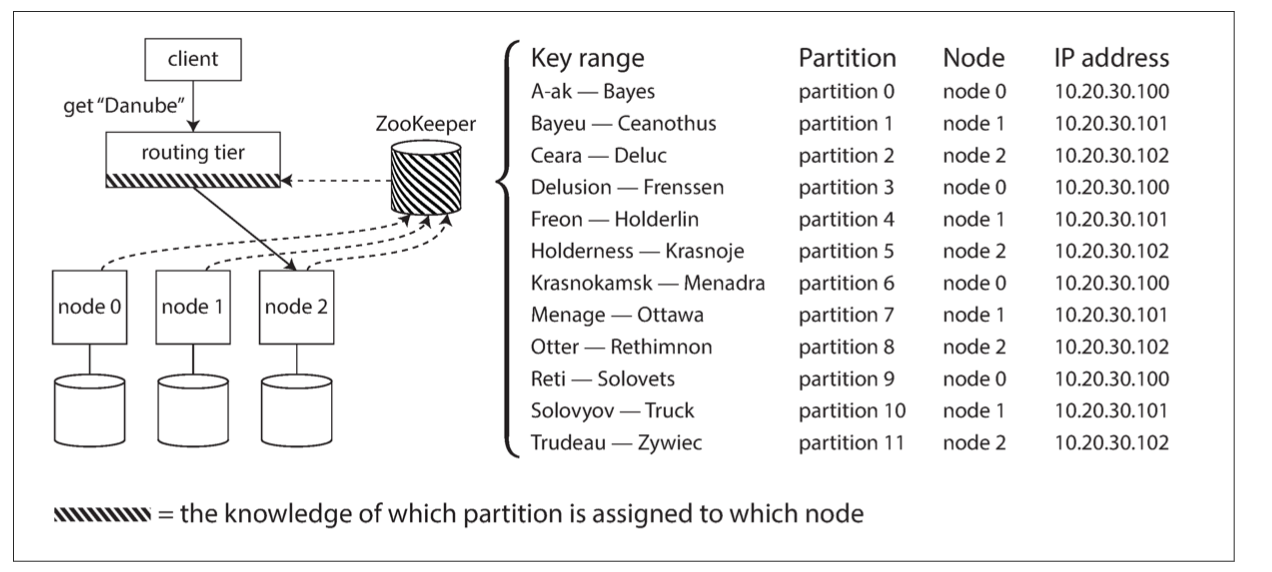
* + 1. **Partitioning Secondary Indices by Term:** Global Index. A global index also needs to be partitioned.

****

Prevents scatter gather over all partitions. Writes are slower as write of a single document or record affects multiple partitions. Updates to the secondary indices done asynchronously. Amazon DynamoDB.

* 1. **Rebalancing Partitions**. Objectives – After rebalancing, load distribution should be fair and less data should be moved. Hash Mod N is a bad partitioning strategy because a lot of data needs to be moved – expensive rebalancing. Some approaches:
     1. **Fixed number of partitions**: Just create many more partitions than the nodes and map partitions to nodes. If a node is added, move some partitions to that node. Riak and ElasticSearch use this. Operationally simpler. Choosing the right number of partitions is difficult. If the partition size is too large, rebalancing and recovery become expensive. If the partition size Is too low, high overhead. So if the total dataset size is variable, right sized partitions are hard to achieve if the number of partitions is fixed.
     2. **Dynamic Partitioning**: For range based partitioning, the fixed number of partitions scheme can be trouble if you get the boundaries wrong. Hbase used Dynamic Partitioning. When a partition exceeds a configured size (default 10 GB in Hbase), it is split into two partitions each of which can assigned to separate nodes to balance the load. Advantage – the number of partitions adapt to the dataset size.
     3. **Partitioning proportionally to the number of nodes**: key idea is the number of partitions per node. Size of the partitions grow in size as the dataset size increases and the nodes remain unchanged. As the new node gets added, it splits a fixed number of randomly chosen partitions. These partitions are known as vnodes. For example, initially there could be 2 nodes and 512 vnodes (256 vnodes per node), when a new node joins. It randomly picks 256 nodes and splits them. One way of understanding vnodes is that for each physical node you compute multiple hashes( 256, for example) and place the nodes on the ring. So whenever a new physical node is added, its positions of the vnodes split as many partitions. Advantages: Rebalancing – it receives data from a lot of nodes. Heterogenity – Some nodes can have more vnodes.
  2. **Request Routing:** Where is a particular key stored? This is an instance of service discovery problem in general.
     1. Allow the client to connet to any node and that node can either serve the request or act as the coordinator, forwards the request to the correct node, receive the reply and then forwards the reply back to the client.
     2. Send the request to a routing tier first. Partition aware load balancer.
     3. Client be aware of partitioning scheme and connect directly

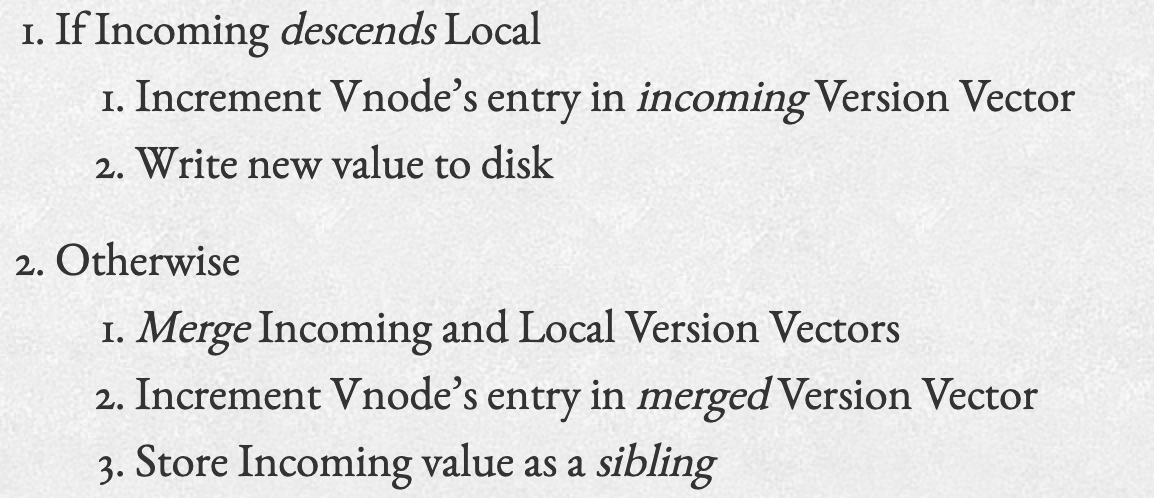
This is essentially a consensus problem and is hard. Many systems use a separate coordination service like Zookeper to track the metadata of the cluster. Nodes register with Zookeper and routing tier subscribe to this information in Zookeper. Whenever mapping changes, Zookeeper notifies the routing tier.



This approach is used by HBase, Kafka to use Zookeeper to track partition management.

Cassandra takes a different approach – gossip protocol. Approach 1 of request routing. It puts complexity in database nodes but avoids dependency on Zookeeper.

1. Primary Secondary Replication is not suitable for key-value stores if we intend to have highly available writes. That is because the writes always go to primary so are limited by the performance of primary. Morevery if Primary fails, writes are not available during the switchover time.
2. **Vector Clocks:** Based on the r and w configurations, the data may diverge in a key value store in case of concurrent writes. This is especially true if the partition has occurred. For example if there are 4 nodes and w is 2, if node A and B can’t reach C and D, then concurrent writes to A and C will be success and the data has diverged. How to resolve these conflicts? Vector clocks. It’s important to know which version of the data is being written so that it can be repaired when read later. API changes – get(key) returns the value and a context which describes the vector clock. Put(key, value, context) -> writes the value for the context returned before. Example from Riak (<https://riak.com/posts/technical/vector-clocks-revisited-part-2-dotted-version-vectors/index.html?p=9929.html>) :

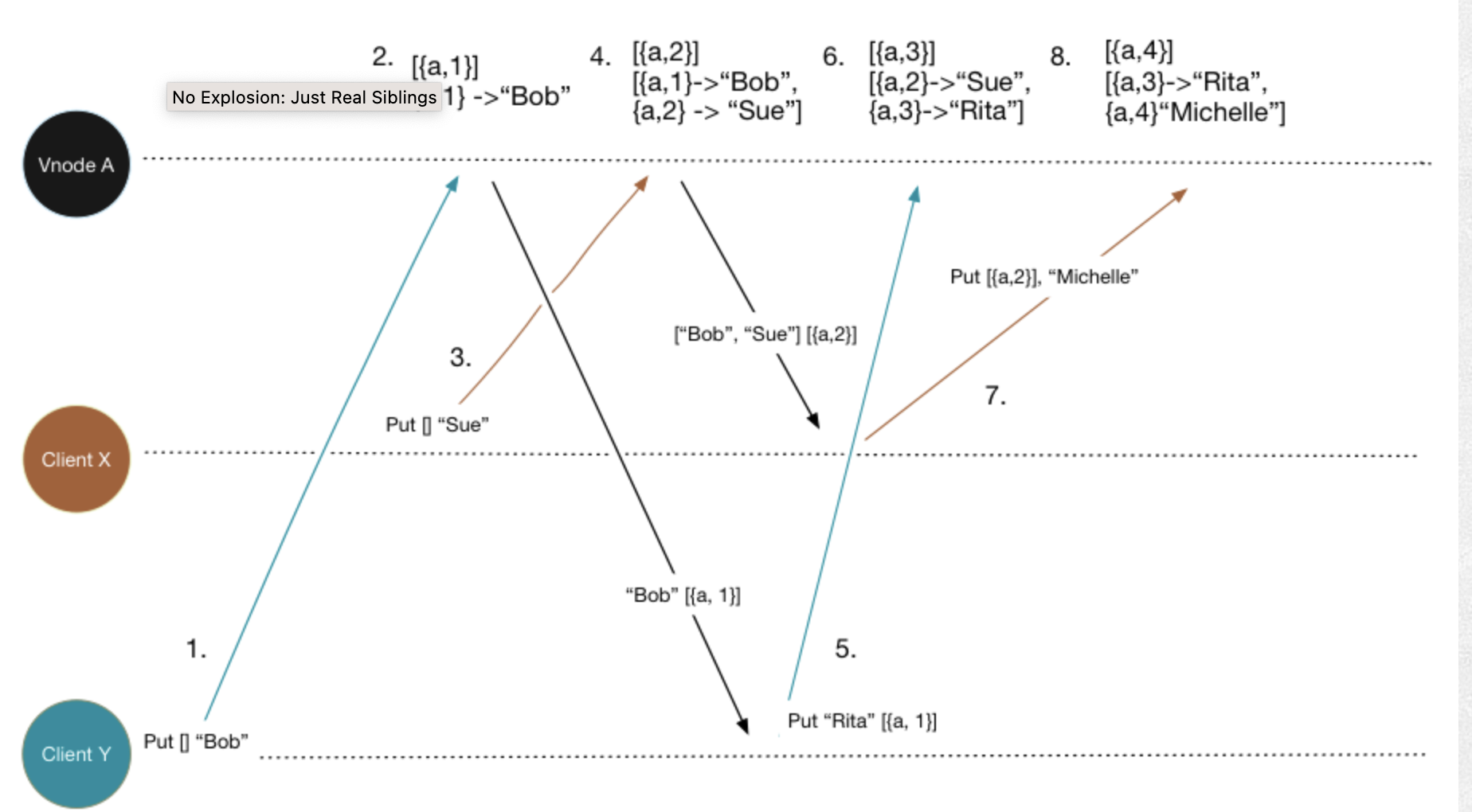
****

This works but creates sibling explosion:

**Diagram

Description automatically generated**

At step 5, Client Y already knows about Bob and [{a,1}] and it was trying to update the value. There was no need to retain Bob in the final vector at step 6. This problem is solved by the dotted version vectors. Also store the values as part of the vectors. For example, when 3 happens, Riak detects that it’s a conflict, version number needs to be merged. So the final version becomes [{a,2}] but it also stores the “casuality” as [{a,1}=>Bob, {a,2}=-Sue]. So at step 5 when Y plans to update Bob from {a,1}, it can be removed from the final version vector at step 6. This keeps the number of siblings under control



It's important to understand that while trying to update a key x, it can be first read from Node A with a specific context, let’s say, {a, 1} and the write may go B which could result in a conflict. Let’s say B has {b,1} initially. How this sequence can potentially play out:

Graphical user interface, application

Description automatically generated

So the basic idea is that you should know which version you are trying to update so that casuality can be captured in the DB at the time of the write. It can either cause a conflict or be accepted. If it causes conflict, it can either be fixed at that time itself by the user who can issue one more write or the conflicted information can be synced in the DB and later be fixed at the time of read (Read Repair).