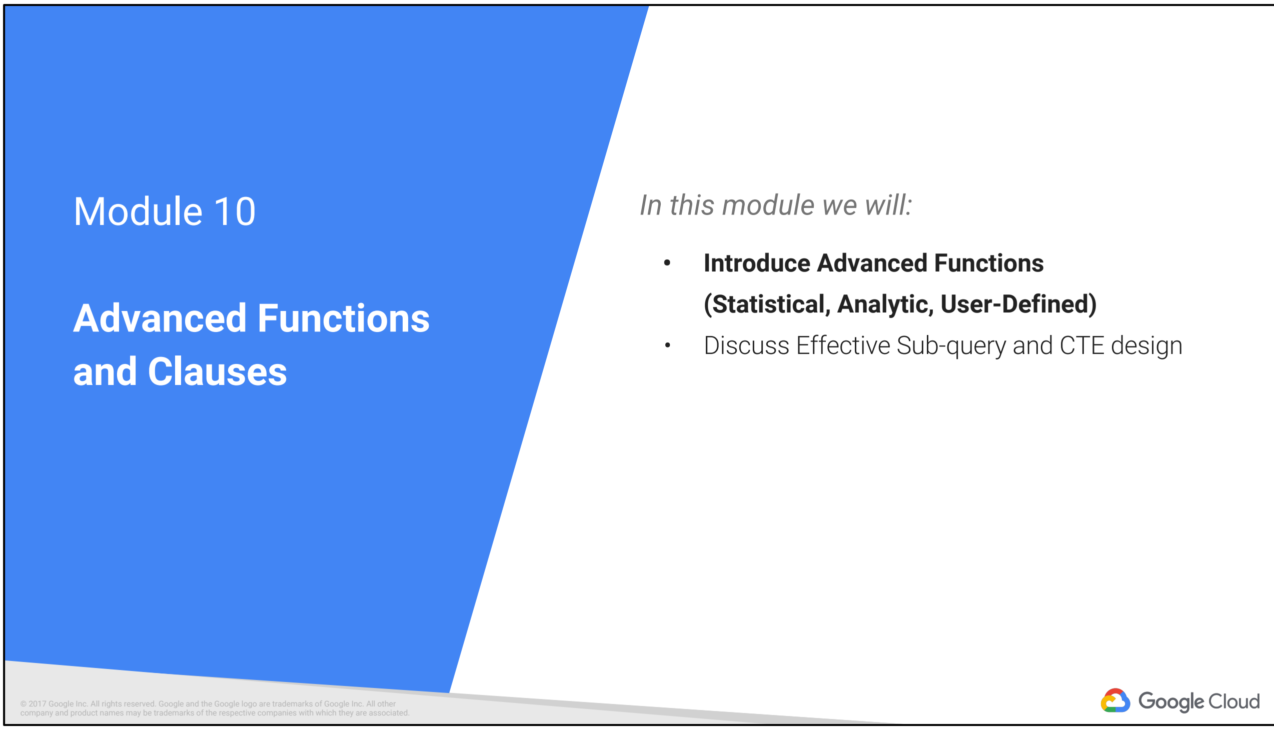



In this module we continue our journey with SQL by delving into a few more advanced concepts like statistical approximation functions and user-defined functions. Then we will explore how to break apart really complex data questions into step-by-step modular pieces in SQL with common table expressions and subqueries.

Let's start by revisiting the SQL functions we've covered so far.

# Use the Right Function for the Right Job

- String Manipulation Functions - FORMAT( )
- Aggregation Functions - SUM( ) COUNT( ) AVG( ) MAX( )
- Data Type Conversion Functions - CAST( )
- Date Functions - PARSE_DATETIME( )
- **Statistical Functions**
- **Analytic Functions**
- **User-defined Functions**

[BigQuery Functions Reference]



Google Cloud

Aggregation = perform calculations over a set of values (like SUM, COUNT, MIN, MAX)

String Manipulation = make every letter uppercase, pull the left 5 characters, format

Statistical = standard deviation, variance, and more

Analytic = perform aggregations over a subset or window of data

User-defined = write your own function recipe in SQL or even Javascript

# **Run Statistical Functions** over Values

```
SELECT
  STDDEV(noemplyeesw3cnt) AS st_dev_employee_count,
  CORR(totprgmrevnue, totfuncexpns) AS corr_rev_expenses
FROM
  `bigquery-public-data.irs_990.irs_990_2015`
```

How **correlated** do you think Program Revenue and Total Functional Expenses are?

More SQL Statistical Functions



Google Cloud

Standard Deviation = How different each group member is from the mean

Corr = Correlation (0 is no Correlation, -1 or 1 indicate very strong correlations)
Returns the Pearson coefficient of correlation of a set of number pairs. For each number pair, the first number is the dependent variable and the second number is the independent variable. The return result is between -1 and 1. A result of 0 indicates no correlation.

Question for class:
How correlated do you think Program Revenue and Total Functional Expenses are?

# **Run Statistical Functions** over Values

```
SELECT
  STDDEV(noemplyeesw3cnt) AS st_dev_employee_count,
  CORR(totprgmrevnue, totfuncexpns) AS corr_rev_expenses
FROM
  `bigquery-public-data.irs_990.irs_990_2015`
```

| Row | st_dev_employee_count | corr_rev_expenses |
|---|---|---|
| 1 | 1579.8005361247351 | 0.9761801901905149 |

More SQL Statistical Functions



Google Cloud

As you might have expected, expenses are highly (almost 1 for 1) associated with revenue. It takes money to make money for these nonprofits!

## Try **Approximate Aggregate Functions** when Close Enough will do

```
#standardSQL
SELECT
  APPROX_COUNT_DISTINCT(ein) AS approx_count,
  COUNT(DISTINCT ein) AS exact_count
FROM
  `bigquery-public-data.irs_990.irs_990_2015`
```

| Row | approx_count | exact_count |
|---|---|---|
| 1 | 276880 | 275077 |

Table JSON

More SQL Approximation Functions

Google Cloud

Background Reading:
https://cloud.google.com/blog/big-data/2017/07/counting-uniques-faster-in-bigquery-with-hyperloglog

Approximate aggregate functions are scalable in terms of memory usage and time, but produce approximate results instead of exact results.

**Not that useful on small datasets** but scales very well in the TB and PB range.

### HyperLogLog++ Functions

BigQuery supports several functions that use HyperLogLog++ for estimating the number of unique values in a large dataset.

They are similar to APPROX_COUNT_DISTINCT, but they are more flexible in the following ways:

- These functions operate on sketches that compress an arbitrary set into a fixed-memory representation. The sketches can be merged to produce a new sketch that represents the *union*, before a final estimate is extracted from the sketch.
- The cardinality of the set can be estimated with a probabilistic error.

https://cloud.google.com/bigquery/docs/reference/standard-sql/functions-and-operator

HLL Paper:
https://research.google.com/pubs/pub40671.html

## **Approximate Users Per Year** of All Github User Logins

```
#standardSQL
SELECT
   CONCAT('20', _TABLE_SUFFIX) year,
   APPROX_COUNT_DISTINCT(actor.login) approx_cnt
FROM `githubarchive.year.20*`
GROUP BY year
ORDER BY year

# 3.8s elapsed, 8.37 GB processed
```

| Row | year | approx_cnt |
|---|---|---|
| 1 | 2011 | 540440 |
| 2 | 2012 | 1188211 |
| 3 | 2013 | 2208240 |
| 4 | 2014 | 3117587 |
| 5 | 2015 | 4440679 |
| 6 | 2016 | 6643627 |

Example from Google Big Data Blog

Google Cloud

Background Reading:
https://cloud.google.com/blog/big-data/2017/07/counting-uniques-faster-in-bigquery-with-hyperloglog

#standardSQL
SELECT
CONCAT('20', _TABLE_SUFFIX) year, APPROX_COUNT_DISTINCT(actor.login) approx_cnt
FROM `githubarchive.year.20*`
GROUP BY year
ORDER BY year

## **Bonus: Approximate Unique Github Users** Since 2011

```
#standardSQL
WITH github_year_sketches AS (
SELECT
    CONCAT('20', _TABLE_SUFFIX) AS year,
   APPROX_COUNT_DISTINCT(actor.login) AS approx_cnt,
   HLL_COUNT.INIT(actor.login) AS sketch # HyperLogLog Estimation
FROM `githubarchive.year.20*`
GROUP BY year
ORDER BY year)

SELECT HLL_COUNT.MERGE(sketch) AS approx_unique_users
FROM `github_year_sketches`
#4.2s elapsed, 8.37 GB processed
#11,334,294 Unique Github Users, Only 0.3% off exact count
```

← we'll cover WITH clauses shortly

[Example from Google Big Data Blog](#)



Google Cloud

HLL = HyperLogLog

Background Reading:
https://cloud.google.com/blog/big-data/2017/07/counting-uniques-faster-in-bigquery-with-hyperloglog

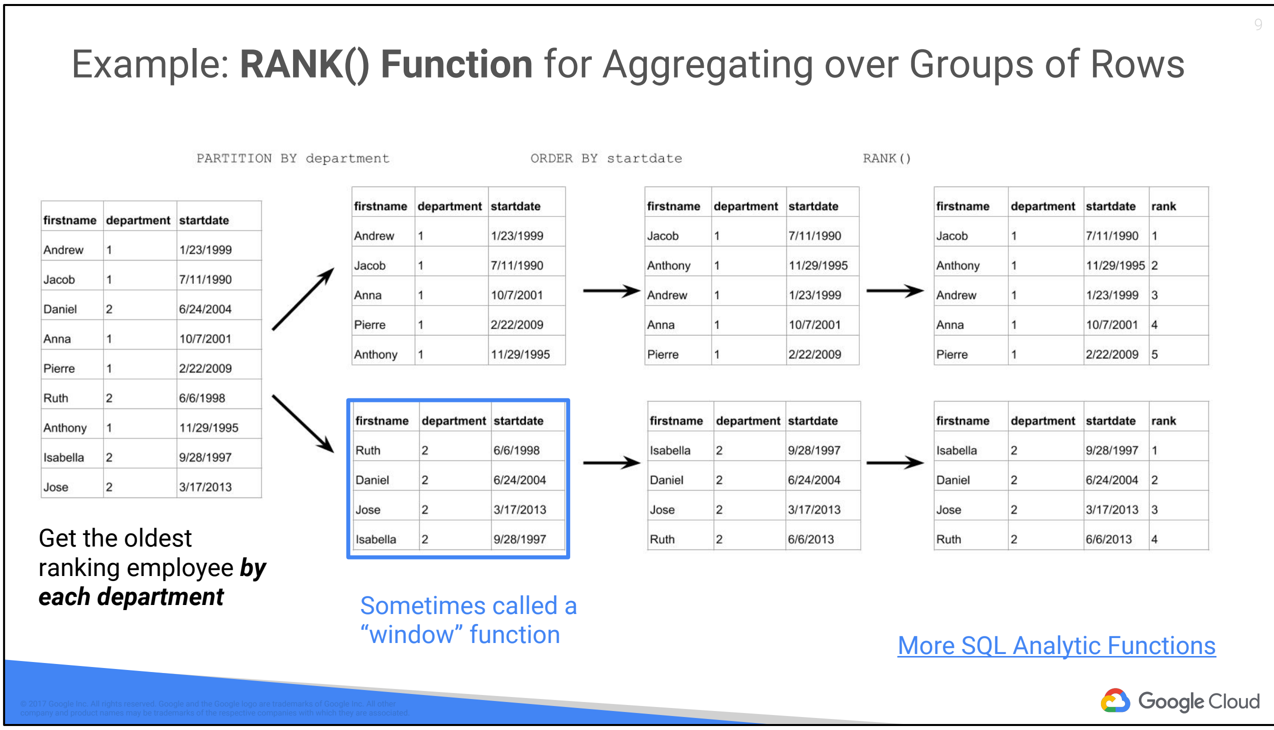
#standardSQL
SELECT
CONCAT('20', _TABLE_SUFFIX) year, APPROX_COUNT_DISTINCT(actor.login) approx_cnt
FROM `githubarchive.year.20*`
GROUP BY year
ORDER BY year

# Use **Analytic Window Functions** for Advanced Analysis

- Standard aggregations
  - `SUM`, `AVG`, `MIN`, `MAX`, `COUNT`, etc.
- Navigation functions
  - `LEAD()` – Returns the value of a row *n* rows ahead of the current row
  - `LAG()` – Returns the value of a row *n* rows behind the current row
  - `NTH_VALUE()` – Returns the value of the *n*th value in the window
- Ranking and numbering functions
  - `CUME_DIST()` – Returns the cumulative distribution of a value in a group
  - `DENSE_RANK()` – Returns the integer rank of a value in a group
  - `ROW_NUMBER()` – Returns the current row number of the query result
  - `RANK()` – Returns the integer rank of a value in a group of values
  - `PERCENT_RANK()` – Returns the rank of the current row, relative to the other rows in the partition



Google Cloud




**Get the oldest ranking employee by each department**

https://cloud.google.com/bigquery/docs/reference/standard-sql/functions-and-operators#supported-functions
RANK( )

In databases, an analytic function is a function that computes aggregate values over a group of rows. Unlike aggregate functions, which return a single aggregate value for a group of rows, analytic functions return a single value for each row by computing the function over a group of input rows.

Analytic functions are a powerful mechanism for succinctly representing complex analytic operations, and they enable efficient evaluations that otherwise would involve expensive self-JOINs or computation outside the SQL query.

Analytic functions are also called "(analytic) window functions" in the SQL standard and some commercial databases. This is because an analytic function is evaluated over a group of rows, referred to as a window or window frame. In some other databases, they may be referred to as Online Analytical Processing (OLAP) functions.

## Example: **RANK() Function** for Aggregating over Groups of Rows

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



Google Cloud

**Option to do demo with IRS dataset:**

```
#standardSQL
# Largest employer per U.S. state per 2015 filing
WITH employer_per_state AS (
SELECT
  ein,
  name,
  state,
  noemplyeesw3cnt AS number_of_employees,
  RANK() OVER (PARTITION BY state ORDER BY noemplyeesw3cnt DESC ) AS
rank
FROM
  `bigquery-public-data.irs_990.irs_990_2015`
JOIN
  `bigquery-public-data.irs_990.irs_990_ein`
USING(ein)
GROUP BY 1,2,3,4 #remove duplicates
)

# Get the top employer per state and order highest to lowest states
SELECT *
FROM employer_per_state
WHERE rank = 1
```

```sql
ORDER BY number_of_employees DESC;
```

# Components of a **User-Defined Function (UDF)**

- CREATE TEMPORARY FUNCTION. Creates a new function. A function can contain zero or more named_parameters
- RETURNS [data_type]. Specifies the data type that the function returns.
- Language [language]. Specifies the language for the function.
- AS [external_code]. Specifies the code that the function runs.

```
CREATE TEMPORARY FUNCTION greeting(a STRING)
RETURNS STRING
LANGUAGE js AS """
  return "Hello, " + a + "!";
  """;
SELECT greeting(name) as everyone
FROM names
```

```
+----------------+
| everyone       |
+----------------+
| Hello, Hannah! |
| Hello, Max!    |
| Hello, Jakob!  |
+----------------+
```

[BigQuery UDFs Reference](https://cloud.google.com/bigquery/docs/reference/standard-sql/user-defined-functions)



Google Cloud

https://cloud.google.com/bigquery/docs/reference/standard-sql/user-defined-functions

Optional UDF demo:

```
#standardSQL
# define two custom javascript UDFs:

# multiply inputs
CREATE TEMPORARY FUNCTION multiplyInputs(x FLOAT64, y FLOAT64)
RETURNS FLOAT64
LANGUAGE js AS """
  return x*y;
""";

# divide by two
CREATE TEMPORARY FUNCTION divideByTwo(x FLOAT64)
RETURNS FLOAT64
LANGUAGE js AS """
  return x / 2;
""";

# generate some data
WITH numbers AS
  (SELECT 1 AS x, 5 as y
```

```
  UNION ALL
  SELECT 2 AS x, 10 as y
  UNION ALL
  SELECT 3 as x, 15 as y)

# invoke UDFs
SELECT
  x,
  y,
  multiplyInputs(x, y) as product,
  divideByTwo(x) as half_x,
  divideByTwo(y) as half_y
FROM numbers;
```

# Pitall: User-Defined Functions **hurt Performance**

- Use native SQL functions whenever possible
- Concurrent rate limits:
  - for non-UDF queries: 50
  - for UDF-queries: 6

[BigQuery Quota Policy]



Google Cloud

Amount of data UDF outputs per input row should be <=5 MB

Each user can run 6 concurrent JavaScript UDF queries per project

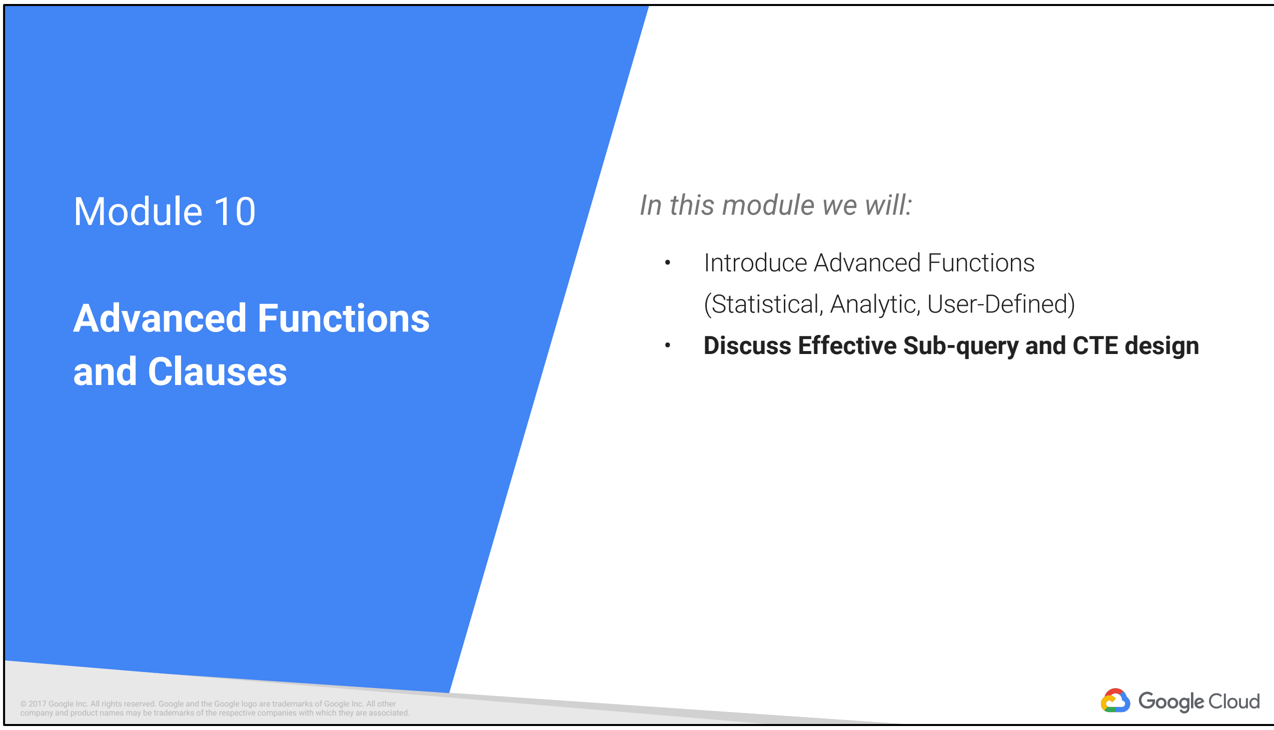
Native code JavaScript functions aren't supported

JavaScript handles only the most significant 32 bits

A query job can have a maximum of 50 JavaScript UDF resources

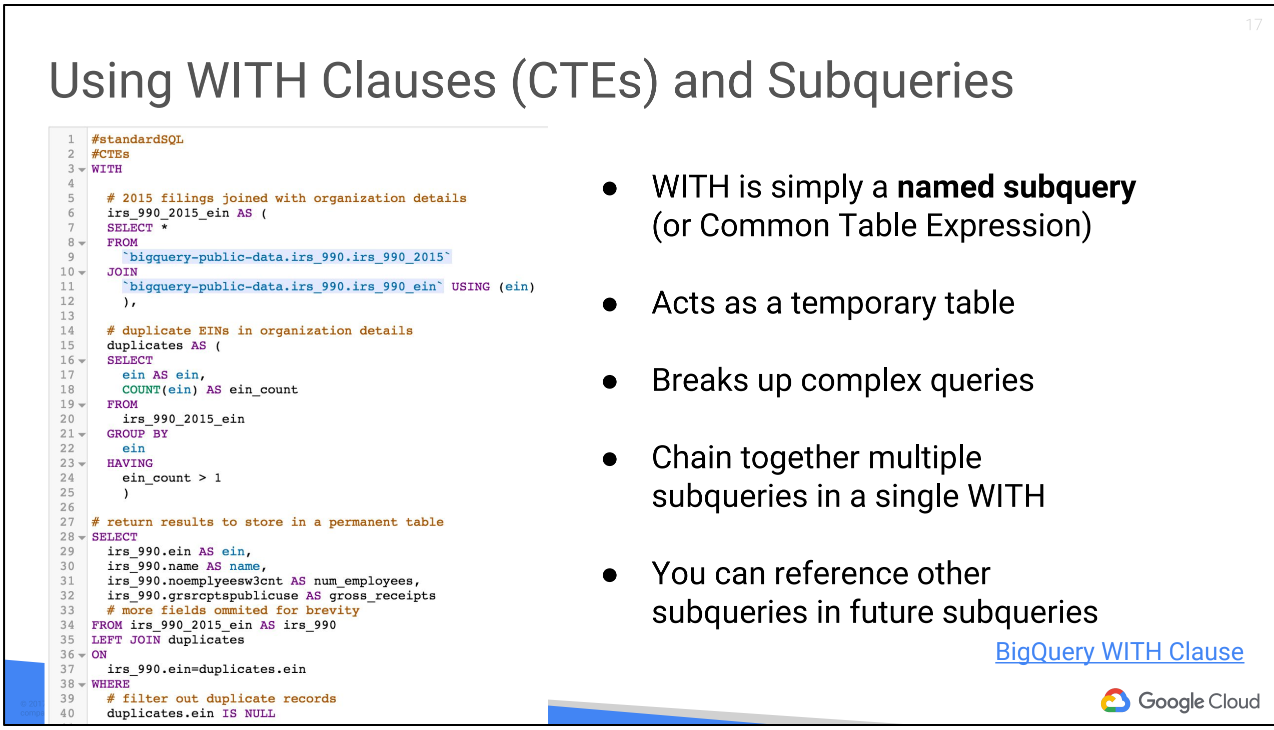
Each inline code blob is limited to maximum size of 32 KB

Each external code resource limited to maximum size of 1 MB




In this module we continue our journey with SQL by delving into a few more advanced concepts like statistical approximation functions and user-defined functions. Then we will explore how to break apart really complex data questions into step-by-step modular pieces in SQL with common table expressions and subqueries.

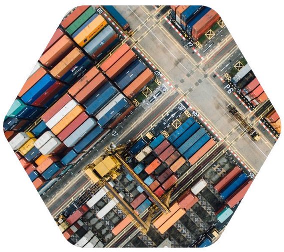
Let’s start by revisiting the SQL functions we’ve covered so far.




```
#standardSQL
#CTEs
WITH

 # 2015 filings joined with organization details
 irs_990_2015_ein AS (
 SELECT *
 FROM
   `bigquery-public-data.irs_990.irs_990_2015`
 JOIN
   `bigquery-public-data.irs_990.irs_990_ein` USING (ein)
   ),

 # duplicate EINs in organization details
 duplicates AS (
 SELECT
   ein AS ein,
   COUNT(ein) AS ein_count
 FROM
   irs_990_2015_ein
 GROUP BY
   ein
 HAVING
   ein_count > 1
```

```
    )

# return results to store in a permanent table
SELECT
  irs_990.ein AS ein,
  irs_990.name AS name,
  irs_990.noemplyeesw3cnt AS num_employees,
  irs_990.grsrcptspublicuse AS gross_receipts
  # more fields ommited for brevity
FROM irs_990_2015_ein AS irs_990
LEFT JOIN duplicates
ON
  irs_990.ein=duplicates.ein
WHERE
  # filter out duplicate records
  duplicates.ein IS NULL

  LIMIT 10
```

## Summary: Answer more complex questions with advanced SQL



Consider using approximation functions for really large datasets

Operate over sub-groups of rows with analytical window functions

User-defined functions add sophistication at the expense of performance

Break apart complex questions into steps with `WITH` and temporary tables



Google Cloud

To wrap up, we finished covering SQL functions which included some pretty neat ones that allow you to statistically estimate with great accuracy across huge datasets. It's your option here whether to trade processing time for 100% accuracy.

Next we covered an example where we wanted to break apart a single table into sub groups of rows and perform a ranking inside each sub-group by using analytical window functions.

After that we introduced UDFs or user-defined functions which can be written in SQL or Javascript. Remember the caveat that query performance is impacted.

Lastly and probably most importantly make liberal use of the WITH clause to break apart a complex question into many smaller steps and tables instead of trying to write one massive combined SQL statement.

Let's practice this in our next lab.

Image (shipping containers) cc0:
https://pixabay.com/en/container-van-aerial-view-block-2568204/
Image (window) cc0: [https://pixabay.com/en/window-prague-twins-941625/](https://pixabay.com/en/window-prague-twins-941625/)
Image (fireworks) cc0: https://unsplash.com/photos/AEnv9_J605M
Image (cube) cc0: https://pixabay.com/en/magic-cube-cube-puzzle-play-378543/

Lab 9

**Deriving Insights with Advanced SQL Functions**



Google Cloud

Lab 9 in Qwiklabs

# Deriving Insights with Advanced SQL Functions

In this lab, you will explore Deriving Insights from Advanced SQL Functions

```
WITH temp_table AS (
...
)

SELECT * FROM temp_table
```



Google Cloud