## 2. Modernizing Data Lakes and Data Warehouses with Google Cloud

* The two key components of any data pipeline are data lakes and warehouses. This course highlights use-cases for each type of storage and dives into the available data lake and warehouse solutions on Google Cloud in technical detail.

## 2.1 Introduction to Data Engineering

* This module describes the role of a data engineer and motivates the claim why data engineering should be done in the Cloud
* The role of a data engineer 4 minutes -https://youtu.be/wEv2pslwdO8
* Data engineering challenges 5 minutes - https://youtu.be/VbBWK3kQcEo
* Introduction to BigQuery 2 minutes - https://youtu.be/2UGA6b5MFI0
* Data lakes and data warehouses 4 minutes - https://youtu.be/Qct49pj7ilM
* Transactional databases versus data warehouses 3 minutes - https://youtu.be/0iMdieh32-g
* Partner effectively with other data teams 5 minutes - https://youtu.be/uteLYDVYtys
* Manage data access and governance 1 minute - https://youtu.be/uteLYDVYtys
* Demo: Finding PII in your dataset with the DLP API 1 minute - https://youtu.be/OQ2sOXaYgUo
* Build production-ready pipelines 2 minutes - https://youtu.be/gO8m8SWu0jk
* Google Cloud customer case study 1 minute - https://youtu.be/OQPpEEMm6Gg
* Using BigQuery to do Analysis 45 minutes
  + Overview
    - In this lab you analyze 2 different public datasets, run queries on them, separately and then combined, to derive interesting insights.
  + What you'll learn
    - Carry out interactive queries on the BigQuery console.
    - Combine and run analytics on multiple datasets.
  + Introduction
    - This lab uses two public datasets in BigQuery: weather data from the US National Oceanic and Atmospheric Administration (NOAA), and bicycle rental data from New York City.
    - You will encounter, for the first time, several aspects of Google Cloud Platform that are of great benefit to scientists:
      * Serverless. No need to download data to your machine in order to work with it - the dataset will remain on the cloud.
      * Ease of use. Run ad-hoc SQL queries on your dataset without having to prepare the data, like indexes, beforehand. This is invaluable for data exploration.
      * Scale. Carry out data exploration on extremely large datasets interactively. You don't need to sample the data in order to work with it in a timely manner.
      * Shareability. You will be able to run queries on data from different datasets without any issues. BigQuery is a convenient way to share datasets. Of course, you can also keep your data private, or share them only with specific persons -- not all data need to be public.
    - The end-result is that you will find if there are lesser bike rentals on rainy days.
  + Explore bicycle rental data
    - Open BigQuery Console
      * Navigation > BigQuery > ADD DATA > Explore public datasets
      * Search "NYC bike" then press Enter. Select “NYC Citi Bike Trips” Dataset
      * BigQuery Console > select bigquery-public-data > new\_york\_citibike > citibike\_trips table.
      * click the Preview tab > Examine the columns and some of the data values.
      * Click Compose New Query and enter the following:

SELECT MIN(start\_station\_name) AS start\_station\_name,

MIN(end\_station\_name) AS end\_station\_name,

APPROX\_QUANTILES(tripduration, 10)[OFFSET (5)] AS typical\_duration,

COUNT(tripduration) AS num\_trips

FROM `bigquery-public-data.new\_york\_citibike.citibike\_trips`

WHERE start\_station\_id != end\_station\_id

GROUP BY start\_station\_id, end\_station\_id

ORDER BY num\_trips DESC

LIMIT 10

* + - * Look at the result and try to determine what this query does ? (Hint: typical duration for the 10 most common one-way rentals)
      * find another interesting fact: total distance travelled by each bicycle in the dataset. Note that the query limits the results to only top 5.

WITH

trip\_distance AS (

SELECT

bikeid,

ST\_Distance(ST\_GeogPoint(s.longitude, s.latitude),

ST\_GeogPoint(e.longitude, e.latitude)) AS distance

FROM

`bigquery-public-data.new\_york\_citibike.citibike\_trips`,

`bigquery-public-data.new\_york\_citibike.citibike\_stations` as s,

`bigquery-public-data.new\_york\_citibike.citibike\_stations` as e

WHERE

start\_station\_id = s.station\_id

AND end\_station\_id = e.station\_id )

SELECT bikeid, SUM(distance)/1000 AS total\_distance

FROM trip\_distance

GROUP BY bikeid

ORDER BY total\_distance DESC

LIMIT 5

* + - * Note: For this query, we also used the other table in the dataset called citibike\_stations to get bicycle station information.
  + Explore the weather dataset
    - select or add bigquery-public-data project > ghcn\_d > ghcnd\_2015.
    - Examine the columns and some of the data values.

SELECT wx.date, wx.value/10.0 AS prcp

FROM `bigquery-public-data.ghcn\_d.ghcnd\_2015` AS wx

WHERE id = 'USW00094728' AND qflag IS NULL AND element = 'PRCP'

ORDER BY wx.date

* + - This query will return rainfall (in mm) for all days in 2015 from a weather station in New York whose id is provided in the query (the station corresponds to NEW YORK CNTRL PK TWR )
  + Find correlation between rain and bicycle rentals
    - How about joining the bicycle rentals data against weather data to learn whether there are fewer bicycle rentals on rainy days?

WITH bicycle\_rentals AS (

SELECT

COUNT(starttime) as num\_trips,

EXTRACT(DATE from starttime) as trip\_date

FROM `bigquery-public-data.new\_york\_citibike.citibike\_trips`

GROUP BY trip\_date

),

rainy\_days AS

(

SELECT date, (MAX(prcp) > 5) AS rainy

FROM (

SELECT wx.date AS date, IF (wx.element = 'PRCP', wx.value/10, NULL) AS prcp

FROM `bigquery-public-data.ghcn\_d.ghcnd\_2015` AS wx

WHERE wx.id = 'USW00094728' )

GROUP BY date

)

SELECT

ROUND(AVG(bk.num\_trips)) AS num\_trips, wx.rainy

FROM bicycle\_rentals AS bk

JOIN rainy\_days AS wx

ON wx.date = bk.trip\_date

GROUP BY wx.rainy

* + - Now you can see the results of joining the bicycle rental dataset with a weather dataset that comes from a completely different source.
      * Table

        Description automatically generated
    - Running the query yields that, yes, New Yorkers ride the bicycle 47% fewer times when it rains.
* Quiz: Introduction to Data Engineering
  + Which of the following are the jobs of a data engineer? (Choose all that apply)
    - Manage the data
    - Get the data to where it can be useful
    - Get the data into a usable condition
    - Productionize data processes
    - Add new value to the data
  + Which of the following statements are true? (Choose TWO)
    - Cloud SQL is optimized for high-throughput writes
    - BigQuery is optimized for high-read data

## 2.2 Building a Data Lake

* In this module, we describe what data lake is and how to use Cloud Storage as your data lake on Google Cloud.
* Introduction to data lakes 8 minutes - https://youtu.be/PkAuvuJDC24
* Data storage and ETL options on Google Cloud 4 minutes - https://youtu.be/KEDmYdVDgkM
* Build a data lake using Cloud Storage 9 minutes - https://youtu.be/1gV73e\_l3ao
* Secure Cloud Storage 6 minutes - https://youtu.be/WUddGaz0f4A
* Store all sorts of data types 4 minutes - https://youtu.be/SLbDqonhPh8
* Cloud SQL as a relational data lake 6 minutes - https://youtu.be/eX5tAdEdy9A
* Loading Taxi Data into Google Cloud SQL 2.5 1 hour 7 minutes
  + Overview
    - In this lab, you will learn how to import data from CSV text files into Cloud SQL and then carry out some basic data analysis using simple queries.
    - The dataset used in this lab is collected by the NYC Taxi and Limousine Commission <https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page> and includes trip records from all trips completed in Yellow and Green taxis in NYC from 2009 to present, and all trips in for-hire vehicles (FHV) from 2015 to present. Records include fields capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, and driver-reported passenger counts.
    - This dataset can be used to demonstrate a wide range of data science concepts and techniques and will be used in several of the labs in the Data Engineering curriculum.
  + Objectives
    - Create Cloud SQL instance
    - Create a Cloud SQL database
    - Import text data into Cloud SQL
    - Check the data for integrity
  + Preparing your Environment
    - Create environment variables that will be used later in the lab for your project ID and the storage bucket that will contain your data:
      * export PROJECT\_ID=$(gcloud info --format='value(config.project)')
      * export BUCKET=${PROJECT\_ID}-ml
  + Create a Cloud SQL instance
    - Enter the following commands to create a Cloud SQL instance:
      * gcloud sql instances create taxi --tier=db-n1-standard-1 --activation-policy=ALWAYS
    - This will take a few minutes to complete.
    - Set a root password for the Cloud SQL instance:
      * gcloud sql users set-password root --host % --instance taxi --password Passw0rd
    - Now create an environment variable with the IP address of the Cloud Shell:
      * export ADDRESS=$(wget -qO - http://ipecho.net/plain)/32
    - Whitelist the Cloud Shell instance for management access to your SQL instance.
      * gcloud sql instances patch taxi --authorized-networks $ADDRESS
    - When prompted press Y to accept the change.
    - Get the IP address of your Cloud SQL instance by running:
      * MYSQLIP=$(gcloud sql instances describe taxi --format="value(ipAddresses.ipAddress)")
    - Check the variable MYSQLIP:
      * echo $MYSQLIP
    - you should get an IP address as an output.
    - Create the taxi trips table by logging into the mysql command line interface.
      * mysql --host=$MYSQLIP --user=root --password --verbose
    - create database if not exists bts;
      * use bts;
      * drop table if exists trips;
      * create table trips (
      * vendor\_id VARCHAR(16),
      * pickup\_datetime DATETIME,
      * dropoff\_datetime DATETIME,
      * passenger\_count INT,
      * trip\_distance FLOAT,
      * rate\_code VARCHAR(16),
      * store\_and\_fwd\_flag VARCHAR(16),
      * payment\_type VARCHAR(16),
      * fare\_amount FLOAT,
      * extra FLOAT,
      * mta\_tax FLOAT,
      * tip\_amount FLOAT,
      * tolls\_amount FLOAT,
      * imp\_surcharge FLOAT,
      * total\_amount FLOAT,
      * pickup\_location\_id VARCHAR(16),
      * dropoff\_location\_id VARCHAR(16)
      * );
    - In the mysql command line interface check the import by entering the following commands:
      * describe trips;
      * select distinct(pickup\_location\_id) from trips;
      * exit
  + Add data to Cloud SQL instance
    - Now you'll copy the New York City taxi trips CSV files stored on Cloud Storage locally. To keep resource usage low, you'll only be working with a subset of the data (~20,000 rows).
    - Run the following in the command line:
      * gsutil cp gs://cloud-training/OCBL013/nyc\_tlc\_yellow\_trips\_2018\_subset\_1.csv trips.csv-1
      * gsutil cp gs://cloud-training/OCBL013/nyc\_tlc\_yellow\_trips\_2018\_subset\_2.csv trips.csv-2
    - Import the CSV file data into Cloud SQL using mysql:
      * mysqlimport --local --host=$MYSQLIP --user=root --password --ignore-lines=1 --fields-terminated-by=',' bts trips.csv-\*
    - Connect to the mysql interactive console:
      * mysql --host=$MYSQLIP --user=root --password
  + Checking for data integrity
    - Whenever data is imported from a source it's always important to check for data integrity. Roughly, this means making sure the data meets your expectations.
    - In the mysql interactive console select the database:
      * use bts;
    - Query the trips table for unique pickup location regions:
      * select distinct(pickup\_location\_id) from trips;
    - digging into the trip\_distance column. Enter the following query into the console:
      * Select max(trip\_distance), min(trip\_distance) from trips;
    - expect the trip distance between 1 to say 1000 miles. The max of 85 miles seems reasonable but the minimum of 0 seems buggy. How many trips in the dataset have a trip distance of 0?
      * select count(\*) from trips where trip\_distance = 0;
    - There are 155 such trips in the database. These trips warrant further exploration. You'll find that these trips have non-zero payment amounts associated with them. Perhaps these are fraudulent transactions? Let's see if we can find more data that doesn't meet our expectations. We expect the fare\_amount column to be positive. Enter the following query to see if this is true in the database:
      * select count(\*) from trips where fare\_amount < 0;
    - There should be 14 such trips returned. Again, these trips warrant further exploration. There may be a reasonable explanation for why the fares take on negative numbers. However, it's up to the data engineer to ensure there are no bugs in the data pipeline that would cause such a result.
    - Finally, let's investigate the payment\_type column.
      * Select payment\_type, count(\*) from trips group by payment\_type;
    - The results of the query indicate that there are four different payment types, with:
      * payment type = 1 has 13863 rows
      * payment type = 2 has 6016 rows
      * payment type = 3 has 113 rows
      * payment type = 4 has 32 rows
    - Digging into the documentation <https://www1.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_yellow.pdf> , a payment type of 1 refers to credit card use, payment type of 2 is cash, and a payment type of 4 refers to a dispute. The figures make sense.
* Quiz: Building a Data Lake
  + Which statement best describes a data lake?
    - The place where you capture every aspect of your business operations. Data is stored in its natural, raw format.
  + Which of the following statements on Cloud Storage are true?
    - Cloud Storage implements both IAM policy and Access Control Lists
    - Cloud Storage simulates a file system
    - Cloud Storage allows you to set retention policies on all objects in a bucket

## 2.3 Building a Warehouse

* In this module, we talk about BigQuery as a data warehousing option on Google Cloud
* The modern data warehouse 3 minutes - <https://youtu.be/ZZ5WQnjtw1Q>
* Introduction to BigQuery 6 minutes - <https://youtu.be/ee6wsaOM12Y>
* Demo: Querying TB of data in seconds 7 minutes - <https://youtu.be/DAPiUo3sAFA>
* Get started with BigQuery 11 minutes - <https://youtu.be/p10DSZ9T_Mg>
* Load data into BigQuery 11 minutes - <https://youtu.be/TKHt3ujcC0c>
* Loading data into BigQuery 1 hour 30 minutes
  + Overview
    - BigQuery is Google's fully managed, NoOps, low cost analytics database. With BigQuery you can query terabytes and terabytes of data without having any infrastructure to manage or needing a database administrator. BigQuery uses SQL and can take advantage of the pay-as-you-go model. BigQuery allows you to focus on analyzing data to find meaningful insights.
    - In this lab you will ingest subsets of the NYC taxi trips data into tables inside of BigQuery.
  + What you'll learn
    - Loading data into BigQuery from various sources
    - Loading data into BigQuery using the CLI and Console
    - Using DDL to create tables
  + Open BigQuery Console
    - Navigation > BigQuery > Done > top of the page, Disable Editor Tabs.
      * This adjusts the BigQuery user interface to non-preview mode.
  + Create a new dataset to store tables
    - Create Dataset > Set the Dataset ID to nyctaxi. Leave the other fields at their default values.
  + Ingest a new Dataset from a CSV
    - In this section, you will load a local CSV into a BigQuery table.
    - Download a subset of the NYC taxi 2018 trips data locally onto your computer from here : <https://storage.googleapis.com/cloud-training/OCBL013/nyc_tlc_yellow_trips_2018_subset_1.csv>
    - In the BigQuery Console, Select the nyctaxi dataset then click Create Table
    - Specify the below table options:
      * Source:
        + Create table from: Upload
        + Choose File: select the file you downloaded locally earlier
        + File format: CSV
      * Destination: Table name: 2018trips Leave all other setting at default.
      * Schema: Check Auto Detect (tip: Not seeing the checkbox? Ensure the file format is CSV and not Avro)
      * Advanced Options: Leave at default values
      * Click Create Table.
    - You should now see the 2018trips table below the nyctaxi dataset.
  + Running SQL Queries
    - * SELECT \* FROM nyctaxi.2018trips ORDER BY fare\_amount DESC LIMIT 5
  + Ingest a new Dataset from Google Cloud Storage
    - load another subset of the same 2018 trip data from Cloud Storage, use the CLI tool to do it.
    - In your Cloud Shell, run the following command :
      * bq load --source\_format=CSV --autodetect --noreplace nyctaxi.2018trips \
      * gs://cloud-training/OCBL013/nyc\_tlc\_yellow\_trips\_2018\_subset\_2.csv
    - this subset is to be appended to the existing 2018trips table that you created above.
    - When the load job is complete, you will get a confirmation on the screen.
    - Confirm that the row count has now almost doubled.
  + Create tables from other tables with DDL
    - The 2018trips table now has trips from throughout the year. Interested in January trips? use DDL to extract this data and store it in another table, run the following CREATE TABLE command :
      * CREATE TABLE nyctaxi.january\_trips AS
      * SELECT \* FROM nyctaxi.2018trips WHERE EXTRACT(Month FROM pickup\_datetime)=1;
    - Now run the below query, find the longest distance traveled in the month of January:
      * SELECT \* FROM nyctaxi.january\_trips ORDER BY trip\_distance DESC LIMIT 1
* Explore schemas 1 minute - <https://youtu.be/pKDZ5XTD2ig>
* Demo: Exploring Schemas 10 minutes - <https://youtu.be/TR7myixeUgE>
* Schema design 3 minutes - <https://youtu.be/sqKVS2w1HwU>
* Nested and repeated fields 9 minutes - <https://youtu.be/QSgFu9bAjoo>
* Demo: Nested and repeated fields 15 minutes - <https://youtu.be/jiBSkTa_N50> <https://github.com/jonesevan/training-data-analyst>
* Design the optimal schema for BigQuery 1 minute - <https://youtu.be/pa0Yhe0g2g8>
* Working with JSON and Array data in BigQuery 2.5 1 hour
  + Overview
    - BigQuery is Google's fully managed, NoOps, low cost analytics database. With BigQuery you can query terabytes and terabytes of data without having any infrastructure to manage or needing a database administrator. BigQuery uses SQL and can take advantage of the pay-as-you-go model. BigQuery allows you to focus on analyzing data to find meaningful insights.
    - This lab is an in-depth walkthrough of working with semi-structured data (ingesting JSON, Array data types) inside of BigQuery. Denormalizing your schema into a single table with nested and repeated fields can yield performance improvements, but the SQL syntax for working with array data can be tricky. You will practice loading, querying, troubleshooting, and unnesting various semi-structured datasets.
  + Objectives
    - Loading semi-structured JSON into BigQuery
    - Creating and querying arrays
    - Creating and querying structs
    - Querying nested and repeated fields
  + Open BigQuery Console
    - Navigation menu > BigQuery > Done
  + Create a new dataset to store our tables
    - Create a dataset (fruit\_store). Leave the other options at their default values.
  + Practice working with Arrays in SQL
    - Normally in SQL you will have a single value for each row like this list of fruits below:

|  |  |
| --- | --- |
| **Row** | **Fruit** |
| 1 | raspberry |
| 2 | blackberry |
| 3 | strawberry |
| 4 | cherry |

* + - What if you wanted a list of fruit items for each person at the store?:

|  |  |  |
| --- | --- | --- |
| **Row** | **Fruit** | **Person** |
| 1 | raspberry | sally |
| 2 | blackberry | sally |
| 3 | strawberry | sally |
| 4 | cherry | sally |
| 5 | orange | frederick |
| 6 | apple | frederick |

* + - In traditional relational database SQL, you would look at the repetition of names and immediately think to split the above table into two separate tables: Fruit Items and People. That process is called normalization (going from one table to many). This is a common approach for transactional databases like mySQL.
    - For data warehousing, data analysts often go the reverse direction (denormalization) and bring many separate tables into one large reporting table.
    - What are some potential issues if you stored all your data in one giant table?
      * The table row size could be too large for traditional reporting databases
    - Stores data at different levels of granularity all in one table using repeated fields:

|  |  |  |
| --- | --- | --- |
| **Row** | **Fruit (array)** | **Person** |
| 1 | raspberry | sally |
| blackberry |  |
| strawberry |  |
| cherry |  |
| 2 | orange | frederick |
| apple |  |

* + - What looks strange about the previous table?
      * It's only two rows.
      * There are multiple field values for Fruit in a single row.
      * The people are associated with all of the field values.
    - What the key insight? The array data type!
    - An easier way to interpret the Fruit array:

|  |  |  |
| --- | --- | --- |
| **Row** | **Fruit (array)** | **Person** |
| 1 | [raspberry, blackberry, strawberry, cherry] | sally |
| 2 | [orange, apple] | frederick |

* + - Both of these tables are exactly the same. There are two key learnings here:
      * An array is simply a list of items in brackets [ ]
      * BigQuery visually displays arrays as flattened. It simply lists the value in the array vertically (note that all of those values still belong to a single row)
    - Try it yourself. Enter the following in the BigQuery Query Editor:
      * SELECT ['raspberry', 'blackberry', 'strawberry', 'cherry'] AS fruit\_array
    - Now try executing this one:
      * SELECT ['raspberry', 'blackberry', 'strawberry', 'cherry', 1234567] AS fruit\_array
      * You should get an error that looks like the following:
      * Error: Array elements of types {INT64, STRING} do not have a common supertype at [3:1]
    - Why did we get this error?
      * Data in an array [ ] must all be the same type
    - Here's the final table to query against:
      * SELECT person, fruit\_array, total\_cost FROM `data-to-insights.advanced.fruit\_store`;
        + Table

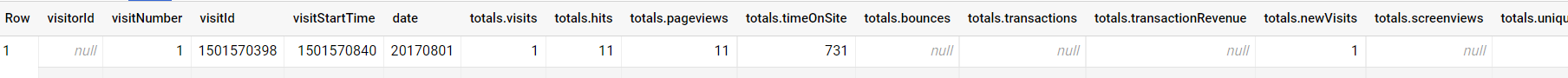
          Description automatically generatedText, letter

          Description automatically generated
    - Loading semi-structured JSON into BigQuery
      * What if you had a JSON file that you needed to ingest into BigQuery?
      * Create a new table in the fruit\_store dataset.
      * Add the following details for the table:
        + Source: Choose Google Cloud Storage in the Create table from dropdown.
        + Select file from GCS bucket (type or paste the following): data-insights-course/labs/optimizing-for-performance/shopping\_cart.json
        + File format: JSONL (Newline delimited JSON) {This will be auto-populated}
        + Schema: Check Auto detect (Schema and input parameters).
        + Call the new table "fruit\_details".
        + Click Create table.
      * In the schema, note that fruit\_array is marked as REPEATED which means it's an array.
    - Recap
      * BigQuery natively supports arrays
      * Array values must share a data type
      * Arrays are called REPEATED fields in BigQuery
  + Creating your own arrays with ARRAY\_AGG()
    - Don't have arrays in your tables already? You can create them!
    - query to explore this public dataset
      * + SELECT fullVisitorId, date, v2ProductName, pageTitle
        + FROM `data-to-insights.ecommerce.all\_sessions`
        + WHERE visitId = 1501570398 ORDER BY date
      * Graphical user interface, text, application, email

        Description automatically generated
      * How many rows are returned? 111
    - use the ARRAY\_AGG() function to aggregate our string values into an array.
      * + SELECT fullVisitorId, date, ARRAY\_AGG(v2ProductName) AS products\_viewed,
        + ARRAY\_AGG(pageTitle) AS pages\_viewed
        + FROM `data-to-insights.ecommerce.all\_sessions`
        + WHERE visitId = 1501570398 GROUP BY fullVisitorId, date ORDER BY date
      * How many rows are returned? 2 - one for each day
      * Graphical user interface, text, application

        Description automatically generated
    - use the ARRAY\_LENGTH() function to count the number of pages and products that were viewed.
      * + SELECT fullVisitorId, date, ARRAY\_AGG(v2ProductName) AS products\_viewed,
        + ARRAY\_LENGTH(ARRAY\_AGG(v2ProductName)) AS num\_products\_viewed,
        + ARRAY\_AGG(pageTitle) AS pages\_viewed
        + ARRAY\_LENGTH(ARRAY\_AGG(pageTitle)) AS num\_pages\_viewed
        + FROM `data-to-insights.ecommerce.all\_sessions`
        + WHERE visitId = 1501570398 GROUP BY fullVisitorId, date ORDER BY date
      * How many pages were visited by this user on 20170801? 109
      * Graphical user interface

        Description automatically generated with medium confidence
    - lets deduplicate the pages and products so we can see how many unique products were viewed. We'll simply add DISTINCT to our ARRAY\_AGG()
      * + SELECT fullVisitorId, date,
        + ARRAY\_AGG(DISTINCT v2ProductName) AS products\_viewed,
        + ARRAY\_LENGTH(ARRAY\_AGG(DISTINCT v2ProductName)) AS distinct\_products\_viewed, ARRAY\_AGG(DISTINCT pageTitle) AS pages\_viewed,
        + ARRAY\_LENGTH(ARRAY\_AGG(DISTINCT pageTitle)) AS distinct\_pages\_viewed
        + FROM `data-to-insights.ecommerce.all\_sessions`
        + WHERE visitId = 1501570398 GROUP BY fullVisitorId, date ORDER BY date
      * How many DISTINCT pages were visited by this user on 20170801? 8
      * A picture containing application

        Description automatically generated
    - Recap
      * You can do some pretty useful things with arrays like:
        + finding the number of elements with ARRAY\_LENGTH(<array>)
        + deduplicating elements with ARRAY\_AGG(DISTINCT <field>)
        + ordering elements with ARRAY\_AGG(<field> ORDER BY <field>)
        + limiting ARRAY\_AGG(<field> LIMIT 5)
  + Querying datasets that already have ARRAYs
    - The BigQuery Public Dataset for Google Analytics bigquery-public-data.google\_analytics\_sample has many more fields and rows than our course dataset data-to-insights.ecommerce.all\_sessions. More importantly, it already stores field values like products, pages, and transactions natively as ARRAYs.
    - query to explore the available data and see if you can find fields with repeated values (arrays)
      * + SELECT \*
        + FROM `bigquery-public-data.google\_analytics\_sample.ga\_sessions\_20170801`
        + WHERE visitId = 1501570398
      * Scroll right in the results until you see the hits.product.v2ProductName field (we will discuss the multiple field aliases shortly).
      * 
      * Text

        Description automatically generated with low confidence
      * Graphical user interface, text, application, email

        Description automatically generated
    - You will notice a lot of seemingly 'empty' cells in the results as you scroll. These cells are grayed out and not marked as null. Why do you think that is?
      * The grayed out cells are visual placeholders to make it possible to show each item in an array type column on its own row within the context of a row in the result set
    - The amount of fields available in the Google Analytics schema can be overwhelming for our analysis. Let's try to query just the visit and page name fields like we did before.
      * + SELECT visitId, hits.page.pageTitle
        + FROM `bigquery-public-data.google\_analytics\_sample.ga\_sessions\_20170801`
        + WHERE visitId = 1501570398
      * You will get an error: Cannot access field product on a value with type ARRAY> at [5:8]
      * Before we can query REPEATED fields (arrays) normally, you must first break the arrays back into rows.
      * For example, the array for hits.page.pageTitle is stored currently as a single row like:
        + ['homepage','product page','checkout']
      * and we need it to be
        + ['homepage',
        + 'product page',
        + 'checkout']
    - How do we do that with SQL? Answer: Use the UNNEST() function on your array field:
      * + SELECT DISTINCT visitId, h.page.pageTitle
        + FROM `bigquery-public-data.google\_analytics\_sample.ga\_sessions\_20170801`,
        + UNNEST(hits) AS h
        + WHERE visitId = 1501570398 LIMIT 10
      * We'll cover UNNEST() more in detail later but for now just know that:
        + You need to UNNEST() arrays to bring the array elements back into rows
        + UNNEST() always follows the table name in your FROM clause (think of it conceptually like a pre-joined table)
        + Text, table

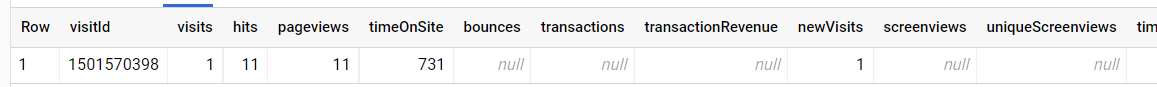
          Description automatically generated
  + Introduction to STRUCTs
    - You may have wondered why the field alias hit.page.pageTitle looks like three fields in one separated by periods. Just as ARRAY values give you the flexibility to go deep into the granularity of your fields, another data type allows you to go wide in your schema by grouping related fields together. That SQL data type is the STRUCT data type.
    - The easiest way to think about a STRUCT is to consider it conceptually like a separate table that is already pre-joined into your main table.
    - A STRUCT can have:
      * one or many fields in it
      * the same or different data types for each field
      * it's own alias
    - Let's explore a dataset with STRUCTs
      * Under Explorer find the bigquery-public-data dataset (if it's not present already, use this link <https://console.cloud.google.com/bigquery?p=bigquery-public-data&d=google_analytics_sample&t=ga_sessions_20170801&page=table> to pin the dataset)
      * Click bigquery-public-data > Find and open google\_analytics\_sample
      * Click the ga\_sessions table
      * Start scrolling through the schema and answer the following question by using the find feature of your browser (i.e. CTRL + F)
      * In a BigQuery schema, a STRUCT field is noted as a RECORD Type. Search for RECORD in the Google Analytics schema. How many STRUCTs are present in this dataset? 32
      * What are the names of some of the STRUCT (RECORD Type) fields?
        + Totals, TrafficSource , trafficSource.adwordsClickInfo - All of the above
      * How can both TrafficSource and trafficSource.adwordsClickInfo both be STRUCTs?
        + A STRUCT can have another STRUCT as one of its fields (you can nest STRUCTs)
      * In a BigQuery schema, an ARRAY field is noted as a REPEATED Mode. Search for REPEATED in the Google Analytics schema. How many ARRAYs are present in this dataset? 11
      * As you can imagine, there is an incredible amount of website session data stored for a modern ecommerce website. The main advantage of having 32 STRUCTs in a single table is it allows you to run queries like this one without having to do any JOINs:

SELECT visitId, totals.\*, device.\*

FROM `bigquery-public-data.google\_analytics\_sample.ga\_sessions\_20170801`

WHERE visitId = 1501570398

LIMIT 10

* + - 
      * + Note: The .\* syntax tells BigQuery to return all fields for that STRUCT (much like it would if totals.\* was a separate table we joined against)
        + Storing your large reporting tables as STRUCTs (pre-joined "tables") and ARRAYs (deep granularity) allows you to:

gain significant performance advantages by avoiding 32 table JOINs

get granular data from ARRAYs when you need it but not be punished if you don't (BigQuery stores each column individually on disk)

have all the business context in one table as opposed to worrying about JOIN keys and which tables have the data you need

* + Practice with STRUCTs and ARRAYs
    - The next dataset will be lap times of runners around the track. Each lap will be called a "split".
    - With this query, try out the STRUCT syntax and note the different field types within the struct container:
      * SELECT STRUCT("Rudisha" as name, 23.4 as split) as runner

|  |  |  |
| --- | --- | --- |
| **Row** | **runner.name** | **runner.split** |
| 1 | Rudisha | 23.4 |

* + - What do you notice about the field aliases? Since there are fields nested within the struct (name and split are a subset of runner) you end up with a dot notation.
    - What if the runner has multiple split times for a single race (like time per lap)?
    - How could you have multiple split times within a single record? Hint: the splits all have the same numeric datatype.
      * Store each split time as an element in an ARRAY of splits
    - With an array of course! Run the below query to confirm:
      * SELECT STRUCT("Rudisha" as name, [23.4, 26.3, 26.4, 26.1] as splits) AS runner

|  |  |  |
| --- | --- | --- |
| **Row** | **runner.name** | **runner.splits** |
| 1 | Rudisha | 23.4 |
| 26.3 |
| 26.4 |
| 26.1 |

* + - To recap:
      * Structs are containers that can have multiple field names and data types nested inside.
      * An arrays can be one of the field types inside of a Struct (as shown above with the splits field).
  + Practice ingesting JSON data
    - Create a new dataset titled racing.
    - Create a new table titled race\_results.
    - Ingest this Google Cloud Storage JSON file:
      * data-insights-course/labs/optimizing-for-performance/race\_results.json
    - Source: Google Cloud Storage under Create table from dropdown.
    - Select file from GCS bucket: data-insights-course/labs/optimizing-for-performance/race\_results.json
    - File format: JSONL (Newline delimited JSON)
    - In Schema, move the Edit as text slider and add the following:
      * [
      * {
      * "name": "race",
      * "type": "STRING",
      * "mode": "NULLABLE"
      * },
      * {
      * "name": "participants",
      * "type": "RECORD",
      * "mode": "REPEATED",
      * "fields": [
      * {
      * "name": "name",
      * "type": "STRING",
      * "mode": "NULLABLE"
      * },
      * {
      * "name": "splits",
      * "type": "FLOAT",
      * "mode": "REPEATED"
      * }
      * ]
      * }
      * ]
    - Click Create table.
    - After the load job is successful, preview the schema for the newly created table:
      * Graphical user interface, text, application, email

        Description automatically generated
    - Which field is the STRUCT? How do you know?
      * The participants field is the STRUCT because it is of type RECORD
    - Which field is the ARRAY?
      * The participants.splits field is an array of floats inside of the parent participants struct. It has a REPEATED Mode which indicates an array. Values of that array are called nested values since they are multiple values inside of a single field.
    - Practice querying nested and repeated fields
      * Let's see all of our racers for the 800 Meter race.

SELECT \* FROM racing.race\_results

* + - * + How many rows were returned? 1

Table

Description automatically generated

* + - * What if you wanted to list the name of each runner and the type of race?

SELECT race, participants.name FROM racing.race\_results

* + - * + Error: Cannot access field name on a value with type ARRAY\<STRUCT\<name STRING, splits ARRAY\<FLOAT64\>>>> at [1:21]
      * Much like forgetting to GROUP BY when you use aggregation functions, here there are two different levels of granularity. One row for the race and three rows for the participants names. So how do you change this...

|  |  |  |
| --- | --- | --- |
| **Row** | **race** | **participants.name** |
| 1 | 800M | Rudisha |
| 2 | ??? | Makhloufi |
| 3 | ??? | Murphy |

* + - * ...to this:

|  |  |  |
| --- | --- | --- |
| **Row** | **race** | **participants.name** |
| 1 | 800M | Rudisha |
| 2 | 800M | Makhloufi |
| 3 | 800M | Murphy |

* + - * In traditional relational SQL, if you had a races table and a participants table what would you do to get information from both tables? You would JOIN them together. Here the participant STRUCT (which is conceptually very similar to a table) is already part of your races table but is not yet correlated correctly with your non-STRUCT field "race".
      * Can you think of what two word SQL command you would use to correlate the 800M race with each of the racers in the first table? Answer: CROSS JOIN
      * Great! Now try running this:
        + SELECT race, participants.name FROM racing.race\_results CROSS JOIN participants # this is the STRUCT (it's like a table within a table)
      * Error: Table name "participants" cannot be resolved: dataset name is missing.
      * Even though the participants STRUCT is like a table, it is still technically a field in the racing.race\_results table.
      * Add the dataset name to the query:
        + SELECT race, participants.name FROM racing.race\_results CROSS JOIN race\_results.participants # full STRUCT name
      * Wow! You've successfully listed all of the racers for each race!

|  |  |  |
| --- | --- | --- |
| **Row** | **race** | **name** |
| 1 | 800M | Rudisha |
| 2 | 800M | Makhloufi |
| 3 | 800M | Murphy |
| 4 | 800M | Bosse |
| 5 | 800M | Rotich |
| 6 | 800M | Lewandowski |
| 7 | 800M | Kipketer |
| 8 | 800M | Berian |

* + - You can simplify the last query by:
      * Adding an alias for the original table
      * Replacing the words "CROSS JOIN" with a comma (a comma implicitly cross joins)
    - This will give you the same query result:
      * SELECT race, participants.name FROM racing.race\_results AS r, r.participants
    - If you have more than one race type (800M, 100M, 200M), wouldn't a CROSS JOIN just associate every racer name with every possible race like a cartesian product?
      * Answer: No. This is a correlated cross join which only unpacks the elements associated with a single row. For a greater discussion, see working with ARRAYs and STRUCTs
    - Recap of STRUCTs:
      * A SQL STRUCT is simply a container of other data fields which can be of different data types. The word struct means data structure. Recall the example from earlier:
      * \_\_STRUCT(\_\_"Rudisha" as name, [23.4, 26.3, 26.4, 26.1] as splits\_\_)\_\_AS runner
      * STRUCTs are given an alias (like runner above) and can conceptually be thought of as a table inside of your main table.
      * STRUCTs (and ARRAYs) must be unpacked before you can operate over their elements. Wrap an UNNEST() around the name of the struct itself or the struct field that is an array in order to unpack and flatten it.
  + Lab Question: STRUCT()
    - Answer the below questions using the racing.race\_results table you created previously.
    - Task: Write a query to COUNT how many racers were there in total.
    - To start, use the below partially written query:
      * SELECT COUNT(participants.name) AS racer\_count FROM racing.race\_results
    - Hint: need to cross join in your struct name as an additional data source after the FROM.
    - Possible Solution:
      * SELECT COUNT(p.name) AS racer\_count FROM racing.race\_results AS r, UNNEST(r.participants) AS p

|  |  |
| --- | --- |
| **Row** | **racer\_count** |
| 1 | 8 |

* + - * Answer: There were 8 racers who ran the race.
  + Lab Question: Unpacking ARRAYs with UNNEST( )
    - Write a query that will list the total race time for racers whose names begin with R. Order the results with the fastest total time first. Use the UNNEST() operator and start with the partially written query below.
    - Complete the query:
      * SELECT p.name, SUM(split\_times) as total\_race\_time FROM racing.race\_results AS r
      * , r.participants AS p , p.splits AS split\_times
      * WHERE GROUP BY ORDER BY ;
    - Hint:
      * You will need to unpack both the struct and the array within the struct as data sources after your FROM clause
      * Be sure to use aliases where appropriate
    - Possible Solution:
      * SELECT p.name, SUM(split\_times) as total\_race\_time
      * FROM racing.race\_results AS r
      * , UNNEST(r.participants) AS p , UNNEST(p.splits) AS split\_times
      * WHERE p.name LIKE 'R%' GROUP BY p.name ORDER BY total\_race\_time ASC;

|  |  |  |
| --- | --- | --- |
| **Row** | **name** | **total\_race\_time** |
| 1 | Rudisha | 102.19999999999999 |
| 2 | Rotich | 103.6 |

* + Lab Question: Filtering within ARRAY values
    - You happened to see that the fastest lap time recorded for the 800 M race was 23.2 seconds, but you did not see which runner ran that particular lap. Create a query that returns that result.
    - Task: Complete the partially written query:
      * SELECT p.name, split\_time FROM racing.race\_results AS r
      * , r.participants AS p , p.splits AS split\_time WHERE split\_time = ;
    - Possible Solution:
      * SELECT p.name, split\_time FROM racing.race\_results AS r
      * , UNNEST(r.participants) AS p , UNNEST(p.splits) AS split\_time
      * WHERE split\_time = 23.2;

|  |  |  |
| --- | --- | --- |
| **Row** | **name** | **split\_time** |
| 1 | Kipketer | 23.2 |

* Optimize with partitioning and clustering 7 minutes - <https://youtu.be/wu_wCs99vgc>
* Quiz: Building a Data Warehouse
  + True or False:
  + ARRAYS can be part of regular fields or STRUCTS in BigQuery?
    - True
  + Which of the following statements on BigQuery is incorrect?
    - The number of slots allotted to a query is independent of query complexity