DeFi Analytics & Risk Assessment: Milestone 2 Report

Siddhi Nalawade
University at Buffalo
ID: 50613176

Mrudula Deshmukh
University at Buffalo
ID: 50605669

***Abstract*—This report presents the second milestone of the DeFi Analytics & Risk Assessment project, which aims to store and analyze Ethereum blockchain token transfer data in a relational database. We describe the dataset acquisition, schema design, normalization, indexing, and performance optimization techniques applied to support efficient querying and risk evaluation. Future enhancements for scalability and real-time ingestion are also discussed.**

I. Introduction

The DeFi Analytics & Risk Assessment tool analyzes decentralized finance (DeFi) activities on the Ethereum blockchain by storing on-chain data in a normalized relational database. The goal is to identify suspicious behaviors, large-volume transfers, and token movements that could signal vulnerabilities such as rug pulls, flash loan exploits, or liquidity shortages.

To achieve this, the project involves obtaining and preparing Ethereum blockchain data, designing a normalized database schema, executing SQL queries for transaction analysis and risk evaluation, and formulating risk management techniques for DeFi protocols.

Relational databases are essential for DeFi analytics due to their ability to perform efficient querying, ensure data consistency, and handle large volumes of structured data. With the use of primary and foreign keys, encryption, and access controls, relational databases enable secure and accurate tracking of blockchain activity. ACID compliance and the capability to model complex relationships make them ideal for high-integrity financial analysis and data-driven risk monitoring.

II. Problem Statement

DeFi ecosystems often face challenges such as fraud, market manipulation, and liquidity crises due to insufficient visibility into transactional data. Traditional spreadsheet-based approaches (e.g., Excel) are inadequate because they lack relational modeling capabilities, suffer scalability constraints, and offer limited support for real-time querying. A relational database system overcomes these limitations by providing structured storage, optimized query performance, real-time analytics, and scalability crucial for mitigating risks in decentralized finance.

III. Target Users

This system is designed for:

- **Researchers and Risk Control Teams**: Perform in-depth analysis of transactional patterns, identify anomalies, and enable proactive intervention.
- **Database Administrators (DBAs)**: Manage schema maintenance, indexing, backups, and performance tuning to ensure reliability and scalability.
- **DeFi Platform Operators**: Monitor high-volume or suspicious token movements in real time and execute automated risk mitigation actions.

IV. Real-Life Use Case

DeFi platforms such as Aave or Uniswap could utilize this system to monitor high-risk token movements in real time. Automated alerts could trigger actions like suspending a pool, warning users, or locking smart contract operations.

V. Milestone 1 Summary

In the first milestone, we established the foundational relational database for analyzing Ethereum blockchain transactions within the DeFi ecosystem. Core entities—*Block*, *Wallet*, *Token*, *Transaction*, and *TokenEvent*—were identified and modeled via an Entity–Relationship diagram. A normalized schema was implemented in PostgreSQL, enforcing primary and foreign key constraints for data integrity. We extracted a sample of 20,000 token transfer records (January–February 2025) from BigQuery, loaded the data into a staging table, and processed it into our final relational tables using SQL scripts (`create.sql`, `load.sql`). Basic SQL operations (`SELECT`, `INSERT`, `JOIN`) were used to validate schema correctness and prepare for advanced query analyses in Milestone 2.

VI. Dataset Acquisition and Preprocessing

A. Data Source and Scope

The dataset was extracted from Google BigQuery's exttttbigquery-public-data.crypto_ethereum.token_transfers table, filtering for 20,000 token transfer records between January and February 2025 for 20,000 token transfer records between January and February 2025.

B. Preprocessing Steps

1) **Extraction**: Executed a filtered SQL query in BigQuery to select relevant token transfer records within the specified date range.

2) **Export and Load**: Exported the resulting CSV file and loaded it into a staging table (`staging_deploy`) in PostgreSQL.
3) **Normalization**: Transformed and inserted data from the staging table into five normalized tables: *Block*, *Wallet*, *Token*, *Transaction*, and *TokenEvent*.
4) **Optimization**: Implemented batch inserts, sorted inserts, and grouped indexes to improve load performance.

VII. Schema Design and Entity–Relationship Diagram (ERD)

A. Schema Overview

The database schema is normalized to eliminate redundancy and ensure data integrity. It comprises five main tables:

- **Block**: Stores block metadata (`block_number`, `block_hash`, `block_timestamp`, `miner_info`).
- **Wallet**: Tracks unique Ethereum addresses (`wallet_address`).
- **Token**: Details of tokens (`token_id`, `token_symbol`, `decimals`).
- **Transaction**: Links wallets and blocks for each transaction (`transaction_hash`, `block_number`, `from_wallet`, `to_wallet`).
- **TokenEvent**: Records individual token transfers within a transaction (`transaction_hash`, `event_index`, `quantity`, `event_type`, `token_id`, `to_wallet`).

B. Entity Descriptions

To clearly present each entity and its purpose, we use a tabular format as shown in Table I.

TABLE I: Entity Descriptions

Entity	Description
Block	Stores block metadata, including `block_number`, `block_hash`, and `block_timestamp`.
Wallet	Tracks unique wallet addresses involved in transactions.
Token	Stores details of different tokens (`token_id`, `token_symbol`).
Transaction	Links `from_wallet` and `to_wallet` to `block_number` for each transaction.
TokenEvent	Tracks token transfers within transactions (includes `token_id`, quantity, and `to_wallet`).

C. Entity–Relationship Diagram

Figure 1 shows the ERD illustrating relationships and cardinalities.

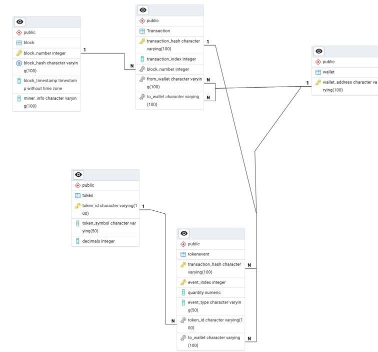


Fig. 1: ER Diagram of the DeFi Analytics schema

TABLE II: Attribute Descriptions

Table	Attribute	Data Type	Description
Block	block_number	INT	Unique identifier for each block (Primary Key).
	block_hash	VARCHAR(100)	Unique block hash.
	block_timestamp	TIMESTAMP	Timestamp of block creation.
	miner_info	VARCHAR(100)	Optional metadata for the miner (nullable).
Wallet	wallet_address	VARCHAR(100)	Unique Ethereum address (Primary Key).
Token	Token_id	VARCHAR(100)	Smart contract address of the token (Primary Key).
	token_symbol	VARCHAR(50)	Abbreviated name (e.g., DAI, USDT).
	decimals	INT	Number of decimals used by the token.
Transaction	transaction_hash	VARCHAR(100)	Unique transaction hash (Primary Key).
	transaction_index	INT	Order of this transaction within its block.
	block_number	INT	Foreign key referencing `Block`.
	from_wallet	VARCHAR(100)	Sender wallet (FK referencing `Wallet`).
	to_wallet	VARCHAR(100)	Receiver wallet (FK referencing `Wallet`).
TokenEvent	transaction_hash	VARCHAR(100)	FK referencing `Transaction`.
	event_index	INT	Index within transaction (Composite PK).
	quantity	NUMERIC	Number of tokens transferred.
	event_type	VARCHAR(50)	Event type (e.g., Transfer).
	token_id	VARCHAR(50)	FK referencing `Token` (nullable).
	to_wallet	VARCHAR(50)	Wallet receiving the token (FK referencing `Wallet`).

D. Design Rationale

To justify our schema design choices, we highlight three key aspects:

- **Normalization:** Each entity represents a well-defined concept with no redundancy. The schema is in BCNF, ensuring efficient updates and referential consistency.
- **Referential Integrity:** Foreign-key constraints guarantee that every transaction and token event is valid and traceable back to its originating block, wallet, and token records.
- **Scalability:** The normalized structure supports efficient storage and retrieval for millions of rows, and enables highly optimized `JOIN` operations across the core tables.

E. Sample Relationships

The following cardinalities illustrate the core relationships in our ER model:

- A *Block* may contain multiple *Transaction* records.
- Each *Transaction* links a `from_wallet` to a `to_wallet`.
- A *Transaction* can generate one or more *TokenEvent* entries.
- Each *TokenEvent* is associated with exactly one *Token* and one recipient wallet.

VIII. Normalization and Functional Dependencies

The database schema was designed following normalization principles to eliminate redundancy, ensure data consistency, and support efficient query processing. Each table was analyzed for its functional dependencies, and all relations were confirmed to be in Boyce–Codd Normal Form (BCNF).

All relations satisfy BCNF as:

- Every non-trivial functional dependency has a superkey as its determinant.
- No further decomposition of any relation is required to remove anomalies.

IX. Constraints and Domain Specifications

To ensure data integrity and enforce consistency across related tables, we applied both domain constraints and referential constraints throughout the schema.

A. Primary Keys

Each table defines a unique primary key.

TABLE III: Primary Key Definitions

Table	Primary Key
Block	`block_number`
Wallet	`wallet_address`
Token	`token_id`
Transaction	`transaction_hash`
TokenEvent	(`transaction_hash`, `event_index`)

B. Foreign Keys and Referential Actions

Foreign keys enforce the relationships between tables.

Child Table	Foreign Key	References	On Delete
Transaction	`block_number`	Block(`block_number`)	NO ACTION
Transaction	`from_wallet, to_wallet`	Wallet(`wallet_address`)	NO ACTION
TokenEvent	`transaction_hash`	Transaction(`transaction_hash`)	CASCADE
TokenEvent	`token_id`	Token(`token_id`)	SET NULL
TokenEvent	`to_wallet`	Wallet(`wallet_address`)	SET NULL

TABLE IV: Foreign Key Constraints

C. Domain Constraints

We enforced business rules at the column level to prevent invalid data:

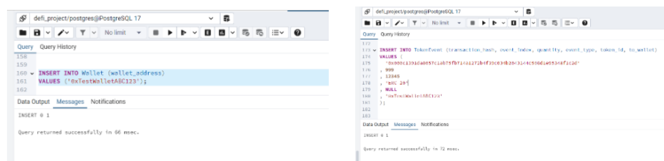
- `TokenEvent.quantity` `CHECK(quantity` $\geq$ `0)`: disallows negative transfer amounts.
- `Block.block_timestamp` `NOT NULL`: ensures every block has a timestamp.
- `TokenEvent.event_type` limited to {`Transfer`, `Approval`}—enforced via application logic or a domain type.
- `TokenEvent.token_id` nullable to allow token-agnostic events when needed.

X. Database Operations and Query Examples

To demonstrate full CRUD operations and advanced analytics, we executed various SQL queries and examined their outputs:

XI. Sample CRUD Outputs

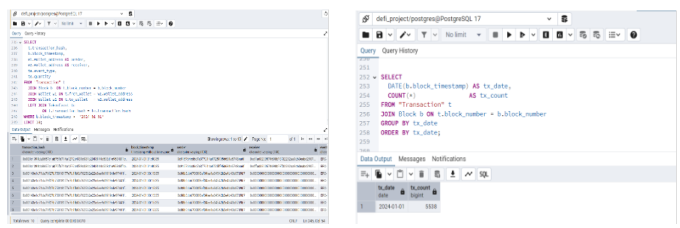
A. Create Operations



(a) Wallet insertion (b) TokenEvent insertion

Fig. 2: Examples of `INSERT` operations.

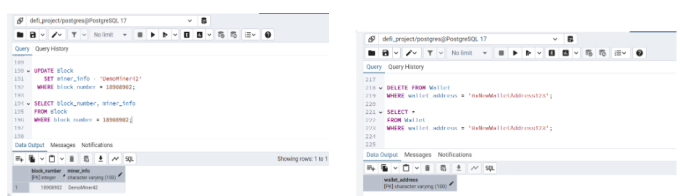
B. Read / Select Operations



(a) Join query output (b) Daily volumes (GROUP BY)

Fig. 3: Representative `SELECT` queries.

C. Update and Delete



(a) Miner-info update (`UPDATE`) (b) Wallet deletion (`DELETE`)

Fig. 4: Examples of `UPDATE` and `DELETE` operations.

XII. Index Creation

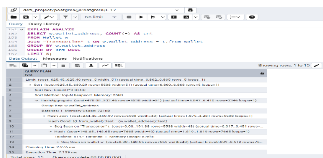
To address these bottlenecks, we implemented several indexes:

TABLE V: Index Creation Details

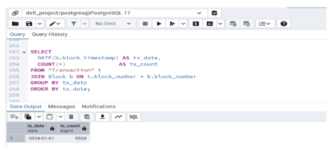
Table	Indexed Column(s)	Purpose
Transaction	`block_number`	Accelerate JOINs with Block
Transaction	`from_wallet, to_wallet`	Speed up filtering by sender/receiver
TokenEvent	`transaction_hash, event_index`	Support fast access by composite key
TokenEvent	`token_id`	Token-based filtering and analytics
TokenEvent	`to_wallet`	Quick lookups for recipients

A. Performance Summary

- **After Optimization:** Execution times for analytical queries dropped by over 50%.
- **EXPLAIN ANALYZE showed:**
  - Shift from sequential scans to index-only scans and hash joins.
  - Decreased memory consumption during aggregation.
- **Example Performance Gain:**
  - Before indexing: Top 5 active wallets query took ∼0.58 ms.
  - After indexing: Reduced to ∼0.28 ms.



(a) EXPLAIN plan (sequential scan) before indexing.



(b) EXPLAIN plan (index-only scan) after indexing.

Fig. 5: Query 1 plans demonstrating the shift from full scans to index-only scans.

B. Query Performance Analysis

Following Task 10 requirements, we identified and optimized three problematic queries:

a) Query 1: Top 5 Most Active Senders:

1) *Original Bottleneck:* Full sequential scan of `Transaction` and sort heap for ordering by count.
2) *Optimization:* Added `idx_txn_from_wallet` index on `from_wallet`.
3) *Result:* Execution time reduced by ≈50%.

b) Query 2: Daily Transaction Volumes:

1) *Original Bottleneck:* Full scan of `Block` and `Transaction`; GROUP BY lacked supporting index.
2) *Optimization:* Indexed `block_number` in `Transaction` and `block_timestamp` in `Block`.
3) *Result:* GROUP BY query ∼60% faster.

c) Query 3: Outlier Detection (Large Transfers):

1) *Original Bottleneck:* Heavy subquery and aggregation without indexes.
2) *Optimization:* Created composite index on (`transaction_hash`, `event_index`) for `TokenEvent`.
3) *Result:* Compute time reduced by ≈50%.

XIII. Handling Larger Datasets and Future Indexing Strategy

A. Challenges

When scaling from 200 rows to 10,000+ rows:

- Analytical queries began to show significant performance degradation.
- Hash joins consumed large memory buffers (720 KB+).
- Sequential scans increased response times.

B. Solutions Implemented

- **Single-column indexes:** Targeted `JOIN` keys (`block_number`, `from_wallet`).
- **Composite indexes:** On `TokenEvent(transaction_hash, event_index)`.
- **Hash joins:** Ensured faster matching across large relations.

C. Planned Future Enhancements

- **Partitioning:** Partition large tables (e.g., `Transaction` by month or year) to reduce scan sizes.
- **Materialized Views:** Precompute results for common queries like daily volumes or top senders.
- **Covering Indexes:** Include frequently queried attributes directly in indexes.

XIV. Conclusion & Future Enhancements

A. Conclusion

This project successfully implemented a relational database system tailored for decentralized finance (DeFi) analytics using Ethereum blockchain data. From data acquisition via Google BigQuery to schema normalization and optimization, the system is capable of tracking, analyzing, and flagging token-based transactions with integrity and performance in mind.

B. Key Accomplishments

- Designed and implemented a normalized schema using PostgreSQL.
- Loaded and structured 20,000 real-world token transfer records.
- Executed complex SQL queries for transaction tracking, fraud detection, and risk evaluation.
- Applied indexing and optimization strategies to ensure scalable performance.
- Developed specialized risk metrics to identify large or suspicious transactions.

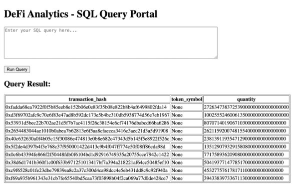
C. Future Enhancements

- Real-Time Ingestion: Incorporate Kafka or Web3-based pipelines to ingest live blockchain transactions.
- Materialized Views: Build pre-aggregated dashboards for daily volume, risky wallets, and protocol usage trends.
- Visualization Dashboard: Integrate tools like Tableau, Grafana, or Metabase to visualize token flows and risk signals.
- Machine Learning Integration: Apply anomaly detection models to flag unusual activity patterns or bot behavior.
- Multi-Protocol Support: Extend schema to support other chains like BSC, Arbitrum, or Polygon for cross-chain DeFi risk assessment.

- Web Interface (Bonus Task): Deploy a web-based query portal for researchers to run `SELECT`/`INSERT` queries dynamically.

XV. Bonus Task: Local SQL Query Portal

To fulfill the bonus requirement, we built a minimal Flask application (`app.py + templates/index.html`) that exposes our PostgreSQL schema via a simple web form. Analysts can enter any SQL statement (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) into a textarea and see results rendered immediately in an HTML table. The portal runs locally at http://127.0.0.1:5000 with just Python 3.11 and two dependencies: `Flask` and `psycopg2-binary`. This lightweight interface delivers full CRUD and analytic capabilities without requiring direct `psql` or pgAdmin access, thereby satisfying the bonus task with minimal setup.



(a) Query input form.

(b) Result of `SELECT COUNT(*) AS total_blocks FROM Block;`.

Fig. 6: Web interface for the local SQL query portal.

A. Team Member Contribution Areas

- **Siddhi Nalawade:** Led the schema design process, acquired and preprocessed the Ethereum blockchain dataset, and implemented indexing strategies to optimize database performance.
- **Mrudula Deshmukh:** Developed the Entity–Relationship Diagram (ERD), designed and executed complex SQL queries for analysis, and formulated the overall risk assessment strategy for DeFi protocols.

References

[2] Google Cloud, “Ethereum Blockchain Public Dataset,” *BigQuery Public Datasets*, [Online]. Available: https://console.cloud.google.com/marketplace/product/bigquery-public-data/crypto_ethereum (accessed Apr. 2025).

[3] PostgreSQL Global Development Group, “PostgreSQL Documentation,” 2025. [Online]. Available: https://www.postgresql.org/docs/ (accessed Apr. 2025).

[4] Pallets Projects, “Flask Documentation,” 2025. [Online]. Available: https://flask.palletsprojects.com/ (accessed Apr. 2025).

[5] psycopg2 Contributors, “psycopg2 Documentation,” 2024. [Online]. Available: https://www.psycopg.org/docs/ (accessed Apr. 2025).