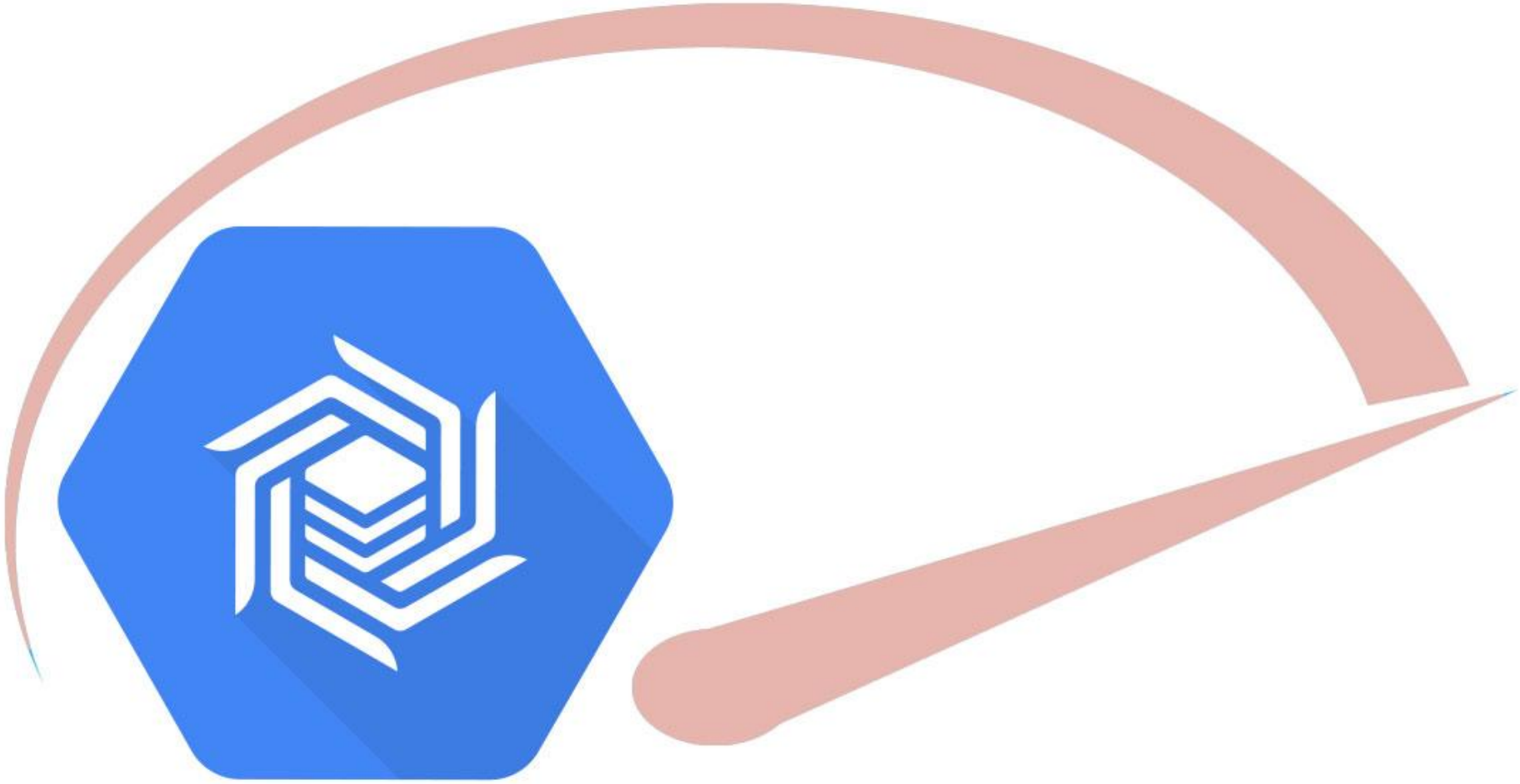


Bigtable - Preventing key hotspots at scale

Date: 2 July 2020

Presented: Irvi Aini (@irvifa)

Google Cloud Data User Group

slido

Join at

slido.com

#gcpdata

Agenda

1. Intro to BigTable
    - High-level architecture
    - How to use / key structures / tools
2. When to use BigTable
    - High-level overview of use case
3. Problems encountered.. i.e. Hotspots after Scale-Up
    - How solved / addressed

When to Use Bigtable

- At least > 1 TB, otherwise the overhead is too high
- Each row only contains < 10 MB
- No SQL or multi row transactions
- Stores key value pairs

Good Use Cases Using Bigtable

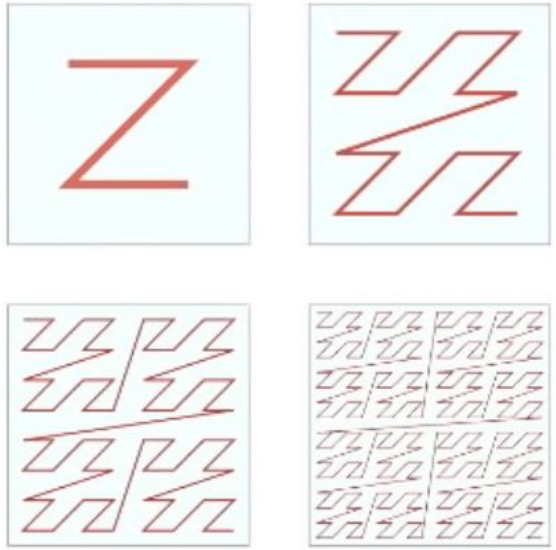
- Google Analytics (Google)
- Personalized Search (Google)
- Youtube (Google)
- Advertising Analytics (Twitter)
- IoT and Sensor Data Management and Analysis (GeoMesa)
- etc

IoT and Sensor Data Management and Analysis (GeoMesa)

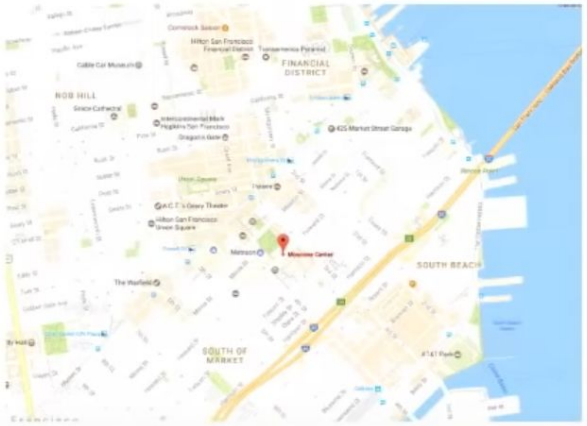
- Geographics
  - Roads: Lines
  - Natural
    - Rivers: Lines
    - Lakes: Polygons
  - Political
    - Capital City: Points
    - Region: Polygon
- Satellite Imagery
- Weather
- etc

Indexing Spatio-Temporal Data using Space - Filling Curve (GeoMesa)

- Encode latitude, longitude, and time
  - nearby located space is located in nearby index
- 2-D Z-Order: simple



Moscone Center coordinates
37.7839° N, 122.4012° W



Encode coordinates to a 32 bit Z

1. Scale latitude and longitude to use 16 available bits each

   scaled_x = (-122.4012 + 180)/360 * 2^16
   = 10485
   scaled_y = (37.7839 + 90)/180 * 2^16
   = 46524

2. Take binary representation of scaled coordinates

   bin_x = 0010100011110101
   bin_y = 1011010110111100

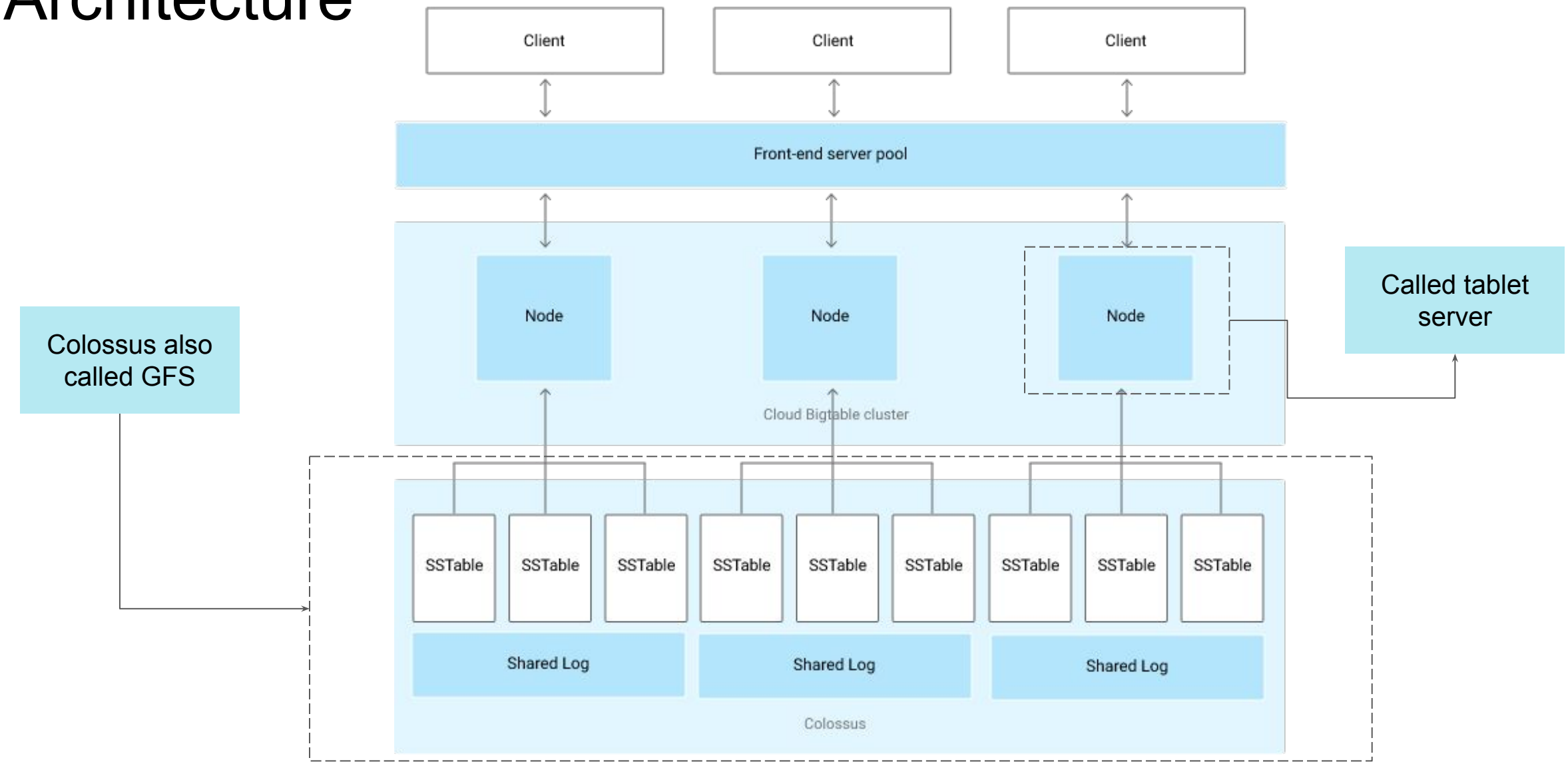
3. Interleave bits of x and y and convert back to an integer

   bin_z = 01001101100100011110111101110010

z = 1301409650

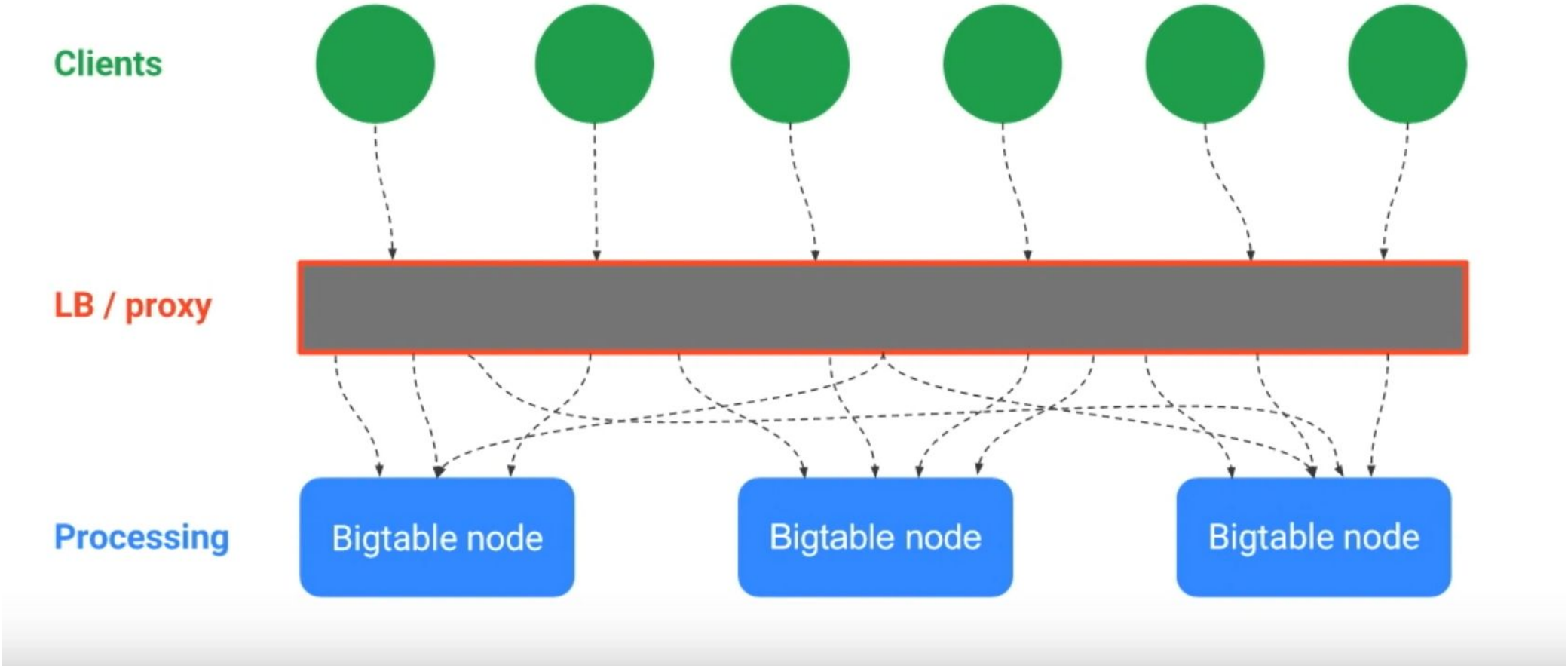
Bigtable Key

Architecture



https://cloud.google.com/bigtable/docs/overview

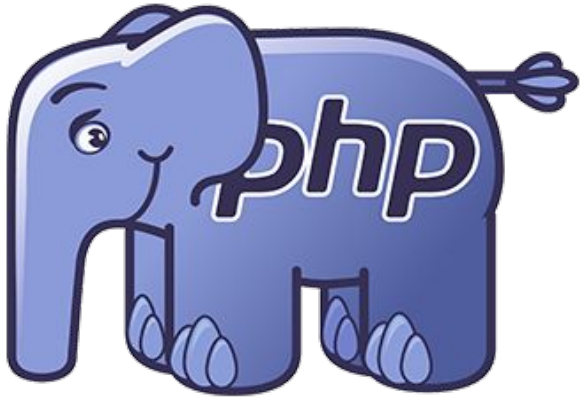
Client’s Connectivity



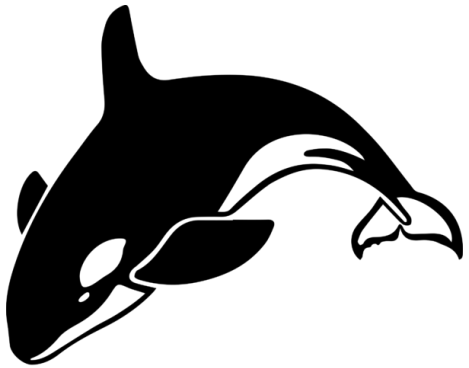
Available Clients











https://cloud.google.com/bigtable/docs/reference/libraries

API

- The API provides functions to:
  - create and delete tables and column families
  - changing cluster, table, and column family metadata (access control rights)
- Client have the ability to write and delete row values, lookup values, or iterate through subset of data
- Quick start can be used using cbt or hbase shell

Bigtable Data Model

Bigtable is 3 dimension map, indexed by:

- Row Key: can be arbitrary string up to 64 kB, RW operation in row is atomic
- Column Key
- Timestamp

Each of this value is represented as uninterpreted array of bytes.

Illustration on How Data is Stored

- **Column:**
  - Are grouped into sets called column families, which form the basic unit of access control.
  - All data stored in a column family is usually of the same type, compressed in the same column family together

	Column-Family-1		Column-Family-2	
Row Key	*Column-Qualifier-1*	*Column-Qualifier-2*	*Column-Qualifier-1*	*Column-Qualifier-2*
r1	r1, cf1:cq1	r1, cf1:cq2	r1, cf1:cq1	r1, cf1:cq2
r2	r2, cf1:cq1	r2, cf1:cq2	r2, cf1:cq1	r2, cf1:cq2

- **Row:**
  - Data is maintained in lexicographic order by row key.
  - Row range is dynamically partitioned, each row range is called a tablet
  - Tablet: a unit of distribution and load balancing
  - That's why reading a short range data will be more efficient since it only involved several machines

- **Cells**
  - versioned based on TIMESTAMP (default)
  - TTL/expiration in the CF level

Case Study: Advertising Analytics (Twitter)

Sample schema with sample dimension:

- **timestamp**
- advertiser id
- campaign id
- engagement type
- display location

1st iteration:

- Concatenate all of the dimension
- Pros
  - Easy to scan all advertisers based on time
  - Good data locality
- Cons
  - Possibility of hotspot because timestamp in the front will cause monotonically increasing key

Case Study: Advertising Analytics (Twitter)

Sample schema with sample dimension:

- **advertiser id**
- campaign id
- **timestamp**
- engagement type
- display location

2nd iteration:

- Concatenate all of the dimension
- Pros
  - Easy to scan all advertisers based on time
  - Good data locality
- Cons
  - Harder to query all advertisers (however this is not common use case for them)

Case Study: Advertising Analytics (Twitter)

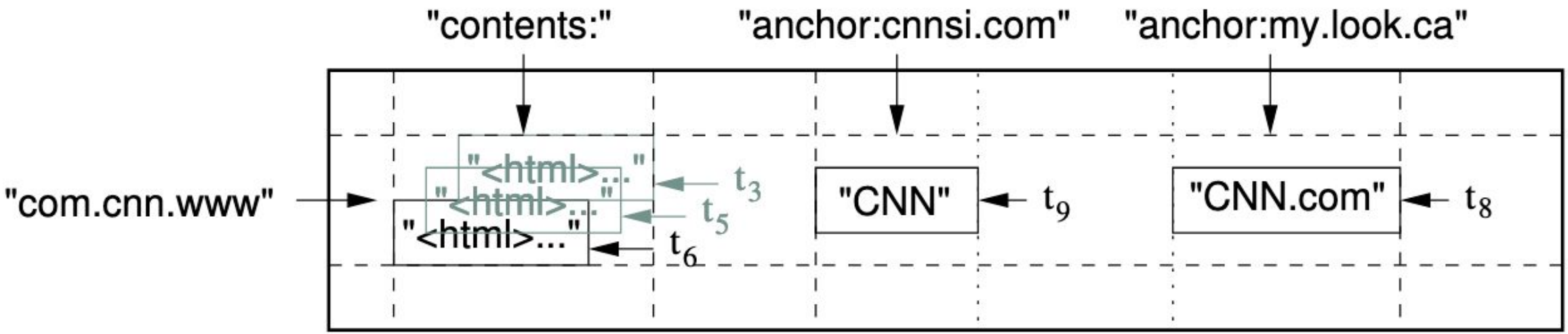
Sample schema with sample dimension:

- **advertiser campaign (reversed)**
- campaign id
- timestamp
- engagement type
- display location

3rd iteration:

- Concatenate all of the dimension
- Pros
  - Easy to scan all advertisers based on time
  - Good data locality
  - Well distributed key
- Cons
  - Harder to query all advertisers (however this is not common use case for them)

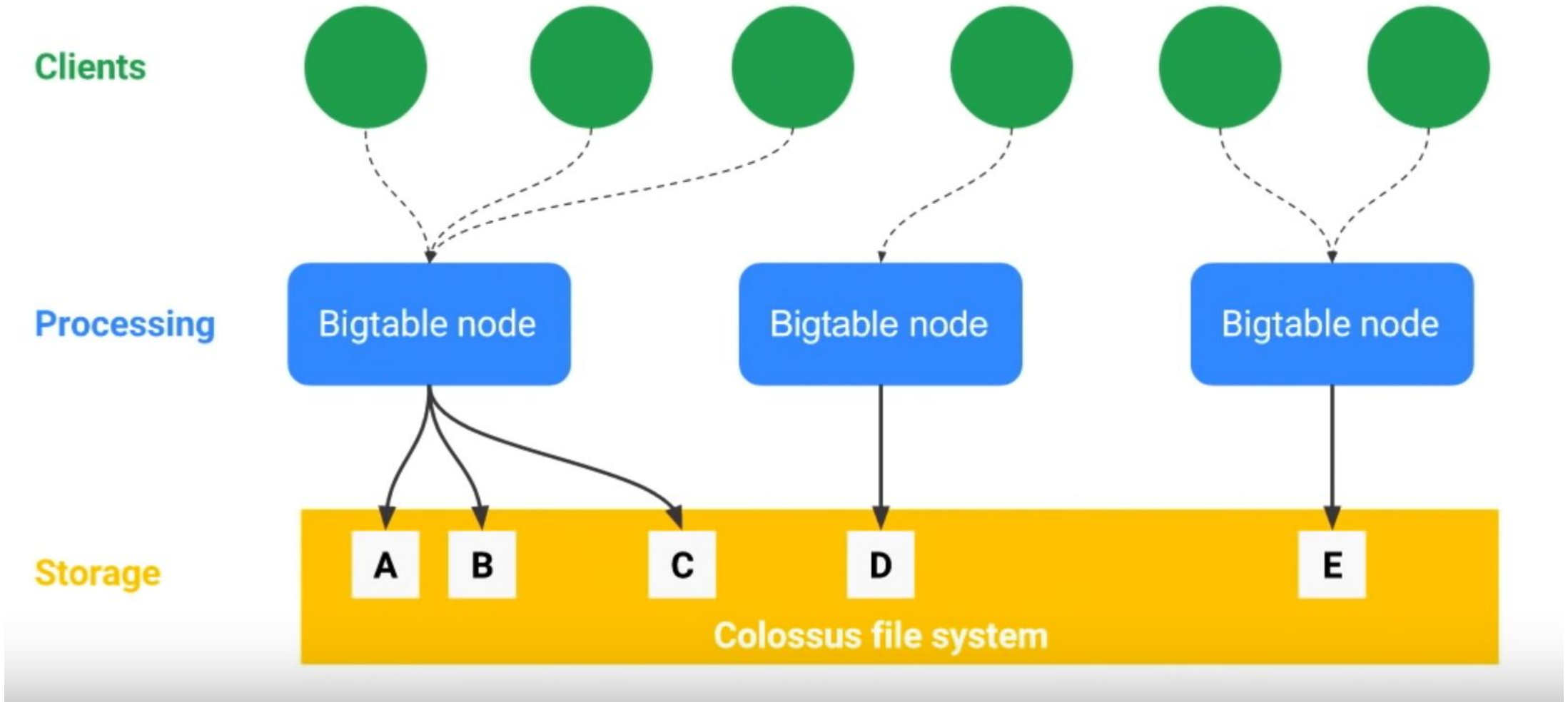
Illustration on Row and Column in Bigtable



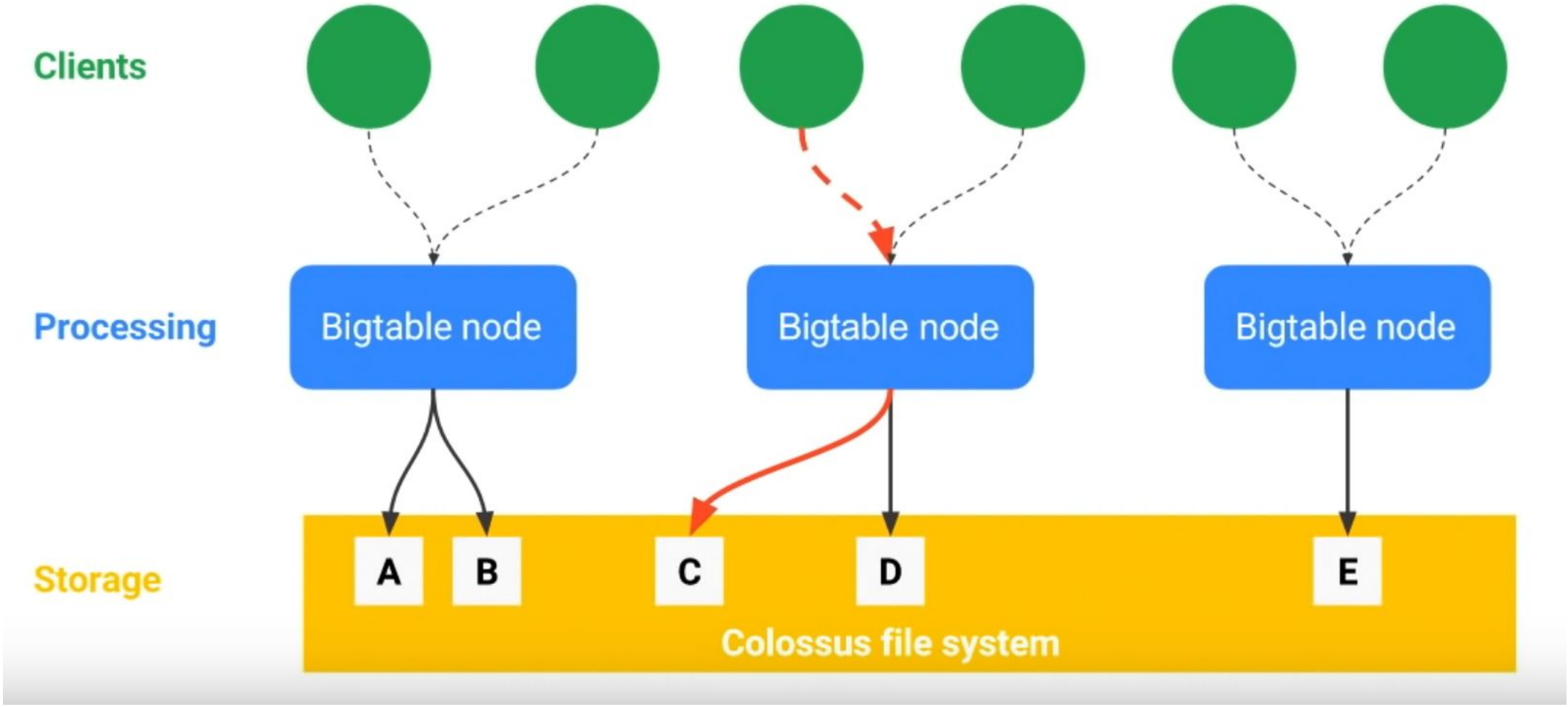
Defining a Row Key

- Key strategy that accommodate common query
  - Can be hashing or anything that can make sure the data is distributed evenly, in this case it will prevent key hotspot
- Avoid monotonically increasing keys, e.g:
  - Timestamp
  - Sequence type ID
- Combined key will be helpful as well

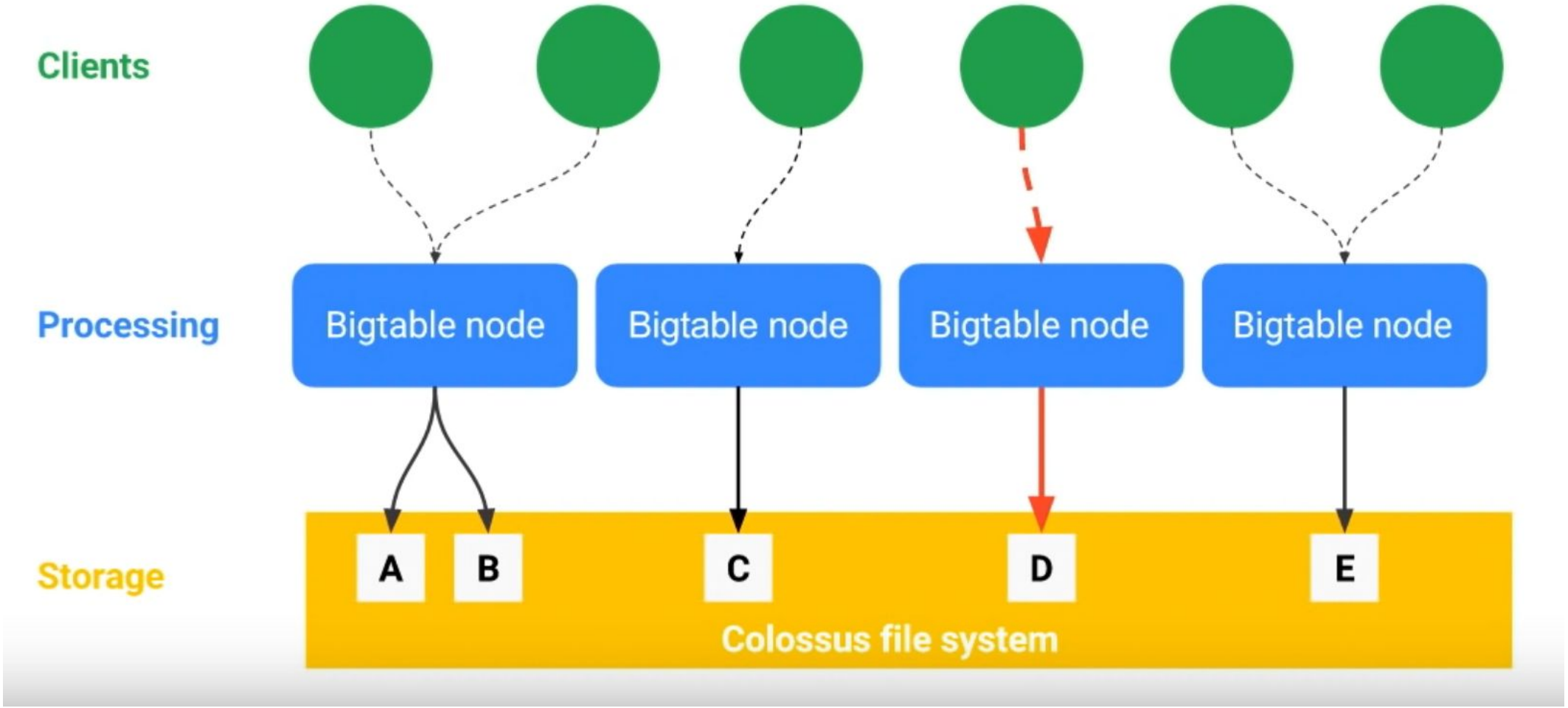
Rebalancing



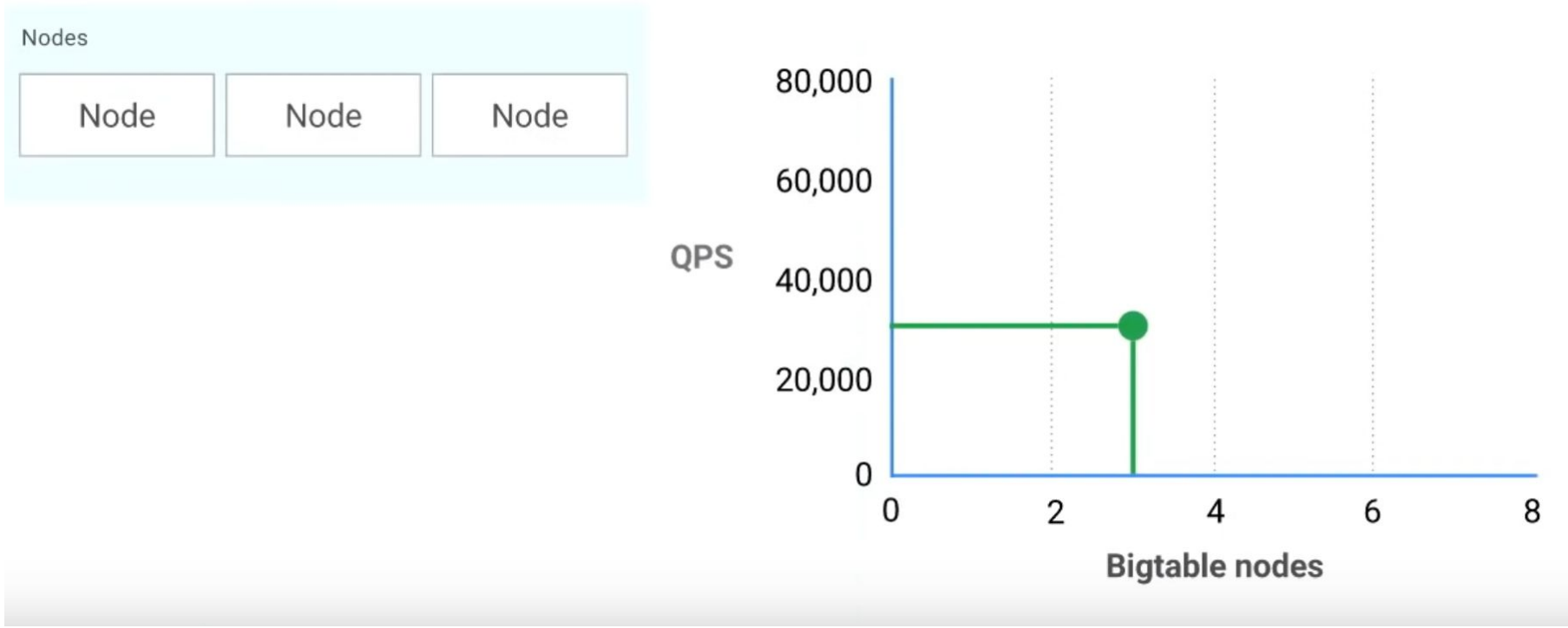
Rebalancing (Con’t)



Resizing


Clients
Processing
Storage
Bigtable node
Bigtable node
Bigtable node
Bigtable node
A
B
C
D
E
Colossus file system

Throughput is Increased in Linear Behavior



However how write and read pattern affecting the rebalancing?

Designing Schema

- General concepts:
  - Each table has only one index, the row key
  - Rows are sorted lexicographically by row key
  - Columns are grouped by column family and sorted in lexicographic order within the column family
  - All operations are atomic at the row level
  - Ideally, both reads and writes should be distributed evenly
  - Related entities should be stored in adjacent rows, make sense since reads will be more efficient
  - Cloud Bigtable tables are sparse.
  - It's better to have a few large tables than many small tables

https://cloud.google.com/bigtable/docs/schema-design

Row Schema and Performance

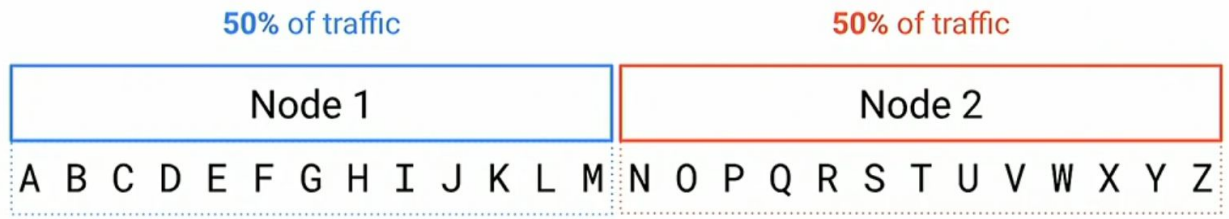
- Good row schema design is crucial
- Row schema guide access pattern
- Uneven access pattern causing hotspot
- Balanced access pattern will resulting in linear scaling

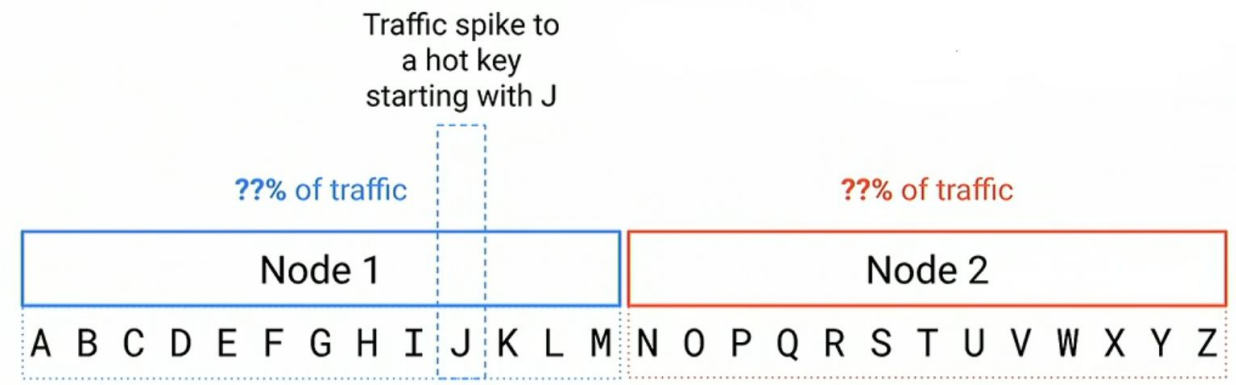
Problems Encountered

- Whenever there’s a drastic increase in the traffic
  - The system have a high write and read operation
    - Need more nodes to serve the sudden peak in the traffic, however Bigtable by default doesn’t provide autoscale mechanism.
  - Even when we increase the nodes before the traffic, we should make sure the rebalancing performed as expected, typically rebalancing will take about 20 minutes or so, depends on the traffic coming, the data we have, and also the size of the nodes addition
  - We can check it using key visualizer

Simulation

- Cluster size: 2
  - p 99th Read: 10 ms
- Cluster size: 2
  - begin a hotspot key in J
  - p 99th Read: 60 ms
- Cluster size: 4
  - p 99th Read: 57 ms





Assuming one key with significantly more traffic than the rest

$$p_{99} \approx \frac{traffic_{cluster} \cdot \frac{traffic_{\text{hottest key}}}{traffic_{\text{median key}}}}{|cluster| - 1 + \frac{traffic_{\text{hottest key}}}{traffic_{\text{median key}}}}$$

Common Way to Make the Key Distributed Evenly

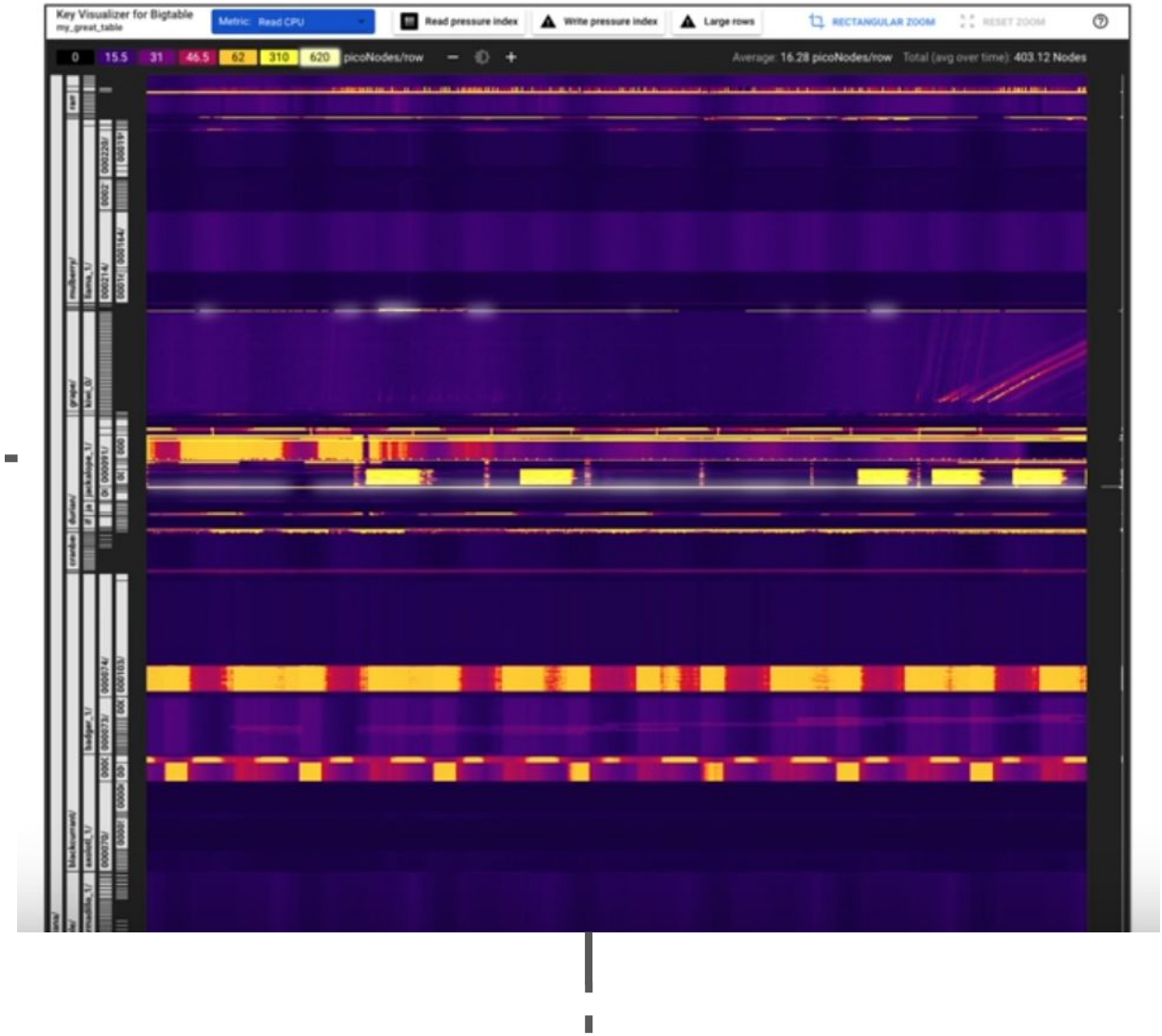
- Hash all key with fast non-cryptographic hash function
- Reverse all key (assuming it’s in numerical order)
- Promote most random field in your data as row key

Getting the Insight of Resource Usage

Does it help to know what's the reason of High CPU usage? Or the possibility of key hotspot?

Using Key Visualizer

- What is key visualizer?
  - Read/Write access pattern over time,
    - based on heatmap
- Usage:
  - Find/prevent hotspot
  - Find rows with too much data
  - See if the schema is balanced



Vertical: Key Space

Range from Blue - White (Silver)
From low to high

Horizontal: Time

Preventing Key Hotspot

- Before expected increase in the traffic we can increase the size of the Bigtable nodes
- Try to do random access all of the whole key available in the bigtable. This way we can make sure the key is rebalanced evenly, also we can try to simulate incoming traffic to see whether the nodes provided are enough.

Simple random access code:

- https://github.com/irvifa/bigtable-load-test

Installing cbt

$ gcloud components update

$ gcloud components install cbt

PS:

- If you're using MacOS:

```
version=$(ls /usr/local/Cellar/openssl)

brew switch openssl $version
```

- Reference for cbt: https://cloud.google.com/bigtable/docs/cbt-reference

Bigtable: Quickstart

Using CBT

```
$ cbt ls
my-table

$ cbt set my-table r1 cf1:c1=my-value

$ cbt read my-table

----------------------------------------
r1
  cf1:c1    @ 2017/02/18-13:20:33.346000
    "my-value"
```

Using HBase Shell

```
hbase(main):001:0> create 'my-table', 'cf1'
...
row(s) in 1.7190 seconds

=> Hbase::Table - my-table

hbase(main):002:0> list
TABLE
my-table
1 row(s) in 0.1560 seconds

=> ["my-table"]
hbase(main):003:0>
```

https://cloud.google.com/bigtable/docs/quickstarts

References

1. Storage Architecture and Challenges
2. SSTable and Log Structured Storage: LevelDB
3. Overview of Key Visualizer
4. Bigtable: A Distributed Storage System for Structured Data
5. Prevent Key Hotspots in Bigtable After Scale-Up
6. Designing your Schema
7. Basic Operations for Bigtable


Q
A