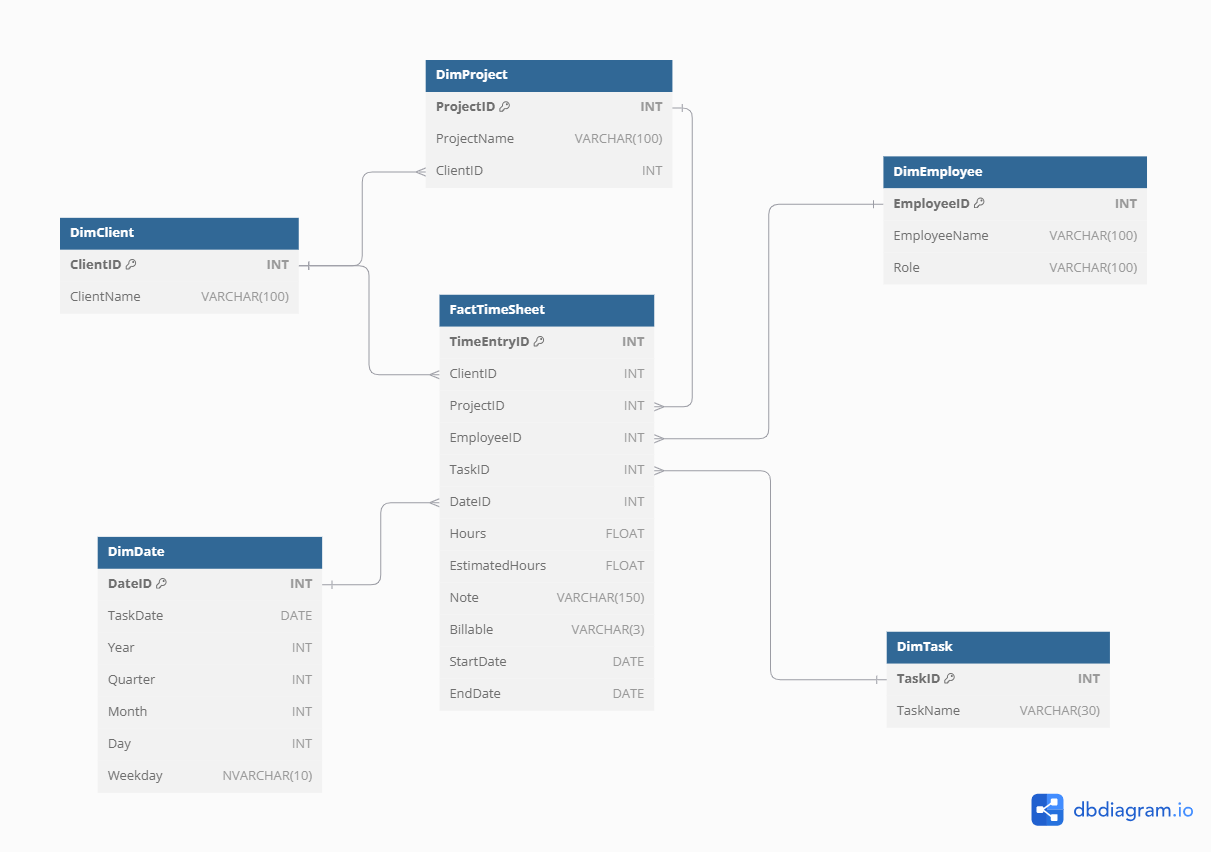
# **Question 1**

## Introduction

The ClicksUp and Float table contains data for hours logged for different tasks, different clients, different date ranges. We also have additional information such as if a task is considered billable which in project management is hours worked by the employee that can be charged to the client. With the data from the two tables, the dimension model can be designed to answer some questions which cut across various metrics in Performance analysis , financial analysis and employee utilization.



Question that cut across this include such as :

* How many hours did each employee work on a specific project?
* What are the total billable hours for each client?
* Total hours worked in each task
* Average hours worked by employees per day
* What is the total billable amount for each project or client?
* How much time is spent on non-billable tasks?
* How is each employee’s time distributed across different projects and tasks?
* Which employees are working the most billable hours?

## What tools did I use to design this solution?

* Azure Synapse dedicated pool - Data Warehouse
* SQL - for Data loading and transformation
* Azure Data Lake Storage(ADLS) - to store the datasets to be used to build the tables
* dbdiagram.io - schema design

I built the data warehouse on the existing files in the data lake by utilizing external tables. The benefit of external tables is that tables can be populated in the data warehouse without the need of moving data from the data lake storage.

## What loading technique was used?

The loading technique would involve loading the data through 3 stages:

**Staging layer** -> **Dimension Tables** -> **Fact table**

The staging layer is populated first, afterwards, dimension tables are built and then fact table(s).

The staging layer acts as a temporary storage area where raw data from various source systems is initially loaded.

## 

## How is the data staged?

A staging layer would be used as a temporary storage area where raw data from the 2 tables would be ingested to. Any errors or discrepancies can be seen early enough in the process without having an adverse effect in the production dimension and fact tables. The dimension and facts tables would be populated from the staging tables.

There would be 3 staging tables, however I would only explain the first two(2), the last one would be discussed when explaining the fact table load:

* **stgFloat** - to store raw float table
* **stgClick** - to store raw ClickUp table

**Notes**

* stg is used to shorten the word staging. Below is the script used to create a populate staging tables via external tables.

--Create sora schema

CREATE SCHEMA sora

--Create Staging click table

CREATE EXTERNAL TABLE [sora].[StgClick] (

Client VARCHAR(32),

Project VARCHAR(85),

Name VARCHAR(60),

Task VARCHAR(30),

TaskDate DATE,

Hours FLOAT,

Note VARCHAR(150),

Billable NVARCHAR(9)

)

WITH (

LOCATION = 'data/clickup.csv', --external table location

DATA\_SOURCE = [SoraData],

FILE\_FORMAT = [SoraDataCSV]

);

-- Create staging float table

CREATE EXTERNAL TABLE [sora].[StgFloat] (

Client VARCHAR(25),

Project VARCHAR(90),

Role VARCHAR(90),

Name VARCHAR(60),

Task VARCHAR(36),

StartDate DATE,

EndDate DATE,

EstimatedHours INT

)

WITH (

LOCATION = 'data/Float-allocations.csv', --external table location

DATA\_SOURCE = [SoraData],

FILE\_FORMAT = [SoraDataCSV]

);

## What would the dimension tables include?

The dimension tables would be used to make sense of the numerical values in the facts. The tables should provide an accurate description. Based on the question I am looking to answer, I thought the following should be used as descriptions. The dimension tables would look to store distinct information segments e.g distinct employees, dates, project, clients etc.

* **DimClient** - storing client name information
* **DimTask** - storing task name information
* **DimEmployee** - storing employee information
* **DimProject**- storing project information alongside a relationship to the client table. It would make it easier to retrieve all projects for a specific client.
* **DimDate** - this will store pre-calculated date fields to enable advanced time-based analysis and would prevent performing date related aggregations ad-hoc.

**Notes**

* Azure SQL pool does not support identity columns and so I had to improvise the means of ensuring uniqueness of columns in the dimension tables hence, I made use of row\_number to assign unique integers to all distinct values. By using this, uniqueness of rows are enforced.
* Data duplication is also avoided in this layer
* Creating tables with REPLICATE distribution, this means that the table's data will be fully copied to each compute node in the data warehouse. Since the table is available on all nodes, joins with large tables can be performed locally on each node, reducing data movement. This method is typically used for small dimension tables or lookup tables that are frequently joined with larger fact tables.
* I also made use of CTEs to join existing fields in both float and ClickUp staging tables to ensure that I am having dimensions that are present in both tables.
* A pre-calculated date table provides pre-calculated fields like week number, quarter, etc., reducing the need for on-the-fly calculations. It also simplifies and speeds up joins between fact tables and the date dimension.
* During the table creation, a NOT NULL constraint is added to the table to ensure that only rows with the NOT NULL values are populated.

----Dim Client Table

CREATE TABLE [sora].[DimClient] (

ClientID INT NOT NULL,

ClientName VARCHAR(100) NOT NULL

)

WITH (DISTRIBUTION = REPLICATE);

WITH ClientCTE AS (

SELECT DISTINCT

Client,

ROW\_NUMBER() OVER (ORDER BY Client) AS DimClientID

FROM [sora].[StgFloat]

WHERE Client NOT IN (SELECT ClientName FROM [sora].[DimClient])

GROUP BY Client

)

INSERT INTO [sora].[DimClient] (ClientID, ClientName)

SELECT

DimClientID,

Client

FROM ClientCTE;

-----DimTask table

CREATE TABLE [sora].[DimTask] (

TaskID INT NOT NULL,

TaskName VARCHAR(30) NOT NULL

)

WITH (DISTRIBUTION = REPLICATE);

-- Combine tasks from both StgClick and StgFloat

WITH CombinedTasks AS (

SELECT DISTINCT Task

FROM [sora].[StgClick]

UNION

SELECT DISTINCT Task

FROM [sora].[StgFloat]

),

-- Select tasks that are not already in DimTask

NewTasks AS (

SELECT Task,

ROW\_NUMBER() OVER (ORDER BY Task) AS RowNum

FROM CombinedTasks

WHERE Task NOT IN (SELECT TaskName FROM [sora].[DimTask])

)

-- Insert new tasks into DimTask

INSERT INTO [sora].[DimTask] (TaskID, TaskName)

SELECT

RowNum,

Task

FROM NewTasks;

----DimEmployee

CREATE TABLE sora.DimEmployee (

EmployeeID INT NOT NULL,

EmployeeName VARCHAR(60) NOT NULL,

Role VARCHAR(100) NOT NULL

)

WITH (DISTRIBUTION = REPLICATE);

WITH CombinedEmployees AS (

SELECT DISTINCT

Name AS EmployeeName,

Role

FROM [sora].[StgFloat]

UNION

SELECT DISTINCT

Name AS EmployeeName,

NULL AS Role

FROM [sora].[StgClick]

WHERE Name NOT IN (SELECT Name FROM [sora].[StgFloat])

),

-- Assign Row Numbers for EmployeeID

EmployeeCTE AS (

SELECT

EmployeeName,

Role,

ROW\_NUMBER() OVER (ORDER BY EmployeeName) AS EmployeeID

FROM CombinedEmployees

)

-- Insert into DimEmployee

INSERT INTO [sora].[DimEmployee] (EmployeeID, EmployeeName, Role)

SELECT

EmployeeID,

EmployeeName,

Role

FROM EmployeeCTE;

----DimProject

CREATE TABLE [sora].[DimProject] (

ProjectID INT NOT NULL,

ProjectName NVARCHAR(90) NOT NULL,

ClientID INT NOT NULL

)

WITH (DISTRIBUTION = REPLICATE);

WITH ProjectCTE AS (

SELECT DISTINCT

C.ClientID,

F.Project,

ROW\_NUMBER() OVER (ORDER BY F.Project) AS ProjectID

FROM sora.StgFloat F

INNER JOIN sora.DimClient C ON F.Client = C.ClientName

WHERE F.Project NOT IN (SELECT ProjectName FROM sora.DimProject)

GROUP BY C.ClientID, F.Project

)

INSERT INTO sora.DimProject (ProjectID, ProjectName, ClientID)

SELECT

ProjectID,

Project,

ClientID

FROM ProjectCTE;

---DimDate

CREATE TABLE [sora].[DimDate] (

DateID INT NOT NULL,

[TaskDate] DATE NOT NULL,

Year INT NOT NULL,

Quarter INT NOT NULL,

Month INT NOT NULL,

Day INT NOT NULL,

Weekday NVARCHAR(10) NOT NULL

)

WITH (DISTRIBUTION = REPLICATE);

WITH DateCTE AS (

SELECT DISTINCT

[TaskDate],

YEAR([TaskDate]) AS Year,

DATEPART(QUARTER, [TaskDate]) AS Quarter,

MONTH([TaskDate]) AS Month,

DAY([TaskDate]) AS Day,

DATENAME(WEEKDAY, [TaskDate]) AS Weekday,

ROW\_NUMBER() OVER (ORDER BY [TaskDate]) AS RowNum

FROM [sora].[StgClick]

WHERE [TaskDate] NOT IN (SELECT [TaskDate] FROM [sora].[DimDate])

GROUP BY TaskDate

)

INSERT INTO [sora].[DimDate] (DateID, [TaskDate], Year, Quarter, Month, Day, Weekday)

SELECT

RowNum,

[TaskDate],

Year,

Quarter,

Month,

Day,

Weekday

FROM DateCTE;

## **What was done before loading the fact table?**

To ensure that I am not joining based on names which can be problematic as it could contain white spaces or different cases of letters that might be subtle to detect, I wanted to ensure consistency of data by joining on integers as opposed to string. I did this by creating a unique identity for fields in the [stg.Click](https://stg.click) table which was chosen because it contained not just the start and end dates but because some tasks in the [stg.Click](https://stg.click) table are not tracked in the Float tables for instance **Ana Suarez**. The Click and Float tables do not have unique identities.

-------Clicks with IDs for the joins

CREATE TABLE [sora].[StgClickWithIDs] (

Client NVARCHAR(100),

Project NVARCHAR(100),

Name NVARCHAR(100),

Task NVARCHAR(100),

Date DATE,

Hours FLOAT,

Note VARCHAR(150),

Billable NVARCHAR(3),

ClientID INT,

ProjectID INT,

EmployeeID INT,

TaskID INT,

DateID INT

)

WITH (DISTRIBUTION = REPLICATE);

INSERT INTO [sora].[StgClickWithIDs] (

Client, Project, Name, Task, Date, Hours, Note, Billable, ClientID, ProjectID, EmployeeID, TaskID, DateID

)

SELECT

c.Client,

c.Project,

c.Name,

c.Task,

c.TaskDate,

c.Hours,

c.Note,

c.Billable,

dc.ClientID,

dp.ProjectID,

de.EmployeeID,

dt.TaskID,

dd.DateID

FROM [sora].[StgClick] c

LEFT JOIN [sora].[DimClient] dc ON dc.ClientName = c.Client

LEFT JOIN [sora].[DimProject] dp ON dp.ProjectName = c.Project

LEFT JOIN [sora].[DimEmployee] de ON de.EmployeeName = c.Name

LEFT JOIN [sora].[DimTask] dt ON dt.TaskName = c.Task

LEFT JOIN [sora].[DimDate] dd ON dd.TaskDate = c.TaskDate;

## **How was the fact table loaded?**

Based on the tables, data for time logging and billing can be aggregated to generate insights. The table is called **FactTimeSheet** as the hours logged and estimated are the key aggregate data fields. A unique key called **TimeEntryID** is used to store unique fact rows. With the fact now populated, several questions can be answered. Using a ROW\_NUMBER function is used to enforce an IDENTITY constraint to ensure uniqueness of rows.

**Notes**

* The table distribution is HASH. This distributes rows based on the hash value of the ClientID column. It ensures that rows with the same ClientID are sorted on the same compute node/distribution node. This improves query performance for joins and aggregations. It would reduce data movement during joins on the ClientID column, it distributes data and avoids data skews
* The table contains primary keys from the dimension table layer as well as additional information to be aggregated(EstimatedHours, Billable, Hours). It also has the start and end date which can be used to filter data by this range.
* Before loading data into the fact table, the code includes validation steps to ensure that all necessary IDs (e.g., ClientID, TaskID, DateID) are not NULL. This is achieved through the WHERE clause in the INSERT statement.
* By using LEFT JOIN operations and filtering out NULL values, the code ensures that only valid and complete records are inserted into the fact table.

--Fact table

CREATE TABLE [sora].[FactTimeSheet](

TimeEntryID INT NOT NULL,

ClientID INT NOT NULL,

ProjectID INT NOT NULL,

EmployeeID INT NOT NULL,

TaskID INT NOT NULL,

DateID INT NOT NULL,

Hours FLOAT NOT NULL,

EstimatedHours FLOAT,

Note VARCHAR(150),

Billable VARCHAR(3) NOT NULL,

StartDate DATE,

EndDate DATE

)

WITH (DISTRIBUTION = HASH(ClientID), CLUSTERED COLUMNSTORE INDEX);

-- Insert into FactTimeSheet

INSERT INTO [sora].[FactTimeSheet] (

TimeEntryID, ClientID, ProjectID, EmployeeID, TaskID, DateID, Hours, EstimatedHours, Note, Billable, StartDate, EndDate

)

SELECT

ROW\_NUMBER() OVER (ORDER BY (SELECT NULL)) AS TimeEntryID,

sc.ClientID,

sc.ProjectID,

sc.EmployeeID,

sc.TaskID,

sc.DateID,

sc.Hours,

f.EstimatedHours,

sc.Note,

CASE WHEN sc.Billable = 'Yes' THEN 1 ELSE 0 END AS Billable,

f.StartDate,

f.EndDate

FROM sora.StgClickWithIDs sc

LEFT JOIN [sora].[StgFloat] f ON sc.Client = f.Client AND sc.Project = f.Project AND sc.Name = f.Name AND sc.Task = f.Task

## **Generated insight**

### What are the total hours worked by each employee?

--Total Hours Worked by Each Employee

SELECT

de.EmployeeName,

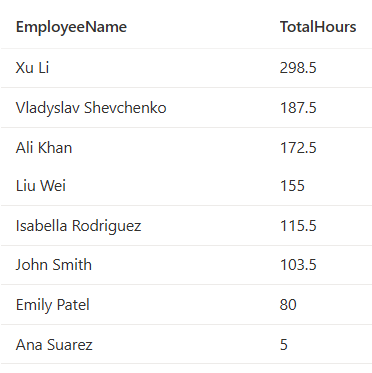
SUM(ft.Hours) AS TotalHours

FROM [sora].[FactTimeSheet] ft

JOIN [sora].[DimEmployee] de ON ft.EmployeeID = de.EmployeeID

GROUP BY de.EmployeeName

ORDER BY TotalHours DESC;



### What is the total billable hours to the client per project?

--Total billable hours

SELECT

dp.ProjectName,

SUM(ft.Hours) AS TotalBillableHours

FROM [sora].[FactTimeSheet] ft

JOIN [sora].[DimProject] dp ON ft.ProjectID = dp.ProjectID

WHERE ft.Billable = 1

GROUP BY dp.ProjectName

ORDER BY TotalBillableHours DESC;



### What is the average hours worked by employees per day?

--Average hours worked by employees per day

SELECT

de.EmployeeName,

ROUND(AVG(ft.Hours),2) AS AverageHoursPerDay

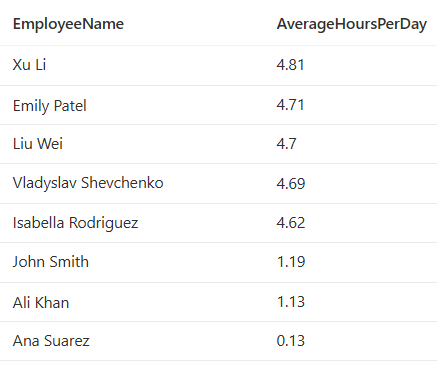
FROM [sora].[FactTimeSheet] ft

JOIN [sora].[DimEmployee] de ON ft.EmployeeID = de.EmployeeID

JOIN [sora].[DimDate] dd ON ft.DateID = dd.DateID

GROUP BY de.EmployeeName

ORDER BY AverageHoursPerDay DESC;



### 

### 

### **What is the hours worked by each employee on each project**

--Hours worked by each employee on each project

SELECT

de.EmployeeName,

dp.ProjectName,

SUM(ft.Hours) AS TotalHours

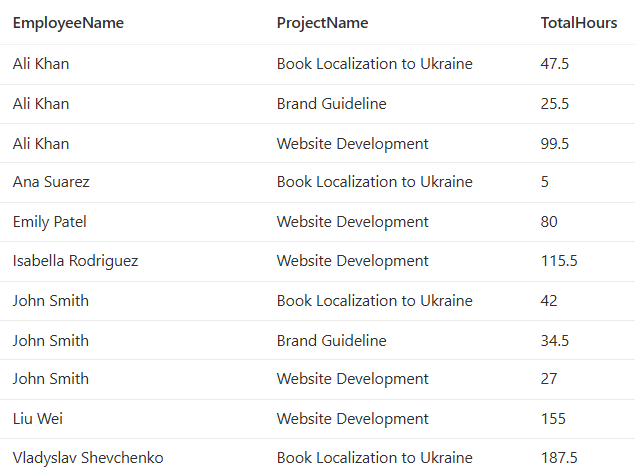
FROM [sora].[FactTimeSheet] ft

JOIN [sora].[DimEmployee] de ON ft.EmployeeID = de.EmployeeID

JOIN [sora].[DimProject] dp ON ft.ProjectID = dp.ProjectID

GROUP BY de.EmployeeName, dp.ProjectName

ORDER BY de.EmployeeName, dp.ProjectName;



# 

# 

# 

# **Question 2**

**Unoptimized code**

SELECT

c.Name,

f.Role,

SUM(c.hours) AS Total\_Tracked\_Hours,

SUM(f.EstimatedHours) AS Total\_Allocated\_Hours,

c.TaskDate

FROM sora.stgClick c

JOIN sora.stgFloat f on c.Name = f.Name

GROUP BY c.Name, f.Role

HAVING SUM(c.hours) > 100

ORDER BY Total\_Allocated\_Hours DESC;

Steps to reflector code:

1. Firstly, the TaskDate has to be included in the group by clause, otherwise it throws an error. Also changed the column name Estimted\_Hours to EstimatedHours to properly reflect the column.

SELECT

c.Name,

f.Role,

SUM(c.hours) AS Total\_Tracked\_Hours,

SUM(f.EstimatedHours) AS Total\_Allocated\_Hours,

c.TaskDate

FROM sora.stgClick c

JOIN sora.stgFloat f on c.Name = f.Name

GROUP BY c.Name, f.Role, c.TaskDate

HAVING SUM(c.hours) > 100

ORDER BY Total\_Allocated\_Hours DESC;

1. Create indexes on the Click and Float tables

Columnstore indexes are used to enable high efficient reads, it improves query performance. It creates columnstore indexes on columns which is great for reading heavy analytical reads. Creating on the Name column is vital as they are used on the join condition.

-- Create clustered columnstore indexes for better performance on large tables

-- Create an index on the Name column in the ClickUp table

CREATE INDEX IX\_ClickUp\_Name ON ClickUp(Name);

-- Create an index on the Name column in the Float table

CREATE INDEX IX\_Float\_Name ON Float(Name);

1. Since the tables were created with REPLICATE distribution, data movement is minimized between compute nodes thereby reducing I/O operations. There for there was little need to partition the table by dropping and rebuilding it.

Ensure that your queries are written efficiently. Avoid unnecessary columns in the SELECT clause and use appropriate filtering conditions to reduce the amount of data processed.

**Final Code**

**CREATE INDEX IX\_ClickUp\_Name ON ClickUp(Name);**

**CREATE INDEX IX\_Float\_Name ON Float(Name);**

**SELECT**

**c.Name,**

**f.Role,**

**SUM(c.hours) AS Total\_Tracked\_Hours,**

**SUM(f.EstimatedHours) AS Total\_Allocated\_Hours,**

**c.TaskDate**

**--CAST(MAX(c.TaskDate) AS DATE) AS Last\_Tracked\_Date**

**FROM sora.stgClick c**

**JOIN sora.stgFloat f ON c.Name = f.Name**

**GROUP BY c.Name, f.Role,**

**HAVING SUM(c.hours) > 100**

**ORDER BY Total\_Allocated\_Hours DESC;**

# 

# 

# **Question 3**

## Introduction

### What Dataset would be used?

I would be making use a a stock market data set from Kaggle - [https://www.kaggle.com/datasets/jakewright/9000-tickers-of-stock-market-data-full-history?select=all\_stock\_data](https://www.kaggle.com/datasets/jakewright/9000-tickers-of-stock-market-data-full-history?select=all_stock_data.parquet). I would be analyzing the CSV file format. It has about 34 million rows of records or stock market data

| Date | Ticker | Open | High | Low | Close | Volume | Dividends | Stock Splits |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1962-01-02 | ED | 0 | 0.2658275556 | 0.2617876232 | 0.2617876232 | 25600 | 0 | 0 |
| 1962-01-02 | CVX | 0 | 0.04680890217 | 0.04606926601 | 0.04680890217 | 105840 | 0 | 0 |
| 1962-01-02 | GD | 0 | 0.2100327595 | 0.2030607079 | 0.2082897425 | 2648000 | 0 | 0 |
| 1962-01-02 | BP | 0 | 0.1414393309 | 0.1395279765 | 0.1395279765 | 77440 | 0 | 0 |
| 1962-01-02 | MSI | 0 | 0.7649229763 | 0.7452535214 | 0.7518101931 | 65671 | 0 | 0 |
| 1962-01-02 | HON | 0 | 1.559642297 | 1.549127912 | 1.556137562 | 40740 | 0 | 0 |
| 1962-01-02 | FL | 0 | 0.9722493029 | 0.9538055894 | 0.9590751529 | 49200 | 0 | 0 |
| 1962-01-02 | GT | 0 | 1.94690013 | 1.914270519 | 1.936023593 | 32000 | 0 | 0 |
| 1962-01-02 | JNJ | 0 | 0.06776573509 | 0.06741443839 | 0.06776573509 | 0 | 0 | 0 |
| 1962-01-02 | MMM | 0 | 0.5418908527 | 0.5259528843 | 0.5299372673 | 254509 | 0 | 0 |

### What is the transformation objective?

The goal is to calculate the max\_high of each stock, per month per year. Month and year would be extracted from the Date column. The dataframe would then be grouped by Ticker, year and month and the max value of the High column would be aggregated for each group. The query execution time would be calculated

### ***How was it implemented?***

Necessary functions were imported. Data was read from the data lake and counted. After which, the transformation logic was implemented. And the time of execution was noted

import time

from pyspark.sql.functions import col, max, year, month

df = spark.read.load(

'abfss://filesystem@azprdneudwcest01.dfs.core.windows.net/synapse/workspaces/azprd-neu-dwce-synw01/warehouse/all\_stock\_data.csv',

format='csv',

header=True,

inferSchema=True

)

df.count()

df2 = df.select("Date", "Ticker", "High")\

.withColumn("Year", year(col("Date")))\

.withColumn("Month", month(col("Date")))

start\_time = time.time()

df = df.select("Date", "Ticker", "High").withColumn("Year", year(col("Date"))).withColumn("Month", month(col("Date")))

# Group by Ticker, Year, and Month, and calculate max High

max\_high\_df = df2.groupBy("Ticker", "Year", "Month").agg(max("High").alias("MaxHigh"))

# Show the result

max\_high\_df.show()

end\_time = time.time()

# Calculate the elapsed time

elapsed\_time = end\_time - start\_time

print(f"Time taken to run the query: {elapsed\_time} seconds")

### Optimization of query

* I would create a partition on this file using a new column called Year, which is derived from the Date column.
* This partitioned file would then be saved in Parquet format. What are the benefits of this? Parquet supports schema evolution which supports flexibility of data structures over time. Parquet is also a columnar file format meaning it stores data by columns and not by row which is better for compression. For example, the CSV format of the dataset is almost 4GB in size while the parquet format is less than 1GB. This is great for working with large datasets
* As there would not be numerous operations that have to make use of the dataset, I chose not to cache or repartition the dataframe

df = spark.read.load('abfss://filesystem@azprdneudwcest01.dfs.core.windows.net/synapse/workspaces/azprd-neu-dwce-synw01/warehouse/all\_stock\_data.csv', format='csv', header = True)

# Extract year from Date and add it as a new column

df = df.withColumn("Year", year(col("Date")))

# Repartition the DataFrame by the Year column

partitioned\_df = df.repartition("Year")

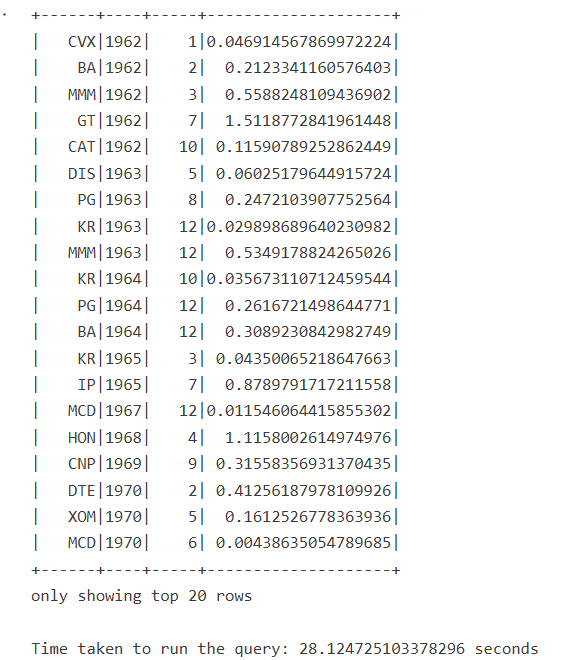
# Save the partitioned DataFrame to a Parquet file

partitioned\_df.write.partitionBy("Year").parquet("abfss://filesystem@azprdneudwcest01.dfs.core.windows.net/synapse/workspaces/azprd-neu-dwce-synw01/warehouse/all\_stock\_data.parquet")

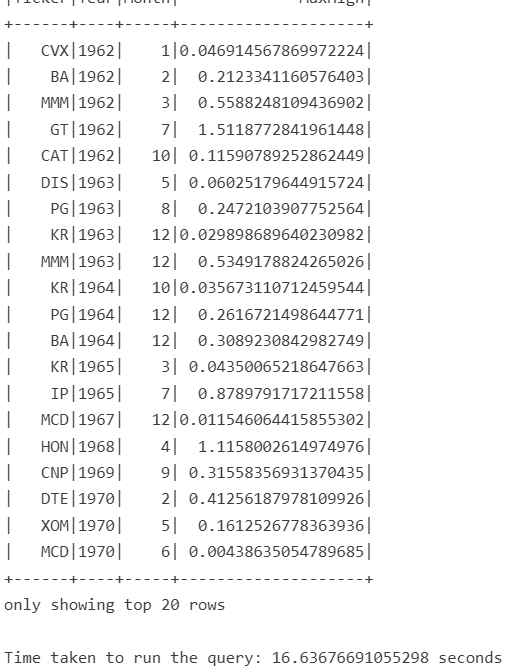
### **Results**

Without the optimisation, the query took almost 28 seconds, while with the optimisation, it took about 16 seconds

1. Non - optimized



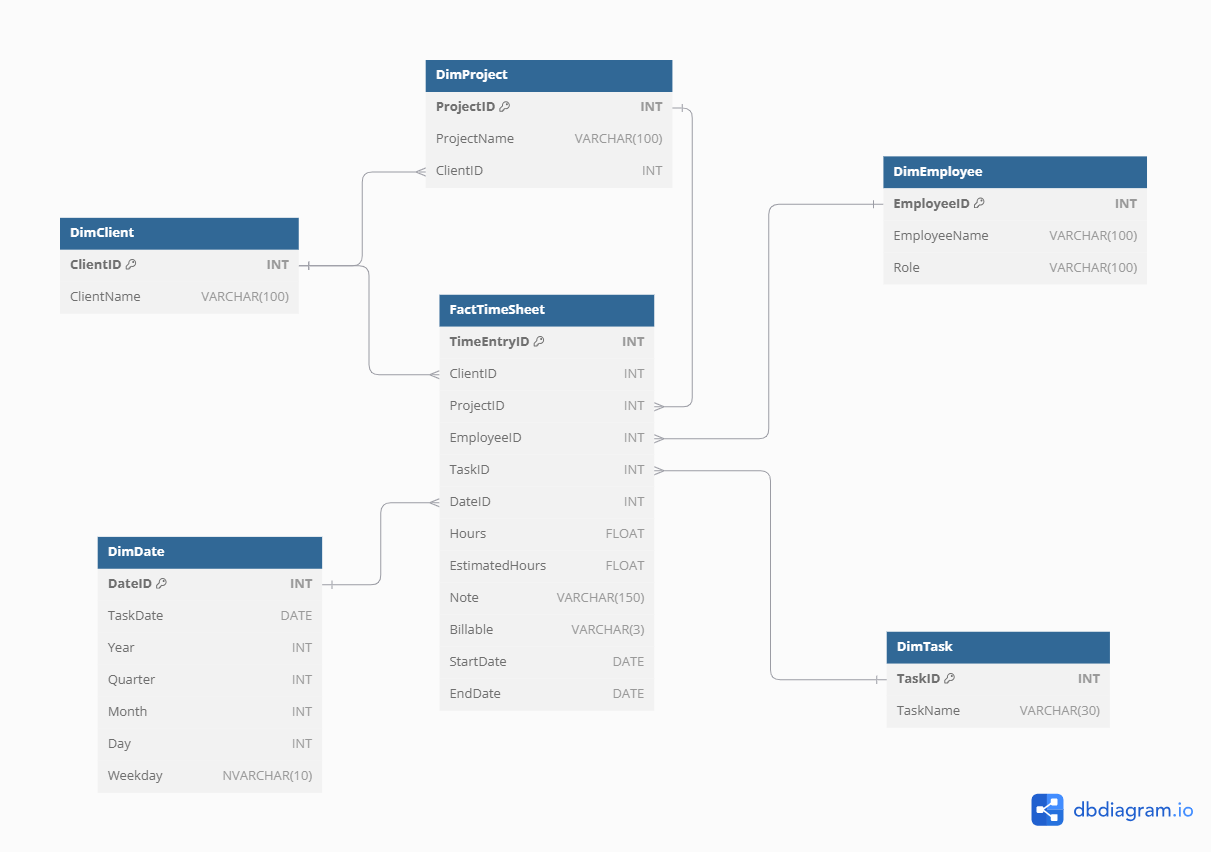
1. Optimized



# Question 4

## Analytical model

It is derived based on the values from question 1. I would design the Analytical model exactly as I designed the answer to **question 1.**



## Operation model

The entity relationship model should be able to support project/employee management systems to track clients, projects and time tracking.

### What are the decisions taken?

1. **Entities**: I created entities under the following self describing names Client, Project, Person, Role, Task, TimeLog, WorkSchedule. This is to ensure that there is proper establishment of data boundaries. This would support scalability as new attributes can be added to existing entities without compromising the others. For example, I could add new projects or employees in the future and not cause data redundancy(data duplication).
2. **Relationships**: all rows in the tables would have primary keys to ensure data integrity and foreign keys will be used to establish relationships to ensure data referential integrity. I want to be sure that foreign keys properly reference the appropriate data.
3. **Time log** and **Work-schedule** tables:

* Time log table will be used to track details of real/actual work hours such as the date, hours worked, notes and billable columns. This enables work to be tracked per person per task. It further enables billing and performance analysis.
* Work schedule houses data such as RoleID, TaskID, EstimatedHours - this will be used to assign employees to roles and tasks.’

1. **Cardinality**: This defines the relationship between rows of one table and rows of another

* **Client** - **Project**: *One-to-Many* - meaning that a client can have multiple projects. Each project is associated with one client but a client can be associated with many projects.
* **Project** - **Task**: *One-to-Many* - a project can have multiple task and each task is associated with one project
* **Employee** - **TimeLog**: One-to-Many - an employee can have multiple time entries and each time entry is associated with one employee
* **Employee** - **WorkSchedule**: *One -to -many* - An employee can have multiple work schedules across different projects. Each work schedule is associated with one employee
* **Task** - **TimeLog**: *one-to-many*: a task can have multiple time logs. Each time log entry is associated with one task, but a task can have many time log entries.
* **Project** - **WorkSchedule**: *one-to-many*: A project can have multiple estimations for its tasks. Each estimation is associated with one project, but a project can have many estimations.

