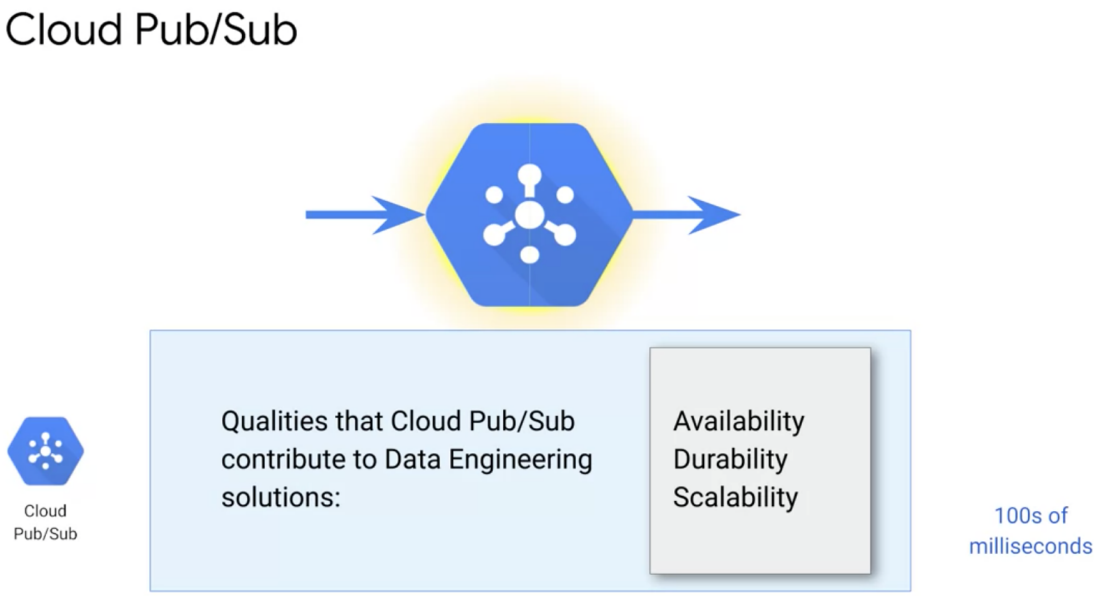
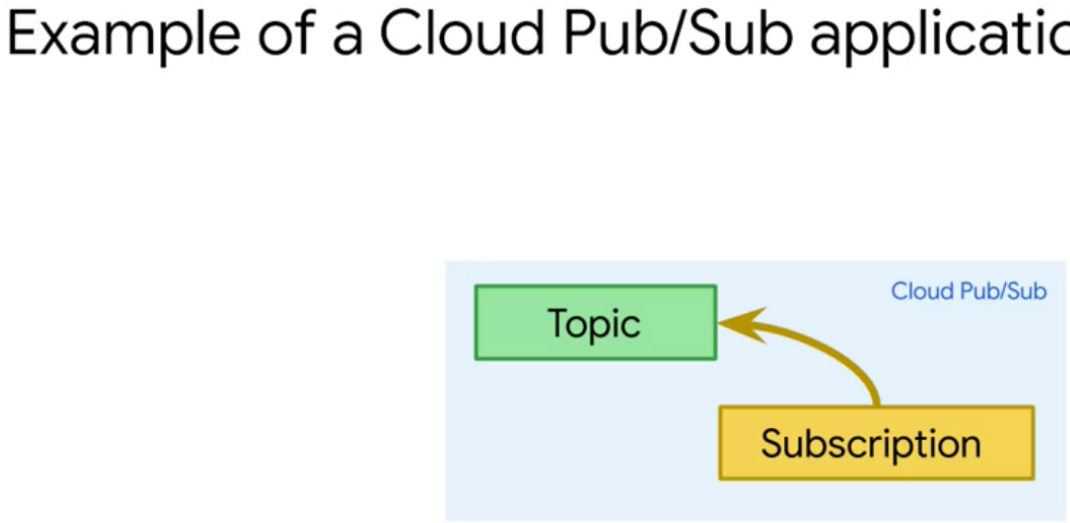
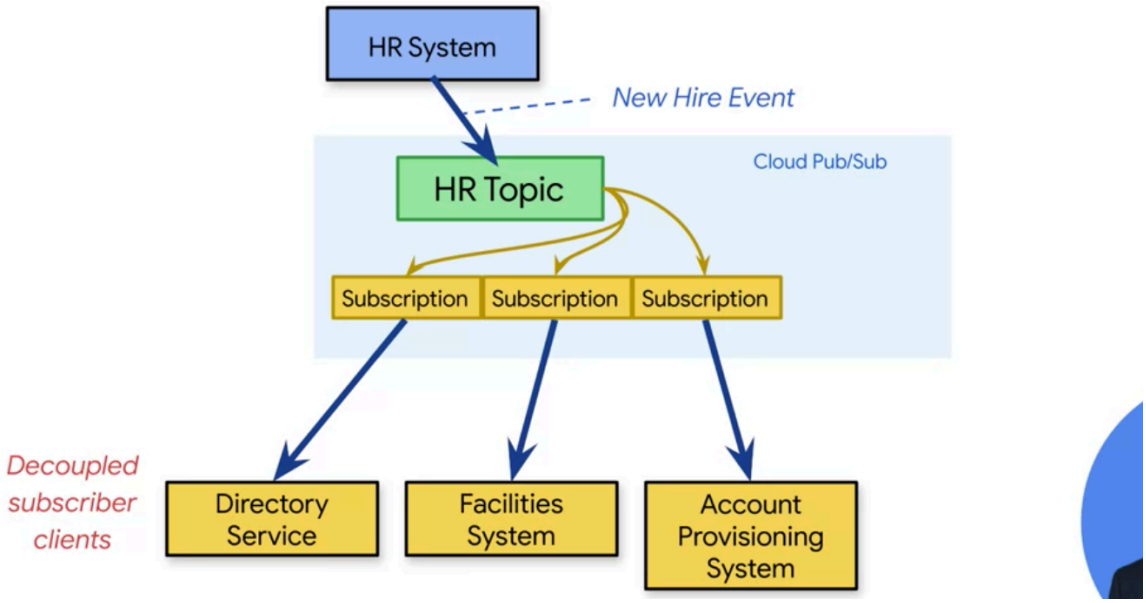


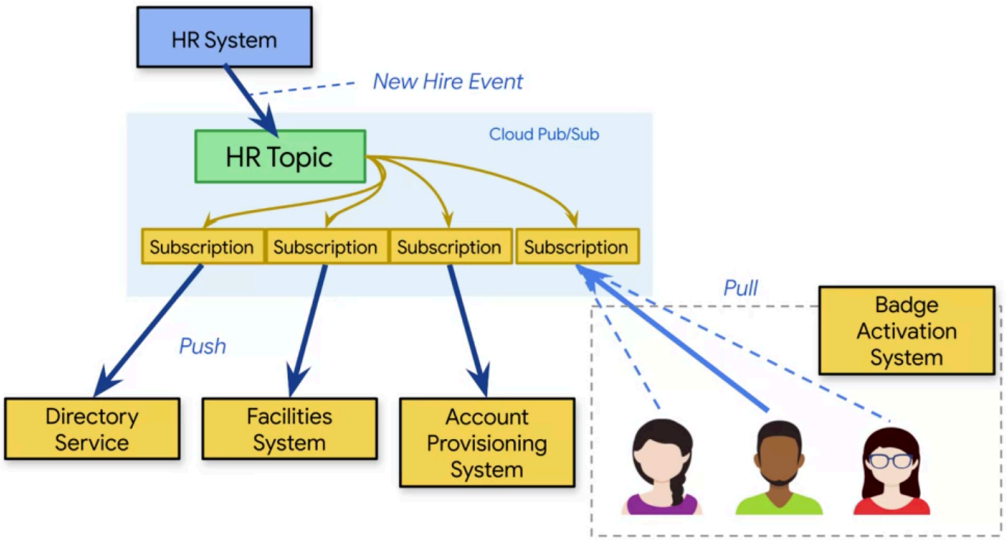


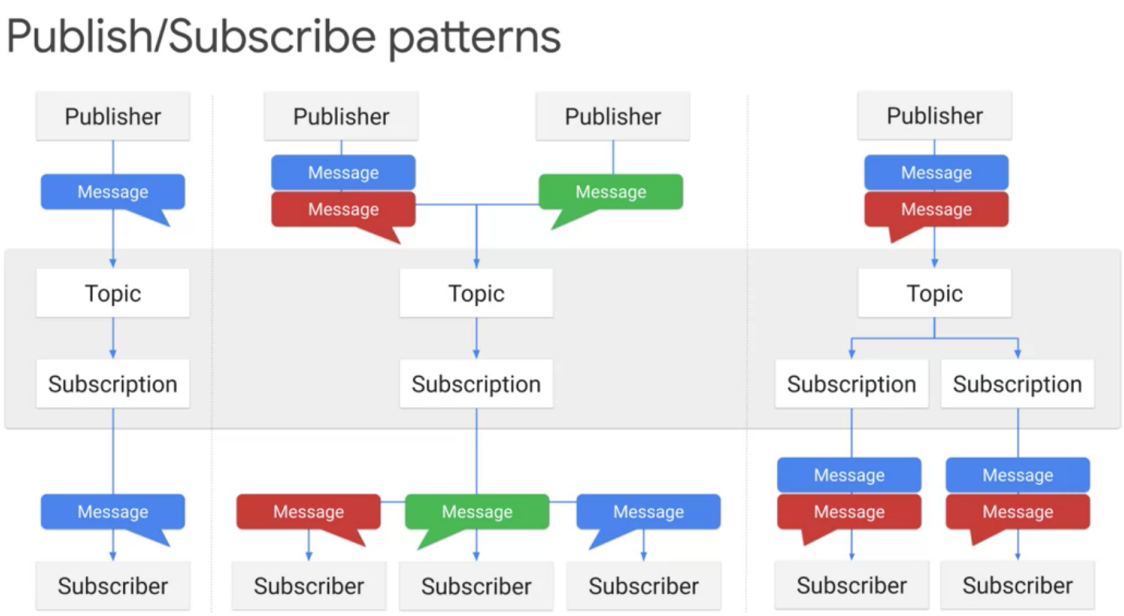
The story of Cloud Pub/Sub is the story of two data structures, the topic and the subscription. Both the topic and the subscription are abstractions which exists in the Pub/Sub framework, independently of any workers, subscribers, or anything else. The Cloud Pub/Sub Client that creates the topic is called the publisher, and the Cloud Pub/Sub Client that creates the subscription is called the subscriber. Publisher will publish events and messages into a topic to be distributed and used for further processing. To receive messages published to a topic, you must create a subscription to that topic.
In this example, the subscription is subscribed to the topic. Only messages published to the topic after the subscription is created are available to subscriber applications. The subscription connects the topic to a subscriber application that receives and processes messages published to the topic. **A topic can have multiple subscriptions, but a given subscription belongs to a single topic.**

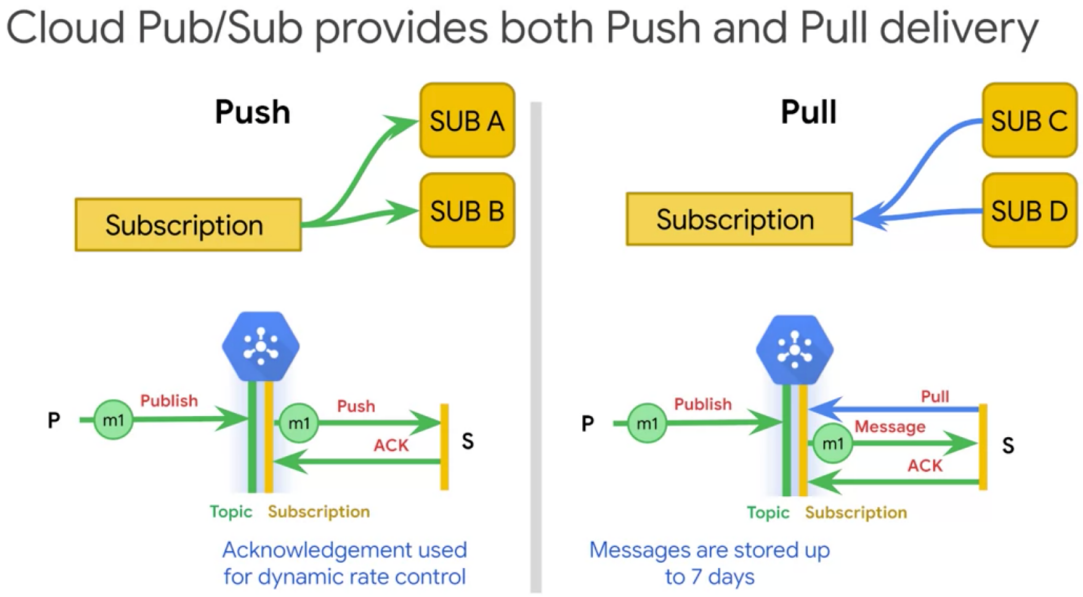
The message is sent from Topic to Subscription



And there can be multiple subscribers per Subscription



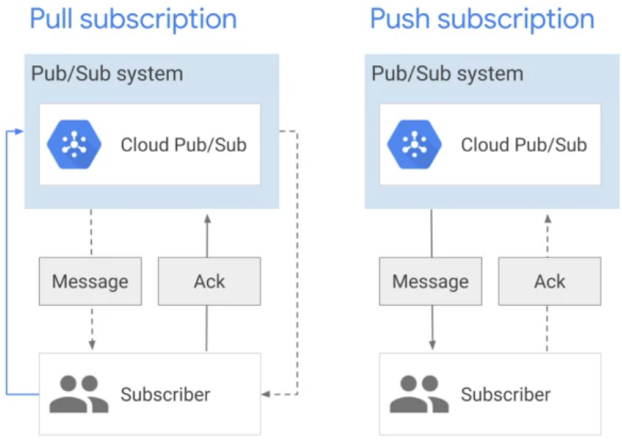




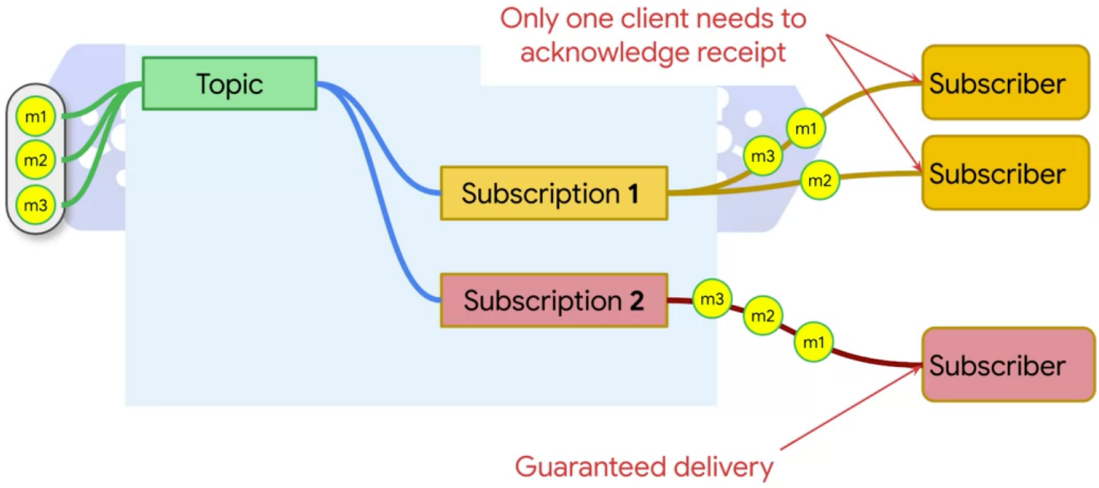
In the pull model, the messages are stored for up to seven days. In push delivery, Cloud Pub/Sub initiates requests your subscriber application, to deliver messages. The Cloud Pub/Sub servers sends each message as an HTTPS request to the subscriber application at a preconfigured endpoint.
In the push scenario, you just respond with status 200 Okay for the HTTP call, and that tells Pub/Sub the message delivery was successful.
Push delivery is ideal when multiple topics must be processed by the same webhook for example.

At least once delivery guarantee

- A subscriber ACKs each message for every subscription
- A message is resent if subscriber takes more than `ackDeadline` to respond
- Messages are stored for up to 7 days
- A subscriber can extend the deadline per message



Subscribers can work as a team or separately



Publishing with Cloud Pub/Sub

```
gcloud pubsub topics create sandiego
```

Create topic

```
gcloud pubsub topics publish sandiego "hello"
```

Publish to topic

```python
import os
from google.cloud import pubsub_v1

publisher = pubsub_v1.PublisherClient()

topic_name = 'projects/{project_id}/topics/{topic}'.format(
    project_id=os.getenv('GOOGLE_CLOUD_PROJECT'),
    topic='MY_TOPIC_NAME',
)

publisher.create_topic(topic_name)
publisher.publish(topic_name, b'My first message!', author='dylan')
```

Python

Create a client

Set topic name

Message

Send attribute

Subscribing with Cloud Pub/Sub using async pull

```python
import os
from google.cloud import pubsub_v1

subscriber = pubsub_v1.SubscriberClient()
topic_name = 'projects/{project_id}/topics/{topic}'.format(
    project_id=os.getenv('GOOGLE_CLOUD_PROJECT'),
    topic='MY_TOPIC_NAME',
)
subscription_name = 'projects/{project_id}/subscriptions/{sub}'.format(
    project_id=os.getenv('GOOGLE_CLOUD_PROJECT'),
    sub='MY_SUBSCRIPTION_NAME',
)
subscriber.create_subscription(
    name=subscription_name, topic=topic_name)

def callback(message):
    print(message.data)
    message.ack()

future = subscriber.subscribe(subscription_name, callback)
```

Python

Create a client

Select topic name

Set subscription name

callback when message received

Callback function

In general, asynchronous pull is preferable for latency sensitive applications.

Subscribing with Cloud Pub/Sub using synchronous pull

```
gcloud pubsub subscriptions create --topic sandiego mySub1
```

Create subscription

```
gcloud pubsub subscriptions pull --auto-ack mySub1
```

Pull subscription

```
import time

from google.cloud import pubsub_v1

subscriber = pubsub_v1.SubscriberClient()

subscription_path = subscriber.subscription_path(project_id, subscription_name)

NUM_MESSAGES = 2
ACK_DEADLINE = 30
SLEEP_TIME = 10

# The subscriber pulls a specific number of messages.
response = subscriber.pull(subscription_path, max_messages=NUM_MESSAGES)
```

Set subscription name

Create a client

`projects/{project_id}/subscriptions/{subscription_name}` ← subscription_path format

Subscriber is non-blocking

Keep the main thread from exiting to allow it to process messages asynchronously

By default, the Publisher batches messages; turn this off if you desire lower latency

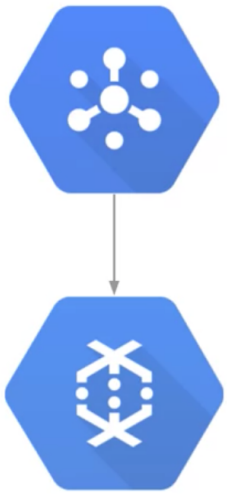
Batching messages: throughput versus latency

Pub/Sub: latency, out-of-order, duplication will happen

- Latency - no guarantees
- Messages can be delivered in any order, especially with large backlog
- Duplication may happen

Cloud Pub/sub with Dataflow: Exactly once, ordered processing

- ✓ Cloud Pub/Sub delivers at least once
- ✓ Cloud Dataflow: Deduplicate, order, and window
- ✓ Separation of concerns → scale



- Dataflow will de-duplicate messages based on the message ID because in Pub/Sub, if a message is delivered twice, it will have the same ID in both cases.
- BigQuery can also be used for this purpose but has limited capabilities.
- Data flow will not be able to order in the sense of providing exact sequential order of when messages were published. However, it will help us deal with late data.
- Using Pub/Sub and Dataflow together allows us to get a scale at that wouldn't be possible otherwise.

Create Pub/Sub topic and subscription

```
git clone https://github.com/GoogleCloudPlatform/training-data-analyst
```

1. On the **training-vm** SSH terminal, navigate to the directory for this lab.

```
cd ~/training-data-analyst/courses/streaming/publish
```

2. Create your topic and publish a simple message.

```
gcloud pubsub topics create sandiego
```

3. Publish a simple message.

```
gcloud pubsub topics publish sandiego --message "hello"
```

4. Create a subscription for the topic.

```
gcloud pubsub subscriptions create --topic sandiego mySub1
```

5. Pull the first message that was published to your topic.

```
gcloud pubsub subscriptions pull --auto-ack mySub1
```

Do you see any result? If not, why?

6. Try to publish another message and then pull it using the subscription.

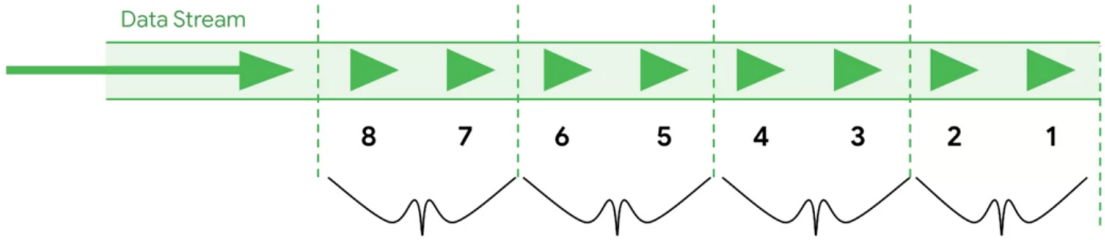
```
gcloud pubsub topics publish sandiego --message "hello again"
gcloud pubsub subscriptions pull --auto-ack mySub1
```

Did you get any response this time?

Output:

DATA	MESSAGE_ID	ATTRIBUTES
hello again	38138015771622	

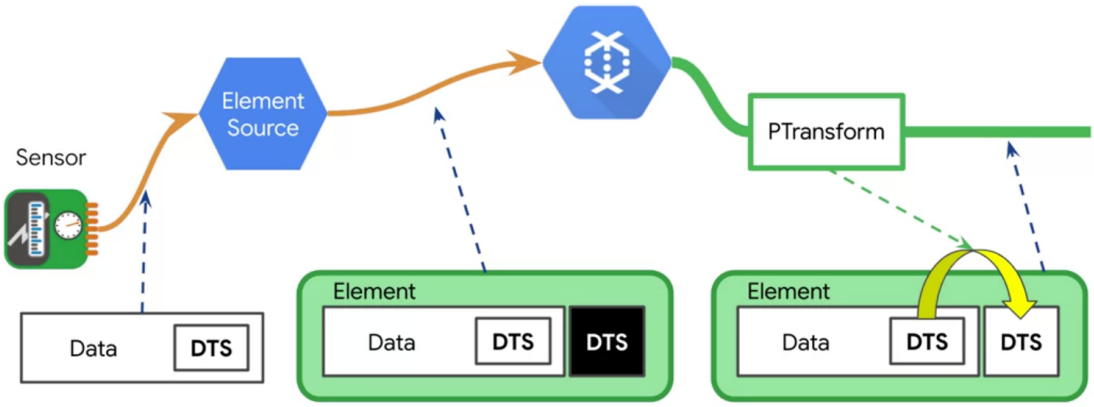
Divide the stream info a series of finite windows



Windows are defined by element date-timestamp (metadata, usually Pub/Sub ingest time)

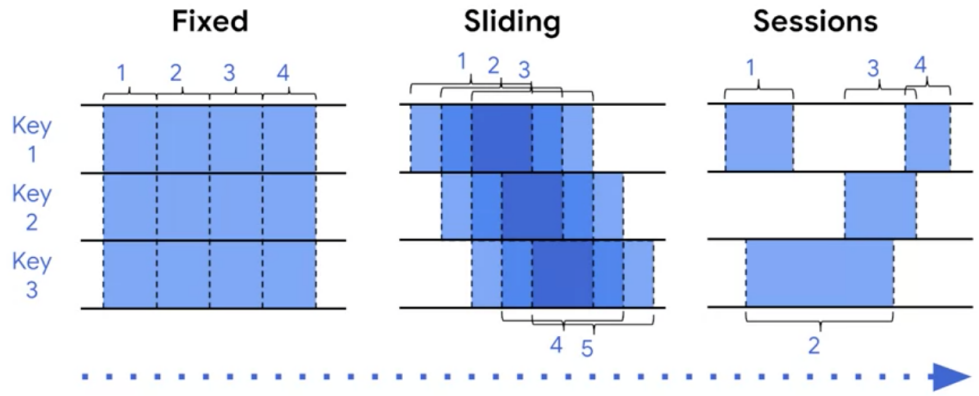
$$\sum_{i=w}^{n} x_i \qquad \sum_{i=w}^{n} x_i \qquad \sum_{i=w}^{n} x_i$$

Modify the date-timestamp with a PTransform if needed



Cloud Dataflow Windowing

Three kinds of windows fit most circumstances



Windowing divides data into time-based finite chunks

Often required when doing aggregations over unbounded data

Setting time windows

Remember:

you can apply windows to batch data, although you may need to generate the metadata date-timestamp on which windows operate.

Fixed-time windows

```
from apache_beam import window                                    Python
fixed_windowed_items = (
    items | 'window' >> beam.WindowInto(window.FixedWindows(60)))
```

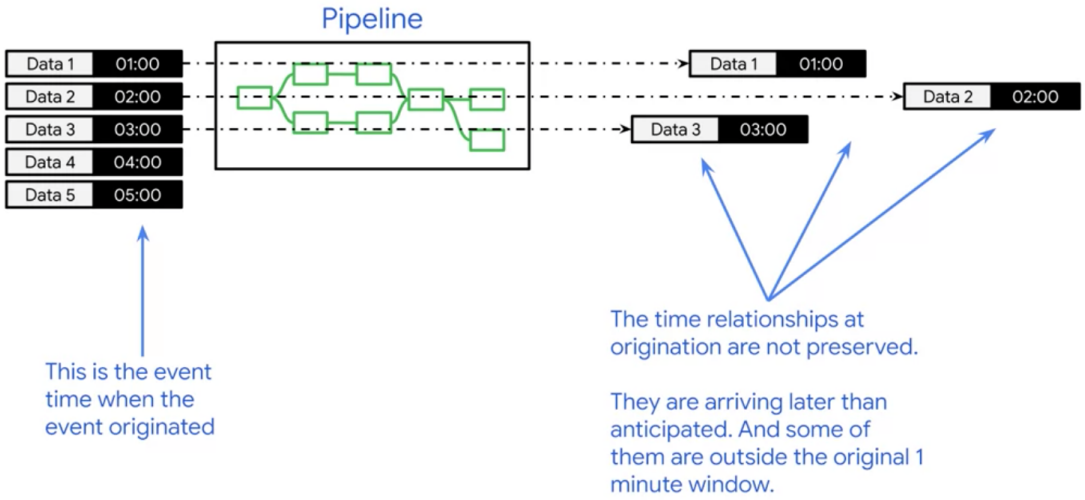
Sliding time windows

```
from apache_beam import window                                    Python
sliding_windowed_items = (
    items | 'window' >> beam.WindowInto(window.SlidingWindows(30, 5)))
```

Session windows

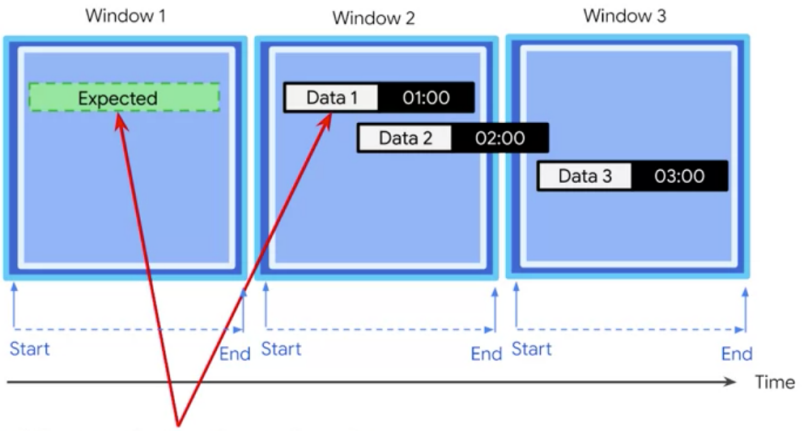
```
from apache_beam import window                                    Python
session_windowed_items = (
    items | 'window' >> beam.WindowInto(window.Sessions(10 * 60)))
```

Pipeline processing can introduce latency



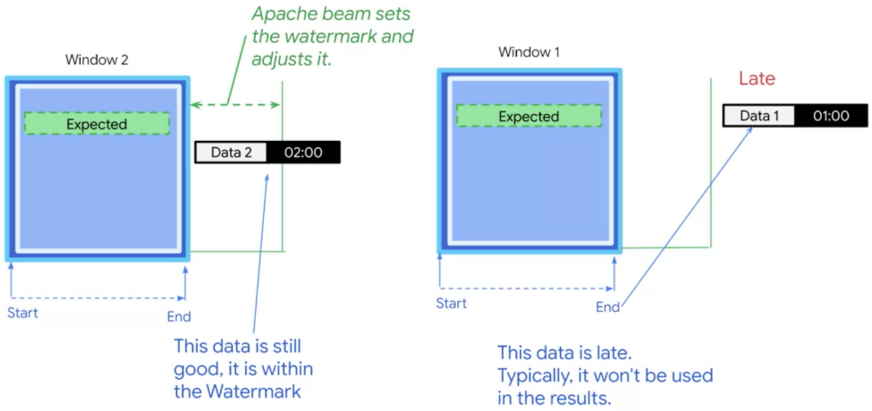
How should Cloud Dataflow deal with handling latency?

The data could be a little past the window or a lot. Data 2 is a little outside of Window 2. Data 1 is completely outside of Window 1.

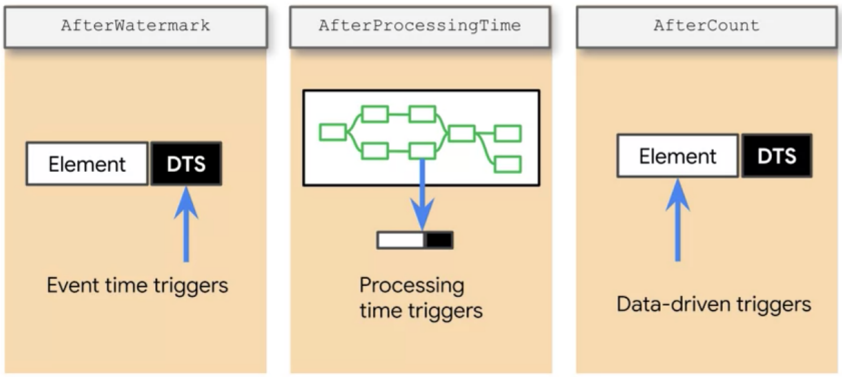


The difference in time from when data was expected to when it actually arrived is called the **lag time**.

Watermarks provide flexibility for a little lag time



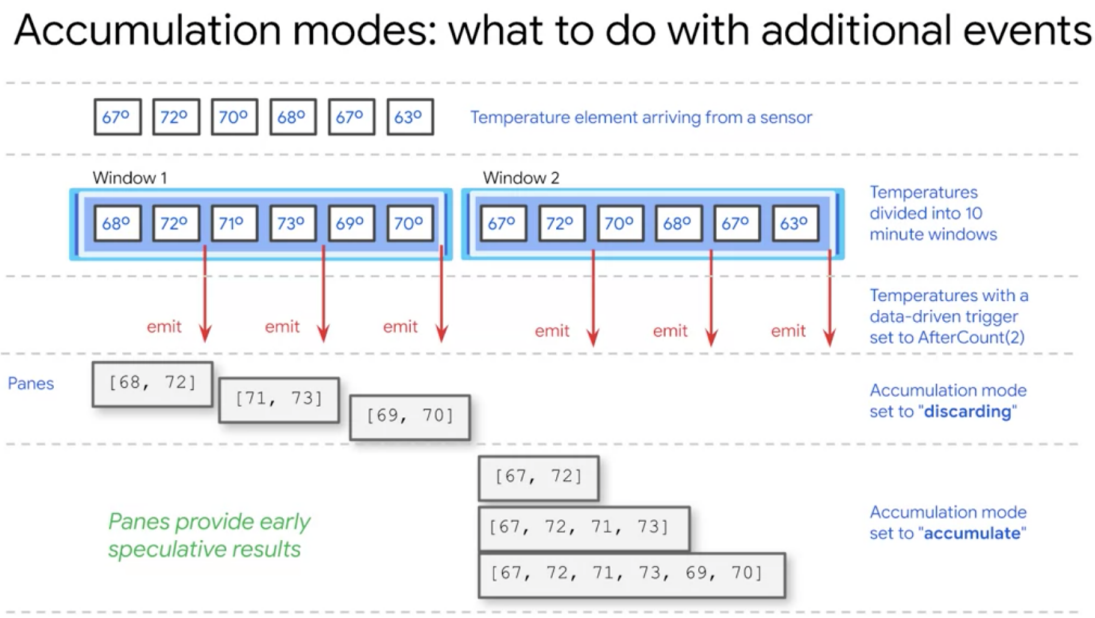
The default is to trigger at the watermark, but we can also add custom trigger(s)



Some example triggers

```
pcollection | WindowInto(
    SlidingWindows(60, 5),                          # Sliding window of 60 seconds, every 5 seconds
    trigger=AfterWatermark(                         # Relative to the watermark, trigger:
      early=AfterProcessingTime(delay=30),          #     -- 30 seconds early
      late=AfterCount(1))                           #     -- and for every late record (< allowedLateness)
    accumulation_mode=AccumulationMode.ACCUMULATING)  # the pane should have all the records
```

```
pcollection | WindowInto(
    FixedWindows(60),                               # Fixed window of 60 seconds
    trigger=Repeatedly(                             # Set up a composite trigger that triggers ...
        AfterAny(                                   # whenever either of these happens:
            AfterCount(100),                        #   -- 100 elements accumulate
            AfterProcessingTime(1 * 60))),          #   -- every 60 seconds (ignore watermark)
    accumulation_mode=AccumulationMode.DISCARDING)  # the trigger should be with only new records
```



To Create a dataflow pipeline =>

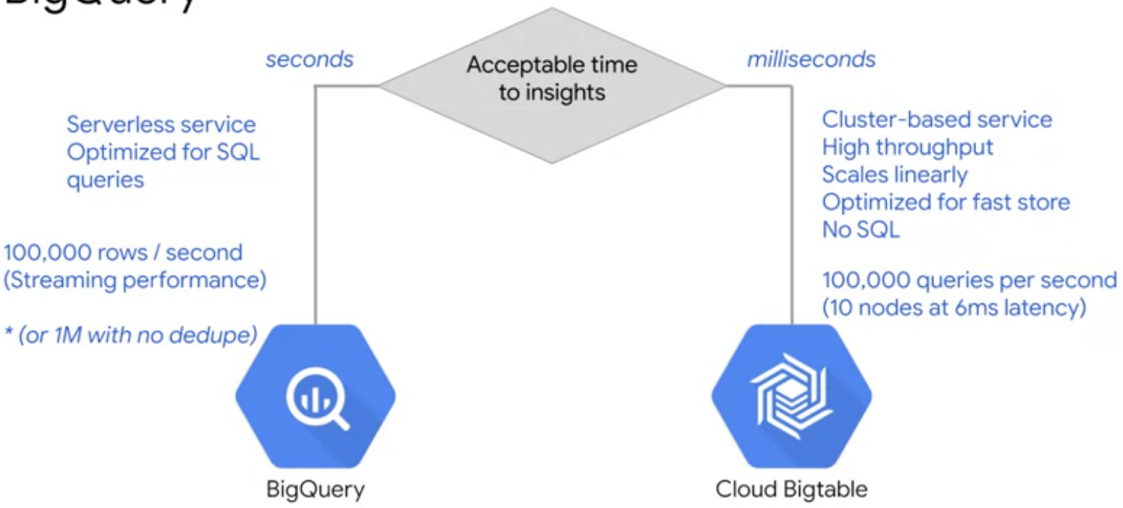
https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/courses/streaming/process/sandiego/run_oncloud.sh

Example Dataflow pipelines =>

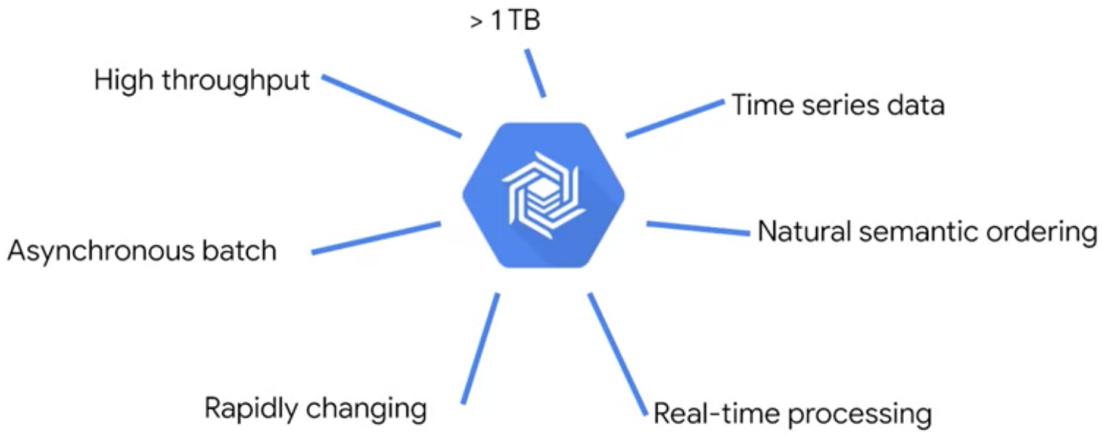
https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/courses/streaming/process/sandiego/src/main/java/com/google/cloud/training/dataanalyst/sandiego/AverageSpeeds.java

https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/courses/streaming/process/sandiego/src/main/java/com/google/cloud/training/dataanalyst/sandiego/CurrentConditions.java

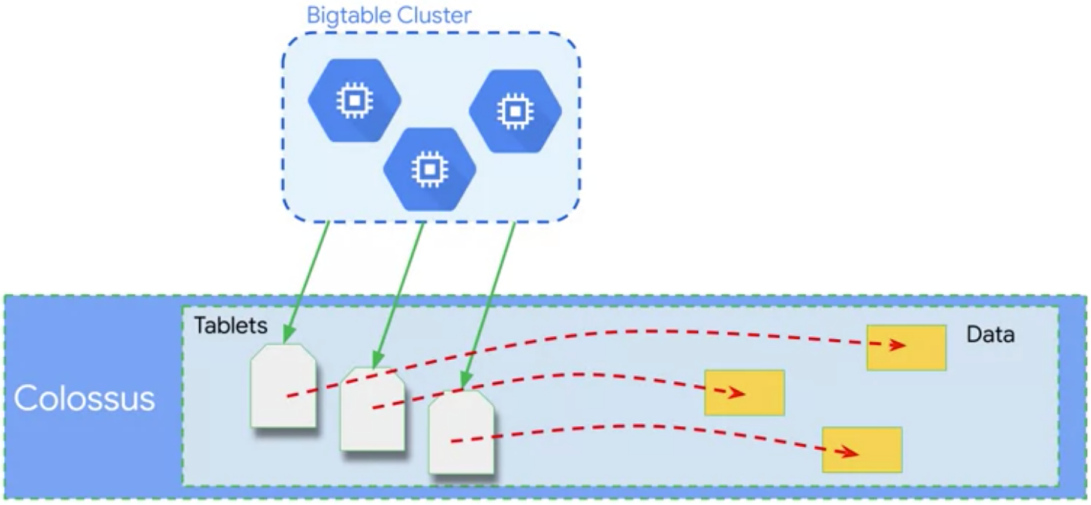
How to choose between Cloud Bigtable and BigQuery



Consider Cloud Bigtable for these requirements



How does Cloud Bigtable work?



Cloud Bigtable design idea is “simplify for speed”

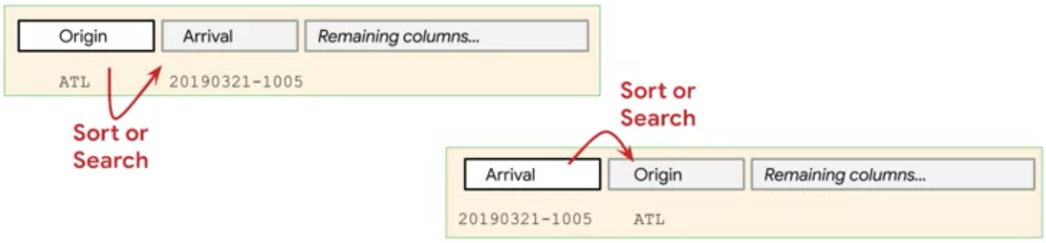
The Row Key is the index.

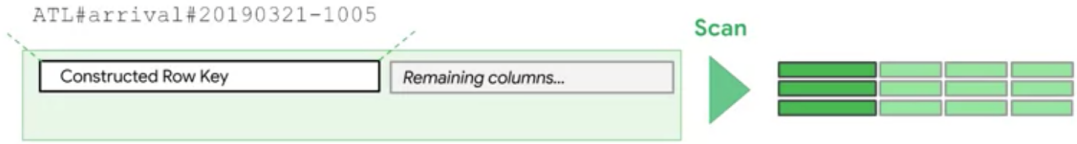
And you get only one.

But speed depends on your data and Row Key

What is the best Row Key?

Query: All flights originating in Atlanta and arriving between March 21st and 29th





Cloud Bigtable schema organization

Column Families

Row Key	Flight_Information					Aircraft_Information			
	Origin	Destination	Departure	Arrival	Passengers	Capacity	Make	Model	Age
ATL#arrival#20190321-1121	ATL	LON	20190321-0311	20190321-1121	158	162	B	737	18
ATL#arrival#20190321-1201	ATL	MEX	20190321-0821	20190321-1201	187	189	B	737	8
ATL#arrival#20190321-1716	ATL	YVR	20190321-1014	20190321-1716	201	259	B	757	23

Queries that use the row key, a row prefix, or a row range are the most efficient

Query: Current arrival delay for flights from Atlanta

1

ROW KEY BASED ON ATLANTA ARRIVALS
E.G. ORIGIN#arrival
(ATL#arrival#20190321-1005)

Puts latest flights at bottom of table

2

REVERSE TIMESTAMP TO THE ROWKEY
E.G. ORIGIN#arrival#RTS
(ATL#arrival#12345678)

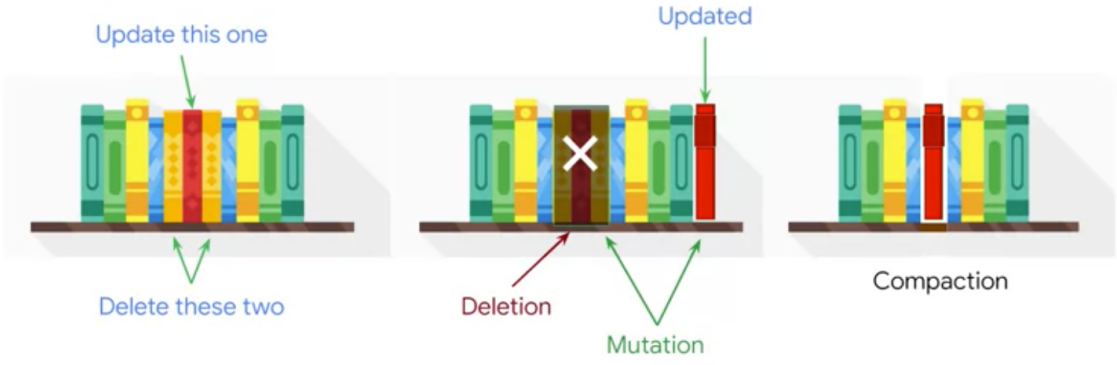
Puts latest flights at top of table

Use reverse timestamps when your most common query is for the latest values

Query: Current arrival delay for flights from Atlanta

```
// key is ORIGIN#arrival#REVTS
String key = info.getORIGIN() //
   + "#arrival"  //
   + "#" + (Long.MAX_VALUE - ts.getMillis()); // reverse timestamp
```

What happens when data in Cloud Bigtable is changed?



Optimizing data organization for performance

Group related data for more efficient reads

Example row key:

```
DehliIndia#2019031411841
```

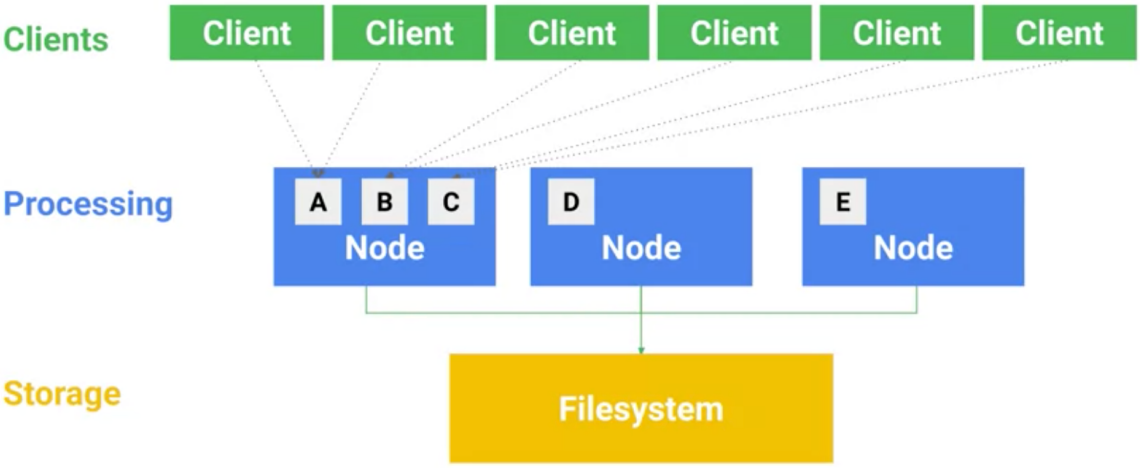
Use column families

Distribute data evenly for more efficient writes

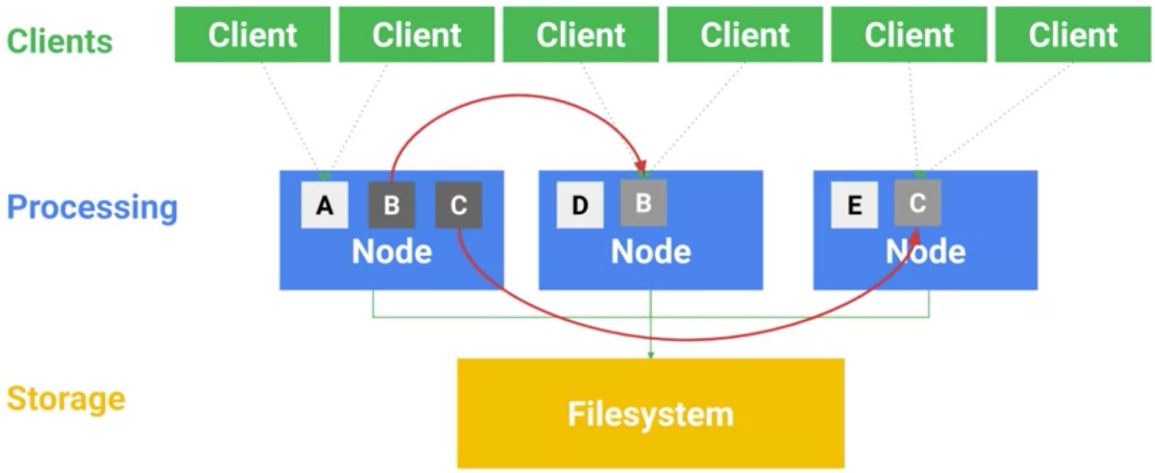
Place identical values in the same row or adjoining rows for more efficient compression

Use row keys to organize identical data

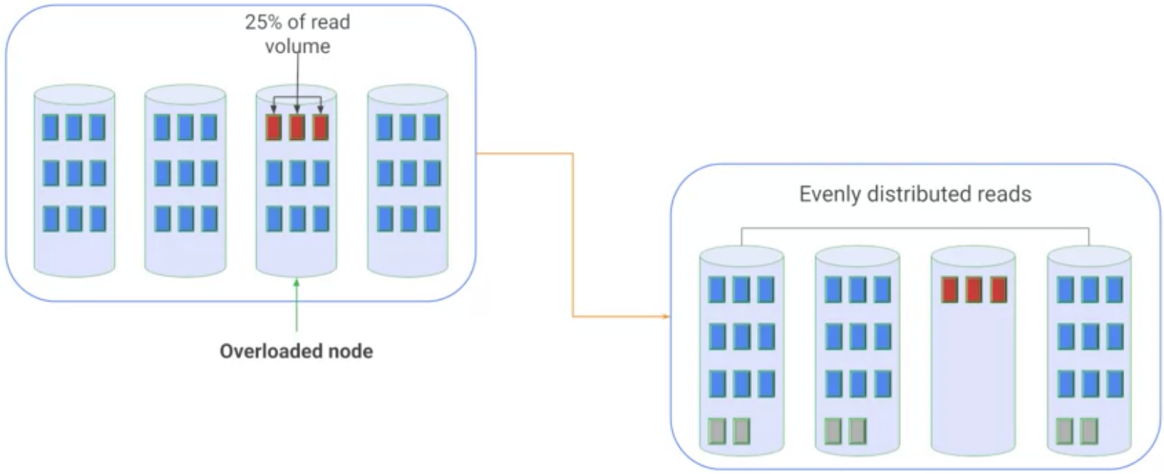
Cloud Bigtable self-improves by learning access patterns...



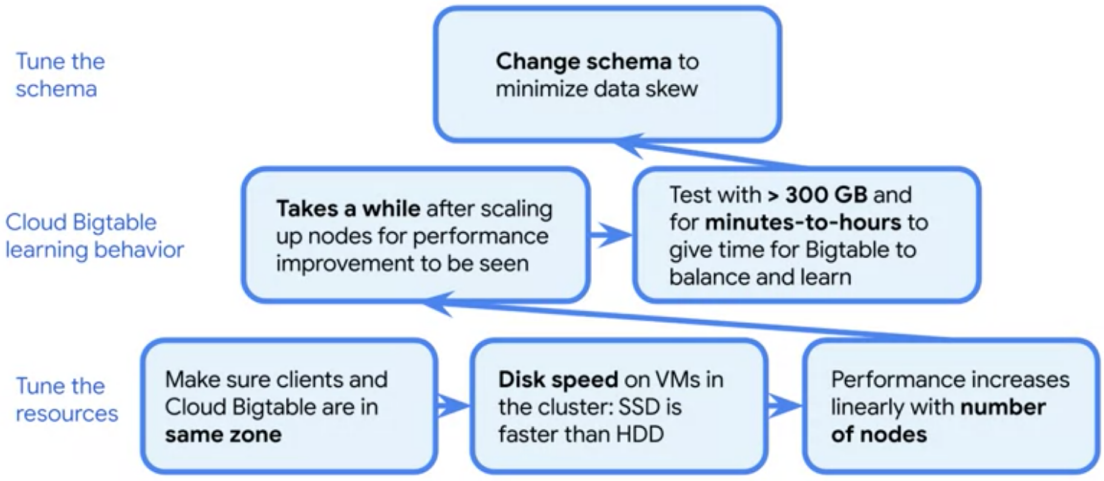
...and rebalances data accordingly



Rebalance strategy: distribute reads



Optimizing Cloud Bigtable Performance



Use Cloud Bigtable replications to improve availability

Why perform replication?

- Isolate serving applications from batch reads
- Improve availability
- Provide near-real-time backup
- Ensure your data has a global presence

```
gcloud bigtable clusters create CLUSTER_ID \
    --instance=INSTANCE_ID \
    --zone=ZONE \
    [--num-nodes=NUM_NODES] \
    [--storage-type=STORAGE_TYPE]
```

GIS Functions =>

GIS Functions

You can use STRUCT() to organize columns from different tables

```
SELECT
  STRUCT(
    start_stn.name,
    start_stn.longitude,
    start_stn.latitude,
    start_stn.docks_count,
    start_stn.install_date
  ) AS starting,
  STRUCT(
    end_stn.name,
    end_stn.longitude,
    end_stn.latitude,
    end_stn.docks_count,
    end_stn.install_date
  ) AS ending,
  STRUCT(
    rental_id,
    bike_id,
    duration, -- seconds
    -- placeholder: Distance ?
    start_date,
    end_date
  ) AS bike
FROM `bigquery-public-data.london_bicycles.cycle_stations` AS start_stn
LEFT JOIN `bigquery-public-data.london_bicycles.cycle_hire` AS b
ON start_stn.id = b.start_station_id
LEFT JOIN `bigquery-public-data.london_bicycles.cycle_stations` AS end_stn
ON end_stn.id = b.end_station_id
```

- STRUCT() are a SQL data type that are simply containers for columns. Super useful for readability for really wide and deep schemas like Google Analytics
- Why are there three tables in the JOIN condition?

STRUCT and their nested column names are separated with a dot '.'

Query complete (10.8 sec elapsed, 1.3 GB processed)

Job information | Results | JSON | Execution details

Row	starting.longitude	starting.latitude	ending.longitude	ending.latitude	bike.rental_id	bike.bike_id	bike.duration	bike.start_date	bike.end_date
1	-0.122759625866	51.4878074485	-0.121394101	51.49369988	56894333	1307	3360	2016-08-07 00:11:00 UTC	2016-08-07 01:07:00 UTC
2	-0.122759625866	51.4878074485	-0.09968067	51.49337264	64108629	11022	1560	2017-04-16 13:28:00 UTC	2017-04-16 13:54:00 UTC
3	-0.122759625866	51.4878074485	-0.130431727	51.52168078	62078071	14045	1380	2017-02-01 10:25:00 UTC	2017-02-01 10:48:00 UTC
4	-0.122759625866	51.4878074485	-0.117286	51.486575	56595963	11411	540	2016-07-30 16:56:00 UTC	2016-07-30 17:05:00 UTC
5	-0.122759625866	51.4878074485	-0.116911864	51.4908679	64360044	5028	120	2017-04-24 17:55:00 UTC	2017-04-24 17:57:00 UTC
6	-0.122759625866	51.4878074485	-0.116542278	51.5046364	55716706	9574	1320	2016-07-08 20:16:00 UTC	2016-07-08 20:38:00 UTC
7	-0.122759625866	51.4878074485	-0.119047563	51.50963123	60064076	5354	840	2016-11-04 08:36:00 UTC	2016-11-04 08:50:00 UTC
8	-0.122759625866	51.4878074485	-0.113936001	51.50013942	65340664	6540	600	2017-05-25 09:35:00 UTC	2017-05-25 09:45:00 UTC
9	-0.122759625866	51.4878074485	-0.132102166	51.49812559	56160135	12591	540	2016-07-20 09:10:00 UTC	2016-07-20 09:19:00 UTC
10	-0.122759625866	51.4878074485	-0.158456089	51.50295379	65575024	12735	1140	2017-05-31 14:23:00 UTC	2017-05-31 14:42:00 UTC

Starting Station

Ending Station

Bike rental and trip details

Many separate tables joined into one is a process called **denormalization** and it's generally a BigQuery best practice

Calculating distance between geopoints with ST_DISTANCE()

Query complete (10.8 sec elapsed, 1.3 GB processed)

Job information Results JSON Execution details

Row	starting.longitude	starting.latitude	ending.longitude	ending.latitude	bike.rental_id	bike.bike_id	bike.duration
1	-0.122759625866	51.4878074485	-0.121394101	51.49369988	56894333	1307	3360

```
SELECT
ST_DISTANCE(
 ST_GEOGPOINT(-0.122759625866, 51.4878074485), -- starting long/lat
 ST_GEOGPOINT(-0.121394101, 51.49369988)       -- ending long/lat
) AS distance_meters
```

Job information Results

Row	distance_meters
1	661.9951844869499

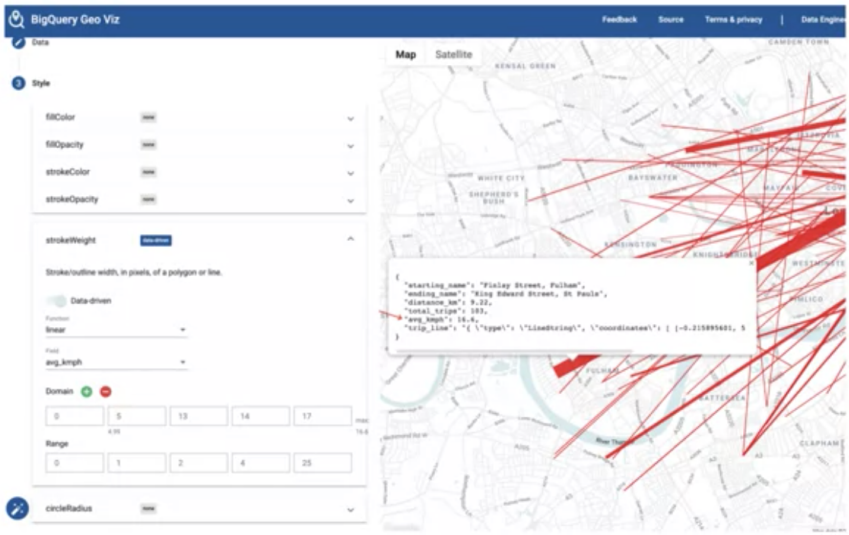
- ST_DISTANCE() gives you the straight line distance between two geographic points in meters.
- We must first convert our lat/long with ST_GEOGPOINT()
- Tip: You can use ST_MAKELINE() to draw a line between those two points on a GIS visualization tool (more on this later)

Creating the final query with aggregation functions

```
-- Find the fastest avg biking pace for rides over 30 mins
SELECT
  starting.name AS starting_name,
  ending.name AS ending_name,
  ROUND(bike.distance/1000,2) distance_km,
  COUNT(bike.rental_id) AS total_trips,
  ROUND(
      AVG(
        (bike.distance / 1000) -- meters --> km
        / (bike.duration / 60 / 60) -- seconds --> hours
      )
    ,2)
  AS avg_kmph
FROM staging
WHERE bike.duration > (30 * 60) -- at least 30 minutes = 1800 seconds
GROUP BY
  starting.name,
  ending.name,
  bike.distance
HAVING total_trips > 100
ORDER BY avg_kmph DESC
 LIMIT 100;
```

- Notice the conversions of meters to kilometers and duration to hours
- We also filter out trips less than 30 minutes in length and stations that have 100 or fewer total trips
- The data itself is pulled from a 'staging' table which we will discuss soon (hint: it's a WITH clause that came before this query)

Visualizing the fastest bike commuter station pairs with BigQuery GeoViz



- BigQuery GeoViz UI
- Useful for visualizing geographic insights
- You specify your query and project to pull in data
- You can style the points / lines by weight etc.

Use WITH clauses to stage complex analysis

```sql
WITH staging AS (
  SELECT
    STRUCT(
      start_stn.name,
      ST_GEOGPOINT(start_stn.longitude, start_stn.latitude) AS point,
      start_stn.docks_count,
      start_stn.install_date
    ) AS starting,
    STRUCT(
      end_stn.name,
      ST_GEOGPOINT(end_stn.longitude, end_stn.latitude) AS point,
      end_stn.docks_count,
      end_stn.install_date
    ) AS ending,
    STRUCT(
      rental_id,
      bike_id,
      duration, -- seconds
      ST_DISTANCE(
        ST_GEOGPOINT(start_stn.longitude, start_stn.latitude),
        ST_GEOGPOINT(end_stn.longitude, end_stn.latitude)
        ) AS distance, -- meters
      ST_MAKELINE(
        ST_GEOGPOINT(start_stn.longitude, start_stn.latitude),
        ST_GEOGPOINT(end_stn.longitude, end_stn.latitude)
        ) AS trip_line, -- straight line (for GeoViz)
      start_date,
      end_date
    ) AS bike
  FROM `bigquery-public-data.london_bicycles.cycle_stations` AS start_stn
  LEFT JOIN `bigquery-public-data.london_bicycles.cycle_hire` AS b
  ON start_stn.id = b.start_station_id
  LEFT JOIN `bigquery-public-data.london_bicycles.cycle_stations` AS end_stn
  ON end_stn.id = b.end_station_id
)
```

- We needed to preprocess all of our raw data with JOINs and GIS functions before our last analysis
- You can use WITH clauses to create common table expressions which are **named subqueries** (no data is stored)
- You can chain multiple WITH clause names together with commas ‘,’

Should you always store WITH clause as a table?

```sql
WITH staging AS (
  SELECT
    STRUCT(
      start_stn.name,
      ST_GEOGPOINT(start_stn.longitude, start_stn.latitude) AS point,
      start_stn.docks_count,
      start_stn.install_date
    ) AS starting,
    STRUCT(
      end_stn.name,
      ST_GEOGPOINT(end_stn.longitude, end_stn.latitude) AS point,
      end_stn.docks_count,
      end_stn.install_date
    ) AS ending,
    STRUCT(
      rental_id,
      bike_id,
      duration, -- seconds
      ST_DISTANCE(
        ST_GEOGPOINT(start_stn.longitude, start_stn.latitude),
        ST_GEOGPOINT(end_stn.longitude, end_stn.latitude)
        ) AS distance, -- meters
      ST_MAKELINE(
        ST_GEOGPOINT(start_stn.longitude, start_stn.latitude),
        ST_GEOGPOINT(end_stn.longitude, end_stn.latitude)
        ) AS trip_line, -- straight line (for GeoViz)
      start_date,
      end_date
    ) AS bike
  FROM `bigquery-public-data.london_bicycles.cycle_stations` AS start_stn
  LEFT JOIN `bigquery-public-data.london_bicycles.cycle_hire` AS b
  ON start_stn.id = b.start_station_id
  LEFT JOIN `bigquery-public-data.london_bicycles.cycle_stations` AS end_stn
  ON end_stn.id = b.end_station_id
)
```

- Should we replace WITH with just CREATE OR REPLACE TABLE AS ?
- PROs for Permanent tables:
  - Speed - preprocess the data once
  - Shared insight - give others can query your new table
- CONs
  - May process more data than you need (can't do predicate pushdown)
  - Readability - it's not always clear what preprocessing steps were taken

Review our overall analysis approach

Goal: Stations with the fastest and slowest bike turnover times

Analyze the schema

1. **cycle_hire** has **start_date** and **end_date** for each bike (timestamp)
2. **cycle_hire** has the starting and ending bike station ids
3. **cycle_station** has the count of total bike docks available (capacity)

Plan the query

1. JOIN the two datasets
2. Track the previous time a given bike_id was returned before it was checked out again
3. Calculate the timestamp difference between last return and this checkout
4. Aggregate by each station
5. Filter the data as necessary to remove anomalies

Use navigation functions (LAG, LEAD) with window functions to navigate to the previous return timestamp for a given bike_id

```
lag_end_date AS (
-- Find how long after one ride ends,
-- another one begins (on average)
SELECT
  starting.name,
  starting.docks_count,
  starting.install_date,
  bike.bike_id,
  LAG(bike.end_date) OVER (
    PARTITION BY bike.bike_id
    ORDER BY bike.start_date)
  AS last_end_date,
  bike.start_date,
  bike.end_date
FROM staging
)
```

- We use a LAG() function to go back one record in the dataset and pull the previous value for bike.end_date
- We are using an analytical window function to logically partition our data so it's meaningful when we look one record backward

LAG() defaults to previous 1 record in the dataset. The last_end_date for the current row is the previous record's end_date

Query complete (2.4 sec elapsed, 926.1 MB processed)

Job information | Results | JSON | Execution details

Row	name	docks_count	install_date	bike_id	start_date	end_date	last_end_date
1	Gloucester Road Station, South Kensington	15	2011-03-09	9842	2017-06-13 19:39:00 UTC	2017-06-13 19:49:00 UTC	2017-06-13 19:16:00 UTC
2	Hyde Park Corner, Hyde Park	36	2010-07-17	9842	2017-06-13 19:09:00 UTC	2017-06-13 19:16:00 UTC	2017-06-13 18:53:00 UTC
3	Black Lion Gate, Kensington Gardens	24	2010-07-22	9842	2017-06-13 18:32:00 UTC	2017-06-13 18:53:00 UTC	2017-06-13 18:05:00 UTC
4	Green Park Station, Mayfair	28	null	9842	2017-06-13 17:46:00 UTC	2017-06-13 18:05:00 UTC	2017-06-13 16:10:00 UTC
5	Storey's Gate, Westminster	21	2014-02-10	9842	2017-06-13 15:49:00 UTC	2017-06-13 16:10:00 UTC	2017-06-13 13:56:00 UTC
6	Rochester Row, Westminster	13	2010-07-14	9842	2017-06-13 13:47:00 UTC	2017-06-13 13:56:00 UTC	2017-06-13 08:57:00 UTC
7	Simpson Street, Clapham Junction	25	2013-10-23	9842	2017-06-13 08:39:00 UTC	2017-06-13 08:57:00 UTC	2017-06-12 20:40:00 UTC
8	Austin Road, Battersea Park	24	2013-10-28	9842	2017-06-12 20:27:00 UTC	2017-06-12 20:40:00 UTC	2017-06-12 17:34:00 UTC

Use analytical window functions like OVER(PARTITION BY) when you need to analyze a subset or “window” of data

Query complete (2.4 sec elapsed, 926.1 MB processed)

Job information | Results | JSON | Execution details

Row	name	docks_count	install_date	bike_id	start_date	end_date	last_end_date
1	Gloucester Road Station, South Kensington	15	2011-03-09	9842	2017-06-13 19:39:00 UTC	2017-06-13 19:49:00 UTC	2017-06-13 19:16:00 UTC
2	Hyde Park Corner, Hyde Park	36	2010-07-17	9842	2017-06-13 19:09:00 UTC	2017-06-13 19:16:00 UTC	2017-06-13 18:53:00 UTC
3	Black Lion Gate, Kensington Gardens	24	2010-07-22	9842	2017-06-13 18:32:00 UTC	2017-06-13 18:53:00 UTC	2017-06-13 18:05:00 UTC
4	Green Park Station, Mayfair	28	null	9842	2017-06-13 17:46:00 UTC	2017-06-13 18:05:00 UTC	2017-06-13 16:10:00 UTC
5	Storey's Gate, Westminster	21	2014-02-10	9842	2017-06-13 15:49:00 UTC	2017-06-13 16:10:00 UTC	2017-06-13 13:56:00 UTC
6	Rochester Row, Westminster	13	2010-07-14	9842	2017-06-13 13:47:00 UTC	2017-06-13 13:56:00 UTC	2017-06-13 08:57:00 UTC
7	Simpson Street, Clapham Junction	25	2013-10-23	9842	2017-06-13 08:39:00 UTC	2017-06-13 08:57:00 UTC	2017-06-12 20:40:00 UTC
8	Austin Road, Battersea Park	24	2013-10-28	9842	2017-06-12 20:27:00 UTC	2017-06-12 20:40:00 UTC	2017-06-12 17:34:00 UTC

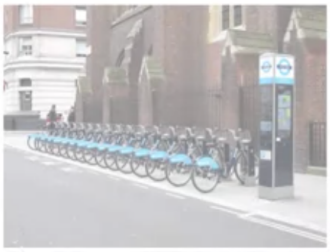
Finally, take the TIMESTAMP_DIFF and aggregate across all bike station names

```
SELECT
  name,
  docks_count,
  install_date,
  COUNT(bike_id) AS total_trips,
  ROUND(
    AVG(
      TIMESTAMP_DIFF(
       start_date, last_end_date, HOUR)
      )
  ,2) AS time_at_station_hrs
FROM lag_end_date
GROUP BY
  name,
  docks_count,
  install_date
HAVING total_trips > 0
ORDER BY time_at_station_hrs ASC -- fastest turnover first
 LIMIT 10;
```

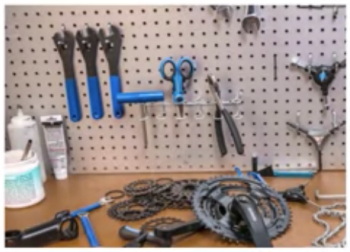
Row	name	docks_count	install_date	total_trips	time_at_station_hrs
1	Wormwood Street, Liverpool Street	16	2010-07-19	118444	1.15
2	Hyde Park Corner, Hyde Park	36	2010-07-17	213568	1.66
3	South Kensington Station, South Kensington	15	2010-10-25	74467	1.86
4	Tooley Street, Bermondsey	17	2010-07-21	68854	1.88
	.....				
764	Manfred Road, East Putney	29	2013-12-06	3950	64.38
765	St. John's Park, Cubitt Town	30	2012-01-30	4434	71.48
766	Teviot Street, Poplar	33	null	3851	90.2
767	Colet Gardens, Hammersmith	30	null	3001	95.44
768	Here East South, Queen Elizabeth Olympic Park	28	2017-03-09	267	161.11

3) Rank which bikes need maintenance the most based on usage metrics

Finding the fastest bike share commuters in London (avg kmph between stations)



Find the stations with the fastest and slowest bike turnover times



Rank which bikes need maintenance the most based on usage metrics

Combine RANK with window functions for multiple rankings within a dataset

```
SELECT
  bike.bike_id,

STRUCT(
  RANK() OVER(
    PARTITION BY bike.bike_id
    ORDER BY bike.start_date
    ) AS current_trip_number,

  SUM(bike.duration/60/60) OVER(
    PARTITION BY bike.bike_id
    ORDER BY bike.start_date
    ) AS cumulative_duration_hr,

  SUM(bike.distance/1000) OVER(
    PARTITION BY bike.bike_id
    ORDER BY bike.start_date
    ) AS cumulative_distance_km,

  bike.start_date,
  bike.end_date
  ) AS stats
 FROM staging
```

- We want to collect how much use each bike has received and prioritize maintenance
- Let's RANK every trip each bike took based on timestamp
- Let's SUM common metrics for **duration** and **distance**
- We'll store these metrics in a STRUCT called stats

Challenge: What if we wanted to rank each bike against all other bikes AND we also wanted granular details for every trip each bike took?

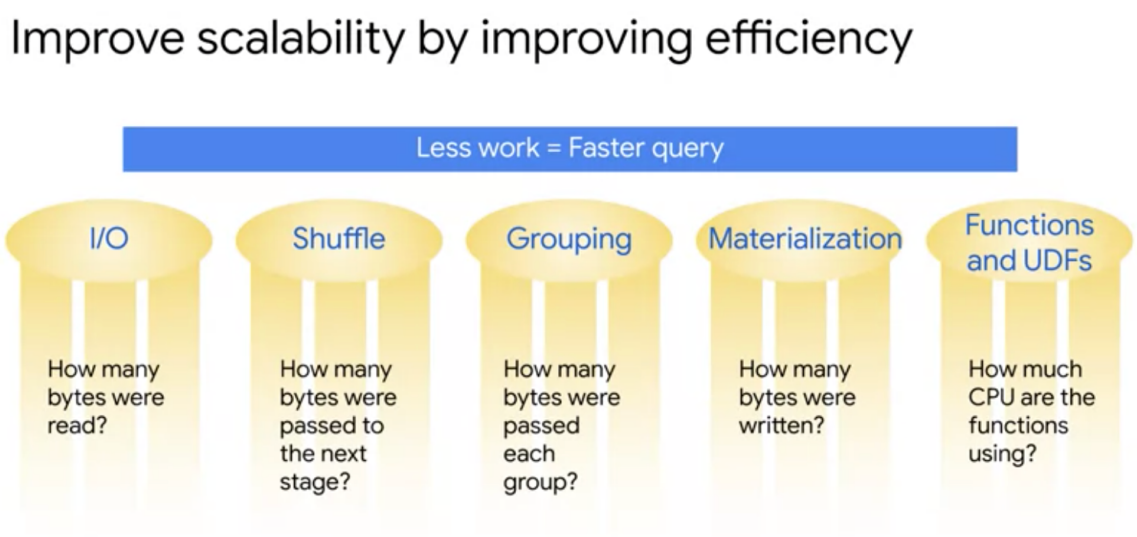
Query complete (23.5 sec elapsed, 1.1 GB processed)

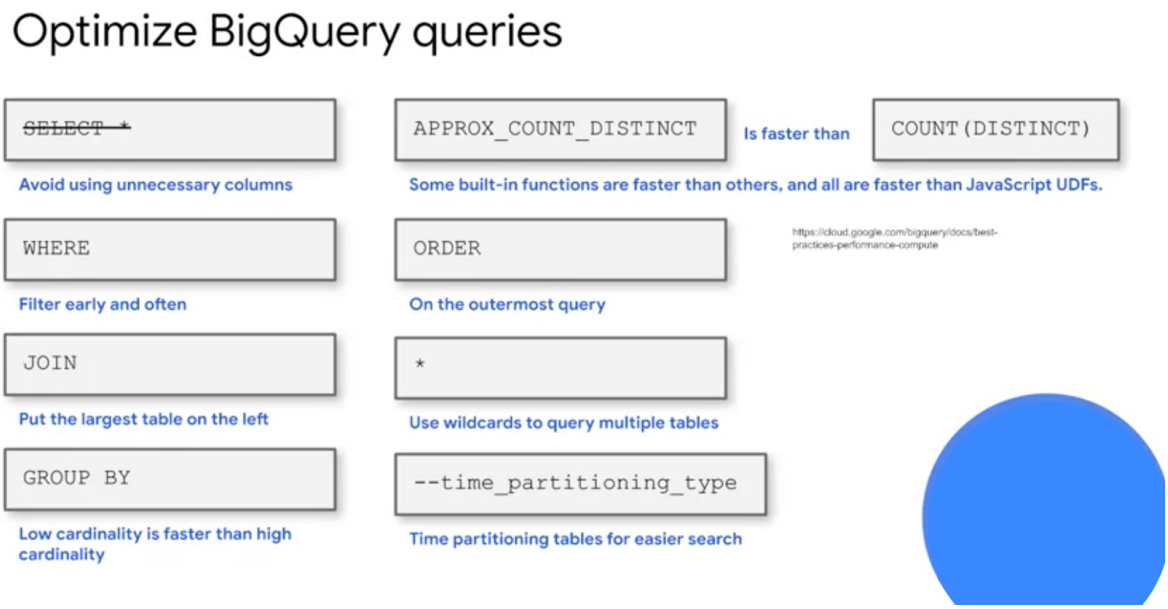
Job information | Results | JSON | Execution details

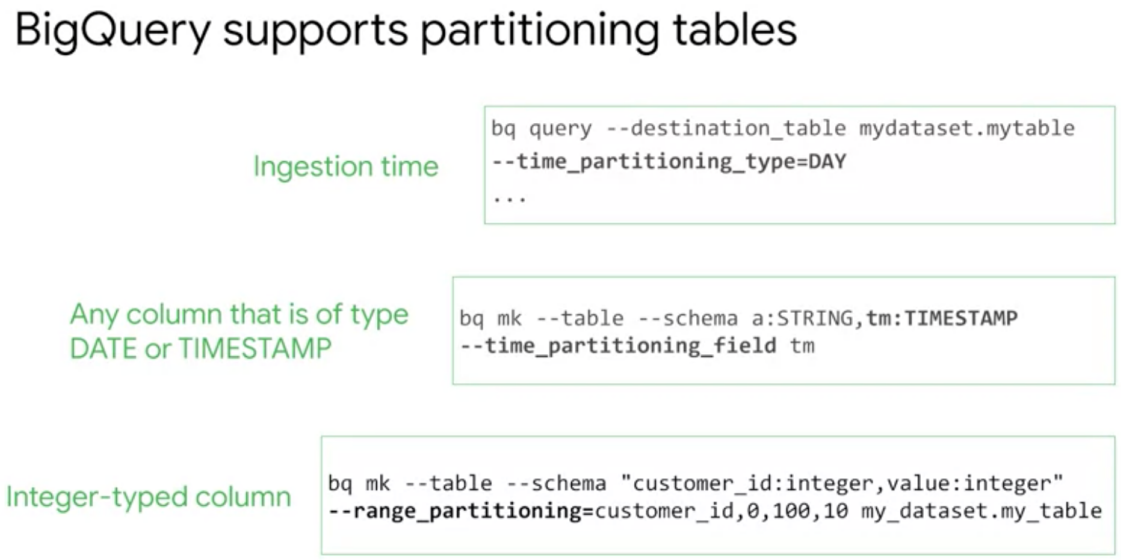
Row	most_distance_km_rank	bike_id	distance_travelled	maint_stats.current_trip_number	maint_stats.cumulative_duration_hr	maint_stats.cumulative_distance_km	maint_stats.start_date	maint_stats.end_date
1	1	12757	5846.469623639495	2746	963.8833333333333	5846.469623639495	2017-06-13 18:00:00 UTC	2017-06-13 18:09:00 UTC
				2745	963.7333333333332	5844.421787976495	2017-06-13 17:42:00 UTC	2017-06-13 17:52:00 UTC
				2744	963.5666666666666	5842.5291360107485	2017-06-13 09:43:00 UTC	2017-06-13 09:49:00 UTC
				2743	963.4666666666666	5841.687255976212	2017-06-12 17:43:00 UTC	2017-06-12 18:02:00 UTC
				2742	963.15	5837.581675192751	2017-06-12 17:33:00 UTC	2017-06-12 17:41:00 UTC
				2741	963.0166666666667	5835.860305651863	2017-06-12 08:57:00 UTC	2017-06-12 09:04:00 UTC
				2740	962.9	5834.536903703587	2017-06-12 08:37:00 UTC	2017-06-12 08:51:00 UTC
				2739	962.6666666666666	5831.547099983607	2017-06-12 07:49:00 UTC	2017-06-12 08:01:00 UTC
				2738	962.4666666666666	5829.3522309666105	2017-06-09 14:26:00 UTC	2017-06-09 14:42:00 UTC
				2737	962.1999999999999	5825.135931342222	2017-05-31 16:26:00 UTC	2017-05-31 16:29:00 UTC

Rows per page: 1 | 1 - 1 of 10 | First page

Performance Considerations







Approximate aggregation functions

BigQuery provides fast, low-memory approximations of aggregate functions. Instead of using COUNT(DISTINCT …), we can use APPROX_COUNT_DISTINCT on large data streams when a small statistical uncertainty in the result is tolerable.

The approximate algorithm is much more efficient than the exact algorithm only on large datasets and is recommended in use-cases where errors of approximately 1% are tolerable. Before using the approximate function, measure on your use case!

Use a window function instead of a self-join

Suppose you wish to find the duration between a bike being dropped off and it being rented again, i.e., the duration that a bicycle stays at the station. This is an example of a dependent relationship between rows. It might appear that the only way to solve this is to join the table with itself, matching the `end_date` of one trip against the `start_date` of the next.

```
SELECT
  bike_id,
  start_date,
  end_date,
  TIMESTAMP_DIFF( start_date, LAG(end_date) OVER (PARTITION BY bike_id ORDER BY
start_date), SECOND) AS time_at_station
FROM
  `bigquery-public-data`.london_bicycles.cycle_hire
LIMIT
  5
```