SERVERLESS DATA ANALYSIS WITH GOOGLE BIGQUERY AND CLOUD DATAFLOW

Introduction

In this first course of the data engineering track, you will gain an overview of the data and machine learning parts of Google Cloud platform, and not just a cursory overview. In each module, we first explore products that will help you accomplish certain things on Google Cloud, and second, we look at very common and specific use cases that involve machine learning, data processing and data analysis. By the time we are done with this course, you will have a good overview of all the moving parts of the data and Machine Learning parts of the platform.

Who is a Data Engineer?

A data engineer is someone who enables decision making. Typically, they do this by building data pipelines which ingesting, process data, and analyze data, building dashboards, and building machine learning models. The data engineer's job is to enable decision-making within the company in a very systematic way. In order to be a good data engineer, you needed to know both programming and statistics to a great deal of depth. With the advent of the fully managed, auto-scaling services on Google Cloud, the amount of infrastructure and programming that you need to know has become simpler. At the same time, the statistics realm has also gotten a lot simpler. You now have libraries that take care of a lot of the low-level programs that you once had to write. Also reduced are the mathematical concepts that you had to know; to the extent that you can now program with data. It is much simpler to build statistical machine learning models when utilize existing libraries and packages.

Course Overview and Agenda

Course overview

Learn how to carry out no-ops data warehousing, analysis and pipeline processing

Google Authorized Trainer

BigQuery

Cloud Dataflow

Data analysts, business analysts, data scientists, data engineers with experience on:

Google Cloud Platform Big Data & Machine Learning Fundamentals

Using a SQL-like query language to analyze data

Either Python or Java

Welcome to our class on Serverless Data Analysis. In this course, you will learn how to carry out no-ops data warehousing using Bigquery and pipeline processing using Cloud Dataflow. Google Bigquery is a data warehouse that you interact with primarily through SQL and Cloud Dataflow is a data processing pipeline system that you can program against in either Python or Java. This class is meant for people who build data pipelines and data analytics. It is prerequisite to have knowledge of SQL because you're going to be interacting with Bigquery and you need to know either Python or Java because we're going to be looking at Dataflow.

Agenda

Module	Description	Topics	Labs
1. BigQuery	No-ops data warehousing and analytics	• Queries • Functions • Load/Export • Nested, repeated fields • Windows • UDFs	• Queries and Functions • Loading and exporting data • Demos
2. Cloud Dataflow	No-ops data pipelines for reliable, scalable data processing	• Pipeline concepts • MapReduce • Side inputs • Streaming	• Simple pipeline • MapReduce • Side inputs • Demos
3. Summary	Course summary	• Resources	

We first look at Bigquery which is the no-ops. No-ops in this context essentially means that there is no infrastructure for you to manage, i.e., no operations. Bigquery is a no-ops data warehousing and analytics system. It is a place that you can store your data, analyze the data, and export your data from. We will look at how to do queries, functions, how to load and export data, and we'll also look at some advanced features like nested and repeated fields, how to do window functions and how to do user defined functions. Along the way we'll do a few labs on queries and functions, on loading and exporting data, and on demos.

Then, we'll move on to talking about Cloud Dataflow which is also no-ops in the sense that you don't need to manage any infrastructure. However, we will write programs that process data, the kind of processing that can operate on batch data or streaming data. In Dataflow, you get to write the exact same code. It works on both batch and stream. If you need reliable scalable data processing on GCP, the best option is Cloud Dataflow. We will define a data pipeline. We will discuss MapReduce which is the most common type of big data algorithms. We look at how you would do those kinds of algorithms in Dataflow. With that, we get the benefit or working with real time data and historical data in exactly the same way.

We will look at what are called side inputs. For the most part, when you're writing a data pipeline, you'll have this one massive data set that you're processing as it comes in and you're writing things out. But usually, along with that big data set, you will have other smaller data that you need to get and join with. Those smaller data sets are called side inputs. We will discuss how to manage with side inputs in Dataflow.

We discuss streaming because one of the key advantages of Dataflow is the idea that you can process batch data and streaming data the same way. Having looked at examples of used cases that used batch data, we will then look at how to do streaming data and Dataflow. The code itself would be very similar to the way you deal with batch, with a few extra concepts that come along because you're dealing with but infinite data, unbounded data.

In terms of labs, we will look at how to create a simple pipeline, how to write MapReduce jobs, how to incorporate side inputs and along the way, we'll also do some demos. Our goal here is to discuss how to handle very large data sets, both from a declarative way with Bigquery and the programmatic way with Dataflow, such that you don't need to spin up any servers or manage any infrastructure. Your code can be SQL statements or Python or Java programs. This code logic can be executed on the cloud against massive data set or a small data set against data sets without having to manage infrastructure. It can be extremely liberating as a data scientist and as a data engineer if you can get to focus on your data. These two key products of GCP let you focus on your core logic and leave the infrastructure management to the cloud provider: Bigquery and Dataflow.

MODULE 1: SERVERLESS DATA ANALYSIS WITH BIGQUERY

Serverless Data Analysis with BigQuery Introduction

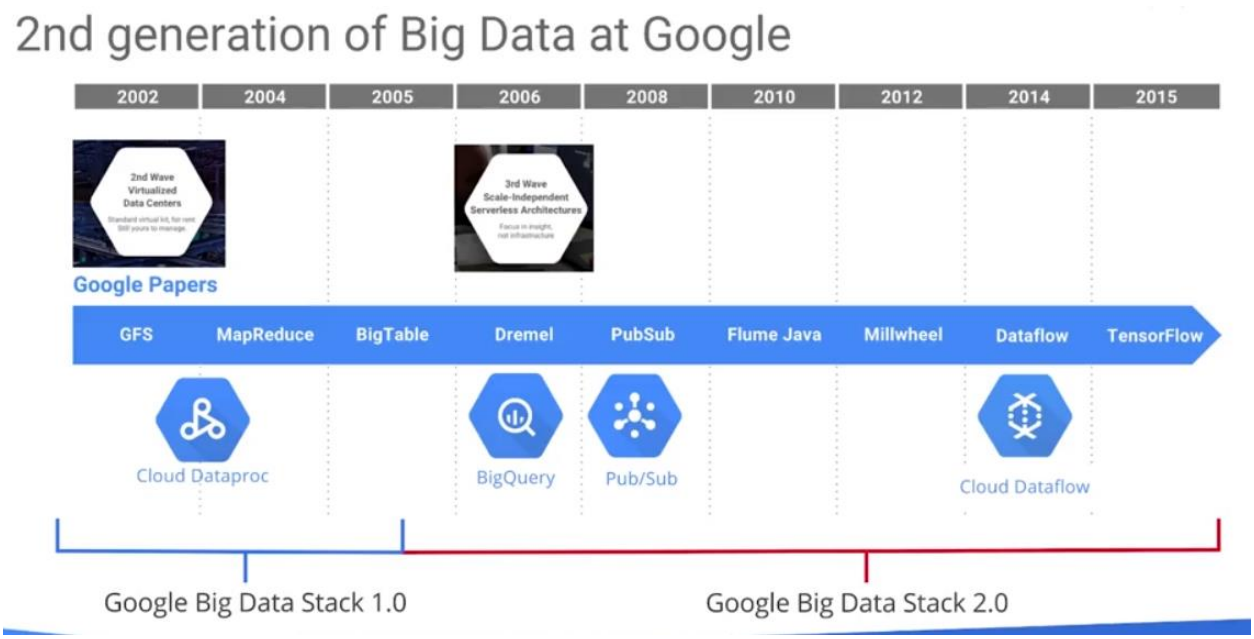
In this module, we look at BigQuery, how to write queries and functions in BigQuery, how to load data into it, and how to export data from it. We also look at advanced capabilities, performance tuning, and pricing.

All along the way, we do labs where labs you write functions, export data, load data.

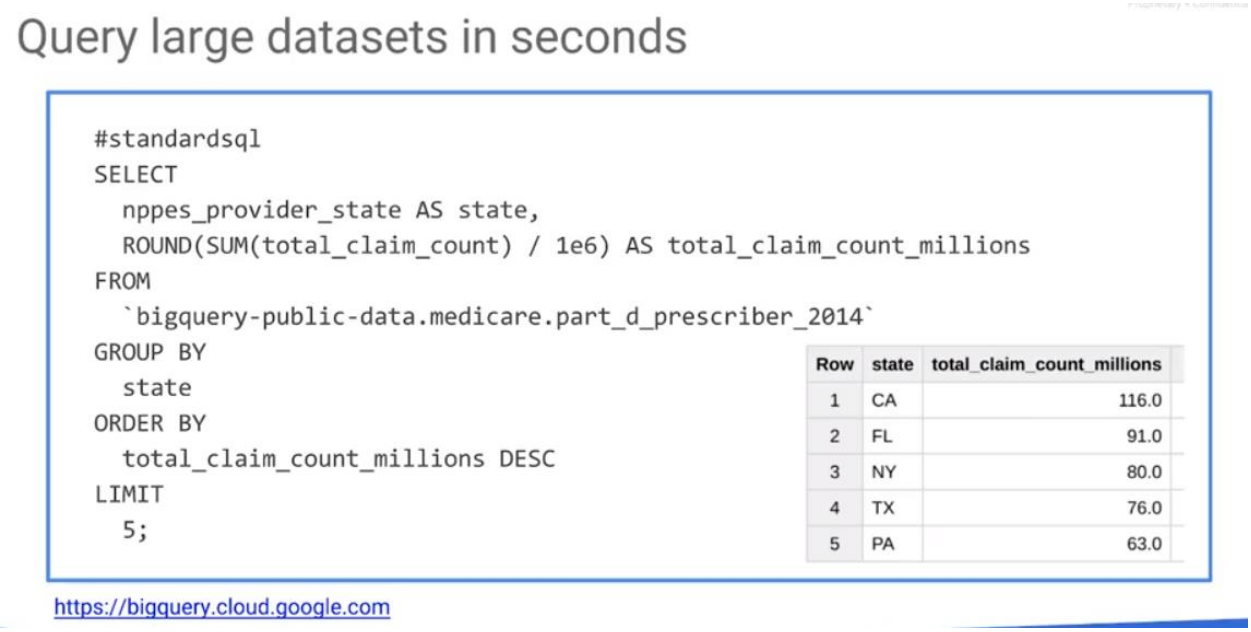
What is BigQuery?

Let's begin by looking at serverless data analysis using structured query language, or SQL. BigQuery is the data warehouse that petabyte scale data warehouse on Google Cloud. We discuss how to write queries and functions in BigQuery. Do a lab that takes what we've learned and demonstrates it in the form of sample code. Then we look at how to load data into BigQuery, export data from BigQuery, from tables in BigQuery into other formats such as a CSV file. We also explore advanced capabilities that BigQuery allows beyond straight forward structured query language including nesting, repeated fields, etc. Then we finish off with looking at performance and pricing.

Then, we apply what we learned in a lab. A word of caution, all of these labs are going to be very structured, stepping through writing a query, loading data, exporting data, etc. After doing the lab, then try what you learn on your own dataset. The labs here are primarily about showing you how it's done, but you will actually learn a lot more if you try to do it by yourself. There are a variety of public data sets on BigQueries, I'll show you some of those as we go along. And then you can go ahead and try doing the exact same things that we're doing, but on that other query dataset.



BigQuery is part of the second generation of Big Data at Google. Big Data at Google started with the Google file system, GFS, and MapReduce. MapReduce is basically the idea that if you want to process very large datasets, you can't do them on one machine - you shard the datasets, that is, you split them up into small pieces and you put those small pieces on a distributed cluster of machines. Then for any kind of analysis on that data, each machine that has a shard of the complete dataset carries out subset of the operations. You parallelize what are called the map operations, so that each map operation on each compute node only processes its shard of data. Then you take the results from all of those map operations and you combine them in some way on a different set of machines, compute nodes that are called reduce operations. These reduce operations aggregate the results out of all of the map nodes, and then that's essentially your result. With MapReduce, you first have to shard the data - split it and store it on multiple machines. That doesn't scale very well because now you've mixed up storage and compute. The number of machines that you need to do your compute is going drive how to shard the data. That is not optimal: every time you need to do a compute job, you first need to determine where the data is stored. That is what led to the second generation of Big Data at Google, starting with things like Dremel, things like PubSub. These are ways in which you can autoscale and carry out your operations and the system infrastructure determines out how many machines are required to perform the operations. When you look at things like Dremel and PubSub and Flume, you have completely serverless, autoscaled ways of performing data analysis.

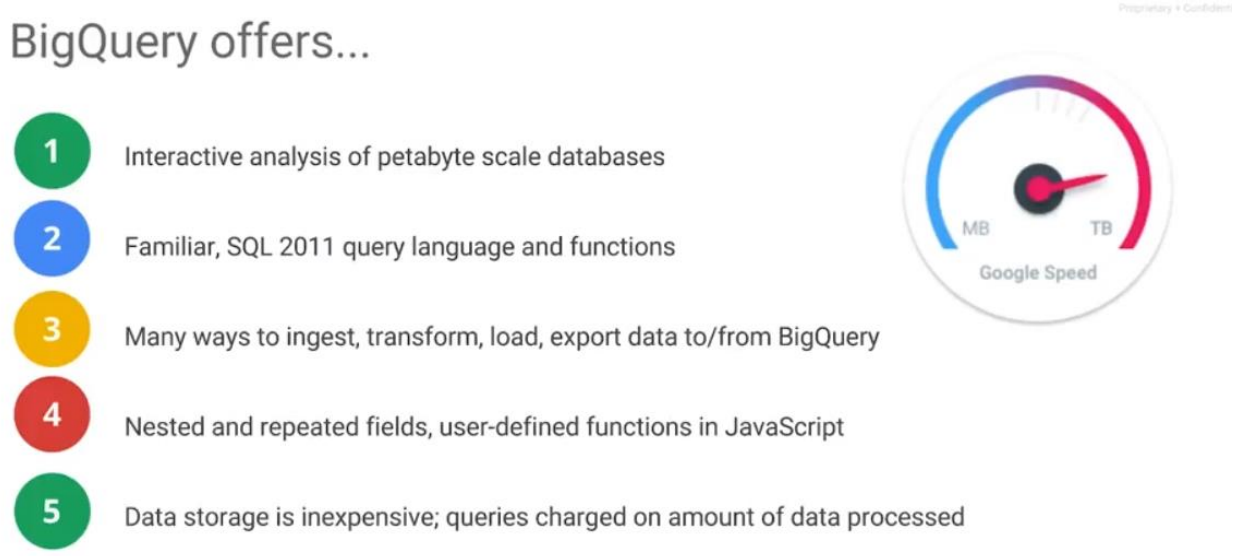


Dremel which was used within Google – it is basically what on Google Cloud is available as BigQuery. The same engineering team that builds Dremel which is an internal product also builds BigQuery, the external product – they are the same product. BigQuery allows you to query very large datasets in seconds. How large? Well, the dataset could be as big as a few petabytes. Data are stored on the Google Cloud. Whenever we want to do an ad hoc query, just write the query and run it. We don't need to provision clusters. We don't need to create indexes. We don't need to do anything other than have the data on Google Cloud in a

denormalized form in the form of BigQuery. Although the data is said to be in denormalized form, it can be accessed as regular tables. It is much more effective and efficient if the data is denormalized. We can access BigQuery through the Web console, a typical console, or at console.cloud.google.com.

BigQuery queries are standard SQL or legacy SQL". At this point, we can query the dataset. For example, a Medicare dataset across every state. For example, group by State and find the millions of claims across each state. The operation takes seconds and generates a quarter of a gig of data. There is no indexing in this query. You pay for the amount of data that you process in BigQuery which is based on the number of columns that processed.

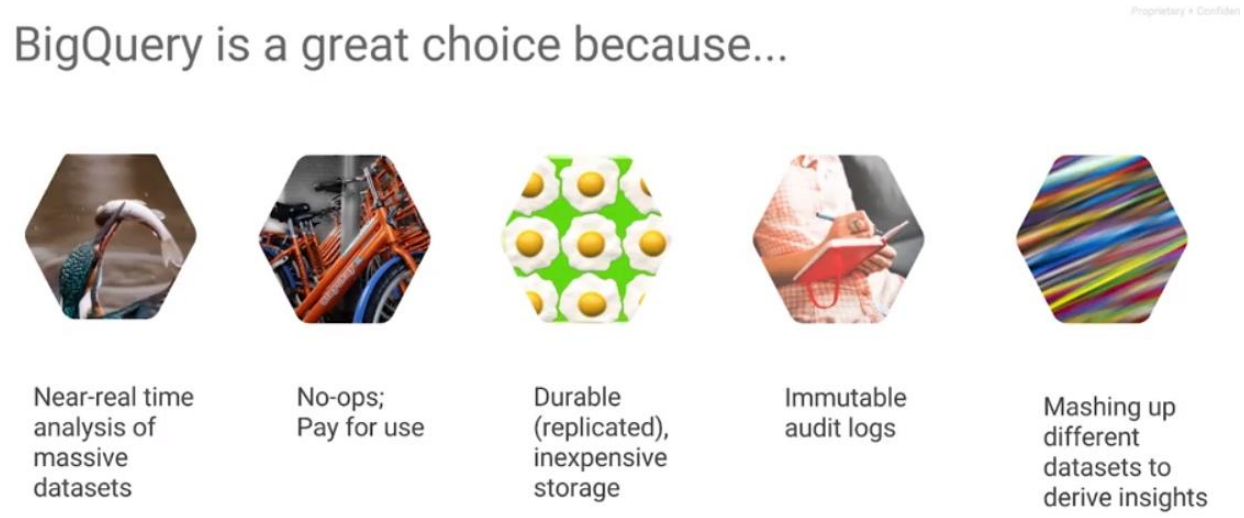
Evaluating BigQuery



BigQuery offers us interactive analysis. It is possible to run SQL queries in BigQuery without any preparation of any sort. It gives you an ability to do interactive ad hoc analysis of databases that can go up to petabytes of data. The BigQuery language itself is standard SQL, SQL 2011. It supports a variety of SQL functions as well, so it's very familiar. If you know SQL, you know how to use BigQuery. To get your data into BigQuery, you have a variety of ways to ingest data, and ETL (transform, load, and export) data to and from BigQuery. There are a lot of tools out there that have connectors into BigQuery. Whether you're using Tableau or using Looker or you using Qlik or using Data Studio, they all have connectors into BigQuery. And more and more partners are creating connectors to BigQuery. BigQuery supports nested and repeated fields. BigQuery can ingest JSON data: because JSON is a hierarchical data format, many of the logs may come in as JSON and then can be queries using an SQL-like language. It is not standard SQL at that point because there are some extensions to help you deal with nested and repeated fields.

BigQuery separates out storage and compute. You pay separately for storage for the data entered into BigQuery. The cost of storage in BigQuery is about the same as the cost of storage in Google Cloud Storage.

You can store your data, if it's structured or tabular data in BigQuery and you will get the same sort of discounts that you get in Cloud Storage. Iif you have some data and you haven't edited it in a few weeks, you start getting automatically the discounted rates for older data. Storing data in BigQuery will get the discounts of Nearline.



BigQuery is a great choice because it allows you near real time analysis. We were able to take that Medicare dataset and were able to do our analysis and we got our results back in seconds. It is not really real time, so if you want a transactional database that gives you millisecond, microsecond responses, BigQuery's not the answer for that. The answer for that would be something like Cloud SQL, or something like Spanner. But for ad hoc analysis of very large datasets, for data warehousing, for business intelligence, and similar kinds of operations, BigQuery is a great choice. BigQuery is completely no-ops - ingest data onto BigQuery and you are not managing any clusters beyond that point. When you query on BigQuery, run the query on BigQuery: there is no need to create a cluster ahead of time. You only pay when you query the data. Separately, you do pay for the storage but as we all know, storage is cheap. What you pay for are the queries that you run. It is possible to get a flat rate pricing as well. You have two forms of pricing in BigQuery, you have the pay for use and you have the flat rate pricing. However, it is recommended to try out the pay for use because it helps you deal with these very rarely used datasets business intelligence sums. You can set billing limits and if you find that your bills are pretty high, you can talk to Google’s sales representatives and figure out a way to get on to the flat rate plan. By default, the best plan for most people tends to be the pay per use.

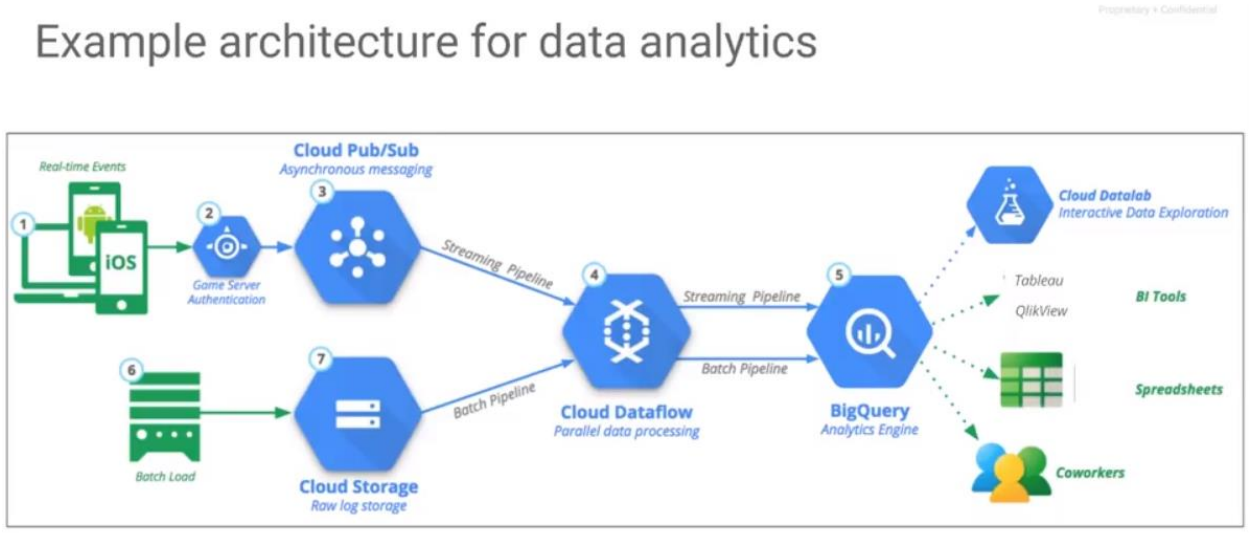
BigQuery, just like cloud storage, is durable. If you put your data on BigQuery, it's replicated in multiple places and it's pretty inexpensive, in terms of storage. BigQuery gives you immutable audit logs. Once your data is in BigQuery, you know every time somebody runs a query on it and you basically get those audit logs. Those audit logs can't be tampered with, so you know whenever someone has actually used this dataset or they've edited this dataset in any way.

The key thing which is difficult to explain is that BigQuery is a way to share data beyond the silos of your company's structure.

What is meant by that? BigQuery is clusterless, once data is in BigQuery and you can do ad hoc queries on it, you can take your BigQuery dataset and you can share it with somebody else in your company. Or if you want, you can share it with anybody at your company's domain.

What this means is that when someone wants to query on your data, they do not need to ask for permission. They can just make an ad hoc query on it because you have given them access. Of course, you can control access if you need to. But the point being that anybody that you have allowed to access your table, is able to access your table without any of the ops that you're typically required to do. They don't need to think about how do I log into this person's cluster so I can use their data? Well, it's completely cluster free, right? What this means is that they can take their data, combine it with your data, etc. What happens after a while is that BigQuery becomes this forum in which you get data collaboration across your company, and not just data, but also the techniques such as how to get at the data. For example, you can share links for data requests (queries) to BigQuery. That is, it allows sharing your analysis as well. BigQuery creates a path that is hard to explain. After experiencing this daily for a while for example, it becomes commonplace. For example, some 70 to 80% of all Googlers use BigQuery at least once a week. This is across Google, so it's not just engineers, it's not just data analysts, it's everybody.

Architecting a BigQuery Project



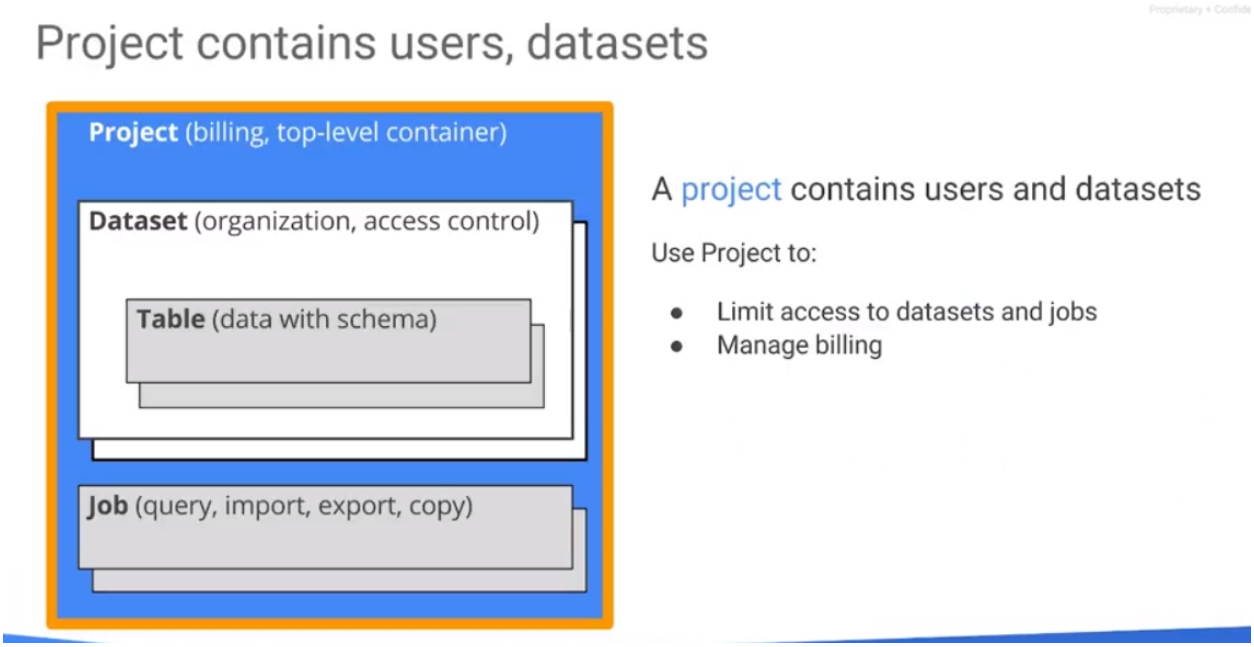
Build a mobile gaming analytics platform - a reference architecture

BigQuery is a huge part of all the kinds of data analysis performed on Google Cloud. Here, for example is a typical architecture that you might do for Data Analytics. In this case, this is a gaming architecture, but this is a very common theme, a pattern, if you will for data analytics on to the cloud. In this case, with games, the idea is that basically every time somebody plays a game on a mobile device, real time events gets created, this person has crossed this level, another person has clicked on this thing, a person has made in app up-

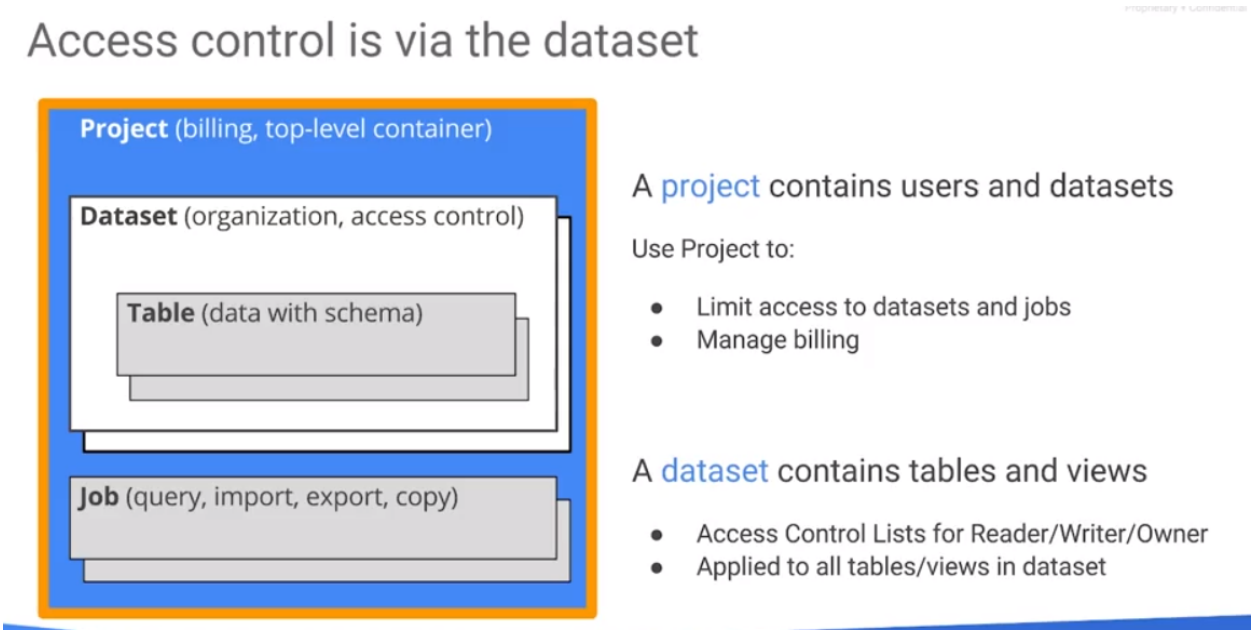
purchase, another has called this amount that this person has conquered this whatever, etc. Each of those events then gets sent to a gate server, which is running on app engine in this case, and app engine is basically taking those events and basically putting them into a topic on Cloud Pub/Sub (Pub/Sub is an autoscaling message queue). Then the messages and pub/sub are consumed in real time by DataFlow which is an autoscaling data processing pipeline which processes these messages. If there are more messages, there are more machines to process them. If there are fewer messages, there are fewer machines to process them. DataFlow can create aggregates that track customer behavior over some period of time. It is condensing the data that's being created from these mobile apps, perhaps by user, time slot, region, domain, or whatever it aggregates on, and writes the aggregate into Big Query.

Now you can perform operations on these Big Query aggregates. How do you know what kinds of things to do with them? When you build tools is you have to go look at historical data. Traditionally, when we handle historical data, the data pipeline for historical data is very different from the data pipeline to process real time data. But that is not so in Pub/Sub, because cloud data flow lets you process both real time data and batch data with the same code. Historical Data may be loaded from Cloud storage into Dataflow, and a batch pipeline that is probably running once a day, for example, which does essentially the same kinds of things and is writing them into BigQuery, creating a data warehouse. So regardless of whether the data comes in through a streaming pipeline in real time, or a batch pipeline based on historical data, it's all in BigQuery and BigQuery, as we talked about, is connected to a variety of business intelligence tools. You can analyze it, if you're a data scientist, you can analyze it using DataLab. If you are a business person, you want business types of charts, you can analyze them with tools like Tableau and Q-Liv. You can take data from BigQuery and use it to populate a Google sheet. As we talked about, you can take a BigQuery table, your data set, your query, and you can send links to those to your coworkers. That is basically the way you tend to do analytics and in Google Cloud and BigQuery is a huge part of it.

The way you work with anything on Google Cloud including BigQuery is from a project. A project, of course, is wh

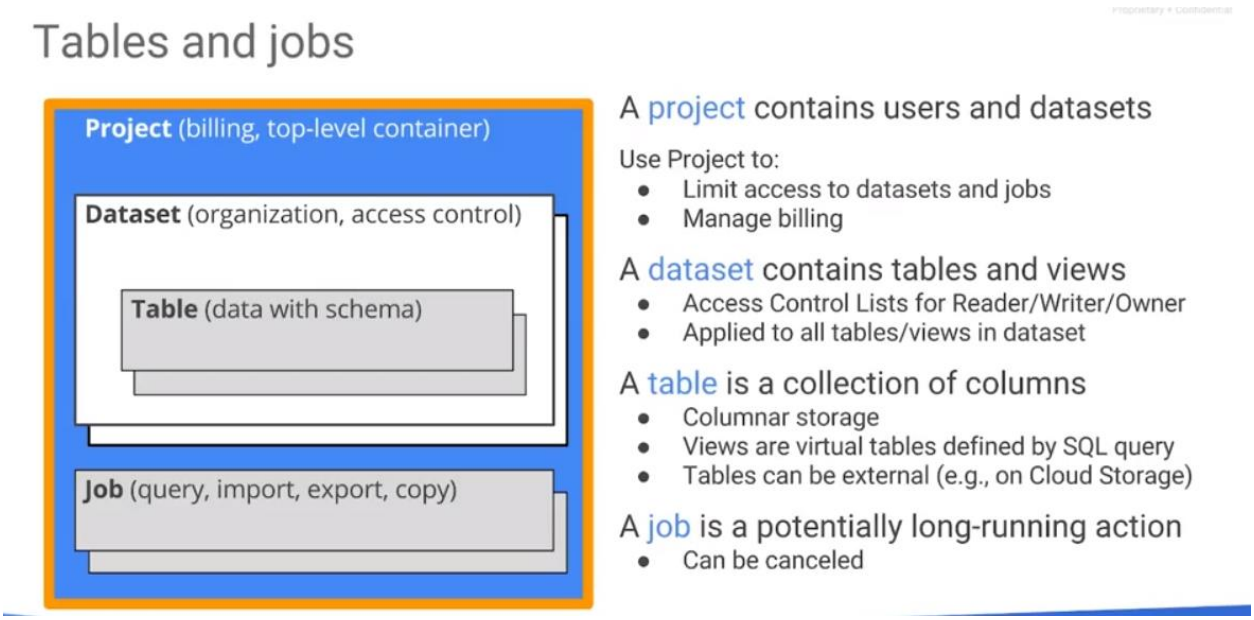


at sets billing. It knows where to send the check to when you run a query on BigQuery. When you load the dataset into BigQuery, the storage bill is assigned to the project that you've created the dataset. The project provides billing information. It contains users – there are people who belong to a project, they have owner rights to anything that's created in the project. Typically, you don't want to have one owner, you will have a few owners so that if that a person leaves, other people can use that project.

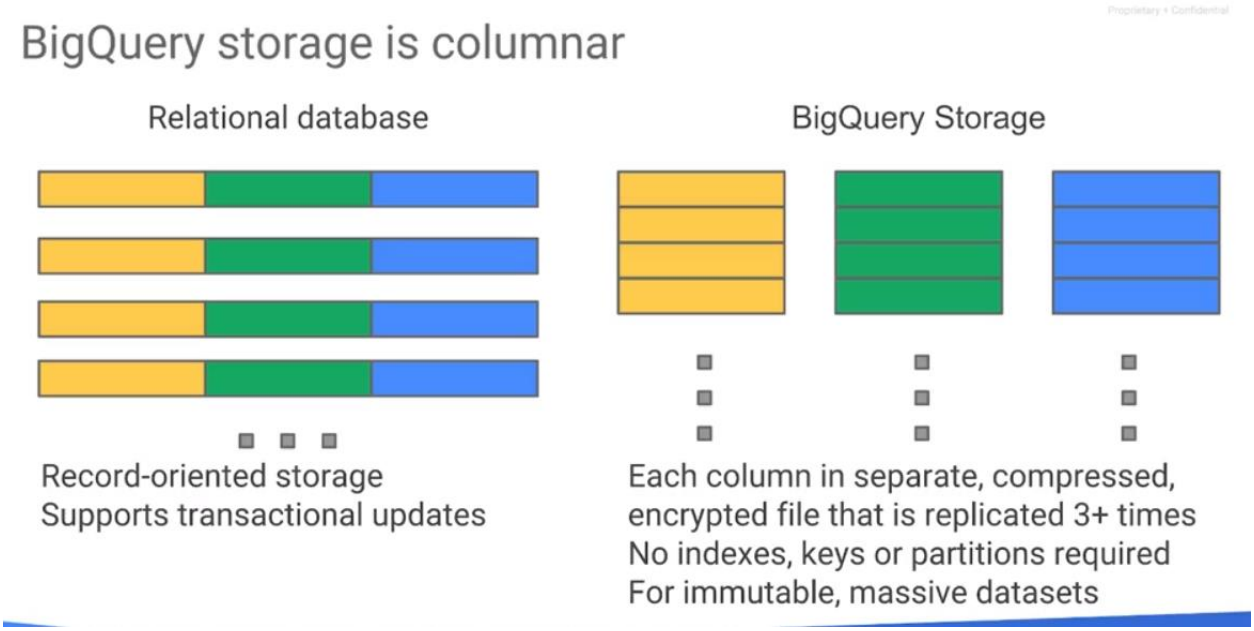


In the project, you create a dataset. A dataset is basically a collection of tables, typically belonging to an organization. Access control is on a dataset basis, not on a table basis, because you want to join tables together which are typically within a dataset. If you don't have access control on a single table, you have access control on a dataset that consists of multiple tables. Whenever you want to do a query, that's a job,

and the query's imports, exports, copies are all jobs that actually interact with this dataset in some way. Therefore, the access control is through this data set.



A data set contains tables and will also contains views. A view is a live view of a table. So essentially you can think of a view as a query. And that query returns a result set, then you can write a query to process that view, just like you are writing something on a table. You can use views as a way to restrict control to your data set because a view is a select. It's a select - you can select specific rows and specific columns out of your data set, out of your table. Then you can make that a view in a separate dataset, and that view, because it's part of a new dataset, can have its new access control. That access control might be a lot more limited. You can designate columns and/or rows which users don't have access to, because the view has done a select and restricted it.



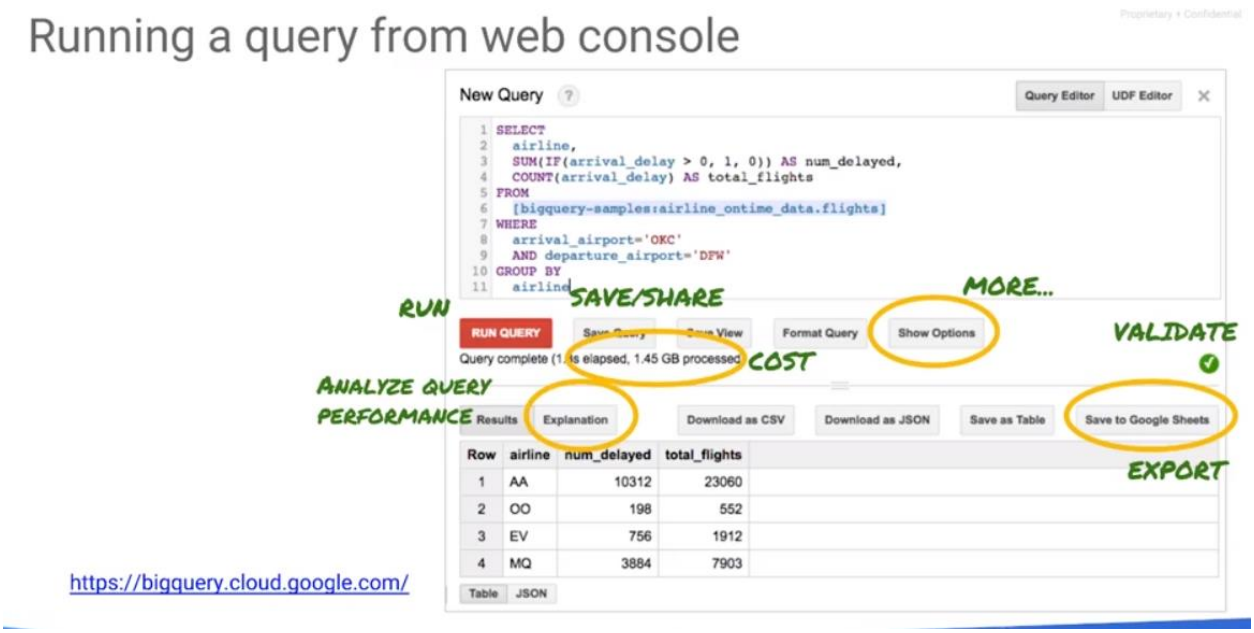
Tables are collections of columns. We say that tables are a collection of columns because BigQuery's a columnar database.

Views themselves are virtual tables – they are defined by a SQL query. Even though we normally think of tables as being part of BigQuery, BigQuery is a really a query engine that separates storage and compute. BigQuery also comes with its own native storage format. And the BigQuery query engine can work with tables that are external to BigQuery. For example, a table could be on cloud storage - the table could work on a Google Sheet.

A job is something that works on the dataset. It is potentially long running. It could take a few seconds or it could take a few hours. You can cancel a job. When you cancel a job, what happens to cost? Well, you get charged for the amount of data that you have processed up to that point. Remember that with BigQuery, the storage is columnar. In a typical relational database, you tend to think of it as rows of data, records of data. And you basically use it that way because you want to support transactional updates. Well, BigQuery lets you update it. You have ways of deleting rows and updating rows, etc. But that's not the primary use case for BigQuery. BigQuery is not a transactional storage. The way BigQuery works is that every column Is stored in a separate file. That file is encrypted, it's replicated, it's compressed. And because it's stored as separate files, you don't need keys or partitions. You can use partitions to basically save on cost. We'll talk about that later. BigQuery works on a column-by-column basis, so one of the ways that you can save money with BigQuery is to limit the columns that you run your queries on. You are charged for the number of columns of data that you're processing. BigQuery is primarily meant for immutable, very large datasets.

Running a Query

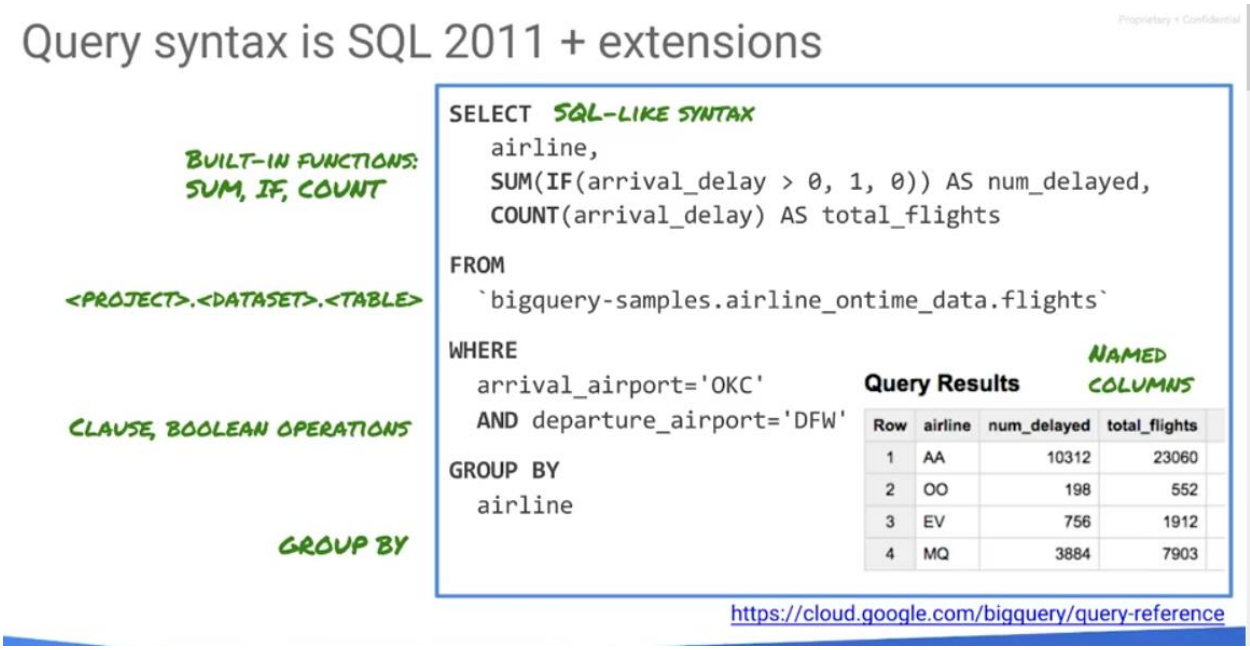
How do you write queries and functions in BigQuery? To run a query, use a web console as demonstrated earlier, okay.



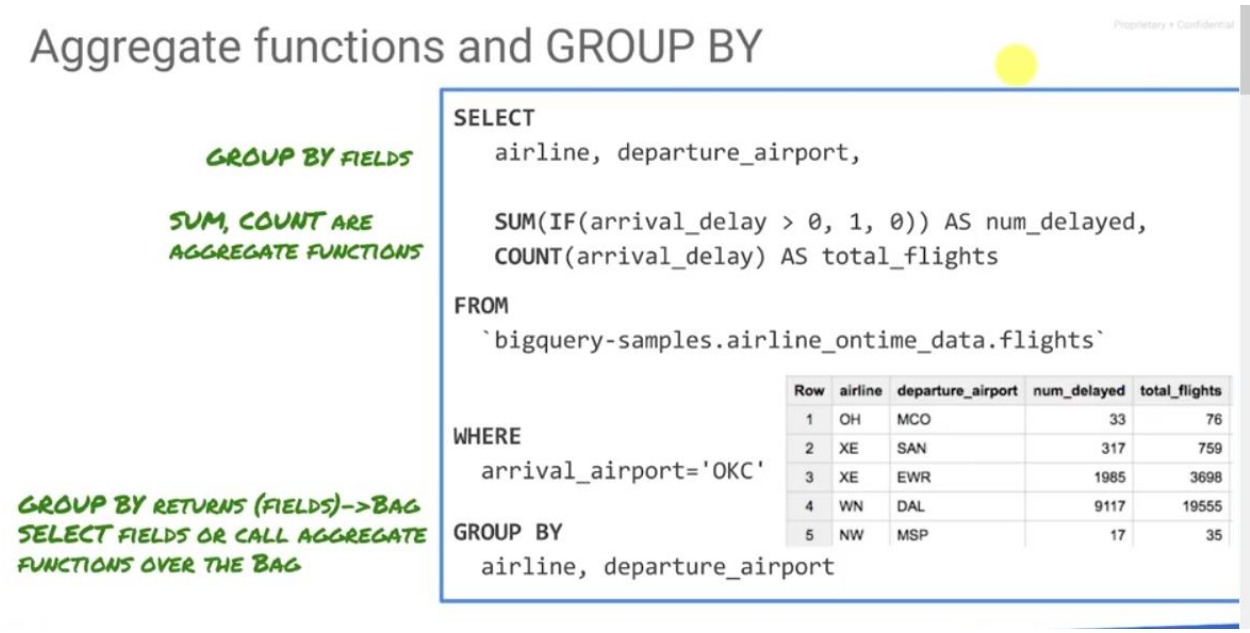
This is what the web console looks like - it changes over time so don't worry about if your web console doesn't look exactly the same. It's the basic concepts that matter. That red button runs the query. You write the query in that text box where it says 'New Query'. The rest of the menu is about how to save and share things. You have ways to save the query and save the view. There are options to turn on/off the legacy sequel or use the Google sequel (the default for BigQuery); turn off the toggle to use standard sequel and not Google sequel. You can download the results of a query as a CSV file, as JSON, save it as another table. The difference between a view and a table is that when a new table is materialized, it is no longer live. A view is not materialized and therefore it is live

The green arrow will validate the SQL, it will tell you how much data is going to get processed. Use this validate button to get an idea of cost if you are on a per use billing. Perhaps you want to know how much data gets processed because that's the amount of resources that you're using and that's the amount of resources that somebody else on that same flat rate plan doesn't have available to them at that exact time your query is running.

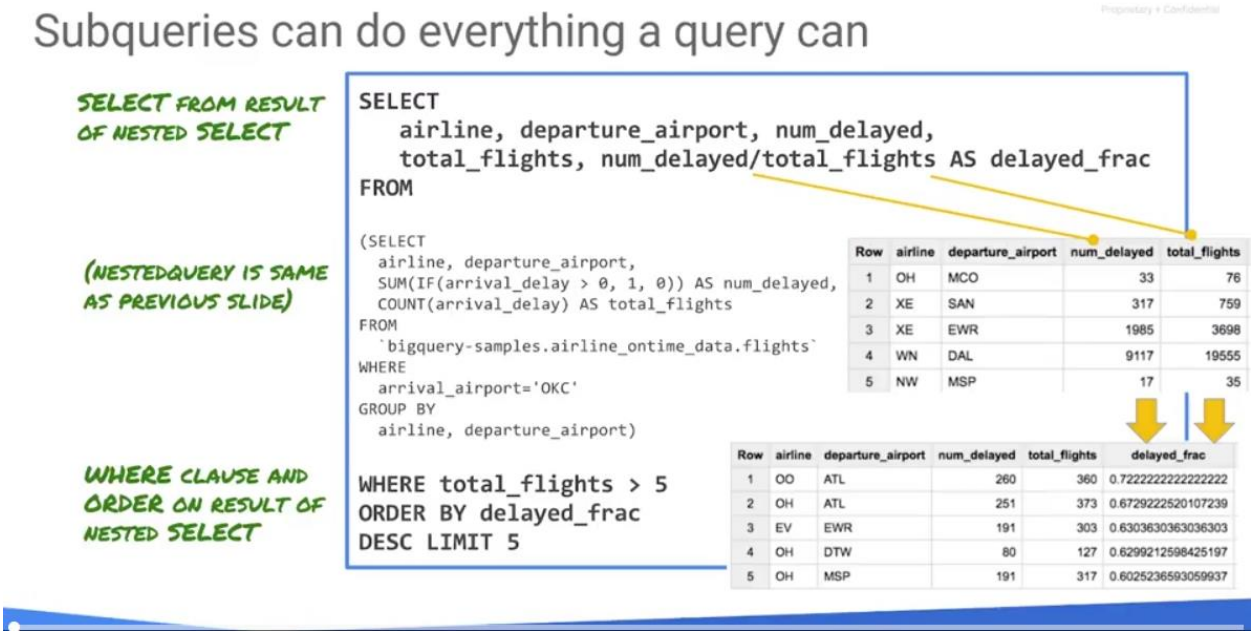
You can export into Google sheets as well not just as CSV or JSON but you can export to Google sheets. In fact, you can define a Google sheet as your external table. That is the reason you can write a complex query that runs in a very large petabyte scale dataset and then your Google sheet can be this really tiny table, human editable table, that maybe has like 10 rows in it that says this is the customer IDs I'm interested, these are the products I'm interested in, these are the dates that I'm interested in. You can then do a join of the Google sheet with this massive dataset and that's what your query could be. Your query could be joining a very large data set with a human editable Google sheet and then you can take the result and you can export it to Google sheet as well.



The query syntax itself is sequel 2011 plus a few extensions of the look alike. The table itself, the format is <project>.<dataset>.<table>. The project here is bigquery-samples. That dataset is airline_ontime_data, and the table is flights. If you leave out the project, by default, it will be the current project. You have access to BigQuery through the console, but there is also a ReST API making possible to BigQuery programmatically. There are a variety of client libraries that use that exact same rest API. It can be called from Python in which case you are returned a Pandas data frame, You will always need data set dot table. BigQuery uses "fully-functional" SQL supporting joins, aggregates (sum and counter, etc.) functions. It supports grouping and "Select As".



Complex Queries and Functions



Take the entire query that it did on the previous slide – the one not in bold. That's a nested query. Notice the SELECT on airline and airport is the same query as on the previous slide; it is an inner query which returns some results set. This can be treated as a table or as a view, and now you can SELECT on that.

A query could process gigabytes or more of data.

This is legacy SQL: SELECT something FROM. You can use date ranges: TABLE_DATE_RANGE and wildcards. Use clauses, inner selects, join on fields across table, use built-in functions like LPAD, etc.

In BigQuery, you have a variety of other projects. You can switch to, display, and explore these problems. One of the ones that should be explored are BigQuery samples: https://cloud.google.com/bigquery/sample-tables. If you enter the project, the BigQuery samples will show up on the left hand side of your BigQuery console with a variety of data sets there. Pick one of these datasets and try an analysis on it. How do I write a query to look at the Nasdaq stock quote for the Google stock for example.

You should look at a holistic example of how something is put together and that's the purpose of these labs.

Lab: Queries and Functions

https://codelabs.developers.google.com/codelabs/cpb101-bigquery-query/

Load and Export Data

Why load data into BigQuery? Primarily because BigQuery is the end of many roads in GCP. Wherever you get your data from, whether you're ingesting and capturing it from App Engine or through log files, or from Google Analytics, or you're receiving data in real time from a sensor out in the field with Pub/Sub, inevitably, at some point, your data makes its way into BigQuery. BigQuery acts as your storage. It also acts as the data

warehousing mechanism where your data is ingested and you can execute SQL queries on the data. You can process it with Dataflow and with Dataproc. You may be store it on Cloud Storage. However, very commonly, if the data are tabular in nature, you store it in BigQuery. You can perform analysis in BigQuery and then use all of the connected tools to visualize your data and to share your data around. It is important to look at how to get data into BigQuery, because that's something that you'll be doing quite a bit,

You can load data into big query using command line interface called BQ. It comes with the Gcloud SDK. You can use the web user interface or you can use an API: a python API, a data flow API, pretty much all of the tools in Google cloud. You can make queries and write data into the query.

Lab: Load and Export Data

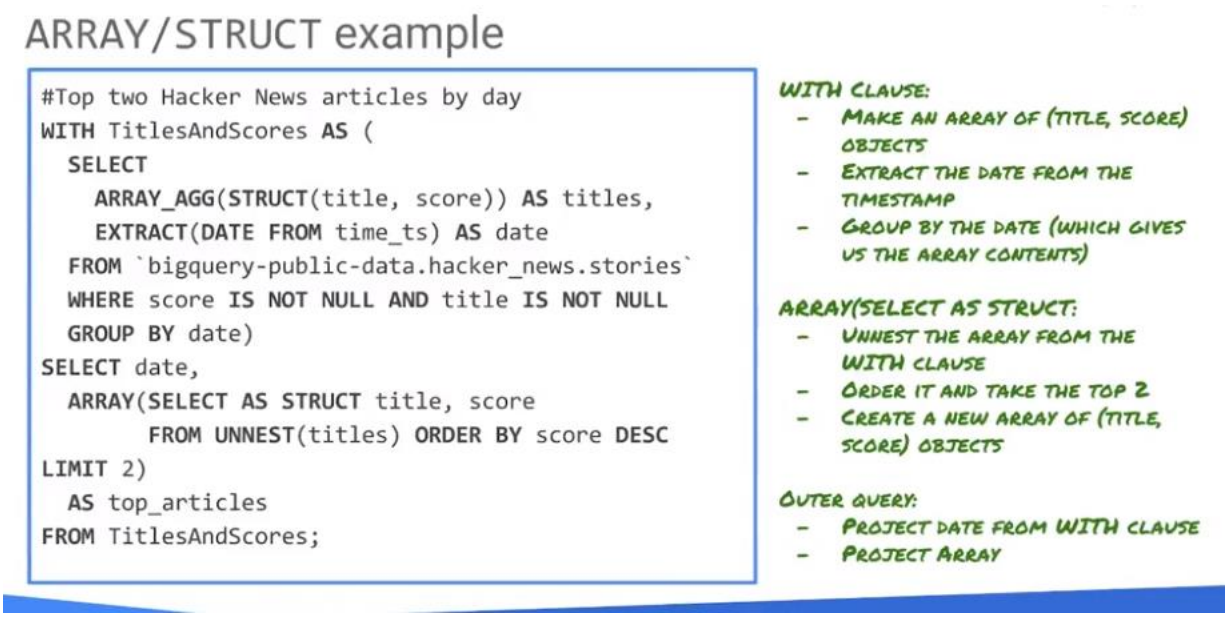
https://codelabs.developers.google.com/codelabs/cpb101-bigquery-data/

Advance Capabilities in Big Query

BigQuery supports all of the standard SQL types: strings, 64-bit integers, double precision floats, booleans, and arrays, which are an ordered list of elements which may be empty. Every array essentially contains many elements of integers, floats, Booleans, strings or even structs. A struct contains fields. Each of those properties or fields has a type and has a field name. Typically, give it a name so it can be selected using that name. And in addition, there is a type called timestamp which is to millisecond precision and you can have years between 1 and 9999.

Processing BigQuery Data Types

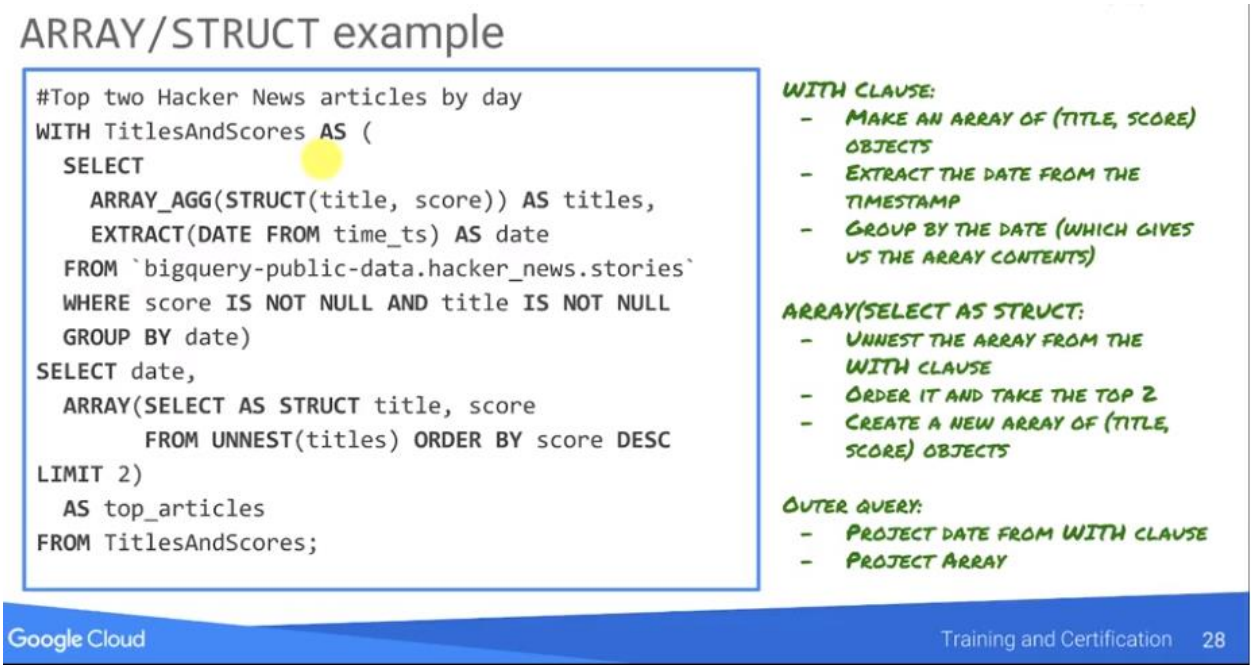
We have considered queries with the WITH clause and the COUNT function. In this example, we consider the use of ARRAYS and the use of STRUCTS. The first part of this query is to select the title and the date from news stories and the score. WITH Titles selects every article that has a publication title, score and date. We will query by Hacker News. We will group by date – for every date we get a collection of news articles. That is the inner query.



In the inner query, group by date, where score is not null and title is not null. Select the news article with a title and a score from this table of Hacker News stories. From that, select the date and there is no date field as such there's only a timestamp. The function extract extracts the date from the time stamp. Retrieve the date, title and the score. Before we look at the array_aggregate(array_agg), let's look at what selecting the title and the score would do for us.

As you can see, this is not going to run because there are many titles and many scores for any specific date. We need do reduce the number of items with that date. For example, we could find maximum score and/or the maximum title. That doesn't really help us – it is not all that useful because this score and this title are not related. All we do is just arrange the title. We've arranged all of the titles, we've arranged all of the scores but what we want is to find the title with the maximum score. Perform an aggregation where we aggregate the title and score into an array. We have not yet located the maximum.

Running this, we notice that for any date, for the date 2012-05-07, we now get a large group of titles and the corresponding scores. These are all of the articles that were published on that specific date. We can process other dates as required. We've aggregated an array and now we can process this array and find the article with the maximum score.



The inner query with SELECT ARRAY_AGG finds for every date all of the articles published on that date and their corresponding scores. We create a new result set WITH TitlesAndScores. From TitlesAndScores we are selecting the date and we are taking the titles. We unnest the titles, and retrieve the title and score, ordering by score, limit two. We find thetwo highest rated articles on that particular day. The inner query basically gives us the array. And now we're selecting the date. And we are selecting for each of those dates we are looking at this array, un-nesting it, getting back title and score. And from that we are ordering it by score in descending order. We find the highest rated score first and limiting it to the first two articles. If we run it we're not going to get a bazillion articles on a particular day, we're going to get only two articles for every specific day.

Standard SQL and Window Functions

Thus far, we've looked at only JOIN on equality conditions. But JOINs are not limited to just equality conditions. You can actually JOIN on two tables using any Boolean condition including less than, greater than and also functions that return a Boolean. For example, the “startswith” method which determines if a word starts with some prefix returns a Boolean.

JOIN condition example

```
#Top name frequency in Shakespeare
WITH TopNames AS (
  SELECT name, SUM(number) AS occurrences
  FROM `bigquery-public-data.usa_names.usa_1910_2013`
  GROUP BY name
  ORDER BY occurrences DESC LIMIT 100)
SELECT name, SUM(word_count) AS frequency
FROM TopNames
JOIN 'bigquery-public-data.samples.shakespeare'
ON STARTS_WITH(word, name)
GROUP BY name
ORDER BY frequency DESC LIMIT 10;
```

WITH CLAUSE:
- COUNT THE WORD OCCURRENCES IN `USA_1910_2013`

OUTER QUERY:
- JOIN `SHAKESPEARE` TO `TOPNAMES` WHERE THE WORD IN SHAKESPEARE STARTS WITH THE TOPNAMES RESULT
- SUM THE NUMBER OF WORD_COUNTS MATCHING THE NAME

In the above query, we have a table called USA baby names between 1910 and 2013. Count the number of times that a baby has been given that name and order descending, limit 100 such that we find the 100 most common baby names.

What is word count? Word count is a count of words ordered by frequency.

Over the course of these example queries, we've looked at a variety of standard SQL functions. There are a variety of standard SQL functions, aggregate functions like sum, count, max, etc. String functions, analytic functions, datetime functions, and array functions. A string function, REGEXP_CONTAINS is a string function that performs regular expression matching. We show the count of the matches against words, grouping the word and showing you the counts.

Standard SQL functions

- Standard SQL provides many functions:
  - Aggregate functions
  - String functions
  - Analytic (window) functions
  - Datetime functions
  - Array functions
  - Other functions and operators

You can also do analytical window functions: sums, averages, min, max, count. You can also do navigation: if you are on a particular row, you can say show me the row that is n rows ahead of this row, or n rows behind this row, so lead and lag. If you just do a select on BigQuery, the order in which the things come back is pretty arbitrary. However, if you order the query, you can also do things like medium, things like the 10th percentile, 20th percentile, by using and NTH value.

Analytical window functions

- Standard aggregations
  - `SUM, AVG, MIN, MAX, COUNT, etc.`
- Navigation functions
  - `LEAD()` – Returns the value of a row *n* rows ahead of the current row
  - `LAG()` – Returns the value of a row *n* rows behind the current row
  - `NTH_VALUE()` – Returns the value of the *n*th value in the window
- Ranking and numbering functions
  - `CUME_DIST()` – Returns the cumulative distribution of a value in a group
  - `DENSE_RANK()` – Returns the integer rank of a value in a group
  - `ROW_NUMBER()` – Returns the current row number of the query result
  - `RANK()` – Returns the integer rank of a value in a group of values
  - `PERCENT_RANK()` – Returns the rank of the current row, relative to the other rows in the partition

These are all within Windows – they are applied on some Group of records. You also have ways to rank, percent rank, row number within a result set, cumulative distribution, etc.

Window function example

Window function using RANK

```
SELECT
   corpus,
   word,
   word_count,
   RANK() OVER (
     PARTITION BY corpus
     ORDER BY word_count DESC) rank,
FROM
   `publicdata.samples.shakespeare`
WHERE
   length(word) > 10 AND word_count > 10
LIMIT 40
```

Row	corpus	word	word_count	rank
1	kinghenryv	WESTMORELAND	15	1
2	kinghenryvi	Chamberlain	53	1
3	3kinghenryvi	NORTHUMBERLAND	21	1
4	3kinghenryvi	Plantagenet	14	2
5	othello	handkerchief	29	1
6	1kinghenryiv	WESTMORELAND	16	1
7	1kinghenryiv	NORTHUMBERLAND	13	2
8	1kinghenryiv	Westmoreland	12	3
9	1kinghenryvi	PLANTAGENET	32	1
10	1kinghenryvi	Plantagenet	12	2

The thing to realize is that all analytical functions compute aggregate values over groups of rows, and or windows of rows. Therefore, you use the over clause as shown in the example above.

User Defined Functions

Date and time functions

- Enable date and time manipulation for timestamps, date strings, and `TIMESTAMP` data types
- BigQuery uses Epoch time; returns values based on the UTC time zone by default
- Many date and time functions, including:
  - `DATE(year, month, day)` — Constructs a DATE from INT64 values representing year, month, and day
  - `DATE(timestamp)` — Converts a timestamp_expression to DATE and supports time zone
  - `DATETIME(date, time)` — Constructs a DATETIME object using DATE and TIME objects

There are a variety of date and time functions that operate on time stamp. For example, extract the date from a time stamp. BigQuery uses the epoch time - UTC. You can create a date using year, month and date. You can create a date using a time stamp. You can also create a Date Time object using separate date and time objects. Standard SQL UDFs are scalar; They provide functionality to do things like loops and non-trivial string parsing.

SQL user-defined function

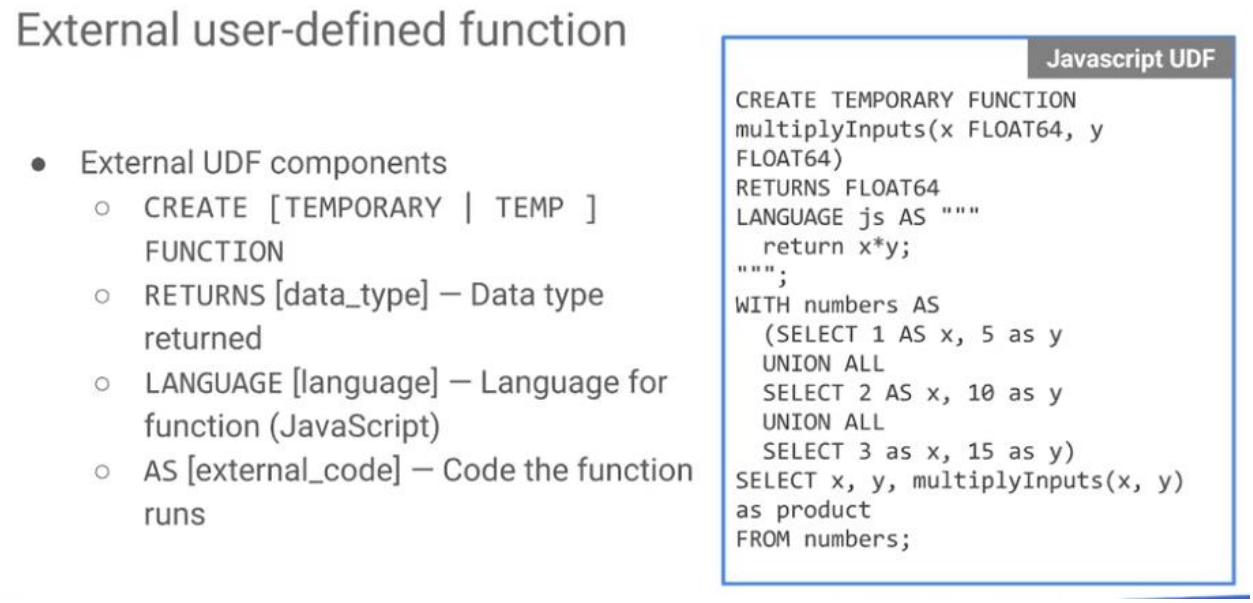
- Declare function with SQL query
  - `CREATE [TEMPORARY | TEMP ] FUNCTION` creates function
- Function can contain zero or more named parameters
  - Comma-separated (name, type) pairs

SQL UDF

```
CREATE TEMPORARY FUNCTION
addFourAndDivide(x INT64, y
INT64) AS ((x + 4) / y);

WITH numbers AS
  (SELECT 1 as val
  UNION ALL
  SELECT 3 as val
  UNION ALL
  SELECT 4 as val
  UNION ALL
  SELECT 5 as val)
SELECT val, addFourAndDivide(val,
2) AS result
FROM numbers;
```

Consider a SQL function we name addFourAndDivide. X is an INT64, Y is an INT64. Those are the two parameters, and it's AS, so you're now defining it, ((x + 4) / y). WITH numbers where SELECT 1 as val UNION ALL, SELECT 3 as val UNION ALL, SELECT 4 as val UNION ALL. Select the val addFourAndDivide val comma 2 as a result from numbers. We use an array of numbers - the key thing to do is look at the first part which is how you define a temporary function, addFourAndDivide. You essentially define the function, define its parameters, define their types, you say, ask, and then you basically write what that function needs to do.



You can also create external UDF components. For example, here the language is JavaScript. Here's a temporary function, multi multiple inputs but it's not written in SQL, it's written in JavaScript. And so the rest of it between the three codes is all JavaScript. You can write JavaScripts UDF's, you can write SQL UDF's There are some constraints as far as user-defined functions are concerned. UDFs can't return way too much data – they need to return reasonable amounts of data, less than 5 megabytes. You cannot run lots of JavaScript UDFs concurrently – you can only run six at the time that we're recording here. You can't do native code JavaScript – there's only things that can run within a sandbox. And JavaScript is a 32-bit language inside BigQuery so even though BigQuery has int64s, it will truncate to 32 bits.

User-defined function constraints

- Amount of data UDF outputs per input row should be <=5 MB
- Each user can run 6 concurrent JavaScript UDF queries per project
- Native code JavaScript functions aren't supported
- JavaScript handles only the most significant 32 bits
- A query job can have a maximum of 50 JavaScript UDF resources
  - Each inline code blob is limited to maximum size of 32 KB
  - Each external code resource limited to maximum size of 1 MB

Lab: Advanced Capabilities

Lab - Serverless Data Analysis (Java/Python) : Part 3

In this lab, you write a query that uses advanced SQL concepts:

- Nested fields
- Regular expressions
- With statement
- Group and Having

Answer the question: what programming languages do open-source programmers program in on weekends?

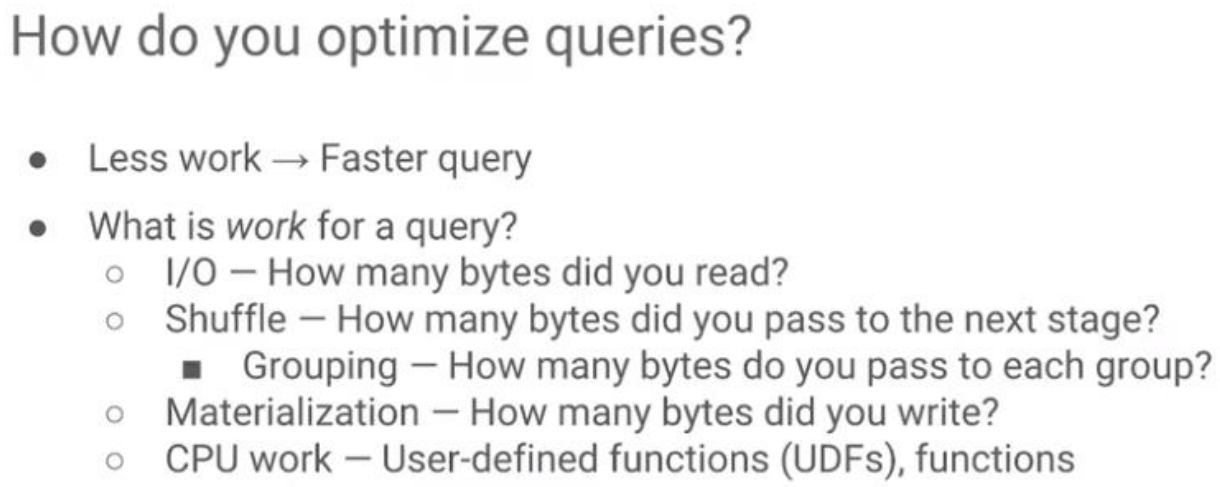
Codelab:

https://codelabs.developers.google.com/codelabs/cpb101-advanced-queries/

Review:

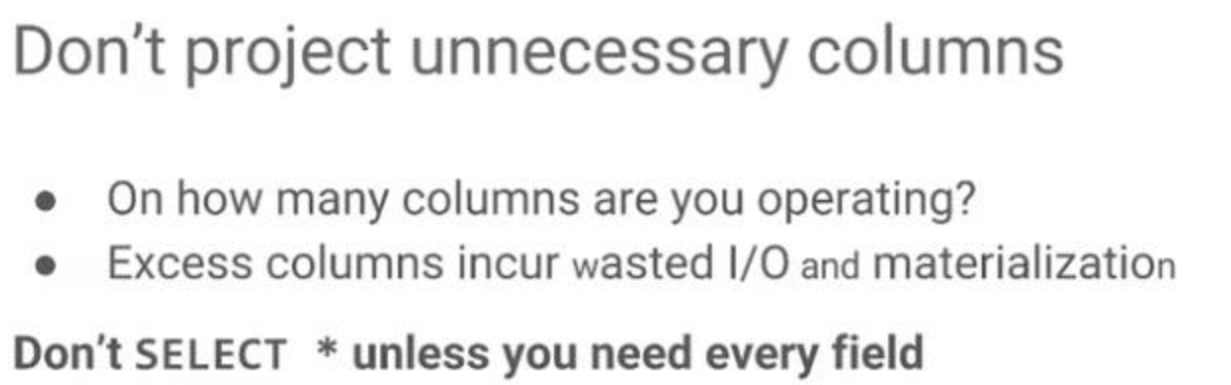
https://www.coursera.org/learn/serverless-data-analysis-bigquery-cloud-dataflow-gcp/lecture/wwFkp/lab-1c-review

Optimize for Performance and Pricing

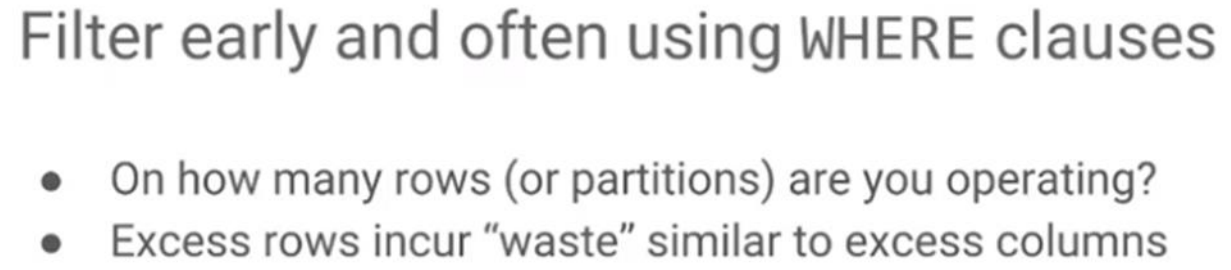


This is a discussion of performance and pricing of BigQuery. In terms of performance, we're looking at optimizing queries. And a query is better if it's faster, if it does less work. What constitutes work for a query? Work can be one of four things:

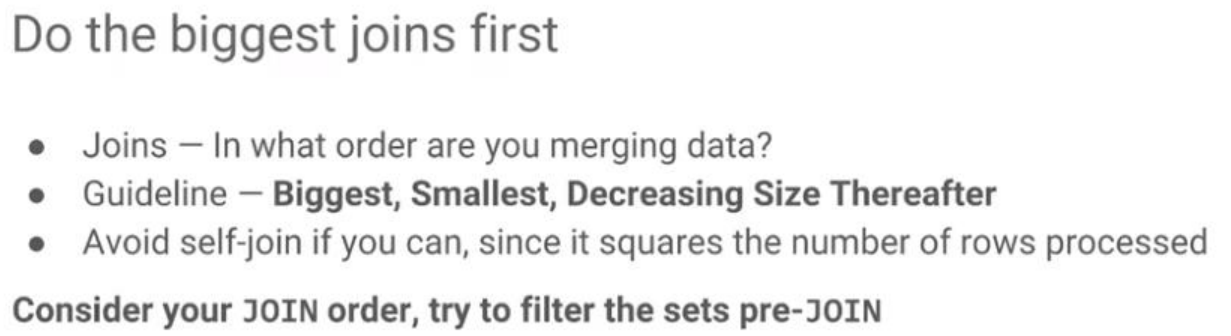
1. It's the amount of data that the query is reading the input and output of the query.
2. It could be the shuffle because any query will have a plan and the amount of data that basically gets passed from one stage to the next stage is the shuffle. The lower the amount of data that's passed from one stage to the next, the faster the query is going to be.
3. You also have to worry about whether anything requires materialization. If we can just hold it in memory between stages and keep going, such a query is going to be faster than a query that requires intermediate outputs or final outputs to be materialized for the bytes to be written out.
4. You have to think in terms of the CPU overhead associated with the functions that you're calling. Some functions, like sum or count, are probably very cheap. Other functions, such as sine or cosine, are more expensive. Depending on the CPU overhead of the functions, some queries will take longer than others.



How do you improve the performance of a query? Because BigQuery is a columnar database, limit the number of columns that you're processing. Don't write a SELECT * unless you actually need every field. Excess columns will involve a lot more input/output, a lot more materialization, more use of the network, it's going to be slower. Do not project unnecessary columns in your query. Select only the fields that you actually want.



Second row - because we want to limit the amount of data that's passed from one stage to the next stage, you want to carry out filtering or WHERE clauses as early as possible. Try to perform WHERE clauses as early as possible to reduce the amount of data that's parsed between stages of your query. The more rows you're processing, the more data you're processing and therefore the slower your query is going to be.



When you are joining, consider the order in which you're merging the data. The guideline is to perform the largest joins first and do smaller joins later. Essentially, reduce the amount of data that need to travel through your query. Do that by doing your big joins first.

Low Cardinality GROUP BYs are faster

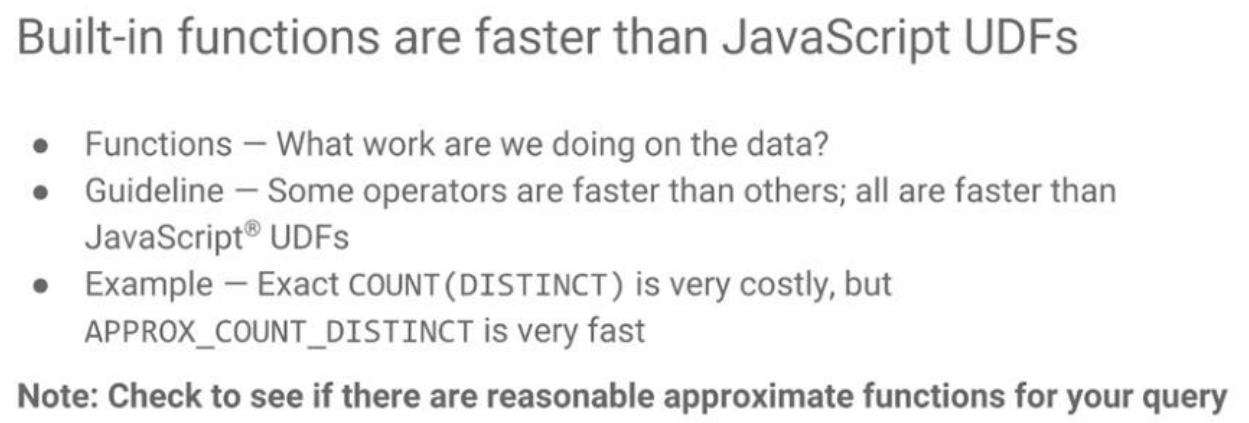
- Grouping — How much data are we grouping per-key for aggregation?
- Guideline — Low-cardinality keys/groups → fast, high-cardinality → slower
- However, higher key cardinality (more groups) leads to more shuffling; key skew can lead to increased tail latency

Note: Get a count of your groups when trying to understand performance

Whenever you're doing a GROUP BY, you should think about how much data is actually getting grouped per key because GROUP BY is per key. For example, in the programming example, we grouped it by language so it now depends on how many files are there and how many commits are there per language – that's the cardinality of the group. The lower the cardinality, the faster the query is going to be. The higher the cardinality, the slower it's going to be.

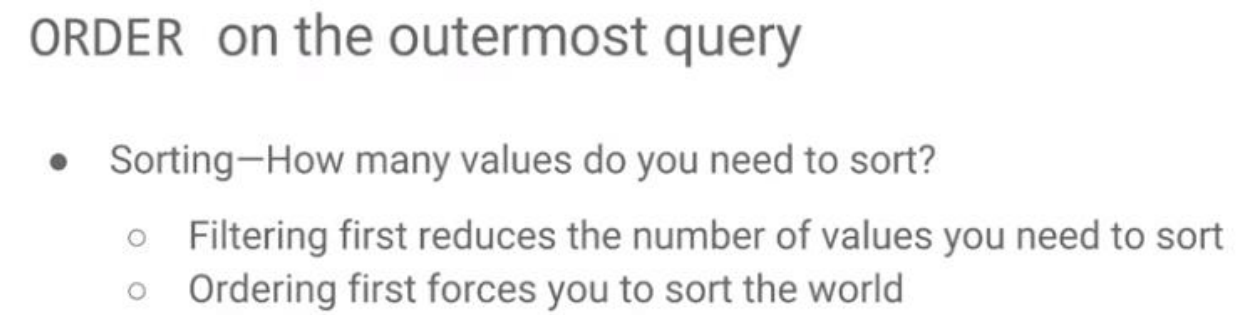
> In mathematics, the cardinality of a set is a measure of the "number of elements of the set". For example, the set A = {2, 4, 6} contains 3 elements, and therefore A has a cardinality of 3. There are two approaches to cardinality – one which compares sets directly using bijections and injections, and another which uses cardinal numbers.[1] The cardinality of a set is also called its size, when no confusion with other notions of size[2] is possible.

If you have a language, like say JavaScript which has many commits, that key, that JavaScript key is going to be slow. If on the other hand you have a key of Haskell – and Haskell is a language that's not used that often, it's a low-cardinality key, and so anything associated with that key is going be a lot faster. At the same time, if you have many keys or you have many GROUP BYs, then will have more shuffling and you will have a lot of tail latency (tail latency in this context is that you will have some keys that don't actually have lots of data associated with them). In other words, to understand the performance of a query that involves a GROUP BY, you should have a pretty good idea of how many rows each of the GROUP BYs is actually processing. It is a good idea to think about the number of rows, and if you look at our query earlier, we were counting the number of commits, that is the number of rows. So that gives us an idea of the amount of data that each of those keys is associated with. Some keys you could avoid processing with a where clause. For example, process only those keys that have more than a 100 commits, i.e., num_commits greater than 100, and that's one way to reduce tail latency.

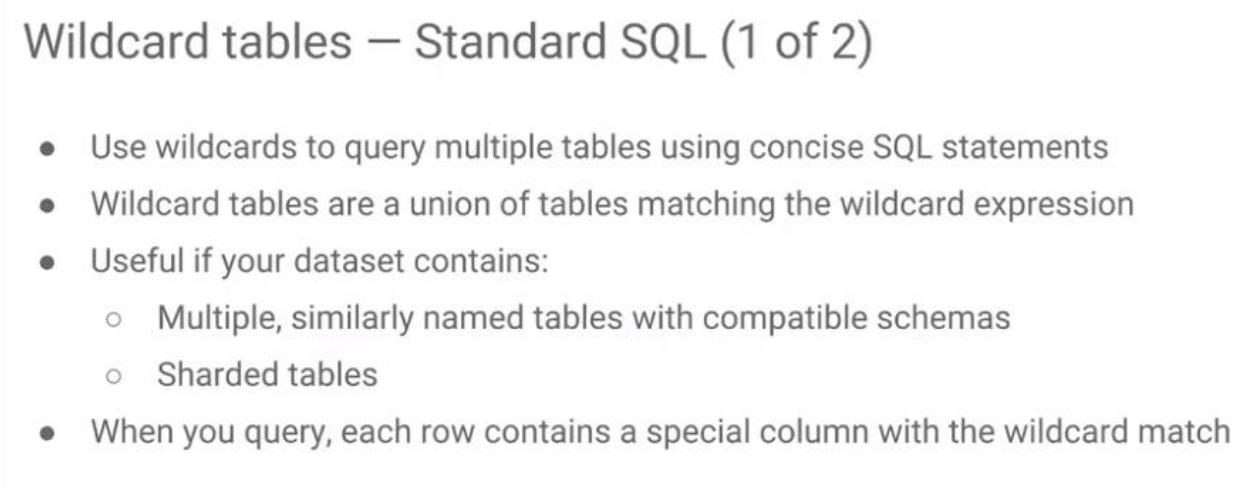


When choosing functions, consider the CPU overhead of those functions. Every function is going to be different. But there are a few guidelines.

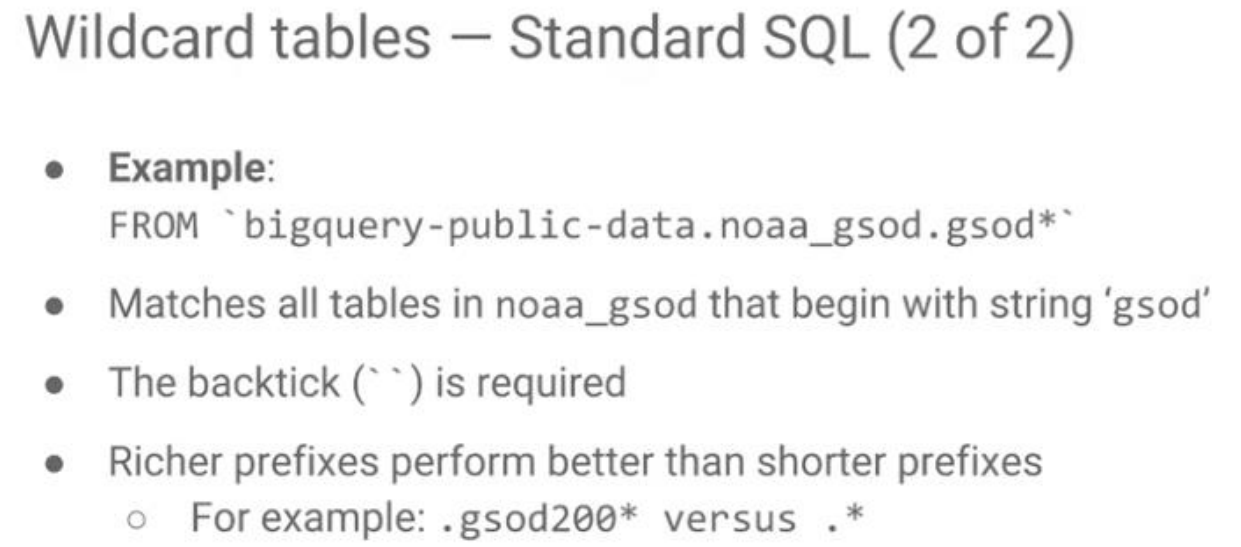
- Built-in functions are going to be the fastest. If you have a SQL function that exists, you should try to use that first. JavaScript UDFs tend to be the slowest. A function that you write, that you define with Sequel is going to be somewhere in between. Use a built-in function when you can, write your own UDF in sequel, if you can, and as a last thing where you have no other option, then write a JavaScript UDF.
- Consider differences between functions that are similar to each other. For example, there are a variety of APPROX functions that are supported by BigQuery. An APPROX quintile will be faster than a regular quintile. An APPROX count will be faster than a regular count. If an approximate result is good enough, then use it. Note that approximate in this instance typically means that it's within one percent of the actual number.
- When ordering, apply an order only on the outermost query. Doing an order on an inner query usually won't affect the final output. Sort on the smallest amount of data: that tends to be your outermost query. So, you want to filter first, and then you want to order.



Tables and Partitioning for Performance



The other thing that you should think about is whether you can use wildcards. So, you can query multiple tables using your wildcard, and the number of tables that you're matching obviously affects your performance. So, when you do your query, if you have multiple tables with very similar names, you can use wildcard tables.



For example, from bigquery-public-data.gsod* which matches all of the tables in noaa_gsod that begin with the string 'gsod'. Because we use a wildcard, we need to backtick to make sure to escape it. Also, when you're doing these wildcards, the richer your prefix, in other words, the more information you can provide to BigQuery, the faster it's going to be. Specify as much of the common letters as possible. But where we talked about wildcards, there is another option that you have, which is partitioning the tables by timestamp.

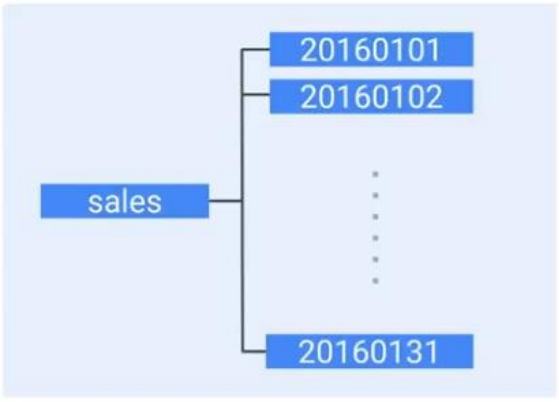
Table partitioning

- Time-partitioned tables are a cost-effective way to manage data
- Easier to write queries spanning time periods
- When you create tables with time-based partitions, BigQuery automatically loads data in correct partition
  - Declare the table as partitioned at creation time using this flag: `--time_partitioning_type`
  - To create partitioned table with expiration time for data, using this flag: `--time_partitioning_expiration`

You can specify a particular column in your data, that is, your timestamp and partition behind the scenes based on timestamp. Timestamps will give you similar performance benefits as having a sharder table, right? If you have a sharder table, you would basically be splitting it and maybe doing your queries only on a few shards, a few dates. Well you can do the exact same thing with time-partitioned tables.

Example — Table partitioning

```
SELECT ...
FROM `sales`
WHERE _PARTITIONTIME
BETWEEN TIMESTAMP("20160101")
    AND TIMESTAMP("20160131")
```



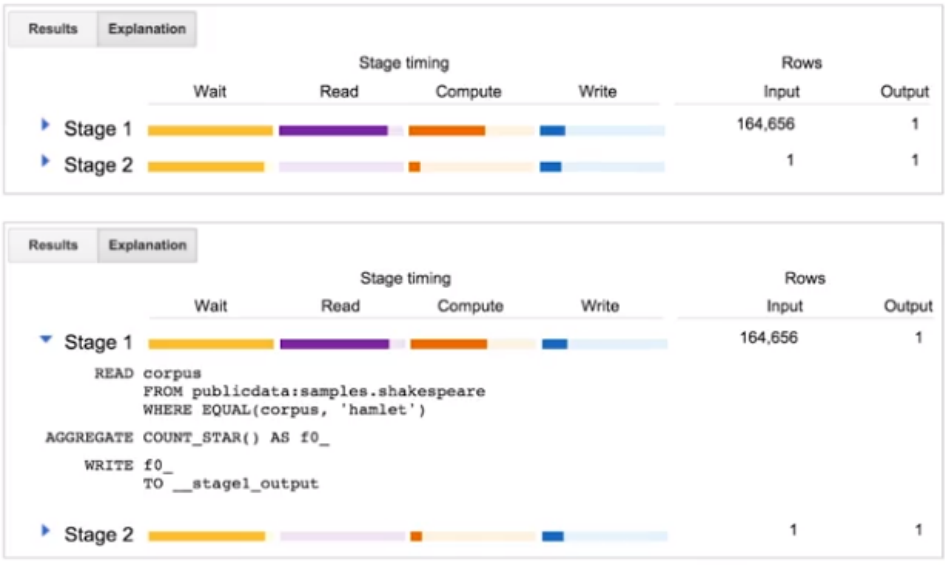
The way time partitioning is that you 'select from sales' where sales is a table but you partition between timestamps. It is very similar to having a wildcard table where the wildcard is your date except that your BigQuery dataset only contains one table. It's just that it happens to be partitioned by time, by default. Wildcard tables involve many tables. Partitioning gives you a cleaner interface just as one table, but it imposes some cognitive load on everybody writing queries to remember that this thing contains a partition column. The BigQuery console also helps, it warns you if there is a partition, if there is a partition column and you're not using it in your query. But it's just a warning - you can ignore it and actually do it if that's exactly what you want to do.

Understand query performance

- You can optimize our queries and your data, but still need to monitor performance
- Two primary approaches:
  - Per-query explain plans
    - "What did my query do?"
  - Project-level monitoring through Google Stackdriver
    - "What is going on with all my resources in this project?"

In order to improve query performance, you have to understand the query performance. The way to this is to run the query and then investigate what the query did. The other way is to monitor the query as it runs using stat driver which allows monitoring how CPU usage and disk usage.

BigQuery explanation plans (1 of 2)



This is what an explanation plan looks like – there are multiple stages in your query. You can see in the query which part of the query corresponds to what stage. Or you can just go ahead and click on the arrow. For example, this arrow and that it shows you the exact part of the query that falls into stage one. And with every stage it's waiting for the data to become available from the previous stage. It's then reading the data, it's then doing the computation, and it's writing the data to the next stage

What you should consider is the number of rows that are being processed at each stage. What is desired is that the number of rows that get processed are greatly decreasing.

How do you know if there is skew associated with any particular stage? If the maximum is dramatically bigger than the average, then you know that you have some things that take a long time to process, compared to the typical query. That indicates that you have tail skew.

BigQuery explanation plans (2 of 2)

The following ratios are also available for each stage in the query plan.

API JSON Name	Web UI*	Ratio Numerator **
waitRatioAvg	AVG	Time the average worker spent waiting to be scheduled.
waitRatioMax	MAX	Time the slowest worker spent waiting to be scheduled.
readRatioAvg	AVG	Time the average worker spent reading input data.
readRatioMax	MAX	Time the slowest worker spent reading input data.
computeRatioAvg	AVG	Time the average worker spent CPU-bound.
computeRatioMax	MAX	Time the slowest worker spent CPU-bound.
writeRatioAvg	AVG	Time the average worker spent writing output data.
writeRatioMax	MAX	Time the slowest worker spent writing output data.

* The labels 'AVG' and 'MAX' are for illustration only and do not appear in the web UI.

** All of the ratios share a common denominator that represents the longest time spent by any worker in any segment.

BigQuery Plans and Categories

Understand BigQuery plans

- Significant difference between avg and max time?
  - Probably data skew—use APPROX_TOP_COUNT to check
  - Filter early to workaround
- Most time spent reading from intermediate stages
  - Consider filtering earlier in the query
- Most time spent on CPU tasks
  - Consider approximate functions, inspect UDF usage, filter earlier

When you look at a BigQuery plan, consider any stage where there is a significant difference between the average and the max time. When this happens, it indicates a significant data skew. One of the ways that you

can fix this is to remove the tail, for example, with the having clause, filtering them out so that you're not processing those tail latencies. Also consider the amount of time that's been spent waiting – that might indicate that you might want to do your filtering earlier. You should also look at the time that spent on CPU tasks on the compute and if there's a lot of time that's spent on the compute. Whatever is taking the longest period of time, that's the thing that you should look at optimizing.

Monitor BigQuery with Stackdriver

- Available for all BigQuery customers
- Fully interactive GUI. Customers can create custom dashboards displaying up to 13 BigQuery metrics, including:
  - Slots Utilization
  - Queries in Flight
  - Uploaded Bytes (not shown)
  - Stored Bytes (not shown)

The other thing that you can do is that you can monitor BigQuery with Stackdriver. It is a fully interactive GUI. You can create dashboards, you can look at how many slots were utilized, what queries are in flight, how many bytes were uploaded or stored.

Three categories of BigQuery pricing

Storage

- Amount of data in table
- Ingest rate of streaming data
- Automatic discount for old data

Processing

- On-demand OR Flat-rate plans
- On-demand based on amount of data processed
- 1 TB/month free
- Have to opt-in to run high-compute queries

Free

- Loading
- Exporting
- Queries on metadata
- Cached queries
- Queries with errors

In terms of pricing BigQuery, there are several categories or considerations of BigQuery pricing.

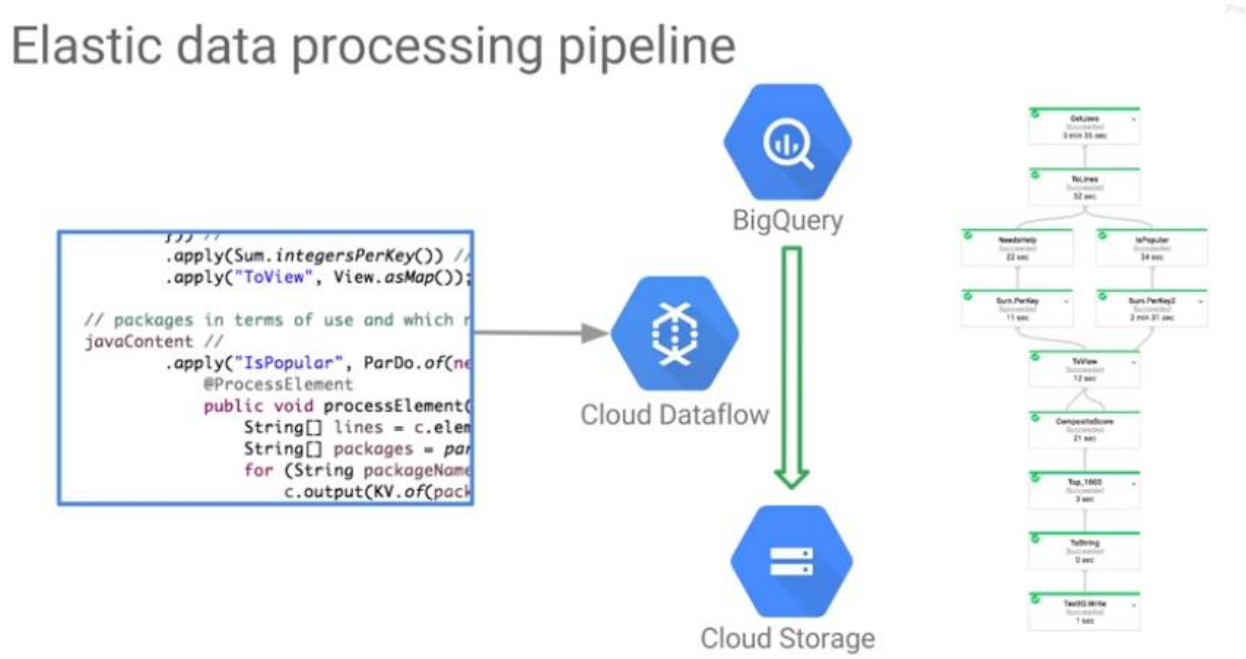
- Free Tier - all the things that are free. Loading data into BigQuery, exporting data from BigQuery, any queries on metadata. How many rows are there in this table? How many columns are there in this table? What are the names of the columns? Queries like that are always free. Any cached query is free. If you run a query and you run the exact same query in that project, it is free. Anything that has already been cached and you're getting it back is free. The caches are per user for privacy reasons. It is a per user cache, but any query whose results are returned from the cache is free. Any query that has an error is also free.
- There are things that are not free - storage and processing. You are charged based on the amount of data in the table. If you have streaming data, you get charged based on the ingest rate of the streaming data. And you will get an automatic discount for data that you haven't edited in a while. You automatically get a discount for older data.
- The other category is pricing is processing. For processing prices, you can have on-demand pricing. Each query we look at the amount of data that that query processes. And the charge for the query is based on the amount of data that's being processed, with the first terabyte a month being free. Processing beyond 1 terabyte is what you pay for. You also have a flat rate plan, but the flat rate plan is something that you talk to your sales team about.
- The last thing is that you may have some functions that are extremely high compute, especially JavaScript functions that are very high compute. You have to opt in to run them. There is a special charge for those.

Resources: the BigQuery documentation, tutorials on BigQuery, pricing information on BigQuery and client libraries for BigQuery. APIs for BigQuery from Python, Java are all available.

MODULE 2: AUTOSCALING DATA PROCESSING PIPELINES WITH DATAFLOW

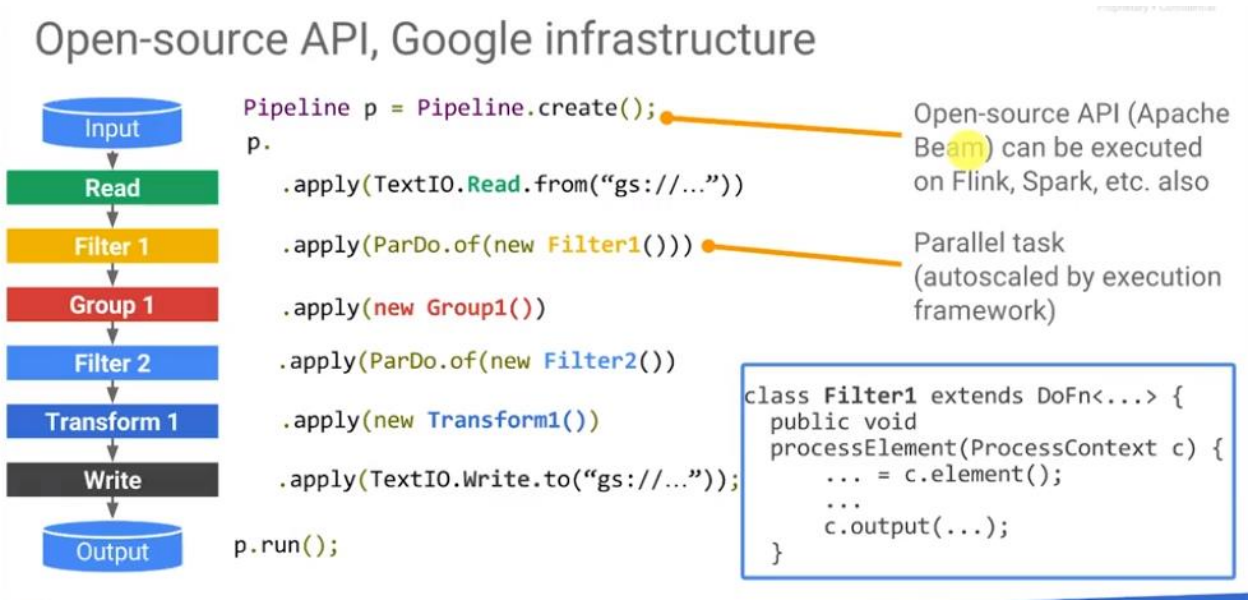
DataFlow and its Capabilities

Autoscaling data processing pipelines - In this chapter, we look at Cloud Dataflow, which is a way to execute Apache Beam data pipelines on Google Cloud platform. We look at how to write such data pipelines and how to carry out MapReduce kinds of programs in Dataflow and also some concepts such as how to deal with side inputs and how to carry out a streaming.

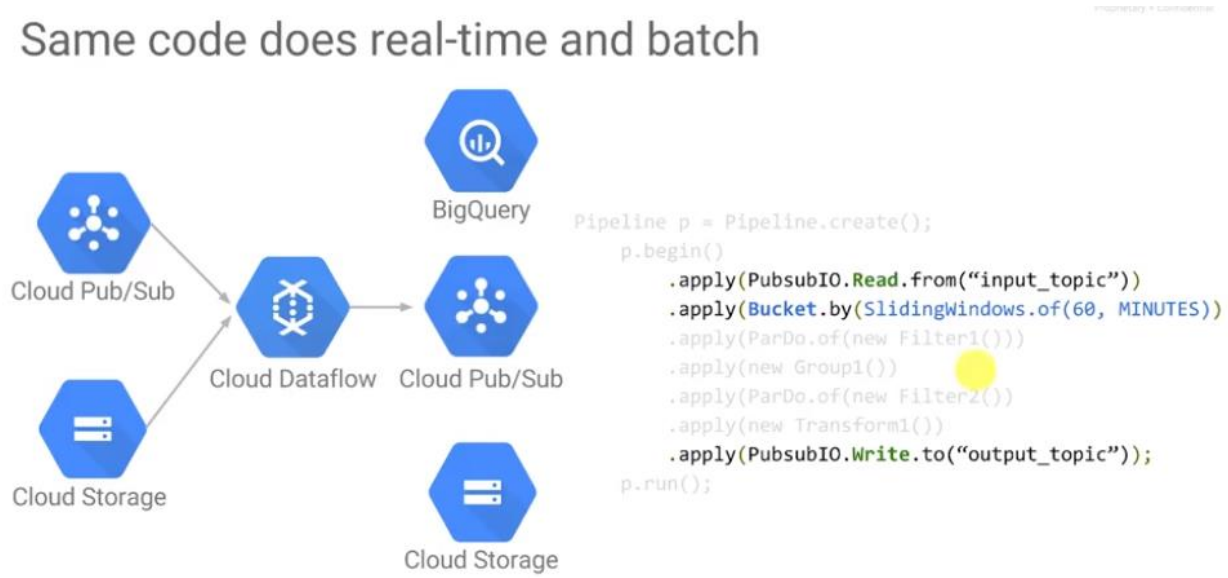


Dataflow is a way by which you can execute data processing pipelines on the cloud. In this case, for example, we're using Dataflow to carry out a pipeline that reads from BigQuery and writes into Cloud Storage. And it does so in a series of steps.

The key thing about Dataflow is that these steps, called transforms, can be elastically scaled. For example, if it turns out that one of these steps needs to get executed in parallel on 15 machines, then it can get auto-scaled to those 15 machines. And the results can then move onto the next step of the transform, which itself may get scaled only to five machines. It is elastic - each of whose steps are automatically scaled by the execution service.



Apache Bean is an source API and DataFlow is not the component that uses a Apache Beam in pipelines. Beam can be executed them on Flink and Spark, etc. We consider cloud DataFlow as the execution service for when we have a data pipeline that we would like to execute on the cloud. First, create your pipeline and then do a series of applies. For example, in this pipeline I'm reading from TextIO and I'm reading from Google Cloud storage. That's why there's a GS and then doing some kind of filtering, some kind of grouping, some kind of filtering, some kind of transformation and then writing those results out to another storage file on cloud storage. Each of these steps, Filter 1 group 1, Filter 2 Transform 1, the steps are user defined code. These Java classes and Filter 1 exhibits a special property, Filter one is executed within a Parallel Do. These means is that Filter one is going to get auto scaled, run on a whole number of machines scaled out, and then the results of Filter 1 are going to go streaming into Group 1 and the results of Group 1 again are going to get applied to Filter 2 but Filter 2 is again going to be completely parrallelized and run in parallel on many machines and the results of Filter 2 are going to stream in just Transform 1 and the results of Transform 1 are going to go into the output file.

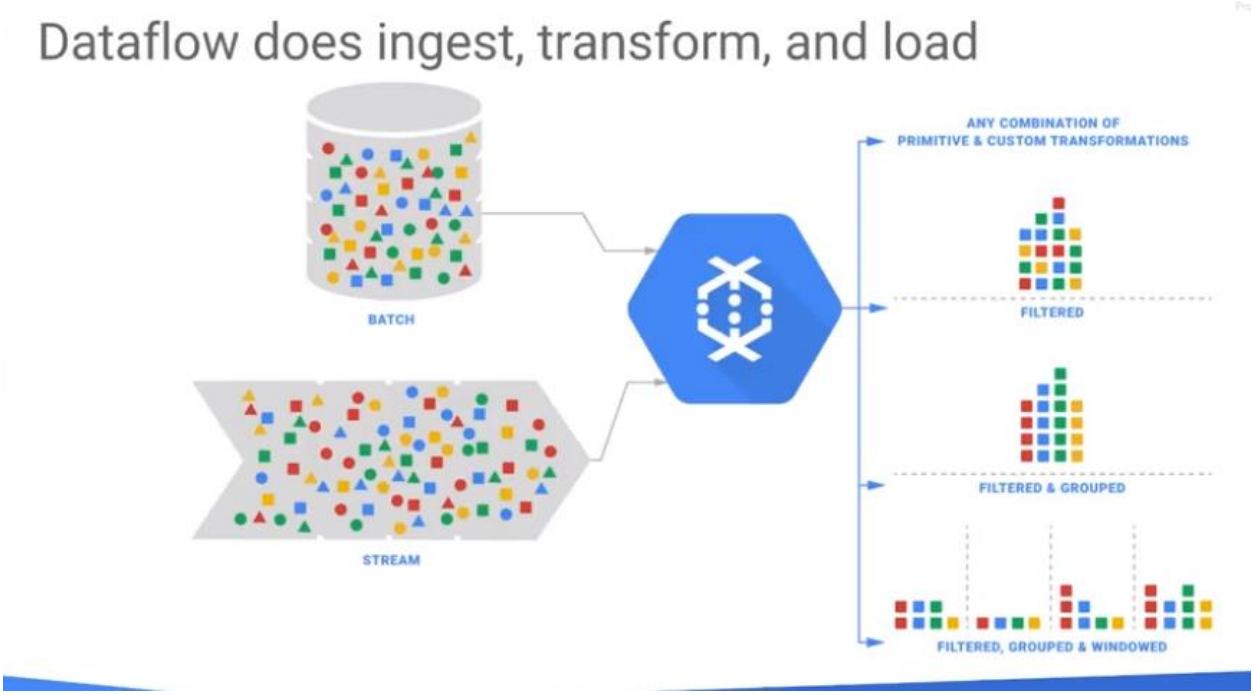


The use of a stream there was explicit because what data flow lets you do is that it lets you write code that processes both historical batch data, data that's complete, what we'll call bounded sources and sinks. The data is unbounded in nature.

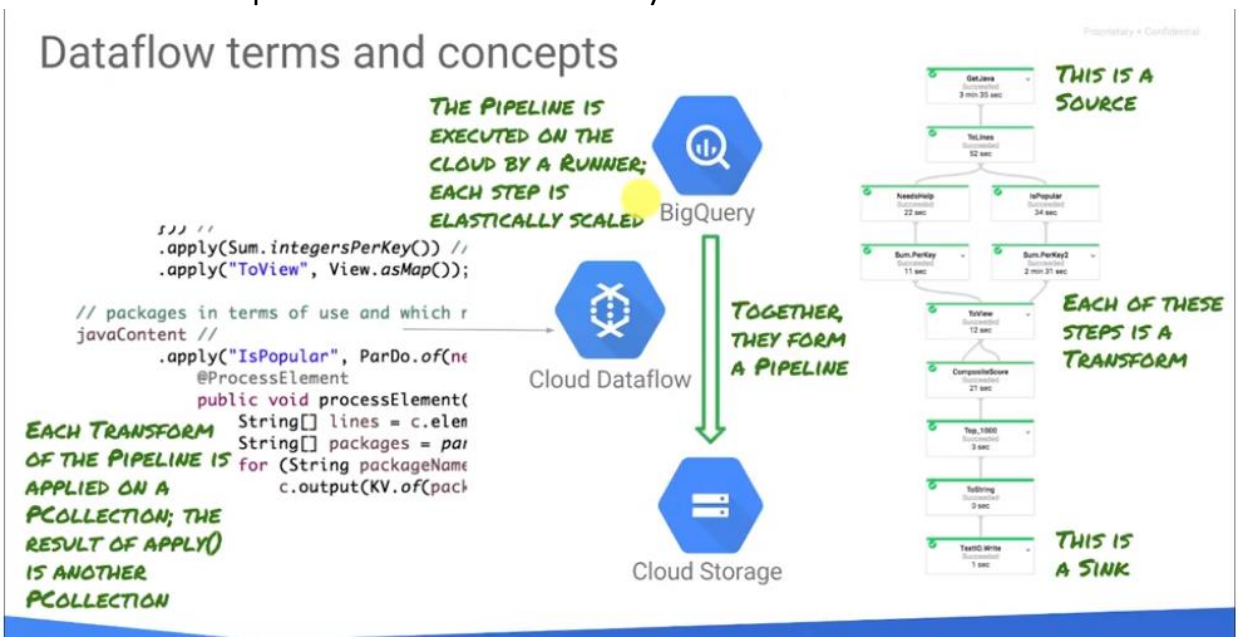
How can we group something that is unbounded in nature? You cannot, because you do not know if you are going to get some new member of the group. Anything calculated, e.g., calculating an average and mean will change over time as new data come in. When reading streaming data, typically you also apply a window to it. For a sliding window of 60 minutes, each of these transforms perform a mean over sixty minutes, in other words it is a moving average.

Dataflow lets you ingest data both from batch and from stream, use the same code, the same filtering, transforming, grouping code to carry out operations on both batch data and on streaming data. Of course, when you do a mean on batch data, you would get a mean over the entire dataset, whereas if you're doing

means on a streaming data, you will have to do a window, a window of, for example, 60 minutes, but you could do windows based on other things, e.g., the number of records, etc. You can apply the same windowing transforms to batch as well. In general, the practice is to apply the same code that processes streaming data is also the same code that processes batch data.



Write Data Pipelines in Java and Python



Before writing data pipelines, data pipelines that work both on batch data and on streaming data, let's review a few concepts. A pipeline is a set of steps. All of these steps are executed on the cloud by a runner. Dataflow

is a runner - an execution framework for this code that we have written. The code is written against Apache Beam and can be written in Python or Java. Steps are elastically scaled so they can be scaled and run on as many machines as possible. Each step is called a transform, and for this particular transform, its source is BigQuery and its sink is Cloud Storage. A typical pipeline goes from a source to a sink, involves branching, involves a number of transforms, and each of those transforms.

> There seem to be a lot of similar concepts in Spark 2.0 and Apache Bean – see https://cloud.google.com/dataflow/blog/dataflow-beam-and-spark-comparison.

The input to that transform is a parallel collection: a PCollection. When you do an apply, the apply is to a parallel collection or a PCollection. A PCollection is a collection – a list or a map of items if it's a key-value pair. The collection need not be bounded. That collection need not fit into memory, that's why we're calling it a PCollection.

A Pipeline is a directed graph of steps

- Read in data, transform it, write out
  - Can branch, merge, use if-then statements, etc.

```
import org.apache.beam.sdk.Pipeline;  // etc.

public static void main(String[] args) {
    // Create a pipeline parameterized by commandline flags.
    Pipeline p = Pipeline.create(PipelineOptionsFactory.fromArgs(arg));

    p.apply(TextIO.Read.from("gs://..."))   // Read input.
     .apply(new CountWords())               // Do some processing.
     .apply(TextIO.Write.to("gs://..."));   // Write output.

    // Run the pipeline.
    p.run();
}
```

A Pipeline is a directed set of steps, of these transforms. A typical Pipeline reads in data, does some transformations to it, and then writes it out. It can branch and merge. It can do anything that code can do, because it is code. So here is an example of a Pipeline. This is Java. I'll show you Python in a little bit. So here I'm creating a Pipeline.

A Pipeline is a directed graph which is not run or executed until the command p.run is executed on a specified runner. A runner may be a direct runner in which case it's running locally, right on your laptop. Or it could be a dataflow runner, in which case, this graph gets launched on the cloud, and all of the compute is now happening on the cloud.

Python API conceptually similar

- Read in data, transform it, write out
  - Pythonic syntax

```
import apache_beam as beam

if __name__ == '__main__':
   # create a pipeline parameterized by commandline flags
   p = beam.Pipeline(argv=sys.argv)

   (p
      | beam.io.ReadFromText('gs://...') # read input
      | beam.FlatMap(lambda line: count_words(line)) # do some processing
      | beam.io.WriteToText('gs://...') # write output
   )

   p.run() # run the pipeline
```

The Python API is very similar. We read from a source, apply a transform, and writing to a sink. The pipe operator in Python has been overloaded to mean apply. In Python, transforms are applied sequentially as in Java, but instead of the.apply, in Python, we're essentially using the pipe operator to carry out one transform after another. And just as in Java, we first create a graph, and then we run it.

The data in a pipeline, as we talked about, is represented by a parallel collection, so every transform is applied to a parallel collection. The input to a transform is a parallel collection, the output of the transform is a parallel collection. A parallel collection which is not applied results a parallel collection. This is the same whether it's Java or Python except that in Java, because Java is a type safe language, you explicitly define it.

Apply Transform to PCollection

- Data in a pipeline are represented by `PCollection`
  - Supports parallel processing
  - Not an in-memory collection; can be unbounded

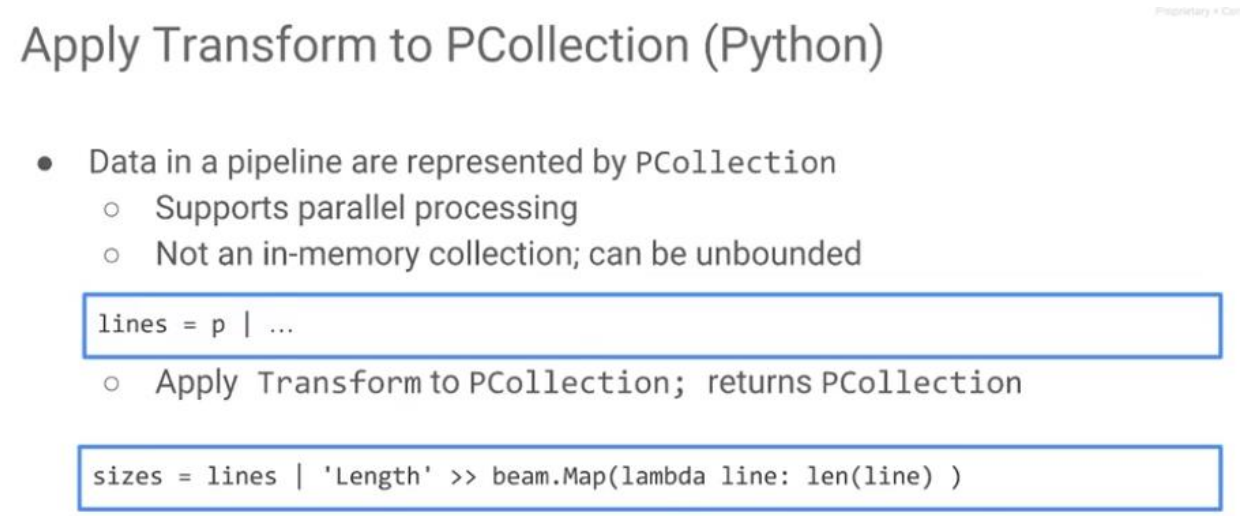
```
PCollection<String>  lines = p.apply(...)   //
```

  - Apply `Transform` to `PCollection`; returns `PCollection`

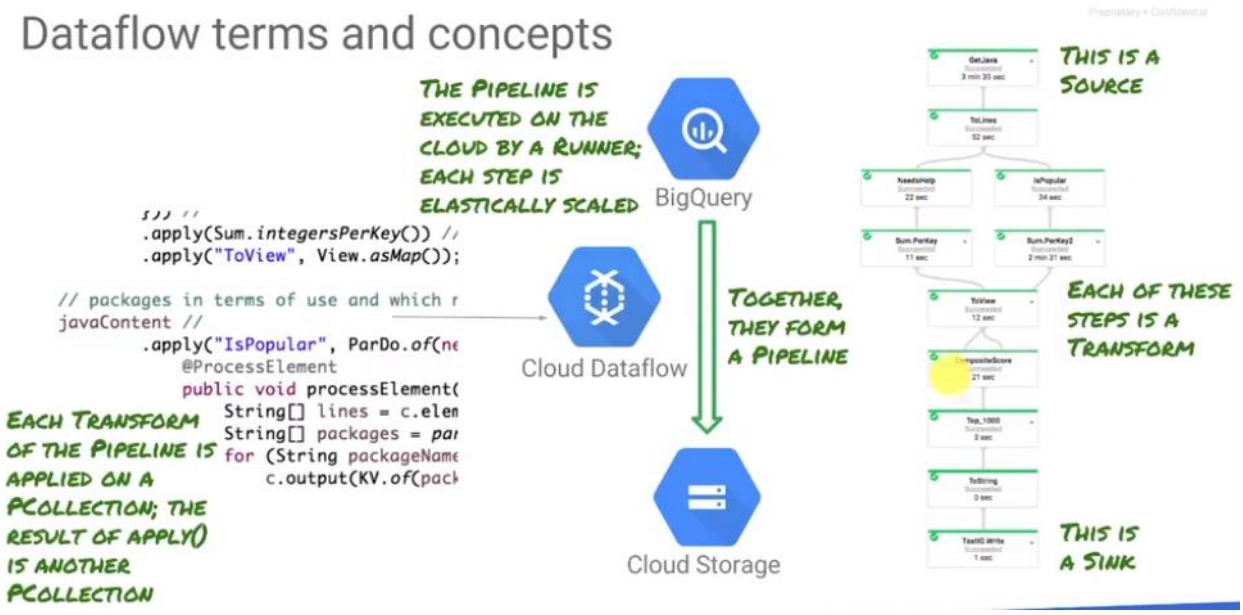
```
PCollection<Integer> sizes =
    lines.apply("Length", ParDo.of(new DoFn<String, Integer>() {
              @ProcessElement
              public void processElement(ProcessContext c) throws Exception {
                        String line = c.element();
                        c.output(line.length());
}}))
```

An anonymous do function inherits from a do function and implements a method called process element

with this annotation add process element to run in parallel. At this point then, for every line in my input, I have a corresponding integer, and the sizes now represents that parallel collection. A parallel collection doesn't need to be in-memory, it can be unbounded, because it may be streaming data that you're reading.

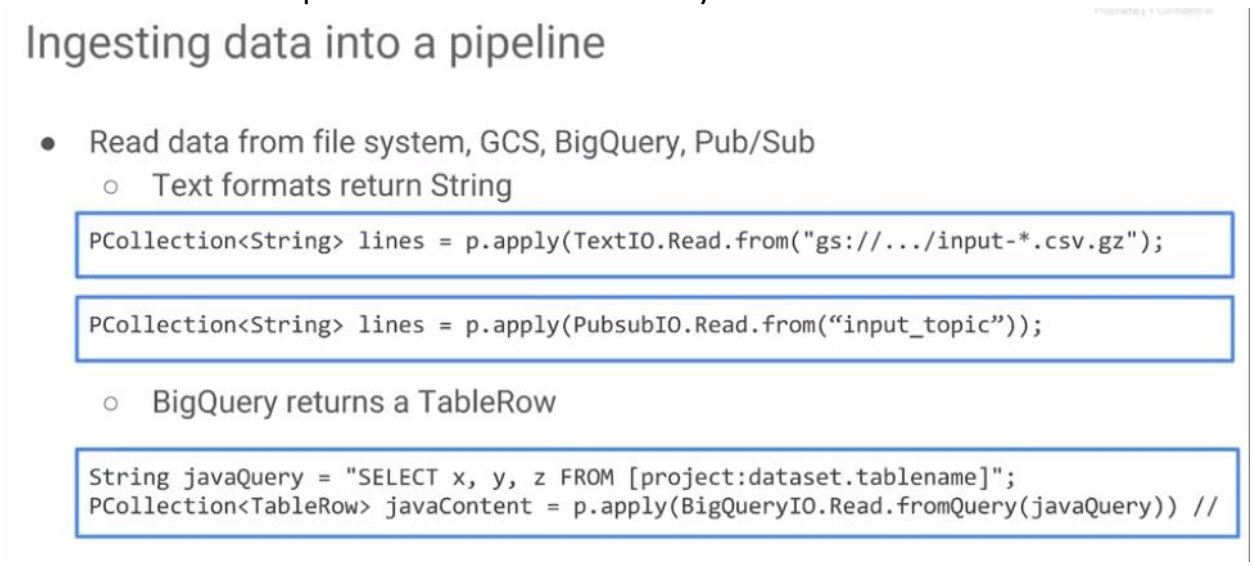


In Python, the code is very similar, the concepts are very similar. All of the data is represented again by a PCollection, 'P' for 'parallel'. It is not in memory collection, it can be unbounded, but the code is a lot simpler. So where we said 'line is p.apply', here we basically say 'p with a pipe symbol'. We want to apply a transform to those lines, we say lines, the pipe symbol again, and what we want to do is we want to do a map and for every line that comes in, lambda line, for every line that comes in, return the length of the line.



One of the neat things that dataflow will let you do is that you can have a running pipeline, and you can change the code and you can essentially replace the running pipeline. This is important because it means that when you replace a running pipeline, you don't lose any data. Any data processed by the old pipeline will get processed by the new pipeline. In order for that replacement to work, your transforms have to have names - unique names. So best practice, always, whenever you write a transform, give it a name that's unique within the pipeline.

Execute Data Pipelines in Java and Python



To ingest data into a pipeline, we need to read data. Data can be read from text, from big table, from BigQuery, from Pub/Sub - from a variety of different sources:

- Text - use TextIO.read from some file on cloud storage but it doesn't need to be a single file. You could have a wildcard in here. Read from a bunch of files on cloud storage, and this line now represents the PCollection of lines that are read from these files, whether it's one file or multiple files but it's a PCollection. What that means is that it's not all in memory at the same time but you will get to process all of these lines in your data pipeline.
- pubsubIO – in this case you're reading from an input topic.
- BigQuery – For a query use a BigQueryIO read which returns a PCollection of TableRows. Go to the table row and retrieve the value for a specific column.

Can write data out to same formats

- Write data to file system, GCS, BigQuery, Pub/Sub

```
lines.apply(TextIO.Write.to("/data/output").withSuffix(".txt"))
```

- Can prevent sharding of output (do only if it is small)

```
.apply(TextIO.Write.to("/data/output").withSuffix(".csv").withoutSharding())
```

- May have to transform PCollection<Integer>, etc. to PCollection<String> before writing out

Whatever you can read, you can also write. You can write into TextIO: TextIO.Write.to/data/output with a suffix. The thing to remember is that data flow is meant for big data, streaming data. When you write things out, they will be sharded – they will be split and written out into multiple files. What if you're just writing just a really small file, for example, a file of 100 numbers? It can be annoying to see the sharding syntax. 0 of 36, 1 of 36, etc. If you don't want to shard the output, that is, you want the output to explicitly go to the file that you set, /data/output.csv, then you can add the call withoutSharding. However, when using withoutSharding, you force all of the writing to happen on a single machine. This is not the recommended thing to do but if you have a really small file and you want to control the naming of this file in a very deterministic way, then you can use withoutSharding.

TextIO only writes out strings. If you have a PCollection of something else, e.g., integers, user defined objects, etc., then you must transform each of the elements in that PCollection to a string, so that you get back a PCollection of strings. It is the PCollection of strings on which you use a TextIO operation.

Executing pipeline (Java)

- Simply running `main()` runs pipeline locally

```
java -classpath ...        com...
```

```
mvn compile -e exec:java -Dexec.mainClass=$MAIN
```

- To run on cloud, submit job to Dataflow

```
mvn compile -e exec:java \
      -Dexec.mainClass=$MAIN \
      -Dexec.args="--project=$PROJECT \
      --stagingLocation=gs://$BUCKET/staging/ \
      --tempLocation=gs://$BUCKET/staging/ \
      --runner=DataflowRunner"
```

To run the pipeline, just run the main program. Typically, the pipeline is in a main program and you simply execute it: “java - <classpath>: provide all of the classes, classpath, all the jar files, and then the name of the class, where the name of the class, the fully qualified name, com.google.something. It can get very tiresome to keep typing in the classpath each time and the classpaths can be really very long. A better way to execute the program is to use Maven because Maven will also take care of downloading dependencies for you and managing them. Use Maven if you are using the Java version of data flow - execute the program with “mvn compile exec:java” where exec's main class is this fully qualified name, com.something.something. By default, this runs locally, the default runner is a local runner.

To execute on the cloud, you need to submit the job to cloud data flow on GCP. To do this, use the same Maven commands as before and add the following:

- specify a project. The reason you specify a project is because a project controls billing. Since data flow is going to launch compute nodes for you, we need to know where to send the bill to. So that's where the project is.
- In addition, data flow will have to take all of your code and stage it on the cloud storage bucket. Provide the staging location. You optionally may provide a temporary location. You can provide either the staging or the temporary location and you can do both. They can be different.
- Specify a runner. The runner is a DataflowRunner to indicate that you want to run it on the cloud.

Executing pipeline (Python)

- Simply running `main()` runs pipeline locally

```
python ./grep.py
```

- To run on cloud, specify cloud parameters

```
python ./grep.py \
      --project=$PROJECT \
      --job_name=myjob \
      --staging_location=gs://$BUCKET/staging/ \
      --temp_location=gs://$BUCKET/staging/ \
      --runner=DataflowRunner
```

To execute in Python, follow the same pattern. To run the main program, use “python grep.py”, where grep.py contains your main program that has your Python pipeline. To run it on the cloud, run it similar to Java: specify the project, the staging location, the runner etc. In Python you have to also specify a job name and the job name should be unique. You should not have submitted a job with that name before. So probably the best thing to do is add a timestamp to the name of the job.

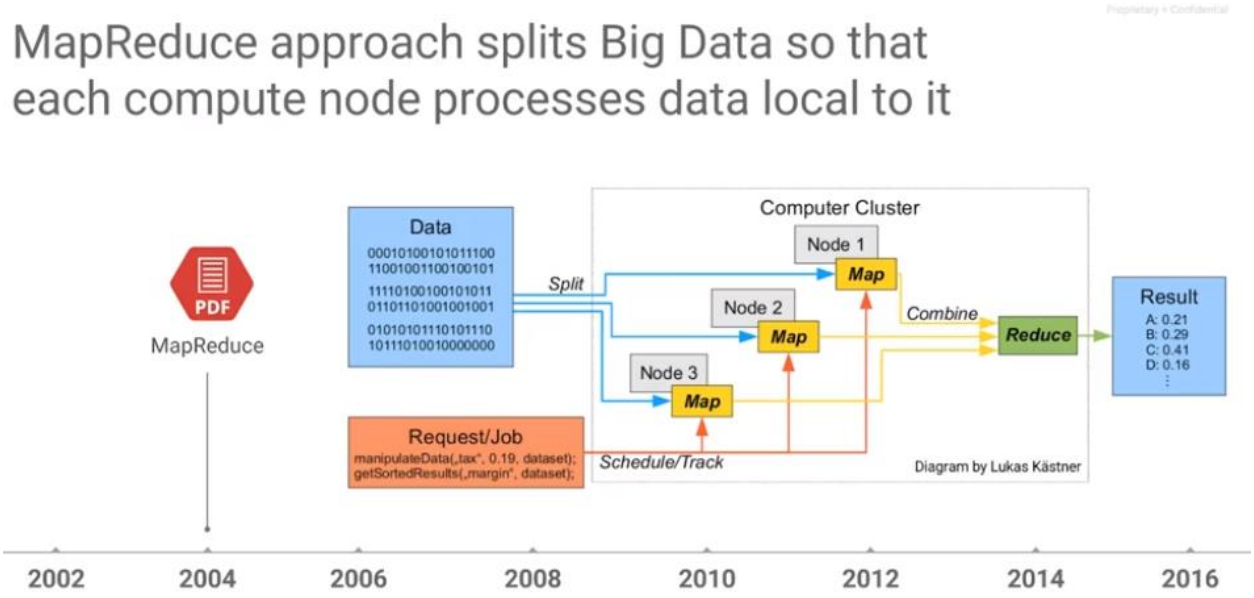
Lab: A Simple DataFlow Pipeline

Java: https://codelabs.developers.google.com/codelabs/cpb101-simple-dataflow/

Python: https://codelabs.developers.google.com/codelabs/cpb101-simple-dataflow-py/

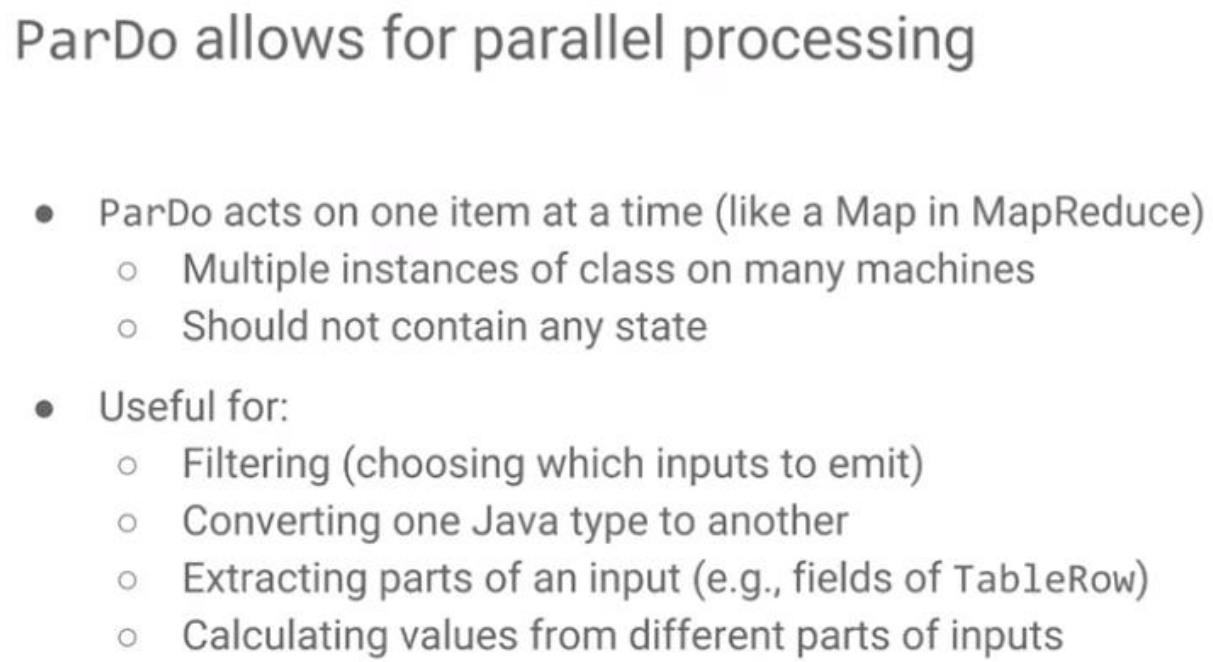
Review: https://www.coursera.org/learn/serverless-data-analysis-bigquery-cloud-dataflow-gcp/lecture/PMQ2Y/lab-4-review

MapReduce and Parallel Processing



Now that we've looked at how to write a basic pipeline in Dataflow, let's look at how to write a more useful pipeline. A common thing done with Big Data is to use MapReduce. If you want to process a very large data set, you break up the data set into pieces that each compute node processes as local data. For example, if we need to manipulate this data, we break it up such that this node carries out operations on a third of the data, this node carries out operations on another third of the data, and the first node carries out operations on the remainder of the data. The map operations occur in parallel on chunks of the original input data. The result of these maps is sent to one or more reduce nodes. There is a shuffle, if you have multiple reduce nodes, each of those reduce nodes processes one key or one set of keys. The output of the maps is sent to the appropriate reduce nodes, and each of the reduce nodes then calculates the aggregate, e.g., a count, sum, mean, on the values for that particular key and that is what gets written out.

The idea behind MapReduce is, to break it into two types of steps: (1) Steps that have to be done in parallel on all of the data. Those are your map operations. (2) Aggregations that have to be carried out on many roles at a time - those your reduce operations. Map operations are highly paralyzable. Reduce operations tend to be aggregates that you compute on many rows at a time. Usually all of the rows have a common key to them. For example, there may be tax data that we need to process. Take all of the tax returns that we have in our system and process each of those tax returns calculate the marginal tax rate on each of those returns. Then, we might emit out of the map node the corresponding state. Then in the reduce, the average tax rate per state could be emitted or calculated.



That is the basic idea behind a MapReduce approach. Take a large data set, break it into pieces, and process them. Everything that you do in MapReduce can be done in Dataflow. In Dataflow, the parallel processing, which is done by the Map operations in a MapReduce is carried out using a parallel do. A parallel do acts on one item at a time. In the tax form example, the transform inside the parallel do processes one tax return. There will be many instances of this class because we may have millions of tax returns that we need to process, which means there are many classes. Maybe you have 14 different instances of your tax processor, which is your Map operation: each of these Map operations needs to just process the data that it is given and emit the data out. It should not keep history. The map operations do not maintain state, you don't want your map operation to compute the average tax rate across users. Instead, you want the map operation to only calculate information from a single tax document. That is an example of a Parallel Do – the same a Map operation.

What do you use a Parallel Do for? You can use it, as we just talked about, to process one item at a time. You can also use it to filter. For example, if you get a tax return, and you emit the tax return only if that tax rate was greater than 10%, because we don't want to include students and juniors and teenagers etc. in our calculation. We may filter our data by the age of the customer or we might filter the data based on the full-time employment. That is what filtering is – choose which things that you want to emit for the next step of the processing pipeline.

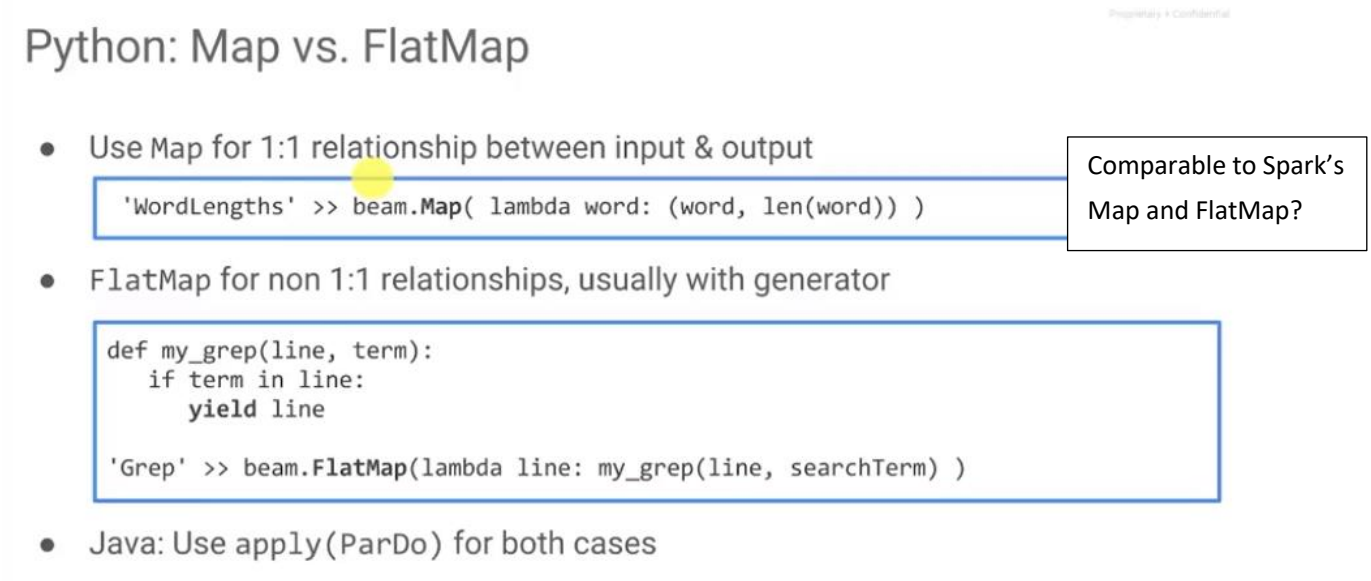
It can also be useful for convergence. For example, determine the average tax amount. It is a floating point number – but our output, for example, TextIO wants a string. The input may be a parallel collection of floats, the output will be a parallel collection of strings.

It can also be useful for extracting parts of your input. For example, for each tax return, extract the overall income of the person from their tax return – so you're extracting a part of the input. The input is a parallel collection of tax documents, the output is a parallel collection of floating-point numbers, where the floating point-number is the salary or the income of the person that's been identified in the tax document.

It may be used calculate values from different parts of your inputs. For example, for each tax document coming in, calculate the percent of income that this person paid in taxes. You might have two pieces of input that you're extracting from the tax document, you might be extracting their salary and you might be extracting the final tax amount and you're dividing one by the other and that is what you're emitting.

In conclusion, parallel do is useful as long as you're processing one item at a time. Recall that all of my examples here, whether I'm choosing which tax document to have it move on, whether I'm converting a float to a string, whether I'm extracting the salary from a tax document, whether I'm computing the average return from a tax document, all of those are on one tax document. That's a point of a parallel do, a Map operation, operates on one item at a time.

Transforms in Cloud DataFlow



Comparable to Spark's Map and FlatMap?

To do this in code, for example, in Python, use lambda: such that for every input, I emit something. For example, for every input word, I'm going to be omitting a tuple and the tuple is a word and the length of the word.

If you have a filtering relationship, you like will not have a return for every input. However, map assumes that for every input you have one output. When filtering, you choose whether to return or not. If that's the case, then you use a flat map in Python. For every input, if It is something that matches then yield. In Python, use flat maps with generators that do a yield. You use map for anything that's one on one. In Java, on the other

hand, it's always a parallel dual. You do a parallel dual with a do function, and the code looks like what we looked at earlier with our grep.

Apply Transform to PCollection

- Data in a pipeline are represented by `PCollection`
  - Supports parallel processing
  - Not an in-memory collection; can be unbounded

```
PCollection<String>  lines = p.apply(...)   //
```

  - Apply `Transform` to `PCollection`; returns `PCollection`

```
PCollection<Integer> sizes =
    lines.apply("Length", ParDo.of(new DoFn<String, Integer>() {
              @ProcessElement
              public void processElement(ProcessContext c) throws Exception {
                        String line = c.element();
                        c.output(line.length());
}}))
```

This is an example of code using a parallel do. In our example, this should be a parallel do with a do function where the input will be a tax document. And the output might be a float. The process element will be a tax document doc equals c.element. I have an object (tax document) and process on that object. The output (a float) is collected into a PCollection of floats. Once the map is complete (processed all tax documents and computed the average tax rate over all tax documents), then process the reduce (compute the average tax rate by state). Emit a key value pair of the state, the amount for the state.

Combine lets you aggregate

- Can be applied to a `PCollection` of values:

```
PCollection<Double> salesAmounts = ...;
PCollection<Double> totalAmount = salesAmounts.apply(
    Combine.globally(new Sum.SumDoubleFn()));
```

- And also to a grouped Key-Value pair:

```
PCollection<KV<String, Double>> salesRecords = ...;
PCollection<KV<String, Double>> totalSalesPerPerson =
    salesRecords.apply(Combine.<String, Double, Double>perKey(
      new Sum.SumDoubleFn()));
```

- Many built-in functions: `Sum`, `Mean`, etc.

This is a Pcollection of these key value pairs. Now we want to group them. We have an iterable of values to process. If you want to perform an aggregate function, e.g., min, max or sum, you have to iterate on all objects, you can directly apply aggregates to a PCollection of values. For example, we have a PCollection of doubles, and we want to basically find the total amount regardless of key. Apply the aggregate function (sum) globally to this PCollection.

GroupBy and Combine

Prefer Combine over GroupByKey

```
collection.apply(Count.perKey())
```

Is faster than:

```
collection
  .apply(GroupByKey.create())
  .apply(ParDo.of(new DoFn() {
     void processElement(ProcessContext c) {
       c.output(KV.of(c.element().getKey(), c.element().getValue().size()));
```

In Python the syntax is cleaner. A PCollection of tuples is indicated using parentheses. The first part of the tuple is a key. The second part of the tuple is a value. Group it by key gives me the grouped PCollections. Note that it is beam.Map and the reason that the packet is called is because this is Apache Beam. It's an open source API and Dataflow is the execution frame work for this API. Even on the Java side, the package names as you will see in our code, are all not beam. Beam packages, are going to use the package of beam.

Combine lets you aggregate

- Can be applied to a PCollection of values:

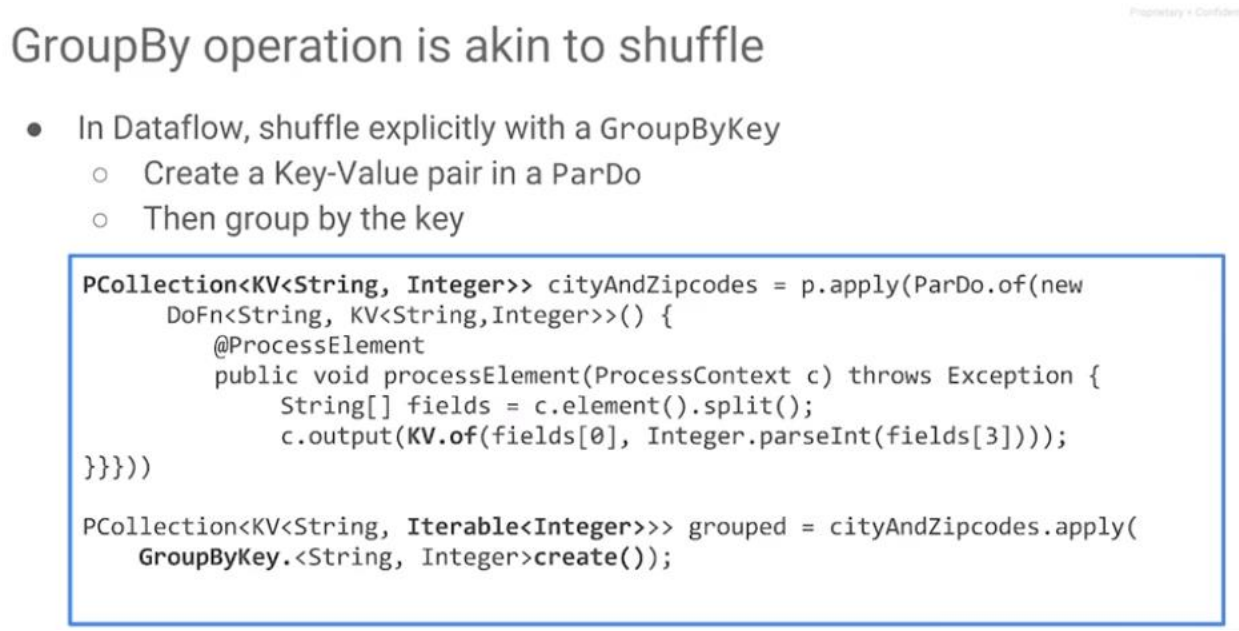
```
PCollection<Double> salesAmounts = ...;
PCollection<Double> totalAmount = salesAmounts.apply(
    Combine.globally(new Sum.SumDoubleFn()));
```

- And also to a grouped Key-Value pair:

```
PCollection<KV<String, Double>> salesRecords = ...;
PCollection<KV<String, Double>> totalSalesPerPerson =
    salesRecords.apply(Combine.<String, Double, Double>perKey(
      new Sum.SumDoubleFn()));
```

- Many built-in functions: Sum, Mean, etc.

If you have a PCollection of floating point numbers and you want to compute the sum, you would Combine.globally and calculate the sum. If you have the PCollection of tuples, then Combine.perKey and sum.



Python does not declare types, so for a PCollection, you have to remember the types, e.g., float when using Combine.glocally or for tuples, a Combine.perKey. In Java, declare types for the PCollection – e.g., key value pairs of strings and doubles.

Whatever you can do with a combine you could also do with a group by key explicitly. That is, you get an iterable and process on that, e.g., add them all up. That would be the same as doing combined per key and sum. Which one should you do? Do you do a combined per key, or do you do a GroupByKey and process every element?

Combine per key is going to be faster because it is processed in DataFlow at the server. The operations are optimized. Only use a combine operation if it is something special, that you don't have a direct implementation. In that case, it is necessary to resort to a group by key followed by explicitly processing the interval of values.

Lab: MapReduce in DataFlow

- Lab in Java: https://codelabs.developers.google.com/codelabs/cpb101-mapreduce-dataflow/
- Lab in Python: https://codelabs.developers.google.com/codelabs/cpb101-mapreduce-dataflow-py/

Review: https://www.coursera.org/learn/serverless-data-analysis-bigquery-cloud-dataflow-gcp/lecture/LbQtw/lab-5-review

Side Inputs and In-Memory Objects

So far, we've been assuming that everything that we're processing is in a PCollection. We read PCollection from a source and we process it and we write it to a sink.

Providing other inputs to a ParDo

- In-memory objects can be provided as usual:

```
public class Match extends DoFn<String, String> {
         private final String searchTerm;
         public Match(String searchTerm) {
              this.searchTerm = searchTerm;
         }
         @Override
         public void processElement(ProcessContext c) throws Exception {
              String line = c.element();
              if (line.contains(searchTerm)) {
                   c.output(line);
}}}
```

```
p.apply("Grep", ParDo.of(new Match(searchTerm))
```

In reality, you might have to read multiple pieces of data. You may need information in your transforms that is not just embedded in that element that you're processing. Now if you're processing a tax document, all the information that you need to process a single tax document may not be in that tax document itself. You may have some external piece of data that you need. How do you get external piece of data into a dataflow transform? One type of external data is an in-memory object which is immutable.

In-memory objects can be provided as a constructor parameter to your transform. For example, we perform a grep and the search term has to be set. When we create this match transform, we set the search term which is set as a field of my object. In my process element, I can determine whether the line contains that search term or not.

To apply the grep, perform a ParDo of (new Match), where Match is this transform passing the search term. The search term is immutable – it only needs to be known by this object and it is not something that is calculated dynamically.

There may be a second PCollection that you may need to provide – you may be reading from multiple sources. While processing the elements from one of those sources, you need the entirety of the data from the other source.

To pass in a PCollection...

- Convert the PCollection to a View (asList, asMap)

```
PCollection<KV<String, Integer>> cz = ...
PCollectionView<Map<String, Integer>> czmap = cz.apply("ToView", View.asMap());
```

- Call the ParDo with side input(s)

```
.apply("...", ParDo.withSideInputs(czmap) //
                        .of(new DoFn<KV<String, Integer>, KV<String, Double>>()
```

- Within ParDo, get the side input from the context

```
public void processElement(ProcessContext c) throws Exception {
     Integer fromcz = c.sideInput(czmap).get(czkey); // .get() because Map
```

To compose two or more sources, take the source that is smaller (typically) and convert it into a view. This view could be either a list or a map. It is a list if it is a PCollection of objects. It is a map if it's a PCollection of key value pairs. Take this PCollection of key value pairs and convert it to a map. This object can be provided as a side input. The word “side” is very evocative tells you exactly what it's doing: it is a side input, it's not the main input. The second PCollection is made into a view which is passed in as a side input. When you apply a ParDo with side inputs, it operates on both inputs.

After the apply inside the ParDo function or inside the process element, how do you get the secondary PCollection? Use ProcessContext: call c.sideInput to retrieve the side input. This is a map – use a key to retrieve a value. Inside this process element to get access to this PCollection, you have to do two things. You have to convert that PCollection to a PCollectionView and you have to take that view and pass it as a sideInput. Once it's a sideInput, from the ProcessContext, you can get that sideInput. And then you can use it just like a regular map or just like a regular list.

Lab: Side Inputs

Lab: https://codelabs.developers.google.com/codelabs/cpb101-bigquery-dataflow-sideinputs/

Review 1: https://www.coursera.org/learn/serverless-data-analysis-bigquery-cloud-dataflow-gcp/lecture/Wq09K/lab-6-review

Review 2: https://www.coursera.org/learn/serverless-data-analysis-bigquery-cloud-dataflow-gcp/lecture/FUbmX/lab-6-review-part-2

Streaming and Inputs

The pipeline that we just ran was a batch pipeline. It was run on a massive amount of data. It's runs in parallel but it was a batch pipeline. The Beam Programming Model is that the same code can be used on both batch and on streaming.

Can associate a timestamp with inputs

- Automatic timestamp when reading from PubSub
  - Timestamp is the time that message was published to topic

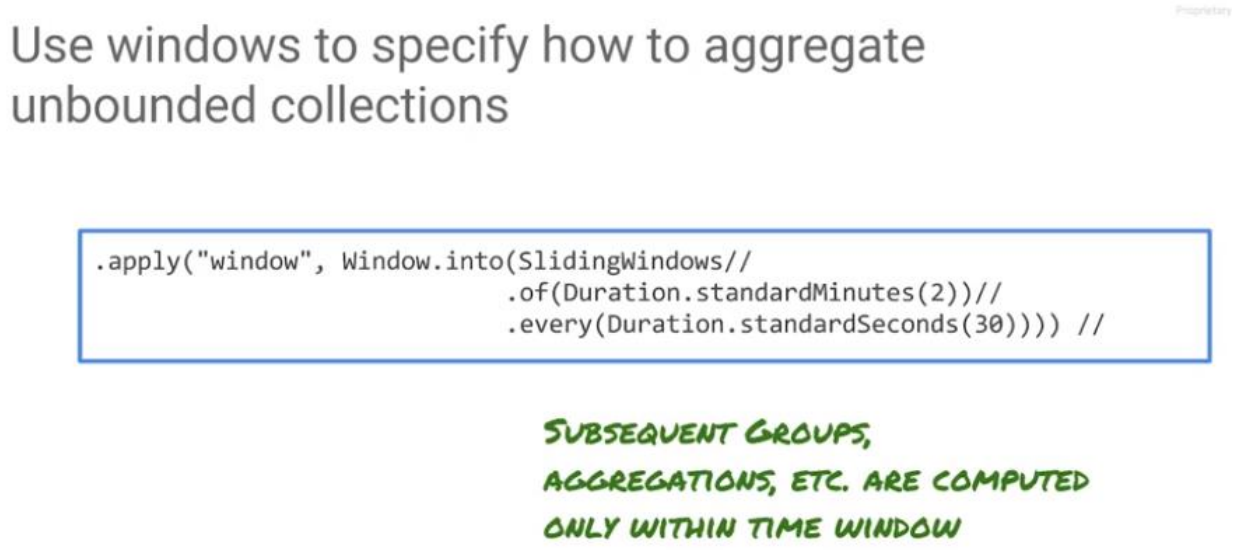
```
PCollection<String> lines = p.apply(PubsubIO.read().topic("input_topic"));
```

- For batch inputs, explicitly assign timestamp when emitting at some step in your pipeline:
  - Instead of `c.output()`

```
c.outputWithTimestamp(f, Instant.parse(fields[2]));
```

When you read any data from PubSub, you automatically get the time stamp associated with a message for the time it was inserted into PubSub. The timestamp for messages in stream processing takes into account the natural delays due to network latency, data flow accounts handles, late arriving records, out of order records, etc. The time stamp that PubSub has is from when the message is published into PubSub. It is a system time. There are times that you don't want to use the system time, but rather provide your own time which you can do. If we use the system time, in real time you're basically using the time that's in PubSub.

When writing batch pipelines, you read lines from a text file. Rather than doing a c.output when you read your data you do a c.output with time stamp and that lets you simulate the behavior of how your pipeline would behave if you had a time stamp associated with every message. If you are processing logs for example, the time stamp is part of the message itself, so you can parse the message, extract the time stamp and the next step you do a c.output with that time stamp, and then the rest of your code will basically do all of your processing as if it were real time, because it has a time stamp associated with the message.



You can perform aggregations in Windows using an apply. For a window of two minutes, an average is for the two-minute window. Any time I do a group by key, sums, counts, groups, aggregations are all carried out in the context of this window.

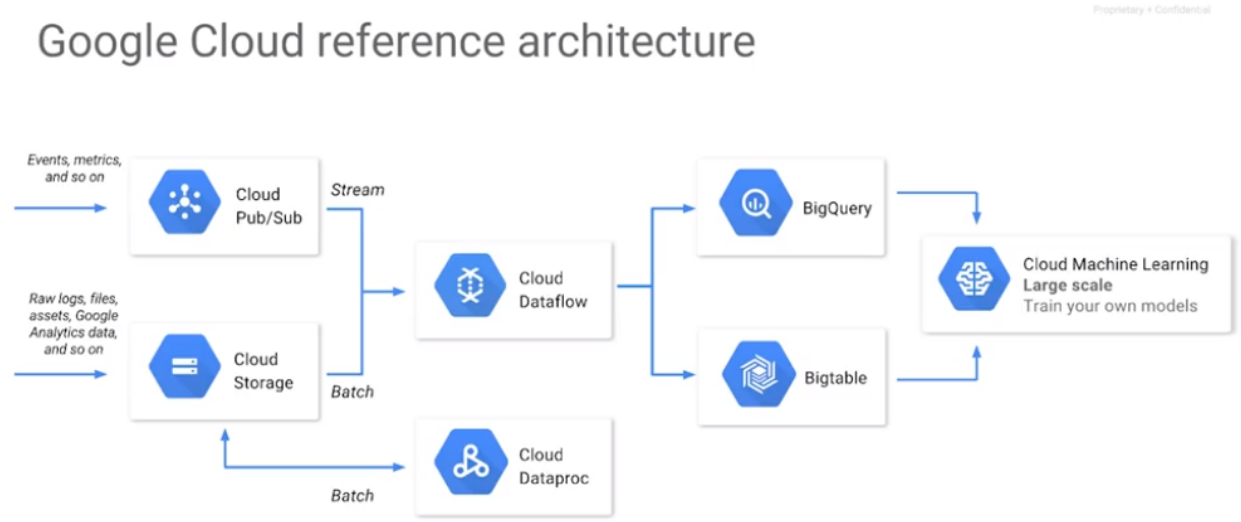
This is a very common way to handle real-time data. Data input from an IoT sensor, or log files, etc. is streamed into Pub/Sub. Pub/Sub provides a caching layer and autoscales. It can handle any throughput. Input data can be inserted into Pub/Sub and then processed by code in DataFlow. Dataflow ingests data from Pub/Sub, and it can stream them into BigQuery, which is your data warehouse. Then you can carry out SQL statements on the BigQuery warehouse even as the data are streaming in.

Lab: Streaming into BigQuery

https://codelabs.developers.google.com/codelabs/cpb101-bigquery-dataflow-streaming/

Review: https://www.coursera.org/learn/serverless-data-analysis-bigquery-cloud-dataflow-gcp/lecture/RfLG4/lab-7-review

Google Cloud Reference Architecture



Dataflow provides a way to process events, metrics, log files, etc., that you receive in real time through Pub/Sub. These can be streamed into BigQuery for analysis in real time. BigQuery allows us to use SQL queries on streaming data as it came in. Code that we wrote in Dataflow could also process batch data such as historical data with the same analyses. Those analyses may also be applied to real-time data. This idea that you can apply the same code that you apply on historical data to real-time data is extremely important for things like machine learning. Machine learning model are trained on historical data, but you apply your machine learning model to predict on real-time data that's coming in. It is important that all of the kinds of processing carried out on your historical data are also carried out on your real-time data. This is one of the reasons why this is a very common reference architecture on Google Cloud. Process real-time data the same way you process historical data using Dataflow. From Dataflow, you choose whether to stream it into BigQuery or into Bigtable depending on your throughput and latency considerations.

Resources

- Cloud Dataflow: http://cloud.google.com/dataflow/
- Which Java projects need help? https://medium.com/google-cloud/popular-java-projects-on-github-that-could-use-some-help-analyzed-using-bigquery-and-dataflow-dbd5753827f4#.t82wsxd2c
- Processing logs at scale using Cloud Dataflow: https://cloud.google.com/solutions/processing-logs-at-scale-using-dataflow.