
# Building a Data Warehouse

# Building a Data Warehouse

| | |
|---|---|
| 01 | The modern data warehouse |
| 02 | Introduction to BigQuery |
| 03 | Get started with BigQuery |
| 04 | Load data into BigQuery |
| 05 | Explore schemas |
| 06 | Schema design |
| 07 | Nested and repeated fields |
| 08 | Optimize with partitioning and clustering |

# Building a Data Warehouse

| | |
|---|---|
| 01 | The modern data warehouse |
| 02 | Introduction to BigQuery |
| 03 | Get started with BigQuery |
| 04 | Load data into BigQuery |
| 05 | Explore schemas |
| 06 | Schema design |
| 07 | Nested and repeated fields |
| 08 | Optimize with partitioning and clustering |

# A data warehouse should consolidate data from many sources

# The data in a warehouse should have quality, consistency, and accuracy

# A data warehouse should be optimized for simplicity of access and high-speed query performance

# A modern data warehouse

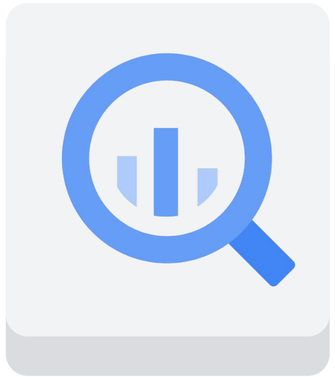
- Gigabytes to petabytes
- Serverless and no-ops, including ad hoc queries
- Ecosystem of visualization and reporting tools
- Ecosystem of ETL and data processing tools
- Up-to-the-minute data
- Machine learning
- Security and collaboration

# Building a Data Warehouse

| | |
|---|---|
| 01 | The modern data warehouse |
| 02 | Introduction to BigQuery |
| 03 | Get started with BigQuery |
| 04 | Load data into BigQuery |
| 05 | Explore schemas |
| 06 | Schema design |
| 07 | Nested and repeated fields |
| 08 | Optimize with partitioning and clustering |

# BigQuery has many capabilities that make it an ideal data warehouse

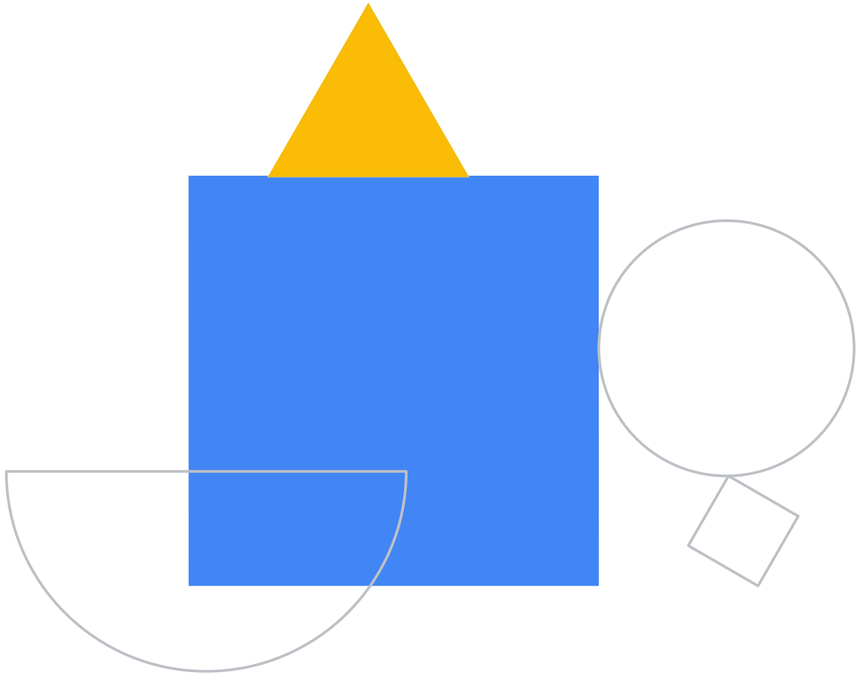
- Interactive SQL queries over large datasets (petabytes) in seconds
- Serverless and no-ops, including ad hoc queries
- Ecosystem of visualization and reporting tools
- Ecosystem of ETL and data processing tools
- Up-to-the-minute data
- Machine learning
- Security and collaboration



BigQuery

# Demo

Query TB+ of data in seconds



# BigQuery is a serverless fully-managed service

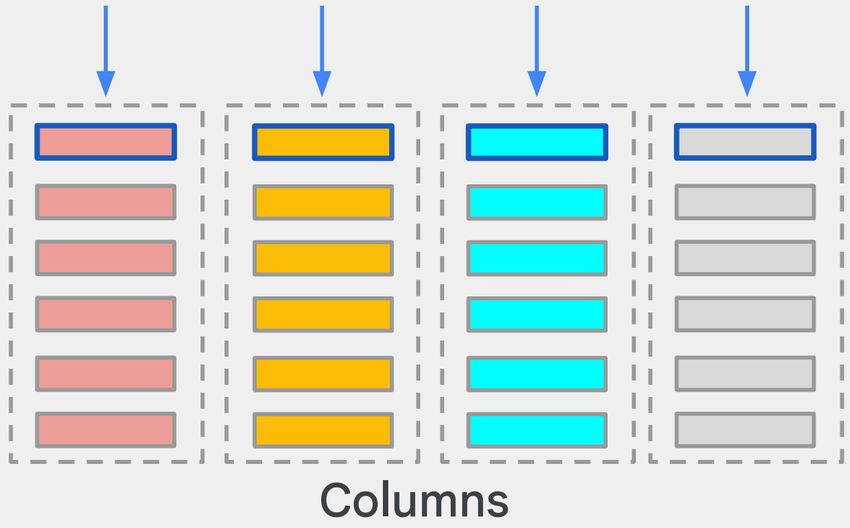
- Data aging
- Query engine optimization
- Storage management
- Hardware
- Fault recovery
- Updates

Free up real people-hours by not having to worry about common tasks.

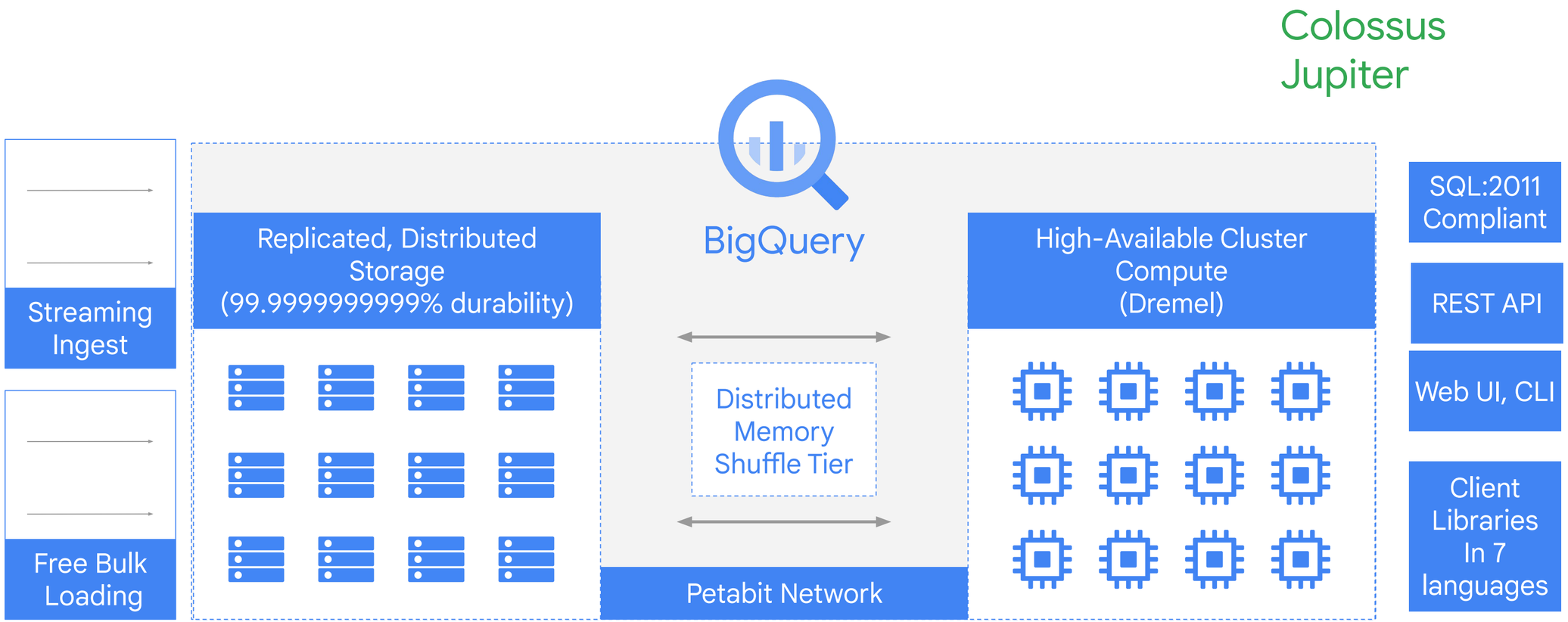
# What makes BigQuery fast?

## Relational Database tables are row-oriented

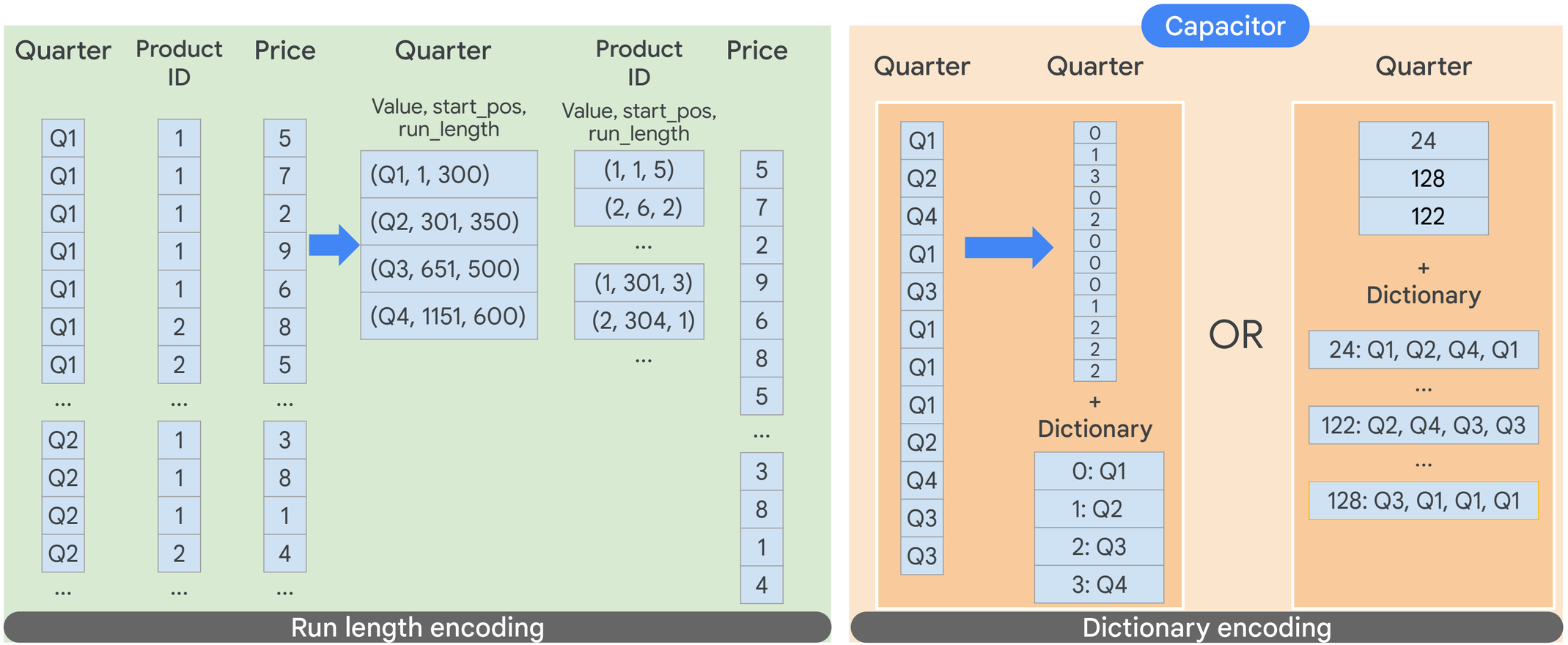

## BigQuery tables are column-oriented



# The data is physically stored in a redundant way separate from the compute cluster

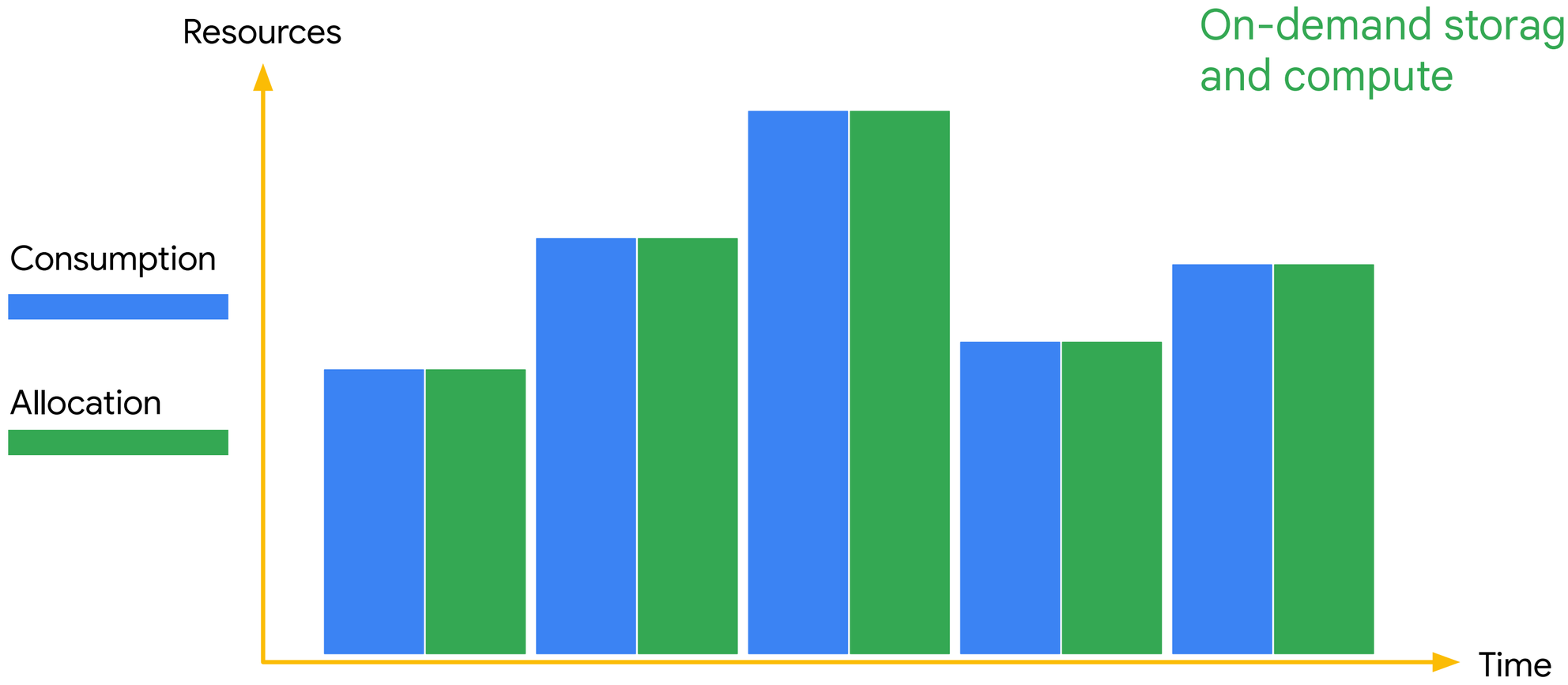


# The data are also run length-encoded and dictionary-encoded



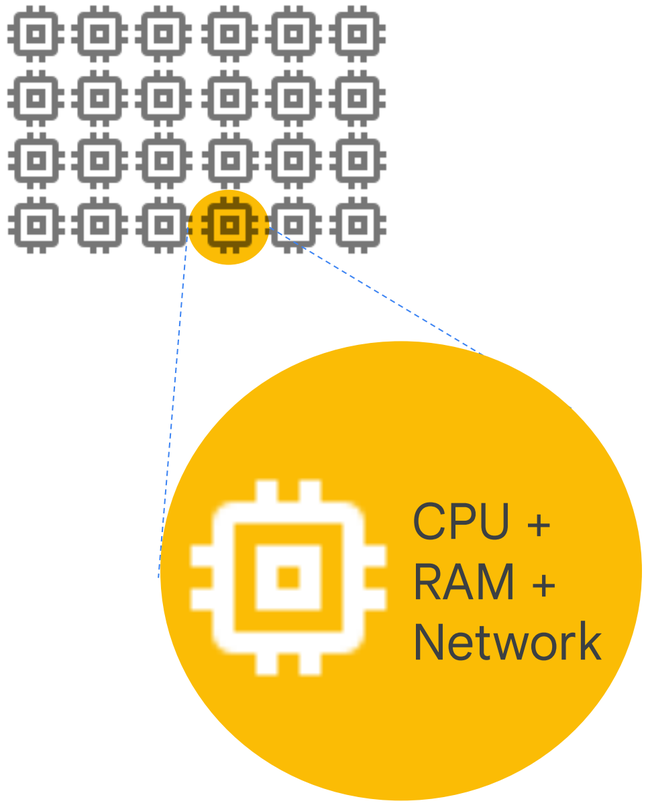

Examples taken from VLDB 2009 tutorial on Column Oriented Database Systems

# You don't need to provision resources before using BigQuery



# A BigQuery slot is a combination of CPU, memory, and networking resources

BigQuery slots



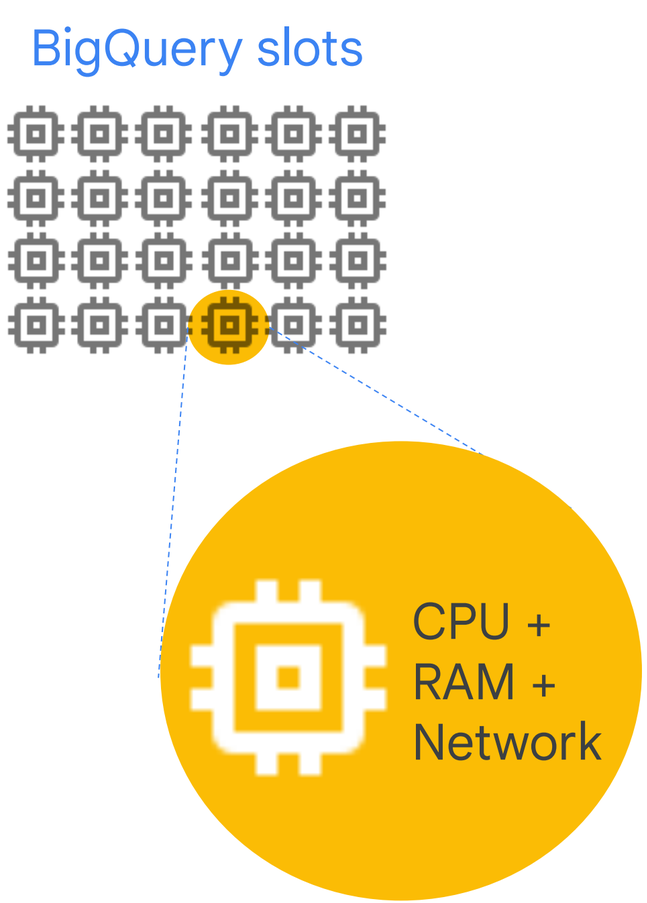

| | Stage timing | | | | Rows | | |
|---|---|---|---|---|---|---|---|
| | Wait | Read | Compute | Write | Parallel Inputs | Input | Output |
| S00: Input | | | | | 7 | 24.4 M | 6.14 K (348 KB) |
| S01: Sort+ | | | | | 1 | 6.14 K | 5 (257 B) |
| S02: Output | | | | | 1 | 5 | 5 (257 B) |

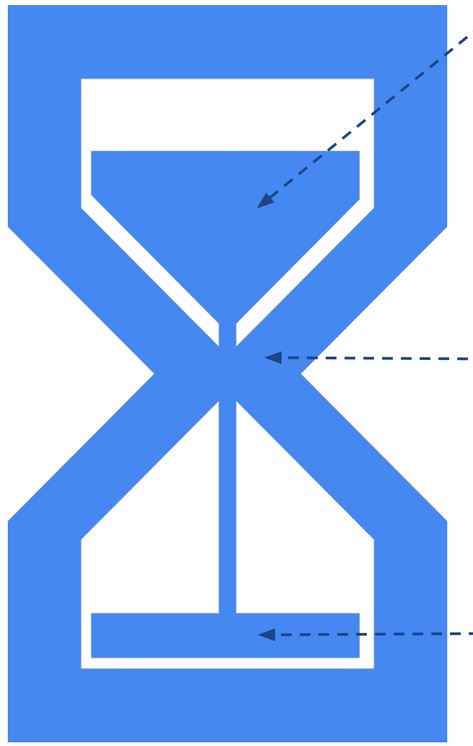
```
READ $1:duration, $2:start_station_name
     FROM bigquery-public-data.london_bicycles.cycle_hire

AGGREGATE GROUP BY $30 := $2
          $20 := SHARD_AVG($1)
          $21 := COUNT($1)

WRITE $30, $20, $21
      TO __stage00_output
      BY HASH($30)
```

```
READ $30, $20, $21
     FROM __stage00_output

SORT $11 DESC
     LIMIT 5

AGGREGATE GROUP BY $40 := $30
          $10 := ROOT_AVG($20)
          $11 := SUM_OF_COUNTS($21)

WRITE $50, $51, $52
      TO __stage01_output
```

```
READ $50, $51, $52
     FROM __stage01_output

SORT $52 DESC
     LIMIT 5

WRITE $60, $61, $62
      TO __stage02_output
```

# The actual number of slots allotted to a query depends on query complexity and project quota



Slots requested

More complex queries = More slots requested

Slots available

Single, simple query = Faster execution rate

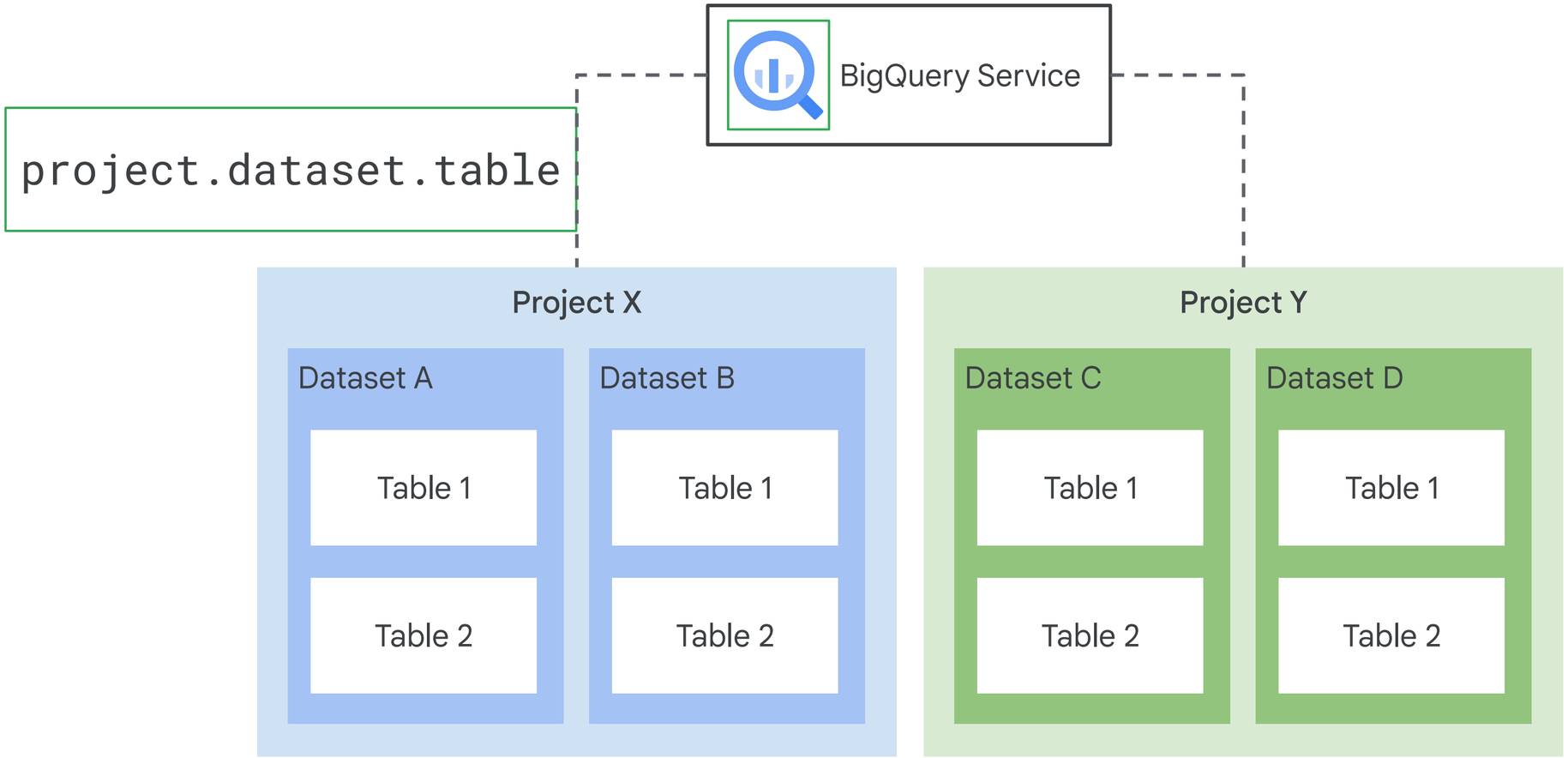
More concurrent queries = Slower execution rate

Completed work

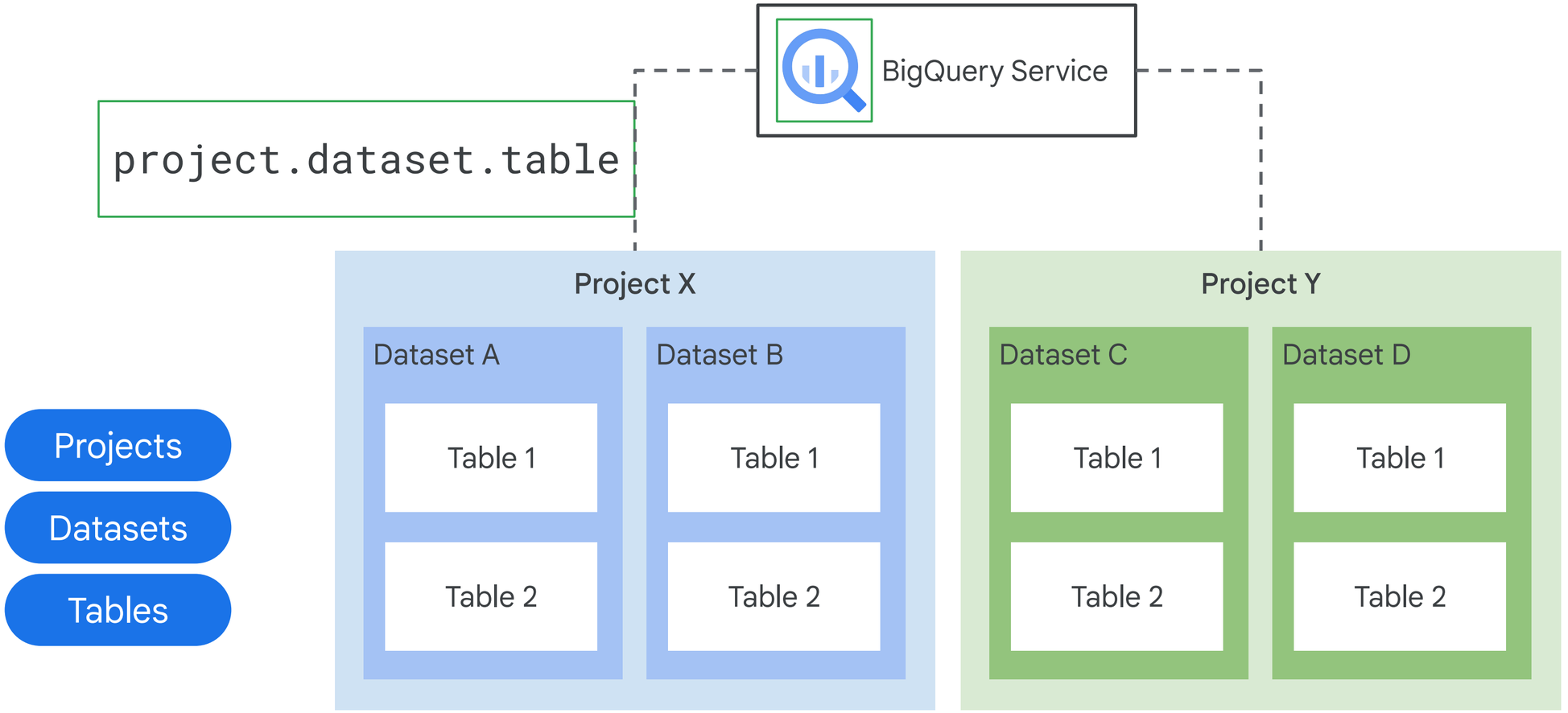
# Building a Data Warehouse



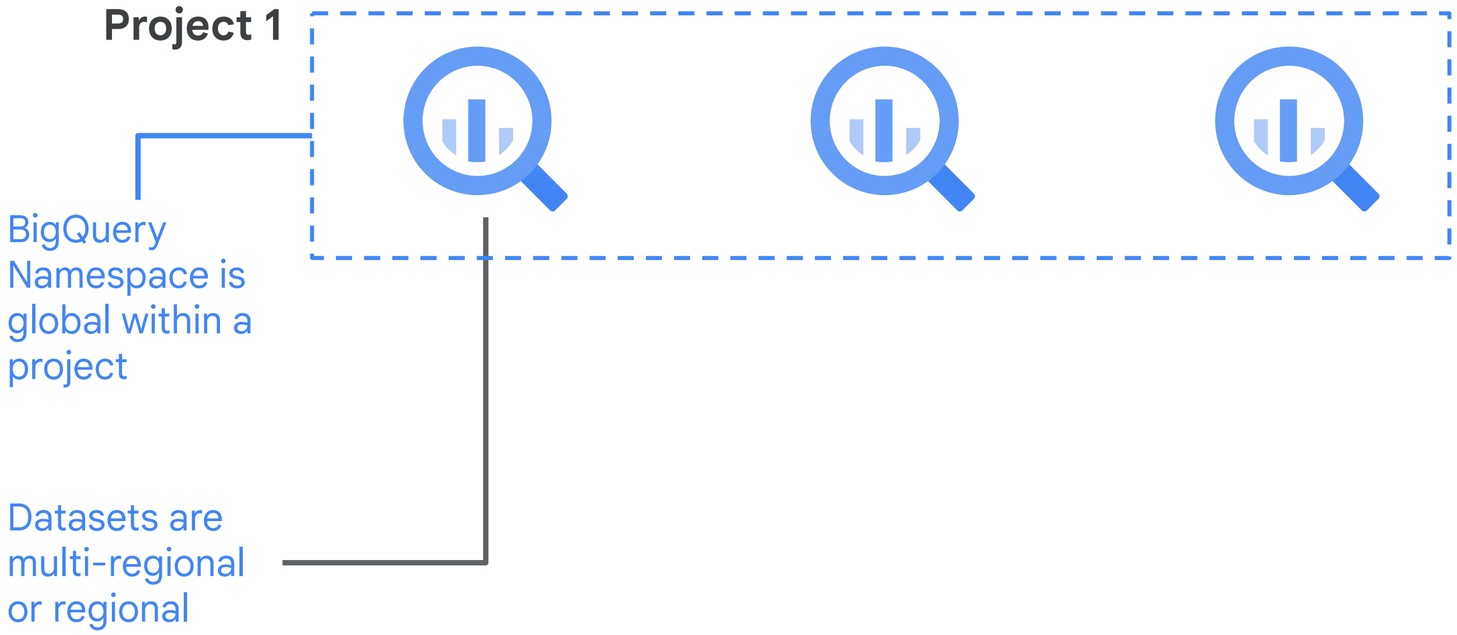
# BigQuery organizes data tables into units called datasets



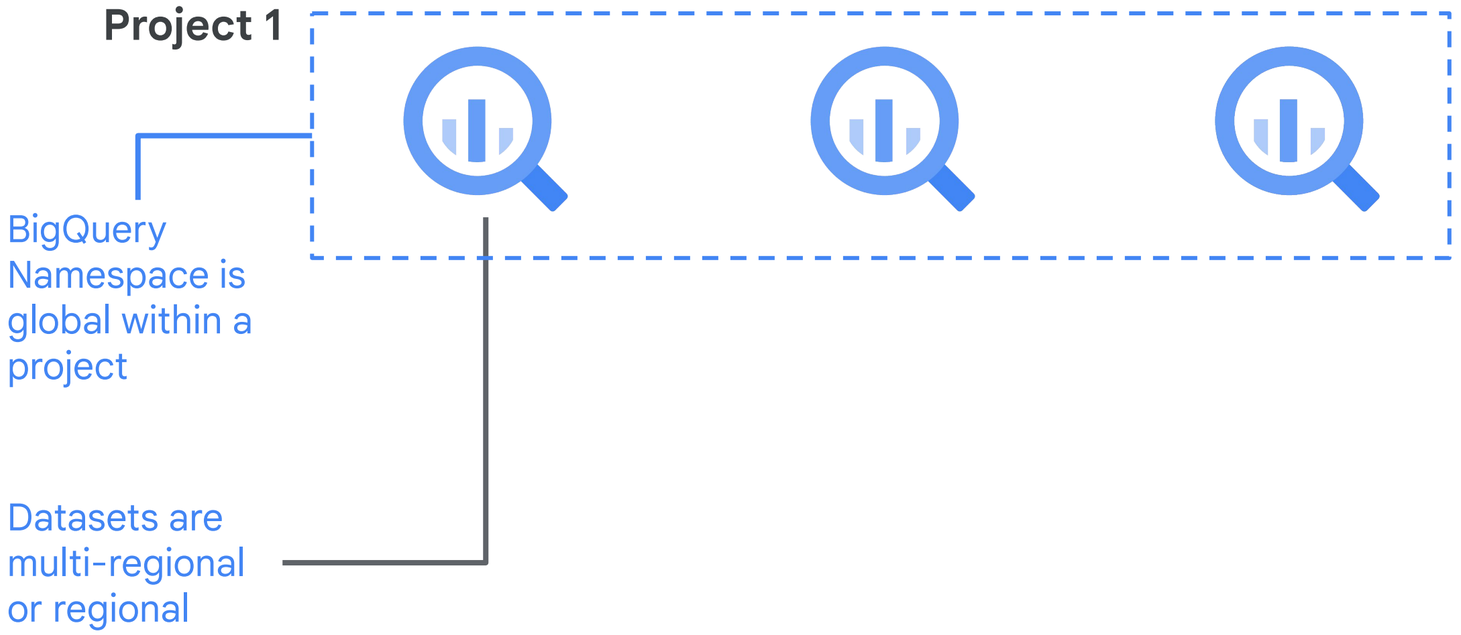
# What are some reasons to structure your information?



# BigQuery datasets belong to a project



# Access control to run a query is via IAM



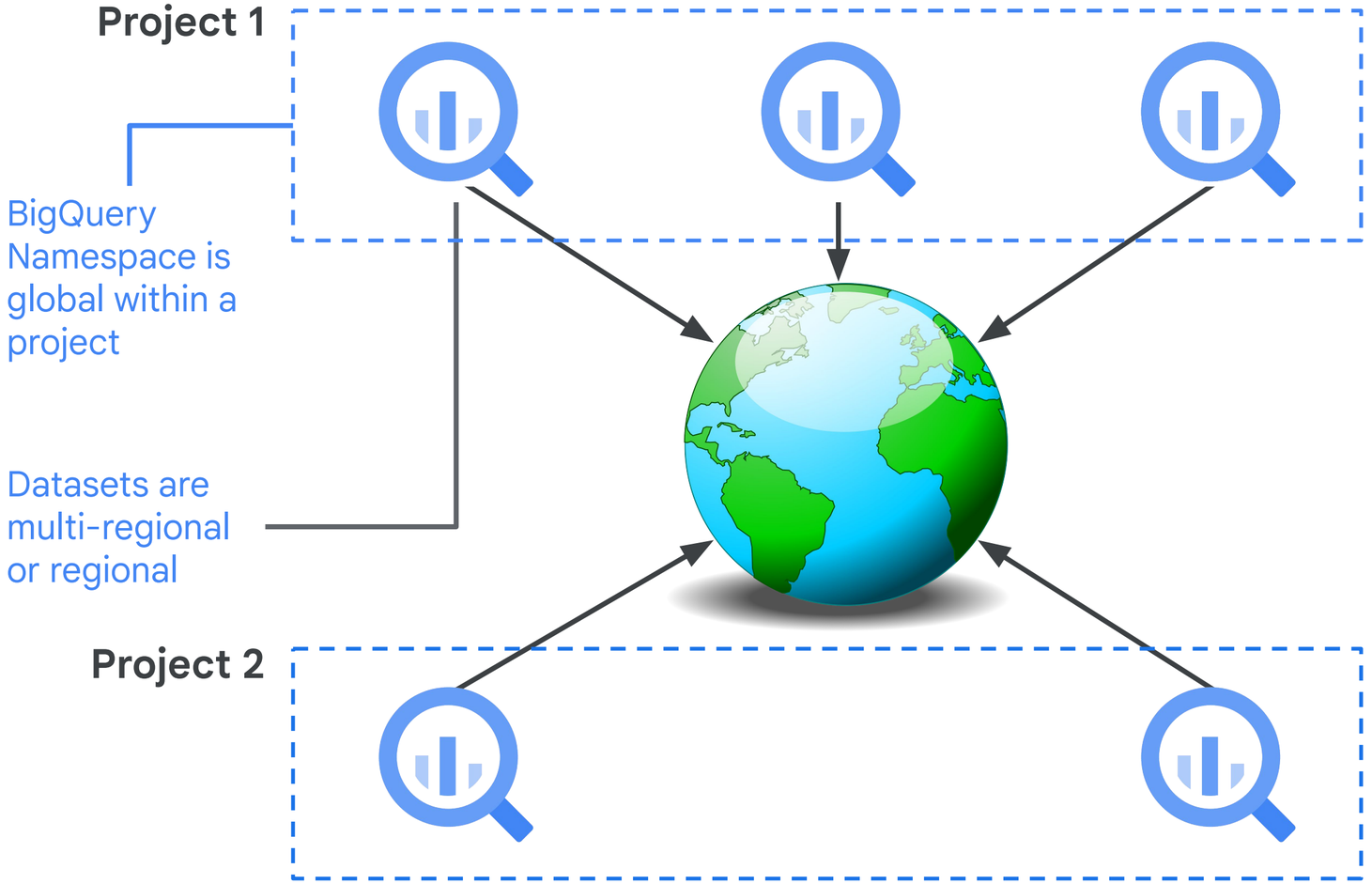

Access control is through IAM

Organization
Project
Dataset

Access control is to the Dataset, Table, View, or Column

A user has to run jobs to query datasets

# BigQuery datasets can be regional or multi-regional



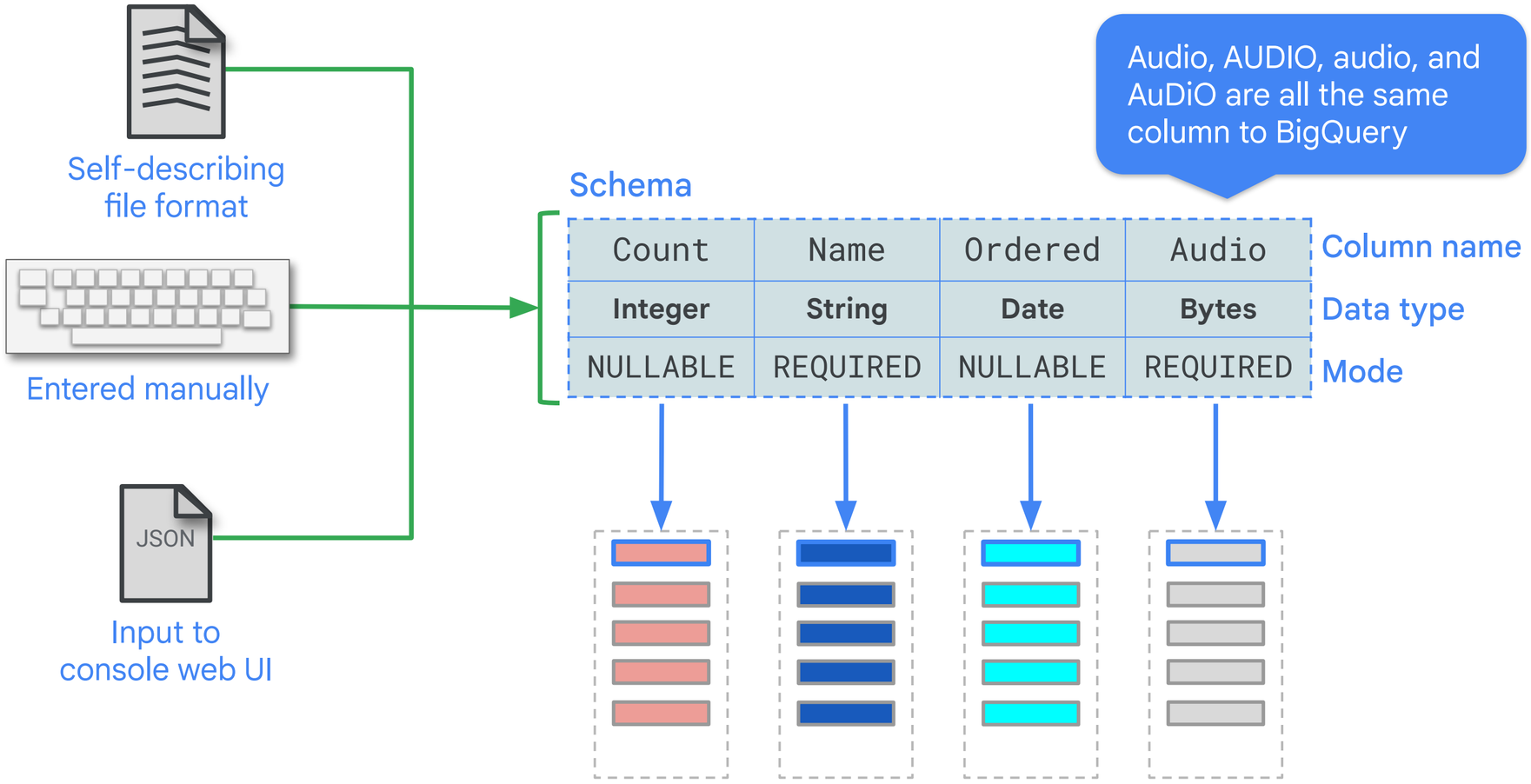

Access control is through IAM

Organization
Project
Dataset

Access control is to the Dataset, Table, View, or Column

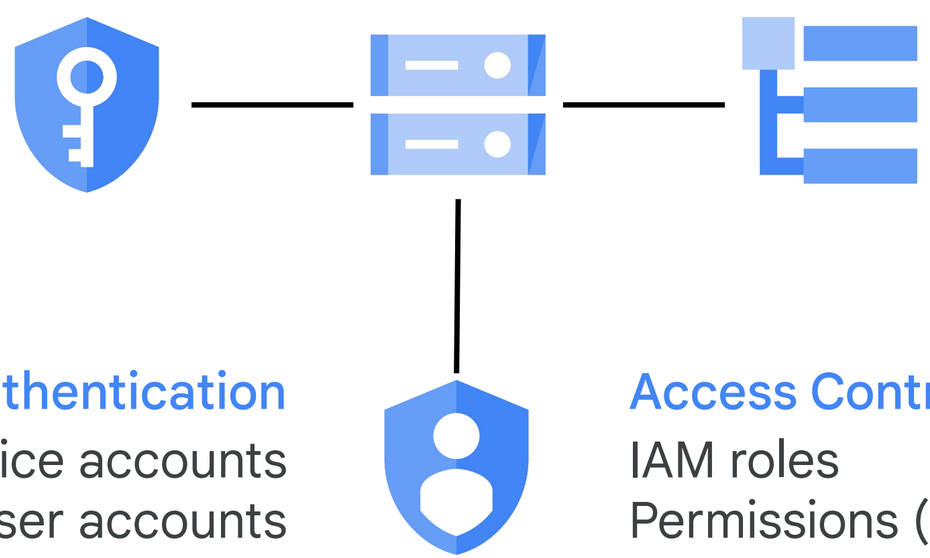
A user has to run jobs to query datasets

# The table schema provides structure to the data



# Security, encryption, and auditing for BigQuery

**Key Management**
Customer-managed encryption keys



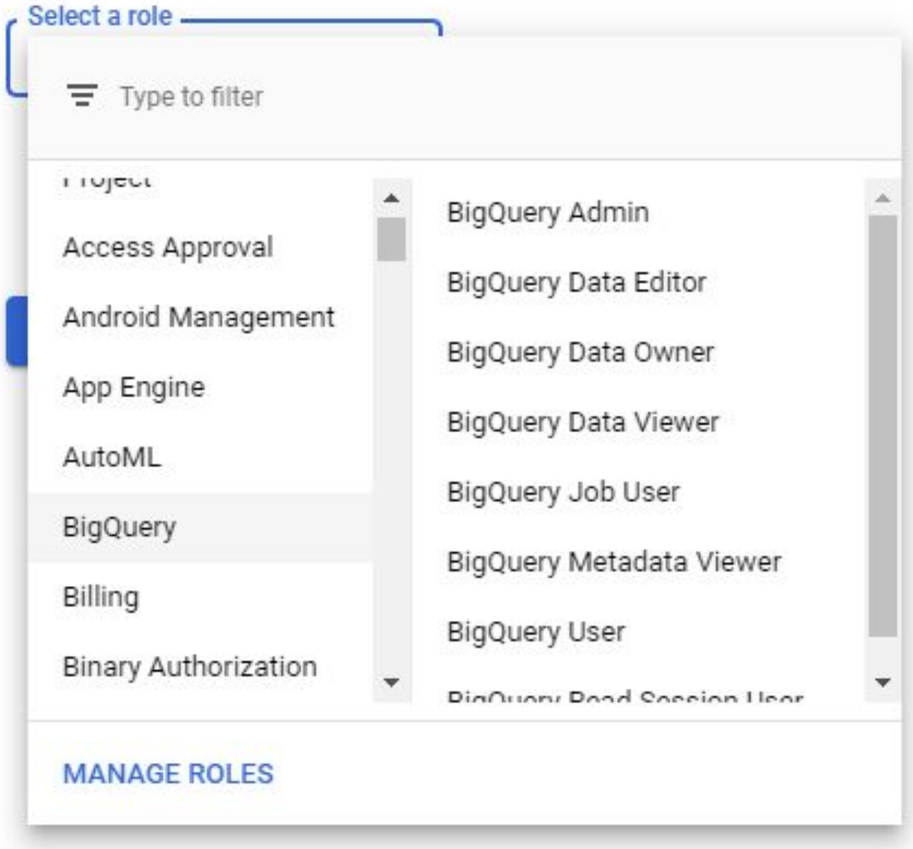
**Cloud Audit Logs**
Admin activity
System event
Data access

**Authentication**
Service accounts
User accounts

**Access Control**
IAM roles
Permissions (custom)

Project, Dataset,
Table, View, Column

# BigQuery provides predefined roles for controlling access to resources




Grants

IAM grants permission to perform specific actions

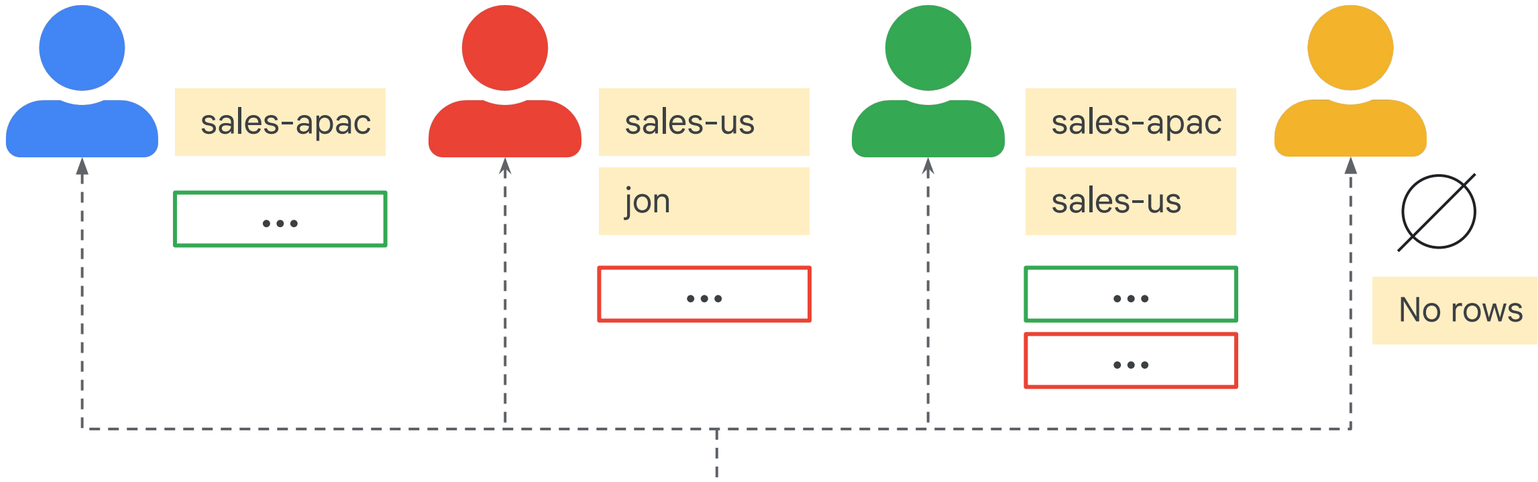
Access control is to datasets, tables, views, or columns.

Use authorized views to restrict at a finer-grained level.

# Row-level security in BigQuery

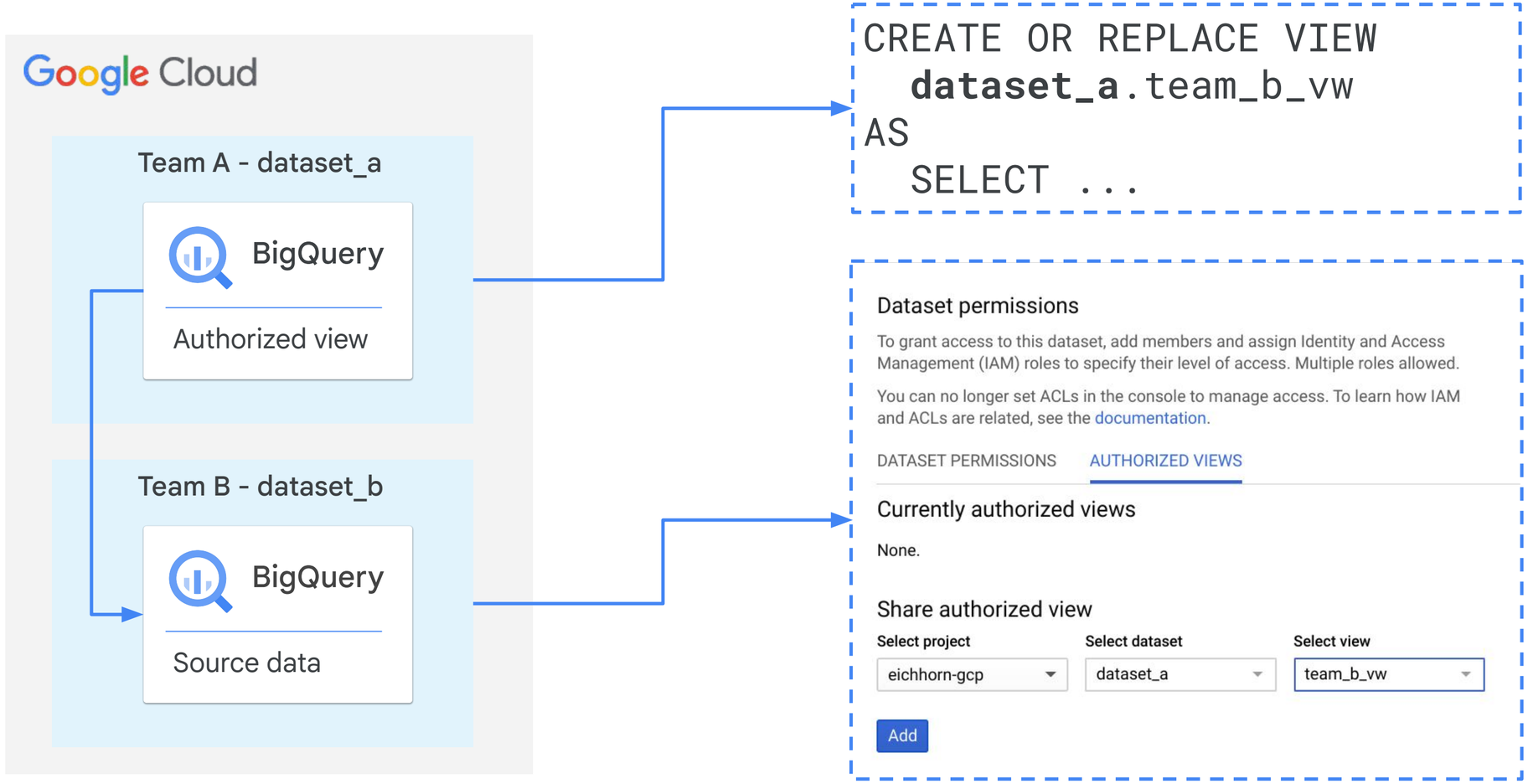
```
CREATE ROW ACCESS POLICY
    apac_filter
ON
    dataset1.table1
GRANT TO
("group:sales-apac@example.com")
FILTER USING
    (Region="APAC");
```

```
CREATE ROW ACCESS POLICY
    us_filter
ON
    dataset1.table1
GRANT TO
("group:sales-us@example.com",
 "user:jon@example.com")
FILTER USING
    (Region="US");
```

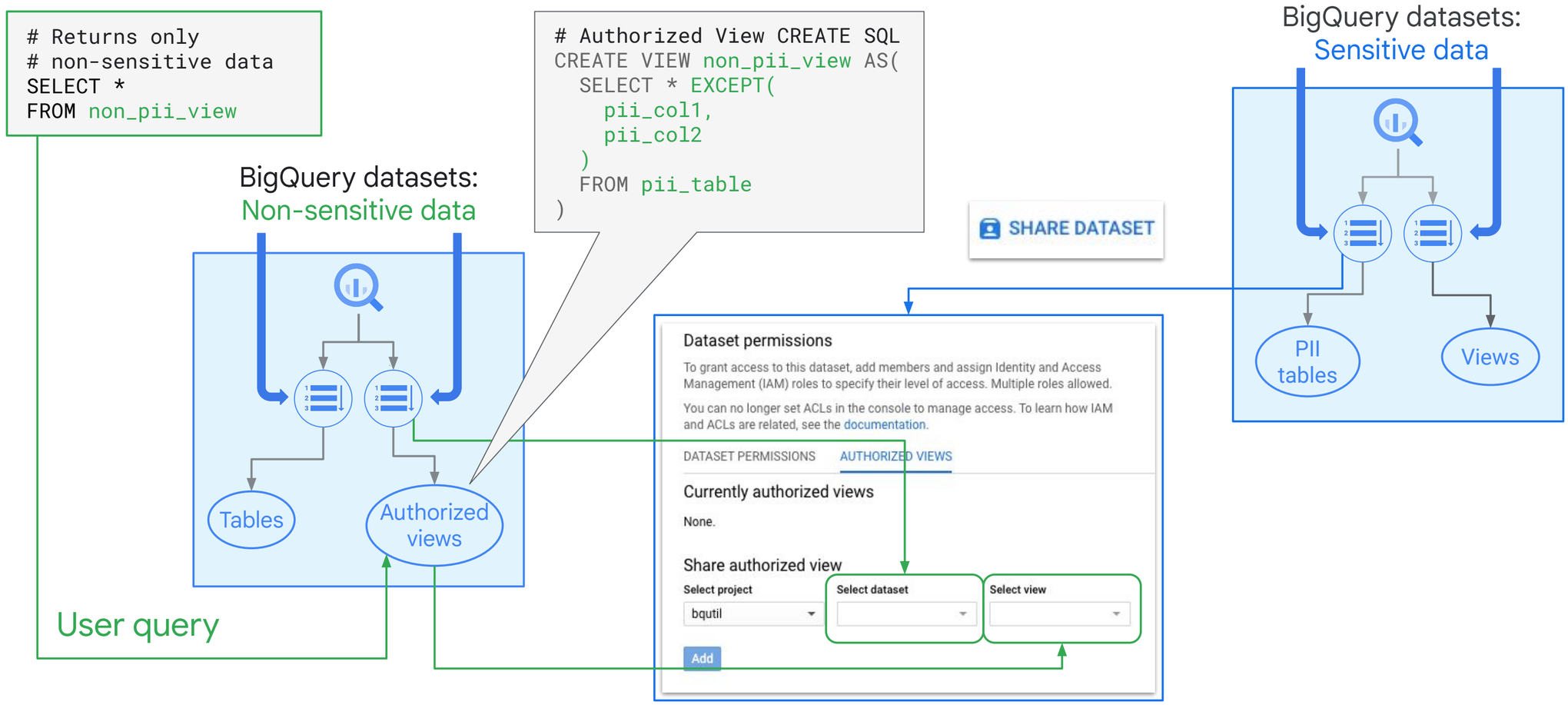



| Partner | Contact | Country | Region |
|---|---|---|---|
| Example Customers Corp | alice@examplecustomers.com | Japan | APAC |
| Example Enterprise Group | bob@exampleenterprisegroup.com | Singapore | APAC |
| Example HighTouch Co. | carrie@examplehightouch.com | USA | US |
| Example Buyers Inc. | david@examplebuyersinc.com | USA | US |

# Creating an authorized view



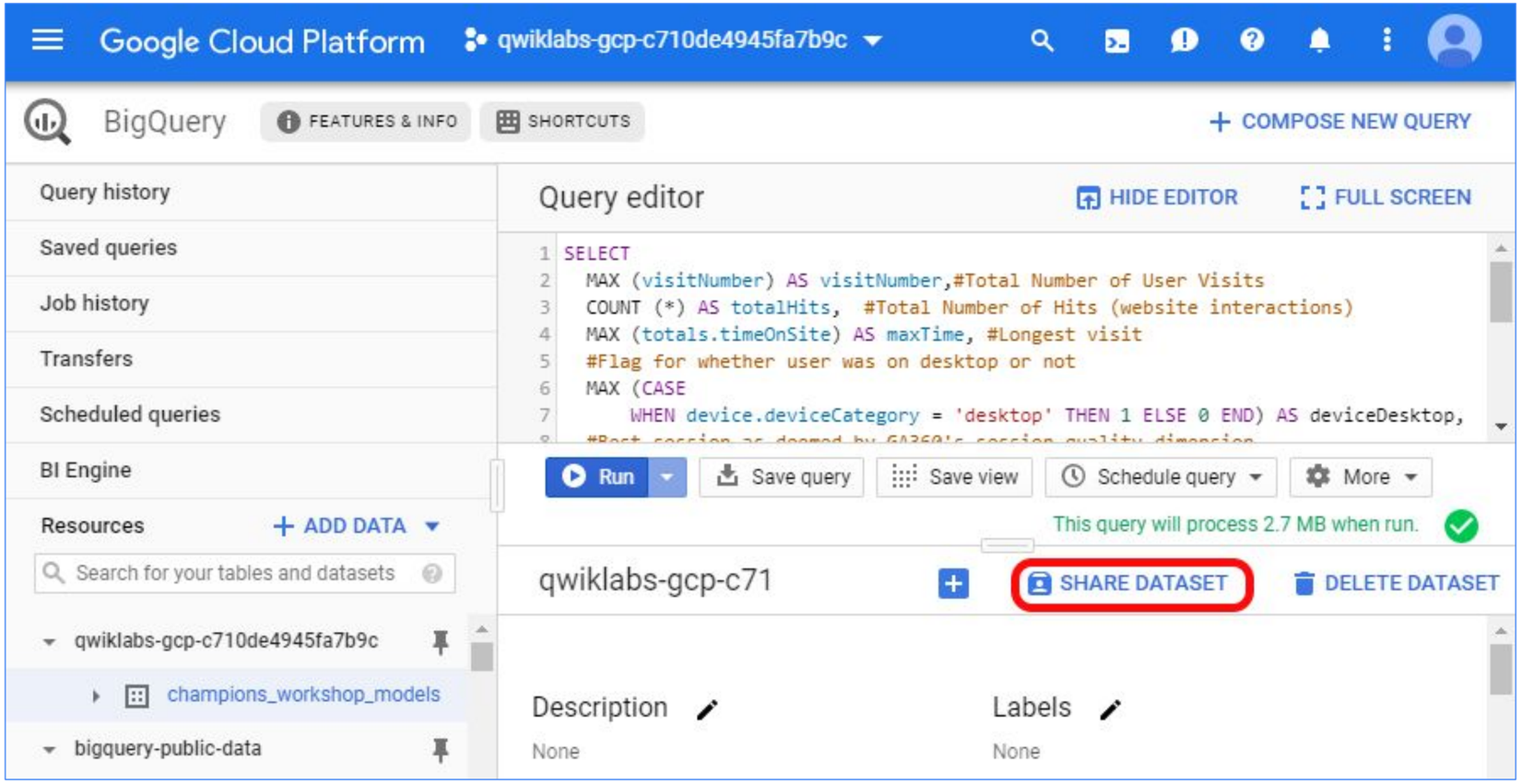
# Protect columns with authorized views




```
# Returns only
# non-sensitive data
SELECT *
FROM non_pii_view
```

```
# Authorized View CREATE SQL
CREATE VIEW non_pii_view AS(
  SELECT * EXCEPT(
    pii_col1,
    pii_col2
  )
  FROM pii_table
)
```

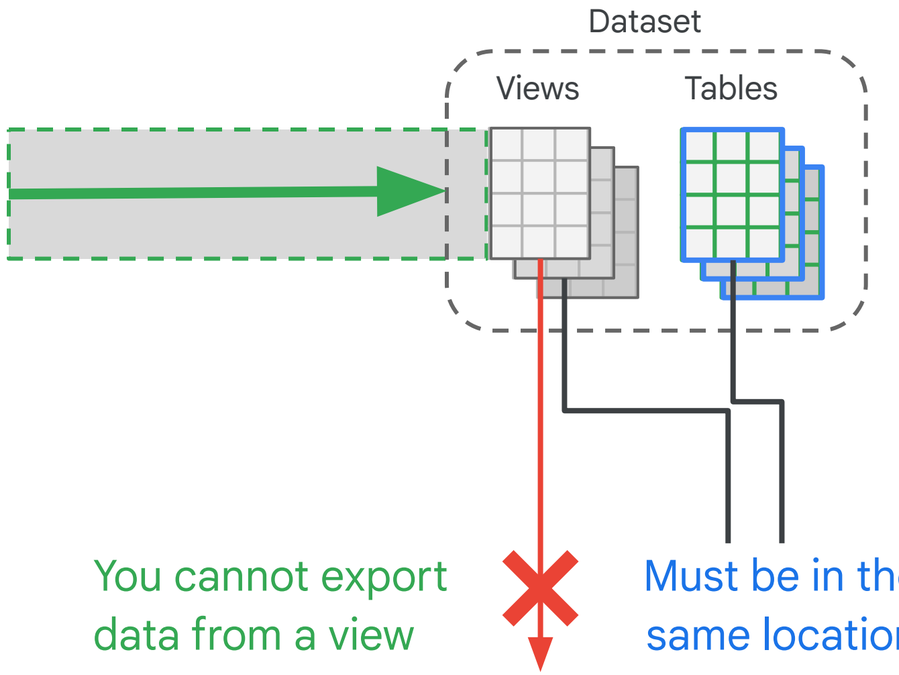
# It’s easy to share access to datasets with other analysts



# Views add another degree of access control

A *view* is a virtual table defined by a SQL query

An authorized view allows you to share data externally without sharing the underlying table




```
CREATE VIEW dsB.myview AS
SELECT
name, number
FROM dsB.mytable
WHERE year >1950
```

You cannot export data from a view

Must be in the same location

# Introduction to materialized views

**base_table**

| date | user | item | amount |
|---|---|---|---|
| 20180801 | Tom | Toaster | 40 |
| 20180801 | Polly | Toaster | 60 |
| 20180802 | Bob | Kettle | 30 |
| 20180804 | Wendy | Kettle | 10 |

1

```
SELECT date,
    item,
    sum (amount) as amt
FROM base_table
GROUP BY 1.2
```

**materialized_view**

| date | item | amt |
|---|---|---|
| 20180801 | Toaster | 100 |
| 20180802 | Kettle | 30 |
| 20180804 | Kettle | 10 |

2

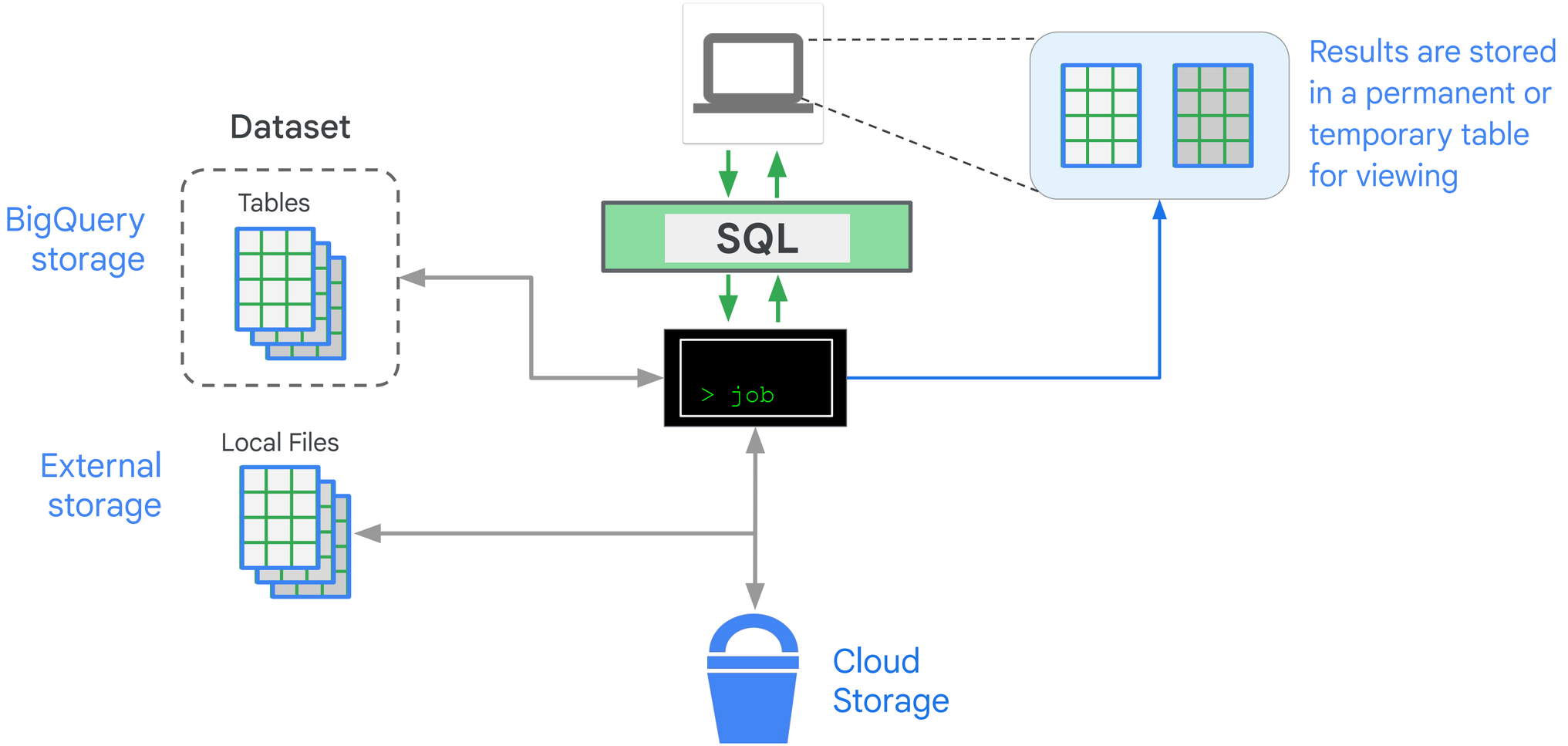
```
SELECT date,
    item
FROM base_table
Where user = “Policy”
```

3

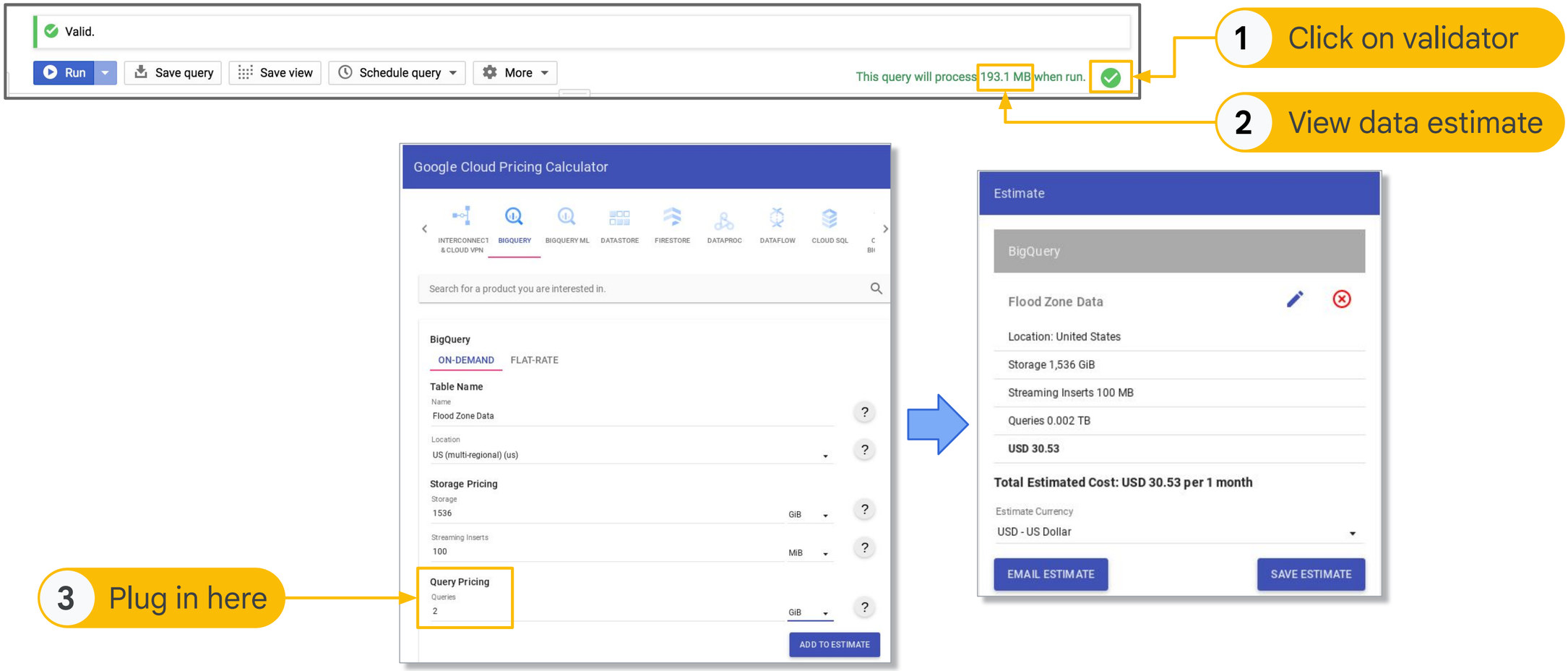
```
SELECT date,
    item,
    sum (amount) as amount
FROM base_table
Where date =  “20180101”
```

1. MV is created and results are periodically refreshed
2. By default, queries continue to go to base table unaffected
3. Applicable queries are rerouted under the hood to MV for faster execution

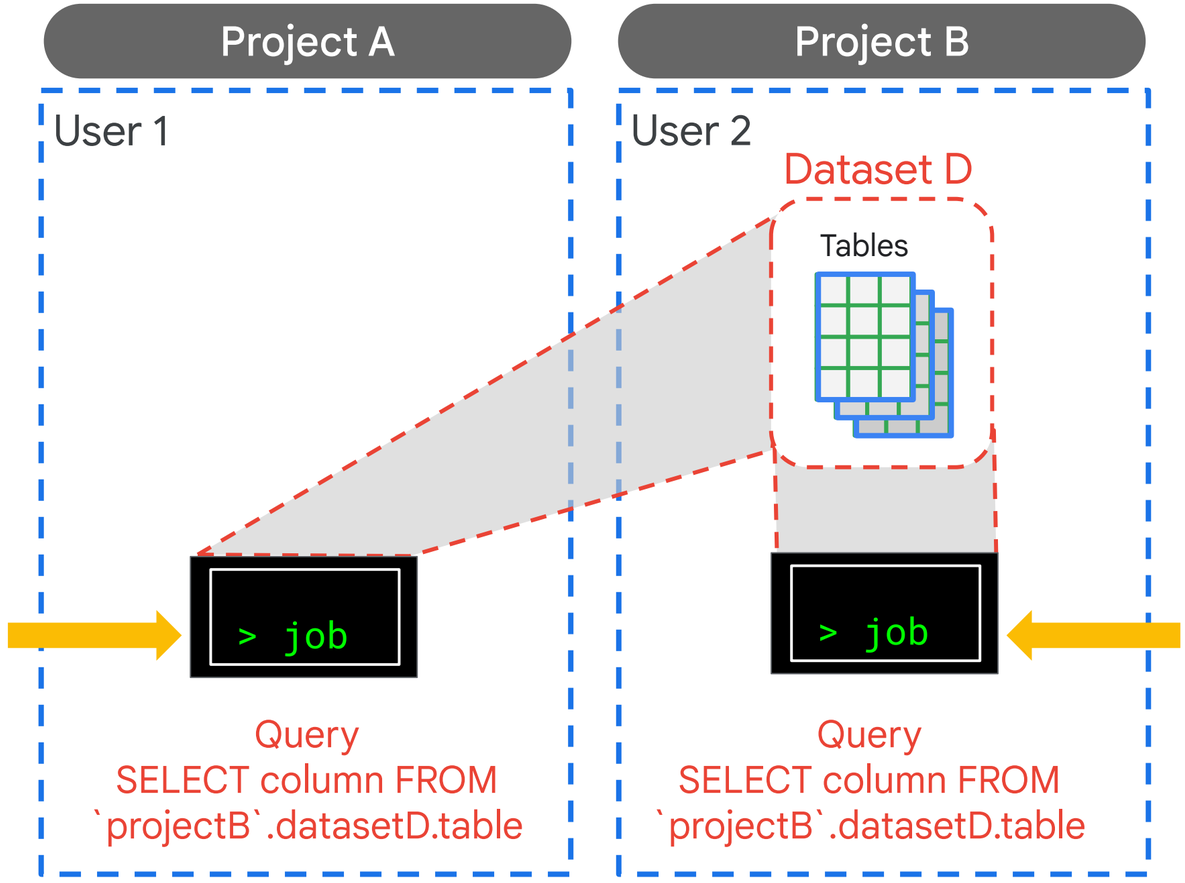
# The life of a BigQuery SQL query



# Use query validator with pricing calculator for estimates



# You can separate cost of storage and cost of queries




**Project A** pays for User 1's queries against data stored in Project B, Dataset D (assuming data access is shared).

**Project B** pays for storage and queries by User 2 executed from Project B.

**NOTE**: The cost of a query is always assigned to the active project from where the query is executed.

Control costs with user and/or project quotas

# Building a Data Warehouse

| | |
|---|---|
| 01 | The modern data warehouse |
| 02 | Introduction to BigQuery |
| 03 | Get started with BigQuery |
| 04 | Load data into BigQuery |
| 05 | Explore schemas |
| 06 | Schema design |
| 07 | Nested and repeated fields |
| 08 | Optimize with partitioning and clustering |

# The method you use to load data depends on how much transformation is needed

## EL

**E**xtract and **L**oad

## ELT

**E**xtract, **L**oad, and **T**ransform

## ETL

**E**xtract, **T**ransform, and **L**oad

# If the data is usable in its original form, just load it

EL

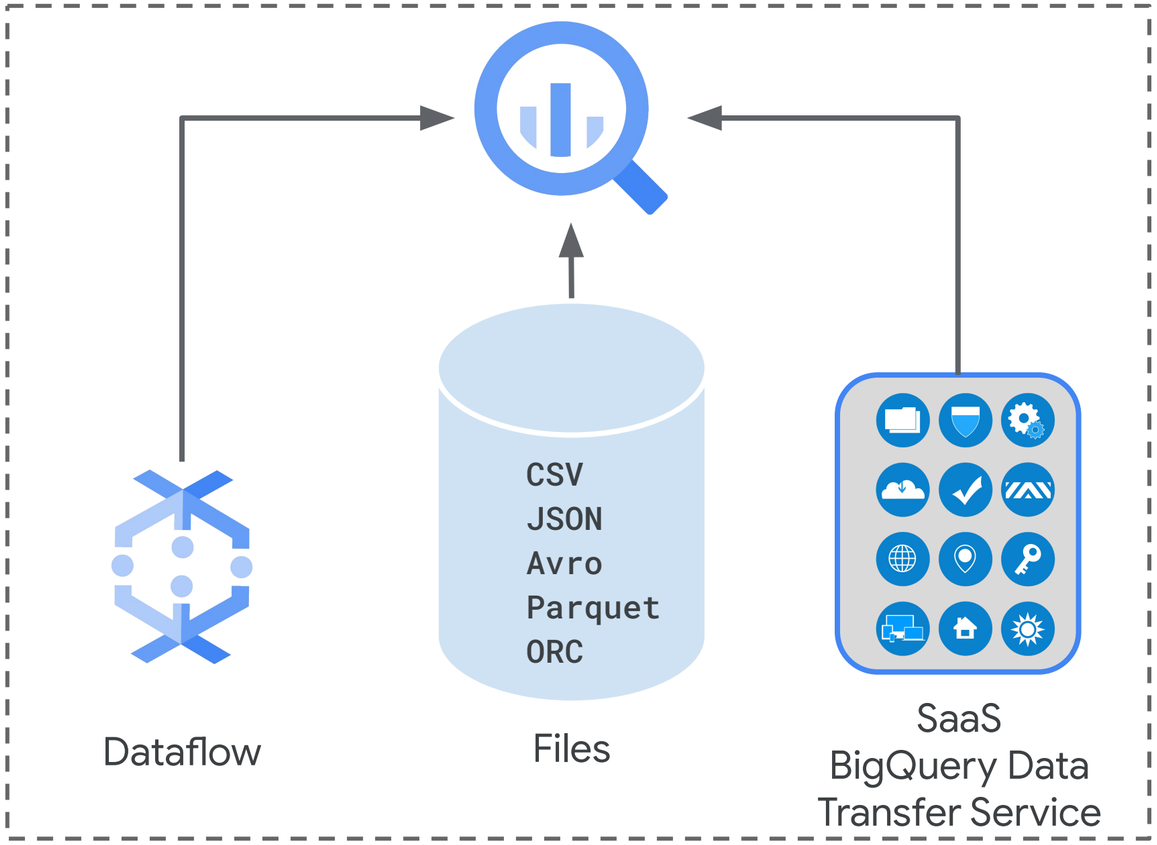
BigQuery
Data Transfer
Service

BigQuery

# Batch load supports different file formats

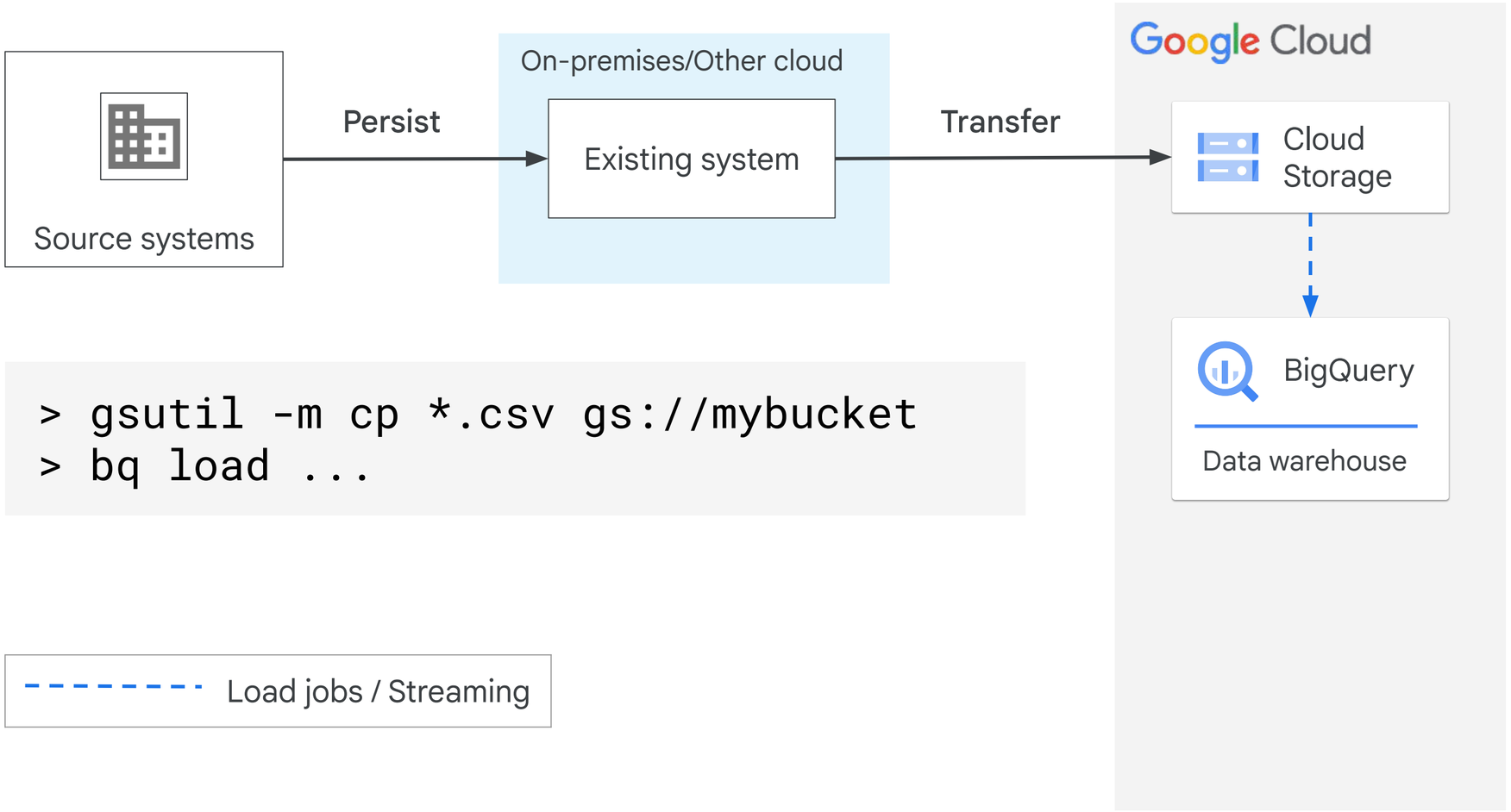
- CSV
- NEWLINE_DELIMITED_JSON
- AVRO
- DATASTORE_BACKUP
- PARQUET
- ORC

# Most common is loading data into BigQuery tables (batch, periodic)




Loading data into BigQuery tables (batch, periodic) offers the best performance.

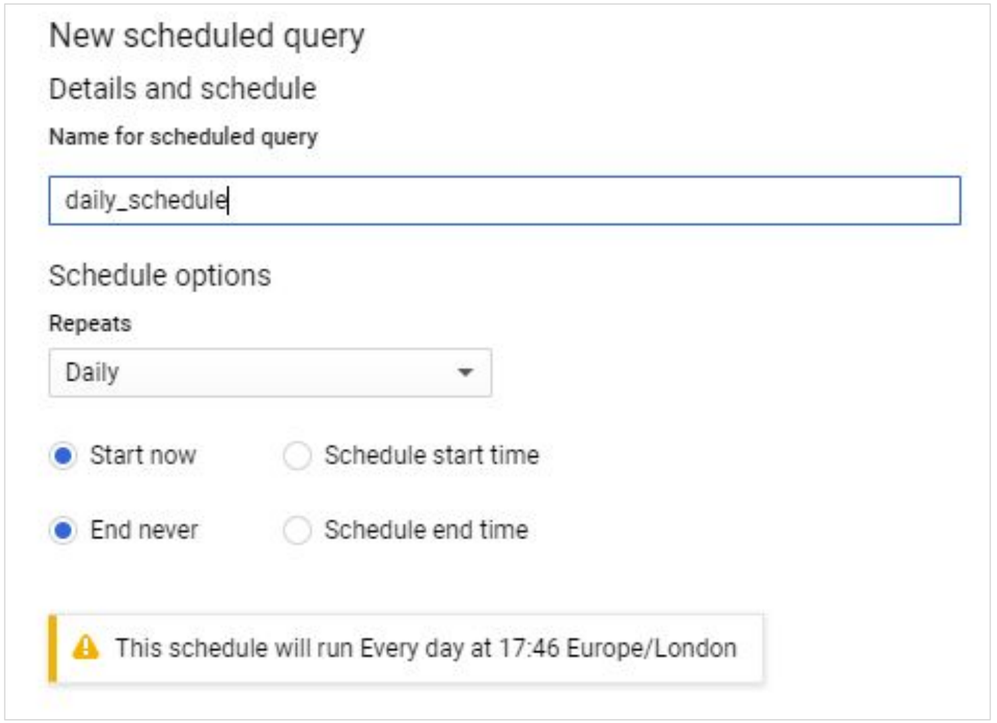
# Loading data through Cloud Storage

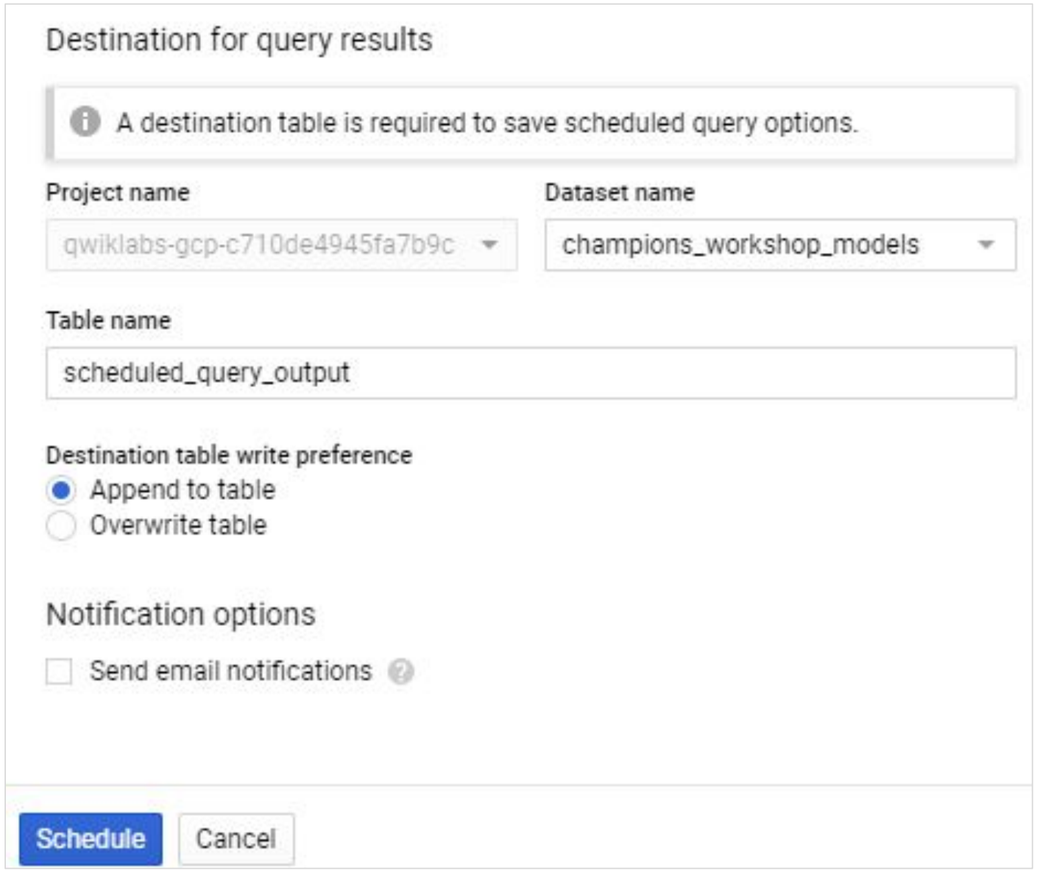



```
> gsutil -m cp *.csv gs://mybucket
> bq load ...
```

Load jobs / Streaming

# Automate the execution of queries based on a schedule



# BigQuery addresses backup and disaster recovery at the service level (time travel)

```
CREATE OR REPLACE TABLE ch10eu.restored_cycle_stations AS
SELECT
*
FROM bigquery-public-data.london_bicycles.cycle_stations
FOR SYSTEM_TIME AS OF
TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 24 HOUR)
```

Query a point-in-time snapshot (up to 7 days).

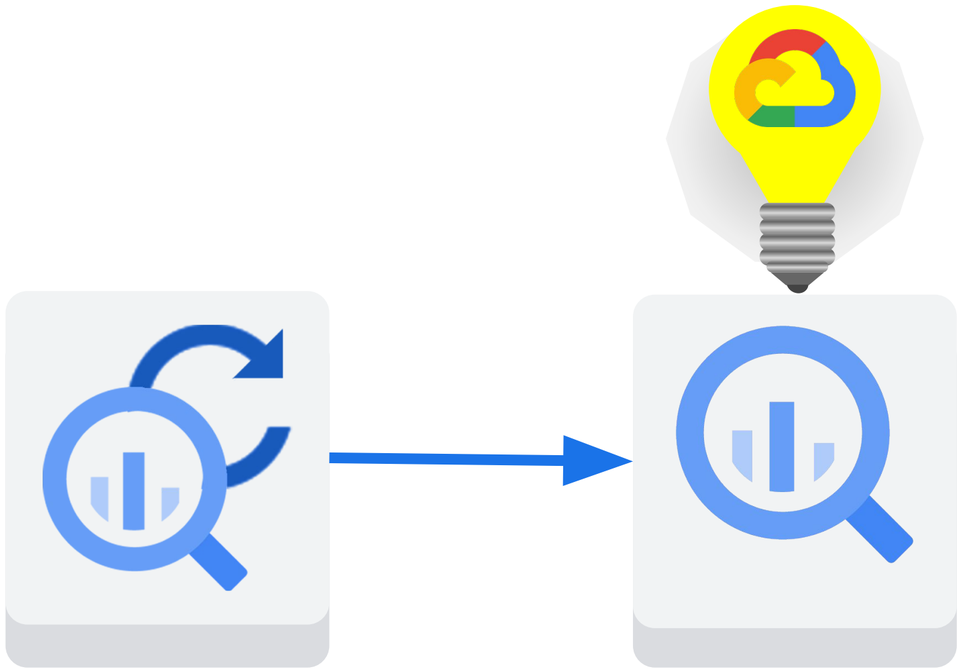
Use it to create a backup.

```
NOW=$(date +%s)
SNAPSHOT=$(echo "($NOW - 120)*1000" | bc)
bq --location=EU cp \
ch10eu.restored_cycle_stations@$SNAPSHOT \
ch10eu.restored_table
```

Restore a 2-min old copy.

(Deleted tables are flushed after 2 days or if recreated with the same name).

# BigQuery Data Transfer Service helps you build and manage your data warehouse



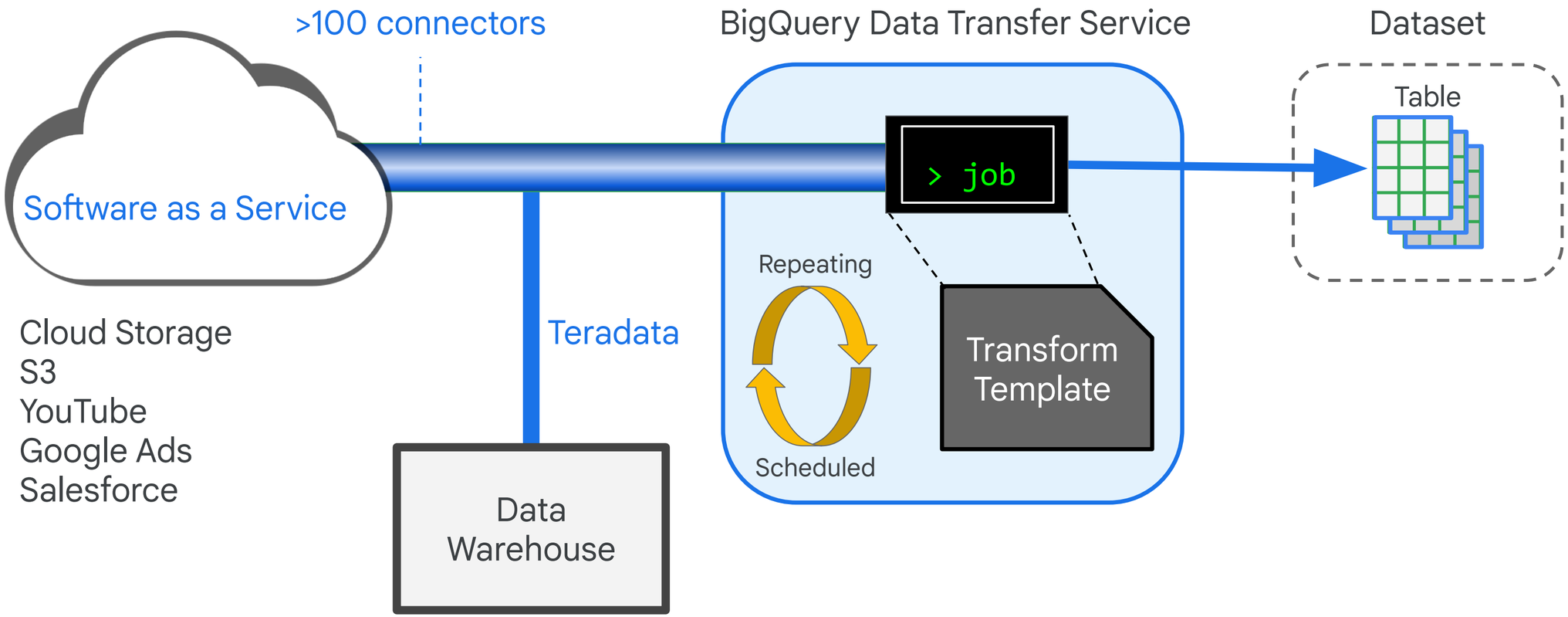
BigQuery Data Transfer Service

Cloud data warehouse

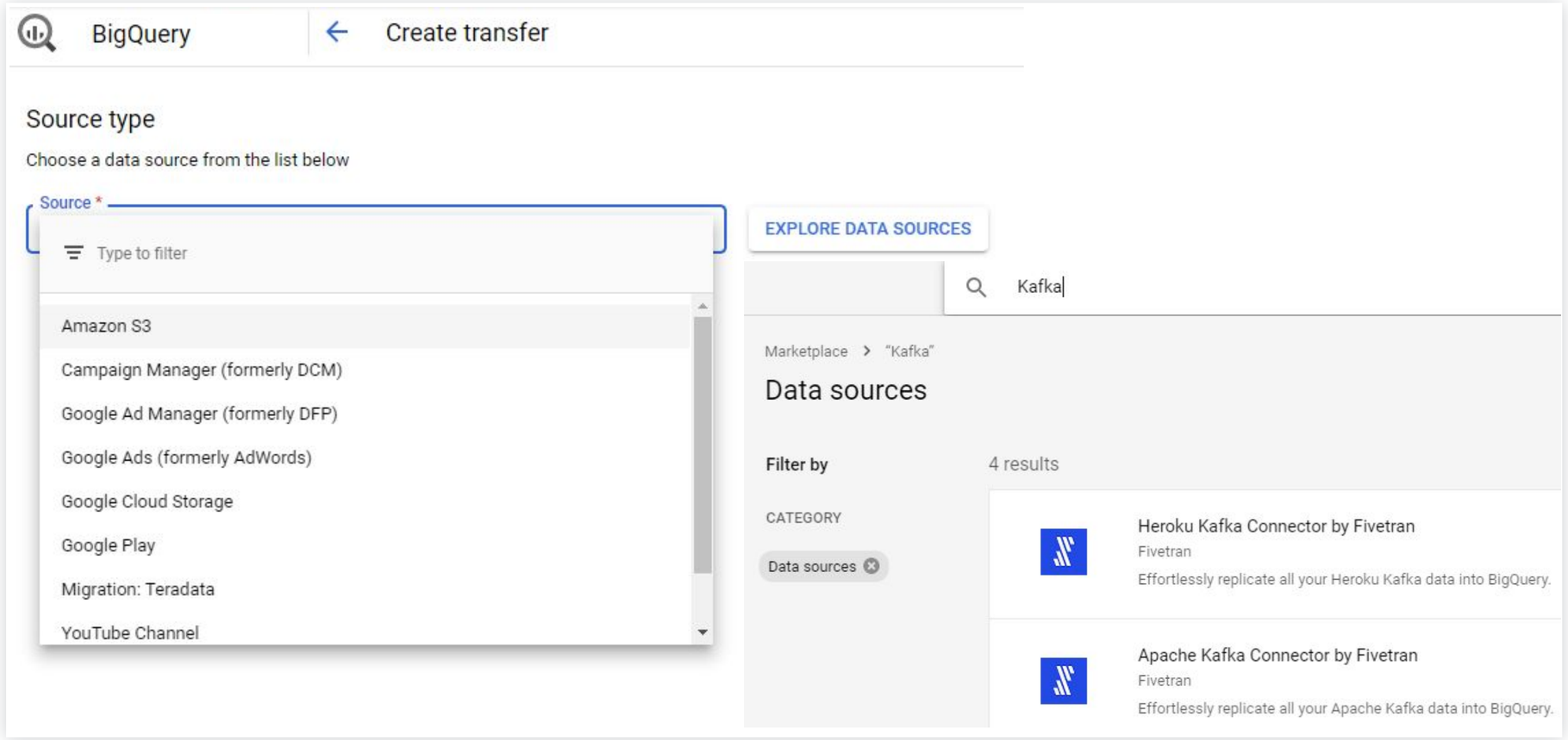
**EL**

- Managed service
- Automatic transfers
- Scheduled
- Data staging
- Data processing
- Data backfills

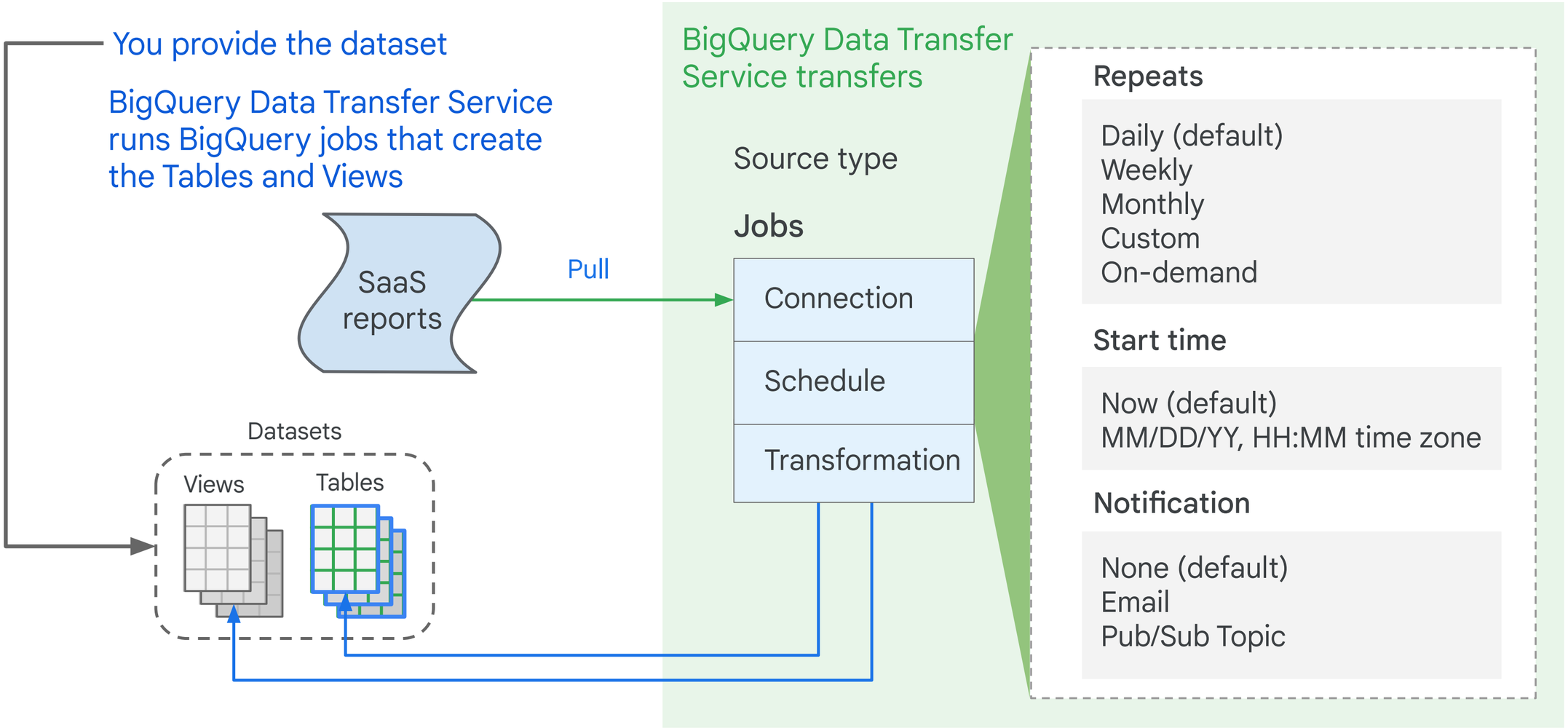
# BigQuery Data Transfer Service provides SaaS connectors



# BigQuery Data Transfer Service supports 100+ SaaS applications



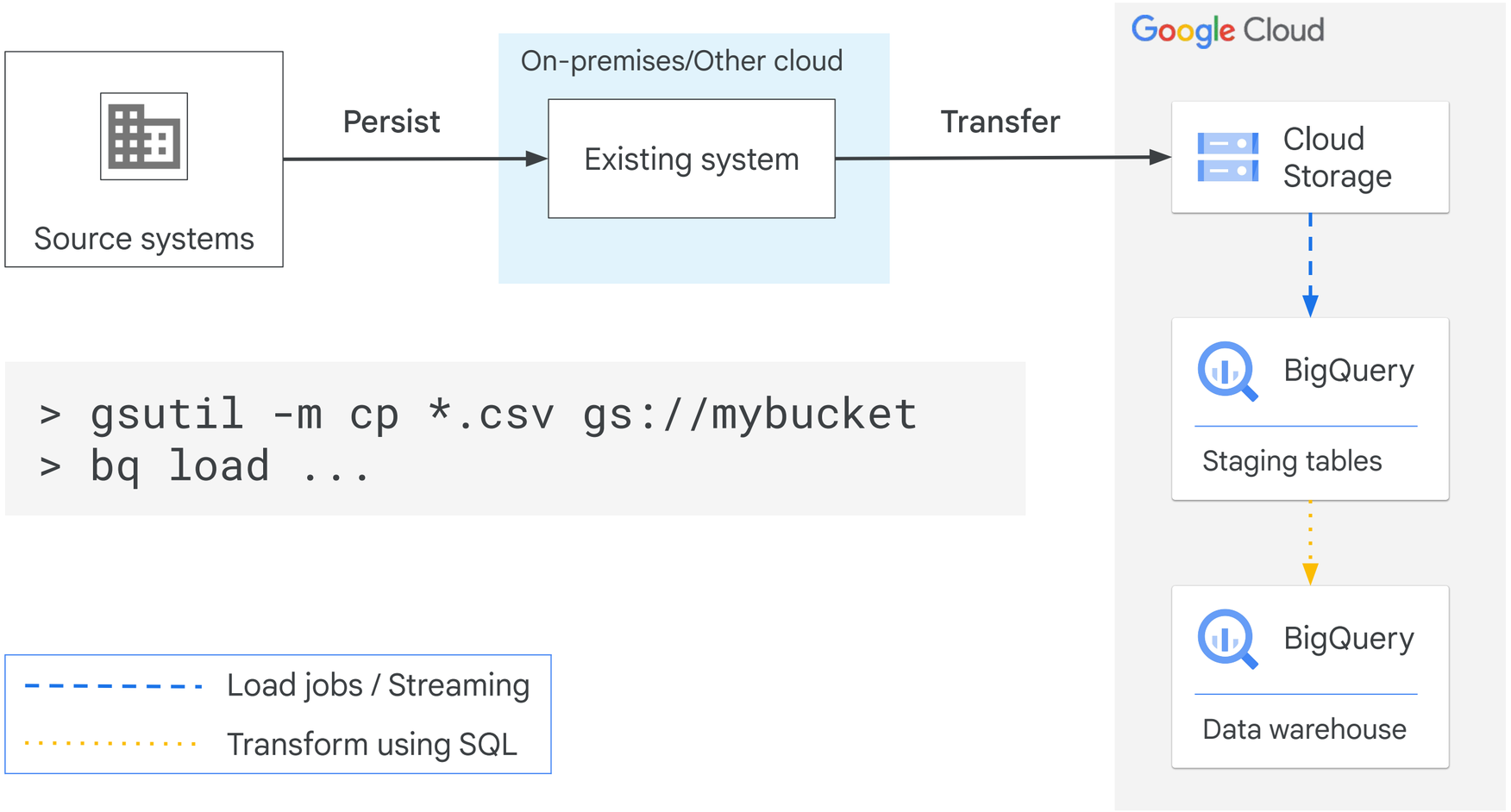
# How BigQuery Data Transfer Service transfers work



# If the data requires simple transformations, such as scaling, maybe it can be handled in SQL

BigQuery

# Loading and transforming data in BigQuery



# Modify table data with standard DML statements

**INSERT**, **UPDATE**, **DELETE**, MERGE records into tables

```
UPDATE table_A
SET
  y = table_B.y,
  z = table_B.z + 1
FROM table_B
WHERE table_A.x = table_B.x
  AND table_A.y IS NULL;

INSERT INTO table VALUES (1,2,3), (4,5,6), (7,8,9);

DELETE FROM table WHERE TRUE;
```

# Create new tables from data with SQL DDL

```
SELECT
    *
FROM
    movielens.movies_raw
WHERE
    movieId < 5;
```

| | movieId | title | genres |
|---|---|---|---|
| 0 | 3 | Grumpier Old Men (1995) | Comedy\|Romance |
| 1 | 4 | Waiting to Exhale (1995) | Comedy\|Drama\|Romance |
| 2 | 2 | Jumanji (1995) | Adventure\|Children\|Fantasy |
| 3 | 1 | Toy Story (1995) | Adventure\|Animation\|Children\|Comedy\|Fantasy |

```
CREATE OR REPLACE TABLE
movielens.movies AS
SELECT
    * REPLACE(SPLIT(genres,
"|") AS genres)
FROM
    Movielens.movies_raw;
-- Execute multiple statements.
SELECT * FROM movielens.movies;
```

| | movieId | title | genres |
|---|---|---|---|
| 0 | 4 | Waiting to Exhale (1995) | [Comedy, Drama, Romance] |
| 1 | 3 | Grumpier Old Men (1995) | [Comedy, Romance] |
| 2 | 2 | Jumanji (1995) | [Adventure, Children, Fantasy] |
| 3 | 1 | Toy Story (1995) | [Adventure, Animation, Children, Comedy, Fantasy] |

**Question:** What's the difference between CREATE OR REPLACE TABLE and CREATE TABLE IF NOT EXISTS ? When would you use each?

# Custom transformations? BigQuery supports user-defined functions in SQL, JavaScript, and scripting

Query editor

```
1  CREATE TEMP FUNCTION multiplyInputs(x FLOAT64, y FLOAT64)
2  RETURNS FLOAT64
3  LANGUAGE js AS """
4    return x*y;
5  """;
6  WITH numbers AS
7    (SELECT 1 AS x, 5 as y
8    UNION ALL
9    SELECT 2 AS x, 10 as y
10   UNION ALL
11   SELECT 3 as x, 15 as y)
12 SELECT x, y, multiplyInputs(x, y) as product
13 FROM numbers;
```

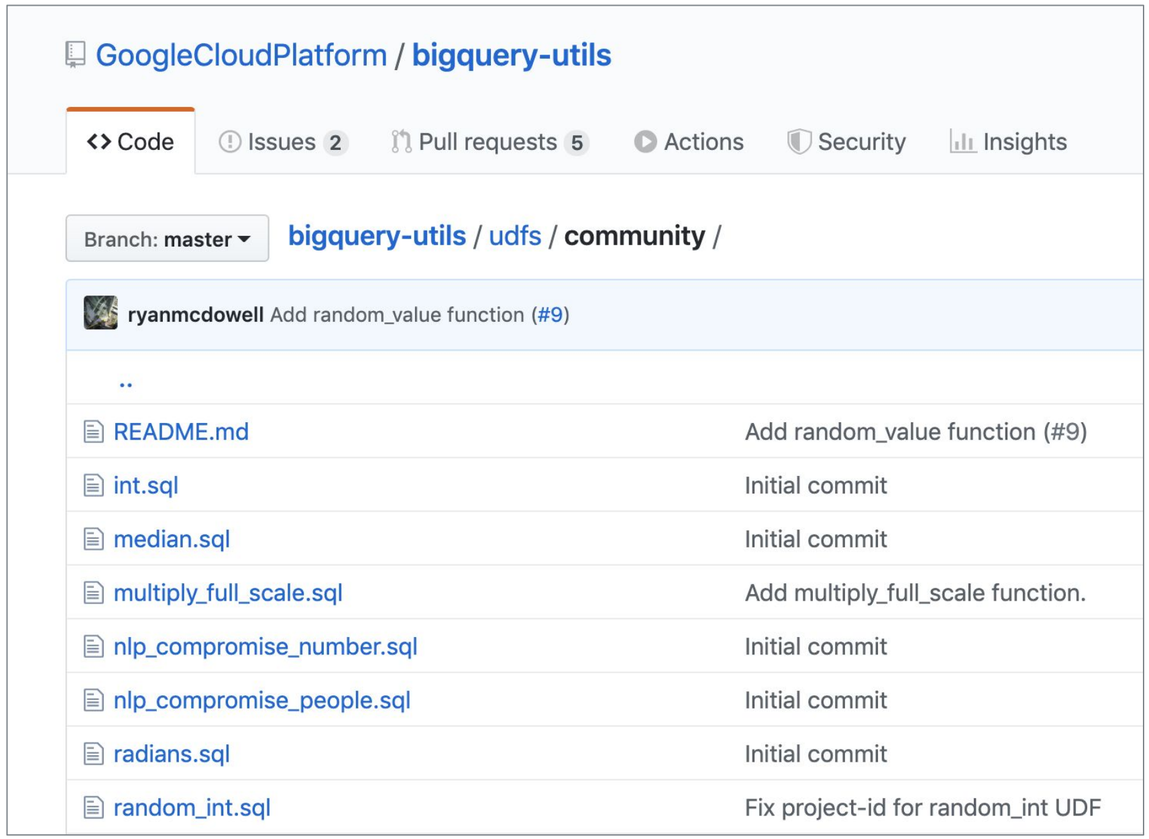
Run | Save query | Save view | Schedule query | More

Query complete (2.5 sec elapsed, 0 B processed)

Job information | Results | JSON | Execution details

| Row | x | y | product |
|---|---|---|---|
| 1 | 1 | 5 | 5.0 |
| 2 | 2 | 10 | 20.0 |
| 3 | 3 | 15 | 45.0 |

# You can persist and share your UDF objects with other team members or publically

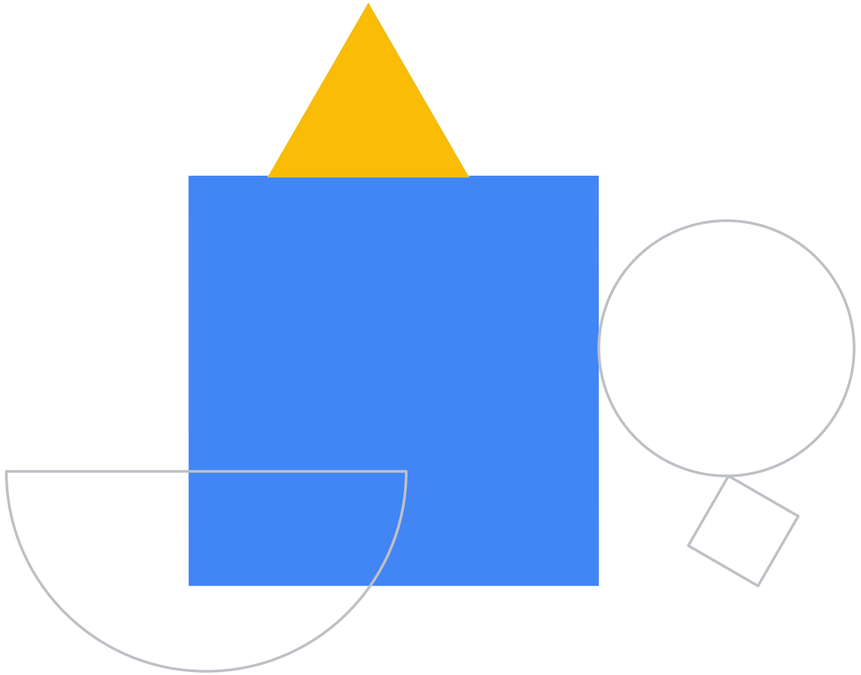



The BigQuery team has a public GitHub repo for common User Defined Functions

https://github.com/GoogleCloudPlatform/bigquery-utils/tree/master/udfs/community

# Lab Intro

Loading Data into BigQuery



# Lab objectives

01 Load data into BigQuery from various sources.

02 Load data into BigQuery using the CLI and the Cloud Console.

03 Use DDL to create tables.

# Building a Data Warehouse



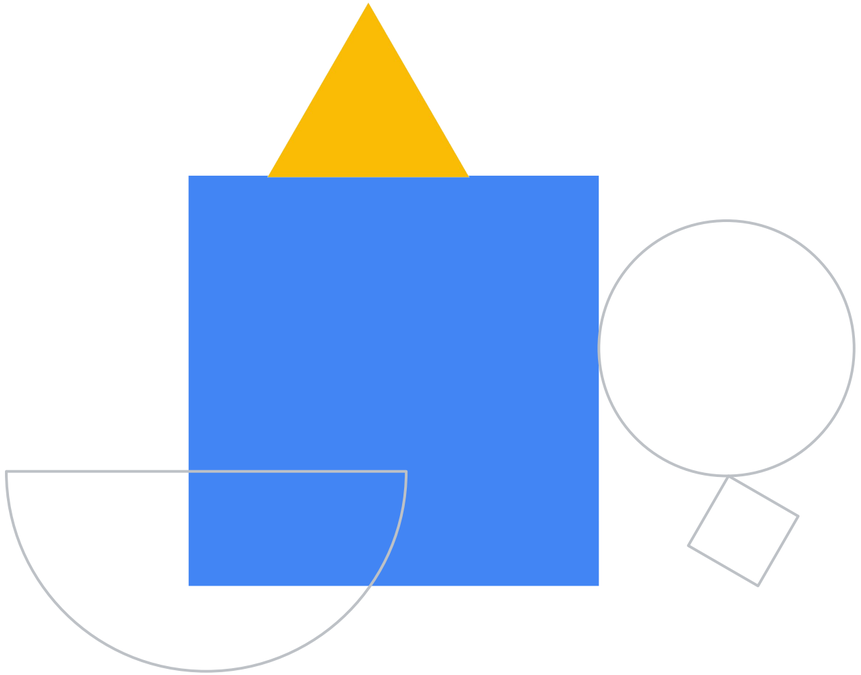
# Designing schemas that scale is a core job of data engineers -- let's explore BigQuery Public Dataset schemas


Public datasets include flights, taxi cab logs, weather recordings, and many more.

https://cloud.google.com/bigquery/public-data/

# Demo

Exploring BigQuery Public Datasets with SQL using INFORMATION_SCHEMA



# Building a Data Warehouse

| | |
|---|---|
| 01 | The modern data warehouse |
| 02 | Introduction to BigQuery |
| 03 | Get started with BigQuery |
| 04 | Load data into BigQuery |
| 05 | Explore schemas |
| 06 | Schema design |
| 07 | Nested and repeated fields |
| 08 | Optimize with partitioning and clustering |

# Transactional databases often use normal form

## Original data

| Customer | OrderID | Date | Items | |
|---|---|---|---|---|
| Doug | 1600p | 8/20/19 | | |
| | | | Product | Quantity |
| | | | Caulk | 3 boxes |
| | | | Soffit | 34 meters |
| | | | Sealant | 2 liters |
| Tom | 221b | 10/29/19 | | |
| | | | Product | Quantity |
| | | | Sealant | 1 liter |
| | | | Soffit | 17 meters |
| | | | Caulk | 4 tubes |

## Normalized data

**Orders**

| Date | OrderID |
|---|---|
| 8/20/2019 | 1600p |
| 10/29/2018 | 221b |

**Order_Items**

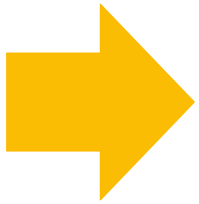
| OrderID | Product | Quantity |
|---|---|---|
| 1600p | Caulk | 3 boxes |
| 221b | Sealant | 1 liter |
| 1600p | Soffit | 34 meters |
| 221b | Soffit | 17 meters |
| 221b | Caulk | 4 tubes |
| 1600p | Sealant | 2 liters |

# Data warehouses often denormalize

## Normalized data

| Orders | |
|---|---|
| Date | OrderID |
| 08/20/2019 | 1600p |
| 10/29/2018 | 221b |

| Order_Items | | |
|---|---|---|
| OrderID | Product | Quantity |
| 1600p | Caulk | 3 boxes |
| 221b | Sealant | 1 liter |
| 1600p | Soffit | 34 meters |
| 221b | Soffit | 17 meters |
| 221b | Caulk | 4 tubes |
| 1600p | Sealant | 2 liters |



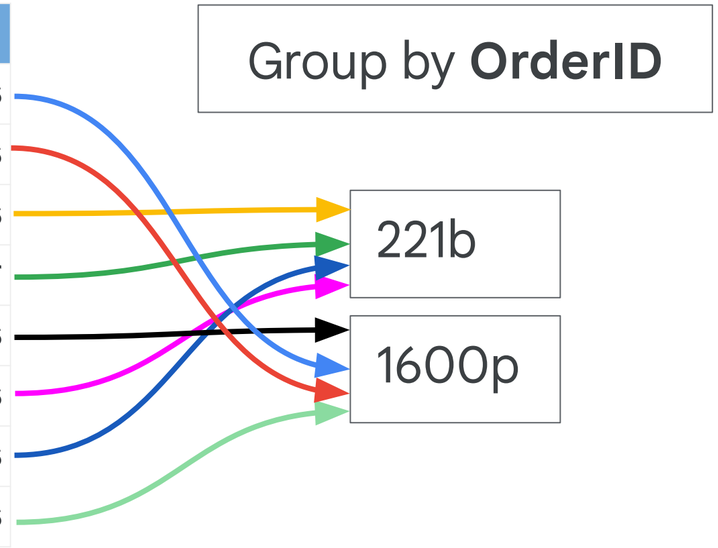
## Denormalized flattened data

| Customer | OrderID | Date | Product | Quantity |
|---|---|---|---|---|
| Doug | 1600p | 08/20/2019 | Siding | 3 boxes |
| Doug | 1600p | 08/20/2019 | Caulk | 12 tubes |
| Tom | 221b | 10/29/2019 | Soffit | 17 meters |
| Tom | 221b | 10/29/2019 | Sealant | 1 liter |
| Doug | 1600p | 08/20/2019 | Soffit | 34 meters |
| Tom | 221b | 10/29/2019 | Siding | 2 boxes |
| Tom | 221b | 10/29/2019 | Caulk | 4 tubes |
| Doug | 1600p | 08/20/2019 | Sealant | 2 liters |

# Grouping on a 1-to-many field in flattened data can cause shuffling of data over the network

Denormalized flattened table

| Customer | OrderID | Date | Product | Quantity |
|---|---|---|---|---|
| Doug | 1600p | 08/20/2019 | Siding | 3 boxes |
| Doug | 1600p | 08/20/2019 | Caulk | 12 tubes |
| Tom | 221b | 10/29/2019 | Soffit | 17 meters |
| Tom | 221b | 10/29/2019 | Sealant | 1 liter |
| Doug | 1600p | 09/09/2018 | Soffit | 34 meters |
| Tom | 221b | 10/29/2019 | Siding | 2 boxes |
| Tom | 221b | 10/29/2019 | Caulk | 4 tubes |
| Doug | 1600p | 08/20/2018 | Sealant | 2 liters |



# Nested and repeated columns improve the efficiency of BigQuery with relational source data

## Denormalized flattened table

| Customer | OrderID | Date | Product | Quantity |
|---|---|---|---|---|
| Doug | 1600p | 08/20/2019 | Siding | 3 boxes |
| Doug | 1600p | 08/20/2019 | Caulk | 12 tubes |
| Tom | 221b | 10/29/2019 | Soffit | 17 meters |
| Tom | 221b | 10/29/2019 | Sealant | 1 liter |
| Doug | 1600p | 8/20/2019 | Soffit | 34 meters |
| Tom | 221b | 10/29/2019 | Siding | 2 boxes |
| Tom | 221b | 10/29/2019 | Caulk | 4 tubes |
| Doug | 1600p | 08/20/2019 | Sealant | 2 liters |

## Denormalized with nested and repeated data

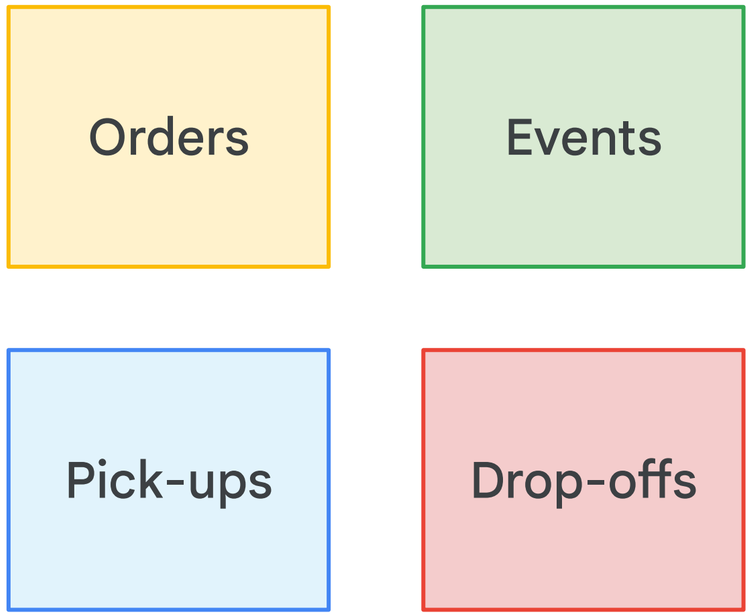
| Order.ID | Order.Date | Order.Product | Order.Quantity |
|---|---|---|---|
| 1600p | 08/20/2019 | Siding | 3 boxes |
| | | Caulk | 12 tubes |
| | | Soffit | 34 meters |
| | | Sealant | 2 liters |
| 221b | 10/29/2019 | Soffit | 17 meters |
| | | Sealant | 1 liter |
| | | Siding | 2 boxes |
| | | Caulk | 4 tubes |

# Building a Data Warehouse

| | |
|---|---|
| 01 | The modern data warehouse |
| 02 | Introduction to BigQuery |
| 03 | Get started with BigQuery |
| 04 | Load data into BigQuery |
| 05 | Explore schemas |
| 06 | Schema design |
| 07 | Nested and repeated fields |
| 08 | Optimize with partitioning and clustering |


GO JEK
GO-JEK is a ride booking service in Indonesia running on Google Cloud

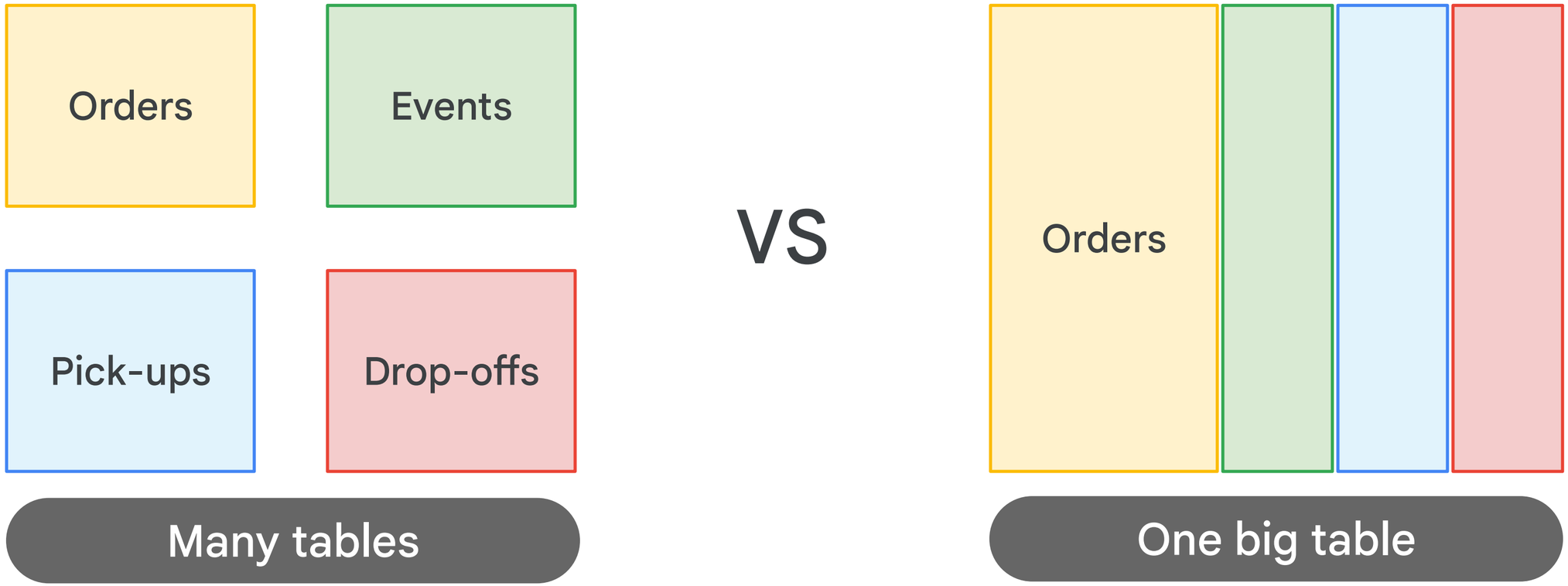
# GO-JEK has 13+PB of data queried each month




- Each ride is stored as an order
- Each ride has a **single** pickup and drop-off
- Each ride can have **one-to-many** events:
  - Ride confirmed
  - Driver en route
  - Pick-up
  - Drop-off

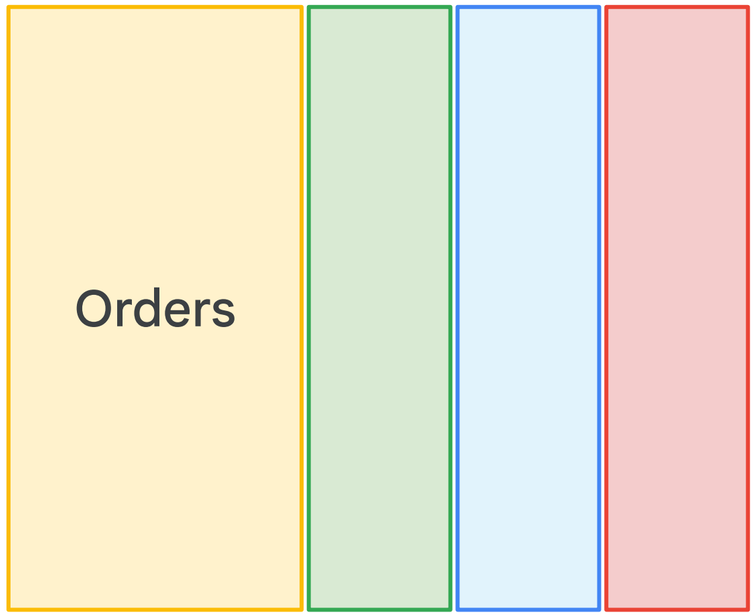
How do you structure your data warehouse for scale? Four separate and large tables that we join together?

# Reporting approach: Should we normalize or denormalize?



# Reporting approach: Should we normalize or denormalize?

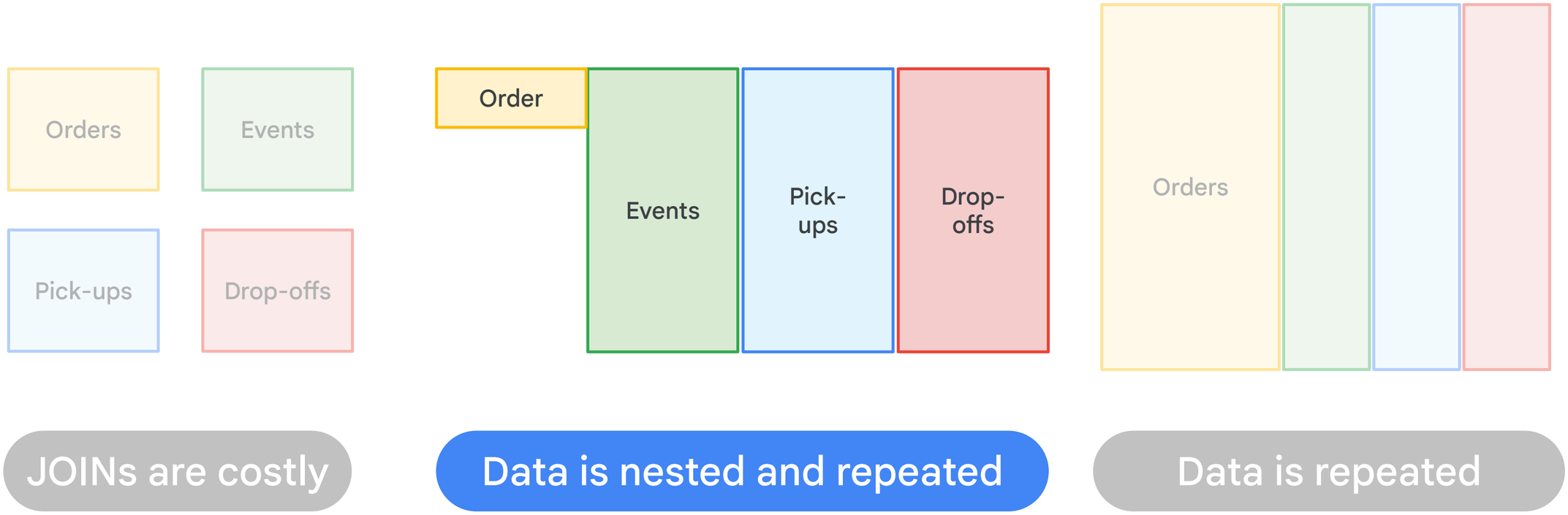

VS




JOINs are costly

Data is repeated

# Nested and Repeated Fields allow you to have multiple levels of data granularity



# Store complex data with nested fields (ARRAYS)

| Row | order_id | service_type | payment_method | event.status | event.time | pickup.latitude | pickup.longitude | destination.latitude | destination.longitude | total_distance_km | pricing_type |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 151 | FD-5117 | GO_FOOD | GOPAY | CREATED | 2018-12-31 04:44:02.545210 UTC | -7.75105 | 110.410561 | -7.7430367 | 110.4046433 | 1.56 | regular |
| | | | | COMPLETED | 2018-12-31 05:06:27.897769 UTC | | | | | | |
| | | | | PICKED_UP | 2018-12-31 04:48:25.945331 UTC | | | | | | |
| | | | | DRIVER_FOUND | 2018-12-31 04:44:06.869010 UTC | | | | | | |
| 152 | FD-6834 | GO_FOOD | CASH | PICKED_UP | 2018-12-31 12:49:52.518880 UTC | 1.121272 | 104.049739 | 1.1368655 | 104.03322 | 4.84 | surge |
| | | | | DRIVER_FOUND | 2018-12-31 12:40:14.214843 UTC | | | | | | |
| | | | | COMPLETED | 2018-12-31 13:04:00.291780 UTC | | | | | | |
| | | | | CREATED | 2018-12-31 12:40:13.431094 UTC | | | | | | |
| 153 | FD-6293 | GO_FOOD | PARTIAL_PAYMENT | PICKED_UP | 2018-12-31 04:33:11.856445 UTC | -7.9657554 | 112.6247491 | -7.9384084 | 112.6227862 | 4.68 | regular |
| | | | | COMPLETED | 2018-12-31 04:56:05.885521 UTC | | | | | | |
| | | | | CREATED | 2018-12-31 04:16:24.356539 UTC | | | | | | |
| | | | | DRIVER_FOUND | 2018-12-31 04:16:25.643766 UTC | | | | | | |
| 154 | FD-7817 | GO_FOOD | CASH | COMPLETED | 2018-12-31 09:14:44.897136 UTC | -6.353915 | 106.247312 | -6.368896 | 106.25787 | 3.51 | regular |
| | | | | PICKED_UP | 2018-12-31 09:01:11.471274 UTC | | | | | | |
| | | | | CREATED | 2018-12-31 08:40:31.821796 UTC | | | | | | |
| | | | | DRIVER_FOUND | 2018-12-31 08:40:32.910319 UTC | | | | | | |

Table JSON

First < Prev Rows 151 - 154 of 1137 Next > Last

# Report on all data in once place with STRUCTS

| Row | order_id | service_type | payment_method | event.status | event.time | pickup.latitude | pickup.longitude | destination.latitude | destination.longitude | total_distance_km | pricing_type |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 151 | FD-5117 | GO_FOOD | GOPAY | CREATED | 2018-12-31 04:44:02.545210 UTC | -7.75105 | 110.410561 | -7.7430367 | 110.4046433 | 1.56 | regular |
| | | | | COMPLETED | 2018-12-31 05:06:27.897769 UTC | | | | | | |
| | | | | PICKED_UP | 2018-12-31 04:48:25.945331 UTC | | | | | | |
| | | | | DRIVER_FOUND | 2018-12-31 04:44:06.869010 UTC | | | | | | |
| 152 | FD-6834 | GO_FOOD | CASH | PICKED_UP | 2018-12-31 12:49:52.518880 UTC | 1.121272 | 104.049739 | 1.1368655 | 104.03322 | 4.84 | surge |
| | | | | DRIVER_FOUND | 2018-12-31 12:40:14.214843 UTC | | | | | | |
| | | | | COMPLETED | 2018-12-31 13:04:00.291780 UTC | | | | | | |
| | | | | CREATED | 2018-12-31 12:40:13.431094 UTC | | | | | | |
| 153 | FD-6293 | GO_FOOD | PARTIAL_PAYMENT | PICKED_UP | 2018-12-31 04:33:11.856445 UTC | -7.9657554 | 112.6247491 | -7.9384084 | 112.6227862 | 4.68 | regular |
| | | | | COMPLETED | 2018-12-31 04:56:05.885521 UTC | | | | | | |
| | | | | CREATED | 2018-12-31 04:16:24.356539 UTC | | | | | | |
| | | | | DRIVER_FOUND | 2018-12-31 04:16:25.643766 UTC | | | | | | |
| 154 | FD-7817 | GO_FOOD | CASH | COMPLETED | 2018-12-31 09:14:44.897136 UTC | -6.353915 | 106.247312 | -6.368896 | 106.25787 | 3.51 | regular |
| | | | | PICKED_UP | 2018-12-31 09:01:11.471274 UTC | | | | | | |
| | | | | CREATED | 2018-12-31 08:40:31.821796 UTC | | | | | | |
| | | | | DRIVER_FOUND | 2018-12-31 08:40:32.910319 UTC | | | | | | |

Table JSON

First < Prev Rows 151 - 154 of 1137 Next > Last

# Nested ARRAY fields and STRUCT fields allow for differing data granularity in the same table

| Row | order_id | service_type | payment_method | event.status | event.time | pickup.latitude | pickup.longitude | destination.latitude | destination.longitude | total_distance_km | pricing_type |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 151 | FD-5117 | GO_FOOD | GOPAY | CREATED | 2018-12-31 04:44:02.545210 UTC | -7.75105 | 110.410561 | -7.7430367 | 110.4046433 | 1.56 | regular |
| | | | | COMPLETED | 2018-12-31 05:06:27.897769 UTC | | | | | | |
| | | | | PICKED_UP | 2018-12-31 04:48:25.945331 UTC | | | | | | |
| | | | | DRIVER_FOUND | 2018-12-31 04:44:06.869010 UTC | | | | | | |
| 152 | FD-6834 | GO_FOOD | CASH | PICKED_UP | 2018-12-31 12:49:52.518880 UTC | 1.121272 | 104.049739 | 1.1368655 | 104.03322 | 4.84 | surge |
| | | | | DRIVER_FOUND | 2018-12-31 12:40:14.214843 UTC | | | | | | |
| | | | | COMPLETED | 2018-12-31 13:04:00.291780 UTC | | | | | | |
| | | | | CREATED | 2018-12-31 12:40:13.431094 UTC | | | | | | |
| 153 | FD-6293 | GO_FOOD | PARTIAL_PAYMENT | PICKED_UP | 2018-12-31 04:33:11.856445 UTC | -7.9657554 | 112.6247491 | -7.9384084 | 112.6227862 | 4.68 | regular |
| | | | | COMPLETED | 2018-12-31 04:56:05.885521 UTC | | | | | | |
| | | | | CREATED | 2018-12-31 04:16:24.356539 UTC | | | | | | |
| | | | | DRIVER_FOUND | 2018-12-31 04:16:25.643766 UTC | | | | | | |
| 154 | FD-7817 | GO_FOOD | CASH | COMPLETED | 2018-12-31 09:14:44.897136 UTC | -6.353915 | 106.247312 | -6.368896 | 106.25787 | 3.51 | regular |
| | | | | PICKED_UP | 2018-12-31 09:01:11.471274 UTC | | | | | | |
| | | | | CREATED | 2018-12-31 08:40:31.821796 UTC | | | | | | |
| | | | | DRIVER_FOUND | 2018-12-31 08:40:32.910319 UTC | | | | | | |

Table JSON

First < Prev Rows 151 - 154 of 1137 Next > Last

# Your turn

- Practice reading the new schema
- Spot the STRUCTS
- Type RECORD = STRUCTS

booking

Schema | Details | Preview

| Field name | Type | Mode |
|---|---|---|
| **order_id** | STRING | NULLABLE |
| **service_type** | STRING | NULLABLE |
| **payment_method** | STRING | NULLABLE |
| **event** | RECORD | REPEATED |
| event. **status** | STRING | NULLABLE |
| event. **time** | TIMESTAMP | NULLABLE |
| **pickup** | RECORD | NULLABLE |
| pickup. **latitude** | FLOAT | NULLABLE |
| pickup. **longitude** | FLOAT | NULLABLE |
| **destination** | RECORD | NULLABLE |
| destination. **latitude** | FLOAT | NULLABLE |
| destination. **longitude** | FLOAT | NULLABLE |
| **total_distance_km** | FLOAT | NULLABLE |
| **pricing_type** | STRING | NULLABLE |
| **duration** | RECORD | NULLABLE |
| duration. **booking_to_dispatch** | FLOAT | NULLABLE |
| duration. **booking_to_pickup** | FLOAT | NULLABLE |

# Practice reading the new schema

- Practice reading the new schema
- Spot the STRUCTS
- Type RECORD = STRUCTS

booking

Schema | Details | Preview

| Field name | Type | Mode |
|---|---|---|
| **order_id** | STRING | NULLABLE |
| **service_type** | STRING | NULLABLE |
| **payment_method** | STRING | NULLABLE |
| **event** | RECORD | REPEATED |
| event. **status** | STRING | NULLABLE |
| event. **time** | TIMESTAMP | NULLABLE |
| **pickup** | RECORD | NULLABLE |
| pickup. **latitude** | FLOAT | NULLABLE |
| pickup. **longitude** | FLOAT | NULLABLE |
| **destination** | RECORD | NULLABLE |
| destination. **latitude** | FLOAT | NULLABLE |
| destination. **longitude** | FLOAT | NULLABLE |
| **total_distance_km** | FLOAT | NULLABLE |
| **pricing_type** | STRING | NULLABLE |
| **duration** | RECORD | NULLABLE |
| duration. **booking_to_dispatch** | FLOAT | NULLABLE |
| duration. **booking_to_pickup** | FLOAT | NULLABLE |

Events

Pick-ups

Destination

Duration

# Your turn

- Practice reading the new schema
- Spot the ARRAYS
- Hint: Look at Mode

booking

Schema | Details | Preview

| Field name | Type | Mode |
|---|---|---|
| **order_id** | STRING | NULLABLE |
| **service_type** | STRING | NULLABLE |
| **payment_method** | STRING | NULLABLE |
| **event** | RECORD | REPEATED |
| event. **status** | STRING | NULLABLE |
| event. **time** | TIMESTAMP | NULLABLE |
| **pickup** | RECORD | NULLABLE |

# Practice reading the new schema

- Practice reading the new schema
- Spot the ARRAYS
- REPEATED = ARRAY

booking

Schema | Details | Preview

| Field name | Type | Mode |
|---|---|---|
| **order_id** | STRING | NULLABLE |
| **service_type** | STRING | NULLABLE |
| **payment_method** | STRING | NULLABLE |
| **event** | RECORD | REPEATED |
| event. **status** | STRING | NULLABLE |
| event. **time** | TIMESTAMP | NULLABLE |
| **pickup** | RECORD | NULLABLE |

Events

| Row | order_id | service_type | payment_method | event.status | event.time |
|---|---|---|---|---|---|
| 151 | FD-5117 | GO_FOOD | GOPAY | CREATED | 2018-12-31 04:44:02.545210 UTC |
| | | | | COMPLETED | 2018-12-31 05:06:27.897769 UTC |

Status and Time in an ARRAY of Event STRUCTs

# Recap

- STRUCTS (RECORD)
- ARRAYS (REPEATED)
- ARRAYS can be part of regular fields or STRUCTS
- A single table can have many STRUCTS

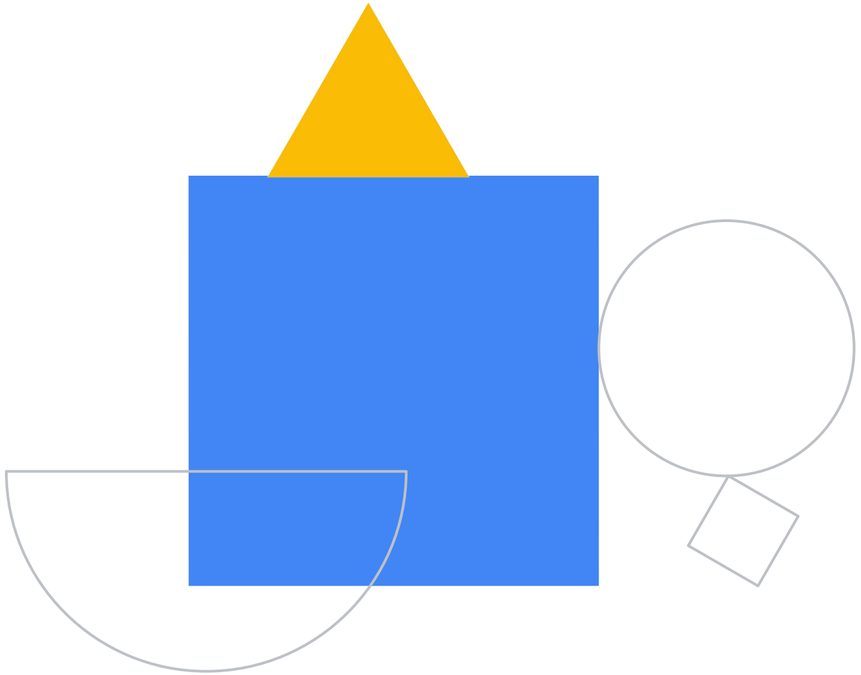
booking

Schema | Details | Preview

| Field name | Type | Mode |
|---|---|---|
| **order_id** | STRING | NULLABLE |
| **service_type** | STRING | NULLABLE |
| **payment_method** | STRING | NULLABLE |
| **event** | RECORD | REPEATED |
| event. **status** | STRING | NULLABLE |
| event. **time** | TIMESTAMP | NULLABLE |
| **pickup** | RECORD | NULLABLE |
| pickup. **latitude** | FLOAT | NULLABLE |
| pickup. **longitude** | FLOAT | NULLABLE |
| **destination** | RECORD | NULLABLE |
| destination. **latitude** | FLOAT | NULLABLE |
| destination. **longitude** | FLOAT | NULLABLE |
| **total_distance_km** | FLOAT | NULLABLE |
| **pricing_type** | STRING | NULLABLE |
| **duration** | RECORD | NULLABLE |
| duration. **booking_to_dispatch** | FLOAT | NULLABLE |
| duration. **booking_to_pickup** | FLOAT | NULLABLE |

# Demo

Nested and repeated fields in BigQuery

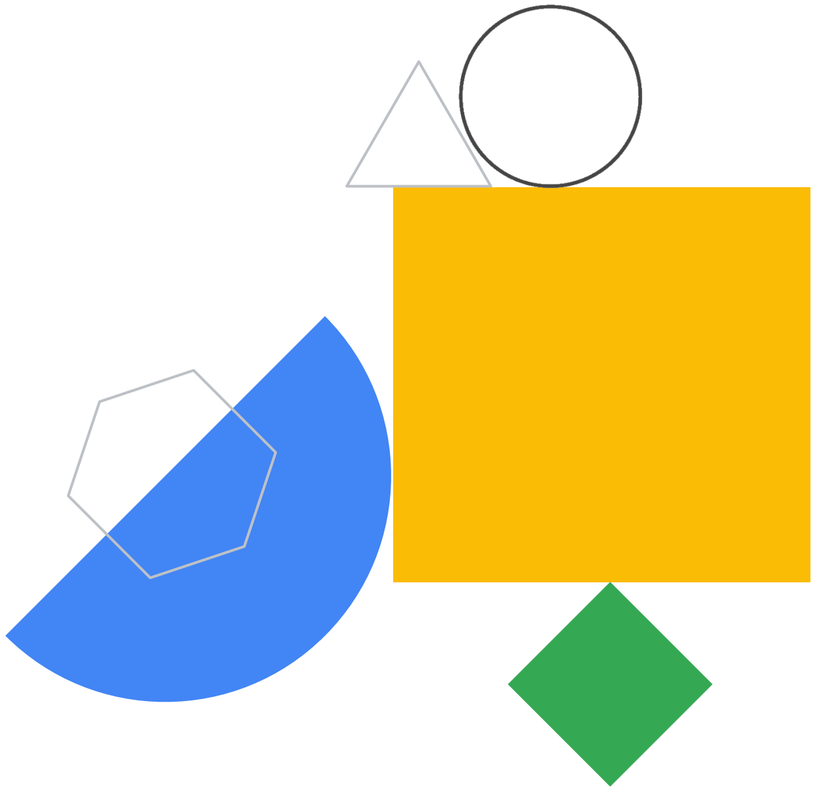


# General guidelines to design the optimal schema for BigQuery

- Instead of joins, take advantage of nested and repeated fields in denormalized tables.
- Keep a dimension table smaller than 10 gigabytes normalized, unless the table rarely goes through `UPDATE` and `DELETE` operations.
- Denormalize a dimension table larger than 10 gigabytes, unless data manipulation or costs outweigh benefits of optimal queries.

# Lab Intro

Working with JSON and Array Data in BigQuery



# Lab objectives

01 Load semi-structured JSON data into BigQuery

02 Learn how to create and query arrays and structs

03 Query nested and repeated fields

# Building a Data Warehouse

| | |
|---|---|
| 01 | The modern data warehouse |
| 02 | Introduction to BigQuery |
| 03 | Get started with BigQuery |
| 04 | Load data into BigQuery |
| 05 | Explore schemas |
| 06 | Schema design |
| 07 | Nested and repeated fields |
| 08 | Optimize with partitioning and clustering |

# Spot the difference!

```
1 SELECT wiki, SUM(views) views
2 FROM `fh-bigquery.wikipedia_v2.pageviews_2017`
3 WHERE datehour >= '2017-01-01'
4 AND wiki IN('en', 'en.m')
5 AND title = 'Kubernetes'
6 GROUP BY wiki ORDER BY wiki
7
```

v2 is not partitioned

Query results SAVE AS EXPLORE IN DATA STUDIO

Query complete (20.706 sec elapsed, 2.2 TB processed)

Job information | Results | JSON | Execution details

| Row | wiki | views |
|---|---|---|
| 1 | en | 343047 |
| 2 | en.m | 90939 |

```
1 SELECT wiki, SUM(views) views
2 FROM `fh-bigquery.wikipedia_v3.pageviews_2017`
3 WHERE datehour >= '2017-01-01'
4 AND wiki IN('en', 'en.m')
5 AND title = 'Kubernetes'
6 GROUP BY wiki ORDER BY wiki
7
```

v3 is partitioned

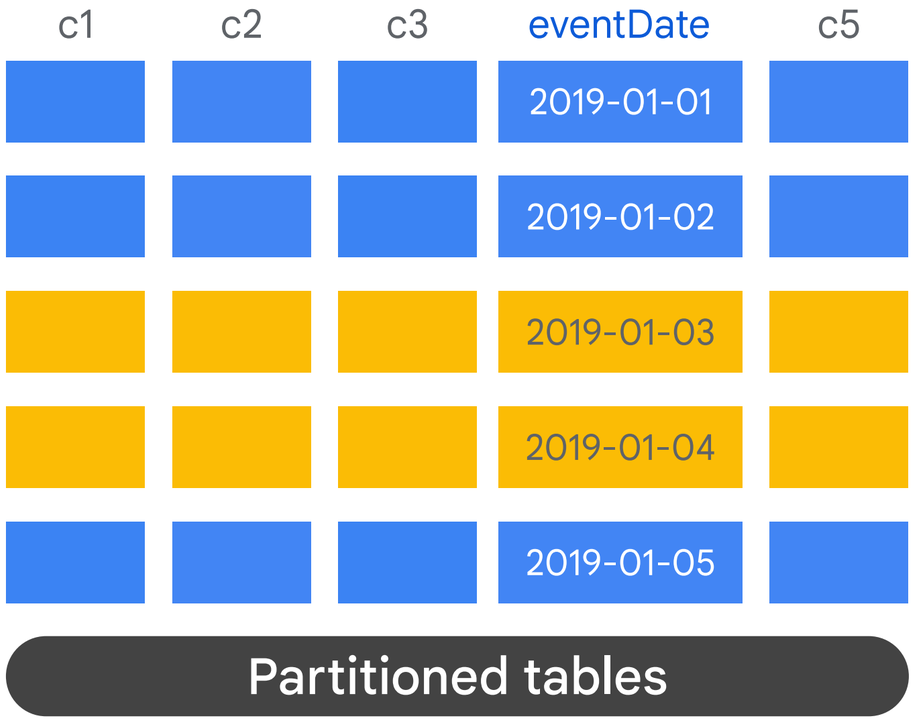
Query results SAVE AS EXPLORE IN DATA STUDIO

Query complete (5.413 sec elapsed, 227.76 GB processed)

Job information | Results | JSON | Execution details

| Row | wiki | views |
|---|---|---|
| 1 | en | 343047 |
| 2 | en.m | 90939 |

# Reduce cost and amount of data read by partitioning your tables




```
SELECT c1, c3 FROM ...
WHERE eventDate BETWEEN
“2019-01-03” AND “2019-01-04”
```

# BigQuery supports three ways of partitioning tables

Ingestion time

```
bq query --destination_table mydataset.mytable
--time_partitioning_type=DAY
...
```

Any column that is of type DATETIME, DATE or TIMESTAMP

```
bq mk --table --schema a:STRING,tm:TIMESTAMP
--time_partitioning_field tm
```

Integer-typed column

```
bq mk --table --schema "customer_id:integer,value:integer"
--range_partitioning=customer_id,0,100,10
my_dataset.my_table
```

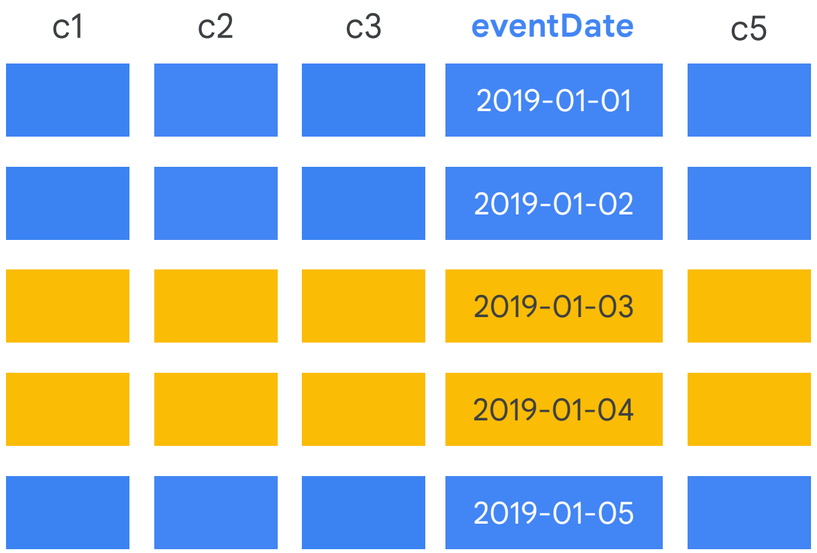
# Partitioning can improve query cost and performance by reducing data being queried

```
SELECT
  field1
FROM
  mydataset.table1
WHERE
  _PARTITIONTIME > TIMESTAMP_SUB(TIMESTAMP('2016-04-15'), INTERVAL 5 DAY)
```

Isolate the partition field in the left-hand side of the query expression!

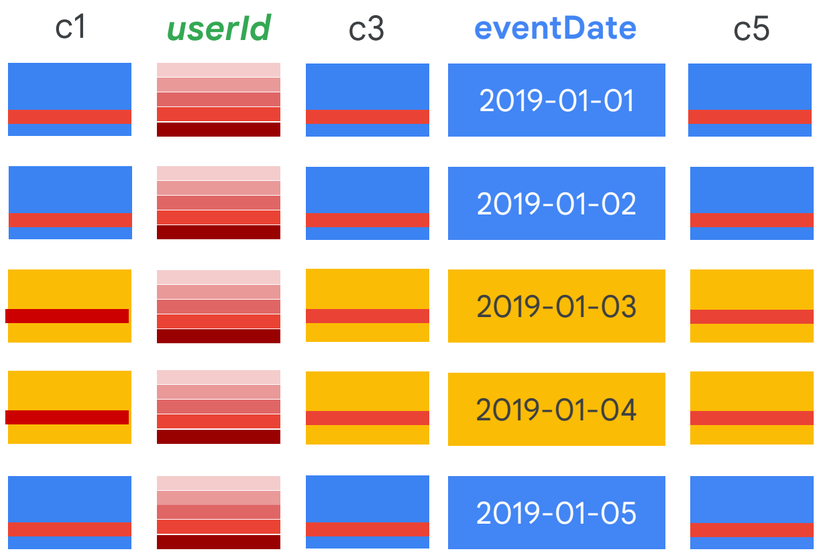
```
bq query \
--destination_table mydataset.mytable
--time_partitioning_type=DAY --require_partition_filter
...
```

# BigQuery automatically sorts the data based on values in the clustering columns




SELECT c1, c3, c5 FROM ...
WHERE eventDate BETWEEN “2019-01-03” AND “2019-01-04”

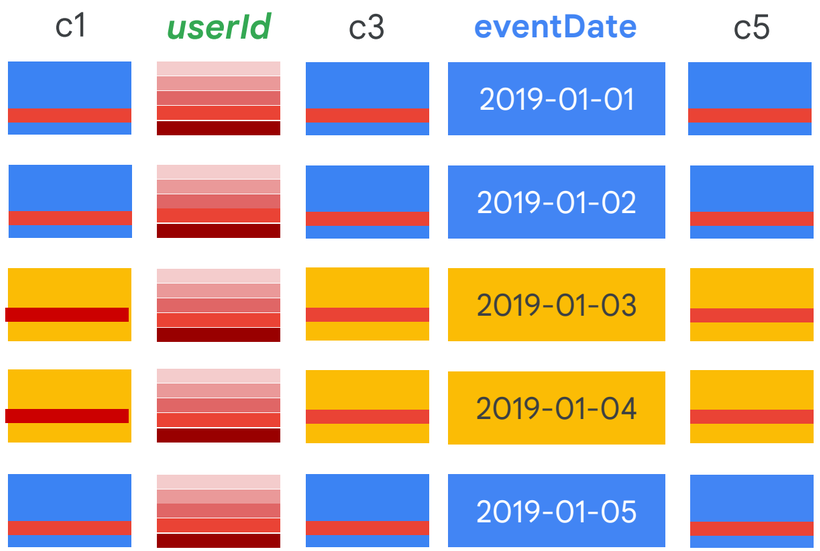
**Partitioned tables**




SELECT c1, c3, c5 FROM ... WHERE *userId* BETWEEN 52 and 63 AND eventDate BETWEEN “2019-01-03” AND “2019-01-04”

**Clustered tables**

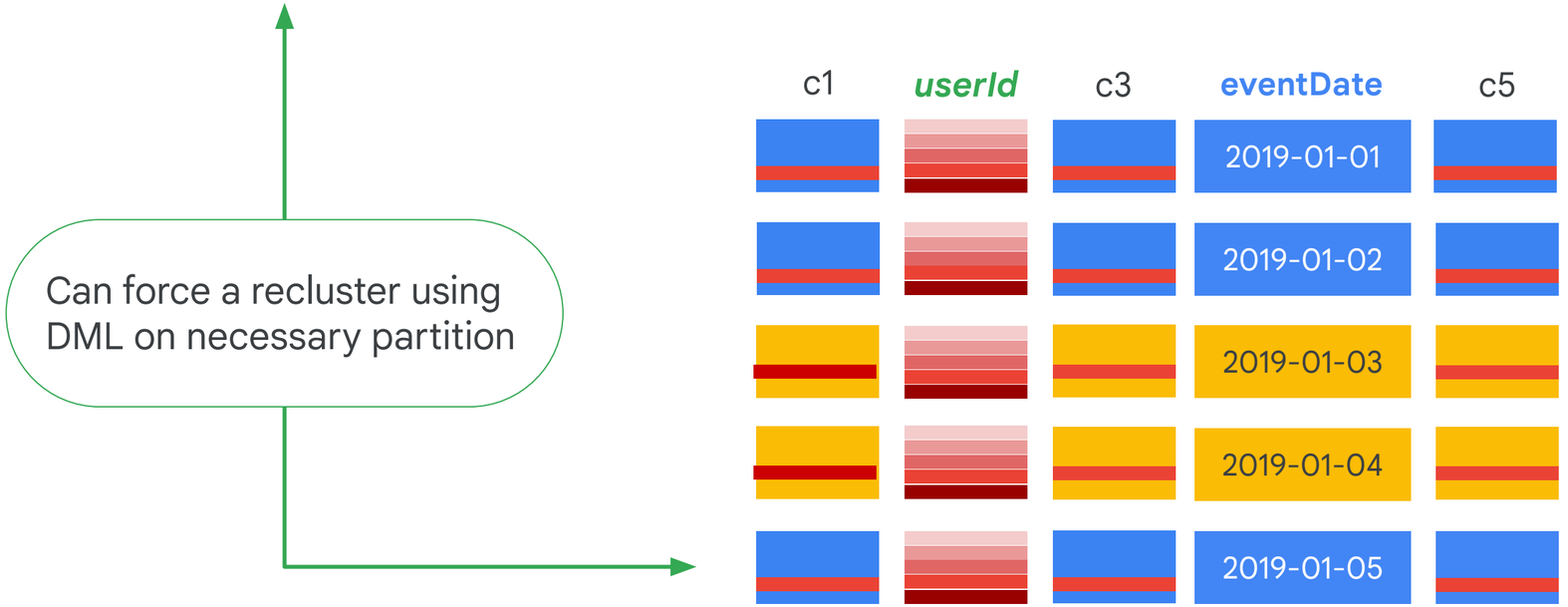
# Set up clustering at table creation time




```
CREATE TABLE mydataset.myclusteredtable
 (
   c1 NUMERIC,
   userId STRING,
   c3 STRING,
   eventDate TIMESTAMP,
   C5 GEOGRAPHY
  )
 PARTITION BY DATE(eventDate)
 CLUSTER BY userId
 OPTIONS (
   partition_expiration_days=3,
   description="cluster")
 AS SELECT * FROM mydataset.myothertable
```

# In streaming tables, the sorting fails over time, and so BigQuery has to recluster

```
UPDATE ds.table
SET c1 = 300
WHERE c1 = 300
AND eventDate > TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 1 DAY)
```



# BigQuery will automatically recluster your data

## Automatic re-clustering

**free**

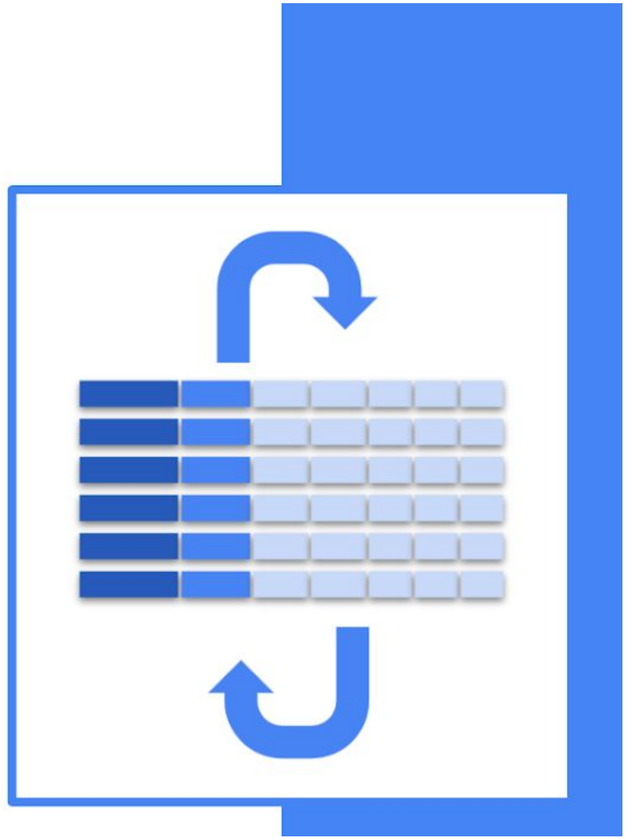
Doesn't consume your query resources

**maintenance-free**

Requires no setup or maintenance

**autonomous**

Automatically  happens in the background



# Organize data through managed tables

## Partitioning

Filtering storage before query execution begins to reduce costs.

Reduces a full table scan to the partitions specified.

A single column results in lower cardinality (e.g., thousands of partitions).

- Time partitioning (Pseudocolumn)
- Time partitioning (User Date/Time column)
- Integer range partitioning

## Clustering

Storage optimization within columnar segments to improve filtering and record colocation.

Clustering performance and cost savings can't be assessed before query begins.

Prioritized clustering of up to 4 columns, on more diverse types (but no nested columns).

# When to use clustering

- Your data is already partitioned on a `DATE`, `DATETIME`, `TIMESTAMP` or Integer Range.
- You commonly use filters or aggregation against particular columns in your queries.

# Review

BigQuery has many capabilities that make it an ideal moderns data warehouse.

BigQuery enables you to structure your information into datasets, projects, and tables.

There are several ways to ingest data into BigQuery.

BigQuery lets you specify a table's schema when you load data into a table.