Google Cloud

Serverless SQL data analysis

Data Engineering on Google Cloud Platform

Notes:

3.5 hours. 2 hours of lecture + 1.5 hours of lab, demo

Note that all the queries in this module are Standard SQL. If you are using the BigQuery console, you may need to uncheck the box that says “Legacy SQL”

Agenda

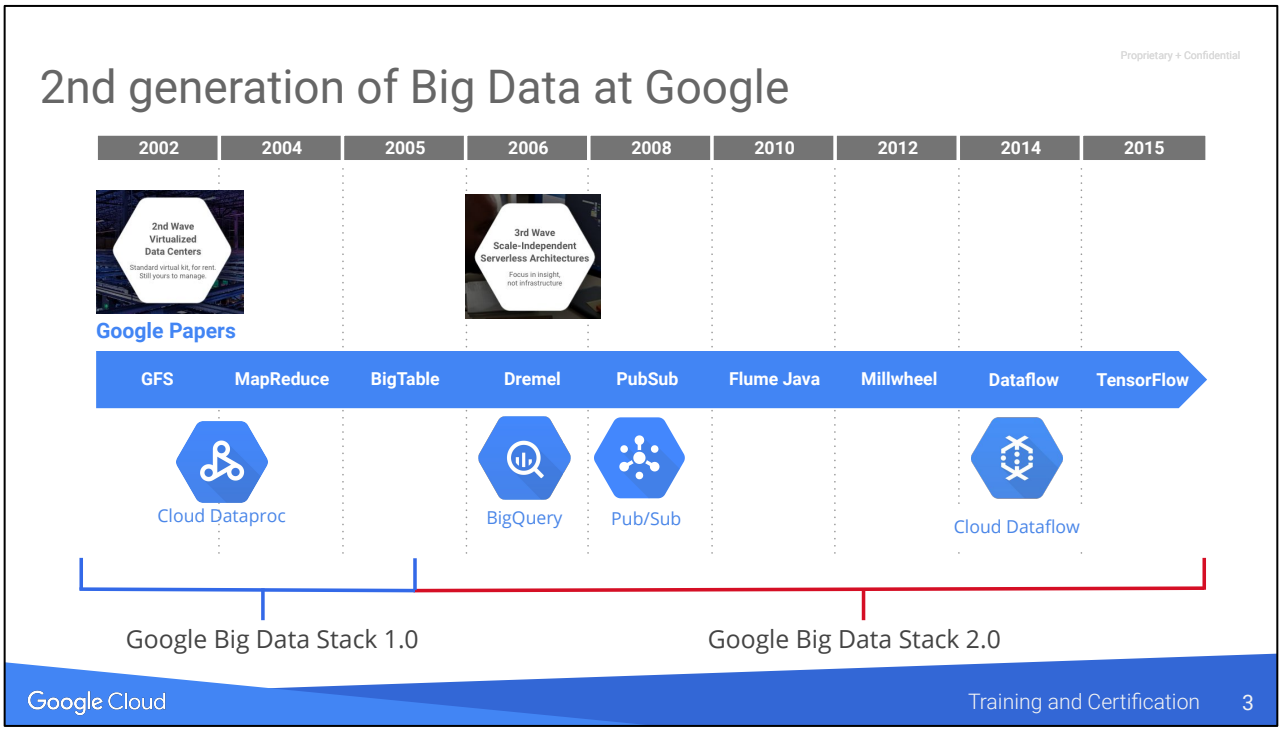
What is BigQuery?

Queries and functions + Lab

Load and export data + Lab

Advanced capabilities + Demo

Performance and pricing



Notes:

Papers on MapReduce and GFS led to open-source implementations Hadoop and HDFS. Dremel and Colossus are offered directly as BigQuery and Google Cloud Storage. For new projects, use 2nd generation products immediately.

Big Data Stack 1.0

In the early 2000's Google laid the foundation for Big Data Strategy –

- Design software with failure in mind
- Use only commodity components
- The cost of twice the amount of capacity should not be considerably more than the cost of twice the amount of hardware
- Be consistent

These principles inspired new computation architectures

- GFS – a distributed, cluster-based filesystem, GFS assumes that any disk can fail so data is stored in multiple locations
- MapReduce – A computing paradigm that divides problems into parallelized pieces across a cluster of machines
- Bigtable – Enables structured storage to scale out to multiple servers

Big Data Stack 2.0

2.0 refined the ideas from the 1.0 stack

- Dremel – A distributed SQL query engine that can perform complex queries over data stored on GFS, Colossus, and others – the basis of BigQuery
- Colossus – A distributed file system that resolves some of the limitations with GFS
- Megastore – A geographically replicated, consistent NoSQL-type data store that insures consistent reads and writes
- Spanner – Solves the problem of global time ordering in a geographically distributed system using atomic clocks

Query large datasets in seconds

```
#standardsql
SELECT
  departure_airport,
  count(1) as num_flights
FROM
  `bigquery-samples.airline_ontime_data.flights`
GROUP BY
   departure_airport
ORDER BY
  num_flights DESC
LIMIT 10
```

Row	departure_airport	num_flights
1	ATL	4184402
2	ORD	3610491
3	DFW	3121179
4	LAX	2300774
5	DEN	2212442
6	IAH	2006291
7	PHX	1999672
8	LAS	1698985

https://bigquery.cloud.google.com

Notes:

This is STANDARD SQL, so you have click on the appropriate tag.
Large here could be petabytes.

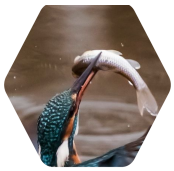
BigQuery offers...

1. Interactive analysis of petabyte scale databases
2. Familiar, SQL 2011 query language and functions
3. Many ways to ingest, transform, load, export data to/from BigQuery
4. Nested and repeated fields, user-defined functions in JavaScript
5. Data storage is inexpensive; queries charged on amount of data processed

Notes:

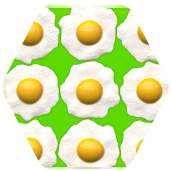
Note that this is a different slide than main course. The points here match the layout of this chapter.

BigQuery is a great choice because...



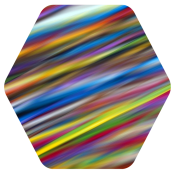
Near-real time analysis of massive datasets

No-ops; Pay for use



Durable (replicated), inexpensive storage

Immutable audit logs



Mashing up different datasets to derive insights

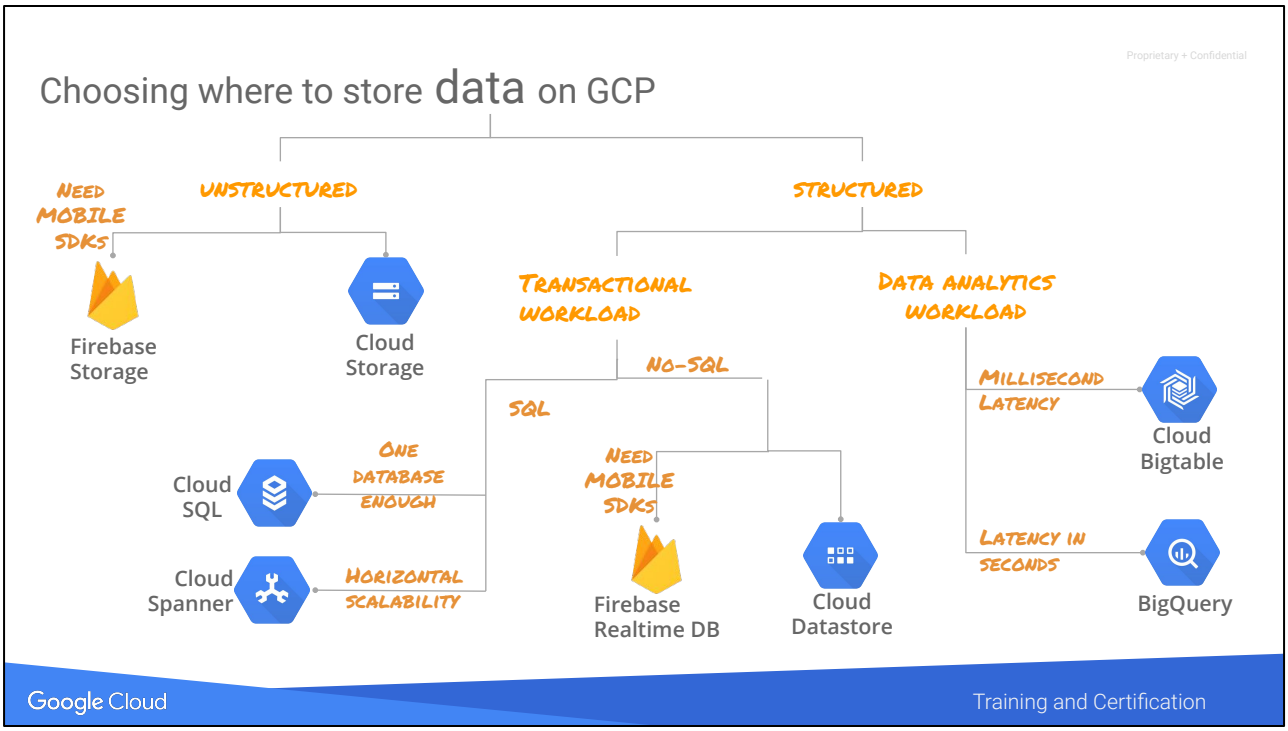
Notes:

BigQuery provides near-real time queries. If you use Hadoop-like systems, you may have to wait an entire weekend for something that you can do in real-time in BigQuery, and this can allow your business to be far more agile and nimble. Think about changing prices, ordering more supplies or buying options every hour in response to demand rather than once a week. BigQuery is one of the transformative Google technologies -- it's no-ops. There is no cluster or software to maintain. You submit a query and you pay for the compute nodes only for the duration of that query. You don't have to pay to keep a compute cluster up and running. Whenever you ingest data into BigQuery, it's auto-replicated, and you get that reliability built-in to the cost of the storage. It's on the order of Nearline storage (i.e., less than standard cloud storage costs!). Google Cloud Audit Logs track Admin Activity and Data Access. These Immutable logs can tell you "who did what, where, and when?" in BigQuery.

Finally, one of the really transformational things is that because it is completely no-ops and a common query format, you can use BigQuery as the way to collaborate more within your company, to tie together datasets from across sales, catalog, warehouse, etc. into a common analysis framework

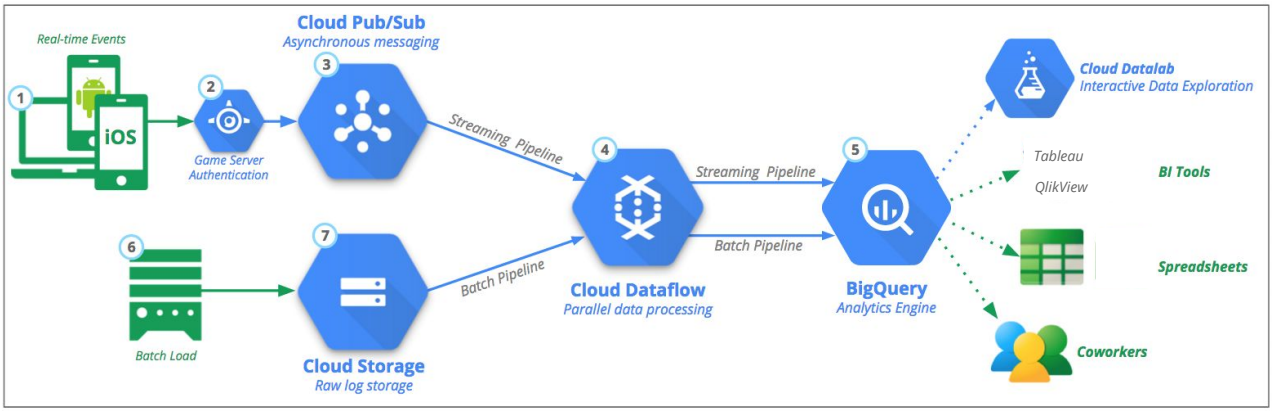
Image credits:

https://pixabay.com/en/kingfisher-bird-alcedo-atthis-1068480/ (cc0)
https://pixabay.com/en/rotterdam-cycling-bicycle-rental-1199442/ (cc0)
https://pixabay.com/en/egg-food-many-duplicate-easter-85641/ (cc0)
https://pixabay.com/en/female-diary-write-beautiful-865110/ (cc0)
https://pixabay.com/en/abstract-blurred-blur-color-mix-859315/ (cc0)



Positioning BigQuery in flow-chart of choices: structured data, primarily for analytics, latency of seconds okay
We'll look at Bigtable and Spanner later in the course, when we talk about streaming.
Cloud SQL was covered in Fundamentals course.
Firebase and Datastore are covered in the App Dev course & App Engine content
Horizontal scalability == multiple databases, even globally distributed, with consistency

Example architecture for data analytics



Build a mobile gaming analytics platform - a reference architecture

Notes:

Popular mobile games can attract millions of players and generate terabytes of game-related data in a short burst of time. This places extraordinary pressure on the infrastructure powering these games and requires scalable data analytics services to provide timely, actionable insights in a cost-effective way.

To address these needs, a growing number of successful gaming companies use Google's web-scale analytics services to create personalized experiences for their players. They use telemetry and smart instrumentation to gain insight into how players engage with the game and to answer questions like: At what game level are players stuck? What virtual goods did they buy? And what's the best way to tailor the game to appeal to both casual and hardcore players?

A new reference architecture describes how you can collect, archive and analyze vast amounts of gaming telemetry data using Google Cloud Platform's data analytics products. The architecture demonstrates two patterns for analyzing mobile game events:

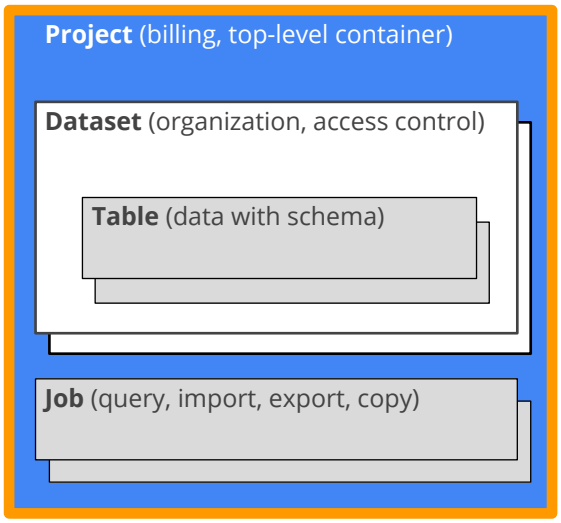
- Batch processing: This pattern helps you process game logs and other large files in a fast, parallelized manner. For example, leading mobile

- gaming company DeNA moved to BigQuery from Hadoop to get faster query responses for their log file analytics pipeline.
- Real-time processing: Use this pattern when you want to understand what's happening in the game right now. Cloud Pub/Sub and Cloud Dataflow provide a fully managed way to perform a number of data-processing tasks like data cleansing and fraud detection in real-time. For example, you can highlight a player with maximum hit-points outside the valid range. Real-time processing is also a great way to continuously update dashboards of key game metrics, like how many active users are currently logged in or which in-game items are most popular.

Some Cloud Dataflow features are especially useful in a mobile context since messages may be delayed from the source due to mobile Internet connection issues or batteries running out. Cloud Dataflow's built-in session windowing functionality and triggers aggregate events based on the actual time they occurred (event time) as opposed to the time they're processed so that you can still group events together by user session even if there's a delay from the source.

But why choose between one or the other pattern? A key benefit of this architecture is that you can write your data pipeline processing once and execute it in either batch or streaming mode without modifying your codebase. So if you start processing your logs in batch mode, you can easily move to real-time processing in the future. This is an advantage of the high-level Cloud Dataflow model that was released as open source by Google.
Cloud Dataflow loads the processed data into one or more BigQuery tables. BigQuery is built for very large scale, and allows you to run aggregation queries against petabyte-scale datasets with fast response times. This is great for interactive analysis and data exploration, like the example screenshot above, where a simple BigQuery SQL query dynamically creates a Daily Active Users (DAU) graph using Google Cloud Datalab.

Project contains users, datasets



A project contains users and datasets

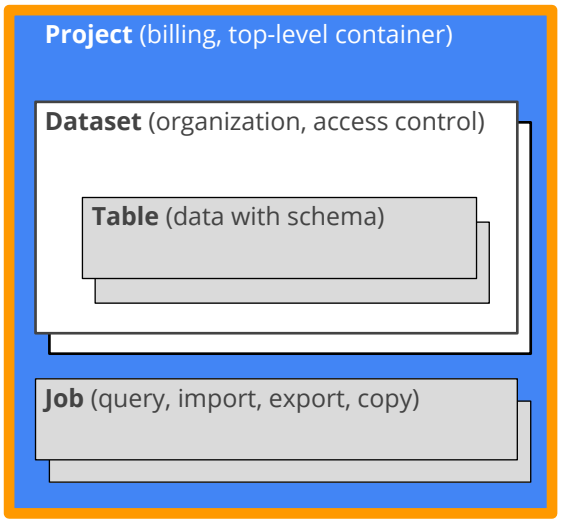
Use Project to:

- Limit access to datasets and jobs
- Manage billing

Notes:

Datasets are owned by projects, which control billing and serve as a global namespace root - all of the object names in BigQuery are relative to the project.

Access control is via the dataset



A project contains users and datasets

Use Project to:

- Limit access to datasets and jobs
- Manage billing

A dataset contains tables and views

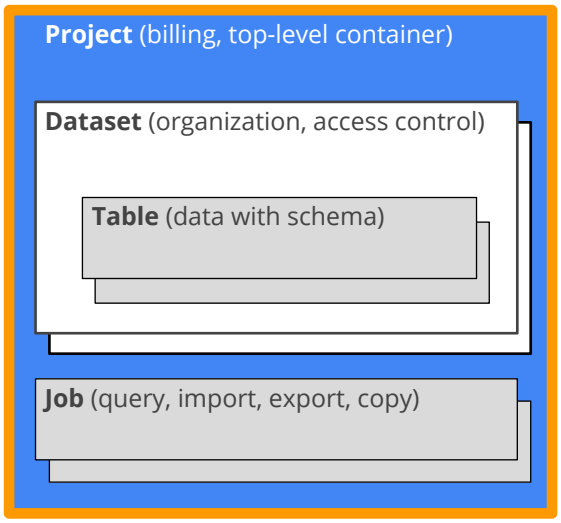
- Access Control Lists for Reader/Writer/Owner
- Applied to all tables/views in dataset

Notes:

BigQuery currently does not manage access to individual tables or views within the dataset. You can implement access control to a table through the use of views which would reside in a separate dataset.

A dataset's ACL can include users who are not in project -- this would be the case, for example, for public datasets

Tables and jobs



A project contains users and datasets

Use Project to:

- Limit access to datasets and jobs
- Manage billing

A dataset contains tables and views

- Access Control Lists for Reader/Writer/Owner
- Applied to all tables/views in dataset

A table is a collection of columns

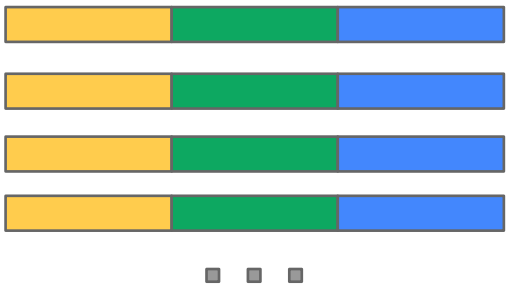
- Columnar storage
- Views are virtual tables defined by SQL query
- Tables can be external (e.g., on Cloud Storage)

A job is a potentially long-running action

- Can be canceled

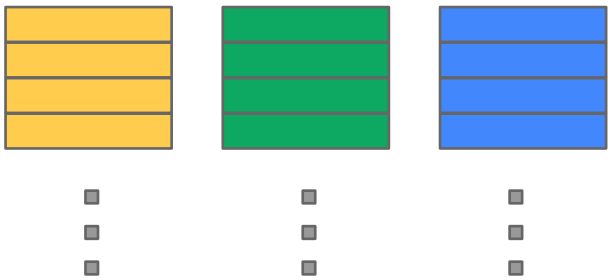
BigQuery storage is columnar

Relational database



Record-oriented storage
Supports transactional updates

BigQuery Storage



Each column in separate, compressed, encrypted file that is replicated 3+ times
No indexes, keys or partitions required
For immutable, massive datasets

Notes:

BigQuery stores data in columns.

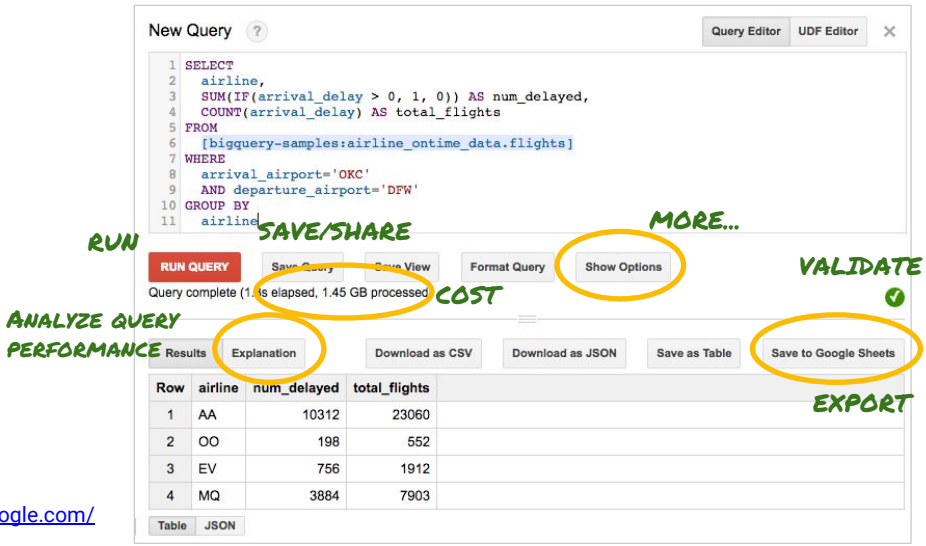
Most queries only work on a small number of fields and BigQuery only needs to read those relevant columns to execute a query. Since each column has data of same type, BigQuery could compress the column data much more effectively.

You can stream (append data) easily to BigQuery tables, but not change existing values.
Replicating the data 3+ times also helps in finding optimal compute nodes to do filtering, mixing, etc.

Agenda

Queries and functions + Lab

Running a query from web console



https://bigquery.cloud.google.com/

Notes:

Everything you can do with the web console can also be done with a Python client.

Options include a destination BigQuery table, whether to cache, etc.

You will get to look at this in the lab, but if you want to copy-paste and try the query, see next slide.

Query syntax is SQL 2011 + extensions

BUILT-IN FUNCTIONS: SUM, IF, COUNT

<PROJECT>.<DATASET>.<TABLE>

CLAUSE, BOOLEAN OPERATIONS

GROUP BY

```
SELECT  SQL-LIKE SYNTAX
   airline,
   SUM(IF(arrival_delay > 0, 1, 0)) AS num_delayed,
   COUNT(arrival_delay) AS total_flights

FROM
  `bigquery-samples.airline_ontime_data.flights`

WHERE
  arrival_airport='OKC'
  AND departure_airport='DFW'

GROUP BY
  airline;
```

NAMED COLUMNS

Query Results

Row	airline	num_delayed	total_flights
1	AA	10312	23060
2	OO	198	552
3	EV	756	1912
4	MQ	3884	7903

https://cloud.google.com/bigquery/query-reference

Notes:

Spend some time explaining this query.

The extensions have to do with nested fields.

Aggregate functions and GROUP BY

GROUP BY FIELDS

SUM, COUNT ARE AGGREGATE FUNCTIONS

```
SELECT
   airline,
   departure_airport,
   SUM(IF(arrival_delay > 0, 1, 0)) AS num_delayed,
   COUNT(arrival_delay) AS total_flights
FROM
  `bigquery-samples.airline_ontime_data.flights`

WHERE
  arrival_airport='OKC'

GROUP BY
  airline, departure_airport;
```

Row	airline	departure_airport	num_delayed	total_flights
1	OH	MCO	33	76
2	XE	SAN	317	759
3	XE	EWR	1985	3698
4	WN	DAL	9117	19555
5	NW	MSP	17	35

GROUP BY RETURNS (FIELDS)->BAG
SELECT FIELDS OR CALL AGGREGATE FUNCTIONS OVER THE BAG

Notes:

Slightly different from previous slide, since departure_airport is part of aggregation.
We can not have arrival_delay in SELECT clause because it is neither a GROUP_BY field not is it an aggregate function. Aggregate of arrival_delay is okay.

Subqueries can do everything a query can

SELECT FROM RESULT OF NESTED SELECT

(NESTED QUERY IS SAME AS PREVIOUS SLIDE)

WHERE CLAUSE AND ORDER ON RESULT OF NESTED SELECT

```
SELECT
   airline, departure_airport, num_delayed,
   total_flights, num_delayed/total_flights AS delayed_frac
FROM

# Nested Query
(SELECT
  airline, departure_airport,
  SUM(IF(arrival_delay > 0, 1, 0)) AS num_delayed,
  COUNT(arrival_delay) AS total_flights
FROM
  `bigquery-samples.airline_ontime_data.flights`
WHERE
  arrival_airport='OKC'
GROUP BY
  airline, departure_airport)

WHERE total_flights > 5
ORDER BY delayed_frac DESC
LIMIT 5;
```

Row	airline	departure_airport	num_delayed	total_flights
1	OH	MCO	33	76
2	XE	SAN	317	759
3	XE	EWR	1985	3698
4	WN	DAL	9117	19555
5	NW	MSP	17	35

Row	airline	departure_airport	num_delayed	total_flights	delayed_frac
1	OO	ATL	260	360	0.7222222222222222
2	OH	ATL	251	373	0.6729222520107239
3	EV	EWR	191	303	0.6303630363036303
4	OH	DTW	80	127	0.6299212598425197
5	OH	MSP	191	317	0.6025236593059937

Notes:

Num_delayed, total_flights come out of nestedquery results.

The nested query goes in parentheses.

More efficient than a join.

Can query from multiple tables using UNIONS

PULL DATA FROM MULTIPLE TABLES WITH UNION ALL

USE COLUMN INDEXES IN GROUP BY AND ORDERING

```
# Find the 5 hottest NOAA stations since 2015
SELECT
  # Find the average temperature
  ROUND(AVG(temp),2) AS temp_avg_f,
  stn AS noaa_station_number
FROM
  (
  SELECT * FROM `bigquery-public-data.noaa_gsod.gsod2015` UNION ALL
  SELECT * FROM `bigquery-public-data.noaa_gsod.gsod2016` UNION ALL
  SELECT * FROM `bigquery-public-data.noaa_gsod.gsod2017`
  )
GROUP BY 2
ORDER BY 1 DESC
LIMIT 5;
```

temp_avg_f	noaa_station_number
101.4	615100
98.4	626400
96.73	651670
95.0	416360
93.9	416380

Notes:

Standard SQL in BQ supports the use of UNION and UNION ALL (the ‘all’ just means include duplicate records across tables)

Since we’re doing a UNION, we need to actually bring together multiple queries (hence the SELECT *) and not just the table names

Ask: Once, 2017 is over, how would we add table data from 2018? Yes we could continue to hardcode but if we had a standard structure for **dated tables** like you see here in gsod**YYYY** we can also do somthing more…. [slide transition]

Can query from multiple tables with Wildcards

Use the table name wildcard *

And filter tables in your where clause

```
# Find the 5 hottest NOAA stations since 2015
SELECT
  # Find the average temperature
  ROUND(AVG(temp),2) AS temp_avg_f,
  stn AS noaa_station_number
FROM
  `bigquery-public-data.noaa_gsod.gsod*`
WHERE
   _TABLE_SUFFIX >= '2015'
GROUP BY 2
ORDER BY 1 DESC
LIMIT 5;
```

temp_avg_f	noaa_station_number
101.4	615100
98.4	626400
96.73	651670
95.0	416360
93.9	416380

Notes:

In BQ, we can add a wildcard to the table name and filter on all matching data tables within our project in the WHERE clause using _TABLE_SUFFIX

Standard SQL operator matching can be used (<, >, <>, IN, NOT IN, AND, OR etc..). Now we won't need to update our query when 2018 data is loaded.

JOIN ON fields across tables

Casting, String functions

Fields on the two tables (INNER JOIN by default)

```
SELECT
  f.airline,
  SUM(IF(f.arrival_delay > 0, 1, 0)) AS num_delayed,
  COUNT(f.arrival_delay) AS total_flights
FROM
  `bigquery-samples.airline_ontime_data.flights` AS f
JOIN (
  SELECT
    # Convert date fields to YYYY-MM-DD
    CONCAT(CAST(year AS STRING), '-', LPAD(CAST(month AS STRING),2,'0'), '-',
LPAD(CAST(day AS STRING),2,'0')) AS rainyday
  FROM
    `bigquery-samples.weather_geo.gsod`
  WHERE
    station_number = 725030
    # Only return days it rained
    AND total_precipitation > 0) AS w
ON
  w.rainyday = f.date
WHERE f.arrival_airport = 'LGA'
GROUP BY f.airline;
```

Inner SELECT returns days that it rained at LaGuardia Airport (station 725030)

Notes:

725030 happens to be LaGuardia. I ran a separate query on stations table to find this ...

To understand this query, use the query editor and copy-paste the inner query first.

Then, run the outerquery without the JOIN ON
Add AND f.date = '2008-05-03' to the WHERE clause to do it on a single day

This query essentially combines the above 3 ideas and is what they will do in the lab.

```
SELECT
  f.airline,
  SUM(IF(f.arrival_delay > 0, 1, 0)) AS num_delayed,
  COUNT(f.arrival_delay) AS total_flights
FROM
  [bigquery-samples:airline_ontime_data.flights] AS f
JOIN (
```

```
  SELECT
    CONCAT(STRING(year), '-', LPAD(STRING(month), 2, '0'), '-',
LPAD(STRING(day), 2, '0')) AS norainday
  FROM
    [bigquery-samples:weather_geo.gsod]
  WHERE
    station_number = 725030
    AND total_precipitation = 0) AS w
ON
  w. norainday = f.date
WHERE f.arrival_airport = 'LGA'
GROUP BY f.airline
```

Lab 1: Build a BigQuery Query

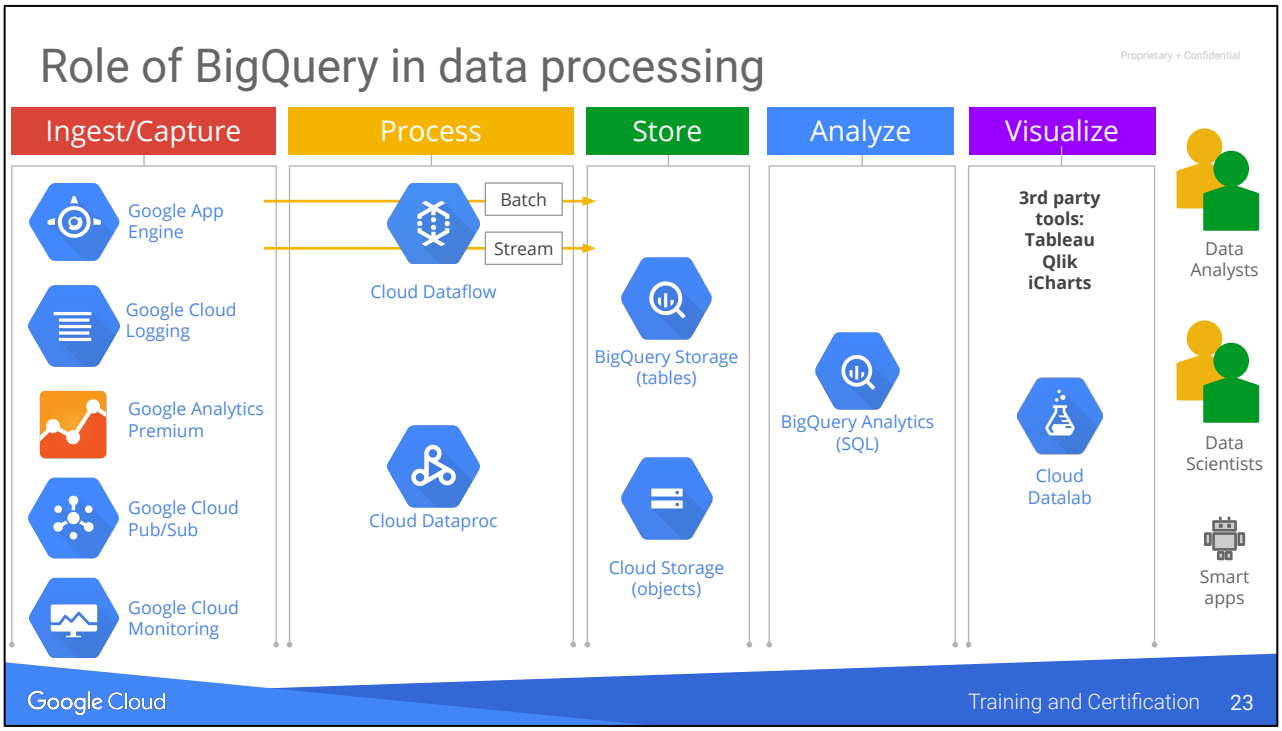
- Build up a complex BigQuery using clauses, inner selects, built-in functions, and joins
- Let's find the fraction of flights that depart late from New York's LaGuardia Airport on rainy days

Notes:

Image: https://pixabay.com/en/rain-drops-wet-rainy-airplane-926765/ (cc0)

Agenda

Load and export data + Lab



Notes:

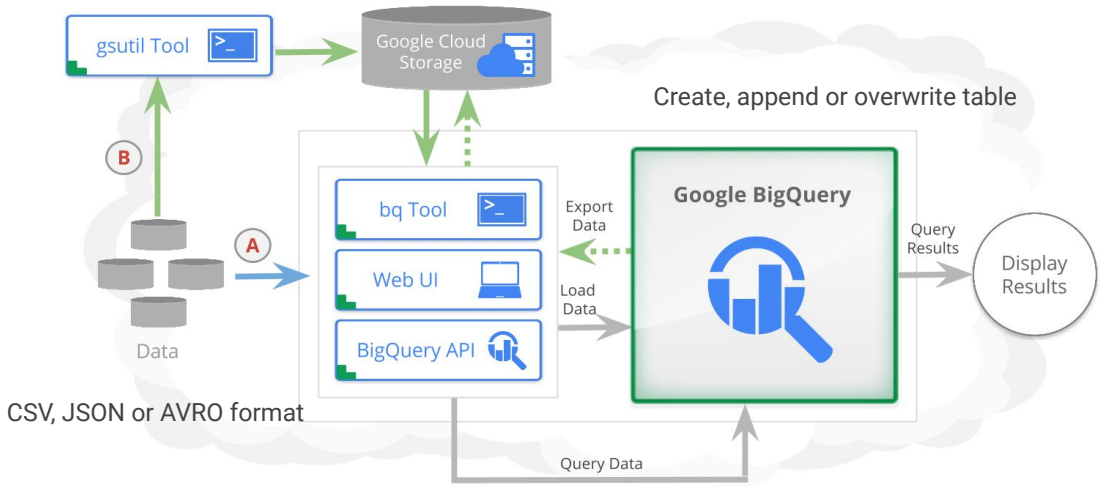
BigQuery is an inexpensive data store for tabular data. It is cost-comparable with Cloud Storage, so it makes sense to ingest into BigQuery and leave the data there.

Ingesting the data depends on where it is coming from. Cloud logs, GAP can directly ingested into BQ. From pub/sub, you have an API. In the most general case, you can use Dataflow and write code to ingest the data in batch/stream. You could also use OSS tools like Spark or Hadoop to do the processing, in which case you'd use Dataproc.

Analysis itself is done by BigQuery. The results can be visualized in a iPython notebook (Datalab) or in 3rd party tools.

So, BigQuery's role is in both storage and analysis. In other words, it is a data warehousing solution.

Load data using CLI, web UI, or API



Notes:

BigQuery CLI:

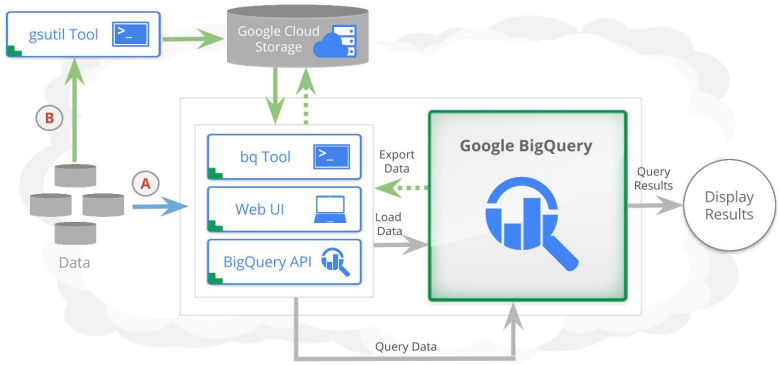
- Good for uploading large data files, scheduling data file uploads
- Create table, define schema, and load data with one command
- Can also run queries from the command line
  - Interactively or batch mode queries
  - Automating scripts using scripting language
- Run the bq command-line tool from:
  - A Compute Engine instance
  - Cloud Shell
  - Client machine (requires installing the Google Cloud SDK)

Syntax for loading data via CLI:
bq load [--source_format=NEWLINE_DELIMITED_JSON|CSV] destination_table data_source_uri table_schema

More details: https://developers.google.com/bigquery/bq-command-line-tool

Lab 2: Loading and Exporting Data

- Load and export data to/from BigQuery using bq tool and using the web UI



Agenda

Advanced capabilities + Demo

BigQuery Data Types — Standard SQL

Data type	Possible value
`STRING`	Variable-length character (Unicode) data
`INT64`	64-bit integer
`FLOAT64`	Double-precision (approximate) decimal values
`BOOL`	True or false (case insensitive)
`ARRAY`	Ordered list of zero or more elements of any non-ARRAY type
`STRUCT`	Container of ordered fields each with a type (required) and field name (optional)
`TIMESTAMP`	Represents an absolute point in time, with precision up to microseconds. Values range between the years 1 and 9999, inclusive

Notes:

For more information on all data types supported by standard SQL, see: https://cloud.google.com/bigquery/sql-reference/data-types.

WITH, COUNT(DISTINCT)

WITH clause defines a subquery that joins `stations` with `gsod2015` and filters by state = 'WA'

```
WITH WashingtonStations AS (
  SELECT weather.stn AS station_id, ANY_VALUE(station.name) AS name
  FROM `bigquery-public-data.noaa_gsod.stations` AS station
  INNER JOIN `bigquery-public-data.noaa_gsod.gsod2015` AS weather
  ON station.usaf = weather.stn
  WHERE station.state = 'WA' AND station.usaf != '999999'
  GROUP BY station_id
)
SELECT washington_stations.name,
  (SELECT COUNT(DISTINCT CONCAT(year, mo, da))
   FROM `bigquery-public-data.noaa_gsod.gsod2015` AS weather
   WHERE washington_stations.station_id = weather.stn
   AND prcp > 0 AND prcp < 99) AS rainy_days
FROM WashingtonStations AS washington_stations
ORDER BY rainy_days DESC;
```

Subquery: Counts the number of rainy days at a station even if a station has repeated records for some day

Using count(distinct concat(year, mo, da)) instead of simply count(*) helps with the case that there are repeated records for the same day. A station might have repeated records in the case of streaming updates, etc. This particular table has been cleaned up so that it doesn't, but the slide shows you how to get around such repetition with a COUNT(DISTINCT)

COUNT(DISTINCT) example result

Results | Explanation | Job Information

Row	name	rainy_days
1	QUILLAYUTE AIRPORT	179
2	BELLINGHAM INTL AIRPORT	177
3	BOWERMAN AIRPORT	165
4	GRAY AFF AIRPORT	164
5	SANDERSON FIELD AIRPORT	162
6	SNOHOMISH CO	160
7	BOEING FLD/KING CO INTL AP	160

Normalize for efficiency

Original data

ID	Name	Transaction		
1	Bob			
2	Doug			
		OrderID	Date	Quantity
		11537	10/29/2018	11
		78244	11/05/2018	14
		32367	09/09/2018	18
3	Tom			
		OrderID	Date	Quantity
		13323	04/09/2018	6
		45452	11/10/2018	13
		17671	08/20/2018	15

Normalized data

Customer Table

ID	Name
1	Bob
2	Doug
3	Tom

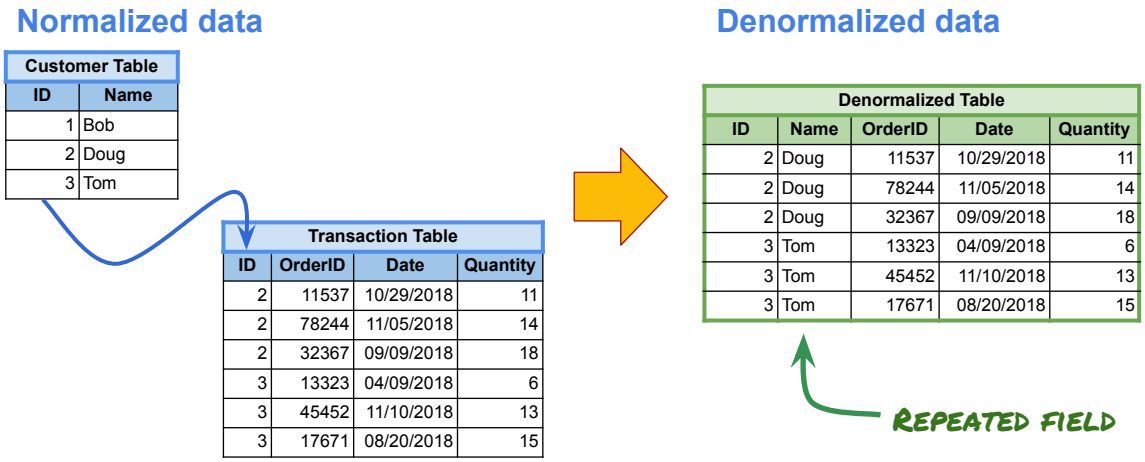
Transaction Table

ID	OrderID	Date	Quantity
2	11537	10/29/2018	11
2	78244	11/05/2018	14
2	32367	09/09/2018	18
3	13323	04/09/2018	6
3	45452	11/10/2018	13
3	17671	08/20/2018	15

The original data is organized visually, but if you had to write an algorithm to process the data, how might you approach it? Could be by rows, by columns, by rows-then-fields. And the different approaches would perform differently based on the query. Also, your method might not be parallelizable.

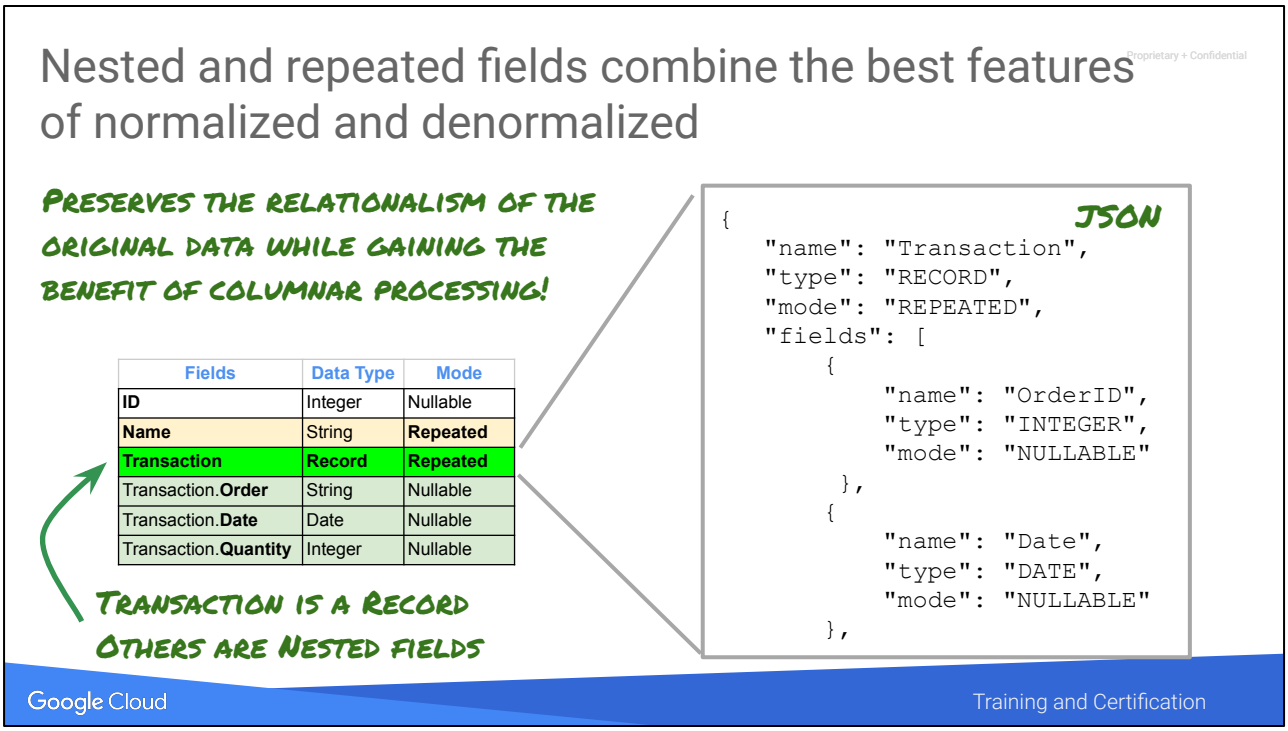
The original data can be interpreted and stored in many ways in a database. Normalizing the data means turning it into a relational system. This stores the data efficiently and makes query processing a clear and direct task. Normalizing increases the orderliness of the data.

Denormalize for performance



Denormalizing is the strategy of accepting repeated fields in the data to gain processing performance.

Data must first be normalized before it can be denormalized. Denormalization is another increase in the orderliness of the data. Because of the repeated fields (in the example, the Name field is repeated), the denormalized form takes more storage. However, because it is no longer relational, queries can be processed more efficiently and in parallel using columnar processing.



Nested columns can be understood as a form of repeated field. It preserves the relationalism of the original data and schema while enabling columnar and parallel processing of the repeated nested fields. Nested and repeated fields helps BigQuery more easily interact with existing databases, enabling easier transitions to BigQuery and hybrid solutions where BigQuery is used in conjunction with traditional databases.

Article on "Why Nesting Is So Cool"
https://looker.com/blog/why-nesting-is-so-cool

ARRAY/STRUCT example

```
# Top two Hacker News articles by day
WITH TitlesAndScores AS (
  SELECT
    ARRAY_AGG(STRUCT(title, score)) AS titles,
    EXTRACT(DATE FROM time_ts) AS date
  FROM `bigquery-public-data.hacker_news.stories`
  WHERE score IS NOT NULL AND title IS NOT NULL
  GROUP BY date)

SELECT date,
  ARRAY(SELECT AS STRUCT title, score
        FROM UNNEST(titles) ORDER BY score DESC
        LIMIT 2)
  AS top_articles
FROM TitlesAndScores;
```

WITH Clause:
- Make an array of (title, score) objects
- Extract the date from the timestamp
- Group by the date (which gives us the array contents)

ARRAY(SELECT AS STRUCT:
- Unnest the array from the WITH clause
- Order it and take the top 2
- Create a new array of (title, score) objects

Outer query:
- Project date from WITH clause
- Project Array

Notes:

With standard SQL, you can now store data in arrays natively. In BigQuery SQL, you had to store a list of values in a record and then use string concatenation.

You can also store STRUCTs natively. Previously, you had to store two pieces of data as one record and then use string concatenation.

ARRAY/STRUCT example result

Row	date	top_articles.title	top_articles.score
1	2010-08-23	Why GNU grep is Fast	512
		Readme Driven Development	244
2	2010-04-26	Police raid Gizmodo editor's house	257
		Not even in South Park?	257
3	2009-09-15	Learning Advanced JavaScript	257
		Sub-pixel re-workings of YouTube and BBC favicons	154

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

Notes:

In this example, the JOIN condition uses a non-equality comparison between keys.

JOIN condition example result

Row	name	frequency
1	John	282
2	Henry	226
3	Edward	200
4	Richard	180
5	Helen	97
6	Mark	97
7	Jack	84

Standard SQL functions

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

Notes:

For more information on BigQuery functions in standard SQL, see: https://cloud.google.com/bigquery/docs/reference/standard-sql/functions-and-operators. For information on functions in legacy SQL, see: https://cloud.google.com/bigquery/docs/reference/legacy-sql.

String function example

REGEX example

```
SELECT
     word,
     COUNT(word) as count
FROM `publicdata.samples.shakespeare`
WHERE (
     REGEXP_CONTAINS(word,r'^\w\w\'\w\w')
     )
GROUP BY word
ORDER BY count DESC
LIMIT 3;
```

Query Results 12:07pm, 17 Dec 2013

Row	word	count
1	ne'er	42
2	we'll	35
3	We'll	33

Notes:

This is a query that matches any five letter words with an apostrophe in the middle. This query demonstrated the flexibility of BigQuery. You don't need to pre-index the table in order for the search to return quickly. MySQL supports regexp as well. Here is an example - Select name from world.City where name REGEXP '^.*burg.*$'.]. Oracle supports SELECT first_name, last_name FROM employees WHERE REGEXP_LIKE (first_name, '^Ste(v|ph)en$');

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

Analytical SQL and window functions

- Analytic functions compute aggregate values over groups or “windows” of rows
- Window functions require an **OVER** clause that specifies the window frame
- The **PARTITION BY** clause allows window functions to partition data and parallelize execution

WILL LOOK AT PARTITIONS IN NEXT SECTION

- The **ORDER BY** clause (optional) defines order within each partition
  - No ORDER BY → nondeterministic row ordering

Window function example

Window function using RANK

```
SELECT
   corpus,
   word,
   word_count,
   RANK() OVER (
     PARTITION BY corpus
     ORDER BY word_count DESC) AS 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	kinghenryviii	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

Notes:

The sample query select over Shakespeare's work and find all the words that are longer than 10 with a minimum of 10 word counts, and rank by word count.

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

Notes:

Date/timestamp parsing and conversion functions in legacy SQL include:

- NOW() function: returns microseconds elapsed from 1/1/1970 (Epoch)
- CURRENT_TIMESTAMP() returns YYYY-MM-DD HH:MM:SS
- CURRENT_DATE() returns YYYY-MM-DD
- CURRENT_TIME() returns HH:MM:SS

Legacy SQL doesn't support the TRUNCATE function as in other systems; instead, use CURRENT DATE to get the date without the time. Standard SQL supports the DATE function which constructs a DATE from INT64 values representing the year, month, and day.

User-defined functions

- Allow SQL queries to use programming logic
- Allow functionality not supported by standard SQL
  - Loops, complex conditions, non-trivial string parsing
- Standard SQL UDFs are *scalar*
  - Legacy SQL UDFs are *tabular* (takes a whole row as input)
- User-defined functions (UDFs) are written in SQL or JavaScript®, and are temporary
  - You can only use UDFs for the current query or command-line session

Notes:

User defined functions allow BigQuery to execute code snippets against data.

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;
```

External user-defined function

- External UDF components
  - `CREATE [TEMPORARY | TEMP ] FUNCTION`
  - `RETURNS` [data_type] — Data type returned
  - `LANGUAGE` [language] — Language for function (JavaScript)
  - `AS` [external_code] — Code the function runs

Javascript UDF

```
CREATE TEMPORARY FUNCTION
multiplyInputs(x FLOAT64, y
FLOAT64)
RETURNS FLOAT64
LANGUAGE js AS """
  return x*y;
""";
WITH numbers AS
  (SELECT 1 AS x, 5 as y
  UNION ALL
  SELECT 2 AS x, 10 as y
  UNION ALL
  SELECT 3 as x, 15 as y)
SELECT x, y, multiplyInputs(x, y)
as product
FROM numbers;
```


Notes:

For more information on UDFs in standard SQL, see: https://cloud.google.com/bigquery/docs/reference/standard-sql/user-defined-functions.

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

Notes:

Also note: The DOM objects Window, Document and Node, and functions that require them, are unsupported for UDFs.

Lab 3: Advanced SQL Queries

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?

Agenda

Performance and pricing

How do you optimize queries?

- Less work → Faster query
- What is *work* for a query?
  - I/O — How many bytes did you read?
  - Shuffle — How many bytes did you pass to the next stage?
    - Grouping — How many bytes do you pass to each group?
  - Materialization — How many bytes did you write?
  - CPU work — User-defined functions (UDFs), functions

Don’t project unnecessary columns

- On how many columns are you operating?
- Excess columns incur wasted I/O and materialization

Don’t `SELECT` * unless you need every field

Filter early and often using `WHERE` clauses

- On how many rows (or partitions) are you operating?
- Excess rows incur “waste” similar to excess columns

Do the biggest joins first

- Joins — In what order are you merging data?
- Guideline — **Biggest, Smallest, Decreasing Size Thereafter**
- Avoid self-join if you can, since it squares the number of rows processed

Consider your `JOIN` order, try to filter the sets pre-`JOIN`

Notes:

Often you can replace self-join by a group-by:
http://weblogs.sqlteam.com/jeffs/archive/2007/06/12/using-group-by-to-avoid-self-joins.aspx

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

Built-in functions are faster than JavaScript UDFs

- Functions — What work are we doing on the data?
- Guideline — Some operators are faster than others; all are faster than JavaScript® UDFs
- Example — Exact `COUNT(DISTINCT)` is very costly, but `APPROX_COUNT_DISTINCT` is very fast

Note: Check to see if there are reasonable approximate functions for your query

`ORDER` on the outermost query

- Sorting—How many values do you need to sort?
  - Filtering first reduces the number of values you need to sort
  - Ordering first forces you to sort the world

Wildcard tables — Standard SQL (1 of 2)

- Use wildcards to query multiple tables using concise SQL statements
- Wildcard tables are a union of tables matching the wildcard expression
- Useful if your dataset contains:
  - Multiple, similarly named tables with compatible schemas
  - Sharded tables
- When you query, each row contains a special column with the wildcard match

Notes:

For more information on wildcard tables in standard SQL, see: https://cloud.google.com/bigquery/docs/wildcard-tables.

Wildcard tables — Standard SQL (2 of 2)

- **Example**:
  `FROM` `` `bigquery-public-data.noaa_gsod.gsod*` ``
- Matches all tables in noaa_gsod that begin with string 'gsod'
- The backtick (``) is required
- Richer prefixes perform better than shorter prefixes
  - For example: `.gsod200* versus .*`

Table partitioning

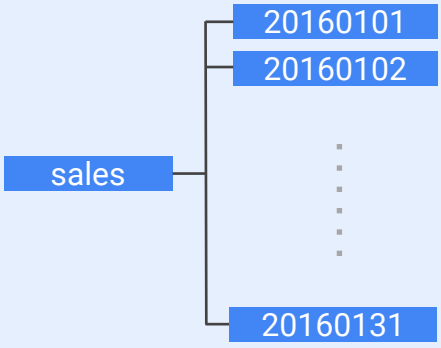
- Time-partitioned tables are a cost-effective way to manage data
- Easier to write queries spanning time periods
- When you create tables with time-based partitions, BigQuery automatically loads data in correct partition
  - Declare the table as partitioned at creation time using this flag: `--time_partitioning_type`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

Notes:

Partitioned tables include a pseudo column named _PARTITIONTIME that contains a date-based timestamp for data loaded into the table. The timestamp is based on UTC time and represents the number of microseconds since the unix epoch. For example, if data is appended to a table on April 15, 2016, all of the rows of data appended on that day contain the value TIMESTAMP("2016-04-15") in the _PARTITIONTIME column.

Example — Table partitioning

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



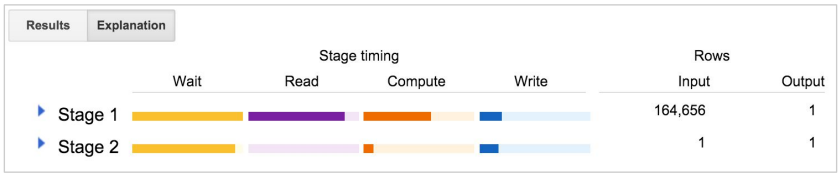
Understand query performance

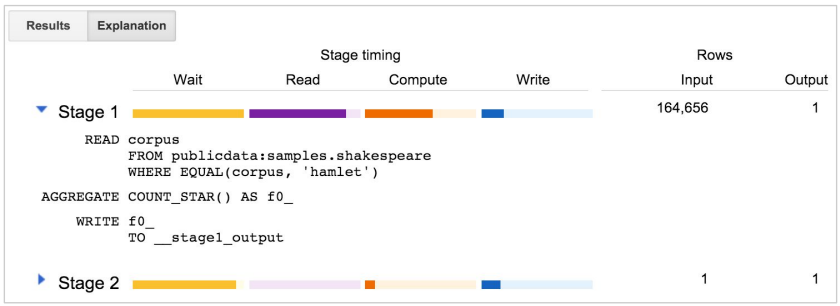
- You can optimize our queries and your data, but still need to monitor performance
- Two primary approaches:
  - Per-query explain plans
    - "What did my query do?"
  - Project-level monitoring through Google Stackdriver
    - "What is going on with all my resources in this project?"

Notes:

For information on table partitioning best practices, see: https://cloud.google.com/bigquery/docs/partitioned-tables#best_practices.

BigQuery explanation plans (1 of 2)





BigQuery explanation plans (2 of 2)

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

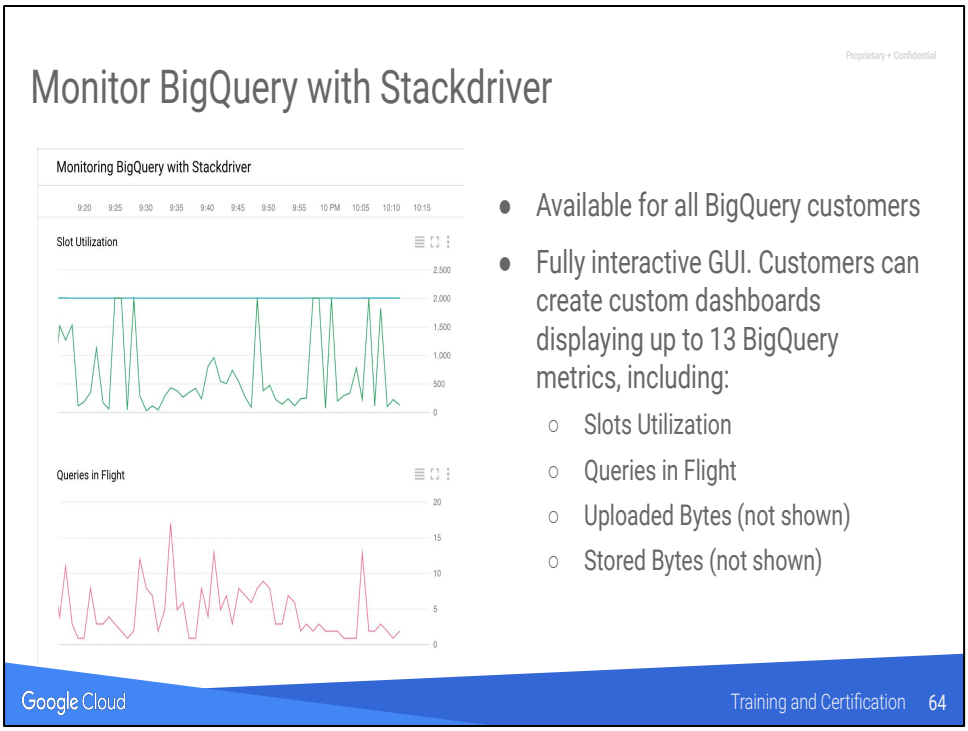
API JSON Name	Web UI*	Ratio Numerator **
`waitRatioAvg`	AVG	Time the average worker spent waiting to be scheduled.
`waitRatioMax`	MAX	Time the slowest worker spent waiting to be scheduled.
`readRatioAvg`	AVG	Time the average worker spent reading input data.
`readRatioMax`	MAX	Time the slowest worker 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.

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



Notes:

These charts show Slot Utilization, Slots available and queries in flight for a 1 hr period.

The Stackdriver charting tools offer

- Graphical User Interface to create custom dashboards for multiple GCP Products
- virtually real time data on many parameters (the lag on slot utilization for example is less than 5 minutes)
- Interactive graphical controls (zooming, creating new charts, selecting display modes, etc)

Known Issues:

- There is a known issue when Stackdriver reports slots available for customers that have subreservations. Please direct any questions to me.

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

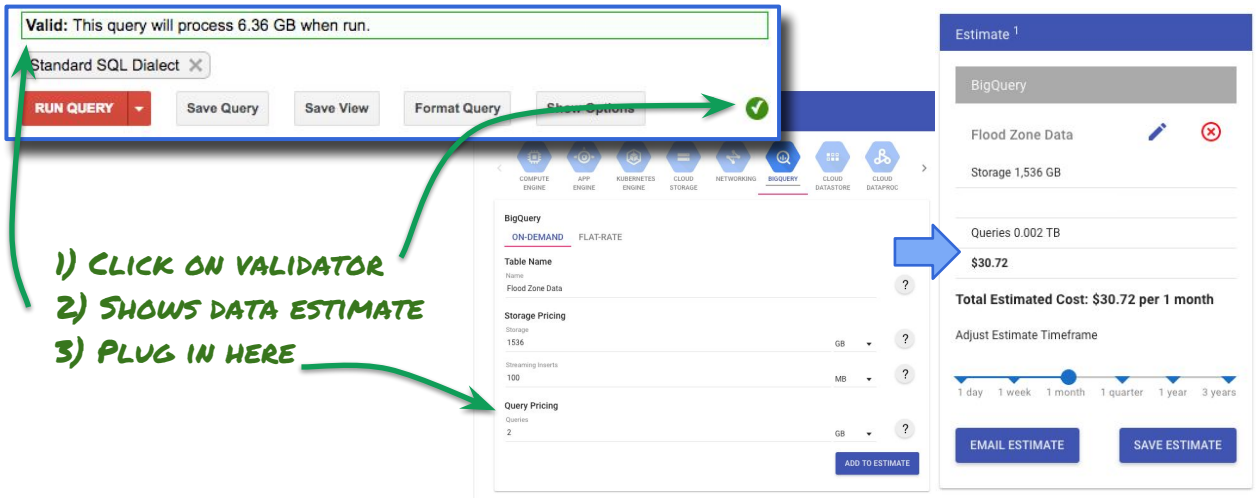
Free

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

Notes:

Image credits:
https://pixabay.com/en/storage-papers-office-cabinet-1209059/ (cc0)
https://pixabay.com/en/jet-engine-turbine-jet-airplane-371412/ (cc0)
https://pixabay.com/en/skydiving-jump-falling-parachuting-678168/ (cc0)

Use query validator with pricing calculator for estimates



https://cloud.google.com/products/calculator/
https://cloud.google.com/bigquery/docs/estimate-costs

Resources

BigQuery documentation	https://cloud.google.com/bigquery/docs/
Tutorials	https://cloud.google.com/bigquery/docs/tutorials
Pricing	https://cloud.google.com/bigquery/pricing
Client libraries	https://cloud.google.com/bigquery/client-libraries


cloud.google.com
Image by Connie Zhou