

ANALYTICS SUMMIT

NOVEMBER 4-5, 2025 IN HAMBURG

ANALYTICS SUMMIT

Who we are



Julian Holzner
Senior BI Analyst



Nassim Gharbi
Senior BI Analyst

BigQuery and SQL: Hands-on Intro (Part 1)

ANALYTICS SUMMIT

Housekeeping



Wifi

Please log into this Wifi
`OMCGroup-Guest`

You will be asked to create an account



Slides

You will receive all the slides, including solutions after the seminar.

If you want to take notes:
`analytics-summ.it/slides`



Access to course data

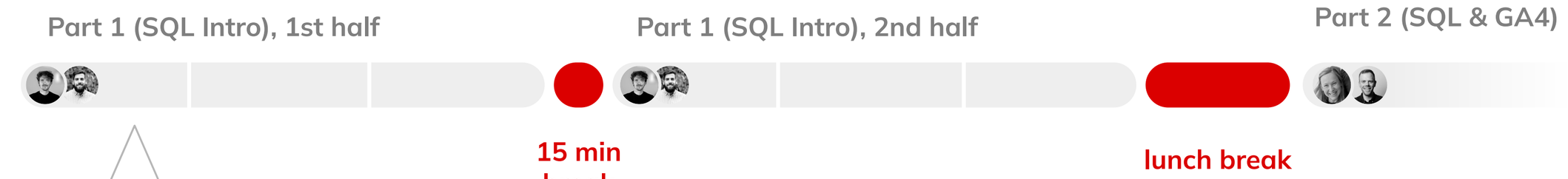
You have been given access to our training data set, please make sure you're logged in with the right Google account.

To test access, please open:
`console.cloud.google.com/bigquery`

| ***TRKKN*** *Proprietary & Confidential*

What to expect

overall structure



chapter structure



Getting to know the topic

Getting to know the details

Seeing what we just learned in action.

(incl. quiz)

Hands-On!

(incl. quiz)

What to expect

Getting data into BigQuery, understanding it, querying it, visualising the results

01 The BQ Interface

02 Importing Data into BQ

03 SQL Basics in 3 Parts

15 min break

04 Mastering Joins

05 How to query nested fields

06 CTE & Subqueries

07 SQL Hygiene & Best Practices

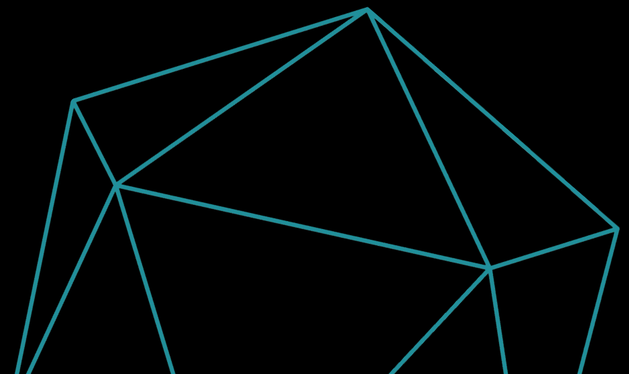
08 Connecting & Visualising Data

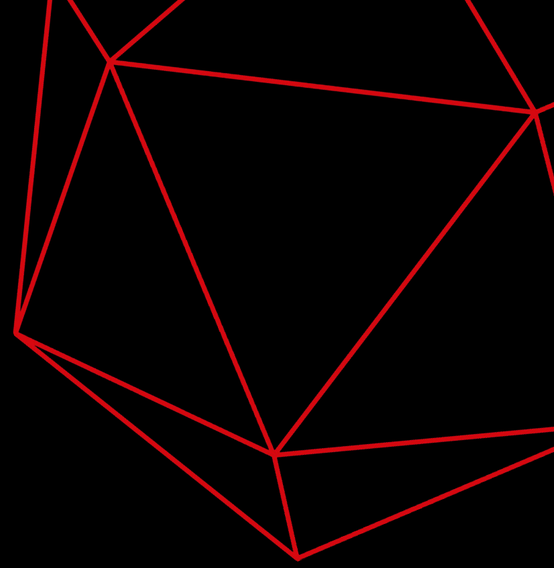
09 E2E Capstone

ANALYTICS SUMMIT

01

The BigQuery Interface



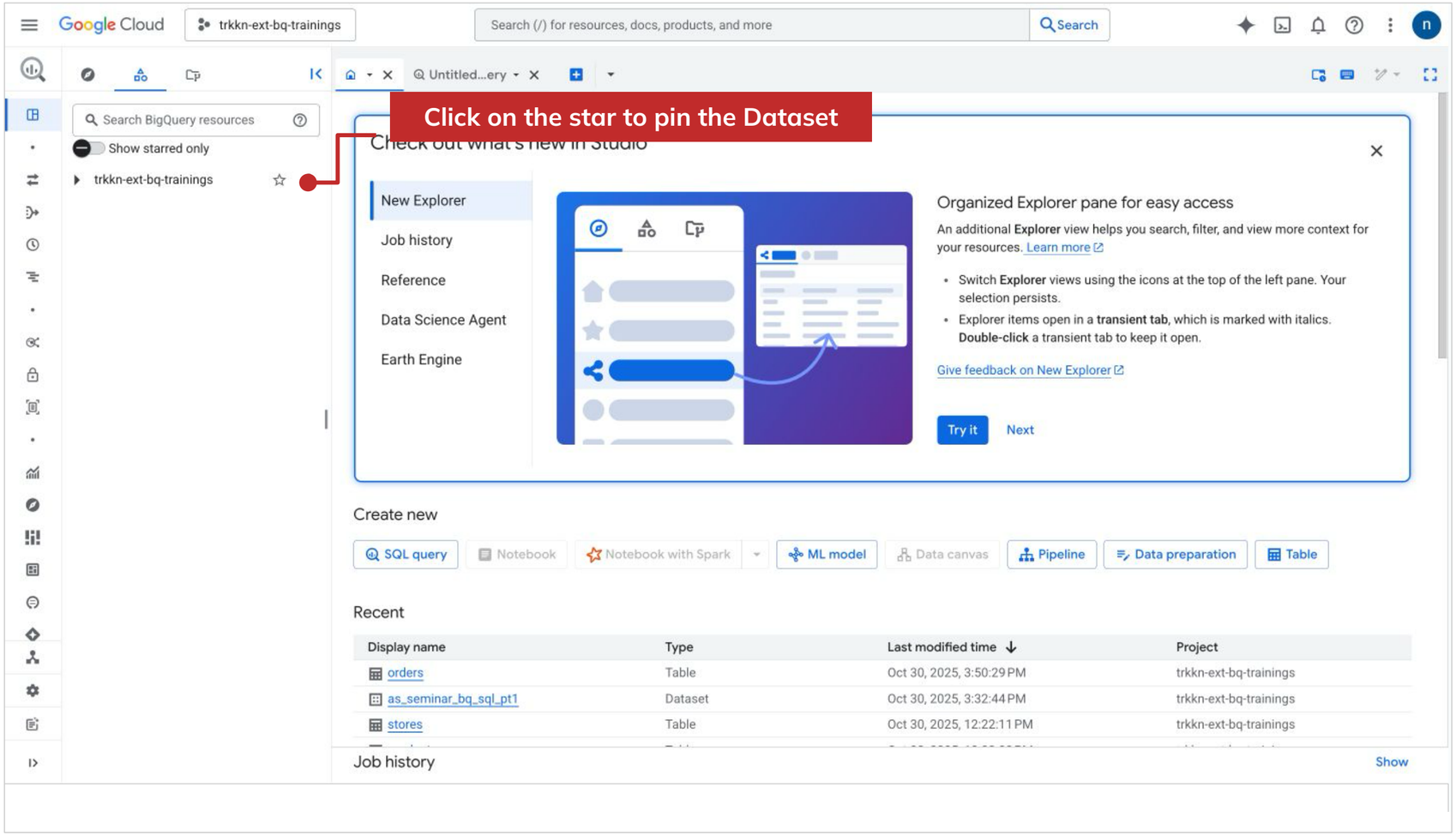


ANALYTICS SUMMIT

BigQuery Walkthrough

Intro — A Closer look — Application — Exercise

BQ Home



Please access
console.cloud.google.com/bigquery?project=trkkn-ext-bq-trainings

BigQuery Walkthrough

Intro | A Closer look | Application | Exercise

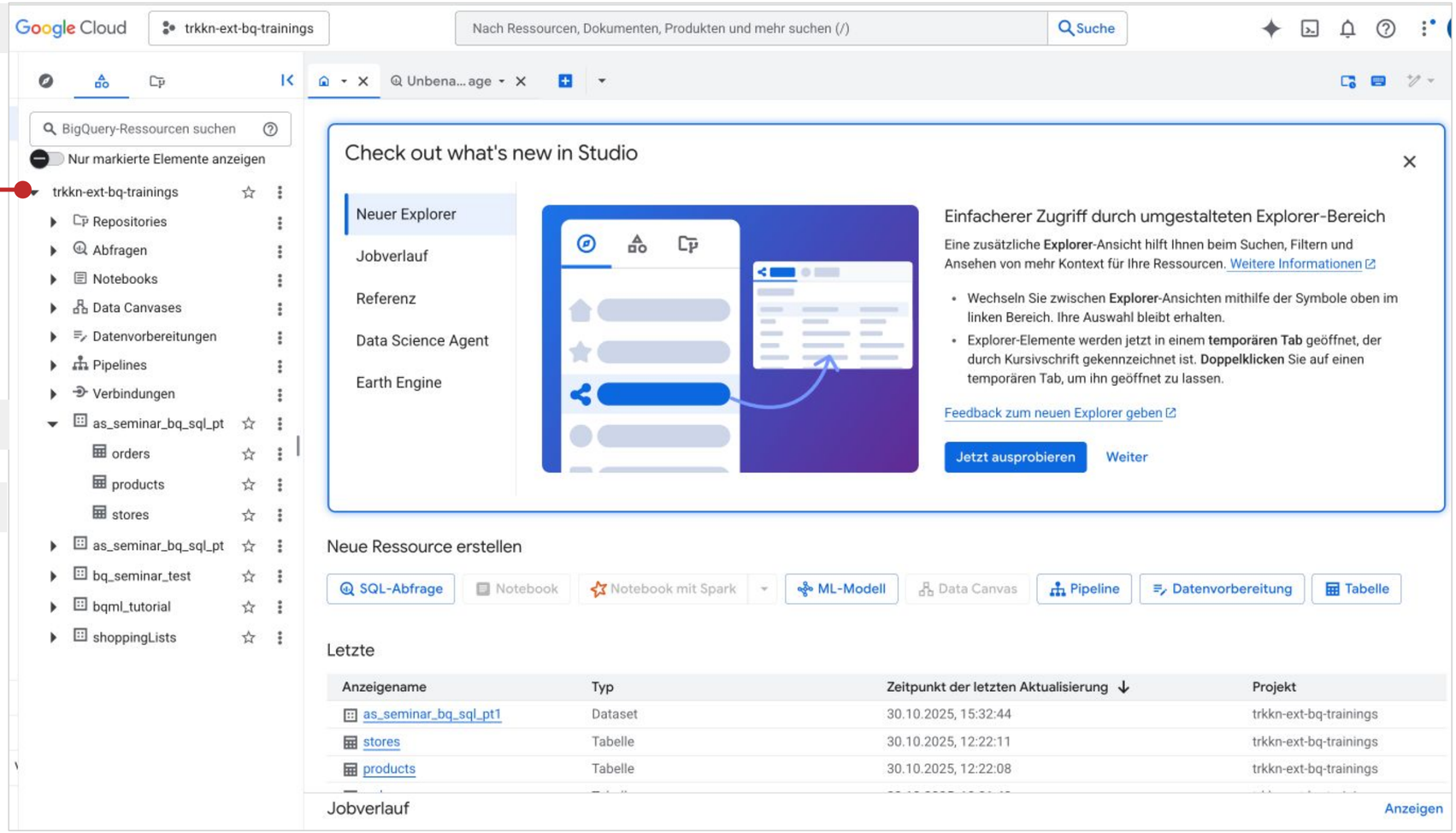
BQ Hierarchy

project

Click here to expand

dataset

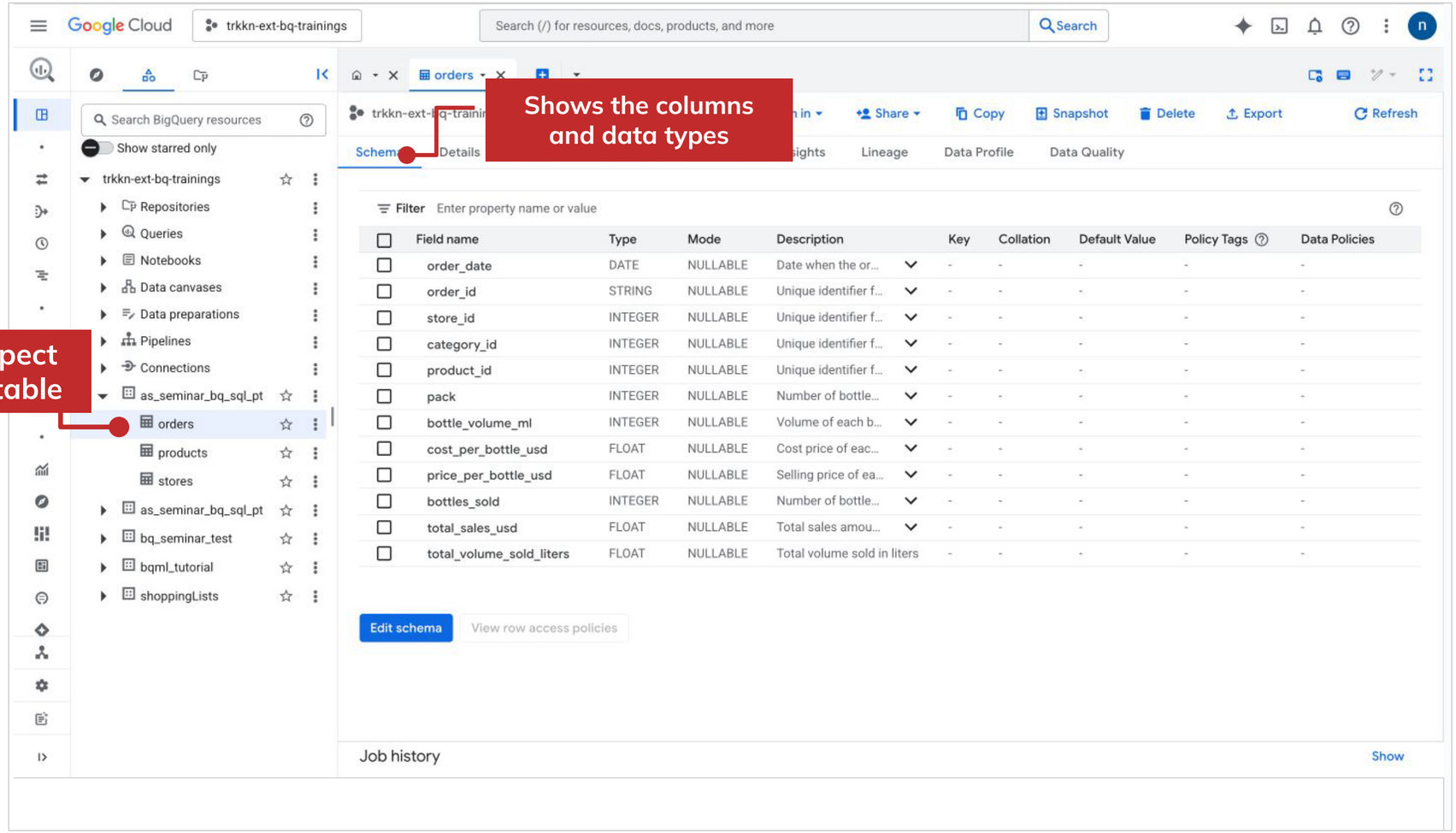
tables



BigQuery Walkthrough

Intro | A Closer look | Application | Exercise

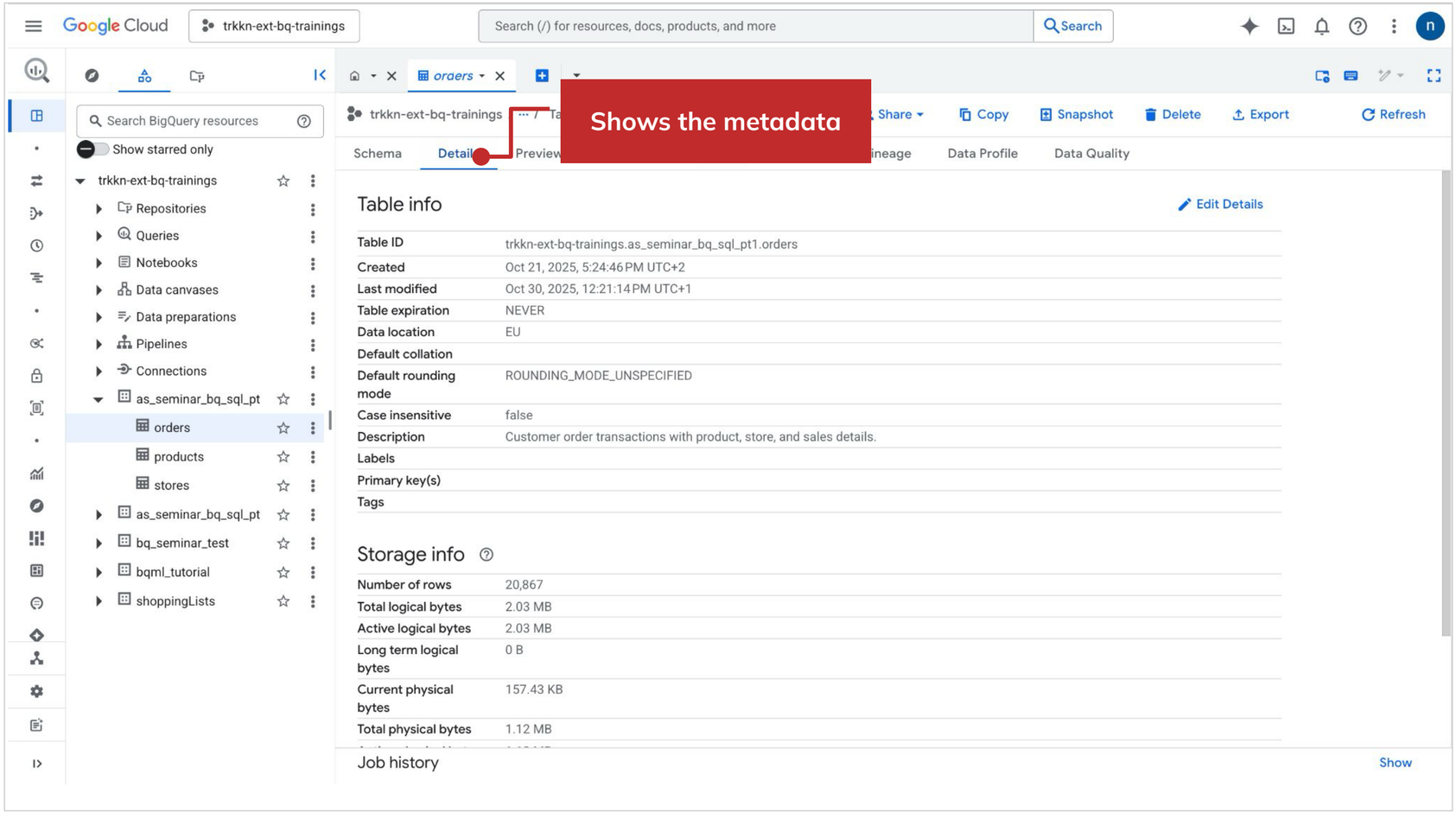
Inspecting Datasets & Tables



BigQuery Walkthrough

Intro | A Closer look | Application | Exercise

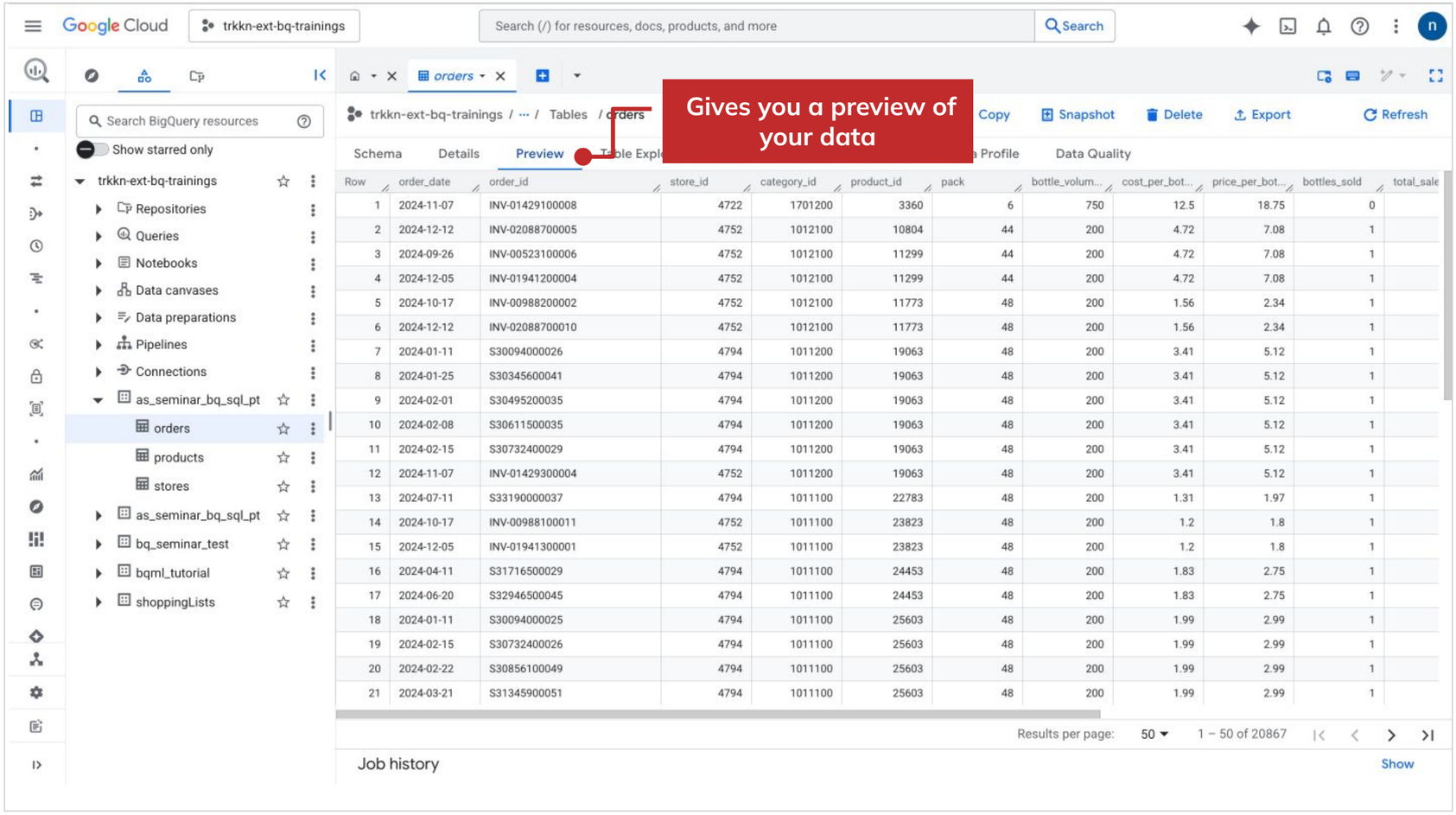
Inspecting Metadata e.g. number of rows



BigQuery Walkthrough

Intro | A Closer look | Application | Exercise

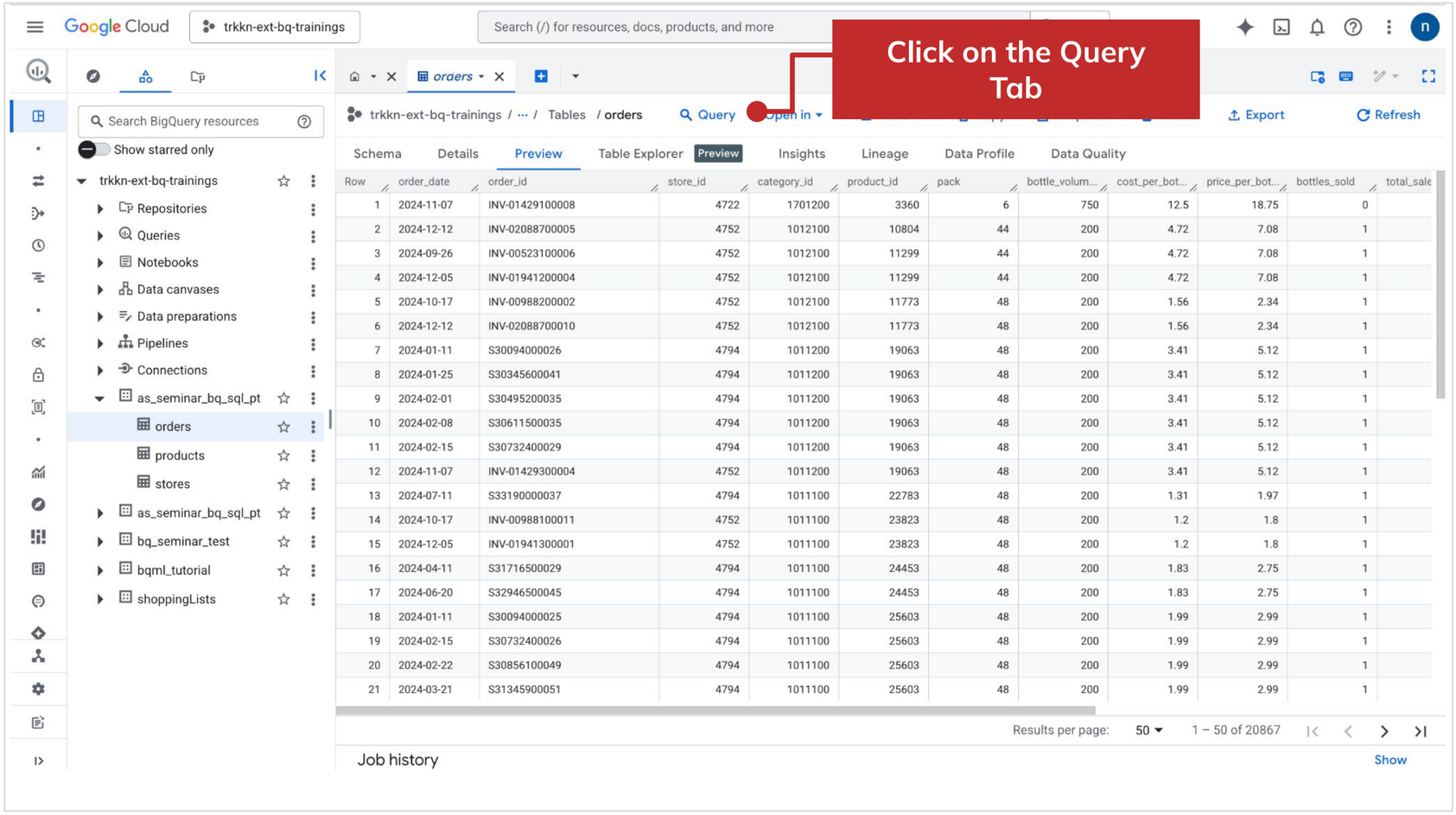
Preview data without querying



BigQuery Walkthrough

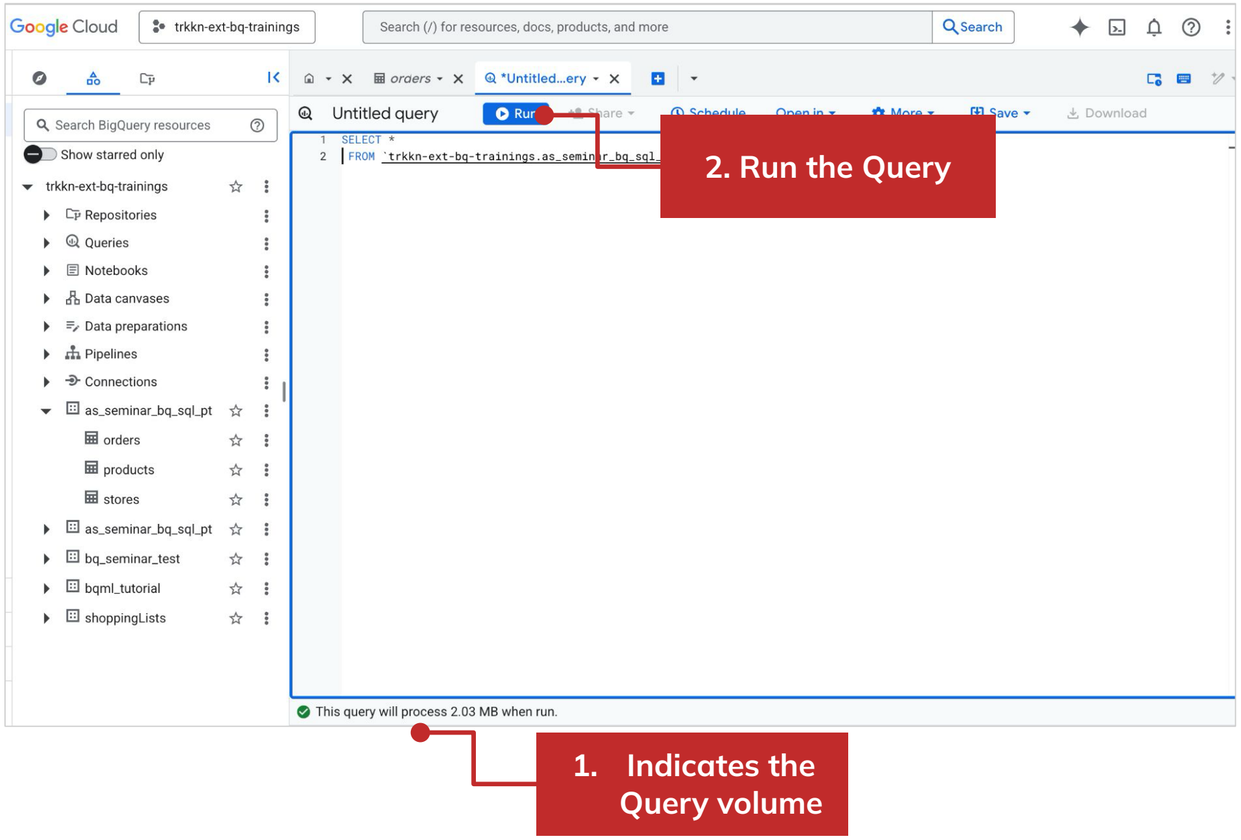
Intro | A Closer look | Application | Exercise

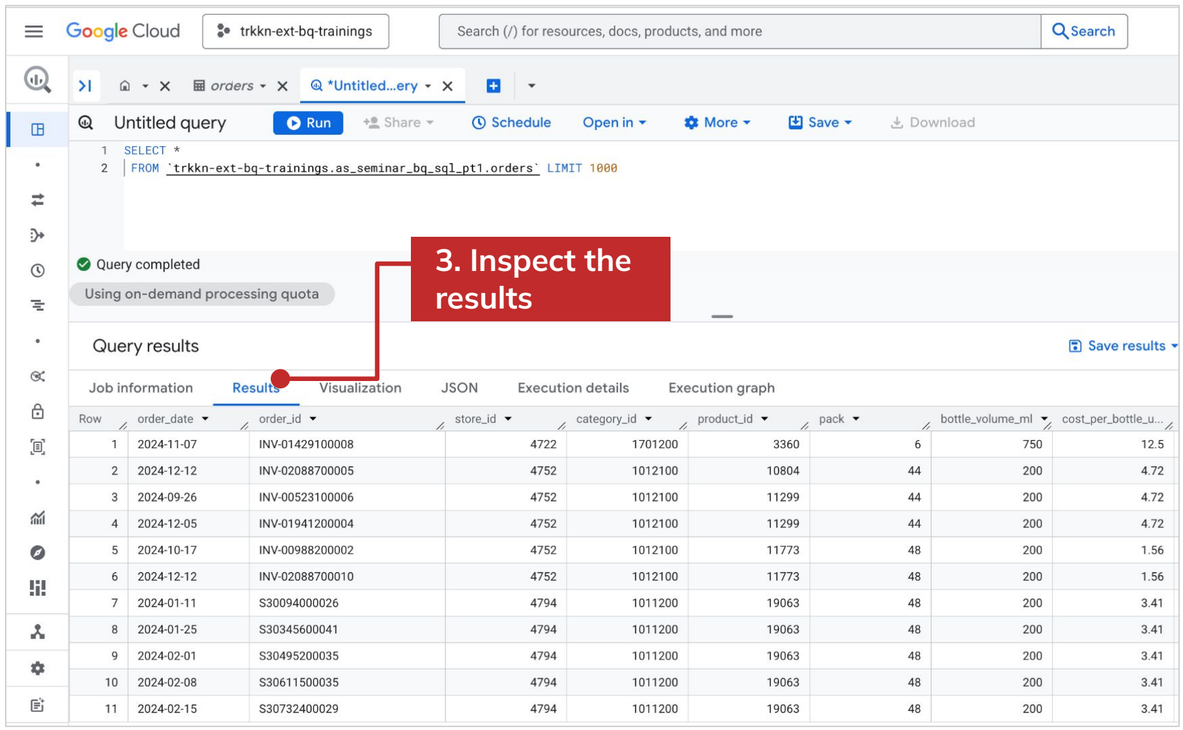
Let's run our first query



BigQuery Walkthrough

Let's run our first query

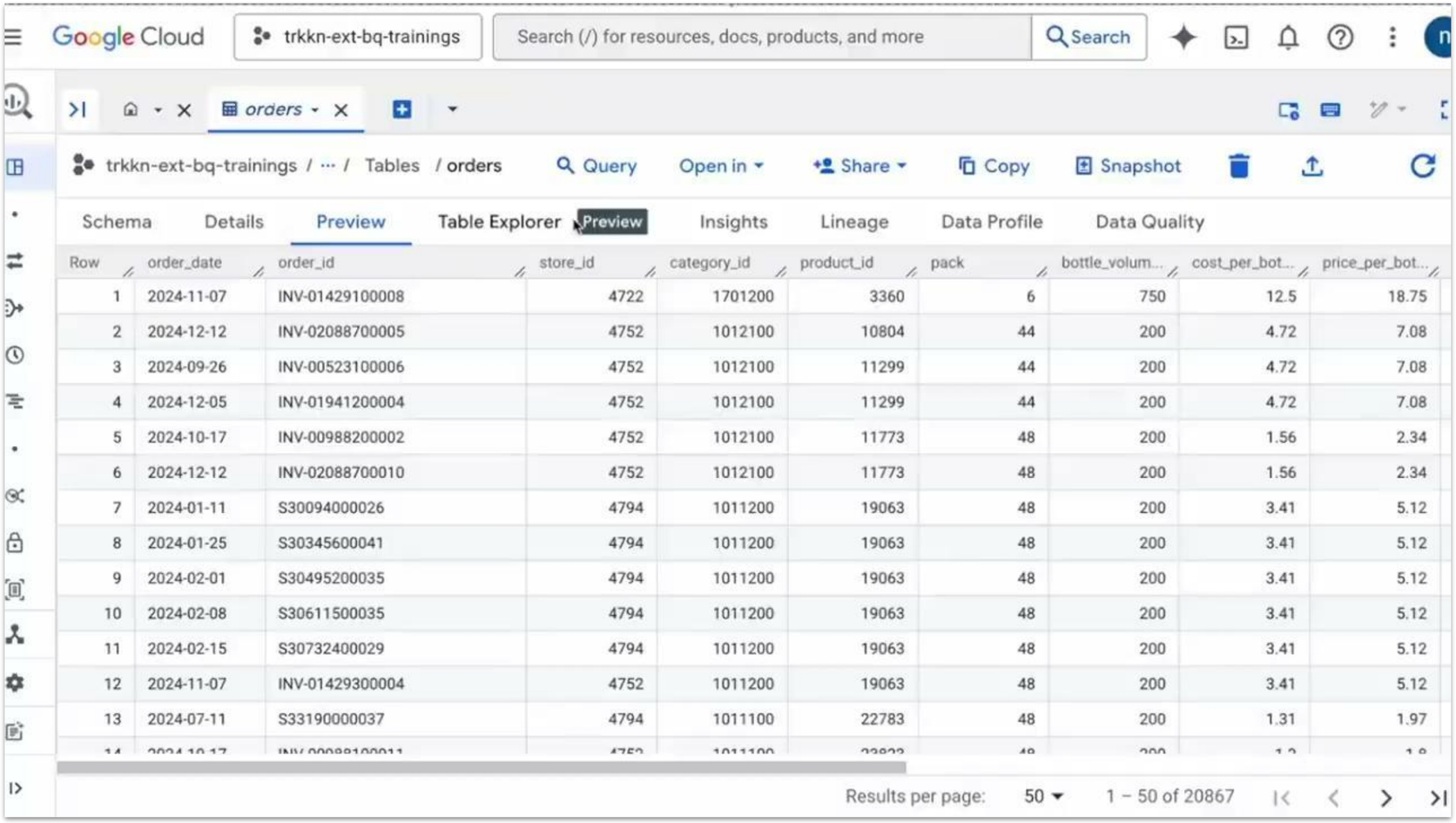




Row	order_date	order_id	store_id	category_id	product_id	pack	bottle_volume_ml	cost_per_bottle_u...
1	2024-11-07	INV-01429100008	4722	1701200	3360	6	750	12.5
2	2024-12-12	INV-02088700005	4752	1012100	10804	44	200	4.72
3	2024-09-26	INV-00523100006	4752	1012100	11299	44	200	4.72
4	2024-12-05	INV-01941200004	4752	1012100	11299	44	200	4.72
5	2024-10-17	INV-00988200002	4752	1012100	11773	48	200	1.56
6	2024-12-12	INV-02088700010	4752	1012100	11773	48	200	1.56
7	2024-01-11	S30094000026	4794	1011200	19063	48	200	3.41
8	2024-01-25	S30345600041	4794	1011200	19063	48	200	3.41
9	2024-02-01	S30495200035	4794	1011200	19063	48	200	3.41
10	2024-02-08	S30611500035	4794	1011200	19063	48	200	3.41
11	2024-02-15	S30732400029	4794	1011200	19063	48	200	3.41

BigQuery Walkthrough

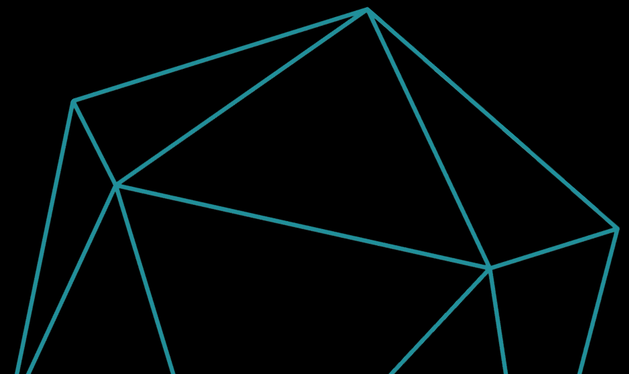
Let's run our first query



Row	order_date	order_id	store_id	category_id	product_id	pack	bottle_volum...	cost_per_bot...	price_per_bot...
1	2024-11-07	INV-01429100008	4722	1701200	3360	6	750	12.5	18.75
2	2024-12-12	INV-02088700005	4752	1012100	10804	44	200	4.72	7.08
3	2024-09-26	INV-00523100006	4752	1012100	11299	44	200	4.72	7.08
4	2024-12-05	INV-01941200004	4752	1012100	11299	44	200	4.72	7.08
5	2024-10-17	INV-00988200002	4752	1012100	11773	48	200	1.56	2.34
6	2024-12-12	INV-02088700010	4752	1012100	11773	48	200	1.56	2.34
7	2024-01-11	S30094000026	4794	1011200	19063	48	200	3.41	5.12
8	2024-01-25	S30345600041	4794	1011200	19063	48	200	3.41	5.12
9	2024-02-01	S30495200035	4794	1011200	19063	48	200	3.41	5.12
10	2024-02-08	S30611500035	4794	1011200	19063	48	200	3.41	5.12
11	2024-02-15	S30732400029	4794	1011200	19063	48	200	3.41	5.12
12	2024-11-07	INV-01429300004	4752	1011200	19063	48	200	3.41	5.12
13	2024-07-11	S33190000037	4794	1011100	22783	48	200	1.31	1.97

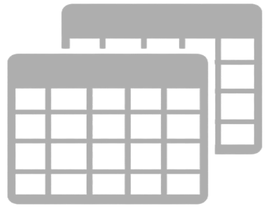
02

Loading Data into BigQuery

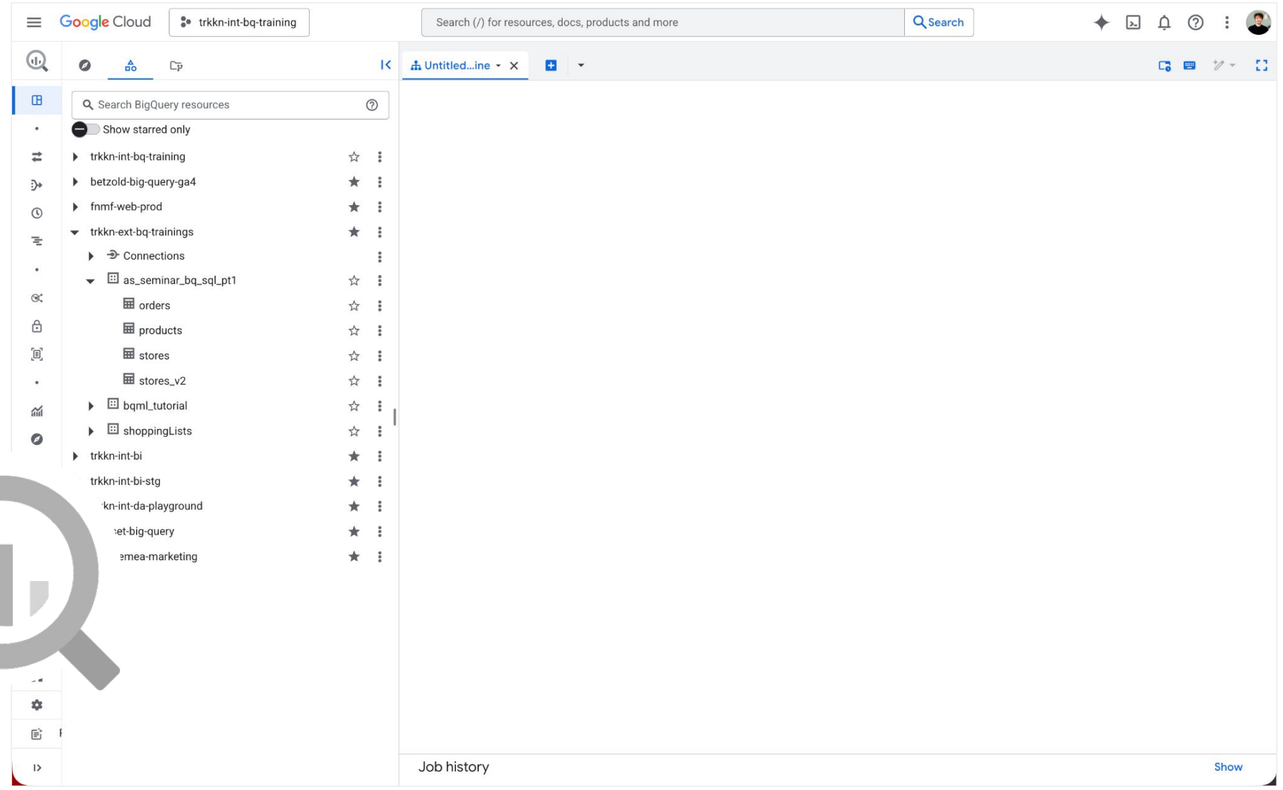


ANALYTICS SUMMIT

Intro: **What we're trying to achieve**



Loading our table from our source of choice ...



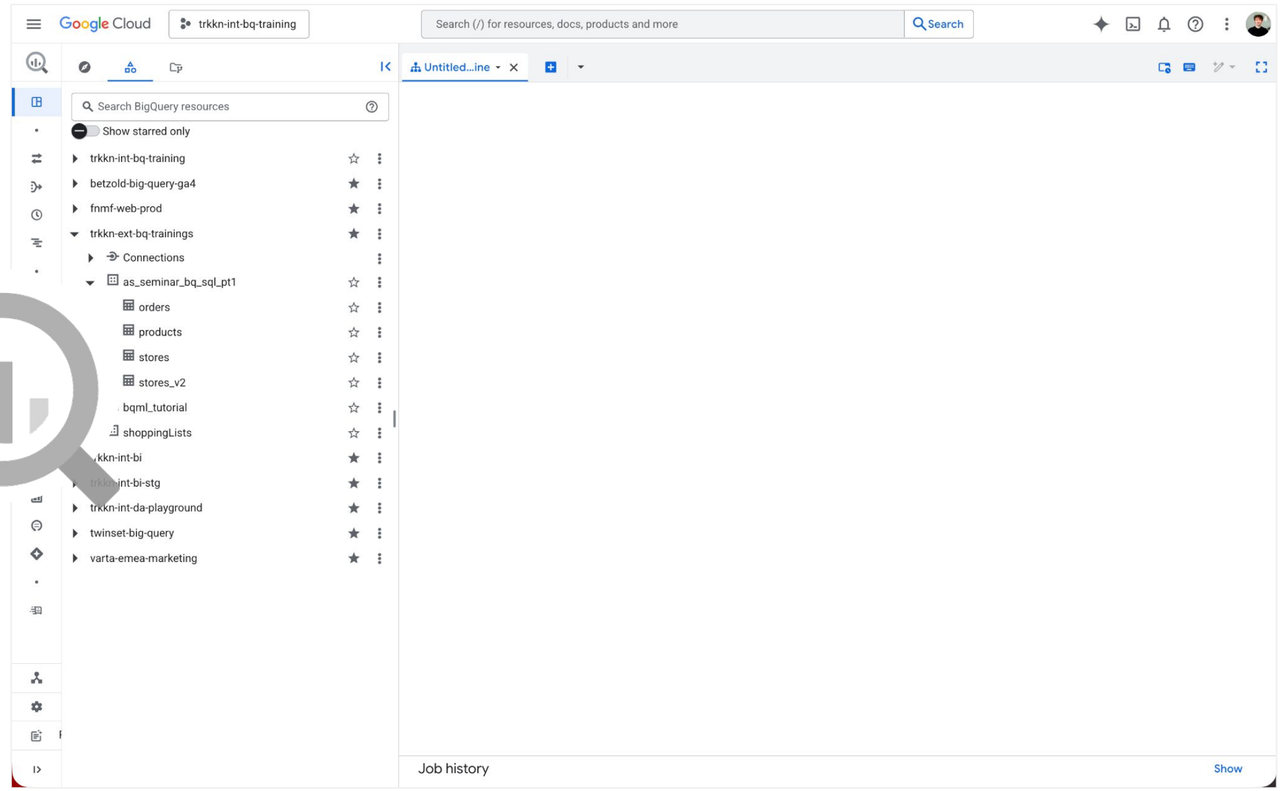
... into BigQuery

A Closer Look: How data can be loaded into BigQuery

Manual uploads
(e.g. CSVs)

Data transfers
(daily exports by platforms)

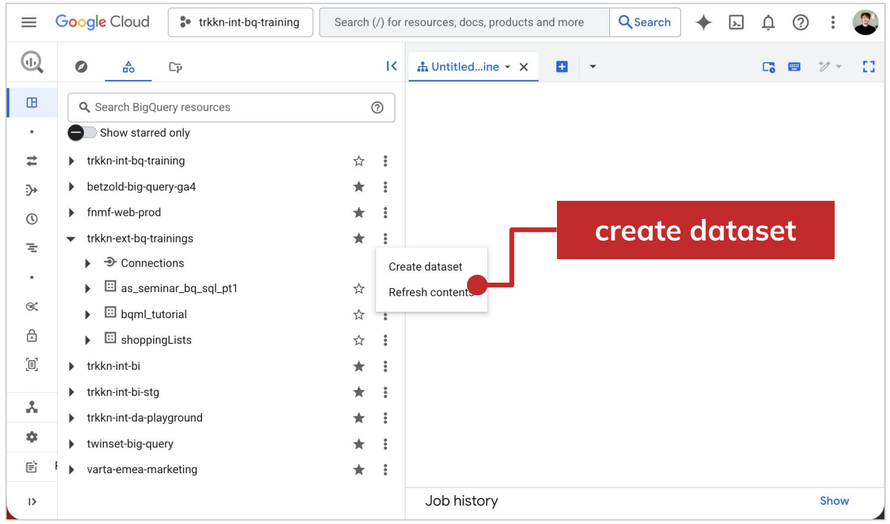
Custom extractions
(via platform APIs)

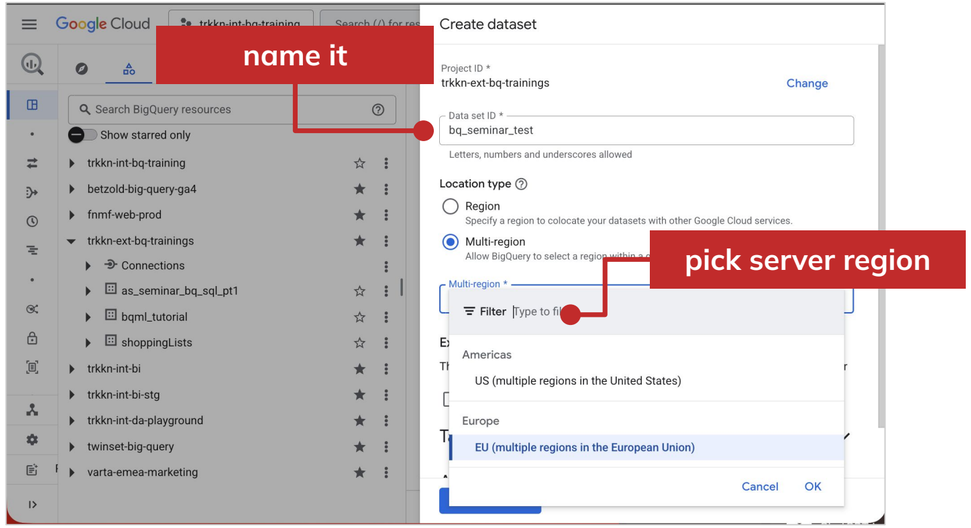


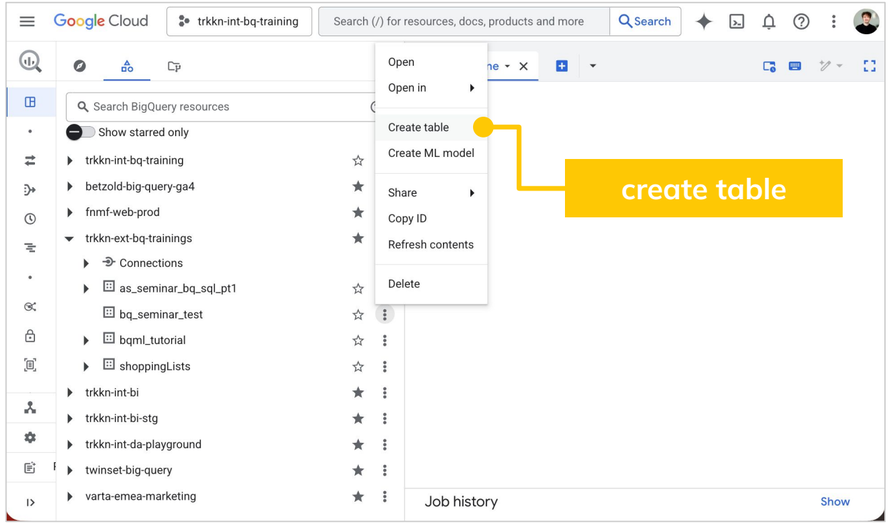
Hands-On: Importing a CSV into BigQuery

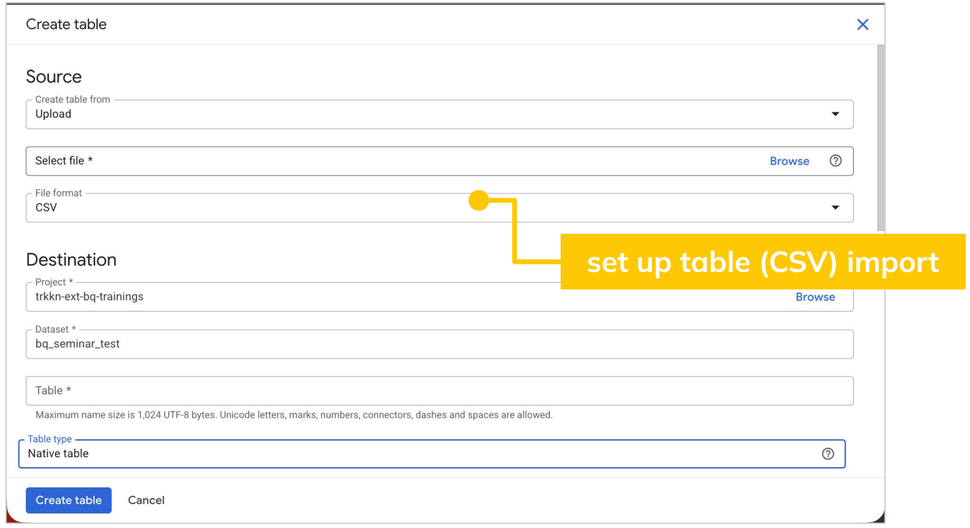
reminder:
BQ hierarchy

- project
  - dataset
    - table





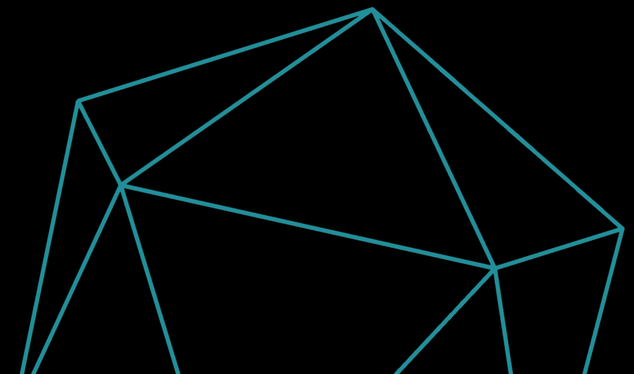




03

Basic SQL, Part 1

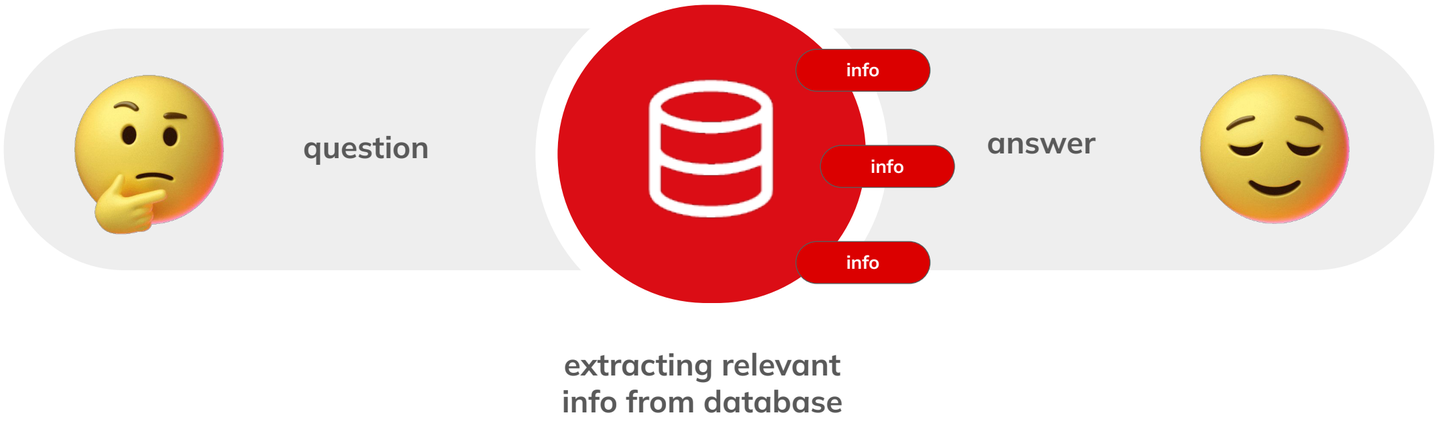
Querying Data



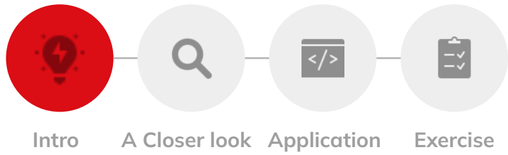
ANALYTICS SUMMIT

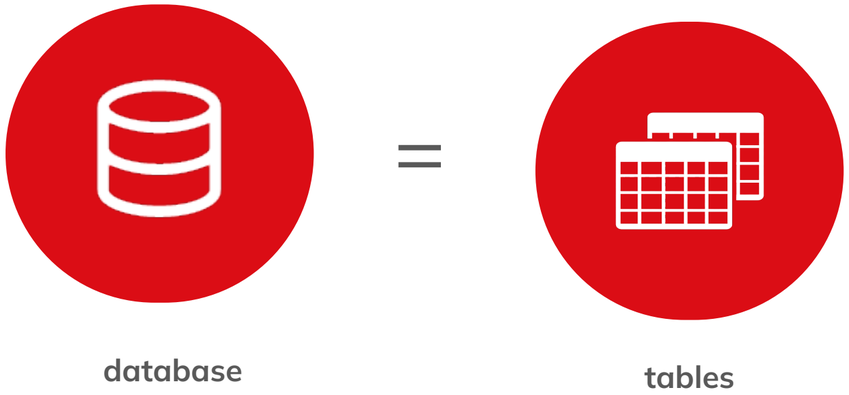
What’s SQL?

SQL, short for **S**tructured **Q**uery **L**anguage and sometimes called "sequel," is a programming language designed for working with databases.



SQL, Databases, Tables

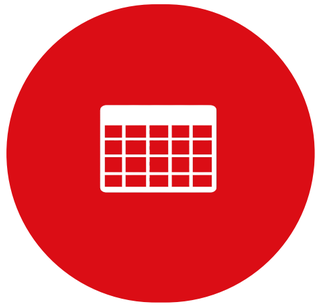




database = tables

SQL = Querying info from a table

The anatomy of a table



table

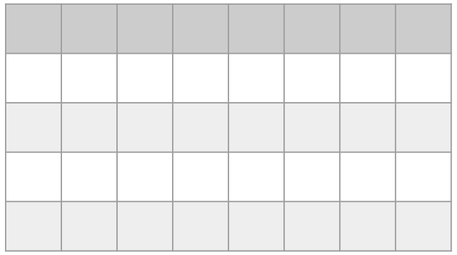
=

`seminar_attendees.csv`

signup_date	name	company	country	language	attending_both_seminars	number_of_seminars	price	email
2025-02-18	Julian Holzner	TRKKN	Germany	DE	FALSE	1	0	julian@trkkn.com
2025-02-18	Nassim Gharbi	TRKKN	Germany	DE	FALSE	1	0	nassim@trkkn.com
2025-04-06	Jack Müller	NextGen (UK) Ltd.	Switzerland	DE	TRUE	2	500	jack.mueller@nextgen.ch
2025-04-08	David Berger	Synergy Ops	Austria	DE	TRUE	2	500	david.berger@synergy.at

Different fields, different data types

`seminar_attendees.csv`



Numeric Types

`INTEGER` – Whole numbers without decimals (`price` = 500)

`FLOAT`– Approximate numbers with decimals (`price` = 499.99996)

`NUMERIC` – Exact numbers with decimals (`price` = 499.99995999123)

Logical Types

`BOOLEAN` – TRUE or FALSE values (`is_attending_both_seminars` = TRUE)

Text Types

`STRING` – Variable-length text supporting Unicode characters (`email` = “julian@trkkn.com”)

Temporal Types

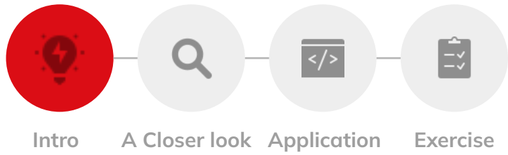
`DATE` – Calendar date without time (`signup_date` = “2025-08-31”)

`DATETIME` – Date with time components (`signup_date` = “2025-08-31 09:15:30”)

`TIMESTAMP` – Absolute point in time with microsecond precision (`signup_date` = “2025-08-31 09:15:30:00.123456 UTC”)

| ***TRKKN*** *Proprietary & Confidential*

What's SQL?



Databases don't know our data needs and require explicit instructions about what data to fetch. – SQL **SELECT** queries let us specify precisely what we need.

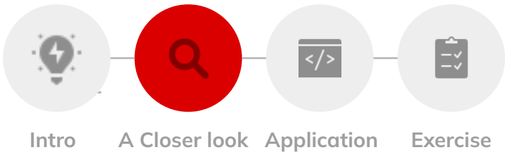
A basic **SELECT** query uses two essential keywords:
SELECT (what data are we fetching?) and **FROM** (which table are we fetching it from?).

```
SELECT
 name,
 company,
 seminar,
 seminar_signup_date
FROM
 trkkn-ext-bq-trainings.seminar_1.signups;
```

project | dataset | table

```
SELECT
 *
FROM
 trkkn-ext-bq-trainings.seminar_1.signups;
```

Filtering rows with a WHERE clause

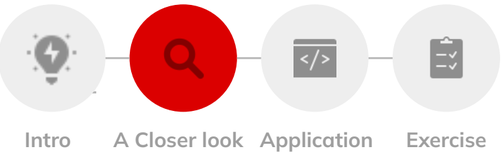


You can add a **`WHERE`** clause to your **`SELECT`** statement to filter the results and show only the rows that match the conditions you specify.

I want to see Trakken employees only

```
SELECT
 *
FROM
 `trkkn-ext-bq-trainings.seminar_1.signups`
WHERE
 company = "TRKKN";
```

Sophisticated Filtering with IN, OR, AND



Sometimes you need to filter data using multiple conditions or check against several values at once. SQL provides powerful tools for this.

I want to see attendees from either DE or AT

```
SELECT
 *
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`
WHERE
 country IN ("Germany","Austria") ;
```

the **IN** operator lets you match against a list of values

I want to see attendees who are either German or speak it,

```
SELECT
 *
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`
WHERE
 country = "Germany"
 OR language = "DE" ;
```

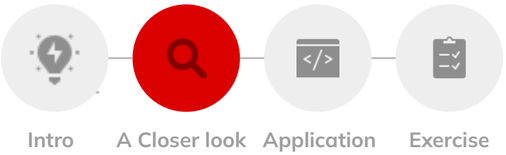
OR allows rows to meet any of several conditions

I want to see attendees who are not from Trakken and attend both seminars

```
SELECT
 *
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`
WHERE
 company != "TRKKN"
 AND attending_both_seminars = TRUE ;
```

AND requires all conditions to be true

Getting unique values with DISTINCT



I want to see all the countries the attendees are from.

```
SELECT
 country
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`;
```

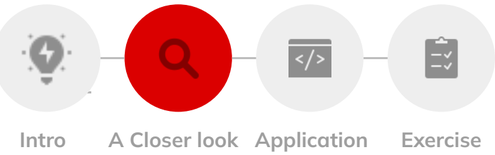
When you select a column without any special keywords, SQL returns every single row – including duplicates – for example: *Germany, Germany, Germany, UK, Germany, Austria, UK.*

```
SELECT
 DISTINCT country
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`;
```

The `DISTINCT` keyword eliminates these duplicates, returning only unique values. So the same query with `DISTINCT` would give you just: *Germany, UK, Austria*.

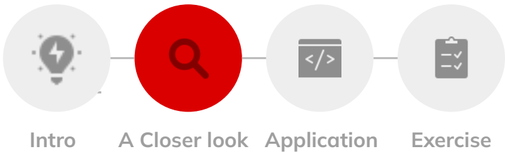
Link to BQ project & table

DISTINCT also works for unique combinations




```
SELECT
 DISTINCT country, language
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`;
```

Sorting data with ORDER BY



I want to see all the attendees in an alphabetical order.

```
SELECT
 *
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`
ORDER BY name ASC;
```

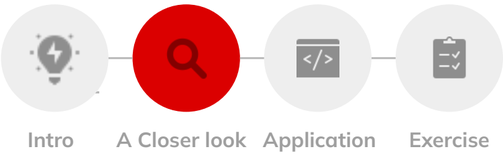
I want to see all the attendees with most seminars first

```
SELECT
 *
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`
ORDER BY number_of_seminars DESC;
```

The `ORDER BY` clause sorts your query results based on one or more columns, letting you arrange data in ascending (`ASC`, default) or descending (`DESC`) order to make it easier to analyse and interpret. It works alphabetically as well as numerically.

| *TRKKN Proprietary & Confidential*

Counting rows with COUNT



The `COUNT(*)` function returns the total number of rows in your query results, making it useful for quickly determining how many records meet your filter conditions or exist in a table.

I'd like to know the total number of attendees for the BQ seminar

```
SELECT
 COUNT(*)
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`
;
```

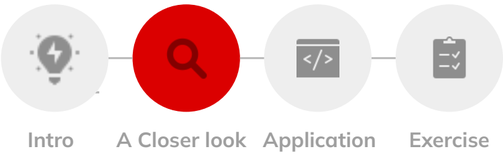
I'd like to know the total number of countries the attendees are from

```
SELECT
 COUNT(DISTINCT country)
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`
;
```

[Link to BQ project & table]

| TRKKN Proprietary & Confidential

Further aggregation functions



SQL provides aggregate functions to perform calculations across multiple rows: **`SUM`** adds up numeric values, **`AVG`** calculates the average, **`MIN`** finds the smallest value, and **`MAX`** identifies the largest value in a column.

Total ticket revenue

```
SELECT
 SUM(ticket_price)
FROM
 `trkkn-ext-bq-trainings.seminar_1.signups`
;
```

Average ticket price paid

```
SELECT
 AVG(ticket_price)
FROM
 `trkkn-ext-bq-trainings.seminar_1.signups`
;
```

maximum ticket price paid

```
SELECT
 MIN(ticket_price)
FROM
 `trkkn-ext-bq-trainings.seminar_1.signups`
;
```

minimum ticket price paid

```
SELECT
 MAX(ticket_price)
FROM
 `trkkn-ext-bq-trainings.seminar_1.signups`
;
```

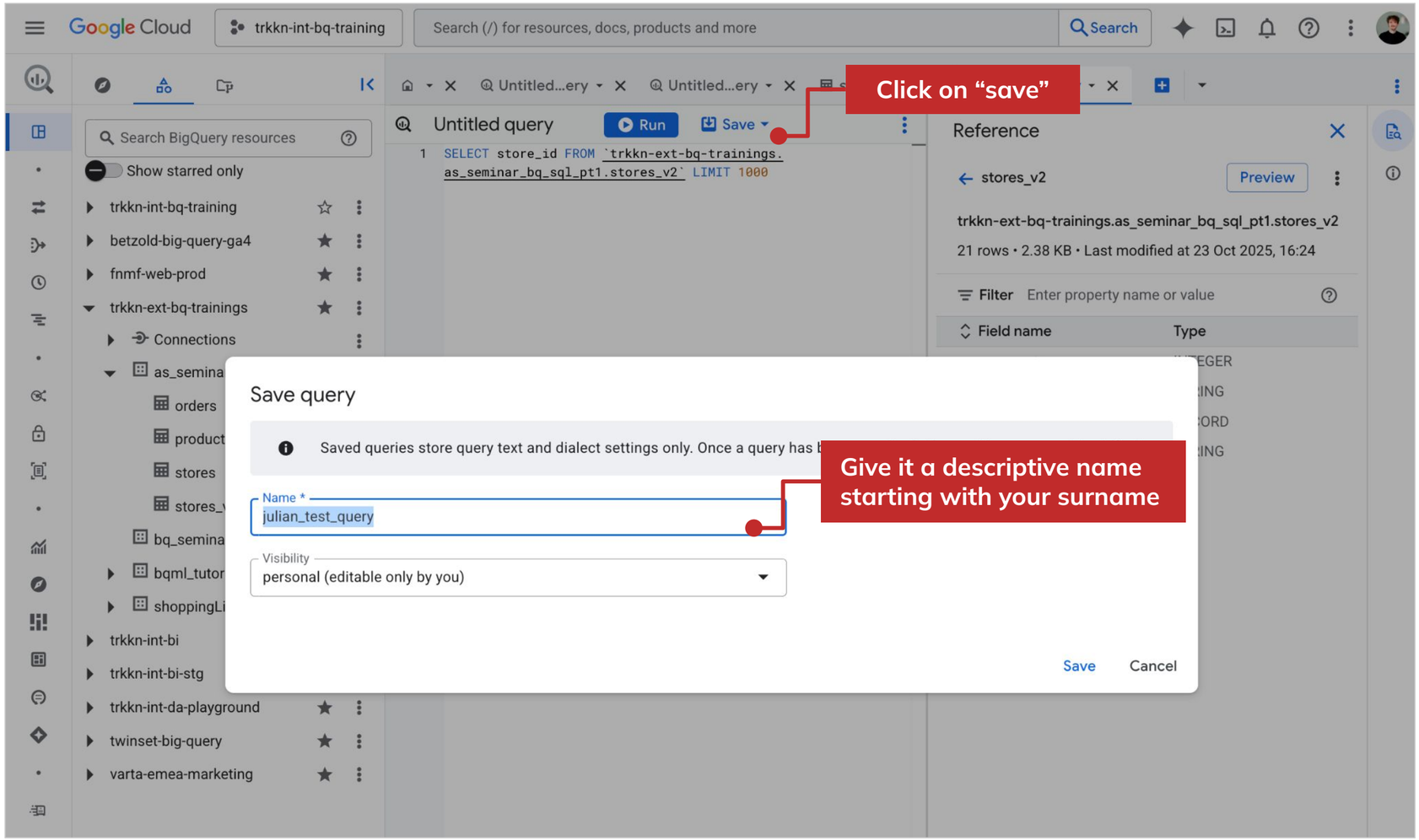
| ***TRKKN*** *Proprietary & Confidential*



Admin & Hands-On

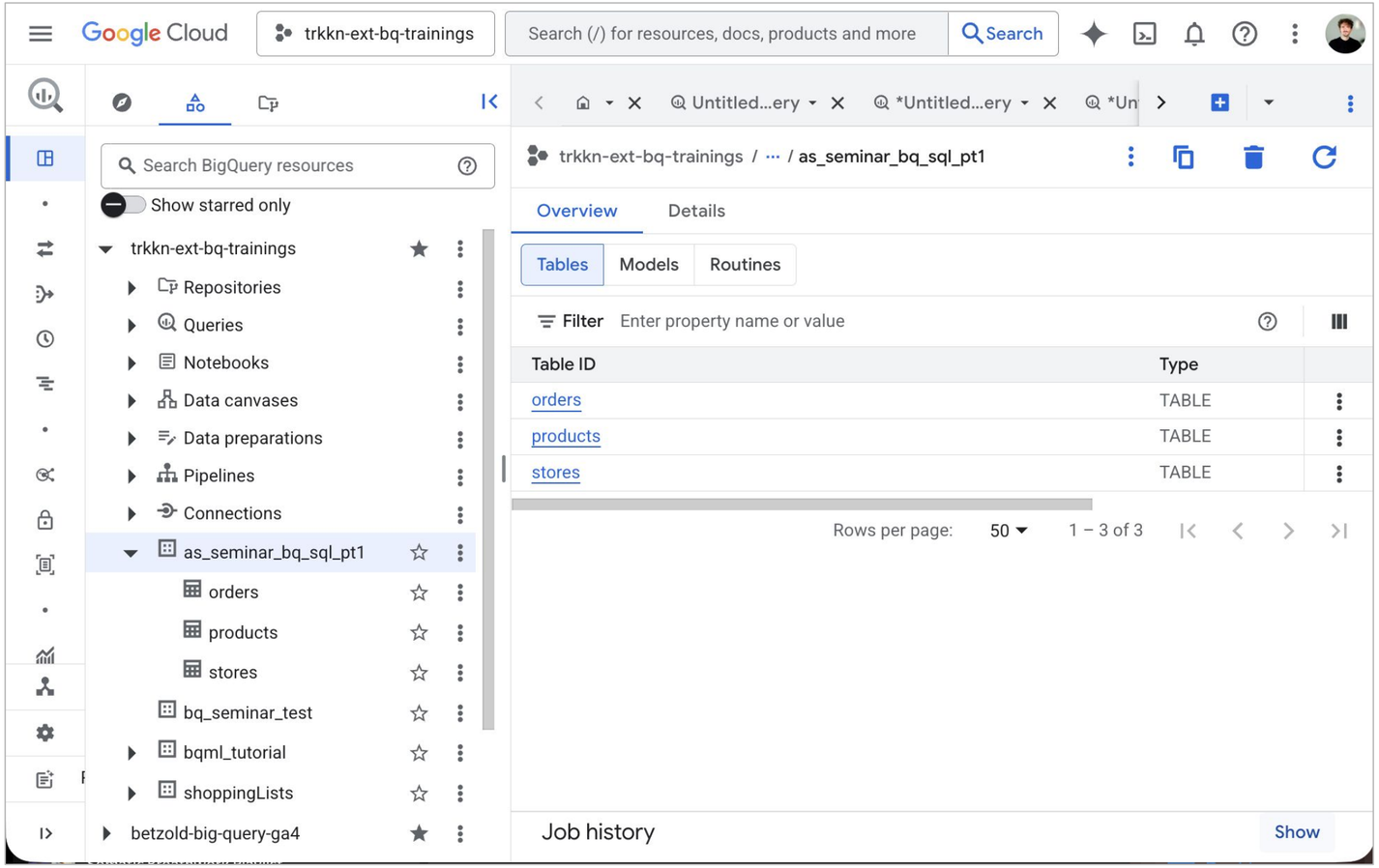
Admin: Saving your queries

Intro · A Closer look · Application · Exercise



Admin: Accessing the training data set

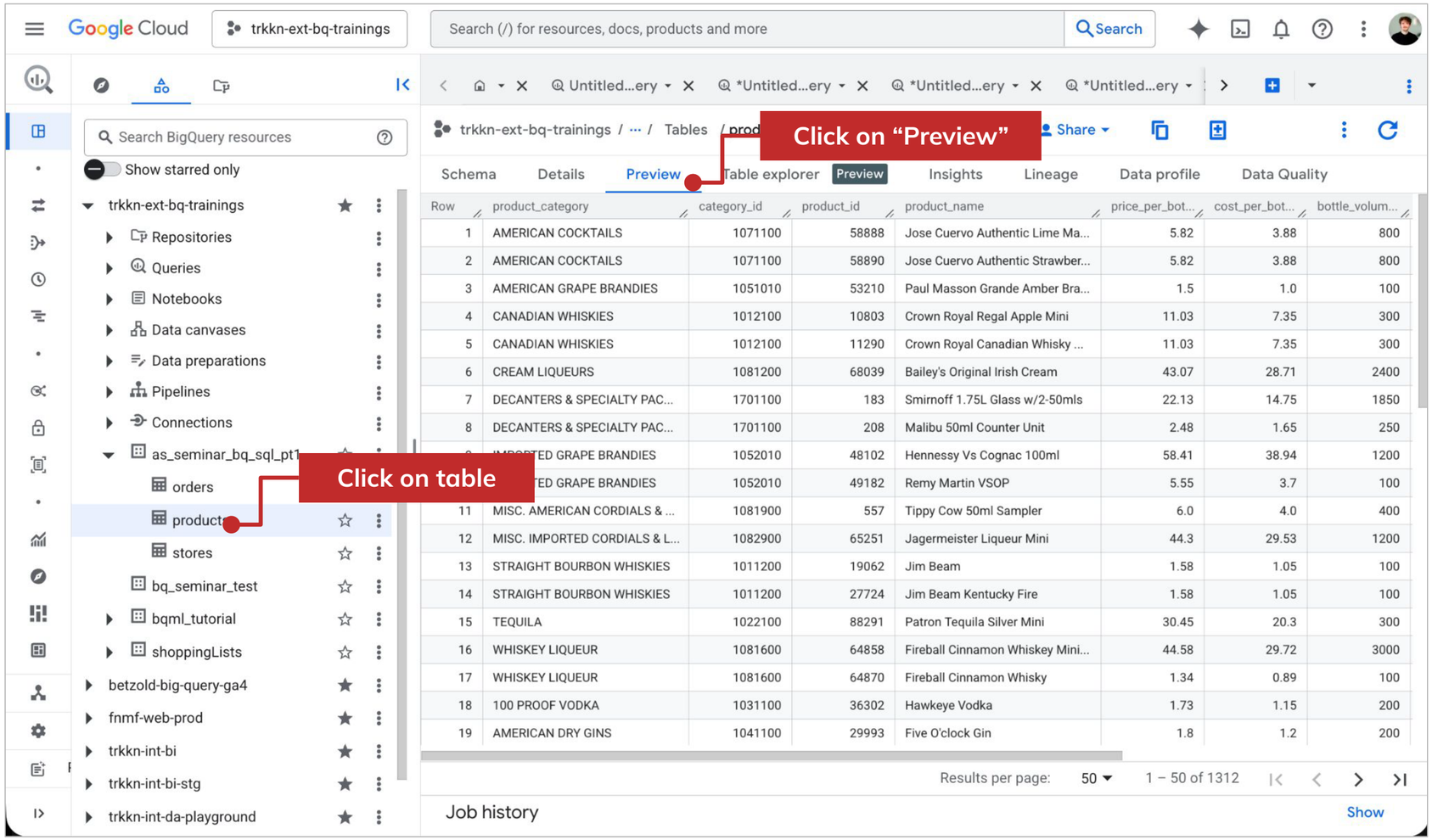
- trkkn-ext-bq-trainings
  - as_seminar_bq_sql_pt1
    - orders
    - products
    - stores




```
SELECT * FROM `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.products`
```

Hands-On: Discover the products table

Intro — A Closer look — Application — Exercise



Hands-On: Query the products table

Use the `products` table and answer the following questions by writing queries.

1. Get only beverages that are either `product_category` “TEQUILA” or “SPICED RUM”
2. How many beverages have a `bottle_volume_ml` higher than 2000?

| ***TRKKN*** *Proprietary & Confidential*

Hands-On: Query the products table

Get only beverages that are either `product_category` “TEQUILA” or “SPICED RUM”

```
SELECT
 *
FROM
 `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.products`
WHERE
 product_category IN (“TEQUILA”,”SPICED RUM”) ;
;
```

Query results

Save results | Open in

Job information | Results | Visualisation | JSON | Execution details | Execution graph

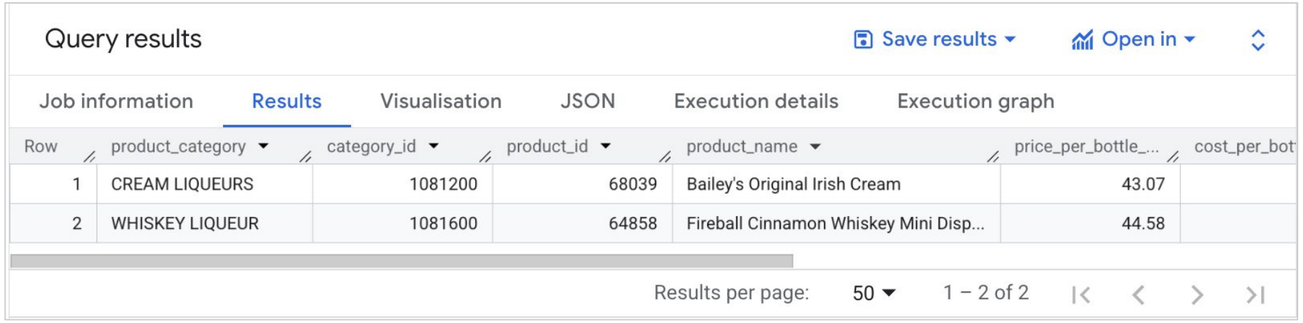
Row	product_category	category_id	product_id	product_name	price_per_bottle_...	cost_
1	TEQUILA	1022100	88291	Patron Tequila Silver Mini	30.45	
2	SPICED RUM	1062310	43333	Captain Morgan Spiced Rum Pet	3.75	
3	TEQUILA	1022100	87402	Jose Cuervo Especial Silver Teq...	4.5	
4	TEQUILA	1022100	88290	Patron Silver Tequila	12.75	

Results per page: 50 1 – 50 of 116

Hands-On: Query the products table

How many beverages have a `bottle_volume_ml` higher than 2000?

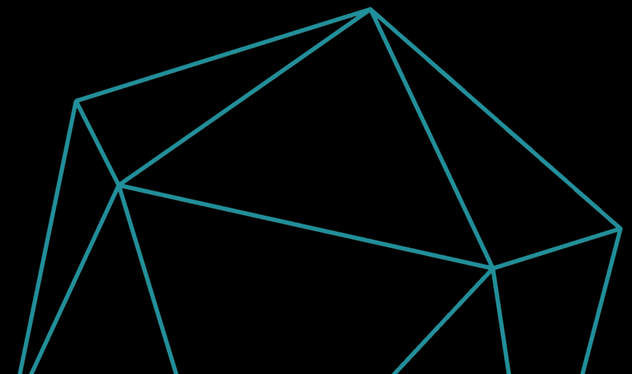
```
SELECT
 *
FROM
 `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.products`
WHERE
 bottle_volume_ml > 2000 ;
;
```



03

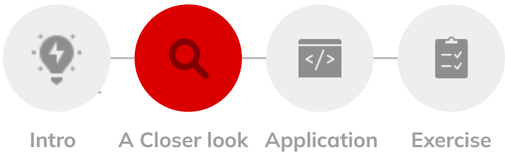
Basic SQL, Part 2

Aggregating and transforming data

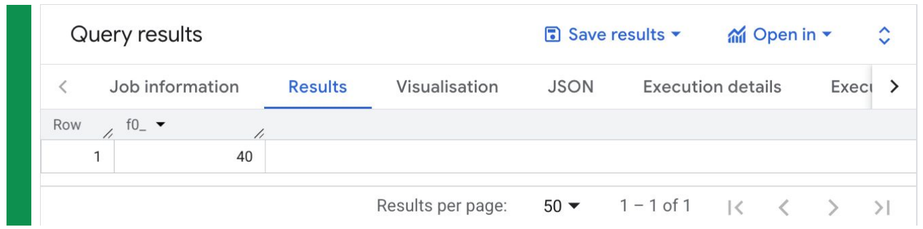


ANALYTICS SUMMIT

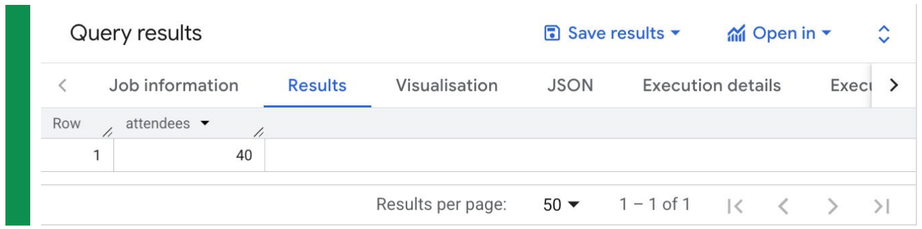
Recap: Counting rows with COUNT + Aliases




```
SELECT
 COUNT(*)
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`
;
```

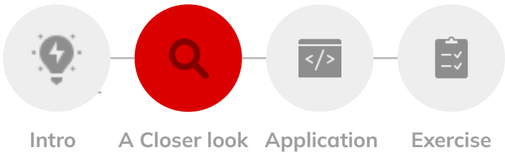



```
SELECT
 COUNT(*) AS attendees
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`
;
```



Aliases let you rename columns or tables in your query results using the `AS` keyword, making output more readable or giving shorter names to complex expressions.

Counting rows per group with COUNT(*) + GROUP BY

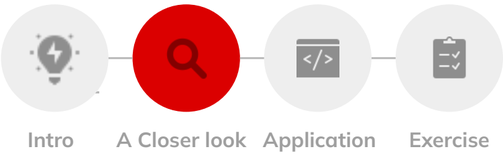


The `GROUP BY` clause organizes rows into groups based on shared column values, allowing you to count how many rows belong to each group using `COUNT`.

I'd like to know the number of attendees per country

```
SELECT
 country,
 COUNT(*) AS attendees
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`
GROUP BY
 country;
```

GROUP BY works with all aggregation functions




```
SELECT
 field1, count(*)
FROM
 table
GROUP BY
 field1;
```

```
SELECT
 field1, count(distinct field2)
FROM
 table
GROUP BY
 field1;
```

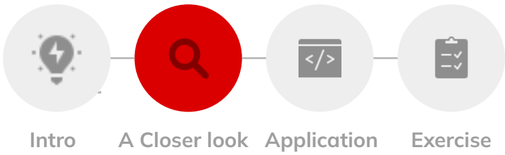
```
SELECT
 field1, min(value)
FROM
 table
GROUP BY
 field1;
```

```
SELECT
 field1, sum(value)
FROM
 table
GROUP BY
 field1;
```

```
SELECT
 field1, avg(value)
FROM
 table
GROUP BY
 field1;
```

```
SELECT
 field1, max(value)
FROM
 table
GROUP BY
 field1;
```

GROUP BY needs to list all non-aggregated columns



You can group data in three ways: `GROUP BY` with column names (e.g., `GROUP BY country`) groups rows by specific columns, `GROUP BY` with column numbers (e.g., `GROUP BY 1`) references columns by their position in the `SELECT` statement, and `GROUP BY ALL` automatically groups by all non-aggregated columns.

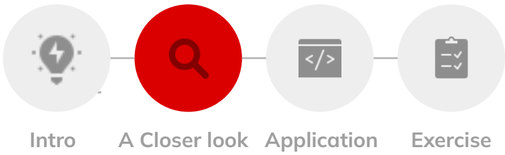
```
SELECT
 country,
 company,
 COUNT(*)
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`
GROUP BY
 country,
 company;
```

```
SELECT
 country,
 company,
 COUNT(*)
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`
GROUP BY
 1,2
```

```
SELECT
 country,
 company,
 COUNT(*)
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`
GROUP BY
 ALL
```

| *TRKKN Proprietary & Confidential*

Calculations on numeric columns



SQL lets you perform mathematical calculations directly on numeric columns using standard operators like + (addition), - (subtraction), * (multiplication), and / (division), allowing you to create new values or transform existing data in your queries.

```
SELECT
 column1 + column2
FROM
 table
;
```

```
SELECT
 column2 - column1
FROM
 table
;
```

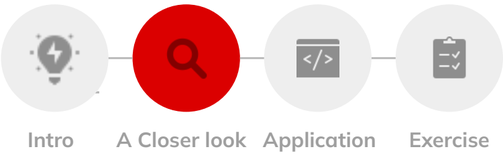
```
SELECT
 column1*column2
FROM
 table
;
```

```
SELECT
 column1/column2
FROM
 table
;
```

| *TRKKN Proprietary & Confidential*

attention: division by 0 throws an error → **workaround**: SAFE_DIVIDE gives NULL (missing value) , As usual in math: parenthesis before dot before dash

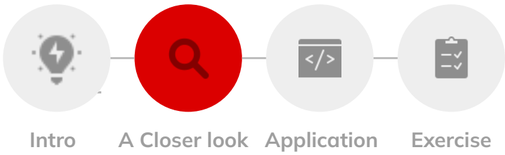
SELECT * EXCEPT



The `EXCEPT` operator returns all columns of the table except for one

```
SELECT
 * EXCEPT (field1)
FROM
 table;
```

Date Extraction

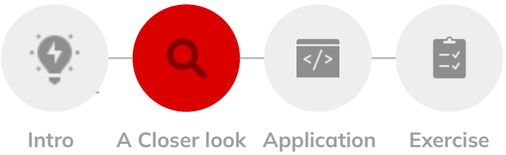


SQL allows you to extract specific parts of a date like year, month, or day using the `EXTRACT()` function.

I'd like to know the number of signups per month

```
SELECT
 EXTRACT(MONTH FROM seminar_signup_date) AS month,
 COUNT(*) AS signups
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`
GROUP BY
 month;
```

Comments in a Query



Comments let you add notes and explanations within your SQL queries: use `--` for single-line comments or `/* */` for multi-line comments to document your code without affecting query execution.

```
SELECT
/* This is a comment which can be
distributed
across
several
rows */
 *
FROM -- this is an in-line comment (only valid for this row)
 table # this is an in-line comment (only valid for this row)
;
```



Hands-On

Hands-On: Query the products table

Use the `products` table and answer the following questions by writing queries.

1. How many unique `product_ids` fall into either the "TEQUILA" or "SPICED RUM" `product_category`?
2. What's the average `price_per_bottle_usd` for the product_category "TEQUILA"?

| **TRKKN** *Proprietary & Confidential*

Hands-On: **Query the products table**

How many unique `product_ids` fall into either the "TEQUILA" or "SPICED RUM" `product_category`?

```
SELECT
 product_category,
 COUNT(DISTINCT product_id) AS beverages
FROM
 `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.products`
-- WHERE product_category IN ("TEQUILA","SPICED RUM")
GROUP BY product_category
```

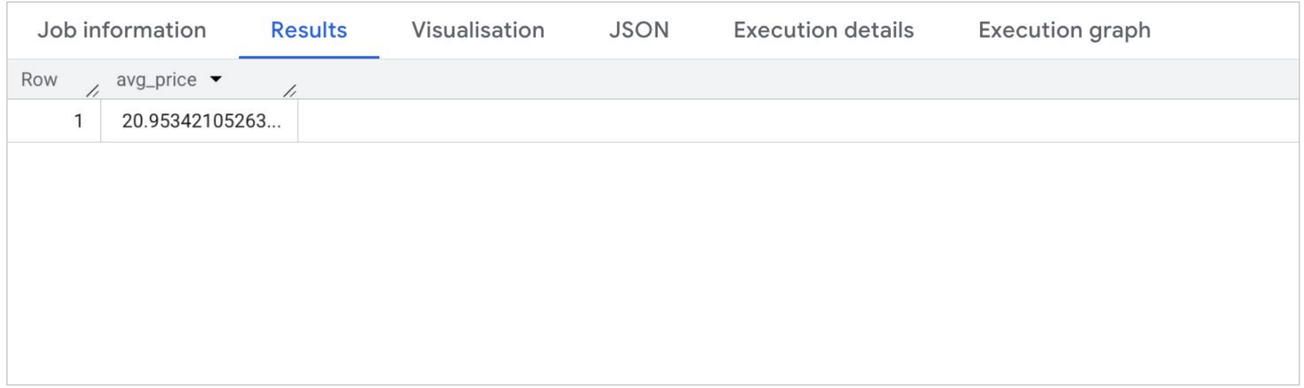
Job information | Results | Visualisation | JSON | Execution details | Execution graph

Row	product_category	beverages
1	TEQUILA	76
2	SPICED RUM	40

Hands-On: Query the products table

What's the average `price_per_bottle_usd` for the product_category "TEQUILA"?

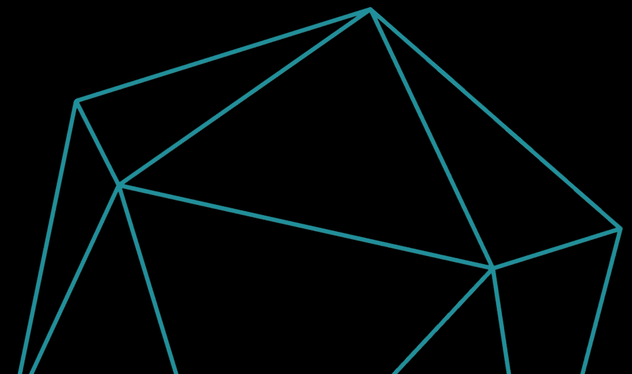
```
SELECT
 AVG(price_per_bottle_usd) AS avg_price
FROM
 `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.products`
WHERE
 product_category = "TEQUILA"
;
```



03

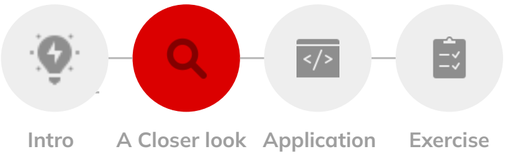
Basic SQL, Part 3

Conditional logics



ANALYTICS SUMMIT

Bucketing with CASE WHEN

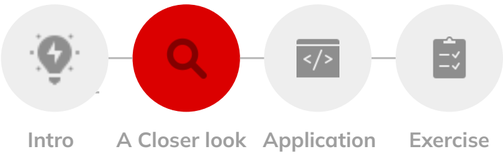


The `CASE WHEN` statement allows you to create conditional logic in SQL, letting you categorize or "bucket" rows into different groups based on specified conditions – similar to if-else statements in other programming languages.

```
SELECT
  name,
  seminar_signup_date,
  CASE
    WHEN seminar_signup_date < '2025-01-01' THEN "Early"
    WHEN seminar_signup_date < '2025-09-01' THEN "Regular"
    ELSE "Late"
  END AS seminar_signup_bucket
FROM
    `trkkn-ext-bq-trainings.seminar_1.signups`
;
```

I'd like to group the workshop attendees in early birds, regular and late sign-ups

Bucketing with CASE WHEN



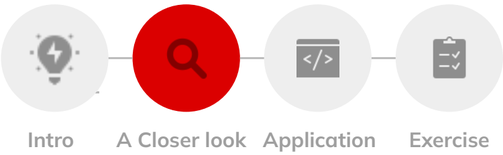

```
SELECT
 field1,
 CASE
       WHEN field1 in (value1,value2) THEN "bucket1"
       WHEN field1 = value3 THEN "bucket2"
       WHEN field1 = value4 OR field1 = value5 THEN "bucket3"
       ELSE "fallback"
 END as bucketing_column
FROM
 table;
```

```
SELECT
 field1,
 CASE
       WHEN field1 = "test" then 1
       ELSE 0
 END as is_test
FROM
 table;
```

- `CASE WHEN` works sequentially – the first match wins!
- You can use several columns in the `WHEN` clause
- Don’t mix up data types in the `THEN` clause – all values in the `THEN` clause need to be of the same type (e.g. string)

| *TRKKN Proprietary & Confidential*

IF and COALESCE



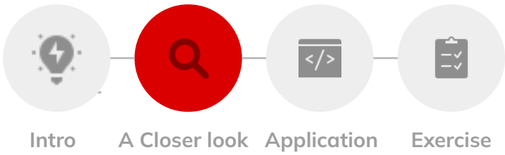
The **`IF`** function provides simple conditional logic with a true/false outcome: **`IF(condition, value_if_true, value_if_false)`**.

```
SELECT
 IF(company = "TRKKN","internal", "external") AS attendee_type
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`
```

The **`COALESCE`** function returns the first non-null value from a list of columns or expressions, making it useful for handling missing data.

```
SELECT
 COALESCE(phone_no, email_address, 'no contact') AS primary_contact_type
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`
```

Combine several values to a single one with CONCAT



The **`CONCAT`** function combines multiple text values or columns into a single string, allowing you to merge data like first and last names or create formatted output from separate fields.

```
SELECT
 CONCAT(first_name, " ", last_name AS full_name
FROM
`trkkn-ext-bq-trainings.seminar_1.signups`
```



Hands-On

Hands-On: Query the products table

Use the `products` table and answer the following question by writing a query: Create a new column called `bottle_type` that groups bottles based on the `bottle_volume_ml` field into “big” (>1500 ml),“regular” (500-1500 ml) and “small” (>500 ml)

Use the `stores` table and answer the following question by writing a query: Create a new column called `full_address` that combines `store_location.address` and `store_location.city` separated by a comma.

Hands-On: **Query the products table**

Create a new column called **`bottle_type`** that groups bottles based on the **`bottle_volume_ml`** field into “big” (>1500 ml),“regular” (500-1500 ml) and “small” (>500 ml).

```
SELECT
*,
 CASE
    WHEN bottle_volume_ml > 1500 THEN 'big'
    WHEN bottle_volume_ml BETWEEN 500 AND 1500 THEN 'regular'
    WHEN bottle_volume_ml < 500 THEN 'small'
  END AS bottle_type
FROM
 `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.products`
```

Job information | Results | Visualisation | JSON | Execution details | Execution graph

Row	product_category	category_id	product_id	product_name	price_per_bottle_...	cost_per_bottle_u...	bottle_vol
1	AMERICAN COCKTAILS	1071100	58888	Jose Cuervo Authentic Lime Ma...	5.82	3.88	
2	AMERICAN COCKTAILS	1071100	58890	Jose Cuervo Authentic Strawber...	5.82	3.88	
3	AMERICAN GRAPE BRANDIES	1051010	53210	Paul Masson Grande Amber Bra...	1.5	1.0	
4	CANADIAN WHISKIES	1012100	10803	Crown Royal Regal Apple Mini	11.03	7.35	
5	CANADIAN WHISKIES	1012100	11290	Crown Royal Canadian Whisky ...	11.03	7.35	

| ***TRKKN*** *Proprietary & Confidential*

Hands-On: **Query the products table**

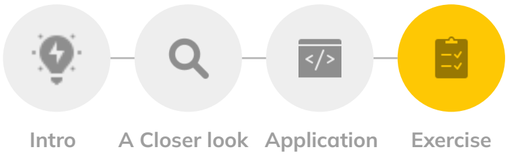
Create a new column called `full_address` that combines `store_location.address` and `store_location.city` separated by a comma.

```
SELECT
*,
  CONCAT(store_location.address, ', ', store_location.city) AS full_address
FROM
 `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.stores`
```

Job information | Results | Visualisation | JSON | Execution details | Execution graph

ti...	store_location. county	product_categories	full_address
501	Des Moines	Convenience	421 N MAIN ST, BURLINGTON
		Snacks	
		Beverages	
501	Des Moines	Convenience	2107 AGENCY ST, BURLINGTON
		Snacks	

Bringing it all together



When writing SQL queries, it's important to follow a specific order:

`SELECT` specifies the columns,
`FROM` identifies the table,
`WHERE` filters rows,
`GROUP BY` organizes data into groups,
`ORDER BY` sorts the results,
`LIMIT` restricts the number of rows returned

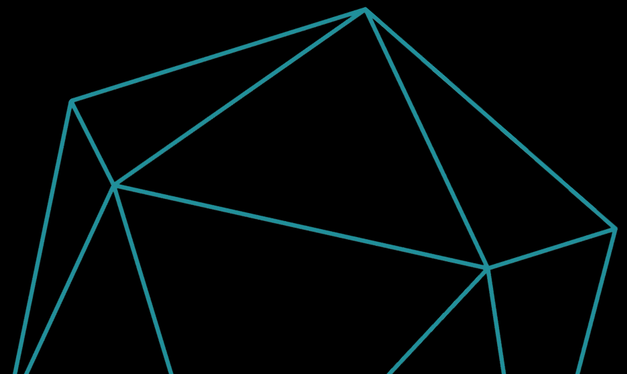
This sequence ensures your query executes correctly and returns the data you need.

```
SELECT
 field1,
 field2,
 count(*)
FROM
 table
WHERE
 field1 = value
GROUP BY
 field1,
 field2
ORDER BY
 field1 DESC
LIMIT 1000;
```

| **TRKKN** *Proprietary & Confidential*

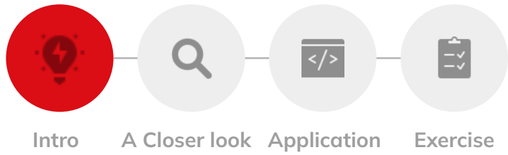
04

Mastering JOINs



ANALYTICS SUMMIT

What are JOINs ?



💡 A `JOIN` clause lets you combine rows from two or more tables, which allows you to analyze data in one place

💡 You match tables using `JOIN keys` — columns with the same identifiers in both tables (e.g. product_id, store_id).

Fields from the Table 1

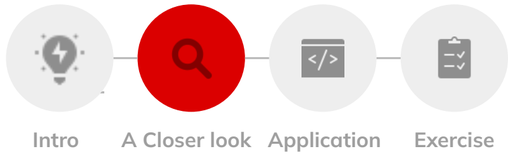
Fields from the Table 2

Join Type

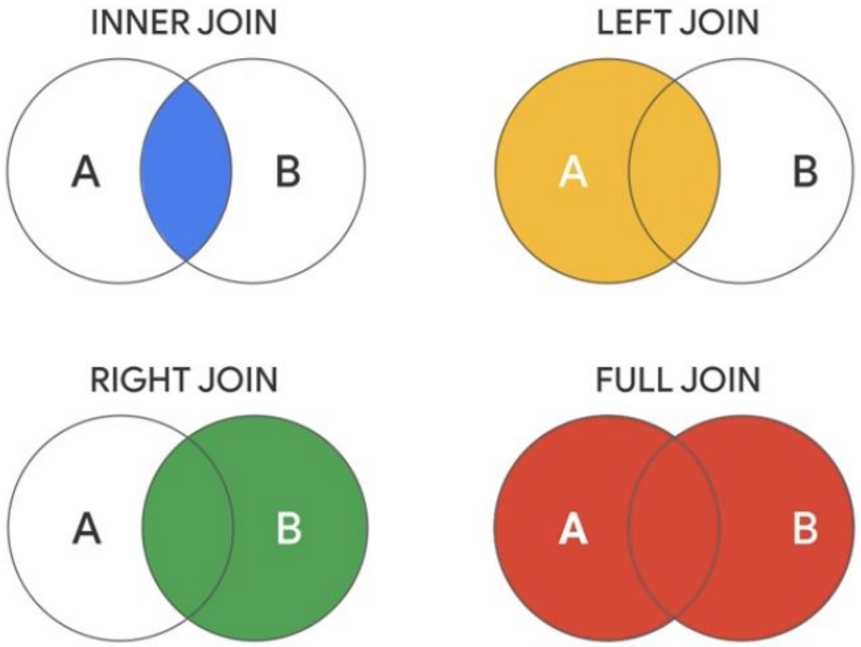
Join Keys

```
SELECT
 table1.field1,
 table2.field1,
 table1.field2,
 table2.field2
FROM
 table1
LEFT JOIN
 table2 ON table1.field1 = table2.field1;
```

What type of JOINs exists?



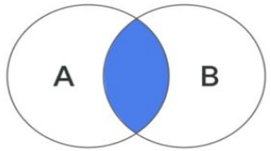
Think of a `JOIN` as a Venn Diagram, each Circle represents a Table and each `JOIN` Type specifies which rows to keep.



=

```
SELECT
*
FROM
 TABLE_A
INNER JOIN | LEFT JOIN | RIGHT JOIN | FULL JOIN
 TABLE_B ON TABLE_A.field1 = TABLE_B.field1;
```

Inner JOIN



A `INNER JOIN` only keeps rows where the values of the `JOIN keys` match in both tables.

Example

```
FROM A INNER JOIN B ON A.w = B.y

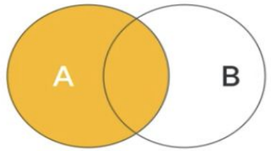
/*
Table A       Table B       Result
+-------+     +-------+     +---------------+
| w | x |  *  | y | z |  =  | w | x | y | z |
+-------+     +-------+     +---------------+
| 1 | a |     | 2 | k |     | 2 | b | 2 | k |
| 2 | b |     | 3 | m |     | 3 | c | 3 | m |
| 3 | c |     | 3 | n |     | 3 | c | 3 | n |
| 3 | d |     | 4 | p |     | 3 | d | 3 | m |
+-------+     +-------+     | 3 | d | 3 | n |
                            +---------------+

*/
```

🔹 A(1,2,3) 🔸 B(2,3,4) → ⚫ Result(2,3)

| ***TRKKN*** *Proprietary & Confidential*

Left JOIN



A `LEFT JOIN` returns all rows from the left table (A), and the matching rows from the right table (B). If no match is found on the right side, the missing columns are filled with `NULL`s.

Example

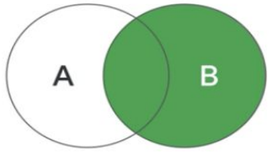
```
FROM A LEFT OUTER JOIN B ON A.w = B.y

/*
Table A       Table B       Result
+-------+     +-------+     +---------------------------+
| w | x |  *  | y | z |  =  | w    | x    | y    | z    |
+-------+     +-------+     +---------------------------+
| 1 | a |     | 2 | k |     | 1    | a    | NULL | NULL |
| 2 | b |     | 3 | m |     | 2    | b    | 2    | k    |
| 3 | c |     | 3 | n |     | 3    | c    | 3    | m    |
| 3 | d |     | 4 | p |     | 3    | c    | 3    | n    |
+-------+     +-------+     | 3    | d    | 3    | m    |
                            | 3    | d    | 3    | n    |
                            +---------------------------+

*/
```

🔹 A(1,2,3) 🔸 B(2,3,4) → ⚫ Result(1,2,3)

Right JOIN



A **`RIGHT JOIN`** returns all rows from the right table (B), and the matching rows from the left table (A). If no match is found on the left side, the missing columns are filled with **`NULL`**s.

Example

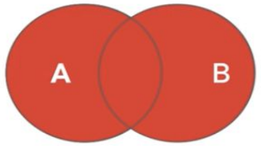
```
FROM A RIGHT OUTER JOIN B ON A.w = B.y

/*
Table A       Table B       Result
+-------+     +-------+     +---------------------------+
| w | x |  *  | y | z |  =  | w    | x    | y    | z    |
+-------+     +-------+     +---------------------------+
| 1 | a |     | 2 | k |     | 2    | b    | 2    | k    |
| 2 | b |     | 3 | m |     | 3    | c    | 3    | m    |
| 3 | c |     | 3 | n |     | 3    | c    | 3    | n    |
| 3 | d |     | 4 | p |     | 3    | d    | 3    | m    |
+-------+     +-------+     | 3    | d    | 3    | n    |
                            | NULL | NULL | 4    | p    |
                            +---------------------------+

*/
```

🔹 A(1,2,3) 🔸 B(2,3,4) → ⚫ Result(2,3,4)

Full JOIN



A **`FULL JOIN`** returns all rows from both tables (A and B).

- When a row has a match in both tables, the data is combined.
- When a row exists only in one table, the missing side is filled with **`NULL`**s.

Example

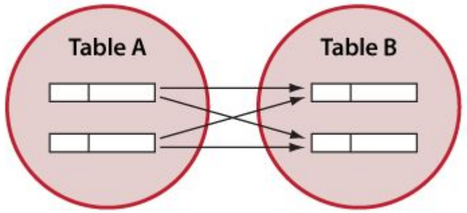
```
FROM A FULL OUTER JOIN B ON A.w = B.y

/*
Table A       Table B       Result
+-------+     +-------+     +---------------------------+
| w | x |  *  | y | z |  =  | w    | x    | y    | z    |
+-------+     +-------+     +---------------------------+
| 1 | a |     | 2 | k |     | 1    | a    | NULL | NULL |
| 2 | b |     | 3 | m |     | 2    | b    | 2    | k    |
| 3 | c |     | 3 | n |     | 3    | c    | 3    | m    |
| 3 | d |     | 4 | p |     | 3    | c    | 3    | n    |
+-------+     +-------+     | 3    | d    | 3    | m    |
                            | 3    | d    | 3    | n    |
                            | NULL | NULL | 4    | p    |
                            +---------------------------+

*/
```

🔹 A(1,2,3) 🔸 B(2,3,4) → ⚫
Result(1,2,3,4)

Cross JOIN



A `CROSS JOIN` combines every row from the first table with every single row from the second table. The `CROSS JOIN` has no `ON` clause, as it creates every possible combination regardless of whether a match exists.

Example

```
FROM A CROSS JOIN B

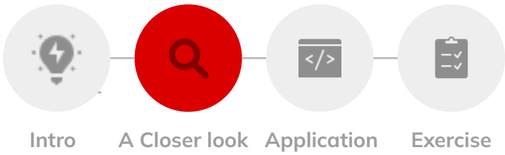
/*
Table A       Table B       Result
+-------+     +-------+     +---------------+
| w | x |  *  | y | z |  =  | w | x | y | z |
+-------+     +-------+     +---------------+
| 1 | a |     | 2 | c |     | 1 | a | 2 | c |
| 2 | b |     | 3 | d |     | 1 | a | 3 | d |
+-------+     +-------+     | 2 | b | 2 | c |
                            | 2 | b | 3 | d |
                            +---------------+

*/
```

🔹 A(1,2) 🔸 B(2,3)

→⚫Result({1,2},{1,3}},{2,2},{2,3})

What you need to know about JOINs



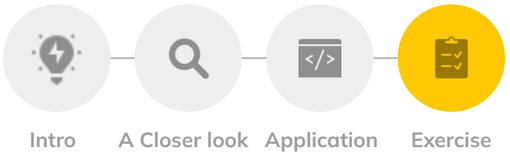
- Joined tables must share at least one common column — the `JOIN key` (except in `CROSS JOIN`s).
- The `JOIN` condition (key or expression) must be defined in the `ON` clause.
- Always check your result size and duplicates — `JOIN`s can multiply rows if keys aren't unique.
- Make sure you know what you want to keep (only matches, or all rows from one/both tables).
- `Alias` your tables clearly (e.g., orders AS o, customers AS c) for readability.
- Be careful with `NULL` values — they don't match in `JOIN`s (`NULL` ≠ `NULL`).

| ***TRKKN*** *Proprietary & Confidential*



Hands-On

Hands-On: **Let’s practice some Joins**

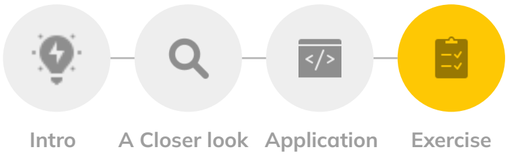


What is the name of the store with the highest total revenue ? Join the `order` and `stores` table and use `GROUP BY` to calculate the sum.

How many different products does the store “`Gasland / Burlington`“ sell in the `Tequila` category ? Join the `stores`, `orders` and `products` tables and use `COUNT distinct` to count unique `products_ids`.

| ***TRKKN*** *Proprietary & Confidential*

Hands-On: Let's practice some Joins



What is the name of the store with the highest total revenue ? Join the `order` and `stores` table and use `GROUP BY` to calculate the sum

```sql
# store with highest revenue
SELECT
    s.store_name,
    ROUND(SUM(o.total_sales_usd),2) AS total_revenue
FROM
    `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.orders` AS o
LEFT JOIN `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.stores` AS s
    ON o.store_id = s.store_id
GROUP BY
    s.store_name
ORDER BY
    total_revenue DESC;
```

Query results

Job information | Results | Visualisation

Row	store_name	total_revenue
1	Hy-Vee #1044 / Burlington	911943.58
2	Wal-Mart 0797 / W Burlington	367307.16
3	Smokin' Joe's #17 Tobacco and ...	213136.85
4	Hy-Vee Food Store #1 / Burlington	182969.38
5	Gasland #102 / Burlington	134435.99
6	Gasland / Burlington	115883.14
7	Fareway Stores #009 / Burlington	95086.84
8	Burlington Shell	63178.31
9	Mepo Foods	53725.63

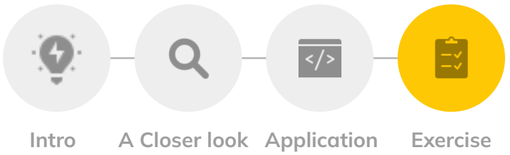
SOLUTION

✅ **store with the highest revenue ?**

Correct Answer: `Hy-Vee #1044 / Burlington`

| ***TRKKN*** *Proprietary & Confidential*

Hands-On: Let's practice some Joins



How many different products does the store "`Gasland / Burlington`" sell in the "`TEQUILA`" category? Join the `stores`, `orders` and `products` tables and use `COUNT distinct` to count the unique `products_ids`.

```sql
SELECT
distinct
    s.store_name,
    COUNT(distinct p.product_id) num_products,

FROM `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.stores` as s
    LEFT JOIN `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.orders`  as o
        ON s.store_id = o.store_id
    LEFT JOIN `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.products` as p
        ON o.product_id = p.product_id
WHERE s.store_name = "Gasland / Burlington" AND p.product_category ="TEQUILA"
GROUP BY 1;
```

Query results

Job information | Results | Visualisation | J

Row	store_name	num_products
1	Gasland / Burlington	6

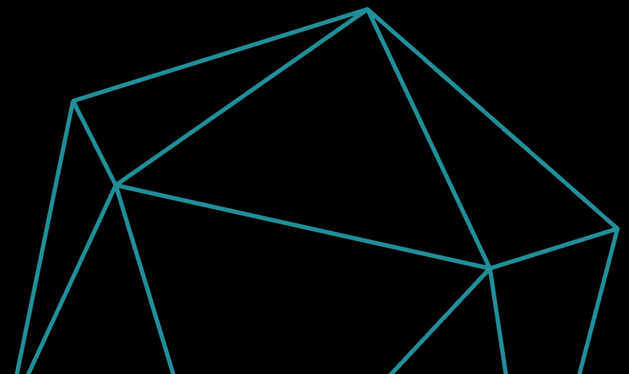
SOLUTION

✅ # sells of Tequila products for `Gasland / Burlington` ?

Correct Answer: `6`

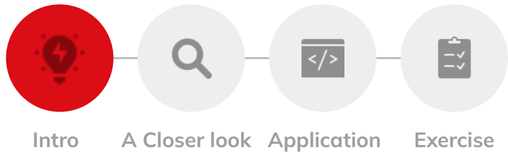
05

How to query nested fields



ANALYTICS SUMMIT

Not all Data is flat !



Data is called **flat** when every value sits in its own row and column, which makes it easy to query.

BUT certain data can be stored in compact, single-cell lists called **`arrays`**. This saves storage, but it traps the information inside.

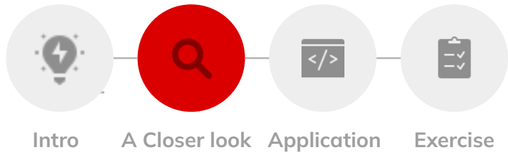
Table_1

column_1 flat	column_2 nested
1	a b
2	a b

```
# This returns an array not a single value
SELECT column_2
FROM Table_1

Result: ["a","b"]
```

Introducing the UNNEST Operator



`UNNEST` turns an array into a table by outputting one row for each element. It's the key tool for flattening nested structures. Different data types can affect how `UNNEST` behaves.

Simple Array

["Milk, "Honey"]

```
SELECT * FROM UNNEST(['Milk',
'Honey']) AS products
```

Row	products
1	Milk
2	Honey

Array of Structs (repeated)

[{ name: Milk, price: 4.99 } , name: Honey, price: 5.99 }]

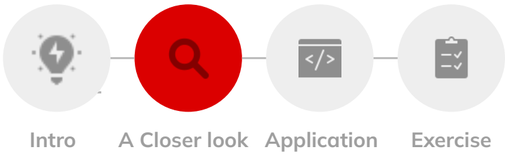
```
SELECT * FROM UNNEST([ STRUCT('Milk' AS
name, 4.99 AS price), STRUCT('Honey' AS
name, 5.99 AS price)])
```

Row	name	price
1	Milk	4.99
2	Honey	5.99

`UNNEST` turns arrays into rows; `STRUCT` defines columns for those rows.

| ***TRKKN*** *Proprietary & Confidential*

Unnesting Simple Arrays



`UNNEST` expands a simple array into separate rows — one for each value.

Table

Order_ID	tags
101	Tech
	Hardware
102	Office
	Accessory

Query

```
SELECT
  Order_ID,
  tags
FROM
  ORDERS,
  UNNEST(tags) as tags
```

Result

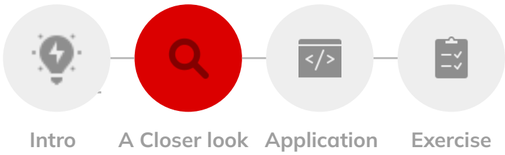
Query results

Job information | Results | Visualisation

Row	Order_ID	tags
1	101	Tech
2	101	Hardware
3	102	Office
4	102	Accessory

| *TRKKN Proprietary & Confidential*

Unnesting Array of Structs (repeated)



If a column contains a list of `STRUCT`s , `UNNEST` makes one row per `STRUCT` and turns each `STRUCT` field into a column. Use dot notation to reference fields (`STRUCT.field`).

Table

Row	Order_ID	Shipping_Details. Warehouse	Shipping_Details. Status
1	101	NY_HUB	Shipped
		CA_PORT	In Transit
2	102	TX_HUB	Processing

Query

```
SELECT
  Order_ID,
  -- Access nested fields using dot notation
  Shipping_Details.Warehouse,
  Shipping_Details.Status
FROM
  ORDERS,
  UNNEST(Shipping_Details) AS Shipping_Details
```

Result

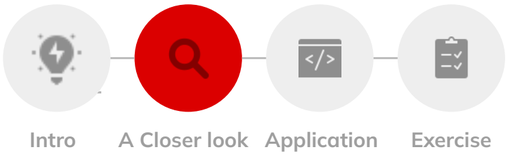
Query results

Job information | Results | Visualisation

Row	Order_ID	Warehouse	Status
1	101	NY_HUB	Shipped
2	101	CA_PORT	In Transit
3	102	TX_HUB	Processing

| **TRKKN** *Proprietary & Confidential*

Chaining Unnests to flatten Tables



When a table has multiple **`arrays`**, you can chain **`UNNEST`** calls to fully flatten the data.

Table

Row	Order_ID	Shipping_Details. Warehouse	Shipping_Details. Status
1	101	NY_HUB	Shipped
		CA_PORT	In Transit
2	102	TX_HUB	Processing

Query

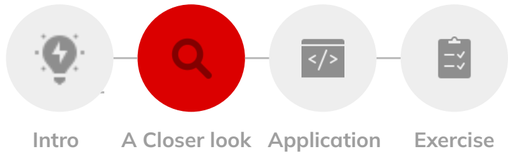
```
SELECT
  Order_ID,
  tags,
  Shipping_Details.Warehouse,
  Shipping_Details.Status
FROM
  ORDERS,
  UNNEST(tags) as tags,
  UNNEST(Shipping_Details) AS Shipping_Details
```

Result

Row	Order_ID	tags	Warehouse	Status
1	101	Tech	NY_HUB	Shipped
2	101	Tech	CA_PORT	In Transit
3	101	Hardware	NY_HUB	Shipped
4	101	Hardware	CA_PORT	In Transit
5	102	Office	TX_HUB	Processing
6	102	Accessory	TX_HUB	Processing

| **TRKKN** *Proprietary & Confidential*

Pro Tip: Use Unnest in SELECT



`UNNEST` in **`SELECT`** lets you peek into **`arrays`** without creating extra rows — safer and faster when you just need the value.

Table

Row	Order_ID	tags	Shipping_Details. Warehouse	Shipping_Details. Status
1	101	Tech	NY_HUB	Shipped
		Hardware	CA_PORT	In Transit
2	102	Office	TX_HUB	Processing
		Accessory		

Query

```
SELECT
  Order_ID,
  /*
  When using in SELECT statements make sure that the
  INNER Select returns only one value and column
  */
  (
    SELECT
      sd.Status
    FROM  UNNEST(Shipping_Details) as sd
    WHERE Status = 'Shipped'
  ) AS Shipped_Status,

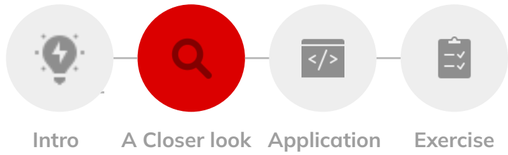
FROM
  ORDERS
```

Result

Row	Order_ID	Shipped_Status
1	101	Shipped
2	102	*null*

| **TRKKN** *Proprietary & Confidential*

Caveat: Rows with empty arrays



`UNNEST` acts like a `CROSS JOIN` by default, so rows with empty arrays disappear. Use `LEFT JOIN UNNEST` when you want to keep those rows and return `NULL` instead.

Table

Row	Order_ID	tags
1	101	Tech
		Hardware
2	102	*0 rows*

Query 1

```
SELECT
  Order_ID,
  tags
FROM
  ORDERS,
  UNNEST(tags) as tags
```

Query 2

```
SELECT
  Order_ID,
  tags
FROM
  ORDERS
  LEFT JOIN
  UNNEST(tags) as tags
```

Result Query 1

Query results

Job information | Results | Visualization

Row	Order_ID	tags
1	101	Tech
2	101	Hardware

Result Query 2

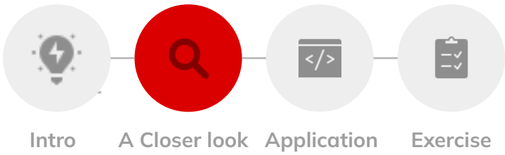
Query results

Job information | Results | Visualization

Row	Order_ID	tags
1	101	Tech
2	101	Hardware
3	102	*null*

| ***TRKKN*** *Proprietary & Confidential*

What to keep in mind



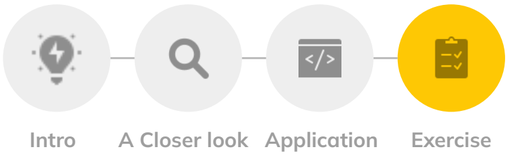
- 💡 `UNNEST` expands each input row into N rows, where N = array length.
- 💡 Chaining multiple `UNNEST`s creates a Cartesian product (lengths multiply).
- 💡 Build incrementally (add one `UNNEST` at a time) and inspect intermediate results.
- 💡 The default syntax (FROM t, UNNEST(arr)) is a `CROSS JOIN`.
- 💡 Unnesting an empty Array makes the entire row disappear, use a `LEFT JOIN` with `CROSS JOIN`

| ***TRKKN*** *Proprietary & Confidential*



Hands-On

Hands-On: **Let’s practice Unnesting**



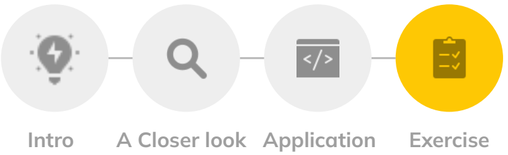
Use a simple dot notation to access the `store_location` fields from the `stores` table.

`Unnest` the `product_category` column in the `stores` table. Each store should have a row for each category in the `product_category` field.

`Unnest` the `product_category` as in the previous exercise. Return `distinct` product categories across all stores.

| ***TRKKN*** *Proprietary & Confidential*

Hands-On: Let's practice Unnesting



Use a simple dot notation to access the `store_location` fields from the `stores` table.

Query

```
SELECT
store_id,
store_name,
store_location.city,
store_location.address,
store_location.zip_code,
store_location.county,
FROM
  `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.stores`
```

Result

Row	store_id	store_name	city	address	zip_code	county
1	4859	Circle K #6604 / Burlington	BURLINGTON	421 N MAIN ST	52601	Des Moines
2	4933	River Mart	BURLINGTON	2107 AGENCY ST	52601	Des Moines
3	2506	Hy-Vee #1044 / Burlington	BURLINGTON	3140 AGENCY	52601	Des Moines
4	2507	Hy-Vee Food Store #1 / Burlington	BURLINGTON	939 ANGULAR	52601	Des Moines
5	5204	Quik Stop / Burlington	BURLINGTON	1812 OSBORN ST	52601	Des Moines
6	4794	Smokin' Joe's #17 Tobacco and ...	BURLINGTON	110 S ROOSEVELT HWY 61	52601	Des Moines
7	4258	Fareway Stores #009 / Burlington	BURLINGTON	920 LAWRENCE DR	52601	Des Moines
8	4898	Burlington Shell	BURLINGTON	130, S ROOSEVELT AVE	52601	Des Moines

| ***TRKKN*** *Proprietary & Confidential*

Hands-On: Let's practice Unnesting

Intro | A Closer look | Application | Exercise

`Unnest` the **`product_category`** column in the **`stores`** table. Each store should have a row for each category in the **`product_category`** field.

Query

```
SELECT
store_id,
store_name,
product_categories
FROM
  `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.stores`
  Unnest(product_categories) as product_categories
```

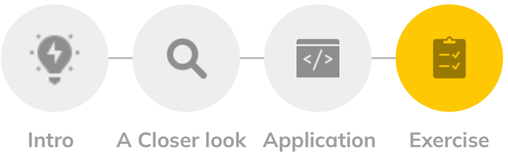
Result

Query results

Job information | Results | Visualisation | JSON | Execution details

Row	store_id	store_name	product_categories
1	4859	Circle K #6604 / Burlington	Convenience
2	4859	Circle K #6604 / Burlington	Snacks
3	4859	Circle K #6604 / Burlington	Beverages
4	4933	River Mart	Convenience
5	4933	River Mart	Snacks
6	4933	River Mart	Beverages
7	2506	Hy-Vee #1044 / Burlington	Groceries
8	2506	Hy-Vee #1044 / Burlington	Pharmacy
9	2506	Hy-Vee #1044 / Burlington	Bakery

Hands-On: Let's practice Unnesting



`Unnest` the **`product_category`** as in the previous exercise.
Return **`distinct`** product categories across all stores.

Query

```
SELECT
  distinct (product_categories) unique_categories
FROM
  `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.stores` ,
  Unnest(product_categories) as product_categories
```

Result

Query results

Job information | Results

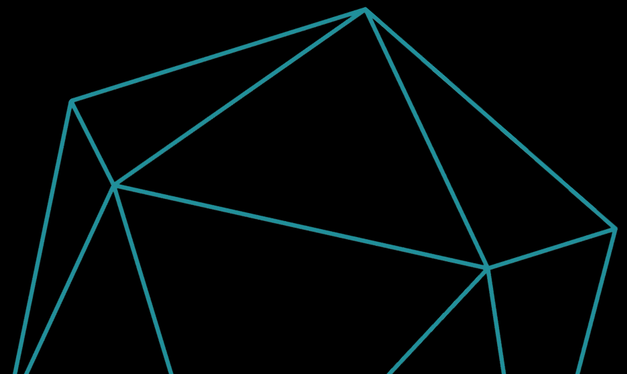
Row	unique_categories
1	Convenience
2	Snacks
3	Beverages
4	Groceries
5	Pharmacy
6	Bakery
7	Deli
8	Meat
9	Produce
10	Health & Beauty
11	Electronics
12	Clothing

| *TRKKN Proprietary & Confidential*

06

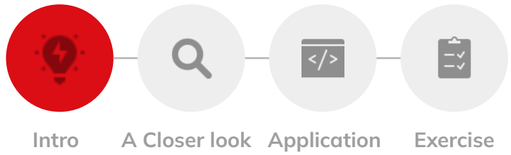
CTEs and Subqueries

Breaking down complex Logic



ANALYTICS SUMMIT

Why Use CTEs and Subqueries



As questions get more complex, single-step SQL queries often aren't enough.

`CTEs` *(Common Table Expression)* and **`Subqueries`** help you break a query into logical steps, reuse intermediate results, and keep your logic readable.

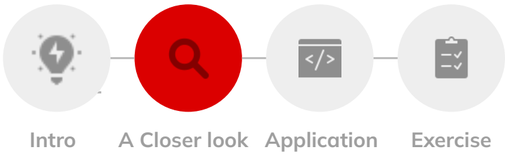
Let's explore when and how to use each.

Example of a CTE:

```sql
WITH step_one AS (
 -- Build an intermediate result
 SELECT ...
 FROM ...
 WHERE ...
),
step_two AS (
 -- Reuse results from step_one
 SELECT ...
 FROM step_one
 JOIN ...
)
-- Final query
SELECT *
FROM step_two;
```

| **TRKKN** *Proprietary & Confidential*

What is a CTE ?



A `CTE` defines a temporary result you can reference in the same query.

Written with `WITH`, it works like a named intermediate result that improves clarity and structure.

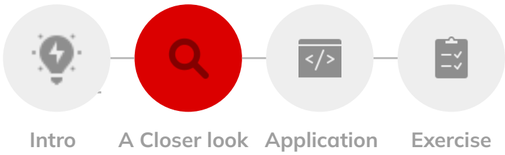
```
WITH numbers AS (          ← CTE definition  of "numbers"

SELECT *
FROM UNNEST([1, 2, 3, 4]) AS value

)

SELECT value
FROM numbers               ← Main query referencing CTE
WHERE value >              ← Reusing CTE to filter
(SELECT AVG(value) FROM numbers);
```

What is a Subquery ?



A **`subquery`** is a query inside a query.

It is typically enclosed in parentheses and used in the **`SELECT`**, **`FROM`**, or **`WHERE`** clause.

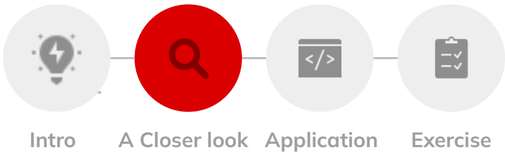
```sql
SELECT
    numbers
FROM UNNEST([1, 2, 3, 4]) AS numbers
WHERE numbers >
(
 SELECT
    AVG(value)
 FROM
    UNNEST([1, 2, 3, 4]) as value
);
```

← Using a Subquery in WHERE clause to Filter

```sql
SELECT
    AVG(numbers) as value
FROM
(
    SELECT *
    FROM
        UNNEST([1, 2, 3, 4]) as numbers
)
```

← Using a Subquery in FROM clause to aggregate its result

CTE vs Subquery - When to use each ?



Subquery

```sql
SELECT
   numbers
FROM UNNEST([1, 2, 3, 4]) AS numbers
WHERE numbers >
(
 SELECT
   AVG(value)
 FROM
   UNNEST([1, 2, 3, 4]) as value
);
```

✅ Best for simple, one-off calculations

❌ Can become hard to read

❌ Must be re-written or repeated if used multiple times.

CTE

```sql
WITH numbers AS (
    SELECT *
    FROM UNNEST([1, 2, 3, 4]) AS value
)

SELECT value
FROM numbers
WHERE value > (SELECT AVG(value) FROM numbers);
```

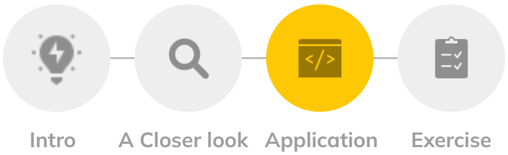
✅ More readable - logic separated and named

✅ Can reference the same CTE multiple times

✅ Best for multi-step logic

| ***TRKKN*** *Proprietary & Confidential*

Combining `With` and `Join`



I want to get the total revenue per product_category

```sql
WITH total_sales_by_category AS (                    ← Step 1
    --Aggregate revenue by product category
    SELECT
      products.product_category,
      SUM(orders.total_sales_usd) AS total_revenue
    FROM `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.orders` AS orders
    --JOIN orders with products table to retrieve category name
    LEFT JOIN                                        ← Step 2
      `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.products` AS products
      ON orders.category_id = products.category_id
    GROUP BY 1                                       ← Step 3
)
SELECT
  product_category,
  ROUND(total_revenue) as total_revenue
FROM total_sales_by_category
ORDER BY total_revenue DESC;                         ← Step 4
```

Building the CTE incrementally:

Step 1: Define `CTE` using `With`

Step 2: `Join` `products` on `orders` to get category names using `category_id` as `Join Key`. Use `Left Join` to keep all orders

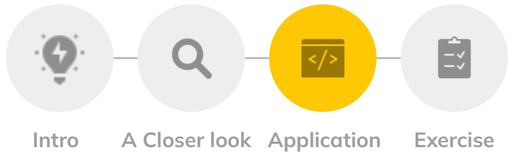
💡 *We join here so we can group by category. But — you can place joins at any step in a CTE, depending on what makes the logic clearest.*

Step 3: `GROUP BY` product_category to return one row per category with total revenue

Step 4: Reuse the `CTE` result and format the number (rounded). `Order by` highest revenue

| TRKKN Proprietary & Confidential

Combining `With` and `Join`



I want to get the total revenue per product_category

```sql
WITH total_sales_by_category AS (
    --Aggregate revenue by product category
    SELECT
      products.product_category,
      SUM(orders.total_sales_usd) AS total_revenue
    FROM `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.orders` AS orders
    --JOIN orders with products table to retrieve category name
    LEFT JOIN
      `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.products` AS products
      ON orders.category_id = products.category_id
    GROUP BY 1

)
SELECT
  product_category,
  ROUND(total_revenue) as total_revenue
FROM total_sales_by_category
ORDER BY total_revenue DESC;
```

Result

Query results

Job information | Results | Visualisation

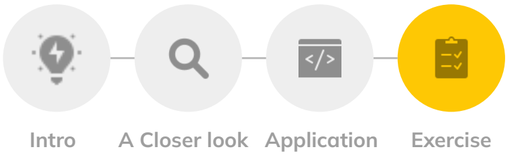
Row	product_category	total_revenue
1	VODKA 80 PROOF	24350627.0
2	CANADIAN WHISKIES	18178195.0
3	STRAIGHT BOURBON WHISKIES	11750994.0
4	TEQUILA	7806839.0
5	VODKA FLAVORED	7694936.0
6	SPICED RUM	7390719.0
7	IMPORTED VODKA	5878052.0
8	BLENDED WHISKIES	5677399.0
9	WHISKEY LIQUEUR	4651793.0
10	IMPORTED GRAPE BRANDIES	4219049.0
11	IMPORTED VODKA - MISC	3951065.0
12	TENNESSEE WHISKIES	3028085.0
13	PUERTO RICO & VIRGIN ISLAND...	2380156.0

| TRKKN Proprietary & Confidential



Hands-On

Hands-On: Let’s practice CTEs or Subqueries

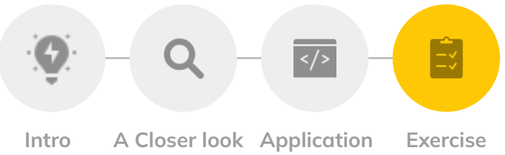


Write a `CTE` to get the total number of products from the `products` table. Use the `product_id` to count products.

Write a `CTE` to get the revenue share for each store of the overall revenue. Use a `Join` with the `store` table to select the `store_name`. Divide the revenue per store by the overall revenue.

| ***TRKKN*** *Proprietary & Confidential*

Hands-On: Let’s practice CTEs or Subqueries



Write a **`CTE`** to get the total number of products from the **`products`** table. Use the **`product_id`** to count products.

Query

```sql
WITH product_counts AS (

    SELECT
        COUNT(DISTINCT product_id) AS total_products
    FROM `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.products`
)
SELECT
    total_products
FROM
    product_counts ;
```

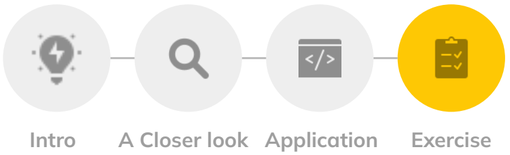
Result

Query results

Job information Res

Row	total_products
1	1312

Hands-On: Let's practice CTEs or Subqueries



Write a **`CTE`** to get the revenue share for each store of the overall revenue. Use a **`Join`** with the **`store`** table to select the **`store_name`**. Divide the revenue per store by the overall revenue.

Query

```sql
WITH store_sales AS (
    --- GROUP BY store
    SELECT
        s.store_name,
        SUM(o.total_sales_usd) AS total_sales
    FROM
        `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.orders` AS o
    JOIN
        `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.stores` AS s
        ON o.store_id = s.store_id
    GROUP BY s.store_name
    )
SELECT
    -- USE CTE to calculate perc of total
    store_name,
    ROUND(total_sales) as total_sales,
    ROUND((total_sales /
    (SELECT SUM(total_sales) FROM   store_sales))*100) perc_of_total
FROM
    store_sales
ORDER BY 3 desc;
```

Result

Query results

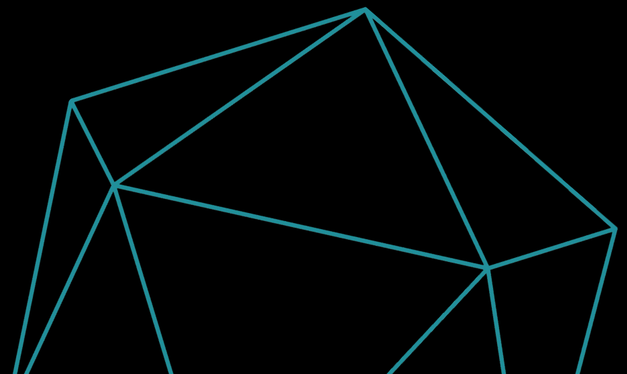
Job information | Results | Visualisation | JSON | Executi

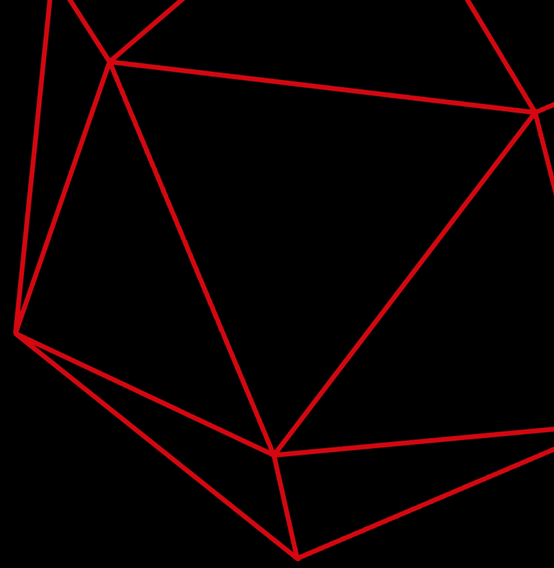
Row	store_name	total_sales	perc_of_total
1	Hy-Vee #1044 / Burlington	911944.0	38.0
2	Wal-Mart 0797 / W Burlington	367307.0	15.0
3	Smokin' Joe's #17 Tobacco and ...	213137.0	9.0
4	Hy-Vee Food Store #1 / Burlington	182969.0	8.0
5	Gasland #102 / Burlington	134436.0	6.0
6	Gasland / Burlington	115883.0	5.0
7	Fareway Stores #009 / Burlington	95087.0	4.0
8	Burlington Shell	63178.0	3.0
9	Quik Stop / Burlington	36605.0	2.0
10	B S Mini Mart Inc	49316.0	2.0
11	Mepo Foods	53726.0	2.0

| *TRKKN Proprietary & Confidential*

07

SQL Hygiene & Best Practices





ANALYTICS SUMMIT

SQL Code Structure

```
SELECT
name,company,seminar, seminar_signup_date
FROM `trkkn-ext-bq-trainings.seminar_1.signups`
```

```
SELECT
    name,
    company,
    seminar,
    seminar_signup_date
FROM
    `trkkn-ext-bq-trainings.seminar_1.signups`
```

- Each column should appear on its own line in `SELECT` statements for better readability.
- Use consistent indentation with 4 spaces recommended for column names and conditions.
- Align keywords like `SELECT`, `FROM`, `WHERE`, and `JOIN` to the left margin without indentation.
- Remove all trailing spaces after column names to maintain clean code.
- Keep the `FROM` clause on the same indentation level as `SELECT`

| *TRKKN Proprietary & Confidential*

Commenting & Naming Conventions

Commenting Best Practices: Comments help explain the purpose and context of your queries. Use single-line comments for brief notes and multi-line comments for detailed explanations

```
-- in-line commenting for brief note
SELECT
   name,
   company,
   seminar,
   seminar_signup_date

/* multi-line comments for
   Detailed explanations */

FROM
    `trkkn-ext-bq-trainings.seminar_1.signups`
```

Commenting & Naming Conventions

Naming Conventions: Consistent naming conventions make code more readable and maintainable. Always use lowercase letters with underscores to separate words (snake_case) for all database objects.

- All column names should use snake_case format, such as seminar_signup_date, customer_id, and order_total.
- Choose descriptive names that clearly indicate the data they contain.
- Use consistent prefixes for boolean fields (is_active, has_subscription) and aggregated values (`total_*`, `avg_*`, `count_*`).
- When using table aliases, keep them short but recognizable
- Avoid using SQL reserved keywords as column names, such as "user", "date", or "order".

SQL Code Structure

Redundant Subqueries: Multiple correlated subqueries scanning the same table repeatedly hurt performance. Replace them with `JOIN`s and aggregation for better efficiency.

Get a list of all orders, but instead of the store_id, you want the actual store_name.

```sql
SELECT
  order_id,
  (SELECT
     store_name
   FROM
     `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.stores`
     AS s
   WHERE
     s.store_id = o.store_id  -- would run for each order
  ) AS store_name
FROM
  `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.orders`
   AS o;
```

```sql
SELECT
  o.order_id,
  s.store_name
FROM
  `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.orders`
   AS o
JOIN
  `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.stores`
   AS s
ON
  o.store_id = s.store_id; -- This connects the tables
```

SQL Code Structure

Missing Table Aliases: Without table aliases, it becomes unclear which table columns belong to, especially when column names overlap. Always use aliases for clarity.

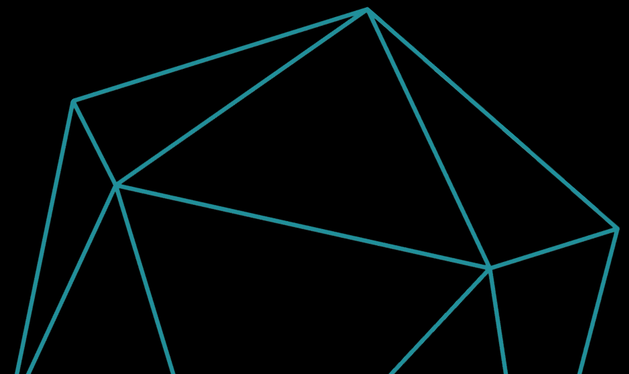
Get a list of every order_id and the store_name where it was sold. To do this, you must JOIN the orders and stores tables.

```
SELECT
  `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.orders`.order_id,
  `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.stores`.store_name
FROM
  `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.orders`
JOIN
  `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.stores`
ON
  -- hard to read …
  `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.orders`.store_id =
`trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.stores`.store_id;
```

```
SELECT
  o.order_id,    -- "Get the order_id from 'o' (orders)"
  s.store_name   -- "Get the store_name from 's' (stores)"
FROM
  `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.orders` AS o
-- Here we create the Alias 'o'
JOIN
  `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.stores` AS s
-- Here we create the Alias 's'
ON
  o.store_id = s.store_id;
```

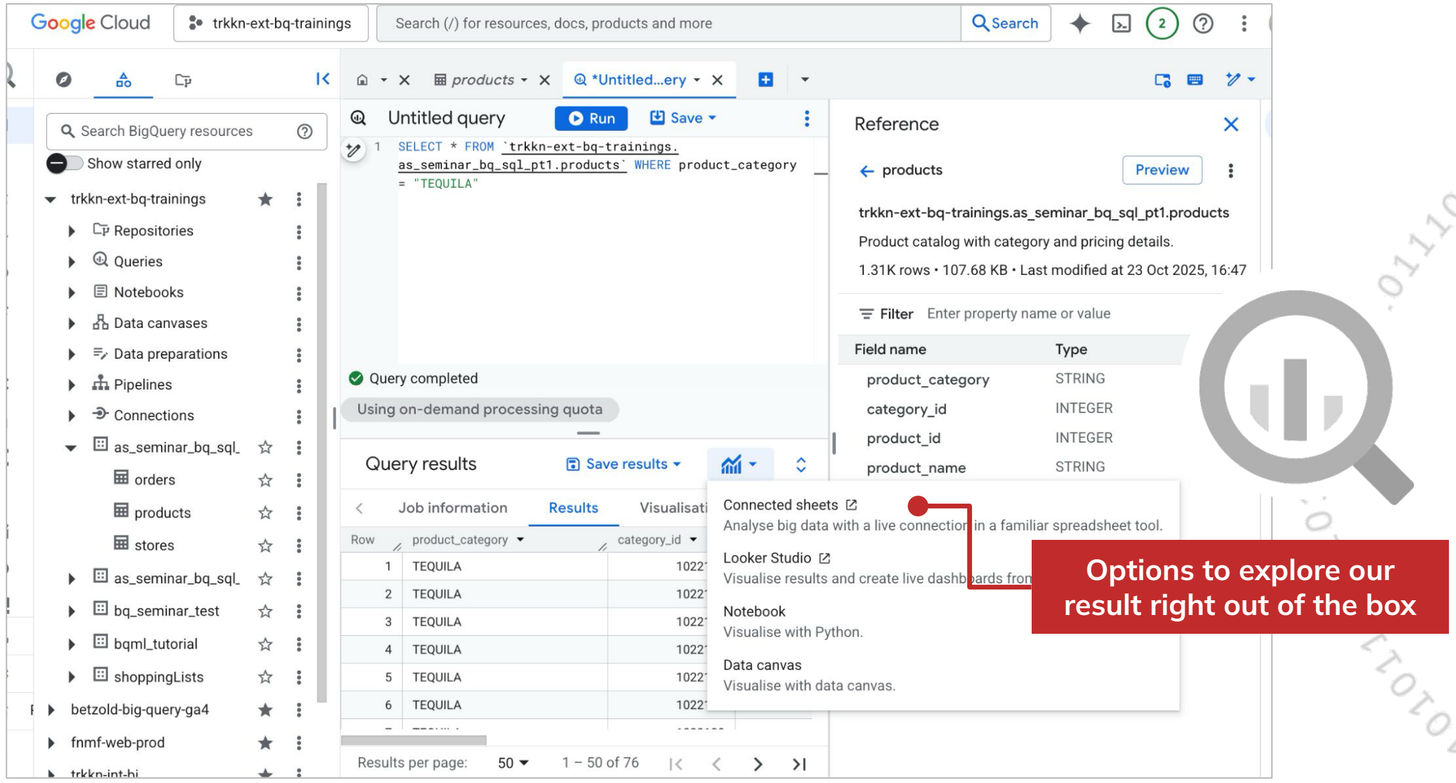
08

Connecting & Visualising Data



ANALYTICS SUMMIT

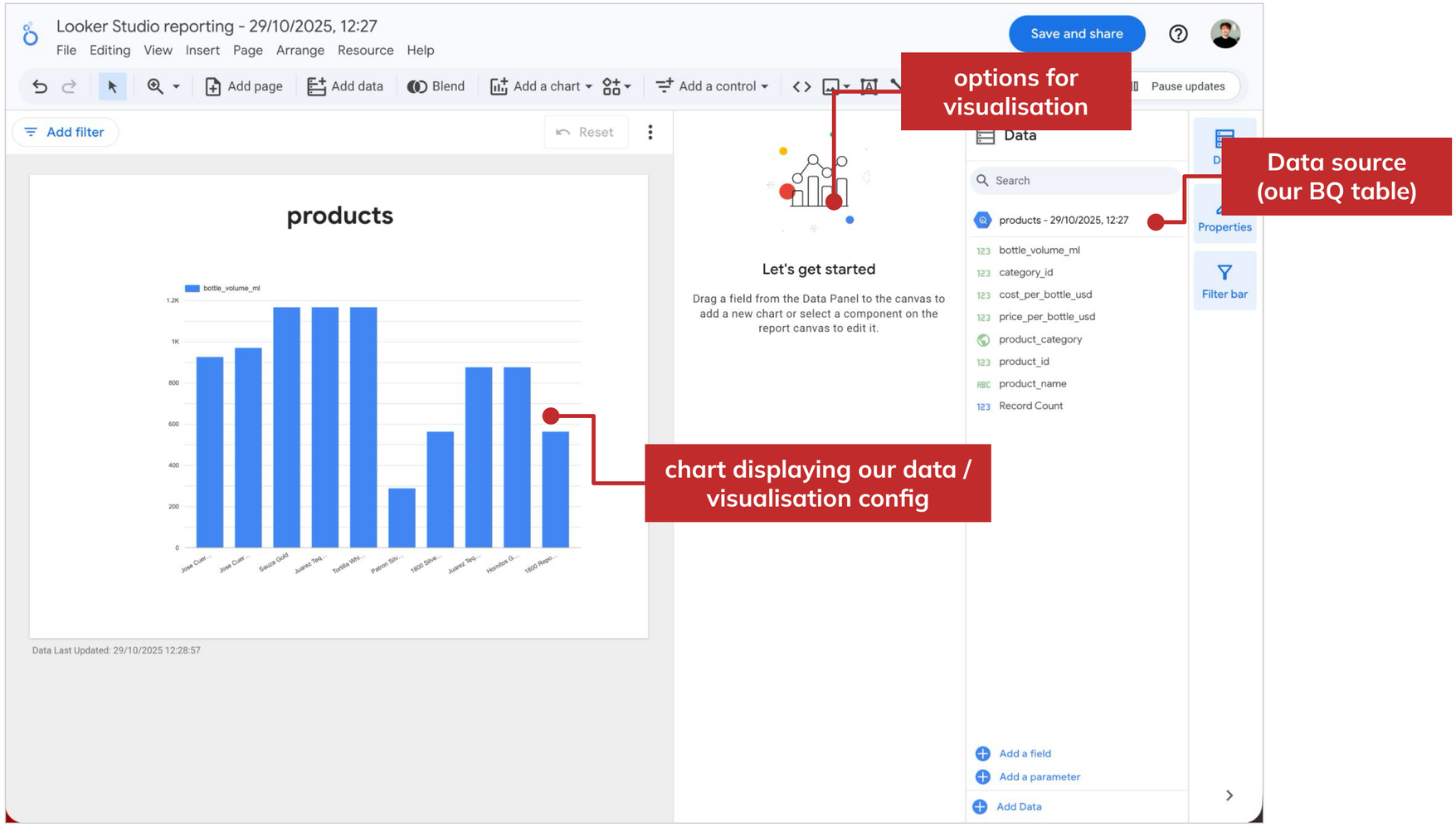
Discovering BigQuery data outside BigQuery



Google Sheets

Looker Studio

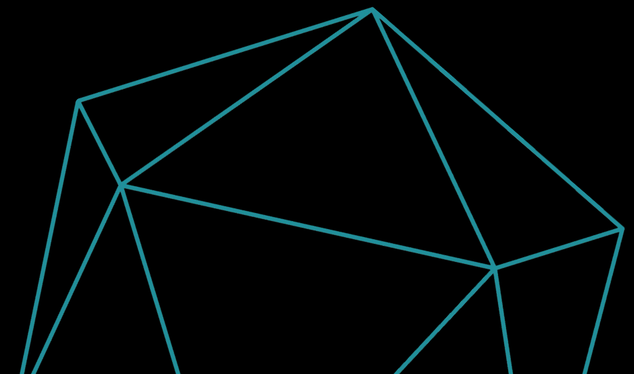
Discovering BigQuery data outside BigQuery



09

Capstone Assignment

Let’s bringing it all together.



ANALYTICS SUMMIT

Final Capstone - Scenario

🧭 Scenario:

You are a **Data Analyst** for Store **“Gasland / Burlington“**, responsible for supporting inventory management and resource planning. Your goal is to explore the sales data and uncover trends.

🎯 Your Assignment

For each task, you will

- Write at least one `SQL` query
- Use `CTEs` or `Subqueries` and other concepts learned so far
- Include clear `comments`, consistent naming, and clean formatting
- Deliver a result table that directly answers the question(s)

| ***TRKKN*** *Proprietary & Confidential*

Final Capstone - Overview

Topics	Your Tasks
🏆 **1. Top / Worst Selling Products**	✅ Aggregate Sales Metrics By Product
📉 **2. Sales Peaks and Drops**	✅ Aggregate Sales metrics by Year and Month
📊 **3. Visualize your results**	✅ Connect your results to Looker Studio

| ***TRKKN*** *Proprietary & Confidential*

Final Capstone - 1. Assignment

🏆 1. Top / Worst Selling Products

Answer the following questions

🔎What is the best-selling product by `revenue` ?

🔎 What is the worst selling product by `revenue` ?

🔎 Which product has the highest `average_revenue_per_bottle` ?

| ***TRKKN*** *Proprietary & Confidential*

Final Capstone - 1. Assignment

🏆 1. Top / Worst Selling Products

Strategy

- Use a `CTE` or Subquery to build your logic
- Start with the `orders` table
- Join the `products` table to get the `product_name`
- Join the `stores` table to filter by `store_name`
- Use `GROUP BY` to group by `product_name`
- Calculate the following metrics:
  - *total_revenue* = `total_sales_usd` *per product*
  - *total_bottles_sold* = `bottles_sold` *per product*
  - *average_revenue_per_bottle = total_revenue / total_bottles_sold*
- Use `ORDER BY` + `LIMIT` to get an ordered list of products

| ***TRKKN*** *Proprietary & Confidential*

Final Capstone - 1. Assignment

🏆 1. Top / Worst Selling Products

Strategy

- Use a `CTE` or Subquery to build your logic
- Start with the `orders` table
- Join the `products` table to get the `product_name`
- Join the `stores` table to filter by `store_name`
- Use `GROUP BY` to group by `product_name`
- Calculate the following metrics:
  - *total_revenue* = `total_sales_usd` *per product*
  - *total_bottles_sold* = `bottles_sold` *per product*
  - *average_revenue_per_bottle = total_revenue / total_bottles_sold*
- Use `ORDER BY` + `LIMIT` to get an ordered list of products

Expected Result

Job information | Results | Visualisation | JSON | Execution details | Execution graph

Row	product_name	total_revenue	total_bottles_sold	avg_revenue_per_bottle
1	Hennessy VS	14179.0	1220.0	11.62
2	Hennessy Vs Cognac	8917.0	313.0	28.49
3	Remy Martin Vsop Cognac	6256.0	407.0	15.37
4	Remy Martin Vsop (flask)	4471.0	276.0	16.2
5	Crown Royal Regal Apple	4312.0	358.0	12.05
6	Paul Masson Grande Amber Bra...	4232.0	879.0	4.82
7	Grey Goose Vodka	3481.0	280.0	12.43

| **TRKKN** *Proprietary & Confidential*

🔍 Final Capstone - 2. Assignment

📉 2. Sales Peaks and Drops

Answer the following questions

🔎Which is the month with the highest `revenue` ?

🔎 Which is the month with the highest number of `orders` ?

🔎 Is the month with the second highest `revenue` also the month with the second highest number of `bottles sold` ?

| ***TRKKN*** *Proprietary & Confidential*

Final Capstone - 2. Assignment

2. Sales Peaks and Drops

Strategy

- Use a `CTE` or Subquery to build your logic
- Use `EXTRACT( {Year| MONTH} from` ... to extract the date parts from the `order_date` column
- `GROUP BY` year and month
- Use aggregation functions (`SUM`, `Count distinct`) to calculate:
  - total_revenue
  - total_orders
  - total_bottles

Final Capstone - 2. Assignment

📉 2. Sales Peaks and Drops

Strategy

- Use a `CTE` or Subquery to build your logic
- Use `EXTRACT( {Year| MONTH} from` ... to extract the date parts from the `order_date` column
- `GROUP BY` year and month
- Use aggregation functions (`SUM`, `Count distinct`) to calculate:
  - total_revenue
  - total_orders
  - total_bottles

Expected Result

Query results

Job information | Results | Visualisation | JSON | Execution details | Execution

Row	year	month	total_revenue	total_orders	total_bottles
1	2024	1	15449.0	84	1663
2	2024	2	14170.0	96	1625
3	2024	3	17107.0	94	1746
4	2024	4	17934.0	104	1817
5	2024	5	13815.0	85	1542
6	2024	6	1822.0	10	156
7	2024	7	15383.0	96	1729
8	2024	8	12852.0	99	1467
9	2024	9	1332.0	90	112

| **TRKKN** *Proprietary & Confidential*

Final Capstone - 3. Assignment

📊 3. Visualize your result(s)

Strategy

- Connect at least one of your query results to Looker Studio
- Choose a appropriate Chart Type (e.g. Line Chart for Time Series)
- Add an informative Head Line
- Format the layout appropriately (Sorting, Font Size, Legend etc.)
- Save the Result

Expected Result





Solutions

Final Capstone - SOLUTION

🏆 1. Top / Worst Selling Products

```sql
WITH total_sales_per_item as(
  -- Join all tables to filter and get the product name field
  SELECT
    p.product_name,
    o.total_sales_usd ,
    o.bottles_sold
  FROM  `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.orders` as o
    LEFT JOIN `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.stores` as s USING(store_id)
    LEFT JOIN `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.products` as p USING(product_id)
  WHERE store_name = "Gasland / Burlington"
),
aggregated_sales_per_item as(
  -- Aggregate by product
  SELECT
    product_name,
    SUM(total_sales_usd) as total_revenue,
    SUM(bottles_sold) as total_bottles_sold
  FROM total_sales_per_item
  GROUP BY 1

)
-- Final select and Ordering
SELECT
    product_name,
    ROUND(total_revenue) as total_revenue ,
    ROUND(total_bottles_sold) as total_bottles_sold,
    ROUND(total_revenue/total_bottles_sold,2) as avg_revenue_per_bottle
FROM aggregated_sales_per_item
ORDER BY 2 desc  -- adjuts based on the column you want to order by
LIMIT 10; -- get the first 10 results
```

Row	product_name	total_revenue	total_bottles_sold	avg_revenue_per_...
1	Hennessy VS	14179.0	1220.0	11.62
2	Hennessy Vs Cognac	8917.0	313.0	28.49
3	Remy Martin Vsop Cognac	6256.0	407.0	15.37
4	Remy Martin Vsop (flask)	4471.0	276.0	16.2
5	Crown Royal Regal Apple	4312.0	358.0	12.05
6	Paul Masson Grande Amber Bra...	4232.0	879.0	4.82
7	Grey Goose Vodka	3481.0	280.0	12.43
8	Admiral Nelson Spiced Rum	3138.0	264.0	11.88
9	Seagrams 7 Crown Bl Whiskey	3045.0	358.0	8.51
10	Black Velvet	2436.0	470.0	5.18

SOLUTIONS

✅ What is the best-selling product by `total_revenue` ?
Correct Answer: **`Hennessy VS`**

✅ What is the worst selling product by `total_revenue` ?
Correct Answer:: **`Kessler Blend Whiskey`**

✅ Which product has the highest `average_revenue_per_bottle` ?
Correct Answer:: **`Fireball Cinnamon Whiskey Mini Dispenser`**

| ***TRKKN*** *Proprietary & Confidential*

Final Capstone - SOLUTION

📉 2. Sales Peaks and Drops

```sql
WITH  revenue_per_category as (
  -- Aggregate metrics by year, month
  SELECT
    EXTRACT( Year from CAST( order_date as date)) as year,
    EXTRACT( Month from CAST( order_date as date)) as month,
    ROUND(sum(total_sales_usd)) as total_revenue ,
    COUNT(Distinct order_id) as total_orders,
    SUM(bottles_sold) as total_bottles,
  FROM `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.orders`
  LEFT JOIN `trkkn-ext-bq-trainings.as_seminar_bq_sql_pt1.stores` USING(store_id
  WHERE store_name = "Gasland / Burlington"
  GROUP BY 1,2

)
  -- Final Select and Ordering
SELECT
  year,
  month,
  total_revenue,
  total_orders,
  total_bottles
FROM revenue_per_category as revenue
ORDER BY 1,2
```

Query results

Job information | Results | Visualisation | JSON | Execution details | Execution

Row	year	month	total_revenue	total_orders	total_bottles
1	2024	1	15449.0	84	1663
2	2024	2	14170.0	96	1625
3	2024	3	17107.0	94	1746
4	2024	4	17934.0	104	1817
5	2024	5	13815.0	85	1542
6	2024	6	1822.0	10	156
7	2024	7	15383.0	96	1729
8	2024	8	12852.0	99	1467
9	2024	9	1332.0	90	112
10	2024	10	2748.0	153	200
11	2024	11	1360.0	85	108
12	2024	12	1911.0	113	151

SOLUTIONS

✅ **Which is the month with the highest revenue ?**

Correct Answer: `APRIL`

✅ **Which is the month with the highest number of orders ?**

Correct Answer: `OCTOBER`

✅ **Is the month with the a) second highest revenue also the month with b) the second highest number of bottles sold ?**

Correct Answer: Yes both a) and b) is `March`

| ***TRKKN*** *Proprietary & Confidential*

Final Capstone - SOLUTION

3. Visualize the end result

