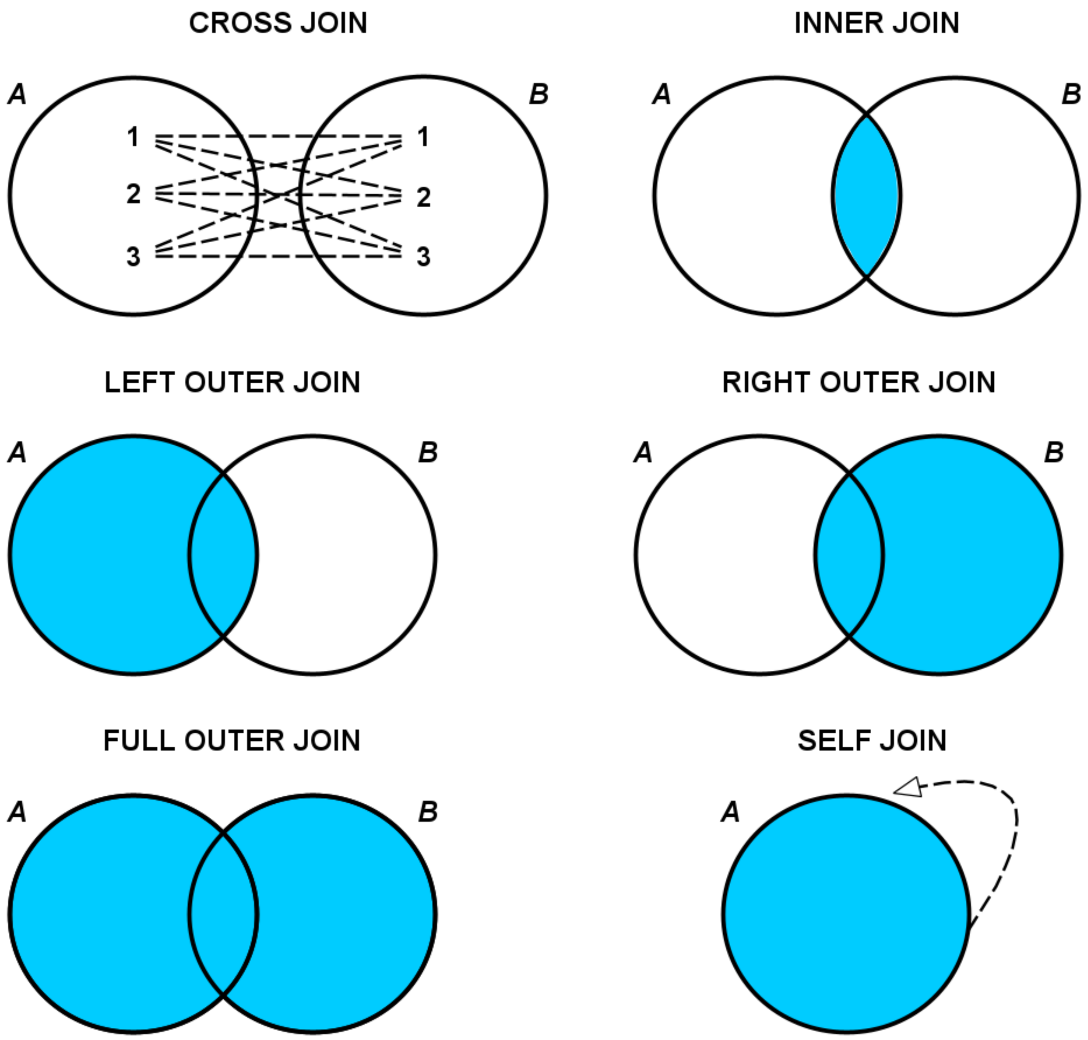
# Joins in BigQuery

INTRODUCTION TO BIGQUERY

**Matthew Forrest**
Field CTO

# Joining data in BigQuery

- `INNER JOIN` : Values exist in both tables.
- `LEFT JOIN` : **All rows in left** table, matches right table.
- `RIGHT JOIN` : **All rows in right** table, matches left table.
- `FULL JOIN` : All rows from **both tables**, matches **and non matches.**
- `CROSS JOIN` : **Every row matched to every row** from both tables.



# Real life examples of joins

**Customers:** left table

**Orders:** right table

- `INNER JOIN` : Matching customers and their orders
- `LEFT JOIN` : Showing all customers, even if they haven't placed any orders
- `RIGHT JOIN` : Showing all orders, even if there is a missing customer ID
- `FULL JOIN` : Showing all customers and all orders, even if some haven't interacted
- `CROSS JOIN` : Match every order to every customer with no conditions

# INNER JOIN

- Only returns matching results from both datasets

```
SELECT
  c.customer_id, s.product_name
FROM customers c
-- The INNER keyword is optional
JOIN sales_data s
ON c.customer_id = s.customer_id;
```

```
| customer_id | product_name          |
|-------------|-----------------------|
| 1           | Bluetooth Headphones |
| 2           | Running Shoes         |
```

# LEFT JOIN

- Returns all rows from the LEFT dataset

```
SELECT
  c.customer_id, s.product_name
FROM customers c
LEFT JOIN sales_data s
ON c.customer_id = s.customer_id;
```

```
| customer_id | product_name         |
|-------------|----------------------|
| 1           | Bluetooth Headphones |
| 2           | Running Shoes        |
| 3           | null                 |
```

# RIGHT JOIN

- Returns all rows from the RIGHT dataset

```
SELECT
  c.customer_id, s.product_name
FROM customers c
RIGHT JOIN sales_data s
ON c.customer_id = s.customer_id;
```

```
| customer_id | product_name          |
|-------------|----------------------|
| 1           | Bluetooth Headphones |
| 2           | Running Shoes        |
| null        | External Microphone  |
```

# OUTER JOIN

- A "RIGHT-LEFT" join: all rows from both RIGHT a and LEFT datasets

```
SELECT
  c.customer_id, s.product_name
FROM customers c
OUTER JOIN sales_data s
ON c.customer_id = s.customer_id;
```

```
| customer_id | product_name         |
|-------------|----------------------|
| 1           | Bluetooth Headphones |
| 2           | Running Shoes        |
| 3           | null                 |
| null        | External Microphone  |
```

# SELF or CROSS JOIN

- A cartesian join - every row with every row

```
SELECT
  c.customer_id,
  s.product_name,

-- Adding table names separated
-- by a comma is a CROSS JOIN
-- Order is determined by the
-- left table, here "customers"
FROM customers c, sales_data s;
```

```
| customer_id | product_name         |
|-------------|----------------------|
| 1           | Bluetooth Headphones |
| 1           | null                 |
| 2           | Bluetooth Headphones |
| 2           | null                 |
| 3           | null                 |
| 3           | Bluetooth Headphones |
```

# Joins and UNNEST

- Also used to join unnested data

```sql
SELECT
  c.customer_id,
  payments.method
FROM customers c,
UNNEST(
  customers.payment_methods
) payments;
```

```
| customer_id | product_name |
|-------------|--------------|
| 1           | Visa         |
| 1           | Mastercard   |
| 1           | Venmo        |
| 1           | Paypal       |
| 2           | Amex         |
| 2           | Visa         |
```

# Let's practice!

INTRODUCTION TO BIGQUERY

# Data manipulation language (DML) statements

INTRODUCTION TO BIGQUERY

**Matt Forrest**
Field CTO

# Overview of data manipulation in BigQuery

- `INSERT` : Add new rows of data.
- `UPDATE` : Modify existing values in a row.
- `DELETE` : Remove unwanted data from a tables.
- `MERGE` : Statement that can combine `INSERT` , `UPDATE` , and `DELETE` statements into one statement.
- `CREATE TABLE AS` : Creates a new table from a query result.

# Considerations and performance

- Group DML statements together when possible rather than running them individually
- You must use a `WHERE` condition when running an `UPDATE` statement
- Consider using table partitions and clusters

[1] https://cloud.google.com/bigquery/docs/reference/standard-sql/data-manipulation-language

# INSERT

- Add records to tables

```sql
-- Define the columns in the parentheses
INSERT INTO customers (customer_id, name, email)

-- Each value is a row to be inserted
VALUES (1, "John Doe", "john.doe@example.com"),
(2, "Jane Doe", "jane.doe@example.com"),
(3, "Alice Smith", "alice.smith@example.com");
```

# UPDATE

- Changing data based on a condition

```
UPDATE customers
-- Set one column for each SET statement
SET email = "john.doe@newdomain.com"
-- Make sure to include where otherwise all
-- rows will be updated
WHERE customer_id = 1;
```

- `UPDATE` together with subqueries or joins

```
UPDATE customers c
SET c.email = e.email
FROM emails e
WHERE c.customer_id = 1;
```

# DELETE

- `DELETE` **permanently** removes records and **can't be reversed**

```sql
DELETE FROM customers

-- Include WHERE to ensure only specific rows are deleted
WHERE customer_id = 3;
```

```sql
DELETE FROM customers c
JOIN emails e USING (customer_id)
WHERE email = 'john.doe@newdomain.com'
```

# MERGE

- Combines `INSERT` , `UPDATE` , and `DELETE` in a single operation

```sql
-- Sets the customers table as the target
MERGE customers AS target
-- The source is set to new_customers
USING new_customers AS source
-- Matching condition
ON target.customer_id = source.customer_id
-- If the emails do not match, update the email
WHEN MATCHED AND target.email != source.email THEN
  UPDATE SET email = source.email
-- If the match is not met, insert the record
WHEN NOT MATCHED THEN
  INSERT (customer_id, name, email) VALUES
  (source.customer_id, source.name, source.email);
```

# CREATE TABLE

- Create new tables from queries

```
CREATE TABLE active_customers AS
SELECT customer_id, name, email
FROM customers
WHERE last_active_date > DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY);
```

# Let's practice!

INTRODUCTION TO BIGQUERY

# Query optimization strategies

INTRODUCTION TO BIGQUERY

**Matt Forrest**
Field CTO

# Three rules of thumb

There are three main optimization rules:

1. **Reduce** the amount of data that needs to be **processed**
2. **Optimize** the query operations
3. **Reduce** the output size of your query

[1] https://cloud.google.com/bigquery/docs/best-practices-performance-compute#use-bi-engine

# Reducing the amount of data

- Avoid using `SELECT *` , and only select the columns of data we need
- Filter the amount of data in our CTEs early and often
- Filter our data using the `WHERE` clause early and often

# Optimizing joins

- Make sure we reduce the data we need using CTEs.
- Join using an `INT64` data type.

```
WITH filter_my_data AS (SELECT
-- Filter data with
-- WHERE in the CTE first
)
SELECT
-- This query will run
-- faster with less data
JOIN a USING (user_id)
```

# Optimizing the WHERE clause

- In BigQuery, use
  - `BOOL`
  - `INT`
  - `FLOAT`
  - `DATE`
- Data types with `WHERE` , `STRING` , or `BYTE` are **not optimal.**

**Not** optimal

```
SELECT user_id, date_ordered
FROM dataset.table
WHERE product = 'shoes'
```

Optimal

```
SELECT user_id, date_ordered
FROM dataset.table
WHERE product_id = 1234
```

# ORDER BY optimizations

- `ORDER BY` should always be at the outermost (end) of our query
- The only exception to this is using `ORDER BY` within a window clause

# ORDER BY without optimization

## Not optimal

```sql
WITH order_total AS (SELECT
user_id,
sum(product_price) as order_sum
FROM orders
GROUP BY user_id
-- Order by is not at the end of the query
ORDER BY last_purchase_date
)
SELECT order_total.order_sum,
users.user_name
FROM dataset.users users
JOIN order_total USING (user_id);
```

# ORDER BY with optimization

## Optimal

```sql
WITH order_total AS (SELECT
user_id,
last_purchase_date
sum(product_price) as order_sum
GROUP BY user_id
)
SELECT order_total.order_sum,
users.user_name
FROM dataset.users users
JOIN a USING (user_id)
-- Order by should always be at the end
ORDER BY orders_total.last_purchase_date;
```

# Using EXISTS vs. COUNT

- If we only need to know if a record is in the table, using `EXISTS`
- Avoid using `COUNT` to solve this use case

```
SELECT EXISTS (

  -- Write the main query as a subquery w
  SELECT
    user_id
  FROM
    dataset.table
  WHERE
    product_category = 'home_goods'
    AND status = 'Closed Account'
);
```

# Other optimization methods

- Use approximate aggregate functions such as `APPROX_TOP_SUM` or `APPROX_COUNT_DISTINCT` .
- Many BigQuery tables are partitioned by date - include dates in the `WHERE` clause.

# Let's practice!

INTRODUCTION TO BIGQUERY

# Congratulations!

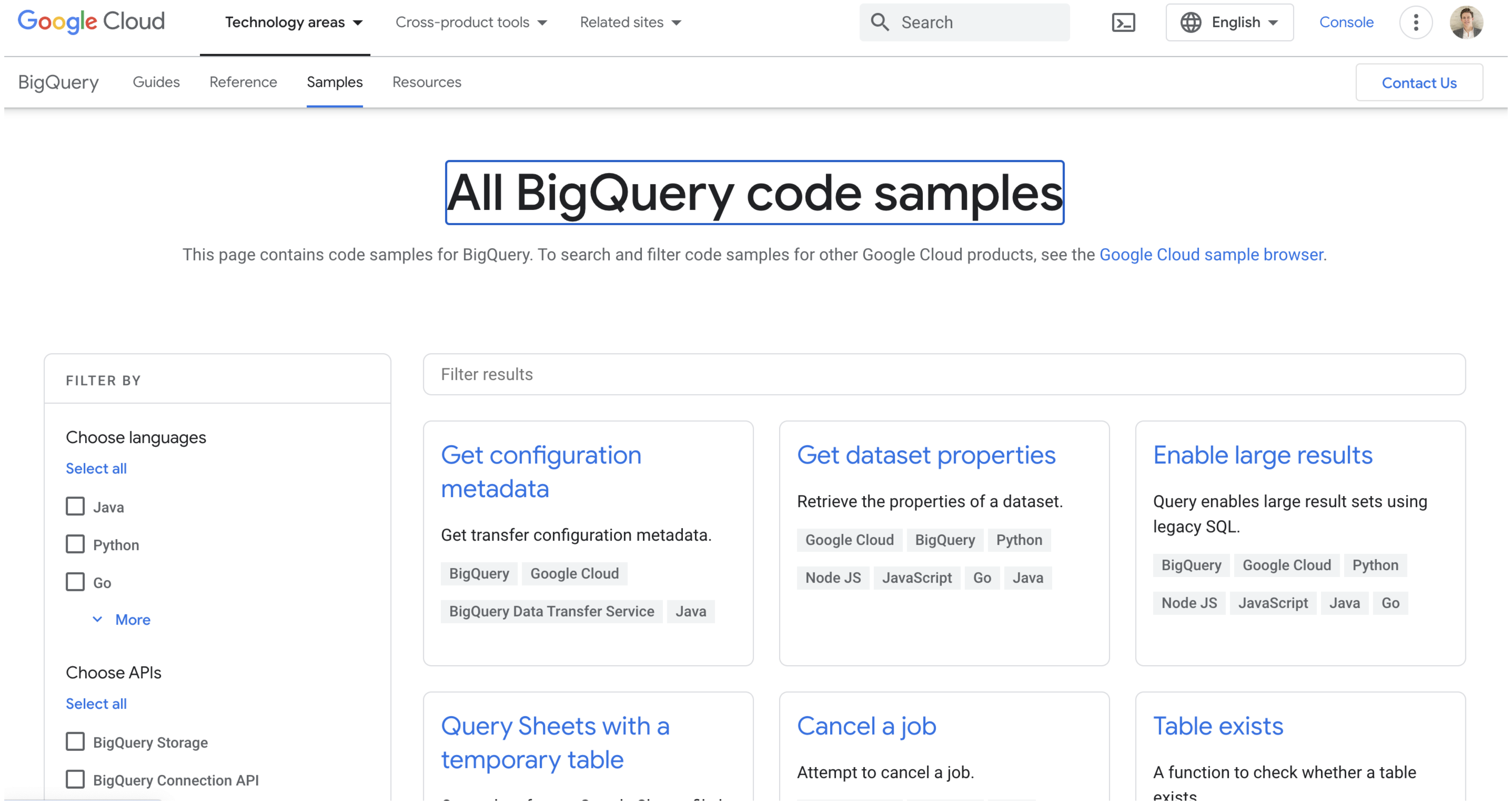
INTRODUCTION TO BIGQUERY

**Matt Forrest**
Field CTO

# What we covered

1. BigQuery architecture, background, and comparisons
2. Data ingestion, data types, and unstructured data
3. Querying data in BigQuery with CTEs, aggregations, and WINDOWs
4. Joins, query optimizations, and data manipulation

# What's next?



## All BigQuery code samples

This page contains code samples for BigQuery. To search and filter code samples for other Google Cloud products, see the Google Cloud sample browser.

FILTER BY

Choose languages

Select all

- □ Java
- □ Python
- □ Go

More

Choose APIs

Select all

- □ BigQuery Storage
- □ BigQuery Connection API

Filter results

### Get configuration metadata

Get transfer configuration metadata.

BigQuery · Google Cloud · BigQuery Data Transfer Service · Java

### Get dataset properties

Retrieve the properties of a dataset.

Google Cloud · BigQuery · Python · Node JS · JavaScript · Go · Java

### Enable large results

Query enables large result sets using legacy SQL.

BigQuery · Google Cloud · Python · Node JS · JavaScript · Java · Go

### Query Sheets with a temporary table

### Cancel a job

Attempt to cancel a job.

### Table exists

A function to check whether a table exists.

# Congrats!

INTRODUCTION TO BIGQUERY