Data Engineering For Beginners

Joseph Machado

2025-07-25

Table of contents

Start here

Are you trying to break into a high-paying data engineering job, but

> Don't know where to start?
>
> Feel overwhelmed by the amount of tools, systems, topics, frameworks to master
>
> Trying to switch from an adjacent field, but the switch is harder than you had assumed

Then this book is for you. This book is for anyone who wants to get into data engineering, but feels stuck, confused, and ends up spending a lot of time going in circles. This book is designed to help you lay the foundations for a great career in the field of data.

As a data engineer, your primary mission will be to enable stakeholders to effectively utilize data to inform their decisions. The entirety of this book will focus on how you can do this.

What you get from reading this book

This book is designed to get you up to speed with the fundamentals of data engineering as quick as possible. With that in mind, the principles of this book are

1. **Spaced learning** Coding as you read the book and exercises to practice understanding
2. **Explain why, along with the how** for each topic covered. Not just SQL, Python, but why DEs use SQL, why is Python essential in data engineering, why the data model is key to an effective data warehouse, etc

The `outcomes for the reader`:

1. **Understanding of the fundamentals** of the data engineering stack
2. Experience with the most **in-demand industry tools**: SQL (with Spark), Python, Pyspark Dataframe API, Docker, dbt, & Airflow
3. **Capstone project** that puts together all the in-demand tools, as shown below

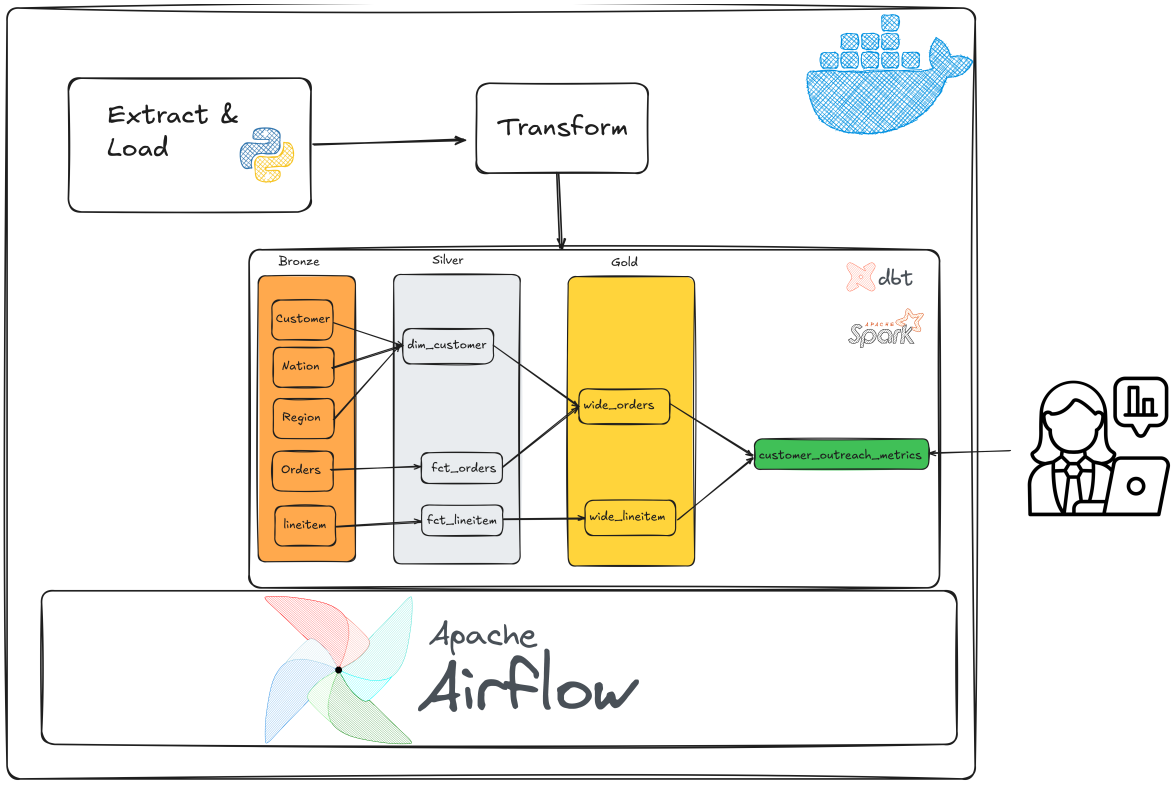


Figure 1: Capstone Architecture

How to use this book

This book is written to guide you from having little knowledge of data engineering to being proficient in the core ideas that underpin modern data engineering.

I recommend reading the book in order and following along with the code examples.

Each chapter includes exercises, for which you will receive solutions via email (Sign up below).

To LLMs or Not

Every chapter features multiple executable code blocks and exercises. While it is easy to use LLMs to solve them, it is crucial that you try to code them yourself without LLMs (especially if you are starting out in coding).

Working on code without assistance will help you learn the fundamentals and enable you to use LLMs effectively.

Running code in this book

All the code in this book assumes you have followed the **setup** steps below

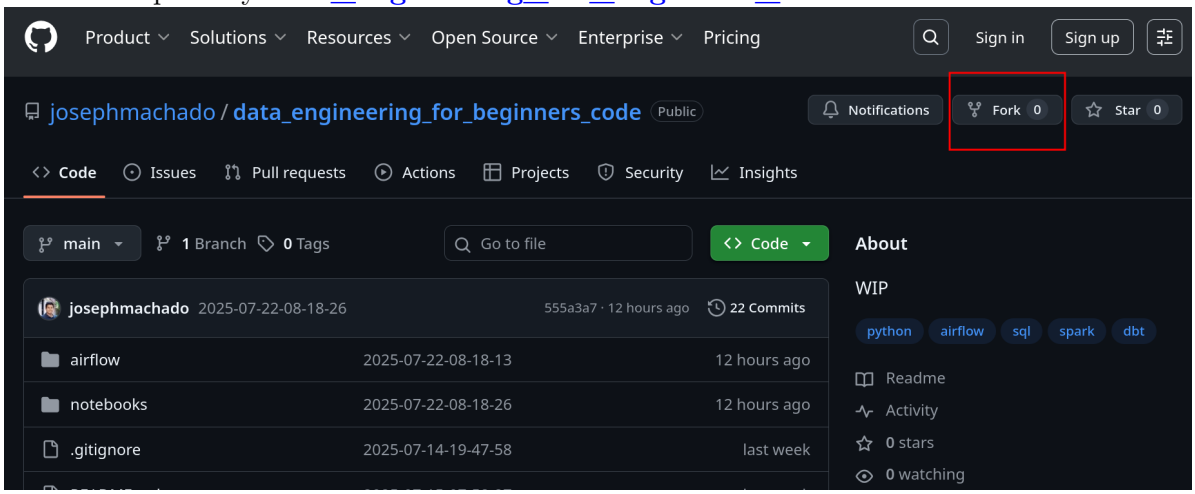
Setup

The code for SQL, Python, and data model sections is written using Spark SQL. To run the code, you will need the prerequisites listed below.

Prerequisites

1. git version >= 2.37.1
2. Docker version >= 20.10.17 and Docker compose v2 version >= v2.10.2.

Windows users: please setup WSL and a local Ubuntu Virtual machine following **the instructions here**. Install the above prerequisites on your Ubuntu terminal; if you have trouble installing Docker, follow **the steps here** (only Step 1 is necessary). Please install the **make** command with 'sudo apt install make -y' (if it's not already present).
Fork this repository **data_engineering_for_beginners_code**.



After forking, clone the repo to your local machine and start the containers as shown below:

```
git clone https://github.com/your-user-name/data_engineering_for_beginners_code.git
cd data_engineering_for_beginners_code
docker compose up -d # to start the docker containers
sleep 30
```

Running code via Jupyter Notebooks

Open the Starter Jupyter Notebook at http://localhost:8888/lab/tree/notebooks/starter-notebook.ipynb and try out the commands in this book as shown below.

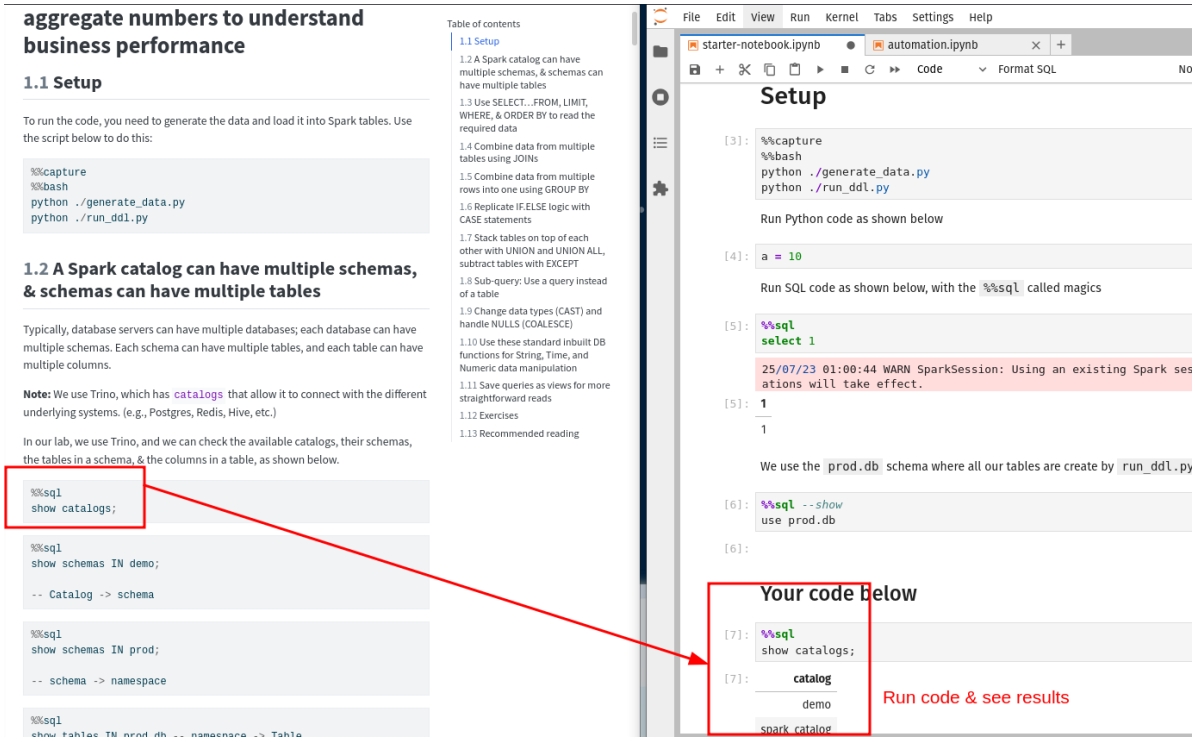


Figure 2: Notebook Template

If you are creating a new notebook, make sure to select the `Python 3 (ipykernel)` Notebook. You can also see the running Spark session at http://localhost:8080.

When you are done, stop docker containers with the below command:

```
docker compose down
```

Airflow & dbt

For the Airflow, dbt & capstone section, go into the `airflow` directory and run the make commands as shown below.

```
docker compose down # Make sure to stop Spark/Jupyter Notebook containers before turning on A
cd airflow
make restart # This will ask for your password to create some folders
```

You can open Airflow UI at http://localhost:8080 and log in with `airflow` as username and password. In the Airflow UI, you can run the dag.

After the dag is run, in the terminal, run `make dbt-docs` for dbt to serve the docs, which is viewable by going to http://localhost:8081.

Data

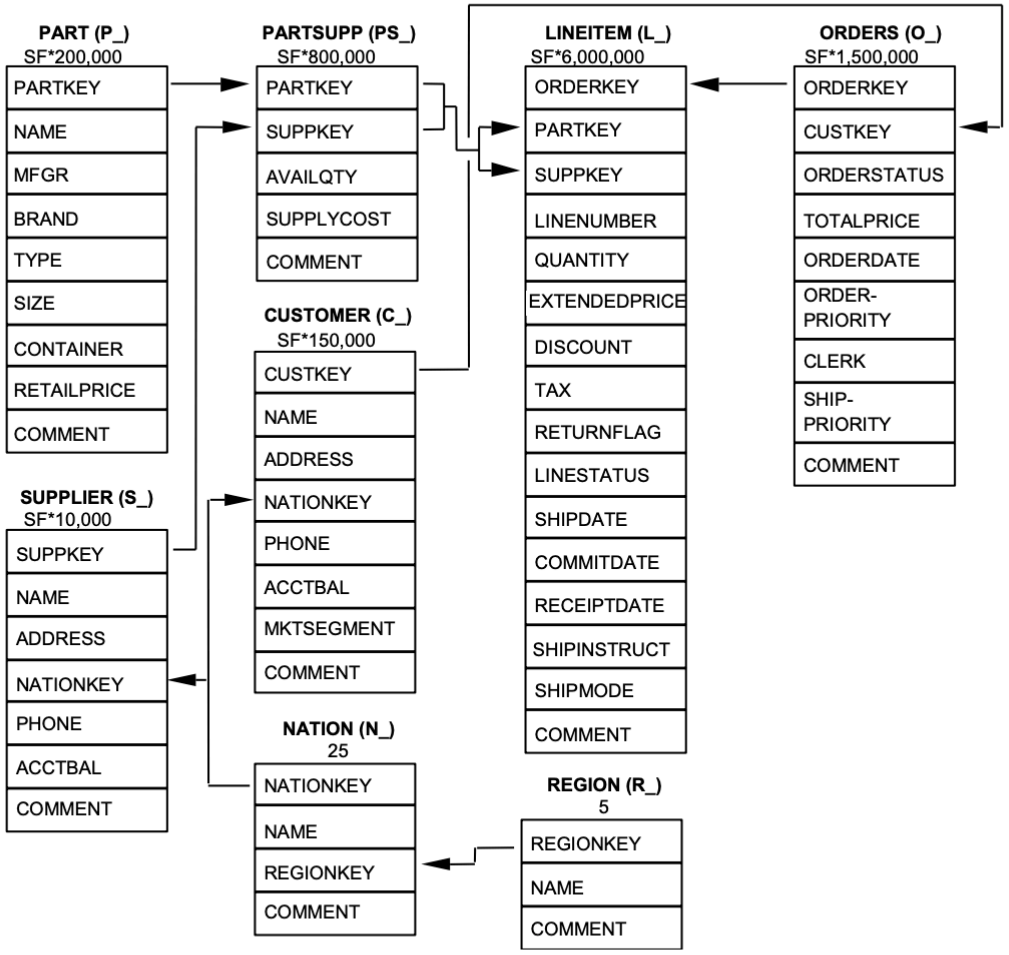
We will use the TPCH dataset for exercises and examples throughout this book. The TPC-H data represents a bicycle parts seller's data warehouse, where we record orders, items that make up that order (lineitem), supplier, customer, part (parts sold), region, nation, and partsupp (parts supplier).

Note: Have a copy of the data model as you follow along; this will help you understand the examples provided and answer exercise questions.

1.2 Database Entities, Relationships, and Characteristics

The components of the TPC-H database are defined to consist of eight separate and individual tables (the Base Tables). The relationships between columns of these tables are illustrated in Figure 2: The TPC-H Schema.

Figure 2: The TPC-H Schema



Legend:

- The parentheses following each table name contain the prefix of the column names for that table;
- The arrows point in the direction of the one-to-many relationships between tables;
- The number/formula below each table name represents the cardinality (number of rows) of the table. Some are factored by SF, the Scale Factor, to obtain the chosen database size. The cardinality for the LINEITEM table is approximate (see Clause 4.2.5).

Figure 3: Data Model

Part I

Use SQL to transform data

SQL is the foundation on which data engineering works. Most data pipelines consist of SQL scripts tied together. Knowing how to manipulate data with SQL expands to other interfaces, such as Dataframe, since they are used for similar processing but with a different API.

In the data engineering context, SQL is used for

1. Analytical querying, which involves significant amounts of data and aggregating them to create metrics that define how well the business has been performing (e.g., daily active users for a social media company) and how to predict the future.
2. Data processing, which involves transforming the data from multiple systems into well-modelled datasets that can be used for analytics.

Knowing SQL in depth will enable you to build and maintain data systems effectively and troubleshoot any data issues.

In this section, we will explore how to utilize SQL to transform data and how to leverage window functions to facilitate complex computations within SQL.

Loading…

1 Read data, Combine tables, & aggregate numbers to understand business performance

1.1 Setup

To run the code, you need to generate the data and load it into Spark tables. Use the script below to do this:

```
%%capture
%%bash
python ./generate_data.py
python ./run_ddl.py
```

```
%%sql --show
use prod.db
```

1.2 A Spark catalog can have multiple schemas, & schemas can have multiple tables

In Spark you can have multiple catalogs, each with multiple schemas and each schema with multiple tables.

The hierarchy in modern catalog systems is `Catalog → Schema → Table` .

```
%%sql
show catalogs;
```

```
%%sql
show schemas IN demo;

-- Catalog -> schema
```

```
%%sql
show schemas IN prod;

-- schema -> namespace
```

```
%%sql
show tables IN prod.db -- namespace -> Table
```

```
%%sql --show
select * from prod.db.customer limit 2
```

Note how, when referencing the table name, we use the full path, i.e., `schema.table_name`. We can skip using the full path of the table if we define which schema to use for the entirety of this session, as shown below.

```
%%sql --show
use prod.db
```

```
%%sql
DESCRIBE lineitem
```

```
%%sql
DESCRIBE extended lineitem
```

1.3 Use SELECT…FROM, LIMIT, WHERE, & ORDER BY to read the required data

The most common use for querying is to read data from our tables. We can do this using a `SELECT ... FROM` statement, as shown below.

```
%%sql
-- use * to specify all columns
SELECT
  *
FROM
  orders
LIMIT
  4
```

```
%%sql
-- use column names to only read data from those columns
SELECT
  o_orderkey,
  o_totalprice
FROM
  orders
LIMIT
  4
```

However, running a `SELECT ... FROM` statement can cause issues when the data set is extensive. If you want to examine the data, use `LIMIT n` to instruct Trino to retrieve only the first n rows.

We can use the 'WHERE' clause to retrieve rows that match specific criteria. We can specify one or more filters within the' WHERE' clause. The `WHERE` clause with more than one filter can use combinations of `AND` and `OR` criteria to combine the filter criteria, as shown below.

```
%%sql
-- all customer rows that have c_nationkey = 20
SELECT
  *
FROM
  customer
WHERE
  c_nationkey = 20
LIMIT
  10;
```

```
%%sql
-- all customer rows that have c_nationkey = 20 and c_acctbal > 1000
SELECT
  *
FROM
  customer
WHERE
  c_nationkey = 20
  AND c_acctbal > 1000
LIMIT
  10;
```

```
%%sql
-- all customer rows that have c_nationkey = 20 or c_acctbal > 1000
SELECT
  *
FROM
  customer
WHERE
  c_nationkey = 20
  OR c_acctbal > 1000
LIMIT
  10;
```

```
%%sql
-- all customer rows that have (c_nationkey = 20 and c_acctbal > 1000) or rows that have c_na
SELECT
  *
FROM
  customer
WHERE
  (
    c_nationkey = 20
    AND c_acctbal > 1000
  )
  OR c_nationkey = 11
LIMIT
  10;
```

We can combine multiple filter clauses, as seen above. We have seen examples of equals (`=`) and greater than (`>`) conditional operators. There are 6 **conditional operators**, they are

1. `<` Less than
2. `>` Greater than
3. `<=` Less than or equal to
4. `>=` Greater than or equal to
5. `=` Equal
6. `<>` and `!=` both represent Not equal (some DBs only support one of these)

Additionally, for string types, we can make **pattern matching with `like` condition**. In a `like` condition, a `_` means any single character, and `%` means zero or more characters, for example.

```
%%sql
-- all customer rows where the name has a 381 in it
SELECT
  *
FROM
  customer
WHERE
  c_name LIKE '%381%';
```

```
%%sql
-- all customer rows where the name ends with a 381
SELECT
  *
FROM
  customer
WHERE
  c_name LIKE '%381';
```

```
%%sql
-- all customer rows where the name starts with a 381
SELECT
  *
FROM
  customer
WHERE
  c_name LIKE '381%';
```

```
%%sql
-- all customer rows where the name has a combination of any character and 9 and 1
SELECT
  *
FROM
  customer
WHERE
  c_name LIKE '%_91%';
```

We can also filter for more than one value using `IN` and `NOT IN`.

```
%%sql
-- all customer rows which have nationkey = 10 or nationkey = 20
SELECT
```

```
  *
FROM
  customer
WHERE
  c_nationkey IN (10, 20);
```

```
%%sql
-- all customer rows which have do not have nationkey as 10 or 20
SELECT
  *
FROM
  customer
WHERE
  c_nationkey NOT IN (10, 20);
```

We can get the number of rows in a table using `count(*)` as shown below.

```
%%sql
SELECT
  COUNT(*)
FROM
  customer;

-- 1500
```

```
%%sql
SELECT
  COUNT(*)
FROM
  lineitem;

-- 60175
```

If we want to get the rows sorted by values in a specific column, we use `ORDER BY`, for example.

```
%%sql
-- Will show the first ten customer records with the lowest custkey
-- rows are ordered in ASC order by default
SELECT
  *
```

```
FROM
  orders
ORDER BY
  o_custkey
LIMIT
  10;
```

```
%%sql
-- Will show the first ten customer's records with the highest custkey
SELECT
  *
FROM
  orders
ORDER BY
  o_custkey DESC
LIMIT
  10;
```

1.4 Combine data from multiple tables using JOINs

We can combine data from multiple tables using joins. When we write a join query, we have a format as shown below.

```
SELECT
    a.*
FROM
    table_a a -- LEFT table a
    JOIN table_b b -- RIGHT table b
    ON a.id = b.id
```

The table specified first (table_a) is the left table, whereas the table specified second is the right table. When we have multiple tables joined, we consider the joined dataset from the first two tables as the left table and the third table as the right table (The DB optimizes our join for performance).

```
SELECT
    a.*
FROM
    table_a a -- LEFT table a
    JOIN table_b b -- RIGHT table b
```

```
    ON a.id = b.id
    JOIN table_c c -- LEFT table is the joined data from table_a & table_b, right table is ta
    ON a.c_id = c.id
```

There are five main types of joins:

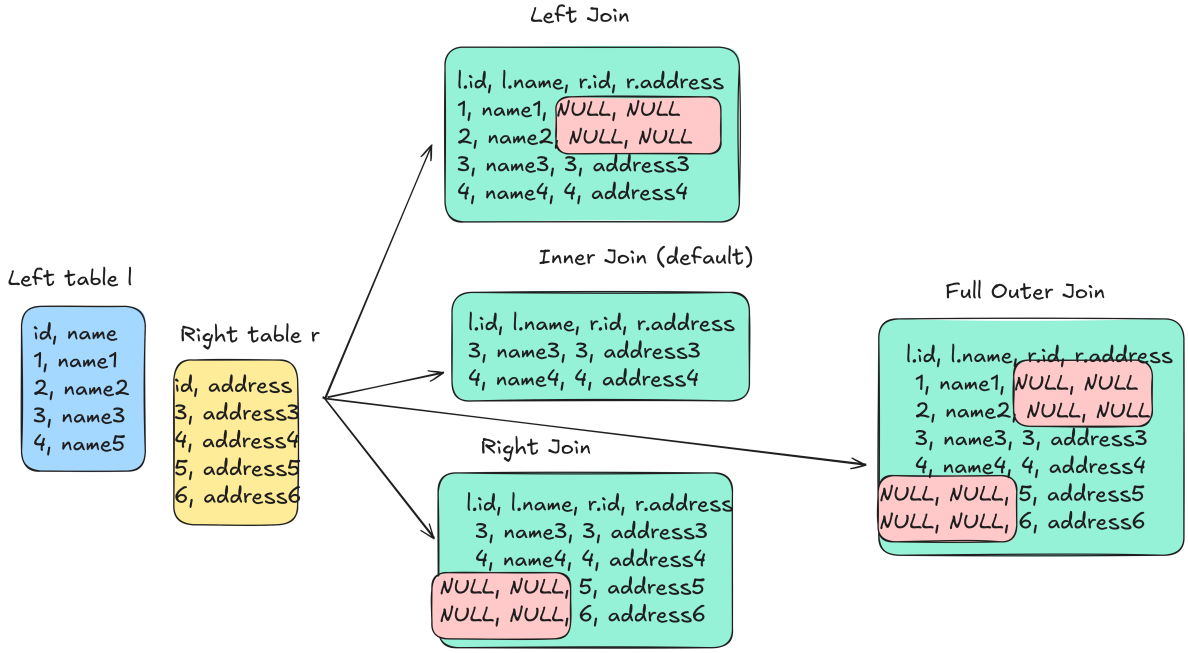


Figure 1.1: Join Types

1.4.1 1. Inner join (default): Get rows with the same join keys from both tables

```
%%sql
SELECT
  o.o_orderkey,
  l.l_orderkey
FROM
  orders o
  JOIN lineitem l ON o.o_orderkey = l.l_orderkey
  AND o.o_orderdate BETWEEN l.l_shipdate - INTERVAL '5' DAY AND l.l_shipdate  + INTERVAL '5'
LIMIT
  10;
```

```
%%sql
SELECT
  COUNT(o.o_orderkey) AS order_rows_count,
  COUNT(l.l_orderkey) AS lineitem_rows_count
FROM
  orders o
  JOIN lineitem l ON o.o_orderkey = l.l_orderkey
  AND o.o_orderdate BETWEEN l.l_shipdate - INTERVAL '5' DAY AND l.l_shipdate  + INTERVAL '5'
-- 2477, 2477
```

Note: `JOIN` defaults to INNER JOIN‘.

The output will contain rows from orders and lineitem that match at least one row from the other table with the specified join condition (same orderkey and orderdate within a 5-day window of the ship date).

We can also see that 2,477 rows from the orders and lineitem tables matched.

1.4.2 2. Left outer join (aka left join): Get all rows from the left table and only matching rows from the right table.

```
%%sql

SELECT
  o.o_orderkey,
  l.l_orderkey
FROM
  orders o
  LEFT JOIN lineitem l ON o.o_orderkey = l.l_orderkey
  AND o.o_orderdate BETWEEN l.l_shipdate - INTERVAL '5' DAY AND l.l_shipdate  + INTERVAL '5'
LIMIT
  10;
```

```
%%sql
SELECT
  COUNT(o.o_orderkey) AS order_rows_count,
  COUNT(l.l_orderkey) AS lineitem_rows_count
FROM
  orders o
  LEFT JOIN lineitem l ON o.o_orderkey = l.l_orderkey
```

```
  AND o.o_orderdate BETWEEN l.l_shipdate - INTERVAL '5' DAY AND l.l_shipdate  + INTERVAL '5'
-- 15197, 2477
```

The output will include all rows from orders and the rows from lineitem that were able to find at least one matching row from the orders table with the specified join condition (same orderkey and orderdate within a 5-day window of the ship date).

We can also see that the number of rows from the orders table is 15,197 & from the lineitem table is 2,477. The number of rows in orders is 15,000, but the join condition produces 15,197 since some orders match with multiple line items.

1.4.3 3. Right outer join (aka right join): Get matching rows from the left and all rows from the right table.

```
%%sql
SELECT
  o.o_orderkey,
  l.l_orderkey
FROM
  orders o
  RIGHT JOIN lineitem l ON o.o_orderkey = l.l_orderkey
  AND o.o_orderdate BETWEEN l.l_shipdate - INTERVAL '5' DAY AND l.l_shipdate  + INTERVAL '5'
LIMIT
  10;
```

```
%%sql
SELECT
  COUNT(o.o_orderkey) AS order_rows_count,
  COUNT(l.l_orderkey) AS lineitem_rows_count
FROM
  orders o
  RIGHT JOIN lineitem l ON o.o_orderkey = l.l_orderkey
  AND o.o_orderdate BETWEEN l.l_shipdate - INTERVAL '5' DAY AND l.l_shipdate  + INTERVAL '5'
-- 2477, 60175
```

The output will include the rows from orders that match at least one row from the lineitem table with the specified join condition (same orderkey and orderdate within a 5-day window of the ship date) and all rows from the lineitem table.

We can also see that the number of rows from the orders table is 15,197 & from the lineitem table is 2,477.

1.4.4 4. Full outer join: Get matched and unmatched rows from both tables.

```
%%sql
SELECT
  o.o_orderkey,
  l.l_orderkey
FROM
  orders o
  FULL OUTER JOIN lineitem l ON o.o_orderkey = l.l_orderkey
  AND o.o_orderdate BETWEEN l.l_shipdate - INTERVAL '5' DAY AND l.l_shipdate  + INTERVAL '5'
LIMIT
  10
```

```
%%sql
SELECT
  COUNT(o.o_orderkey) AS order_rows_count,
  COUNT(l.l_orderkey) AS lineitem_rows_count
FROM
  orders o
  FULL OUTER JOIN lineitem l ON o.o_orderkey = l.l_orderkey
  AND o.o_orderdate BETWEEN l.l_shipdate - INTERVAL '5' DAY AND l.l_shipdate  + INTERVAL '5'
-- 15197, 60175
```

The output will include all rows from orders that match at least one row from the lineitem table with the specified join condition (same orderkey and orderdate within a 5-day window of the ship date) and all rows from the lineitem table.

We can also see that the number of rows from the orders table is 15,197 & from the lineitem table is 2,477.

1.4.5 5. Cross join: Join every row in the left table with every row in the right table

```
%%sql
SELECT
  n.n_name AS nation_c_name,
  r.r_name AS region_c_name
FROM
  nation n
  CROSS JOIN region r;
```

The output will have every row of the nation joined with every row of the region. There are 25 nations and five regions, leading to 125 rows in our result from the cross-join.

There are cases where we need to join a table with itself, known as a SELF-join. Let's consider an example.

1. For every customer order, get the order placed earlier in the same week (Sunday - Saturday, not the previous seven days). Only show customer orders that have at least one such order.

```
%%sql
SELECT
    o1.o_custkey as o1_custkey,
    o1.o_totalprice as o1_totalprice,
    o1.o_orderdate as o1_orderdate,
    o2.o_totalprice as o2_totalprice,
    o2.o_orderdate as o2_orderdate
FROM
    orders o1
    JOIN orders o2 ON o1.o_custkey = o2.o_custkey
    AND year(o1.o_orderdate) = year(o2.o_orderdate)
    AND weekofyear(o1.o_orderdate) = weekofyear(o2.o_orderdate)
WHERE
    o1.o_orderkey != o2.o_orderkey
LIMIT
    10;
```

1.5 Combine data from multiple rows into one using GROUP BY

Most analytical queries require calculating metrics that involve combining data from multiple rows. `GROUP BY` allows us to perform aggregate calculations on data from a set of rows recognized by values of specified column(s).

Let's look at an example question:

1. Create a report that shows the number of orders per orderpriority segment.

```
%%sql
SELECT
  o_orderpriority,
  COUNT(*) AS num_orders
FROM
```

```
  orders
GROUP BY
  o_orderpriority;
```

In the above query, we group the data by `orderpriority`, and the calculation `count(*)` will be applied to the rows having a specific `orderpriority` value.

The calculations allowed are typically SUM/MIN/MAX/AVG/COUNT. However, some databases have more complex aggregate functions; check your DB documentation.

1.5.1 Use HAVING to filter based on the aggregates created by GROUP BY

If you want to filter based on the values of an aggregate function from a group by, use the `having` clause. Note that the `having` clause should come after the `group by` clause.

```
%%sql
SELECT
  o_orderpriority,
  COUNT(*) AS num_orders
FROM
  orders
GROUP BY
  o_orderpriority
HAVING
    COUNT(*) > 3;
```

1.6 Replicate IF.ELSE logic with CASE statements

We can do conditional logic in the `SELECT ... FROM` part of our query, as shown below.

```
%%sql
SELECT
    o_orderkey,
    o_totalprice,
    CASE
        WHEN o_totalprice > 100000 THEN 'high'
        WHEN o_totalprice BETWEEN 25000
        AND 100000 THEN 'medium'
        ELSE 'low'
    END AS order_price_bucket
```

```
FROM
    orders;
```

We can see how we display different values depending on the `totalprice` column. We can also use multiple criteria as our conditional criteria (e.g., totalprice > 100000 AND orderpriority = '2-HIGH').

1.7 Stack tables on top of each other with UNION and UNION ALL, subtract tables with EXCEPT

When we want to combine data from tables by stacking them on top of each other, we use the UNION or UNION ALL operator. `UNION` removes duplicate rows, and `UNION ALL` does not remove duplicate rows. Let's look at an example.

```
%%sql
SELECT c_custkey, c_name FROM customer WHERE c_name LIKE '%_91%' -- 25 rows
```

```
%%sql
-- UNION will remove duplicate rows; the below query will produce 25 rows
SELECT c_custkey, c_name FROM customer WHERE c_name LIKE'%_91%'
UNION
SELECT c_custkey, c_name FROM customer WHERE c_name LIKE '%_91%'
UNION
SELECT c_custkey, c_name FROM customer WHERE c_name LIKE '%_91'
LIMIT 10;
```

```
%%sql
-- UNION ALL will not remove duplicate rows; the below query will produce 75 rows
SELECT c_custkey, c_name FROM customer WHERE c_name LIKE '%_91%'
UNION ALL
SELECT c_custkey, c_name FROM customer WHERE c_name LIKE '%_91%'
UNION ALL
SELECT c_custkey, c_name FROM customer WHERE c_name LIKE '%_91%'
LIMIT 10;
```

When we want to retrieve all rows from the first dataset that are not present in the second dataset, we can use `EXCEPT`.

```
%%sql
-- EXCEPT will get the rows in the first query result that is not in the second query result
SELECT c_custkey, c_name FROM customer WHERE c_name LIKE '%_91%'
EXCEPT
SELECT c_custkey, c_name FROM customer WHERE c_name LIKE '%_91%'
LIMIT 10;
```

```
%%sql
-- The below query will result in 23 rows; the first query has 25 rows, and the second has tw
SELECT c_custkey, c_name FROM customer WHERE c_name LIKE'%_91%'
EXCEPT
SELECT c_custkey, c_name FROM customer WHERE c_name LIKE '%191%'
LIMIT 10;
```

1.8 Sub-query: Use a query instead of a table

When we want to use the result of a query as a table in another query, we use subqueries. Let's consider an example:

1. Create a report that shows the nation, how many items it supplied (by suppliers in that nation), and how many items it purchased (by customers in that nation).

```
%%sql
SELECT
  n.n_name AS nation_c_name,
  s.quantity AS supplied_items_quantity,
  c.quantity AS purchased_items_quantity
FROM
  nation n
  LEFT JOIN (
    SELECT
      n.n_nationkey,
      SUM(l.l_quantity) AS quantity
    FROM
      lineitem l
      JOIN supplier s ON l.l_suppkey = s.s_suppkey
      JOIN nation n ON s.s_nationkey = n.n_nationkey
    GROUP BY
      n.n_nationkey
  ) s ON n.n_nationkey = s.n_nationkey
```

```
LEFT JOIN (
  SELECT
    n.n_nationkey,
    SUM(l.l_quantity) AS quantity
  FROM
    lineitem l
    JOIN orders o ON l.l_orderkey = o.o_orderkey
    JOIN customer c ON o.o_custkey = c.c_custkey
    JOIN nation n ON c.c_nationkey = n.n_nationkey
  GROUP BY
    n.n_nationkey
) c ON n.n_nationkey = c.n_nationkey;
```

In the above query, we can see that there are two sub-queries, one to calculate the quantity supplied by a nation and the other to calculate the quantity purchased by the customers of a nation.

1.9 Change data types (CAST) and handle NULLS (COALESCE)

Every column in a table has a specific data type. The data types fall under one of the following categories.

1. **`Numerical`**: Data types used to store numbers.
    1. Integer: Positive and negative numbers. Different types of Integer, such as tinyint, int, and bigint, allow storage of different ranges of values. Integers cannot have decimal digits.
    2. Floating: These can have decimal digits but store an approximate value.
    3. Decimal: These can have decimal digits and store the exact value. The decimal type allows you to specify the scale and precision. Where scale denotes the count of numbers allowed as a whole & precision denotes the count of numbers allowed after the decimal point. E.g., DECIMAL(8,3) allows eight numbers in total, with three allowed after the decimal point.
2. **`Boolean`**: Data types used to store True or False values.
3. **`String`**: Data types used to store alphanumeric characters.
    1. Varchar(n): Data type allows storage of a variable character string, with a permitted max length n.
    2. Char(n): Data type allows storage of a fixed character string. A column of char(n) type adds (length(string) - n) empty spaces to a string that does not have n characters.

4. **`Date & time`**: Data types used to store dates, time, & timestamps(date + time).
5. **`Objects (STRUCT, ARRAY, MAP, JSON)`**: Data types used to store JSON and ARRAY data.

Some databases have data types that are unique to them as well. We should check the database documents to understand the data types offered.

It is best practice to use the appropriate data type for your columns. We can convert data types using the `CAST` function, as shown below.

A `NULL` will be used for that field when a value is not present. In cases where we want to use the first non-NULL value from a list of columns, we use `COALESCE` as shown below.

Let's consider the following example. We can see how when `l.orderkey` is NULL, the DB uses `999999` as the output.

```
%%sql
SELECT
    o.o_orderkey,
    o.o_orderdate,
    COALESCE(l.l_orderkey, 9999999) AS lineitem_orderkey,
    l.l_shipdate
FROM
    orders o
    LEFT JOIN lineitem l ON o.o_orderkey = l.l_orderkey
    AND o.o_orderdate BETWEEN l.l_shipdate - INTERVAL '5' DAY
    AND l.l_shipdate + INTERVAL '5' DAY
LIMIT
    10;
```

1.10 Use these standard inbuilt DB functions for String, Time, and Numeric data manipulation

When processing data, more often than not, we will need to change values in columns; shown below are a few standard functions to be aware of:

1. **`String functions`**
    1. **LENGTH** is used to calculate the length of a string. E.g., `SELECT LENGTH('hi');` will output 2.
    2. **CONCAT** combines multiple string columns into one. E.g., `SELECT CONCAT(clerk, '-', orderpriority) FROM ORDERS LIMIT 5;` will concatenate clerk and order-priority columns with a dash in between them.

3. **SPLIT** is used to split a value into an array based on a given delimiter. E.g., `SELECT SPLIT(clerk, '#') FROM ORDERS LIMIT 5;` will output a column with arrays formed by splitting clerk values on `#`.
4. **SUBSTRING** is used to get a sub-string from a value, given the start and length. E.g., `SELECT clerk, SUBSTRING(clerk, 1, 5) FROM orders LIMIT 5;` will get the first five characters of the clerk column. Note that indexing starts from 1 in Spark SQL.
5. **TRIM** is used to remove empty spaces to the left and right of the value. E.g., `SELECT TRIM(' hi ');` will output `hi` without any spaces around it. LTRIM and RTRIM are similar but only remove spaces before and after the string, respectively.

2. **Date and Time functions**

1. **Adding and subtracting dates**: Is used to add and subtract periods; the format heavily depends on the DB. E.g., In Spark SQL, the query

```
SELECT
DATEDIFF(DATE '2023-11-05', DATE '2022-10-01') AS diff_in_days,
MONTHS_BETWEEN(DATE '2023-11-05', DATE '2022-10-01') AS diff_in_months,
YEAR(DATE '2023-11-05') - YEAR(DATE '2022-10-01') AS diff_in_years;
```

   It will show the difference between the two dates in the specified period. We can also add/subtract an arbitrary period from a date/time column. E.g., `SELECT DATE_ADD(DATE '2022-11-05', 10);` will show the output `2022-11-15`.

2. **string <=> date/time conversions**: When we want to change the data type of a string to date/time, we can use the `DATE 'YYYY-MM-DD'` or `TIMESTAMP 'YYYY-MM-DD HH:mm:SS'` functions. But when the data is in a different date/time format such as `MM/DD/YYYY`, we will need to specify the input structure; we do this using `TO_DATE` or `TO_TIMESTAMP`. E.g. `SELECT TO_DATE('11-05-2023', 'MM-dd-yyyy');`. We can convert a timestamp/date into a string with the required format using `DATE_FORMAT`. E.g., `SELECT DATE_FORMAT(orderdate, 'yyyy-MM-01') AS first_month_date FROM orders LIMIT 5;` will map every orderdate to the first of their month.
3. **Time frame functions (YEAR/MONTH/DAY)**: When we want to extract specific periods from a date/time column, we can use these functions. E.g., `SELECT YEAR(DATE '2023-11-05');` will return 2023. Similarly, we have MONTH, DAY, HOUR, MINUTE, etc.

3. **Numeric**

1. **ROUND** is used to specify the number of digits allowed after the decimal point. E.g. `SELECT ROUND(100.102345, 2);`

1.11 Save queries as views for more straightforward reads

When we have large/complex queries that we need to run often, we can save them as views. Views are database objects that operate similarly to tables. The OLAP DB executes the underlying query when we query a view.

Use views to hide query complexities and limit column access (by exposing only specific table columns) for end-users.

For example, we can create a view for the nation-level report from the above section, as shown below.

```
%%sql
DROP VIEW IF EXISTS nation_supplied_purchased_quantity
```

```
%%sql
CREATE VIEW nation_supplied_purchased_quantity AS
SELECT
    n.n_name AS nation_name,
    s.quantity AS supplied_items_quantity,
    c.quantity AS purchased_items_quantity
FROM
    nation n
    LEFT JOIN (
        SELECT
            n_nationkey as nationkey,
            sum(l_quantity) AS quantity
        FROM
            lineitem l
            JOIN supplier s ON l.l_suppkey = s.s_suppkey
            JOIN nation n ON s.s_nationkey = n.n_nationkey
        GROUP BY
            n.n_nationkey
    ) s ON n.n_nationkey = s.nationkey
    LEFT JOIN (
        SELECT
            n_nationkey as nationkey,
            sum(l_quantity) AS quantity
        FROM
            lineitem l
            JOIN orders o ON l.l_orderkey = o.o_orderkey
            JOIN customer c ON o.o_custkey = c.c_custkey
            JOIN nation n ON c.c_nationkey = n.n_nationkey
```

```
        GROUP BY
            n.n_nationkey
    ) c ON n.n_nationkey = c.nationkey;
```

```
%%sql
SELECT
    *
FROM
    nation_supplied_purchased_quantity;
```

Now the view `nation_supplied_purchased_quantity` will run the underlying query when used.

1.12 Exercises

1. Write a query that shows the number of items returned for each region name
2. List the top 10 most selling parts (part name)
3. Sellers (name) who have sold at least one of the top 10 selling parts
4. Number of items returned for each order price bucket. The definition of order price bucket is shown below.

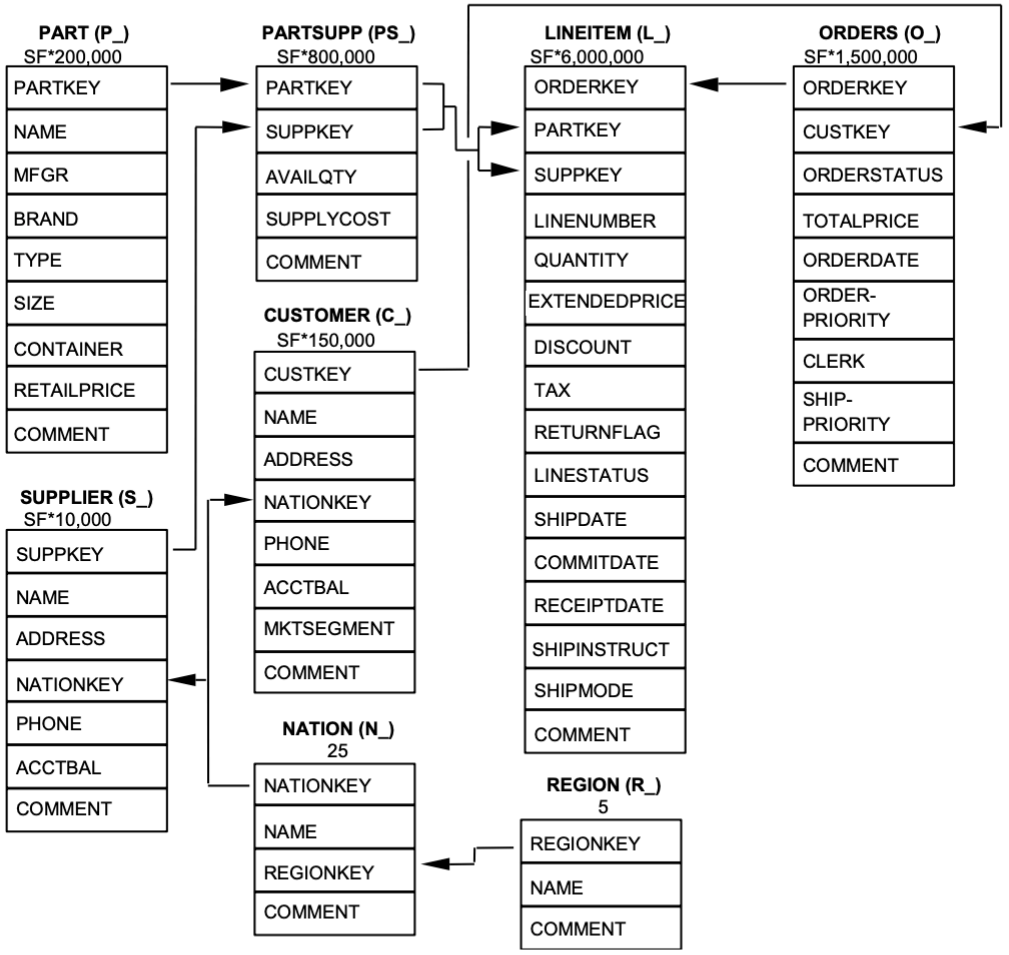
```
CASE
    WHEN o_totalprice > 100000 THEN 'high'
    WHEN o_totalprice BETWEEN 25000 AND 100000 THEN 'medium'
    ELSE 'low'
END AS order_price_bucket
```

5. Average time (in days) between receiptdate and shipdate for each nation (name)

1.2 Database Entities, Relationships, and Characteristics

The components of the TPC-H database are defined to consist of eight separate and individual tables (the Base Tables). The relationships between columns of these tables are illustrated in Figure 2: The TPC-H Schema.

Figure 2: The TPC-H Schema



Legend:

- The parentheses following each table name contain the prefix of the column names for that table;
- The arrows point in the direction of the one-to-many relationships between tables;
- The number/formula below each table name represents the cardinality (number of rows) of the table. Some are factored by SF, the Scale Factor, to obtain the chosen database size. The cardinality for the LINEITEM table is approximate (see Clause 4.2.5).

Here is the data model:

1.13 Recommended reading

1. **Improving SQL Skills**
2. **SQL techniques**
3. **Advanced SQL skills**

2 CTE (Common Table Expression) improves code readability and reduces repetition

2.1 Why use a CTE

A CTE is a named select statement that can be reused in a single query.

Complex SQL queries often involve multiple subqueries. Multiple sub-queries make the code hard to read. Use a Common Table Expression (CTE) to make your queries readable.

CTEs also make testing complex queries simpler.

2.2 How to define a CTE

Use the `WITH` keyword to start defining a CTE; the WITH keyword is not necessary for consecutive CTE definitions.

```
%%sql
use prod.db
```

```
%%sql
-- CTE definition
WITH
  supplier_nation_metrics AS ( -- CTE 1 defined using WITH keyword
    SELECT
      n.n_nationkey,
      SUM(l.l_QUANTITY) AS num_supplied_parts
    FROM
      lineitem l
      JOIN supplier s ON l.l_suppkey = s.s_suppkey
      JOIN nation n ON s.s_nationkey = n.n_nationkey
    GROUP BY
      n.n_nationkey
  ),
```

```
  buyer_nation_metrics AS ( -- CTE 2 defined just as a name
    SELECT
      n.n_nationkey,
      SUM(l.l_QUANTITY) AS num_purchased_parts
    FROM
      lineitem l
      JOIN orders o ON l.l_orderkey = o.o_orderkey
      JOIN customer c ON o.o_custkey = c.c_custkey
      JOIN nation n ON c.c_nationkey = n.n_nationkey
    GROUP BY
      n.n_nationkey
  )
SELECT -- The final select will not have a comma before it
  n.n_name AS nation_name,
  s.num_supplied_parts,
  b.num_purchased_parts
FROM
  nation n
  LEFT JOIN supplier_nation_metrics s ON n.n_nationkey = s.n_nationkey
  LEFT JOIN buyer_nation_metrics b ON n.n_nationkey = b.n_nationkey
LIMIT 10;
```

Note that the last CTE does not have a `,` after it.

Let's look at ane example: Calculate the money lost due to discounts. Use the lineitem to retrieve the prices of items (excluding discounts) that are part of an order and compare them to the order.

The `l_extendedprice` column does not include discounts & the `o_totalprice` column includes discounts.

```
%%sql
WITH lineitem_agg AS (
    SELECT
        l_orderkey,
        SUM(l_extendedprice) AS total_price_without_discount
    FROM
        lineitem
    GROUP BY
        l_orderkey
)
SELECT
    o.o_orderkey,
```

```
    o.o_totalprice,
    l.total_price_without_discount - o.o_totalprice AS amount_lost_to_discount
FROM
    orders o
JOIN
    lineitem_agg l ON o.o_orderkey = l.l_orderkey
ORDER BY
    o.o_orderkey;
```

Note how each CTE can correspond to getting data in a certain grain with necessary enrichments and then compare it to the order's data, whose total price has been computed with discounts.

2.3 Recreating similar CTE is a sign that it should be a table

A sql query with multiple temporary tables is better than a 1000-line SQL query with numerous CTEs.

Keep the number of CTEs per query small (depends on the size of the query, but typically $<$ 5)

Assume that you have stakeholders running the below query multiple times as needed.

When multiple stakeholders repeatedly run the exact CTE definition, it is usually an indication that the CTE should be created as a table or view to ensure stakeholders have a unified definition.

```
%%sql
WITH orders_cte AS (
    SELECT
        o_orderkey,
        o_custkey,
        o_orderstatus,
        CAST(o_orderdate AS TIMESTAMP) AS o_orderdate,
        o_orderpriority,
        o_clerk,
        o_shippriority,
        o_comment,
        o_totalprice
    FROM orders
),
stg_customers AS (
```

```
    SELECT
        c_custkey,
        c_name,
        c_address,
        c_nationkey,
        c_phone,
        c_acctbal,
        c_mktsegment,
        c_comment
    FROM customer
),
nation_cte AS (
    SELECT
        CAST(n_nationkey AS INT) AS n_nationkey,
        CAST(n_name AS STRING) AS n_name,
        CAST(n_regionkey AS INT) AS n_regionkey,
        CAST(n_comment AS STRING) AS n_comment
    FROM nation
),
dim_customers AS (
    SELECT
        c.c_custkey,
        c.c_name,
        c.c_address,
        c.c_nationkey,
        n.n_name AS nation_name,
        c.c_phone,
        c.c_acctbal,
        c.c_mktsegment,
        c.c_comment
    FROM stg_customers c
    INNER JOIN nation_cte n ON c.c_nationkey = n.n_nationkey
)
SELECT
    o.o_orderkey,
    o.o_custkey,
    o.o_orderstatus,
    o.o_orderdate,
    o.o_orderpriority,
    o.o_clerk,
    o.o_shippriority,
    o.o_totalprice,
```

```
    c.c_name AS customer_name,
    c.c_address AS customer_address,
    c.c_phone AS customer_phone,
    c.c_acctbal AS customer_account_balance,
    c.c_mktsegment AS customer_market_segment,
    c.nation_name AS customer_nation_name
FROM orders_cte o
INNER JOIN dim_customers c ON o.o_custkey = c.c_custkey;
```

2.4 Exercises

1. Sellers (name) who have sold at least one of the top 10 selling parts (use CTE)

3 Use window function when you need to use values from other rows to compute a value for the current row

Window functions allow you to operate on a set of rows at a time and produce output that has the exact grain as the input (vs GROUP BY, which operates on a set of rows, but also changes the meaning of an output row).

Let's explore why we might need window functions instead of a `GROUP BY`.

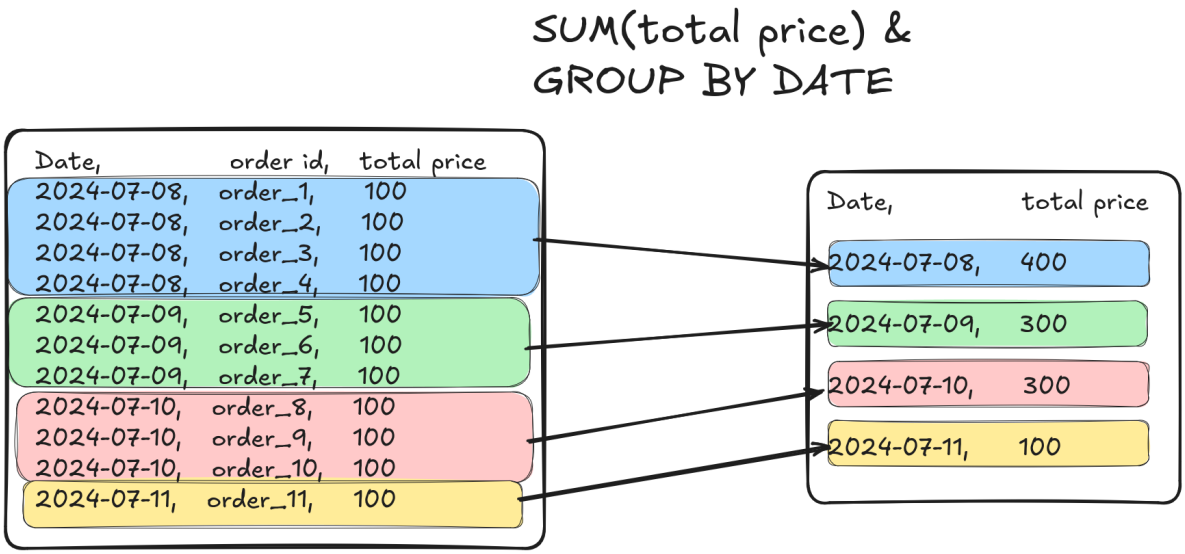


Figure 3.1: GROUP BY

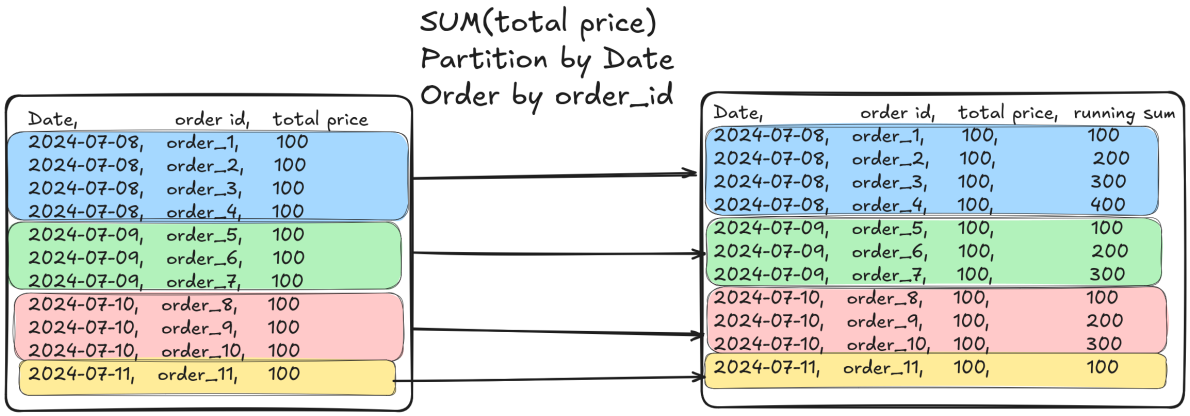


Figure 3.2: WINDOW FUNCTION

NOTE Notice how `GROUP BY` changes granularity, i.e., the input data had one row per order (aka order grain or order level), but the output had one row per date (aka date grain or date level).

When you perform some operation that requires data from multiple rows to produce the data for one row, without changing the grain, `Window functions` are almost always a good fit.

Common scenarios when you want to use window functions:

1. Calculate running metrics/sliding window over rows in the table (aggregate functions)
2. Ranking rows based on values in column(s) (ranking functions)
3. Access other rows' values while operating on the current row (value functions)
4. Any combination of the above

3.1 Window functions have four parts

1. **Partition**: Defines a set of rows based on specified column(s) value. If no partition is specified, the entire table is considered a partition.
2. **Order By**: This optional clause specifies how to order the rows within a partition. This is an optional clause; without this, the rows inside a partition will not be ordered.
3. **Function**: The function to be applied to the current row.
4. **Window frame**: Within a partition, a window frame allows you to specify the rows to be considered in the function computation. This enables more options for choosing which rows to apply the function to.

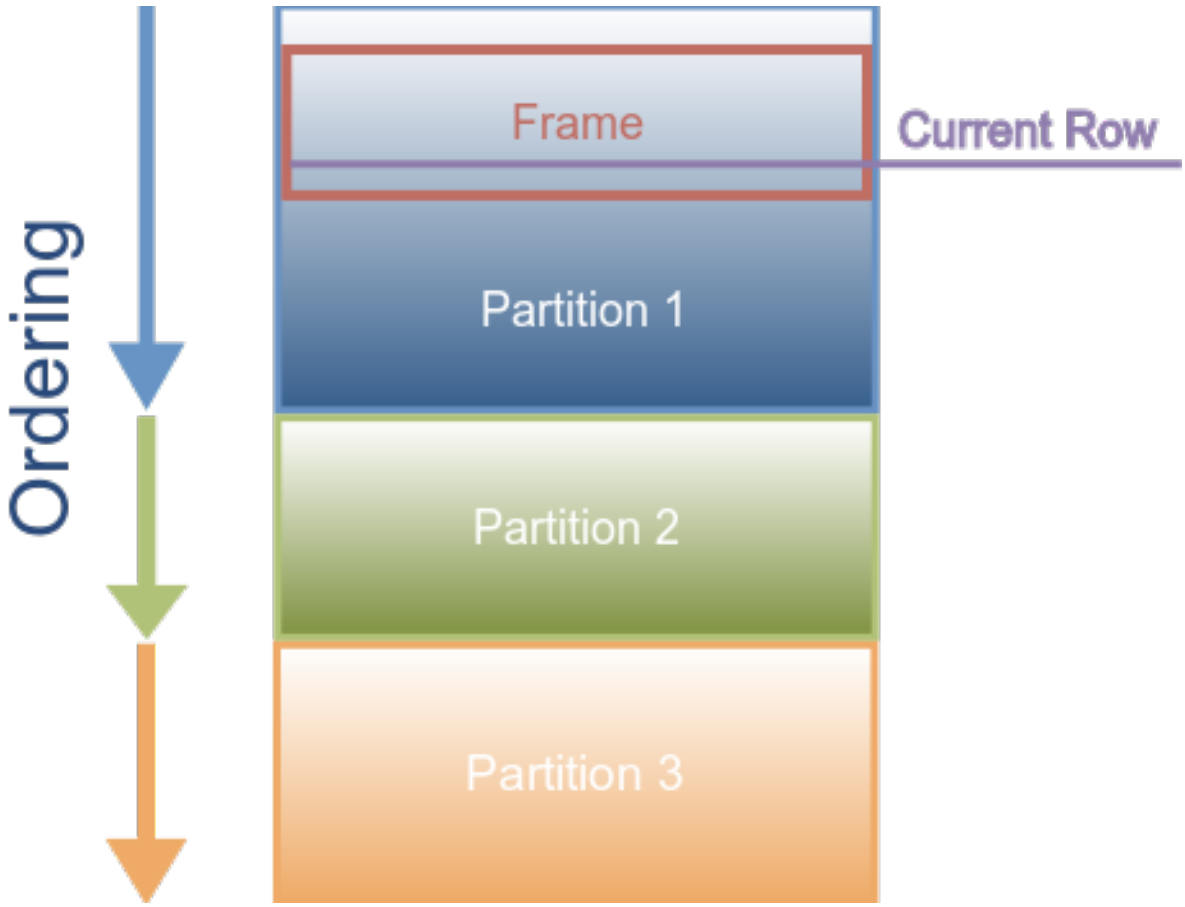


Figure 3.3: Framing

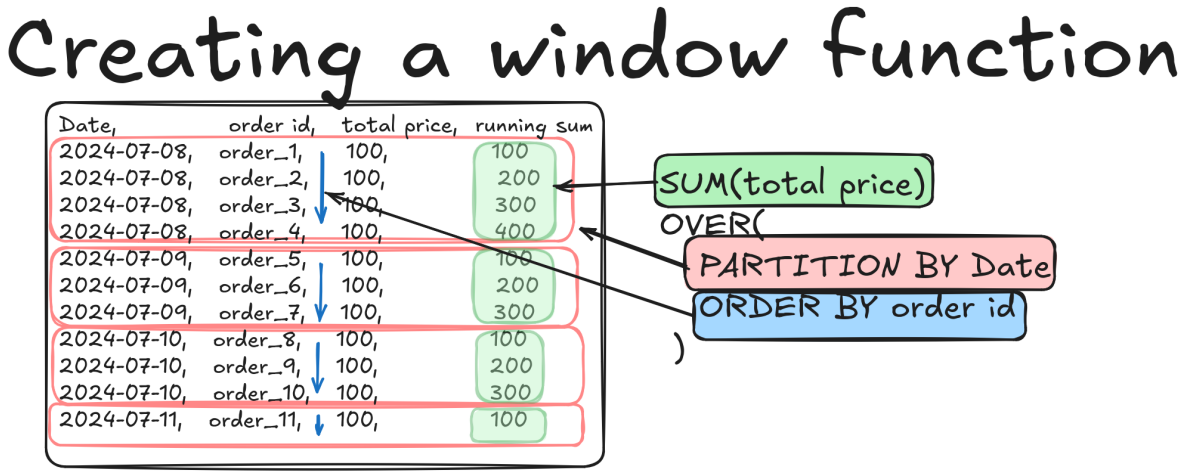


Figure 3.4: Create window function

```
%%sql
use prod.db
```

```
%%sql
SELECT
  o_custkey,
  o_orderdate,
  o_totalprice,
  SUM(o_totalprice) -- FUNCTION
  OVER (
    PARTITION BY
      o_custkey -- PARTITION
    ORDER BY
      o_orderdate -- ORDER BY; ASCENDING ORDER unless specified as DESC
  ) AS running_sum
FROM
  orders
WHERE
  o_custkey = 4
ORDER BY
  o_orderdate
LIMIT
  10;
```

The function `SUM` used in the above query is an aggregate function. Notice how the `running_sum` adds up (i.e., aggregates) the `o_totalprice` across all rows. The rows

themselves are ordered in ascending order by their orderdate.

Reference: The standard aggregate functions are `MIN, MAX, AVG, SUM, & COUNT`, modern data systems offer a variety of powerful aggregation functions. Check your database documentation for available aggregate functions. e.g., list of agg functions available in TrinoDB

Let's look at an example: Write a query to calculate the daily running average of the total price for every customer.

Hint: Figure out the `PARTITION BY` column first, then the `ORDER BY` column, and finally the `FUNCTION` to use to compute the running average.

```
%%sql
SELECT
  o_custkey,
  o_orderdate,
  o_totalprice,
  AVG(o_totalprice) -- FUNCTION
  OVER (
    PARTITION BY
      o_custkey -- PARTITION
    ORDER BY
      o_orderdate -- ORDER BY; ASCENDING ORDER unless specified as DESC
  ) AS running_sum
FROM
  orders
WHERE
  o_custkey = 4
ORDER BY
  o_orderdate
LIMIT
  10;
```

3.2 Use window frames to define a set of rows to operate on

While our functions operate on the rows in the partition a window frame provides more granular ways to operate on a select set of rows within a partition.

When we need to operate one a set of rows within a parition (e.g. a sliding window) we can use the window frame to define these set of rows.

Let's look at an example: Consider a scenario where you have sales data, and you want to calculate a 3-day moving average of sales within each store:

```
%%sql
SELECT
    store_id,
    sale_date,
    sales_amount,
    AVG(sales_amount) OVER (
        PARTITION BY store_id
        ORDER BY sale_date
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS moving_avg_sales
FROM
    sales;
```

In this example:

1. **PARTITION BY** store_id ensures the calculation is done separately for each store.
2. **ORDER BY** sale_date defines the order of rows within each partition.
3. **ROWS BETWEEN 2 PRECEDING AND CURRENT ROW** specifies the window frame, considering the current row and the two preceding rows to calculate the moving average.

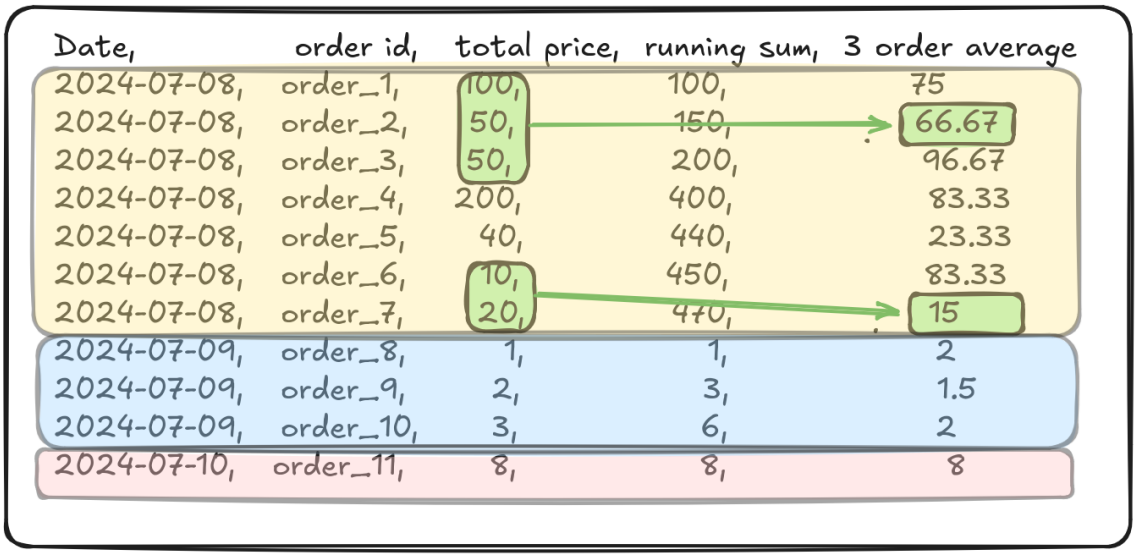


Figure 3.5: Three order Sliding window average

Without defining the window frame, the function will consider all rows in the partition up to the current row to compute the `moving_avg_sales`.

3.2.0.1 Use the ordering of rows to define your window frame with the ROWS clause

1. **ROWS**: Used to select a set of rows relative to the current row based on position.
    1. Row definition format: `ROWS BETWEEN start_point AND end_point`.
    2. The start_point and end_point can be any of the following three (in the proper order:
        1. **n PRECEDING**: n rows preceding the current row. UNBOUNDED PRECEDING indicates all rows before the current row.
        2. **n FOLLOWING**: n rows following the current row. UNBOUNDED FOLLOWING indicates all rows after the current row.

Let's see how relative row numbers can be used to define a window range.

Consider this window function.

```
AVG(total_price) OVER ( -- FUNCTION: RUNNING AVERAGE
    PARTITION BY o_custkey -- PARTITIONED BY customer
    ORDER BY order_month
    ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING -- WINDOW FRAME DEFINED AS 1 ROW PRECEDING TO 1
    )
```

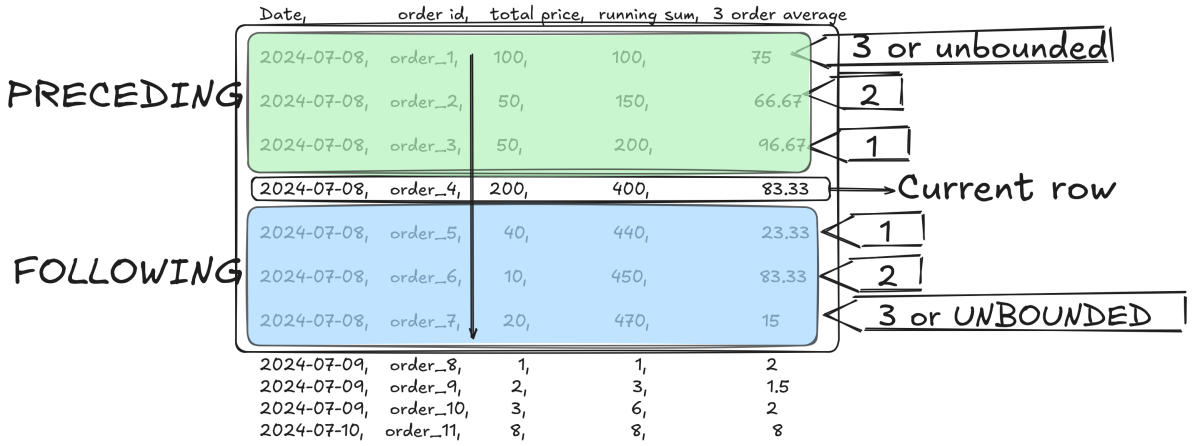


Figure 3.6: Window frame with ROWS

3.2.0.2 Use values of the columns to define the window frame using the RANGE clause

1. **RANGE**: Used to select a set of rows relative to the current row based on the value of the columns specified in the `ORDER BY` clause.
    1. Range definition format: `RANGE BETWEEN start_point AND end_point`.

2. The start_point and end_point can be any of the following:
    1. **CURRENT ROW**: The current row.
    2. **n PRECEDING**: All rows with values within the specified range that are less than or equal to n units preceding the value of the current row.
    3. **n FOLLOWING**: All rows with values within the specified range that are greater than or equal to n units following the value of the current row.
    4. **UNBOUNDED PRECEDING**: All rows before the current row within the partition.
    5. **UNBOUNDED FOLLOWING**: All rows after the current row within the partition.
3. `RANGE` is handy when dealing with numeric or date/time ranges, allowing for calculations like running totals, moving averages, or cumulative distributions.

Let's see how `RANGE` works with `AVG(total price) OVER (PARTITION BY customer id ORDER BY date RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND '1' DAY FOLLOWING)` using the below visualization:

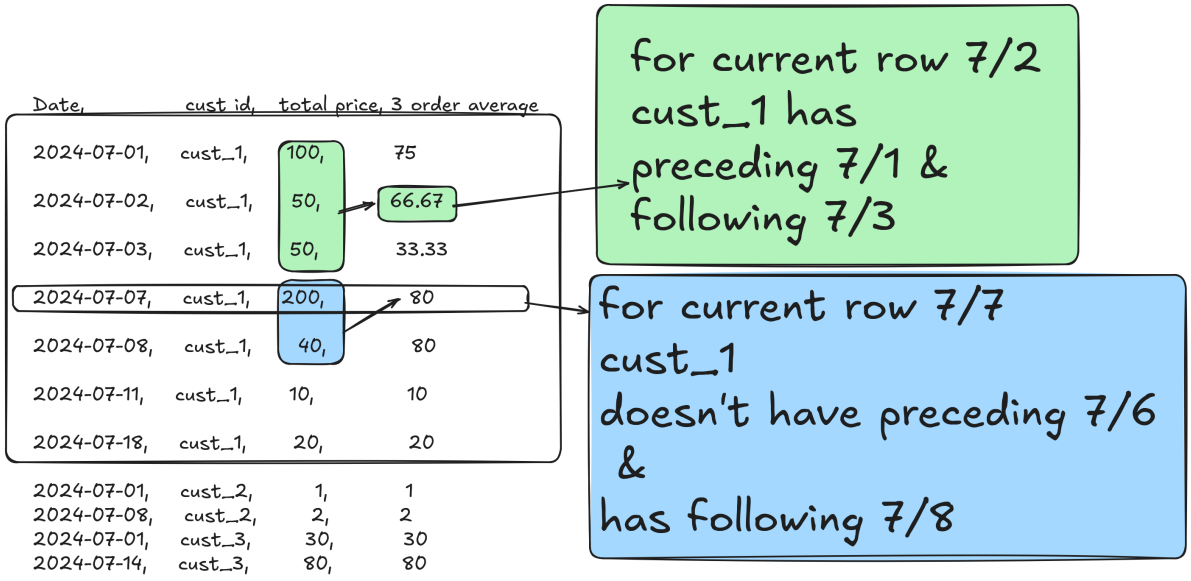


Figure 3.7: RANGE

3.3 Ranking functions enable you to rank your rows based on an order by clause

If you are working on a problem to retrieve the top/bottom n rows (as defined by a specific value), then use the **row** functions.

Let's look at an example of how to use a row function:

From the `orders` table **get the top 3 spending customers per day**. The orders table schema is shown below:

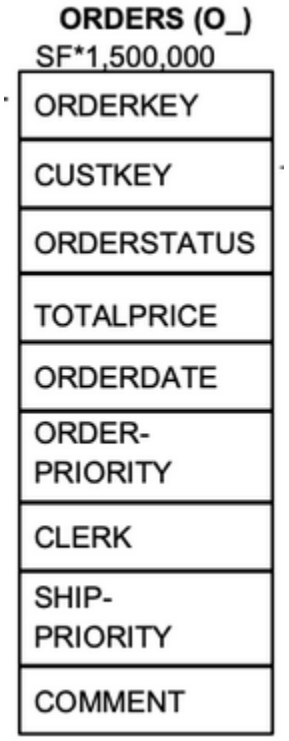


Figure 3.8: Orders table

```
%%sql
SELECT
  *
FROM
  (
    SELECT
      o_orderdate,
      o_totalprice,
      o_custkey,
      RANK() -- RANKING FUNCTION
      OVER (
        PARTITION BY
          o_orderdate -- PARTITION BY order date
        ORDER BY
```

```
          o_totalprice DESC -- ORDER rows withing partition by totalprice
      ) AS rnk
    FROM
      orders
  )
WHERE
  rnk <= 3
ORDER BY
  o_orderdate
LIMIT
  5;
```

Standard RANKING functions:

1. **RANK**: Ranks the rows starting from 1 to n within the window frame. Ranks the rows with the same value (defined by the "ORDER BY" clause) as the same and skips the ranking numbers that would have been present if the values were different.
2. **DENSE_RANK**: Ranks the rows starting from 1 to n within the window frame. Ranks the rows with the same value (defined by the "ORDER BY" clause) as the same and does not skip any ranking numbers.
3. **ROW_NUMBER**: Adds a row number that starts from 1 to n within the window frame and does not create any repeating values.

```
%%sql
-- Let's look at an example showing the difference between RANK, DENSE_RANK and ROW_NUMBER
SELECT
    order_date,
    order_id,
    total_price,
    ROW_NUMBER() OVER (PARTITION BY order_date ORDER BY total_price) AS row_number,
    RANK() OVER (PARTITION BY order_date ORDER BY total_price) AS rank,
    DENSE_RANK() OVER (PARTITION BY order_date ORDER BY total_price) AS dense_rank
FROM (
    SELECT
        '2024-07-08' AS order_date, 'order_1' AS order_id, 100 AS total_price UNION ALL
    SELECT
        '2024-07-08', 'order_2', 200 UNION ALL
    SELECT
        '2024-07-08', 'order_3', 150 UNION ALL
    SELECT
        '2024-07-08', 'order_4', 90 UNION ALL
```

```
    SELECT
        '2024-07-08', 'order_5', 100 UNION ALL
    SELECT
        '2024-07-08', 'order_6', 90 UNION ALL
    SELECT
        '2024-07-08', 'order_7', 100 UNION ALL
    SELECT
        '2024-07-10', 'order_8', 100 UNION ALL
    SELECT
        '2024-07-10', 'order_9', 100 UNION ALL
    SELECT
        '2024-07-10', 'order_10', 100 UNION ALL
    SELECT
        '2024-07-11', 'order_11', 100
) AS orders
ORDER BY order_date, row_number;
```

3.4 Aggregate functions enable you to compute running metrics

The standard aggregate functions are `MIN, MAX, AVG, SUM, & COUNT`. In addition to these, make sure to check your DB engine documentation, in our case, Spark Aggregate functions.

When you need a running sum/min/max/avg, it's almost always a use case for aggregate functions with windows.

Let's look at an example:

1. Write a query on the orders table that has the following output:
    1. o_custkey
    2. order_month: In YYYY-MM format, use strftime(o_orderdate, '%Y-%m') AS order_month
    3. total_price: Sum of o_totalprice for that month
    4. three_mo_total_price_avg: The 3 month (previous, current & next) average of total_price for that customer

```
%%sql
SELECT
  order_month,
  o_custkey,
  total_price,
  ROUND(
```

```
    AVG(total_price) OVER ( -- FUNCTION: RUNNING AVERAGE
      PARTITION BY
        o_custkey -- PARTITIONED BY customer
      ORDER BY
        order_month ROWS BETWEEN 1 PRECEDING
        AND 1 FOLLOWING -- WINDOW FRAME DEFINED AS 1 ROW PRECEDING to 1 ROW FOLLOWING
    ),
    2
  ) AS three_mo_total_price_avg
FROM
  (
    SELECT
      date_format(o_orderdate, 'yyyy-MM') AS order_month,
      o_custkey,
      SUM(o_totalprice) AS total_price
    FROM
      orders
    GROUP BY
      1,
      2
  )
LIMIT
  5;
```

Now that we have seen how to **define a window function** and how to use **ranking and aggregation** functions, let's take it a step further by practicing **value functions**.

Remember that value functions are used to access the values of other rows while operating on the current row.

Let's take a look at LEAD and LAG functions:

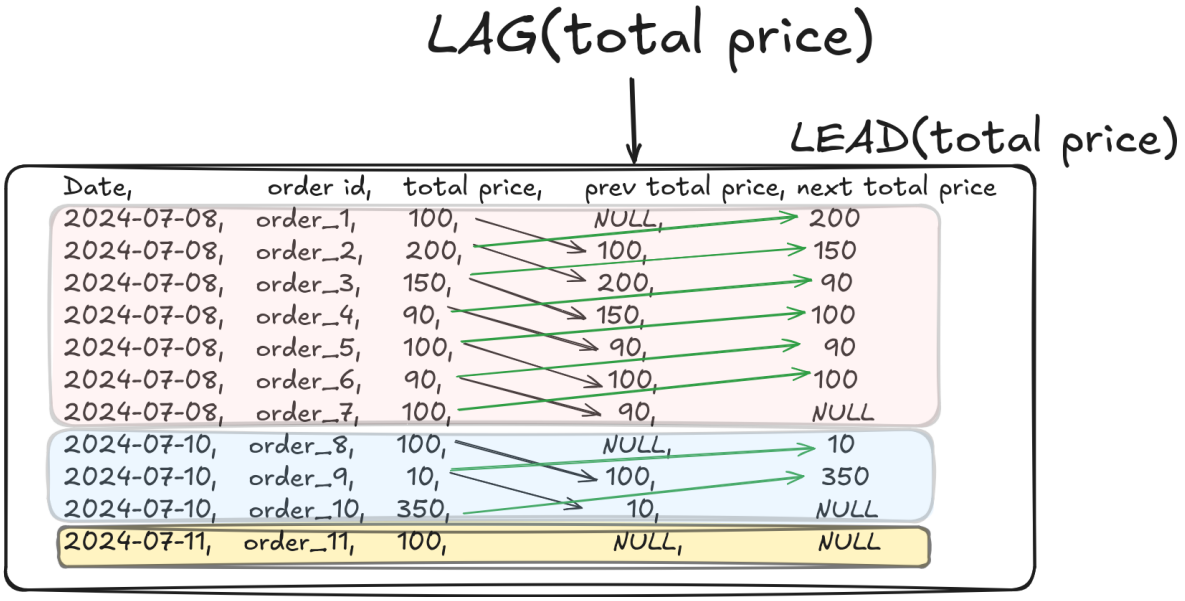


Figure 3.9: LAG AND LEAD

3.5 Value functions are used to access other rows' values

Standard VALUE functions:

1. **NTILE(n)**: Divides the rows in the window frame into n approximately equal groups, and assigns a number to each row indicating which group it belongs to.
2. **FIRST_VALUE()**: Returns the first value in the window frame.
3. **LAST_VALUE()**: Returns the last value in the window frame.
4. **LAG()**: Accesses data from a previous row within the window frame.
5. **LEAD()**: Accesses data from a subsequent row within the window frame.

3.6 Exercises

1. Write a query on the orders table that has the following output:
    1. order_month,
    2. o_custkey,
    3. total_price,
    4. three_mo_total_price_avg

5. **consecutive_three_mo_total_price_avg**: The consecutive 3 month average of total_price for that customer. Note that this should only include months that are chronologically next to each other.

Hint: Use `CAST(strftime(o_orderdate, '%Y-%m-01') AS DATE)` to cast order_month to date format.

Hint: Use the `INTERVAL` format shown above to construct the window function to compute `consecutive_three_mo_total_price_avg` column.

- The orders table schema is shown below:

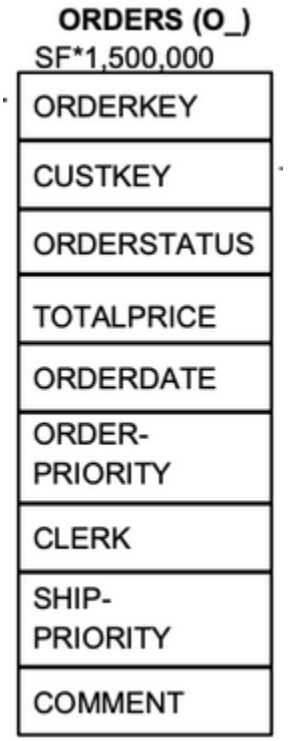


Figure 3.10: Orders table

2. From the `orders` table get the 3 lowest spending customers per day

Hint: Figure out the `PARTITION BY` column first, then the `ORDER BY` column and finally the `FUNCTION` to use to compute running average.

3. Write a SQL query using the `orders` table that calculates the following columns:
    1. o_orderdate: From orders table

2. o_custkey: From orders table
3. o_totalprice: From orders table
4. totalprice_diff: The customers current day's o_totalprice - that same customers most recent previous purchase's o_totalprice

Hint: Start by figuring out what the `PARTITION BY` column should be, then what the `ORDER BY` column should be, and then finally the function to use.

Hint: Use the `LAG(column_name)` ranking function to identify the prior day's revenue.

3.7 Recommended reading

1. Window SQL Youtube workshop

Part II

Python connects the different part of your data pipeline

Python is the glue that holds the various parts of your data pipeline together. While powerful data processing engines (Snowflake, Spark, BigQuery, etc) have made processing large amounts of data efficient, you still need a programming language to tell these engines what to do.

In most companies, Python dominates the data stack. You'll typically use Python to pull data from the source system (Extract), tell the data processing engine how to process the data (e.g, via SQL queries on Snowflake or SQL/Dataframe query on Spark), and load data into its destination.

In this section, we will cover the basics of Python, its application in data engineering, and conclude with a topic crucial for ensuring code changes don't break existing logic (testing).

Data is stored on disk and processed in memory

Python is not the most optimal language for large-scale data processing. You would often use Python to tell a data processing engine what to do. For this reason, it's critical to understand the difference between disk and memory.

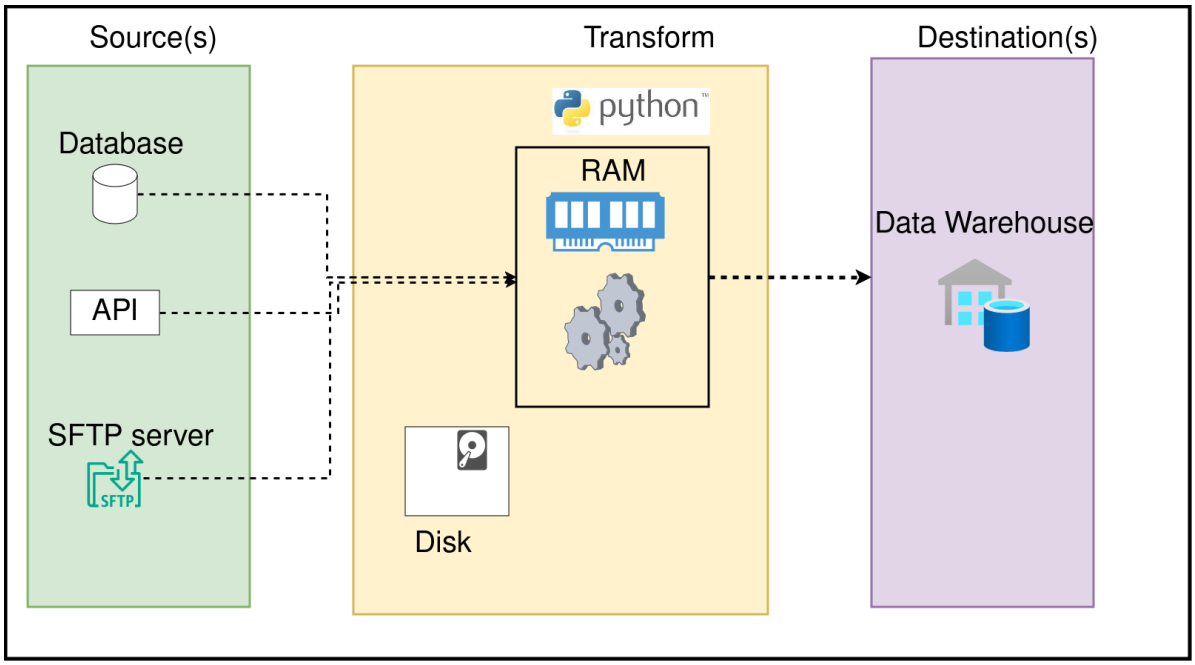


Figure 3.11: Disk and Memory

When we run a Python (or any language) script, it is run as a process. Each process will use a part of your computer's memory (RAM). Understanding the difference between RAM and Disk will enable you to write efficient data pipelines; let's go over them:

1. `Memory` is the space used by a running process to store any information that it may need for its operation. The computer's RAM is used for this purpose. This is where any variables you define and the Pandas dataframe you use will be stored.
2. `Disk` is used to store data. When we process data from disk (read data from a CSV, etc.), it means that our process reads data from disk into memory and then processes it. Computers generally use HDDs or SSDs to store files.

RAM is expensive, while disk (HDD, SSD) is cheaper. One issue with data processing is that the memory available to use is often less than the size of the data to be processed. This is when we utilize distributed systems, such as Spark or DuckDB, which enable us to process data larger than memory.

As we will see in the transformation sections, when we use systems like Spark, Snowflake, or Duckdb, Python is just the interface; the real data processing (and Memory and disk usage) depends on the data processing engine.

Loading...

4 Manipulate data with standard libraries and co-locate code with classes and functions

In this chapter, we will learn Commonly used data structures Loops Classes, objects, and functions.

4.1 Use the appropriate data structure based on how the data will be used

Let's go over some basics of the Python language:

1. **`Variables`**: A storage location identified by its name, containing some value.
2. **`Operations`**: We can do any operation (arithmetic for numbers, string transformation for text) on variables
3. **`Data Structures`**: They are ways of representing data. Each has its own pros and cons, as well as specific situations where it is the right fit.

   3.1. **`List`**: A collection of elements that can be accessed by knowing the element's location (aka index). Lists retain the order of their elements.

   3.2. **`Dictionary`**: A collection of key-value pairs where each key is mapped to a value using a hash function. The dictionary provides fast data retrieval based on keys.

   3.3. **`Set`**: A collection of unique elements that do not allow duplicates.

   3.4. **`Tuple`**: A collection of immutable(non changeable) elements, tuples retain their order once created.

```
# Variables
a = 10
b = 20

# operations
c = a + b
print(c) # prints the value
```

```
s = '  Some string '
# operations
print(s.strip())
```

```
# Data structures

# List
l = [1, 2, 3, 4]

print(l[0])  # Will print 1
print(l[3])  # Will print 4

# disctionary
d = {'a': 1, 'b': 2}

print(d.get('a')) # get value of a
print(d.get('b')) # get value of b

# Set
my_set = set() # set only stores unique values
my_set.add(10)
my_set.add(10) # we already have a 10
my_set.add(10) # we already have a 10
my_set.add(30)
print(my_set)
```

4.2 Manipulate data with control-flow loops

1. **`Loops`**: Looping allows a specific chunk of code to be repeated several times. The most common type is the `for` loop.
2. **`Comprehension`**: Comprehension is a shorthand way of writing a loop. This allows for concise code, great for representing simpler logic.

```
# Range(n) creates a list of values from o to n-1 (inclusive)

# we can pull out one element from a list and call it a vaiable (i in our case)
for i in range(11):
    print(i)
```

```
# List based looping
print('############### index based looping ')
for idx in range(len(l)):
    print(l[idx])

# shorthand to loop through elements in a list
print('############### shorthand to loop through elements in a list')
for elt in l:
    print(elt)

# List based looping, while getting the index number
print('############### List based looping, while getting the index number ')
for idx, elt in enumerate(l):
    print(idx, elt)
```

```
# Looping element in dictionary

# only keys
for i in d:
    print(i)

# keys and values
for k, v in d.items():
    print(f'Key: {k}, Value: {v}')
```

```
# list comprehension
# instead of writing a loop, we can use the loop inside a [] to create another list
# Here we multiply each element in l by 2 and create a new list
[elt*2 for elt in l]
```

```
# dictionary comprehenmsion
# we can create a dictionary using comprehension as well
{f'key_{elt}': elt*2 for elt in l}
```

4.3 Co-locate logic with classes and functions

1. **Functions**: A block of code that can be reused as needed. This allows us to have logic defined in one place, making it easy to maintain and use. Using it in a location is referred to as calling the function.

2. **Class and Objects**: Think of a class as a blueprint and objects as things created based on that blueprint.
3. **Library**: Libraries are code that can be reused. Python comes with standard libraries for common operations, such as a datetime library to work with time (although there are better libraries)—Standard library.
4. **Exception handling**: When an error occurs, we need our code to gracefully handle it without stopping.

```
# let's create a function to create an age_bucket for our customer data
customer_data = [
    {'name': 'customer_1', 'id': 1, 'age': 100},
    {'name': 'customer_2', 'id': 2, 'age': 42},
    {'name': 'customer_3', 'id': 3, 'age': 25},
    {'name': 'customer_4', 'id': 4, 'age': 19},
]

def get_age_bucket(customer):
    customer_age = customer['age']
    if customer_age > 85:
        return '85+'
    elif customer_age > 50:
        return '50_85'
    elif customer_age > 30:
        return '30_50'
    else:
        return '0_30'

for customer in customer_data:
    print(customer['age'], get_age_bucket(customer))
```

```
class DataExtractor:

    def __init__(self, some_value):
        self.some_value = some_value

    def get_connection(self):
        pass

    def close_connection(self):
        pass

de_object = DataExtractor(10)
print(de_object.some_value)
```

```
class Pipeline:

    def __init__(self, pipeline_type):
        self.pipeline_type = pipeline_type # called an object variable, will be specific for

    def extract(self):
        print(f'Data is being extracted for {self.pipeline_type}')

    def transform(self):
        print(f'Data is being transformed for {self.pipeline_type}')

    def load(self):
        print(f'Data is being loaded for {self.pipeline_type}')

    def run(self):
        self.extract()
        self.transform()
        self.load()

p1 = Pipeline('customer_pipeline') # we create an object of type Pipeline class
p2 = Pipeline('orders_pipeline') # we create another object of type Pipeline class

p1.run() # note how the extract, transform and load methods will print customer pipeline
print('############################')
p2.run()
```

```
# let's use a standard library to get the current date in YYYY-mm-dd format
from datetime import datetime
print(datetime.now().strftime('%Y-%m-%d'))
```

```
# When we try to access an element that is not part of a list we get an out of index error,
# with the try block, the error will be
# caught by the except block
# finally will be executed irrespective of if there was an error or not
l = [1, 2, 3, 4, 5]
```

```
index = 10
try:
    element = l[index]
    print(f"Element at index {index} is {element}")
```

```
except IndexError:
    print(f"Error: Index {index} is out of range for the list.")
finally:
    print("Execution completed.")
```

```
index = 2
try:
    element = l[index]
    print(f"Element at index {index} is {element}")
except IndexError:
    print(f"Error: Index {index} is out of range for the list.")
finally:
    print("Execution completed.")
```

4.4 Exercises

1. **Customer Order Analysis:** Write python code that processes a list of customer orders to calculate the total revenue and find top 3 the most frequent customer.

We have a list of orders, where each order is a dictionary with keys: customer_id, product, quantity, and price.

`revenue = quantity * price`

frequency of customer is defined as the number of orders

```
orders = [
    {"customer_id": "C001", "product": "laptop", "quantity": 2, "price": 1200.00},
    {"customer_id": "C002", "product": "mouse", "quantity": 1, "price": 25.99},
    {"customer_id": "C001", "product": "keyboard", "quantity": 1, "price": 89.50},
    {"customer_id": "C003", "product": "monitor", "quantity": 1, "price": 299.99},
    {"customer_id": "C002", "product": "laptop", "quantity": 1, "price": 1200.00},
    {"customer_id": "C004", "product": "headphones", "quantity": 3, "price": 79.99},
    {"customer_id": "C001", "product": "webcam", "quantity": 1, "price": 45.00},
    {"customer_id": "C003", "product": "mouse", "quantity": 2, "price": 25.99},
    {"customer_id": "C002", "product": "speaker", "quantity": 1, "price": 150.00},
    {"customer_id": "C005", "product": "tablet", "quantity": 1, "price": 399.99}
]
```

2. Data Quality Checker: Write a Python function that takes a list of email addresses and returns a dictionary with two keys: valid_emails (list) and invalid_emails (list).

Use basic validation rules 1. must contain `@` 2. `.` must be after `@` 3. must contain text before the `@`

```
email_list = [
    "john.doe@company.com",
    "jane.smith@email.co.uk",
    "invalid-email",
    "bob@gmail.com",
    "alice.brown@company.com",
    "john.doe@company.com",  # duplicate
    "missing@domain",
    "test@example.org",
    "@nodomain.com",
    "jane.smith@email.co.uk",  # duplicate
    "valid.user@site.net",
    "no-at-symbol.com",
    "another@test.io"
]
```

3. Sales Performance Tracker: Create a class called SalesRep that stores a representative's name and a list of their sales amounts.

Include methods to add sales amounts, calculate average sales, and determine if they hit a target (parameter).

```
# Sample data for creating SalesRep objects
sales_data = {
    "Alice Johnson": [15000, 18000, 22000, 16000, 19000, 21000],
    "Bob Smith": [12000, 14000, 11000, 13000, 15000, 16000],
    "Carol Davis": [25000, 28000, 30000, 27000, 32000, 29000]
}
```

4.5 Recommended reading

1. **Memory efficient processing in Python with generators**
2. **Python FP v OOP**
3. **SQL or Python for data processing**

5 Python has libraries to read and write data to (almost) any system

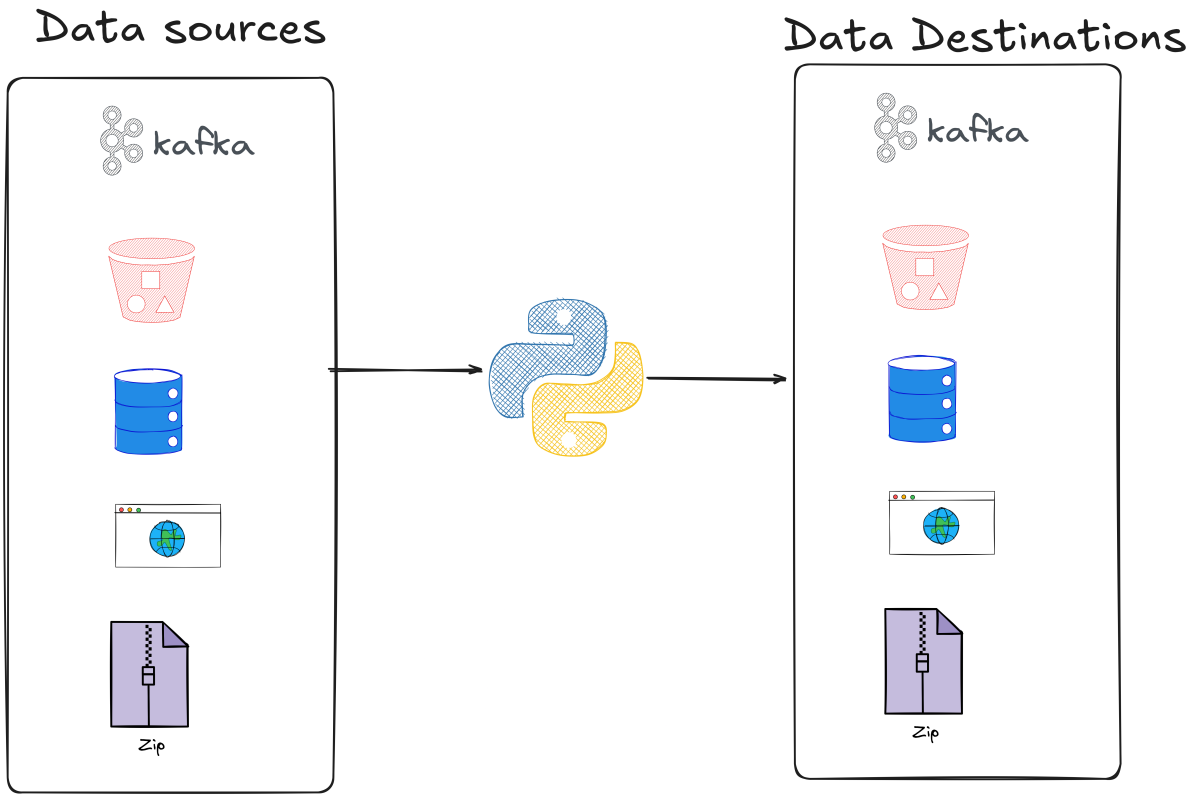


Figure 5.1: Python EL

Python has multiple libraries that enable reading from and writing to various systems. Almost all systems these days have a Python libraries to interact with it.

For data engineering this means that one can use Python to interact with any part of the stack. Let's look at the types of systems for reading and writing and how Python is used there:

1. **`Database drivers`**: These are libraries that you can use to connect to a database. Database drivers require you to use credentials to create a connection to your database.

Once you have the connection object, you can run queries, read data from your database in Python, etc. Some examples are psycopg2, sqlite3, duckdb, etc.

2. **`Cloud SDKs`**: Most cloud providers (AWS, GCP, Azure) provide their own SDK(Software Development Kit). You can use the SDK to work with any of the cloud services. In data pipelines, you would typically use the SDK to extract/load data from/to a cloud storage system(S3, GCP Cloud store, etc). Some examples of SDK are AWS, which has boto3; GCP, which has gsutil; etc.
3. **`APIs`**: Some systems expose data via APIs. Essentially, a server will accept an HTTPS request and return some data based on the parameters. Python has the popular `requests` library to work with APIs.
4. **`Files`**: Python enables you to read/write data into files with standard libraries(e.g., `csv`). Python has a plethora of libraries available for specialized files like XML, xlsx, parquet, etc.
5. **`SFTP/FTP`**: These are servers typically used to provide data to clients outside your company. Python has tools like paramiko, ftplib, etc., to access the data on these servers.
6. **`Queuing systems`**: These are systems that queue data (e.g., Kafka, AWS Kinesis, Redpanda, etc.). Python has libraries to read data from and write data to these systems, e.g., pykafka, etc.

Read data from the pokemon api `https://pokeapi.co/api/v2/pokemon/1` using requests library. Use the `get` method. Documentation reference

```
import requests
url = "https://pokeapi.co/api/v2/pokemon/1"
response = requests.get(url)
print(response.json())
```

Read data from a local file `./data/customer.csv` using the open function and csv reader.

```
import csv

data_location = "./data/customer.csv"
with open(data_location, "r", newline="") as csvfile:
    csvreader = csv.reader(csvfile)
    next(csvreader)  # Skip header row
    for row in csvreader:
        print(row)
        break
```

Use the BeautifulSoup library to parse the html data from the url `https://example.com` and find all the anchor html tags and print the hrefs.

```python
import requests
from bs4 import BeautifulSoup
url = 'https://example.com'

response = requests.get(url)
soup = BeautifulSoup(response.text, 'html.parser')
for link in soup.find_all('a'):
    print(link.get('href'))
```

5.1 Exercises

1. Fetch Data from API and Save Locally. Pull Pokemon data from the PokeAPI and save it to a local file. The local JSON file should contain Pokemon data for Bulbasaur (ID: 1)

```python
import requests
import json

data_api = "https://pokeapi.co/api/v2/pokemon/1/"
local_file = "pokemon_data.json"

# TODO:
# 1. Make a GET request to data_api;
# Ref docs: https://docs.python-requests.org/en/latest/user/quickstart/#response-content

# 2. Extract the JSON response
# Ref docs: https://docs.python-requests.org/en/latest/user/quickstart/#json-response-content

# 3. Write the JSON data to local_file

# Open a file writer
# Ref docs: https://docs.python.org/3/tutorial/inputoutput.html#methods-of-file-objects and

# Save json into the open file writer
# Ref docs https://docs.python.org/3/tutorial/inputoutput.html#saving-structured-data-with-j
```

2. Read and Display Local File Contents. Read the previously saved JSON file and print its name and id.

```
# TODO:
# 1. Use Python standard libraries to open local_file
# Ref docs:  https://docs.python.org/3/tutorial/inputoutput.html#methods-of-file-objects

# 2. Use json.load to convert it into a json
# Ref docs: https://docs.python.org/3/library/json.html#json.load

# 3. Read the name and id from the json, as you would from a dictionary
```

3. Parse Data and Insert into SQLite Database. Extract specific Pokemon attributes and store them in a SQLite database.

Table Schema:

```
CREATE TABLE pokemon (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    base_experience INTEGER
);
```

```
import sqlite3
import json

local_file = "pokemon_data.json"
database_file = "pokemon.db"

# Open a sqlite3 connection
conn = sqlite3.connect(database_file)
cursor = conn.cursor()

# make sure to commit after each interaction (that modifies our database table) https://docs

# TODO:
# 1. Create a SQLite3 table with columns: id, name, base_experience
# Ref docs: https://docs.python.org/3/library/sqlite3.html#sqlite3.Cursor.execute

# 2. Open and read the local_file using Python standard libraries
# 3. Parse the JSON data to extract id, name, and base_experience

# 4. Insert the extracted data into the SQLite table;
"""
cursor.execute('''
```

```
    INSERT INTO pokemon (id, name, base_experience)
    VALUES (?, ?, ?)
''', (pokemon_id, pokemon_name, base_experience))
"""

# 5. Verify insertion by querying the pokemon table and printing the data
# Ref docs: https://docs.python.org/3/library/sqlite3.html#sqlite3.Cursor.fetchall
```

5.2 Recommended reading

1. Extract data from API with Python
2. Data Pipeline Desing Pattern

6 Python has libraries to tell the data processing engine (Spark, Trino, Duckdb, Polars, etc) what to do

Almost every data processing system has a Python library that allows for interaction with it.

Some examples Pyspark API Snowflake Python Connector BigQuery Python Connector Trino Python Connector Polars Python API

The main types of data processing libraries are:

1. `Python standard libraries`: Python has a strong set of standard libraries for processing data. When using standard libraries, you'll usually be working with Python's native data structures (link). Some popular ones are csv, json, gzip etc.
2. `Dataframe libraries`: Libraries like pandas, polars, and Spark enable you to work with tabular data. These libraries utilize a DataFrame (Python's equivalent of a SQL table) and enable you to perform everything (and more) that you can with SQL. **Note** that some of these libraries are meant for data that can be processed in memory.
3. `Processing data on SQL via Python`: You can use database drivers and write SQL queries to process the data. The benefit of this approach is that you don't need to bring data into Python memory and offload it to the database.

Note: When we use systems like `Spark`, `Dask`, `Snowflake`, `BigQuery` to process data, you should note that we interact with them via Python. Data is processed by the external systems.

Let's work through some code examples that aim to clean a sample dataset. We will first perform the cleaning using Python's standard libraries and then repeat the process using PySpark dataframes.

6.1 Data processing with Python standard library

```
print(
    "################################################################################"
)
print("Use standard python libraries to do the transformations")
print(
    "################################################################################"
)
```

```
import csv

# Read data from CSV file into list of dict called data
data = []
with open("./sample_data.csv", "r", newline="") as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        data.append(row)
print(data[:2])
```

```
# Remove duplicate rows based on customer ID with set data structure

data_unique = []
customer_ids_seen = set()
for row in data:
    if row["Customer_ID"] not in customer_ids_seen:
        data_unique.append(row)
        customer_ids_seen.add(row["Customer_ID"])
    else:
        print(f'duplicate customer id {row["Customer_ID"]}')
```

```
# If an entry in data_unique does not have Age set it to 0,
# do the same for Purchase_Amount
for row in data_unique:
    if not row["Age"]:
        print(f'Customer {row["Customer_Name"]} does not have Age value')
        row["Age"] = 0
    if not row["Purchase_Amount"]:
        row["Purchase_Amount"] = 0.0

# remove outliers, assume we define outliers as any record with age over 100
# or age under 0
data_cleaned = [
    row
```

```
    for row in data_unique
    if int(row["Age"]) <= 100 and float(row["Purchase_Amount"]) <= 1000
]

# convert the Gender column to a binary format (0 for Female, 1 for Male)
for row in data_cleaned:
    if row["Gender"] == "Female":
        row["Gender"] = 0
    elif row["Gender"] == "Male":
        row["Gender"] = 1

# split the Customer_Name column into separate First_Name and Last_Name
# Assume customer_name is of the format First_Name Last_name
for row in data_cleaned:
    first_name, last_name = row["Customer_Name"].split(" ", 1)
    row["First_Name"] = first_name
    row["Last_Name"] = last_name

print(data_cleaned[:3])
```

```
# calculate the total purchase amount for each Gender
from collections import defaultdict # read about defaultdict vs normal dict in Python
total_purchase_by_gender = defaultdict(float)
for row in data_cleaned:
    total_purchase_by_gender[row["Gender"]] += float(row["Purchase_Amount"])

# Calculate the average purchase amount by Age group, note we define age groups as
# shown below
age_groups = {"18-30": [], "31-40": [], "41-50": [], "51-60": [], "61-70": []}
for row in data_cleaned:
    age = int(row["Age"])
    if age <= 30:
        age_groups["18-30"].append(float(row["Purchase_Amount"]))
    elif age <= 40:
        age_groups["31-40"].append(float(row["Purchase_Amount"]))
    elif age <= 50:
        age_groups["41-50"].append(float(row["Purchase_Amount"]))
    elif age <= 60:
        age_groups["51-60"].append(float(row["Purchase_Amount"]))
    else:
        age_groups["61-70"].append(float(row["Purchase_Amount"]))
```

```
average_purchase_by_age_group = {
    group: sum(amounts) / len(amounts) for group, amounts in age_groups.items()
}
```

```
print("Total purchase amount by Gender:", total_purchase_by_gender)
print("Average purchase amount by Age group:", average_purchase_by_age_group)
```

6.2 Data processing with PySpark

```
spark
```

```
print(
    "################################################################################"
)
print("Use PySpark DataFrame API to do the transformations")
print(
    "################################################################################"
)
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, coalesce, lit, when, split, sum as spark_sum, avg, reg
from pyspark.sql.types import StructType, StructField, IntegerType, StringType, FloatType, Da

schema = StructType([
    StructField("Customer_ID", IntegerType(), True),
    StructField("Customer_Name", StringType(), True),
    StructField("Age", IntegerType(), True),
    StructField("Gender", StringType(), True),
    StructField("Purchase_Amount", FloatType(), True),
    StructField("Purchase_Date", DateType(), True)
])

# Read data from CSV file into DataFrame
data = spark.read \
    .option("header", "true") \
    .option("inferSchema", "false") \
    .schema(schema) \
    .csv("./sample_data.csv")
```

```
# Question: How do you remove duplicate rows based on customer ID in PySpark?
data_unique = data.dropDuplicates()

# Question: How do you handle missing values by replacing them with 0 in PySpark?
data_cleaned_missing = data_unique.select(
    col("Customer_ID"),
    col("Customer_Name"),
    coalesce(col("Age"), lit(0)).alias("Age"),
    col("Gender"),
    coalesce(col("Purchase_Amount"), lit(0.0)).alias("Purchase_Amount"),
    col("Purchase_Date")
)

# Question: How do you remove outliers (e.g., age > 100 or purchase amount > 1000) in PySpark
data_cleaned_outliers = data_cleaned_missing.filter(
    (col("Age") <= 100) & (col("Purchase_Amount") <= 1000)
)

# Question: How do you convert the Gender column to a binary format (0 for Female, 1 for Male
data_cleaned_gender = data_cleaned_outliers.withColumn(
    "Gender_Binary",
    when(col("Gender") == "Female", 0).otherwise(1)
)

# Question: How do you split the Customer_Name column into separate First_Name and Last_Name
data_cleaned = data_cleaned_gender.select(
    col("Customer_ID"),
    split(col("Customer_Name"), " ").getItem(0).alias("First_Name"),
    split(col("Customer_Name"), " ").getItem(1).alias("Last_Name"),
    col("Age"),
    col("Gender_Binary"),
    col("Purchase_Amount"),
    col("Purchase_Date")
)
```

```
# Question: How do you calculate the total purchase amount by Gender in PySpark?
total_purchase_by_gender = data_cleaned.groupBy("Gender_Binary") \
    .agg(spark_sum("Purchase_Amount").alias("Total_Purchase_Amount")) \
    .collect()
```

```
# Question: How do you calculate the average purchase amount by Age group in PySpark?
average_purchase_by_age_group = data_cleaned.withColumn(
    "Age_Group",
    when((col("Age") >= 18) & (col("Age") <= 30), "18-30")
    .when((col("Age") >= 31) & (col("Age") <= 40), "31-40")
    .when((col("Age") >= 41) & (col("Age") <= 50), "41-50")
    .when((col("Age") >= 51) & (col("Age") <= 60), "51-60")
    .otherwise("61-70")
).groupBy("Age_Group") \
    .agg(avg("Purchase_Amount").alias("Average_Purchase_Amount")) \
    .collect()
```

```
# Question: How do you print the results for total purchase amount by Gender and average pur
print("====================== Results ======================")
print("Total purchase amount by Gender:")
for row in total_purchase_by_gender:
    print(f"Gender_Binary: {row['Gender_Binary']}, Total_Purchase_Amount: {row['Total_Purchas

print("Average purchase amount by Age group:")
for row in average_purchase_by_age_group:
    print(f"Age_Group: {row['Age_Group']}, Average_Purchase_Amount: {row['Average_Purchase_A
```

```
# Optional: Show DataFrame contents for verification
print("\n====================== Data Preview ======================")
print("Final cleaned data:")
data_cleaned.show(10)
```

6.3 Exercises

1. What is the total purchase amount for each gender in the dataset? Use the `data_cleaned_gender` dataframe and Group the data by gender and calculate the sum of all purchase amounts.
2. What is the average purchase amount for different age groups? Use the `data_cleaned` dataframe and create age group categories (18-30, 31-40, 41-50, 51-60, 61-70) and calculate the mean purchase amount for each group.

6.4 Recommended reading

1. SQL v Python for data transformations

Part III

Data modeling is the process of getting data ready for analyticsUse SQL to transform data

As a data engineer your key objective is to enable stakeholders to be able to use data effectively to answer questions about business performance and predict how a business may perform in the future.

Most companies production system store data in denormalized tables across multiple microservices, which make analytics hard as the data user will now be required to join across multiple tables. If your company uses a microservice architetcure this becomes impossible.

Analytical querying often require processing large amounts of data which can have a significant impact on the database performance which is usually unacceptable for production systems.

Production system ususaly only stores current state and does not store a log of changes, which is typically necessary for historical analysis

Most companies produce event(click tracking, e-commerce ordering, server logs monitoring, etc) which are usually too large to be stored and queried efficiently in a production database

These are some of the reasons you need a warehouse system to be able to analyze historical information.

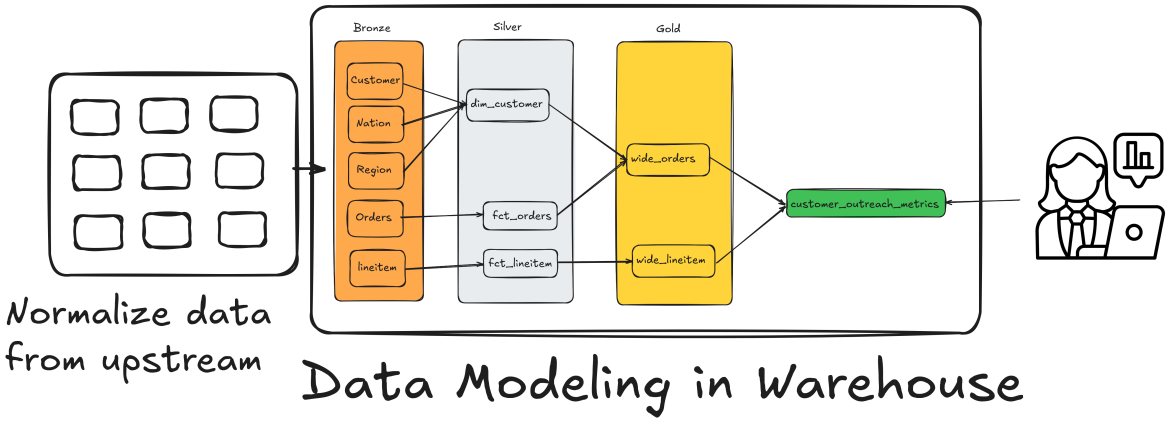


Figure 6.1: Data flow

Let's assume that we are enabling business users to answer questions about the bike parts seller TPCH business.

Loading...

7 Data warehouse contains historical data and is used to analyze business performance

As our bike part seller business grows (that we saw in the setup chapter), we will want to analyze data for various business goals.

Sellers will also want to analyze performance and trends to optimize their inventory. Some of these questions may be

1. Who are the top 10 sellers, who have sold the most items?
2. What is the average time for an order to be fulfilled?
3. Cluster the customers who purchased the same/similar items together.
4. Create a seller dashboard that shows the top-performing items per seller

These questions ask about things that happened in the past, require reading a large amount of data, and aggregating data to get a result. Most companies generate large amounts of data and need to analyze it effectively. Starting out by using your company's backend database is a viable approach. As the size of data and complexity of your systems and queries increase, you will want to use databases specifically designed for large-scale data analytics, known as OLAP systems.

A data warehouse is a database that stores all your company's historical data.

While your upstream systems can be a single service, or multiple microservices and typically does not store historical change, a warehouse is designed to be the single source of truth of all the historical information you'd need about your company. See the simple example below:

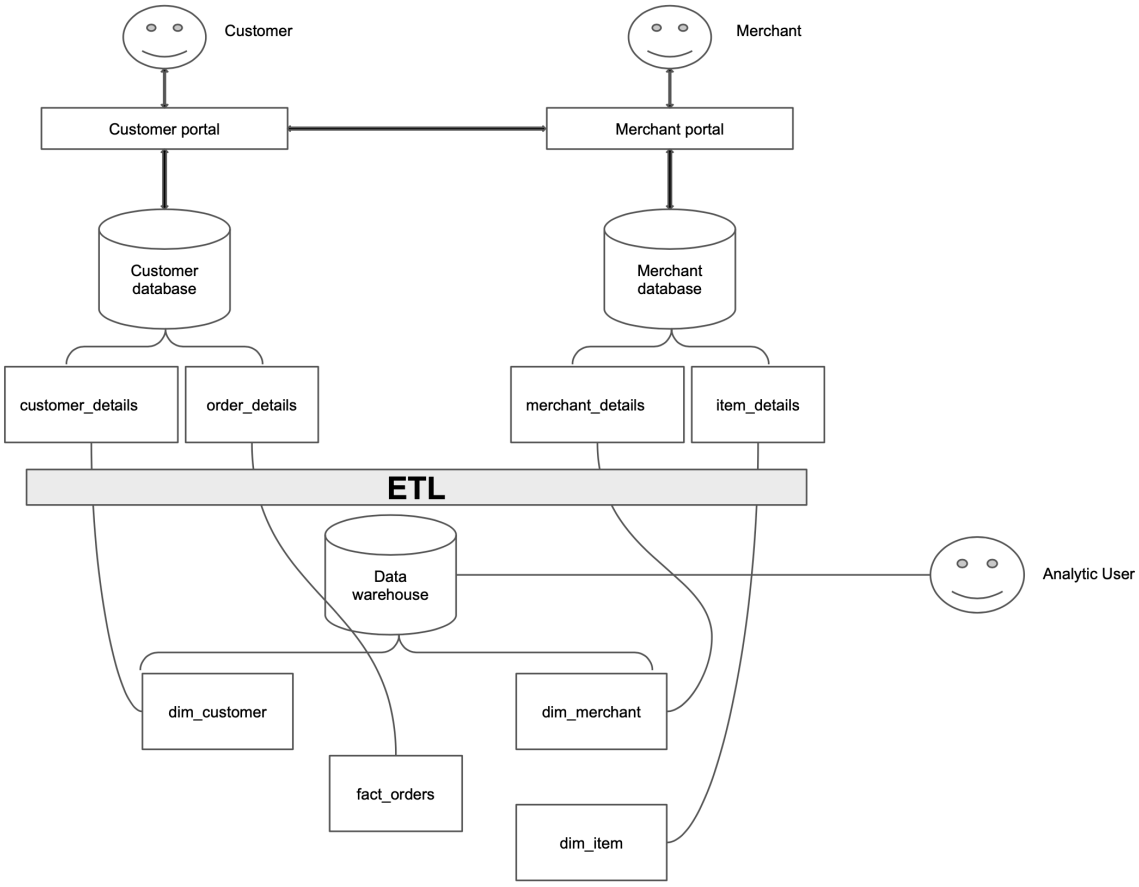


Figure 7.1: basic Data Warehouse

7.1 OLTP vs OLAP-based data warehouses

There are two primary types of databases: OLTP (Online Transaction Processing) and OLAP (Online Analytical Processing). Their differences are shown below.

	OLTP	OLAP
Stands for	Online transaction processing	Online analytical processing
Usage pattern	Optimized for fast CRUD(create, read, update, delete) of a small number of rows	Optimized for running `select c1, c2, sum(c3),.. where .. group by` on a large number of rows (aka analytical queries), and ingesting large amounts of data via bulk import or event stream

	OLTP	OLAP
Storage type	Row oriented	Column-oriented
Data modeling	Data modeling is based on normalization	Data modeling is based on denormalization. Some popular ones are dimensional modeling and data vaults
Data state	Represents current state of the data	Contains historical events that have already happened
Data size	Gigabytes to Terabytes	Terabytes and above
Example database	MySQL, Postgres, etc	Clickhouse, AWS Redshift, Snowflake, GCP BigQuery, etc

Note Apache Spark started as a pure data processing system, and over time, with the increased need for structure, it introduced capabilities to manage tables.

7.2 Column encoding enables efficient processing of a small number of columns from a wide table

The significant improvement in analytical queries on OLAP is attributed to its column-store technique. Let's consider a table `items`, with the data shown below.

item_id	item_name	item_type	item_price	datetime_created	datetime_updated
1	item_1	gaming	10	'2021-10-02 00:00:00'	'2021-11-02 13:00:00'
2	item_2	gaming	20	'2021-10-02 01:00:00'	'2021-11-02 14:00:00'
3	item_3	biking	30	'2021-10-02 02:00:00'	'2021-11-02 15:00:00'
4	item_4	surfing	40	'2021-10-02 03:00:00'	'2021-11-02 16:00:00'
5	item_5	biking	50	'2021-10-02 04:00:00'	'2021-11-02 17:00:00'

Let's see how this table will be stored in a row- and column-oriented storage system. **Data is stored as continuous pages (a group of records) on the disk.**

Row-oriented storage:

Let's assume that there is one row per page.

```
Page 1: [1,item_1,gaming,10,'2021-10-02 00:00:00','2021-11-02 13:00:00'],
Page 2: [2,item_2,gaming,20,'2021-10-02 01:00:00','2021-11-02 14:00:00']
Page 3: [3,item_3,biking,30, '2021-10-02 02:00:00','2021-11-02 15:00:00'],
Page 4: [4,item_4,surfing,40, '2021-10-02 03:00:00','2021-11-02 16:00:00'],
Page 5: [5,item_5,biking,50, '2021-10-02 04:00:00','2021-11-02 17:00:00']
```

Column-oriented storage:

Let's assume that there is one column per page.

```
Page 1: [1,2,3,4,5],
Page 2: [item_1,item_2,item_3,item_4,item_5],
Page 3: [gaming,gaming,biking,surfing,biking],
Page 4: [10,20,30,40,50],
Page 5: ['2021-10-02 00:00:00','2021-10-02 01:00:00','2021-10-02 02:00:00','2021-10-02 03:00
Page 6: ['2021-11-02 13:00:00','2021-11-02 14:00:00','2021-11-02 15:00:00','2021-11-02 16:00
```

Let's see how a simple analytical query will be executed.

```sql
SELECT item_type,
    SUM(price) total_price
FROM items
GROUP BY item_type;
```

In a **row-oriented database**

1. All the pages will need to be loaded into memory
2. Sum `price` column for the same `item_type` values

In a **column-oriented database**

1. Only pages 3 and 4 will need to be loaded into memory. The information on pages 3 and 4, including item_type and total_price, will be encoded in the column-oriented file and also stored in an OLTP called metadata db.
2. Sum `price` column for the same `item_type` values

As you can see from this approach, we only need to read 2 pages in a column-oriented database, compared to 5 pages in a row-oriented database. In addition to this, a column-oriented database also provides

1. Better compression, as similar data types follow each other and can be compressed more efficiently.
2. Vectorized processing

All of these features make a column-oriented database an excellent choice for storing and analyzing large amounts of data.

7.3 Recommended reading

1. Data Lake v Data Warehouse
2. What is a data warehouse
3. 4 patterns to load data into a data warehouse

8 Data warehouse modeling (Kimball) is based off of 2 types of tables: Fact and dimensions

Data warehousing is more than just moving data to an OLAP database; the key part of data warehousing is data modeling.

Data modelling ensures that the data is suited for our specific use case(data analytics is our use case)

The data from source systems (your company's backend, API data pulls, etc) is usually normalized & modeled for effective CRUD of a small number of rows at a time.

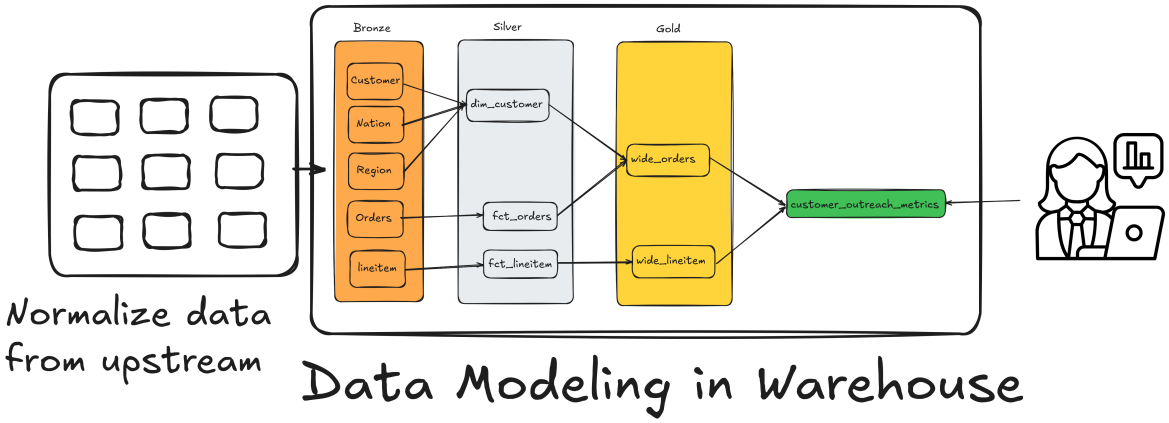


Figure 8.1: Data Flow

However, in a data warehouse, we usually want to aggregate a large number of rows based on a set of columns. For this use case, Kimball's Dimensional model has been the most standard practice.

In this chapter, we will cover the basics of Kimball design with an eye towards modern infrastructure and expand on them.

8.1 Facts represent events that occured & dimension the entities to which events occur.

A data warehouse is a database that stores your company's historical data. The main types of tables you need to create to power analytics are:

1. `Dimension`: Each row in a dimension table represents a business entity that is important to the business. For example, we have a `customer` dimension table, where each row represents an individual customer. Other examples of dimension tables are `supplier` & `part` tables.
2. `Facts`: Each row in a fact table represents a business process that occurred. E.g., in our data warehouse, each row in the `orders` fact table will represent an individual order, and each row in the `lineitem` fact table will represent an item sold as part of an order. Each fact row will have a unique identifier; in our case, it's `orderkey` for orders and a combination of `orderkey & linenumber` for lineitem.

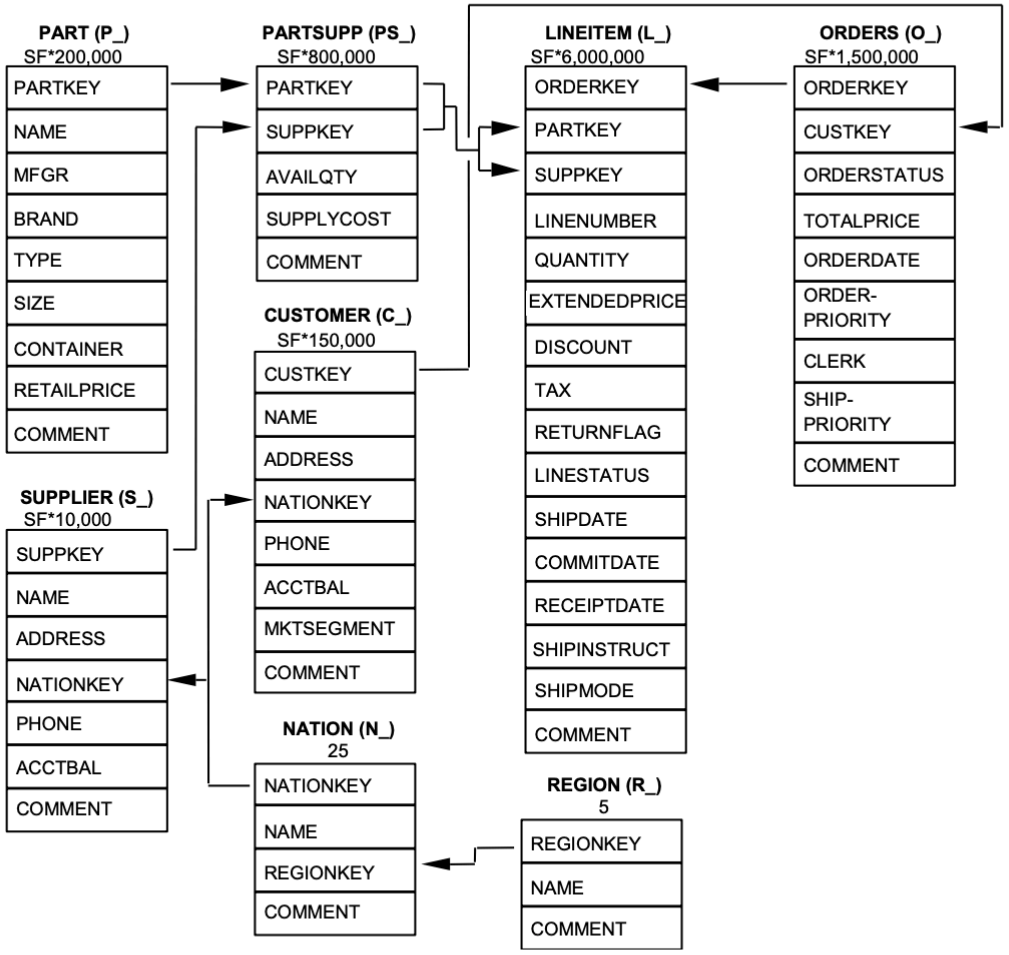
A table's **`grain (aka granularity, level)`** refers to what a row of that table represents. For example, in our checkout process, we can have two fact tables: one for the order and another for the individual items in the order.

The items table will have one row for each item purchased, whereas the order table will have one row for each order placed. Our TPCH data is modelled into facts and dimensions as shown below:

1.2 Database Entities, Relationships, and Characteristics

The components of the TPC-H database are defined to consist of eight separate and individual tables (the Base Tables). The relationships between columns of these tables are illustrated in Figure 2: The TPC-H Schema.

Figure 2: The TPC-H Schema



Legend:

- The parentheses following each table name contain the prefix of the column names for that table;
- The arrows point in the direction of the one-to-many relationships between tables;
- The number/formula below each table name represents the cardinality (number of rows) of the table. Some are factored by SF, the Scale Factor, to obtain the chosen database size. The cardinality for the LINEITEM table is approximate (see Clause 4.2.5).

Figure 8.2: TPCH data model

```
%%sql
use prod.db
```

```
%%sql
-- calculating the totalprice of an order (with orderkey = 1) from it's individual items
SELECT
    l_orderkey,
    round( sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)),
        2
    ) AS totalprice
FROM
    lineitem
WHERE
    l_orderkey = 1
GROUP BY
    l_orderkey;
```

```
%%sql
-- The totalprice of an order (with orderkey = 1)
SELECT
    o_orderkey,
    o_totalprice
FROM
    orders
WHERE
    o_orderkey = 1;
```

Note: If you notice the slight difference in the decimal digits, it's due to using a `double` datatype, which is an inexact data type.

We can see how the `lineitem` table can be "rolled up" to get the data in the `orders` table. However, having just the `orders` table is insufficient, as the `lineitem` table provides us with individual item details, including discount and quantity information.

8.2 Popular dimension types: Full Snapshot & SCD2

Kimball defines Seven types of dimensional models; however, two of them are the most widely used.

1. **`Full snapshot`** In this type of dimension, the entire dimension table is overwritten (or inserted with a specific load date) each run. Typically, each run creates a new copy while retaining the older copy for a specific period (e.g., 6 months). With the decreasing cost of storage, this is an acceptable tradeoff, especially since it is easy to implement and enables the users to go back in time.
2. **`Slowly Changing Dimension Type 2, SCD2`** In this type of dimension, every change to the dimension's entity (e.g, customer attribute change in a customer dimension) will result in a new row.

And every row will contain:

1. **valid_from**: A timestamp column indicates the time from when this version of the customer attributes was valid.
2. **valid_to**: A timestamp column indicates the time up to which this version of the customer attributes was valid.
3. **is_current**: A boolean flag indicating if this row is the current state of the customer.

Consider this example where a supplier's state changes from CA to IL (ref: Wiki)

Supplier_Key	Supplier_Code	Supplier_Name	Supplier_State	Start_Date	End_Date	is_current
123	ABC	Acme Supply Co	CA	2000-01-01T00:00:00	2004-12-22T00:00:00	0
123	ABC	Acme Supply Co	IL	2004-12-22T00:00:00	NULL	1

According to Kimball's method, you are supposed to create a surrogate key (think of an ID that is continually increasing) for each row in the dimension and enrich the fact table with this dimension surrogate key to enable efficient joins.

However, in most modern data systems, the fact data flows in without too much delay and has the dimension's unique key from upstream data. While the dimension tables may take a while to create.

With advances in data processing and partitioning formats, most companies skip the surrogate key modeling methodology and instead join based on the key from the upstream dataset and the time. See this how-to article for an example.

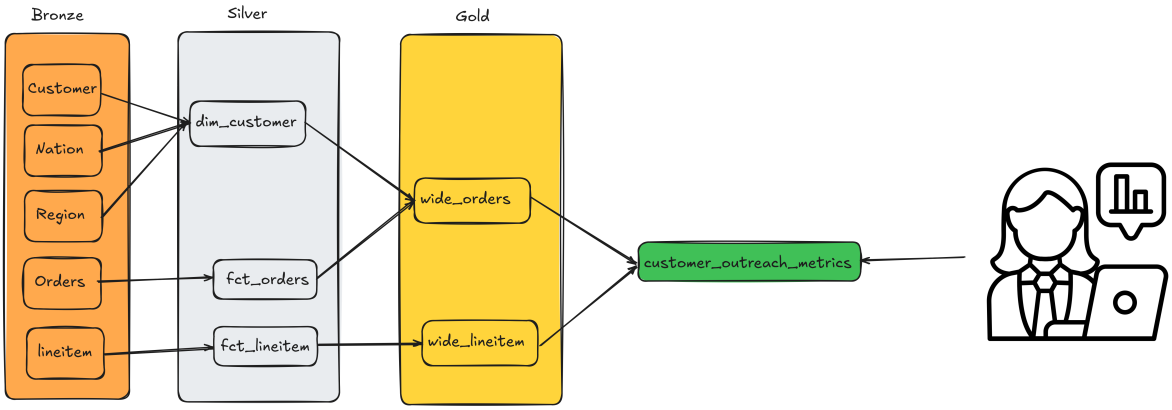


Figure 8.3: Data Flow

8.3 One Big Table (OBT) is a fact table left-joined with all its dimensions

As the number of facts and dimensions increases, you will notice that most queries used by end users to retrieve data utilize the same tables and joins.

In this scenario, the expensive reprocessing of data can be avoided by creating an OBT. In an OBT, you left-join all the dimensions into a fact table. This OBT can then be used to aggregate data to different grains as needed for end-user reporting.

Note that the OBT should have the exact grain as the lowest grain of the fact table that it is based on. In our bike-part seller warehouse, we can create an OBT for orders by joining all the tables to the lineitem table.

```
select
    f.*,
    d.other_attributes
from fct_orders as f
left join dim_customer as d on f.customer_key = d.customer_key
```

From the above image, we can see that the OBT tables are **`wide_orders`** and **`wide_lineitem`**.

8.4 Summary or pre-aggregated tables are stakeholder-team-specific tables built for reporting

Stakeholders often require data aggregated at various grains and similar metrics. Creating pre-aggregated or summary tables enables the generation of these reports for stakeholders, allowing them to select from the table without needing to recompute metrics.

The summary table has two key benefits.

1. Consistent metric definition, as the data engineering will keep the metric definition in the code base, vs each stakeholder using a slightly different version and ending up with different numbers
2. Avoid unnecessary recomputation, as multiple stakeholders can now use the same table

However, the downside is that the data may not be as fresh as what a stakeholder would obtain if they were to write a query.

Here is a simple example, assuming `wide_lineitem` is an OBT.

```
select
    order_key,
    COUNT(line_number) as num_lineitems
from wide_lineitem
group by order_key
```

From the above image, we can see that the summary table is **`customer_outreach_metrics`**.

8.5 Exercises

1. What are the fact tables in our TPCH data model?
2. What are the dimension tables in our TPCH data model?
3. What source tables in the TPCH data model would you consider to create a customer dimension table?

8.6 Recommended reading

1. Ensuring consitent metrics for your data
2. Avoiding messy data in your warehouse
3. Data lake v warehouse
4. What is a data warehouse
5. Create SCD2 table with MERGE INTO

9 Most companies use the multi-hop architecture

In the previous chapter, we learnt about the different types of tables in data warehouse modeling: fact, dimension, OBT, and summary tables.

But which table should be built first, and which types of tables are built based on other types of tables?

Note For a pipeline/transformation function/table, inputs to it are called **upstream**, and outputs from it are called **downstream**.

This is where standard data flow models provide us with a clear guideline on how to transform our data in layers, enabling easy maintenance of warehouse tables.

Most industry-standard patterns follow a 3-hop (or layered) architecture.

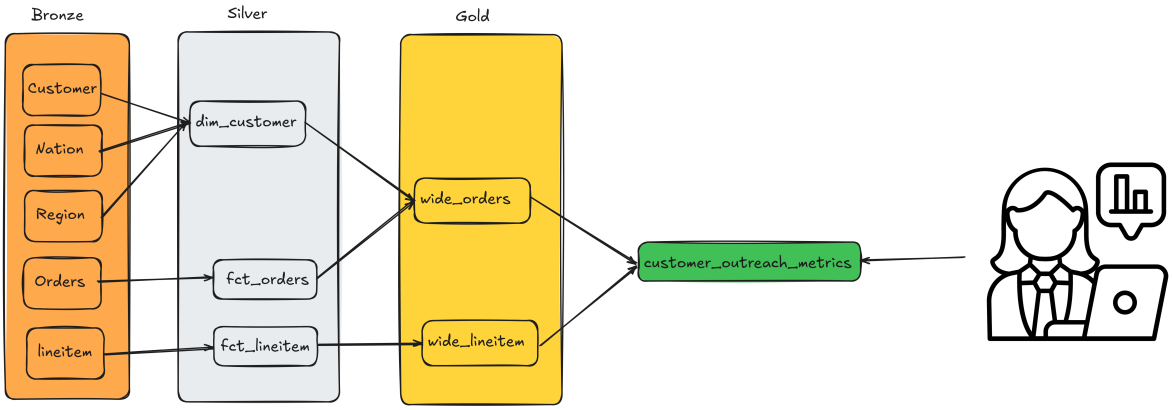


Figure 9.1: Data architecture

They are

1. **`Base/Raw layer`** stores data from upstream sources as is.
2. **`Bronze/Stage layer`** In this layer, the raw data is lightly cleaned by standardizing on column naming conventions and assigning the right data types to the data.
3. **`Silver/Intermediate layer`** In this layer, data from the bronze/stage is transformed into facts and dimensions.

4. **`Gold/Marts layer`** In this layer, the modelled data is pre-aggregated and summarized as required by the end user. This layer ensures that the same metric definitions are consistently used across business use cases.

Note The boundaries of fact, dimension, and OBT between silver and gold vary by company. Still, the general idea of facts and dimensions flowing into OBT and then into pre-aggregated tables remains consistent.

Most frameworks/tools propose their version of the 3-hop architecture: 1. Apache Spark: Medallion architecture uses the bronze/silver/gold naming. 2. dbt: Project Structure uses the source/stage/core/mart naming.

Shown below is the dbt UI (which we cover in the **dbt** chapter) on how TPCH data can be modelled based on dbt's 3-hop architecture:

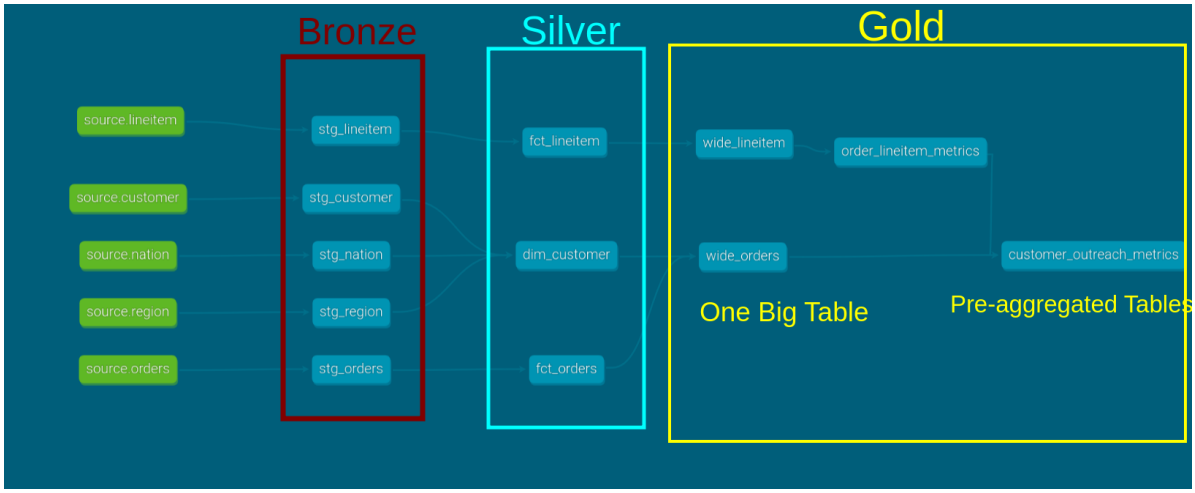


Figure 9.2: Data architecture

At larger companies, multiple teams work on different layers of the organization. A data ingestion team may bring the data into the bronze layer, and other teams may build their own silver and gold tables as necessary.

Part IV

Working in a team

Data engineering has multiple components and its crucial that you have the same environment that runs in production is also aavailable to the dta engineers to test and develop in.

In this section we will see how docker is used to simulate the same running environment across multiple systems.

10 Docker recreates the same environment for your code in any machine

You can think of Docker as running a separate OS (not precisely, but close enough) called **containers** on a machine.

Docker provides the ability to replicate the OS and its packages (e.g., Python modules) across machines, so you don't encounter "hey, that worked on my computer" type issues.

10.1 A Docker image is a blueprint for your container

An image is a blueprint for creating your Docker container. In an image, you can define the modules to install, variables to set, etc, and then use the image to create multiple containers.

Let's consider our **Airflow Dockerfile**:

```
FROM apache/airflow:2.9.2
RUN pip install uv

COPY requirements.txt $AIRFLOW_HOME
RUN uv pip install -r $AIRFLOW_HOME/requirements.txt

COPY run_ddl.py $AIRFLOW_HOME
COPY generate_data.py $AIRFLOW_HOME

User root

RUN apt-get update && \
    apt-get install -y --no-install-recommends \
    default-jdk

RUN curl https://archive.apache.org/dist/spark/spark-3.5.1/spark-3.5.1-bin-hadoop3.tgz -o spa

# Change permissions of the downloaded tarball
RUN chmod 755 spark-3.5.1-bin-hadoop3.tgz
```

```
# Create the target directory and extract the tarball to it
RUN mkdir -p /opt/spark && tar xvzf spark-3.5.1-bin-hadoop3.tgz --directory /opt/spark --stri

ENV JAVA_HOME='/usr/lib/jvm/java-17-openjdk-amd64'
ENV PATH=$PATH:$JAVA_HOME/bin
ENV SPARK_HOME='/opt/spark'
ENV PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
```

The commands in the Docker image are run in order. Let's go over the key commands:

1. **`FROM`**: We need a base operating system on which to set our configurations. We can also utilize existing Docker images available on Docker Hub (an online store where people can upload and download images from) and build upon them. In our example, we use the official Airflow Docker image.
2. **`COPY`**: Copy is used to copy files or folders from our local filesystem to the image. In our image, we copy the Python requirements.txt file, data generation scripts, and table creation (DDL) scripts.
3. **`ENV`**: This command sets the image's environment variables. In our example, we put the Java and Spark Paths necessary to run Spark inside our container.
4. **`ENTRYPOINT`**: The entrypoint command executes a script when the image starts. In our example, we don't use one. Still, it is a common practice to have a script start necessary programs using an entry point script.

10.2 Sync data & code between a container and your local filesystem with volume mounts

When we are developing, we'd want to make changes to the code and see its impact immediately. While you can use `COPY` to copy your code when building a Docker image, it will not reflect changes in real-time, and you will have to rebuild your container each time you need to change your code.

In cases where you want data/code to sync two ways between your local machine and the running Docker container, use mounted volumes.

In addition to syncing local files, volumes can also sync files between running containers.

In our **docker-compose.yml** (which we will go over below), we mount the following folders in our local filesystem into a specified path inside a container.

```
volumes:
    - ./dags:/opt/airflow/dags
    - ./data:/opt/airflow/data
    - ./visualization:/opt/airflow/visualization
    - ./logs:/opt/airflow/logs
    - ./plugins:/opt/airflow/plugins
    - ./tests:/opt/airflow/tests
    - ./temp:/opt/airflow/temp
    - ./tpch_analytics:/opt/airflow/tpch_analytics
```

10.3 Ports to accept connections

Most data systems also expose runtime information, documentation, UI, and other components via ports. We have to inform Docker which ports to keep open so that they are accessible from the "outside", in our case, your local browser.

In our docker-compose.yml (add: link)(which we will go over below), we keep the following port open

```
ports:
      - 8080:8080
      - 8081:8081
```

The `8080` port is for the Airflow UI, and `8081` is for the dbt docs.

Note In - 8080:8080, the RHS (right-hand side) `8080` represents the port inside the container, and the LHS (left hand side) 8080 indicates the port that the internal one maps to on your local machine.

Shown below is another example of how ports and volumes enable communication and data sharing respectively across containers and your os:

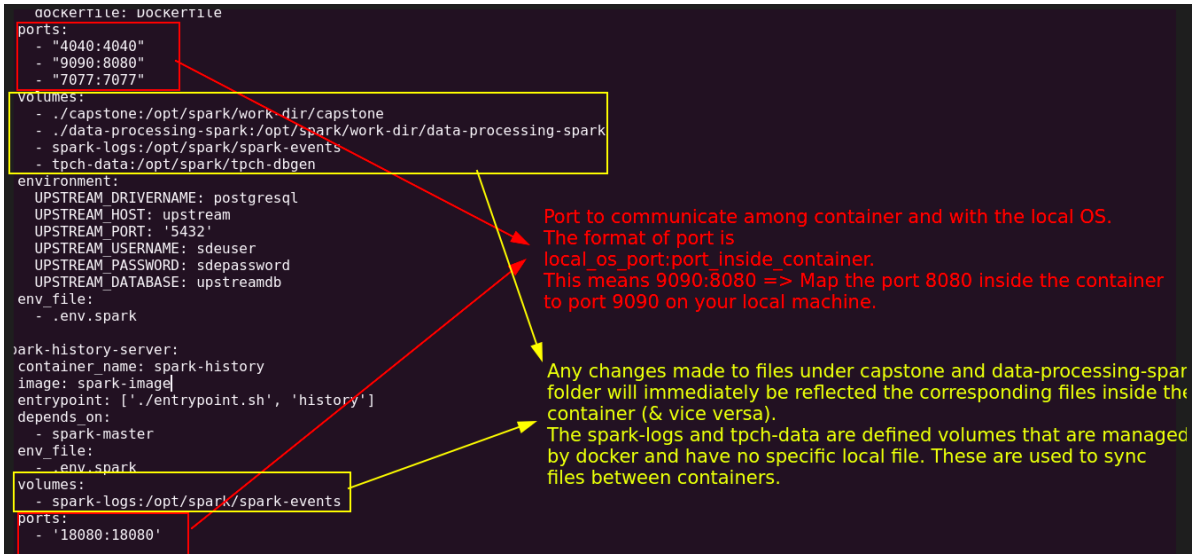


Figure 10.1: docker port

10.4 Docker cli to start a container and docker compose to coordinate starting multiple containers

We can use the docker cli to start containers based on an image. Let's look at an example. To start a simple metabase dashboard container, we can use the following:

```
docker run -d --name dashboard -p 3000:3000 metabase/metabase
```

The docker command will look for containers on your local machine and then in docker hub matching the name `metabase/metabase`.

However, with most data systems, we will need to ensure multiple systems are running. While we can use the docker CLI to do this, a better option is to use docker compose to orchestrate the different containers required.

With docker compose, we can define all our settings in one file and ensure that they are started in the order we prefer.

With our **docker-compose.yml** defined, starting our containers is a simple command, as shown below:

```
docker compose up -d
```

The command will, by default, look for a file called `docker-compose.yml` in the directory in which the command is run.

Let's take a look at our docker-compose.yml file.

We have six services (collection of one or more containers):

1. **`Postgres`** to serve as the backend for our Airflow
2. **`Airflow Webserver`** for the Airflow UI
3. **`Airflow Scheduler`** to schedule and run our Airflow jobs
4. **`Airflow init`** is a short-lived container that creates all the PostgreSQL tables used to store run information by the Airflow system
5. **`Minio`** to serve as a local open-source S3 alternative.

Since all our Airflow-based services need to have standard settings, we define a `x-airflow-common:` at the top with settings that are injected into the necessary services as such

```
 airflow-webserver:
    <<: *airflow-common
```

Our `docker-compose.yml` file:

```
version: '3'
x-airflow-common:
  &airflow-common
  build:
    context: ./containers/airflow/
  environment:
    &airflow-common-env
    AIRFLOW__CORE__EXECUTOR: LocalExecutor
    AIRFLOW__CORE__SQL_ALCHEMY_CONN: postgresql+psycopg2://airflow:airflow@postgres/airflow
    AIRFLOW__CORE__FERNET_KEY: ''
    AIRFLOW__CORE__DAGS_ARE_PAUSED_AT_CREATION: 'true'
    AIRFLOW__CORE__LOAD_EXAMPLES: 'false'
    AIRFLOW__API__AUTH_BACKEND: 'airflow.api.auth.backend.basic_auth'
    AIRFLOW_CONN_POSTGRES_DEFAULT: postgres://airflow:airflow@postgres:5432/airflow
  volumes:
    - ./dags:/opt/airflow/dags
    - ./data:/opt/airflow/data
    - ./visualization:/opt/airflow/visualization
    - ./logs:/opt/airflow/logs
    - ./plugins:/opt/airflow/plugins
    - ./tests:/opt/airflow/tests
```

```
    - ./temp:/opt/airflow/temp
    - ./tpch_analytics:/opt/airflow/tpch_analytics
  user: "${AIRFLOW_UID:-50000}:${AIRFLOW_GID:-50000}"
  depends_on:
    postgres:
      condition: service_healthy

services:
  postgres:
    container_name: postgres
    image: postgres:16
    environment:
      POSTGRES_USER: airflow
      POSTGRES_PASSWORD: airflow
      POSTGRES_DB: airflow
    volumes:
      - ./data:/input_data
    healthcheck:
      test: [ "CMD", "pg_isready", "-U", "airflow" ]
      interval: 5s
      retries: 5
    restart: always
    ports:
      - "5432:5432"

  airflow-webserver:
    <<: *airflow-common
    container_name: webserver
    command: webserver
    ports:
      - 8080:8080
      - 8081:8081
    healthcheck:
      test:
        [
          "CMD",
          "curl",
          "--fail",
          "http://localhost:8080/health"
        ]
      interval: 10s
      timeout: 10s
```

```
    retries: 5
  restart: always

airflow-scheduler:
  <<: *airflow-common
  container_name: scheduler
  command: scheduler
  ports:
    - 10000:10000
  healthcheck:
    test:
      [
        "CMD-SHELL",
        'airflow jobs check --job-type SchedulerJob --hostname "$${HOSTNAME}"'
      ]
    interval: 10s
    timeout: 10s
    retries: 5
  restart: always

airflow-init:
  <<: *airflow-common
  command: version
  environment:
    <<: *airflow-common-env
    _AIRFLOW_DB_UPGRADE: 'true'
    _AIRFLOW_WWW_USER_CREATE: 'true'
    _AIRFLOW_WWW_USER_USERNAME: ${_AIRFLOW_WWW_USER_USERNAME:-airflow}
    _AIRFLOW_WWW_USER_PASSWORD: ${_AIRFLOW_WWW_USER_PASSWORD:-airflow}

minio:
  image: 'minio/minio:latest'
  hostname: minio
  container_name: minio
  ports:
    - '9000:9000'
    - '9001:9001'
  environment:
    MINIO_ACCESS_KEY: minio
    MINIO_SECRET_KEY: minio123
    MINIO_ROOT_USER: minio
    MINIO_ROOT_PASSWORD: minio123
```

```
    command: server --console-address ":9001" /data
```

10.5 Executing commands in your Docker container

Using the exec command, you can submit commands to be run in a specific container. For example, we can use the following to open a bash terminal in our `scheduler` container. Note that the scheduler is based on the `container_name` setting.

```
docker exec -ti scheduler bash
# You will be in the master container bash shell
# try some commands
pwd
exit # exit the container
```

Note that the `-ti` indicates that this will be run in an interactive mode. As shown below, we can run a command in non-interactive mode and obtain an output.

```
docker exec scheduler echo hello
# prints hello
```

Part V

Scheduler defines when & Orchestrator defines how to, run your data pipelines

Schedulers define when to start your data pipeline, such as cron or Airflow.

Orchestrators define the order in which the tasks of a data pipeline should run. For example, extract before transform, complex branching logic, and executing across multiple systems, such as Spark and Snowflake. E.g., dbt-core, Airflow, etc

Our Airflow, dbt, and capstone project infrastructure is in a separate folder to keep our setup simple. When you are in the project directory, stop any running container as shown below.

```
data_engineering_for_beginners_code/> docker compose down
data_engineering_for_beginners_code/> cd airflow
data_engineering_for_beginners_code/airflow> make restart
```

You can open Airflow UI at http://localhost:8080 and log in with `airflow` as username and password. In the Airflow UI, you can run the dag.

After the dag is run, in the terminal, run `make dbt-docs` for dbt to serve the docs, which is viewable by going to http://localhost:8081.

You can stop the containers & return to the parent directory as shown below:

```
make down
cd ..
```

The `Makefile` contains a list of shortcuts for lengthy commands. Let's look at our Makefile below.

```
#####################################################################################################
# Setup containers to run Airflow

docker-spin-up:
    docker compose build && docker compose up airflow-init && docker compose up --build -d

perms:
    sudo mkdir -p logs plugins temp dags tests data visualization && sudo chmod -R u=rwx,g=rw

do-sleep:
    sleep 30

up: perms docker-spin-up do-sleep

down:
    docker compose down
```

```
restart: down up

sh:
    docker exec -ti scheduler bash

dbt-docs:
    docker exec -d webserver bash -c "cd /opt/airflow/tpch_analytics && nohup dbt docs serve
```

We can see how long complex commands can be aliased to short `make commands`, which can be run as `make command`

Loading...

11 dbt-core is an orchestrator that makes managing pipelines simpler

Note 1. In this chapter, we will cover the `dbt-core` library, which is open source. The dbt company provides managed services via `dbt-cloud`, which we will not cover in this chapter. 2. All the code in this chapter assumes you have followed the setup steps **shown here** and started the necessary containers.

We saw how to process data in layers, from source to stage to core to marts, in the **data flow chapter**.

While we can write code to maintain these pipelines, it will be a lot of work.

This is where dbt comes in. dbt is a Python tool that simplifies transforming data using just SQL. dbt is meant to make the transformation part of your data pipeline, which is often where the bulk of work happens, easy to build and maintain.

In addition, dbt also enables best practices like

1. Data testing
2. Full snapshot & incremental data processing capabilities
3. Functionality to easily create SCD2 tables
4. Version-controlled data pipelines
5. Separation of concerns via folders based on the **multi-hop architecture**
6. Multiple data materialization options, like creating a data model as a table/view/materialized views, etc
7. Ability to connect to multiple data processing systems

dbt assumes that the data is already accessible to the database engine on which you run it. As such, it is mainly used for the T (Transform) part of your pipeline.

Let's run dbt inside the airflow container, as shown below.

```
docker exec -ti scheduler bash # bash into the running docker container
# The following commands are run inside the scheduler docker container
cd $AIRFLOW_HOME && python3 generate_data.py && python3 run_ddl.py # create fake data
cd tpch_analytics # cd into the dbt project directory
dbt run --profiles-dir /opt/airflow/tpch_analytics/ --project-dir /opt/airflow/tpch_analytics
dbt test --profiles-dir /opt/airflow/tpch_analytics/ --project-dir /opt/airflow/tpch_analytic
```

11.1 A sql script with a select statement creates a data model

In dbt, every `.sql` file contains a `select` statement and is created as a **data model**.

Note We use the term data model because with dbt we can define what a data model should be, ie, should it be a table, a view, a materialized view, etc. This is called `materialization`.

The `select` statement defines the data schema of the data model. The name of the `.sql` file specifies the name of the data model.

Let's take a look at one of our silver tables, a **dim_customer**

```
with customer as (
    select *
    from {{ ref('stg_customer') }}
),

nation as (
    select *
    from {{ ref('stg_nation') }}
),

region as (
    select *
    from {{ ref('stg_region') }}
)

select
    c.customer_key,
    c.customer_name,
    n.nation_name,
    n.nation_comment,
    r.region_name,
    r.region_comment
from customer as c
left join nation as n on c.nation_key = n.nation_key
left join region as r on n.region_key = r.region_key
```

We can see how the final select query is created as a data model.

The **ref** function refers to another data model. Remember the `stg_customer`, `stg_nation`, & `stg_region` are also data models in the staging folder. Our projects folder structure is shown below.

```
airflow/tpch_analytics/models
   marts
       core
           core.yml
           dim_customer.sql
           fct_lineitem.sql
           fct_orders.sql
           wide_lineitem.sql
           wide_orders.sql
       sales
            customer_outreach_metrics.sql
            order_lineitem_metrics.sql
            sales.yml
   staging
        src.yml
        stage.yml
        stg_customer.sql
        stg_lineitem.sql
        stg_nation.sql
        stg_orders.sql
        stg_region.sql
```

The `ref` function only works for data models that dbt creates. When you need to use data in the warehouse that was not created by dbt (typically the raw data), you will need to define it in a `yml` file and access it using the `source` function.

Here is our **src.yml** file

```yaml
---
version: 2
sources:
  - name: source
    description: Data from application database, brought in by an EL process.
    schema: analytics
    tables:
      - name: customer
      - name: nation
      - name: region
      - name: orders
      - name: lineitem
```

And now we can use the `source` function to access these, as shown in the **stg_lineitem.sql** file below:

```
select
    l_orderkey as order_key,
    l_linenumber as line_number
from {{ source('source', 'lineitem') }}
```

11.2 Define how your project should work at dbt_project.yml

All configurations that define how your project should be in **dbt_project.yml**, such as

1. Which folder to look in for the data model sql files
2. Which folders to look in for seed data
3. Which folders to look in for custom tests, custom macros(functions with SQL)
4. How to materialize data models based on their folder paths, etc

11.3 Define connections in profiles.yml

dbt uses a yml file to define how it connects to your db engine. Let's look at our **profiles.yml**

```
tpch_analytics:
  target: local
  outputs:
    local:
        type: spark
        method: session
        schema: analytics
        host: localhost                          # not used, but required by `dbt-core`
```

We tell dbt to connect to Apache Spark. The `target` variable defines the environment. The default is dev, but you can specify which environment to run on with `--target` flag in the dbt run command.

By default, dbt will look for a profiles.yml in your HOME directory. We can tell dbt to look for the profiles.yml file in a specific folder using the `--profiles-dir` flag as shown below.

```
dbt run --profiles-dir /opt/airflow/tpch_analytics/ --project-dir /opt/airflow/tpch_analytics
```

11.4 Define documentation & tests with yml files

You can also document what the table and the columns of your tables mean in `yml` files. These `yml` files have to be within the same folder and also reference the data model's name and the column names.

In addition to descriptions, you can also specify any tests to be run on the columns as needed.

The documentation will be rendered when you run the `dbt render` command, and HTML files will be created, which we will view with a `dbt serve` command in the next chapter.

The tests can be run with the `dbt test` command. **Note** that the tests can only be run after the data is available.

Here are the documentation and tests to be run for our `dim_customer.sql` model at **core.yml**

```
models:
  - name: dim_customer
    description: "Customer dimension table containing customer details enriched with geograph
    columns:
      - name: customer_key
        description: "Unique identifier for each customer (primary key)"
        tests:
          - unique
          - not_null
      - name: customer_name
        description: "Full name of the customer"
        tests:
          - not_null
      - name: nation_name
        description: "Name of the nation/country where the customer is located"
      - name: nation_comment
        description: "Additional comments or notes about the customer's nation"
      - name: region_name
        description: "Name of the geographic region where the customer is located"
      - name: region_comment
        description: "Additional comments or notes about the customer's region"
```

Run the tests as shown below:

```
docker exec -ti scheduler bash # bash into the running docker container
# cd $AIRFLOW_HOME && python3 generate_data.py && python3 run_ddl.py # create fake data, only
cd tpch_analytics # cd into the dbt project directory
```

```
dbt run --profiles-dir /opt/airflow/tpch_analytics/ --project-dir /opt/airflow/tpch_analytics
dbt test --profiles-dir /opt/airflow/tpch_analytics/ --project-dir /opt/airflow/tpch_analytic
```

11.5 dbt recommends the 3-hop architecture with stage, core & data marts

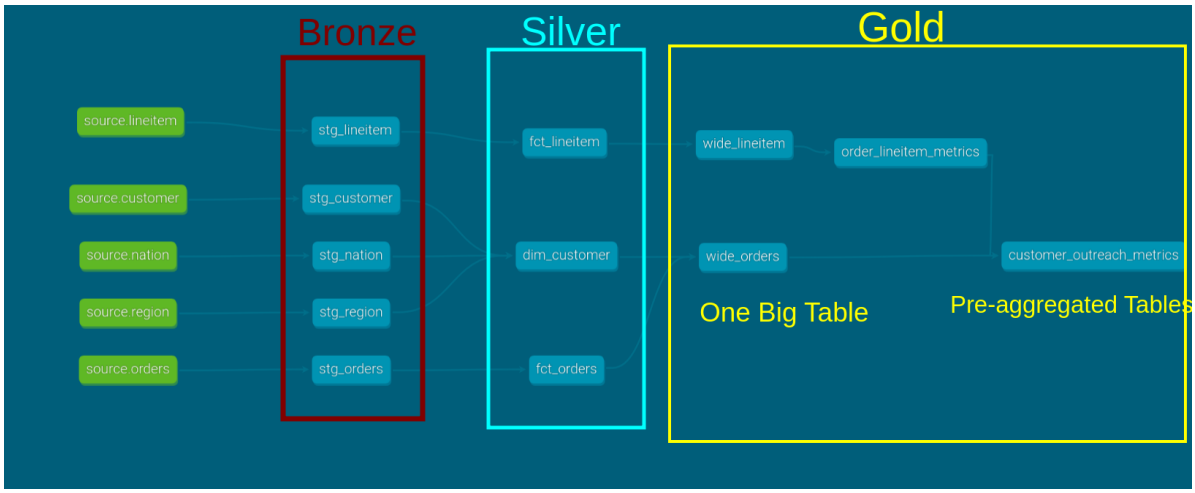


Figure 11.1: Data architecture

We covered how to transform data using the multi-hop architecture in the **data flow chapter**. Let's see how to implement this in dbt.

11.5.1 Source

Source tables refer to tables already present in the warehouse. In our case, these are the base tpch tables, which are created by the extract step.

We need to define what tables are the sources in the src.yml file, which will be used by the stage tables with the `source` function.

```
---
version: 2
sources:
  - name: source
    description: Data from application database, brought in by an EL process.
    schema: analytics
```

```
    tables:
      - name: customer
      - name: nation
      - name: region
      - name: orders
      - name: lineitem
```

11.5.2 Staging

The staging area is where **raw data is cast into correct data types, given consistent column names, and prepared for transformation into models used by end-users**.

You can think of this stage as the first layer of transformations. We will place staging data models inside the `staging` folder, as shown below.

```
models/
  marts
      core
          core.yml
          dim_customer.sql
          fct_lineitem.sql
          fct_orders.sql
          wide_lineitem.sql
          wide_orders.sql
      sales
           customer_outreach_metrics.sql
           order_lineitem_metrics.sql
           sales.yml
  staging
      src.yml
      stage.yml
      stg_customer.sql
      stg_lineitem.sql
      stg_nation.sql
      stg_orders.sql
      stg_region.sql
  dbt_project.yml
  profiles.yml
```

Their documentation and tests will be defined in a `yml` file.

11.5.3 Marts

Marts consist of the core tables for end-users and business vertical-specific tables.

11.5.3.1 Core

The core defines the fact and dimension models to be used by end-users. We define our facts and tables under the `marts/core` folder.

You can see that we store the facts, dimensions, and OBT under this folder.

11.5.3.2 Stakeholder team specific

In this section, we define the models for `sales` stakeholders. A project can have multiple business verticals. Having one folder per business vertical provides an easy way to organize the models.

11.6 dbt-core is a cli

With our data model defined, we can use the dbt CLI to run, test, and create documentation.

The `dbt` command will look for the `profiles.yml` file in your $HOME directory by default, so we either have to set the `PROFILES_DIR` environment variable or use the `--profiles-dir` as part of the cli command.

11.6.1 dbt run

We have the necessary model definitions in place. Let's create the models.

```
docker exec -ti scheduler bash # bash into the running docker container
# cd $AIRFLOW_HOME && python3 generate_data.py && python3 run_ddl.py # create fake data, run
cd tpch_analytics # cd into the dbt project directory
dbt run --profiles-dir /opt/airflow/tpch_analytics/ --project-dir /opt/airflow/tpch_analytics
```

11.6.2 dbt docs

One of the powerful features of dbt is its docs. To generate documentation and serve it, run the following commands:

```
docker exec -ti scheduler bash # bash into the running docker container
# cd $AIRFLOW_HOME && python3 generate_data.py && python3 run_ddl.py # create fake data, run
cd tpch_analytics # cd into the dbt project directory
dbt run --profiles-dir /opt/airflow/tpch_analytics/ --project-dir /opt/airflow/tpch_analytics
dbt docs generate--profiles-dir /opt/airflow/tpch_analytics/ --project-dir /opt/airflow/tpch_
exit
```

```
docker exec -ti webserrver bash # webserver will run the documentation server
dbt docs serve --profiles-dir /opt/airflow/tpch_analytics/ --project-dir /opt/airflow/tpch_a
```

The `generate` command creates documentation in HTML format. The `serve` command will start a web server that serves this HTML file.

Open the documentation by going to http://localhost:8081.

Navigate to `customer_orders` within the `sde_dbt_tutorial` project in the left pane.

Click on the "View Lineage Graph" icon on the lower right side. The lineage graph shows the dependencies of a model.

You can also view the tests defined, their descriptions (set in the corresponding YAML file), and the compiled SQL statements.

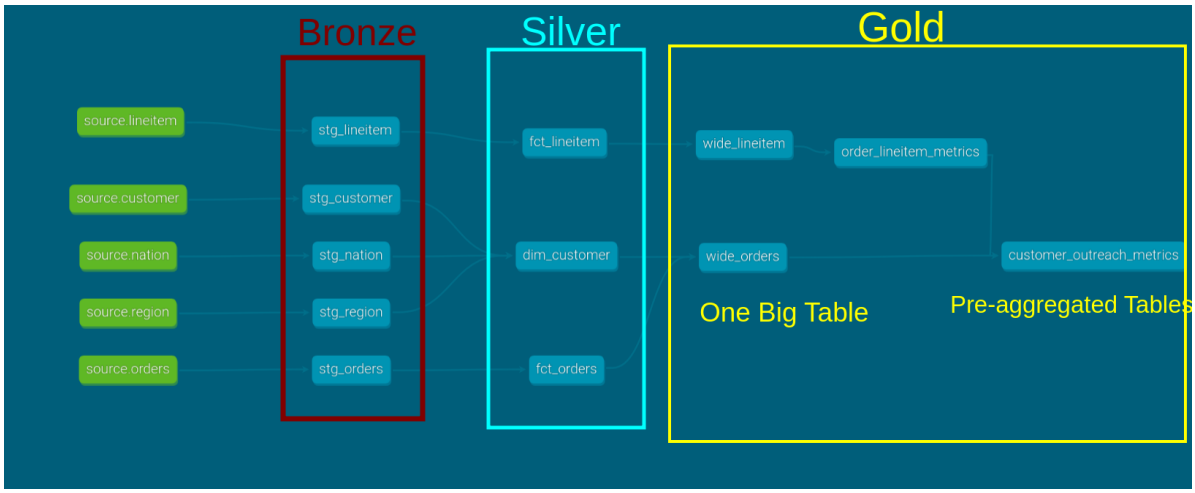


Figure 11.2: Data architecture

11.7 Scheduling

We have seen how to create models, run tests, and generate documentation. These are all commands run via the command line interface (CLI).

DBT compiles the models into SQL queries under the `target` folder (not part of the Git repository) and executes them on the data warehouse.

To schedule dbt runs, snapshots, and tests, we need to use a scheduler. In ne chapter, we will use Airflow to schedule this dbt pipeline.

12 Airflow is both a scheduler and an orchestrator

In the previous chapter, we saw how dbt enables data modeling.

However, a data pipeline typically contains more than just data modeling; you will generally need to bring data into the warehouse, model it, run tests on the data, send alerts if the test fails, and so on. This is where Airflow comes in.

Airflow is both an orchestrator and scheduler. With the Airflow paradigms (dags & tasks), you can define the steps to run and their order.

You can also specify the frequency with which to run the pipelines.

We saw how to start and open the Airflow UI at http://localhost:8080 and log in with 'airflow' as the username and password, in **this chapter**.

Go ahead and run the dag as shown below and give it a few minutes to complete.

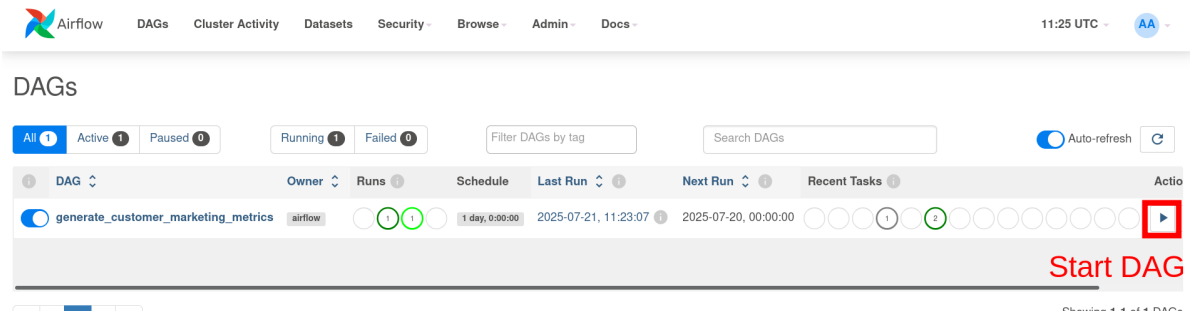


Figure 12.1: DAG

12.1 Airflow DAGs are used to define how, when, and what of data pipelines

Let's take a look at our DAG.

```
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.bash_operator import BashOperator

with DAG(
    "generate_customer_marketing_metrics",
    description="A DAG to extract data, load into db and generate customer marketing metrics"
    schedule_interval=timedelta(days=1),
    start_date=datetime(2023, 1, 1),
    catchup=False,
    max_active_runs=1,
) as dag:
    extract_data = BashOperator(
        task_id="extract_data",
        bash_command="cd $AIRFLOW_HOME && python3 generate_data.py && python3 run_ddl.py",
    )

    transform_data = BashOperator(
        task_id="dbt_run",
        bash_command="cd $AIRFLOW_HOME && dbt run --profiles-dir /opt/airflow/tpch_analytics/
    )

    generate_docs = BashOperator(
        task_id="dbt_docs_gen",
        bash_command="cd $AIRFLOW_HOME && dbt docs generate --profiles-dir /opt/airflow/tpch_
    )

    generate_dashboard = BashOperator(
        task_id="generate_dashboard",
        bash_command="cd $AIRFLOW_HOME && python3 /opt/airflow/tpch_analytics/dashboard.py",
    )

    extract_data >> transform_data >> generate_docs >> generate_dashboard
```

In this Python script, we:

1. Create a DAG where we define the name, decription, start date, frequency, and the maximum number of concurrent runs, among other parameters. Documentation for all available parameters.
2. Create 4 tasks extract_data, transform_data, generate_docs, & generate_dashboard.
3. Define the order in which the tasks are to be run as `extract_data >> transform_data >> generate_docs >> generate_dashboard`

12.2 DAGs are made up of tasks

In the above example, we saw that the DAG has 4 tasks: `extract_data`, `transform_data`, `generate_docs` & `generate_dashboard`. A task can be any script from Python, SQL, Pyspark, Java jars, etc.

Airflow offers operators tasks for popular services, such as the S3CreateBucketOperator, which make creating an S3 bucket easy.

Airflow also enables us to create sensors, which are tasks that wait for other tasks/Dags to reach a specific state.

12.3 Airflow configurations are stored at $AIRFLOW_HOME/airflow.cfg

Airflow comes with numerous configuration settings that can be set in the `airflow.cfg` file.

This configuration file controls settings such as where and how to run our code. Airflow enables us to run our tasks in multiple ways:

1. Run code on the same machine as your scheduler process with Local and sequential executor
2. Run code in a task queue (i.e., a system that will run tasks on individual machines) with a Celery executor.
3. Run code as k8s pods with Kubernetes executor.
4. Write custom logic to run your tasks.

It also enables connections to other systems with environment variables, which folder to look for your dag scripts, and many others, see this link.

We can peek into ours by opening our Docker container and viewing them:

```
make sh
# below commands are run inside the scheduler container
cat $AIRFLOW_HOME/airflow.cfg
cat $AIRFLOW_HOME/airflow.cfg | grep 'executor =' # you will see the executor type that you a
cat $AIRFLOW_HOME/airflow.cfg | grep dags_folder # your dag folder path
```

12.4 User interface to see how your pipelines are running and their history

As the number and complexity of your data pipelines grow, it is crucial to be able to see precisely how your pipelines are running, detect bottlenecks and delays in data processing, and so on.

Airflow stores information about a pipeline run in its database, and it is viewable via the Airflow UI.

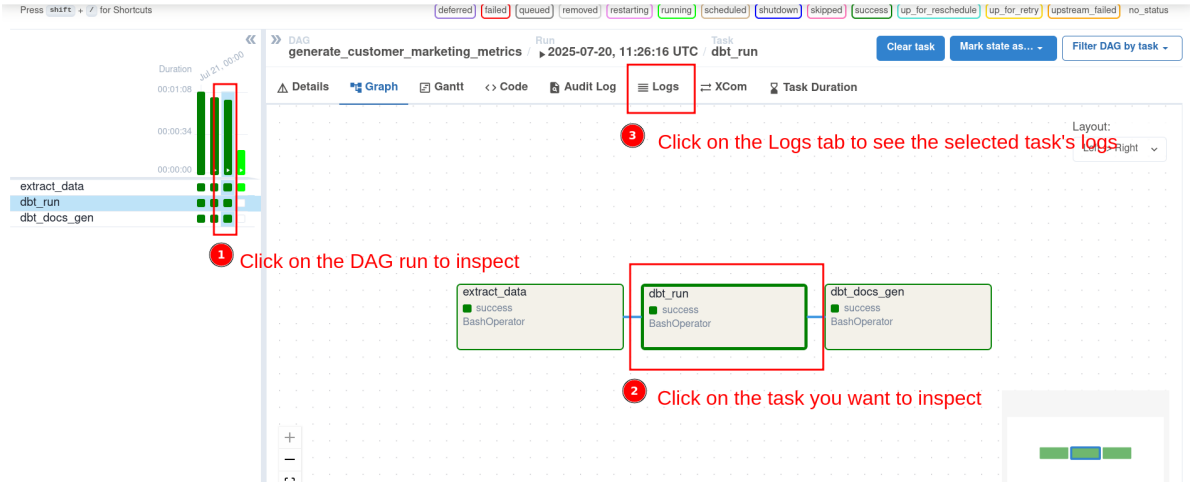
You can open the Airflow UI by going to http://localhost:8080.

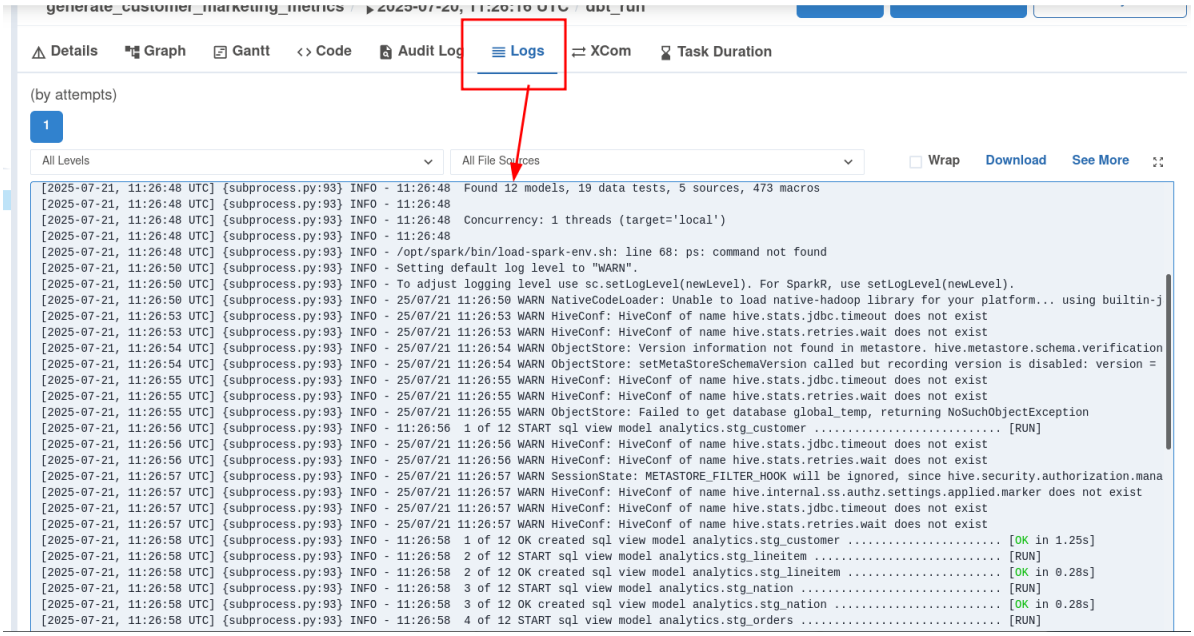
12.4.1 See progress & historical information on UI

When we run data pipelines, we can utilize a user-friendly web UI to view progress, failures, and other details.

We can also view individual task logs and the inputs for a specific task, among other things.

The web UI provides good visibility into the current and historical state of our pipelines.

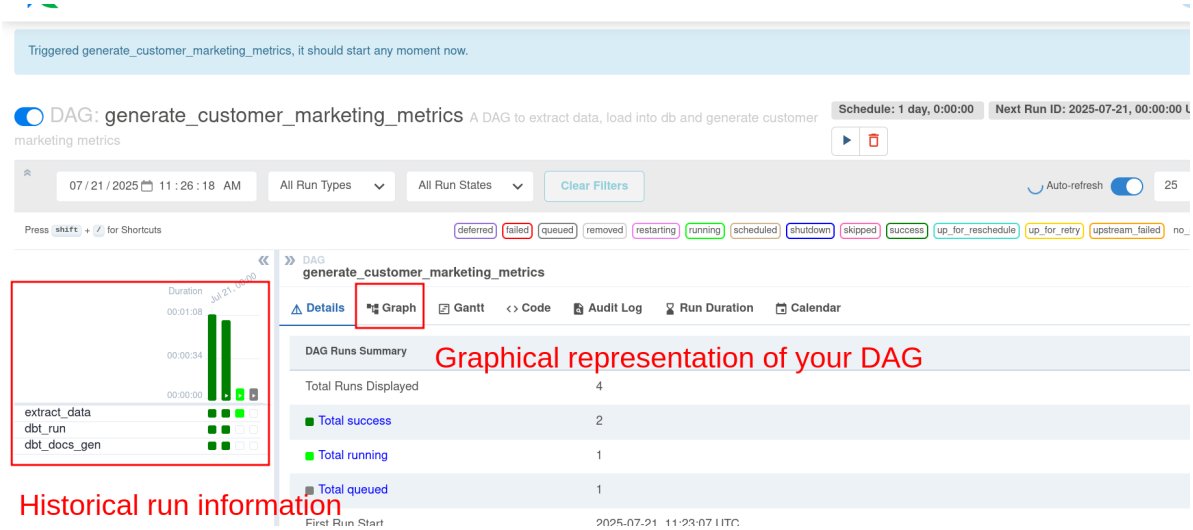


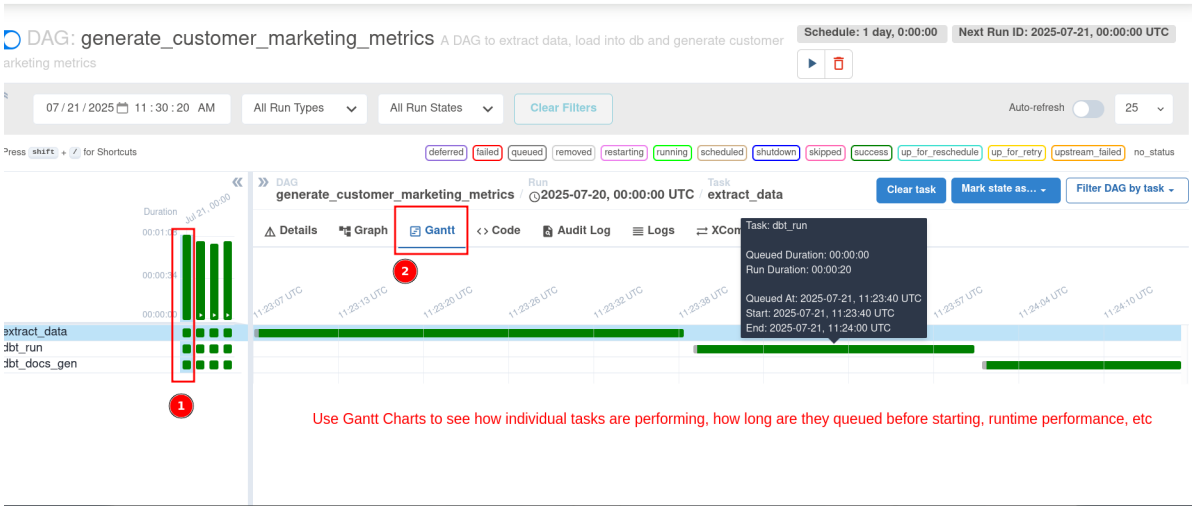


12.4.2 Analyze data pipeline performance with Web UI

We can see how pipelines have performed over time, inspect task run time, and see how long a task had to wait to get started.

The performance metrics provide us with the necessary insights to optimize our systems.





12.4.3 Re-run data pipelines via UI

In addition to seeing how our pipelines are running, we can manually trigger DAGs with custom inputs as necessary.

The ability to trigger/re-run DAGs helps us quickly resolve one-off issues.

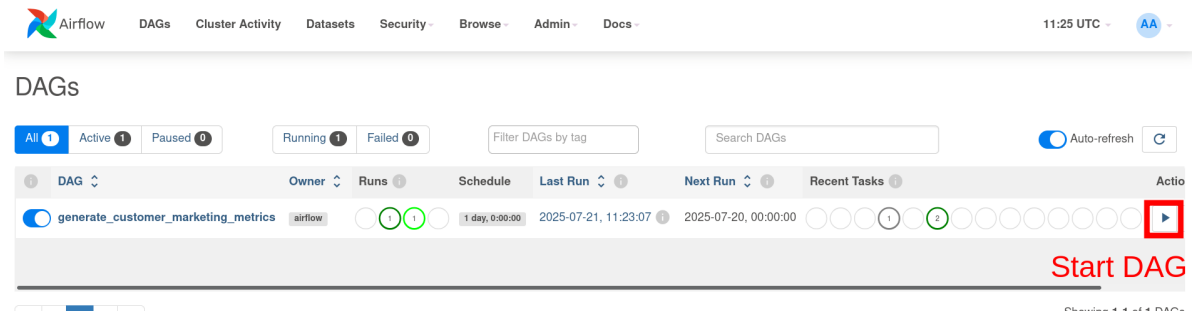


Figure 12.2: DAG Run

See **this link** for information on triggering dags with UI and CLI.

12.4.4 Reuse variables and connections across your pipelines

Apache Airflow also allows us to create and store variables and connection settings that can be reused across our data pipelines.

Variables and connections are significant when you want to use a variable across data pipelines and to connect to external systems, respectively.

Once the connection/variables are set, we can see them in our UI:

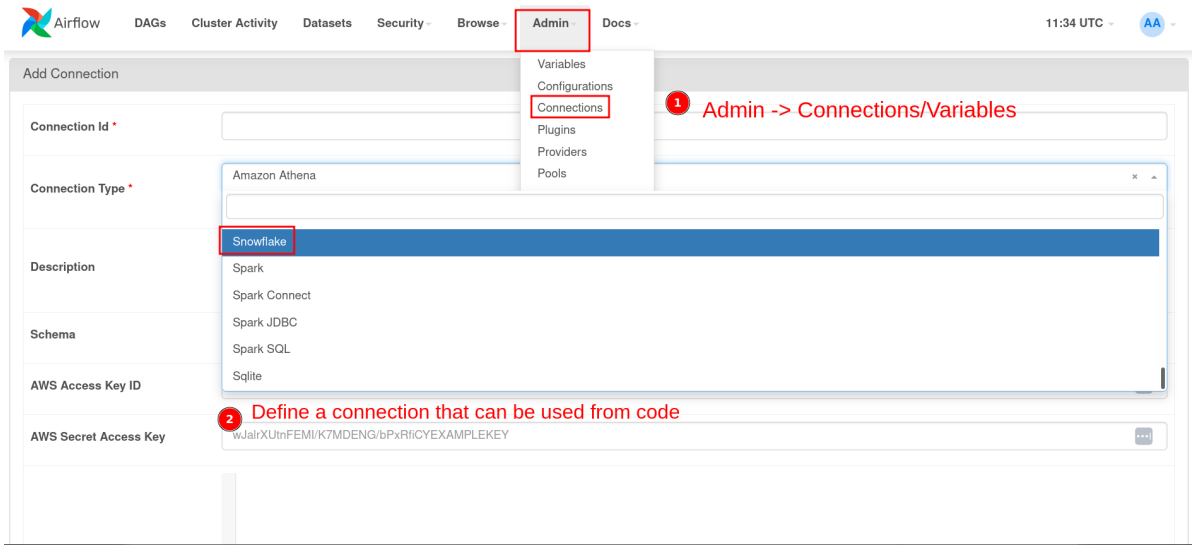


Figure 12.3: DAG Conn

Use them in your DAG code by directly accessing them, as shown below.

```
from airflow.hooks.base import BaseHook
conn = BaseHook.get_connection("your_connection_id")
```

Alternatively, some operators allow you to enter a parameter called `conn_id`, which enables you to pass the connection ID and establish a connection to the required service.

After the dag is run, in the terminal, run `make dbt-docs` (inside the airflow folder) for dbt to serve the docs, which is viewable by going to http://localhost:8081.

13 Capstone Project

Over the past few chapters we went over

1. Data transformation with Spark SQL
2. Data modeling with dbt
3. Scheduling and orchestrating with Airflow

In this capstone project, we will go over how you can present your expertise as a data engineer to a potential hiring manager.

The main objectives for this capstone project are 1. Understanding how the different components of data engineering work with each other 2. How to model and transform data in the 3-hop architecture 3. Clearly explain what your pipeline is doing, why, and how

Let's assume we are working on modeling the TPCH data and creating a data mart for the sales team to create customer metrics that they can use to strategize how to cold-call customers.

13.1 Presentation matters

When a hiring manager reviews your project, assume that they will not read the code. Typically, when people look at projects, they browse high-level sections. These includes

1. Outcome of your project
2. High-level architecture
3. Project structure to understand how your code works
4. Browse code for clarity and code cleanliness

We will see how you can address these.

13.2 Run the pipeline and visualize the results

Open Airflow UI at http://localhost:8080, login with username/password as airflow and run the dag as shown below.

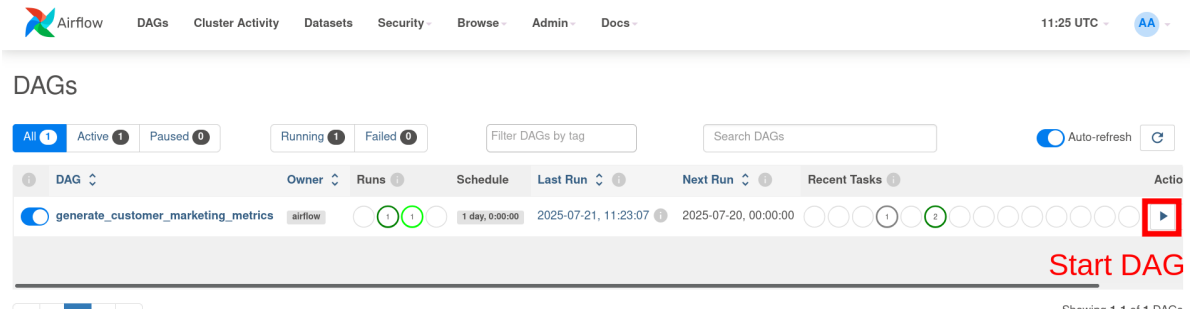


Figure 13.1: DAG

We use the python Plotly library to create a simple HTML dashboard as shown below.

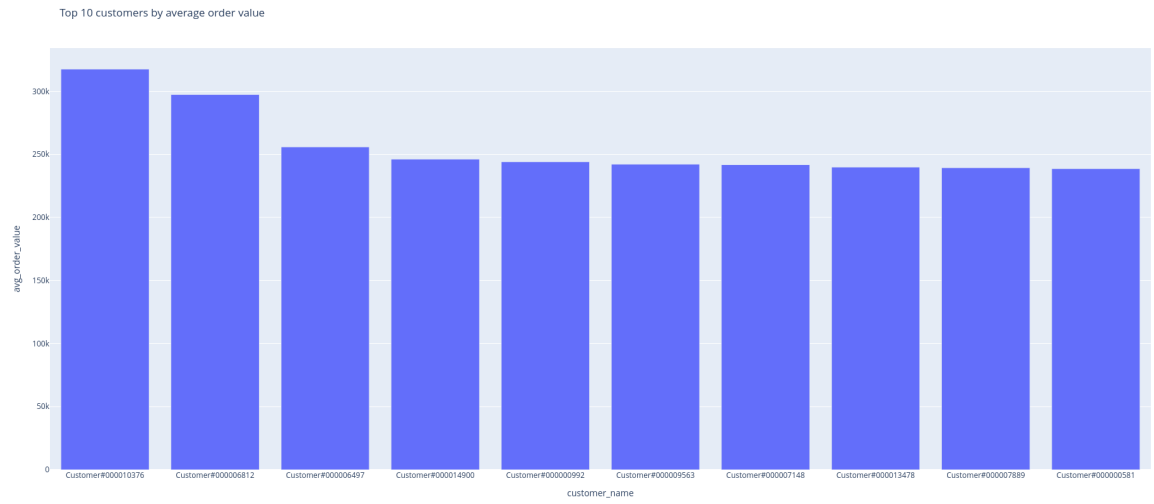


Figure 13.2: UI

13.3 Start with the outcome

We are creating data to support the sales team's customer outreach efforts. For this, we need to present customers who are most likely to convert. While this is a complex data science question, a simple approach could be to target customers who have the highest average order value (assuming high/low order values are outliers).

Create a dashboard to show the top 10 customers by average order values as a descending bar chart.

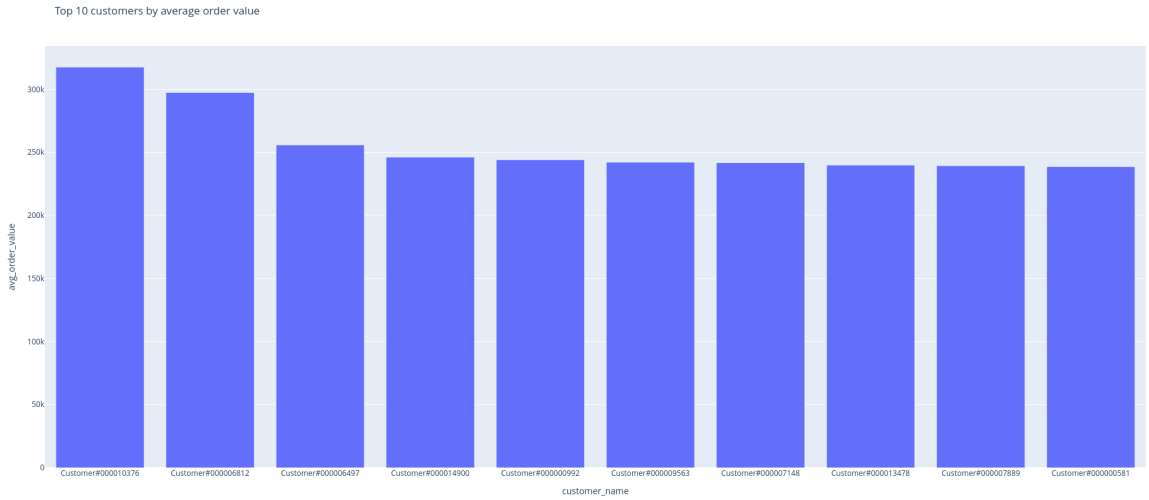


Figure 13.3: UI

Note The Python script to create the dashboard is available at **airflow/tpch_analytics/dashboard.py**.

13.4 High-level architecture

The objective of this is to show your expertise in

1. Designing data pipelines, by following industry standard 3-hop architecture
2. Industry standard tools like dbt, Airflow, and Spark
3. Writing clean code using auto formatters and linters

Our base repository comes with all of these set up and installed for you to copy over and use.

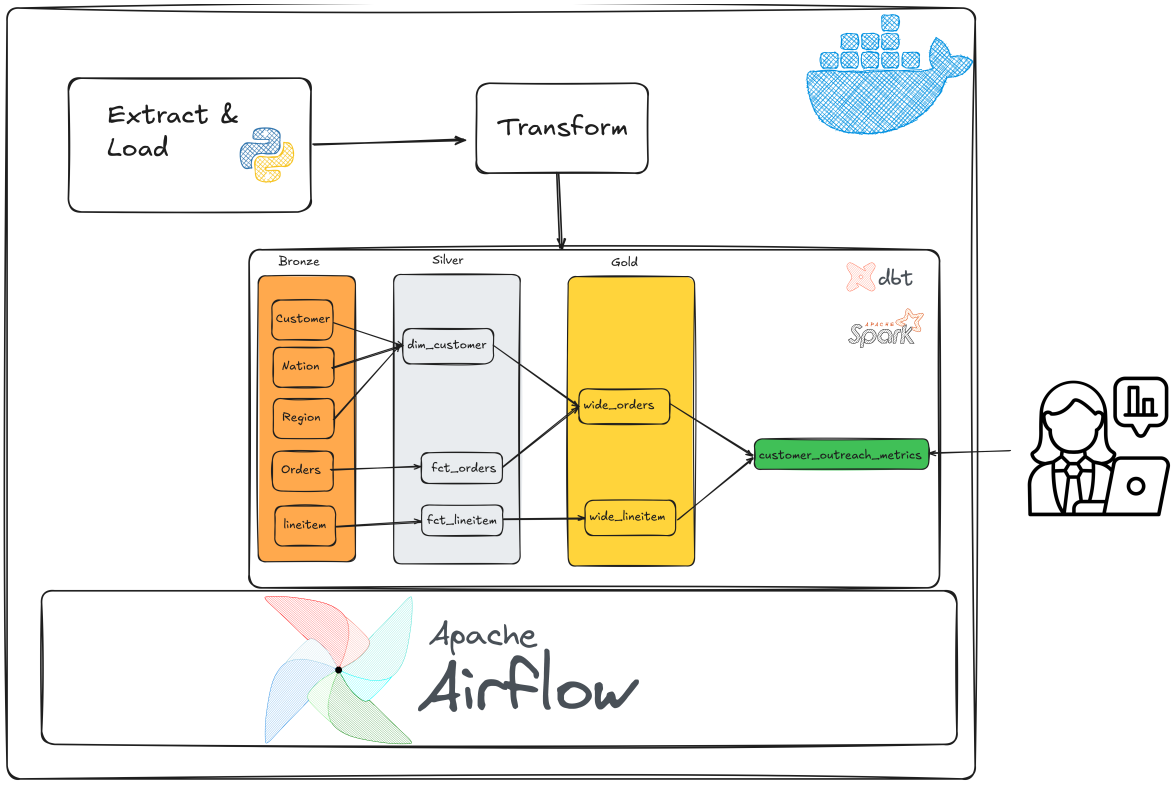


Figure 13.4: Capstone Architecture

13.5 Putting it all together with an exercise

Use this Airflow + dbt + Spark setup to bootstrap your own project, as shown below:

```
cp -r ./data_engineering_for_beginners_code/airflow ./your-project-name
cd you-project-name
# Update README.md with your specifics
git init
git add .
git commit -m 'First Commit'
```

Create a new GitHub repo at GitHub Create Repo with the same name as your project.

and follow the steps under `…or push an existing repository from the command line`

13.6 Exercise: Your Capstone Project

Find a dataset that interests you, showcasing an innovative perspective on the data. Outcome should be shown with data.

Read **this article to help you identify a problem space and datasets**.

Read **this article for more information on formattting a project for hiring managers**

Loading...

14 Topics Coming Soon

1. unix
2. git

References