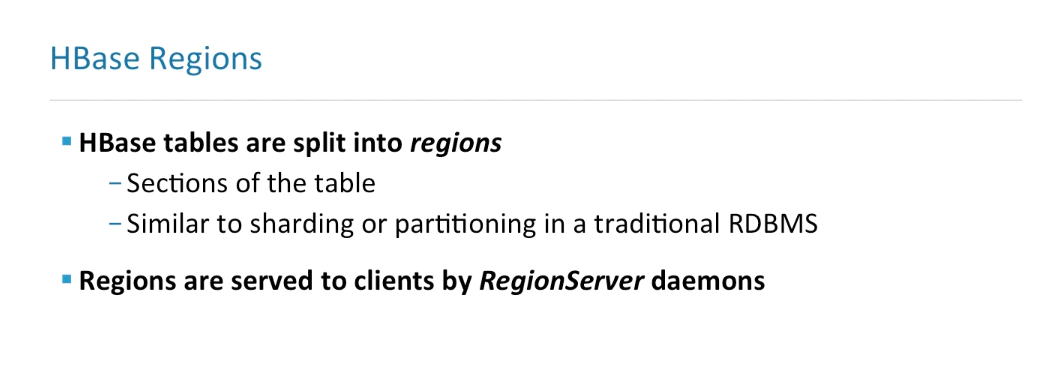
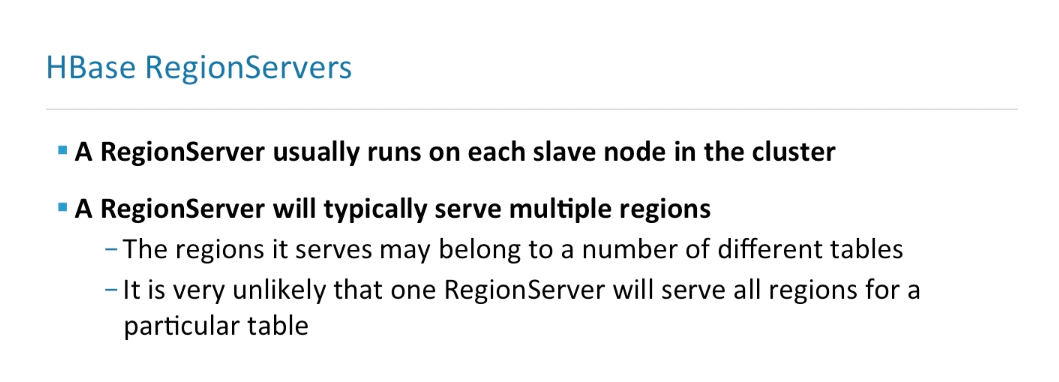
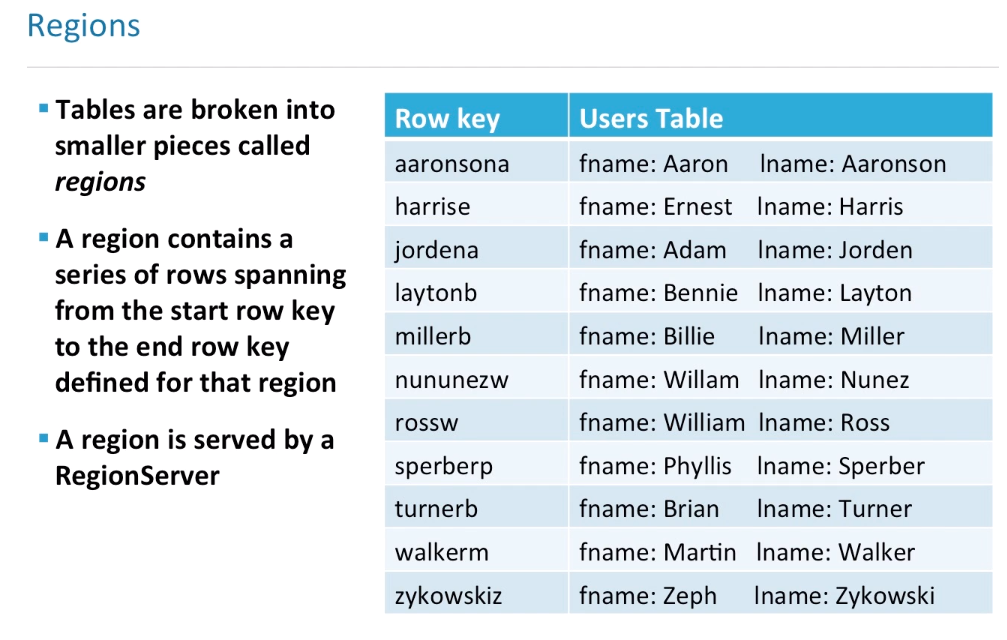


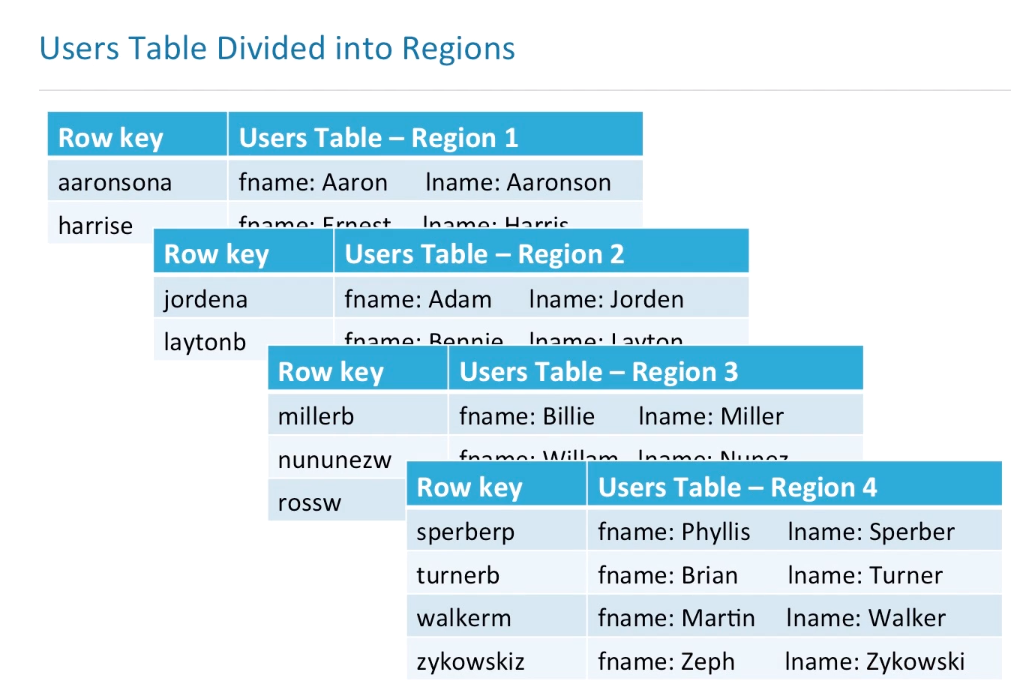
HBase is a bit different than other key value stores, like perhaps Cassandra, in that HBase stores table data as a map, that’s similar, but, **it guarantees that adjacent keys will be stored next to each other physically on disk.This can help with retrieval of information that is to be grouped together**.

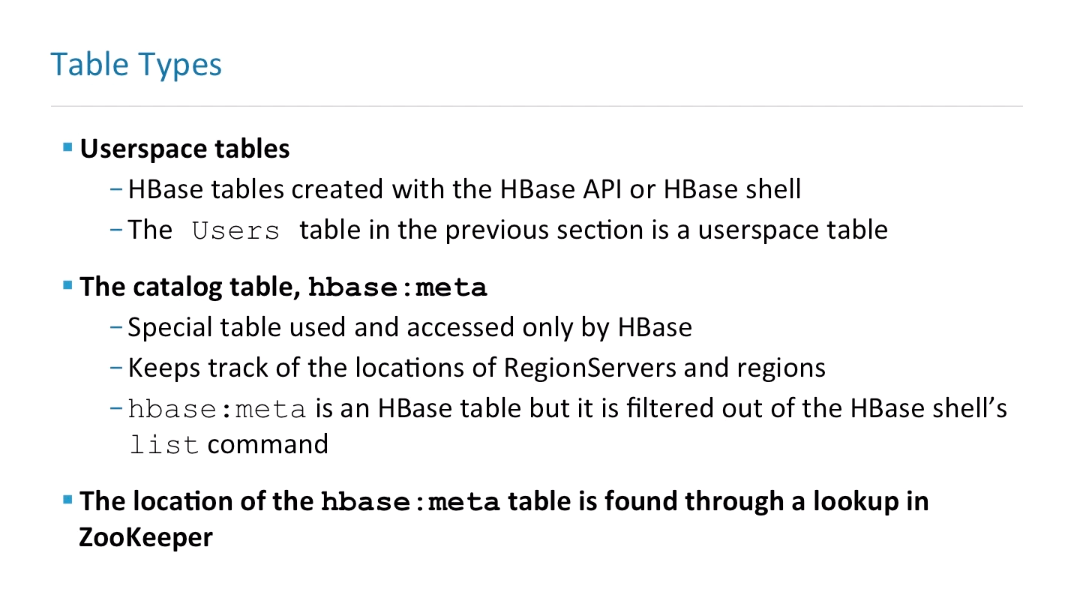
For example, if I am storing stock ticker information or quotes, I could have a key that is comprised of the stock ticker, for example IBM, and then I can have a delimiter and the next part of the key could actually be the timestamp of the quote itself, so I could retrieve the first 100 rows that begin with IBM, and I would essentially be fetching the data from one particular host, then this allows for efficient retrieval of related data.





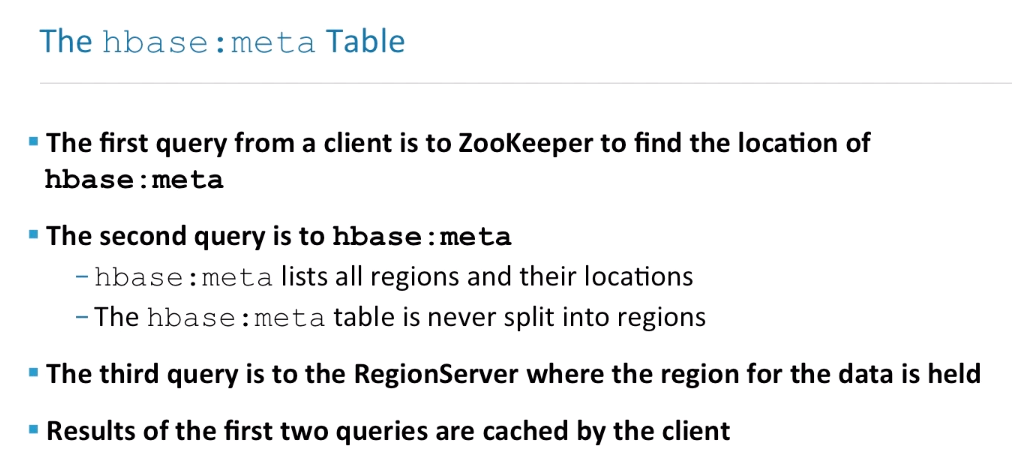
RegionServers typically serve multiple regions.It's not out of the ordinary for a particular RegionServer to be serving tens or hundreds of regions at a time of multiple tables.It's very unlikely that one RegionServer will serve all regions for a particular table and HBase has a process known as a balancer that takes into consideration the tables that the particular regions belong to, and essentially factors in the table when deciding which regions a particular server would be assigned.





There are two different table types in HBase.

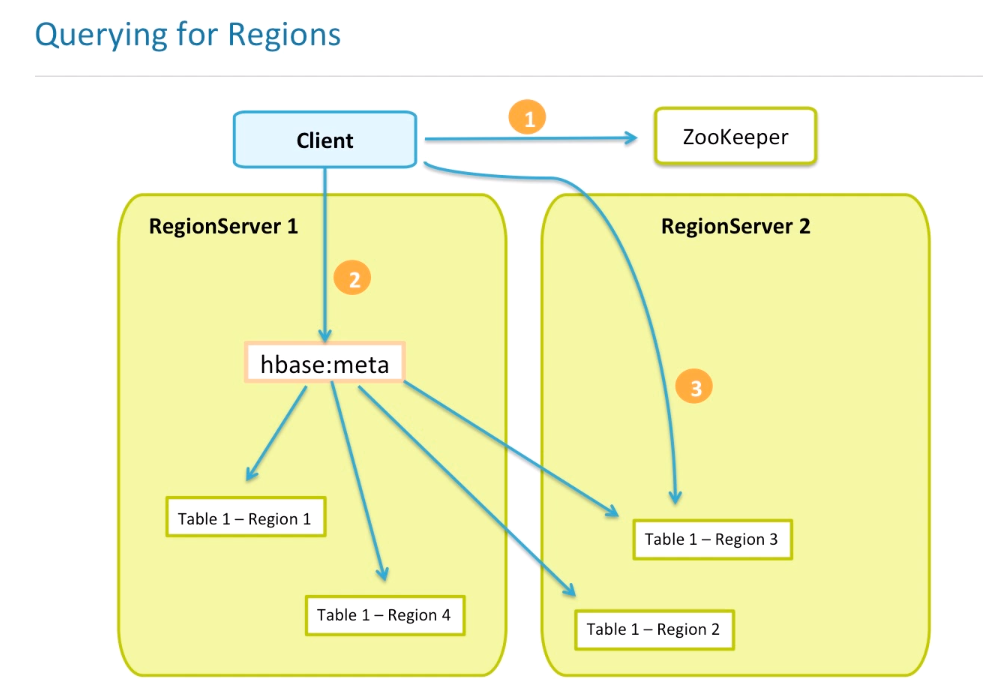
There are userspace tables which contain the actual data that the users of HBase are storing and retrieving.There are catalog tables.

**There's a table called hbase:meta**

It’s in this HBase namespace and it's called meta and this is a special table known as a catalog table that keeps track of the locations of the active RegionServers and the regions that they are serving. So this is how the HBase Master knows which RegionServers are responsible for which regions.

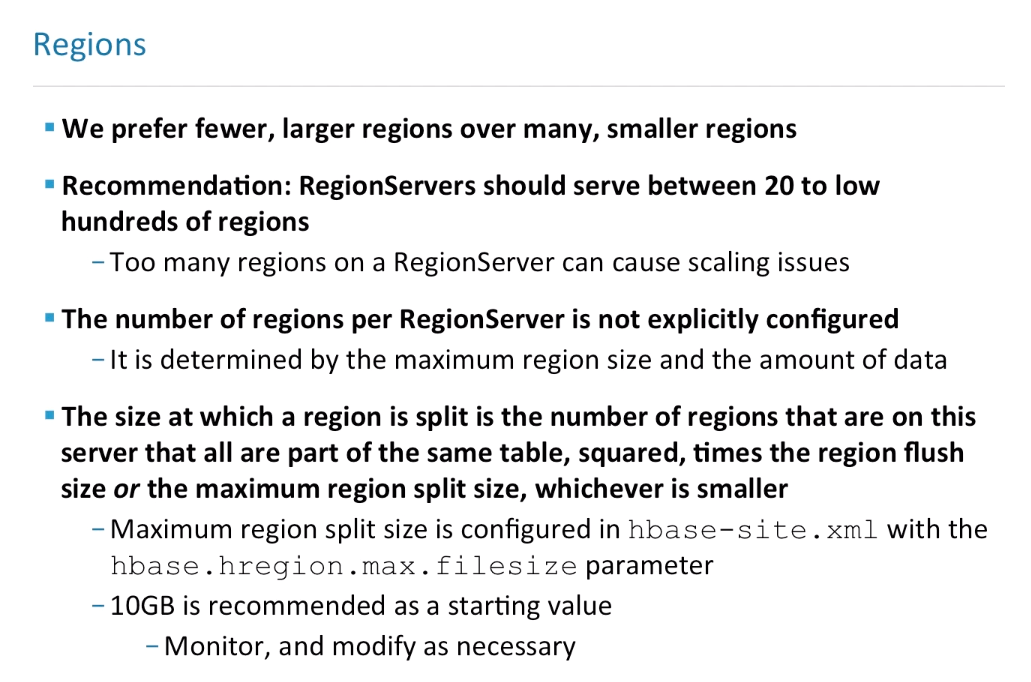
When you issue the list command in the HBase shell, it is not shown by default and in order to find the RegionServer that is serving the hbase:meta table, that server is found through a lookup in ZooKeeper. Any client that wants to access data in HBase must know who the ZooKeeper hosts are.

That's the most important setting that you need to specify when hooking up clients to talk to HBase.



This diagram shows an illustration of the steps needed to query for data in an HBase system. The first step is from the client to find out which host has the meta table. So the client asks the ZooKeeper service, "What is the RegionServer that's serving the meta table?" ZooKeeper says, "It is RegionServer 1.” The client then contacts RegionServer 1 and says, "I would like to have the contents of the hbase:meta table." The RegionServer says, "Here you go, here is the contents of the meta table." And the contents of the meta table have the fact that Table 1-Region 1 is being served by RegionServer 1.

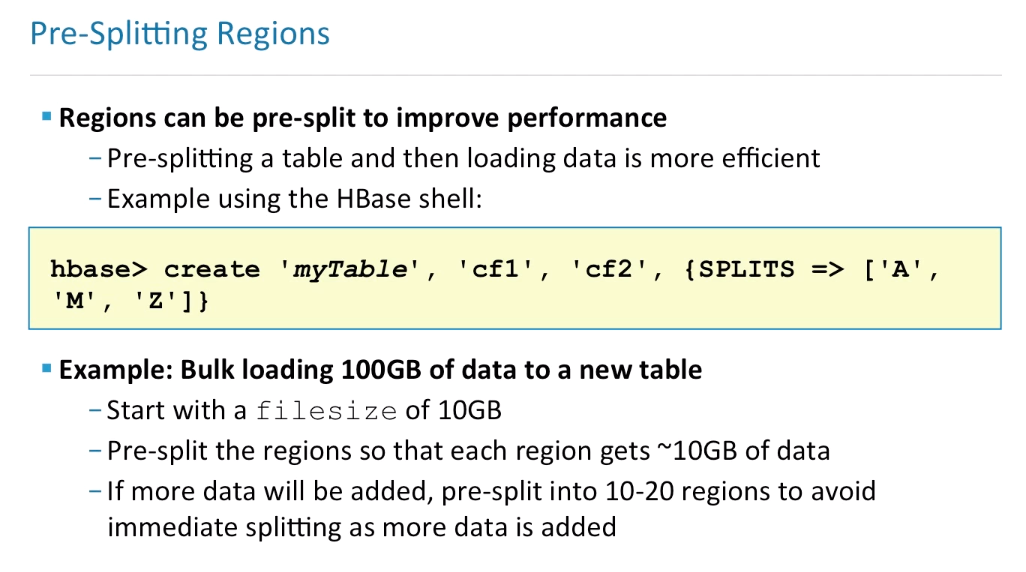
Table 1-Region 4 is also being served by RegionServer 1. Table 1-Region 3 is being served by RegionServer 2 and Table 1-Region 2 is also being served by a RegionServer 2. And each region is marked by a start key and an end key. So now the client knows that if I want to have some key that fits between the start key of Region 3 and the end key of Region 3, then it needs to contact RegionServer 2. That's the basic idea of how data is fetched and pushed to HBase efficiently by the client.

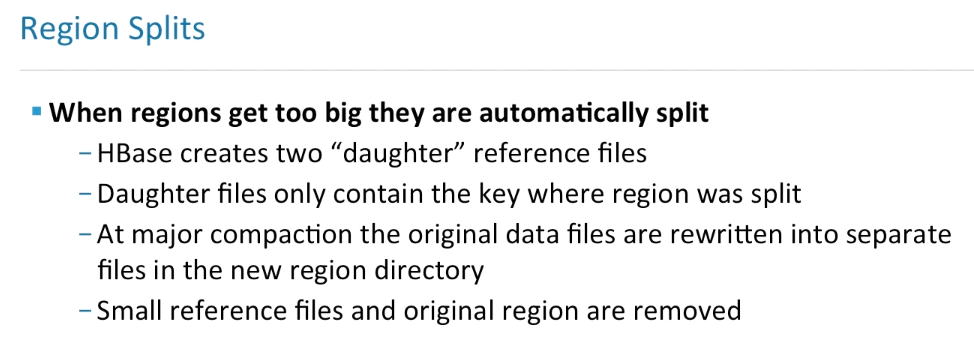


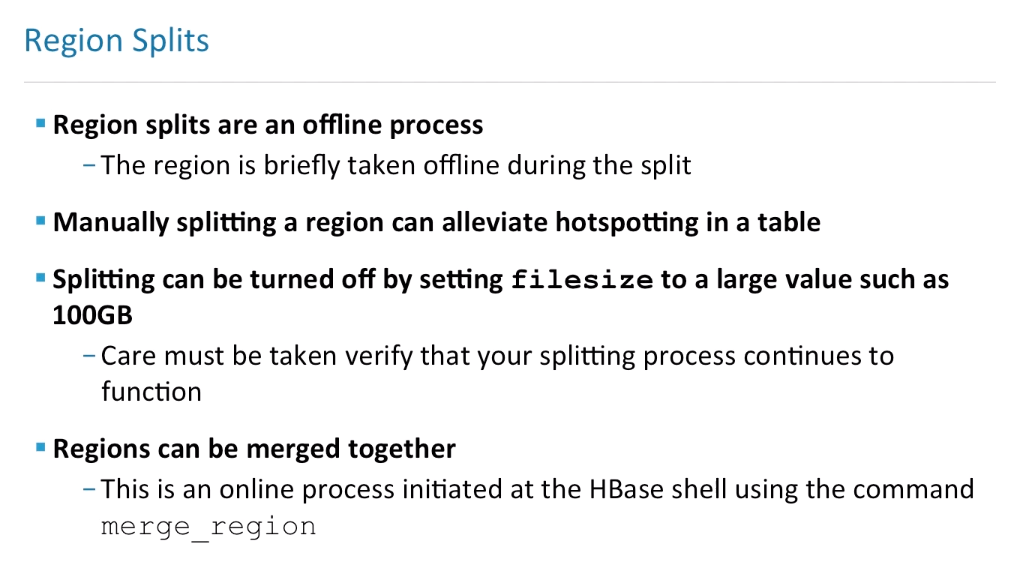
There's been many cases over HBase's history where people that are using HBase for the first time had the default configurations running with HBase, and there were way too many regions on a particular server, in the thousands or tens of thousands. Regions involve overhead.

They're basically the resource management unit for data on HBase, not tables, not necessarily column families, certainly not columns.So we recommend RegionServers should serve between 20 to the low hundreds of regions. When you start to get into the thousands of regions per server,the number of compactions, flushes, resources such as threads, open-file handles, becomes too much for a particular JVM to handle.

This can cause scaling issues. A RegionServer can become unresponsive. It can oversubscribe to the I/O on the system. All kinds of things can ensue when you have too many regions on a particular RegionServer.







When regions split, the parent region is temporarily offline.

The offline is done to make sure that the region flushes, that the data is consistent, and to update the hbase:meta table with the new daughter regions. So, you do not want to have regions splitting during high-usage times. If you see a hotspot on a table or a particular region, you can manually split. You can use the HBase shell to split the region.

We recommend turning off, or postponing, automatic splitting. Splitting can be turned off by setting a file size that's way higher than the recommendation,

so that we do not have automatic splits happen during high-volume times. The idea is to split manually during low-usage times, and then verify and monitor the HBase system to make sure that the splitting process continues to function.

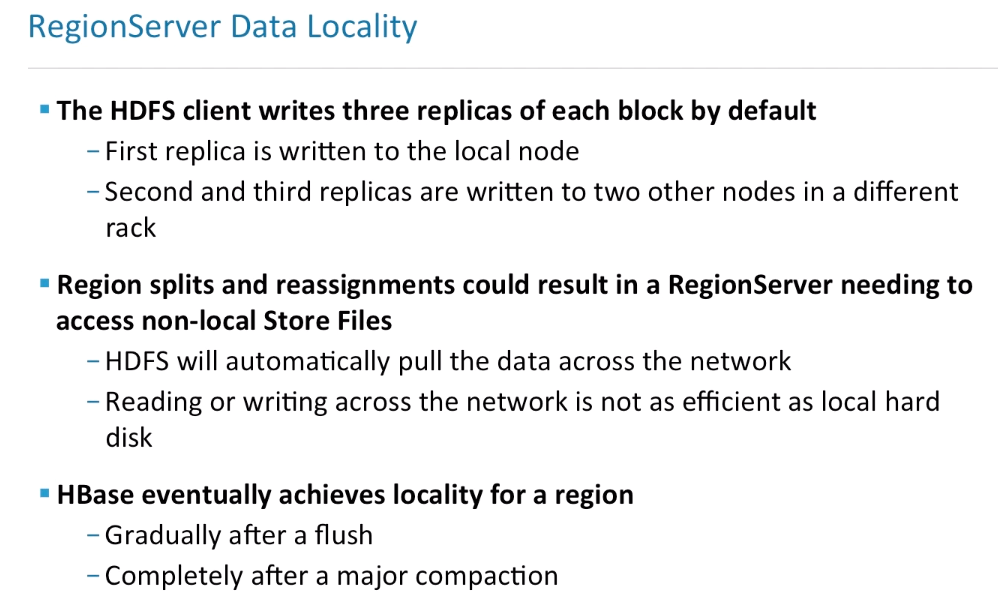


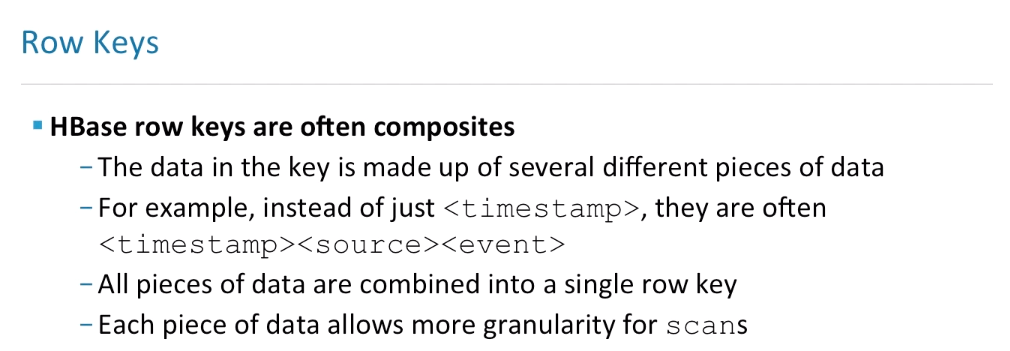
Regions are the elements of availability and distribution of your data. The more regions you have, the better the workload is, the better the workload is distributed, and the more available data is, presumably, because you'll have different servers that are serving different pieces of the table.

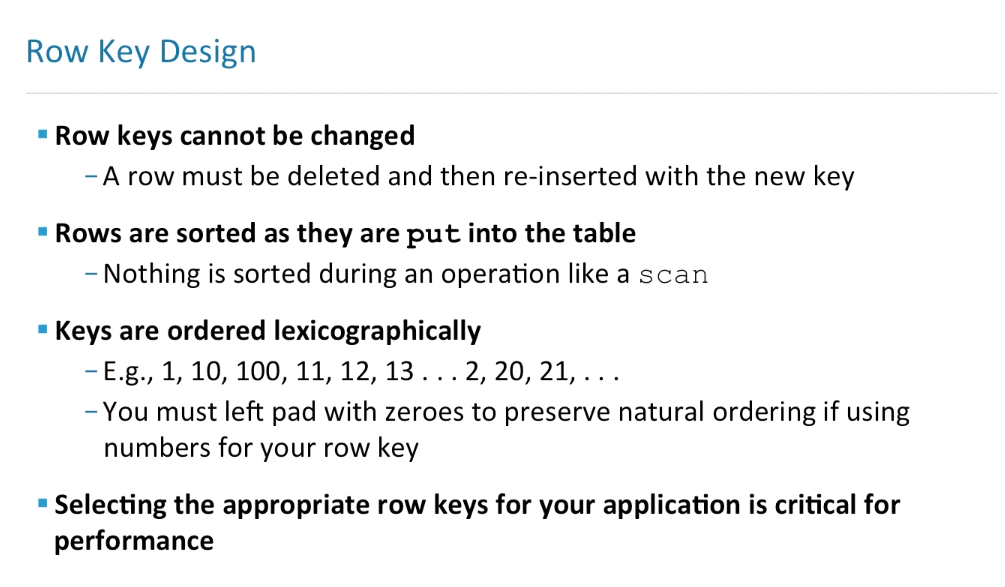
However, you don't want a high number of regions for a small amount of data. Let's say we have 10 RegionServers and we have a very small table, and it's split into 10 regions. If the data is small, then it's sort of a waste of resources, because if we're asking for record number one, we have to ask the server over here. If we ask for record number 100, then we have to ask a different server and they each have to maintain a set of open files in HDFS and they each have to seek through the data physically, anyway, in a lot of cases.

It's pretty much better to have 128 megabytes in one region than it is to have 12 megabytes in 10 regions. So, you don't necessarily want a high number of regions for a small amount of data. The flip side is if I have a gargantuan number of regions, in the thousands per server, the problem is that the RegionServers have threads that loop through the regions to see if they need to be split. They loop through the regions to make sure they do not need to be compacted.

The RegionServers have to maintain a set of files that are open for every region. So, if I have a ton of these regions, regardless of whether they are big or small, they can definitely impact HBase performance. The ballpark is that you generally want to have in the range of 20 to the low hundreds per RegionServer.

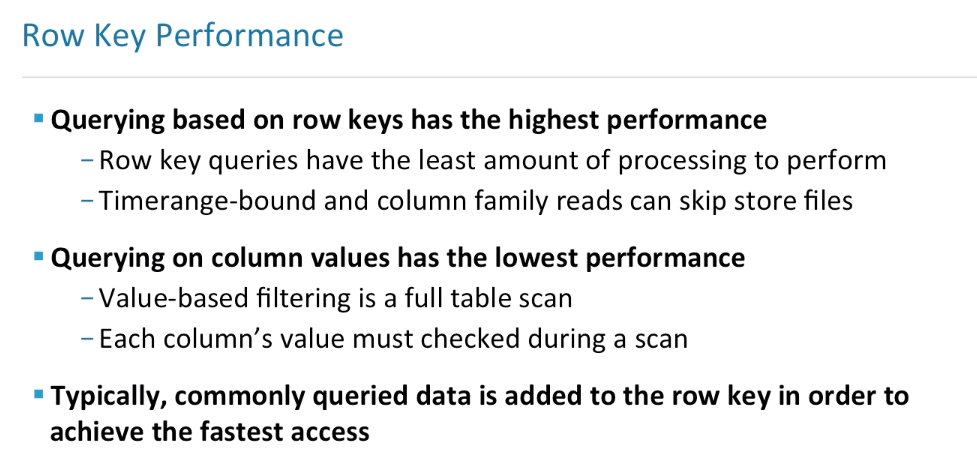


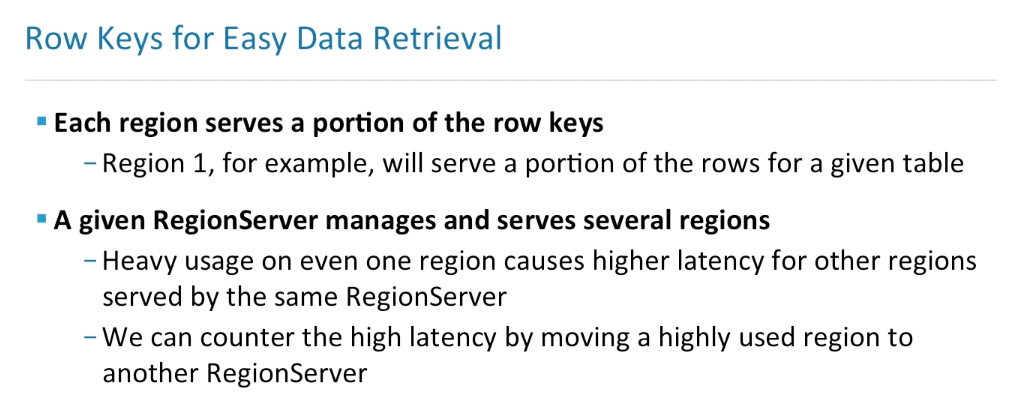


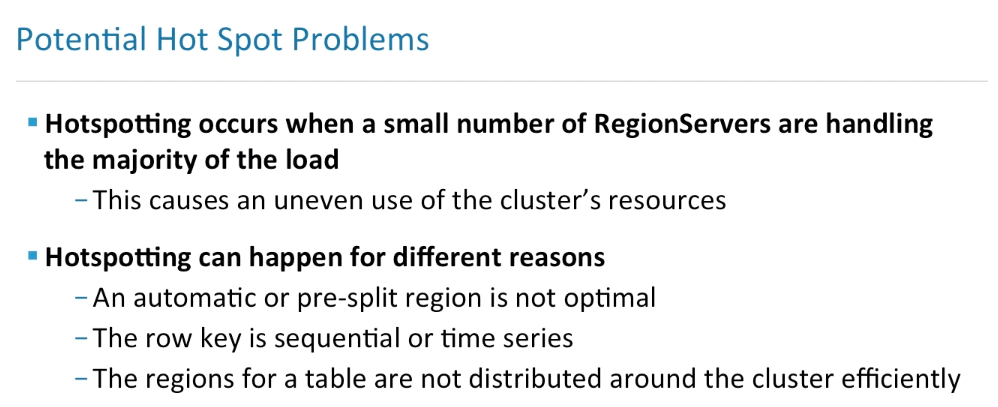


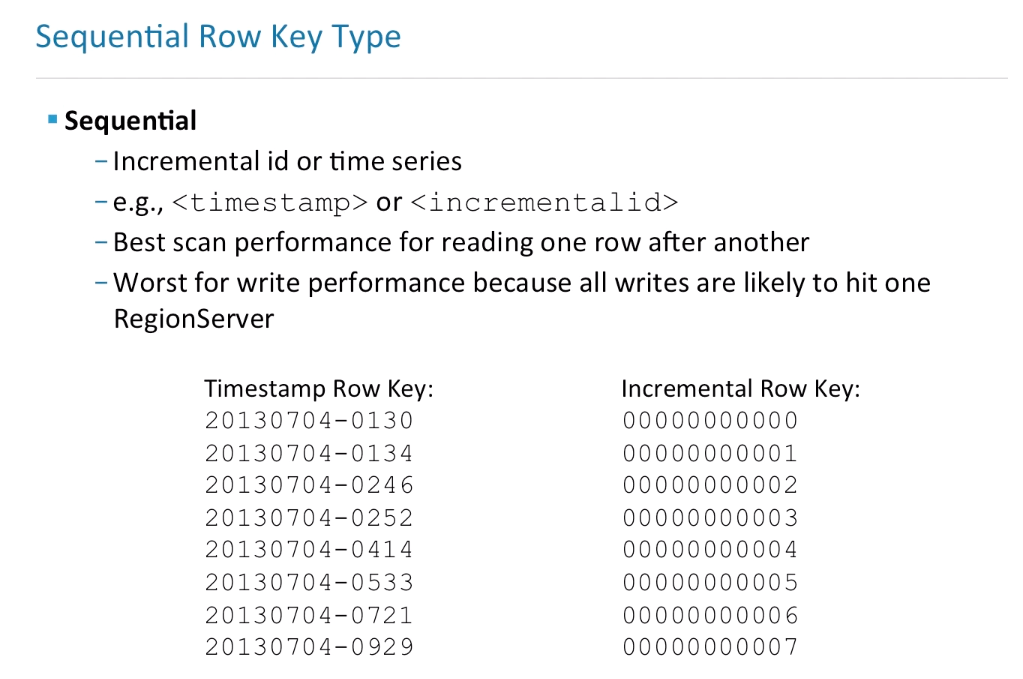
HBase stores rows by the row key. When data is inserted, the cost of sorting the data is performed. Therefore, key lookups for sequential keys are very cheap in HBase. A sort of a beginner trap is that keys are ordered lexicographically. If you recall, everything in HBase that's stored is a byte array.

So if I insert row one, row 10, row 100, and then I insert row two and I do a scan, I would see row one, 10, 100, then two. If I want to preserve some kind of integer ordering, I would insert the actual bytes for the integers themselves or use some kind of padding. And selecting the appropriate row keys is absolutely critical for performance.It's one of the first things that is considered when designing schema.



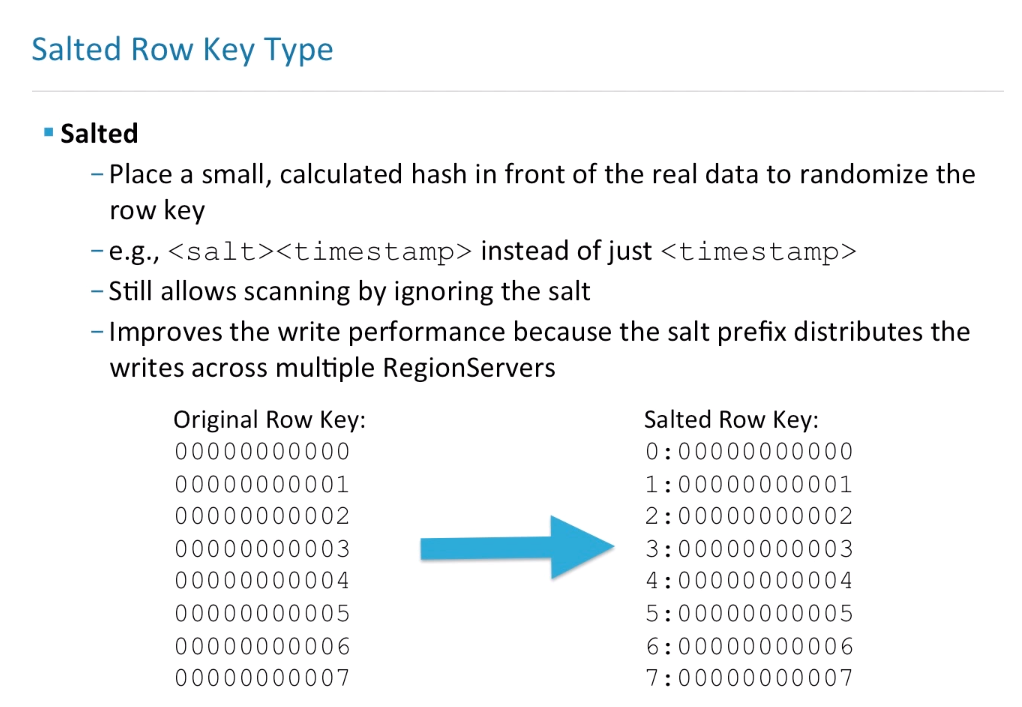






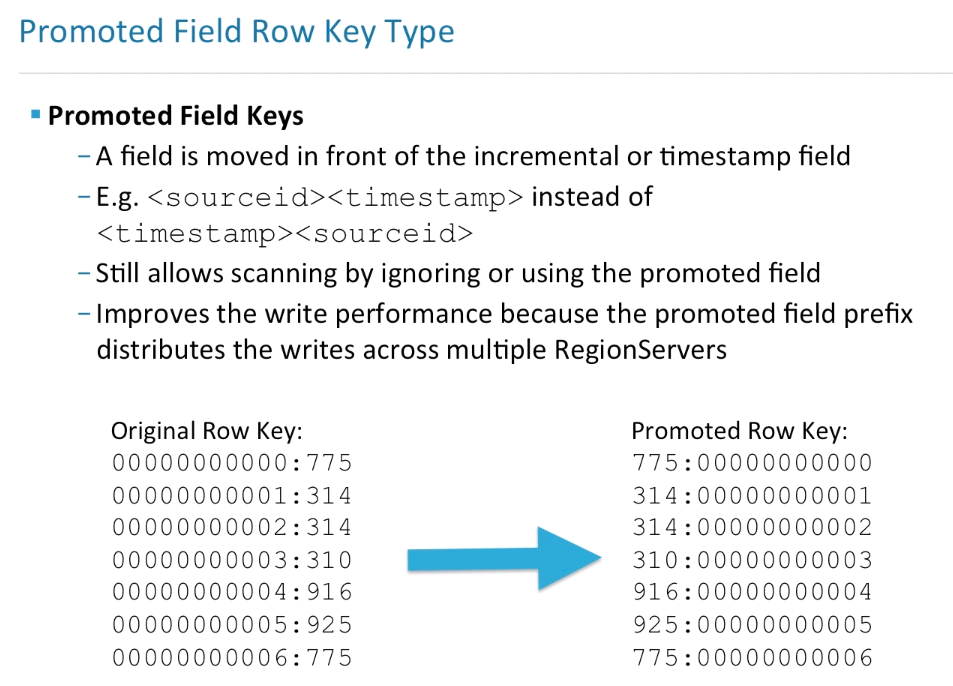
The sequential row key type is an incremental ID, or some kind of time series data. For example, a timestamp of an event log or an incremental ID from a relational database, where the primary key was simply an autogenerated globally-unique ID. **Those offer the best scan performance.**

**However, they are worse for the write performance, because all the writes for a sequence of records are likely to hit one RegionServer, at least in the amount of given time.**



We could put a salt in front of the timestamp, or if there are simply sequential IDs, we could take some other kind of field and place it in front of the key.

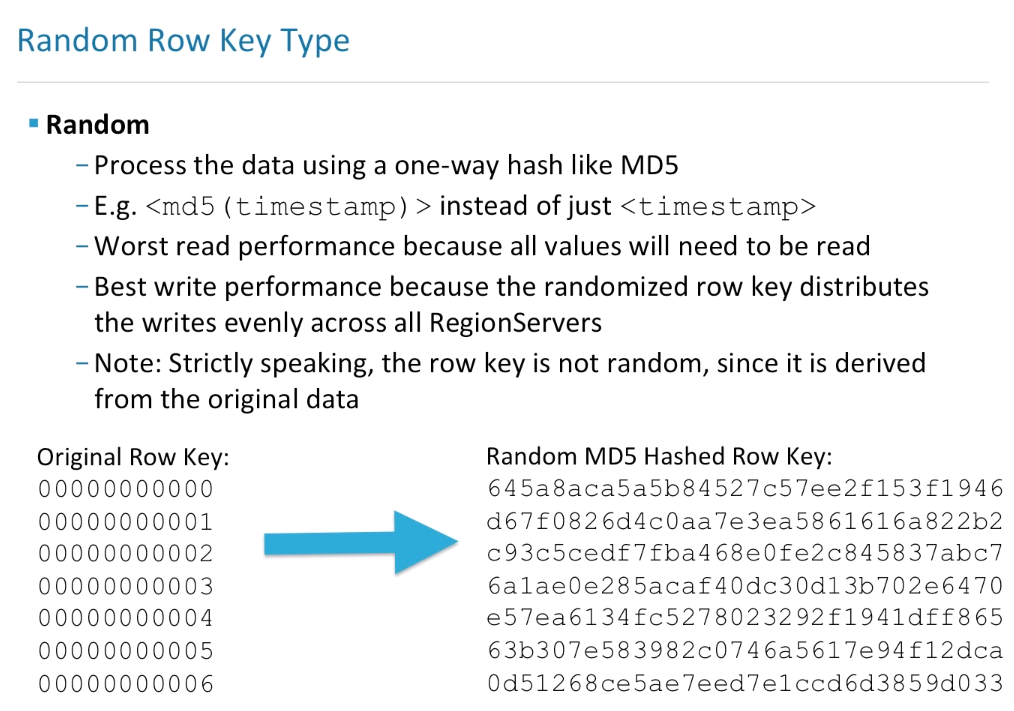
This would improve the write performance, because the salt prefix would distribute the writes across multiple RegionServers.



Another option is to take a piece of data from the record itself and prefix that piece of data to the row key. For example, we could place the host name in front of the timestamp in our keys, if we are logging events. This still allows scanning by ignoring or using the promoted field.

For example, we could have a list of all the servers whose events we are recording, and those essentially would be the buckets that we would need to scan when we want to find the events that happened at 12:00. Now this could be a thousand scans, a thousand micro scans, but it would use the resources in your cluster in a balanced way, rather than when ingesting data, making a single server hot and possibly causing failure.

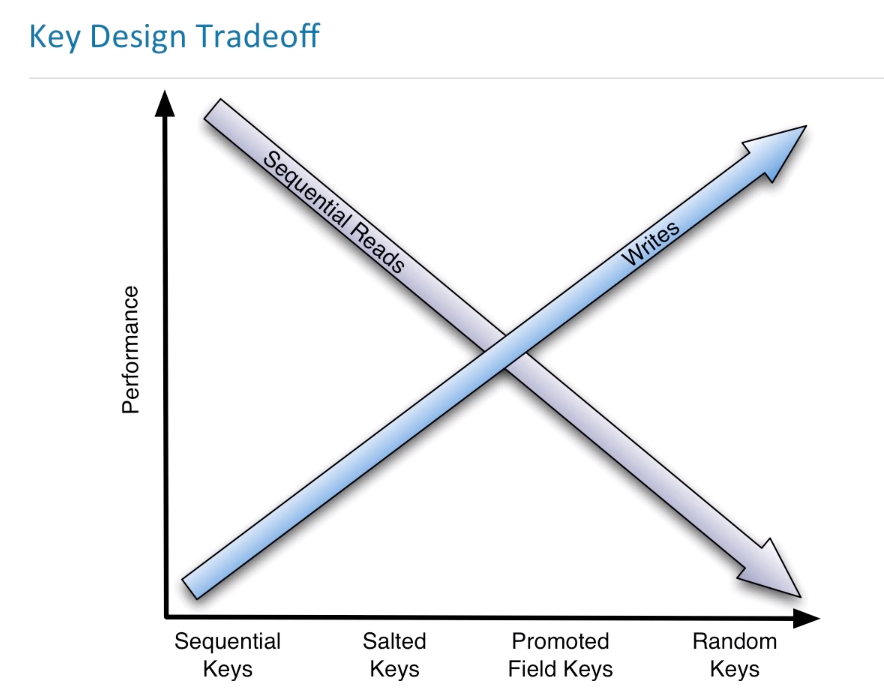
**This does improve the write performance, because the promoted field prefix would distribute the writes across multiple RegionServers.**

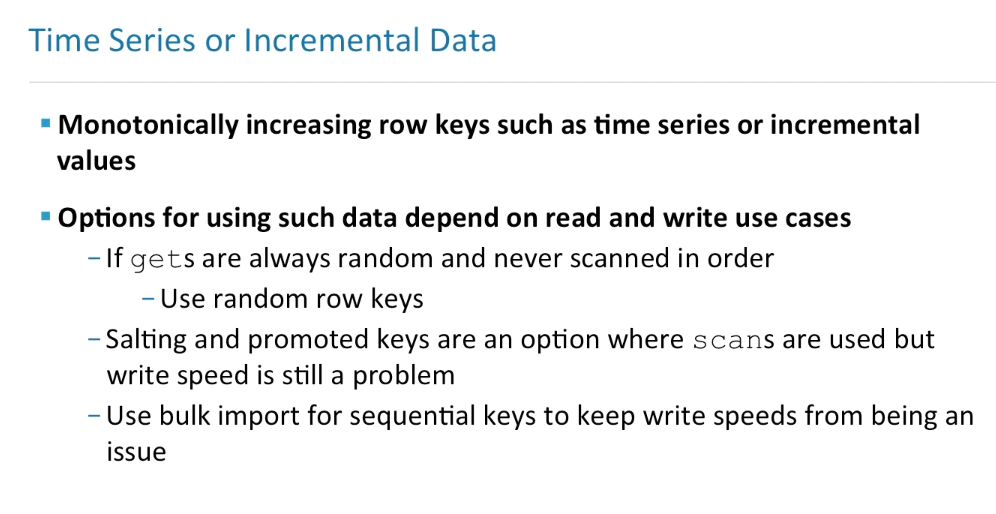


**This would give us the worst sequential read performance,** because if we are trying to put back together our event log, then all the values would need to be read and they would not be in some kind of an order that we wish. It has the best write performance, because the keys would be spread out fairly evenly among RegionServers.

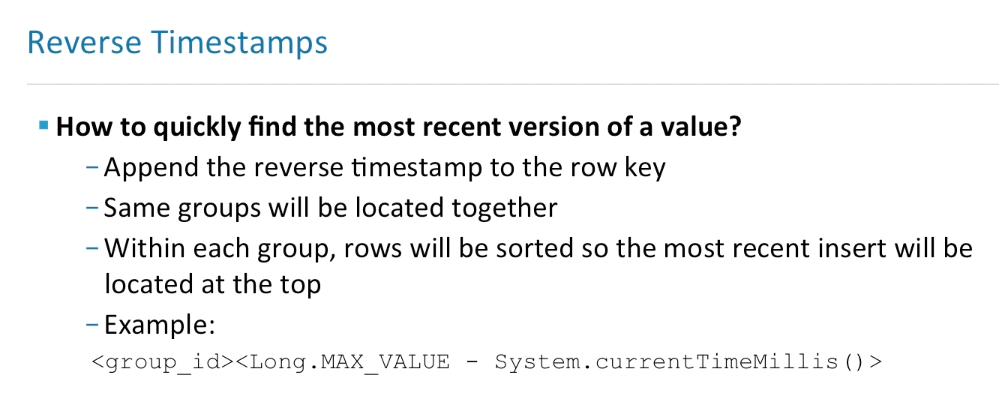
Strictly speaking, the row key, if you're using some kind of a hash, is not random, since it's derived from the original data. This means that if I have time series data and I'm getting millions of records that have the same timestamp, that simply taking the MD5 of the timestamp will provide some relief, but I will have hot spots still, they'll just be shorter.

So, what is done sometimes is that a greatly varying piece of the record is used to create the hash.





Now, salting and promoted keys are an options where you would like to use scans, but the right speed is still a problem, in the case of the event logging.



This is an access pattern. This is something that you could consider for building into your key structure in HBase. This will save a lot of reads, and presumably you're using HBase to provide access to very large data sets, so you could presume that many of these requests will be done per minute and save a lot of time and resources.

