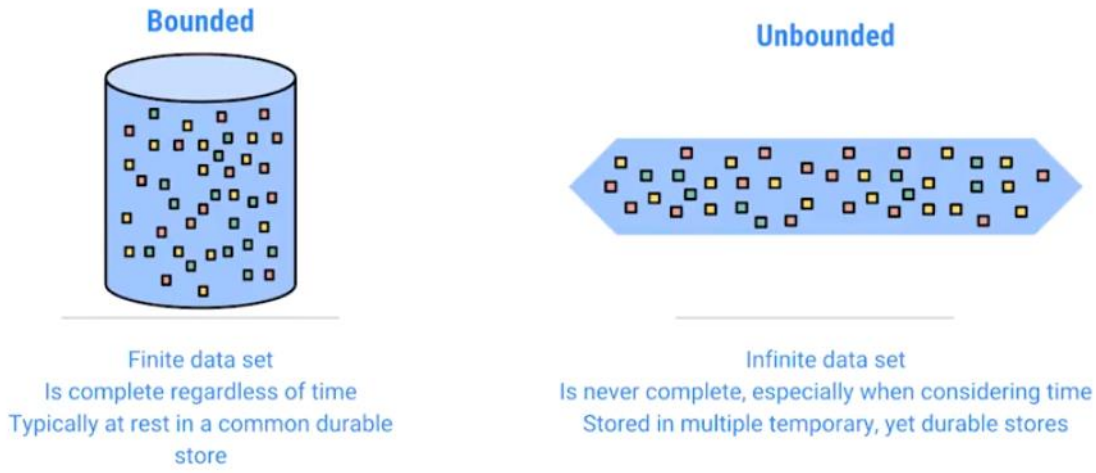
BUILDING RESILIENT STREAMING SYSTEMS ON GOOGLE CLOUD PLATFORM

Welcome to this course on resilient streaming applications on Google Cloud platform. The first chapter is on the architecture of streaming analytics pipelines. This course discusses what stream processing is, how it fits into a big data architecture, when stream processing makes sense, and what Google cloud technologies and products you can use to build a resilient streaming data processing solution. We also discuss the challenges associated with stream data processing. There are three key challenges: handling variable data volumes, dealing with unordered or late data, and deriving insights from data even as it's streaming in. And in this chapter, we also do a pen and paper lab, where we consider typical streaming scenarios and build streaming architectures.

MODULE 1: ARCHITECTURE OF STREAMING ANALYTICS PIPELINES

Steaming – Data Processing for Unbounded Data Sets

Streaming in this context means processing data for **unbounded** data sets.



Typical data sets are bounded. This means is that they're complete. At the very least, you will process the data as if it were complete. Realistically, we know that there'll always be new data but as far as data processing is concerned, we treat it as if it were a complete data set. Another way to think about bounded data processing is that we finish analyzing the data before new data comes in.

On the other hand, an unbounded data set is never complete and there is always new data coming in. Typically, data is coming in while analyzing the data. We tend to think of analysis on unbounded data sets as this temporary thing, carried out many times, bit being valid only at a particular point in time.

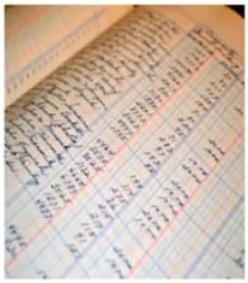
Streaming is data processing on unbounded data. Bounded data is data at rest. Stream processing is how you deal with unbounded data that's not at rest.

More broadly, streaming may be considered as an execution engine - the system, the service, the runner, the thing that you're using to process unbounded data. If designed correctly, such a stream processing engine can have low latency, provide partial results or speculative results, results that can be changed later. Streaming gives the ability to reason about time, to control for correctness of the data, and the power to perform lots of complex analysis, even as the data are coming in. In this course, we learn how to design stream processing systems.

Unbounded Data Sets are Quite Common

Traffic sensors along highways

Usage information of Cloud component by every user with a GCP project



Credit card transactions

User moves in multi-user online gaming

How common are unbounded datasets and why should we care? Unbounded datasets are actually quite common and becoming even more common as data centers get cheaper and network connectivity improves. For example, traffic events are tracked by sensors laid out on thousands of miles of highway.

Another example of monitoring is to track information from some cloud component across all users within a Google Cloud Platform project. Monitoring creates streaming data. This is a very common. You want to derive insights fast and in order to derive insights fast monitor data as it is coming in. Consider credit card transactions of every card member ever since they opened an account – if we track all transactions over time, we have streaming data. This is necessary for things like fraud detection.

Another example is online multi-user games – tracking the moves of every user in the game. This is also a streaming application.

Consider these common themes:

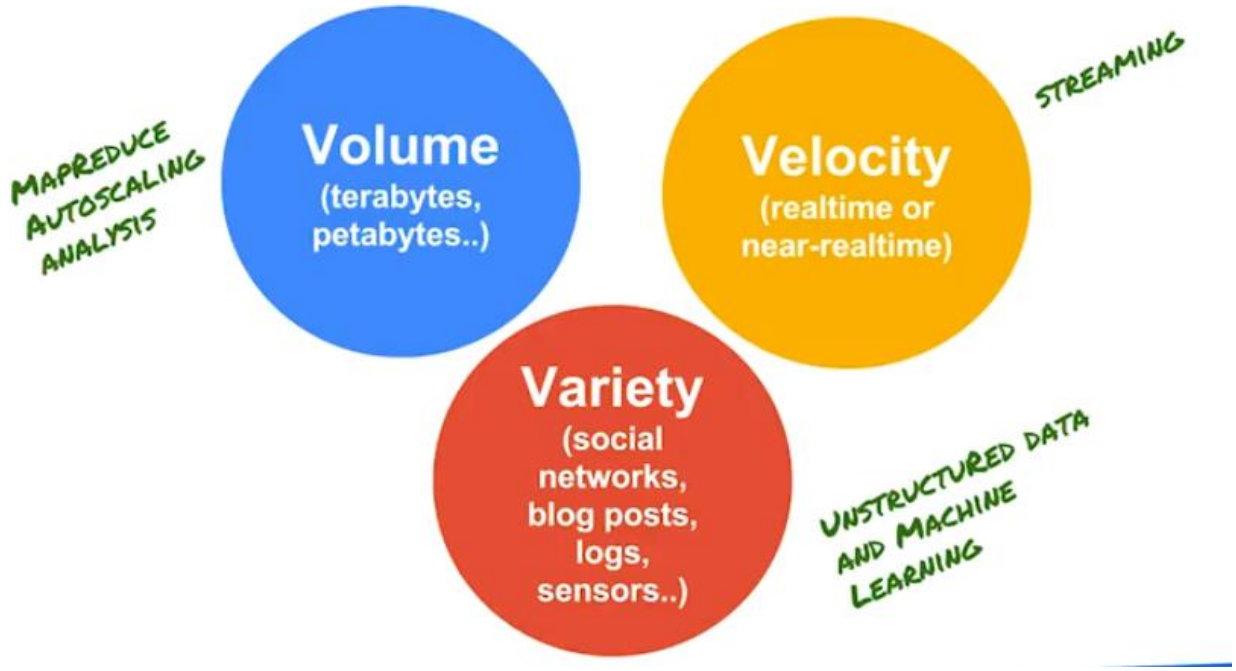
1. There is massive data from a variety of sources that keeps growing over time.

2. The need for fast decisions. For example, it is desired to prevent the credit card fraud from happening. That's the reason you need to process the data as it's coming in. It's not enough to collect the credit card data and process at once a day for example. You need to process that as it's coming in. Results can be displayed in the form of dashboards That can lead to quick decision from streaming data.
3. There is a lot of data from a variety of sources and it keeps growing over time. In cases such as fraud, we consider not only new data, but look at patterns over time. How does a user's behavior compare to past behavior?

The Need for Fast Decisions Leads to Streaming

The demand for stream processing is increasing. It's not enough to process large data. It must be processed fast enough such that a firm can react to changing business conditions in real-time. Real world stream processing use cases include things like trading, fraud detection, system monitoring, order routing, transaction cost analysis, etc. Batch is to look at fraud transactions historically. Streaming is to look at transactions as they come in. Stream processing is used to determine if there are fraudulent activity among all incoming transactions. Once there is a fraudulent transaction, then something must be done quickly. Streaming brings the ability to make faster decisions.

The Three Vs of Big Data



Volume

If you have terabytes and petabytes of data then we should consider using serverless data analysis. One way to handle such large amounts of data is to do MapReduce which will break up the data and perform auto-scaling analysis of this data: e.g., using BigQuery and Dataflow. When these were considered previously, they were used for batch processing solutions over large amounts of data. On Google Cloud Platform, both BigQuery and Dataflow provide support for streaming also.

Another aspect of big data is variety: audio, video, images, unstructured text, blog posts, etc. It is difficult ot deal with such a wide variety of unstructured data. If data are all structured, it is simple to send it to a relational table and query using joins. In the courses on unstructured data, we used machine-learning APIs to process the data.

The third aspect to big data is near real-time data processing; data that's coming in so fast that you need to process it to keep up with the data. If you think about a big data architecture, you will have several parts to it. Often, masses of structured and semi-structured historical data that may have been stored in cloud storage or POP server, or BigQuery. That's volume and variety. On the other side, stream processing is what is used for fast data requirements for velocity. Both complement each other. For example, if you think about Internet of Things, there is increased data volume, probably increased variety and velocity of data. This leads to a dramatic increase and need for things like stream processing technologies.

Stream Data Processing: Challenges

Consider the challenges of stream data processing.

Stream processing makes it possible to derive real-time insights from growing amounts of data. For a good stream processing solution, there are three key challenges that it needs to be addressed:

1. The solution must be able to scale. In many cases, it must be scaled for seasonal needs, e.g., scale up during the holiday season.
2. Use continuous queries. Many times, we are interested in querying the latest arriving data. For example, we might want to consider moving averages looking for unusual spikes. To do so, we have to continuously calculate some a statistical analysis.
3. Insight is not enough. With streaming data, ingest cannot be stopped whenever we want to analyze the data. It is necessary to derive the insights even as the data are coming in. In other words, we want to be able to do SQL-like queries that operate in time windows of the data.

Stream processing systems were first created nearly a decade ago because of the need for low latency processing of large volumes of dynamic time continuous streams: sensors and monitoring devices, etc. Probably the first users were in the financial industry with stock market data. But today you see it everywhere. You're generating stream data through human activities, through social media, through machine data, through sensor data.

Challenge #1: Handling Variable Data Volumes

Variable volumes make it possible to derive real-time insights from growing data

Volumes keep changing, and so you need the ability to ingest to scale.

And in spite of variable volumes, the ingest application shouldn't crash. It needs to be available.

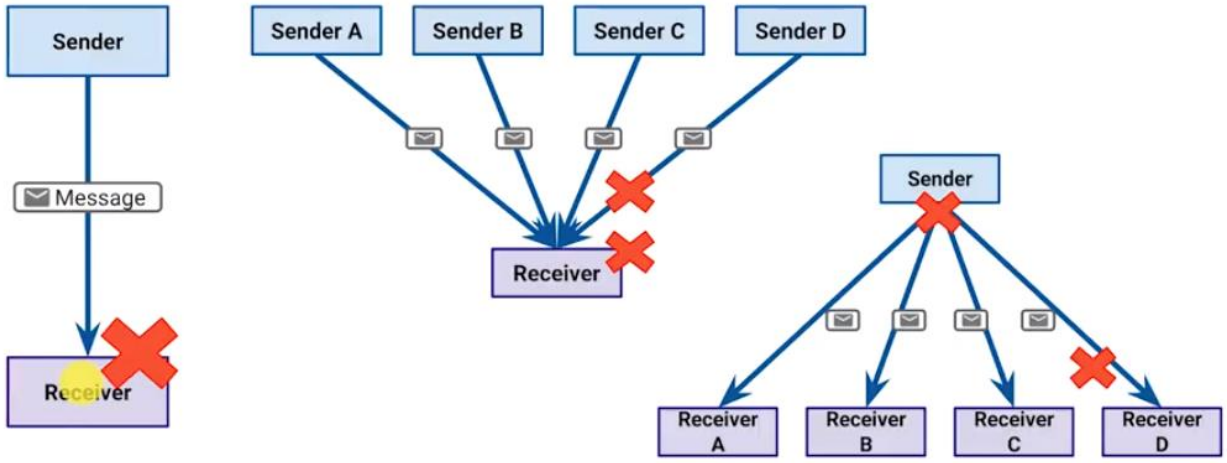
Any messages received need to remain received. For example, records of purchases cannot be lost. They must be *durable*.

When scaling, the application must be able to handle faults as clients, servers and storage systems fail unexpectedly.

How can you build a fault tolerant application while dealing with spikes in data and other changes in the volume of data?

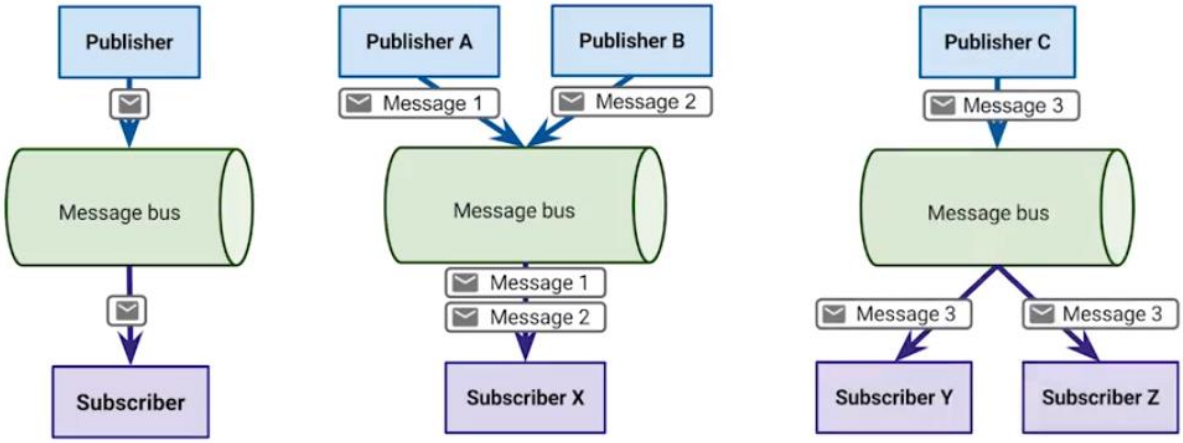
Tightly coupled services propagate failures

What you cannot do is to tightly couple the sender and the receiver. If the sender is directly sending messages to the receiver and the receiver gets overwhelmed and crashes then messages get lost - it's not durable in other words. Perhaps the server itself will get overwhelmed, so the sender won't be available. If you directly couple the sender with the receiver, you either have durability problems or you have availability problems.



This diagram above illustrates various failure scenarios. If there is a single point of failure, then the system is very fragile.

Loosely Couple Systems Scale Better



The solution to resolving a single point of failure is to buffer. Send messages to a message bus which will buffer the messages. If a sender or receiver fails, messages are not lost, but will be sent to another receiver.

Furthermore, this architecture scales much better and handles variable volumes and nearly any number of senders and receivers (or subscriber – we say subscriber because the receiver now subscribes to messages).

The message bus addresses durability and availability. All components are loosely coupled. If a subscriber fails, the message bus keeps those messages to send to another subscriber instance. If a publisher fails, the bus waits till it comes back online and then receives its messages. The system is now fault-tolerant.

Challenge #2: Dealing with Unordered/Late Data

Latency is to be expected

To build a durable, highly available solution on GCP, use Pub/Sub. Pub/Sub is serverless: you don't need to start any clusters, just publish messages to a topic in Pub/Sub, or subscribe to a topic in Pub/Sub.

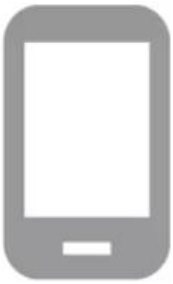
Consider the case where there are two sensors sending data. Smaller packets arrive before larger packets often in most networks, even if it's from the same sensor. Also, for two or more sensors, the network paths to the sensors could be very different.

Different latencies for devices in a network cause messages to be out of order.

Known phenomena for a long time – remember the design for TCP.

Messages will not be in perfect sequence, which must be remembered when making calculations. Consider computing the average or maximum. What happens when data arrives late – data that was already expected? The maximum that was computed is now incorrect. Therefore, the computed maximum in a stream is a speculative result which needs to be updated. How do you include this data? How do you treat it?

Latency Happens for a Variety of Reasons



Transmit	Ingest	Process
Network delay or unavailable	Throughput (read latency)	Starved resources
Ingest delay (write latency)	Backlog	Internal backlog
Ingest failure	Hardware failure	

Considering that latency will occur, what should be done? There are two schools of thought here. One says to design and build hardened systems - build systems such that there is absolutely no error whatsoever. This is an unrealistic expectation.

The second school of thought is to design with the expectation that errors will occur, but errors must be handled with resilience. Assume that errors are normal, and design proper exception handling around those errors.

Latency will occur due to network delays, ingest delays, write latencies, ingest failures, and a variety of other reasons. Latency could happen during processing, perhaps there are not have enough workers processing messages so there is a back log of messages which are queued.

Beam/DataFlow model provides "exactly once" processing of events

- ***What* results are calculated?** Answered via transformations.
- ***Where* in event time are results calculated?** Answered via event-time windowing.
- ***When* in processing time are results materialized?** Answered via watermarks, triggers, and allowed lateness.
- ***How* do refinements of results relate?** Answered via accumulation modes

https://cloud.google.com/blog/big-data/2017/05/after-lambda-exactly-once-processing-in-google-cloud-dataflow-part-1

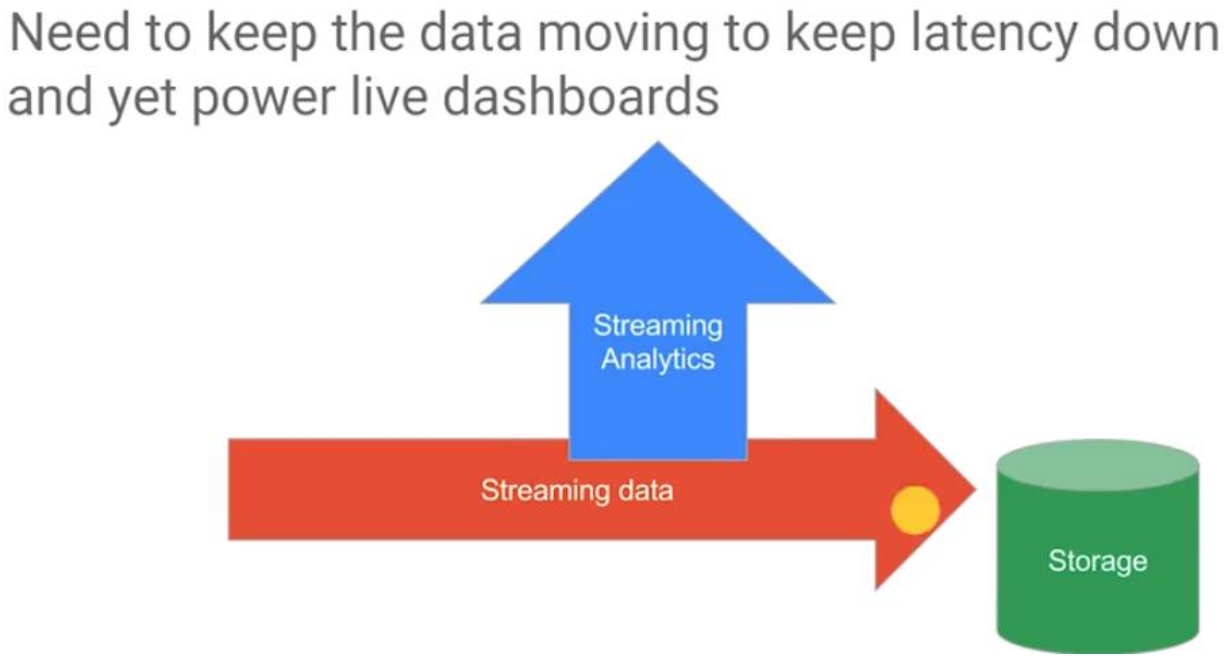
Beam/DataFlow model provides for "exactly once" processing of events. The model allows for wrigin to code that answers these questions:

1. What results are being calculated? For calculating the maximum, use a P transform in DataFlow
2. When in processing time should the results be calculated?
3. When in processing time should the results be saved?
4. How should you change an already computed result? How should you refine it?

Based on these questions, we can design a valid stream processing pipeline. These out of order processing constructs, watermarks, and triggers all give you the ability to maintain eventual correctness of results in a pipeline. However, at the same time they give you the ability to provide a low latency speculation, e.g., the ability to publish a maximum before all the data are in. The provide the ability to refine the results after the fact when you get new data. And you also get an easy way to cap the lifetime of data within your system – for example, the program will refine data up to three hours, but ignore any late record that comes in three hours late.

Challenge #3: Derive Insights from Data

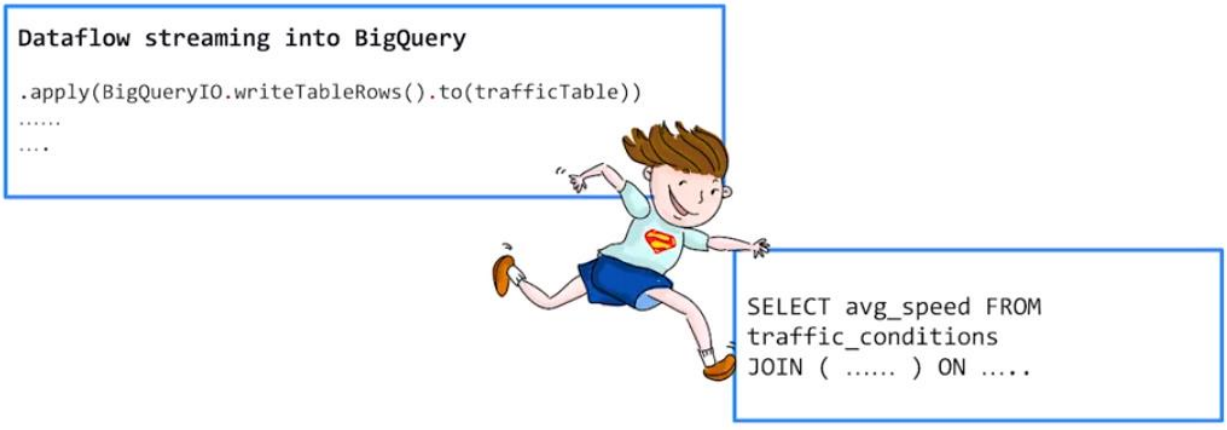
Instant Insights are Needed



The way to handle late data or data out of order is to use Cloud Dataflow. Then, it is necessary to get instant insights. Data needs to keep moving through the pipeline to keep down the latency, otherwise there will be significant delays. The data stream cannot be stopped to analyze it. To achieve low latency, a system must be able to perform message processing without having to store the data.

At the same time while the data is coming in, we want to power live dashboards. You should not have to store data and then query the stored data. It is better to perform streaming analytics on a stream buffer of data, because storage adds latency.

BigQuery lets you ingest streaming data and run queries as the data arrives



BigQuery can ingest streaming data and run queries is the data flies by. Recall that we wrote a p transform in

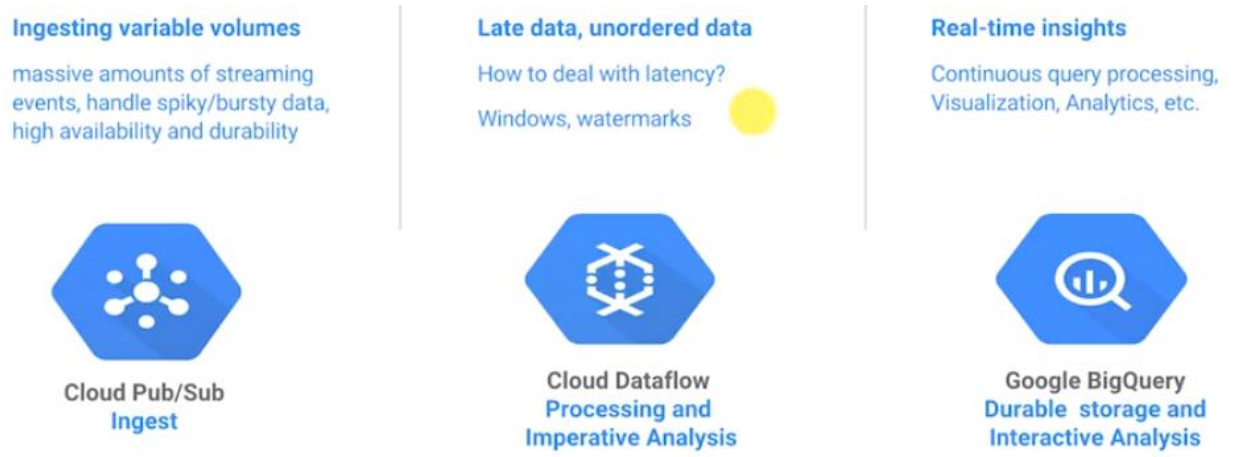
Dataflow: BigQueryIO.writeTableRows.to some table. That would for streaming also, even as Dataflow is streaming table rows to BigQuery.

In the BigQuery console or a BigQuery client and we can run an SQL query. For example, select average speed from this table joined with something else and this SQL query will get carried out on the streaming buffer even before it has been saved to disk.

BigQuery works on stored and streamed data in the same way as both Dataflow and BigQuery. The code is the same: use the same kinds of analysis, the same kinds of queries, the same kinds of speed transforms on both batch data as on streaming data.

Need to use unified language when querying historic and streaming data for seamless integration

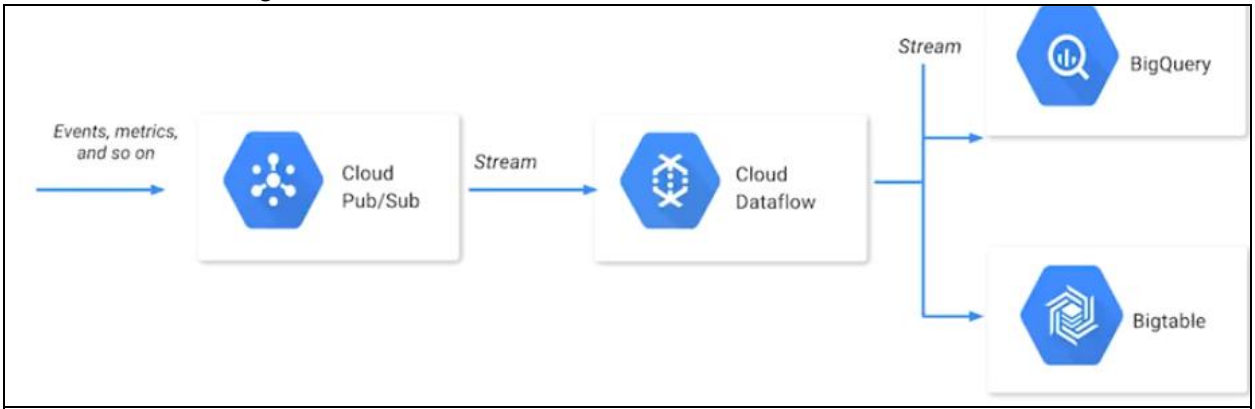
It is desirable to use a unified language when querying historic and streaming data to create a system that can deal with reference data, historical data, and live data. To have real-time insights, it is necessary to analyze both the live data and historical data, and using two different systems for this is hard. It's much better to perform the same kind of analysis on both historical data and live data. This is what BigQuery makes possible.



- Pub/Sub is a global message bus.
- Dataflow is capable of handling batched or streaming data, the code doesn't change. It gives you the ability to deal with late and unordered data.
- BigQuery gives you the power to perform analytics both on historical data and on streaming data.

One issue with BigQuery is that if very high throughput is required with very low latency, it may be necessary to use Bigtable.

Stream Processing on GCP



A Common Configuration which offers fully managed, No-Ops services

This then is the architecture for stream processing on GCP. There is a data source, and that data source provides events, metrics, and so on. And the data source could be a sensor network, a mobile application, a web client, a server log, even something from the Internet of Things.

This goes into a message bus, Pub/Sub, which is reliable, high throughput, and low latency. It buffers messages which can be passed to a stream processing system which is dataflow. This is a computational framework that can carry out computations on data streams, continuous computations, computations which are continuously repeated.

Meanwhile, the computed values, which are typically aggregations of sorts or even raw data, can be streamed into BigQuery or Bigtable. This gives the ability to do ad hoc interactive queries on this data. Dataflow provides the ability to make continuous queries. In contrast, BigQuery and Bigtable provide the ability to perform user-generated queries, ad hoc queries, queries performed occasionally.

The difference between BigQuery and Bigtable is that the latency in BigQuery is on the order of seconds. The latency in Bigtable is on the order of milliseconds. BigQuery is sequel, Bigtable is no sequel. All these are fully managed no-op services.

Streaming Scenarios

https://www.coursera.org/learn/building-resilient-streaming-systems-gcp/lecture/OWPuc/discuss-some-streaming-scenarios

Lab Worksheet

https://d18ky98rnyall9.cloudfront.net/_6fea563bb557571a99ef8afbff6df258_Lab1.pdf?Expires=1527033600&Signature=GsOSWFiKF9A6RQGloiVgXUTYruA7kjypkiXgfB2nv98T3KcutBkx~~GCkMcrI1uCEfTImH8N58EsRXmgW-UmNE93zgls5lpRVj-cjaSx4c4YAyrqXwRkySjtQdAqR34CwcLLH34vUzqx4fdwBWR7j~vJuQnF9IIHtWMpb4KvBrQ_&Key-Pair-Id=APKAJLTNE6QMUY6HBC5A

MODULE 2: INGESTING VARIABLE VOLUMES

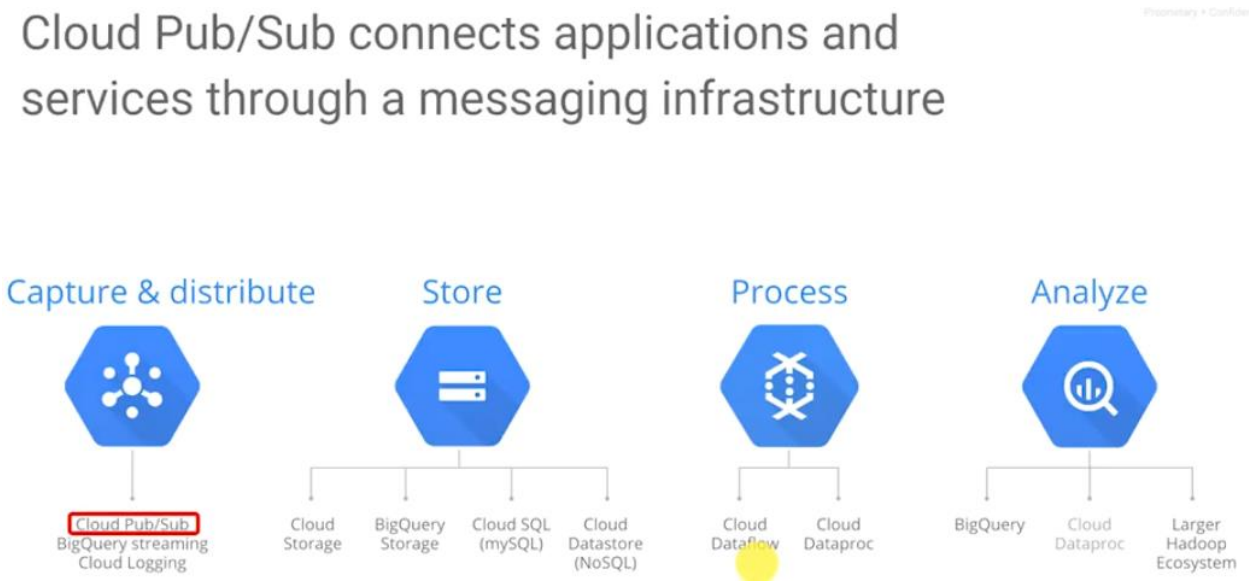
What is Pub/Sub?

In GCP, Cloud Pub/Sub is the message bus used to ingest the traffic data. Pub/Sub works very well with streaming data. It provides high ingest speed, durability, fault tolerance, NoOps, provides auto scaling. It is the way to handle the first challenge of being able to ingest variable volumes. Cloud Pub/Sub is global, multi-talented, managed, real time and it is a messaging service.

Pub/Sub is discoverable. It handles a discovery of subscribers who are interested in messages from specific publishers such that subscribers and publishers don't need to find each other directly. They just look in Pub/Sub for the list of topics for example. This also means with discoverability that moves are handled. So if a different publisher starts publishing to the same topic, no problem. Turndowns can be handled. So again if there are three publishers publishing to a topic and one of them goes away, no problem you're just listening to the topic and the messages will continue showing up in the topic.

Pub/Sub is highly available. A publisher does not need to address whether subscribers are around to receive the messages. Pub/Sub absorbs the requirements of handling any subscribers that cannot keep up with the streaming rate.

Messages are durable. Messages are to be delivered later if subscribers are not available to receive them. Expiry is up to seven days. Finally, Pub/Sub gives you global scale. It's a global messaging service. And it also gives you low latency.



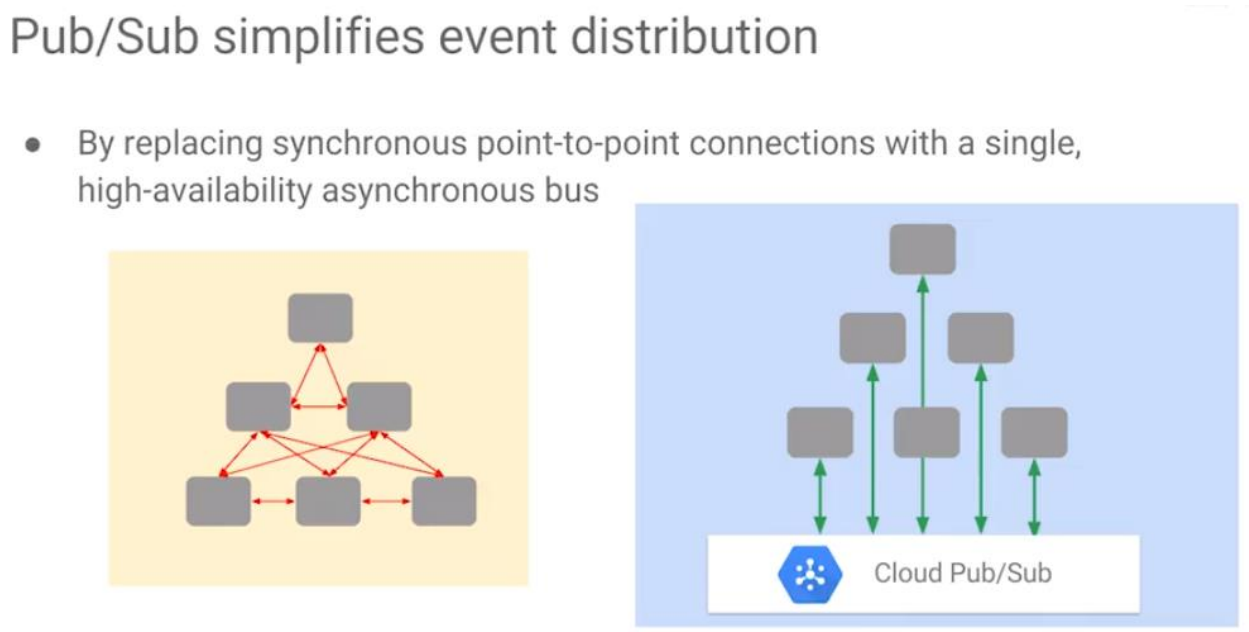
Pub/Sub is a way to connect applications and services through a messaging infrastructure, so don't use Pub/Sub to store data permanently. You don't use it to process data. You don't use it to analyze data. There are other products for that. Pub/Sub captures and distributes data.

There is only one Pub/Sub – it is a serverless, single global service that offers high, consistent throughput and latency. There is no need to start up a cluster of machines on which you run Pub/Sub.

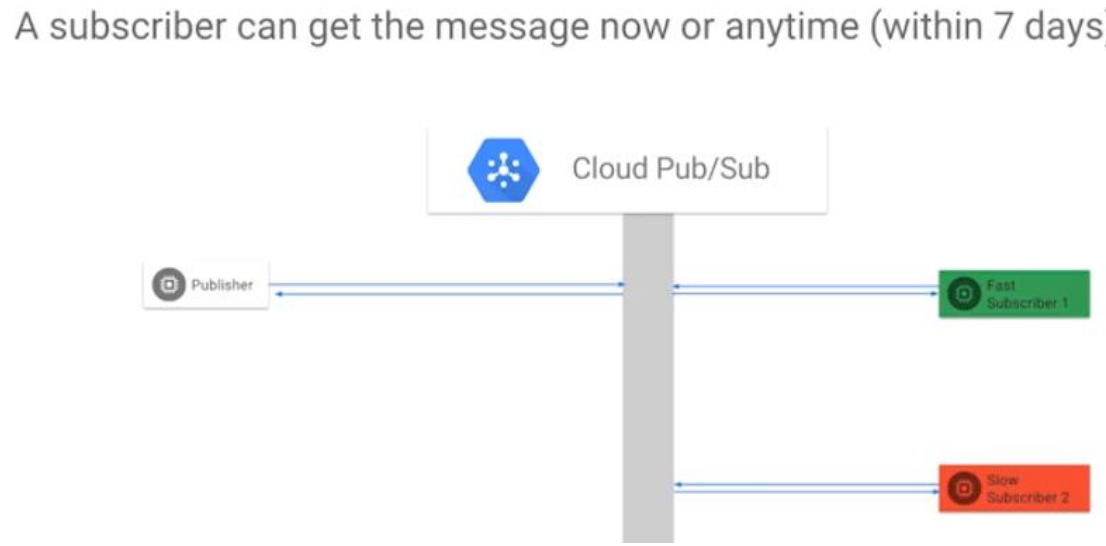
We already have a single, global way of connecting applications which is the network. Isn't that enough? Why do we need Pub/Sub too, in addition to having a network? If the subscriber is not around, the message won't

be transmitted. If the publisher is not around, the subscriber has nothing to process. Everything grinds to a halt unless both parties are online and available all the time.

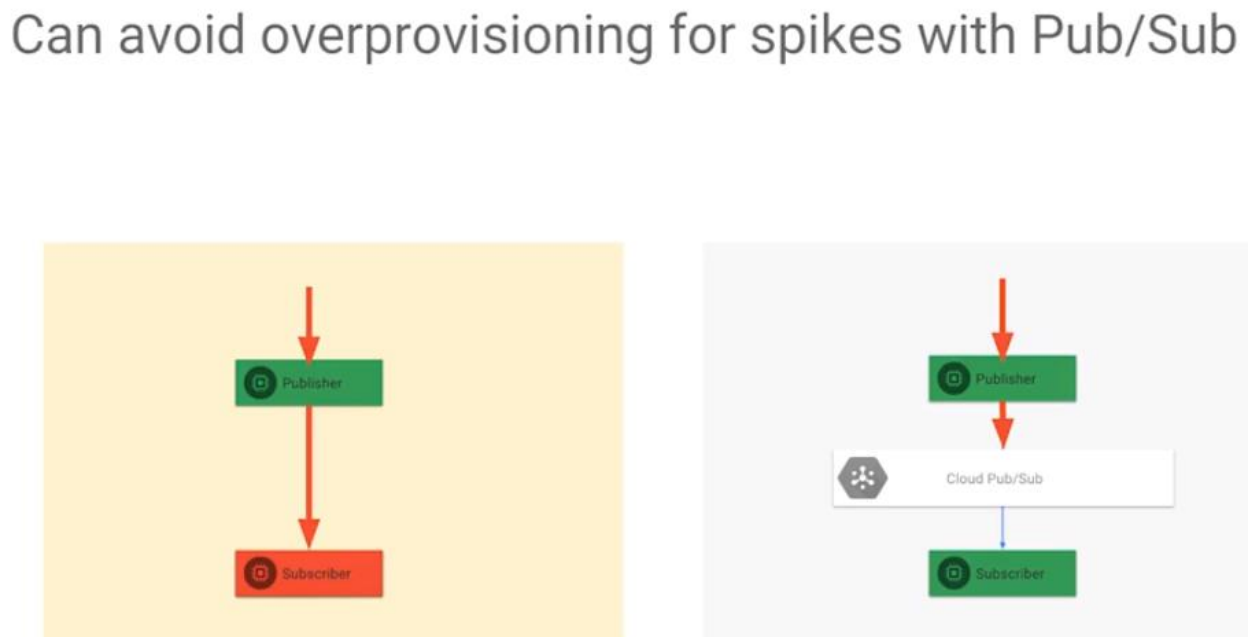
With Pub/Sub however, senders and receivers don't need to be online at the same time. Pub/Sub delivers messages instantly as if everyone is online, and safely holds them until they can be delivered if the subscriber is not online. This is a way that Pub/Sub helps you deal with spikes and variable volumes.



Pub/Sub simplifies event distribution and provides a method of asynchronous communications. The publisher never waits, the publisher keeps publishing to Pub/Sub. Each subscriber can consume messages at its own pace.



Messages are delivered instantly to subscribers that are online and available. If a subscriber is not online, the messages are kept in Pub/Sub and retried for up to seven days. This means that if you are using Pub/Sub, you can avoid over-provisioning for spikes since the messages can be processed after the spike has passed.



This is the answer to handling variable volumes. Pub/Sub, because it holds messages and delivers them when the subscribers are ready, it serves as a smoothing mechanism. This way subscribers don't get overwhelmed.

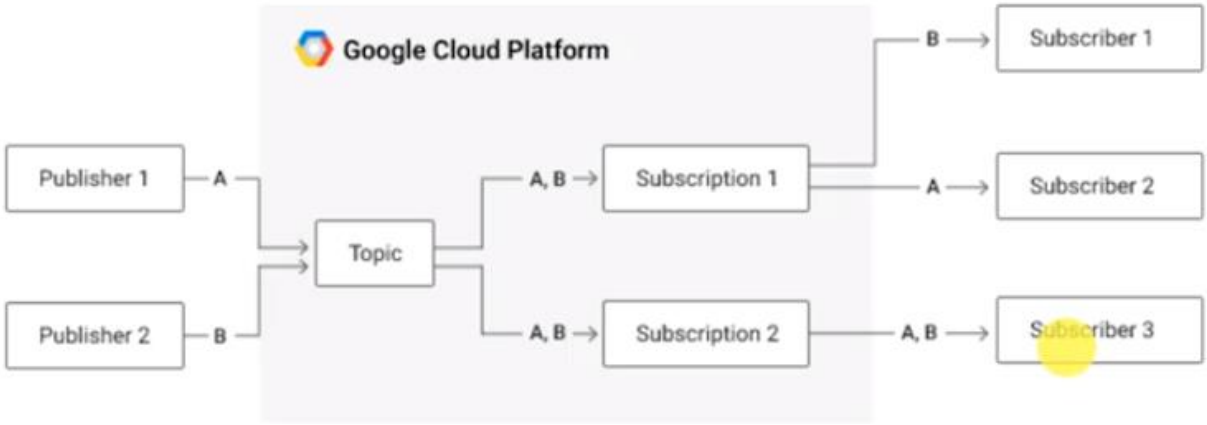
Pub/Sub Features

- Single, global service – move data from anywhere to anywhere. It is fully managed. There's only one Pub/Sub globally and it is always available.
- Messages and Pub/Sub, are guaranteed to be delivered at least once to everybody who subscribe to it. In other words, it is possible (but unlikely) that you will get duplicate deliveries. However, the order isn't guaranteed. It is necessary to choose between low latency and order delivery. *Order is not guaranteed at low latency*: low latency will provide the messages as quickly as we get them, but the messages are different sizes and smaller messages are going to be received typically before very large messages in the same queue. It is also possible to get duplicate messages (the next step in the reference architecture is Cloud Dataflow and Cloud Dataflow which address the problem, e.g., Dataflow gives exactly once processing).
- Pub/Sub is fast. Latency is on the order of hundreds of milliseconds, and it is scale-able from one kilo byte per second to 100 gigabytes per second. At that scale, you get consistent performance whether you're processing 1,000 bytes a second or you're processing 100 gigabytes per second.
- Pub/Sub supports fan in fan out whether you have multiple subscribers listening to a single topic, or you have multiple publishers publishing to a single topic - both are supported.

- Pub/Sub also supports both push and pull. Subscribers can ask to be notified or they can keep checking whether there is a new message in the topic. In terms of interacting with Pub/Sub, you have two types of client libraries that you can use to publish into Pub/Sub or subscribe from Pub/Sub.
- There are hand-built libraries in Java, Python, C#, Ruby, PHP, Node.js. In addition, any language that GRPC (Google Protocol – see https://grpc.io/) supports provides the ability to auto generate a client API for that language.

How Topics and Subscriptions Function

Pub/Sub works is through topics and subscriptions. Pub/Sub implements an asynchronous publish subscribe pattern: a publisher sends messages to one of multiple topics in Pub/Sub. The subscriber creates a subscription to topics of interest. If the subscriber goes down, the messages in the topic are held in the subscription for up to seven days for this subscriber to come back up and collect those messages.



There can be multiple subscribers that process the messages in a subscription. Every message in this topic is going to be in each subscribers' subscription. That is the way that subscribers are kept independent. There may be fast subscribers and slow subscribers, neither waits on the other.

The decoupling of publishers and subscribers means that neither side needs to worry availability - the message bus handles all of that.

Create Topic and Publish Messages

```
gcloud beta pubsub topics create sandiego
```

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

Python code

```
from google.cloud import pubsub
client = pubsub.Client()

topic = client.topic("sandiego")
topic.create()
topic.publish(b'hello')
```

To create a topic, use gcloud which is one of the tools in the GCP SDK. It can be accessed Cloud Shell or from the cloud SDK downloaded onto your local machine and run from the command line. In the example above, the topic *Sandiego* in Pub/Sub. The topic will get created in a project. The namespace of this topic is going to include the project name.

To publish a message into this topic, use gcloud: publish into the topic sandiego the message hello. Messages in Pub/Sub are opaque – it will not parse these messages. It simply hands blob of bytes received to subscribers. It supports a ReST API which can be used from the users' applications.

It is also possible to utilize one of the pubsub client libraries from any of those ten GRPC languages. Or you can use one of the hand coded libraries. That is the third code snippet on the slide which shows the Python library. Both the glcoud commands and the Python code here do essentially the same thing.

Other Publishing Options

Python code

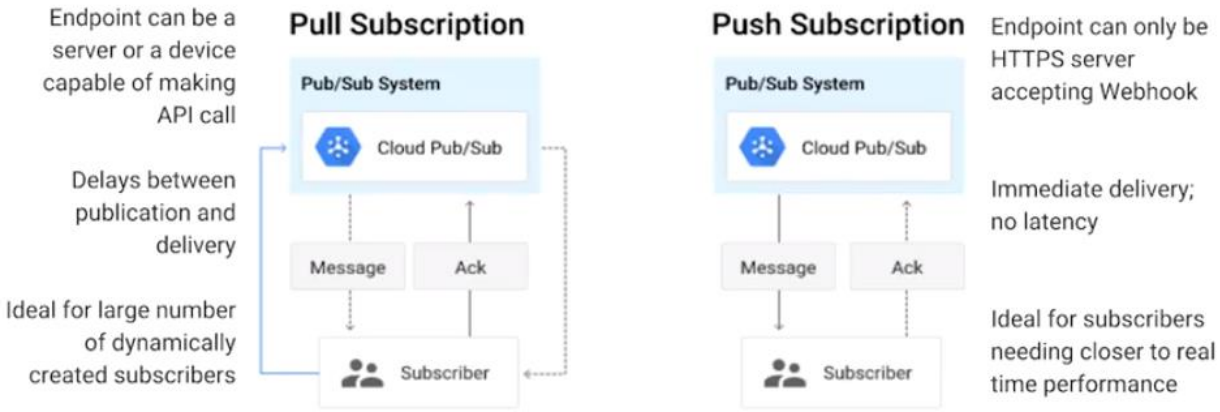
```
TOPIC = 'sandiego'
topic.publish(b'This is the message payload')

#Publish a single message to a topic, with attributes:
topic.publish(b'Another message payload', extra='EXTRA')

#Publish a set of messages to a topic (as a single request):
with topic.batch() as batch:
        batch.publish(PAYLOAD1)
        batch.publish(PAYLOAD2, extra=EXTRA)
```

There are other options. One is to just publish the message and be done. It is also possible to publish a message adding in extra attributes in the form of key value pairs. The key value attributes may be thought of as metadata about the message.

When the subscriber gets the message they also get the metadata. They can inspect the metadata for key value pairs of interest. One common metadata item is the timestamp of the message generated by the publish method.

The timestamp works very well when working in real time. But if not working in real time, e.g., working with historical data, it may be desirable to set metadata which is the time stamp of the message. The other thing that you want to be aware of is that you don't have to call Publish for every message. Every time you call Publish, there is a network call from your application to Pub/Sub which introduce latency. If there are a bunch of messages that you want to publish, it is a good idea to publish as a batch. The last snippet shows how to batch publish in Python and is similar in other languages. By publishing a set of messages as a batch, you help reduce the network costs.

Push vs Pull Delivery



There are two types of delivery flows which a subscriber can choose: push or pull.

In push delivery, Pub/Sub initiates the request to the subscriber application to deliver messages. Whenever there is a new message in the topic, that message gets delivered to the subscriber by Pub/Sub.

In a pull subscription, the subscriber asks Pub/Sub for a new message and if there is, it gets the message.

When the subscriber is pulling messages, what does the subscriber need? The subscriber simply needs to be able to make a web call or an API call.

For pushing, the subscriber needs to be an HTTPS web application. It needs to have a web hook that is accessible via HTTPS because Pub/Sub calls that URL. In a push subscription, the subscriber needs to be a server application.

With a pull subscription, there will be a delay between publishing to the topic and the subscriber getting the topic because a subscriber typically checks only periodically. In a push subscriber, there are no delays. The pull subscription is ideal if there are many subscribers, and those subscribers are dynamically created.

Push is ideal if you want low-latency, close to real-time performance, and immediate processing of messages. What happens in the case of a push subscription if the subscriber isn't running?

The subscriber has set up a subscription. The messages are going into the subscription, but Pub/Sub will not be able to reach the subscriber because the web URL is not accessible. In that case, Pub/Sub does an exponential backoff (see https://en.wikipedia.org/wiki/Exponential_backoff) where Pub/Sub tries again over exponentially increasing intervals. The retry rate can be controlled. Pub/Sub will try for up to seven days.

In a pull subscription, the subscriber explicitly requests delivery of a message that's in that subscription queue. The Pub/Sub server responds to the message or it responds to an error if the queue is empty. Pub/Sub responds with an acknowledgement ID. The subscriber is then responsible for calling the acknowledge method using that acknowledge ID. In that way, Pub/Sub knows that the subscriber has received (and processed) the message. If it doesn't, Pub/Sub will try to resend the message, so it is essential to acknowledge the ID.

There will be duplicate messages if a subscriber invokes a message call. The subscriber gets the message and processes the message. Then, between processing the message and acknowledging it, the subscriber crashes. Because the acknowledge method was not called, Pub/Sub will send the message again. However, this is the issue is handled well by DataFlow as it will only process a message once.

In a push subscription, the Pub/Sub server is going to call this pre-configured HTTPS endpoint. And the subscriber's response to that call serves as an implicit acknowledgement. It there is a success (200) response, it indicates that message has been successfully processed. Pub/Sub system will then delete the message from the subscription.

A non-success response (which is anything not in the range 200-299), indicates that the Pub/Sub server should resend the message. To ensure that subscribers can handle the message flow, Pub/Sub dynamically adjusts the rate at which it is sending messages. It uses a (backoff) algorithm to rate limit retries such that the subscriber endpoint will not be overwhelmed.

Creating Subscription Pull Messages

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

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

Python code

```
subscription = topic.subscription(subscription_name)
subscription.create()

results = subscription.pull(return_immediately=True)

if results:
 subscription.acknowledge([ack_id for ack_id, message in
results])
```

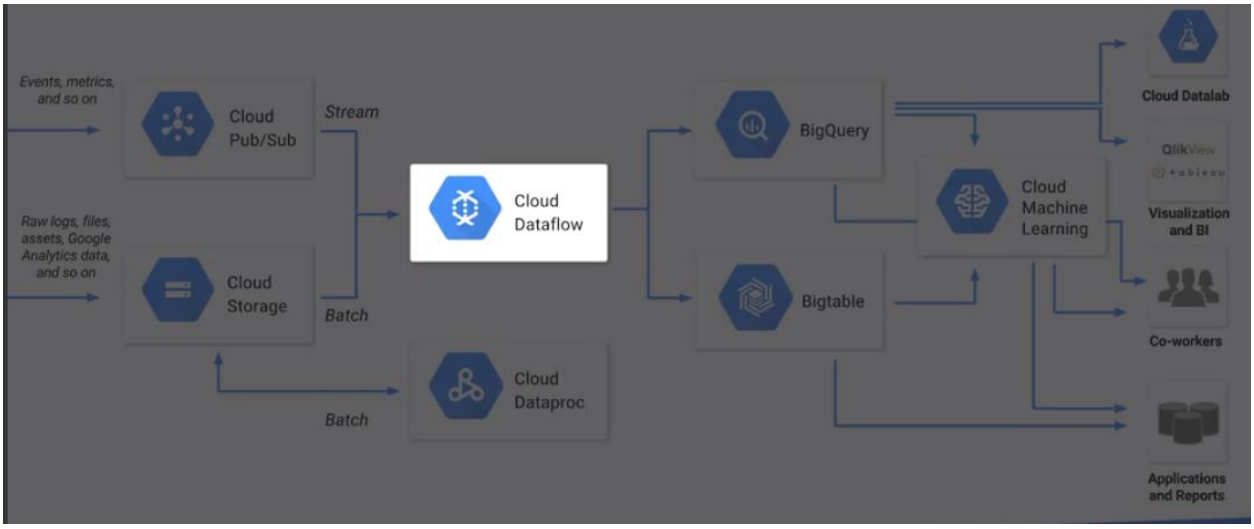
To retrieve messages as a subscriber, you will create a subscription to the topic of interest identified by the topic name. The subscription is created in your project and the project for the message and the topic could be different than the project for the subscribers. Whenever necessary, you can do a pull for a message and then acknowledge the message. In the code above, only one message was processed. All other messages are queuing up in that subscription for the next time for you to call pull.

Create a topic by calling pubsub.Client(). Once you have your topic, create a subscription with the name of your subscription. Then, whenever you're interested in a new message, pull using subscription.pull. Block, or wait for a new message until a new message comes in. Or, you can return_immediately which may return no messages or one or more messages. If there are results (results is *not* none), acknowledge subscription.acknowledge with the acknowledge ID and message in results. Acknowledge all messages.

Lab: Publish Streaming Data into Pub/Sub

Review: https://www.coursera.org/learn/building-resilient-stre1aming-systems-gcp/lecture/y0cUl/lab-review

MODULE 3: IMPLEMENTING STREAMING PIPELINES



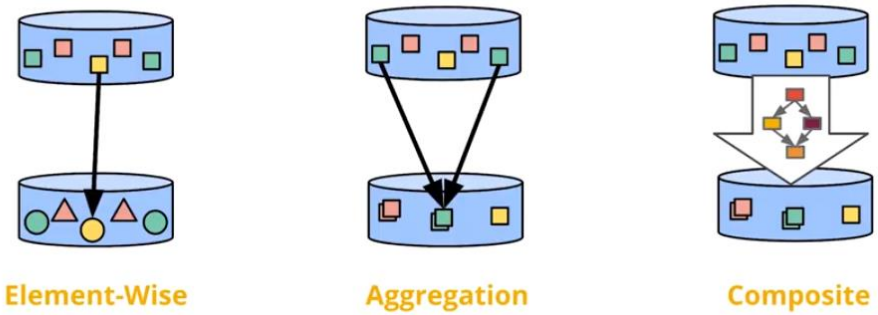
The second challenge of stream processing is to continuous processing data as it comes in. In other words, we want to build a data pipeline that works on streaming data. In the reference architecture on GCP, we now consider Cloud Dataflow. Data comes from a streaming source batch source with which we create a data pipeline to process this data.

In this chapter we look at Cloud Dataflow. We consider the challenges involved in stream processing and how Dataflow helps us address those. We'll then build a stream processing pipeline for traffic data, the hands-on activity that we are doing in this course. And then we'll look at how to handle late data. We'll look at the concepts of watermarks, triggers, accumulations and how those get affected by different choices in terms of how you do your triggers and how you do your windows. And finally, we'll do a lab where we run a Dataflow pipeline on this simulated traffic data stream that's coming into pops up.

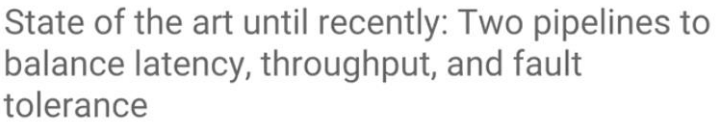
What is Google Cloud DataFlow?

The Challenge

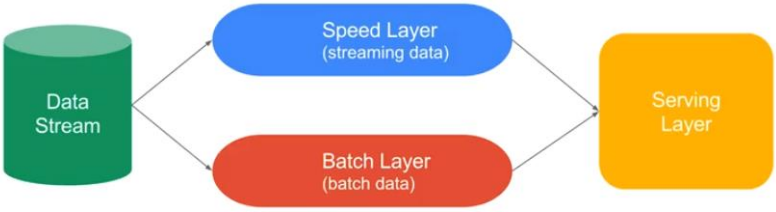
Element-wise stream processing is easy
Aggregation and composite on unbounded data is hard



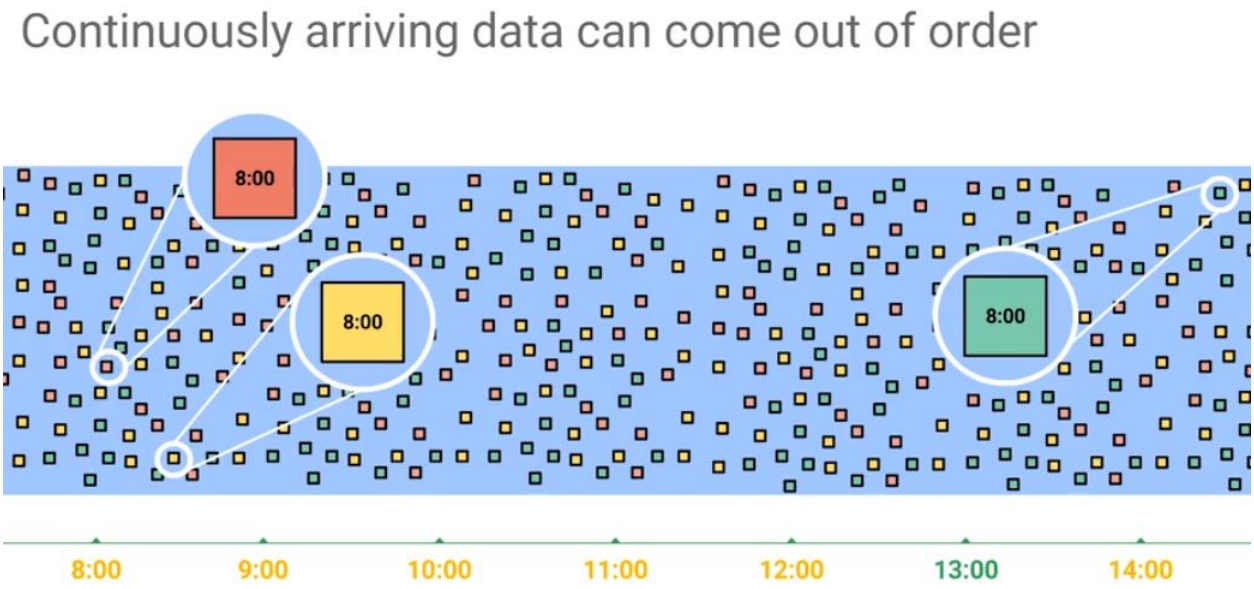
When considering processing of data in a stream, it is easy to process the *independent* data element by element: receive an element, process it, and then output the element. However, if you need to combine elements, perform aggregations (e.g., aggregate on elements with the same key), then things are not straightforward.



One of the most difficult things to do is to *composite* data with different keys, and/or coming from different data sources and then to perform aggregation on the composite data stream. For example, computations made from one source may be dependent on computations from the other source.



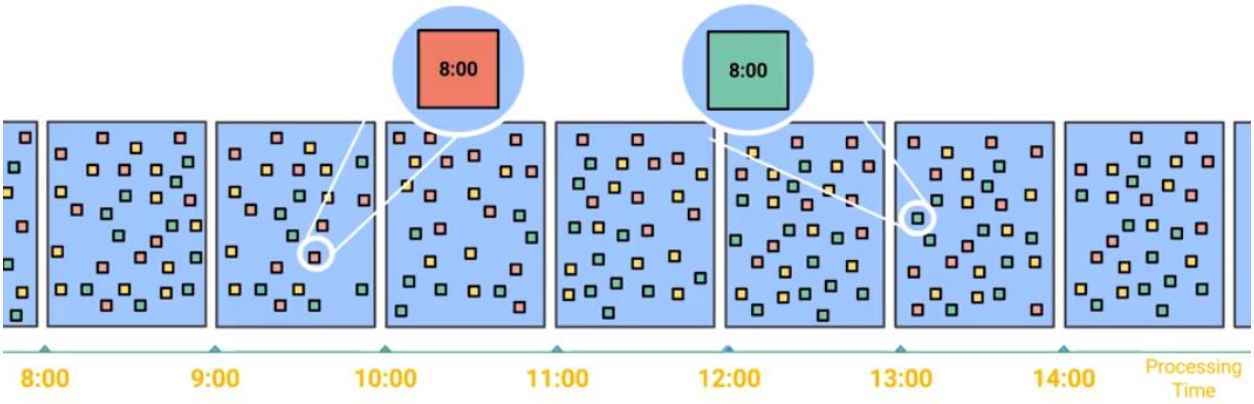
Until recently, it was necessary to write two sets of code. One set of code that handled batch data where the objective is to be as accurate as possible. Another set of code processed streaming data under the assumption that because data comes in a stream, accuracy cannot be guaranteed. There will be latency and missing data. In essence, it was necessary to compute a speculative result and populate the dashboard, but then it would be necessary to come back later and recompute everything. This became the "golden source of data" which would be saved.



You would build two pipelines:

- one pipeline that has low latency but not very accurate, very speculative
- a second pipeline that is very high latency (to handle cases where data is not available till much later as shown above), but it is accurate and you also balanced throughput.

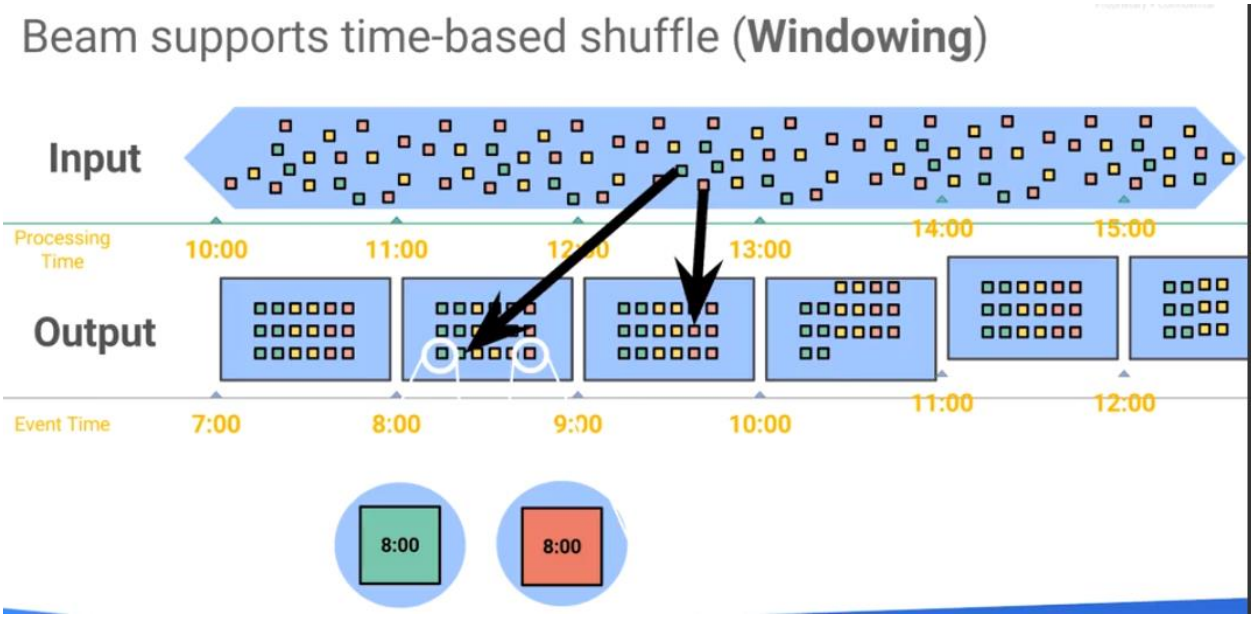
The key was to handle the fault tolerance in building these two data pipelines which was a state of art. It was necessary to build two or more models – sometimes as many as however often it was necessary to revisit the data model. That is very wasteful. But it was necessary to handle data that arrives out of order. Because the data has such latency and continuously arriving data can come out of order, final calculations had to be delayed and it was necessary to build two pipelines.



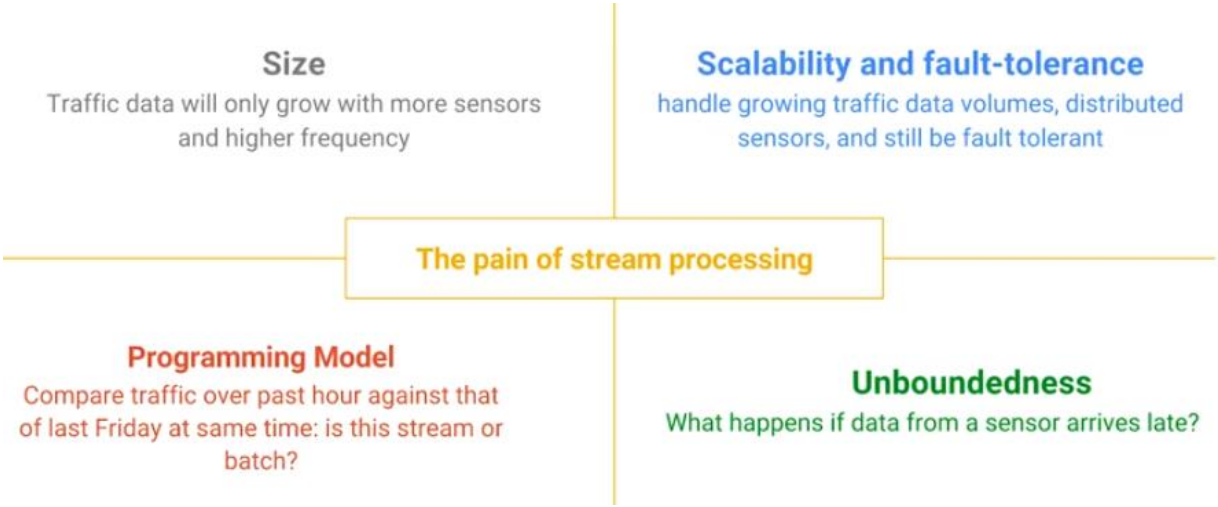
We could split the data into processing time windows … but we would lose information about related time events.

A Programming Model for both Batch AND Stream

A better solution is to have a unified model for batch and stream pipeline that supports the ability to analyze trade-offs and get accuracy, and that deals with low latency, *and* high latency compute situations. Apache Beam is an open source product which provides that solution. It was based on several internal products at Google such as Flume, but it has been open sourced. It is managed by the Apache Foundation, and it can run in multiple runtime environments, not just on Cloud Dataflow. Cloud DataFlow is an execution framework that runs Apache Beam pipelines. Note that Cloud DataFlow runs other frameworks such as Spark and Flink (https://flink.apache.org/).

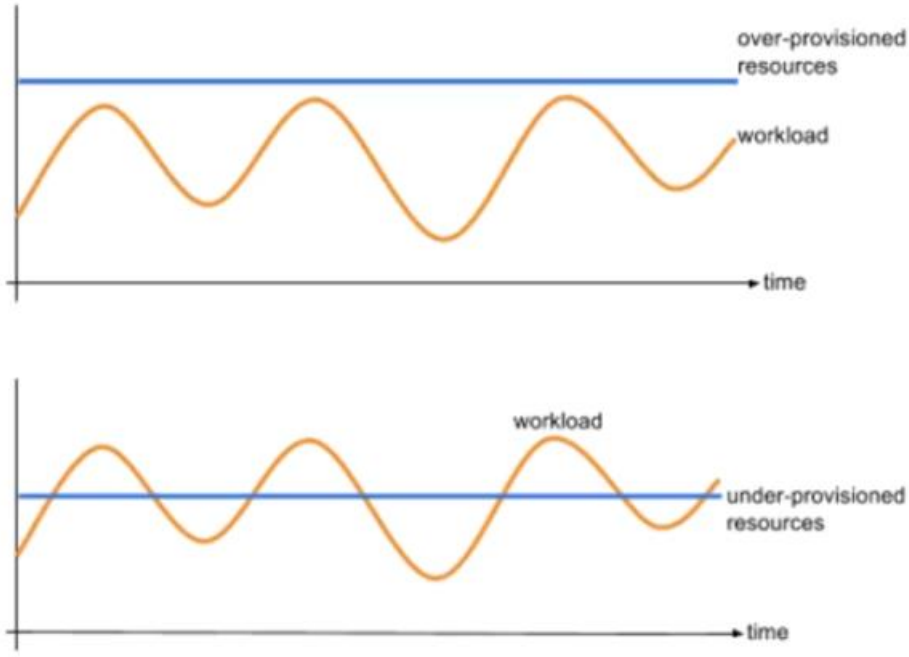


The key concept in Apache Beam is a *window*. A window provides a time-based shuffle. Define a window to cover some time frame. The window will receive all the data timestamped within that time frame, that is, any message with a timestamp in that time frame will be shuffled into that window *regardless of when the data is received*. This is a big distinction between processing time and event time. Long and term short latency are both handled since messages are shuffled into the proper window (by time frame) regardless of when they are processed. This allows us to use one model for all calculations regardless of whether they are streaming or some batched run at a later time.

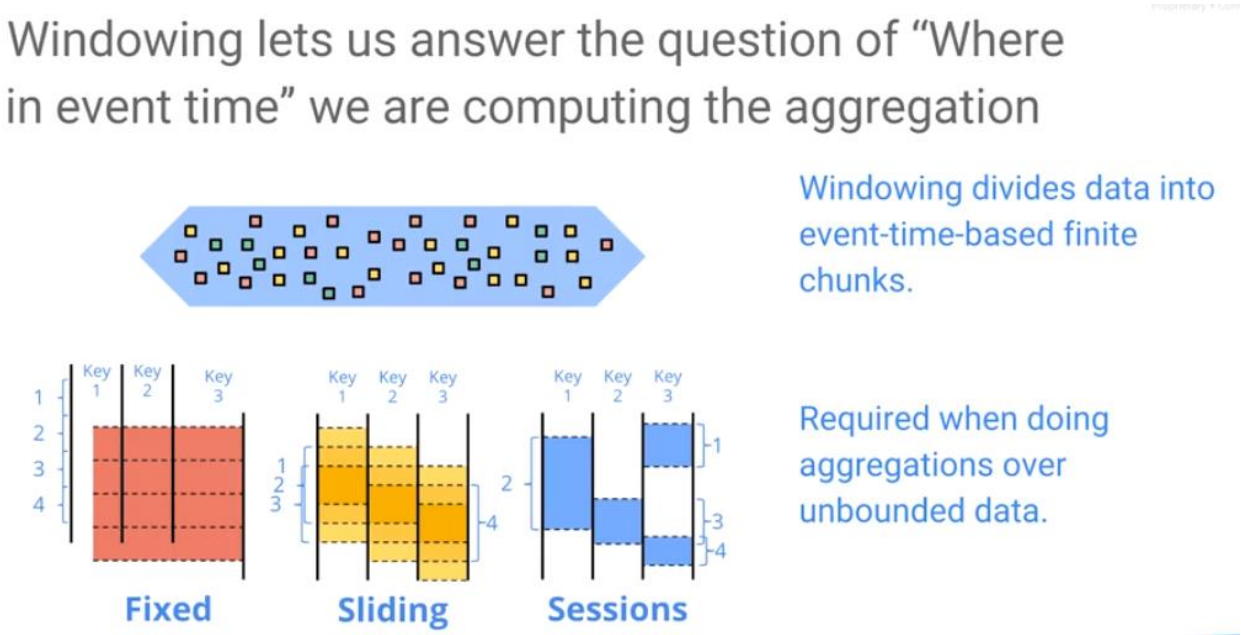


Challenges in Stream Processing

Fixed or slowly scaled clusters are a waste

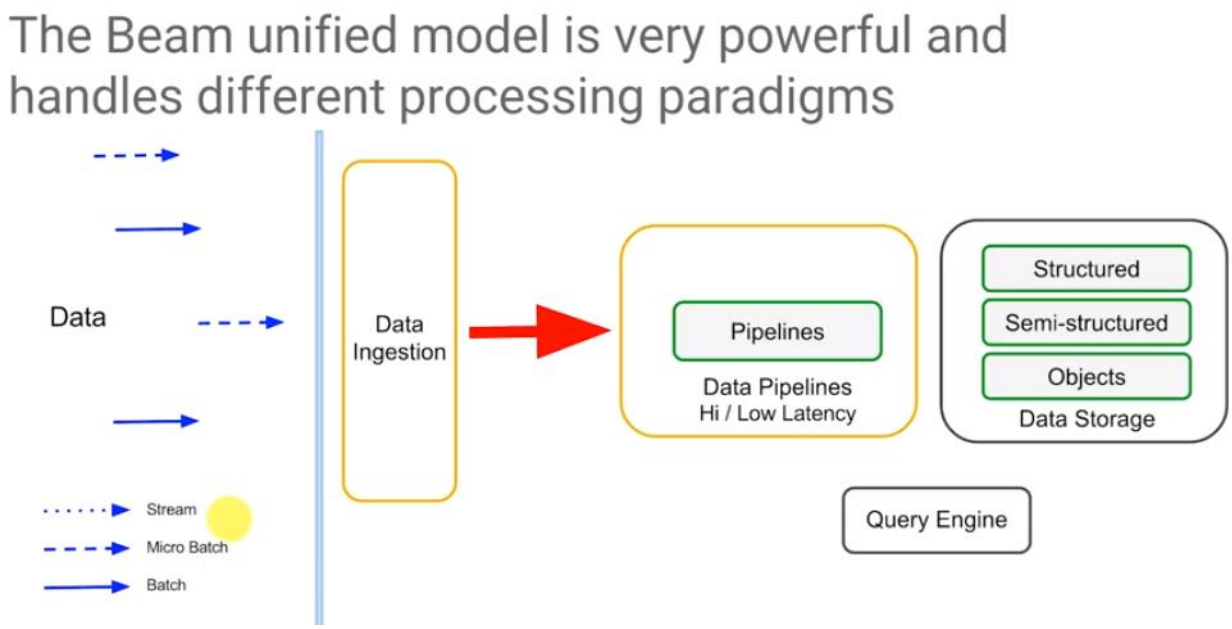


The first issue to address is that data changes over time. Not only is there more data, but the data velocity of the feed varies, and the data patterns change over time. How do we deal with the varying amount of data over time (velocity)? One way is to decide a priori what the work load will be. Create a cluster of fixed size – then there are issues with underprovisioning and overprovisioning – scalability problems. This has to be resolved by auto-scaling the clusters.



The second issue is the use of windows. The key is that we are concerned about event time, not about the time when we receive and process the message. Aggregates and other computations need to be calculated based on event time. It is necessary to address latency due to network delays, etc. when computing aggregates. Windowing allows the processing of an unbounded data stream, dividing the unbounded data stream into finite chunks of data.

- Fixed Window: separate the data into different absolute time periods, e.g., all data corresponding to stock trades that happened from 10:00am to 11:am, etc.
- Sliding Window: Define a window which is updated every five minutes and consider only the last hour of data, e.g., the stock trades that occurred in the last hour.
- Session Window: Gather all the data for a given user session, e.g., all the activity for a given user on a website. A session it could be based on domain or user – it needs to be defined.

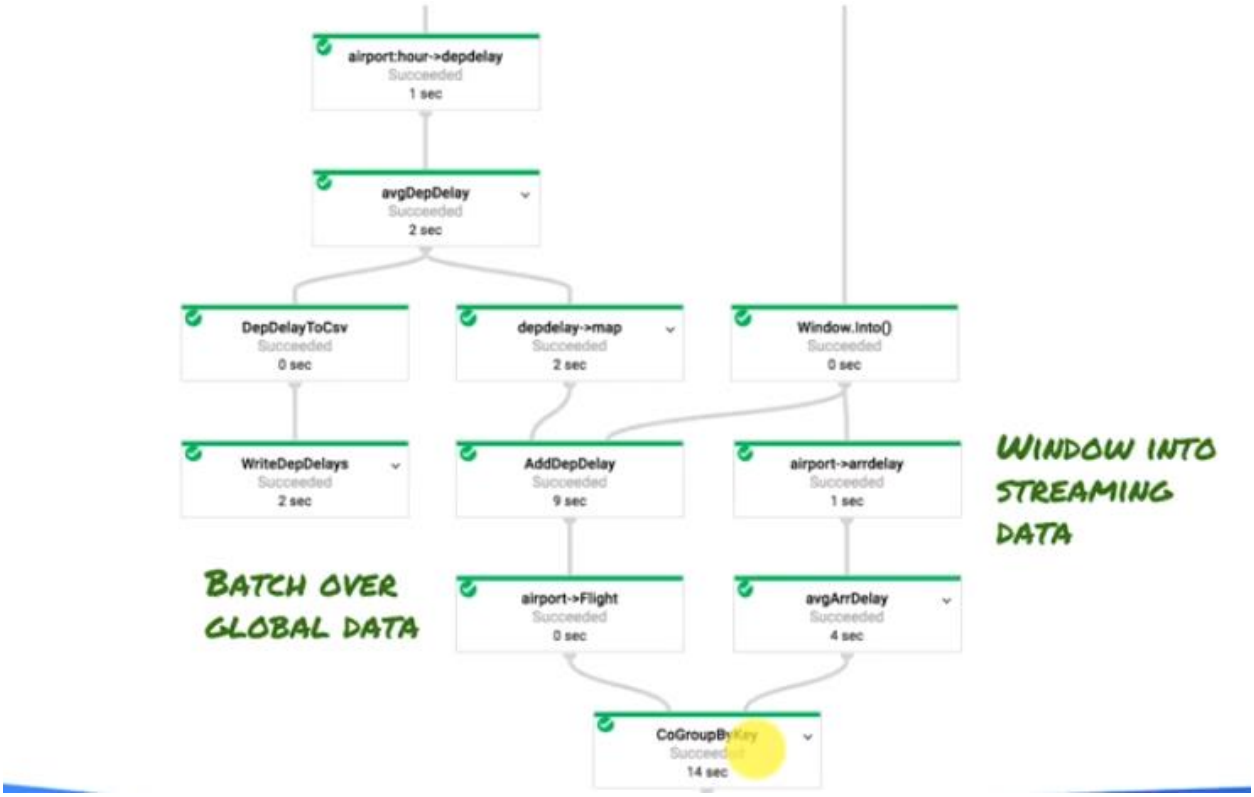


The Beam unified model is extremely powerful because supports different processing paradigms. Data can be treated as batch or subdivided into microbatches – that is, the data is bounded.

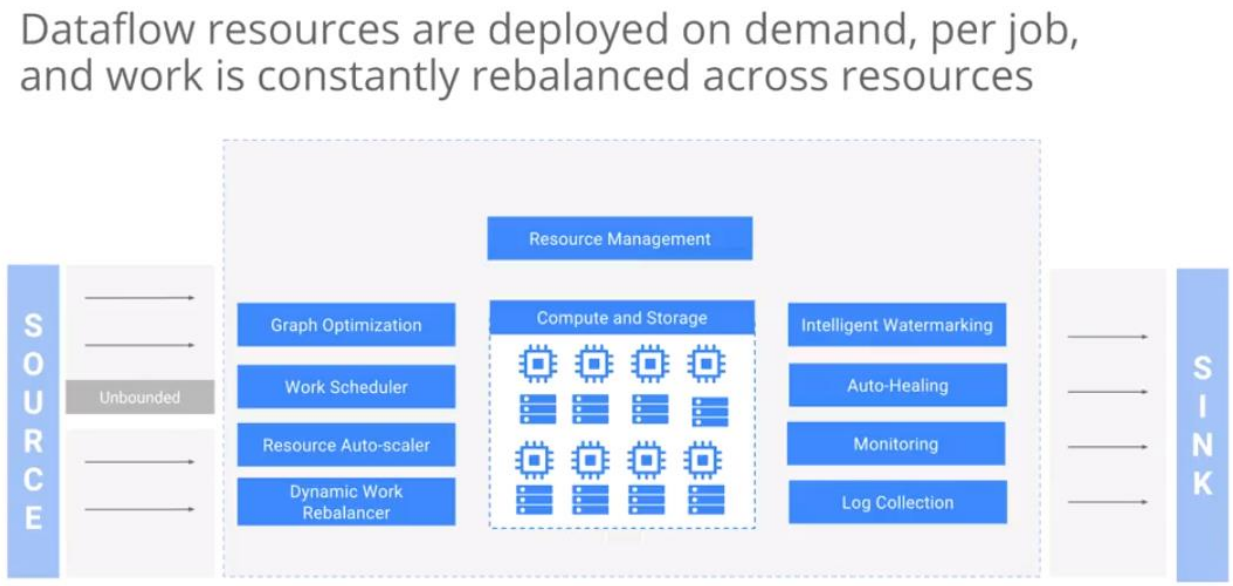
Beam also supports stream processing – handling unbounded data segregated by event time. And Beam gives you this flexibility and it lets you mix and match. There may be situations where calculations need to be done in both/either batch or stream and you want to do them in the same pipeline – Beam supports that.

When building a pipeline, Beam allows the choice between high latency and low latency. For example, perhaps this part of my pipeline needs to be batched – it needs to be very accurate and be high-latency (handles late arriving events). On the other hand, other parts of the pipeline could be speculative running at low latency, e.g., just publish a sum, but keep updating the sum.

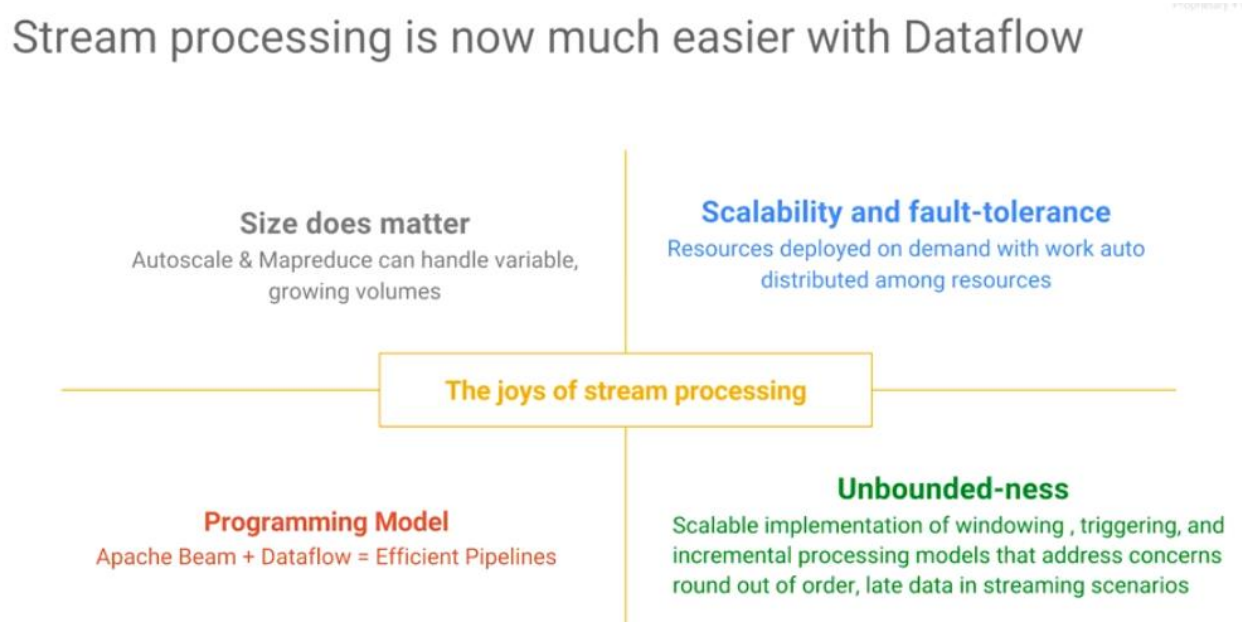
Beam supports different types of data: structured data, semi-structured data, object data, queries, etc. The Beam unified model is extremely powerful – it is a hammer that solves so many problems, so many issues. Furthermore, if run on Cloud DataFlow, Beam runs in an environment that is auto-scaling and fully managed.



Consider the example of batch processing and stream processing in the same pipeline. The objective of this pipeline is to predict the delay of a flight. Delays depend on which airport you are at and the flight network dependences. The dependency patterns can be characterized by the past behavior – it is necessary to do batch and streaming window the same pipeline. This is a surprisingly common thing and can be accomplished with DataFlow and Apache beam.



When running Apache Beam pipelines with DataFlow, you get auto scaling. The work constantly gets rebalanced across compute and storage resources. Dataflow also does watermarking – it monitors the network, determines which machines are overloaded, and heals them. It collects the logs and sends all this information to you. You can write a beam pipeline and run it anywhere, but Dataflow is an execution framework designed to run the beam pipeline very well.



The bottom line is that stream processing is a lot easier with Dataflow. Size does matter and Dataflow auto scales, peforms MapReduce, it lets you handle variable querying volumes. It is fault-tolerant, resources get deployed on demand, and work gets redistributed among the workers. The programming model of Apache Beam supports both batch and string. Apache Beam handles issues associated with unbounded data by giving you a good, scalable implementation of Windows triggers, of incremental processing, etc.

Beam Programming

Apache Beam Programming Guide: https://beam.apache.org/documentation/programming-guide/

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

Video about the code: https://www.coursera.org/learn/building-resilient-streaming-systems-gcp/lecture/dKFJ3/build-a-stream-processing-pipeline-for-live-traffic-data

DataFlow – A great way to work with Cloud PubSub

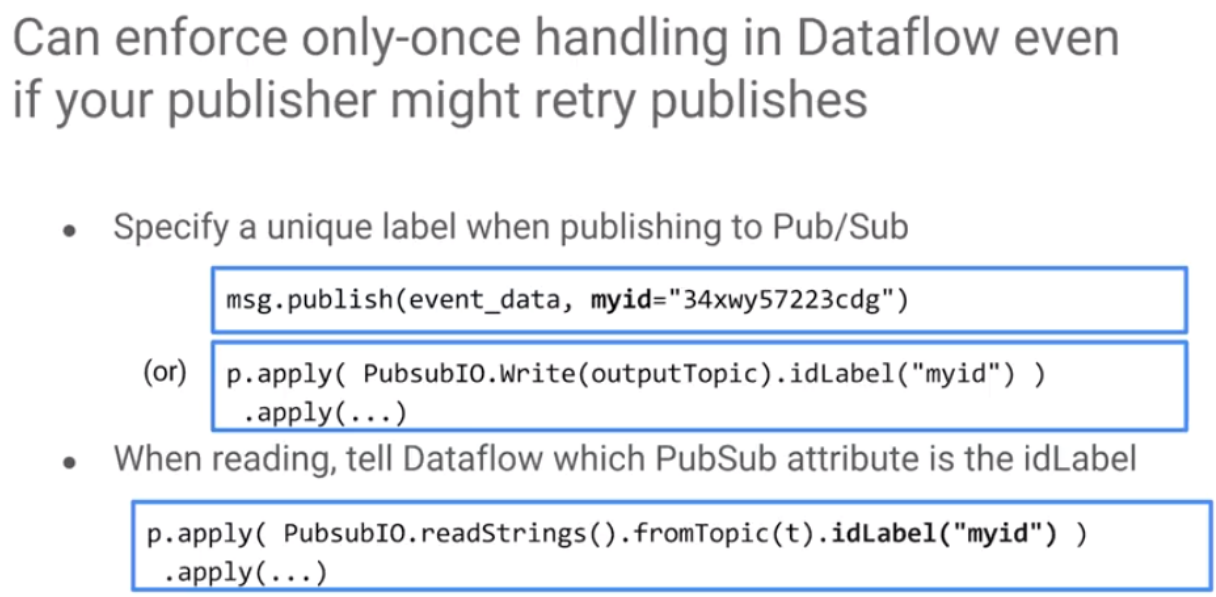
Recall that Pub/Sub is a low-latency global delivery service - a message bus. There are some issues that must be addressed when using PubSub:

1) PubSub does not guarantee order of messages.

2) It only gives you at-least-once delivery guarantee - it is possible that you have repeated delivery of the same message several times.

Stream processing in Dataflow addresses these problems. When computing aggregates, it works with out-of-order messages. Because you're shuffling, not by the time at which you've received it, but by the time at which the message was published, it works automatically with out-of-order messages.

Each message in Cloud PubSub has an internal message ID. Dataflow keeps track of which message IDs it has already processed. Dataflow uses the internal Pub/Sub ID to deduplicate it, remove it, and process each message only once. Dataflow provides exactly once processing guarantees.

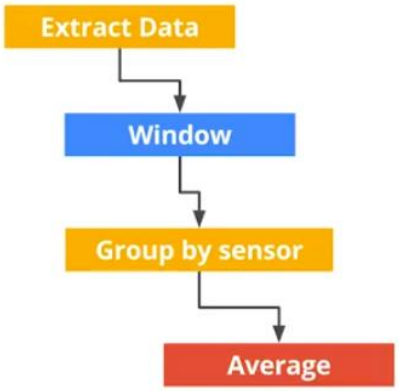


Only-once handling in dataflow is enforced automatically. But, this only works about messages that are going to PubSub. What happens if the person publishing to PubSub publishes multiple times? For example, you process data from a sensor, the sensor goes down, and when it comes back up, it resends messages. As far as Pub/Sub is concerned, this is a new message with a new message ID. The message cannot deduplicate since deduplication is based on Message IDs.

When you publishing to Pub/Sub we can specify attributes. Assign an attribute which is unique to the message (e.g., “myid”). When reading from Pub/Sub in dataflow, instruct DataFlow to use this new attribute as the ID/Label. DataFlow will now key off this attribute instead of using the PubSub ID.

[As an aside, a great way to get unique IDs across messages of different content is to hash the content and use that as the ID.]

To compute average speed on streaming data, we need to bound the computation within time-windows



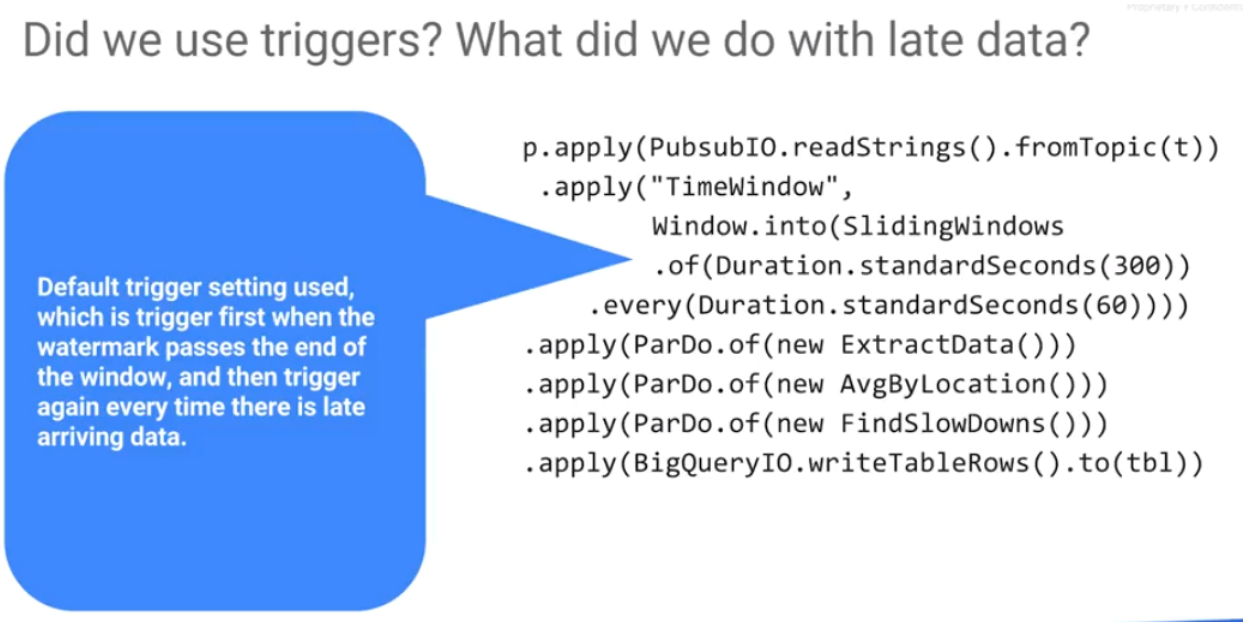

```
PCollection<KV<String, Double>> avgSpeed =
currentConditions //
        .apply("TimeWindow",
            Window.into(SlidingWindows//
              .of(Duration.standardMinutes(5))
              .every(Duration.standardSeconds(60))))
        .apply("BySensor", ParDo.of(new DoFn() {
          ...
            LaneInfo info = c.element();
            String key = info.getSensorKey();
            Double speed = info.getSpeed();
            c.output(KV.of(key, speed));
          ...
        })) //
        .apply("AvgBySensor", Mean.perKey());
```

What if we need to do aggregations? Consider the case where we want to compute average speed. If is hard to define average speed on an unbounded data stream. What is the average over something that has no beginning or end?

As a hint – consider a moving average. Think in terms of an average of a bounded time window. In fact, a moving average is an average in a sliding window.

By itself, just specifying a window on the event doesn't do anything. The key thing happens when you do an aggregation. After applying the window, somewhere later in the pipeline, there is an aggregation. In this case, the aggregate that is a mean per key. This requires some sort of a group by key.

To compute an aggregation (e.g., average) on streaming data, the computation must be bound within a time window.



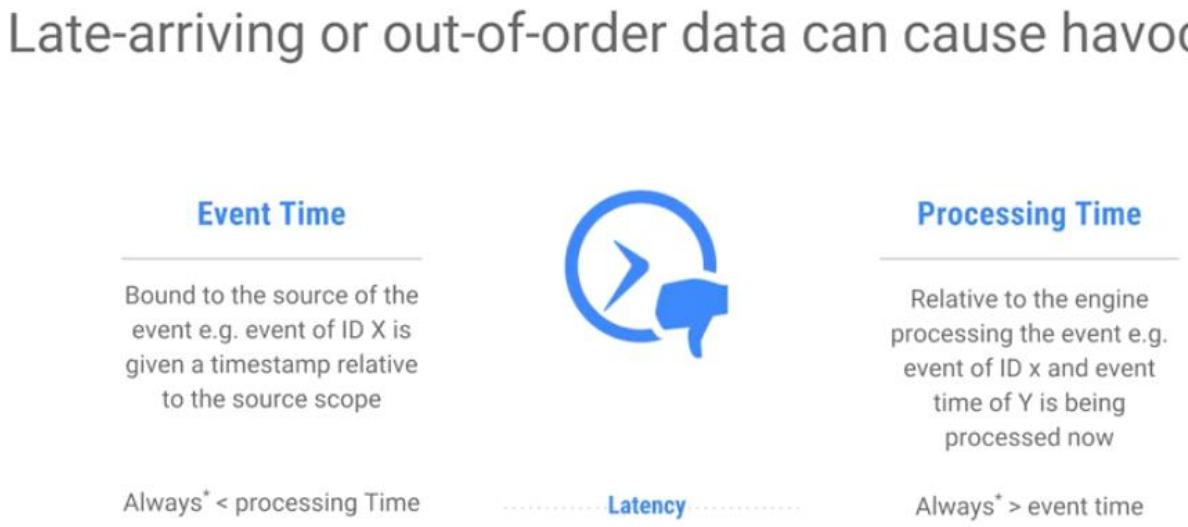
What happens to late data?

What happens if some data comes in late, and the window closed has been closed, the mean computed and passed on to the client. By default, a *trigger* happens when you have a *watermark*. The trigger causes the mean to be written out; late data will come after that watermark. When late data arrives, this mean will be recomputed. Every time you get a new (late) record, the mean is recomputed, and going to stream it out again. DataFlow performs this operation with a default trigger setting. The default trigger errs on the side of accuracy. When the window closes, the mean (an accumulation) is written out, it is a speculative result. A new data point comes in for a window (late), the mean is recomputed, and written out again.

Obviously, you want to be able to change this. The way you change this behavior - to do it the most accurate way, will require some tradeoffs. This has to do with watermarks and triggers.

Handle Late Data: Watermarks and Triggers

To change the default of erring on the side of accuracy provides different ways of dealing with late data. Watermarks, triggers, accumulations.



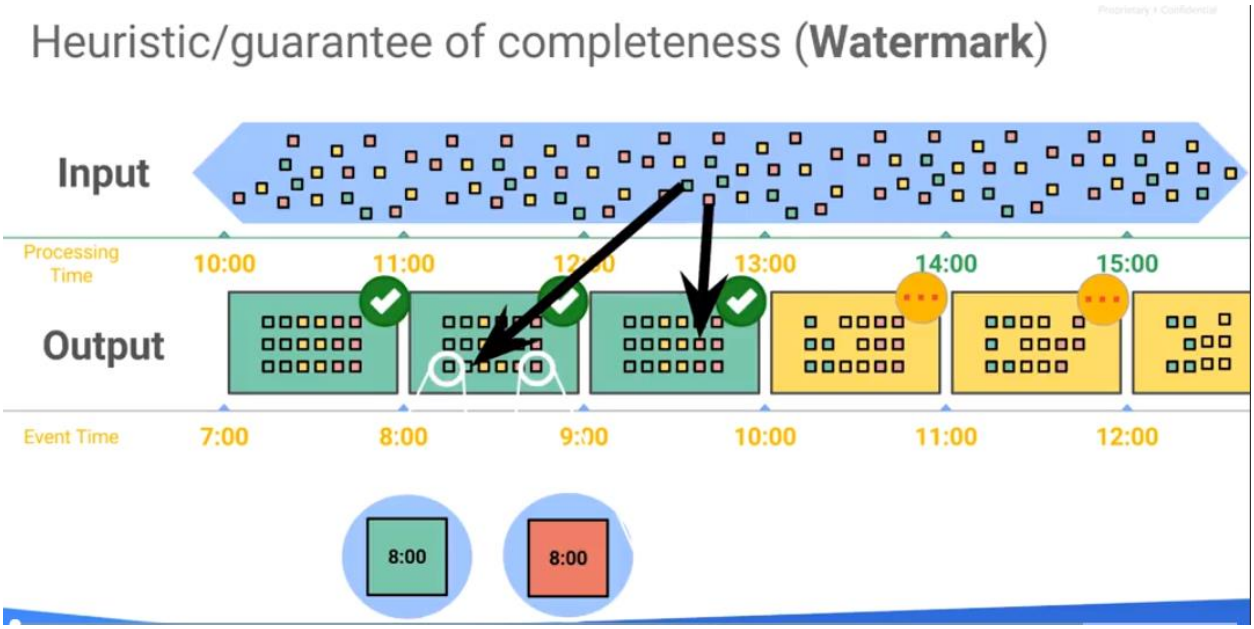
There are two issues to consider:

1. event time which is bound to the source of the event. For example, the time at which something was published into Pub/Sub. And then you have the processing time.
2. processing time is relative to whoever is processing this event. This machine, the workers on which Dataflow is running. In between the publication and the processing, there will be the network latency. Processing time is greater than the event time.

The processing time are typically later than the event time, but sometimes the clocks aren't synchronized, etc. However, in ideal conditions with synchronized clocks, the processing time is always greater than the event time, because there is always a non-zero latency.

This skew, the difference between the ideal and the actual event time and the processing time needs to be tracked. DataFlow tracks this time skew.

This skew is called a *watermark*: the watermark is Dataflow tracking how far behind the processing time is from the event time.

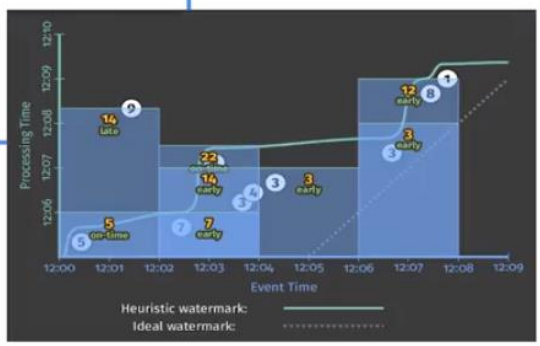


The watermark is a heuristic that's computed from the stream. As the stream comes in, everything gets pushed into Pub/Sub, As an example of a trigger, Pub/Sub may note that a message was received at 8 o'clock. DataFlow records that it is 8:03, therefore the skew is three minutes.

DataFlow averages the skew over a period of time - it learns the heuristic. It is possible to define the watermark provide a guarantee of how complete the data is. DataFlow knows how Pub/Sub assigns its message IDs, so it can determine if for a particular message, it has received all the messages before this particular message. That could be a heuristic, that could be a guarantee. The watermark measures how complete this message set is.

Scenario 4: Streaming with speculative + late data

```
PCollection<KV<String, Integer>> scores = input
    .apply(Window.into(FixedWindows.of(Minutes(2))
              .triggering(AfterWatermark()
                .withEarlyFirings(AtPeriod(Minutes(1)))
                .withLateFirings(AtCount(1))
              .withAllowedLateness(Minutes(30)
          )
    .apply(Sum.integersPerKey());
```

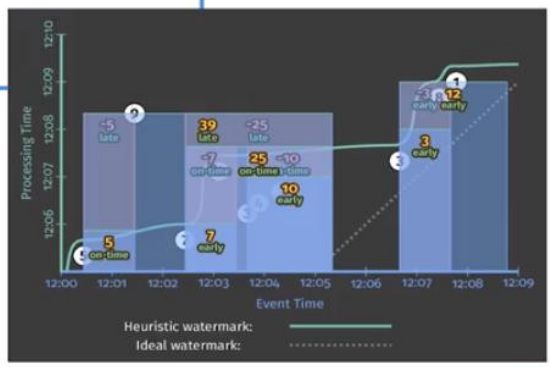


Consider a more complex triggering. Note that we can trigger at the watermark, but it is possible to also trigger WithEarlyFirings, e.g., trigger a minute early.

This trigger event will trigger one minute before the watermark and whenever there is one late record, e.g., AtCount(1). With AtCount(10), trigger after receiving 10 late records. At one minute before the watermark, close the window. And then, every record that arrives after the AtCount (1) trigger, close the window, refine the result. But the allowed lateness is only 30. If I got a record that is more than 30 minutes late, discard it.

Scenario 5: Session windows

```
PCollection<KV<String, Integer>> scores = input
    .apply(Window.into(Sessions.withGapDuration(Minutes(2))
              .triggering(AfterWatermark()
                .withEarlyFirings(AtPeriod(Minutes(1)))
    .apply(Sum.integersPerKey());
```



To do a session window, do sessions with a gap duration of two minutes. The case is to treat all of the events that happen as one session until there are two events that happen at least two minutes apart, and then that becomes the next session. To start the next session, trigger at the watermark, with an early firing of a minute.

In your DoFn, can get information about Window, Triggers

```
.apply("...", ParDo.of(new DoFn<Flight, Flight>() {
      @ProcessElement
       public void processElement(ProcessContext c, IntervalWindow window) throws
Exception {
                Instant endOfWindow = window.maxTimestamp();
       }
}))//
```

In the DoFn, information can be retrieved about the window. Normally, we've just done a process element with a process context, but you can add an extra window parameter to your process element function to get information about the window itself.

Lab: Streaming Data Processing

Lab: https://codelabs.developers.google.com/codelabs/cpb104-dataflow-streaming-pipeline/#0

Review: https://www.coursera.org/learn/building-resilient-streaming-systems-gcp/lecture/kPEBt/lab-review

MODULE 4: STREAMING ANALYTICS AND DASHBOARDS

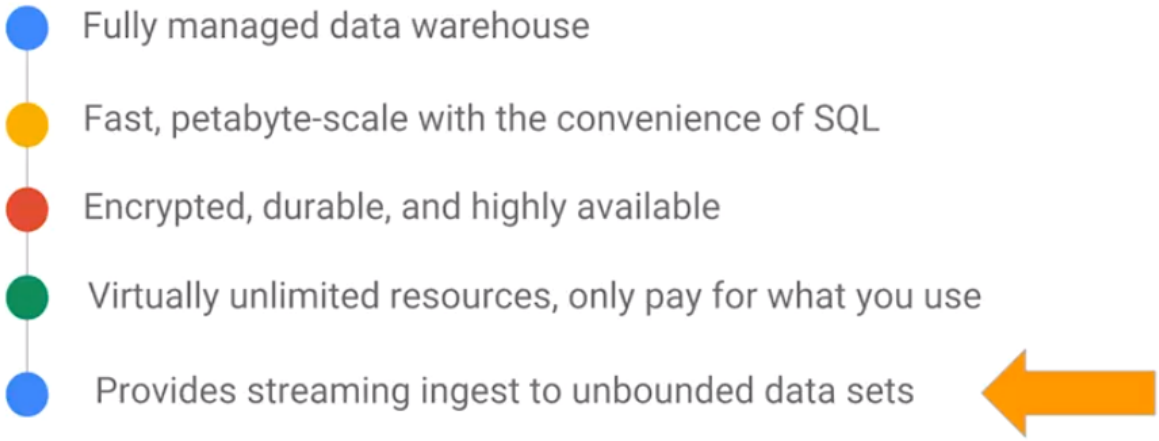
We considered two of the challenges that we started out with:

1) handle variable data volumes during ingest. Pub/Sub provides a way to address that.
2) Be able to perform continuous processing of the data as it's streaming in. Cloud DataFlow provides a method to do that using the concepts of watermark, trigger, and windows to control how to do aggregations and how to make these computations.

The third aspect is that even as the data comes in, it is desirable to do ad hoc analysis. We want to be able to power dashboards. To do so, we consider how to leverage BigQuery.

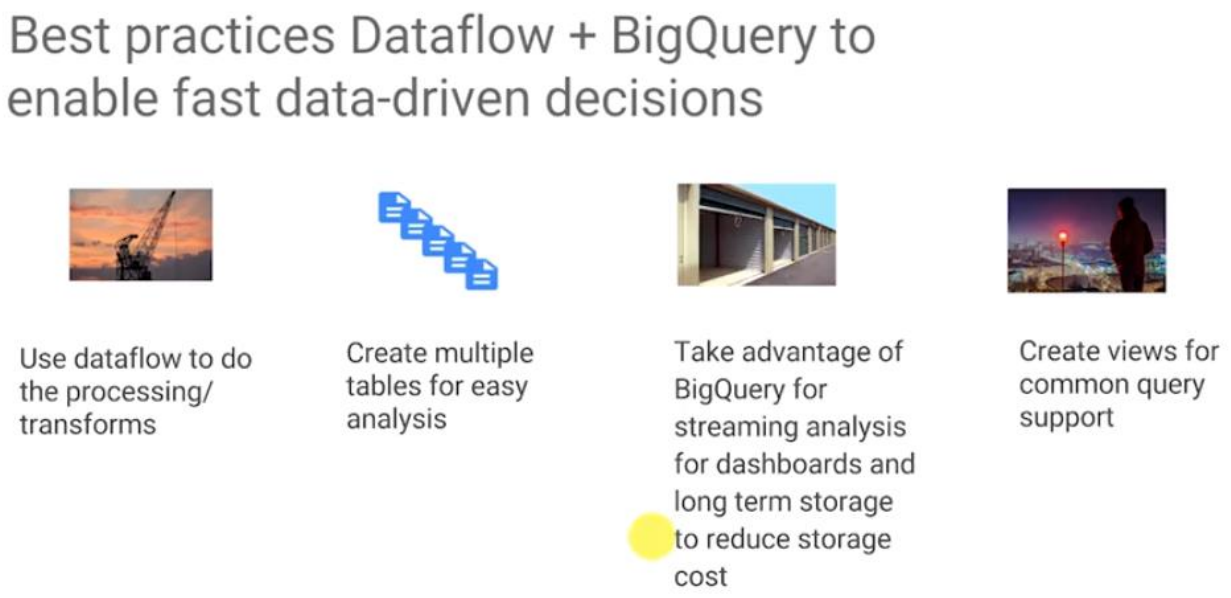
Sometimes in order to handle the datasets for feeding dashboards is that the data has to be written to disk. The problem with that, of course, is latency. Perhaps data is flushed to disk once every five minutes. In that case, “real time” data processing systems have a granularity of five minutes. With BigQuery, it will be possible to perform ad hoc analytics and dashboards even as the data are streaming in.

What is BigQuery?



BigQuery is a fully managed data warehouse. BigQuery separates out computer and storage and scales to petabytes. There is no need to launch a cluster or set up a database instance. As it is fully managed, it is always available and all that is required is to make SQL queries. The data, as is true with everything on GCP, it's encrypted at rest, and it's encrypted in flight. This is true of Pub/Sub and BigQuery as well. All these services are durable. BigQuery is highly available. In addition, you only pay for what you use.

BigQuery also provides streaming ingest to unbounded data sets. It allows you to stream data into BigQuery to the streaming buffer and carry out SQL even as the data are streaming in. The best practice is to combine Dataflow and BigQuery. Routine processing, the things that you always have to do over and over again, is performed in Dataflow. Then, allow Dataflow to write out tables in BigQuery. Those tables allow you to carry out data-driven decisions, do ad hoc queries, and power dashboards.



Take advantage of BigQuery for long-term storage, right. BigQuery gives you long-term storage at about the same cost as Cloud Storage so it is very cheap. And finally, BigQuery create views that provide common query support. These views can be shared to allow others to power their dashboards.

Streaming data into BigQuery

- BigQuery provides streaming ingestion at a rate of 100,000 rows/table/second
  - Provided by the REST APIs `tabledata().insertAll()` method
  - Works for partitioned and standard tables
- Streaming data can be queried as it arrives
  - Data available within seconds
- For data consistency, enter `insertId` for each inserted row
  - De-duplication is based on a best-effort basis, and can be affected by network errors
  - Can be done manually

BigQuery handles streaming up to about 100,000 rows per table per second. BigQuery provides a REST API to stream data in. DataFlow calls BigQuery's REST API to interface with it. The API is exposed outside of DataFlow. It works for standard tables. It also works for partition tables. If tables are partitioned by date, you can stream into those as well.

BigQuery supports queries on streaming data as it arrives. Normally, BigQuery will deduplicate records. If for whatever reason, the same record is inserted twice, it is deduplicated. But this is on a best-effort basis. If you want to be absolutely sure that data is deduplicated when you insert, provide an InsertId for every row. In that way, data rows will be deduplicated. However, normally you don't bother, it's fine. Data is inserted by DataFlow into BigQuery. Once the data is in BigQuery, it can be queried.

The stuff that powers your near real-time dashboards

```
SELECT
  s.*
FROM
  demos.current_conditions AS s
JOIN (
  SELECT
    sensorId,
    MAX(timestamp) AS timestamp
  FROM
    demos.current_conditions
  GROUP BY
    sensorId) AS c
ON
  s.sensorId = c.sensorId
  AND s.timestamp = c.timestamp
```

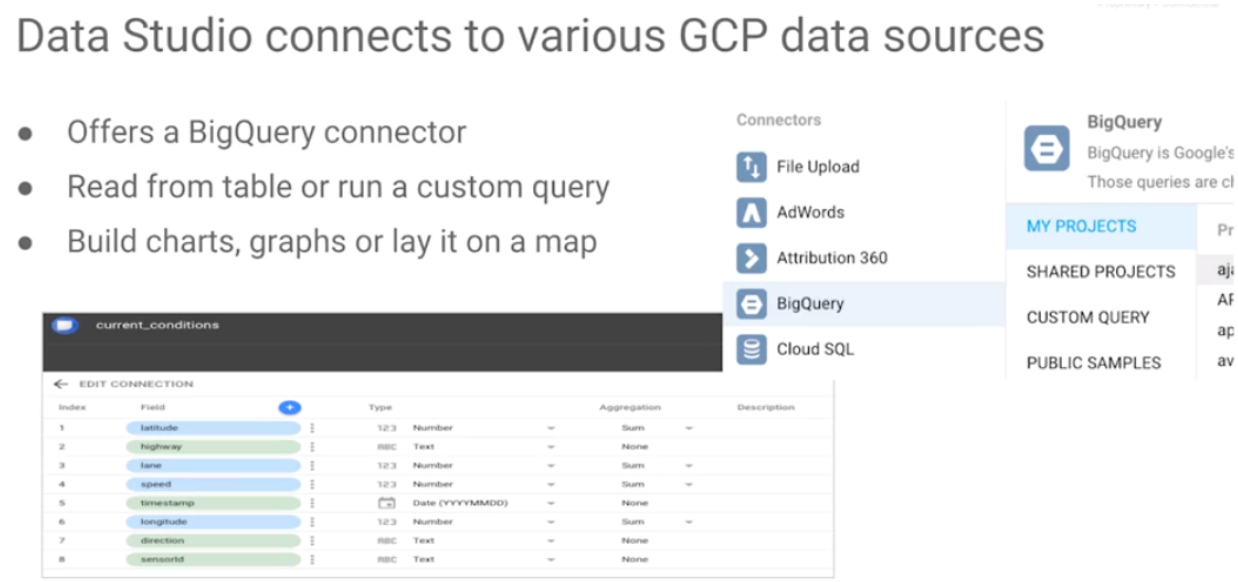
1	2008-11-01 02:10:00 UTC	32.67696	-117.109463	5	S	4	65.0	32.67696,-117.109463,5,S,4
2	2008-11-01 02:10:00 UTC	32.817541	-117.160122	805	S	5	65.4	32.817541,-117.16012,805,S,5
3	2008-11-01 02:10:00 UTC	32.743125	-117.184141	5	S	5	65.3	32.743125,-117.184141,5,S,5
4	2008-11-01 02:10:00 UTC	32.759802	-117.187519	8	W	2	69.2	32.759802,-117.187519,8,W,2
5	2008-11-01 02:10:00 UTC	32.67696	-117.109463	5	S	4	65.0	32.67696,-117.109463,5,S,4
6	2008-11-01 02:10:00 UTC	32.793785	-117.149874	805	S	3	81.7	32.793785,-117.149874,805,S,3
7	2008-11-01 02:10:00 UTC	32.762484	-117.163806	8	W	3	64.5	32.762484,-117.163806,8,W,3

In this example, there is an inner query, select sensorID and MAX of timestamp AS timestamp, from this table GROUP by sensorId. The inner query gives us the latest timestamp for every sensor. We can determine what was the last received data from every sensor. This is what is used to power real time dashboards with the latest data.

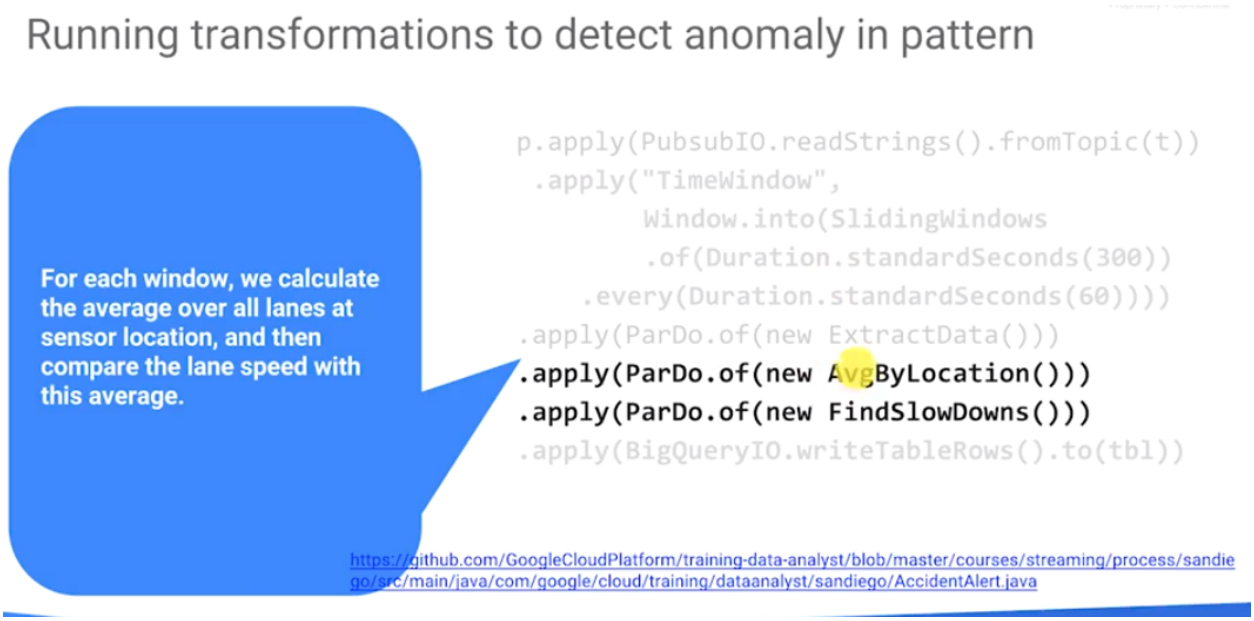
Data Studio lets you build dashboards and reports

- Easy to read, share, and fully customizable
- Handles authentication, access rights, and structuring of data

This dashboard itself can be built with a variety of dashboard tools and can be shown in Data Studio. Data Studio is a dashboard tool by Google. It is free. It is convenient to use. However, there are a variety of other dashboard tools which also can be used. Data Studio lets you build a dashboard and other reports. It works a lot like Google Docs, in the sense that you can create a dashboard, and you can share it with people, and you can collaborate on a dashboard. It is very good for collaboration and sharing scenarios.

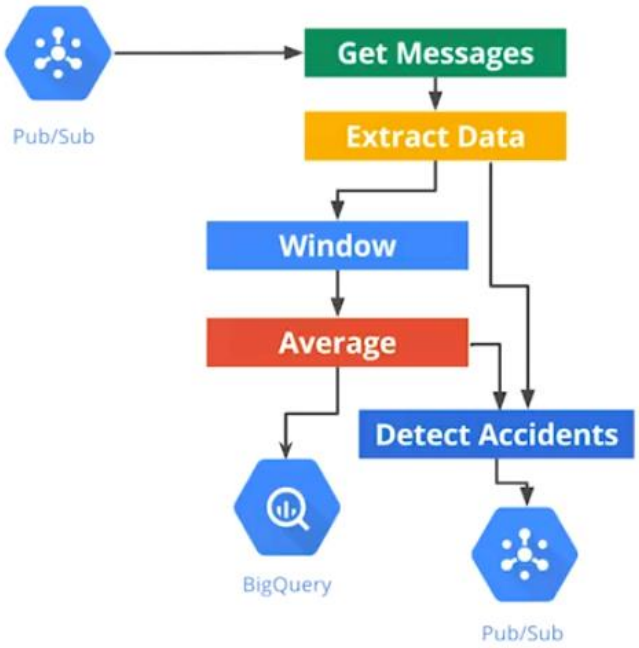


You can do a variety of charts. The charts could be as simple as bar charts or charts of different types of maps. Data Studio can create all these graphs. It can connect to a variety of different GCP data sources: Cloud SQL, Bigtable, BigQuery etc. In particular, Data Studio offers a BigQuery connector which can read from a table, with a custom query, and you can use the queries to populate it. Data Flow is very good for handling an analysis which is repeated often or performed continuously. It can be integrated with Pub/Sub, data can be windowed, and an aggregation can be computed.



Pipeline to detect accidents

In the upcoming lab, the graphic represents the pipeline to detect automobile accidents. Messages are received from Pub/Sub. We extract out the data - information associated with the sensor on a lane.



Meanwhile, while this is occurring, compute the average speed of all of the sensors at some location. Average doesn't make sense in an unbounded data stream just by itself. It's not an average, historically, over the entire data set, because there is no entire data set. Instead, compute the average at this location on the windows spanning the past few minutes. Detect Accidents is going to take the average at that location, and the actual speed of that lane. It compares and also output all the accidents to Pub/Sub (for example). In addition, it's going to take the average speeds and write the average speeds to BigQuery. This is the complete pipeline.

In this lab, complete Data Flow code to find accidents. Build a dashboard so we can look at data that is streaming into Bit Query rather than typing SQL queries - create nice, impactful graphs.

Lab: Streaming data analytics and dashboards

In this lab, you will

Create a streaming pipeline to process traffic events from Pub/Sub
Perform transformations on the data (e.g., averages) and detect anomalies (compare averages and detect which lane is slower than rest
Store results in BigQuery for future analysis

In the streaming pipeline to process traffic events, perform transformations in the data that averages to detect anomalies, to detect which lane is slower than the rest and store all the results in BigQuery for future analysis. Also store accident data to BigQuery.

Lab: https://codelabs.developers.google.com/codelabs/cpb104-bigquery-datastudio/

Lab Review: https://www.coursera.org/learn/building-resilient-streaming-systems-gcp/lecture/eycpA/lab-review

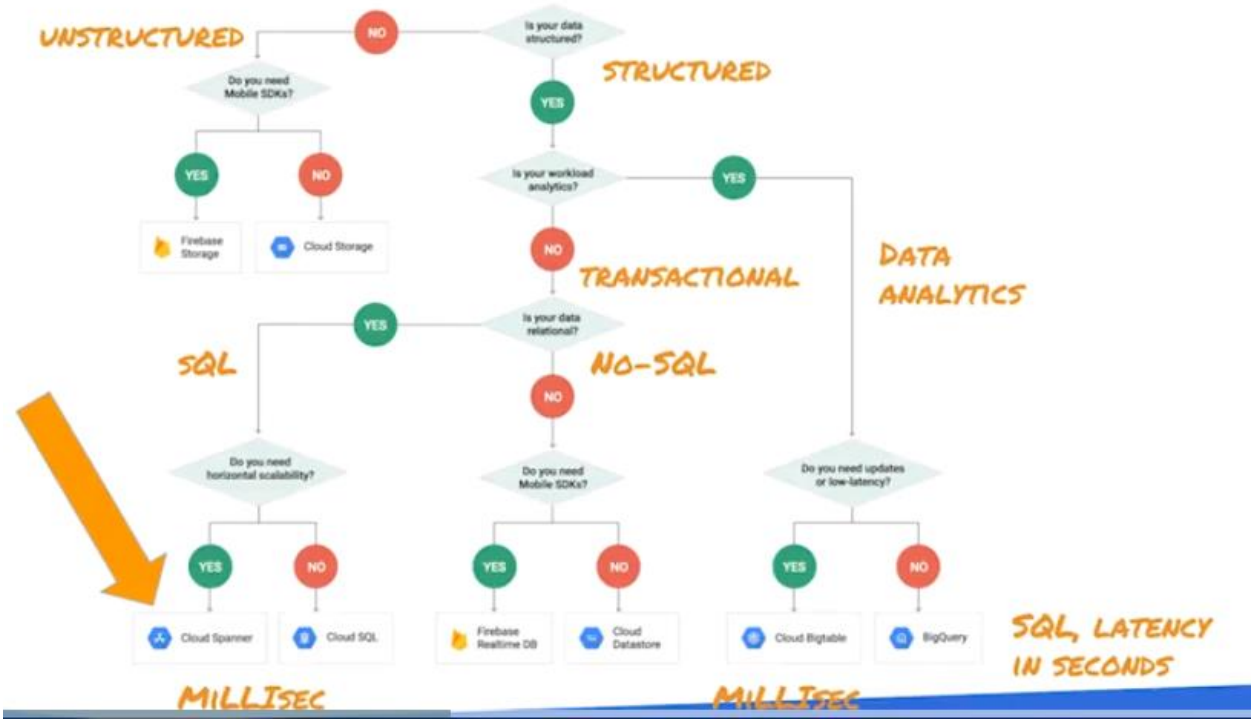
MODULE 5: HANDLING THROUGHPUT AND LATENCY REQUIREMENTS

So far in this course, we have looked at streaming data, how to build resilient streaming pipelines, how to create variable in variable volume ingest, how to process data that could be late or unordered using Dataflow, and finally looked at how to do queries on data even as it's streaming using BigQuery, and displaying that data with Data Studio.

BigQuery is a very good general-purpose solution, something that would work in most cases. However, every once in a while, the latency of BigQuery will be problematic. In BigQuery, the data that is streaming in is available in a matter of seconds. There are latency issues throughout BigQuery, which has a throughput of about 100,000 records a second which may not be enough. When BigQuery is not enough, where do you go?

In this module, we discuss Cloud Spanner and Bigtable. We consider how to design for Bigtable. Specifically, how to design schemas and how to design the row key for Bigtable. We consider how best to ingest data into Bigtable. There is a lab that takes a Dataflow pipeline which it streams into BigQuery and modifies it so that it is streaming streaming with 30 times more data. The objective is to reduce latency such that information is available in a matter of milliseconds, for example, or microseconds.

Choosing where to store data in GCP



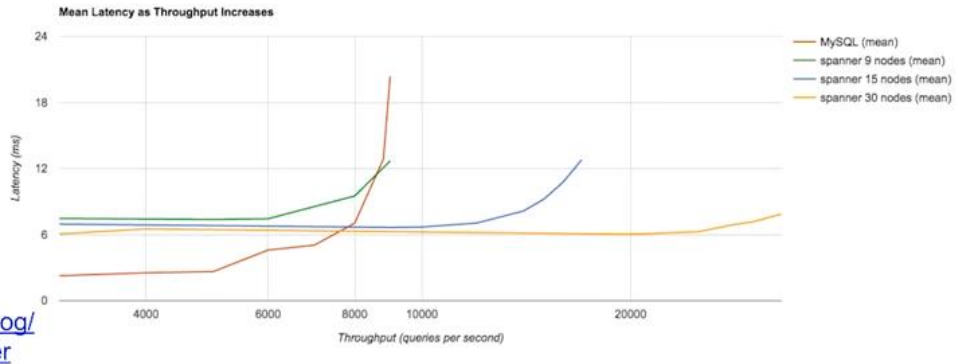
Where should I store data on GCP? Here are the questions to consider:

1) Is the data that to store structured or unstructured data?
2) If it is structured data, then the next question is: are transactions important? Or is the workload primarily read-only? Is your workload primarily around data analytics?
3) If considering transactional workloads, do the transactional workloads SQL or noSQL? In other words, is your data relational? Perhaps it is desired to do object stores.
4) For structured transactional data supporting SQL, there are multiple options.
    - The most common option is to use Cloud SQL. Cloud SQL is a single database. If you need multiple databases or horizontal scalability, then Cloud Spanner is a good solution. Both Cloud Spanner and Cloud SQL will give millisecond latency.
    - If the workload is not transactional, but is for data analytics, then the overhead of locking and just adds extra cost. At that point, consider if updates are required. Low-latency? If neither transactions or low latency are required (order of seconds is OK), then BigQuery is your most cost-effective solution. BigQuery provides an SQL interface).
    - On the other hand, if millisecond latency is required, but the data is for analytics workloads, Bigtable is a good solution.

Unstructured data ideally goes into cloud storage. But if you need mobile SDKs, then Firebase storage would be a good solution for unstructured data. If you don't need mobile, if it's primarily web applications, put it into Datastore. BigQuery is the most common and cost-effective and has latency of seconds. Cloud SQL is also very common for relational and transactional data and is backed by either a MySQL database or a PostgreSQL database.

If you have transactional SQL data and your workload is much larger such that the data does not fit in a single database (you need horizontal scalability), use Cloud Spanner. If you are doing data analytics with more data than BigQuery can support, use Bigtable.

Use cloud spanner if you need globally consistent data or more than one Cloud SQL instance



Source: https://quizlet.com/blog/quizlet-cloud-spanner

When do you use Spanner? Use Spanner if you need globally consistent data – that is the data is consistent over more than one region. You need more than one Cloud SQL instance. The graph shows the mean latency associated with databases. The red line shows us MySQL cloud SQL installation: it hits the wall at about 9000 queries per second. On the other hand, if you have a Cloud Spanner installation you get horizontal scalability. For a Cloud Spanner with nine nodes, you achieve low latency up to about 6000 queries per second. The blue line shows Cloud Spanner with 15 nodes giving good performance up to10,000 queries per second. Thirty (30) nodes supporst up to 20,000 queries per second, etc.

This is not eventual consistency, this is strong consistency on transactions in a global case. If you have throughput needs that are more than what a single Cloud SQL instance can handle or if you need to have multiple databases because you want to be committing from multiple regions in a global application, then Cloud Spanner is a great choice. Spanner and Cloud SQL are transactional data solutions: Spanner is a lot more scalable than Cloud SQL. Should you use Spanner for everything? Absolutely not. Use Cloud SQL if your throughput needs are more reasonable, if a single database is going to be enough. Use Spanner only when your applications scale beyond that. (Remember cost!)

How do you use Cloud Spanner?

It's a fully-managed SQL database Create it from web console (or gcloud) Interact with it via SQL (from Java, Python, etc.)
https://cloud.google.com/spanner/docs/quickstart-console https://cloud.google.com/spanner/docs/getting-started/python/

Cloud Spanner is a fully managed SQL database which can be created from a Web Console, just like a MySQL Database. The quick start guide gives guidance on the graphical user interface, the web UI, and how to create a console. You define how many server nodes, tables, and perform CRUD operations using SQL. There are a client APIs available in a variety of languages including Python.

Bigtable is used for data analytics on streaming data when BigQuery doesn't provide low enough latency – e.g., you need millisecond latencies. BigQuery is easy to use, inexpensive, and the latency is in the order of seconds, you get about 100,000 rows per second of streaming. Bigtable is low latency with very high-throughput. It is more expensive than BigQuery because you're paying for the number of nodes of Bigtable that you're running. But you get great performance: for example, a cluster with ten nodes, has query performance of 100,000 queries per second at six millisecond latency.

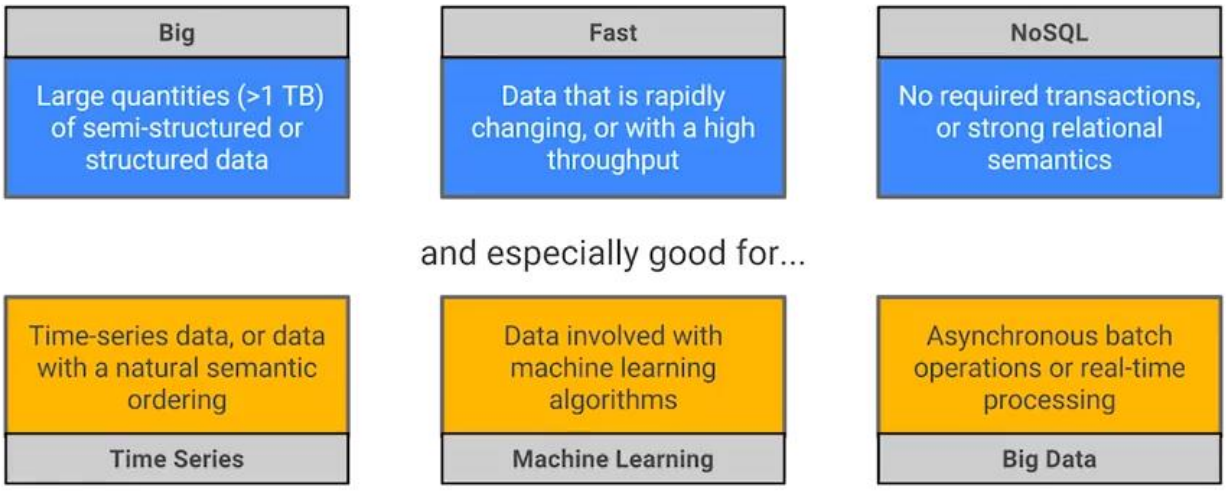
BigQuery:
easy, inexpensive

- latency in order of seconds
- 100k rows/second streaming

Bigtable:
low latency/high-throughput

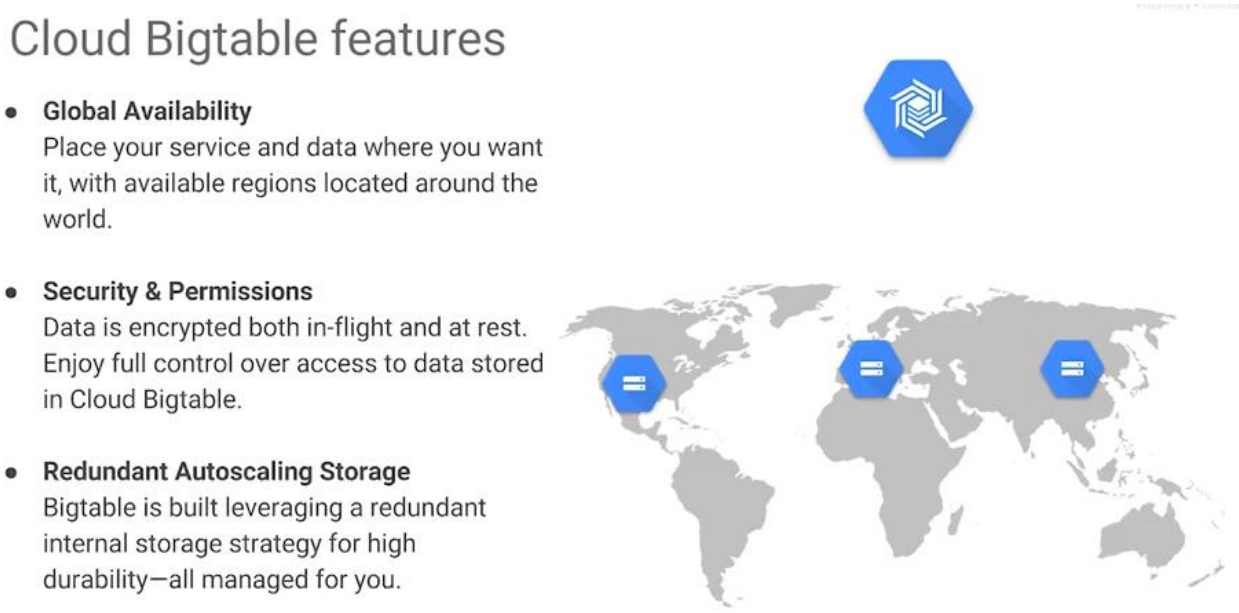
- 100,000 QPS @ 6ms latency for a 10-node cluster

Bigtable: big, fast, autoscaling NoSQL

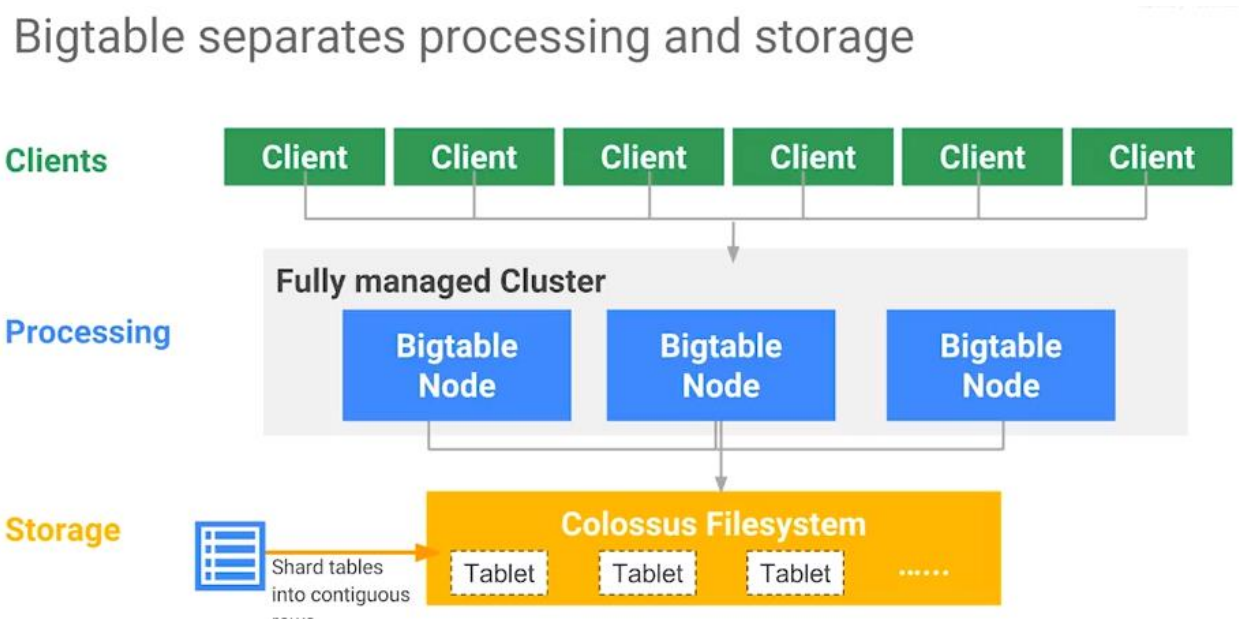


Bigtable handles more than one terabyte of data, it's fast, it's auto scaling, and it's NoSQL. Data can be semi structured or structured. It is meant for data that's very fast changing, that has a very high throughput, is not transactional, and does not require strong relational semantics.

What is Bigtable used for? It tends to be used for time series data, financial data, sensor data - data with this natural ordering in terms of time. It also gets used for real-time processing, asynchronous batch operations, and increasingly, it is used for data feeding machine learning algorithms that do continuous training.



Cloud Bigtable has global availability. You can put your service and data where you want but unlike Pub/Sub or BigQuery, it is not cluster free, so it must be architected in terms of a cluster of nodes. Bigtable encrypts data at flight, at rest, on the wire, and you get full control of the data. There is identity access management that is available in the platform. And you get redundant auto scaling storage. It is all the data that you put into Bigtable, is durable, it's replicated and you can get access to it.



Like BigQuery, Bigtable also separates out computing and storage. The clusters only contain pointers to the data, they don't contain the data itself. Clusters nodes keep only the metadata, the data itself remains on Colossus or on Google Cloud storage. The data should be viewed as being stored in contiguous rows. Rows of contiguous data are called tablets. All the tablets of data are stored on Google Cloud Storage. During processing, nodes basically carry out the operations and data gets shuffled into those nodes to do the

computation. When the nodes read the data, they read contiguous rows of data. This is going to be important when we think about the design of Bigtable. How to optimize the performance of Bigtable.

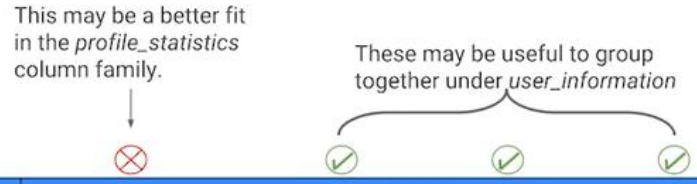
Designing for Bigtable

A table can have only one index (the row key)

	user_information				
Row Key	followers	birthdate	age	gender	messageCount
andrew	34	06_04_1986	34	F	1
brianna	31	07_24_1993	23	F	12
caitlyn	55	03_22_1952	51	F	15

How do we design for Bigtable with this understanding that Bigtable separates out computing and storage? Why does it separate out computing and storage? It separates so that the compute can be auto scaled. If there is a new cluster that comes up, what does that new cluster have to do? It only needs to point at the data, it doesn't need to copy the data. In order to do design the data schema in Bigtable, the first thing to realize is that tables in Bigtable have only one index which is the row key.

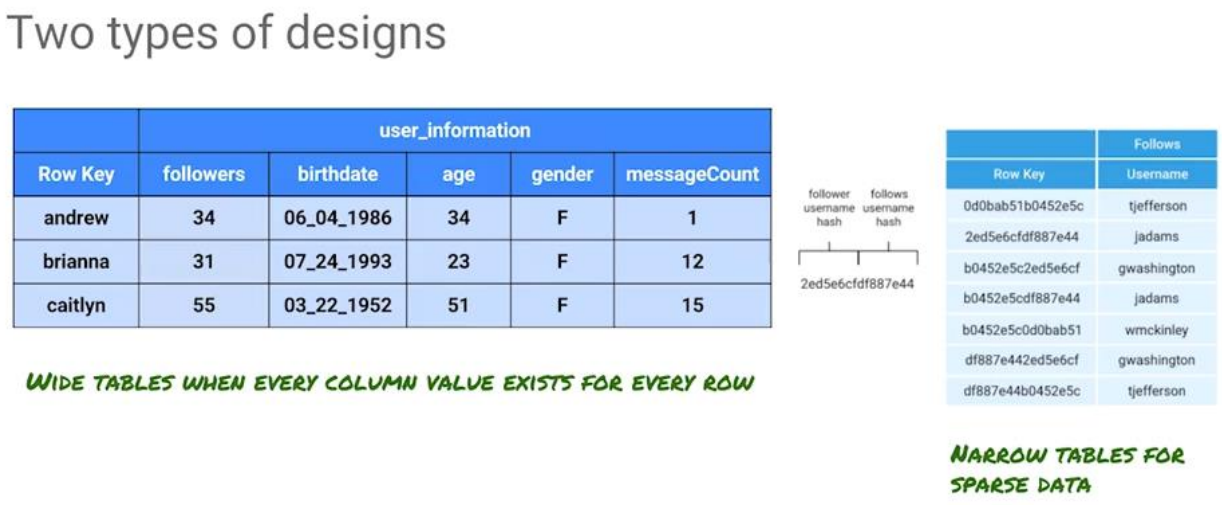
Group related columns into column families



	user_information				profile_statistics
Row Key	followers	birthdate	age	gender	messageCount
andrew	34	06_04_1986	34	F	1
brianna	31	07_24_1993	23	F	12
caitlyn	55	03_22_1952	51	F	15

What is stored in Bigtable is a key-value pair. The values can have columns, the columns contain column families but only one of those columns is the key. In this case, the first column is the row key and all of the

rows are stored in ascending order based on this row key. None of the other columns can be indexed. The only thing that's indexable is a row key. When designing schemas for Bigtable, take information that belongs together and group columns together into what are called column families. For the table shown above, there are two column families: user_information and profile_statistics. What kind of information should be collected together in column families. In this example, birthday, age and gender are all information about the user but the number of messages that the user has sent, that belongs in profile_statistics. User_information is very slow changing information and profile_statistics contains fast-changing information. Consider the number of followers that a user has, where should it be? Should it be in the user_information column family or should it be the profile_statistics column family? Once you say that the user information tends to be static information, it doesn't tend to come in streaming all the time but followers and messageCounts keep changing. Consider followers and the number of messageCounts as being this data that's constantly updated, so it may be good to move those followers into the profile_statistics. Column families contain related columns that tend to be updated together.



When we design Bigtable, you essentially choose among one of two design types. One type of design is where you have a wide table. A wide table is a table where you have a number of columns and each column value exist for every row. In the user_information table, every row is fully populated. This can be considered a wide table. On the other hand, there may be sparse tables, tables in which many of the columns will not have any data in them. For example, not every user may have rated a product. Creating a separate column family for sparse data, a narrow table, reduces waste. A narrow table is great for sparse data.

Each row has to have a unique key. The key consists of this pair of the follower and the person that they follow, and that becomes a unique key which can be stored as plain text; in this case the name of the follower or the name of the followers and other information about when did this follow happen etc. Data dense or data is sparse and depending on whether it's dense or sparse choose between one of these two types of designs.

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

Rows are sorted lexicographically by row key, from lowest to highest byte string

	fam1	fam2	fam3
Row Key	cols 1-20	cols 20-40	cols 40-60
Andrew	...	...	...
Jesse	...	...	...
Zachary	...	...	...
Sandy	...	...	...
Bob	...	...	...

Encoded to raw byte strings (not shown), sorted alphabetically

	fam1	fam2	fam3
Row Key	cols 1-20	cols 20-40	cols 40-60
Andrew	...	...	...
Bob	...	...	...
Jesse	...	...	...
Sandy	...	...	...
Zachary	...	...	...

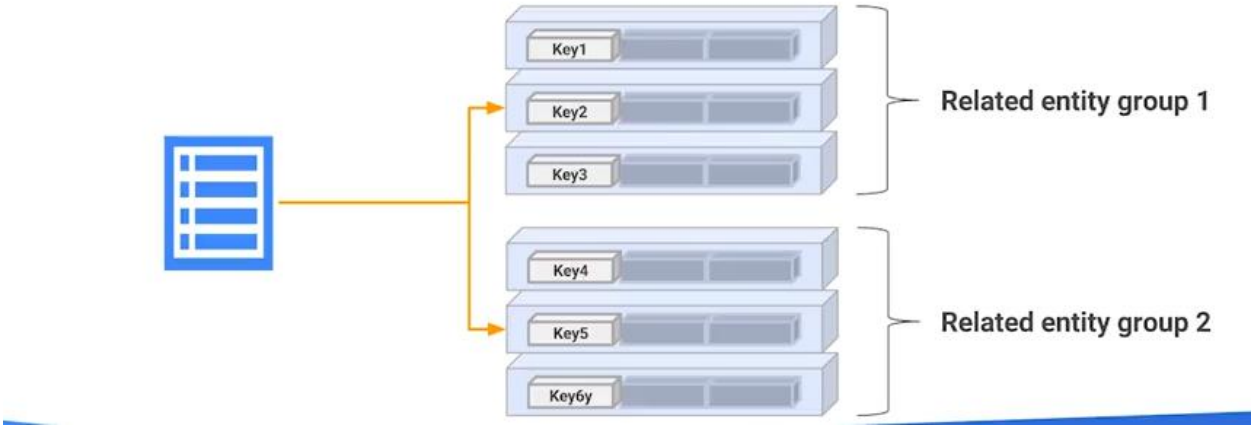
BigQuery rows are sorted and they are stored sorted. Every tablet consists of contiguous rows. Rows are sorted in ascending order by row key. Given this, how can the design of a Bigtable table be optimized?

The basic principle is that when querying the data, queries that use the row key, or a row prefix, or a row range are the most efficient. This makes sense because the only thing that we're ever indexing is the row key. Any query that uses a row key is going to be more efficient because any other query will read the entire table. Use the row key as much as possible, that is the only thing that's indexed. Bigtable will let you use a prefix from the row key such that it is not necessary to use the entire key. This is also efficient because the rows are stored in ascending order.

When designing for Bigtable, what is the most common query that will be made – that will determine which feature should be the row key.

Store Related Entities in Adjacent Rows

- Entities are considered related if users are likely to pull both records in a single query. This makes reads more efficient.



Bigtable's stores data with row keys in sorted order. The objective is to store related entities adjacentl such that a query returns rows that are contiguous to achieve efficient reads (think of reading data off the disk without lifting the head). Contiguous rows are typically in the same tablet. If the results are going to have to get pulled in from multiple tablets, that's going to be a lot slower than if the results come from the same tablet. The only way the results are going to come from the same tablet is to query for records whose keys are contiguous; design keys such that the most common queries return adjacent rows.

Another common use case may be to retrieve latest few records. If retrieving the latest last few records regularly, you may consider adding the time stamp to the key. But you don't want to add the time stamp as is to the key. Why not?

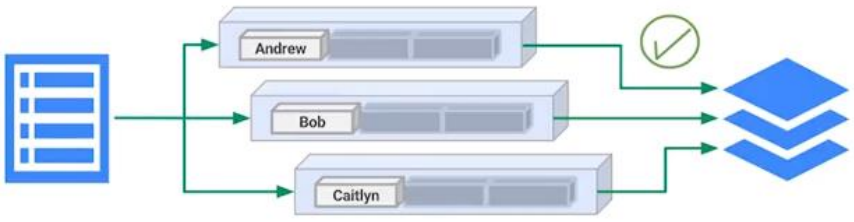
Suppose you add a time stamp to the key where the time stamp is the year, month, and day. If you add that time stamp to the key, what is going to be the first part of the key? What is going to be the first part of the time stamp?

The first part of the time stamp is the year. Earlier years are going to be coming after the later years (sorted by ascending year). The data is in the wrong order if the desire is to grab the latest records. Therefore, rather than storing the data in ascending order of time stamp, store our data (sort the keys) in descending order of time stamp.

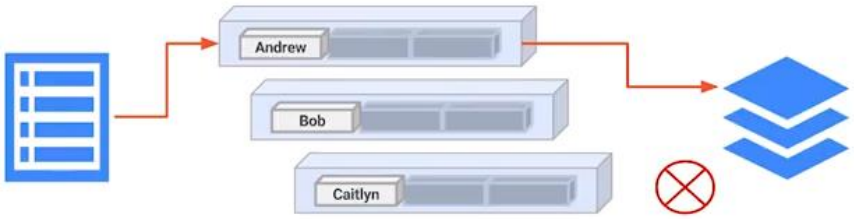
However, Bigtable cannot do that - Bigtable only stores things in ascending order of the row key. What is required in the row key is to use a reverse timestamp so that the latest records are first.

See https://cloud.google.com/bigtable/docs/schema-design-time-series for more on time data design. Also, for an idea of a reverse timestamp, consider Long.MAX_VALUE - System.getCurrentTimeMillis() (http://mail-archives.apache.org/mod_mbox/hbase-user/201108.mbox/%3C4E4FD8E5.4020001@gmail.com%3E).

Distribute writes and reads across rows

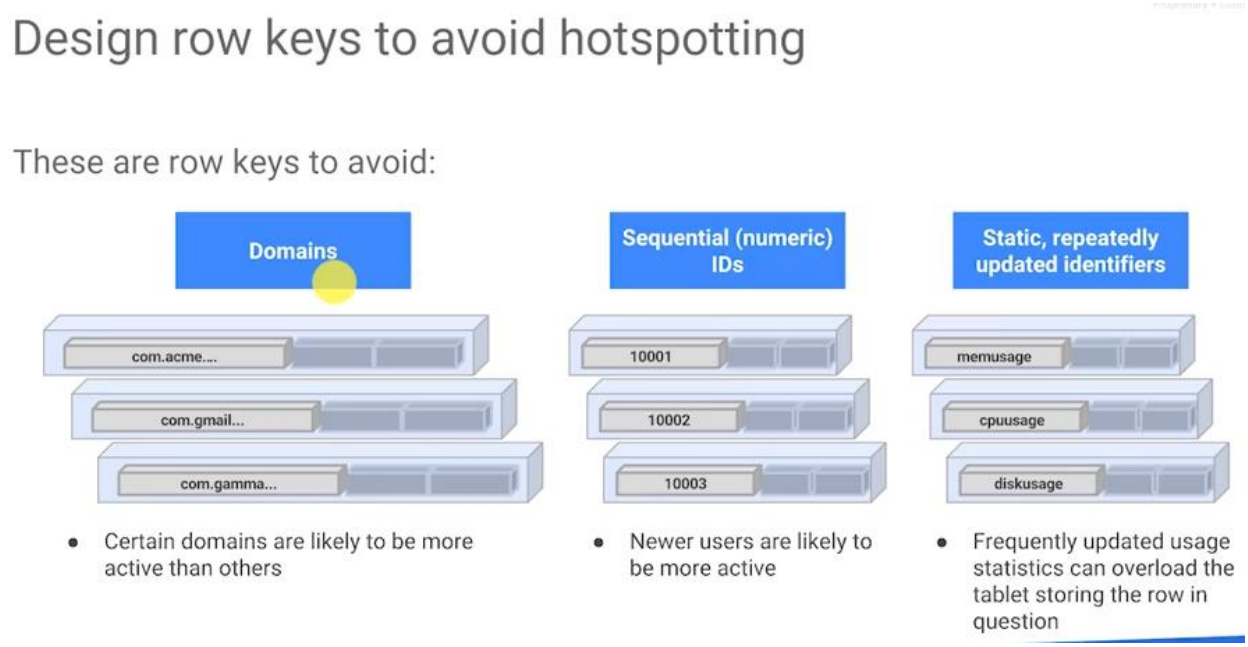


Reads and writes should ideally be **distributed evenly across rows**, so that node workloads are easily balanced.



When read/writes are concentrated in one part of a table, **individual tablets (and their nodes) can be overloaded.**

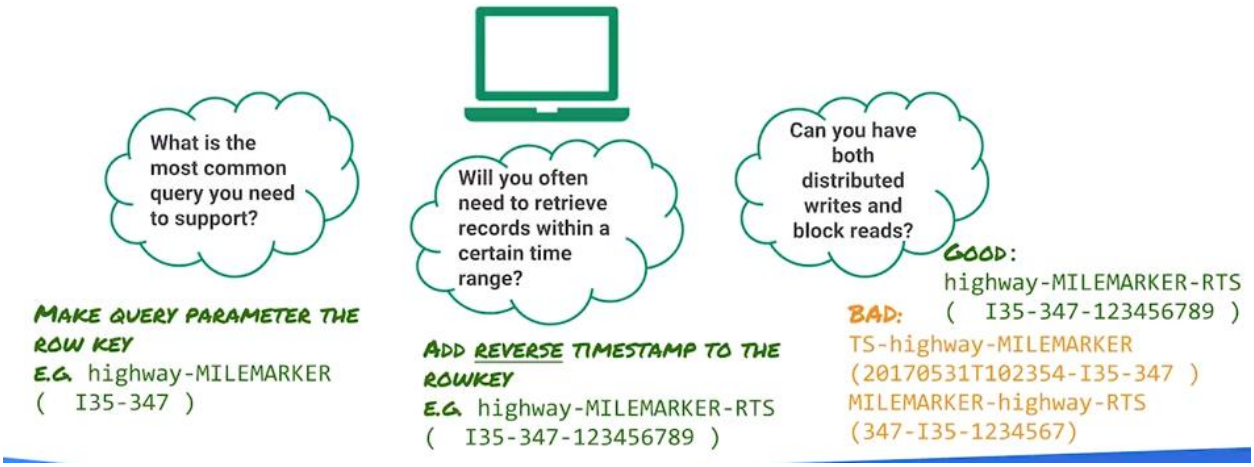
When reading data, it is desirable to read contiguous rows. However, at the same time, recall that Bigtable assets are being used by multiple users over the entire application. It is desirable for reads and writes to be distributed. You don't want all the activity to happen on just one tablet. When the activity occurs in multiple tablets, multiple nodes are carry out the operations. It is desirable for reads and writes to be evenly distributed such that the work load on all of these nodes are balanced. If your reads and writes are concentrated on one set of keys, then the compute occurs in one tablet and one node will perform most of the work which is not a very scalable design.



To avoid hotspotting where a single node performs nearly all compute, consider what kind of row keys might prevent that. One idea is to use the domain of the customers. However, if a small percent of the customers are responsible for a majority of activity, then hotspotting will occur. Domains are okay to use, but don't use the domains to start the row key. Don’t use user IDs to be your row key, especially if user IDs are sequentially assigned. It's okay if your user IDs are randomly assigned, if you're just using a hash code.

Similarly, you don't want to have a static identifier as a key, especially if you have a static identifier that's going to keep getting used. For example, if you have a rogue key that's mem usage or CPU usage or disk usage and you keep updating it over and over again, it could be the case that those tablets, those nodes that are basically doing this processing for this constantly updated data. They're going to basically get overworked.

The key to good row key design is that keys need to be arbitrary and random. See for example this document which discusses good key design: http://archive.cloudera.com/cdh5/cdh/5/hbase-0.98.6-cdh5.3.8/book/rowkey.design.html.



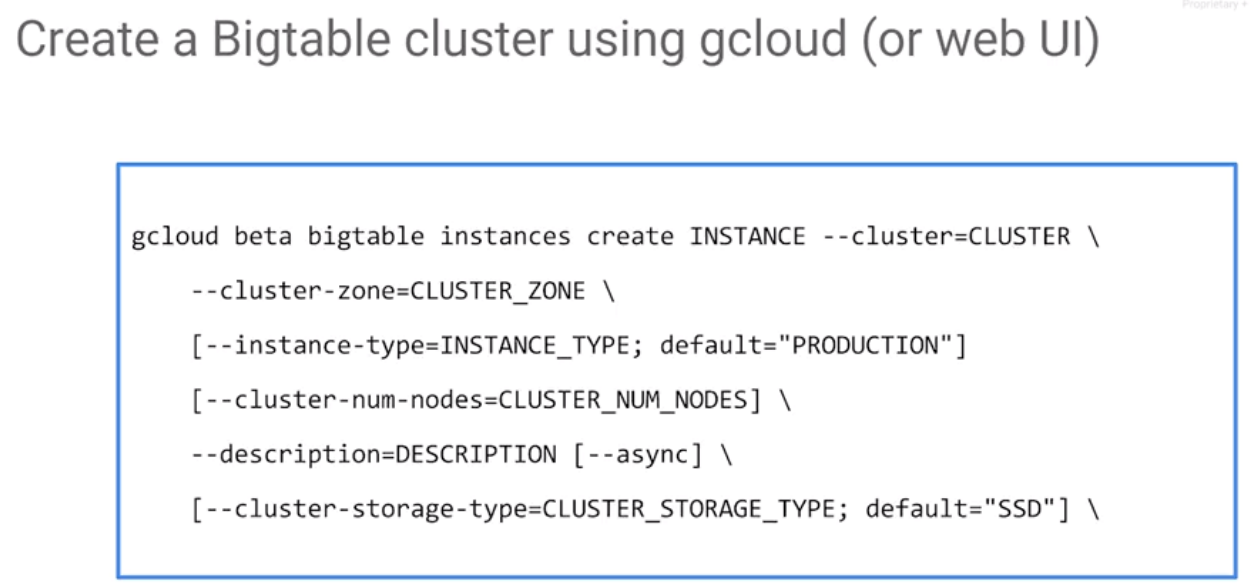
Distribute the writing load across tablets while allowing common queries to return consecutive rows

It is necessary to distribute this writing load such that it is written across one table: do not write one table at a time. At the same time, design row keys such that when queried, rows are returned from contiguous or adjacent rows. Both need to be balanced.

A major issue to watch out for is IDs bunching up, say for example, at the beginning of the trading day, many orders get executed and the key is the time stamp. In this case, there are clusters of trades occurring in just a few tablets which cause major hotspotting.

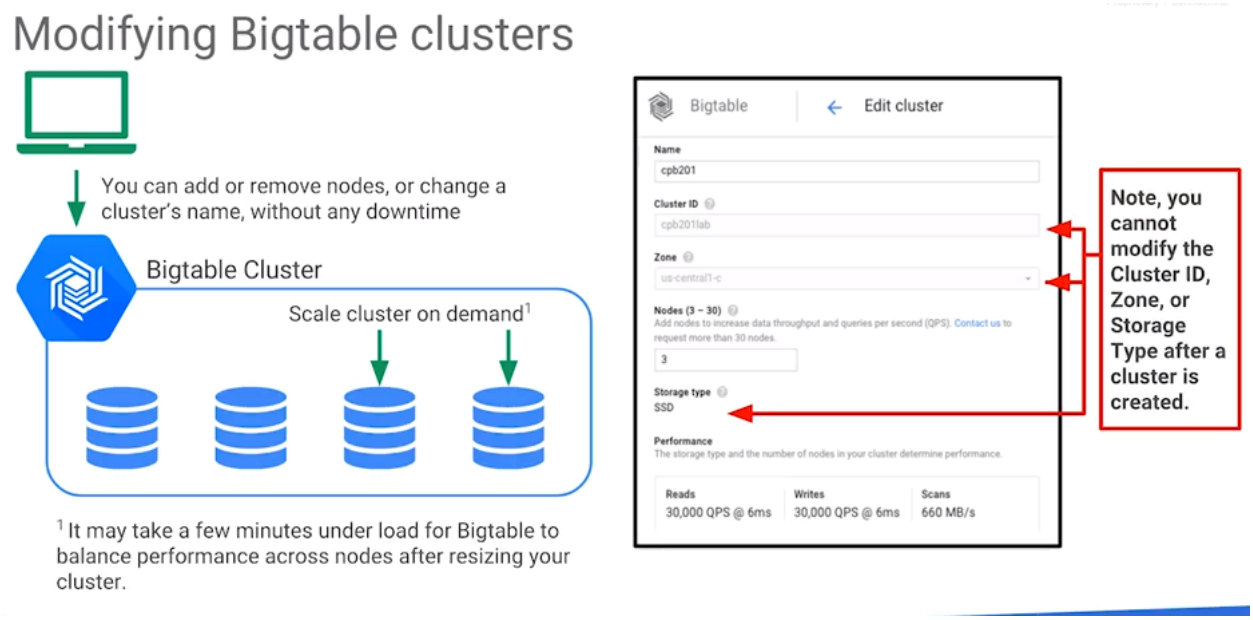
Ingesting into Bigtable

To create a Bigtable cluster, use the Web user interface. You can use gcloud to create the cluster.



Command: gcloud Bigtable instances create, create instance

- Specify the instance type which is a cluster – worker type
- CPU count
- Memory
- cluster storage type.
- Disk type, e.g. SSD disks



Once the cluster is initialized, it is still possible to add or remove nodes without any downtime. This is because the cluster nodes don't hold data but they only the metadata referring to the data. All that happens when you add new clusters is that some metadata gets copied over.

```
from google.cloud import bigtable

client = bigtable.Client(project=project_id, admin=True)
instance = client.instance(instance_id)
```

```
print('Creating the {} table.'.format(table_id))
table = instance.table(table_id)
table.create()
column_family_id = 'cf1'
cf1 = table.column_family(column_family_id)
cf1.create()
```

Once you have your big table instance you can create your table as shown above using the Bigtable client with Python.

Write rows into table

```
column_id = 'greeting'.encode('utf-8')
greetings = ['Hello World!', 'Hello Cloud Bigtable!','Hello Python!',]


for i, value in enumerate(greetings):
    row_key = 'greeting{}'.format(i)
    row = table.row(row_key)
    row.set_cell(
        column_family_id,
        column_id,
        value.encode('utf-8'))
    row.commit()
```

More commonly however, rather than using Python to create the table, use Dataflow. If using Dataflow, then use BigtableIO to commit data to it.

STEPS TO WRITE DATA TO BIGTABLE

1. Create table

```
BigtableOptions.Builder optionsBuilder = new BigtableOptions.Builder()
            .setProjectId(options.getProject())
            .setInstanceId(INSTANCE_ID).setUserAgent("my-program-name");
BigtableSession session = new BigtableSession(
   optionsBuilder.setCredentialOptions(                    GET AUTHENTICATED SESSION
   CredentialOptions.credential(
   options.as(GcpOptions.class).getGcpCredential())).build()));
BigtableTableAdminClient tableAdminClient = session.getTableAdminClient();

CreateTableRequest.Builder createTableRequestBuilder = //       CREATE TABLE
            CreateTableRequest.newBuilder().setParent(getInstanceName(options)) //
            .setTableId(TABLE_ID).setTable(tableBuilder.build());
tableAdminClient.createTable(createTableRequestBuilder.build());
```

2. Convert object to Mutation(s) inside a ParDo

```
LaneInfo info = c.element();
DateTime ts = fmt.parseDateTime(info.getTimestamp().replace('T', ' '));
// key is HIGHWAY#DIR#LANE#SENSORID#REVTS
String key =  info.getHighway() + "#" + info.getDirection() //
            + "#" + info.getLane() + "#" + info.getSensorKey() //
            + "#" + (Long.MAX_VALUE - ts.getMillis()); // reverse time stamp
// all the data is in a wide column table with only one column family
List<Mutation> mutations = new ArrayList<>();
mutations.add(Mutation.newBuilder().setSetCell(Mutation.SetCell.newBuilder()
                      .setValue(Double.toString(info.getLatitude()))//
                      .setFamilyName(CF_FAMILY).setColumnQualifier("latitude")//
                      .setTimestampMicros(ts)).build())
// other columns
c.output(KV.of(ByteString.copyFromUtf8(key), mutations));
```

3. Write mutations to Bigtable

```
mutations.apply("write:cbt",
        BigtableIO.write().withBigtableOptions(
        optionsBuilder.build()).withTableId(TABLE_ID))
```

To read from Bigtable, again, you can do it very programmatically using the HBase API, you can use the Hbase client. You can even use Bigquery to query off Bigtable.

Reading from Bigtable

Typically programmatic (using HBase API)

HBase command-line client

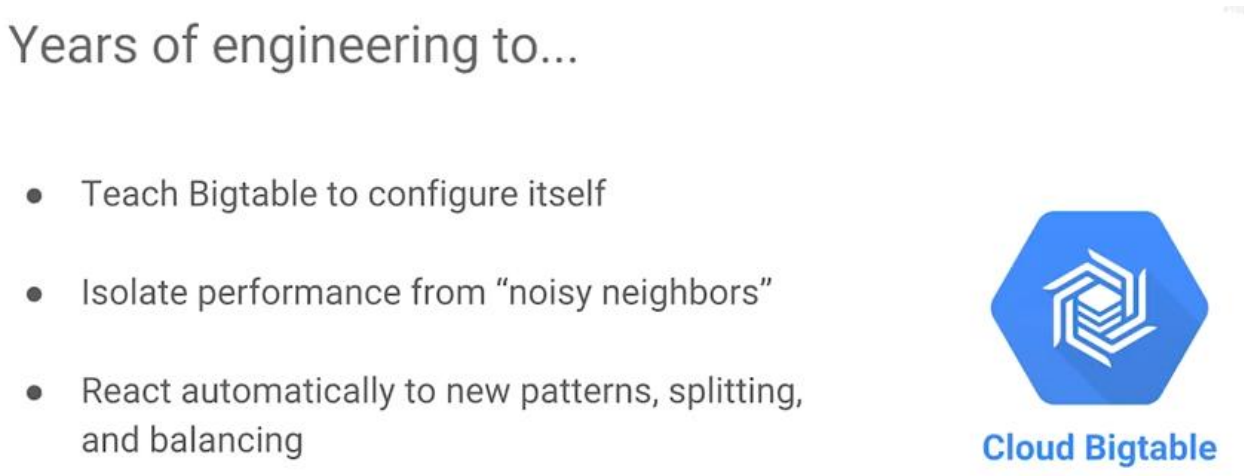
Or even BigQuery
https://cloud.google.com/bigquery/external-data-bigtable

Lab: Streaming into Bigtable at low latency

https://codelabs.developers.google.com/codelabs/cpb104-bigtable-cbt/

Video review: https://www.coursera.org/learn/building-resilient-streaming-systems-gcp/lecture/wlLP1/lab-review

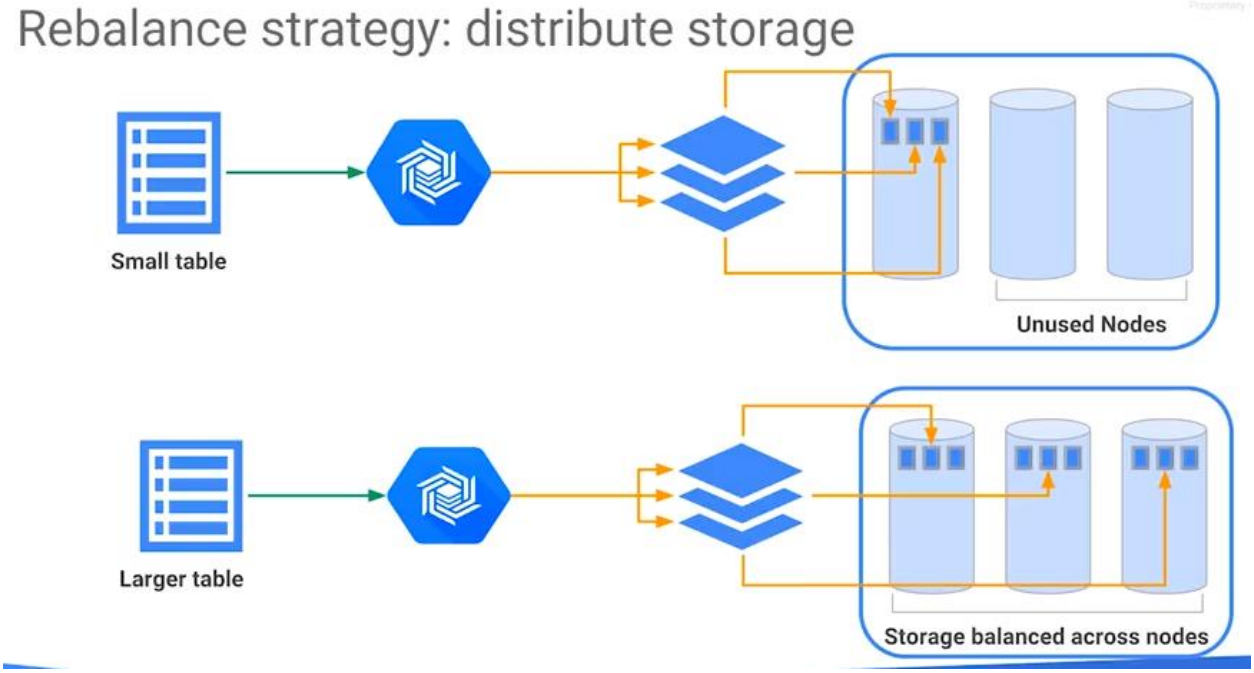
Performance Considerations



What are the performance considerations when using Bigtable? Bigtable essentially configures itself. There's been a lot of engineering effort on Bigtable at Google so that it can react automatically to the way in which you are using it. The key thing is that Bigtable learns over time as it is up and running in a typical usage scenario. The best way to figure out how Bigtable is going to perform is to have it work on the load that you're performing because the way it performs in the first few seconds is not the way that it's going to keep performing.

Bigtable considers access patterns, figures out how many nodes it needs to keep running, figures out where the tablets need to be, how the tablets need to be split up, which machines need to process what data. From this usage data, Bigtable optimizes its configuration. We discussed the design of the row key in such a way as to distribute reads and writes which is important and somewhat independent of optimization self-configuration. However, because of that optimization row keys do not need to have a perfect design, Bigtable self-configuration will take care of hotspots to some extent. The way that Bigtable addresses hotspotting is to first implement a rebalance strategy is that it will redistribute the reads. If it finds that more than 25 percent of the reads are happening on a single overloaded node, Bigtable will attempt to redistribute reads over the other nodes.

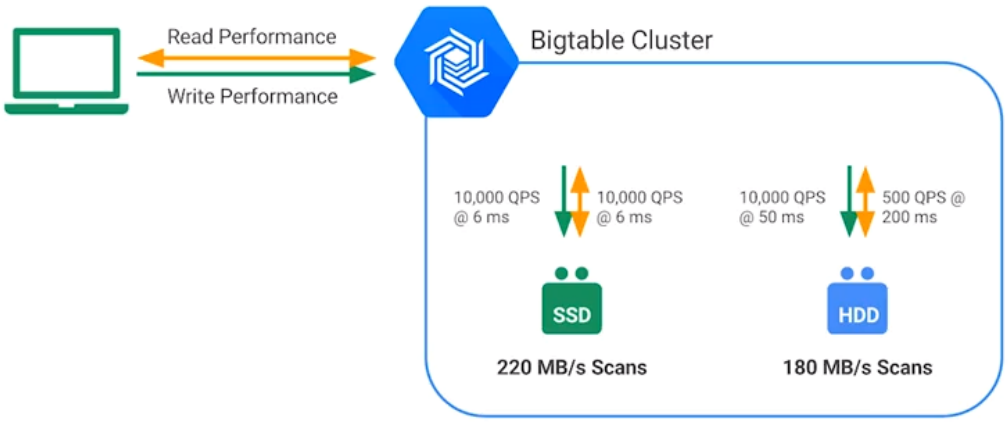
Rebalance Strategy: Distribute Storage



Part of the rebalancing strategy of Bigtable is that the storage itself will also get redistributed. Recall that cluster nodes do not store data, they only store the metadata. The data itself is still stored in cloud storage. Redistribution of the data in cloud storage is not as expensive as you would think. Bigtable is not really copying data around, it's only copying pointers to the data.

Cluster performance

- Under typical workloads, Cloud Bigtable delivers highly predictable performance. When everything is running smoothly, you can expect to achieve the following performance for each node in your Cloud Bigtable cluster, depending on which type of storage your cluster uses.



Note: A *query* is a read or write operation against a single row.

Under typical workloads, Bigtable gives very predictable, consistent performance. For example, you can expect about 10,000 queries per second at about a 6 millisecond latency (read and write) with SSD discs. With HDD, the performance is 500 queries per second at 200 millisecond latency.

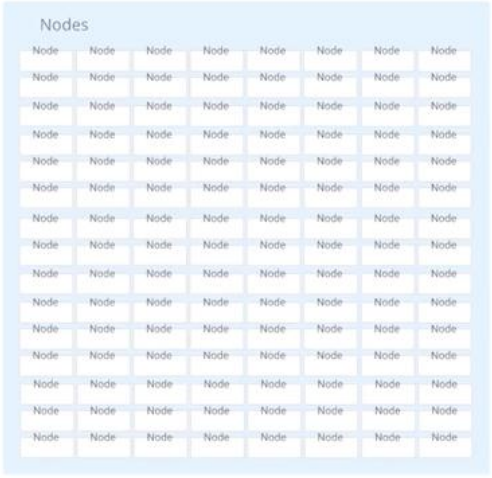
Tips for improving performance

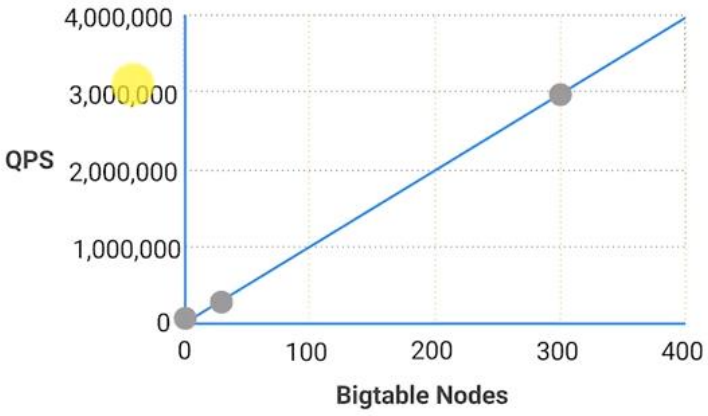
Change schema to minimize data skew	Test with > 300 GB and for minutes-to-hours	Performance increases linearly with number of nodes
Takes a while after scaling up nodes for performance improvement to be seen	Disk speed: SSD faster than HDD	Make sure clients & Bigtable are in same zone

How do you improve the performance of Bigtable?

- Consider the schema. Design the schema such that reads and writes are as distributed as much as possible. Such that there's not much of a data skew.
- Realize that Bigtable learns over time. Don't keep changing your schema or data. Let Bigtable run for a while before you start making changes. At least a couple of hours.
- When testing Bigtable, make sure to test with at least 300 gigs of data.
- SSDs are faster than HDDs.
- The number of nodes is linearly related to performance.
- Ensure that your clients and your Bigtable are in same zone.

Throughput can be controlled by node count





LINKS FOR PREPPING AND STUDYING FOR THE GCP DATA ENGINEER EXAM

Data Engineering/Data Scientist GCP Certification

See https://cloud.google.com/certification/data-engineer. There are different tracks, all of which appear to be covered by the five Coursera course in this specialization. Additionally:

Use this guide: https://cloud.google.com/training/data-ml

Do this quest: https://google.qwiklabs.com/quests/25?utm_source=gcp&utm_medium=website

Consider the case studies: https://cloud.google.com/certification/guides/data-engineer/#sample-case-study

And take the practice exam: https://cloud.google.com/certification/practice-exam/data-engineer.

Condider also:

Guide to the Architecture Exam - https://cloud.google.com/certification/guides/cloud-architect/.