FINAL REPORT
DS3020

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

FINTRACK DATABASE SYSTEM

142301011 Italiya Prisha
142301022 Mayank Tewatia
142301041 Patel Parv

Contents:

1. ER Model
2. Relational Model
3. Relationships
4. Tables
5. Integrity Constraints and General Constraints
6. Integrity constraints and general constraints
7. Functions
    - GetPorfolioValue
    - IsGoalMet
    - GetInvestmentPerformance
    - GetTotalUserReturns
    - CountActiveInvestments
    - get_user_summary
8. Roles
    - Admin
    - Advisor
    - fintrack_user
9. Use of Functions to fetch values on Frontend
10. Triggers
    - trg_check_goal_met
    - trg_log_investment_performance
    - trg_notify_active_investments
    - Trg_recalculate_user_returns

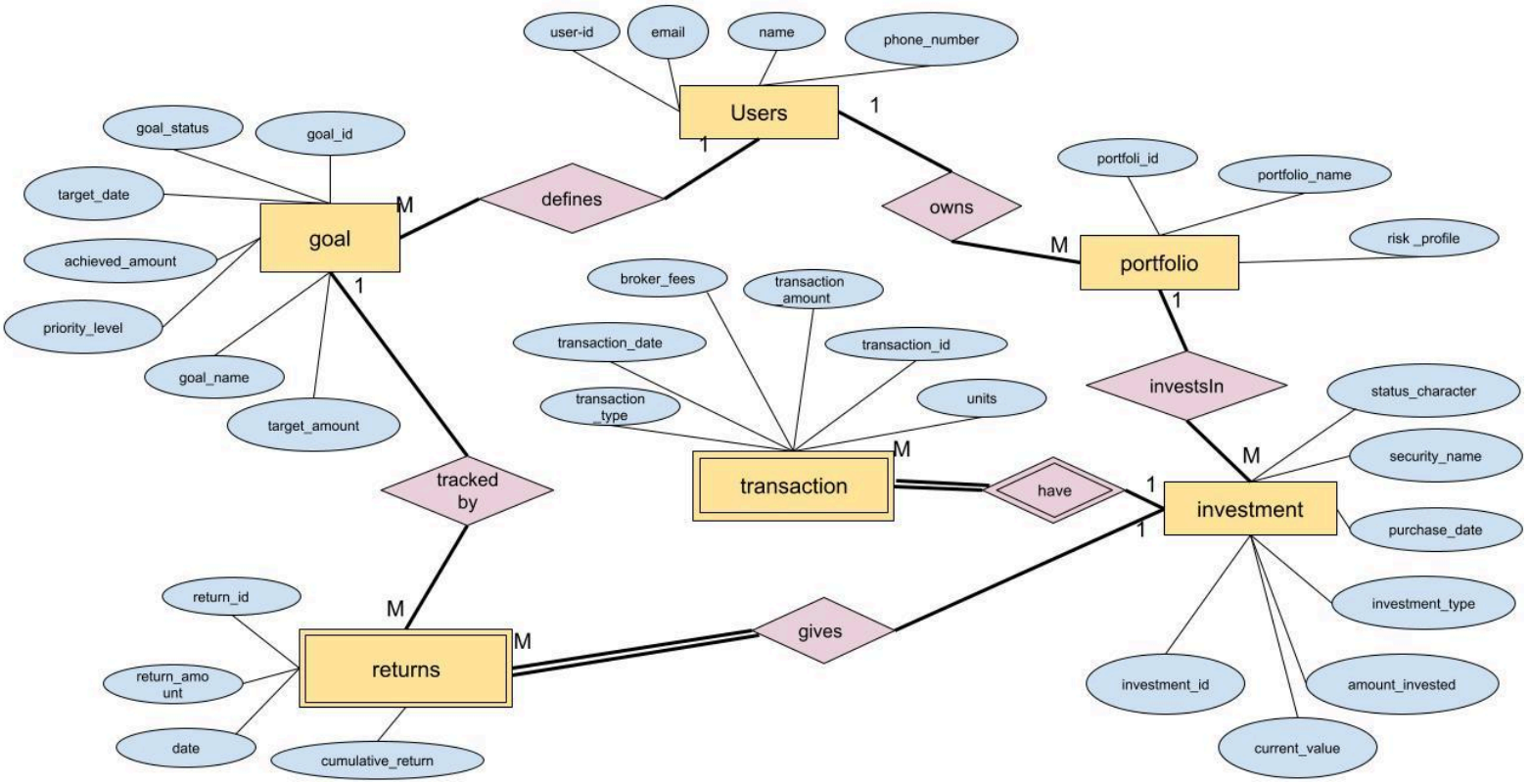
11. Views
    - investor_portfolio view
    - UserInvestmentSummaryView
    - admin_goal_progress view

12. Indices
    - index_risk_profile on portfolio using HASH(risk_profile)
    - index_return_amount on returns using BTREE(return_amount)
    - index_target_date on goal using BTREE(target_date)
    - index_units on transaction using HASH(units)
    - index_transaction_type on transaction using HASH(transaction_type)
    - index_broker_fees on transaction using BTREE(broker_fees)
    - index_status on investment using HASH(status)

13. Queries using indices

Description:
The Personalized Investment Portfolio Tracker is a database-backed app that helps users monitor, manage, and optimize their investments based on their financial goals. It provides insights into portfolio performance and progress toward achieving set goals.

ER Model



Relational Model

1. **User Table**

- `user_id (PK)` → Unique identifier for each user.
- `name` → Full name of the user.
- `email` → User's email address.
- `phone_number` → User’s phone number.

2. **Portfolio Table**

- `portfolio_id (PK)` → Unique identifier for each portfolio.
- `user_id (FK)` → Foreign key referencing the `User` table.
- `portfolio_name` → Name of the portfolio.
- `risk_profile` → Risk level (**high, medium, low**).

3. **Investment Table**

- `investment_id (PK)` → Unique identifier for each investment.
- `portfolio_id (FK)` → Foreign key referencing the `Portfolio` table.
- `security_name` → Name of the stock, bond, or other asset.
- `investment_type` → Type of investment (**equity, mutual fund, fixed deposit, etc.**).
- `amount_invested` → Total amount invested in this asset.
- `current_value` → Current market value of the investment.
- `purchase_date` → Date of purchase.
- `status` → Indicates whether the investment is **active** or **sold**.
- Status: it will tell about the investment like it is sold or active

4. **Transaction Table**

- `transaction_id (PK)` → Unique identifier for each transaction.
- `investment_id (FK)` → Foreign key referencing the `Investment` table**.**
- `transaction_type` → Type of transaction (buy, sell).
- `transaction_date` → Date of the transaction.
- `transaction_amount` → Amount involved in the transaction.
- `units` → Number of units bought or sold.
- `broker_fees` → Fees charged by the broker.

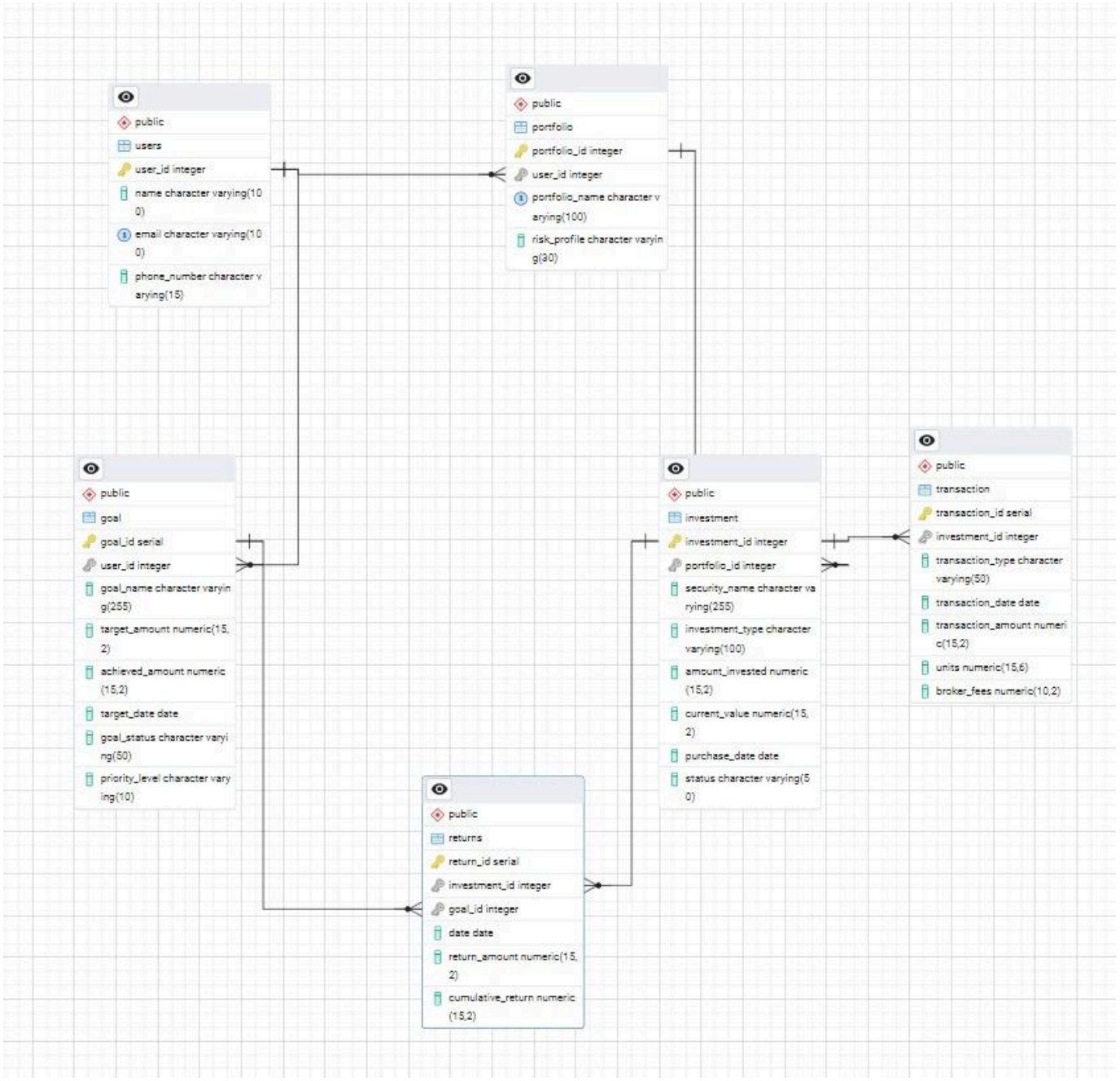
5. **Return Table**

- `return_id (PK)` → Unique identifier for each return entry.
- `investment_id (FK)` → Foreign key referencing the `Investment` table.
- `goal_id (FK)` → Foreign key referencing the `Goal` table.
- `date` → Date of the return calculation.
- `return_amount` → Amount of return generated on this date.

- `cumulative_return` → Cumulative return up to this date.

6. Goal Table

- `goal_id (PK)` → Unique identifier for each goal.
- `user_id (FK)` → Foreign key referencing the `User` table.
- `goal_name` → Name of the financial goal (e.g., retirement, house purchase).
- `target_amount` → Target amount needed to achieve the goal.
- `achieved_amount` → Amount already saved toward the goal.
- `target_date` → Deadline to achieve the goal.
- `goal_status` → Indicates if the goal is **achieved** or **in progress**.
- `priority_level` → Classified as **high, medium, or low** priority


public
users
user_id integer
name character varying(100)
email character varying(100)
phone_number character varying(15)
public
portfolio
portfolio_id integer
user_id integer
portfolio_name character varying(100)
risk_profile character varying(30)
public
goal
goal_id serial
user_id integer
goal_name character varying(255)
target_amount numeric(15,2)
achieved_amount numeric(15,2)
target_date date
goal_status character varying(50)
priority_level character varying(10)
public
investment
investment_id integer
portfolio_id integer
security_name character varying(255)
investment_type character varying(100)
amount_invested numeric(15,2)
current_value numeric(15,2)
purchase_date date
status character varying(50)
public
transaction
transaction_id serial
investment_id integer
transaction_type character varying(50)
transaction_date date
transaction_amount numeric(15,2)
units numeric(15,6)
broker_fees numeric(10,2)
public
returns
return_id serial
investment_id integer
goal_id integer
date date
return_amount numeric(15,2)
cumulative_return numeric(15,2)

Relationships

1. **User → Portfolio** `(1:M)`
    - A **user** can own **multiple portfolios**.
2. **Portfolio → Investment** `(1:M)`
    - Each **portfolio** can contain **multiple investments**.
3. **Investment → Transaction** `(1:M)`
    - Each **investment** can have **multiple transactions**.
4. **User → Goal** `(1:M)`
    - A **user** can have **multiple financial goals**.
5. **Investment → Return** `(1:M)`
    - **Returns** are generated from each **investment**.
6. **Goal → Return** `(1:M)`
    - **Returns** are tracked for **goals** directly (**not through portfolios**).

Tables

1. **users**

```
1  select * from users;
```

Data Output | Messages | Notifications

	user_id [PK] integer	name character varying (100)	email character varying (100)	phone_number character varying (15)
1	1	User1	user1@example.com	6944731350
2	2	User2	user2@example.com	4855652129
3	3	User3	user3@example.com	4200341927
4	4	User4	user4@example.com	6833346219
5	5	User5	user5@example.com	6677191342
6	6	User6	user6@example.com	4506091836
7	7	User7	user7@example.com	3229598815
8	8	User8	user8@example.com	7788587371
9	9	User9	user9@example.com	2058214427
10	10	User10	user10@example.com	3470595340
11	11	User11	user11@example.com	4770772664
12	12	User12	user12@example.com	6038095342
13	13	User13	user13@example.com	6788578636
14	14	User14	user14@example.com	7939353736
15	15	User15	user15@example.com	4107696772
16	16	User16	user16@example.com	7773900061
17	17	User17	user17@example.com	2816437029
18	18	User18	user18@example.com	8531821623
19	19	User19	user19@example.com	2058984214
20	20	User20	user20@example.com	3786434851
21	21	User21	user21@example.com	8516083175
22	22	User22	user22@example.com	9987159663
23	23	User23	user23@example.com	9167485858

2. portfolio

```
select * from portfolio;
```

Data Output | Messages | Notifications

	portfolio_id [PK] integer	user_id integer	portfolio_name character varying (100)	risk_profile character varying (30)
1	1	3407	Portfolio1	Medium
2	2	555	Portfolio2	Low
3	3	1997	Portfolio3	Medium
4	4	1138	Portfolio4	Medium
5	5	173	Portfolio5	Low
6	6	2019	Portfolio6	Low
7	7	4628	Portfolio7	Low
8	8	1397	Portfolio8	High
9	9	780	Portfolio9	Medium
10	10	677	Portfolio10	Medium
11	11	4611	Portfolio11	Low
12	12	1363	Portfolio12	Medium
13	13	4623	Portfolio13	High
14	14	4583	Portfolio14	Medium
15	15	1365	Portfolio15	High
16	16	641	Portfolio16	High
17	17	250	Portfolio17	Low
18	18	370	Portfolio18	Low
19	19	1795	Portfolio19	Low
20	20	1750	Portfolio20	High
21	21	1465	Portfolio21	Low
22	22	2628	Portfolio22	Low
23	23	1937	Portfolio23	Medium

3. investment

```
select * from investment;
```

Data Output | Messages | Notifications

Showing rows: 1 to 1000 | Page No:

	investment_id [PK] integer	portfolio_id integer	security_name character varying (255)	investment_type character varying (100)	amount_invested numeric (15,2)	current_value numeric (15,2)	purchase_date date	status character varying (50)
1	1	4176	Security1	Equity	3083.14	10171.18	2021-10-25	Sold
2	2	198	Security2	Fixed Deposit	8652.29	1531.83	2022-07-18	Sold
3	3	1281	Security3	Fixed Deposit	1541.48	7220.27	2022-07-31	Sold
4	4	850	Security4	Mutual Fund	9539.04	10782.47	2024-12-01	Active
5	5	4244	Security5	Mutual Fund	9806.74	9137.16	2023-01-12	Active
6	6	1510	Security6	Mutual Fund	8661.45	2563.77	2022-05-29	Active
7	7	122	Security7	Equity	3243.07	11763.86	2020-08-26	Active
8	8	4993	Security8	Fixed Deposit	3936.91	7007.74	2021-04-09	Sold
9	9	4577	Security9	Crypto	4409.62	5270.50	2022-07-31	Active
10	10	2762	Security10	Crypto	6874.48	9389.34	2021-08-05	Active
11	11	835	Security11	Fixed Deposit	3207.87	658.05	2021-07-05	Active
12	12	2991	Security12	Fixed Deposit	4780.66	4195.11	2020-06-27	Sold
13	13	1798	Security13	Fixed Deposit	1186.07	11681.47	2020-05-07	Active
14	14	794	Security14	Equity	3656.90	11758.72	2024-03-29	Active
15	15	305	Security15	Equity	2243.79	7330.89	2023-06-05	Sold
16	16	2851	Security16	Crypto	7292.76	8421.35	2020-04-28	Active
17	17	3737	Security17	Crypto	8207.02	9401.90	2020-06-03	Sold
18	18	81	Security18	Equity	8840.42	4533.18	2024-08-13	Sold
19	19	388	Security19	Fixed Deposit	8532.82	8655.35	2024-10-23	Sold
20	20	2178	Security20	Fixed Deposit	7023.68	11699.10	2020-12-02	Sold
21	21	3013	Security21	Crypto	8501.89	1956.41	2021-11-11	Active
22	22	4860	Security22	Crypto	1959.66	3822.32	2020-08-15	Sold
23	23	4863	Security23	Mutual Fund	7965.66	10917.59	2024-09-22	Sold
24	24	4461	Security24	Fixed Deposit	9383.91	4210.22	2020-08-25	Active
25	25	1302	Security25	Mutual Fund	9029.39	626.02	2023-05-09	Sold

4. transaction

```
select * from transaction;
```

Data Output | Messages | Notifications

Showing rows: 1

	transaction_id [PK] integer	investment_id integer	transaction_type character varying (50)	transaction_date date	transaction_amount numeric (15,2)	units numeric (15,6)	broker_fees numeric (10,2)
1	1	496	Buy	2020-08-02	2802.77	50.000000	8.95
2	2	463	Buy	2023-10-07	5971.56	58.000000	45.19
3	3	3441	Sell	2021-09-24	2218.52	37.000000	5.70
4	4	1713	Buy	2020-07-24	3262.03	34.000000	30.41
5	5	1416	Buy	2020-09-29	4540.12	52.000000	49.59
6	6	434	Sell	2020-05-31	5186.58	16.000000	37.14
7	7	3062	Buy	2023-10-24	7885.23	34.000000	2.61
8	8	3605	Buy	2020-08-10	6454.34	33.000000	25.82
9	9	3315	Sell	2024-03-16	6577.11	68.000000	5.80
10	10	2826	Sell	2021-11-27	6005.53	89.000000	43.31
11	11	3271	Sell	2022-02-02	9054.08	18.000000	24.42
12	12	4748	Buy	2023-06-19	111.68	42.000000	39.75
13	13	4419	Buy	2021-08-17	7220.27	70.000000	17.65
14	14	3259	Sell	2020-05-15	150.30	63.000000	24.04
15	15	3926	Buy	2024-05-31	7363.63	60.000000	45.31
16	16	977	Buy	2023-01-20	3283.62	41.000000	8.15
17	17	848	Sell	2022-08-10	6069.13	31.000000	20.04
18	18	1025	Buy	2022-06-05	7737.53	51.000000	12.71
19	19	3382	Buy	2024-12-19	7282.63	92.000000	10.35
20	20	1062	Sell	2023-04-08	4257.38	64.000000	38.59
21	21	4458	Sell	2024-01-26	3797.26	91.000000	29.13
22	22	2359	Sell	2024-05-30	2977.69	29.000000	44.28
23	23	4306	Buy	2021-12-19	7144.05	1.000000	35.57
24	24	2986	Buy	2022-03-21	5296.68	98.000000	24.10
25	25	162	Sell	2023-11-12	5877.26	75.000000	33.85

5. returns

Query | Query History

```
select * from returns;
```

Data Output | Messages | Notifications

	return_id [PK] integer	investment_id integer	goal_id integer	date date	return_amount numeric (15,2)	cumulative_return numeric (15,2)
1	1	3634	2298	2024-07-20	4561.97	4916.60
2	2	638	2650	2020-12-10	3615.94	16307.83
3	3	3569	3600	2023-01-04	2433.28	996.93
4	4	1456	2689	2021-11-22	1635.44	12763.22
5	5	4709	3202	2022-02-20	806.64	7360.75
6	6	922	1585	2023-05-26	2139.50	2585.87
7	7	2336	236	2020-09-27	695.11	13171.59
8	8	4922	3668	2023-07-27	1900.27	19293.59
9	9	4255	4876	2022-07-08	2898.78	971.91
10	10	4857	464	2022-06-01	2913.86	8712.95
11	11	2774	4152	2022-10-31	387.75	5518.77
12	12	1198	3877	2022-05-18	3662.39	1161.06
13	13	2493	4215	2021-08-31	3565.10	19443.07
14	14	1148	4370	2022-05-02	3806.09	2433.56
15	15	2891	59	2021-11-27	2744.23	10209.57
16	16	3762	3807	2023-10-10	4808.39	16239.07
17	17	766	3995	2023-02-28	2650.50	13016.86
18	18	4206	1368	2022-10-03	4333.51	743.32
19	19	1712	861	2025-01-01	75.89	1575.65
20	20	393	2841	2022-01-29	1768.50	1386.67
21	21	4427	1686	2020-10-30	3032.85	17935.15
22	22	4337	981	2021-05-10	4565.47	18523.09
23	23	2599	3230	2024-05-25	3444.66	7752.75

6. goal

```
463  select * from goal;
```

Data Output | Messages | Notifications

Showing rows: 1 to 1000 | Page No: 1 of 5

	goal_id [PK] integer	user_id integer	goal_name character varying (255)	target_amount numeric (15,2)	achieved_amount numeric (15,2)	target_date date	goal_status character varying (15)	priority_level character varying (10)
1	1	1407	Goal1	23574.60	16867.37	2028-07-06	In Progress	Low
2	2	4903	Goal2	17486.72	20178.97	2030-05-12	Achieved	Low
3	3	1193	Goal3	17713.54	23210.55	2031-08-16	Achieved	Low
4	4	2060	Goal4	41109.86	11873.25	2032-09-25	In Progress	Medium
5	5	3030	Goal5	24339.76	16830.97	2026-06-16	In Progress	Low
6	6	3151	Goal6	746.84	14139.04	2031-03-18	Achieved	Medium
7	7	1976	Goal7	6416.10	13007.57	2031-11-11	Achieved	Low
8	8	4505	Goal8	28338.83	4434.93	2026-05-13	In Progress	High
9	9	884	Goal9	44483.79	2439.19	2033-11-29	In Progress	Low
10	10	3146	Goal10	24290.41	5124.61	2025-12-09	In Progress	Medium
11	11	2135	Goal11	32472.19	456.28	2026-11-19	In Progress	Medium
12	12	2373	Goal12	18332.86	29036.39	2027-10-28	Achieved	Medium
13	13	1511	Goal13	37626.05	18235.16	2028-10-01	In Progress	Low
14	14	91	Goal14	29820.20	29822.50	2032-12-08	Achieved	Low
15	15	2218	Goal15	6796.46	9831.91	2029-10-09	Achieved	Low
16	16	3702	Goal16	18693.71	10424.97	2032-06-06	In Progress	High
17	17	968	Goal17	17383.68	22871.19	2033-05-10	Achieved	Medium

Total rows: 5000 | Query complete 00:00:00.742 | CRLF

Integrity Constraints and General Constraints

To ensure the accuracy, consistency, and reliability of data in the FinTrack database system, the following integrity constraints and general constraints have been applied across various tables:

1. Primary Key Constraints

Each table has a uniquely identifying column:

- `user_id` in the User table
- `portfolio_id` in the Portfolio table
- `investment_id` in the Investment table
- `transaction_id` in the Transaction table
- `return_id` in the Return table
- `goal_id` in the Goal table

These ensure uniqueness and prevent duplicate records.

2. Foreign Key Constraints

These ensure referential integrity between tables:

- `user_id` in Portfolio references `User(user_id)`
- `portfolio_id` in Investment references `Portfolio(portfolio_id)`
- `investment_id` in Transaction and Return references `Investment(investment_id)`
- `goal_id` in Return references `Goal(goal_id)`
- `user_id` in Goal references `User(user_id)`

3. NOT NULL Constraints

Important attributes that must always have a value include

- `name`, `email` in User
- `portfolio_name`, `risk_profile` in Portfolio
- `security_name`, `investment_type`, `amount_invested`, `current_value` in Investment
- `transaction_type`, `transaction_date`, `transaction_amount` in Transaction
- `return_amount`, `cumulative_return` in Return
- `goal_name`, `target_amount`, `target_date` in Goal

4. CHECK Constraints

To enforce domain-specific rules:

- `risk_profile` in Portfolio is limited to: `'high'`, `'medium'`, `'low'`
- `status` in Investment is limited to: `'Active'`, `'Sold'`
- `transaction_type` in Transaction is restricted to: `'Buy'`, `'Sell'`
- `goal_status` in Goal is restricted to: `'In Progress'`, `'Achieved'`
- `priority_level` in Goal is limited to: `'High'`, `'Medium'`, `'Low'`
- `amount_invested`, `current_value`, `transaction_amount`, `return_amount`, and `target_amount` must be greater than or equal to 0

5. UNIQUE Constraints

- `Email` in User table is unique to prevent duplicate user registrations.

6. DEFAULT Constraints

- Default values can be used for fields such as `goal_status = 'In Progress'` or `status = 'Active'` in Investment upon insertion, to simplify data entry.

FUNCTIONS

FUNCTION 1

CALCULATE PORTFOLIO VALUE

```
CREATE FUNCTION GetPortfolioValue(portfolio_id INT)
RETURNS DECIMAL AS $$
DECLARE
    total_value DECIMAL;
BEGIN
    SELECT SUM(CurrentValue) INTO total_value
    FROM Investment WHERE PortfolioID = portfolio_id;

    RETURN total_value;
END;
$$ LANGUAGE plpgsql;
```

What It Does?

This function calculates the total value of a portfolio by summing up the current values of all investments in that portfolio.

Useful for: Checking the total worth of a user's portfolio.

How It Works?

Declares a variable total_value to store the sum.

Runs a SELECT SUM(CurrentValue) FROM Investment where PortfolioID matches the input portfolio_id.

Stores the sum into total_value and returns it.

FUNCTION 2

CHECK IF GOAL IS ACHEVED OR NOT?

```
CREATE FUNCTION IsGoalMet(goal_id INT)
RETURNS BOOLEAN AS $$
DECLARE
    target DECIMAL;
    achieved DECIMAL;
BEGIN
    SELECT TargetAmount, AchievedAmount INTO target, achieved
    FROM Goal WHERE GoalID = goal_id;

    RETURN achieved >= target;
END;
$$ LANGUAGE plpgsql;
```

What It Does?
Checks whether a financial goal is met by comparing achieved amount with target amount.
Useful for: Tracking goal progress.

How It Works?
Retrieves the TargetAmount and AchievedAmount for the given goal_id.
Compares achieved with target and returns TRUE if the goal is met, otherwise FALSE.

FUNCTION 3
GET INVESTMENT PERFORMANCE

```
CREATE FUNCTION GetInvestmentPerformance(investment_id INT)
RETURNS DECIMAL AS $$
DECLARE
    invested DECIMAL;
    current DECIMAL;
BEGIN
    SELECT AmountInvested, CurrentValue INTO invested, current
    FROM Investment WHERE InvestmentID = investment_id;

    RETURN ((current - invested) / invested) * 100;
END;
$$ LANGUAGE plpgsql;
```

What It Does?
Calculates investment performance (%), which is the percentage gain or loss based on the difference between CurrentValue and AmountInvested.
Useful for: Checking how well an investment is performing.

How It Works?
Retrieves AmountInvested and CurrentValue for the given investment_id.
Returns the percentage.

FUNCTION 4

CALCULATES TOTAL RETURNS A USER HAS EARNED FROM ALL INVESTMENTS

```
CREATE FUNCTION GetTotalUserReturns(user_id INT)
RETURNS DECIMAL AS $$
DECLARE
    total_returns DECIMAL;
BEGIN
    SELECT COALESCE(SUM(ReturnAmount), 0) INTO total_returns
    FROM Returns
    WHERE InvestmentID IN (
        SELECT InvestmentID FROM Investment WHERE PortfolioID IN (
            SELECT PortfolioID FROM Portfolio WHERE UserID = user_id
        )
    );

    RETURN total_returns;
END;
$$ LANGUAGE plpgsql;
```

What It Does?

Calculates total returns from all investments across all portfolios owned by a user.
Useful for: Checking total investment gains.

How It Works?

Finds all InvestmentIDs related to PortfolioIDs owned by the given user_id.
Sums up ReturnAmount from the Returns table for those investments.
If no returns exist, it returns 0 instead of NULL (using COALESCE).

FUNCTION 5

COUNT ACTIVE INVESTMENTS IN PORTFOLIO

```
CREATE FUNCTION CountActiveInvestments(portfolio_id INT)
RETURNS INT AS $$
DECLARE
    active_count INT;
BEGIN
    SELECT COUNT(*) INTO active_count
    FROM Investment
    WHERE PortfolioID = portfolio_id AND Status = 'Active';

    RETURN active_count;
END;
$$ LANGUAGE plpgsql;
```

What It Does?

Counts the number of active investments in a portfolio.
Useful for: Checking how many investments are currently running.

How It Works?
Counts investments in the Investment table where PortfolioID matches and Status = 'Active'.
Returns the count.

FUNCTION 6
GET USER SUMMARY

```sql
CREATE OR REPLACE FUNCTION get_user_summary(p_user_id INT)
RETURNS JSONB
LANGUAGE plpgsql
AS $$
DECLARE
    result JSONB;
BEGIN
    SELECT jsonb_build_object(
        'user_name', u.name,
        'total_portfolios', COALESCE(p.cnt, 0),
        'total_invested_amount', COALESCE(inv.total_amount, 0),
        'total_current_value', COALESCE(inv.current_value, 0),
        'total_goals', COALESCE(goal_counts.total_goals, 0),
        'goals_achieved', COALESCE(goal_counts.goals_achieved, 0)
    )
    INTO result
    FROM users u
    LEFT JOIN (
        SELECT user_id, COUNT(*) AS cnt
        FROM portfolio
        WHERE user_id = p_user_id
        GROUP BY user_id
    ) p ON u.user_id = p.user_id
    LEFT JOIN (
        SELECT p.user_id,
               SUM(i.amount_invested) AS total_amount,
               SUM(i.current_value) AS current_value
        FROM investment i
        JOIN portfolio p ON i.portfolio_id = p.portfolio_id
        WHERE p.user_id = p_user_id
        GROUP BY p.user_id
    ) inv ON u.user_id = inv.user_id
    LEFT JOIN (
        SELECT user_id,
               COUNT(*) AS total_goals,
               COUNT(*) FILTER (WHERE goal_status = 'Achieved') AS goals_achieved
        FROM goal
        WHERE user_id = p_user_id
        GROUP BY user_id
    ) goal_counts ON u.user_id = goal_counts.user_id
    WHERE u.user_id = p_user_id;

    RETURN result;
END;
$$;
```

What It Does?

Generates a JSON summary of a user's investment profile.

Useful for:
Displaying a snapshot of a user's financial portfolio and goals in a compact format.

How It Works

- Takes a `user_id` as input.
- Retrieves and compiles the following details:
    - User's name.
    - Total number of portfolios they own.
    - Total amount they have invested.
    - Total current value of those investments.
    - Total number of financial goals set.
    - Number of goals achieved (status = `'Achieved'`).
- Uses `LEFT JOIN` to gracefully handle users with missing portfolios, investments, or goals (avoids nulls by using `COALESCE`).
- Returns the result as a **JSONB** object.

Roles

1. Admin

Privileges:

- SELECT, INSERT, UPDATE, and DELETE on all tables.
- EXECUTE on get_user_summary

Purpose: Reserved for system administrators responsible for complete system management, data maintenance, and function execution.

2. Advisor

Privileges:

- Read-only access with limited function execution.
- SELECT on goal and returntable.
- EXECUTE on the following functions only:

  GetTotalUserReturns, IsGoalMet

Purpose: Assigned to financial advisors who need to view user and portfolio data, and assess goal achievement and returns.

3. Fintrack_User

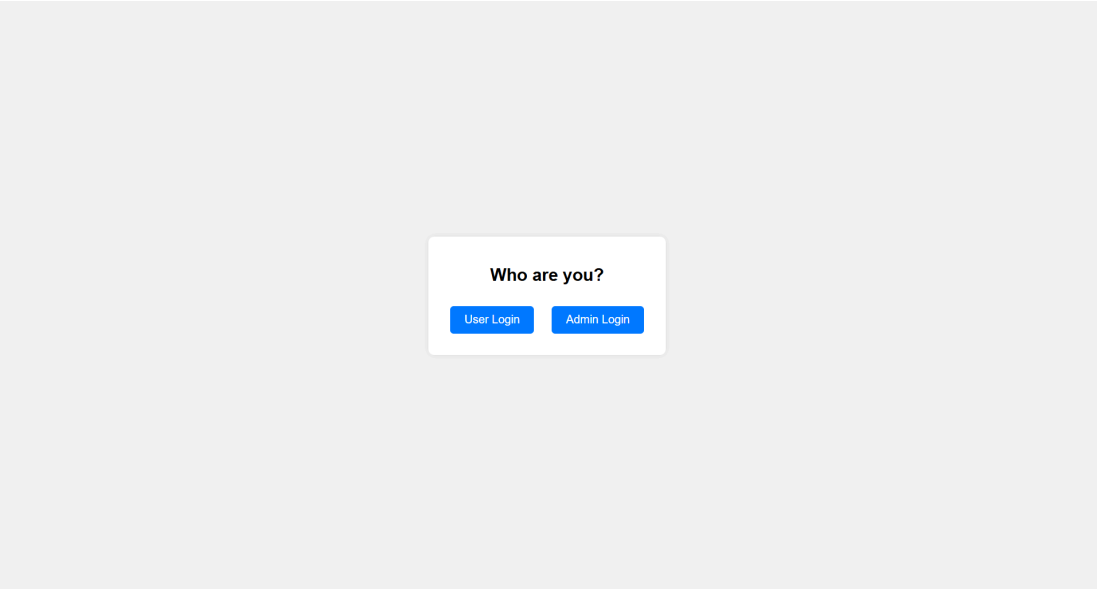
Privileges:

- SELECT, INSERT, UPDATE, and DELETE on all tables.
- EXECUTE on all functions.

Purpose: Assigned to end users of the FinTrack system, allowing them to view their data and perform analysis through available functions without modifying any data.

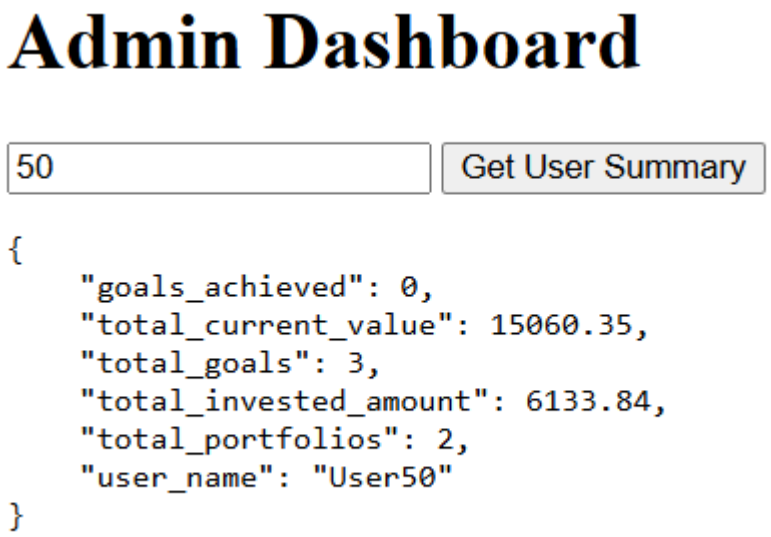
Use of Functions to fetch values on Frontend

1. First page where it is asked, Are you an admin or user?

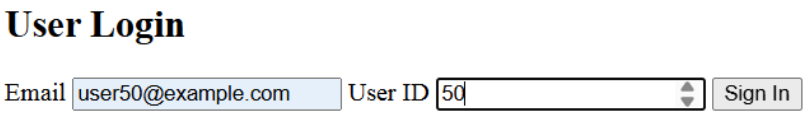


2. If you are admin then it is asked you to enter admin password to enter in admin-dashboard page.

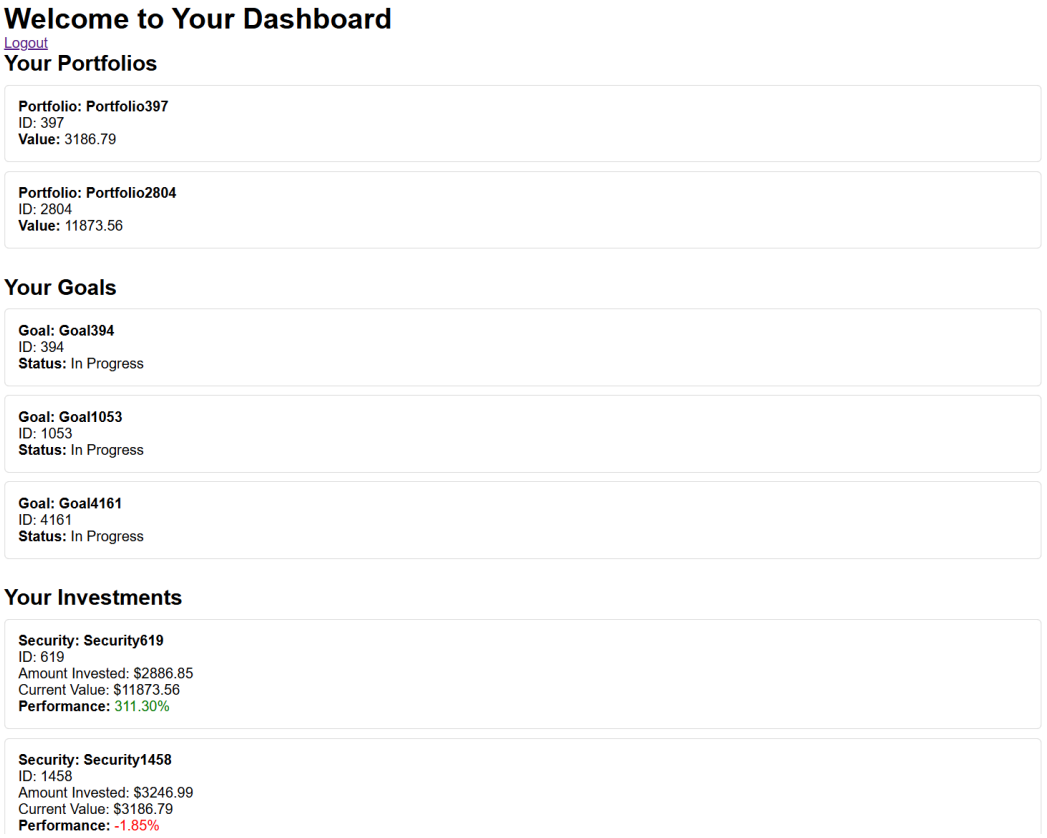
3. After logging in as admin, it is showing the panel to enter user_id and after entering user_id it will show user summary, As we have given permission on get_user_summary function to admin.



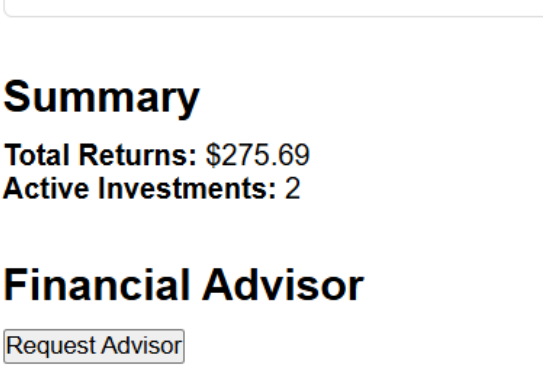
4. If we are logging in as user it will ask us our user_id and email to logging and it will get us to user dashboard.



5. User dashboard will show us value of particular portfolio (using GetPortfolioValue), status of goal, whether it is met or not (using IsGoalMet function), performance of particular investment under portfolio (using GetInvestmentPerformance), total return (using GetTotalUserReturns), and total active investments (using CountActiveInvestments).



It has option of requesting Advisor also advisor will look into users investment performance and total returns generated by them and suggest user that where to invest.



6. After requesting the advisor, the advisor dashboard will be opened, which is accessible to the advisor.

Investments and Total Returns

Security: Security619
ID: 619
Amount Invested: $2886.85
Current Value: $11873.56
Performance: 311.30%

Security: Security1458
ID: 1458
Amount Invested: $3246.99
Current Value: $3186.79
Performance: -1.85%

Total Returns: $275.69
Back to Dashboard

Trigger

1. Trigger-Based Auto-Update of Goal Status

To ensure data consistency and automate goal tracking, a PostgreSQL **trigger function** `trg_check_goal_met()` was implemented. This function dynamically updates the `goal_status` field in the `Goal` table based on changes to the `achieved_amount` column.

Function Logic

- The trigger function compares the updated `achieved_amount` with the `target_amount` for the respective `goal_id`.
- If the achieved amount meets or exceeds the target, the `goal_status` is updated to **'Achieved'**.
- Otherwise, the status is set to **'In Progress'**.
- This logic is encapsulated using the `plpgsql` language, allowing conditional execution within a trigger context.

```
-- Trigger function to auto-update goal status
CREATE OR REPLACE FUNCTION trg_check_goal_met()
RETURNS TRIGGER AS $$
BEGIN
    IF EXISTS (
        SELECT 1 FROM goal WHERE goal_id = NEW.goal_id AND achieved_amount >= target_amount
    ) THEN
        UPDATE goal SET goal_status = 'Achieved' WHERE goal_id = NEW.goal_id;
    ELSE
        UPDATE goal SET goal_status = 'In Progress' WHERE goal_id = NEW.goal_id;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger on update of achieved amount
CREATE TRIGGER trg_goal_achievement
AFTER UPDATE OF achieved_amount ON goal
FOR EACH ROW
EXECUTE FUNCTION trg_check_goal_met();
```

2. Trigger-Based Logging of Investment Performance

To enhance transparency and track real-time changes in investment performance, a trigger function, `trg_log_investment_performance()` was created in the PostgreSQL database system.

Function Logic

- This function is automatically executed when the `current_value` of an investment is updated.
- It calls a user-defined function `GetInvestmentPerformance(investment_id)` to calculate the updated performance metric (e.g., percentage gain/loss).
- A `RAISE NOTICE` command is used to log a message showing the investment ID and the updated performance percentage. This is useful for debugging and audit purposes.

```
-- Trigger function to log performance change
CREATE OR REPLACE FUNCTION trg_log_investment_performance()
RETURNS TRIGGER AS $$
DECLARE
    perf DECIMAL;
BEGIN
    perf := GetInvestmentPerformance(NEW.investment_id);
    RAISE NOTICE 'Updated Performance for Investment %: %%%', NEW.invenstment_id, perf;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger on investment value update
CREATE TRIGGER trg_investment_perf_update
AFTER UPDATE OF current_value ON investment
FOR EACH ROW
EXECUTE FUNCTION trg_log_investment_performance();
```

3.Trigger-Based Notification of Active Investments Count

To enable real-time awareness of investment distribution, a trigger function named `trg_notify_active_investments()` was developed. This function logs the total number of active investments within a portfolio whenever a new active investment is inserted.

Function Logic

- The function queries the count of active investments linked to the portfolio using a custom utility function `CountActiveInvestments(portfolio_id)`.

- A `RAISE NOTICE` statement logs the total number of active investments along with the associated `portfolio_id`. This feedback is useful for internal monitoring and debugging during development.

```
-- Trigger function to notify count of active investments
CREATE OR REPLACE FUNCTION trg_notify_active_investments()
RETURNS TRIGGER AS $$
DECLARE
    count_active INT;
BEGIN
    count_active := CountActiveInvestments(NEW.portfolio_id);
    RAISE NOTICE 'Total Active Investments in Portfolio %: %', NEW.portfolio_id, count_active;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger after inserting new investment
CREATE TRIGGER trg_active_investment_count
AFTER INSERT ON investment
FOR EACH ROW
WHEN (NEW.Status = 'Active')
EXECUTE FUNCTION trg_notify_active_investments();
```

4.Trigger-Based Recalculation of User Returns

To maintain accurate tracking of user-wide financial performance, a trigger function `trg_recalculate_user_returns()` was created. This function is invoked whenever a new investment record is inserted into the `investment`, automatically recalculating and logging the user's total returns.

Function Logic

- The function first determines the `userID` associated with the newly added return. This is done by navigating from the `returns` to the `investment`, and then to the corresponding `Portfolio` record.

- Using the retrieved `userID`, the function calls a user-defined utility function `GetTotalUserReturns(uid)` to compute the user's updated total return.

- A `RAISE NOTICE` statement outputs the result for logging and verification purposes.

```
-- Trigger function to recalculate and log user returns
CREATE OR REPLACE FUNCTION trg_recalculate_user_returns()
RETURNS TRIGGER AS $$
DECLARE
    uid INT;
    total_ret DECIMAL;
BEGIN
    SELECT userID INTO uid
    FROM Portfolio
    WHERE PortfolioID = (
        SELECT PortfolioID FROM investment WHERE investment_id = NEW.investment_id
    );

    total_ret := GetTotalUserReturns(uid);
    RAISE NOTICE 'Total Returns for User %: ₹%', uid, total_ret;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger after a new return is added
CREATE TRIGGER trg_user_return_update
AFTER INSERT ON returntable
FOR EACH ROW
EXECUTE FUNCTION trg_recalculate_user_returns();
```

Views

1. Investor Portfolio View

View Definition

This view is generated by joining three core tables: `Users`, `Portfolio`, and `Investment`. It extracts and displays the following attributes:

- `u.name`: Name of the investor.
- `p.portfolio_name`: Name of the portfolio associated with the user.
- `i.security_name`: The name of the security or investment held.
- `i.current_value`: The current market value of the investment.

```sql
CREATE VIEW investor_portfolio_view AS
SELECT u.name, p.portfolio_name, i.security_name, i.current_value
FROM Users u
JOIN Portfolio p ON u.user_id = p.user_id
JOIN Investment i ON p.portfolio_id = i.portfolio_id;
```

2. User Investment Summary View

View Definition

This view is generated by joining the Users, Portfolio, and Investment tables. It summarizes each user's overall investment performance by aggregating their total investment data. It displays the following attributes:

- u.user_id: Unique identifier of the user.
- u.name: Name of the investor.
- total_amount_invested: The total amount invested by the user across all portfolios (sum of `i.amount_invested`).
- total_current_value: The total current market value of all the user's investments (sum of `i.current_value`).

- net_gain_loss: The overall gain or loss in monetary terms, calculated as total current value minus total amount invested.

- return_percentage: The percentage return on investment, calculated as net gain/loss divided by the amount invested, multiplied by 100.

```
CREATE VIEW UserInvestmentSummaryView AS
SELECT
    u.user_id,
    u.name AS user_name,
    SUM(i.amount_invested) AS total_amount_invested,
    SUM(i.current_value) AS total_current_value,
    ROUND(SUM(i.current_value - i.amount_invested), 2) AS net_gain_loss,
    ROUND((SUM(i.current_value - i.amount_invested) / NULLIF(SUM(i.amount_invested), 0)) * 100, 2) AS
return_percentage
FROM Users u
JOIN Portfolio p ON u.user_id = p.user_id
JOIN Investment i ON p.portfolio_id = i.portfolio_id
GROUP BY u.user_id, u.name;
```

3. Admin Goal Progress View

The view joins the `Users` and `Goal` tables to present the following details:

- `u.name`: Name of the user.

- `g.goal_name`: The title or label of the goal.

- `g.achieved_amount`: The amount already accumulated toward the goal.

- `g.target_amount`: The total target amount required to complete the goal.

```
CREATE VIEW admin_goal_progress AS
SELECT u.name, g.goal_name, g.achieved_amount, g.target_amount
FROM Users u
JOIN Goal g on u.user_id=g.user_id;
```

INDICES

1. `CREATE INDEX index_risk_profile ON portfolio USING HASH(risk_profile);`

Why:
To quickly filter portfolios based on the **risk profile** (`High`, `Medium`, `Low`). This is useful when you're querying like:

```
SELECT * FROM portfolio WHERE risk_profile = 'High';
```

Why HASH:

- Perfect for **equality comparisons**.
- HASH is faster than BTREE for queries using `=` (but not for `<`, `>`, or sorting).

2. `CREATE INDEX index_return_amount ON returns USING BTREE(return_amount);`

Why:
Improves performance for range queries and sorting on `return_amount`

```
SELECT * FROM returns WHERE return_amount > 1000;
```

Why BTREE:

- Ideal for **range-based queries**, sorting (`ORDER BY`), and comparisons (`>`, `<`, `BETWEEN`).

3. `CREATE INDEX index_target_date ON goal USING BTREE(target_date);`

Why:
Optimizes queries that filter or sort goals by their target completion date, like:

```
SELECT * FROM goal WHERE target_date <= CURRENT_DATE;
```

Why BTREE:

- Best for **date comparisons**, which are inherently ordered.
- Supports range scans and ordering efficiently.

4. `CREATE INDEX index_units ON transaction USING HASH(units);`

Why:
Speeds up lookups where you're checking for a specific number of units:

```
SELECT * FROM transaction WHERE units = 100;
```

Why HASH:

- Efficient for exact-match queries.
- Not useful for range-based conditions (`>`, `<`).

5. `CREATE INDEX index_transaction_type ON transaction USING HASH(transaction_type);`

Why:
Improves filtering of transactions by type (`Buy`, `Sell`):

```
SELECT * FROM transaction WHERE transaction_type = 'Sell';
```

Why HASH:

- Great for categorical fields with fixed values and equality checks.
- Fast for `=` operations.

6. `CREATE INDEX index_broker_fees ON transaction USING BTREE(broker_fees);`

Why:
Optimizes queries comparing or sorting based on broker fees:

```
SELECT * FROM transaction WHERE broker_fees > 10 ORDER BY broker_fees DESC;
```

Why BTREE:

- Excellent for numeric **range queries** or **sorting**.

7. `CREATE INDEX index_status ON investment USING HASH(status);`

Why:
Speeds up lookups by investment status (`Active`, `Sold`), like:

```
SELECT * FROM investment WHERE status = 'Active';
```

Why HASH:

- Ideal when using equality filters on categorical fields.
- Faster than BTREE for `=` queries when range logic is not needed.

QUERIES USING INDICES

1. index_risk_profile ON Portfolio USING HASH(risk_profile)

- The `risk_profile` column is used in a simple equality condition.
- Hash indexes are optimized for such equality searches.
- PostgreSQL will quickly locate all rows in the `Portfolio` table where `risk_profile = 'High'`.
- This reduces the need for a sequential scan and speeds up the join and aggregation.

```sql
--query using hash index
SELECT
    u.user_id,u.name AS user_name,u.email,p.portfolio_id,p.portfolio_name,
    p.risk_profile,COALESCE(SUM(i.amount_invested), 0) AS total_invested
FROM Portfolio p
JOIN Users u ON u.user_id = p.user_id
LEFT JOIN Investment i ON i.portfolio_id = p.portfolio_id
WHERE p.risk_profile = 'High'
GROUP BY u.user_id, u.name, u.email, p.portfolio_id, p.portfolio_name, p.risk_profile
ORDER BY u.name;
```

2. index_return_amount ON Returns USING BTREE(return_amount)

- B-tree indexes are great for **range queries** like `> 1000`.
- The `ORDER BY r.return_amount DESC` clause also **benefits from B-tree's sorted structure**.
- PostgreSQL can efficiently **scan the index in reverse order** to get the top values (used with `LIMIT 5`).
- Reduces the number of rows accessed and sorted, improving performance.

```sql
--query using btree index
EXPLAIN ANALYZE
SELECT
    r.return_id,r.investment_id,r.goal_id,r.date,r.return_amount,r.cumulative_return
FROM Returns r
WHERE r.return_amount > 1000
ORDER BY r.return_amount DESC
LIMIT 5;
```

3. index_target_date ON Goal USING BTREE(target_date)

- The **date range query** benefits from the **sorted order** of B-tree.
- The `BETWEEN` condition lets the planner use **index range scan** to fetch only relevant rows.
- `ORDER BY target_date ASC` is naturally supported by the index, so **no extra sorting is needed**.

```sql
--query using btree index
EXPLAIN ANALYZE
SELECT
    goal_id,goal_name,target_amount,achieved_amount,target_date,goal_status
FROM Goal
WHERE target_date BETWEEN CURRENT_DATE AND CURRENT_DATE + INTERVAL '30 days'
ORDER BY target_date ASC;
```

4. index_units ON Transaction USING HASH(units)

index_transaction_type ON Transaction USING HASH(transaction_type)

index_broker_fees ON Transaction USING BTREE(broker_fees)

`t.units = 100` and `t.transaction_type = 'Buy'`:

- Both use **Hash indexes**, optimized for **exact value matches**.
- PostgreSQL may use **bitmap index scans** to combine these efficiently.

`t.broker_fees < 50`

- Uses the **B-tree index** to quickly identify rows with fees under the threshold.
- Ideal for **range filtering**.

Combining Conditions:

- PostgreSQL may perform **bitmap AND/OR operations** to merge multiple index results.
- The **final sort on `transaction_date` DESC** might require additional sorting if no index is defined on that column.

```sql
--query using btree and hash index
EXPLAIN ANALYZE
SELECT i.security_name,t.transaction_type,t.transaction_date,
       t.transaction_amount,t.broker_fees
FROM Transaction t
JOIN Investment i ON t.investment_id = i.investment_id
WHERE t.units = 100
  AND t.transaction_type = 'Buy'
  AND t.broker_fees < 50
ORDER BY t.transaction_date DESC;
```