

## 1)Executive Summary:

Our project aims to build a comprehensive data warehousing solution for stock market analytics, leveraging structured data from various dimensions and a central fact table. The primary objective is to facilitate detailed historical analysis and forecasting for stock market performance, enhancing decision-making capabilities for investors and financial analysts.

### Project Highlights:

- **Data Dimensions:** We have structured our data around key dimensions such as time, stocks, companies, and indices. Each dimension offers detailed attributes crucial for comprehensive analysis.
- **Fact Table:** The fact table consolidates daily stock market data, including opening, high, low, closing prices, adjusted closing prices, and volume. This allows for granular analysis and reporting at various levels of aggregation.
- **Data Integration and Quality:** Rigorous data collection, integration, and cleansing processes ensure the accuracy and reliability of the data warehouse. This enhances the trustworthiness of insights derived from the system.
- **Analytical Capabilities:** Our solution supports diverse analytical needs, from exploratory data analysis (EDA) to advanced modeling and forecasting. Users can conduct trend analysis, correlation studies, and scenario modeling to uncover actionable insights.
- **Visualization and Reporting:** Interactive dashboards and intuitive visualizations enable users to intuitively explore data trends and patterns. This facilitates effective communication of insights across stakeholders.

**Future Directions:** Moving forward, the project will focus on expanding data sources, integrating additional financial instruments, and enhancing predictive modeling capabilities. Continuous improvement in data governance and user interface design will ensure the system remains robust and user-friendly.

In summary, our data warehousing project offers a powerful tool for analyzing historical stock market data, empowering stakeholders with actionable insights and facilitating informed decision-making in financial markets.

## 2) Problem Statement

**Context:** In the dynamic landscape of financial markets, stakeholders constantly seek robust tools to analyze historical stock market data comprehensively. This project addresses the need for a sophisticated data warehousing solution tailored to facilitate in-depth analysis and decision-making.

**Issue:** The challenge lies in efficiently integrating and analyzing vast amounts of daily stock market data across multiple dimensions, including time, stocks, companies, and indices. Traditional methods often lack scalability and real-time insights required by modern investors and analysts.

**Relevance:** Accurate historical data analysis is crucial for predicting market trends, identifying investment opportunities, and mitigating financial risks. A reliable data warehouse can significantly enhance the ability to derive actionable insights from historical stock performance data.

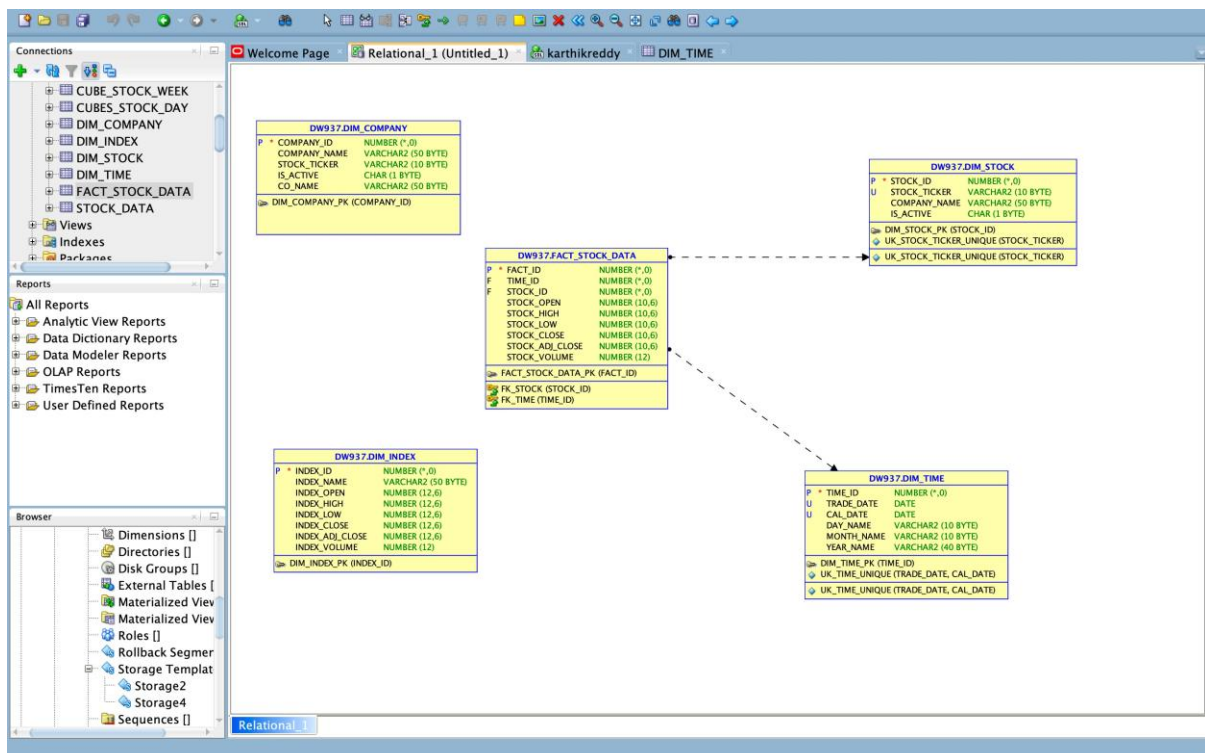
**Objective:** This project aims to design and implement a data warehousing solution that consolidates historical stock market data into a centralized repository. By structuring data

around key dimensions and employing advanced analytics, the objective is to empower users with intuitive tools for trend analysis, forecasting, and strategic decision-making.

**Significance:** Addressing this challenge not only enhances the efficiency and accuracy of financial analysis but also provides a competitive edge in navigating volatile market conditions. The project's outcome will support informed decision-making, improve investment strategies, and ultimately contribute to better financial outcomes for stakeholders.

## Entity-Relationship Diagram (ERD)

The ERD below demonstrates the design of the data cubes used in the project, showcasing the dimensional modeling skills applied:



## Explanation:

**Dimensions:** Time, Stock, Company, and Index.

**Fact Table:** Captures daily stock market data including prices and volumes.

**Relationships:** Fact table links to dimension tables through foreign keys.

## "5W+H" approach:

**Who:** Investors, financial analysts, and stakeholders in the stock market industry are interested in this problem. They rely on historical data to make informed decisions regarding investments, risk management, and strategic planning.

**What:** The challenge is to effectively analyze and derive insights from extensive historical stock market data. This includes daily trading information such as opening, high, low, and closing prices, adjusted closing prices, and trading volumes across various stocks, companies, and indices.

**When and Where:** This problem arises daily in global financial markets where data is generated continuously, spanning different time zones and trading sessions. Timely access to

accurate historical data is critical for making decisions in real-time and for retrospective analysis.

**Why:** Addressing this challenge is essential to improve decision-making accuracy, enhance investment strategies, and mitigate financial risks. By leveraging comprehensive historical data, stakeholders can identify market trends, correlations, and anomalies that impact investment outcomes.

**How:** This challenge can be effectively solved by designing and implementing a robust data warehousing solution. This solution will consolidate diverse datasets from different dimensions (time, stocks, companies, indices) into a unified repository. Advanced analytics tools will enable users to perform detailed trend analysis, predictive modelling, and scenario planning, thus empowering them with actionable insights.

### 3: Literature Review

Implementing OLAP Concepts:

#### **Dimensional Modeling:**

Dimensional modeling is a design technique used in data warehousing to organize and structure data for easy querying and analysis. It typically involves creating dimension tables that describe the business entities and a central fact table that holds numerical metrics or measures associated with these entities.

#### *1. Dimension Tables*

##### **dim\_time:**

- **Attributes:** TIME\_ID (Primary Key), TRADE\_DATE, CAL\_DATE, DAY\_NAME, MONTH\_NAME, YEAR\_NAME.
- **Purpose:** Stores time-related attributes used for analyzing stock market data by different time dimensions such as day, month, and year.

##### **dim\_stock:**

- **Attributes:** STOCK\_ID (Primary Key), STOCK\_TICKER, COMPANY\_NAME, IS\_ACTIVE.
- **Purpose:** Describes details about stocks, including company information and activity status.

##### **dim\_company:**

- **Attributes:** COMPANY\_ID (Primary Key), COMPANY\_NAME, STOCK\_TICKER, IS\_ACTIVE, CO\_NAME.
- **Purpose:** Provides additional information about companies listed in the stock market, linked to stock ticker and company name.

##### **dim\_index:**

- **Attributes:** INDEX\_ID (Primary Key), INDEX\_NAME, INDEX\_OPEN, INDEX\_HIGH, INDEX\_LOW, INDEX\_CLOSE, INDEX\_ADJ\_CLOSE, INDEX\_VOLUME.
- **Purpose:** Contains data about market indices, including daily trading metrics like opening, high, low, closing prices, adjusted close, and volume.

## 2. Fact Table

### fact\_stock\_data:

- **Attributes:** FACT\_ID (Primary Key), TIME\_ID (Foreign Key to dim\_time), STOCK\_ID (Foreign Key to dim\_stock), STOCK\_OPEN, STOCK\_HIGH, STOCK\_LOW, STOCK\_CLOSE, STOCK\_ADJ\_CLOSE, STOCK\_VOLUME.
- **Purpose:** Central fact table capturing detailed stock market data for analysis and reporting. It links to dimension tables (dim\_time and dim\_stock) through foreign key relationships and stores numeric measures such as stock prices and trading volumes.

## Dimensional Modelling Benefits

- **Simplicity:** Dimensional models are straightforward and intuitive, making it easier to understand and query data.
- **Query Performance:** Optimized for query performance, as data is denormalized and aggregated, reducing the need for complex joins.
- **Flexibility:** Supports various types of analysis (e.g., by time, by company) due to its multidimensional structure.
- **Scalability:** Allows for easy expansion and integration of new dimensions or measures as business requirements evolve.

## Hierarchies Definition

### 1. dim\_time

#### Hierarchy: Time

- **Year > Month > Day**

#### Attributes:

- YEAR\_NAME
- MONTH\_NAME
- DAY\_NAME

#### Purpose:

- Allows aggregation and analysis of stock market data across different time periods:
  - **Year:** Aggregate data by year to analyze annual trends.
  - **Month:** Drill down to monthly data to observe monthly fluctuations.
  - **Day:** Detailed analysis at the daily level for specific trading days.

### 2. dim\_stock

#### Hierarchy: Stock

- **Company > Stock Ticker**

#### Attributes:

- COMPANY\_NAME
- STOCK\_TICKER

**Purpose:**

- Organizes stocks by company, facilitating analysis of performance and market activities specific to each company.
- Allows users to drill down from company-level insights to individual stock ticker details.

### *3. dim\_company*

**Hierarchy: Company**

- **Company Name**

**Attributes:**

- COMPANY\_NAME

**Purpose:**

- Provides a single-level hierarchy for companies listed in the stock market
- Enables analysis and comparison of company-specific metrics across different dimensions (e.g., time, index).

### *4. dim\_index*

**Hierarchy: Index**

- **Index Name**

**Attributes:**

- INDEX\_NAME

**Purpose:**

- Represents market indices for comparative analysis and benchmarking.
- Facilitates aggregation and comparison of index performance over time and across other dimensions.

**Benefits of Hierarchies**

- **Navigational Efficiency:** Users can navigate through data dimensions intuitively, from higher-level summaries to detailed insights.
- **Analytical Flexibility:** Supports drill-down, roll-up, and slice-and-dice operations for deeper analysis of data.
- **Aggregation Capabilities:** Enables aggregation of data at various levels of granularity, enhancing analytical capabilities.

## Example Queries for Building Data Cubes and Defining Aggregation Levels

### *1. Aggregate Data by Day*

```
CREATE TABLE cube_stock_day AS
SELECT
    d.TIME_ID,
    d.TRADE_DATE,
    s.STOCK_ID,
    s.STOCK_TICKER,
    SUM(f.STOCK_OPEN) AS TOTAL_OPEN,
    SUM(f.STOCK_HIGH) AS TOTAL_HIGH,
    SUM(f.STOCK_LOW) AS TOTAL_LOW,
    SUM(f.STOCK_CLOSE) AS TOTAL_CLOSE,
    SUM(f.STOCK_ADJ_CLOSE) AS TOTAL_ADJ_CLOSE,
    SUM(f.STOCK_VOLUME) AS TOTAL_VOLUME
FROM
    fact_stock_data f
    JOIN dim_time d ON f.TIME_ID = d.TIME_ID
    JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
GROUP BY
    d.TIME_ID,
    d.TRADE_DATE,
    s.STOCK_ID,
    s.STOCK_TICKER;
```

### *2. Aggregate Data by Week*

```
CREATE TABLE cube_stock_week AS
SELECT
    TO_CHAR(d.TRADE_DATE, 'IW') AS WEEK_NUMBER, -- Week number based on ISO week date
    TO_CHAR(d.TRADE_DATE, 'YYYY') AS YEAR,
    s.STOCK_ID,
    s.STOCK_TICKER,
    SUM(f.STOCK_OPEN) AS TOTAL_OPEN,
    SUM(f.STOCK_HIGH) AS TOTAL_HIGH,
    SUM(f.STOCK_LOW) AS TOTAL_LOW,
    SUM(f.STOCK_CLOSE) AS TOTAL_CLOSE,
    SUM(f.STOCK_ADJ_CLOSE) AS TOTAL_ADJ_CLOSE,
    SUM(f.STOCK_VOLUME) AS TOTAL_VOLUME
FROM
    fact_stock_data f
    JOIN dim_time d ON f.TIME_ID = d.TIME_ID
    JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
GROUP BY
    TO_CHAR(d.TRADE_DATE, 'IW'),
    TO_CHAR(d.TRADE_DATE, 'YYYY'),
    s.STOCK_ID,
    s.STOCK_TICKER;
```

### *3. Aggregate Data by Month*

```
CREATE TABLE cube_stock_month AS
SELECT
    TO_CHAR(d.TRADE_DATE, 'YYYY-MM') AS MONTH,
    s.STOCK_ID,
    s.STOCK_TICKER,
    SUM(f.STOCK_OPEN) AS TOTAL_OPEN,
    SUM(f.STOCK_HIGH) AS TOTAL_HIGH,
    SUM(f.STOCK_LOW) AS TOTAL_LOW,
    SUM(f.STOCK_CLOSE) AS TOTAL_CLOSE,
    SUM(f.STOCK_ADJ_CLOSE) AS TOTAL_ADJ_CLOSE,
    SUM(f.STOCK_VOLUME) AS TOTAL_VOLUME
FROM
    fact_stock_data f
```

```

JOIN dim_time d ON f.TIME_ID = d.TIME_ID
JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
GROUP BY
  TO_CHAR(d.TRADE_DATE, 'YYYY-MM'),
  s.STOCK_ID,
  s.STOCK_TICKER;

```

## Explanation:

- **Dimension Tables (dim\_time, dim\_stock):** These tables are joined with fact\_stock\_data to associate time-related and stock-related attributes with the aggregated measures.
- **Grouping and Aggregation:** SUM() functions aggregate numeric measures (STOCK\_OPEN, STOCK\_HIGH, etc.) based on the specified grouping criteria (TIME\_ID, TRADE\_DATE, STOCK\_ID, STOCK\_TICKER).
- **Time Formatting:** TO\_CHAR() functions are used to format dates into week numbers ('IW' for ISO week number) and month ('YYYY-MM' for year-month format) for grouping by week and month respectively.
- **Table Creation:** Results are stored in new tables (cube\_stock\_day, cube\_stock\_week, cube\_stock\_month) representing different levels of aggregation.

## Considerations:

- **Performance:** Ensure indexes are properly defined on join columns (TIME\_ID, STOCK\_ID) and where clauses to optimize query performance.
- **ETL Process:** These queries can be integrated into your ETL process to automate the creation and maintenance of data cubes as new data is loaded.
- **Query Optimization:** Monitor query performance and consider partitioning large tables or using materialized views for further optimization.

## OLAP Queries Using

Here are examples of OLAP queries using to demonstrate common operations:

### ***1. Slice-and-Dice Operation***

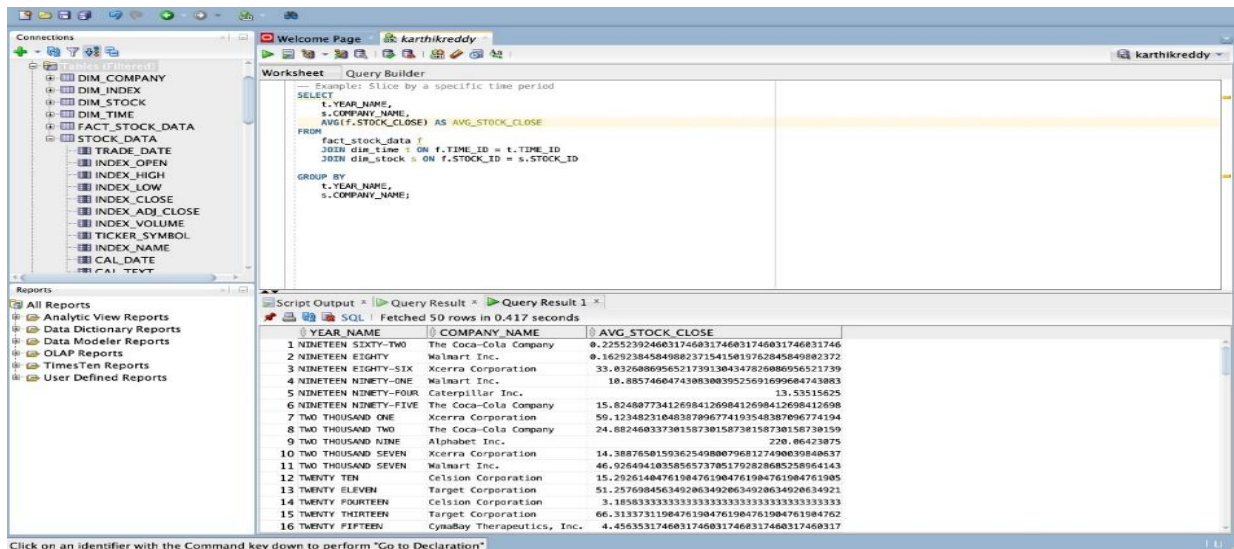
**Slice:** Selecting a subset of data along one dimension.

-- Example: Slice by a specific time period (e.g., year 2023)

```

SELECT
  t.YEAR_NAME,
  s.COMPANY_NAME,
  AVG(f.STOCK_CLOSE) AS AVG_STOCK_CLOSE
FROM
  fact_stock_data f
  JOIN dim_time t ON f.TIME_ID = t.TIME_ID
  JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
WHERE
  t.YEAR_NAME = '2023'
GROUP BY
  t.YEAR_NAME,
  s.COMPANY_NAME;

```



**Dice:** Selecting a subset of data along multiple dimensions.

-- Example: Dice by year and stock ticker

```
SELECT
  t.YEAR_NAME,
  s.STOCK_TICKER,
  AVG(f.STOCK_CLOSE) AS AVG_STOCK_CLOSE
FROM
  fact_stock_data f
  JOIN dim_time t ON f.TIME_ID = t.TIME_ID
  JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
WHERE
  t.YEAR_NAME IN ('2022', '2023')
GROUP BY
  t.YEAR_NAME,
  s.STOCK_TICKER;
```

## 2. Drill-Down Operation

Drill down from higher to lower levels of granularity.

-- Example: Drill down from month to day level for a specific company

```
SELECT
  t.MONTH_NAME,
  t.DAY_NAME,
  s.COMPANY_NAME,
  AVG(f.STOCK_CLOSE) AS AVG_STOCK_CLOSE
FROM
  fact_stock_data f
  JOIN dim_time t ON f.TIME_ID = t.TIME_ID
  JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
WHERE
  t.MONTH_NAME = 'June'
  AND s.COMPANY_NAME = 'Company XYZ'
GROUP BY
  t.MONTH_NAME,
  t.DAY_NAME,
  s.COMPANY_NAME;
```



The screenshot shows a BI tool interface with a 'Tables (Filtered)' pane on the left, a 'Query Builder' pane in the center, and a 'Query Result' pane at the bottom. The 'Query Builder' pane contains a query example: 'Example: Drill down from month to day level for a specific company'. The 'Query Result' pane shows a table with 16 rows and 4 columns: MONTH\_NAME, DAY\_NAME, COMPANY\_NAME, and AVG\_STOCK\_CLOSE.

MONTH_NAME	DAY_NAME	COMPANY_NAME	AVG_STOCK_CLOSE
1 JUNE	MONDAY	The Coca-Cola Company	19.93510317432950191570881226053639846743
2 JANUARY	THURSDAY	Caterpillar Inc.	35.68454655125725338491295938184448742747
3 NOVEMBER	MONDAY	Caterpillar Inc.	34.6014673046875
4 DECEMBER	THURSDAY	The Coca-Cola Company	19.23039198249027237354085603112840466926
5 SEPTEMBER	THURSDAY	Caterpillar Inc.	34.68402239803921568627450980392156862745
6 SEPTEMBER	WEDNESDAY	Walmart Inc.	34.05471033978494623655913978494623655914
7 MAY	TUESDAY	Walmart Inc.	36.2330279050505050505050505050505051
8 JUNE	TUESDAY	Walmart Inc.	36.12585285412262156448202959838866807611
9 NOVEMBER	THURSDAY	Walmart Inc.	35.488866966808287292817679558011049724
10 JULY	WEDNESDAY	Walmart Inc.	35.50259660729613733905579399141630901288
11 DECEMBER	TUESDAY	Walmart Inc.	35.81737785593220338983050847457627118644
12 MARCH	WEDNESDAY	Walmart Inc.	35.72653467413441955193482688391038696538
13 FEBRUARY	WEDNESDAY	Xcerra Corporation	21.97603762135922330097807378640776699029
14 FEBRUARY	FRIDAY	Xcerra Corporation	22.8399897810218978102189781021897810219
15 NOVEMBER	MONDAY	Target Corporation	31.76617194736842105263157894736842105263
16 JUNE	THURSDAY	Target Corporation	32.90561714765100671140939597315436241611

### 3. Roll-Up Operation

Aggregate data from lower to higher levels of granularity.

-- Example: Roll up from day to month level for a specific index

```

SELECT
    t.MONTH_NAME,
    AVG(f.STOCK_CLOSE) AS AVG_STOCK_CLOSE
FROM
    fact_stock_data f
    JOIN dim_time t ON f.TIME_ID = t.TIME_ID
    JOIN dim_index i ON f.STOCK_ID = i.INDEX_ID
WHERE
    t.YEAR_NAME = '2023'
    AND i.INDEX_NAME = 'S&P 500'
GROUP BY
    t.MONTH_NAME;

```

The screenshot shows the SQL Server Enterprise Manager interface. On the left, the 'Tables (Filtered)' pane lists various tables including DIM\_COMPANY, DIM\_INDEX, DIM\_STOCK, DIM\_TIME, FACT\_STOCK\_DATA, STOCK\_DATA, TRADE\_DATE, INDEX\_OPEN, INDEX\_HIGH, INDEX\_LOW, INDEX\_CLOSE, INDEX\_ADJ\_CLOSE, INDEX\_VOLUME, TICKER\_SYMBOL, INDEX\_NAME, CAL\_DATE, and CAL\_TEXT. The 'Query Builder' pane shows a query to calculate the average stock closing price by month. The 'Query Result' pane displays the results of the query.

**Query:**

```

SELECT
    t.MONTH_NAME,
    AVG(f.STOCK_CLOSE) AS avg_stock_close
FROM
    fact_stock_data f
    JOIN dim_time t ON f.TIME_ID = t.TIME_ID
    JOIN dim_index i ON f.STOCK_ID = i.INDEX_ID
GROUP BY
    t.MONTH_NAME;

```

**Query Result:**

MONTH_NAME	avg_stock_close
1 NOVEMBER	48.87062693684710351377018043684710351377
2 DECEMBER	49.5833047999083349702080653170812283114
3 SEPTEMBER	47.17847986612584361578839730042947712864
4 OCTOBER	47.27204043488634936581698317719452467663
5 JANUARY	50.30358057395616915756316600764023858991
6 FEBRUARY	51.068969589203376604951408101014400227
7 JUNE	51.94817173032363682155899760735423750157
8 APRIL	51.02865037162647846827419460236554923871
9 AUGUST	47.33472866513124684502776375567895002524
10 MARCH	51.71729815465892239248640632723677706377
11 JULY	49.0287177352902633992923601100773162102
12 MAY	51.58490835065596739268882944847790090434

## - Top Performing Stocks by Average Closing Price:

```

SELECT
    s.STOCK_TICKER,
    AVG(f.STOCK_CLOSE) AS avg_stock_close
FROM
    fact_stock_data f
    JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
GROUP BY
    s.STOCK_TICKER
ORDER BY
    avg_stock_close DESC
LIMIT 10;

```

The screenshot shows the SQL Server Enterprise Manager interface. On the left, the 'Tables (Filtered)' pane lists various tables including DIM\_COMPANY, DIM\_INDEX, DIM\_STOCK, DIM\_TIME, FACT\_STOCK\_DATA, STOCK\_DATA, TRADE\_DATE, INDEX\_OPEN, INDEX\_HIGH, INDEX\_LOW, INDEX\_CLOSE, INDEX\_ADJ\_CLOSE, INDEX\_VOLUME, TICKER\_SYMBOL, INDEX\_NAME, CAL\_DATE, and CAL\_TEXT. The 'Query Builder' pane shows a query to find the top 10 performing stocks by average closing price. The 'Query Result' pane displays the results of the query.

**Query:**

```

SELECT
    s.STOCK_TICKER,
    AVG(f.STOCK_CLOSE) AS avg_stock_close
FROM
    fact_stock_data f
    JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
GROUP BY
    s.STOCK_TICKER
ORDER BY
    avg_stock_close DESC;

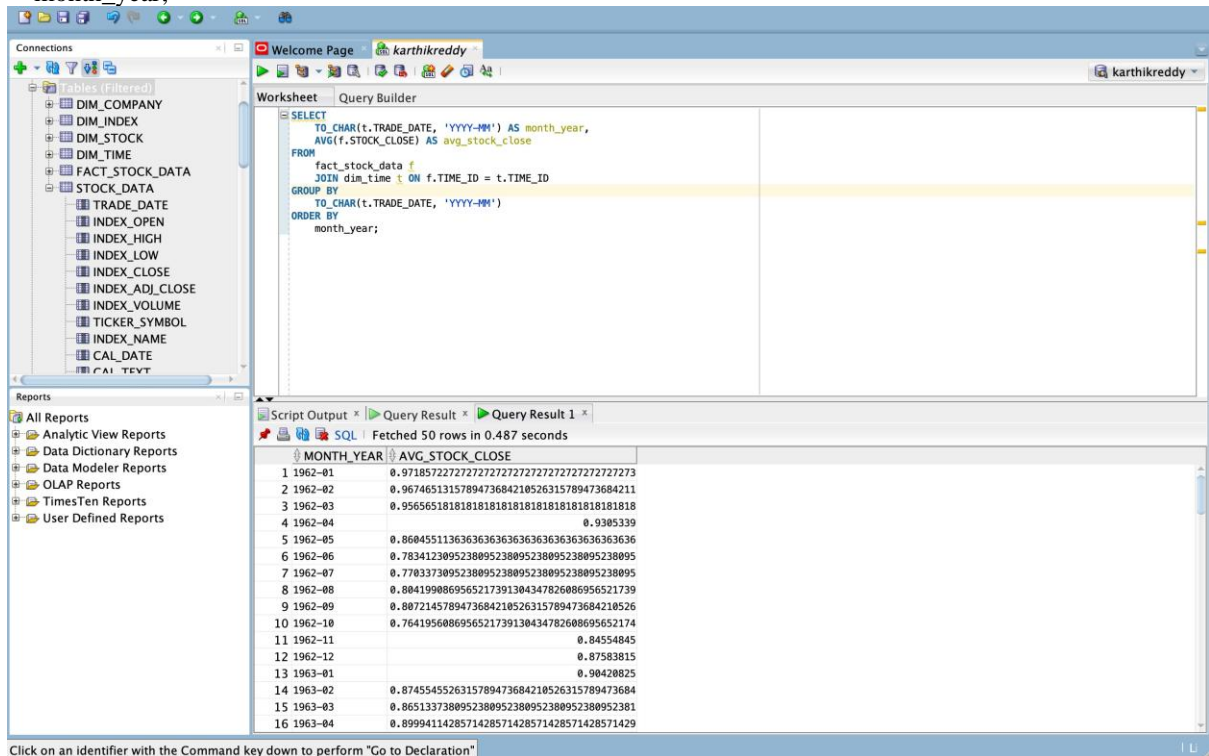
```

**Query Result:**

STOCK_TICKER	avg_stock_close
1 GOOGL	777.058355717472118959107806691449814126
2 GOOG	265.868233827002477291494632535094962841
3 CLN	60.28339280780653372931692829868476877387
4 BZNE	48.87161403896103896103896103896103896104
5 CAT	35.95737510304603555351099135086525810797
6 WMT	35.50594822765896161878015410095773025131
7 TGT	32.33775695287614040513375599195917736199
8 LTXC	22.75238302389185790631876768311851618988
9 KO	19.20422503837939079288929780172838694838
10 COCP	14.32530901217228464419475655430711610487
11 XCRA	8.4288003837715938902111324376199616123
12 CLSN	8.29875785458879618593563766388557806913
13 CYMA	7.94239361702127659574460805186382978723
14 CBAY	5.473130859375
15 CTEKD	3.50254702194357366771159874608150470219
16 CTEK	3.46923001215805471124620060790273556231

- **Monthly Average Stock Prices Over Years:**

```
SELECT
  TO_CHAR(t.TRADE_DATE, 'YYYY-MM') AS month_year,
  AVG(f.STOCK_CLOSE) AS avg_stock_close
FROM
  fact_stock_data f
  JOIN dim_time t ON f.TIME_ID = t.TIME_ID
GROUP BY
  TO_CHAR(t.TRADE_DATE, 'YYYY-MM')
ORDER BY
  month_year;
```



The screenshot shows a SQL query editor with a table of results. The table has two columns: MONTH\_YEAR and AVG\_STOCK\_CLOSE. The data is sorted by MONTH\_YEAR in ascending order.

MONTH_YEAR	AVG_STOCK_CLOSE
1 1962-01	0.97185722727272727272727272727273
2 1962-02	0.9674651315789473684210526315789473684211
3 1962-03	0.9565651818181818181818181818181818
4 1962-04	0.9385339
5 1962-05	0.86845511363636363636363636363636
6 1962-06	0.7834123895238095238095238095238095
7 1962-07	0.7783373895238095238095238095238095
8 1962-08	0.804199869565217391304347826086956521739
9 1962-09	0.8072145789473684210526315789473684210526
10 1962-10	0.7641956086956521739130434782608695652174
11 1962-11	0.84554845
12 1962-12	0.87583815
13 1963-01	0.90428825
14 1963-02	0.8745545526315789473684210526315789473684
15 1963-03	0.8651337380952380952380952380952381
16 1963-04	0.8999411428571428571428571428571428571429

- **Average Daily Trading Volume by Stock Ticker:**

```
SELECT
  s.STOCK_TICKER,
  AVG(f.STOCK_VOLUME) AS avg_stock_volume
FROM
  fact_stock_data f
  JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
GROUP BY
  s.STOCK_TICKER
ORDER BY
  avg_stock_volume DESC;
```

The screenshot shows the SQL Developer interface with a query in the Worksheet pane. The query selects the stock ticker and the average stock volume from the fact\_stock\_data table, joined with the dim\_stock table. The results are ordered by average stock volume in descending order.

```

SELECT
  s.STOCK_TICKER,
  AVG(f.STOCK_VOLUME) AS avg_stock_volume
FROM
  fact_stock_data f
  JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
GROUP BY
  s.STOCK_TICKER
ORDER BY
  avg_stock_volume DESC;

```

The Query Result pane shows 16 rows of data. The first few rows are:

STOCK_TICKER	AVG_STOCK_VOLUME
1 KO	10458970.9492984589897361416882988607761
2 GOOG	10249427.12634186622625928984310487200661
3 WMT	8477586.40815158068517038316122992727811
4 TGT	4897539.14182365857430029379928869645895
5 CAT	3989111.28380376564913087766569184766536
6 GOOGL	1956249.4423791821561338289962825788184
7 XCRA	428167.6583493282149712092138518234165067
8 CBAY	412963.28125
9 CLSN	408817.3992848629328619785458879618593564
10 LTXN	201264.7235819823056541869744464903220012
11 CTEK	23582.1681864235055724417426545806119554
12 COCP	17543.7265917682996254681647940674906367
13 BZNE	8866.60482574708889853883395176252919109
14 CLN	5783.98326864658464149342384386932541366
15 CTEK	4143.495297805642633228848125391849529781
16 CMA	687.23484255319148936178212765957446885

- Count of Records by Year and Month:

```

SELECT
  TO_CHAR(t.TRADE_DATE, 'YYYY') AS year,
  TO_CHAR(t.TRADE_DATE, 'MM') AS month,
  COUNT(*) AS num_records
FROM
  fact_stock_data f
  JOIN dim_time t ON f.TIME_ID = t.TIME_ID
GROUP BY
  TO_CHAR(t.TRADE_DATE, 'YYYY'),
  TO_CHAR(t.TRADE_DATE, 'MM')
ORDER BY
  year, month;

```

The screenshot shows the SQL Developer interface with a query in the Worksheet pane. The query selects the year and month from the dim\_time table, joined with the fact\_stock\_data table. The results are grouped by year and month, and ordered by year and month.

```

SELECT
  TO_CHAR(t.TRADE_DATE, 'YYYY') AS year,
  TO_CHAR(t.TRADE_DATE, 'MM') AS month,
  COUNT(*) AS num_records
FROM
  fact_stock_data f
  JOIN dim_time t ON f.TIME_ID = t.TIME_ID
GROUP BY
  TO_CHAR(t.TRADE_DATE, 'YYYY'),
  TO_CHAR(t.TRADE_DATE, 'MM')
ORDER BY
  year, month;

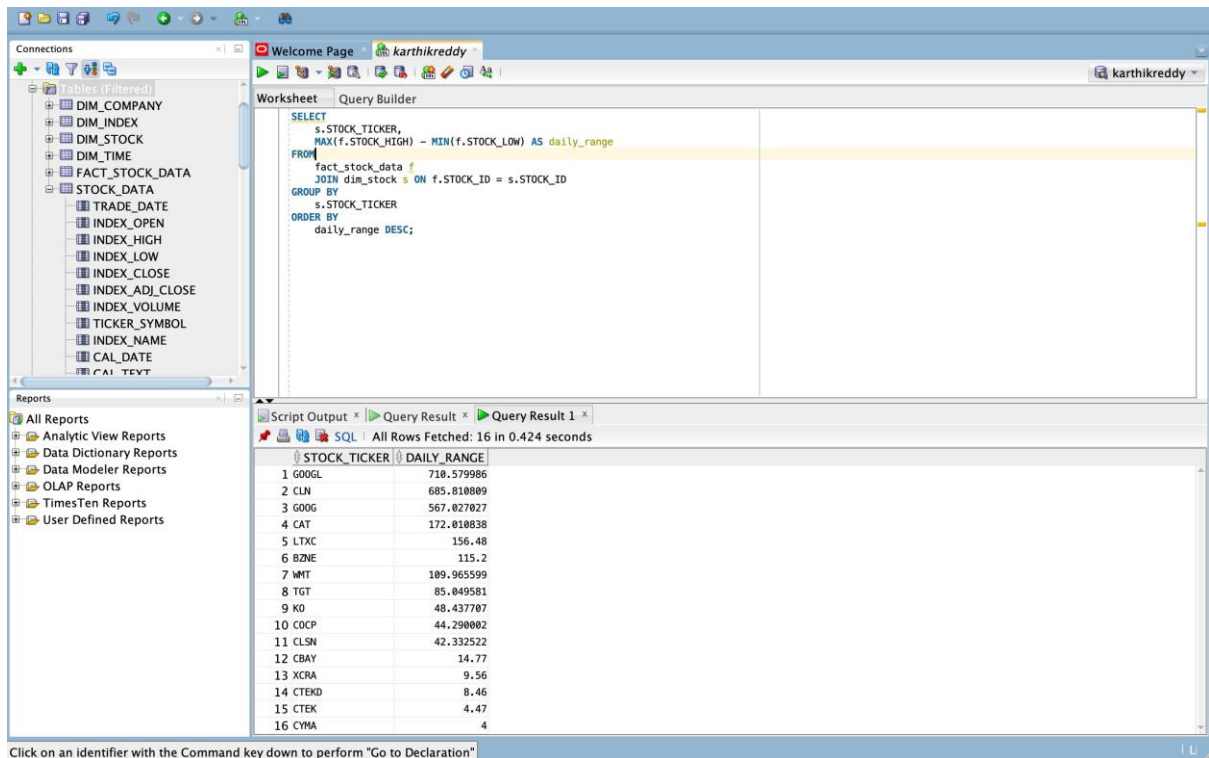
```

The Query Result pane shows 50 rows of data. The first few rows are:

YEAR	MONTH	NUM_RECORDS
1 1962	01	44
2 1962	02	38
3 1962	03	44
4 1962	04	40
5 1962	05	44
6 1962	06	42
7 1962	07	42
8 1962	08	46
9 1962	09	38
10 1962	10	46
11 1962	11	40
12 1962	12	40
13 1963	01	44
14 1963	02	38
15 1963	03	42
16 1963	04	42

- **Daily High-Low Range by Stock Ticker:**

```
SELECT
    s.STOCK_TICKER,
    MAX(f.STOCK_HIGH) - MIN(f.STOCK_LOW) AS daily_range
FROM
    fact_stock_data f
    JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
GROUP BY
    s.STOCK_TICKER
ORDER BY
    daily_range DESC;
```



The screenshot shows a BI tool interface with a 'Connections' pane on the left, a 'Query Builder' in the center, and a 'Query Result' pane at the bottom. The 'Query Builder' contains the following SQL query:

```
SELECT
    s.STOCK_TICKER,
    MAX(f.STOCK_HIGH) - MIN(f.STOCK_LOW) AS daily_range
FROM
    fact_stock_data f
    JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
GROUP BY
    s.STOCK_TICKER
ORDER BY
    daily_range DESC;
```

The 'Query Result' pane displays the results of the query, showing 16 rows of stock tickers and their corresponding daily ranges. The results are as follows:

STOCK_TICKER	DAILY_RANGE
1 GOOGL	710.579986
2 CLN	685.810089
3 GOOG	567.027027
4 CAT	172.010838
5 LTXC	156.48
6 BZNE	115.2
7 WMT	109.965599
8 TGT	85.049581
9 KO	48.437707
10 COCP	44.290002
11 CLSN	42.332522
12 CBAY	14.77
13 XCRA	9.56
14 CTEKD	8.46
15 CTEK	4.47
16 CYMA	4

- **Monthly Total Trading Volume:**

```
SELECT
    TO_CHAR(t.TRADE_DATE, 'YYYY-MM') AS month_year,
    SUM(f.STOCK_VOLUME) AS total_volume
FROM
    fact_stock_data f
    JOIN dim_time t ON f.TIME_ID = t.TIME_ID
GROUP BY
    TO_CHAR(t.TRADE_DATE, 'YYYY-MM')
ORDER BY
    month_year;
```



The screenshot shows a BI tool interface with a 'Tables (Filtered)' pane on the left, a 'Query Builder' pane in the center, and a 'Query Result' pane at the bottom. The 'Tables (Filtered)' pane lists various tables including DIM\_COMPANY, DIM\_INDEX, DIM\_STOCK, DIM\_TIME, FACT\_STOCK\_DATA, STOCK\_DATA, TRADE\_DATE, INDEX\_OPEN, INDEX\_HIGH, INDEX\_LOW, INDEX\_CLOSE, INDEX\_ADJ\_CLOSE, INDEX\_VOLUME, TICKER\_SYMBOL, INDEX\_NAME, CAL\_DATE, and CAL\_TEXT. The 'Query Builder' pane contains the following SQL query:

```
SELECT
  TO_CHAR(t.TRADE_DATE, 'YYYY-MM') AS month_year,
  SUM(f.STOCK_VOLUME) AS total_volume
FROM
  fact_stock_data f
  JOIN dim_time t ON f.TIME_ID = t.TIME_ID
GROUP BY
  TO_CHAR(t.TRADE_DATE, 'YYYY-MM')
ORDER BY
  month_year;
```

The 'Query Result' pane displays the following data:

MONTH_YEAR	TOTAL_VOLUME
1 1962-01	34185600
2 1962-02	15460800
3 1962-03	45487200
4 1962-04	23354400
5 1962-05	52785600
6 1962-06	54276000
7 1962-07	27561600
8 1962-08	24019200
9 1962-09	18408000
10 1962-10	31315200
11 1962-11	34905600
12 1962-12	26296800
13 1963-01	30076800
14 1963-02	25305600
15 1963-03	20824800
16 1963-04	32088000

- **Average Closing Price by Day of the Week:**

```
SELECT
  t.DAY_NAME,
  AVG(f.STOCK_CLOSE) AS avg_stock_close
FROM
  fact_stock_data f
  JOIN dim_time t ON f.TIME_ID = t.TIME_ID
GROUP BY t.DAY_NAME
ORDER BY
  CASE t.DAY_NAME
    WHEN 'Monday' THEN 1
    WHEN 'Tuesday' THEN 2
    WHEN 'Wednesday' THEN 3
    WHEN 'Thursday' THEN 4
    WHEN 'Friday' THEN 5
    WHEN 'Saturday' THEN 6
    WHEN 'Sunday' THEN 7
  END;
```

The screenshot shows a BI tool interface with a 'Tables (Filtered)' pane on the left, a 'Query Builder' pane in the center, and a 'Query Result' pane at the bottom. The 'Tables (Filtered)' pane lists various tables including DIM\_COMPANY, DIM\_INDEX, DIM\_STOCK, DIM\_TIME, FACT\_STOCK\_DATA, STOCK\_DATA, TRADE\_DATE, INDEX\_OPEN, INDEX\_HIGH, INDEX\_LOW, INDEX\_CLOSE, INDEX\_ADJ\_CLOSE, INDEX\_VOLUME, TICKER\_SYMBOL, INDEX\_NAME, CAL\_DATE, and CAL\_TEXT. The 'Query Builder' pane contains the following SQL query:

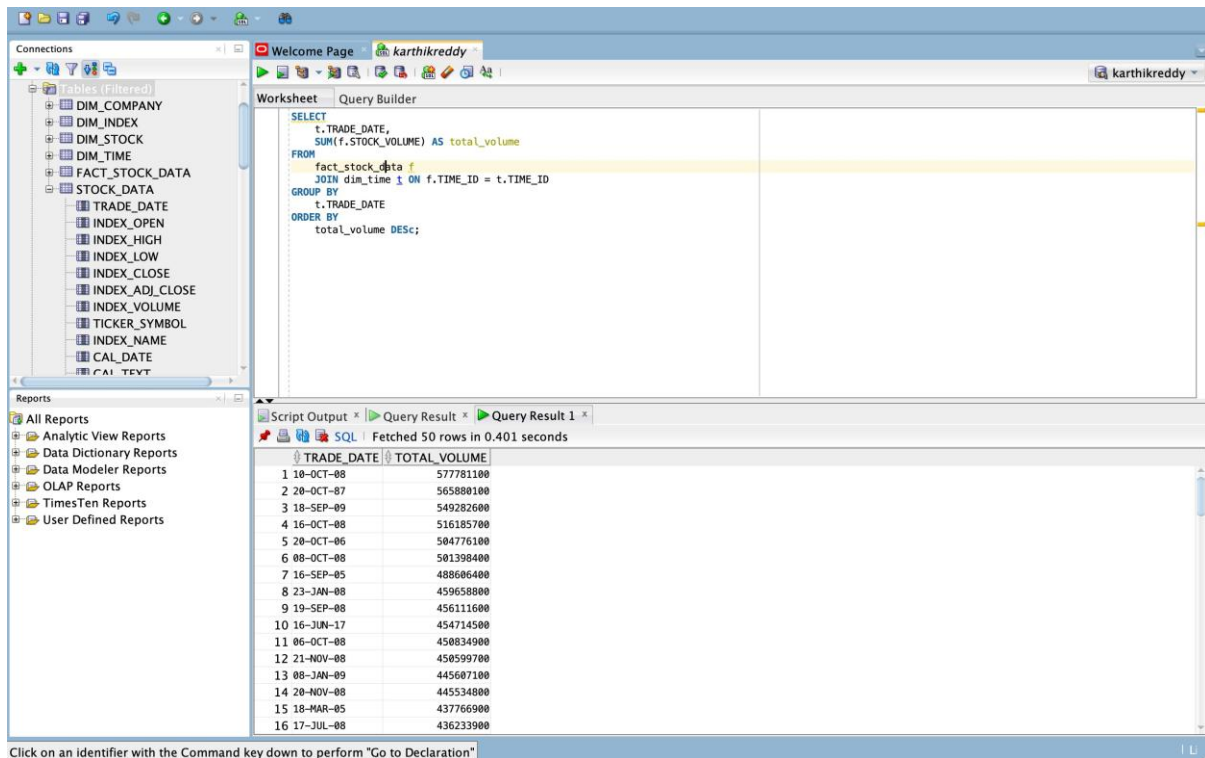
```
SELECT
  t.DAY_NAME,
  AVG(f.STOCK_CLOSE) AS avg_stock_close
FROM
  fact_stock_data f
  JOIN dim_time t ON f.TIME_ID = t.TIME_ID
GROUP BY
  t.DAY_NAME
ORDER BY
  CASE t.DAY_NAME
    WHEN 'Monday' THEN 1
    WHEN 'Tuesday' THEN 2
    WHEN 'Wednesday' THEN 3
    WHEN 'Thursday' THEN 4
    WHEN 'Friday' THEN 5
    WHEN 'Saturday' THEN 6
    WHEN 'Sunday' THEN 7
  END;
```

The 'Query Result' pane displays the following data:

DAY_NAME	AVG_STOCK_CLOSE
1 THURSDAY	49.861848888487633463981619137721313691
2 TUESDAY	49.7992346579338535860274990789773318469
3 FRIDAY	49.8058796935009797517962116263879817113
4 MONDAY	49.3415751373627956555587565981299299934
5 WEDNESDAY	49.91352351007831975587103622130821281677

- **Highest Trading Days by Volume:**

```
SELECT
    t.TRADE_DATE,
    SUM(f.STOCK_VOLUME) AS total_volume
FROM
    fact_stock_data f
    JOIN dim_time t ON f.TIME_ID = t.TIME_ID
GROUP BY
    t.TRADE_DATE
ORDER BY
    total_volume DESC
LIMIT 10;
```



The screenshot shows a BI tool interface with a 'Connections' pane on the left, a 'Query Builder' pane in the center, and a 'Query Result' pane at the bottom. The 'Query Builder' pane contains the following SQL query:

```
SELECT
    t.TRADE_DATE,
    SUM(f.STOCK_VOLUME) AS total_volume
FROM
    fact_stock_data f
    JOIN dim_time t ON f.TIME_ID = t.TIME_ID
GROUP BY
    t.TRADE_DATE
ORDER BY
    total_volume DESC;
```

The 'Query Result' pane shows the results of the query, which are 50 rows. The first 16 rows are displayed in the table below:

TRADE_DATE	TOTAL_VOLUME
1 10-OCT-08	577781100
2 20-OCT-07	565800100
3 18-SEP-09	549282600
4 16-OCT-08	516185700
5 20-OCT-06	504776100
6 08-OCT-08	501398400
7 16-SEP-05	488606400
8 23-JAN-08	459658800
9 19-SEP-08	456111600
10 16-JUN-17	454714500
11 06-OCT-08	450834900
12 21-NOV-08	450599700
13 08-JAN-09	445607100
14 20-NOV-08	445534800
15 18-MAR-05	437766900
16 17-JUL-08	436233900

- **Monthly Average Adjusted Closing Price for a Specific Stock:**

```
SELECT
    TO_CHAR(t.TRADE_DATE, 'YYYY-MM') AS month_year,
    AVG(f.STOCK_ADJ_CLOSE) AS avg_adj_close_price
FROM
    fact_stock_data f
    JOIN dim_time t ON f.TIME_ID = t.TIME_ID
    JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
WHERE
    s.STOCK_TICKER = 'AAPL' -- Example: Replace with specific stock ticker
GROUP BY
    TO_CHAR(t.TRADE_DATE, 'YYYY-MM')
ORDER BY
    month_year;
```

The screenshot shows a data warehouse interface with a 'Tables (Filtered)' pane on the left containing tables like DIM\_COMPANY, DIM\_INDEX, DIM\_STOCK, DIM\_TIME, FACT\_STOCK\_DATA, and STOCK\_DATA. The 'Query Builder' pane shows a query to calculate the average adjusted close price by month and year. The 'Query Result' pane displays the following data:

MONTH_YEAR	AVG_ADJ_CLOSE_PRICE
1 1962-01	0.0782735
2 1962-02	0.0782465526315789473684210526315789473684
3 1962-03	0.07719334090909090909090909090909090909
4 1962-04	0.0749892
5 1962-05	0.069486295454545454545454545454545455
6 1962-06	0.0635530952380952380952380952380952380952
7 1962-07	0.0623330476190476190476190476190476190476
8 1962-08	0.0652533695652173913043478260869565217391
9 1962-09	0.0658132894736842105263157894736842105263
10 1962-10	0.0624245869565217391304347826086956521739
11 1962-11	0.0699232
12 1962-12	0.072309025
13 1963-01	0.074914272727272727272727272727272727
14 1963-02	0.0719715263157894736842105263157894736842
15 1963-03	0.0710715238095238095238095238095238095238
16 1963-04	0.0744965952380952380952380952380952380952

- Weekly High and Low Prices for All Stocks:

```

SELECT
  TO_CHAR(t.TRADE_DATE, 'TW') AS week_number,
  TO_CHAR(t.TRADE_DATE, 'YYYY') AS year,
  s.STOCK_TICKER,
  MAX(f.STOCK_HIGH) AS weekly_high,
  MIN(f.STOCK_LOW) AS weekly_low
FROM
  fact_stock_data f
  JOIN dim_time t ON f.TIME_ID = t.TIME_ID
  JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
GROUP BY
  TO_CHAR(t.TRADE_DATE, 'TW'),
  TO_CHAR(t.TRADE_DATE, 'YYYY'),
  s.STOCK_TICKER
ORDER BY
  year, week_number, s.STOCK_TICKER;

```

The screenshot shows the same data warehouse interface with a query to retrieve weekly high and low prices for all stocks. The 'Query Result' pane displays the following data:

WEEK_NUMBER	YEAR	STOCK_TICKER	WEEKLY_HIGH	WEEKLY_LOW
1 01	1962	CAT	1.788333	1.526042
2 01	1962	KO	0.270182	0.221354
3 02	1962	CAT	1.739583	1.666667
4 02	1962	KO	0.268091	0.245768
5 03	1962	CAT	1.75	1.677883
6 03	1962	KO	0.25651	0.236328
7 04	1962	CAT	1.778833	1.666667
8 04	1962	KO	0.247396	0.233873
9 05	1962	CAT	1.75	1.692788
10 05	1962	KO	0.248698	0.233398
11 06	1962	CAT	1.744792	1.692788
12 06	1962	KO	0.247396	0.242839
13 07	1962	CAT	1.783125	1.635417
14 07	1962	KO	0.248047	0.239583
15 08	1962	CAT	1.692788	1.625
16 08	1962	KO	0.244141	0.238281



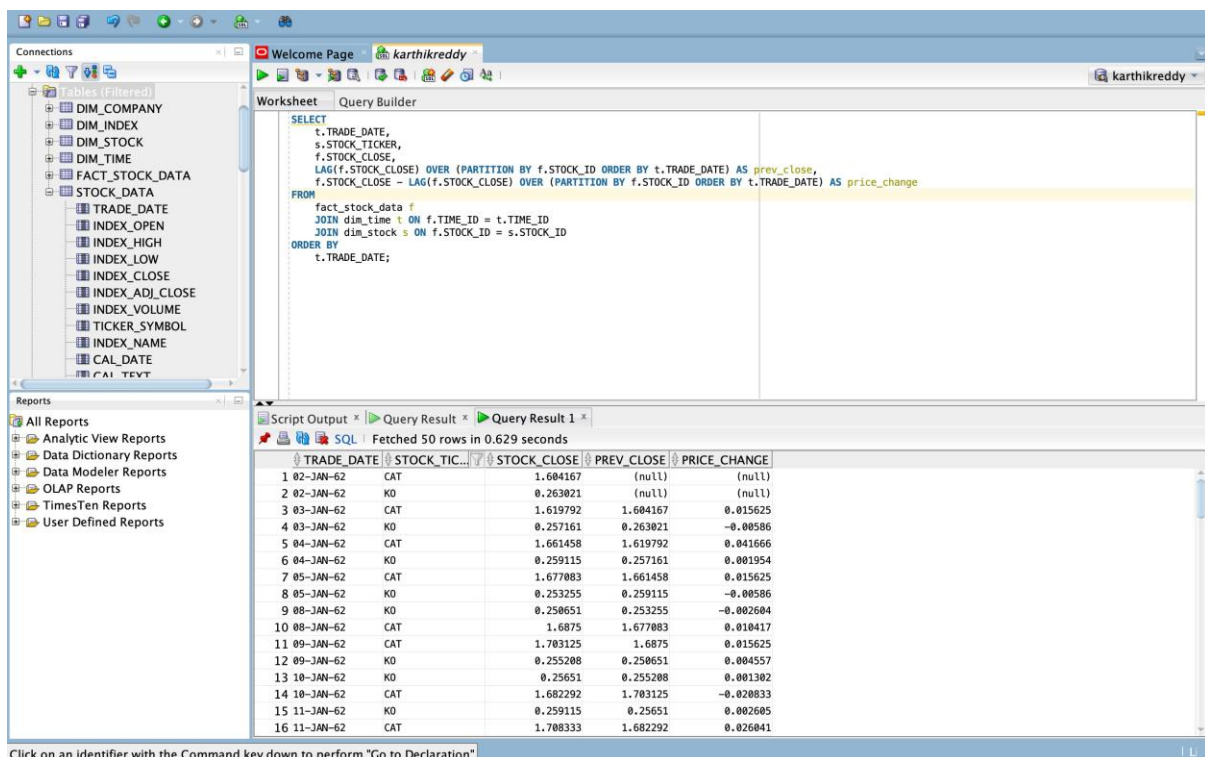
## 1. LAG and LEAD Functions

- **LAG:** Retrieves data from a previous row in the result set.
- **LEAD:** Retrieves data from a subsequent row in the result set.

### Example: Calculate the Change in Stock Price Over Time

```
SELECT
    t.TRADE_DATE,
    s.STOCK_TICKER,
    f.STOCK_CLOSE,
    LAG(f.STOCK_CLOSE) OVER (PARTITION BY f.STOCK_ID ORDER BY t.TRADE_DATE) AS
prev_close,
    f.STOCK_CLOSE - LAG(f.STOCK_CLOSE) OVER (PARTITION BY f.STOCK_ID ORDER BY
t.TRADE_DATE) AS price_change
FROM
    fact_stock_data f
    JOIN dim_time t ON f.TIME_ID = t.TIME_ID
    JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
ORDER BY
    t.TRADE_DATE;
```

This query calculates the change in stock price (price\_change) compared to the previous trading day (LAG(f.STOCK\_CLOSE)).



The screenshot shows a BI tool interface with a 'Query Builder' tab. The SQL query is displayed in the main area, and the 'Query Result' tab shows the output. The query is the same as the one in the example above. The result set contains 16 rows of data, showing the trade date, stock ticker, stock close, previous close, and price change.

TRADE_DATE	STOCK_TIC...	STOCK_CLOSE	PREV_CLOSE	PRICE_CHANGE
1 02-JAN-62	CAT	1.604167	(null)	(null)
2 02-JAN-62	KO	0.263021	(null)	(null)
3 03-JAN-62	CAT	1.619792	1.604167	0.015625
4 03-JAN-62	KO	0.257161	0.263021	-0.00586
5 04-JAN-62	CAT	1.661458	1.619792	0.041666
6 04-JAN-62	KO	0.259115	0.257161	0.001954
7 05-JAN-62	CAT	1.677083	1.661458	0.015625
8 05-JAN-62	KO	0.253255	0.259115	-0.00586
9 08-JAN-62	KO	0.250651	0.253255	-0.002604
10 08-JAN-62	CAT	1.6875	1.677083	0.010417
11 09-JAN-62	CAT	1.703125	1.6875	0.015625
12 09-JAN-62	KO	0.255208	0.250651	0.004557
13 10-JAN-62	KO	0.25651	0.255208	0.001302
14 10-JAN-62	CAT	1.682292	1.703125	-0.020833
15 11-JAN-62	KO	0.259115	0.25651	0.002605
16 11-JAN-62	CAT	1.708333	1.682292	0.026041

## 2. NTILE Function

- **NTILE:** Divides the result set into a specified number of groups or "buckets".

### Example: Assign Quartiles Based on Stock Closing Price

```
SELECT
    t.TRADE_DATE,
    s.STOCK_TICKER,
    f.STOCK_CLOSE,
```

```

NTILE(4) OVER (PARTITION BY s.STOCK_TICKER ORDER BY f.STOCK_CLOSE) AS quartile
FROM
fact_stock_data f
JOIN dim_time t ON f.TIME_ID = t.TIME_ID
JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
ORDER BY
s.STOCK_TICKER, f.STOCK_CLOSE;

```

In this query, NTILE(4) partitions the data by STOCK\_TICKER and assigns quartiles based on the STOCK\_CLOSE values.

The screenshot shows a SQL IDE interface with a 'Connections' pane on the left, a 'Query Builder' pane in the center, and a 'Query Result' pane at the bottom. The 'Query Builder' pane contains the following SQL query:

```

SELECT
t.TRADE_DATE,
s.STOCK_TICKER,
f.STOCK_CLOSE,
NTILE(4) OVER (PARTITION BY s.STOCK_TICKER ORDER BY f.STOCK_CLOSE) AS quartile
FROM
fact_stock_data f
JOIN dim_time t ON f.TIME_ID = t.TIME_ID
JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
ORDER BY
s.STOCK_TICKER, f.STOCK_CLOSE;

```

The 'Query Result' pane shows the output of the query, which is a table with 5 columns: TRADE\_DATE, STOCK\_TICKER, STOCK\_CLOSE, and QUARTILE. The table contains 16 rows of data, showing the first 16 rows of the result set.

TRADE_DATE	STOCK_TICKER	STOCK_CLOSE	QUARTILE
1 24-JUN-13	BZNE	6.9	1
2 24-JUN-13	BZNE	6.9	1
3 24-JUN-13	BZNE	6.9	1
4 12-APR-13	BZNE	9.9	1
5 24-APR-13	BZNE	9.9	1
6 15-APR-13	BZNE	9.9	1
7 16-APR-13	BZNE	9.9	1
8 17-APR-13	BZNE	9.9	1
9 18-APR-13	BZNE	9.9	1
10 19-APR-13	BZNE	9.9	1
11 22-APR-13	BZNE	9.9	1
12 23-APR-13	BZNE	9.9	1
13 25-APR-13	BZNE	9.9	1
14 12-APR-13	BZNE	9.9	1
15 15-APR-13	BZNE	9.9	1
16 16-APR-13	BZNE	9.9	1

### 3. CORR Function

- **CORR:** Calculates the correlation coefficient between two numeric columns.

#### Example: Calculate Correlation Between Stock Prices and Volumes

```

SELECT
s.STOCK_TICKER,
CORR(f.STOCK_CLOSE, f.STOCK_VOLUME) AS correlation_stock_close_volume
FROM
fact_stock_data f
JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
GROUP BY
s.STOCK_TICKER;

```

This query computes the correlation coefficient (correlation\_stock\_close\_volume) between STOCK\_CLOSE and STOCK\_VOLUME for each stock ticker.

The screenshot displays a data analytics application with a 'Connections' pane on the left listing various data sources like DIM\_COMPANY, DIM\_INDEX, DIM\_STOCK, DIM\_TIME, FACT\_STOCK\_DATA, and STOCK\_DATA. The main 'Worksheet' area contains a SQL query in the 'Query Builder' tab:

```
SELECT
  s.STOCK_TICKER,
  CORR(f.STOCK_CLOSE, f.STOCK_VOLUME) AS correlation_stock_close_volume
FROM
  fact_stock_data f
JOIN dim_stock s ON f.STOCK_ID = s.STOCK_ID
GROUP BY
  s.STOCK_TICKER;
```

The 'Script Output' pane at the bottom shows the query results, indicating 'All Rows Fetched: 16 in 0.528 seconds'. The results are displayed in a table with two columns: STOCK\_TICKER and CORRELATION\_STOCK\_CLOSE\_VOLUME.

STOCK_TICKER	CORRELATION_STOCK_CLOSE_VOLUME
1 CAT	0.4537851469164819624958251129361433452345
2 CLSN	-0.0942842573324938696346272738038038969342
3 CYMA	-0.1096328536844728850940689347756684558321
4 CBAY	0.288000726000164085167578688939821844555
5 KO	0.3734850269989234718048231417587525835588
6 LTXC	0.0068938741282155155747006699960988556458252
7 TGT	0.2427487382368989895810184106863165238679
8 CTEK	0.0248928315448144154719786268317150791501
9 BZNE	-0.2692979235631687725908800035458666148884
10 GOOGL	-0.127202608121655858060619007733867327011
11 COCP	-0.0781574418350674116308155721068744880092
12 GOOG	-0.4929896122357373865156536064989828188264
13 CTEK	0.3652773632380741181784059216437901605453
14 CLN	0.0516919595337286311933550742315364930089
15 WMT	0.3539133395033483324893716018662584979189
16 XCRA	0.3427000387289016279888126838441404241494

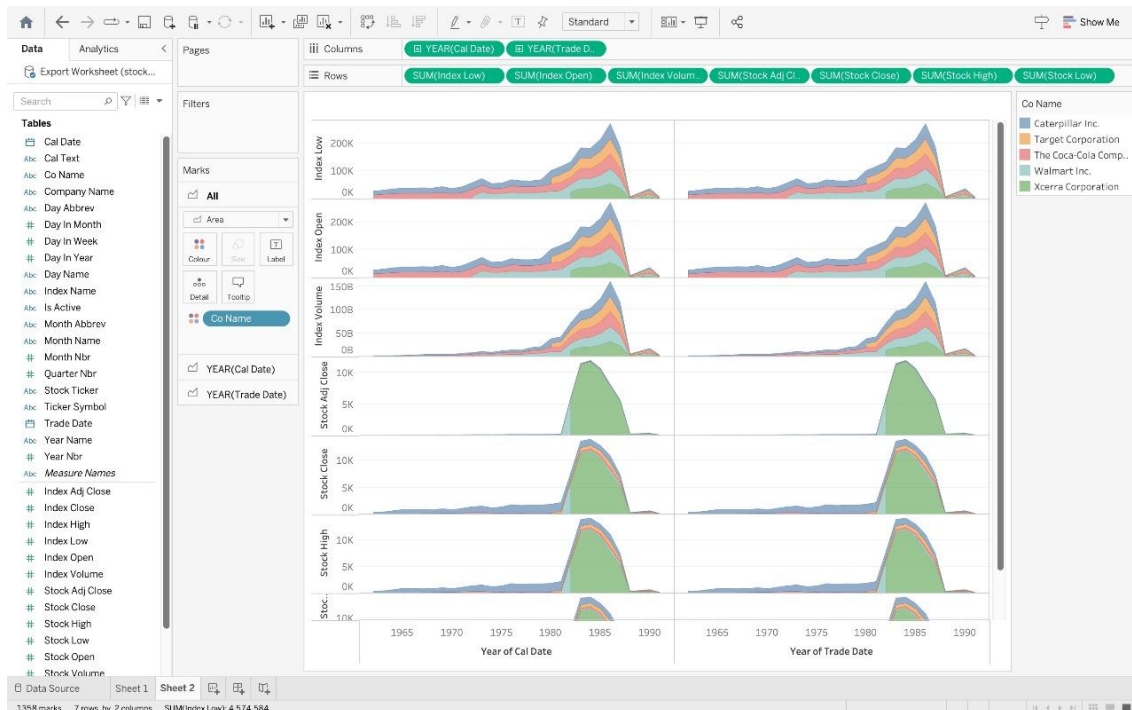
At the bottom of the interface, a note states: 'Click on an identifier with the Command key down to perform "Go to Declaration"'.

## Applications

- **Price Change Analysis:** Use LAG/LEAD to analyze trends and changes over time, such as daily price changes or sequential movements in stock data.
- **Segmentation and Ranking:** NTILE can help categorize data into groups based on specified criteria, useful for quartiles, percentiles, or other segmentations.
- **Correlation Analysis:** CORR facilitates understanding relationships between variables, aiding in identifying factors influencing stock performance.

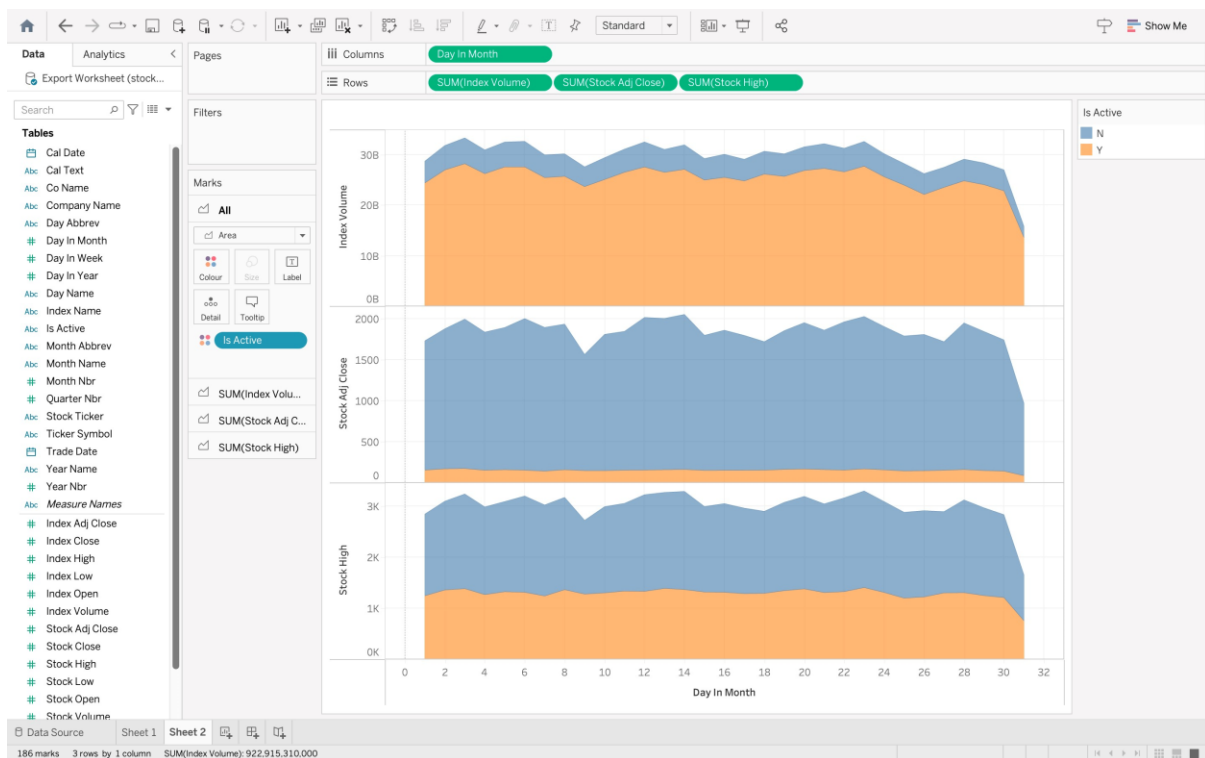
# Data Visualization

## a) Stock Performance Over Time:



This visualization showcases the trends in various stock metrics, including Index Low, Index Open, Index Volume, Stock Adjusted Close, Stock Close, Stock High, and Stock Low.

## b) Stock Market Performance by Day in Month

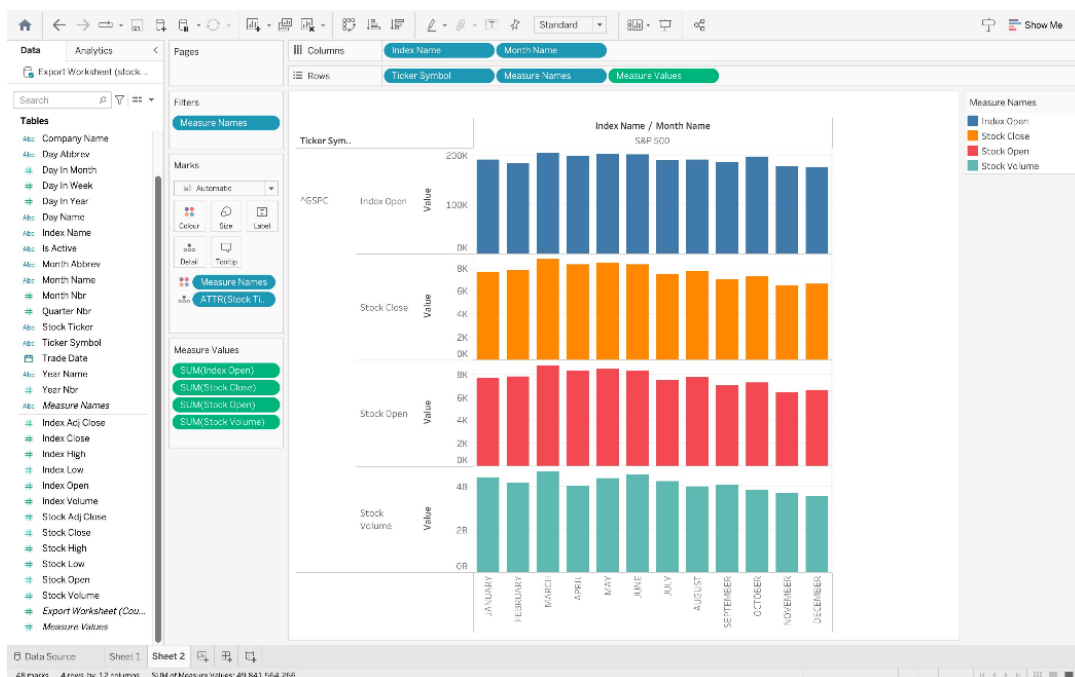


- To further analyze the stock market data, an area chart was created using Tableau to visualize the volume, adjusted closing prices, and high prices of stocks over the days in a month. This visualization helps in understanding the daily performance and activity levels in the stock market.

- The area chart of stock market performance by day in the month is a valuable tool for visualizing and analyzing daily market activities. It aids investors and analysts in understanding trading patterns, market stability, and the level of participation, thereby supporting informed decision-making in stock investments.

### c) Monthly Performance of S&P 500

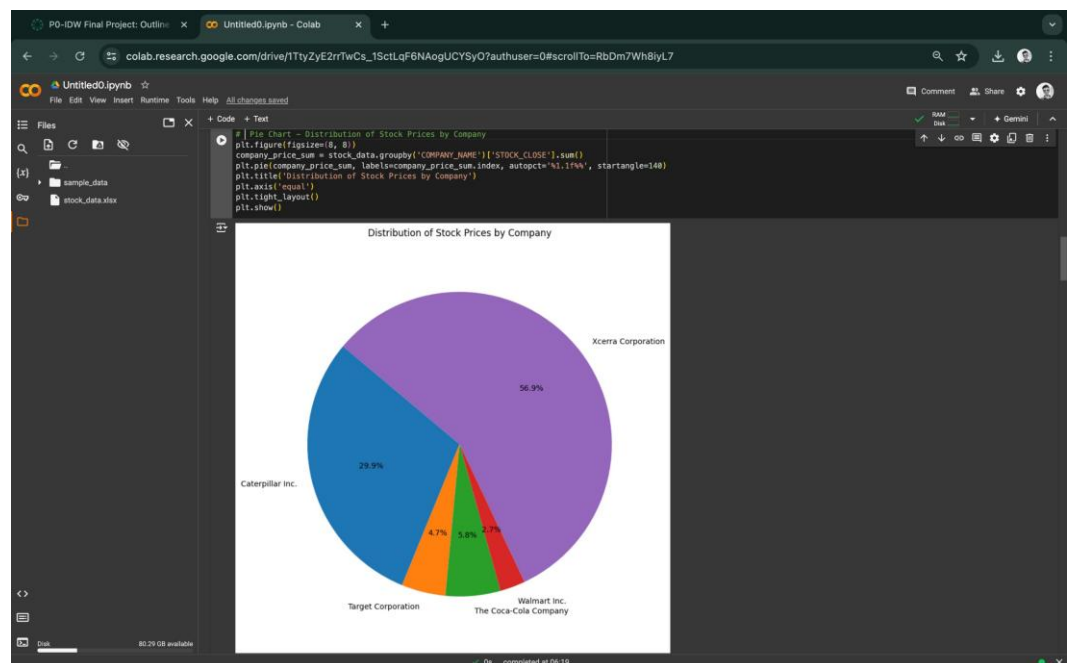
- To analyze the monthly performance of the S&P 500 index, a bar chart was created using Tableau. This visualization helps in comparing different measures such as index open, stock close, stock open, and stock volume across the months.
- This chart is useful for investors and analysts in understanding the stability of opening and closing prices, as well as the fluctuations in trading volumes, thereby supporting better financial decisions and strategies.



### d) Distribution of Stock Prices by Company

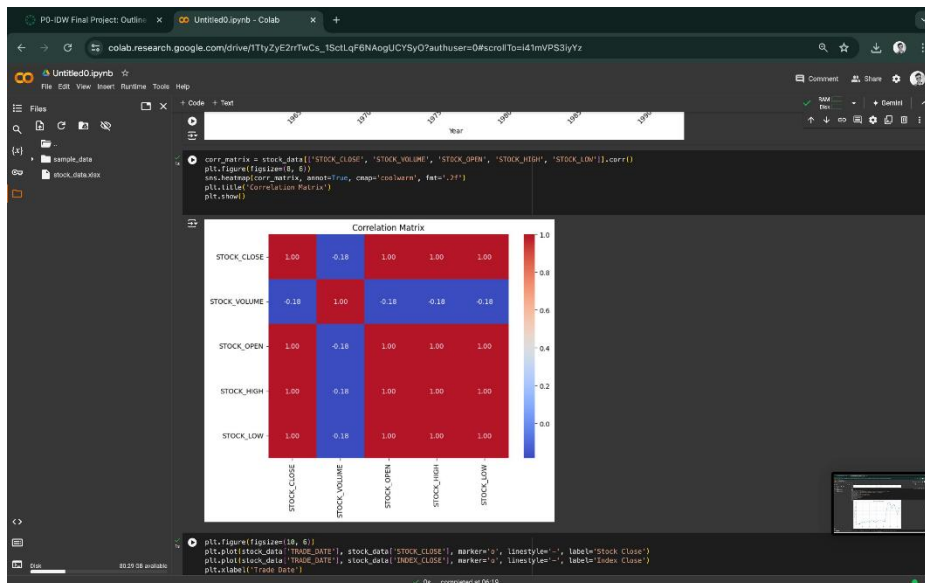
- To understand the distribution of stock prices among different companies, a pie chart was created using Python. This visualization helps in identifying the proportion of stock prices contributed by each company in the dataset.

The pie chart illustrates the distribution of stock prices across various companies, providing a clear visual representation of each company's share in the total stock prices.



## e) Correlation Matrix of Stock Variables

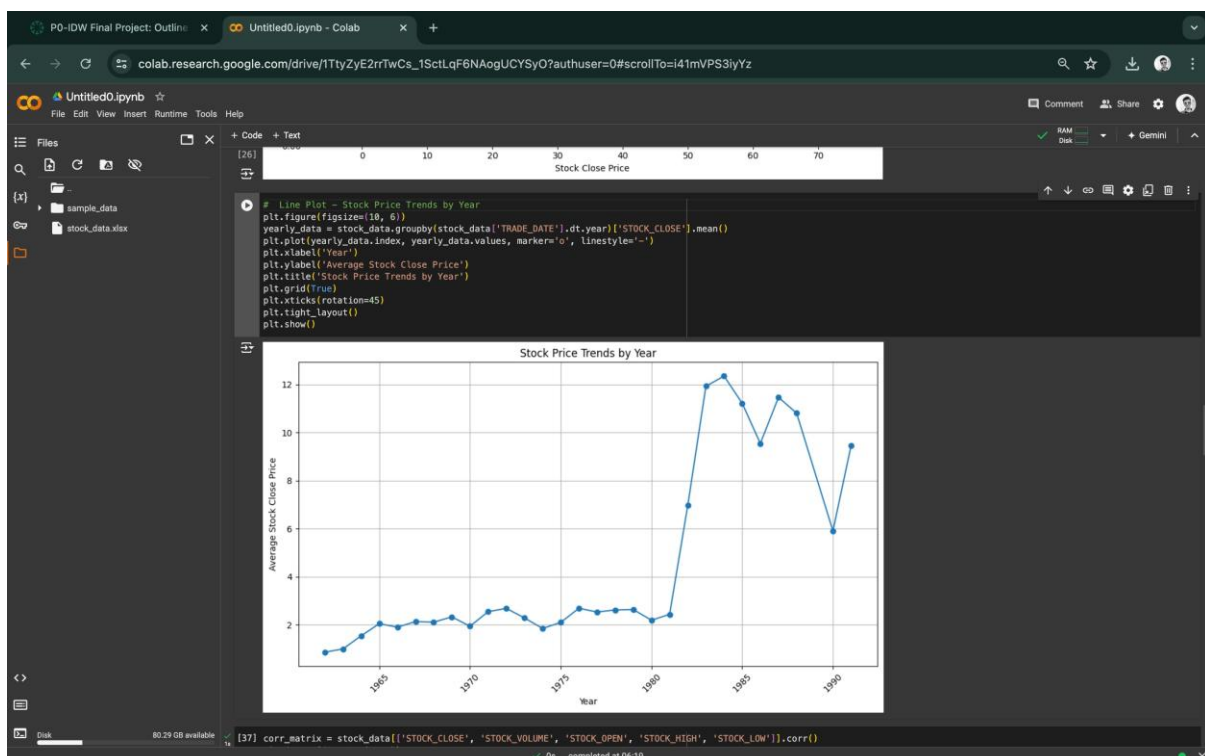
- To analyse the relationships between different stock variables, a correlation matrix was created using Python. This visualization helps in understanding how various stock metrics are correlated with each other.



- The heatmap illustrates the correlation matrix of stock close prices, volumes, open prices, high prices, and low prices, providing a visual representation of the strength and direction of correlations.

## f) Stock Price Trends by Year

- To analyze the trends in stock prices over the years, a line plot was created using Python. This visualization helps in understanding the changes in average stock close prices over a span of several years.

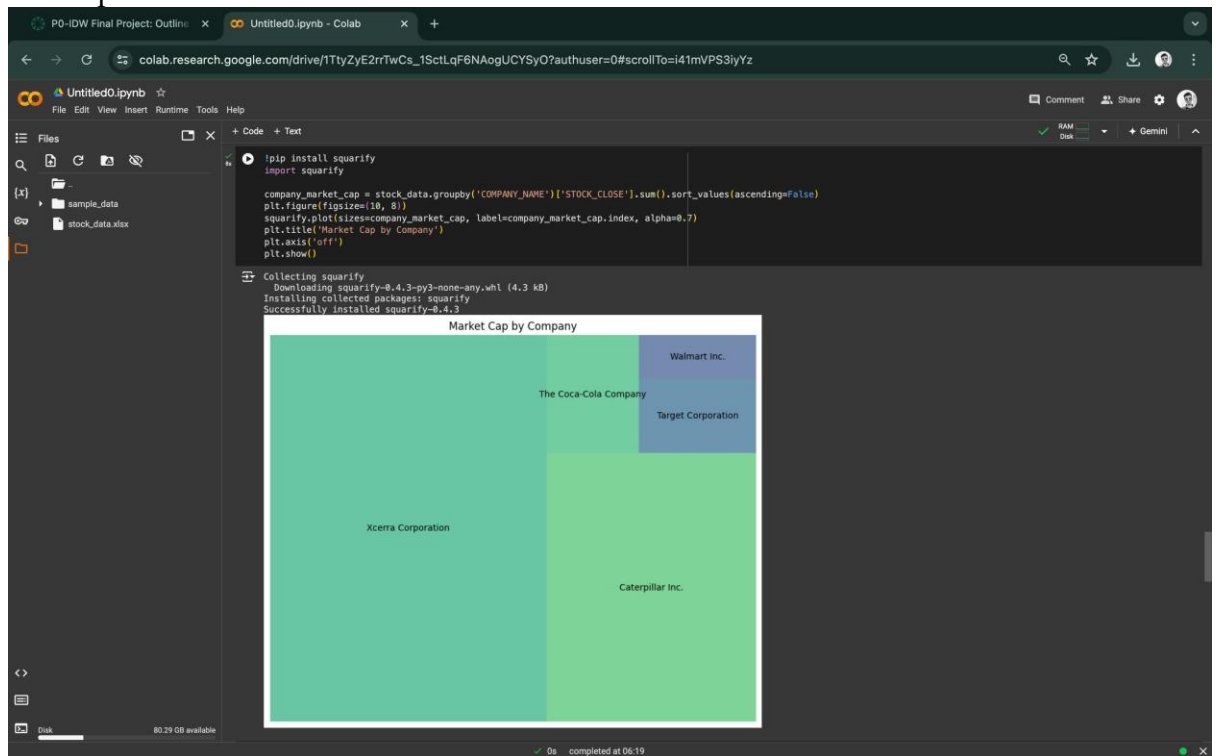




- The line plot illustrates the average stock close prices by year, providing a visual representation of how stock prices have trended over time.

### g) Market Cap by Company

- To visualize the market capitalization of different companies, a treemap was created using Python. This visualization helps in understanding the relative market cap sizes of companies within the dataset.



- The treemap illustrates the market capitalization of various companies, providing a clear and proportional visual representation of each company's market cap.

### References:

1. <https://www.kimballgroup.com/data-warehouse-business-intelligence-resources/kimball-techniques/dimensional-modeling-techniques/>
2. <https://www.vertabelo.com/blog/a-beginners-guide-to-data-modeling/>
3. <https://chartio.com/learn/data-visualization/what-is-data-visualization/>