Analytical Data Processing in SQL

A guide to understanding the core concepts of distributed data processing, analytical functions, and query optimizations in your data warehouse.

Joseph Machado

Contents

1 Preface

1.1 How to ask questions

If you have any questions or feedback, please email help@startdataengineering.com. If you have difficulty starting the docker containers or connecting to them, please open a GitHub issue here.

1.2 Acknowledgments

We use the TPC-H dataset and Trino as our OLAP DB. This book is formatted using this template.

2 Prerequisites

Please try out the code as you read; this will help much more than just reading the text.

2.1 Lab setup

Please install the following software:

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 its not already present).

All the commands shown below are to be run via the terminal (use the Ubuntu terminal for WSL users). We will use docker to set up our containers. Clone and move into the lab repository, as shown below.

```
git clone \
https://github.com/josephmachado/analytical_dp_with_sql.git
cd analytical_dp_with_sql
```

Makefile lets you define shortcuts for commands that you might want to run, E.g., in our Makefile, we set the alias `trino` for the command `docker container exec -it trino-coordinator trino`, so when we run `make trino` the docker command is run.

We have some helpful make commands for working with our systems. Shown below are the make commands and their definitions

1. make up: Spin up the docker containers.
2. make trino: Open trino cli; Use exit to quit the cli. This is where you will type your SQL queries.
3. make down: Stop the docker containers.

You can see the commands in this Makefile. If your terminal does not support make commands, please use the commands in the Makefile directly. All the commands in this book assume that you have the docker containers running.

In your terminal, do the following:

```
# Make sure docker is running using docker ps
make up # starts the docker containers
sleep 60 # wait 1 minute for all the containers to set up
make trino # opens the trino cli
```

In Trino, we can connect to multiple databases (called catalogs in Trino). TPC-H is a dataset used to benchmark analytical database performance. Trino's tpch catalog comes with preloaded tpch datasets of different sizes tiny, sf1, sf100, sf100, sf300, and so on, where sf = scaling factor.

```
-- run "make trino" or
-- "docker container exec -it trino-coordinator trino"
-- to open trino cli

USE tpch.tiny;
SHOW tables;
SELECT * FROM orders LIMIT 5;
```

```
-- shows five rows, press q to quit the interactive results
 ↪  screen
exit -- quit the cli
```

Note: Run `make trino` or `docker container exec -it trino-coordinator trino` on your terminal to open the trino cli. The SQL code shown throughout the book assumes you are running it in trino cli.

Starting the docker containers will also start Minio (S3 alternative); we will use Minio as our data store to explain efficient data storage.

UI: Open the Trino UI at http://localhost:8080 (username: any word) and Minio (S3 alternative) at http://localhost:9001 (username: minio, password: minio123) in a browser of your choice.

2.2 The data model used in this book

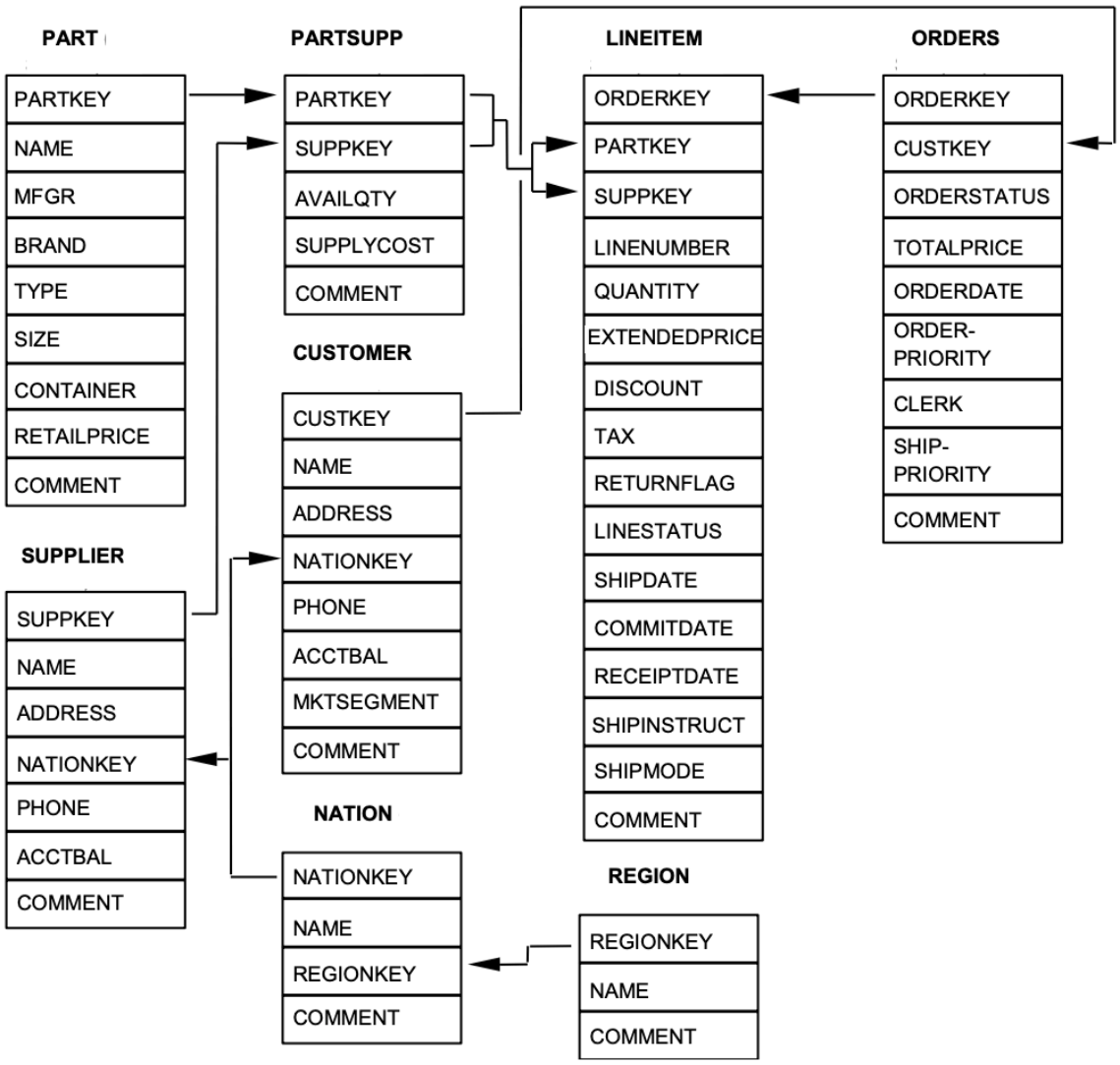
The TPC-H data represents a car parts seller's data warehouse, where we record orders, items that make up that order (lineitem), supplier, customer, part (parts sold), region, nation, and partsupp (parts supplier).

Note: Have a copy of the data model as you follow along; this will help in understanding the examples provided and in answering 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 arrows point in the direction of the one-to-many relationships between tables;

Figure 1: TPC-H data model

2.3 SQL & OLAP Basics

If you are still getting familiar with basic SQL commands, read this appendix chapter first: Appendix: SQL Basics

In this book, we will use an OLAP DB (Online Analytical Processing Database). OLAP DBs are designed to handle analytical data processing on large amounts of data.

Some examples of OLAP DBs are AWS Redshift, Bigquery, and Trino. For a quick introduction to data warehousing, read this article.

3 Understand your data; it's the foundation for data processing

In this chapter, we will learn how data is typically modeled in a data warehouse, understand what analytical queries are, and discuss some questions that can help you understand the data in detail.

3.1 Relate the data model to the business by understanding facts and dimensions

A data warehouse is a database that stores your company's historical data. The tables in a data warehouse are usually of two types, as shown below.

1. Dimension: Each row in a dimension table represents a business entity that is important to the business. For example, A car parts seller's data warehouse will have a `customer` dimension table, where each row will represent an individual customer. Other examples of dimension tables in a car parts seller's data warehouse would be `supplier` & `part` tables. Techniques such as SCD2 are used to store data whose values can change over time (e.g., customers address).
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 fact table's grain (aka granularity, level) refers to what a row in a fact 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 per item purchased, whereas the order table will have one row per order made.

```
use tpch.tiny;

-- calculating the totalprice of an order (with orderkey = 1)
 ↪  from it's individual items
SELECT
    orderkey,
    round( sum(extendedprice * (1 - discount) * (1 + tax)),
        2
    ) AS totalprice
    -- Formula to calculate price paid after discount & tax
FROM
    lineitem
WHERE
    orderkey = 1
GROUP BY
    orderkey;

/*
 orderkey | totalprice
----------+------------
        1 |  172799.56
*/

-- The totalprice of an order (with orderkey = 1)
SELECT
```

```
    orderkey,
    totalprice
FROM
    orders
WHERE
    orderkey = 1;

/*
 orderkey | totalprice
----------+------------
        1 |  172799.49
*/
```

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. But having just the `orders` table is not sufficient since the `lineitem` table will provide us with individual item details, such as discount and quantity details.

3.2 Modifying granularity (aka roll up/group by) is the basis of analytical data processing

The term analytical querying usually refers to aggregating numerical (spend, count, sum, avg) data from the fact table for specific dimension attribute(s) (e.g., name, nation, date, month) from the dimension tables. Some examples of analytical queries are

1. Who are the top 10 suppliers (by totalprice) in the past year?

2. What are the average sales per nation per year?
3. How do customer market segments perform (sales) month-over-month?

The questions above ask about historically aggregating data from the fact tables for one or more business entities(dimensions). Consider the example analytical question below and notice the facts and dimensions.

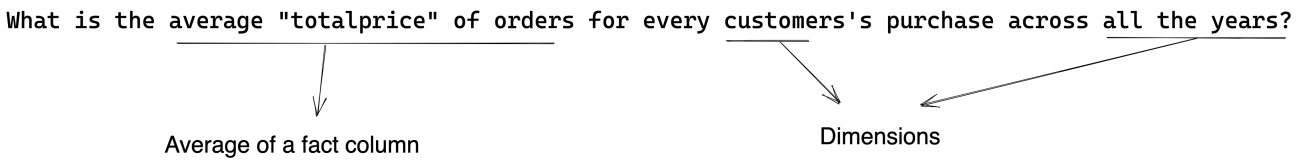


Figure 2: Analytical query

When we dissect the above analytical query, we see that it involves:

1. Joining the fact data with dimension table(s) to get the dimension attributes such as name, region, & brand. In our example, we join the orders fact table with the customer dimension table.
2. Modifying granularity (aka rollup, Group by) of the joined table to the dimension(s) in question. In our example, this refers to `GROUP BY custkey, YEAR(orderdate).`

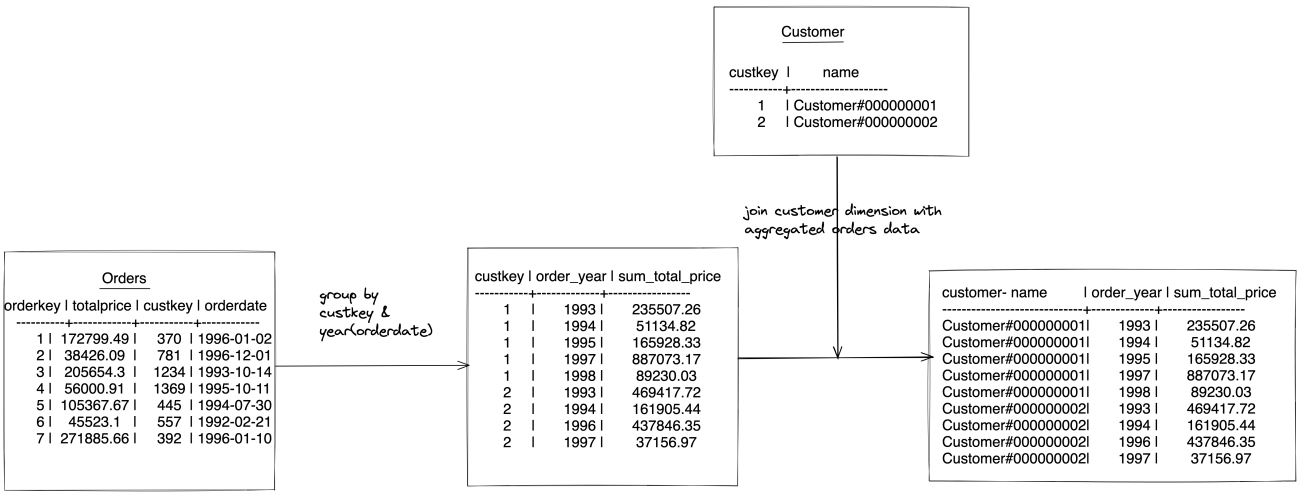


Figure 3: Joining facts and dims

When rolling up, we must pay attention to the type of data stored in the column. For example, in our lineitem table, we can sum up the price but not the discount percentage since percentages cannot be rolled up (don't roll up fractions, distinct counts). `Additive` facts are the columns in the fact table that can be aggregated and still have a meaningful value, while the ones that cannot (e.g., percentages, distinct counts) are called `non-additive` facts.

Understanding the granularity and which columns are additive, is critical for generating accurate numbers.

Note: Throughout this book, we will have examples and exercises that say, "Create the report at dimension1 and dimension2 level". This means that the output must have a granularity of dimension1 and dimension2.

3.3 Understanding what the data represents and the pipeline that generates the data helps you answer most business questions

More often than not, clearly understanding your data can help you answer many business questions. Find the answers to these questions to help you understand your data.

1. Semantic understanding: Do you know what the data represents? What is the business process that generates this data?
2. Data source: Is the data from an application DB or external vendor via SFTP/Cloud store dumps, API data pull, manual upload, or other sources?
3. Frequency of data pipeline: How frequently is the data pipeline run? How long does a complete end-to-end data pipeline run take?
4. Side effects: Does the data pipeline overwrite existing data? Is it possible to see the data as it was from a previous point in time?
5. Data caveats: Does the data have any caveats, such as seasonality affecting size, data skew, inability to join, or weird data unavailability? Are there any known issues with upstream data sources, such as late arrival or skipped data?
6. End-users/Stakeholders: Who are the end-users of your data? How do you manage issue reporting & resolution with the end users? How do they consume the data?

Knowing these about a dataset will help you quickly answer most business questions and debug dashboards/reports.

4 Save time and money by storing data efficiently and reducing data movement (while querying)

In this chapter, we will go over how OLAP DBs process distributed data and techniques to improve the performance of your queries. Understanding how data is stored and processed, how OLAP engines plan to run your query, and knowing data storage patterns to reduce data movement will significantly improve your query performance. You will be able to save money on data processing costs.

4.1 OLAP DB stores large datasets as chunks of data and processes them in parallel

A distributed data processing system refers to multiple machines (aka `nodes in a cluster`) working together. Most OLAP DBs are distributed systems, storing data and running queries across multiple machines. The critical ideas of distributed data processing systems are

1. Splitting up a large dataset into smaller chunks and storing it among the nodes in the cluster
2. The processing of data happening in the node where the data chunk is present, avoiding costly data transmission over the network

While the distribution of a dataset across nodes in the cluster enables processing them in parallel, it also comes with the overhead of managing the separate chunks and coordinating them when performing transformations. Use distributed systems only when the data size is sufficiently large or when aggregating large amounts of data to account for the overhead of managing distributed processing.

You can envision the storage of data as shown below.

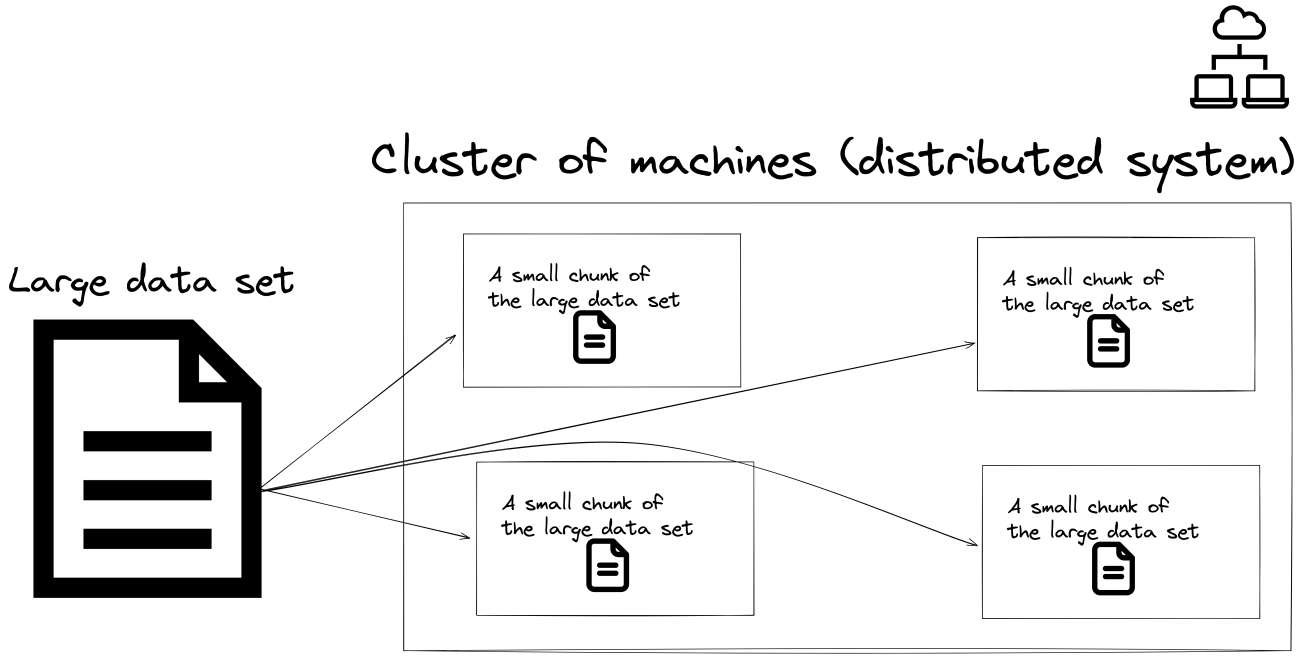


Figure 4: Distributed storage

Examples of distributed data stores are HDFS, AWS S3, GCP Cloud storage, etc.

The fundamental tenets behind query optimizations in any distributed data processing system are

1. Reducing the amount of data to be transferred between nodes in the cluster during querying
2. Reducing the amount of data to be processed by the OLAP engine

4.2 Reduce data movement (data shuffle) by understanding narrow and wide transformations

Now that we have seen how data is stored, it is time to understand data processing in a cluster. We can broadly think of the types of data transformations in 2 categories:

1. Narrow transformations
2. Wide transformations

Narrow transformations refer to transformations that do not require data movement between the nodes in the cluster. These transformations are usually performed per row, independent of other rows. Some examples are operations that operate on one row at a time. For example:

```
USE tpch.tiny;

SELECT
    orderkey,
    linenumber,
    round(
        extendedprice * (1 - discount) * (1 + tax),
        2
    ) AS totalprice
FROM
    lineitem
LIMIT 10;
```

The processing of data happens independently per node. The data is read from the disk in every node, then processed and stored in the node. There is no transfer of data (except to the CLI) to other nodes in the cluster for further processing.

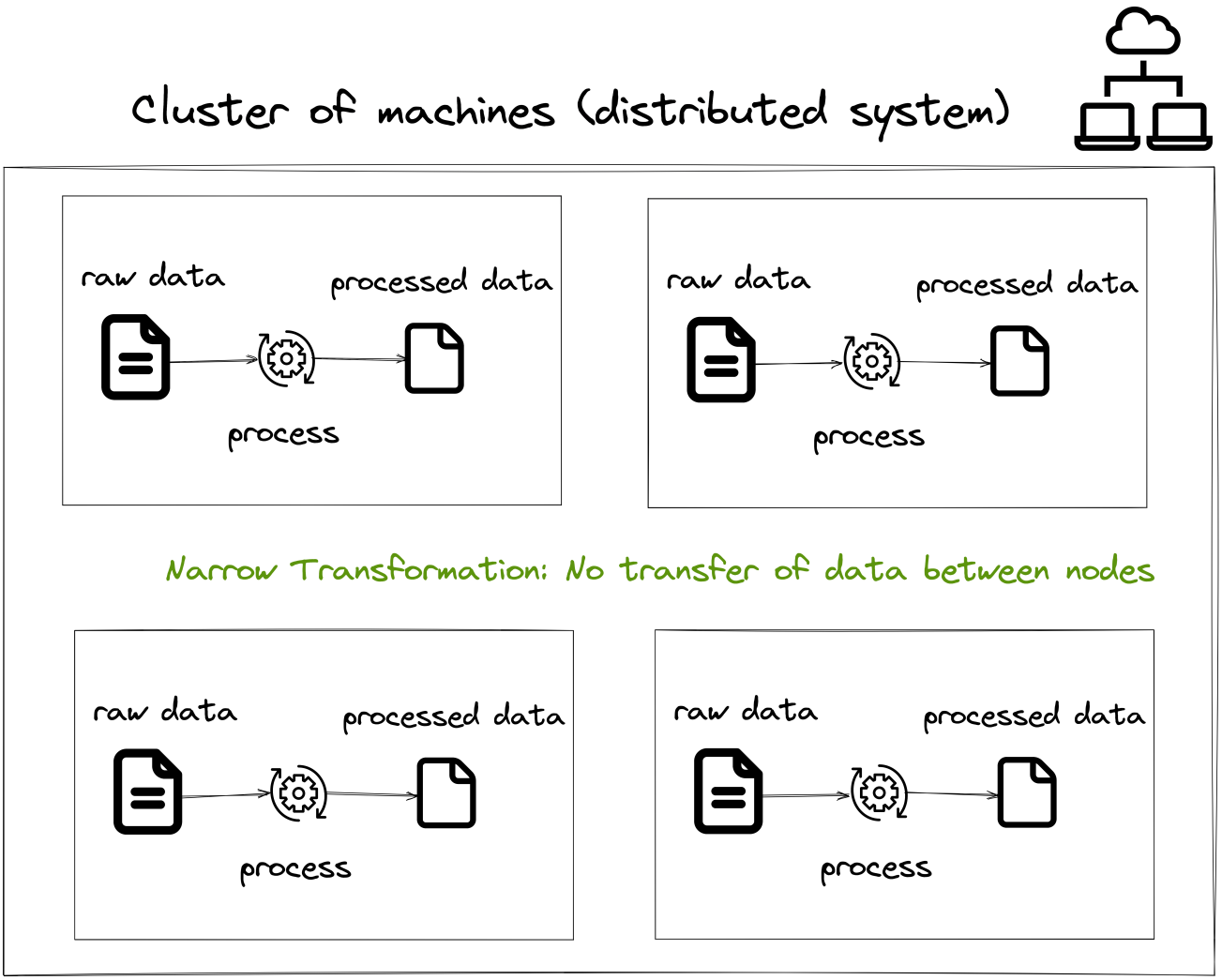


Figure 5: Narrow transformation

Wide transformations involve moving data between nodes in the cluster. You can think of wide transformations as any operation that requires data from multiple rows. E.g., Joining tables require rows from both the tables with matching join keys to be moved to the same machine to get the necessary data in the joined result.

Grouping the lineitem table by `orderkey` will require that the data with the same orderkey be on the same node since the rows with the same orderkey need to be processed together.

A key concept to understand with wide transformation is called data shuffling (aka `exchange`). Data shuffle/exchange refers to the movement of data across the nodes in a cluster. The OLAP DB uses a hash function on the `group by` or `join` column(s) to determine the node to send the data.

Note: Distributed systems use hash functions to create unique identifiers based on the values of a given column(s). Hashing is used when performing a join/group by to identify the rows that need to be processed together.

Distributed systems also use hash functions to identify which node to send a row to. E.g., Given a 4-node distributed system, if we are grouping a dataset, the distributed system can use the formula `Hash (column_value1)%num_of_nodes_in_cluster` (values can be between 0 and num_of_nodes_in_cluster) to identify which node to send all the rows which have a column value of column_value1.

E.g., If we are grouping a table on an id, the OLAP DB will apply the same hash function to the ids of all the data chunks that make up the table and uses it to determine the node to send the data.

```
USE tpch.tiny;

SELECT orderpriority,
ROUND(SUM(totalprice) / 1000, 2) AS total_price_thousands
FROM orders
GROUP BY orderpriority ORDER BY orderpriority;
```

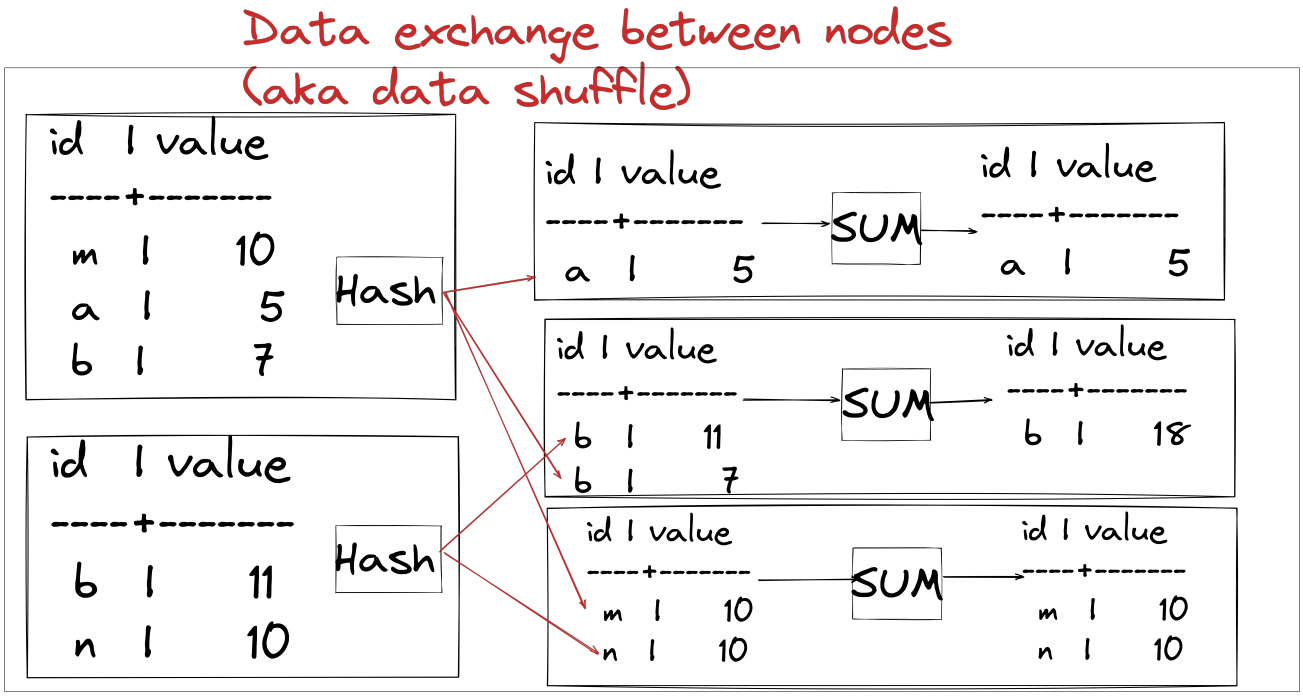


Figure 6: Wide transformation

Wide transformations are expensive (in terms of time) compared to narrow transformations since moving data between nodes takes much more time than just reading from the file system. Thus it is essential to reduce the amount of data shuffled between nodes in the cluster.

In terms of cost (in terms of speed), from highest to lowest is

1. Moving data between nodes in the cluster
2. Reading data from the file system
3. Processing data in memory

We can reduce the data movement by making sure that the data to be shuffled is as tiny as possible by applying filters before joins and only reading in the necessary columns.

4.3 Hash joins are expensive, but Broadcast joins are not

Joins are a vital component of analytical queries. Every OLAP DB has variations of join types, but they all fall under two main types of joins that distributed systems use. The two types are Hash joins, and Broadcast joins.

1. Hash join: This join exchanges the data from both tables based on the join key across nodes in the cluster. The exchange of data over the network is an expensive operation. OLAP DB's use Hash joins to join two sufficiently large tables.

Some OLAP DBs have additional optimizations to help reduce the data movement across the network. E.g., Trino uses a technique called dynamic partitioning where the OLAP DB will exchange data from the smaller table but only exchange the filtered (using ids in the small table) data from the larger table.

An example of a hash join is a join between 2 fact tables. In our example, we have a `lineitem` and an `orders` fact table.

```
USE tpch.sf10;

SELECT
    p.name AS part_name,
    p.partkey,
    l.linenumber,
    ROUND(l.extendedprice * (1 - l.discount), 2)
    AS total_price_wo_tax
FROM
    lineitem l
    JOIN part p ON l.partkey = p.partkey;
```

Note: We used the `tpch.sf10` schema because the tables in `tpch.tiny` are small, and the OLAP DB will trigger a broadcast join (see below).

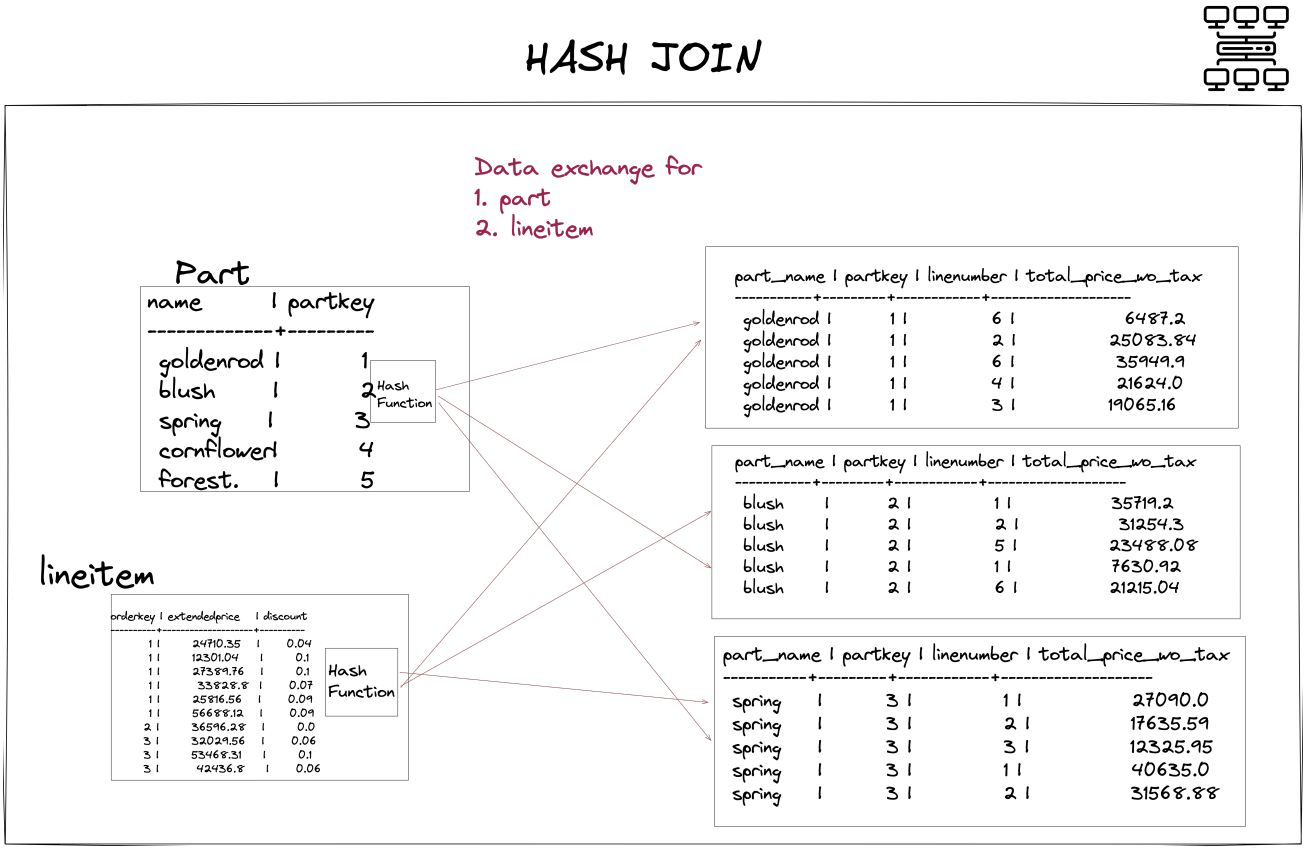


Figure 7: Hash join

2. Broadcast join: This join assumes that the joins are often between facts and dimension tables and that the dimension table is usually significantly smaller than the fact table.
   The OLAP DB sends a copy of the dimension table (usually ten's of millions of rows) to every node in the cluster, which has a chunk of the fact table. In the nodes with the larger tables, the dimension table is kept in memory while reading the fact table data from disk and only keeping the required rows that match the join id in the

dimension table.

This way, we have eliminated the need to exchange large data tables over the network. Transferring only the dimension table over the network allows Broadcast joins to be incredibly fast compared to Hash joins.

The OLAP DB will automatically determine whether to do a broadcast join or a hash join based on the size of the tables. In most OLAP DBs, one can change the size that tells the OLAP DB to use Broadcast join via configs. E.g., in Trino, this is set via the `join_max_broadcast_table_size` config, as such `set session join_max_broadcast_table_size='100MB';`.

An example of a Broadcast join is a join between a fact table (`lineitem`) and a dimension table (`supplier`).

```
USE tpch.tiny;

SELECT
    s.name AS supplier_name,
    l.linenumber,
    ROUND(l.extendedprice * (1 - l.discount), 2)
    AS total_price_wo_tax
FROM
    lineitem l
    JOIN supplier s ON l.suppkey = s.suppkey
LIMIT
    10;
```

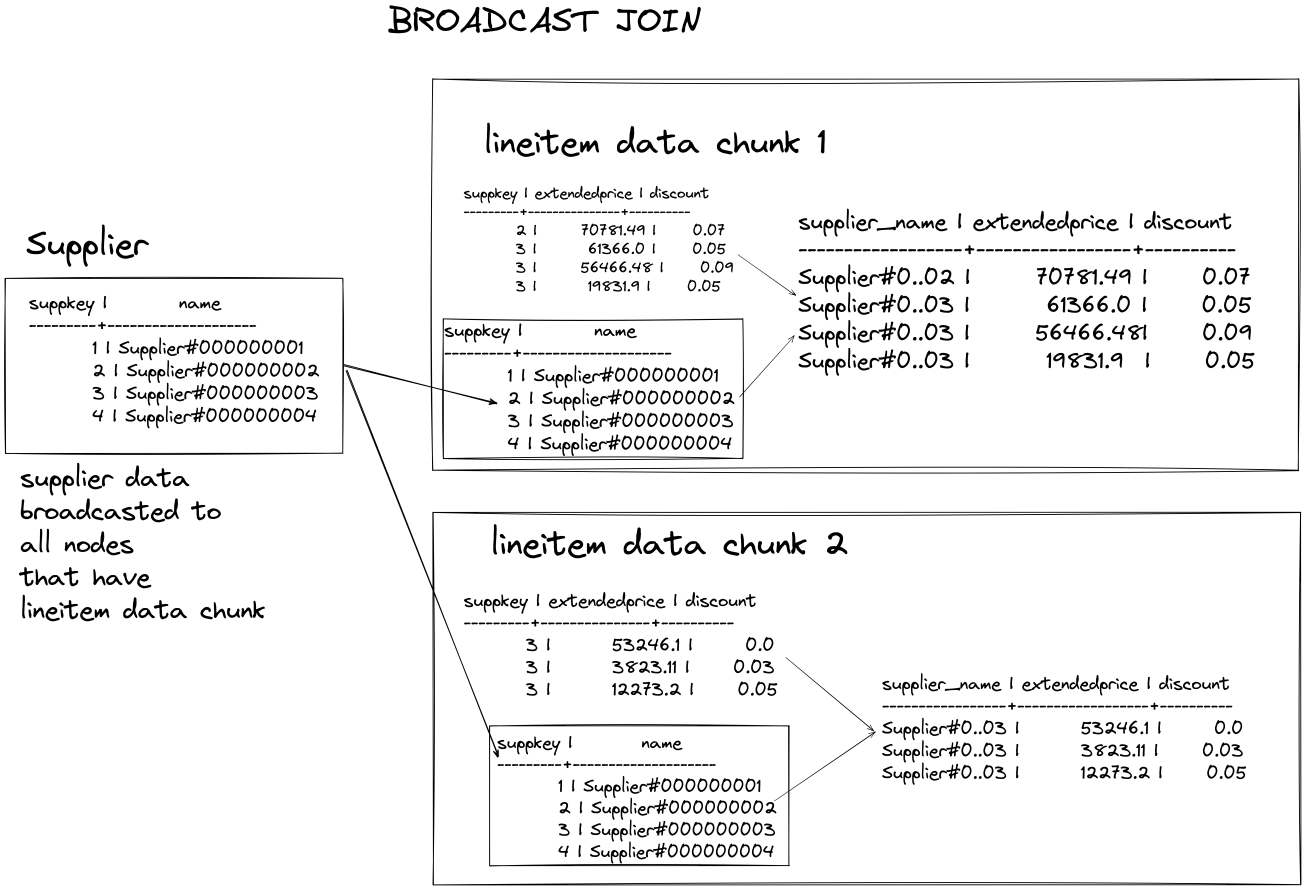


Figure 8: Broadcast join

4.4 Examine how your OLAP DB will process your query, then optimize it

While we can optimize our query, examining the steps the OLAP DB will perform to execute it is better. Use the query plan to explore the OLAP DB's actions. To check the query plan, add the keyword EXPLAIN in front of your query.

Most distributed systems have an EXPLAIN function. Read the results of the `EXPLAIN` function from the bottom up. In general, they have two concepts.

1. Stage: In a query plan, a stage represents a unit of work done in parallel across multiple nodes in the cluster. The boundaries between stages indicate data exchanges. The query plan will show the organization of stages that generates the output. In Trino, we call these `Fragments`.

2. Task: Each stage has one or more tasks executed within nodes. The OLAP DB engine determines running the tasks in parallel or sequentially.

The OLAP DB keeps track of the stages, tasks, and how they associate together. Every distributed system has unique nomenclature. The typical tasks are:

1. Scan: Reads in data from a dataset and brings it into the node's memory.
2. Filter: Uses filter criteria only to keep the eligible rows in memory.
3. Join: Joins two data sets using a hash key. This task is a part of the stage that receives the data chunks of both tables from the exchange.
4. Aggregate: Aggregates data present in the node.
5. Exchange: Hashes data based on column(s) (that is part of the join criteria or group by) and sends data into its corresponding partition node. Some OLAP DBs do `LocalExchange`, which sends data into its corresponding partition thread (within a node process).
6. Statistics: Table statistics are used by the OLAP DB while planning the query and displayed in the query plan. In Trino, these are called `Estimates`. The OLAP DB stores the table statistics information internally. In Trino, we can see table stats using `SHOW STATS FOR lineitem`.
7. Project: Refers to generating(selecting, if no transformation) the columns required from the dataset.

Read your OLAP DBs document to find the equivalent terms for the above concepts. Let's look at an example.

```
USE tpch.tiny;

EXPLAIN
SELECT
    o.orderkey,
    SUM(l.extendedprice * (1 - l.discount))
    AS total_price_wo_tax
FROM
    lineitem l
    JOIN orders o ON l.orderkey = o.orderkey
GROUP BY
    o.orderkey;
```

Let's examine the Fragments (Stages) and the tasks within those.

```
 Fragment 0
     Output layout: [orderkey, sum]
     Output partitioning: SINGLE []
     Output[columnNames = [orderkey, total_price_wo_tax]]
     └─ Aggregate[type =  (STREAMING), keys = [orderkey]]
        └─ Project[]; expr := ("extendedprice" * (1E0 -
         ↪  "discount"))
           └─ InnerJoin[criteria = ("orderkey" =
            ↪  "orderkey_1"), hash = [$hashvalue,
            ↪  $hashvalue_3], distribution = PARTITIONED]
              ├─ ScanFilterProject[table =
               ↪  tpch:tiny:lineitem, dynamicFilters =
               ↪  {"orderkey" = #df_352}]; Layout:
               ↪  [orderkey:bigint, extendedprice:double,
               ↪  discount:double, $hashvalue:bigint]
```

```
                └─ LocalExchange[partitioning = SINGLE];
                ↪  Layout: [orderkey_1:bigint,
                ↪  $hashvalue_3:bigint]
                   └─ RemoteSource[sourceFragmentIds = [1]]

Fragment 1
    Output layout: [orderkey_1, $hashvalue_5]
    Output partitioning: tpch:orders:15000 [orderkey_1]
    ScanProject[table = tpch:tiny:orders]
```

The indentation of the stages/tasks represents the order of execution. Fragments & Tasks along the same indentation run in parallel. We can see that Fragments 0 and 1 are along the same indent; this means they both run in parallel. Fragment 1 scans the orders tables and sends data to Fragment 0, which has lineitem data chunks and tries to join orders data with lineitem data as the orders data flow in.

From the bottom up:

1. Fragment 1:
    1. ScanProject: The table `tpch.tiny.orders` is scanned, and columns `orderkey and hash(orderkey)` are kept. Choosing only the required columns out of all the available columns is called `projection`.
    2. Output partitioning & Layout: The projected data from the previous step is exchanged based on the `orderkey,` and the data is of the format `[orderkey, Hash (orderkey)].`
2. Fragment 0:
    1. RemoteSource: This task receives the data exchanged from Fragment 1. Note we have `sourceFragmentIds = [1]`, which

indicates that this task receives a chunk of data from Fragment 1.
2. LocalExchange: This task exchanges data among multiple threads within the same node.
3. ScanFilterProject: The table `tpch.tiny.lineitem` is scanned, filter applied, & columns `orderkey, extendedprice, discount, & hash(orderkey)` are kept. The dynamic filter is a Trino optimization that aims to reduce the amount of data processed by filtering the larger table on the ids of the smaller table in a join before data exchange.
4. InnerJoin: Joins the data from Fragment 1 (orders) and ScanFilterProject (lineitem).
5. Project: The expression `l.extendedprice * (1 - l.discount)` is calculated.
6. Aggregate: Since we are doing a group by, the OLAP DB aggregates the expression `l.extendedprice * (1 - l.discount)` by orderkey.
7. Output partitioning & layout: This task assigns the column names for the output, does a single partition since we need all the data in our cli, and denotes the order of columns(as orderkey, sums).

An alternative to reading the query plan would be to execute the query and check the plan visualization via Trino UI.

Go to http://localhost:8080 to see the Trino UI (username: any word). Here uncheck the `Finished` state selector, click on the query id, and select the `Live Plan` tab. You will see the plan that the OLAP DB executed. Clicking on a stage will show the operations that we run within that stage.

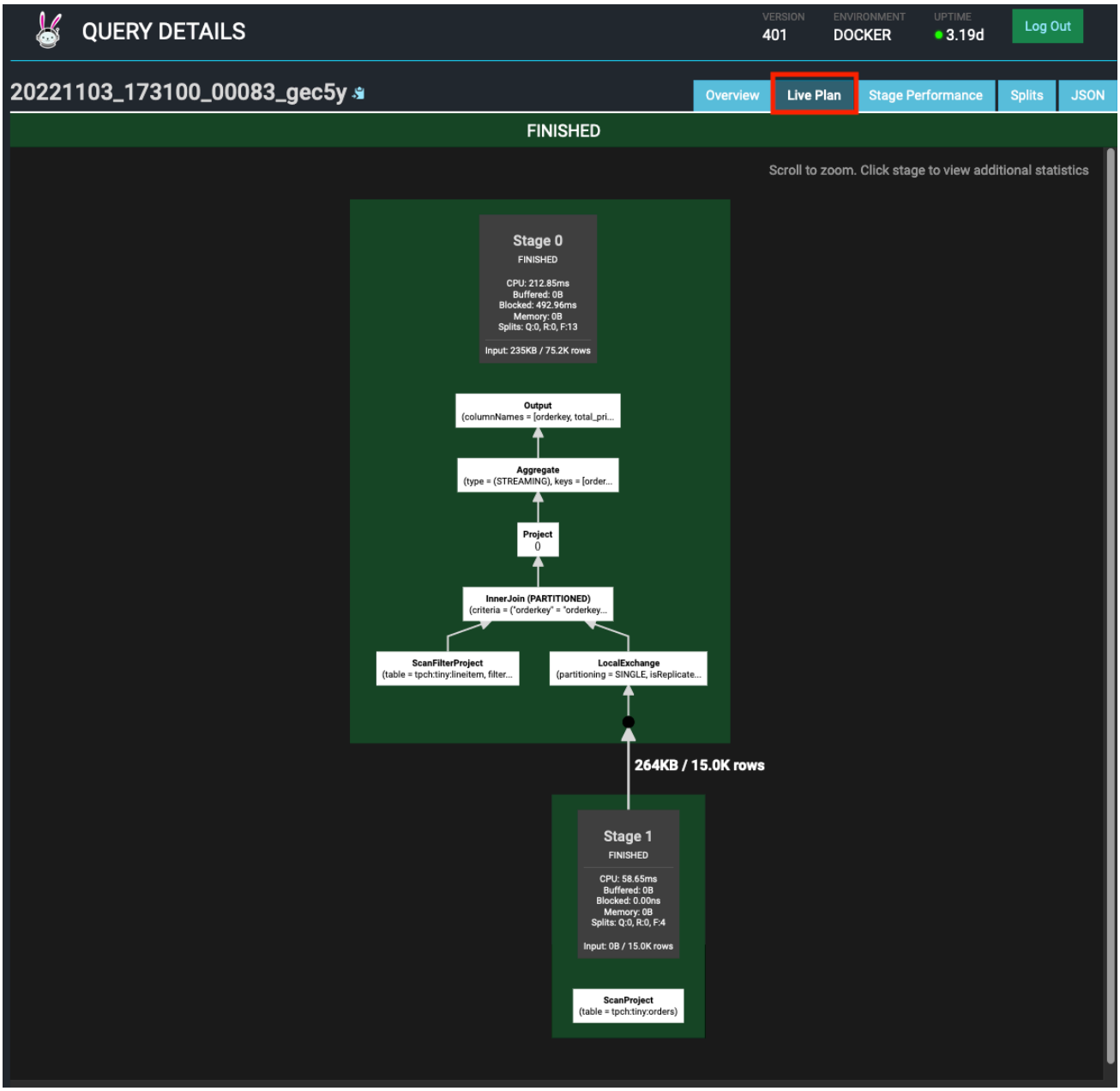


Figure 9: Live plan

If you add a `WHERE o.orderkey = 14983` to the above query, right before the `GROUP BY` clause, you will notice a `ScanFilterProject`(when reading in orders data) with `filterPredicate = ("orderkey_1" = BIGINT '14983')]`.

Filter Pushdown applies a filter while reading the data to avoid pulling unnecessary data into the node's memory.

Exercise:

1. What is the join type of the above query? Hint: Only the `orders` table gets sent over the network.
2. Examine the query plan for the above query with an additional `ORDER BY o.orderkey` clause. How does the query plan change? What do the extra steps(fragments) achieve?
3. Examine the query plan for the following query; what type of join (Hash or Broadcast) does it use?

```
USE tpch.tiny;

EXPLAIN
SELECT
    s.name AS supplier_name,
    SUM(l.extendedprice * (1 - l.discount)) AS
     ↪  total_price_wo_tax
FROM
    lineitem l
    JOIN supplier s ON l.suppkey = s.suppkey
GROUP BY
    s.name;
```

4. Examine the plan for the above query; using `tpch.sf100000`, what changes do you see? How are the partition tasks different, & why?
5. You are tasked with creating an order priority report that counts the number of orders between `1994-12-01` and `1994-13-01` where at least one lineitem was received by the customer later than its

committed date. Display the `orderpriority, order_count` in ascending priority order. Please examine the query plan and explain how it works.

```
USE tpch.tiny;

EXPLAIN
SELECT
    o.orderpriority,
    count(DISTINCT o.orderkey) AS order_count
FROM
    orders o
    JOIN lineitem l ON o.orderkey = l.orderkey
WHERE
    o.orderdate >= date '1994-12-01'
    AND o.orderdate < date '1994-12-01' + INTERVAL '3' MONTH
    AND l.commitdate < l.receiptdate
GROUP BY
    o.orderpriority
ORDER BY
    o.orderpriority;
```

The query plan we get with the EXPLAIN keyword displays the plan the OLAP DB aims to perform. If we want to see the details of CPU utilization, data distribution, partition skew, etc., we can use the EXPLAIN ANALYZE command, which runs the query to get accurate metrics.

When examining a query plan, here are the key points to consider to optimize your query

1. Reduce the amount of data to read (techniques covered in next chapter).

2. Reduce the amount of data to exchange between stages. We can do this by selecting only the columns necessary (instead of `select *`), and column formatting (covered in the next chapter)
3. Check if we can use a Broadcast join instead of a hash join. We can force broadcast join by increasing the broadcast size threshold.

4.5 Reduce the amount of data to be processed with column-oriented formatting, partitioning, and bucketing

4.5.1 Column-oriented formatting

Let's look at an example to see how columnar formatting can increase query efficiency. We will use the `minio` catalog to store data in minio (S3 alternative) and query with Trino. We are using the `Apache Parquet` columnar format.

```
DROP SCHEMA IF EXISTS minio.tpch;

CREATE SCHEMA minio.tpch WITH (location = 's3a://tpch/');
-- tpch is a bucket in minio, precreated for you with your
 ↪  docker container

DROP TABLE IF EXISTS minio.tpch.lineitem_wo_encoding;

CREATE TABLE minio.tpch.lineitem_wo_encoding (
    orderkey bigint,
    partkey bigint,
    suppkey bigint,
    linenumber integer,
    quantity double,
```

```
    extendedprice double,
    discount double,
    tax double,
    shipinstruct varchar(25),
    shipmode varchar(10),
    COMMENT varchar(44),
    commitdate date,
    linestatus varchar(1),
    returnflag varchar(1),
    shipdate date,
    receiptdate date
) WITH (
    external_location = 's3a://tpch/lineitem_wo_encoding/',
    format = 'TEXTFILE'
);

-- we had to specify TEXTFILE as the format since the default
 ↪  is ORC (a columnar format)

USE tpch.sf1;

INSERT INTO
    minio.tpch.lineitem_wo_encoding
SELECT
    orderkey,
    partkey,
    suppkey,
    linenumber,
    quantity,
    extendedprice,
    discount,
```

```
    tax,
    shipinstruct,
    shipmode,
    COMMENT,
    commitdate,
    linestatus,
    returnflag,
    shipdate,
    receiptdate
FROM
    lineitem;

DROP TABLE IF EXISTS minio.tpch.lineitem_w_encoding;

CREATE TABLE minio.tpch.lineitem_w_encoding (
    orderkey bigint,
    partkey bigint,
    suppkey bigint,
    linenumber integer,
    quantity double,
    extendedprice double,
    discount double,
    tax double,
    shipinstruct varchar(25),
    shipmode varchar(10),
    COMMENT varchar(44),
    commitdate date,
    linestatus varchar(1),
    returnflag varchar(1),
    shipdate date,
    receiptdate date
```

```
) WITH (
    external_location = 's3a://tpch/lineitem_w_encoding/',
    format = 'PARQUET'
);

INSERT INTO
    minio.tpch.lineitem_w_encoding
SELECT
    orderkey,
    partkey,
    suppkey,
    linenumber,
    quantity,
    extendedprice,
    discount,
    tax,
    shipinstruct,
    shipmode,
    COMMENT,
    commitdate,
    linestatus,
    returnflag,
    shipdate,
    receiptdate
FROM
    lineitem;
```

If we go to the minio UI (http://localhost:9001) under the tpch bucket, we can see the files under the two paths and their size; note how the parquet encoded file is `142.3 MB` whereas the non-encoded file is `215.1 MB` (33% size decrease). While this size difference may seem small, as the

data set gets larger, the percentage difference gets higher due to the ability of the columnar format to enable better compression.

When we query the data, parquet enables reading more data into memory due to the properties of columnar encoding. Let's look at an example.

```
SELECT
    suppkey,
    sum(quantity) AS total_qty
FROM
    minio.tpch.lineitem_w_encoding
GROUP BY
    suppkey;
-- 2.22 [6M rows, 14.5MB] [2.7M rows/s, 6.54MB/s]

SELECT
    suppkey,
    sum(quantity) AS total_qty
FROM
    minio.tpch.lineitem_wo_encoding
GROUP BY
    suppkey;
-- 10.98 [6M rows, 215MB] [547K rows/s, 19.6MB/s]
```

Look at the resource utilization section for these two queries (go to http://localhost:8080, make sure to check the `Finished` option). We can see that the query using the encoded lineitem table processed `2.51 million rows per sec` whereas the non-encoded lineitem table processed `859K rows per sec`. Thus, we can see how effective columnar encoding is.

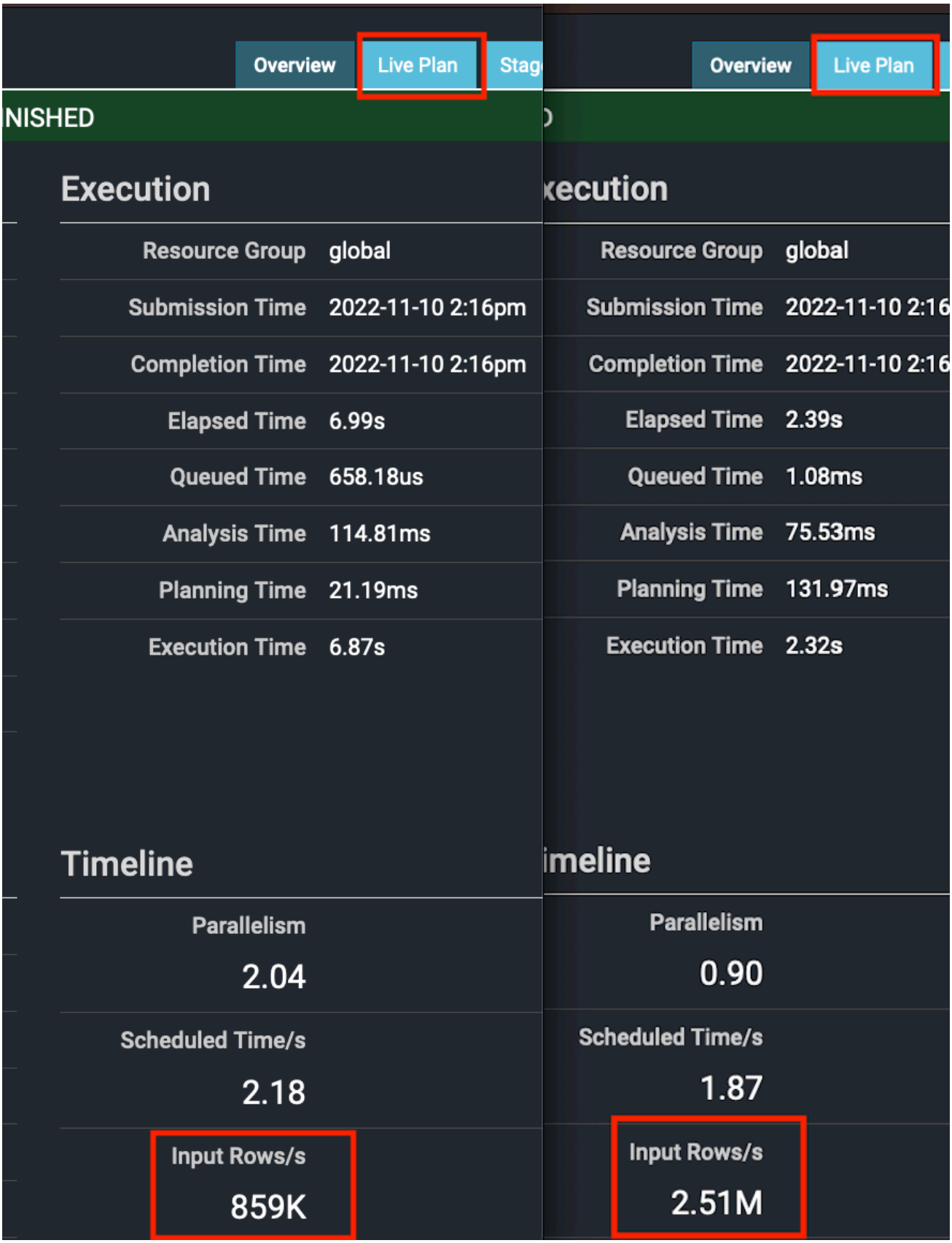


Figure 10: Encoded v Non-encoded rows/sec processed

Note: The number of rows processed per second may vary slightly, caused by varying load on the OLAP DB engine, but the differences between row and column data format will be visible.

Now that we see how columnar formatting is much more efficient, let's understand how it works. The OLAP DB processes data by reading(or streaming) the data into memory and performing the necessary transformations. Analytical queries operating on billions of rows will take a long time to read/stream through all the rows.

Standard OLTP databases(Postgres, MySQL) store data in a row format. You can think of this as a file (lineitem) with the format shown below.

```
orderkey,partkey,suppkey,linenumber,quantity,extendedprice,discount,
tax,returnflag,linestatus,shipdate,commitdate,receiptdate,
shipinstruct,shipmode,comment

1,1552,93,1,17,24710.35,0.04,0.02,N,O,1996-03-13,1996-02-
 ↪  12,1996-03-22,"DELIVER IN PERSON",TRUCK,"egular courts
 ↪  above the"

1,674,75,2,36,56688.12,0.09,0.06,N,O,1996-04-12,1996-02-
 ↪  28,1996-04-20,"TAKE BACK RETURN",MAIL,"ly final
 ↪  dependencies: slyly bold "

1,637,38,3,8,12301.04,0.10,0.02,N,O,1996-01-29,1996-03-
 ↪  05,1996-01-31,"TAKE BACK RETURN","REG AIR","riously.
 ↪  regular, express dep"
```

Let's look at an analytical query that calculates the total quantity for each supplier key and see how the DB will process the data.

```
USE tpch.tiny;

SELECT
    suppkey,
    sum(quantity) AS total_qty
FROM
    lineitem
GROUP BY
    suppkey
ORDER BY
    2 DESC;
```

The DB will read/stream (depending on the table size) all the rows into memory (16 columns) and create a dictionary that keeps track of the suppkey and the sum of the quantity.

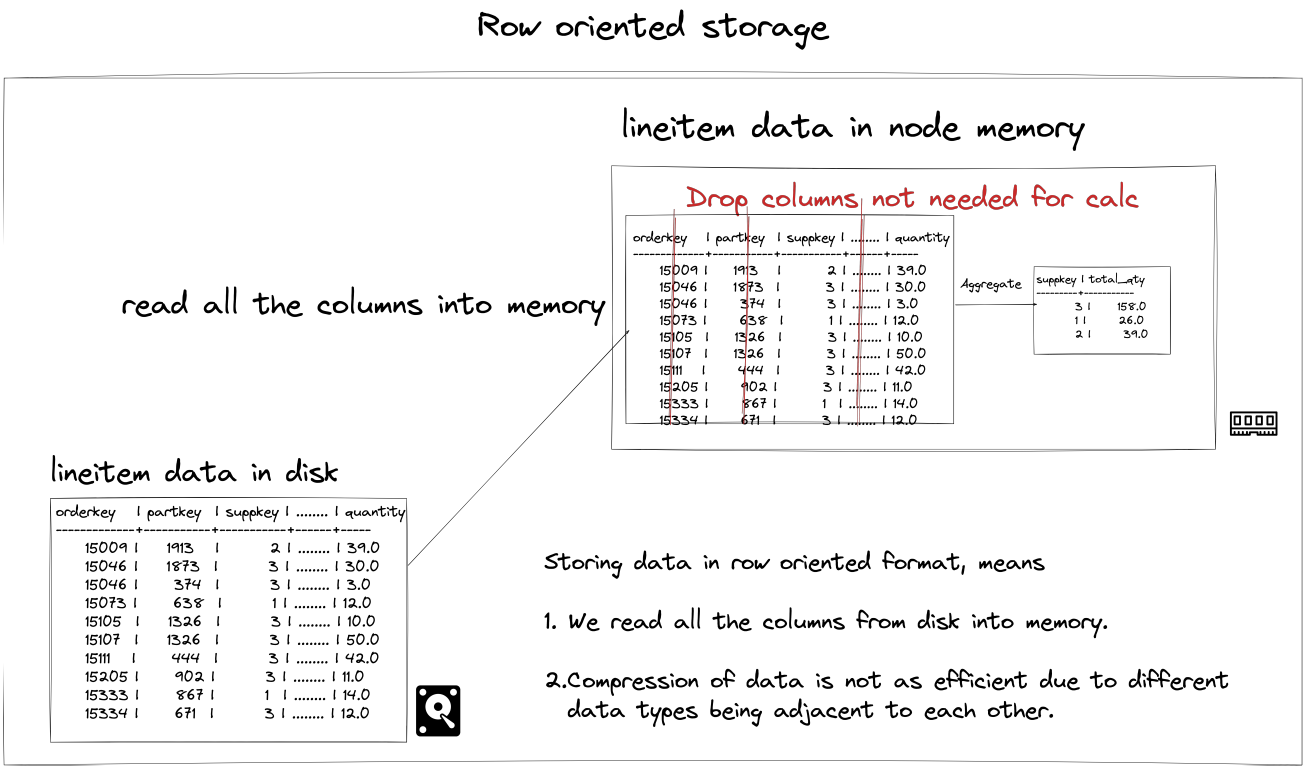


Figure 11: Row oriented storage

We can see how reading and storing 14 columns that we do not need in memory can cause a significant delay. Analytical queries often involve a few columns out of many columns. What if we could only read the required columns into memory? Column-oriented data format helps us do this. You can think of column-oriented storage as a file (lineitem) with the format shown below.

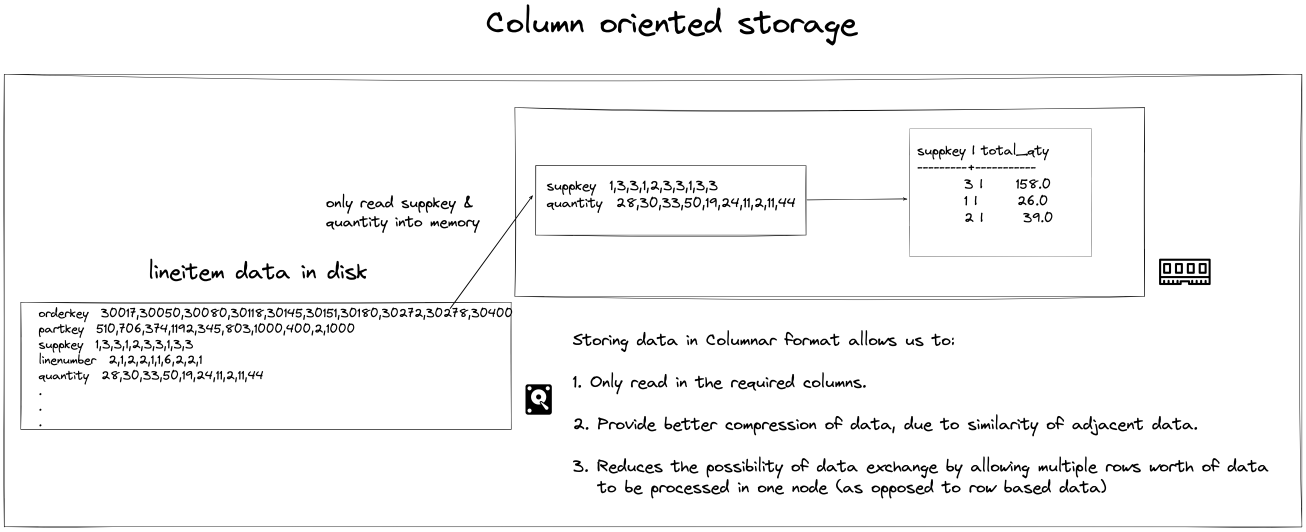


Figure 12: Column oriented storage

For the analytical query that calculates the total quantity for each supplier key, the OLAP DB will read only the `suppkey` and `quantity` columns into memory. Only reading the required columns into memory allows the OLAP DB to fit more data into memory, allowing faster processing. Since the data is stored one column after another, it's called column-oriented formatting. The reading of only required columns is called Column Pruning.

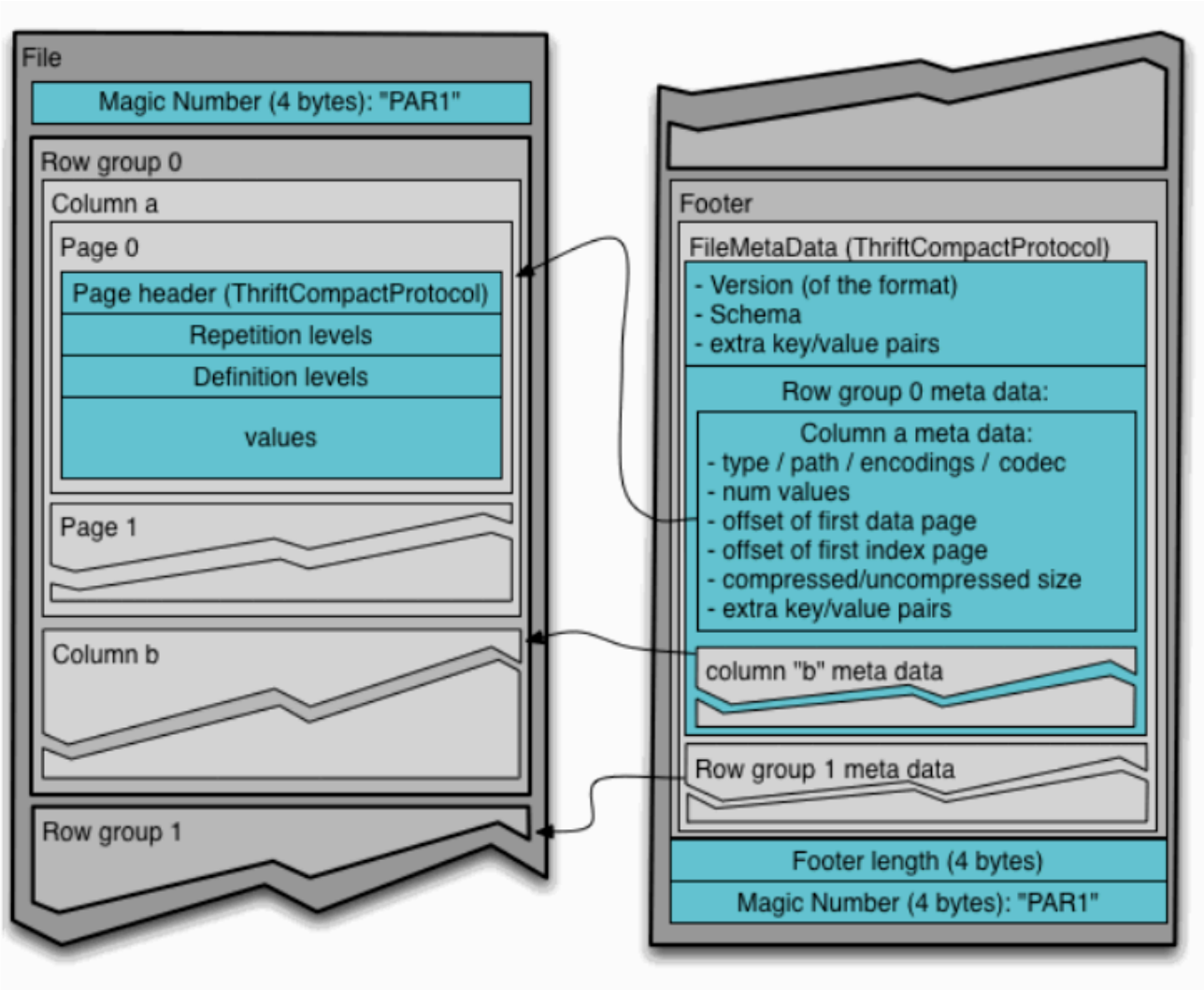
In addition to lowering memory used, column-oriented formatting also enables efficient compression due to the data types of consecutive entries being the same(a column has a single data type). Compression further reduces the data size, allowing the OLAP DB to fit even more data in memory.

In addition to column-oriented formatting, modern data file formats have additional features that help OLAP DB efficiently read only the required data. Some OLAP DBs have their version of column-oriented encoding (e.g., Snowflake); the most popular ones are Apache Parquet, & Apache ORC.

Let's look at how Apache Parquet stores data and helps the OLAP DB read only the necessary data. When we create a table with `Parquet` compression, every file that makes up a table is of parquet format. Each parquet file is made up of the following.

1. RowGroups: Each `RowGroup` denotes a set of rows in the table and has all the column values of those rows. `RowGroup`'s size is tuned to allow OLAP DB engines to read entire sets of rows into memory.
2. ColumnChunks: Within every `RowGroup`, there is a chunk for each column (stored in columnar format). In a parquet file, the column chunks contain the compressed column data.
3. FileMetaData: This is present at the footer of the parquet file and contains metadata about which rows are in which `RowGroup` and information about the column values in `ColumnChunk` within its specific RowGroup. The column chunk metadata includes encoding types, the number of values in that chunk, the file offset of the chunk, & size of the data.

The below image represents the storage of a chunk of our lineitem table.



ref: https://parquet.apache.org/docs/file-format/

When the OLAP DB executes a query on a parquet table, it.

1. Reads the `FileMetaData` to identify all the places in the file(offset) it has to read using the column metadata.
2. Reads only the required columns data by directly reading off the column offset.

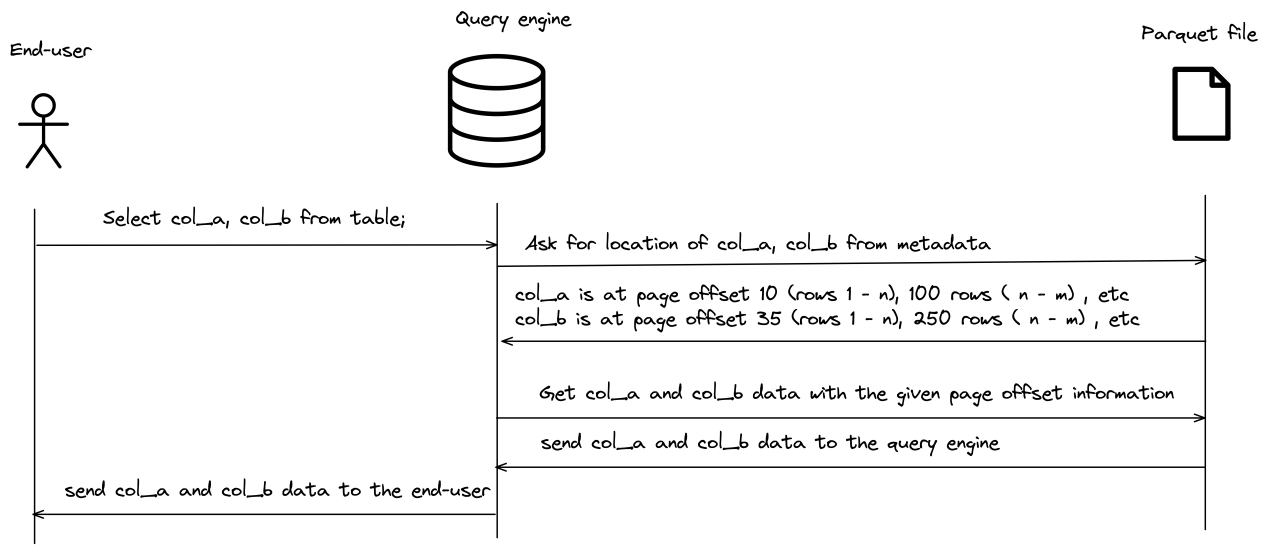


Figure 13: Reading data from parquet

Exercise:

1. For the `orders` table in the `tpch.sf1` schema, create encoded and non-encoded tables in the minio schema using the lineitem example above. Note the difference in size between the encoded and non-encoded orders table. Run the below query on the encoded and non-encoded tables and note the difference in rows processed per second.

```
SELECT
    custkey,
    sum(totalprice) AS total_cust_price
FROM
    minio.tpch.orders_w_encoding -- &
 ↪  minio.tpch.orders_wo_encoding
GROUP BY
    1;
```

Caveats: While encoding has a lot of benefits when reading the data, it also incurs upfront costs when writing the data. The OLAP DB has to perform work to create data in the partition format. Given the nature of analytical queries, which are much more read-heavy than writes & the size of data, which lends itself to better compression, this is a perfectly valid tradeoff.

4.5.2 Partitioning

Columnar encoding enables the OLAP DB to only read the required chunks of data from the encoded files, but it still requires reading the metadata of each encoded file. Reading in metadata of each encoded file requires that the OLAP DB access the file, but what if there is a way to avoid reading the individual file's metadata? Column partitioning is the answer.

Column partitioning (aka HIVE style partitioning) involves creating physical folders based on values in the partitioned column(s). For example, if you partition a table by year, the data will be stored as a set of folders, with one folder per year. So when we filter the data for a specific year, the OLAP DB will know which folder to look into, thereby skipping reading other years' files.

Partition Pruning is the process of elimination of reading unnecessary partitions.

Let's look at an example:

```
DROP TABLE IF EXISTS
minio.tpch.lineitem_w_encoding_w_partitioning;

CREATE TABLE minio.tpch.lineitem_w_encoding_w_partitioning (
```

```
    orderkey bigint,
    partkey bigint,
    suppkey bigint,
    linenumber integer,
    quantity double,
    extendedprice double,
    discount double,
    tax double,
    shipinstruct varchar(25),
    shipmode varchar(10),
    COMMENT varchar(44),
    commitdate date,
    linestatus varchar(1),
    returnflag varchar(1),
    shipdate date,
    receiptdate date,
    receiptyear varchar(4)
) WITH (
    external_location =
    's3a://tpch/lineitem_w_encoding_w_partitioning/',
    partitioned_by = ARRAY ['receiptyear'],
    format = 'PARQUET'
);

USE tpch.tiny;

INSERT INTO
    minio.tpch.lineitem_w_encoding_w_partitioning
SELECT
    orderkey,
    partkey,
```

```
    suppkey,
    linenumber,
    quantity,
    extendedprice,
    discount,
    tax,
    shipinstruct,
    shipmode,
    COMMENT,
    commitdate,
    linestatus,
    returnflag,
    shipdate,
    receiptdate,
    cast(year(receiptdate) AS varchar(4)) AS receiptyear
FROM
    lineitem;
```

Note that we manually create a `receiptyear` column during insert since we do not have that column naturally in the `lineitem` table. If we go to the minio console (http://localhost:9001), under the path `tpch/lineitem_w_encoding_w_partitioning`, we will see one folder per year, named with the format `receiptyear=YYYY`.

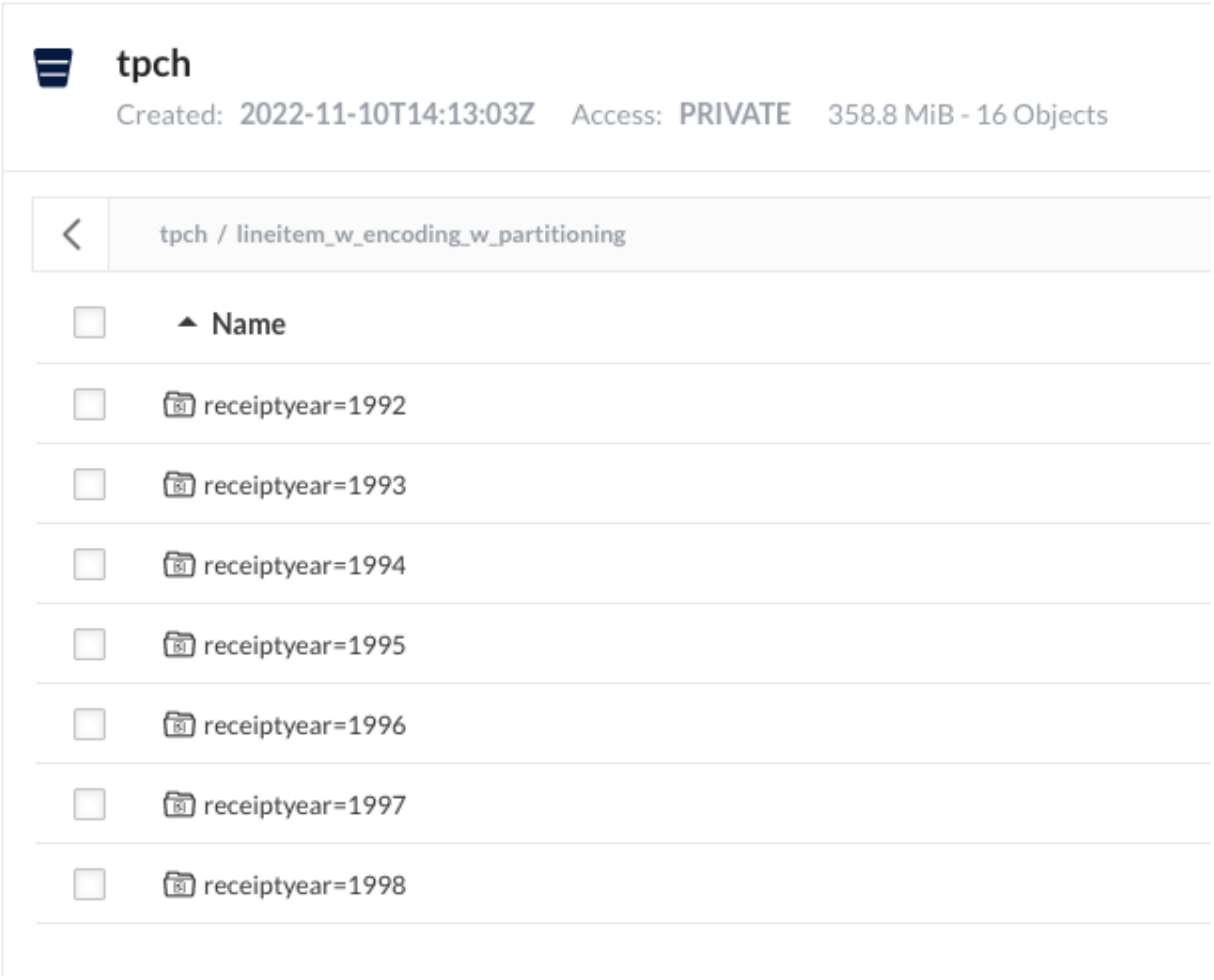


Figure 14: Partitioned folders

The metadata DB will store partition information of a specific table. Let's look at the metadata DB in your project directory using `make metadata-db` to open the metadata DB cli.

```
-- run 'make metadata-db' or
-- 'docker exec -ti mariadb /usr/bin/mariadb -padmin'
-- on you terminal
```

```
SELECT * FROM metastore_db.PARTITIONS;
exit;
```

Let's see how partitioning can improve query performance.

```
-- To get the inputs, look for
-- Estimates: {rows: <input_rows> in
-- the query plan
EXPLAIN ANALYZE
SELECT
    *
FROM
    tpch.tiny.lineitem
WHERE
    year(receiptdate) = 1994;
-- Input: 60175 rows

EXPLAIN ANALYZE
SELECT
    *
FROM
    minio.tpch.lineitem_w_encoding_w_partitioning
WHERE
    receiptyear = '1994';
-- Input: 9525 rows
```

We can see how filtering on the partitioned column (`receiptYear`) allows the OLAP DB to directly read only the required files within the `receiptYear=1994` folder.

We can also partition by multiple columns; for example, we can partition by `receiptyear` and `receiptmonth`, which will create folders of the structure `table_path/receiptyear=YYYY/receiptmonth=MM/`.

Caveats: We can significantly reduce the amount of data scanned when querying a table with partitioning. However, there are a few caveats that one needs to be aware of; they are:

1. Cardinality: Cardinality refers to a column's number of unique values. Partitioning is a great approach when Cardinality is low (e.g., `year(receiptdate)` in lineitem has only seven distinct values). But when the Cardinality of a column is high (e.g., numerical values like price, dates, customer ids, etc.), the number of partitions will be significant. A high number of partitions causes over-partitioning, where each partition will store a small number of values. Queries with filters on non-partitioned columns are affected significantly by over-partitioning. Some OLAP DBs have a limit on the number of partitions, e.g., Bigquery has a limit of 4000 partitions per table.
2. Expensive reprocessing: Partitioning requires processing data into separate folders. Suppose you decide to change the partition columns. In that case, this will involve reprocessing the entire dataset, which can be significant depending on the size of the dataset and use case. Note Newer table formats like Apache Iceberg, & Apache Hudi, have ability to avoid expensive reprocessing.

4.5.3 Bucketing (aka Clustering)

We saw how partitioning is not a good fit for columns with high Cardinality. One approach to overcome this is Bucketing. Bucketing involves

splitting a table into multiple “buckets” based on values in one or more column(s).

When we bucket a table, we also specify the number of buckets. The OLAP DB will send the row to a node determined using a hash function on the bucketed column(s).

When we query the table with a filter on a bucketed column, the OLAP DB will use the hash of the value specified in the filter to determine which bucket to scan.

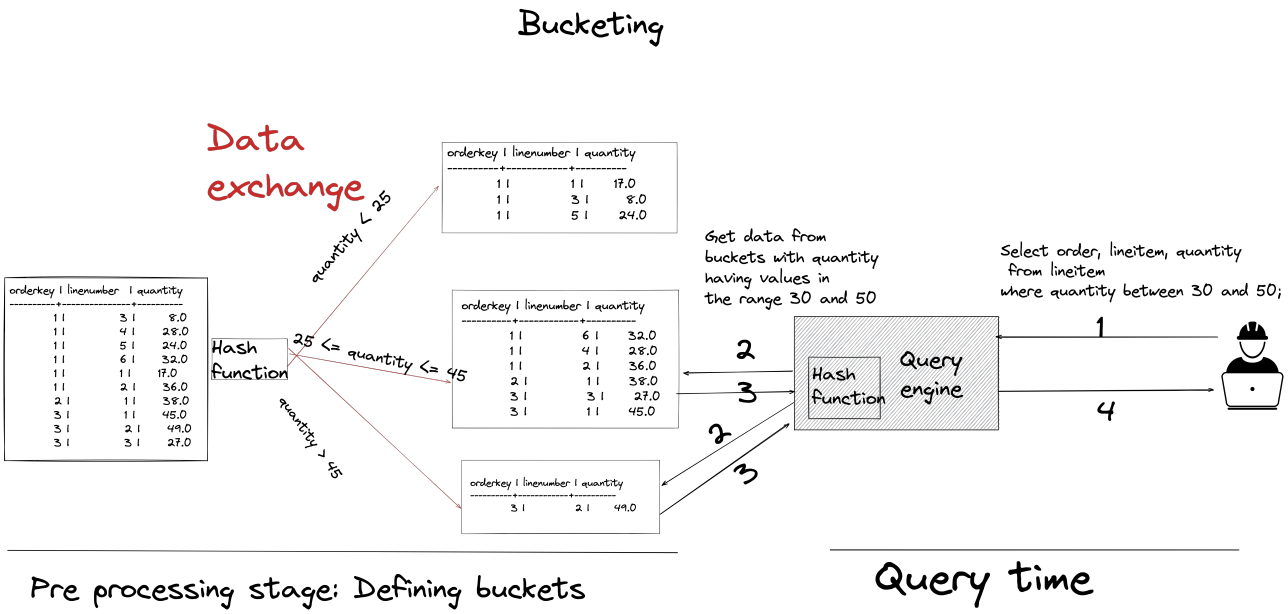


Figure 15: Bucketing

Let’s look at an example.

```
DROP TABLE IF EXISTS
minio.tpch.lineitem_w_encoding_w_bucketing;
```

```
CREATE TABLE minio.tpch.lineitem_w_encoding_w_bucketing (
    orderkey bigint,
    partkey bigint,
    suppkey bigint,
    linenumber integer,
    quantity double,
    extendedprice double,
    discount double,
    tax double,
    shipinstruct varchar(25),
    shipmode varchar(10),
    COMMENT varchar(44),
    commitdate date,
    linestatus varchar(1),
    returnflag varchar(1),
    shipdate date,
    receiptdate date
) WITH (
    external_location =
    's3a://tpch/lineitem_w_encoding_w_bucketing/',
    format = 'PARQUET',
    bucket_count = 75,
    bucketed_by = ARRAY ['quantity']
);

USE tpch.tiny;

INSERT INTO
    minio.tpch.lineitem_w_encoding_w_bucketing
SELECT
    orderkey,
```

```
    partkey,
    suppkey,
    linenumber,
    quantity,
    extendedprice,
    discount,
    tax,
    shipinstruct,
    shipmode,
    COMMENT,
    commitdate,
    linestatus,
    returnflag,
    shipdate,
    receiptdate
FROM
    lineitem;

EXPLAIN ANALYZE
SELECT
    *
FROM
    lineitem
WHERE
    quantity >= 30
    AND quantity <= 45;
-- Input: 60,175 rows (0B), Filtered: 68.14%

EXPLAIN ANALYZE
SELECT
    *
```

```
FROM
    minio.tpch.lineitem_w_encoding_w_bucketing
WHERE
    quantity >= 30
    AND quantity <= 45;
-- Input: 21,550 rows (3.14MB), Filtered: 11.03%
```

We can see how filtering on bucketed columns reduces the amount of data scanned from 60k rows to 21k rows (65% reduction in rows scanned).

Caveats: While bucketing is an excellent alternative to partitioning for high cardinality columns, we should be careful about the distribution of the values in the column. A uniform distribution (number of rows per unique column value) will help make the buckets similar in size, making reads efficient. If some values (in the bucketed column) are much more prevalent, this may cause one bucket to be much more giant compared to the rest, thereby making reads less effective.

Exercise:

1. Create a table bucketed by `quantity` with 100 buckets, and filter for `quantity >= 30 AND quantity <= 45`; what is the input size?

```
DROP TABLE IF EXISTS
minio.tpch.lineitem_w_encoding_w_bucketing_eg;

CREATE TABLE minio.tpch.lineitem_w_encoding_w_bucketing_eg (
    orderkey bigint,
    partkey bigint,
    suppkey bigint,
```

```
    linenumber integer,
    quantity double,
    extendedprice double,
    discount double,
    tax double,
    shipinstruct varchar(25),
    shipmode varchar(10),
    COMMENT varchar(44),
    commitdate date,
    linestatus varchar(1),
    returnflag varchar(1),
    shipdate date,
    receiptdate date
) WITH (
    external_location =
    's3a://tpch/lineitem_w_encoding_w_bucketing_eg/',
    format = 'PARQUET',
    bucket_count = 100,
    bucketed_by = ARRAY ['quantity']
);

USE tpch.tiny;

INSERT INTO
    minio.tpch.lineitem_w_encoding_w_bucketing_eg
SELECT
    orderkey,
    partkey,
    suppkey,
    linenumber,
    quantity,
```

```
    extendedprice,
    discount,
    tax,
    shipinstruct,
    shipmode,
    COMMENT,
    commitdate,
    linestatus,
    returnflag,
    shipdate,
    receiptdate
FROM
    lineitem;

EXPLAIN ANALYZE
SELECT
    *
FROM
    minio.tpch.lineitem_w_encoding_w_bucketing_eg
WHERE
    quantity >= 30
    AND quantity <= 45;
```

5 Calculate aggregate metrics but keep all the rows, rank rows, and compare values across rows with window functions

5.1 Window = A set of rows identified by values present in one or more column(s)

A window refers to a set of rows with the same value for a specified column(s).

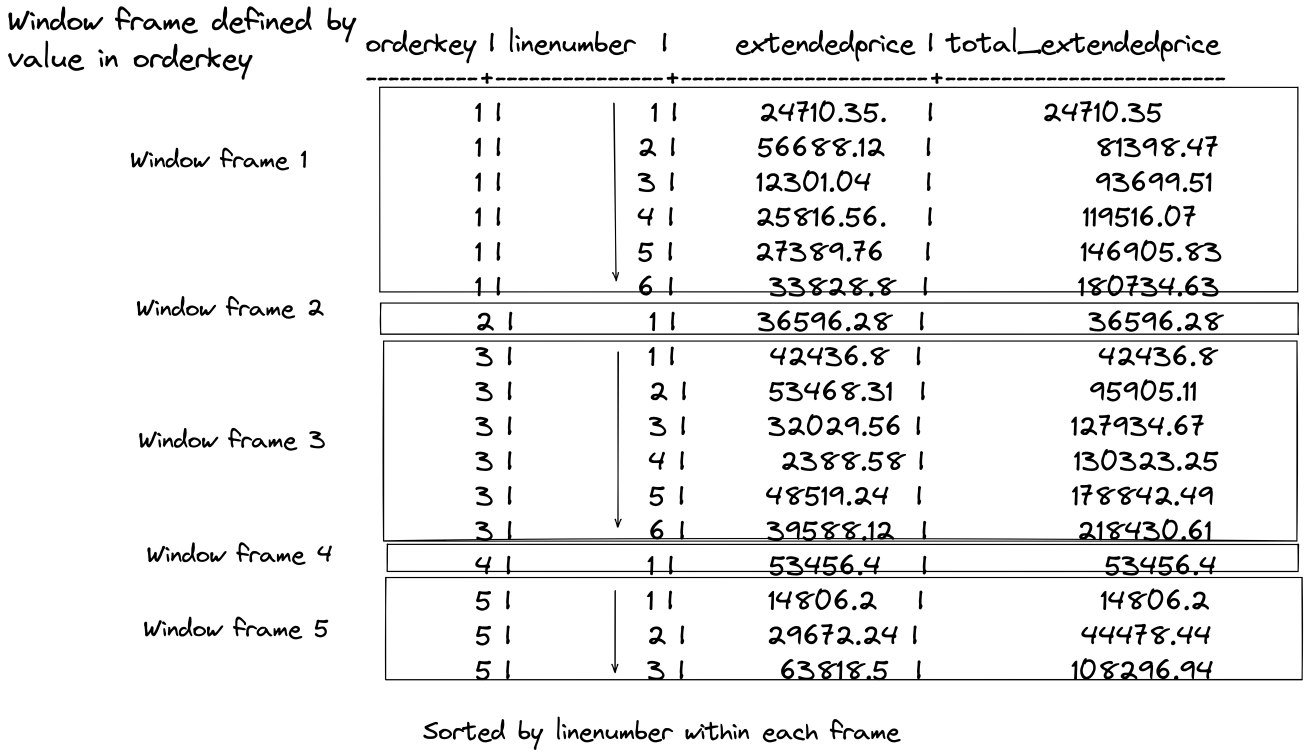


Figure 16: Window

With a window defined, we can

1. Apply a function to the rows in that window
2. Sort the data within the window in any required order

5.2 A window definition has a function, partition (column(s) to identify a window), and order (order of rows within the window)

A window definition involves the following:

1. Function: The function to be applied to the rows in the window.
2. Partition by (Optional): The column(s) that are used to define the windows. `Partition by` is optional; The entire table is considered as one window in the absence of the `Partition by` clause.
3. Order by (optional): We can specify the order of rows based on some column(s) within a window. The ordering will only apply to rows within a window. `Order by` is optional; rows within a window are not ordered without this clause.

Let's consider an example where we want to show the orderkey, linenumber, and the sum of the extendedprice. The sum of the extendedprice (round to 2 decimal digits) should reveal the running sum of the extendedprice of items within an order.

```
USE tpch.tiny;

SELECT
    orderkey,
    linenumber,
    extendedprice,
    ROUND(
        sum(extendedprice) over(
```

```
            PARTITION by orderkey
            ORDER BY
                linenumber
        ),
        2
    ) AS total_extendedprice
FROM
    lineitem
ORDER BY
    orderkey,
    linenumber
LIMIT
    20;
```

Our window is defined by the `orderkey` as the partition, & the rows within the window are ordered by `linenumber`. Note that the sum function creates a cumulative sum adding the extendedprice of one item at a time.

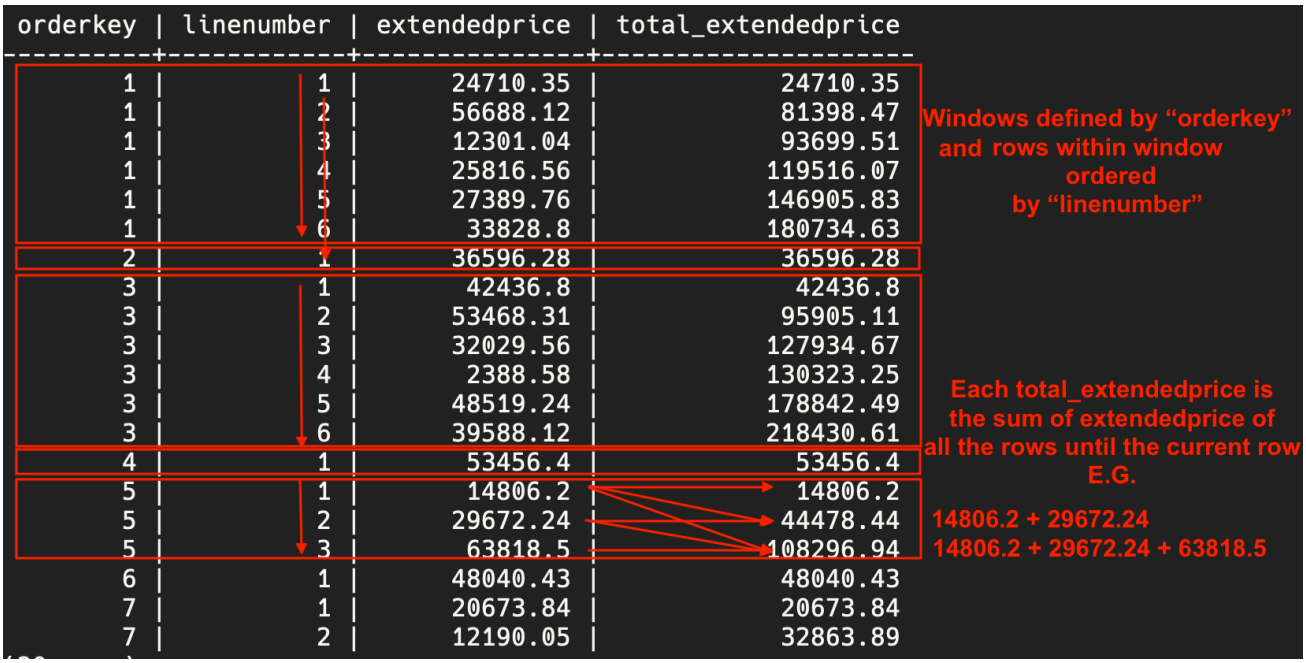


Figure 17: Window calculation

The Order by clause is used to apply the window function in a rolling fashion. Without the Order by clause, the window function calculates the result based on all the values in the window.

Exercise:

1. Run the above query without the `ORDER BY` clause; what do you see? Why does this happen?

5.3 Calculate running metrics with Window aggregate functions

One of the primary use cases for window functions is to calculate aggregates while being able to see all the rows in the output. The standard

aggregate functions are `SUM, MIN, MAX, AVG, & COUNT`. Most OLAP DBs have additional aggregate functions (e.g. Trino aggregate functions)

Example:

1. Create a report at customer_name and orderdate level/granularity with the following metrics
    1. total_price: Sum of totalprice of all the orders
    2. cumulative_sum_total_price: Running sum of the sum of totalprice of all the orders until the current date.

```
USE tpch.tiny;

SELECT
    c.name AS customer_name,
    o.orderdate,
    SUM(o.totalprice) AS total_price, -- We have it here just
     ↪  for comparison purposes
    ROUND(
        SUM(SUM(o.totalprice)) over(
            PARTITION by o.custkey
            ORDER BY
                o.orderdate
        ),
        2
    ) AS cumulative_sum_total_price
FROM
    orders o
    JOIN customer c ON o.custkey = c.custkey
GROUP BY
    c.name,
    o.custkey,
```

```
    o.orderdate
ORDER BY
    o.custkey,
    o.orderdate
LIMIT
    20;
```

Exercise:

1. Create a report at nation_name & order_year level/granularity, with the following metrics
    1. total_price: Sum of the total price (divided by 100,000 for readability)
    2. cumulative_sum_total_price: Average total price for all the years until the current year (divided by 100,000 for readability) and round to 2 decimal digits

```
USE tpch.tiny;

SELECT
    n.name AS nation_name,
    year(o.orderdate) AS order_year,
    ROUND(sum(o.totalprice) / 100000, 2) AS total_price,
    ROUND(
        avg(sum(o.totalprice)) over(
            PARTITION by n.name
            ORDER BY
                year(o.orderdate)
        ) / 100000,
        2
```

```
    ) AS cumulative_sum_total_price
FROM
    orders o
    JOIN customer c ON o.custkey = c.custkey
    JOIN nation n ON c.nationkey = n.nationkey
GROUP BY
    n.name,
    year(o.orderdate)
ORDER BY
    n.name,
    year(o.orderdate)
LIMIT
    20;
```

5.4 Rank rows based on column(s) with Window ranking functions

An everyday use for window functions is to rank the rows within a window. The `ORDER BY` clause determines the ranking.

Example:

1. Create a report showing the list of all customers(who placed orders), order date, and a rank column that ranks customer orders based on descending `totalprice`. We can do this as follows.

```
USE tpch.tiny;

SELECT
    custkey,
```

```
    orderdate,
    format('%,.2f', totalprice) AS totalprice,
    RANK() OVER(
        PARTITION BY custkey
        ORDER BY
            totalprice DESC
    ) AS rnk
FROM
    orders
ORDER BY
    custkey,
    rnk
LIMIT
    15;
```

We see that the rows are ranked based on `totalprice`. We also see the rank reset to 1 after the end of a partition (defined by custkey). The `RANK` function ranks the rows based on the values in the ORDER BY clause.

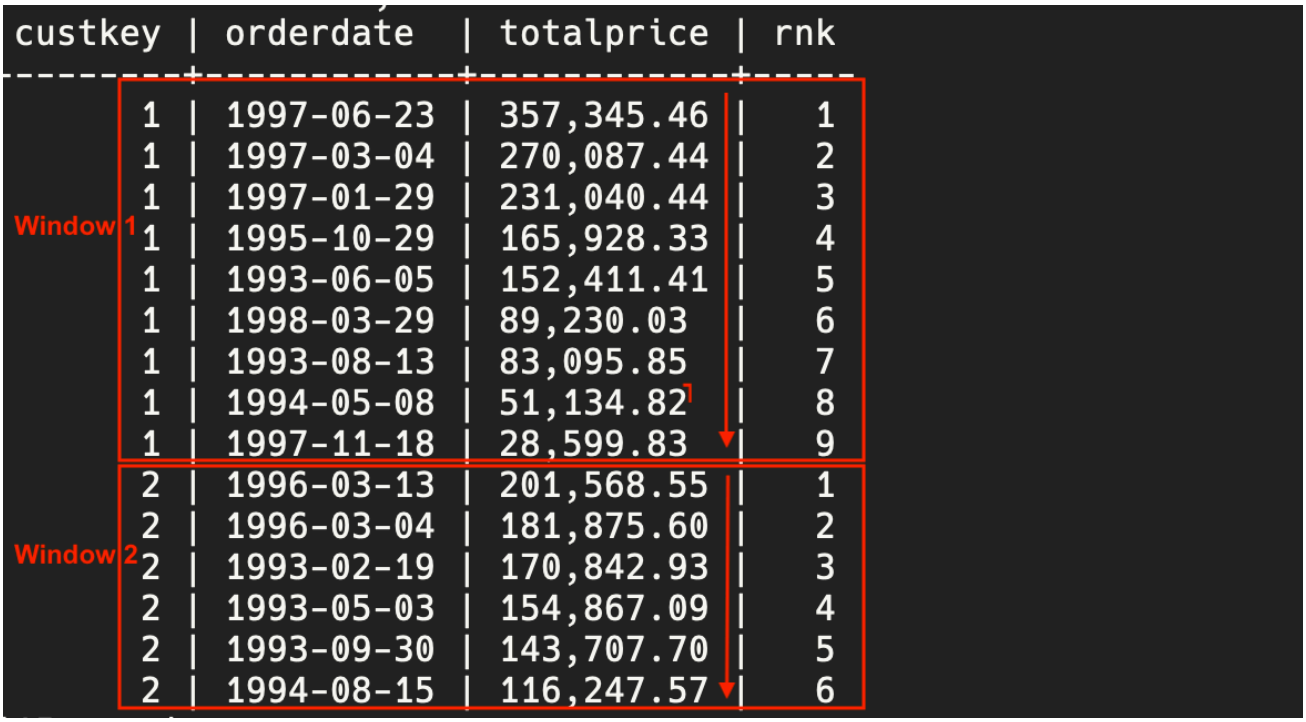


Figure 18: Rank

Exercise:

1. Try the above query without the order by clause; what is the output? Does ranking rows make sense without an `ORDER BY` clause?

Although the `RANK` function works in most cases, there are scenarios where we would want to use the `DENSE_RANK` or `ROW_NUMBER` functions. To see the differences between RANK, DENSE_RANK, & ROW_NUMBER, let's consider an example.

```
USE tpch.tiny;

SELECT
    orderkey,
    discount,
```

```
    RANK() OVER(
        PARTITION BY orderkey
        ORDER BY
            discount DESC
    ) AS rnk,
    DENSE_RANK() OVER(
        PARTITION BY orderkey
        ORDER BY
            discount DESC
    ) AS dense_rnk,
    ROW_NUMBER() OVER(
        PARTITION BY orderkey
        ORDER BY
            discount DESC
    ) AS row_num
FROM
    lineitem
WHERE
    orderkey = 42624 -- this is an example orderkey that has
 ↪  multiple discounts of the same value
LIMIT
    10;
```

orderkey	discount	rnk	dense_rnk	row_num
42624	0.1	1	1	1
42624	0.1	1	1	2
42624	0.1	1	1	3
42624	0.1	1	1	4
42624	0.1	1	1	5
42624	0.08	6	2	6
42624	0.05	7	3	7

Figure 19: Rank, Dense_rank, & Row num

Here is a quick definition of each

1. RANK: Ranks the rows starting from 1. 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. 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 and does not create any repeating values. The absence of an 'ORDER BY" clause will not cause the ROW_NUMBERs to be all 1's.

Exercise:

1. Create a report that shows for every supplier nation the top 3 years and months of orders during which they sold the most items (quantity). If a nation has multiple year-month combinations that qualify for their top 3 spots, show them all.

```
USE tpch.tiny;

SELECT *
FROM (
SELECT
    n.name AS supplier_nation,
    YEAR(o.orderdate) AS order_year,
    MONTH(o.orderdate) AS order_month,
    SUM(l.quantity),
    DENSE_RANK() OVER(
        PARTITION BY n.name
        ORDER BY
            SUM(l.quantity) DESC
    ) AS rnk
FROM
    orders o
    JOIN lineitem l ON o.orderkey = l.orderkey
    JOIN supplier s ON l.suppkey = s.suppkey
    JOIN nation n ON s.nationkey = n.nationkey
GROUP BY
    n.name,
    YEAR(o.orderdate),
    MONTH(o.orderdate))
WHERE rnk <= 3
ORDER BY
    supplier_nation,
    rnk
LIMIT
    30;
```

5.5 Compare column values across rows with Window value functions

Another everyday use for the window function is when you want to use values from other rows to calculate the current row. Two main value functions are

1. LAG(column, n): This function gets the column value from the previous nth (default is 1) row. The `ORDER BY` clause determines the order of the rows.
2. LEAD(column, n): This function gets the column value from the future nth (default is 1) row. The `ORDER BY` clause determines the order of the rows.

For example, if you want the percentage change in sales month over month, you must compare a month's value with the previous month's value. In this case, you can use a lag window function, as shown below.

```
USE tpch.tiny;

SELECT
    ordermonth,
    total_price,
    LAG(total_price) OVER(
        ORDER BY
            ordermonth
    ) AS prev_month_total_price
FROM
    (
        SELECT
            date_format(orderdate, '%Y-%m') AS ordermonth,
            ROUND(SUM(totalprice) / 100000, 2) AS total_price
             ↪  -- divide by 100,000 for readability
```

```
        FROM
            orders
        GROUP BY
            1
    )
LIMIT
    10;
```

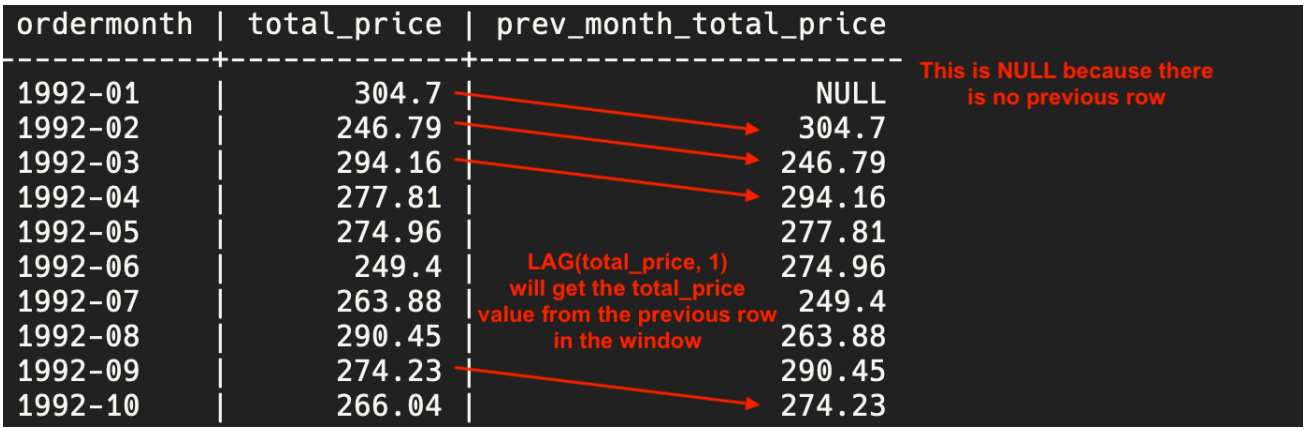


Figure 20: Lag

By default, LAG looks at one row before (order defined by the `ORDER BY` clause) the current row. We can use LAG to look at a specific nth row behind the current row, using an additional parameter, e.g., `LAG(total_price, 2)` will look at the value of total_price from 2 rows before our current row. Let's look at an example.

Exercise:

1. Create a report at ordermonth level/granularity with the following metrics

1. total_price: The sum of totalprice of all orders for the specific ordermonth
2. prev_month_total_price: Previous months total_price
3. prev_prev_month_total_price: Month before the previous month's total_price

```
USE tpch.tiny;

SELECT
    ordermonth,
    total_price,
    LAG(total_price) OVER(
        ORDER BY
            ordermonth
    ) AS prev_month_total_price,
    LAG(total_price, 2) OVER(
        ORDER BY
            ordermonth
    ) AS prev_prev_month_total_price
FROM
    (
        SELECT
            date_format(orderdate, '%Y-%m') AS ordermonth,
            ROUND(SUM(totalprice) / 100000, 2)
            AS total_price
            -- divide by 100,000 for readability
        FROM
            orders
        GROUP BY
            1
    )
```

```
LIMIT
    24;
```

2. Create a report at customer_name, order_date level/granularity, with the following metrics
    1. totalprice: Sum of totalprice spent by a customer on that orderdate
    2. prev_total_price: Sum of the previous orderdate's totalprice spent by a customer
    3. price_change_percentage: The change in price between the current date and the former date as a percentage. Round it to 2 decimal digits. The change percentage is calculated as ((previous value - current value)/previous values) * 100.

```
USE tpch.tiny;

SELECT
    customer_name,
    order_date,
    total_price,
    lag(total_price) over(
        PARTITION by customer_name
        ORDER BY
            order_date
    ) AS prev_total_price,
    ROUND(
        (
            lag(total_price) over(
                PARTITION by customer_name
                ORDER BY
```

```
                    order_date
            ) - total_price
        ) / lag(total_price) over(
            PARTITION by customer_name
            ORDER BY
                order_date
        ) * 100,
        2
    ) AS price_change_percentage
FROM
    (
        SELECT
            c.name AS customer_name,
            o.orderdate AS order_date,
            sum(o.totalprice) AS total_price
        FROM
            orders o
            JOIN customer c ON o.custkey = c.custkey
        GROUP BY
            1,
            2
    )
ORDER BY
    customer_name,
    order_date;
```

3. Create a report at customer_name, order_date level/granularity, with the following metrics

    1. totalprice: The totalprice spent by a customer on that order-date

2. has_total_price_increased: Boolean flag to indicate if the current_price is greater than the previous days price
3. will_total_price_increase: Boolean flag to indicate if the current_price is less than the next days price

```
USE tpch.tiny;

SELECT
    customer_name,
    order_date,
    total_price,
    CASE
        WHEN total_price > LAG(total_price) over(
            PARTITION by customer_name
            ORDER BY
                order_date
        ) THEN TRUE
        ELSE FALSE
    END AS has_total_price_increased,
    CASE
        WHEN total_price < LEAD(total_price) over(
            PARTITION by customer_name
            ORDER BY
                order_date
        ) THEN TRUE
        ELSE FALSE
    END AS will_total_price_increase
FROM
    (
        SELECT
            c.name AS customer_name,
```

```
            o.orderdate AS order_date,
            sum(o.totalprice) AS total_price
        FROM
            orders o
            JOIN customer c ON o.custkey = c.custkey
        GROUP BY
            1,
            2
    )
ORDER BY
    customer_name,
    order_date
LIMIT
    50;
```

5.6 Choose rows to apply functions to within a window frame using ROWS, RANGE, and GROUPS

We now know how to define windows based on column(s) values. What if we need to apply a function to a specific set of rows within a window? We can do this by defining a custom frame within our window.

There are three ways to define a custom frame within our window.

5.6.1 Rows

We can use the ROWS clause to indicate the number of rows preceding and following we want to consider when applying our function on the current row.

The ROW definition follows the format `ROWS BETWEEN start_point AND end_point`. 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. The n is inclusive of the nth preceding row. The value of n can be any integer or the term `UNBOUNDED`, which means all the rows before the current row within a window frame.
2. n FOLLOWING: n rows following the current row. The n is inclusive of the nth following row. The value of n can be any integer or the term `UNBOUNDED`, which means all the rows after the current row within a window frame.
3. CURRENT ROW: Indicates the current row(inclusive).

In the example picture shown below, when operating on a row with order_month = `1996-06`, it is considered the `CURRENT ROW`, and the rows before and after it are considered preceding and following it, respectively.

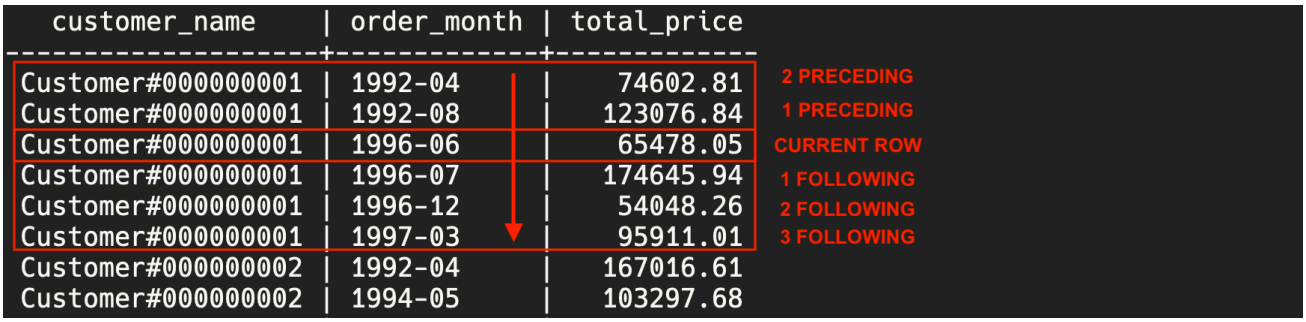


Figure 21: Preceding, Current, & Following

Example:

1. Create a report at customer_name, order_month (YYYY-MM) level/granularity, with the following metrics
    1. total_price: The totalprice spent by a customer on that order_month
    2. three_mo_total_price_avg: The 3 month (previous, current, & next) average of total_price (Round to 2 decimal digits) for that customer

```
USE tpch.tiny;

SELECT
    customer_name,
    order_month,
    total_price,
    ROUND(
        AVG(total_price) OVER(
            PARTITION BY customer_name
            ORDER BY
                order_month ROWS BETWEEN 1 PRECEDING
                AND 1 FOLLOWING
        ),
        2
    ) AS three_mo_total_price_avg
FROM
    (
        SELECT
            c.name AS customer_name,
            DATE_FORMAT(orderdate, '%Y-%m') AS order_month,
            sum(o.totalprice) AS total_price
        FROM
            orders o
```

```
            JOIN customer c ON o.custkey = c.custkey
        GROUP BY
            1,
            2
    )
ORDER BY
    customer_name,
    order_month
LIMIT
    50;
```

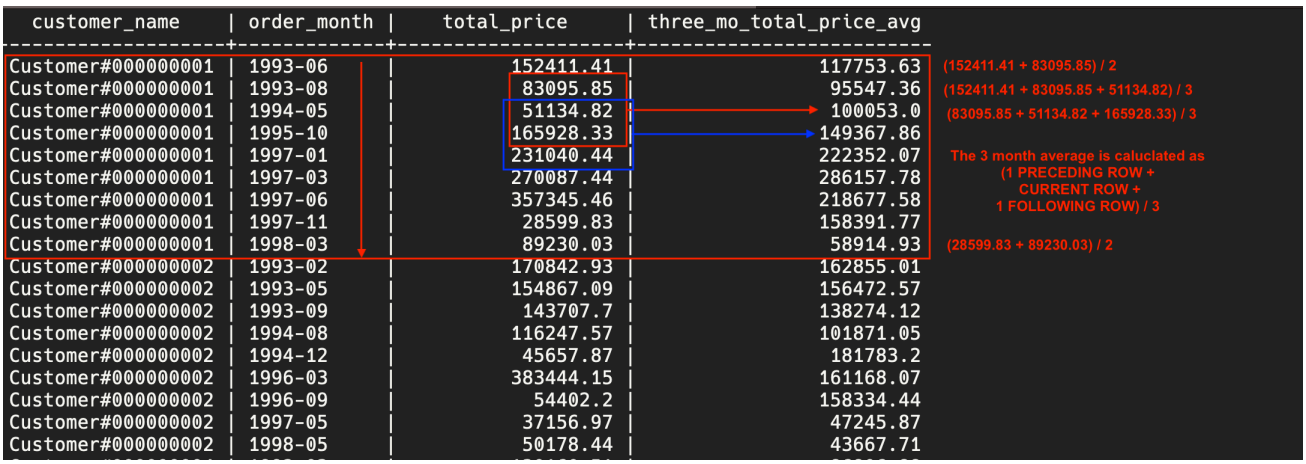


Figure 22: Rows

We use the `ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING` clause to calculate the average of the three months. Note the rows at the window boundaries have only two values to average and do not cross the window frame.

Exercise:

1. Create the example report from above with these additional columns (Round to 2 decimal digits)
    1. running_total_price_avg: The average of total_price for that customer
    2. six_mo_total_price_avg: The 6 month (3 previous, current, & 2 future) average of total_price for that customer
    3. prev_three_mo_total_price_avg: That customer's average of total_price in the past three months. Note that this should not consider the current month but the three months before the current month

```
-- Use the query from the example above
    ROUND(
        AVG(total_price) OVER(
            PARTITION BY customer_name
            ORDER BY
                order_month
        ),
        2
    ) AS running_total_price_avg,
    ROUND(
        AVG(total_price) OVER(
            PARTITION BY customer_name
            ORDER BY
                order_month ROWS BETWEEN 3 PRECEDING
                AND 2 FOLLOWING
        ),
        2
    ) AS six_mo_total_price_avg,
    ROUND(
        AVG(total_price) OVER(
```

```
            PARTITION BY customer_name
            ORDER BY
                order_month ROWS BETWEEN 4 PRECEDING
                AND 1 PRECEDING
        ),
        2
    ) AS prev_three_mo_total_price_avg
```

5.6.2 Range

We can use RANGE to choose rows that fall within a specific range of values of the `ORDER BY` column(s), given that the `ORDER BY` column is of numeric or date or DateTime datatype. Please refer to your DB documentation; not all DB support `RANGE`.

Example:

1. Create a report at customer_name, order_month level/granularity, with the following metrics
    1. total_price: The totalprice spent by a customer on that order_month
    2. avg_3m_all: The three-month (previous, current, & next) average of total_price (Round to 2 decimal digits) for that customer. Note that the three months need not be consecutive.
    3. avg_3m: The three-month (previous, current, & next) average of total_price (Round to 2 decimal digits) for that customer. Note that the three months should be consecutive (not just in order).

```
USE tpch.tiny;

SELECT
    customer_name,
    order_month,
    total_price,
    ROUND(
        AVG(total_price) OVER(
            PARTITION BY customer_name
            ORDER BY
                order_month ROWS BETWEEN 1 PRECEDING
                AND 1 FOLLOWING
        ),
        2
    ) AS avg_3m_all,
    ROUND(
        AVG(total_price) OVER(
            PARTITION BY customer_name
            ORDER BY
                order_month RANGE BETWEEN
                INTERVAL '1' MONTH PRECEDING
                AND INTERVAL '1' MONTH FOLLOWING
        ),
        2
    ) AS avg_3m
FROM
    (
        SELECT
            c.name AS customer_name,
            CAST(
                DATE_FORMAT(orderdate, '%Y-%m-01') AS DATE
```

```
            ) AS order_month,
            sum(o.totalprice) AS total_price
        FROM
            orders o
            JOIN customer c ON o.custkey = c.custkey
        GROUP BY
            1,
            2
    )
ORDER BY
    customer_name,
    order_month
LIMIT
    50;
```

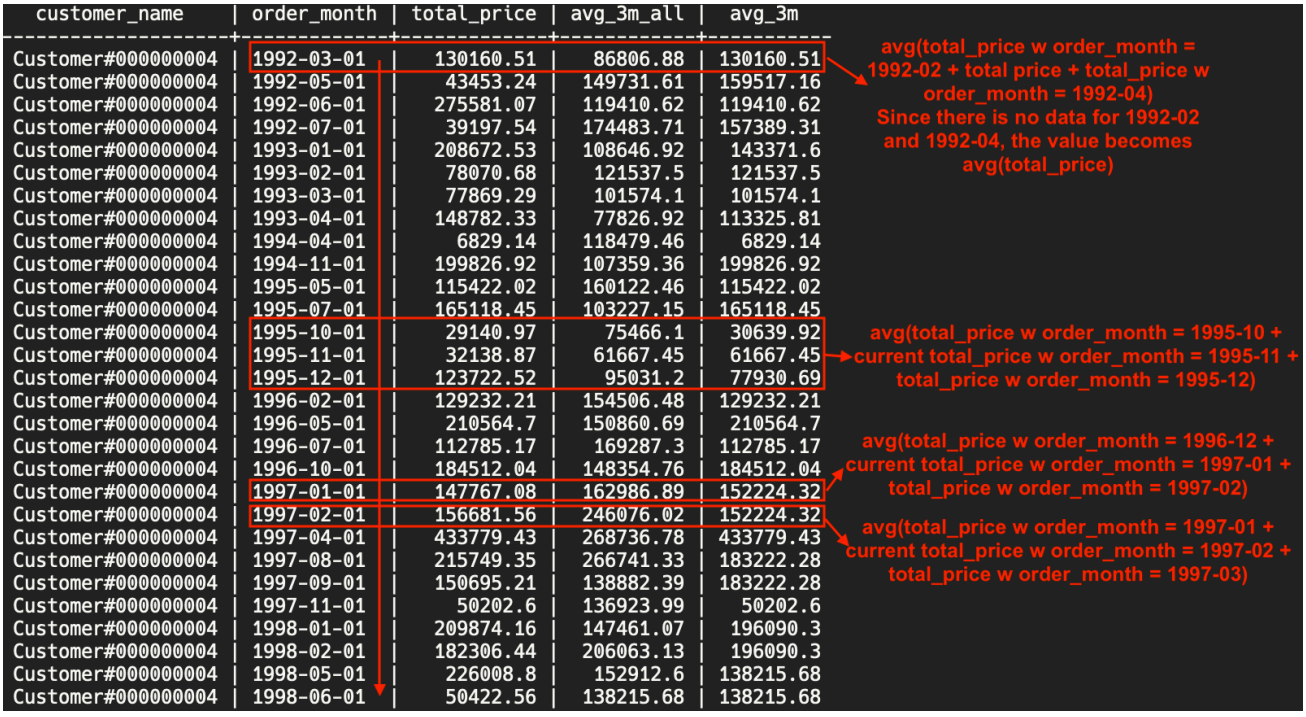


Figure 23: Range

From the above result, we can see how the RANGE clause can be used to define a window based on the actual value of the ORDER BY column.

Exercise:

1. Create a report at supplier_nation (name of item supplier nation) & order_year level/granularity, with the following metrics
    1. total_quantity: The total amount of items supplied by that nation on order_year
    2. avg_total_price_wi_20_quantity: The average of the total amount of items supplied by that nation. The total_quantity of the rows to be considered should lie within +20/-20 of the current total_quantity (round to 2 decimal digits). Divide total_price by 100,000 to make it easy to read.

```
USE tpch.tiny;

SELECT
    supplier_name,
    order_year,
    total_quantity,
    total_price,
    ROUND(
        AVG(total_price) OVER(
            PARTITION BY supplier_name
            ORDER BY
                total_price RANGE BETWEEN 20 PRECEDING
                AND 20 FOLLOWING
        ),
        2
    ) AS avg_total_price_wi_20_quantity
FROM
```

```
    (
        SELECT
            n.name AS supplier_name,
            YEAR(o.orderdate) AS order_year,
            SUM(l.quantity) AS total_quantity,
            ROUND(sum(o.totalprice) / 100000, 2) AS
             ↪  total_price
        FROM
            lineitem l
            JOIN orders o ON l.orderkey = o.orderkey
            JOIN supplier s ON l.suppkey = s.suppkey
            JOIN nation n ON s.nationkey = n.nationkey
        GROUP BY
            n.name,
            YEAR(o.orderdate)
    );
```

5.6.3 Groups

GROUPS allows you to specify rows with the same values in the `ORDER BY` column(s) within a window frame.

Example:

1. Create a report at customer_name, supplier_name, order_month (format YYYY-MM-01) level/granularity, with the following metrics
    1. total_quantity: The total quantity of items from the supplier bought by the customer.

2. max_quantity_over_next_three_months: The maximum quantity of items sold (by this supplier and customer) between this month and the next three months (inclusive).

```
USE tpch.tiny;

SELECT
    customer_name,
    supplier_name,
    order_month,
    total_quantity,
    MAX(total_quantity) OVER(
        PARTITION BY customer_name
        ORDER BY
            order_month GROUPS BETWEEN CURRENT ROW
            AND 3 FOLLOWING
    ) AS max_quantity_over_next_three_months
FROM
    (
        SELECT
            c.name AS customer_name,
            s.name AS supplier_name,
            CAST(DATE_FORMAT(o.orderdate, '%Y-%m-01') AS
             ↪  DATE) AS order_month,
            SUM(l.quantity) AS total_quantity
        FROM
            lineitem l
            JOIN orders o ON l.orderkey = o.orderkey
            JOIN customer c ON o.custkey = c.custkey
            JOIN supplier s ON l.suppkey = s.suppkey
        GROUP BY
```

```
            c.name,
            s.name,
            DATE_FORMAT(o.orderdate, '%Y-%m-01')
        ORDER BY
            1,
            3
    )
LIMIT
    50;
```

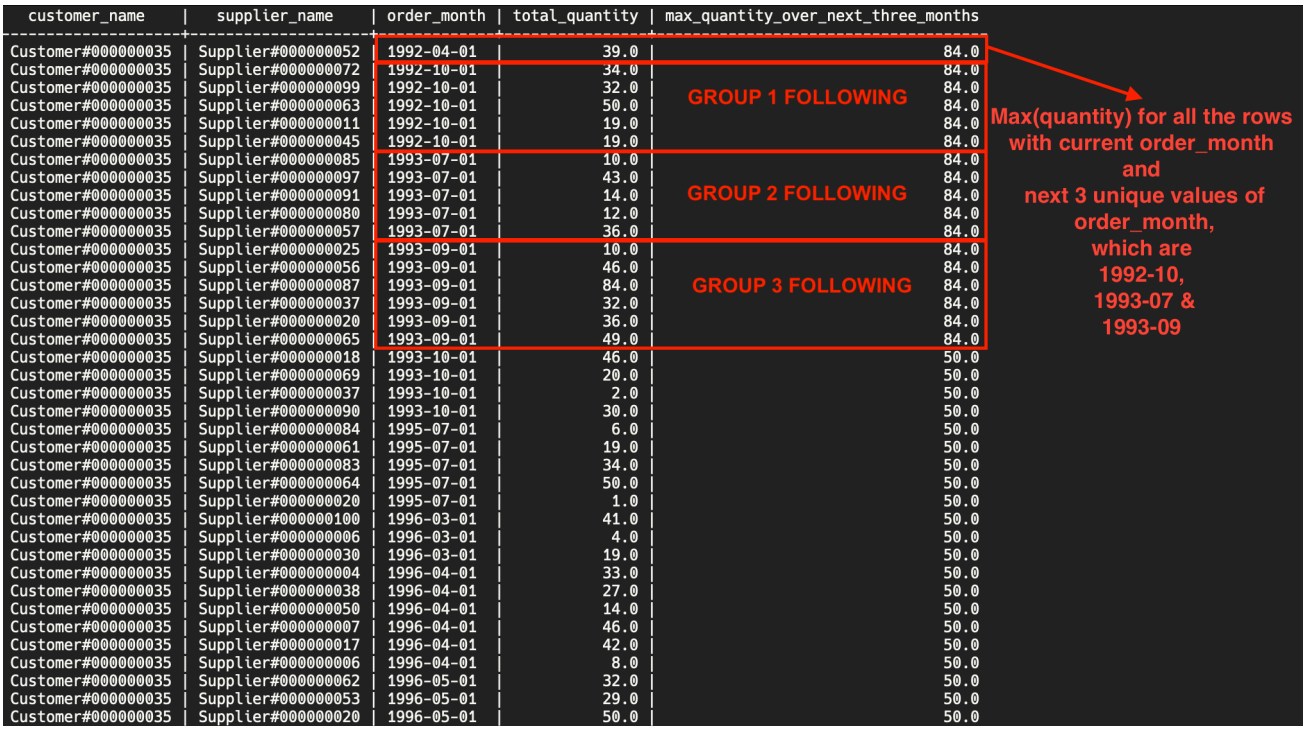


Figure 24: Groups

We can see how the GROUPS clause allows us to identify groups of rows within a window having the same values in a specified column

(order_month in our case).

Exercise:

1. Create a report at customer_nation, supplier_nation, & order_month (Format: YYYY-MM-01) level/granularity, with the following metrics
    1. avg_days_to_deliver: Average days to deliver items; use the formula average of days between ship date & receipt date.
    2. shortest_avg_days_to_deliver_3mo: For every customer_nation, & order_month, get the shortest average days to deliver in the last three months.

```
USE tpch.tiny;

SELECT
    customer_nation,
    supplier_nation,
    order_month,
    avg_days_to_deliver,
    MIN(avg_days_to_deliver) OVER(
        PARTITION BY customer_nation
        ORDER BY
            order_month GROUPS BETWEEN 3 PRECEDING
            AND 1 PRECEDING
    ) shortest_avg_days_to_deliver_3mo
FROM
    (
        SELECT
            cn.name AS customer_nation,
            sn.name AS supplier_nation,
            CAST(
```

```
            DATE_FORMAT(o.orderdate, '%Y-%m-01') AS DATE
        ) AS order_month,
        ROUND(
            AVG(
                DATE_DIFF(
                    'day',
                    l.shipdate,
                    l.receiptdate
                )
            ),
            2
        ) AS avg_days_to_deliver
    FROM
        lineitem l
        JOIN orders o ON l.orderkey = o.orderkey
        JOIN customer c ON o.custkey = c.custkey
        JOIN supplier s ON l.suppkey = s.suppkey
        JOIN nation cn ON c.nationkey = cn.nationkey
        JOIN nation sn ON s.nationkey = sn.nationkey
    GROUP BY
        1,
        2,
        3
);
```

5.7 Use query plan to decide to use window function when performance matters

While window functions are powerful, use them only when needed. If a question can be answered with a `GROUP BY`, then it might be beneficial to check the query plan of both the `Window` & `Group by` approaches.

6 Write easy-to-understand SQL with CTEs and answer common business questions with sample templates

6.1 Use CTEs to write easy-to-understand queries and prevent re-processing of data

Complex SQL queries often involve multiple sub-queries. Having multiple sub-queries makes understanding the query very difficult. You can use a `Common Table Expression (CTE)` to make your queries readable and help with the performance.

Example:

1. Create a report at nation level/granularity, with the following metrics
    1. num_supplied_parts: The number of parts the nation supplies.
    2. num_purchased_parts: The number of parts purchased by the nation.
    3. sold_to_purchase_fraction: The number of parts sold divided by the number of parts purchased.

We can use two sub-queries, one to get the number of parts supplied per nation and the other to get the number of parts purchased per nation. But we can use CTEs to make this query more readable.

```
WITH supplier_nation_metrics AS (
    SELECT
        n.nationkey,
```

```
        SUM(l.QUANTITY) AS num_supplied_parts
    FROM
        lineitem l
        JOIN supplier s ON l.suppkey = s.suppkey
        JOIN nation n ON s.nationkey = n.nationkey
    GROUP BY
        n.nationkey
),
buyer_nation_metrics AS (
    SELECT
        n.nationkey,
        SUM(l.QUANTITY) AS num_purchased_parts
    FROM
        lineitem l
        JOIN orders o ON l.orderkey = o.orderkey
        JOIN customer c ON o.custkey = c.custkey
        JOIN nation n ON c.nationkey = n.nationkey
    GROUP BY
        n.nationkey
)
SELECT
    n.name AS nation_name,
    s.num_supplied_parts,
    b.num_purchased_parts,
    ROUND(CAST(s.num_supplied_parts /b.num_purchased_parts AS
     ↪  DECIMAL(10, 2)), 2 ) * 100 AS sold_to_purchase_perc
FROM
    nation n
    LEFT JOIN supplier_nation_metrics s
      ON n.nationkey = s.nationkey
    LEFT JOIN buyer_nation_metrics b
```

```
    ON n.nationkey = b.nationkey;
```

In the above example, `buyer_nation_metrics` and `supplier_nation_metrics` are the CTEs.

Use CTEs when you have to change the granularity of the table(s) before joining them or when you have to create a different business entity (& its metrics) before joining. The above example shows that the granularity of `buyer_nation_metrics` and `supplier_nation_metrics` CTEs are at the national level before joining the `nation` table. CTEs are defined using the keyword `WITH`, as shown above.

With more complex queries, you will be reusing CTEs at multiple places, which makes the SQL code cleaner by following the DRY (don't-repeat-yourself) rule.

Exercise:

1. Create a report at brand level/granularity, with the following metrics
    1. cust_total_spend: The total amount customers spent buying products of this brand. Note: Spend is calculated as `extendedprice * ( 1 - discount) * (1 + tax)`.
    2. cust_total_spend_wo_tax_discounts: The total amount customers spent buying products of this brand without tax or discounts. Note: Spend without tax or discount is just the `extendedprice`.
    3. supplier_total_sold: The total amount supplier made by selling products of this brand. Note: Spend is calculated as `extendedprice * ( 1 - discount) * (1 + tax)`.

4. supplier_total_sold_wo_tax_discounts: The total amount supplier made by selling products of this brand, without tax or discounts. Note: Spend without tax or discount is just the `extendedprice`.
5. num_suppliers: The total number of suppliers that sold products to this specific brand.
6. num_customers: The total number of customers that purchased products belonging to this specific brand.

```
USE tpch.tiny;

WITH supplier_metrics AS (
    SELECT
        p.brand,
        l.suppkey,
        round(
            sum(l.extendedprice * (1 - l.discount) * (1 +
            ↪  l.tax)),
            2
        ) AS supplier_total_sold,
        round(sum(l.extendedprice), 2) AS
        ↪  supplier_total_sold_wo_tax_discounts
    FROM
        lineitem l
        JOIN partsupp ps ON l.partkey = ps.partkey
        AND l.suppkey = ps.suppkey
        JOIN part p ON ps.partkey = p.partkey
    GROUP BY
        p.brand,
        l.suppkey
),
```

```
customer_metrics AS (
    SELECT
        p.brand,
        o.custkey,
        round(
            sum(
                l.extendedprice * (1 - l.discount) * (1 +
 ↪  l.tax)
            ),
            2
        ) AS cust_total_spend,
        round(sum(l.extendedprice), 2) AS
         ↪  cust_total_spend_wo_tax_discounts
    FROM
        lineitem l
        JOIN orders o ON l.orderkey = o.orderkey
        JOIN partsupp ps ON l.partkey = ps.partkey
        AND l.suppkey = ps.suppkey
        JOIN part p ON ps.partkey = p.partkey
        JOIN customer c ON o.custkey = c.custkey
    GROUP BY
        p.brand,
        o.custkey
)
SELECT
    p.brand,
    sum(num_customers) AS num_customers,
    sum(num_suppliers) AS num_suppliers,
    sum(s.supplier_total_sold) AS supplier_total_sold,
    sum(s.supplier_total_sold_wo_tax_discounts) AS
     ↪  supplier_total_sold_wo_tax_discounts,
```

```
    sum(c.cust_total_spend) AS cust_total_spend,
    sum(c.cust_total_spend_wo_tax_discounts) AS
     ↪  cust_total_spend_wo_tax_discounts
FROM
    part p
    JOIN (
        SELECT
            brand,
            count(suppkey) AS num_suppliers,
            SUM(supplier_total_sold) AS supplier_total_sold,
            SUM(supplier_total_sold_wo_tax_discounts) AS
             ↪  supplier_total_sold_wo_tax_discounts
        FROM
            supplier_metrics
        GROUP BY
            brand
    ) s ON p.brand = s.brand
    JOIN (
        SELECT
            brand,
            count(custkey) AS num_customers,
            SUM(cust_total_spend) AS cust_total_spend,
            SUM(cust_total_spend_wo_tax_discounts) AS
             ↪  cust_total_spend_wo_tax_discounts
        FROM
            customer_metrics
        GROUP BY
            brand
    ) c ON p.brand = c.brand
GROUP BY
    p.brand;
```

Hint: The above query can be significantly simplified; look at the sold and spend numbers in the output; what can you deduce from them?

When replicating a loop construct in SQL, we use recursive CTEs. Defining a recursive CTE is done using the keywords `WITH RECURSIVE`; with the name of the CTE, one should also include the column names in the recursive CTE definition.

For example, if we want to generate a list of months, we can use a recursive CTE, as shown below.

```
USE tpch.tiny;

SET SESSION max_recursion_depth = 12;

WITH RECURSIVE months_2022(mnth) AS (
    SELECT cast('2022-01-01' AS DATE) -- called the
     ↪  base/anchor

    UNION ALL

    SELECT DATE_ADD('month', 1, mnth) -- called the step
    FROM months_2022
    WHERE mnth < cast('2022-12-01' AS DATE)
)
SELECT * FROM months_2022
ORDER BY mnth;
```

The pseudo-code for the above query can be written as

```
1. set base = '2022-01-01'
2. Add base to output(list)
3. Assign step = base (step will be set to '2022-01-01')
4. If step < '2022-12-01', Goto step 5; else step 8
5. step = current step + one month
6. Add the step value to the output
7. Goto step 4
8. Return output
```

The above is a simple example (which can be replicated easily with the `select * from unnest(sequence(cast('2022-01-01' AS DATE), cast('2022-12-01' AS DATE), INTERVAL '1' MONTH));`). Use recursive CTEs to determine the path along a graph. For example, if we have a company reporting structure, as shown below, and we have to figure out the reporting chain, we can use recursive CTEs.

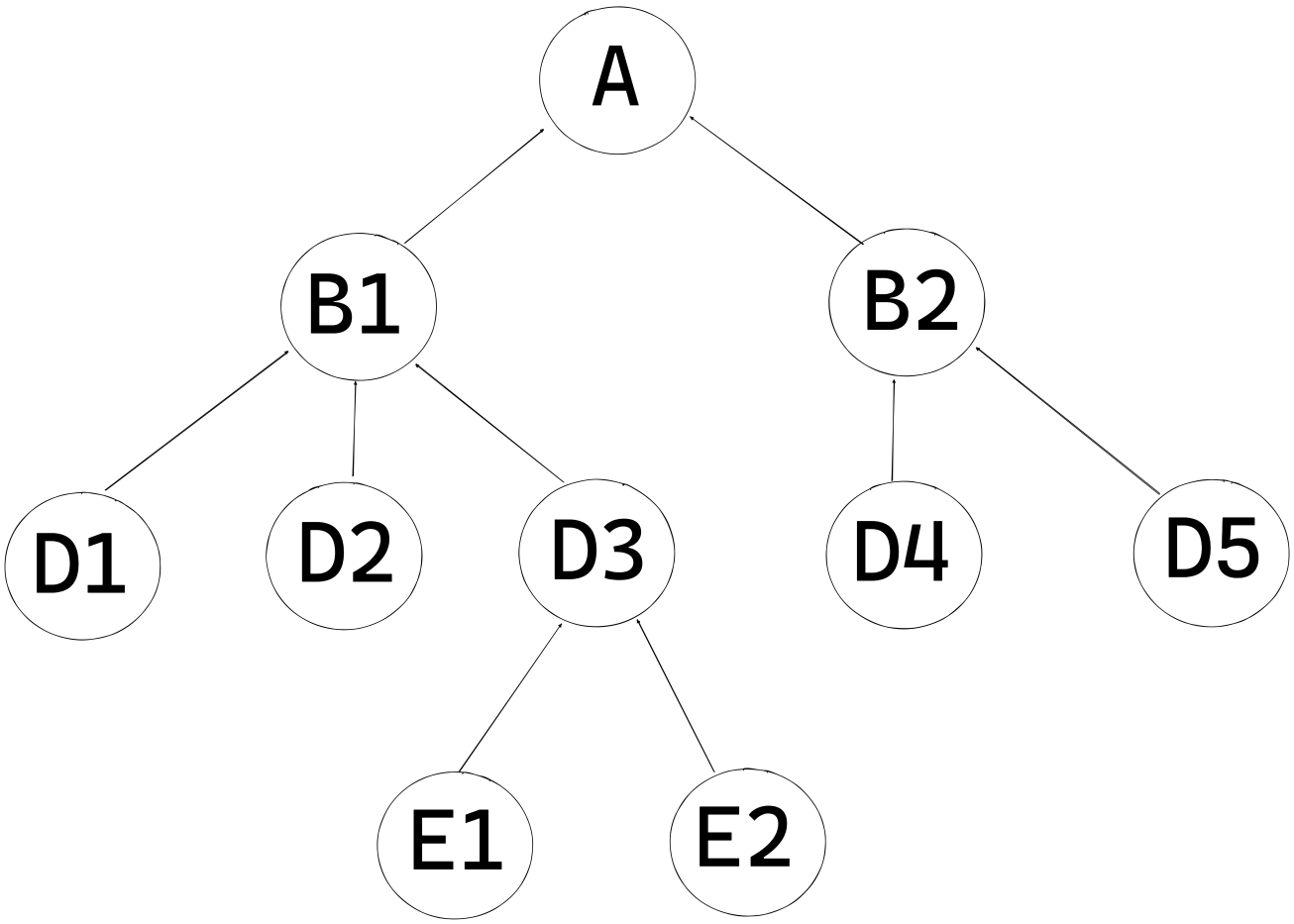


Figure 25: CTE chain

```
CREATE SCHEMA IF NOT EXISTS
minio.warehouse WITH (location = 's3a://warehouse/');

USE minio.warehouse;

DROP TABLE IF EXISTS employee_info;

CREATE TABLE IF NOT EXISTS employee_info(id int, name
 ↪  varchar, reports_to int);
```

```
INSERT INTO
    employee_info
VALUES
    (1, 'A', NULL),
    (2, 'B1', 1),
    (3, 'B3', 1),
    (4, 'D1', 2),
    (5, 'D2', 2),
    (6, 'D3', 2),
    (7, 'D4', 3),
    (8, 'D5', 3),
    (9, 'E1', 6),
    (10, 'E2', 6);

WITH RECURSIVE manager_chain(id, emp, path_to_top) AS (
    SELECT
        id,
        ei.name,
        Array [ei.name] AS path_to_top -- called the
         ↪  base/anchor
    FROM
        employee_info ei
    WHERE
        reports_to IS NULL

    UNION ALL

    SELECT
        ei.id,
        ei.name,
        Array [ei.name] || mc.path_to_top AS path_to_top --
         ↪  called the step
```

```
    FROM
        employee_info ei
        JOIN manager_chain mc ON ei.reports_to = mc.id
        -- No results from join = terminate recursive CTE
)
SELECT
    *
FROM
    manager_chain;
```

The pseudo-code for the above query can be written as

```
1. set base = "A" who does not report to anyone (WHERE
 ↪  reports_to IS NULL)
2. Add base to manager_chain
3. Assign step = base (step will be set to "A"'s record)
4. If there are people who report to the step employee(s)
 ↪  (any result from join), Goto step 5; else, step 8
5. Set step = all the people who report to the current step's
 ↪  employee(s)
6. Add the step value to the manager_chain
7. Goto step 4
8. Return manager_chain
```

We can see how the above query gives the reporting chain from the employee to the company's leader.

6.2 Templates for Deduping, Pivots, Period-over-period (DoD, MoM, YoY) calculations, and GROUPing BY multiple column combinations in one query

There are specific patterns of data requests that will show up in most industries. Let's look at a few of them.

6.2.1 Deduping (aka remove duplicate rows)

When we have a table with duplicate rows or rows with the same data inserted at different dates, we usually have to dedupe them before use.

We can use the `ROW_NUMBER` window function to do this. Let's consider an example where we have duplicate data (duplicated rows) in our `orders` table, and we have to get the unique rows. While we can do a group by all the columns, it will need to be more efficient.

```
USE tpch.tiny;

WITH duplicated_orders AS (
    SELECT
        *
    FROM
        orders
    UNION
    SELECT
        *
    FROM
        orders
),
```

```
ranked_orders AS (
    SELECT
        *,
        row_number() over(PARTITION by orderkey) AS rn
    FROM
        orders
)
SELECT
    COUNT(*)
FROM
    ranked_orders
WHERE
    rn = 1;
```

In the above query, we can see how we use the orders table's primary key (`orderkey`) to dedupe the rows.

We can also use `ROW_NUMBER` in cases where we have multiple similar events that happen one after the other, and we want to pick the latest or the earliest event (using event occurrence DateTime in the ORDER BY clause).

Exercise:

1. For every customer, show their last order placed for each month, with the following columns ordermonth, orderkey, custkey, and totalprice.

```
USE tpch.tiny;

WITH ranked_monthly_orders AS (
```

```
    SELECT
        date_format(orderdate, '%Y-%m') AS ordermonth,
        orderkey,
        custkey,
        totalprice,
        row_number() over(
            PARTITION by date_format(orderdate, '%Y-%m'),
            custkey
            ORDER BY
                orderdate DESC
        ) AS rn
    FROM
        orders
)
SELECT
    ordermonth,
    orderkey,
    custkey,
    totalprice
FROM
    ranked_monthly_orders
WHERE
    rn = 1
ORDER BY
    custkey,
    ordermonth;
```

2. Create the same report above, but show the first and last order of a month.

6.2.2 Pivots

A typical reporting pattern converts the report's dimension values into individual columns, with values being the measured metrics. Let's consider an example.

1. Create a report at ordermonth level, which has the following columns (all rounded to 2 decimal places)
    1. urgent_order_avg_price: Average total price for orders with `1-URGENT` as orderpriority.
    2. high_order_avg_price: Average total price for orders with `2-HIGH` as orderpriority.
    3. medium_order_avg_price: Average total price for orders with `3-MEDIUM` as orderpriority.
    4. not_specified_order_avg_price: Average total price for orders with `4-NOT SPECIFIED` as orderpriority.
    5. low_order_avg_price: Average total price for orders with `5-LOW` as orderpriority.

```
USE tpch.tiny;

SELECT
    date_format(orderdate, '%Y-%m') AS ordermonth,
    ROUND(
        AVG(
            CASE
                WHEN orderpriority = '1-URGENT'
                THEN totalprice
                ELSE NULL
            END
        ),
```

```
        2
    ) AS urgent_order_avg_price,
    ROUND(
        AVG(
            CASE
                WHEN orderpriority = '2-HIGH'
                THEN totalprice
                ELSE NULL
            END
        ),
        2
    ) AS high_order_avg_price,
    ROUND(
        AVG(
            CASE
                WHEN orderpriority = '3-MEDIUM'
                THEN totalprice
                ELSE NULL
            END
        ),
        2
    ) AS medium_order_avg_price,
    ROUND(
        AVG(
            CASE
                WHEN orderpriority = '4-NOT SPECIFIED'
                THEN totalprice
                ELSE NULL
            END
        ),
        2
```

```
    ) AS not_specified_order_avg_price,
    ROUND(
        AVG(
            CASE
                WHEN orderpriority = '5-LOW'
                THEN totalprice
                ELSE NULL
            END
        ),
        2
    ) AS low_order_avg_price
FROM
    orders
GROUP BY
    date_format(orderdate, '%Y-%m');
```

This pattern of a `case statement within an aggregate` is proper when calculating an aggregate conditional on one or more column(s) values.

6.2.3 Period-over-period metrics calculation

One of the critical metrics companies use is period-over-period growth, typically calculated as month-over-month or year-over-year growth. We can use window-based value functions to get the previous period's metric.

Example:

1. Create a report at ordermonth (format YYYY-MM) level with the following fields

1. totalprice: The sum of totalprice (in 100,000 multiples) of all the orders in a month.
2. MoM_totalprice_change: The percentage change of totalprice compared to the previous month. Use this formula ((current - previous) / previous) × 100,

```
USE tpch.tiny;

WITH monthly_orders AS (
    SELECT
        date_format(orderdate, '%Y-%m') AS ordermonth,
        ROUND(SUM(totalprice) / 100000, 2) AS totalprice
    FROM
        orders
    GROUP BY
        date_format(orderdate, '%Y-%m')
)
SELECT
    ordermonth,
    totalprice,
    ROUND(
        (
            totalprice - lag(totalprice) over(
                ORDER BY
                    ordermonth
            )
        ) * 100 / (
            lag(totalprice) over(
                ORDER BY
                    ordermonth
            )
```

```
        ),
        2
    ) AS MoM_totalprice_change
FROM
    monthly_orders
ORDER BY
    ordermonth;
```

Exercise:

1. Create a report at ordermonth & customer_nation level with the following fields
    1. totalprice: The sum of totalprice (divided by 100,000 for readability) of all the order's in a month by a customer nation.
    2. MoM_totalprice_change: The percentage change of totalprice compared to the previous month for the customer nation. Use this formula ((current - previous) / previous) × 100.

```
USE tpch.tiny;

WITH monthly_orders AS (
    SELECT
        DATE(date_format(o.orderdate, '%Y-%m-01')) AS
        ↪  ordermonth,
        n.name AS customer_nation,
        ROUND(SUM(o.totalprice) / 100000, 2) AS totalprice
    FROM
        orders o
        JOIN customer c ON o.custkey = c.custkey
        JOIN nation n ON c.nationkey = c.nationkey
```

```
    GROUP BY
        date_format(o.orderdate, '%Y-%m-01'),
        n.name
)
SELECT
    ordermonth,
    customer_nation,
    totalprice,
    ROUND(
        (
            totalprice - lag(totalprice) over(
                PARTITION BY customer_nation
                ORDER BY
                    ordermonth
            )
        ) * 100 / (
            lag(totalprice) over(
                PARTITION BY customer_nation
                ORDER BY
                    ordermonth
            )
        ),
        2
    ) AS MoM_totalprice_change
FROM
    monthly_orders
ORDER BY
    customer_nation,
    ordermonth;
```

6.2.4 GROUPing BY multiple column combinations in one query

Sometimes, you will be required to get the results of grouping by different column combinations (s) into one dataset. When trying to get results of the different group-by-column combinations into one dataset, you can try doing a UNION ALL on multiple groups by queries, or you can use the `GROUPING SETS` command.

Example:

1. Create a report which has the sum of totalprice spent at
    1. month (from orderdate) level
    2. customer nation level
    3. month (from orderdate) and customer nation level

```
USE tpch.tiny;

WITH monthly_cust_nation_orders AS (
    SELECT
        date_format(o.orderdate, '%Y-%m') AS ordermonth,
        n.name AS customer_nation,
        totalprice
    FROM
        orders o
        JOIN customer c ON o.custkey = c.custkey
        JOIN nation n ON c.nationkey = c.nationkey
)
SELECT
    ordermonth,
    customer_nation,
    ROUND(SUM(totalprice) / 100000, 2) AS totalprice --
     ↪  divide by 100,000 for readability
```

```
FROM
    monthly_cust_nation_orders
GROUP BY
    GROUPING SETS (
        (ordermonth),
        (customer_nation),
        (ordermonth, customer_nation)
    );
```

7 Appendix: SQL Basics

In this chapter, we will go over SQL basics.

7.1 The hierarchy of data organization is a database, schema, table, and columns

Typically database servers (Trino, MySQL, HIVE, etc.) can have multiple databases; each database can have multiple schemas. Each schema can have multiple tables, and each table can have multiple columns.

Note: We use Trino, which has `catalogs` (in place of databases) that allow it to connect with the different underlying systems. (e.g., Postgres, Redis, Hive, etc.). We can consider the catalog as Trino's database equivalent.

In our lab, we use Trino, and we can check the available catalogs, their schemas, the tables in a schema, & the columns in a table, as shown below. Start the Trino cli using `make trino`(and exit with `exit`).

```
SHOW catalogs;

SHOW schemas IN tpch;

SHOW TABLES IN tpch.tiny;

DESCRIBE tpch.tiny.lineitem;
```

Note how, when referencing the table name, we use the full path, i.e., `database.schema.table_name`. We can skip using the full path of the table if we let Trino know which schema to use by default, as shown below.

```
USE tpch.tiny;

DESCRIBE lineitem;
```

7.2 Use SELECT...FROM, LIMIT, WHERE, & ORDER BY to read the required data from tables

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

```
USE tpch.tiny;

-- use * to specify all columns
SELECT * FROM orders;

-- use column names only to read data from those columns
SELECT orderkey, totalprice FROM orders;
```

However, running a `SELECT ... FROM` statement can cause issues when the data set is extensive. If you want to look at the data, use `LIMIT n` to tell Trino only to get n number of rows.

```
USE tpch.tiny;

SELECT orderkey, totalprice FROM orders LIMIT 5;
```

We can use the `WHERE` clause if we want to get the 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.

```
USE tpch.tiny;

-- all customer rows that have nationkey = 20
SELECT * FROM customer WHERE nationkey = 20 LIMIT 10;

-- all customer rows that have nationkey = 20 and acctbal >
 ↪  1000
SELECT * FROM customer
WHERE nationkey = 20 AND acctbal > 1000 LIMIT 10;

-- all customer rows that have nationkey = 20 or acctbal >
 ↪  1000
SELECT * FROM customer
WHERE nationkey = 20 OR acctbal > 1000 LIMIT 10;

-- all customer rows that have (nationkey = 20 and acctbal >
 ↪  1000) or rows that have nationkey = 11
SELECT * FROM customer
WHERE (nationkey = 20 AND acctbal > 1000)
OR 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.

```
USE tpch.tiny;

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

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

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

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

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

```
USE tpch.tiny;

-- all customer rows which have nationkey = 10 or nationkey =
 ↪  20
SELECT * FROM customer WHERE nationkey IN (10,20);
```

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

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

```
USE tpch.tiny;

SELECT COUNT(*) FROM customer; -- 1500
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.

```
USE tpch.tiny;

-- Will show the first ten customer records with the lowest
 ↪  custkey
-- rows are ordered in ASC order by default
SELECT * FROM orders ORDER BY custkey LIMIT 10;

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

7.3 Combine data from multiple tables using JOINs (there are different types of JOINs)

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

```
-- not based on real tables
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 established 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 the joins for performance).

```
-- not based on real tables
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 table_c
    ON a.c_id = c.id
```

There are five main types of joins, they are:

7.3.1 1. Inner join (default): Get only rows in both tables

```
USE tpch.tiny;

SELECT
    o.orderkey,
    l.orderkey
FROM
    orders o
    JOIN lineitem l ON o.orderkey = l.orderkey
    AND o.orderdate BETWEEN l.shipdate - INTERVAL '5' DAY
    AND l.shipdate + INTERVAL '5' DAY
LIMIT
    100;

SELECT
    COUNT(o.orderkey) AS order_rows_count,
    COUNT(l.orderkey) AS lineitem_rows_count
FROM
    orders o
    JOIN lineitem l ON o.orderkey = l.orderkey
    AND o.orderdate BETWEEN l.shipdate - INTERVAL '5' DAY
    AND l.shipdate + INTERVAL '5' DAY;
-- 2477, 2477
```

Note: JOIN defaults to INNER JOIN.

The output will have rows from orders and lineitem that found at least one matching row from the other table with the specified join condition (same orderkey and orderdate within ship date +/- 5 days).

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

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

```
USE tpch.tiny;

SELECT
    o.orderkey,
    l.orderkey
FROM
    orders o
    LEFT JOIN lineitem l ON o.orderkey = l.orderkey
    AND o.orderdate BETWEEN l.shipdate - INTERVAL '5' DAY
    AND l.shipdate + INTERVAL '5' DAY
LIMIT
    100;

SELECT
    COUNT(o.orderkey) AS order_rows_count,
    COUNT(l.orderkey) AS lineitem_rows_count
FROM
    orders o
    LEFT JOIN lineitem l ON o.orderkey = l.orderkey
    AND o.orderdate BETWEEN l.shipdate - INTERVAL '5' DAY
```

```
    AND l.shipdate + INTERVAL '5' DAY;
-- 15197, 2477
```

The output will have all the 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 ship date +/- 5 days).

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 lineitems.

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

```
USE tpch.tiny;

SELECT
    o.orderkey,
    l.orderkey
FROM
    orders o
    RIGHT JOIN lineitem l ON o.orderkey = l.orderkey
    AND o.orderdate BETWEEN l.shipdate - INTERVAL '5' DAY
    AND l.shipdate + INTERVAL '5' DAY
LIMIT
    100;
```

```
SELECT
    COUNT(o.orderkey) AS order_rows_count,
    COUNT(l.orderkey) AS lineitem_rows_count
FROM
    orders o
    RIGHT JOIN lineitem l ON o.orderkey = l.orderkey
    AND o.orderdate BETWEEN l.shipdate - INTERVAL '5' DAY
    AND l.shipdate + INTERVAL '5' DAY;
-- 2477, 60175
```

The output will have the rows from orders that found at least one matching row from the lineitem table with the specified join condition (same orderkey and orderdate within ship date +/- 5 days) and all the rows from the lineitem table.

We can also see that the number of rows from the orders table is 2477 & from the lineitem table is 60,175.

7.3.4 4. Full outer join: Get all rows from both the left and right tables.

```
USE tpch.tiny;

SELECT
    o.orderkey,
    l.orderkey
FROM
    orders o
    FULL OUTER JOIN lineitem l ON o.orderkey = l.orderkey
    AND o.orderdate BETWEEN l.shipdate - INTERVAL '5' DAY
```

```
    AND l.shipdate + INTERVAL '5' DAY
LIMIT
    100;

SELECT
    COUNT(o.orderkey) AS order_rows_count,
    COUNT(l.orderkey) AS lineitem_rows_count
FROM
    orders o
    FULL OUTER JOIN lineitem l ON o.orderkey = l.orderkey
    AND o.orderdate BETWEEN l.shipdate - INTERVAL '5' DAY
    AND l.shipdate + INTERVAL '5' DAY;
-- 15197, 60175
```

The output will have all the rows from orders that found at least one matching row from the lineitem table with the specified join condition (same orderkey and orderdate within ship date +/- 5 days) and all the 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 60,175.

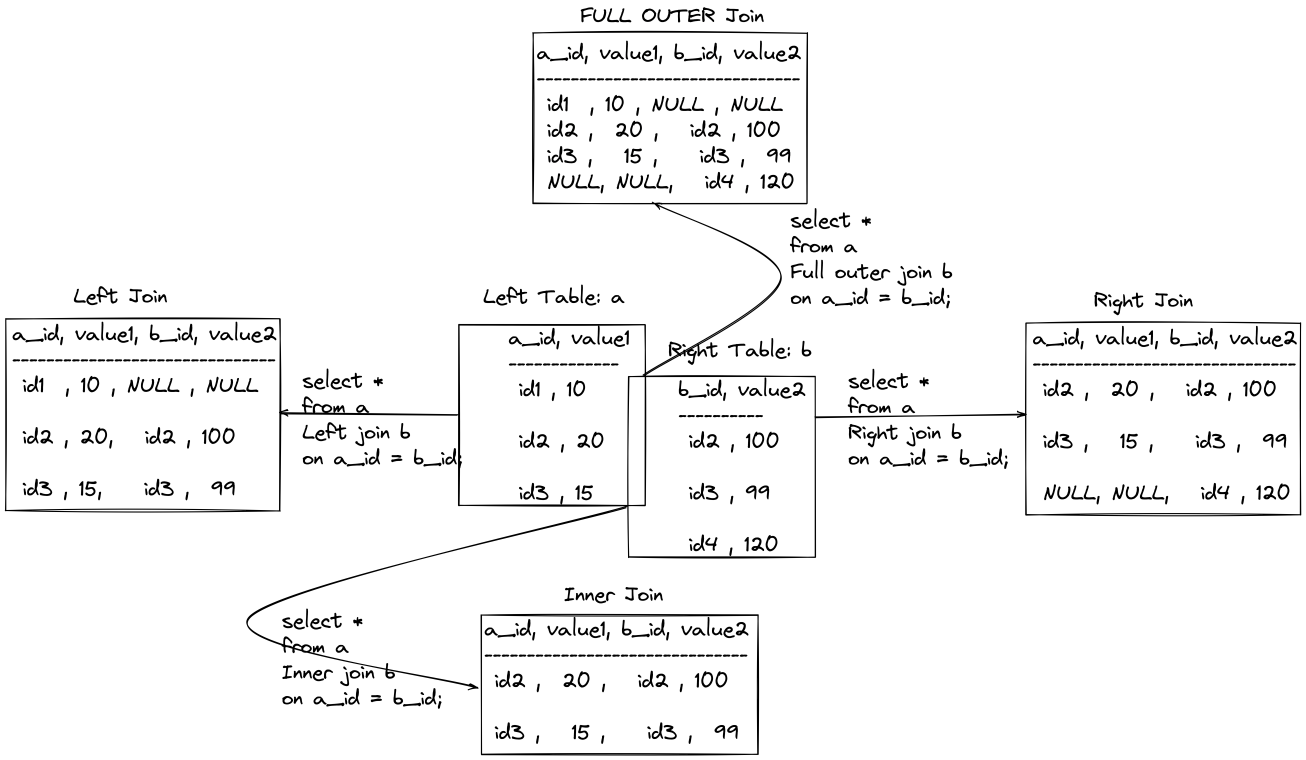
7.3.5 5. Cross join: Get the cartesian product of all rows

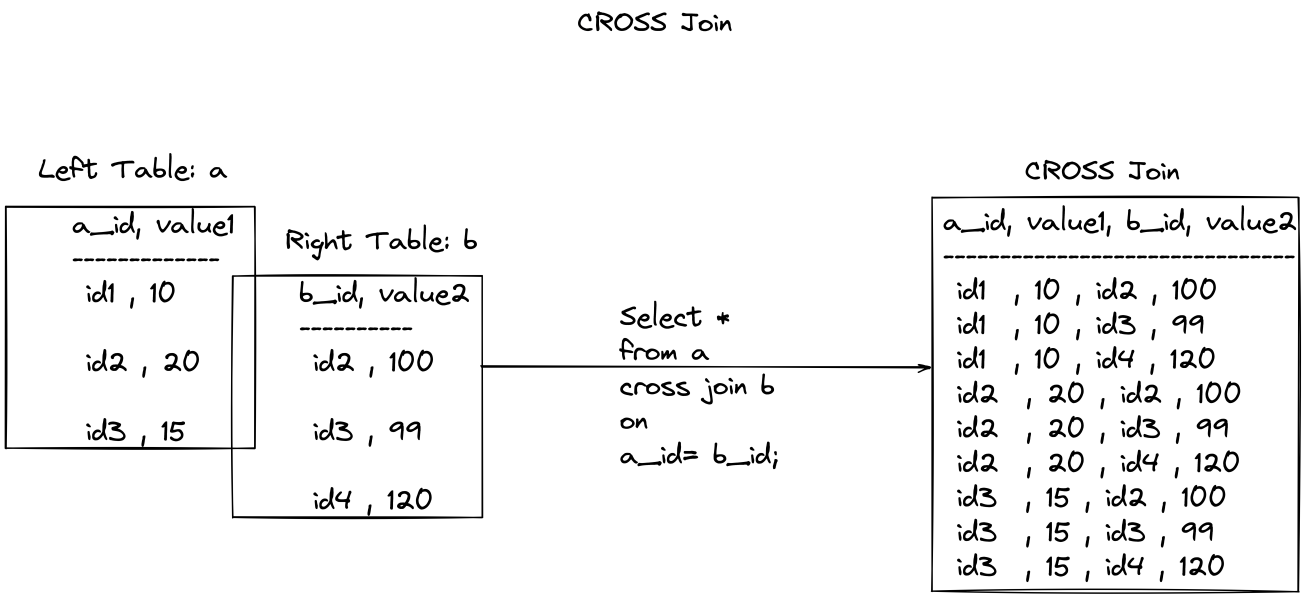
```
USE tpch.tiny;

SELECT
    n.name AS nation_name,
    r.name AS region_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 will need to join a table with itself, called a SELF-join.

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.

```
USE tpch.tiny;

SELECT
    o1.custkey
FROM
    orders o1
    JOIN orders o2 ON o1.custkey = o2.custkey
    AND year(o1.orderdate) = year(o2.orderdate)
    AND week(o1.orderdate) = week(o2.orderdate)
WHERE
    o1.orderkey != o2.orderkey;
```

7.4 Generate metrics for your dimension(s) 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 grouped together based on values of specified column(s).

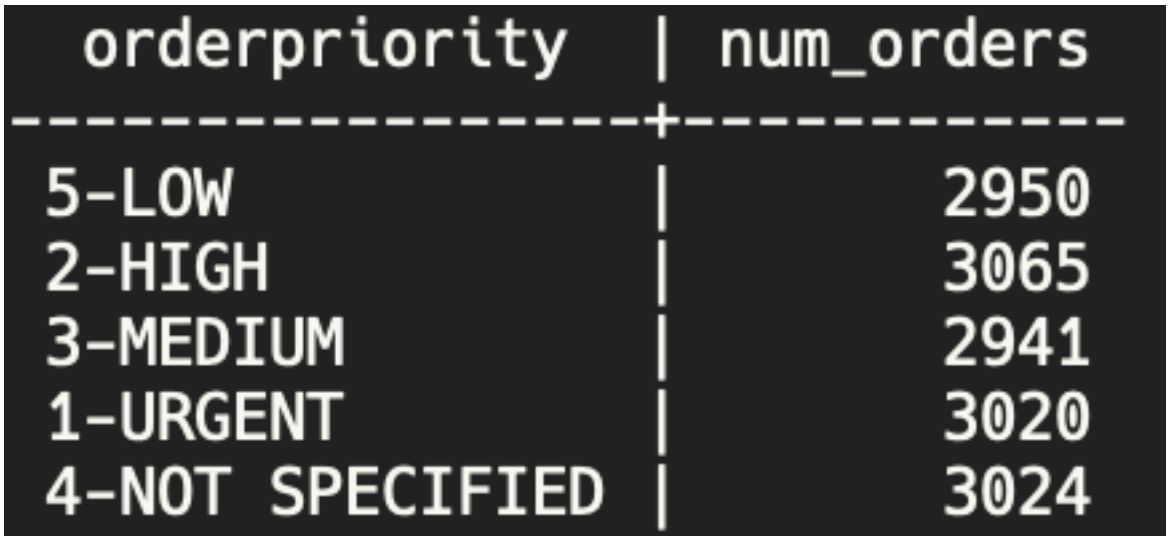
Example:

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

```
USE tpch.tiny;

SELECT
    orderpriority,
    count(*) AS num_orders
FROM
    orders
GROUP BY
    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 output will consist of one row per unique value of `orderpriority` and the `count(*)` calculation.



orderpriority	num_orders
5-LOW	2950
2-HIGH	3065
3-MEDIUM	2941
1-URGENT	3020
4-NOT SPECIFIED	3024

Figure 26: Group by

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

7.5 Use the result of a query within a query using sub-queries

When we want to use the result of a query as a table in another query, we use subqueries. 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).

```
USE tpch.tiny;

SELECT
    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,
            sum(l.quantity) AS quantity
        FROM
            lineitem l
            JOIN supplier s ON l.suppkey = s.suppkey
            JOIN nation n ON s.nationkey = n.nationkey
        GROUP BY
            n.nationkey
    ) s ON n.nationkey = s.nationkey
    LEFT JOIN (
        SELECT
            n.nationkey,
            sum(l.quantity) AS quantity
        FROM
            lineitem l
            JOIN orders o ON l.orderkey = o.orderkey
            JOIN customer c ON o.custkey = c.custkey
            JOIN nation n ON c.nationkey = n.nationkey
        GROUP BY
            n.nationkey
    ) c ON n.nationkey = c.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.

7.6 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 integers, such as tinyint, int, and bigint, allow the 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 variable character string, with a permitted max length n.

2. Char(n): Data type allows storage of 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 (JSON, ARRAY): 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.

Functions such as `DATE_DIFF` and `ROUND` are specific to a data type. 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.

```
USE tpch.tiny;

SELECT
    DATE_DIFF('day', '2022-10-01', '2022-10-05'); -- will
↪   fail due to in correct data type

SELECT
    DATE_DIFF(
        'day',
        CAST('2022-10-01' AS DATE),
        CAST('2022-10-05' AS DATE)
    );
```

A `NULL` indicates the absence of value. 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 an example as shown below. We can see how when `l.orderkey` is NULL, the DB uses `999999` as the output.

```
USE tpch.tiny;

SELECT
    o.orderkey,
    o.orderdate,
    COALESCE(l.orderkey, 9999999) AS lineitem_orderkey,
    l.shipdate
FROM
    orders o
    LEFT JOIN lineitem l ON o.orderkey = l.orderkey
    AND o.orderdate BETWEEN l.shipdate - INTERVAL '5' DAY
    AND l.shipdate + INTERVAL '5' DAY
LIMIT
    100;
```

7.7 Replicate IF.ELSE logic with CASE statements

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

```
USE tpch.tiny;

SELECT
    orderkey,
    totalprice,
    CASE
        WHEN totalprice > 100000 THEN 'high'
```

```
        WHEN 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').

7.8 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 UNION or UNION ALL. `UNION` removes duplicate rows, and `UNION ALL` does not remove duplicate rows. Let's look at an example.

```
USE tpch.tiny;

SELECT custkey, name FROM customer WHERE name LIKE '%_91%';
 ↪  -- 25 rows

-- UNION will remove duplicate rows; the below query will
 ↪  produce 25 rows
SELECT custkey, name FROM customer WHERE name LIKE '%_91%'
UNION
SELECT custkey, name FROM customer WHERE name LIKE '%_91%'
```

```
UNION
SELECT custkey, name FROM customer WHERE name LIKE '%_91%';

-- UNION ALL will not remove duplicate rows; the below query
 ↪  will produce 75 rows
SELECT custkey, name FROM customer WHERE name LIKE '%_91%'
UNION ALL
SELECT custkey, name FROM customer WHERE name LIKE '%_91%'
UNION ALL
SELECT custkey, name FROM customer WHERE name LIKE '%_91%';
```

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

```
USE tpch.tiny;

-- EXCEPT will get the rows in the first query result that is
 ↪  not in the second query result, 0 rows
SELECT custkey, name FROM customer WHERE name LIKE '%_91%'
EXCEPT
SELECT custkey, name FROM customer WHERE name LIKE '%_91%';


-- The below query will result in 23 rows; the first query
 ↪  has 25 rows, and the second has two rows
SELECT custkey, name FROM customer WHERE name LIKE '%_91%'
EXCEPT
SELECT custkey, name FROM customer WHERE name LIKE '%191%';
```

7.9 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 DB objects that operate similarly to a table. 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.

```
CREATE SCHEMA IF NOT EXISTS
minio.warehouse
WITH (location = 's3a://warehouse/');

USE minio.warehouse;

CREATE VIEW nation_supplied_purchased_quantity AS
SELECT
    n.name AS nation_name,
    s.quantity AS supplied_items_quantity,
    c.quantity AS purchased_items_quantity
FROM
    tpch.tiny.nation n
    LEFT JOIN (
        SELECT
            n.nationkey,
            sum(l.quantity) AS quantity
        FROM
            tpch.tiny.lineitem l
            JOIN tpch.tiny.supplier s
```

```
            ON l.suppkey = s.suppkey
        JOIN tpch.tiny.nation n
            ON s.nationkey = n.nationkey
    GROUP BY
        n.nationkey
) s ON n.nationkey = s.nationkey
LEFT JOIN (
    SELECT
        n.nationkey,
        sum(l.quantity) AS quantity
    FROM
        tpch.tiny.lineitem l
        JOIN tpch.tiny.orders o
            ON l.orderkey = o.orderkey
        JOIN tpch.tiny.customer c
            ON o.custkey = c.custkey
        JOIN tpch.tiny.nation n
            ON c.nationkey = n.nationkey
    GROUP BY
        n.nationkey
) c ON n.nationkey = c.nationkey;

SELECT
    *
FROM
    nation_supplied_purchased_quantity;
```

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

Views are equivalent to running queries, hence can be slow if there are

complex transformations, and data is recomputed every time someone queries the view.

If we want better performance, we can use a MATERIALIZED VIEW. The OLAP DB will automatically run the query (used to create materialized view) and store the result when changes occur to the source tables.

The pre-computation of materialized views means that when users query them, the performance will be fast, and the data processing is not performed each time there is a query to the view.

```
CREATE SCHEMA IF NOT EXISTS iceberg.warehouse
WITH (location = 's3a://icebergwarehouse/');

USE iceberg.warehouse;

DROP TABLE IF EXISTS sample_table;

CREATE TABLE sample_table
(
    sample_key bigint,
    sample_status varchar
);
INSERT INTO sample_table VALUES (1, 'hello');

DROP MATERIALIZED VIEW IF EXISTS mat_sample_table;

CREATE MATERIALIZED VIEW mat_sample_table AS
SELECT
    sample_key * 100 AS sample_key,
    UPPER(sample_status) AS sample_status
FROM sample_table;
```

```
SELECT * FROM mat_sample_table; -- 1 row

INSERT INTO sample_table VALUES (2, 'world');

SELECT * FROM mat_sample_table; -- 2 rows
```

Note: We use the `iceberg` catalog, since it's one of the few catalogs that support `materialized views` in Trino at the time of writing this book.

7.10 Use these standard inbuilt DB functions for common 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 clear and orderpriority 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 end character indices. E.g., `SELECT clerk, SUBSTR(clerk, 1, 5) FROM orders LIMIT 5;` will get the first five (1 - 5) characters of the clerk column. Note that the indexing starts from 1 in Trino.
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. In Trino, `date_diff` accepts 3 parameters, the outputs unit (day, month, year), the datetime/date values a and b such that the output will be a - b. The `date_add(unit, value, timestamp)` adds the value (in specified units) to the timestamp value.

```
SELECT
date_diff(
    'DAY',
    DATE '2022-10-01',
    DATE '2023-11-05') diff_in_days,
date_diff(
    'MONTH',
    DATE '2022-10-01',
    DATE '2023-11-05') diff_in_months,
date_diff(
    'YEAR',
    DATE '2022-10-01',
    DATE '2023-11-05') diff_in_years,
```

```
    date_add(
        'DAY',
        400,
        DATE '2022-10-01' -- should give 2023-11-05
    );
```

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 '2022-11-05' + INTERVAL '10' DAY;` will show the output `2022-11-15` (try subtraction of dates).

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 non-standard date/time format such as `MM/DD/YYYY`, we will need to specify the input structure; we do this using `date_parse`, E.g., `SELECT date_parse('11-05-2023', '%m-%d-%Y');`.

We can convert a timestamp/date into a string with the required format using `date_format`. E.g., `USE tpch.tiny;SELECT DATE_FORMAT(orderdate, '%Y-%m-01') AS first_month_date FROM orders LIMIT 5;` will map every orderdate to the first of their month.

See this page on how to set the proper date time format.

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, min, 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);`
    2. ABS is used to get the absolute value of a given number. E.g. `SELECT ABS(-100), ABS(100);`
    3. Mathematical operations these are +,-,*,/.
    4. Ceil/Floor is used to get the next higher and most recent lower integers, given a decimal digit. E.g. `SELECT CEIL(100.1), FLOOR(100.1);`

7.11 Create a table, insert data, delete data, and drop the table

We can create a table using the `CREATE TABLE` statement. We need to specify the table name, column name, and data types, as shown below.

```
CREATE SCHEMA IF NOT EXISTS
minio.warehouse
WITH (location = 's3a://warehouse/');

USE minio.warehouse;

DROP TABLE IF EXISTS sample_table2;
CREATE TABLE sample_table2 (
  sample_key bigint,
  sample_status varchar
);

SELECT * FROM sample_table2;
```

```
INSERT INTO sample_table2 VALUES (1, 'hello');

SELECT * FROM sample_table2;
```

Note: We can create a temporary table, which is a table that only exists for the duration of the SQL session (open-exit CLI or closing a connection). Unfortunately, temp tables are not available in Trino as of writing this book (github issue for temp table support in Trino).

Typically there are two main ways of inserting data into a table.

```
USE minio.warehouse;

-- inserting hardcoded values
INSERT INTO sample_table2 VALUES (1, 'hello');

-- inserting values from another table
INSERT INTO sample_table2
SELECT nationkey, name FROM tpch.tiny.nation;
```

We can remove the data from a table using the delete or truncate, as shown below.

```
USE minio.warehouse;

-- deletes all the rows in sample_table2, but table still
 ↪  present
DELETE FROM sample_table2;
```

```
-- drops the table entirely, the table will need to be
 ↪  re-created
DROP TABLE sample_table2;
```