NoSQL Data with Cloud BigTable:

Bigtable Concepts:

* Managed wide column NoSQL DB.
* Can be thought of as a series of key/value pairs with values split into columns.
* Designed for high throughput – around 10k w/r per second.
* Low latency ~ around 6 sec per node
* Scalability – scale linearly
* HA – cross cluster replication.
* HBase created an open-source implementation of the Bigtable Model.
* HBase was the adopted as a top-level Apache project.
* Cloud Bigtable supports the Apache HBase library for Java.

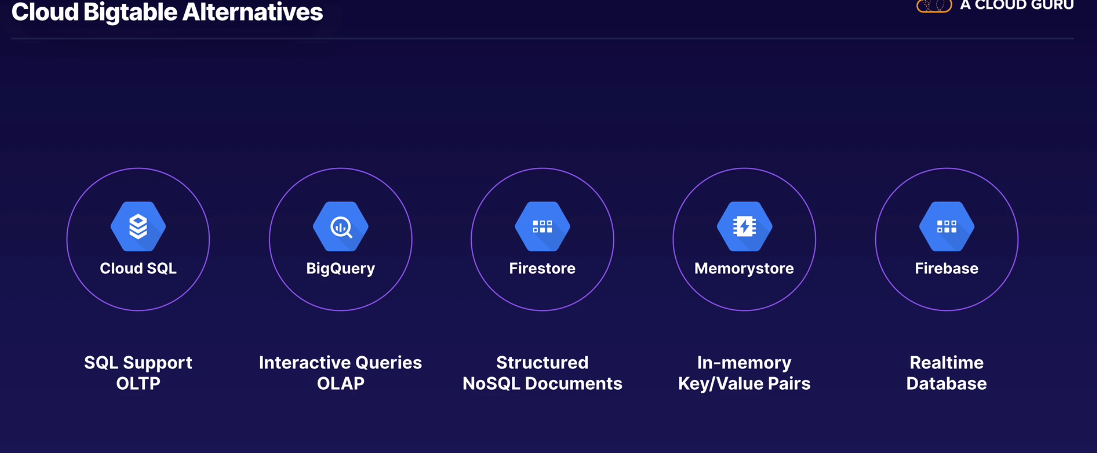
Cloud Bigtable Tables:

* Every table has a Row Key which identifies it – table is indexed only on this key.
* Each row can then have several columns which can be grouped into families.
* Row keys must be strings whereas the value stored in a cell is just stored as an array of bytes.
* Bg is a sparse DB – which means empty cells don’t use any space in your DB.
* Designed to scale to thousands of columns & billions of rows.
* Blocks of contiguous rows are sharded into tablets.
* Tablets are just big chunks of sorted rows which are put together to form a complete table.
* These tablets are what is managed by nodes in your bg cluster not the overall table itself- which means workloads can be scaled very easily.
* Tablet data itself is not stored on the cluster nodes , it’s kept in Google Colossus – Google’s internal file system which makes it very easy to scale cluster sizes up & down without having to worry about storage on the cluster nodes themselves.
* Splitting, merging & rebalancing of tablets happen automatically.

Big Table Use Cases:

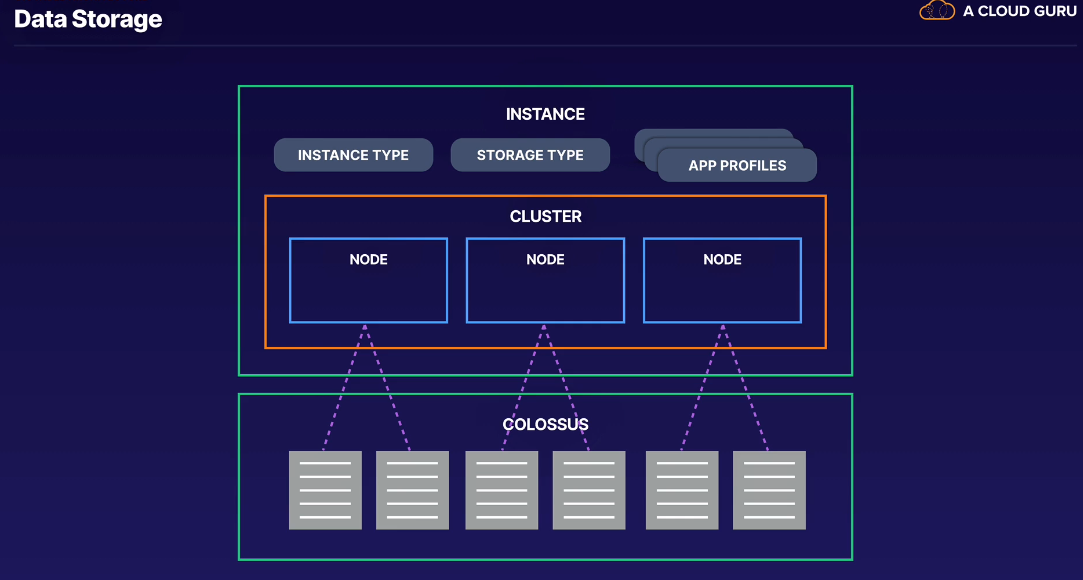
* Marketing, Transactional & Financial Data.
* Time series + IoT.
* Storing quantities of small pieces of data & very quickly.
* Streaming Analytics.
* Storage Engine for Batch MapReduce ops & ML apps.

Cloud Bigtable Alternatives:

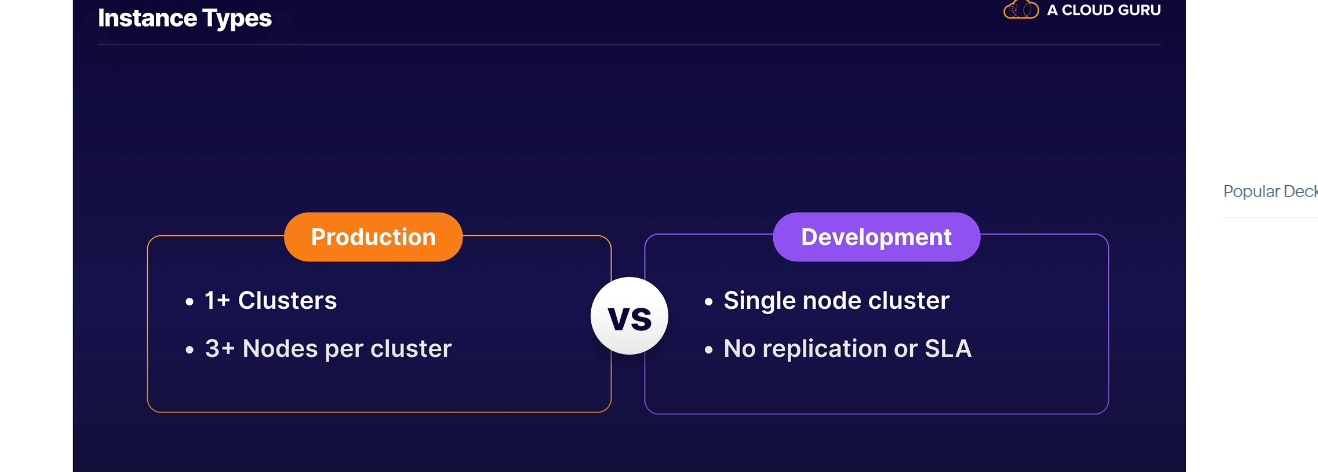


Bigtable Architecture:

* When you use bg, you create an instance – a logical container for your bg clusters that will share the same config.
* Instance is defined by instance type (dev or prod), storage type (HDD or SSD), & associated application profiles (which describes parameters for incoming connections)
* When you connect to bg from your applications you’ll use an Instance ID, & an app profile.
* All instances have a default app profile but it’s good practice to create a custom profile per app.
* Inside your instance you’ll have one or more clusters – clusters contain nodes – workhouse of bg.
* Each node handles the compute responsibility of dealing with requests from big table – so you can scale up/down your cluster.
* This flexibility comes from separating storage from cluster nodes.
* Each node handles a number of tablets which contains set of contaguious rows in your table.
* A tablet can’t be shared by more than one node but as the table grows the data in tablets can be split, rebalanced and reemerged as necessary & this happens automatically.

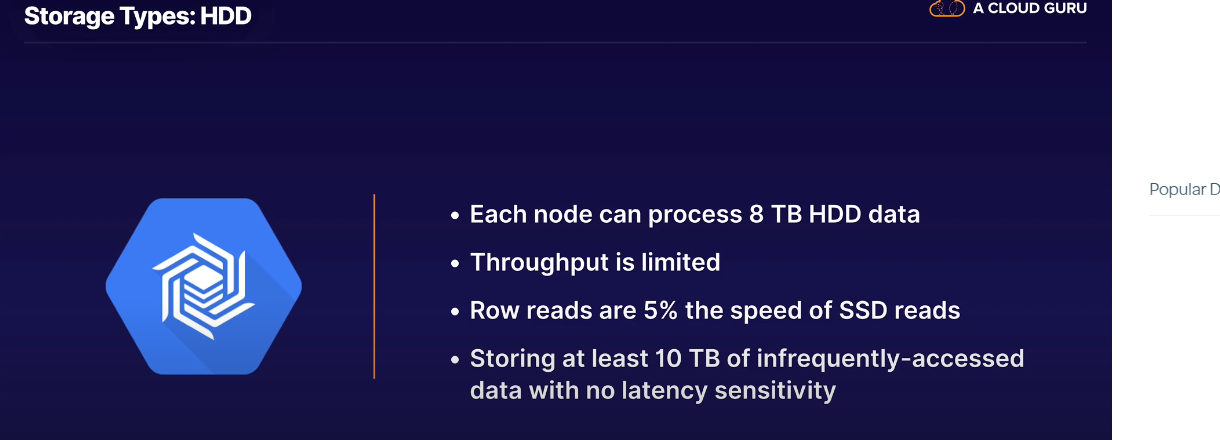


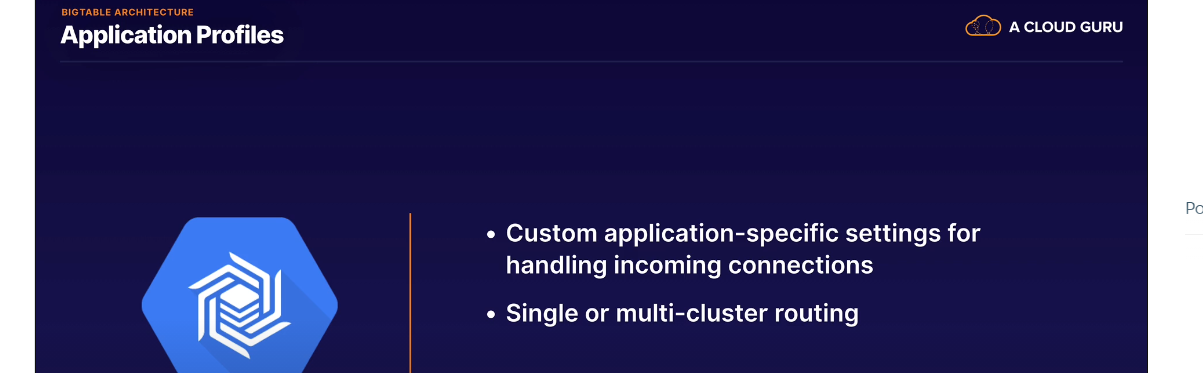
* Instance can be upgraded from dev to prod, but the change is permanent & doesn’t work the other way round.



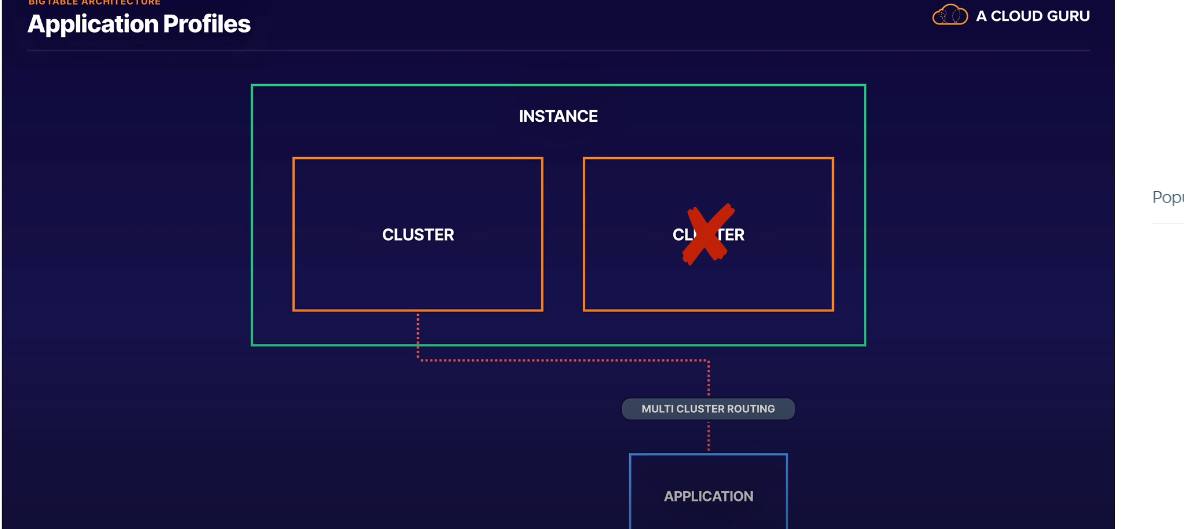
* When creating an instance you have 2 options to chose from:







* If cluster becomes unavailable in a single cluster set up, you’ve to manually repoint to another cluster.
* In multi-cluster routing, app routes to the nearest bg cluster.



* When do you need to chose Single Routing??
  + Comes down to when you application needs to support single row transactions. – tx that permit atomic transactions to single rows in a way that could not be made strongly consistent across multiple clusters

Bigtable Configuration:

* A single instance can run upto 4 clusters.
* Clusters exist in a single zone -to achieve mutli-zone redundancy is to run additional clusters in the instance in different zones in a region.
* Production clusters must run a min of 3 nodes each & default quota allows up to 30 nodes per GCP project.
* Each instance supports a max of 1000 tables per instance.

Bigtable Access Control:

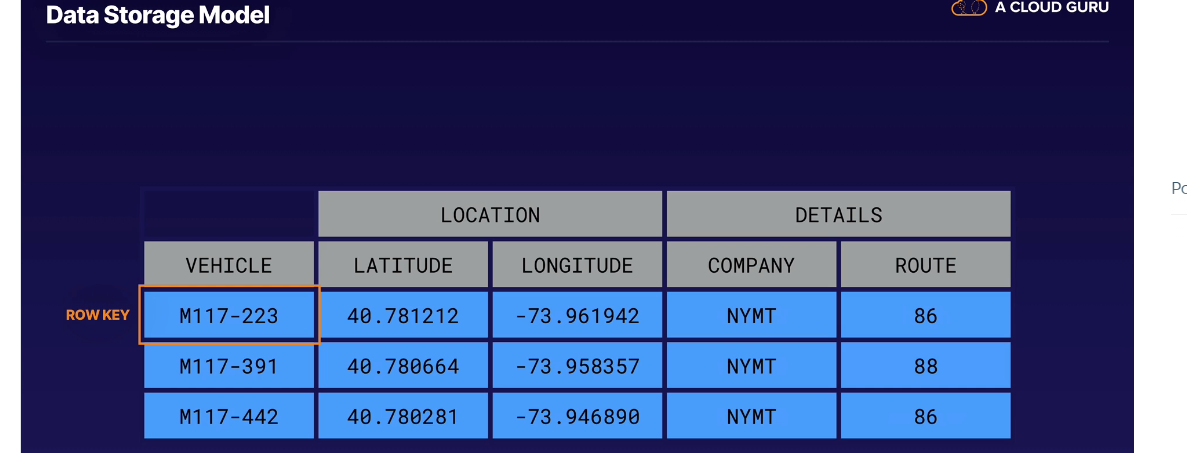
* Cloud IAM roles.
* Applied at project or instance level to:

Restrict access or admin.

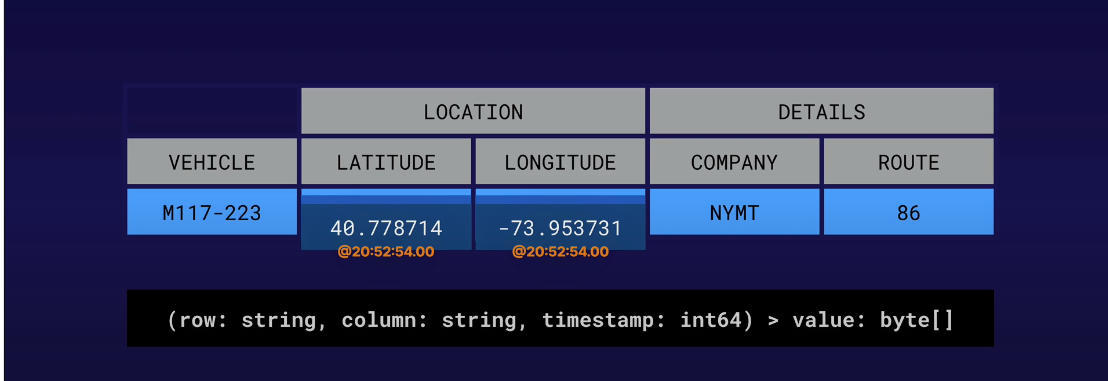
Restrict reads & writes.

Restrict dev or prod access.

**Big Data Model:**

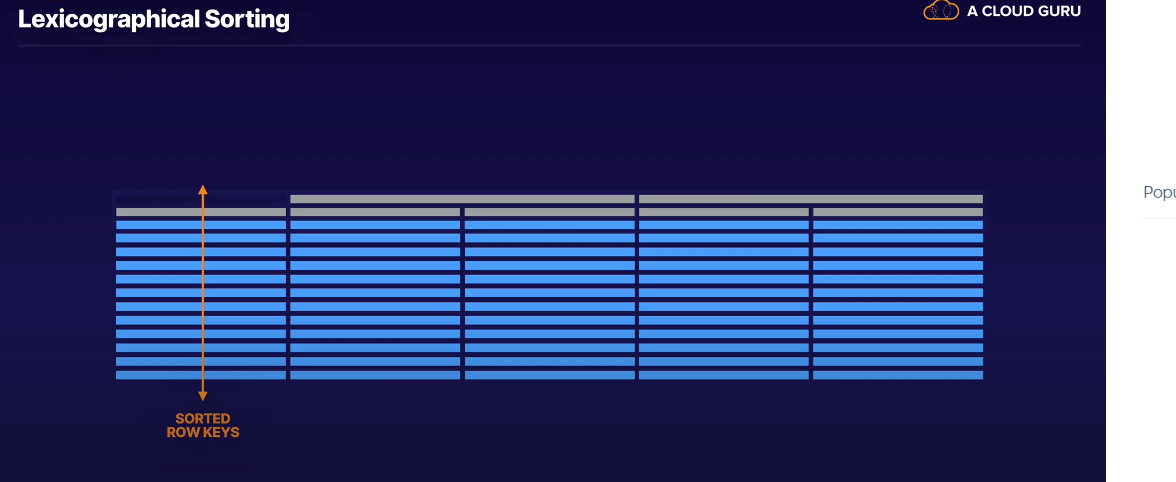


* Using Colum Families can make it possible to retrieve just the data we need rather than an entire row.
* You can have upto 100 column families in a table.
* Name of the column is referred to as the column qualifier.
* In a 2D table, at the intersection of a row & column we find a cell. However, bg is a 3D table, every cell is written with a timestamp & if the cell is mutated new values can be written without overwriting the original values.
* How much cell history you store & how long is configurable. This makes it possible to retrieve data with more granularity specifying row, column & timestamp – value of any cell you achieve is an array of bytes.



Lexicographical Sorting:

* Within a tablet, rows are sorted lexicographically – potentially alphabetically by row keys.
* This allows you to optimize searches across specific ranges of row keys.



Atomic Operations By Row:

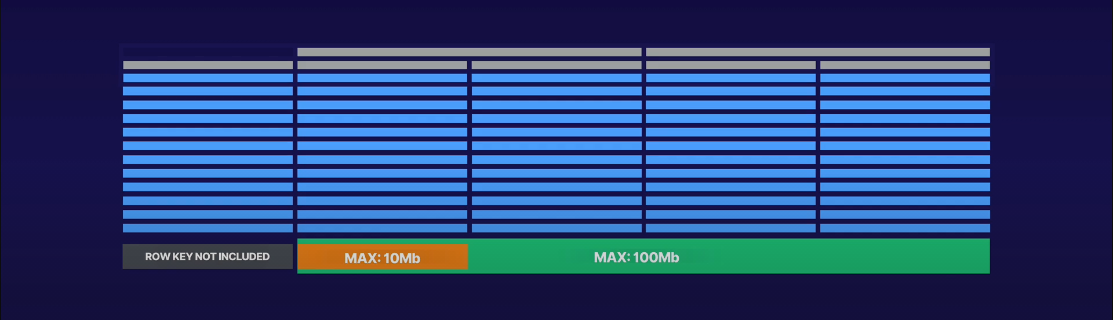
* Only by row as oppose to some other transactional DBs.
* This means e.g, that if you update 2 rows in a table, it’s possible that one of the updates may fail & leave the row unchanged

Sparse Tables:

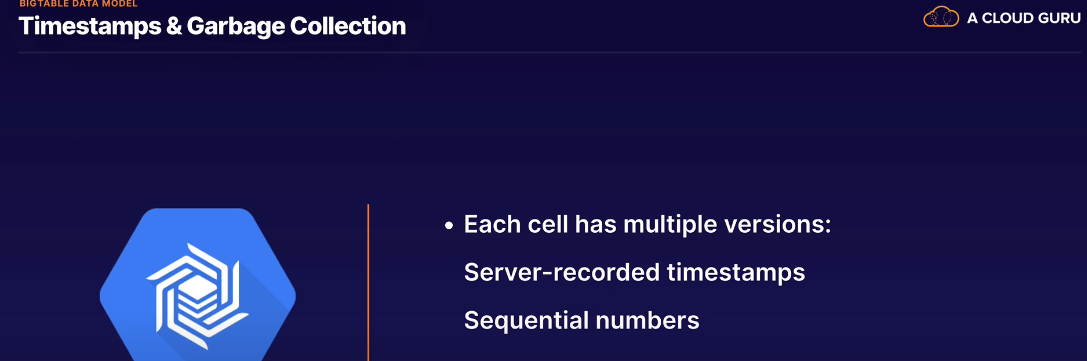
* Empty cells don’t consume any space in the DB. – won’t

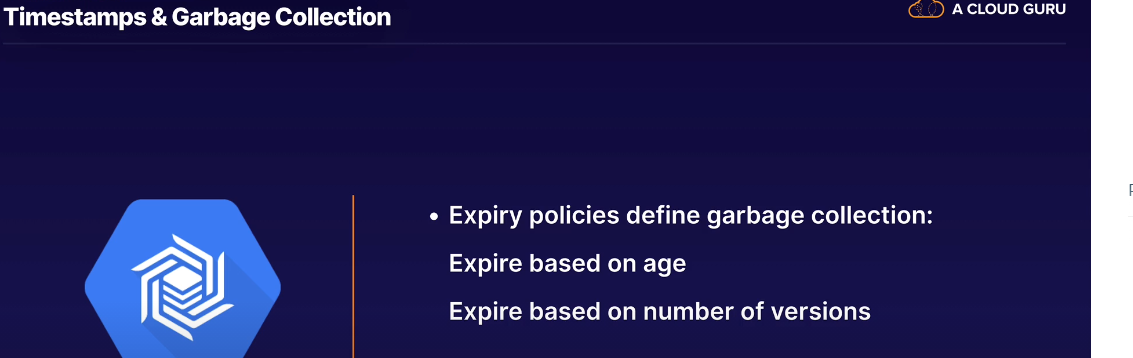
Row Sizing:

* Individual cells should be no larger than 10Mb.
* A row should be under 100MB; although bg supports rows up to 256MB.



* To help you keep the sizing under control, bg uses timestamps & garbage collection.

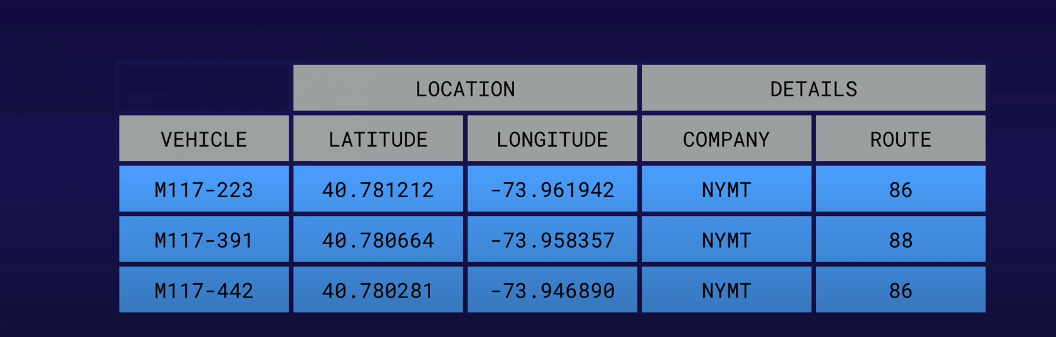




* Creating a column family with HBase client will set the policy to only retain the latest version of cell.
* If you use any other client library it will set the policy to store infinite versions.
* If you can update the policy to expire on a specific age or number of versions if using sequential numbering as opposed to timestamps.

Bigtable Schema Design:

* Consider the fictional transport vehicle table.



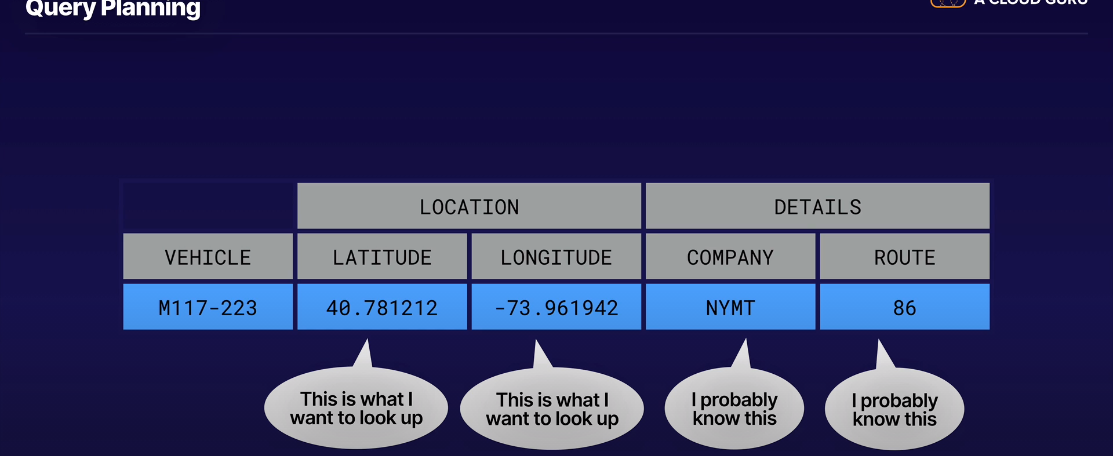
* Our row key is the vehicle identifier, so retrieving the row if we know the identifier will be quick & efficient.
* What if we wanted to get all the vehicles for a particular route? E.g Route 86.
* In SQL DB we might create a secondary index for that column – but there are no secondary indexes in bigtable.
* Instead we have to scan the results in the entire table – filtering results based on a regex match to the string contained in that column cell. – This is the most time expensive way of querying bigtable.
* To help improve on this, we really need to think about query planning:

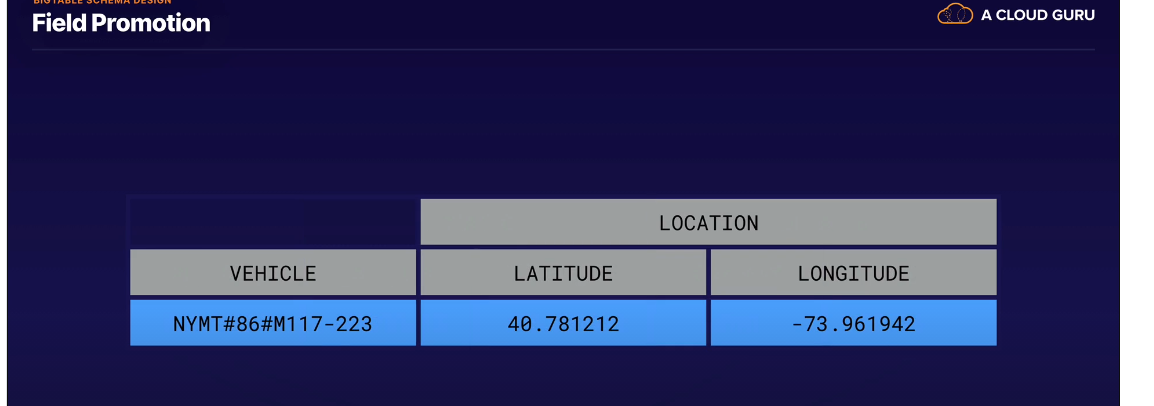
Query Planning:

* Fundamental to good schema & row key design.
* It really means what sort of questions you are going to be asking about our Data
* Let’s say we want to get data for all vehicles that serve route 86
* For the data we have, we already know the company & the route – what we are looking for is the latitude & longitude
* Since we already know the Company & Route number for a vehicle, there is no benefit storing it in a column – a practice called Field Promotion.

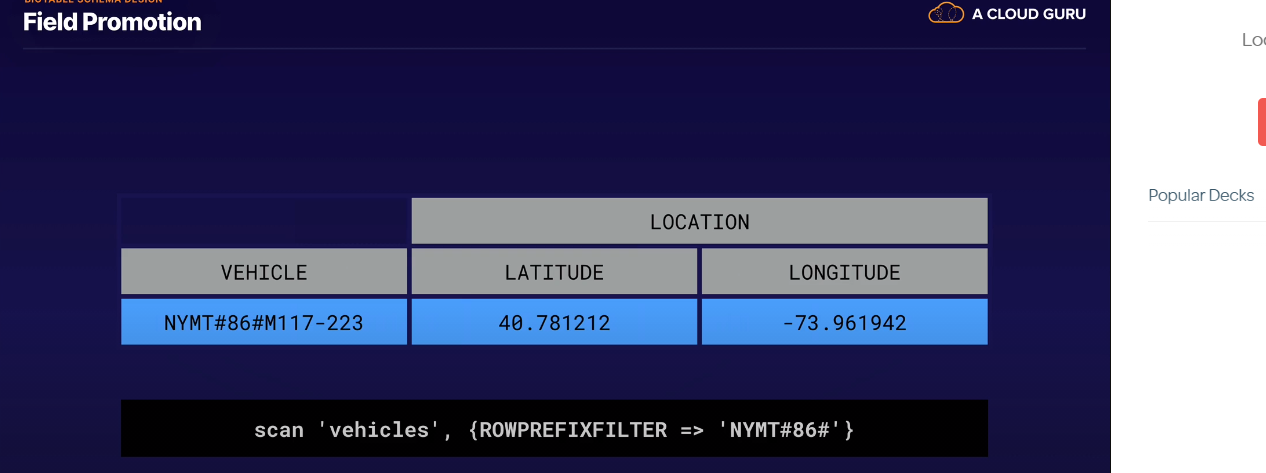
Field Promotion:

* Means taking data that we already know and moving it to the row key itself to create a better row key.
* We end up with a row key design that gives us a couple of benefits.

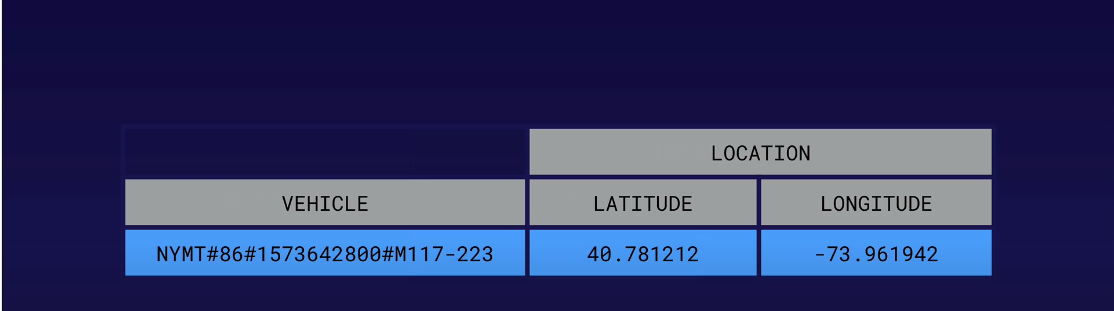




* We can retrieve all the vehicles from NYMT and route 86 without having to scan the entire table. – Rows are automatically sorted lexicographically.



* Bigtable doesn’t have to scan the entire table to find these rows.
* Depending on app needs, you can optionally include a timestamp in your row design.



* Never put a timestamp at the start of a row key.

Designing Row Keys:

* Consider what queries you’ll run & how they can be optimized.
* Queries will either use: a specific row key, row key prefix or a combination of prefixes that ultimately return a row range in the sorted table.
* Some consideration for the components of a row key are:
  + Reverse domain names – e.g when storing multiple entiries related to URLs. Reversing the domain will keep rows for the same website closer together
  + String Identifiers – unique string ids are good components coz they lend themselves to natural distribution
  + Timestamps – only as part of bigger row key design

Row keys to avoid:

* + Domain names – maybe distributed but reads are not going to be contagious.
  + Sequential numbers – writes are always going to be on the end of the table rather than being distributed
  + Frequently updated identifiers – repeatedly updating same row with same information is not as performant as adding new rows.
  + Hashed values

Design for Performance:

* Lexicographic sorting.
* Store related entities in adjacent rows.
* Distribute reads and writes evenly.
* Balanced access patterns enable linear scaling of performance

Time series Data:

* Often uses tall & narrow tables where each row may contain a key & possibly only a single column.
* It’s efficient to use new rows for every event stored. – use rows instead of versioned cells.
* With tall & narrow tables logically separate event data into separate tables.
* Don’t reinvent the wheel – OpenTSB

Avoid Hotsposts:

* To avoid hotspots:
  + Create well distributed row keys using Field promotions.
  + If struggling to make your writes non-contiguous consider adding a salted hash to your row key that artificially distributes rows. Your salt calculation should be based on the total number of nodes so that you can achieve uniform distribution.
  + Use Key Visualizer tool. Shows you CPU Heatmap of the load for row key components.

Bigtable Advanced Concepts:

1. Monitoring:

* You can monitor via GCP console or Stackdriver.
* Key thing to look out for is the overall CPU utilization of the cluster & the average CPU utilization of the hottest node. If there’s a big disparity between these two metrics you likely have a hotspot somewhere in your table.
* CPU overhead varies depending on your instance config. For a single cluster instance you should aim for a 70% CPU utilization with the hottest node not going above 90%.
* For an instance node with 2 clusters, multi-cluster routing introduces additional overhead so 35% and 45% respectively.
* Storage Utilization – keep at 70% per node. Writes will fail if you go above write limit.
* Separate applications using application profiles to enable monitoring.



1. Autoscaling:

Stackdriver metrics can used for programmatic scaling. – Run your own SW that will use the bg client libraries metrics then update cluster node counts via API.

Rebalancing of tablets takes time; performance may not improve for ~20

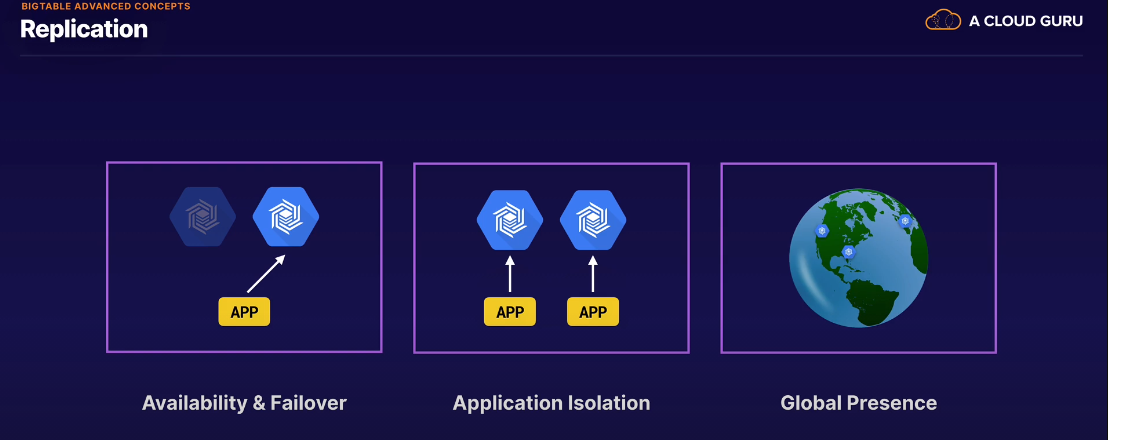
Adding nodes to a cluster doesn’t solve the problem of a bad schema.

Not an inbuilt solution.

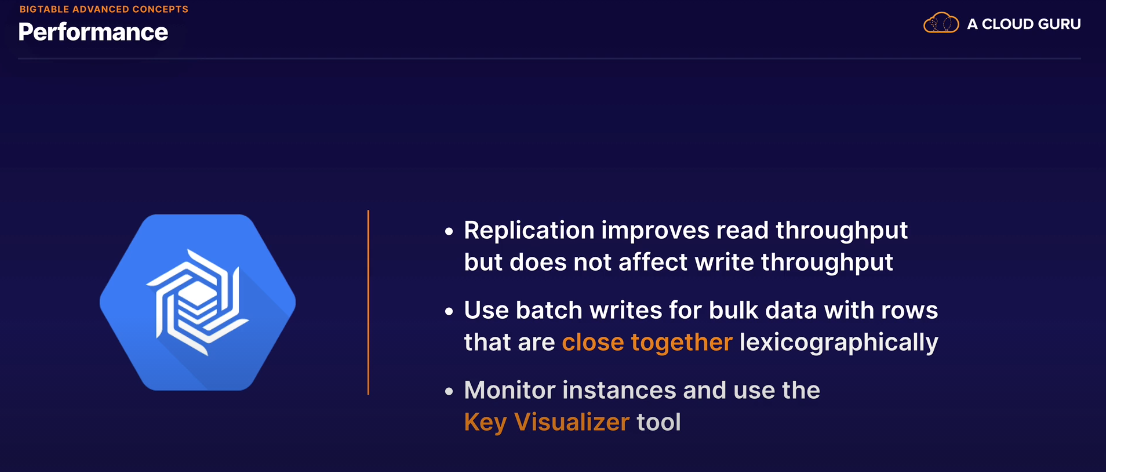
1. Replication:

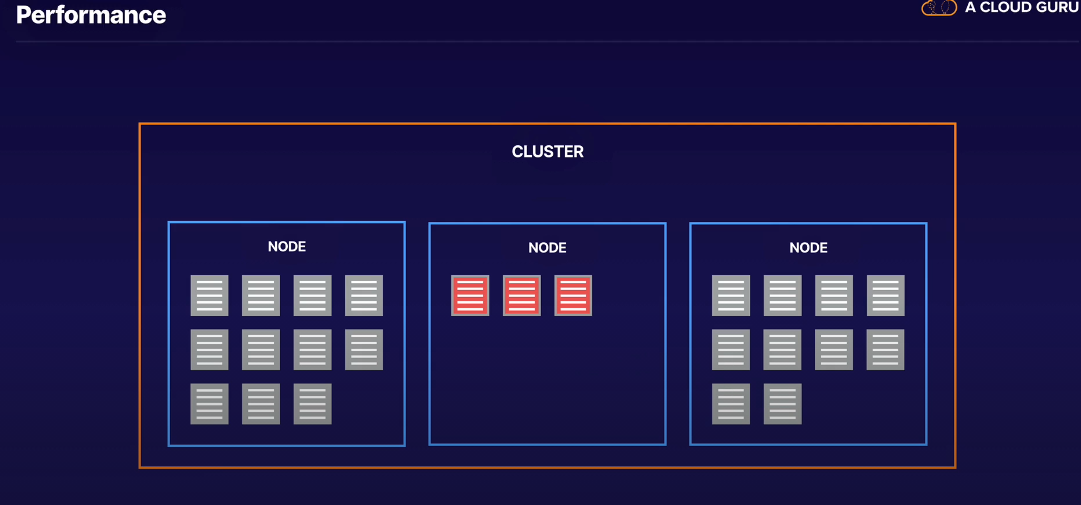
Additional clusters automatically start replication, i.e. data synchronization.

Replication is eventually consistent. – limit your cluster to single routing if this is going to be a problem.









Exam Tips:

