# Project Report - Milestone 2

Krishnanshu Jain 2019CS10368
Prakhar Jagwani 2019CS10382
Souvagya Ranjan Sethi 2019CS10405

18 April, 2023

## Contents

**1 Database Schema** 3

**2 Queries** 5
- 2.1 Insert and Update Queries . . . 5
  - 2.1.1 Insert / Update User Details . . . 5
  - 2.1.2 Insert / Update Portfolio Details . . . 5
- 2.2 Technical Indicators . . . 6
  - 2.2.1 CAGR . . . 6
  - 2.2.2 Volatility Vs Index . . . 7
  - 2.2.3 Sharpe Ratio . . . 7
- 2.3 Portfolio Comparisons . . . 8
  - 2.3.1 SIP return . . . 8
  - 2.3.2 Percentage Comparison between 2 stocks . . . 10
- 2.4 Portfolio Valuation . . . 10
- 2.5 Portfolio Optimizations . . . 11
- 2.6 Back-Testing . . . 13
  - 2.6.1 Function . . . 13
  - 2.6.2 Query . . . 15
- 2.7 Moving Average . . . 16
  - 2.7.1 Simple Moving Average . . . 17
  - 2.7.2 Exponential Moving Average . . . 18
- 2.8 Forecasting . . . 18
  - 2.8.1 Using Linear-Regression Model for stock_price prediction . . . 18
- 2.9 Search options . . . 19
  - 2.9.1 Show complete price history of a stock . . . 19
  - 2.9.2 Search on Name and ticker via prefix . . . 19
  - 2.9.3 Search for Competing ETFs . . . 19
  - 2.9.4 Search for Competing Stocks . . . 20
- 2.10 Filters . . . 20

**3 Query Optimization** 21
- 3.1 Materialized View . . . 21
- 3.2 Indexes . . . 21

**4 Database Size and Performance** **21**
4.1 Database Size . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
4.2 Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

# 1 Database Schema

```
CREATE TABLE securities(
    ticker VARCHAR (25) NOT NULL,
    type VARCHAR(5) check(type in ('stock', 'etf', 'index')),
    PRIMARY KEY (ticker)
);

CREATE TABLE stock (
    ticker VARCHAR(25) NOT NULL, -- check lenght for ticker
    name VARCHAR(255),
    sector VARCHAR(255),
    market_cap float,
    pe_ration float,
    PRIMARY KEY (ticker),
    CONSTRAINT stock_securities FOREIGN KEY (ticker) REFERENCES
        securities(ticker) ON DELETE CASCADE INITIALLY DEFERRED
);

CREATE TABLE etf (
    ticker VARCHAR(25) NOT NULL, -- check lenght for ticker
    underlying_asset VARCHAR(255),
    PRIMARY KEY (ticker),
    CONSTRAINT etf_securities FOREIGN KEY (ticker) REFERENCES
        securities(ticker) ON DELETE CASCADE INITIALLY DEFERRED
);

CREATE TABLE index (
    ticker VARCHAR(25) NOT NULL, -- check lenght for ticker
    name VARCHAR(255) Unique,
    PRIMARY KEY (ticker),
    CONSTRAINT index_securities FOREIGN KEY (ticker) REFERENCES
        securities(ticker) ON DELETE CASCADE INITIALLY DEFERRED
);

CREATE TABLE constituent(
    security_ticker VARCHAR(25) NOT NULL,
    index_ticker VARCHAR(25) NOT NULL,
    PRIMARY KEY (security_ticker, index_ticker),
    CONSTRAINT security_ticker_stock FOREIGN KEY (security_ticker)
        REFERENCES stock(ticker) ON DELETE CASCADE,
    CONSTRAINT index_ticker_index FOREIGN KEY (index_ticker) REFERENCES
        index(ticker) ON DELETE CASCADE
);

CREATE TABLE daily_price (
    ticker VARCHAR(25) NOT NULL,
    date DATE NOT NULL,
    open_price float,
    close_price float,
    avg_price float,
    PRIMARY KEY (ticker, date),
```

```sql
    CONSTRAINT price_securities FOREIGN KEY (ticker) REFERENCES
        securities(ticker) ON DELETE CASCADE
);

CREATE TABLE mutual_fund (
    id VARCHAR(255) NOT NULL,
    name VARCHAR(255),
    amc  VARCHAR(50), -- check type
    category VARCHAR(50),
    risk_type VARCHAR(50) check(risk_type in ('Very High Risk', 'Moderately
        High Risk', 'High Risk', 'Moderate Risk', 'Low to Moderate Risk',
        'Moderately Low Risk', 'Low Risk')),
    PRIMARY KEY (id)
);

CREATE TABLE daily_nav (
    mf_id VARCHAR(255) NOT NULL,
    date DATE NOT NULL,
    nav float,
    PRIMARY KEY (mf_id, date),
    CONSTRAINT id_mutual_fund FOREIGN KEY (mf_id) REFERENCES
        mutual_fund(id) ON DELETE CASCADE
);

CREATE TABLE users(
    id SERIAL,
    name VARCHAR(255),
    email VARCHAR(255),
    password VARCHAR(255),
    PRIMARY KEY (id)
);

CREATE TABLE user_has_mf (
    user_id int NOT NULL,
    mf_id VARCHAR(255) NOT NULL,
    quantity int,
    PRIMARY KEY (user_id, mf_id),
    CONSTRAINT user_id_user FOREIGN KEY (user_id) REFERENCES users(id) ON
        DELETE CASCADE,
    CONSTRAINT mf_id_mutual_fund FOREIGN KEY (mf_id) REFERENCES
        mutual_fund(id) ON DELETE CASCADE
);

CREATE TABLE user_has_security (
    user_id int NOT NULL,
    ticker VARCHAR(25) NOT NULL,
    quantity int,
    PRIMARY KEY (user_id, ticker),
    CONSTRAINT user_id_user FOREIGN KEY (user_id) REFERENCES users(id) ON
        DELETE CASCADE,
    CONSTRAINT user_securities FOREIGN KEY (ticker) REFERENCES
        securities(ticker) ON DELETE CASCADE
```

```sql
);

create table weights(
    name VARCHAR(25) NOT NULL,
    weight1 float,
    weight2 float,
    weight3 float,
    weight4 float,
    weight5 float,
    weight6 float,
    weight7 float,
    weight8 float,
    weight9 float,
    weight10 float,
    PRIMARY KEY (name)
);
```

# 2 Queries

## 2.1 Insert and Update Queries

### 2.1.1 Insert / Update User Details

The User enters his/her details from the frontend and his/her data is added to the database.

```sql
-- insert
INSERT INTO users (name, email, password) VALUES ('John Doe',
    'johndoe@example.com', 'mypassword');
-- update
UPDATE users
SET name = 'Jane Doe', password = 'newpassword'
WHERE email = 'johndoe@eamxple.com';
-- delete
DELETE FROM users
WHERE email = 'johndoe@eamxple.com';
```

### 2.1.2 Insert / Update Portfolio Details

The User enters his/her details from the frontend and this portfolio is updated into the database.

```sql
-- insert
insert into user_has_mf (user_id, mf_id, quantity) values (1, 'direct
    growth small uti', 100);
insert into user_has_security (user_id, ticker, quantity) values (1,
    'ADANIENT', 100);
-- update
UPDATE user_has_security
SET quantity = 20
WHERE user_id = 1 AND ticker = 'RELIANCE';
-- delete
DELETE FROM user_has_security
WHERE user_id = 1 AND ticker = 'RELIANCE';
```

## 2.2 Technical Indicators

### 2.2.1 CAGR

CAGR stands for Compound Annual Growth Rate. It is a measure of the average annual growth rate of an investment over a certain period of time, taking into account the effect of compounding. To calculate the CAGR, you need to know the beginning value, ending value, and the number of years of the investment. The formula for calculating CAGR is:

$$CAGR = (EndingValue/BeginningValue)^{(1/NumberofYears)} - 1$$

CAGR is a useful metric for comparing the performance of investments over different time periods and can help investors determine the long-term growth potential of an investment.
The below Query calculates CAGR for all the securities for 1,2 and 5 years.

```sql
create materialized view if not exists cagr as
with date_1 as (
  select date
  from daily_price
  where date > get_current_date() - interval '1 year'
  order by date asc
  limit 1
),
date_2 as (
  select date
  from daily_price
  where date > get_current_date() - interval '2 year'
  order by date asc
  limit 1
),
date_5 as (
  select date
  from daily_price
  where date > get_current_date() - interval '5 year'
  order by date asc
  limit 1
),
prices_1 as (
  select ticker,avg_price
  from daily_price
  where date = (select date from date_1)
),
prices_2 as (
  select ticker,avg_price
  from daily_price
  where date = (select date from date_2)
),
prices_5 as (
  select ticker,avg_price
  from daily_price
  where date = (select date from date_5)
),
prices as(
```

```
  select daily_price.ticker,daily_price.avg_price as
      price,prices_1.avg_price as price_1,prices_2.avg_price as
      price_2,prices_5.avg_price as price_5
  from daily_price full outer join prices_1 on
      daily_price.ticker=prices_1.ticker full outer join prices_2 on
      daily_price.ticker=prices_2.ticker full outer join prices_5 on
      daily_price.ticker=prices_5.ticker
  where date = get_current_date()
)
  select ticker,((price/price_1)^(1/1)-1)*100 as
      cagr_1,((price/price_2)^(1/2.0)-1)*100 as
      cagr_2,((price/price_5)^(1/5.0)-1)*100 as cagr_5
  from prices;
```

### 2.2.2 Volatility Vs Index

Volatility is a statistical measure that quantifies the degree of variation of an asset's returns over time. It is commonly used as a measure of risk, as assets with higher volatility are considered riskier due to their larger potential fluctuations in value.

Volatility is often calculated as the standard deviation of an asset's returns over a certain period of time, typically expressed as an annualized percentage. This is calculated using historical returns data and represents the variability of returns around the average return for the period. Volatility Vs Index is just the ratio of the volatility of security and the Volatility of the Index.

The below Query is used to calculate annualized volatility over 3 years for every security. One thing to note here is that we are calculating volatitly vs Nifty 50 index.

```
create materialized view if not exists volatility as
with table1 as (
  select d1.ticker,(d1.close_price-d1.open_price)/d1.close_price AS p_change
  from daily_price d1
  where (d1.date > get_current_date() - interval '3 year')
),
table2 as(
  select table1.ticker, (stddev(p_change)) as volatility
  from table1
  group by table1.ticker
)
select p1.ticker, p1.volatility/p2.volatility AS volatility_ratio
from table2 as p1,table2 as p2
where p2.ticker='NIFTY_50';
```

### 2.2.3 Sharpe Ratio

The Sharpe ratio is used to evaluate the performance of an individual stock. The Sharpe ratio indicates how well an security performs in comparison to the rate of return on a risk-free investment, such as Fixed deposits in banks in India.
Here, we are calculating the Sharpe Ratio for a given stock by first extracting monthly average prices for the past year, calculating the monthly returns, and then computing the Sharpe Ratio using the

formula:

$$SharpeRatio = (AverageMonthlyReturn - RiskFreeRate)/(StandardDeviationofMonthlyReturn)$$

where the average monthly return and the standard deviation of monthly return are calculated for the past year of data. The risk-free rate used in this calculation is assumed to be 7%.
The below query is used to Calculate the Sharpe ratio over 2 years for each security.

```
create materialized view if not exists sharpe_ratio as
with monthly_dates as(
  select DISTINCT ON (month,year) EXTRACT(MONTH FROM date) as month,
      EXTRACT(YEAR FROM date) as year,  date
  from daily_price
  where date>get_current_date() - interval '2 year'
  Order by month,year,date asc
),
daily_price_t as (
    select monthly_dates.date,month,year,ticker,avg_price
    from daily_price,monthly_dates
    where daily_price.date=monthly_dates.date
),
daily_price_ret_montly as(
    select p1.ticker,(p2.avg_price/p1.avg_price-1)*100 as ret,p2.date
    from daily_price_t as p1,daily_price_t as p2
    where p1.ticker=p2.ticker
    and ((p1.month=p2.month-1 and p1.year=p2.year) OR (p1.month=12 and
        p2.month=1 and p1.year=p2.year-1))
)
select ticker,(avg(ret)-7.0)/(24.0^(0.5)*stddev(ret)) as sharpe_ration
from daily_price_ret_montly
group by ticker;
```

## 2.3 Portfolio Comparisons

### 2.3.1 SIP return

SIP (Systematic Investment Plan) return is the return earned by an investor who invests a fixed amount of money at regular intervals (such as monthly) in a mutual fund or other investment product. SIP return is calculated based on the total amount invested, the time period of investment, and the net asset value (NAV) of the investment product. SIP return provides a way to track the performance of an investment over time and compare it to other investment options.

To Calculate SIP for a security over a time period we have used a recursive Query, where at the start of each month we are investing a fixed amount defined by the user and calculating current value.

```
with recursive stock_table as (
  select
    date,
    avg_price,
    ROW_NUMBER() over (
      order by
        date asc
```

```
    ) as rank
  from
    daily_price
  where
    ticker = 'RELIANCE'
    AND date > '2019-01-01'
    ),
table1 as (
  select
    date,
    avg_price,
    rank,
    1000 as amount,
    cast(1000 as double precision) as total
  from
    stock_table
  where
    rank = 1
  union all
  select
    d2.date,
    d2.avg_price,
    d2.rank,
    d1.amount + case when (
      extract(
        month
        from
          d1.date
      ) != extract(
        month
        from
          d2.date
      )
    ) then 1000 else 0 end as amount,
    d1.total * d2.avg_price / d1.avg_price + case when (
      extract(
        month
        from
          d1.date
      ) != extract(
        month
        from
          d2.date
      )
    ) then 1000 else 0 end as total
  from
    table1 d1,
    stock_table d2
  where
    (d2.rank = d1.rank + 1)
    AND (d1.date < get_current_date())
)
```

```sql
select
  date,
  amount as investment,
  total as value,
  (
    (total - amount) / amount * 100
  ) as return
from
  table1;
```

### 2.3.2 Percentage Comparison between 2 stocks

Here we calculate the percentage change in the average price of a stock (RELIANCE) and an index (NIFTY_50) over time. It helps us to analyse the change in a stock price with respect to a given index.

```sql
with table1 as (
    select ticker, date, avg_price from daily_price
    where ticker = 'RELIANCE'
    and date >= '2019-01-01'
), table2 as (
    select * from daily_price
    where ticker = 'NIFTY_50'
    and date >= '2019-01-01'
), table3 as (
  select t1.date, t1.avg_price as avg_price_stock, t2.avg_price as
     avg_price_index
  from table1 t1 join table2 t2
  on t1.date = t2.date
)
select date,
100 * (avg_price_stock - base_price_stock)/base_price_stock as
   percentage_change_stock,
100 * (avg_price_index - base_price_index)/base_price_index as
   percentage_change_index
from table3, (
  select avg_price_stock as base_price_stock, avg_price_index as
     base_price_index
  from table3
  where date = '2019-01-01'
) base
order by date;
```

## 2.4 Portfolio Valuation

A user on our website should be able to see the current value of their investments.

```sql
with mf_portfolio as (
  select
    sum(quantity * daily_nav.nav) as value
  from
    user_has_mf, daily_nav
```

```sql
  where
    user_id = 1
    and user_has_mf.mf_id = daily_nav.mf_id
    and daily_nav.date = '2021-01-01'
),
security_portfolio as (
  select
    sum(quantity * daily_price.close_price) as value
  from
    user_has_security, daily_price
  where
    user_id = 1
    and user_has_security.ticker = daily_price.ticker
    and daily_price.date = '2021-01-01'
)
select mf_portfolio.value as mf_v, security_portfolio.value as security_v,
    mf_portfolio.value + security_portfolio.value as value from
    mf_portfolio, security_portfolio;
```

### 2.5 Portfolio Optimizations

The main goal of portfolio optimization is to find the optimal allocation of assets in a portfolio that provides the best possible risk-reward trade-off. Here to compute the risk-reward trade-off we have used Information Ration(I.R.).

$$IR = \frac{AVG(MonthlyPortfolioReturn - MonthlyIndexReturn)}{STDDEV(MonthlyPortfolioReturn - MonthlyIndexReturn)}$$

Here, we have randomly generated portfolios and calculated their expected returns and risks. We select the portfolio with the highest information ratio as the optimal portfolio.

```sql
with recursive monthly_dates as(
  select DISTINCT ON (month,year) EXTRACT(MONTH FROM date) as month,
      EXTRACT(YEAR FROM date) as year,  date
  from daily_price
  where date>'2019-01-01'::date
  Order by month,year,date asc
),
daily_price_t as (
    select monthly_dates.date,ticker,avg_price
    from daily_price,monthly_dates
    where daily_price.date=monthly_dates.date
    AND ticker in ('RELIANCE','TCS','HDFCBANK','NIFTY_50')
),
p1 as(
    select date,avg_price
    from daily_price_t
    where ticker='RELIANCE'
),
p2 as(
    select date,avg_price
    from daily_price_t
```

```sql
    where ticker='TCS'
),
p3 as(
    select date,avg_price
    from daily_price_t
    where ticker='HDFCBANK'
),
pindex as(
    select date,avg_price
    from daily_price_t
    where ticker='NIFTY_50'
),
final_prices as(
  select p1.avg_price as s1_price,p2.avg_price as s2_price, p3.avg_price as
      s3_price,pindex.avg_price as index_price,ROW_NUMBER() over (order by
      p1.date asc) as rank
  from pindex,p1,p2,p3
  where p1.date=p2.date
  and p1.date=p3.date
  and p1.date=pindex.date
),
final_returns as(
  select (p2.s1_price/p1.s1_price-1)*100 as s1_ret,
      (p2.s2_price/p1.s2_price -1)*100 as s2_ret,(p2.s3_price/p1.s3_price
      -1)*100 as s3_ret,(p2.index_price/p1.index_price -1)*100 as
      ind_ret,p2.rank
  from final_prices as p1,final_prices as p2
  where p1.rank=p2.rank-1
),
--recursivly genrate random tuples
random_tuples as(
  select random() as r1,random() as r2,random() as r3,0 as cnt
  union all
  select random() as r1,random() as r2,random() as r3,(random_tuples.cnt+1)
      as cnt
  from random_tuples
  where  random_tuples.cnt<10000
),
normalized_tuples as(
  select r1/(r1+r2+r3) as w1,r2/(r1+r2+r3) as w2,r3/(r1+r2+r3) as w3,cnt
  from random_tuples
),
portfolio_diff as(
  select w1*s1_ret+w2*s2_ret+w3*s3_ret-ind_ret as portfolio_diff,cnt,rank
  from normalized_tuples,final_returns
),
portfolio_IR as(
  select avg(portfolio_diff) / stddev(portfolio_diff) as IR,cnt
  from portfolio_diff
  group by cnt
),
best_portfolio as(
```

```
  select cnt,IR
  from portfolio_IR
  where IR=(select max(IR) from portfolio_IR)
),
weights_best_portfolio as(
  select normalized_tuples.w1, normalized_tuples.w2, normalized_tuples.w3,
      best_portfolio.IR
  from normalized_tuples, best_portfolio
  where normalized_tuples.cnt=best_portfolio.cnt
)
select * from weights_best_portfolio;
```

## 2.6 Back-Testing

We have written a simple framework for backtesting. It allows users to define their own trading rules based on price movements and buying/selling signals. Users can set specific conditions for when to buy or sell a stock, such as when the price rises or falls to a certain value, or when it goes above or below the average buy price. The framework then tests these rules on historical market data to see how well they would have performed in the past, allowing users to evaluate the effectiveness of their trading strategy.
We have defined some functions and types to make it easy to determine if a signal activates.

### 2.6.1 Function

```
CREATE type backtesting_rule AS (
  value decimal,                    -- exact price of a stock
  daily_price_change integer,       -- percent change in a day
  avg_price_diff integer,           -- difference from average buy price
  month_change boolean,             -- whether a new month has begun
  year_change boolean,              -- whether a new year has begun
  quantity integer,                 -- quantity to buy/sell when this
                                    -- signal activates
  comparator text,                  -- >=, <=, == comparison with any of
                                    -- the first four columns
  action text                       -- BUY/SELL/KEEP
);

CREATE type backtesting_row AS (
  cash decimal,
  investment integer,
  date_t date,
  ticker_price decimal,
  watch_price decimal,
  avg_price decimal,
  rank bigint                       -- number of trading days before this
      date
);

CREATE type backtesting_input AS (
  date_t date,
  ticker_price decimal,             -- price of the ticker on this date
```

```
  watch_price decimal,                  -- can watch NIFTY index of the ticker
      invested in
  rank bigint
);

-- when the rule is triggered, the function returns true
CREATE OR REPLACE FUNCTION signal_activated(r1 backtesting_row, r2
    backtesting_input, rule backtesting_rule) RETURNS boolean LANGUAGE
    plpgsql AS $$
BEGIN
  IF rule.action = 'BUY' AND r1.cash < rule.quantity * r2.ticker_price THEN
    RETURN FALSE;
  END IF;
  IF rule.action = 'SELL' AND r1.investment < rule.quantity THEN
    RETURN FALSE;
  END IF;
  IF rule.month_change THEN
    IF extract(month from r1.date_t) != extract(month from r2.date_t) THEN
      RETURN TRUE;
    END IF;
  END IF;
  IF rule.year_change THEN
    IF extract(year from r1.date_t) != extract(year from r2.date_t) THEN
      RETURN TRUE;
    END IF;
  END IF;
  IF rule.comparator = '>=' THEN
    IF r2.watch_price >= rule.value THEN
      RETURN TRUE;
    END IF;
    IF (r2.watch_price - r1.watch_price) / r1.watch_price * 100 >=
        rule.daily_price_change THEN
      RETURN TRUE;
    END IF;
    IF (r2.ticker_price - r1.avg_price) / r1.avg_price * 100 >=
        rule.avg_price_diff THEN
      RETURN TRUE;
    END IF;
  ELSIF rule.comparator = '<=' THEN
    IF r2.watch_price <= rule.value THEN
      RETURN TRUE;
    END IF;
    IF (r2.watch_price - r1.watch_price) / r1.watch_price * 100 <=
        rule.daily_price_change THEN
      RETURN TRUE;
    END IF;
    IF (r2.ticker_price - r1.avg_price) / r1.avg_price * 100 <=
        rule.avg_price_diff THEN
      RETURN TRUE;
    END IF;
  ELSIF rule.comparator = '==' THEN
    IF r2.watch_price = rule.value THEN
```

```
      RETURN TRUE;
    END IF;
    IF (r2.watch_price - r1.watch_price) / r1.watch_price * 100 =
        rule.daily_price_change THEN
      RETURN TRUE;
    END IF;
    IF (r2.ticker_price - r1.avg_price) / r1.avg_price * 100 =
        rule.avg_price_diff THEN
      RETURN TRUE;
    END IF;
  END IF;
  IF rule.action = 'KEEP' THEN
    RETURN TRUE;
  END IF;
  RETURN FALSE;
END;
$$;
```

### 2.6.2 Query

The output for this query will be a time series, showing how the users portfolio and the ticker being watched changed over time.

```
with recursive rules as (
  -- SIP every month
  select
    null::decimal as value,
    null::integer as daily_price_change, null::integer as avg_price_diff,
        true::boolean as month_change, null::boolean as year_change,
        10::integer as quantity, '==' as comparator, 'BUY' as action
  union all
  -- Book profits when price >= 1.05 x avg_buy_price
  select
    null as value,
    null as daily_price_change, 5 as avg_price_diff, null as month_change,
        null as year_change, 10 as quantity, '>=' as comparator, 'SELL' as
        action
  union all
  -- Buy on dips when price <= 0.99 x avg_buy_price, to recude the
      avg_buy_price
  select
    null as value,
    null as daily_price_change, -1 as avg_price_diff, null as month_change,
        null as year_change, 10 as quantity,'<=' as comparator, 'BUY' as
        action
  union all
  -- Default
  select
    null as value,
    null as daily_price_change, null as avg_price_diff, null as
        month_change, null as year_change, 0 as quantity,'>=' as
        comparator, 'KEEP' as action
```

```
),
stock_table as (
  select
    d1.date,
    d1.avg_price::decimal as ticker_price,
    d2.avg_price::decimal as watch_price,
    ROW_NUMBER() over (
      order by
        d1.date asc
    ) as rank
  from
    daily_price d1, daily_price d2
  where d1.ticker = 'RELIANCE' AND d2.ticker = 'RELIANCE' AND d1.date =
     d2.date
    AND d1.date > '2019-01-01'
),
backtesting as (
  SELECT 500000::decimal as cash, 100::integer as investment, date,
      ticker_price::decimal, watch_price::decimal, watch_price::decimal as
      avg_price, rank FROM stock_table
  WHERE rank = 1
  UNION ALL
  SELECT DISTINCT ON(backtesting.date)
    CASE WHEN action = 'BUY' THEN cash - quantity * stock_table.ticker_price
    WHEN action = 'SELL' THEN cash + quantity * stock_table.ticker_price
    ELSE cash END as cash,
    CASE WHEN action = 'BUY' THEN investment  + quantity
    WHEN action = 'SELL' THEN investment - quantity
    ELSE investment END as investment,
    stock_table.date, stock_table.ticker_price,  stock_table.watch_price,
    CASE WHEN action = 'BUY' THEN (investment * avg_price + quantity *
       stock_table.ticker_price) / (investment + quantity)
    WHEN investment = quantity THEN stock_table.ticker_price ELSE avg_price
       END,
    stock_table.rank
  FROM stock_table, backtesting, rules
  WHERE stock_table.rank = backtesting.rank + 1
    AND signal_activated(backtesting, stock_table, rules)
)
select date, cash, investment, cash + investment * ticker_price as value,
    ticker_price, watch_price, avg_price from backtesting;
```

## 2.7 Moving Average

Moving averages are stock indicators commonly used in technical analysis. The reason for calculating the moving average of a stock is to help smooth out the price data by creating a constantly updated average price.

### 2.7.1 Simple Moving Average

We calculate the moving average to show the trend of the stock price. The SMA is calculated by taking the average of the last ’t’ days.

$$SMA_n = \frac{1}{n}\sum_{i=1}^{n} P_i$$

We have used recursive query to compute simple moving average for a given security over a given period of time.

```sql
with recursive view_sma as (
select ticker, date, avg_price, ROW_NUMBER() OVER (PARTITION BY ticker ORDER BY date) as row_num
from daily_price
where ticker = ’RELIANCE’ --replace the arguments
and date >= ’2022-10-01’ --replace the arguments for where to start the forecast
),
table0 as(
  select ticker, date, avg_price
  from daily_price
  where ticker = ’RELIANCE’ --replace the arguments
  and date < ’2022-10-01’ --replace the arguments for where to start the forecast
  order by date desc
  limit 5 -- replace this with the number of days you want to use for the simple moving average period - 1
),
table1 as(
  select ticker, AVG(avg_price) as base_simple_moving_avg
  from table0
  group by ticker
),
sma AS (
  select ticker, date, avg_price, row_num, (select base_simple_moving_avg from table1)::double precision as simple_moving_avg
  from view_sma
  where row_num = 1
  union all
  select dp.ticker, dp.date, dp.avg_price, dp.row_num, (sma.simple_moving_avg*4 + dp.avg_price)/5 as simple_moving_avg
  from view_sma dp
  join sma on sma.ticker = dp.ticker
  and dp.row_num = sma.row_num + 1
)
select ticker, date, avg_price, simple_moving_avg
from sma
where row_num <= (select max(row_num) from view_sma)
and date <= get_current_date();
```

### 2.7.2 Exponential Moving Average

Exponential Moving average keeps track of the change in values of stocks giving more weightage to the recent prices and less weightage to the previous prices of the respective stock. The Exponential Moving Average is calculated using

$$EMA_{today} = Value_{today} * (1 + \frac{smoothing}{1+t}) + EMA_{yesterday} * (1 - \frac{smoothing}{1+t})$$

1. here smoothing refers to the smoothing factor. We have used `smoothing` = 2
2. `t` refers to the period. here t = 20.

We have used a recursive query to compute the exponential moving average for a given security over a given period of time.

```
with recursive view_ema as (
select ticker, date, avg_price, ROW_NUMBER() OVER (PARTITION BY ticker
    ORDER BY date) as row_num
from daily_price
where ticker = 'RELIANCE' --replace the arguments
and date >= '2022-10-01' --replace the arguments for where to start the
    forecast
),
ema AS (
  SELECT ticker, date, avg_price, row_num,
         avg_price::double precision AS ema
  FROM view_ema
  WHERE ticker = 'RELIANCE' --replace the arguments
  AND row_num = 1
  UNION ALL
  SELECT dp.ticker, dp.date, dp.avg_price, dp.row_num,
         ((dp.avg_price - ema.ema) * 2 / (1 + 20)::numeric) + ema.ema as
             ema -- here i have taken 20 as the period
  FROM view_ema dp
  JOIN ema ON ema.ticker = dp.ticker
  AND dp.row_num = ema.row_num + 1
)
SELECT ticker, date, avg_price, ema
FROM ema
where row_num <= (select max(row_num) from view_ema)
and date <= get_current_date();
```

## 2.8 Forecasting

Stock Price Prediction using statistical and machine learning methods helps user discover the future value of company stock and other financial assets traded on an exchange. The entire idea of predicting stock prices is to gain significant profits.

### 2.8.1 Using Linear-Regression Model for stock_price prediction

We have used Linear Regression Model to predict the stock_price value in a particular day. It takes into account the preious values of stock_prices and trains model to find the optimal value of hyper-parameters or weights. In order to implement, this, we have trained our model on the test data to find multiple weights i.e, 10 in our case. The predicted price is calculated using

$$price_{t+1} = \sum_{i=0}^{i=9} price_{t-i} * weight_i$$

We will train the model on the backend and put the weights in the database. By this, we can implement various sorts of models and use them directly for forecasting.

## 2.9 Search options

### 2.9.1 Show complete price history of a stock

We want to show the price history of a stock on its page.

```
SELECT * FROM daily_price
WHERE ticker = 'RELIANCE';
```

### 2.9.2 Search on Name and ticker via prefix

The below query implements prefix search for the search bar implementation.

```
select ticker,name
from stock
where starts_with(name,'A') OR starts_with(ticker, 'A')-- replace the prefix
limit 10;
```

### 2.9.3 Search for Competing ETFs

All ETFs tracks certain Index named as underlying_asset and provides some guarantees above that for example volatility less that 30 percent.
The below query finds all the other ETFs which has the same underlying_asset as the given ETF. It then orders them based on their CAGR value.

```
with comp_etf as (
    select ticker as etf_ticker,underlying_asset
    from etf
    where underlying_asset in (
        select underlying_asset
        from etf
        where ticker = 'EBANK'
    )
    and ticker <> 'EBANK'
)
select ticker,cagr_1,cagr_2,cagr_5,underlying_asset
from comp_etf,cagr
where comp_etf.etf_ticker = cagr.ticker
order by cagr_5 desc, cagr_2 desc, cagr_1 desc
limit 10;
```

### 2.9.4 Search for Competing Stocks

All the stocks have a sector to which they belong.
The below query finds all the other stocks which belong to same sector and orders than by their CAGR values.

```sql
with comp_stocks as (
    select ticker,name,sector
    from stock
    where sector in (
        select sector
        from stock
        where ticker = 'RELIANCE'       -- change to user input
    )
    and ticker <> 'RELIANCE'            -- change to user input
)
select cagr.ticker,name,sector,cagr_1,cagr_2,cagr_5
from comp_stocks,cagr
where comp_stocks.ticker = cagr.ticker
order by cagr_5 desc, cagr_2 desc, cagr_1 desc
limit 10;
```

## 2.10 Filters

The below query implements standard filters which can be used by the user to filter out stocks to his needs.
The below query has all the combinations of filters, but this can be easily modified to apply only a subset of filters.

```sql
select stock.name, stock.ticker, stock.sector, stock.market_cap,
    daily_price.open_price, daily_price.close_price, daily_price.avg_price,
    stock.pe_ration, cagr.cagr_1, cagr.cagr_2, cagr.cagr_5,
    volatility.volatility_ratio, sharpe_ratio.sharpe_ration
from stock,daily_price,cagr,volatility,sharpe_ratio
where stock.ticker='RELIANCE'               -- Condition on stock ticker
and stock.sector='IT'                       -- Condition on stock attributes
and stock.market_cap> 1000000000
and stock.pe_ration< 20
and stock.ticker=daily_price.ticker           -- Condition on daily_price attribute
and daily_price.date=get_current_date()
and daily_price.avg_price>100
and daily_price.open_price>100
and daily_price.close_price>100
and stock.ticker=cagr.ticker                 -- Condition on cagr attribute
and cagr.cagr_1>0.5
and cagr.cagr_2>0.5
and cagr.cagr_5>0.5
and stock.ticker=volatility.ticker           -- Condition on volatility attribute
and volatility.volatility_ratio<0.5
and stock.ticker=sharpe_ratio.ticker         -- Condition on sharpe_ratio attribute
```

```
and sharpe_ratio.sharpe_ration>0.5;
```

# 3 Query Optimization

## 3.1 Materialized View

We have created these materialized view, because they take a lot of time (see performance) and are often used in other tables (eg. filters,search, optimisation):

1. CAGR
2. Volatility
3. Sharpe Ratio

## 3.2 Indexes

We have made indices on the following columns:

1. *stock(name)*, used in search
2. *stock(sector)*, used in finding competing stocks
3. *etf(underlying_asset)*, used in finding competing ETFs

# 4 Database Size and Performance

## 4.1 Database Size

| Table | Number of Rows | Size(KB) |
|---|---|---|
| Constituent | 589 | 16 |
| daily_nav | 1135734 | 61000 |
| daily_price | 4754351 | 174000 |
| etf | 164 | 8 |
| index | 12 | 4 |
| mutual_fund | 1067 | 120 |
| Securities | 2143 | 32 |
| Stock | 2000 | 148 |

## 4.2 Performance

| Query | Time(ms) |
|---|---|
| SIP | 114.9 |
| Comparison | 20.8 |
| Portfolio Optimization | 1824.2 |
| BackTesting | 155.6 |
| Exponential Moving Average | 29.7 |
| Simple Moving Average | 9.4 |
| Linear Regression | 38.6 |
| Competing Stocks | 5.2 |
| Competing ETF | 12.4 |
| User Portfolio | 3.1 |
| Search by name | 3.4 |
| All filters | 3.7 |
| Price History of a Stock | 111.9 |
| CAGR | 1810.7 |
| Volatility | 822 |
| Sharpe | 2861.9 |

We have skipped some very simple and common queries, (eg. get user details) for now but we will use them in our application.