SQL, GEO, Analytics and Streaming with BigQuery

Juan Carlos Fernández
Lead Data Services @ Vodafone Group

Agenda

Google Cloud Platform - Trial

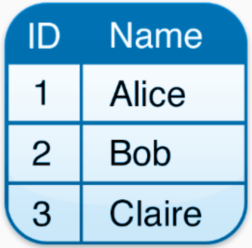
Periodo de Prueba de 90 días y 300 $ -

https://cloud.google.com/free/

SQL Introducción

What is a table?

- A table is a collection of data consisting of fields and values, similar to a spreadsheet.
- A table has a specified number of columns but can have any number of rows.



What is a database?

- Collection of tables
- Contains tables, but in some systems, also reports, queries, views, and other objects.
- In BigQuery called a “dataset”

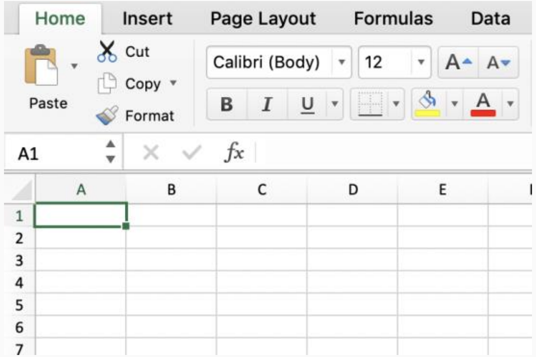
The anatomy of a table: Rows

Data rows

id	first_name	last_name	username	joined_in	home_base
1	Tommy	Liu	tommy	2015-11-07	KLIA
2	Katie	Wilson	kat	2013-10-13	SIN
3	Henry	Wan	henry	2012-05-19	KLIA
4	Iko	Uwais	ikouwais	2011-09-07	CGK
5	Atiqah	Basuki	atiqah	2010-02-16	KLIA
6	Grace	Chan	gracechan	2016-06-01	SIN

What is a row?

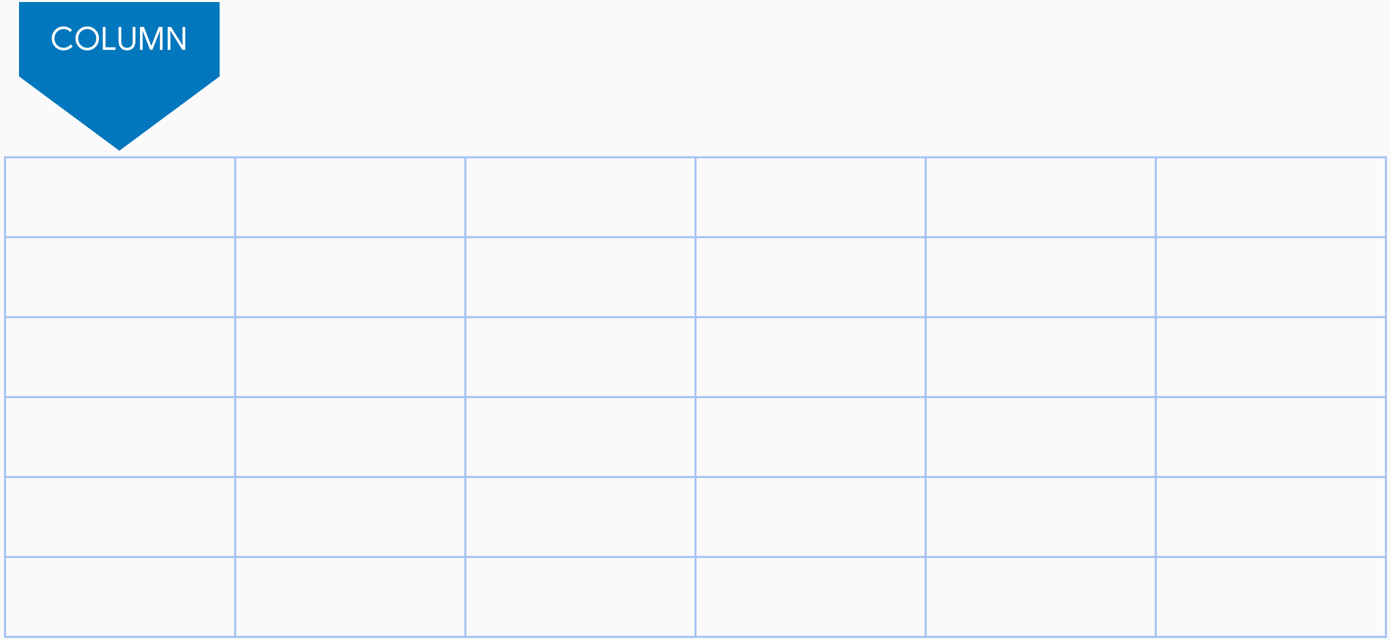
- A table has a specified number of columns but can have any number of rows. Rows have a number (“1” in Excel)
- In BigQuery, tables can be *extremely* large, with 10,000 columns and almost unlimited rows, 15TB per single import load



What is a column?

- A similar type of values or cells, has a name (“A” in Excel)
- Types of columns in BigQuery are for instance:
  - **STRING**, any text (there is no VARCHAR, CHAR or CLOB)
  - **INT64** or INTEGER, a number with a decimal point
  - **NUMERIC**, decimal numbers e.g. prices
  - **DATE, TIME, DATETIME or TIMESTAMP**

The anatomy of a table: Columns



The anatomy of a table

Integer

id	first_name	last_name	username	joined_in	home_base
1	Tommy	Liu	tommy	2015-11-07	KLIA
2	Katie	Wilson	kat	2013-10-13	SIN
3	Henry	Wan	henry	2012-05-19	KLIA
4	Iko	Uwais	ikouwais	2011-09-07	CGK
5	Atiqah	Basuki	atiqah	2010-02-16	KLIA
6	Grace	Chan	gracechan	2016-06-01	SIN

The anatomy of a table

String

id	first_name	last_name	username	joined_in	home_base
1	**Tommy**	Liu	tommy	2015-11-07	KLIA
2	**Katie**	Wilson	kat	2013-10-13	SIN
3	**Henry**	Wan	henry	2012-05-19	KLIA
4	**Iko**	Uwais	ikouwais	2011-09-07	CGK
5	**Atiqah**	Basuki	atiqah	2010-02-16	KLIA
6	**Grace**	Chan	gracechan	2016-06-01	SIN

The anatomy of a table

Date

id	first_name	last_name	username	joined_in	home_base
1	Tommy	Liu	tommy	**2015-11-07**	KLIA
2	Katie	Wilson	kat	**2013-10-13**	SIN
3	Henry	Wan	henry	**2012-05-19**	KLIA
4	Iko	Uwais	ikouwais	**2011-09-07**	CGK
5	Atiqah	Basuki	atiqah	**2010-02-16**	KLIA
6	Grace	Chan	gracechan	**2016-06-01**	SIN

How to get information from a database

- In BigQuery, databases are called "datasets"
- Database information is stored in tables of any size
- Tables are accessed using a "query language"
- The query language can combine different data (Pivots)
- You can write them yourself or use a tool
- A result of a query is also just a table
  - Has any shape you like
  - Can be downloaded (not so great)
  - Can be accessed by other tools (great!)

Querying databases using SQL

/ˈɛs kjuː ˈɛl/ or /ˈsiːkwəl/

is a special-purpose programming language designed for managing data held in a relational database management system (RDBMS).

- **SELECT** is a keyword, extracts data from the database * is the result set definition (* is a character that gets **all** column data)
- **FROM** is a keyword
- **employees** is the table name

```
SELECT * FROM employees;
```

id	first_name	last_name	username	joined_in	home_base
1	Tommy	Liu	tommy	2015-11-07	KLIA
2	Katie	Wilson	kat	2013-10-13	SIN
3	Henry	Wan	henry	2012-05-19	KLIA
4	Iko	Uwais	ikouwais	2011-09-07	CGK
			....		

- **LIMIT** limits the results*

```
SELECT * FROM employees LIMIT 2;
```

id	first_name	last_name	username	joined_in	home_base
1	Tommy	Liu	tommy	2015-11-07	KLIA
2	Katie	Wilson	kat	2013-10-13	SIN

*In BigQuery be careful with LIMIT clause it will still scan the full table and won't limit cost.

- **SELECT** extracts data from the database
- **first_name, last_name** are specified columns

```
SELECT first_name, last_name FROM employees;
```

first_name	last_name
Tommy	Liu
Katie	Wilson
Henry	Wan
Iko	Uwais
...	...

- **WHERE** clause is used to filter records.

```
SELECT * FROM employees WHERE home_base = "KLIA";
```

id	first_name	last_name	username	joined_in	home_base
1	Tommy	Liu	tommy	2015-11-07	KLIA
3	Henry	Wan	henry	2012-05-19	KLIA
5	Atiqah	Basuki	atiqah	2010-02-16	KLIA

- **WHERE** clause is used to filter records
  You can use several operators to filter records, depending on the column data type.

=, !=, <>	equal, not equal, not equal
<	less than
>	greater than
[NOT] LIKE	value does [not] match the pattern
[NOT] BETWEEN	value is [not] within the range
[NOT] IN	value is [not] in the set of values
IS [NOT] NULL/TRUE/FALSE	value is [not] NULL/TRUE/FALSE

- **ORDER BY** orders the result list by a selected field ascending (default) or descending

```
SELECT * FROM employees ORDER BY start_day DESC;
```

id	first_name	last_name	username	joined_in	home_base
6	Grace	Chan	gracechan	2016-06-01	SIN
1	Tommy	Liu	tommy	2015-11-07	KLIA
2	Katie	Wilson	kat	2013-10-13	SIN
3	Henry	Wan	henry	2012-05-19	KLIA
			....		

- **FUNCTIONS** have a lot of different outputs, and can be used almost everywhere in a field (column), WHERE or ORDER

```
SELECT * FROM employees WHERE EXTRACT(YEAR FROM joined_in) = 2016;
```

id	first_name	last_name	username	joined_in	home_base
6	Grace	Chan	gracechan	**2016-06-01**	SIN
5	Atiqah	Basuki	atiqah	**2016-02-16**	KLIA
			....		

Standard SQL | Filtering

```
SELECT
 gender,
 tripduration
FROM
 `bigquery-public-data`.new_york_citibike.citibike_trips
WHERE
 tripduration >= 300 AND tripduration < 600 AND gender = 'female'
LIMIT
 5
```

Standard SQL | SubQueries WITH

```
SELECT * FROM (
 SELECT
   gender, tripduration / 60 AS minutes
 FROM
   `bigquery-public-data`.new_york_citibike.citibike_trips
)
WHERE minutes < 10
LIMIT 5
```

Standard SQL | SubQueries WITH

```
WITH all_trips AS (
 SELECT
   gender, tripduration / 60 AS minutes
 FROM
   `bigquery-public-data`.new_york_citibike.citibike_trips
)

SELECT * from all_trips
WHERE minutes < 10
LIMIT 5
```

Standard SQL | Intro to Functions

Cast Functions (link) to change date type

```
SELECT
 CAST(fare AS string) AS castedfare
FROM
 `bigquery-public-data.chicago_taxi_trips.taxi_trips`
LIMIT
 10
```

Standard SQL | Control flow functions

Operator	**Meaning**
IF(condition, true_return, false_return)	Returns either **true_return** or **false_return**, depending on the condition.
CASE WHEN when_expr1 THEN then_expr1 WHEN when_expr2 THEN then_expr2 ... ELSE else_expr END	Use **CASE** to choose among two or more alternate expressions.
IFNULL(expr1, expr2)	If **expr1** is not NULL, returns **expr1**, otherwise it returns **expr2**

Standard SQL | Intro to Functions

Date Functions (link)

SELECT CURRENT_DATE() as the_date; ⟶ Current Date
SELECT EXTRACT(WEEK from CURRENT_DATE()) as the_date_day; ⟶ Extract Week
SELECT EXTRACT(WEEK(SUNDAY) from CURRENT_DATE()) as the_date_day; ⟶ Extract Week starting on sunday
SELECT
date(trip_start_timestamp,"Europe/Madrid") as date
from `bigquery-public-data.chicago_taxi_trips.taxi_trips` -->Extract date from a timestamp
SELECT DATE_ADD(CURRENT_DATE(), INTERVAL 5 DAY) as five_days_later; ⟶ Add dates
SELECT DATE_SUB(CURRENT_DATE(), INTERVAL 5 DAY) as five_days_ago; ⟶ substract dates
SELECT DATE_DIFF('2017-12-30', '2014-12-30', YEAR) AS year_diff;
SELECT DATE_DIFF('2017-12-30', '2014-12-30', DAY) AS day_diff;
SELECT PARSE_DATE("%x", "12/25/08") as parsed; ⟶ parse date 2008-12-25
SELECT FORMAT_DATE("%x", DATE "2008-12-25") as US_format; ⟶ parse_date 12/25/08
SELECT DATE_TRUNC(DATE '2008-12-25', month) as start_of_month;
SELECT TIMESTAMP_MILLIS(1230219000000) as timestamp; ⟶ change miliseconds to timestamp

Standard SQL | Intro to Functions

String Functions (link)

SELECT cast(CONCAT('1','2') as float64) as concated; ⟶ Concatenate string and convert to float

SELECT lower("APPLE") as lowered; ⟶ Lower capital letters

Standard SQL | Intro to Functions

Aggregation Functions (link)

```
SELECT
 AVG(tripduration / 60) AS avg_trip_duration
FROM
 `bigquery-public-data`.new_york_citibike.citibike_trips
WHERE
 gender = 'male'
```

Standard SQL | Intro to Functions

Aggregation Functions (link)

```
SELECT
 gender, AVG(tripduration / 60) AS avg_trip_duration
FROM
 `bigquery-public-data`.new_york_citibike.citibike_trips
WHERE
 tripduration is not NULL
GROUP BY
 gender
ORDER BY
 avg_trip_duration
```

Standard SQL | Intro to Functions

COUNT (link)

```
SELECT
 gender,
 COUNT(*) AS rides,
 AVG(tripduration / 60) AS avg_trip_duration
FROM
 `bigquery-public-data`.new_york_citibike.citibike_trips
WHERE
 tripduration IS NOT NULL
GROUP BY
 gender
ORDER BY
 avg_trip_duration
```

Standard SQL | Intro to Functions

Filtering with HAVING

```
SELECT
 gender, AVG(tripduration / 60) AS avg_trip_duration
FROM
 `bigquery-public-data`.new_york_citibike.citibike_trips
WHERE tripduration IS NOT NULL
GROUP BY
 gender
HAVING avg_trip_duration > 14
ORDER BY
 avg_trip_duration
```

Standard SQL | Intro to Functions

Unique values with DISTINCT

```
SELECT DISTINCT
  gender
FROM
  `bigquery-public-data`.new_york_citibike.citibike_trips
```

Standard SQL | Intro to Functions

Statistical Functions (link)

```
SELECT
stddev(noemplyeesw3cnt) as st_dev_employee_count,
avg(noemplyeesw3cnt) as avg_employee_count,
APPROX_QUANTILES(noemplyeesw3cnt, 100)[OFFSET(99)] AS employee_count_percentile_99,
APPROX_QUANTILES(noemplyeesw3cnt, 100)[OFFSET(90)] AS employee_count_percentile_90,
APPROX_QUANTILES(noemplyeesw3cnt, 100)[OFFSET(70)] AS employee_count_percentile_70,
APPROX_QUANTILES(noemplyeesw3cnt, 100)[OFFSET(50)] AS employee_count_percentile_50,
corr( totprgmrevnue, totfuncexpns) as corr_rev_expense,
approx_count_distinct(ein) as approx_nonprofits,
count(distinct ein) as nonprofits
    from `bigquery-public-data.irs_990.irs_990_2016`
    where frgnofficecd = "N"
```

Break

BigQuery Introducción

Google BigQuery

Google Cloud Platform's
enterprise data warehouse
for analytics

Gigabyte- to **petabyte-scale**
storage and SQL queries

Encrypted, durable,
And highly available

GeoVizualizations
And geometric operations

Unique

Real-time insights from streaming data

Unique

Built-in **ML and GIS** for out-of-the-box
predictive insights

Unique

High-speed, in-memory **BI Engine**
for faster reporting and analysis

Unique

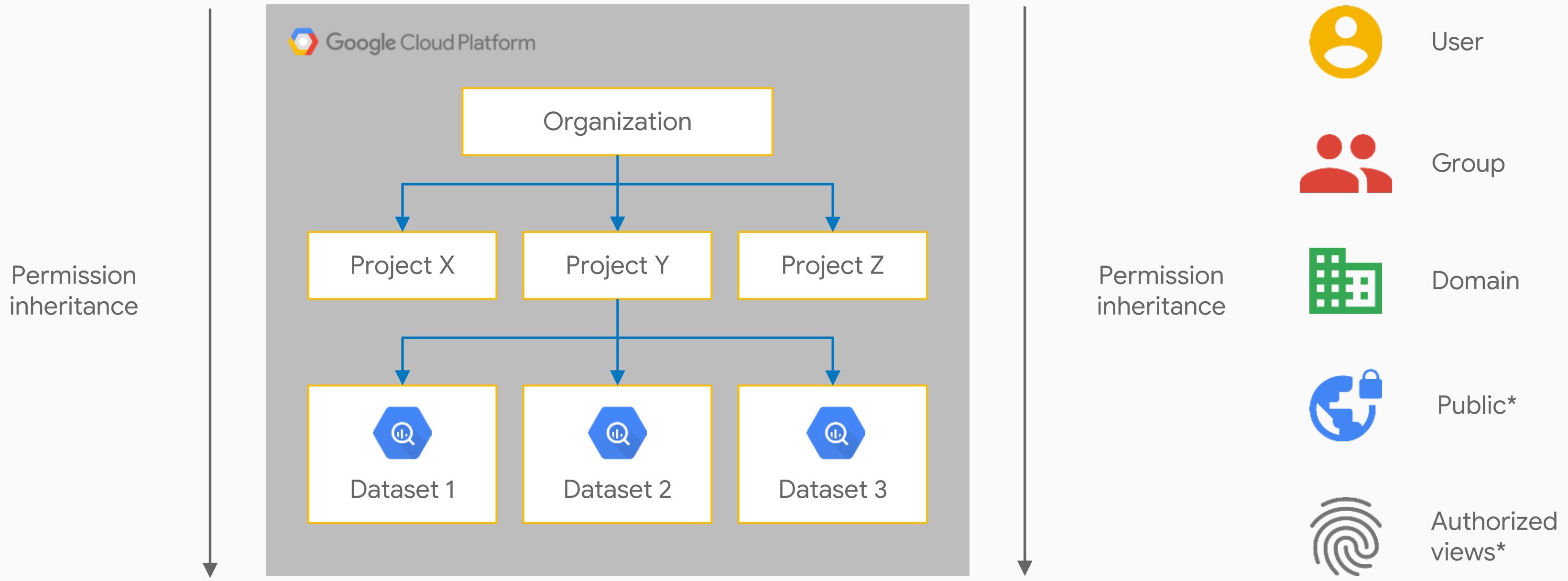
BigQuery | Why is so powerful

1. **Storage Differentiated from compute:** Permanent Storage Vs Temporal compute makes cheaper and faster

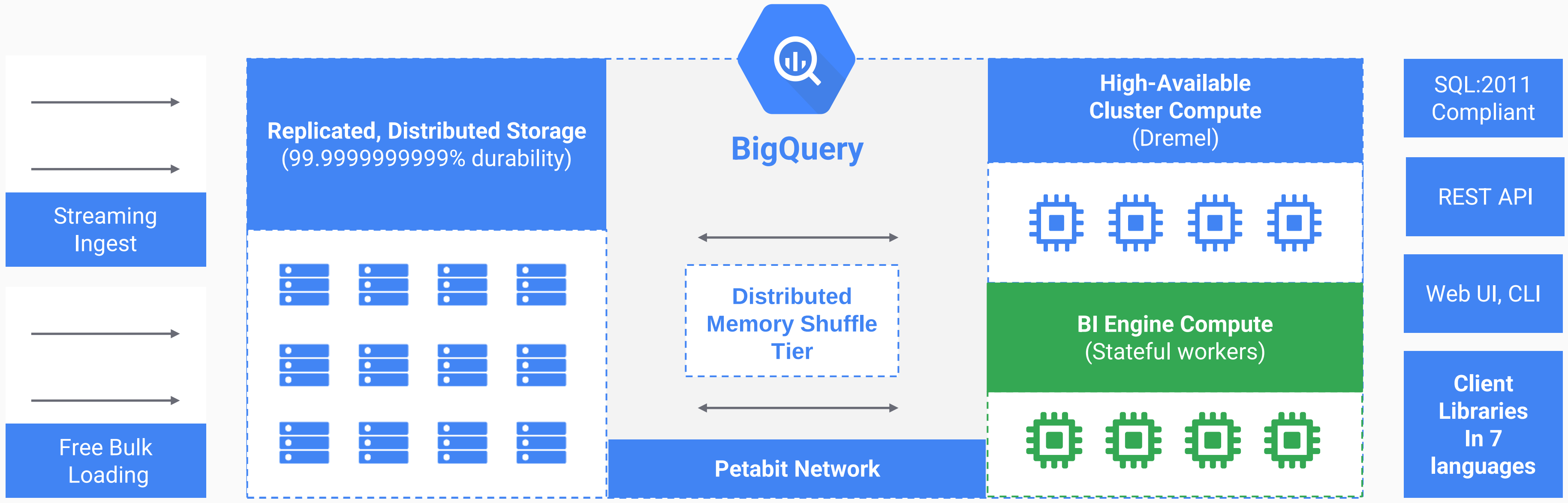
2. **Columnar Based storage:** Data model is based on columns vs registers making faster and cheaper

3. **Serverless:** Let BigQuery do the heavy lifting for you

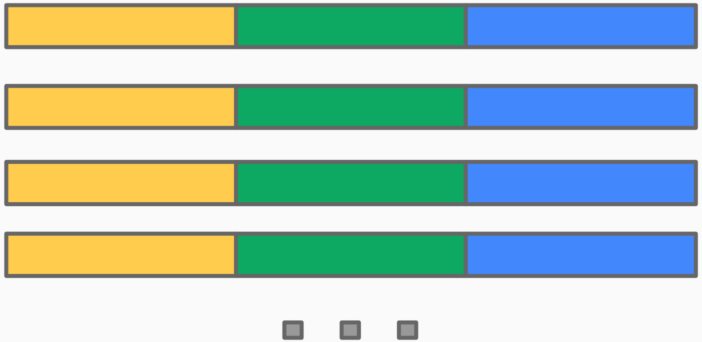
BigQuery: uses Cloud IAM



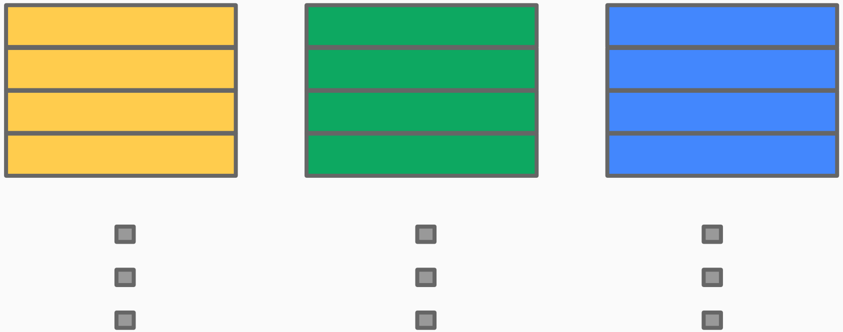
BigQuery: Arquitectura



BigQuery: Columnar Storage

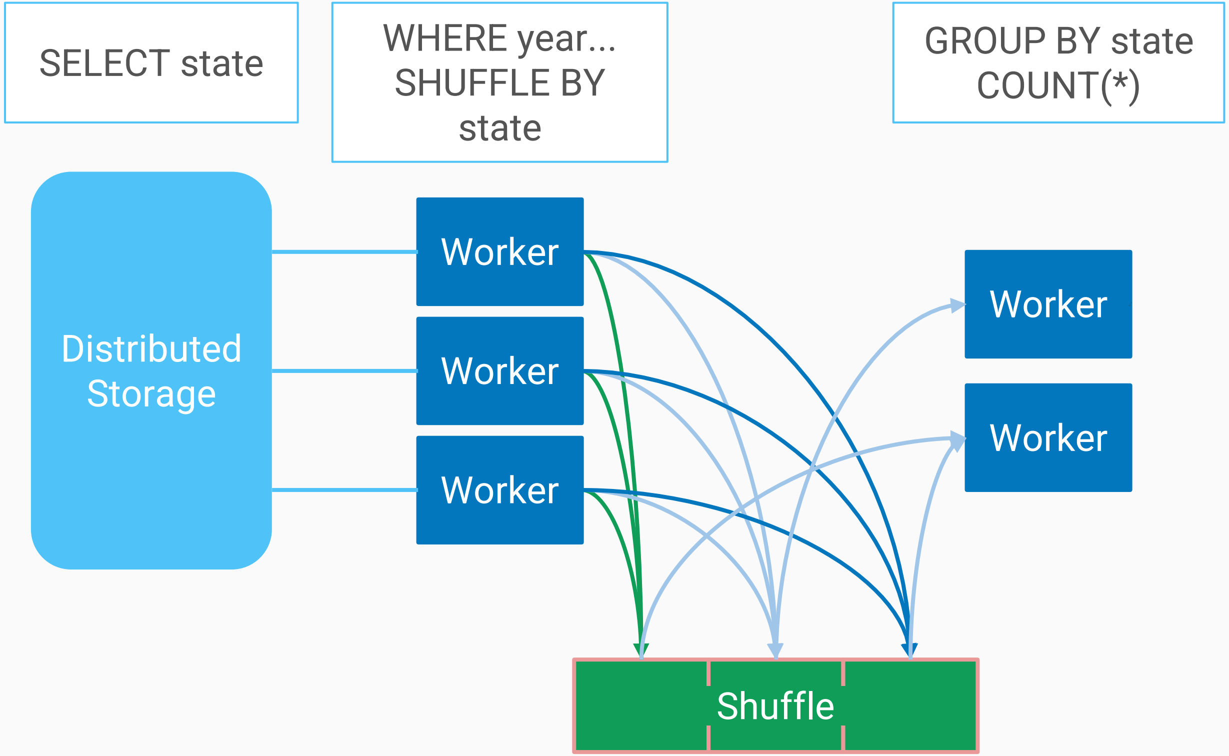


Record-Oriented Storage



Column-Oriented Storage

BigQuery: Remote Memory Shuffle



Shuffle does not block future stages

BigQuery uses dynamic partitioning to distribute shuffle optimally

Substantial key-skew can still impact performance

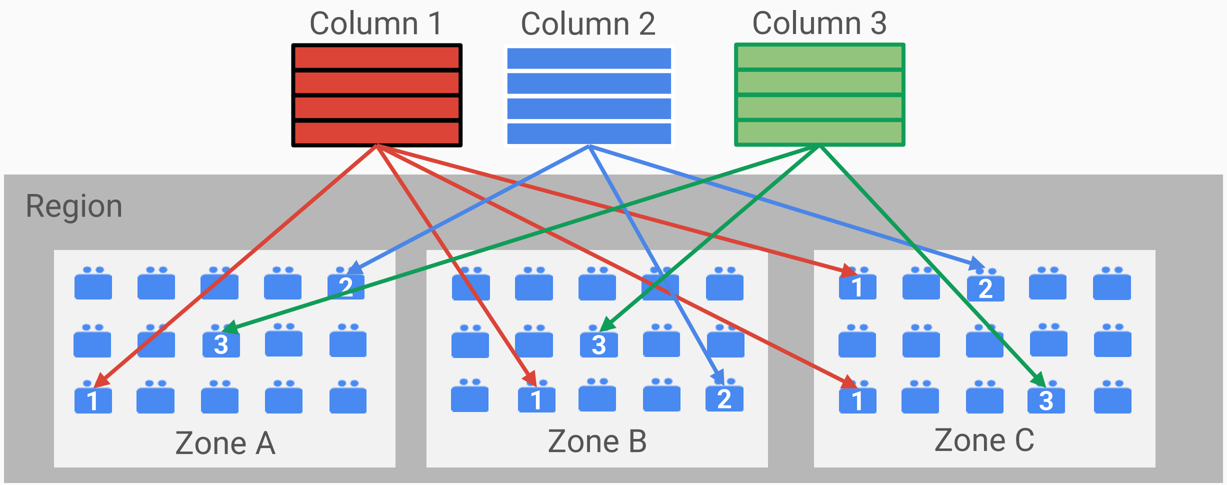
BigQuery: Managed Storage

Each column is stored in its own file

Each file is compressed and encrypted on disk

Storage is durable

Each file is replicated across datacenters



BigQuery: Slots

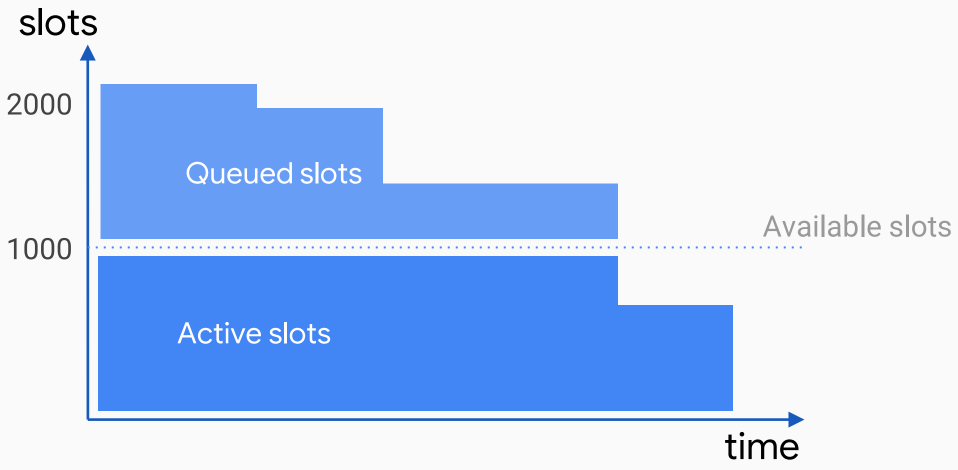
- Unit of scheduling
- Logically, one parallel ‘work unit’
- Measures throughput, not performance
- Maps to resource sizes (X GB RAM, X CPU)
- Can be restarted if slow / failed
- Can be cancelled if no longer needed

BigQuery: Slots

- Default slot allocation: 2,000 per project
- Fair scheduler between queries within a project
- Fair scheduler between projects
- Slots scheduled for each stage of query
- Slot usage depends on underlying representation and data propagation

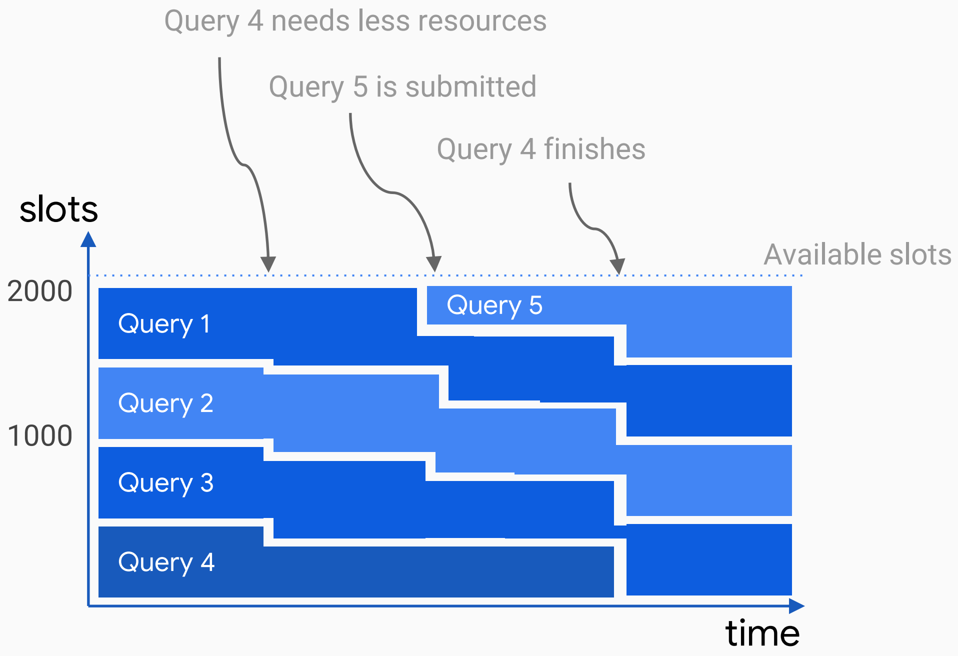
BigQuery | Resource queueing

- If resource demands exceed available capacity, BigQuery queues up additional slots.
- As query execution progresses, BigQuery automatically works through the queued up work until none is left.

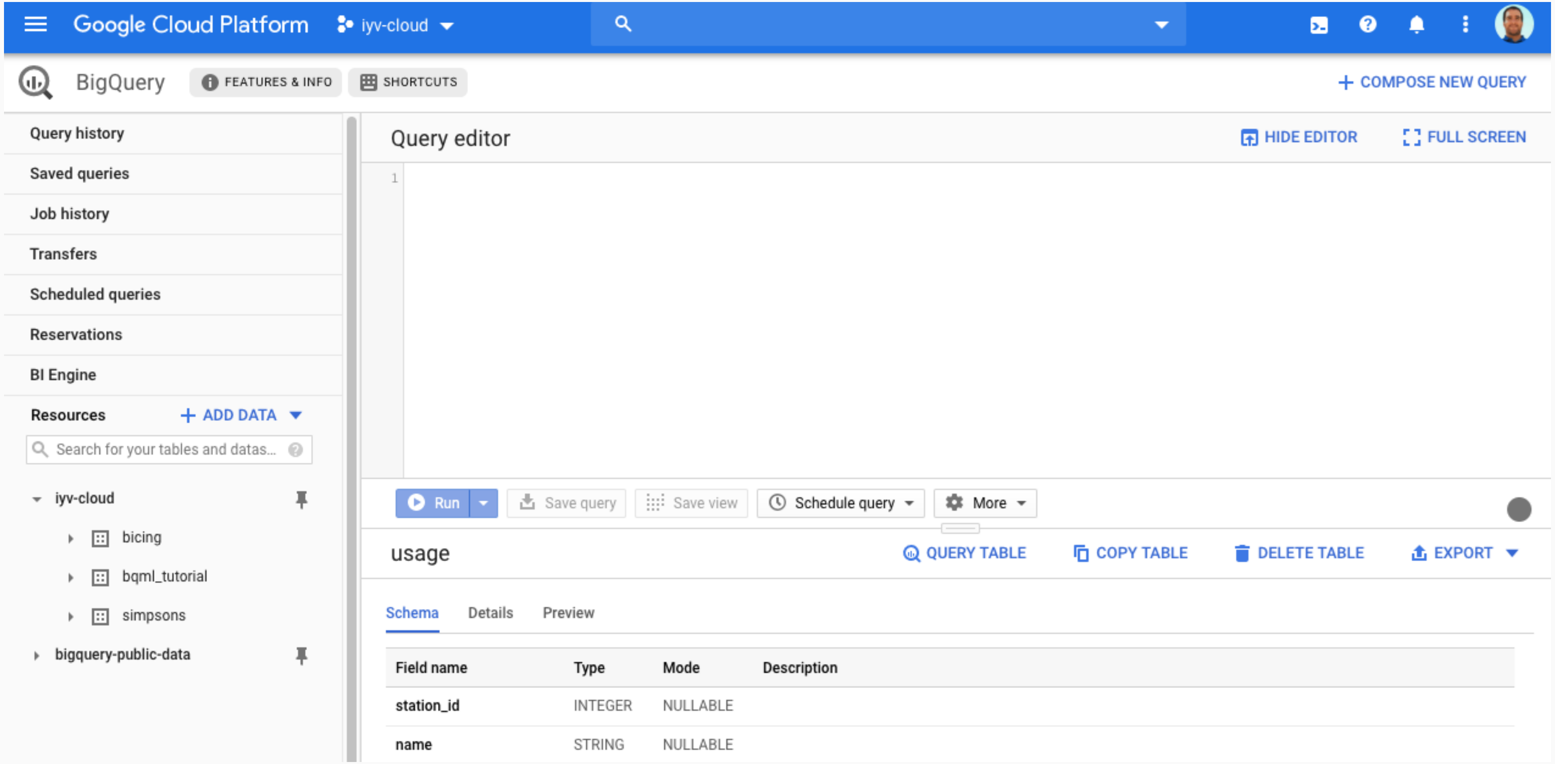


BigQuery | Fair scheduling

- BigQuery dynamically allocates capacity
- Each query gets "fair share" of resources
- Bits of work are gracefully paused
- Avoids workload starvation



BigQuery - UI



Google Cloud Platform
iyv-cloud
BigQuery
FEATURES & INFO
SHORTCUTS
COMPOSE NEW QUERY
Query history
Saved queries
Job history
Transfers
Scheduled queries
Reservations
BI Engine
Resources
ADD DATA
Search for your tables and datas...
iyv-cloud
bicing
bqml_tutorial
simpsons
bigquery-public-data
Query editor
HIDE EDITOR
FULL SCREEN
1
Run
Save query
Save view
Schedule query
More
usage
QUERY TABLE
COPY TABLE
DELETE TABLE
EXPORT
Schema
Details
Preview
Field name
Type
Mode
Description
station_id
INTEGER
NULLABLE
name
STRING
NULLABLE

BigQuery - Sample Query

```
SELECT COUNT(*)
FROM `bigquery-
samples.wikipedia_benchmark.Wiki1B`
WHERE REGEXP_CONTAINS(title,"G.*o.*o.*g")
```

BigQuery - Query Plan Explanation

Query complete (5.1 sec elapsed, 37.2 GB processed)

Job information | Results | JSON | **Execution details**

For help debugging or optimizing your query, check our documentation. Learn more

Elapsed time	Slot time consumed	Bytes shuffled	Bytes spilled to disk
5.1 sec	14 min 37.097 sec	1.93 KB	0 B

Worker timing

Stages		Wait	Read	Compute	Write		Rows
S00: Input	Avg:	792 ms	853 ms	2242 ms	3 ms	Input:	1,249,541,131
	Max:	1585 ms	1397 ms	2757 ms	38 ms	Output:	220
S01: Output	Avg:	1799 ms	0 ms	5 ms	14 ms	Input:	220
	Max:	1799 ms	0 ms	5 ms	14 ms	Output:	1

Data ingestion options

Batch ingestion

Data from GCS or via HTTP POST

Multiple File Formats Supported

Snapshot-based arrival - All Data arrives at once, or not at all

Streaming ingestion

Continuous ingestion from many sources (web/mobile apps, point of sale, supply chain)

Immediate query availability from buffer

Deferred creation of managed storage

Query materialization

SELECT results yield data in the form of tables, either anonymous (cached) or named destinations

ETL/ELT Ingest + Transform via Federated Query

Data Transfer Service (DTS)

Managed ingestion of other sources (doubleclick, adwords, youtube)

Newer: Scheduled Queries, Scheduled GCS Ingestion

Options for third-party integration

Loading data

Batch ingest is free

Doesn't consume query capacity

ACID semantics

Load petabytes per day

Streaming API for real-time

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:

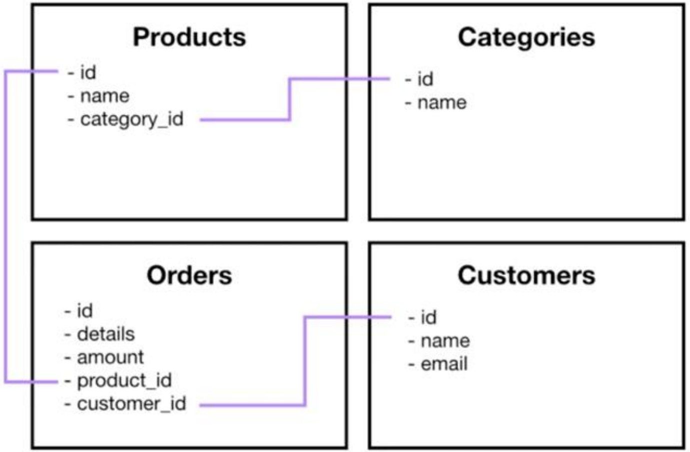
- Loading
- Exporting
- Queries on metadata
- Cached queries
- Queries with errors
- 10GB storage

Esquemas en Data Warehouses

To normalize or to not normalize?

Normalized

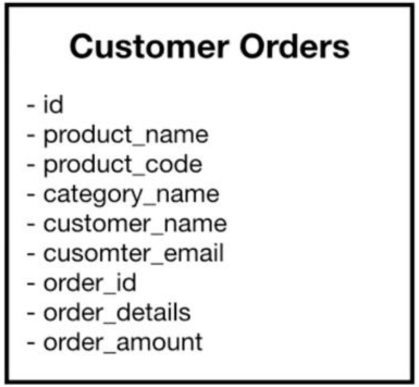
A schema design to store **non-redundant** and **consistent data**



- Data Integrity is maintained
- Little to no redundant data
- Many tables
- Optimizes for storage of data

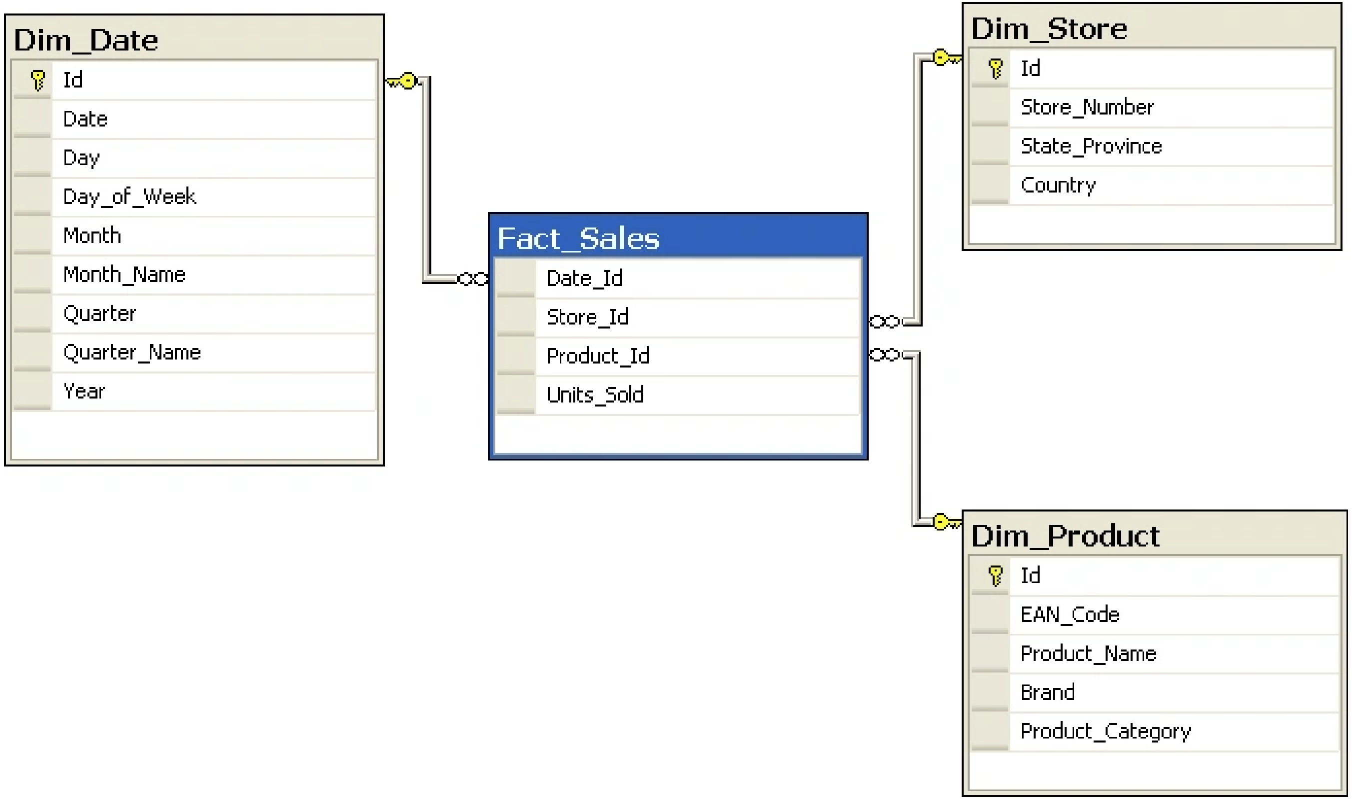
Denormalized

A schema that **combines data** so that **accessing data (querying) is fast**



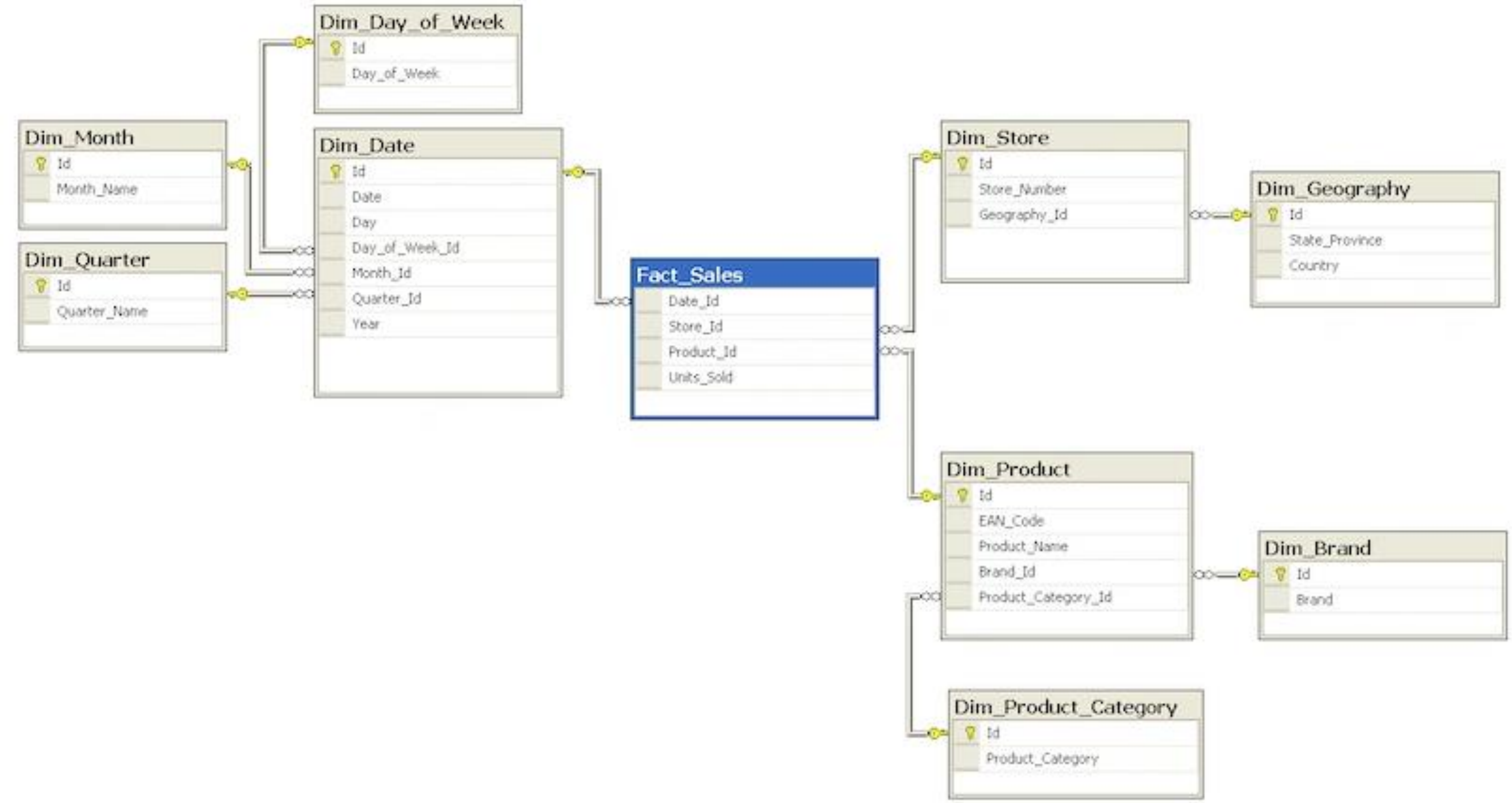
- Data Integrity is not maintained
- Redundant data is common
- Fewer tables
- Excessive data, storage is less optimal

Star Schema



Dim_Date
Id
Date
Day
Day_of_Week
Month
Month_Name
Quarter
Quarter_Name
Year
Fact_Sales
Date_Id
Store_Id
Product_Id
Units_Sold
Dim_Store
Id
Store_Number
State_Province
Country
Dim_Product
Id
EAN_Code
Product_Name
Brand
Product_Category

Snowflake Schema



BigQuery: Use denormalized schema when possible

Transaction Fact					
Order Id	timestamp	CustomerId	sku	quantity	price
1000001	12/18/2017 15:02:00	65401	ABC123456	3	36.3
1000001	12/18/2017 15:02:00	65401	TBL535522	1	878.4
1000001	12/18/2017 15:02:00	65401	CHR762222	6	435.6
1000002	12/16/2017 11:34:00	74682	GCH635354	4	345.7
1000002	12/16/2017 11:34:00	74682	GRD828822	2	9.5

Customer Dimension		
CustomerId	CustomerName	Location
65401	John Doe	Faraway
74682	Jane Michaels	Nearland
63636	Jose Carlos	Nearland

Product Dimension	
sku	description
ABC123456	Redwood 8x4
TBL535522	Sapient Table
CHR762222	Cherrywood Chair
GCH635354	Garden chairs
GRD828822	Ceramic Pots

BigQuery: Use denormalized schema when possible

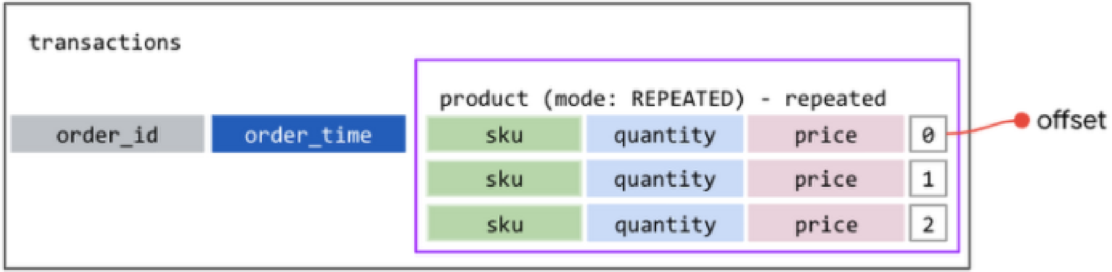
OrderId	CustomerId	CustomerName	timestamp	Location	purchasedItems			
					sku	**description**	**quantity**	**price**
1000001	65401	John Doe	12/18/2017 15:0	Faraway	ABC123456	Redwood 8x4	3	36.3
					TBL535522	Sapient Table	1	878.4
					CHR762222	Cherrywood Ch	6	435.6
					sku	**description**	**quantity**	**price**
1000002	74682	Jane Michaels	12/16/2017 11:3	Nearland	GCH635354	Garden chairs	4	345.7
					GRD828822	Ceramic Pots	2	9.5
					sku	**description**	**quantity**	**price**
1000003	63636	Jose Carlos	12/16/2017 13:4	Nearland				

SQL Intermedio

Nested Fields

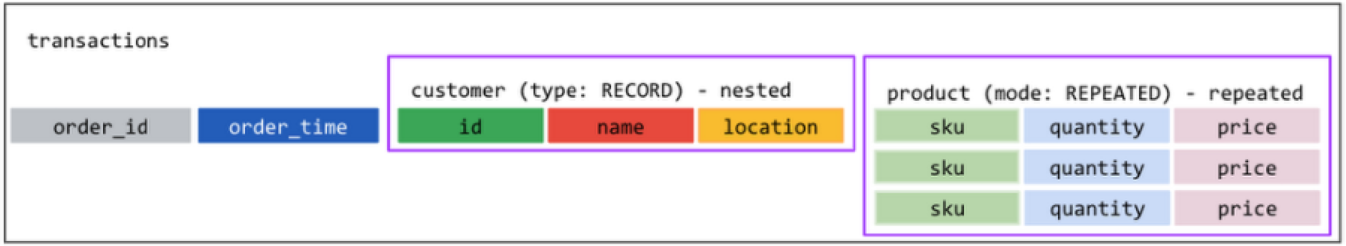
- **STRUCT/RECORD** data type contains ordered fields with a type and name.
- Use dot notation to query a nested column.
  E.g. **customer.name** refers to **name** field in **customer** column.

Repeated Fields



- **`ARRAY`** data type is an ordered list of zero or more elements of the same data type. For e.g. **`product`** is an **`ARRAY`** of **`STRUCT`** here.
- Use `UNNEST()` to flatten the repeated data and **`OFFSET/ORDINAL`** to access individual element

Nested Repeated Fields

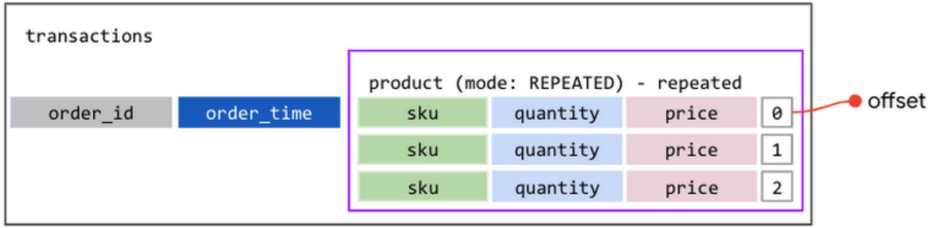


- Combining nested and repeated fields denormalizes a 1:many relationship without joins.
- Use dot notation to query a nested column and `UNNEST()` to flatten the repeated data.

Nested Fields

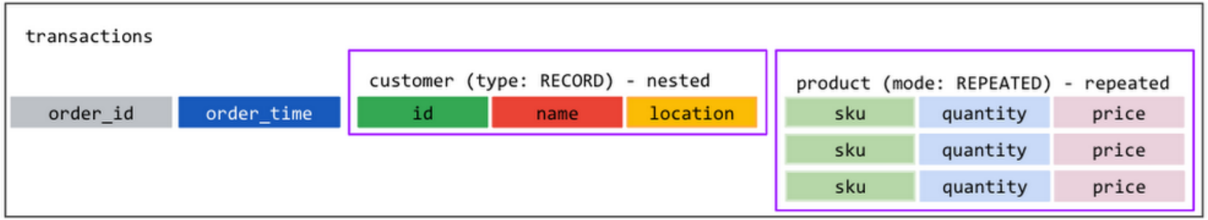
- **`STRUCT/RECORD`** data type contains ordered fields with a type and name.
- Use dot notation to query a nested column.
  E.g. **`customer.name`** refers to **`name`** field in **`customer`** column.

Repeated Fields



- **`ARRAY`** data type is an ordered list of zero or more elements of the same data type.
  For e.g. **`product`** is an **`ARRAY`** of **`STRUCT`** here.
- Use `UNNEST()` to flatten the repeated data and **`OFFSET/ORDINAL`** to access individual element

Nested Repeated Fields



- Combining nested and repeated fields denormalizes a 1:many relationship without joins.
- Use dot notation to query a nested column and `UNNEST()` to flatten the repeated data.

Standard SQL | Arrays & structs

Arrays (link)

Arrays are **ordered lists** of zero or more data values that must have the **same data type**

Standard SQL | Arrays & structs

Structs (link)

STRUCT are a container of ordered fields each with a type (required) and field name (optional).

You can store multiple data types in a STRUCT (even Arrays!)

Standard SQL | Arrays and Structs

```
SELECT
 city, SPLIT(city, ' ') AS parts
FROM (
 SELECT * from UNNEST([
    'Seattle WA', 'New York', 'Singapore'
 ]) AS city
)
```

Row	city	parts
1	Seattle WA	Seattle
		WA
2	New York	New
		York
3	Singapore	Singapore

Standard SQL | Arrays and Structs

```
SELECT
 gender
 , EXTRACT(YEAR from starttime) AS year --
 , COUNT(*) AS numtrips
FROM
 `bigquery-public-data`.new_york_citibike.citibike_trips
WHERE gender != 'unknown' and starttime IS NOT NULL
GROUP BY gender, year
HAVING year > 2016
```

Row	gender	year	numtrips
1	female	2018	1260893
2	male	2017	9306602
3	female	2017	3236735
4	male	2018	3955871

Standard SQL | Arrays and Structs (ARRAY_AGG)

```
SELECT
 gender
 , ARRAY_AGG(numtrips order by year) AS numtrips
FROM (
 SELECT
   gender
   , EXTRACT(YEAR from starttime) AS year
   , COUNT(1) AS numtrips
 FROM
   `bigquery-public-data`.new_york_citibike.citibike_trips
 WHERE gender != 'unknown' and starttime IS NOT NULL
 GROUP BY gender, year
 HAVING year > 2016
)
GROUP BY gender
```

Row	gender	numtrips
1	male	9306602
		3955871
2	female	3236735
		1260893

Standard SQL | Arrays and Structs -> JSON

```
[
  {
    "gender": "male",
    "numtrips": [
      "9306602",
      "3955871"
    ]
  },
  {
    "gender": "female",
    "numtrips": [
      "3236735",
      "1260893"
    ]
  }
]
```

Standard SQL | Arrays and Structs (ARRAY_AGG)

```
SELECT
 gender
 , ARRAY_AGG(numtrips order by year) AS numtrips
FROM (
 SELECT
   gender
   , EXTRACT(YEAR from starttime) AS year
   , COUNT(1) AS numtrips
 FROM
   `bigquery-public-data`.new_york_citibike.citibike_trips
 WHERE gender != 'unknown' and starttime IS NOT NULL
 GROUP BY gender, year
 HAVING year > 2016
)
GROUP BY gender
```

Row	gender	numtrips
1	male	9306602
		3955871
2	female	3236735
		1260893

Standard SQL | ARRAY of STRUCT

```
SELECT
 [
   STRUCT('male' as gender, [9306602, 3955871] as numtrips)
   , STRUCT('female' as gender, [3236735, 1260893] as numtrips)
 ] AS bikerides
```

Row	bikerides.gender	bikerides.numtrips
1	male	9306602
		3955871
	female	3236735
		1260893

Standard SQL | Working with Arrays

```
SELECT
 ARRAY_LENGTH(bikerides) as num_items
 , bikerides[ OFFSET(0) ].gender as first_gender
FROM
(SELECT
 [
   STRUCT('male' as gender, [9306602, 3955871] as numtrips)
   , STRUCT('female' as gender, [3236735, 1260893] as numtrips)
 ] AS bikerides)
```

Row	num_items	first_gender
1	2	male

Standard SQL | UNNEST

```
SELECT * from UNNEST(
 [
   STRUCT('male' as gender, [9306602, 3955871] as numtrips)
   , STRUCT('female' as gender, [3236735, 1260893] as numtrips)
 ])
```

Row	gender	numtrips
1	male	9306602
		3955871
2	female	3236735
		1260893

Standard SQL | UNNEST

```
SELECT numtrips from UNNEST(
 [
   STRUCT('male' as gender, [9306602, 3955871] as numtrips)
   , STRUCT('female' as gender, [3236735, 1260893] as numtrips)
 ])
```

Row	numtrips
1	9306602
	3955871
2	3236735
	1260893

SQL Avanzado

Joining Tables

You can actually query data from multiple tables with a single query.

It's a little like a "Vlookup" in Excel.

employeeId	firstName	lastName	jobCode	salary
001	Montgomery	Burns	AAA	1000000
003	Lenny	None	SFI	60000
002	Homer	Simpson	SFI	15000

jobCode	jobDesc	jobArea
AAA	Overload	Executive
SFI	Safety Inspector	Safety
WBE	Worker Bee	Operations

Joining Tables

Suppose we want the employee name and the job description of every worker.

employeeId	firstName	lastName	jobCode	salary
001	Montgomery	Burns	AAA	1000000
003	Lenny	None	SFI	60000
002	Homer	Simpson	SFI	15000
004	Carl	Carlson	WBE	50000

jobCode	jobDesc	jobArea
AAA	Overload	Executive
SFI	Safety Inspector	Safety
WBE	Worker Bee	Operations

Joining Tables

SELECT
Employee.firstName,
Employee.lastName,
JobDetails.JobDesc
FROM
Employee
INNER JOIN
JobDetails
ON Employee.jobCode =
JobDetails.jobCode

employeeId	firstName	lastName	jobCode	salary
001	Montgomery	Burns	AAA	1000000
003	Lenny	None	SFI	60000
002	Homer	Simpson	SFI	15000
004	Carl	Carlson	WBE	50000

jobCode	jobDesc	jobArea
AAA	Overload	Executive
SFI	Safety Inspector	Safety
WBE	Worker Bee	Operations

Joining Tables

```
SELECT
    Employee.firstName,
    Employee.lastName,
    JobDetails.JobDesc
FROM
        Employee
INNER JOIN
        JobDetails
        ON Employee.jobCode =
JobDetails.jobCode
```

Employee, JobDetails prefix, when selecting from more than one table

employeeId	firstName	lastName	jobCode	salary
001	Montgomery	Burns	AAA	1000000
003	Lenny	None	SFI	60000
002	Homer	Simpson	SFI	15000
004	Carl	Carlson	WBE	50000

jobCode	jobDesc	jobArea
AAA	Overload	Executive
SFI	Safety Inspector	Safety
WBE	Worker Bee	Operations

Joining Tables

SELECT
Employee.firstName,
Employee.lastName,
JobDetails.JobDesc
FROM
Empl
INNER JOIN
JobDe
ON Employee.jobCode =
JobDetails.jobCode

Different kinds of joins.

employeeId	firstName	lastName	jobCode	salary
001	Montgomery	Burns	AAA	1000000
003	Lenny	None	SFI	60000
002	Homer	Simpson	SFI	15000
004	Carl	Carlson	WBE	50000

jobCode	jobDesc	jobArea
AAA	Overload	Executive
SFI	Safety Inspector	Safety
WBE	Worker Bee	Operations

Joining Tables

SELECT
Employee.firstName,
Employee.lastName,
JobDetails.JobDesc
FROM
Employee
INNER JOIN
JobDetails
ON Employee.jobCode =
JobDetails.jobCode

employeeId	firstName	lastName	jobCode	salary
001	Montgomery	Burns	AAA	1000000
003	Lenny	None	SFI	60000
002	Homer	Simpson	SFI	15000
004	Carl	Carlson	WBE	50000

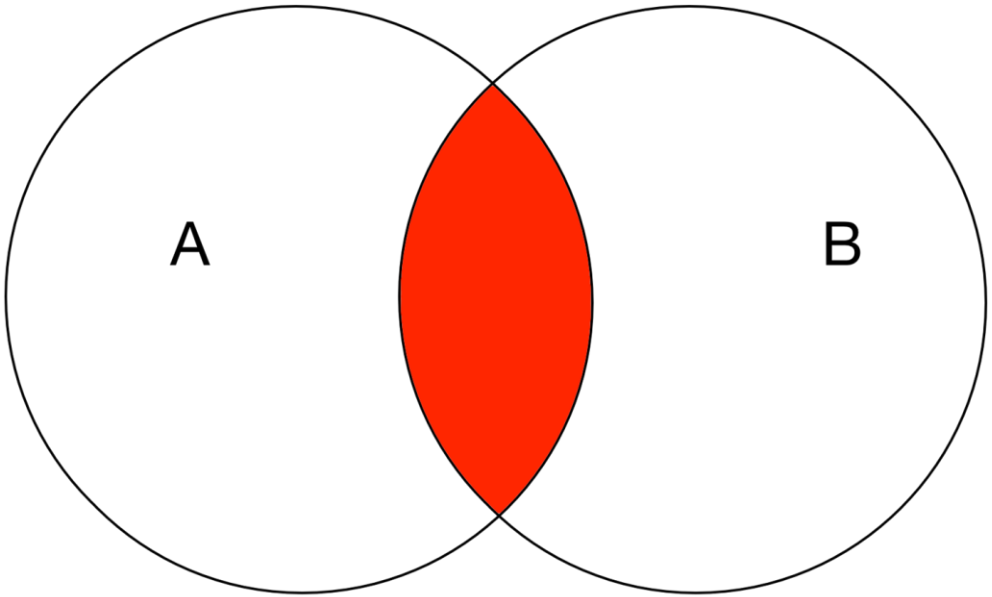
Specify the criteria you want to join the tables by

AAA	Overload	Executive
SFI	Safety Inspector	Safety
WBE	Worker Bee	Operations

Inner Join

Only returns results when both table satisfy the join condition

It's the default



Joining Tables

Who works in Springfield?

```
SELECT
    e.firstName,
    e.lastName,
    s.address
FROM
        Employee e
INNER JOIN
        Site s
        ON e.SiteID = s.SiteID
WHERE
        s.Address = 'Springfield';
```

Employee

employeeId	firstName	lastName	jobCode	siteID
001	Montgomery	Burns	AAA	SPF
002	Homer	Simpson	SFI	SPF
003	Lenny	None	SFI	MTV

Site

SiteID	Address	State
SPF	Springfield	UNK
MTV	Mountain View	CA
NYC	New York	NY

Joining Tables

Who works in Springfield?

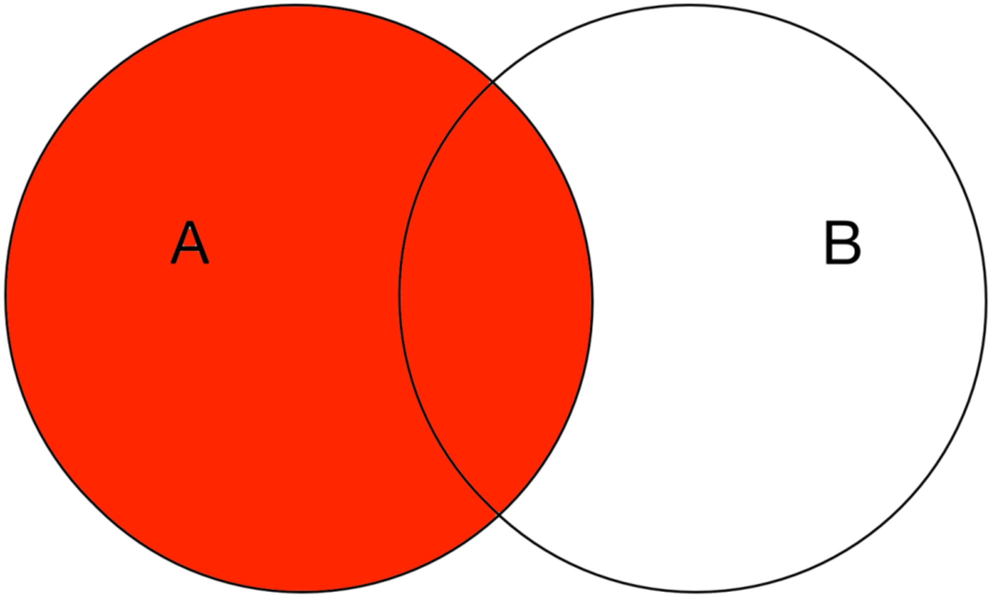
```
SELECT
    e.firstName AS firstName,
    e.lastName AS lastName,
    s.address AS address
FROM
        Employee e
INNER JOIN
        Site s
        ON e.SiteID = s.SiteID
WHERE
        s.Address = 'Springfield';
```

firstName	lastName	Address
Montgomery	Burns	Springfield
Homer	Simpson	Springfield

Break

Left Join

Returns **ALL** records from the left table and only those from the right table which satisfy the join condition



Left Join

We want the workers listed for EVERY job code

```
SELECT
        j.jobCode,
        j.JobDesc,
        e.firstName,
        e.lastName
FROM
                JobDetails j
LEFT JOIN
                Employee e
                ON j.JobCode = e.JobCode
```

Employee

employeeId	firstName	lastName	jobCode	siteID
001	Montgomery	Burns	AAA	SPF
002	Homer	Simpson	SFI	SPF
003	Lenny	None	SFI	MTV
004	Carl	Carlson	WBE	NYC

JobDetails

jobCode	jobDesc	jobArea
EXF	Executive Dogsbody	Dogsbody
SFI	Safety Inspector	Safety
WBE	Worker Bee	Operations

Left Join

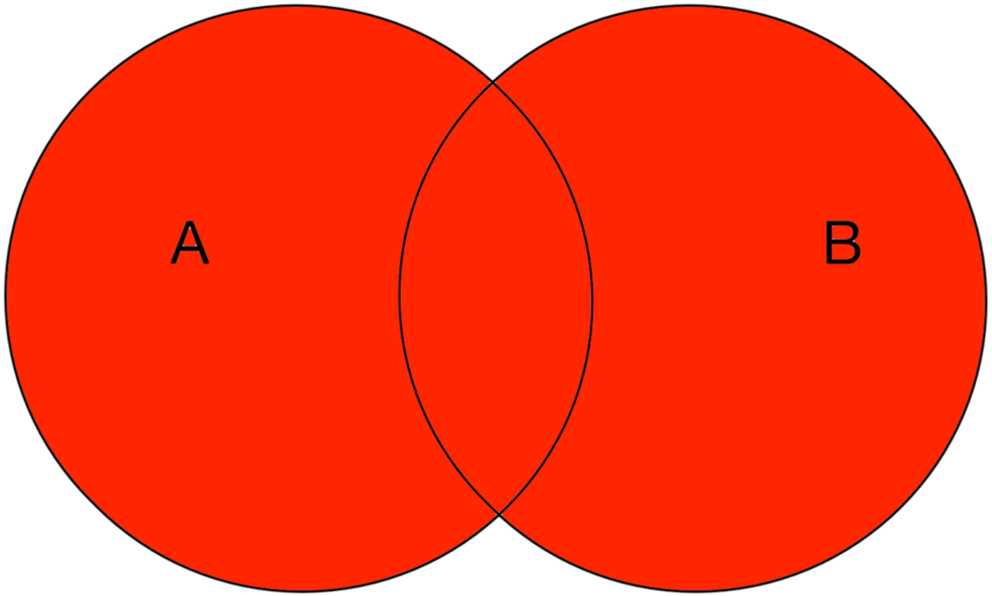
We want the workers listed for EVERY job code

```
SELECT
        j.jobCode,
        j.JobDesc,
        e.firstName,
        e.lastName
FROM
                JobDetails j
LEFT JOIN
                Employee e
                ON j.JobCode = e.JobCode
```

jobCode	jobDesc	firstName	lastName
EXF	Executive Dogsbody	NULL	NULL
SFI	Safety Inspector	Lenny	None
WBE	Worker Bee	Carl	Carlson
SFI	Safety Inspector	Homer	Simpson

Full Outer Join

Returns **ALL** records from both tables. Records which do not satisfy the join condition will have **NULL** values in the joined fields.



FULL OUTER JOIN

```
SELECT
        j.jobCode,
        j.JobDesc,
        e.firstName,
        e.lastName
FROM
                JobDetails j
FULL OUTER JOIN
                Employee e
                ON j.JobCode = e.JobCode
```

jobCode	jobDesc	firstName	lastName
EXF	Executive Dogsbody	NULL	NULL
SFI	Safety Inspector	Lenny	None
WBE	Worker Bee	Carl	Carlson
SFI	Safety Inspector	Homer	Simpson
NULL	NULL	Montgomery	Burns

Cross Join

Cartesian Product

```
WITH winners AS (
 SELECT 'John' as person, '100m' as event
 UNION ALL SELECT 'Hiroshi', '200m'
 UNION ALL SELECT 'Sita', '400m'
),
gifts AS (
 SELECT 'Google Home' as gift, '100m' as event
 UNION ALL SELECT 'Google Hub', '200m'
 UNION ALL SELECT 'Pixel3', '400m'
)
SELECT winners.*, gifts.gift
FROM winners
JOIN gifts USING (event)
-- JOIN gifts ON gifts.event = winners.event
```

Row	person	event	gift	
1	John	100m	Google Home	
2	Hiroshi	200m	Google Hub	
3	Sita	400m	Pixel3	

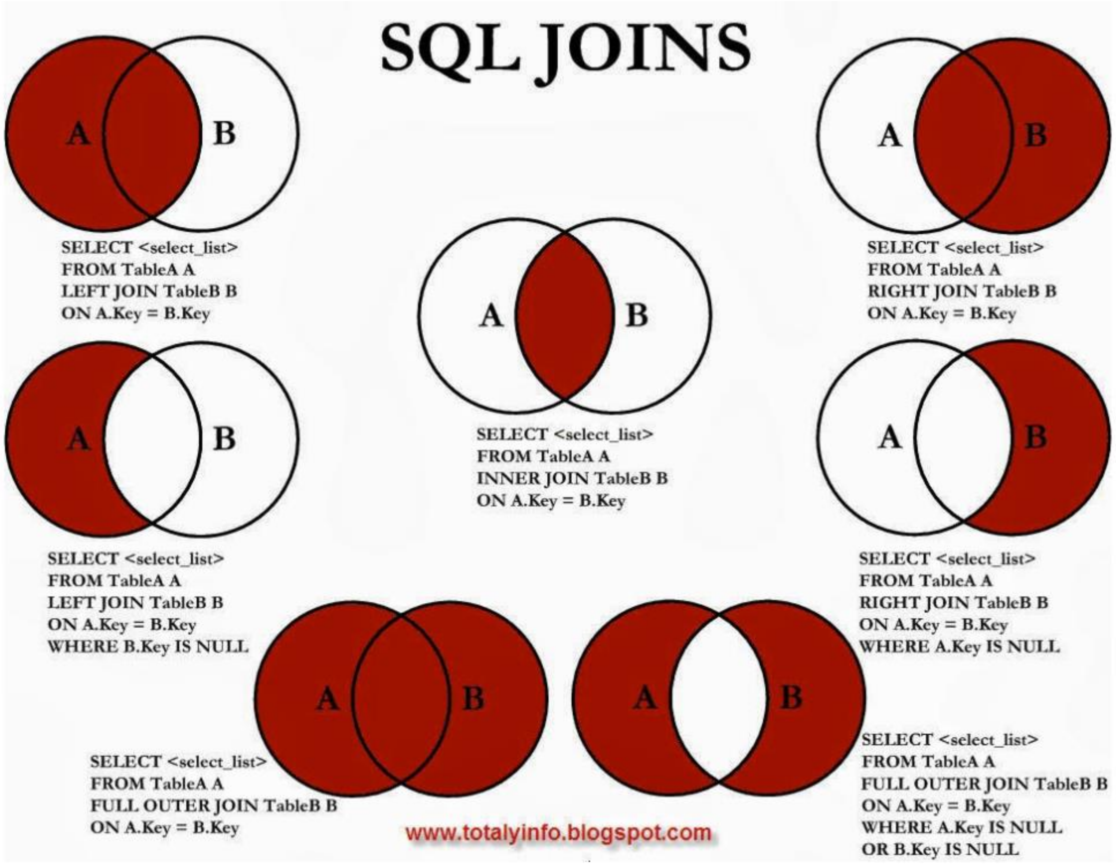
Cross Join

Cartesian Product

```
WITH winners AS (
 SELECT 'John' as person, '100m' as event
 UNION ALL SELECT 'Hiroshi', '200m'
 UNION ALL SELECT 'Sita', '400m'
),
gifts AS (
 SELECT 'Google Home' as gift
 UNION ALL SELECT 'Google Hub'
 UNION ALL SELECT 'Pixel3'
)
SELECT person, gift
FROM winners
CROSS JOIN gifts
```

Row	person	gift
1	John	Google Home
2	John	Google Hub
3	John	Pixel3
4	Hiroshi	Google Home
5	Hiroshi	Google Hub
6	Hiroshi	Pixel3
7	Sita	Google Home
8	Sita	Google Hub
9	Sita	Pixel3

Standard SQL | Intro to Joins



<table>
<tr><th>Syntax</th><th>What happens</th><th>Output</th></tr>
<tr>
<td>SELECT person, gift
FROM winners
INNER JOIN gifts
ON winners.event = gifts.event</td>
<td>Only rows that meet the join condition are retained</td>
<td><table>
<tr><th>Row</th><th>person</th><th>gift</th></tr>
<tr><td>1</td><td>John</td><td>Google Home</td></tr>
<tr><td>2</td><td>Hiroshi</td><td>Google Hub</td></tr>
<tr><td>3</td><td>Sita</td><td>Pixel3</td></tr>
</table></td>
</tr>
<tr>
<td>SELECT person, gift
FROM winners
FULL OUTER JOIN gifts
ON winners.event = gifts.event</td>
<td>All rows are retained even if the join condition is not met</td>
<td><table>
<tr><th>Row</th><th>person</th><th>gift</th></tr>
<tr><td>1</td><td>John</td><td>Google Home</td></tr>
<tr><td>2</td><td>Hiroshi</td><td>Google Hub</td></tr>
<tr><td>3</td><td>Sita</td><td>Pixel3</td></tr>
<tr><td>4</td><td>Kwame</td><td>null</td></tr>
<tr><td>5</td><td>null</td><td>Google Mini</td></tr>
</table></td>
</tr>
<tr>
<td>SELECT person, gift
FROM winners
LEFT OUTER JOIN gifts
ON winners.event = gifts.event</td>
<td>All the winners are retained, but some gifts are discarded</td>
<td><table>
<tr><th>Row</th><th>person</th><th>gift</th></tr>
<tr><td>1</td><td>John</td><td>Google Home</td></tr>
<tr><td>2</td><td>Hiroshi</td><td>Google Hub</td></tr>
<tr><td>3</td><td>Sita</td><td>Pixel3</td></tr>
<tr><td>4</td><td>Kwame</td><td>null</td></tr>
</table></td>
</tr>
<tr>
<td>SELECT person, gift
FROM winners
RIGHT OUTER JOIN gifts
ON winners.event = gifts.event</td>
<td>All the gifts are retained, but some winners aren’t</td>
<td><table>
<tr><th>Row</th><th>person</th><th>gift</th></tr>
<tr><td>1</td><td>John</td><td>Google Home</td></tr>
<tr><td>2</td><td>Hiroshi</td><td>Google Hub</td></tr>
<tr><td>3</td><td>Sita</td><td>Pixel3</td></tr>
<tr><td>4</td><td>null</td><td>Google Mini</td></tr>
</table></td>
</tr>
</table>

Standard SQL | Intro to temporary tables

Temporary tables (link)

```
with WY_state as (select ein as filter from `bigquery-public-
data.irs_990.irs_990_ein` where state = "WY")
select
ein as ein,
noemplyeesw3cnt as nom_of_employees

    from `bigquery-public-data.irs_990.irs_990_2016`

    where ein in (select filter from WY_state)
```

Standard SQL | Intro to Functions

Analytical Functions - Rank (link)

`PARTITION BY department`

firstname	department	startdate
Andrew	1	1/23/1999
Jacob	1	7/11/1990
Daniel	2	6/24/2004
Anna	1	10/7/2001
Pierre	1	2/22/2009
Ruth	2	6/6/1998
Anthony	1	11/29/1995
Isabella	2	9/28/1997
Jose	2	3/17/2013

firstname	department	startdate
Andrew	1	1/23/1999
Jacob	1	7/11/1990
Anna	1	10/7/2001
Pierre	1	2/22/2009
Anthony	1	11/29/1995

firstname	department	startdate
Ruth	2	6/6/1998
Daniel	2	6/24/2004
Jose	2	3/17/2013
Isabella	2	9/28/1997

`ORDER BY startdate`

firstname	department	startdate
Jacob	1	7/11/1990
Anthony	1	11/29/1995
Andrew	1	1/23/1999
Anna	1	10/7/2001
Pierre	1	2/22/2009

firstname	department	startdate
Isabella	2	9/28/1997
Daniel	2	6/24/2004
Jose	2	3/17/2013
Ruth	2	6/6/2013

`RANK()`

firstname	department	startdate	rank
Jacob	1	7/11/1990	1
Anthony	1	11/29/1995	2
Andrew	1	1/23/1999	3
Anna	1	10/7/2001	4
Pierre	1	2/22/2009	5

firstname	department	startdate	rank
Isabella	2	9/28/1997	1
Daniel	2	6/24/2004	2
Jose	2	3/17/2013	3
Ruth	2	6/6/2013	4

Standard SQL | Intro to Functions

Analytical Functions - Rank (link)

```
SELECT firstname, department, startdate,
  RANK() OVER ( PARTITION BY department ORDER
BY startdate ) AS rank
FROM Employees;
```

Standard SQL | Intro to Functions

Navigation - Rolling average of 10 last employees (link)

```
SELECT firstname, department, startdate,
  Sum(salary) OVER ( PARTITION BY department
ORDER BY startdate asc ROWS BETWEEN 9
PRECEDING AND 0 FOLLOWING) AS
salary_rolloing_sum
FROM Employees;
```

Standard SQL | Intro to Functions

Navigation functions - Lead or next joined employee (link)

```
SELECT firstname, department, startdate,
  Lead(firstname) OVER ( PARTITION BY department
ORDER BY startdate asc) AS next_employee
FROM Employees;
```

Hands On Exercise K
Analytical functions

Standard SQL | Intro to Functions

Analytical Functions - Rank (link)

Extraer la ONG con más empleados de cada estado

```
with ranked as (

SELECT
t1.ein as ein,
t2.name as name,
t1.noemplyeesw3cnt as nom_of_employees,
t2.state as state,
rank() over (partition by state order by t1.noemplyeesw3cnt desc) as rank
    from `bigquery-public-data.irs_990.irs_990_2015` as t1
     INNER JOIN `bigquery-public-data.irs_990.irs_990_ein`as t2
    USING(ein)
    group by 1,2,3,4
    )

    select * from ranked where rank = 1
    order by nom_of_employees desc
```

Standard SQL | Intro to Functions

Navigation Functions - Rank (link)

Extraer el número de empleados de la siguiente ONG más grande

```
with ranked as (

SELECT
t1.ein as ein,
t2.name as name,
t1.noemplyeesw3cnt as nom_of_employees,
t2.state as state,
rank() over (partition by state order by t1.noemplyeesw3cnt desc) as rank,

lead(t1.noemplyeesw3cnt,1) over (partition by state order by t1.noemplyeesw3cnt desc) as next_num_employees
    from `bigquery-public-data.irs_990.irs_990_2015` as t1
     INNER JOIN `bigquery-public-data.irs_990.irs_990_ein`as t2
    USING(ein)
    group by 1,2,3,4
    )

    select * from ranked where state = 'CA' and rank = 1
    order by nom_of_employees desc
```

Standard SQL | Intro to Functions

Navigation Functions - Rank (link)

Extraer el número medio de empleados de las top 10 ongs

```
with ranked as (
SELECT
t1.ein as ein,
t2.name as name,
t1.noemplyeesw3cnt as nom_of_employees,
t2.state as state,
rank() over (partition by state order by t1.noemplyeesw3cnt desc) as rank,
lead(t1.noemplyeesw3cnt,1) over (partition by state order by t1.noemplyeesw3cnt desc) as next_num_employees,
Sum(t1.noemplyeesw3cnt) OVER ( PARTITION BY state ORDER BY t1.noemplyeesw3cnt asc  ROWS BETWEEN 9 PRECEDING AN
0 FOLLOWING)/10 AS employee_rolloing_sum
    from `bigquery-public-data.irs_990.irs_990_2015` as t1
     INNER JOIN `bigquery-public-data.irs_990.irs_990_ein`as t2
    USING(ein)
    group by 1,2,3,4
    )
    select * from ranked --where state = 'CA' --
    where rank = 1
    order by nom_of_employees desc
```

Standard SQL | Arrays & structs

Arrays (link)

select ['a','b','c'] as array_sample

Row	array_sample
1	a
	b
	c

with sample as (select ['a','b','c'] as array_sample)
select array_length(array_sample) as array_sample_length from sample

Row	array_sample_length
1	3

select ['a','b','c'] as array_sample, 'field' as field → BigQuery Creates Nested Field Structures

Results Details

Row	array_sample	field
1	a	field
	b	
	c	

Standard SQL | Arrays & structs

Unnest (link)

with table as (select ['a','b','c'] as array_sample, 'field' as field)

select array_sample,field from table,unnest(array_sample) as array_sample

Row	array_sample	field
1	a	field
2	b	field
3	c	field

Create array (link)

with table as (select 'a' as field union all select 'b' as field union all select 'c' as field)

select array_agg(field order by field desc) as array_created from table

Row	array_created
1	a
	b
	c

Standard SQL | Arrays & structs

Structs (link)

select struct(35 as age, ['alicia','pedro'] as names) as info

Row	info.age	info.names
1	35	alicia
		pedro

select struct(35 as age, 'pedro' as names, ['p1','p2','p3'] as products) as info

Row	info.age	info.names	info.products
1	35	pedro	p1
			p2
			p3

Arrays of Structs:
select

[struct(35 as age, 'pedro' as names, ['p1','p2','p3'] as products), struct(30 as age, 'maria' as names, ['p1','p6','p8'] as products)] as info

Row	info.age	info.names	info.products
1	35	pedro	p1
			p2
			p3
	30	maria	p1
			p6
			p8

Standard SQL | Arrays & structs

Arrays & Structs - filter customers that bought p1 (link)

```
with table as (

select

[struct(35 as age, 'pedro' as names, ['p1','p2','p3'] as products),
struct(30 as age, 'maria' as names, ['p1','p6','p8'] as products),
struct(37 as age, 'juan' as names, ['p2','p7','p9'] as products)
] as info)
select
names
from table
, unnest(info) as info
where 'p1' in unnest(info.products)
```

Standard SQL | Declare variables

Declare and set variables (link)

```
DECLARE target_word STRING DEFAULT 'bespoke';
DECLARE corpus_count, num_palabra INT64;

SET (corpus_count, num_palabra) = (
  SELECT AS STRUCT COUNT(DISTINCT corpus), SUM(word_count)
  FROM `bigquery-public-data`.samples.shakespeare
  WHERE LOWER(word) = target_word
);

SELECT
  FORMAT('Found %d occurrences of "%s" across %d Shakespeare works',
         num_palabra, target_word, corpus_count) AS result;
```

Standard SQL | Run various ordered scripts

Run various scripts (link)

```
DECLARE x INT64 DEFAULT 10;
BEGIN
  DECLARE y INT64;
  SET y = x;
  SELECT y;

SELECT x;

END;
```

Standard SQL | if conditions

Run with if conditions (link)

```
DECLARE target_product_id INT64 DEFAULT 3;

IF EXISTS (

with products as ( select product_id,product_name from (select 1  as product_id, 'a' as
product_name UNION ALL
select 2  as product_id, 'b' as product_name UNION ALL
select 3  as product_id, 'c' as product_name ) )
SELECT target_product_id FROM products
        WHERE product_id = target_product_id) THEN
  SELECT CONCAT('found product ', CAST(target_product_id AS STRING));
ELSE
  SELECT CONCAT('did not find product ', CAST(target_product_id AS STRING));
END IF;
```

Standard SQL | Loops

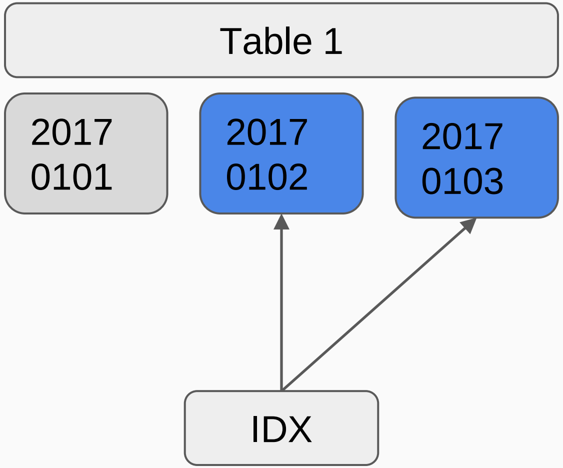
Loops - Create loops in BigQuery (link)

```
DECLARE x INT64 DEFAULT 0;
LOOP
  SET x = x + 1;
  IF x >= 10 THEN
    LEAVE;
  END IF;
END LOOP;
SELECT x;
```

BigQuery

Particionado & Clustering

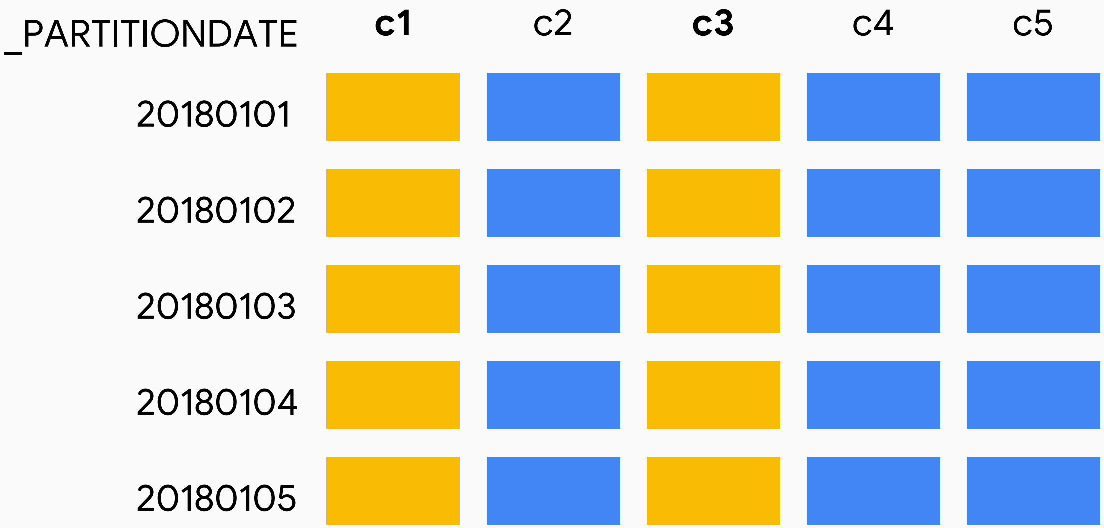
Design Pattern: Partitioning



SELECT … WHERE date >= “20170102”

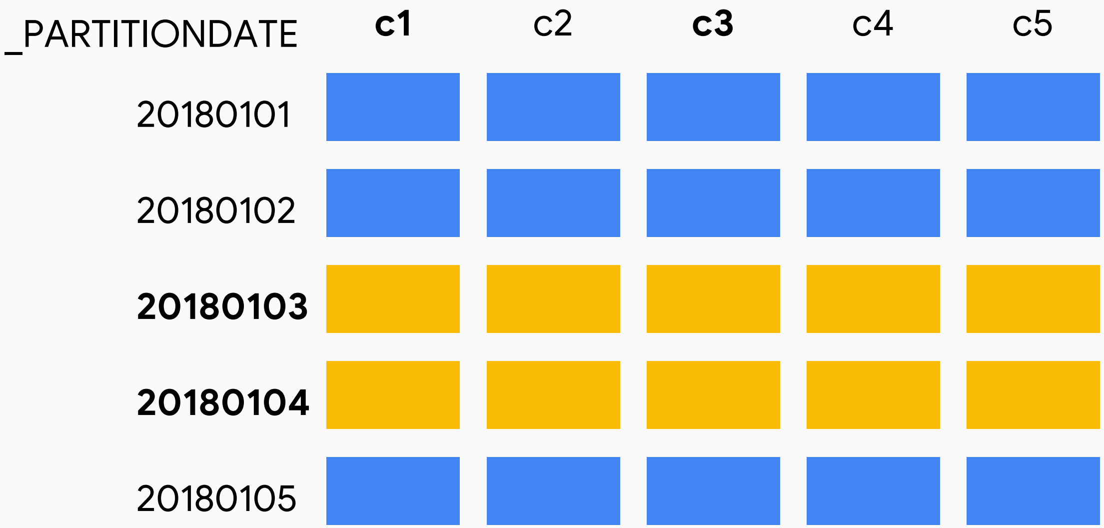
- Simpler Data Management
  - fewer tables
  - consistent schema
- Faster Queries
  - less metadata overhead
  - data pruning
- Less Expensive
  - filtering reduces cost while improving performance
- Available partitioning: Ingestion Time, Date/Timestamp, Integer.

Ingestion time-based partitioning



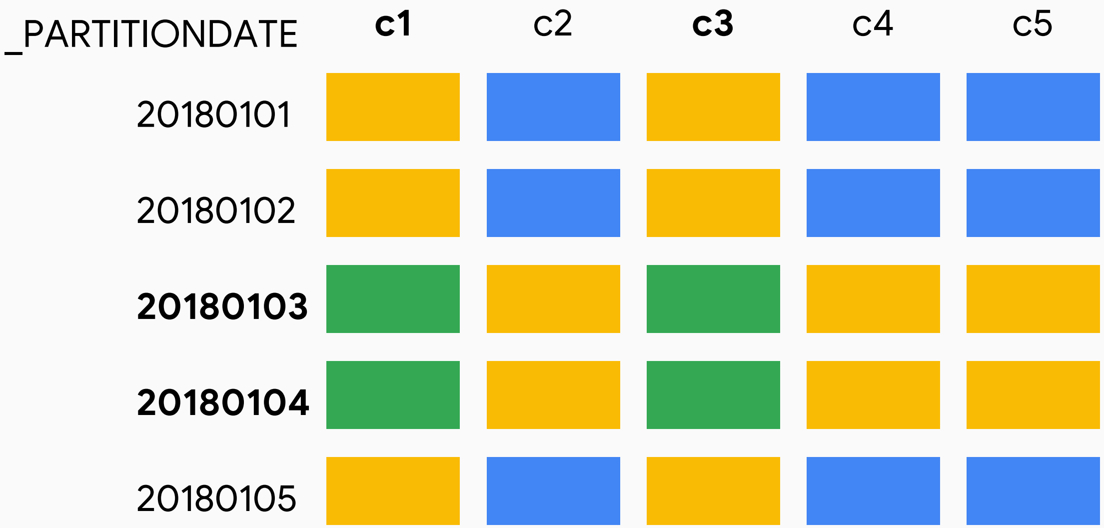
SELECT **c1, c3** FROM ...

Ingestion time-based partitioning



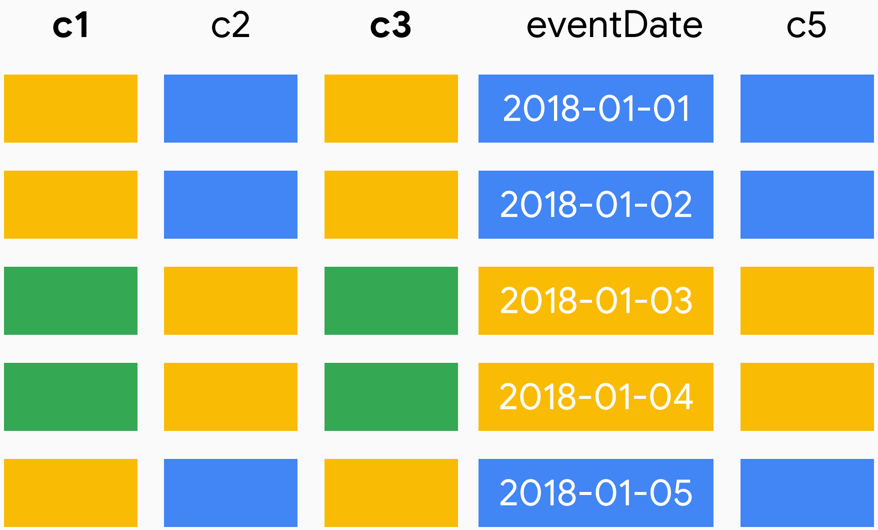
SELECT c1, c3 FROM ...
WHERE _PARTITIONDATE BETWEEN “2018-01-03” AND “2018-01-04”

Ingestion time-based partitioning



SELECT c1, c3 FROM ...
WHERE _PARTITIONDATE BETWEEN “2018-01-03” AND “2018-01-04”

Column-based partitioning

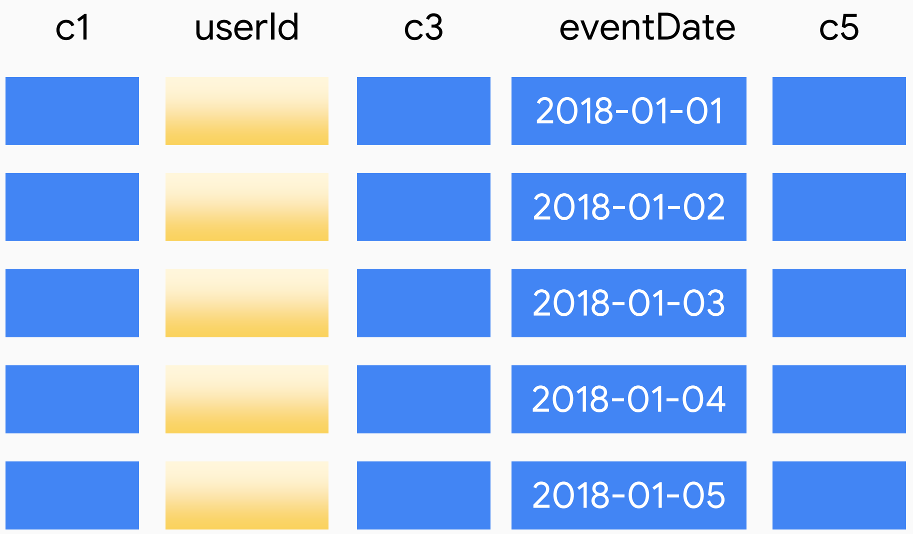


SELECT c1, c3 FROM ...
WHERE eventDate BETWEEN “2018-01-03” AND “2018-01-04”

Partitioning limits

- Maximum number of partitions per partitioned table — 4,000
- Maximum number of partitions modified by a single job — 4,000
- Maximum number of partition modifications per ingestion time partitioned table — 5,000
- Maximum number of partition modifications per column partitioned table — 30,000
- Maximum rate of partition operations — 50 partition operations every 10 seconds

Clustering



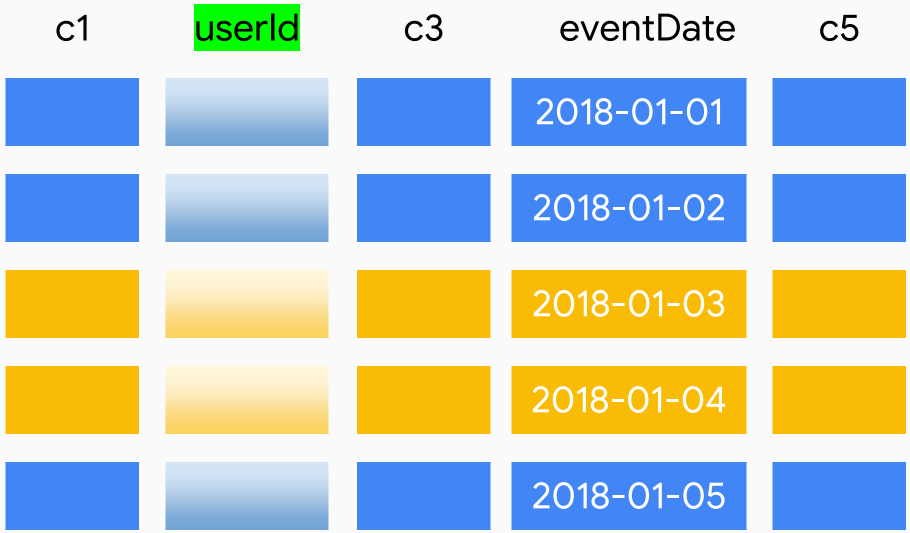
SELECT c1, c3 FROM ... WHERE userId BETWEEN 52 and 63
AND eventDate BETWEEN “2018-01-03” AND “2018-01-05”

Clustering

c1	userId	c3	eventDate	c5
			2018-01-01	
			2018-01-02	
			2018-01-03	
			2018-01-04	
			2018-01-05	

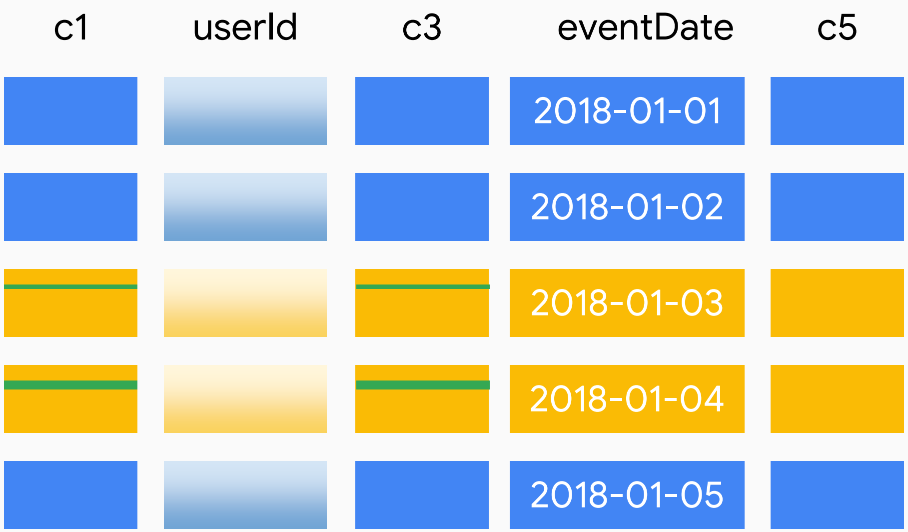
SELECT c1, c3 FROM ... WHERE userId BETWEEN 52 and 63
AND eventDate BETWEEN “2018-01-03” AND “2018-01-05”

Clustering



SELECT c1, c3 FROM ... WHERE userId BETWEEN 52 and 63
AND eventDate BETWEEN “2018-01-03” AND “2018-01-05”

Clustering



SELECT c1, c3 FROM ... WHERE userId BETWEEN 52 and 63
AND eventDate BETWEEN “2018-01-03” AND “2018-01-05”

Clustering

Filter on a high-cardinality column

Less expensive:

- Only pay for scanning blocks with cluster key

Faster:

- Data is stored sorted within the partition

Easier to manage:

- No need to manually shard tables

BigQuery - Partitioning & Clustering

Partitioning

Cardinality: Less than 4k

Dry Run Pricing: Available

Query Pricing: Exact

Performance Overhead: Small

Data Management: Like a Table

Clustering

Cardinality: Unlimited

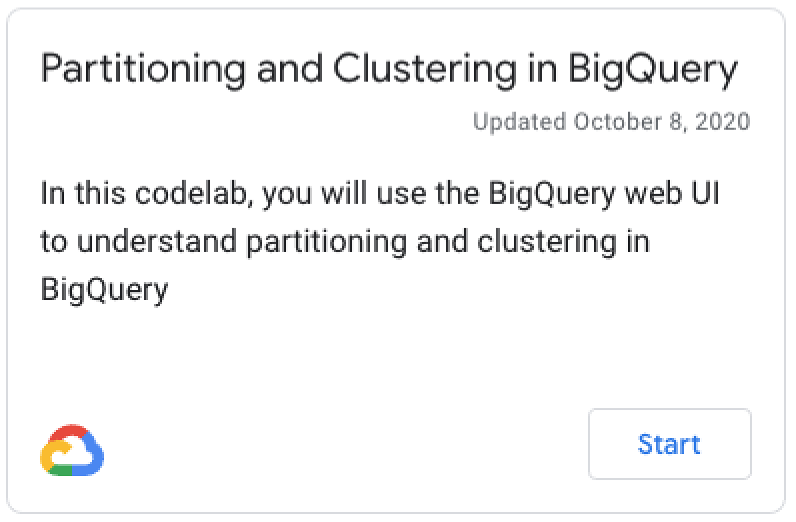
Dry Run Pricing: Not available

Query Pricing: Best Effort

Performance Overhead: None

Data management: Use DML

BigQuery - Partitioning & Clustering



BigQuery Best Practices

BigQuery - Query Settings

Query settings

Query engine

- (•) BigQuery engine
- ( ) Cloud Dataflow engine
  Deploy your data processing pipelines on the Cloud Dataflow service.

Destination

- (•) Save query results in a temporary table
- ( ) Set a destination table for query results

Project name

iyv-cloud

Dataset name

bicing

Table name

Letters, numbers, underscores, and template system characters allowed

Destination table write preference

- ( ) Write if empty
- ( ) Append to table
- ( ) Overwrite table

Results size

- [ ] Allow large results (no size limit)

Resource management

Job priority

- (•) Interactive
- ( ) Batch

Cache preference

- [x] Use cached results

Additional settings

SQL dialect

- (•) Standard
- ( ) Legacy

Processing location

Auto-select

Advanced options

Encryption

Data is encrypted automatically. Select an encryption key management solution.

- (•) Google-managed key
  No configuration required
- ( ) Customer-managed key
  Manage via Google Cloud Key Management Service

Maximum bytes billed

Control projection - Avoid `SELECT *`

Best practice: Control projection - Query only the columns that you need.

Projection refers to the number of columns that are read by your query. Projecting excess columns incurs additional (wasted) I/O and materialization (writing results).

Using `SELECT *` is the most expensive way to query data. When you use `SELECT *`, BigQuery does a full scan of every column in the table.

Prune partitioned queries

Best practice: When querying a time-partitioned table, use the `_PARTITIONTIME` pseudo column to filter the partitions.

When you query partitioned tables, use the `_PARTITIONTIME` pseudo column. Filtering the data using `_PARTITIONTIME`allows you to specify a date or range of dates. For example, the following `WHERE` clause uses the `_PARTITIONTIME`pseudo column to specify partitions between January 1, 2016 and January 31, 2016:

```
WHERE _PARTITIONTIME
BETWEEN TIMESTAMP(“20160101”)
    AND TIMESTAMP(“20160131”)
```

The query processes data only in the partitions that are indicated by the date range, reducing the amount of input data. Filtering your partitions improves query performance and reduces costs.

BigQuery - Best Practises - Input data and Data Sources

Using nested and repeated fields

BigQuery doesn't require a completely flat denormalization. You can use nested and repeated fields to maintain relationships.

- Nesting data (STRUCT)
  - Nesting data allows you to represent foreign entities inline.
  - Querying nested data uses "dot" syntax to reference leaf fields, which is similar to the syntax using a join.
- Repeated data (ARRAY)
  - Creating a field of type RECORD with the mode set to REPEATED allows you to preserve a 1:many relationship inline (so long as the relationship isn't high cardinality).
  - With repeated data, shuffling is not necessary.
- Nested and repeated data (ARRAY of STRUCTs)
  - Nesting and repetition complement each other.
  - For example, in a table of transaction records, you could include an array of line item STRUCTs.

BigQuery - Best Practises - Input data and Data Sources

Use external data sources appropriately

Best practice: If query performance is a top priority, do not use an external data source.

Querying tables in BigQuery managed storage is typically much faster than querying external tables in Google Cloud Storage, Google Drive, or Google Cloud Bigtable.

Reduce data before using a JOIN

Best practice: Reduce the amount of data that is processed before a `JOIN` clause.

Trim the data as early in the query as possible, before the query performs a `JOIN`. If you reduce data early in the processing cycle, shuffling and other complex operations only execute on the data that you need.

Do not treat `WITH` clauses as prepared statements

Best practice: Use `WITH` clauses primarily for readability.

`WITH` clauses are used primarily for readability because they are not materialized. For example, placing all your queries in `WITH` clauses and then running `UNION ALL` is a misuse of the `WITH` clause. If a query appears in more than one `WITH` clause, it executes in each clause.

Avoid tables sharded by date

Best practice: Do not use tables sharded by date (also called date-named tables) in place of time-partitioned tables.

Partitioned Tables perform better than date-named tables. When you create tables sharded by date, BigQuery must maintain a copy of the schema and metadata for each date-named table. Also, when date-named tables are used, BigQuery might be required to verify permissions for each queried table. This practice also adds to query overhead and impacts query performance.

Avoid oversharding tables

Best practice: Avoid creating too many table shards. If you are sharding tables by date, use time-partitioned tables instead.

Table sharding refers to dividing large datasets into separate tables and adding a suffix to each table name. If you are sharding tables by date, use time-partitioned tables instead.

Because of the low cost of BigQuery storage, you do not need to optimize your tables for cost as you would in a relational database system. Creating a large number of table shards has performance impacts that outweigh any cost benefits.

Sharded tables require BigQuery to maintain schema, metadata, and permissions for each shard. Because of the added overhead required to maintain information on each shard, oversharding tables can impact query performance.

Avoid repeatedly transforming data via SQL queries

Best practice: If you are using SQL to perform ETL operations, avoid situations where you are repeatedly transforming the same data.

For example, if you are using SQL to trim strings or extract data by using regular expressions, it is more performant to materialize the transformed results in a destination table. Functions like regular expressions require additional computation. Querying the destination table without the added transformation overhead is much more efficient.

Avoid JavaScript user-defined functions

Best practice: Avoid using JavaScript user-defined functions. Use native UDFs instead.

Calling a JavaScript UDF requires the instantiation of a subprocess. Spinning up this process and running the UDF directly impacts query performance. If possible, use a native (SQL) UDF instead.

Use approximate aggregation functions

Best practice: If your use case supports it, use an approximate aggregation function.

If the SQL aggregation function you're using has an equivalent approximation function, the approximation function will yield faster query performance. For example, instead of using `COUNT(DISTINCT)`, use `APPROX_COUNT_DISTINCT()`. For more information, see approximate aggregation functions in the standard SQL reference.

You can also use HyperLogLog++ functions to do approximations (including custom approximate aggregations). For more information, see HyperLogLog functions in the standard SQL reference.

Order query operations to maximize performance

Best practice: Use `ORDER BY` only in the outermost query or within window clauses (analytic functions). Push complex operations to the end of the query.

If you need to sort data, filter first to reduce the number of values that you need to sort. If you sort your data first, you sort much more data than is necessary. It is preferable to sort on a subset of data than to sort all the data and apply a `LIMIT` clause.

When you use an `ORDER BY` clause, it should appear only in the outermost query. Placing an `ORDER BY` clause in the middle of a query greatly impacts performance unless it is being used in a window (analytic) function.

Another technique for ordering your query is to push complex operations, such as regular expressions and mathematical functions to the end of the query. Again, this technique allows the data to be pruned as much as possible before the complex operations are performed.

Optimize your join patterns

Best practice: For queries that join data from multiple tables, optimize your join patterns. Start with the largest table.

When you create a query by using a `JOIN`, consider the order in which you are merging the data. The standard SQL query optimizer can determine which table should be on which side of the join, but it is still recommended to order your joined tables appropriately. The best practice is to place the largest table first, followed by the smallest, and then by decreasing size.

When you have a large table as the left side of the `JOIN` and a small one on the right side of the `JOIN`, a broadcast join is created. A broadcast join sends all the data in the smaller table to each slot that processes the larger table. It is advisable to perform the broadcast join first.

To view the size of the tables in your `JOIN`, see getting information about tables.

Avoid repeated joins and subqueries

Best practice: Avoid repeatedly joining the same tables and using the same subqueries.

If you are repeatedly joining the same tables, consider revisting your schema. Instead of repeatedly joining the data, it might be more performant for you to use nested repeated data to represent the relationships. Nested repeated data saves you the performance impact of the communication bandwidth that is required by a join. It also saves you the I/O costs that are incurred by repeatedly reading and writing the same data. For more information, see using nested and repeated fields.

Similarly, repeating the same subqueries impacts performance through repetitive query processing. If you are using the same subqueries in multiple queries, consider materializing the subquery results in a table. Then consume the materialized data in your queries.

Materializing your subquery results improves performance and reduces the overall amount of data that is read and written by BigQuery. The small cost of storing the materialized data outweighs the performance impact of repeated I/O and query processing.

BigQuery - Best Practises - Managing Query Outputs

Carefully consider materializing large result sets

Best practice: Carefully consider materializing large result sets to a destination table. Writing large result sets has performance and cost impacts.

BigQuery limits cached results to approximately 128MB compressed. Queries that return larger results overtake this limit and frequently result in the following error: Response too large.

This error often occurs when you select a large number of fields from a table with a considerable amount of data. Issues writing cached results can also occur in ETL-style queries that normalize data without reduction or aggregation.

You can overcome the limitation on cached result size by:

- Using filters to limit the result set
- Using a `LIMIT` clause to reduce the result set, especially if you using an `ORDER BY` clause
- Writing the output data to a destination table

Use a LIMIT clause with large sorts

Best practice: If you are sorting a very large number of values, use a `LIMIT` clause.

Writing results for a query with an `ORDER BY` clause can result in [`Resources exceeded`]() errors. Because the final sorting must be done on a single slot, if you are attempting to order a very large result set, the final sorting can overwhelm the slot that is processing the data. If you are using an `ORDER BY` clause, also use a `LIMIT` clause.

For example, the following query orders a very large results set and throws a `Resources exceeded` error. The query sorts by the `title` column in the `Wiki1B` table. The `title` column contains millions of values.

```
SELECT title
FROM bigquery-samples.wikipedia_benchmark.Wiki1B
ORDER BY title DESC
LIMIT 1000
```

Self-joins

Best practice: Avoid self-joins. Use a window function instead.

Typically, self-joins are used to compute row-dependent relationships. The result of using a self-join is that it potentially doubles the number of output rows. This increase in output data can cause poor performance.

Instead of using a self-join, use a window (analytic) function to reduce the number of additional bytes that are generated by the query.

```
SELECT SUM(x) OVER (
  window_name
  PARTITION BY...
  ORDER BY...
  window_frame_clause)
FROM ...
```

Data skew

Best practice: If your query processes keys that are heavily skewed to a few values, filter your data as early as possible.

Partition skew, sometimes called data skew, is when data is partitioned into very unequally sized partitions. This creates an imbalance in the amount of data sent between slots. You can't share partitions between slots, so if one partition is especially large, it can slow down, or even crash the slot that processes the oversized partition.

When a slot's resources are overwhelmed, a resources exceeded error results. Reaching the shuffle limit for a slot (2TB in memory compressed) also causes the shuffle to write to disk and further impacts performance. If you examine the query explain plan and see a significant difference between avg and max compute times, your data is probably skewed.

To avoid performance issues that result from data skew:

- Use an approximate aggregate function such as APPROX_TOP_COUNT to determine if the data is skewed.
- Filter your data as early as possible.

BigQuery - Anti Patterns

Unbalanced joins

Data skew can also appear when you use JOIN clauses. Because BigQuery shuffles data on each side of the join, all data with the same join key goes to the same shard. This shuffling can overload the slot.

To avoid performance issues that are associated with unbalanced joins:

- Pre-filter rows from the table with the unbalanced key.
- If possible, split the query into two queries.

Cross joins (Cartesian product)

Best practice: Avoid joins that generate more outputs than inputs. When a CROSS JOIN is required, pre-aggregate your data.

Cross joins are queries where each row from the first table is joined to every row in the second table (there are non-unique keys on both sides). The worst case output is the number of rows in the left table multiplied by the number of rows in the right table. In extreme cases, the query might not finish.

If the query job completes, the query plan explanation will show output rows versus input rows. You can confirm a Cartesian product by modifying the query to print the number of rows on each side of the JOIN clause, grouped by the join key.

To avoid performance issues associated with joins that generate more outputs than inputs:

- Use a GROUP BY clause to pre-aggregate the data.
- Use a window function. Window functions are often more efficient that using a cross join. For more information, see analytic functions.

DML statements that update or insert single rows

Best practice: Avoid point-specific DML statements (updating or inserting 1 row at a time). Batch your updates and inserts.

Using point-specific DML statements is an attempt to treat BigQuery like an Online Transaction Processing (OLTP) system. BigQuery focuses on Online Analytical Processing (OLAP) by using table scans and not point lookups. If you need OLTP-like behavior (single-row updates or inserts), consider a database designed to support OLTP use cases such as Google Cloud SQL.

BigQuery DML statements are intended for bulk updates. `UPDATE` and `DELETE` DML statements in BigQuery are oriented towards periodic rewrites of your data, not single row mutations. The `INSERT`DML statement is intended to be used sparingly. Inserts consume the same modification quotas as load jobs. If your use case involves frequent single row inserts, consider streaming your data instead.

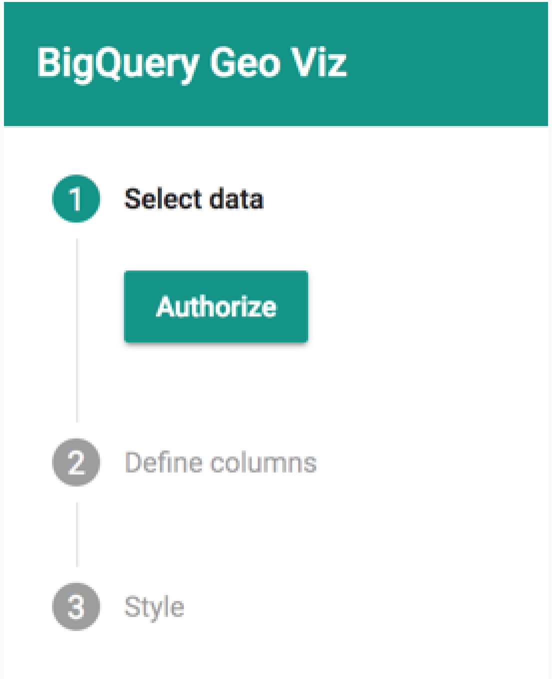
BigQuery GIS

BigQuery - GIS

Geospatial Datatypes and Functions

BigQuery GIS[BETA] brings SQL support for the most commonly used GIS functions right into your data warehouse. With support for arbitrary points, lines, polygons, and multi-polygons in WKT and GeoJSON format, you can simplify your geospatial analyses, see your location-based data in new ways, or unlock entirely new lines of business with the power of BigQuery.

BigQuery - GIS




```
#standardSQL
-- Finds Citi Bike stations with > 30 bikes
SELECT
  ST_GeogPoint(longitude, latitude)  AS WKT,
  num_bikes_available
FROM
  `bigquery-public-data.new_york.citibike_stations`
WHERE num_bikes_available > 30
```

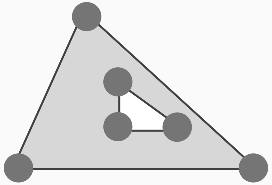
BigQuery - GIS

Data type:

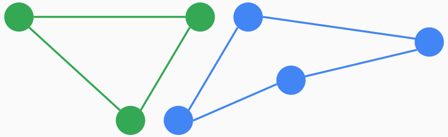
Point

Linestring

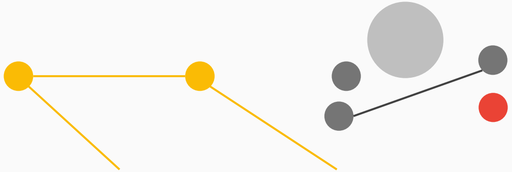
Polygon



Multi-polygon



Collections



Formats:

WKT
GeoJSON
WKB

BigQuery - GIS

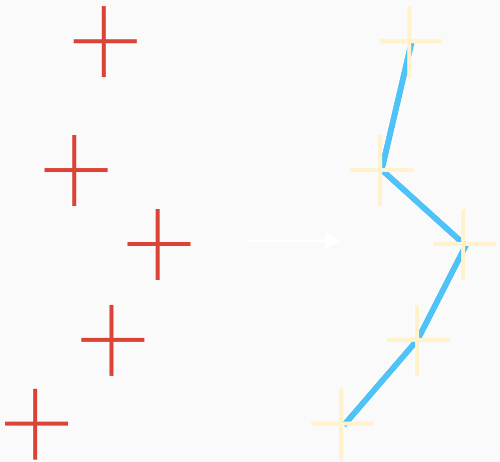
Native SQL support for the most commonly used ST_* functions and geographic data types

Functions	Description
Constructors	Constructive operations build new geography literals from coordinates or existing geographies.
Transformations	Operations that return a single Geography from one or more distinct geographies (e.g., ST_Union)
Predicates	Predicate operations return true/false for some spatial relationship between two geometries. Most frequently used in filter clauses.
Accessors	Operations that let users navigate and select between multiple ways of handling a record based on its type, or select a particular element.
Measures	Measure operations compute some property of the geography such as perimeter, area, or distance to another geography.
Parsers	Operations that construct a Geography from raw coordinates or other geographies.
Formatters	Formatting operations return a geography converted into a standardized (usually string) format suitable for presenting in query results.

Constructors

ST_GEOGPOINT(longitude, latitude)
ST_MAKELINE(geography_1, geography_2)
ST_MAKELINE(array_of_geography)
ST_MAKEPOLYGON(geography_expression)
ST_MAKEPOLYGON(geography_expression, array_of_geography)
ST_MAKEPOLYGONORIENTED(array_of_geography)

Build geographies from
coordinates or existing geographies



BigQuery - GIS

BigQuery Geo Viz

```
WITH
point_1 as (SELECT ST_GEOGPOINT(longitude, latitude) as point from `bigquery-
public-data.covid19_italy.data_by_province` WHERE province_name = "Torino"
limit 1 ),
point_2 as (SELECT ST_GEOGPOINT(longitude,latitude) as point from `bigquery-
public-data.covid19_italy.data_by_province` WHERE province_name = "Milano"
limit 1)

SELECT ST_MAKELINE([(SELECT point from point_1),(SELECT point from point_2)])
as line
```

BigQuery - GIS

BigQuery Geo Viz

```
WITH
point_1 as (SELECT ST_GEOGPOINT(longitude, latitude) as point from `bigquery-public-data.covid19_italy.data_by_province` WHERE province_name = "Torino" limit 1 ),
point_2 as (SELECT ST_GEOGPOINT(longitude,latitude) as point from `bigquery-public-data.covid19_italy.data_by_province` WHERE province_name = "Bologna" limit 1),
point_3 as (SELECT ST_GEOGPOINT(longitude,latitude) as point from `bigquery-public-data.covid19_italy.data_by_province` WHERE province_name = "Roma" limit 1)

SELECT ST_MAKEPOLYGON (ST_MAKELINE([(SELECT point from point_1),(SELECT point from point_2),(SELECT point from point_3)])) as triangle
```

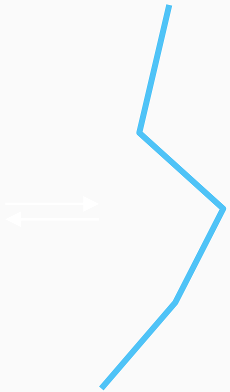
Parsers & formatters

```
ST_GEOGFROMGEOJSON(geojson_string)
ST_GEOGFROMTEXT(wkt_string)
```
`ST_GEOGFROMWKB(wkb_bytes)`

```
ST_ASGEOJSON(geography_expression)
ST_ASTEXT(geography_expression)
```
`ST_ASBINARY(geography_expression)`

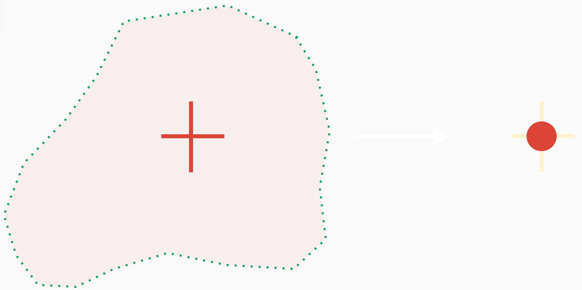
Create/export geographies
between formats

((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0)),
((0 0 0, 0 1 0, 0 1 1, 0 0 1, 0 0 0)),
((0 0 0, 1 0 0, 1 0 1, 0 0 1, 0 0 0)),
((1 1 1, 1 0 1, 0 0 1, 0 1 1, 1 1 1)),
((1 1 1, 1 0 1, 1 0 0, 1 1 0, 1 1 1))



Transformations

```
ST_INTERSECTION(geography_1, geography_2)
ST_UNION(geography_1, geography_2)
ST_UNION(array_of_geography)
ST_UNION_AGG(geography)
ST_DIFFERENCE(geography_1, geography_2)
ST_CENTROID(geography_expression)
ST_CLOSESTPOINT(geography_1, geography_2[,spheroid=FALSE])
ST_BOUNDARY(geography_expression)
ST_SNAPTOGRID(geography_expression, grid_size)
```

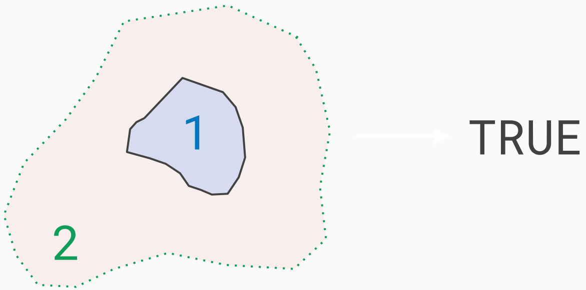


Create new geographies
with similar properties

Predicates

```
ST_CONTAINS(geography_1, geography_2)
ST_COVEREDBY(geography_1, geography_2)
ST_COVERS(geography_1, geography_2)
ST_DISJOINT(geography_1, geography_2)
ST_DWITHIN(geography_1, geography_2, distance[, spheroid=FALSE])
ST_EQUALS(geography_1, geography_2)
ST_INTERSECTS(geography_1, geography_2)
ST_INTERSECTSBOX(geography, lng1, lat1, lng2, lat2)
ST_TOUCHES(geography_1, geography_2)
ST_WITHIN(geography_1, geography_2)
```

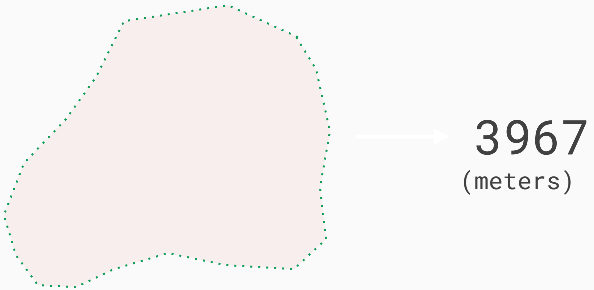
Filter geographies
(TRUE/FALSE)



Measures

```
ST_DISTANCE(geography_1, geography_2[, spheroid=FALSE])
ST_LENGTH(geography_expression[, spheroid=FALSE])
ST_PERIMETER(geography_expression[, spheroid=FALSE])
ST_AREA(geography_expression[, spheroid=FALSE])
ST_MAXDISTANCE(geography_1, geography_2[, spheroid=FALSE])
```

Compute measurements
of geographies



BigQuery - GIS

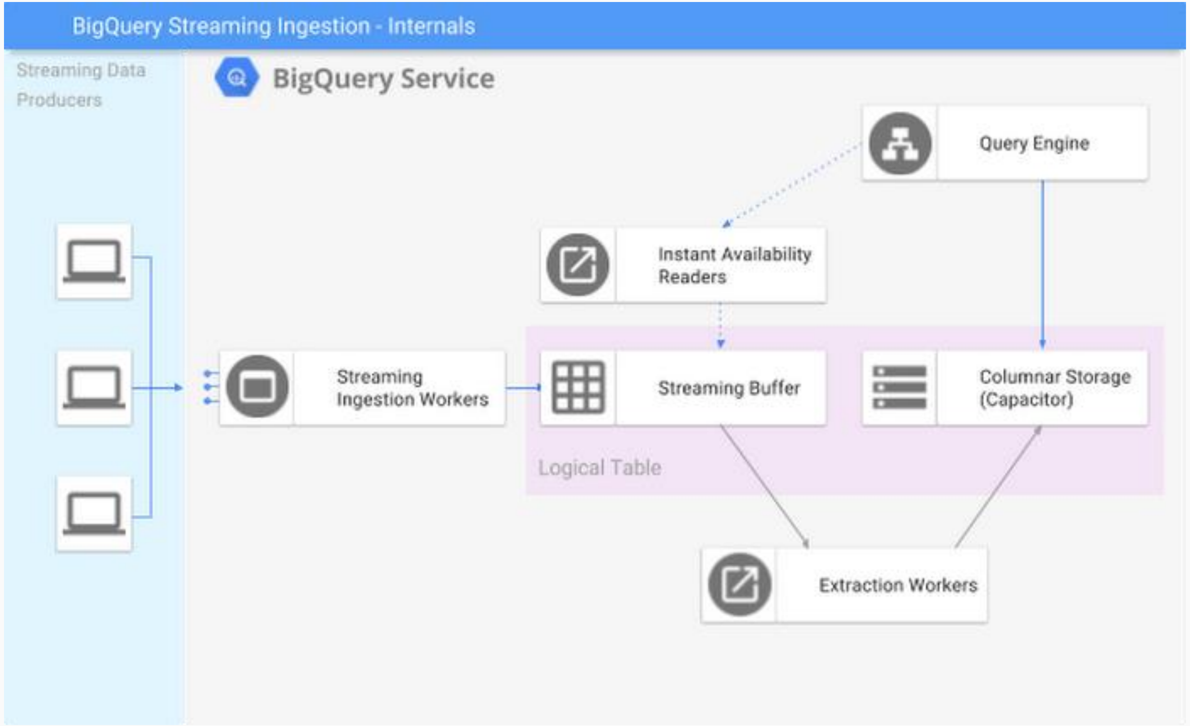
BigQuery Geo Viz

```
WITH
point_1 as (SELECT ST_GEOGPOINT(longitude, latitude) as point FROM `bigquery-
public-data.covid19_italy.data_by_province` WHERE province_name = "Torino"
limit 1 ), point_2 as (SELECT ST_GEOGPOINT(longitude,latitude) as point FROM
`bigquery-public-data.covid19_italy.data_by_province` WHERE province_name =
"Bologna" limit 1), point_3 as (SELECT ST_GEOGPOINT(longitude,latitude) as
point FROM `bigquery-public-data.covid19_italy.data_by_province` WHERE
province_name = "Roma" limit 1)

SELECT ST_MAKEPOLYGON (ST_MAKELINE([(SELECT point FROM point_1),(SELECT point
FROM point_2),(SELECT point FROM point_3)])) as triangle,
ST_AREA(ST_MAKEPOLYGON (ST_MAKELINE([(SELECT point FROM point_1),(SELECT point
FROM point_2),(SELECT point FROM point_3)]))) as area
```

Streaming en BQ con Dataflow

Streaming architecture



Streaming inserts

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

Streaming limits

- Maximum bytes per second: 1GB
- Maximum rows per second per project in the us and eu multi-regions: 500.000 rows
- Max rows / second / table: 100,000
- Max rows / request: 50.000

Use Cases

- **Not transactional.** High volume, continuously appended rows. The app can tolerate a rare possibility that duplication might occur or that data might be temporarily unavailable.
- **Aggregate analysis.** Queries generally are performed for trend analysis, as opposed to single or narrow record selection.

Stream Analytics on Google Cloud

01 Ingest

Ingest and distribute
data reliably

Pub/Sub

02 Transform

Fast, correct computations
quickly and simply

Dataflow

beam

03 Analyze / Serve

Machine learning
& data warehouse

AI Platform

BigQuery

Cloud BigTable

Cloud SQL

Stream Analytics Open Source

01 Ingest

Ingest and distribute
data reliably

Pub/Sub

kafka

confluent cloud

02 Transform

Fast, correct computations
quickly and simply

Dataflow

Apache Spark

Apache Flink

beam

03 Analyze / Serve

Machine learning
& data warehouse

AI Platform

BigQuery

mongoDB

Cloud BigTable

Cloud SQL

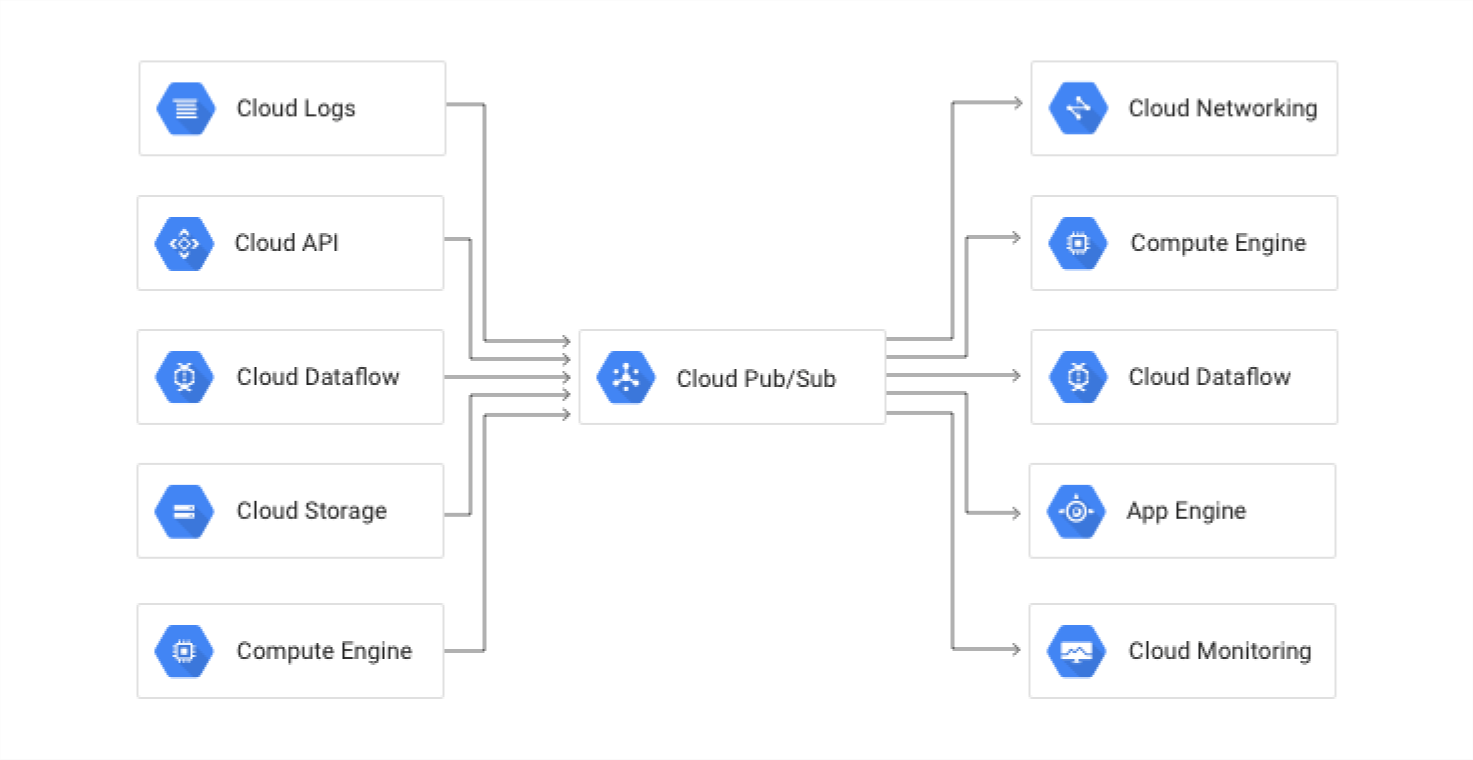
influxdb

Pub/Sub: 100% serverless event delivery

Google Cloud
Pub/Sub

- Reliable and real-time messaging
- Global by design and highly available
- Uses Google's private fiber network and worldwide points of interconnect
- Only pay for what you use

Pub/Sub: Google Cloud Pub/Sub passes messages



What is Beam & Dataflow?

Apache Beam

Portability
Open-source programming model

Elegant Design
Unified batch & streaming

Flexibility
Runner & language portability

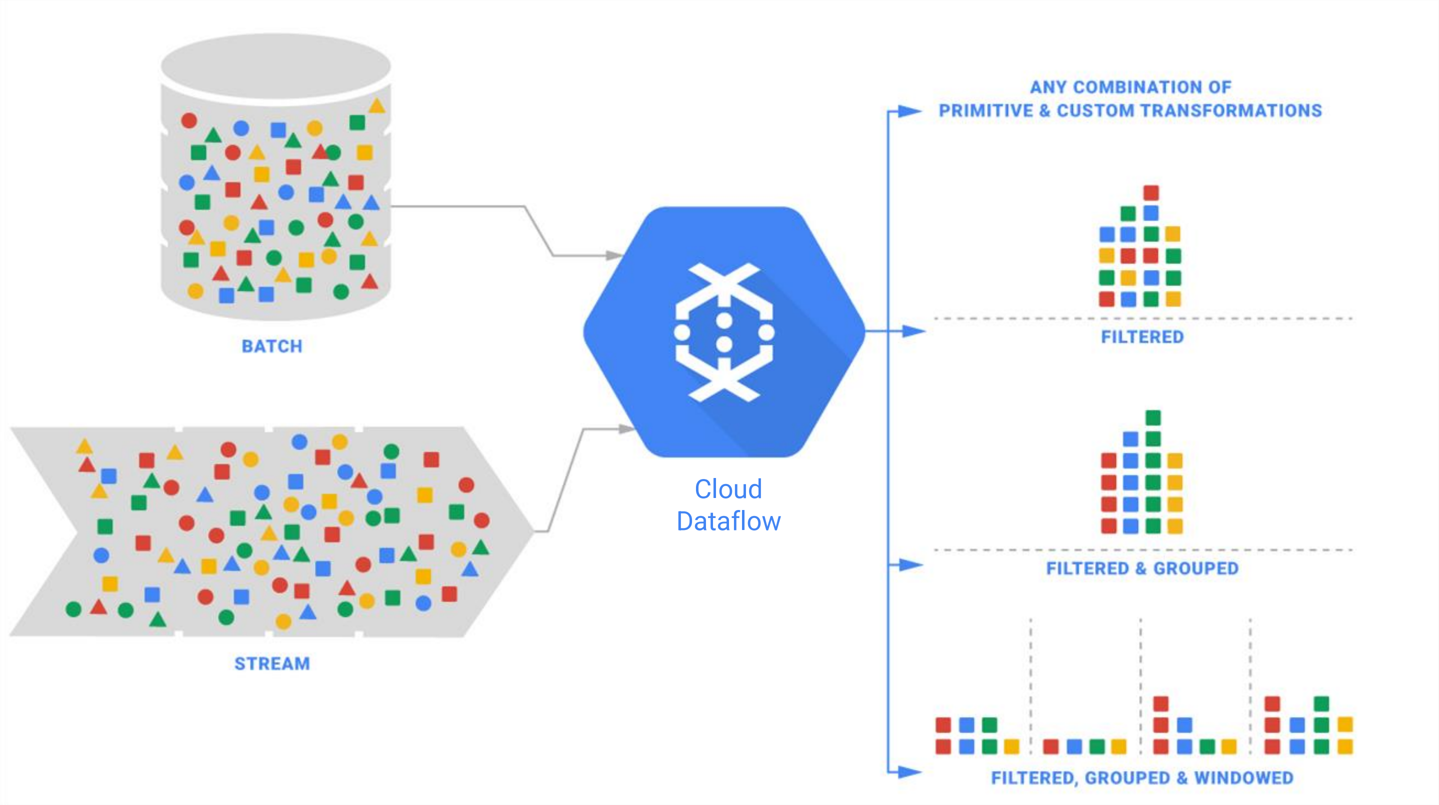
Dataflow

Fully Managed & Scalable
Serverless data processing that scales to millions of QPS

Accuracy
Exactly-once streaming semantics

Efficiency
State storage in Shuffle & Streaming Engine

How Cloud Dataflow works ?



BigQuery - Streaming Sample

Running your first SQL statements using Google Cloud Dataflow

4 minutes

Updated October 11, 2020

The page explains how to use Dataflow SQL and create Dataflow SQL jobs.

Start

¿ Preguntas ?

Nombre