INDIVIDUAL PROJECT – ADVANCED DATABASES CLASS

PL/SQL DATABASE INSPIRED IN THE TORNADO CASH PROTOCOL

DEVELOPED BY JOSÉ BRAZ, ERASMUS STUDENT IN UNIVERSIDAD DE CANTABRIA

⟹ Introduction

The narrative adopted for the implementation of all the concepts learned during the classes of Advanced Databases related to the Relational Paradigm was the decentralized anonymous protocol Tornado Cash, based on the Ethereum blockchain and supporting other chain's interactions.

In this document, the topics covered will be the concepts about the Tornado Cash protocol, followed by the relation between the database and the protocol, adding the physical implementation of the database with the respective explanations.

⟹ Tornado Cash Protocol

Tornado Cash is a decentralized, anonymous, and trustless platform built on the Ethereum blockchain that allows users to send and receive cryptocurrency without revealing their identity or transaction history. It is based on a concept called "zero-knowledge proofs," which allows users to prove the authenticity of a statement without revealing the underlying information.

Ethereum is a decentralized, open-source blockchain platform that enables the creation of smart contracts and decentralized applications (dApps). Tornado Cash is built on the Ethereum blockchain, which means that it is maintained by a network of users who contribute their computing power to verify and validate transactions.

One of the key features of Tornado Cash is anonymity. When using the platform, users can send and receive cryptocurrency without revealing their identity or transaction history to the rest of the network. This makes it a useful tool for privacy-conscious individuals and organizations.

Tornado Cash is also trustless, meaning that users do not need to trust any central authority or third party to use the platform. Transactions are verified and validated by the network, so users can be confident that their transactions will be processed securely and accurately.

Smart contracts are self-executing contracts with the terms of the agreement between buyer and seller being directly written into lines of code. Tornado Cash uses smart contracts to ensure that the terms of the contract are followed and to automate processes.

Decentralized applications (dApps) are applications that run on a decentralized platform, such as the Ethereum blockchain. Tornado Cash is a dApp that allows users to send and receive cryptocurrency anonymously. It is a useful tool for anyone looking to maintain their privacy while using cryptocurrency.

In 2021, The U.S Department of Treasury sanctioned the protocol, banning Americans from using a service that the government said, "landers the proceeds of cybercrimes.". While Tornado Cash is used by some people just as legitimate way to protect their privacy, the government says it fosters illicit activity, including "facilitation of heists, ransomware schemes, fraud and other cybercrimes."

Now, some Tornado Cash main developers have been arrested causing a lot of controversy in the crypto space. Although the main developers are still in custody, the protocol is still operated by the community due to its decentralized, open-source philosophy.
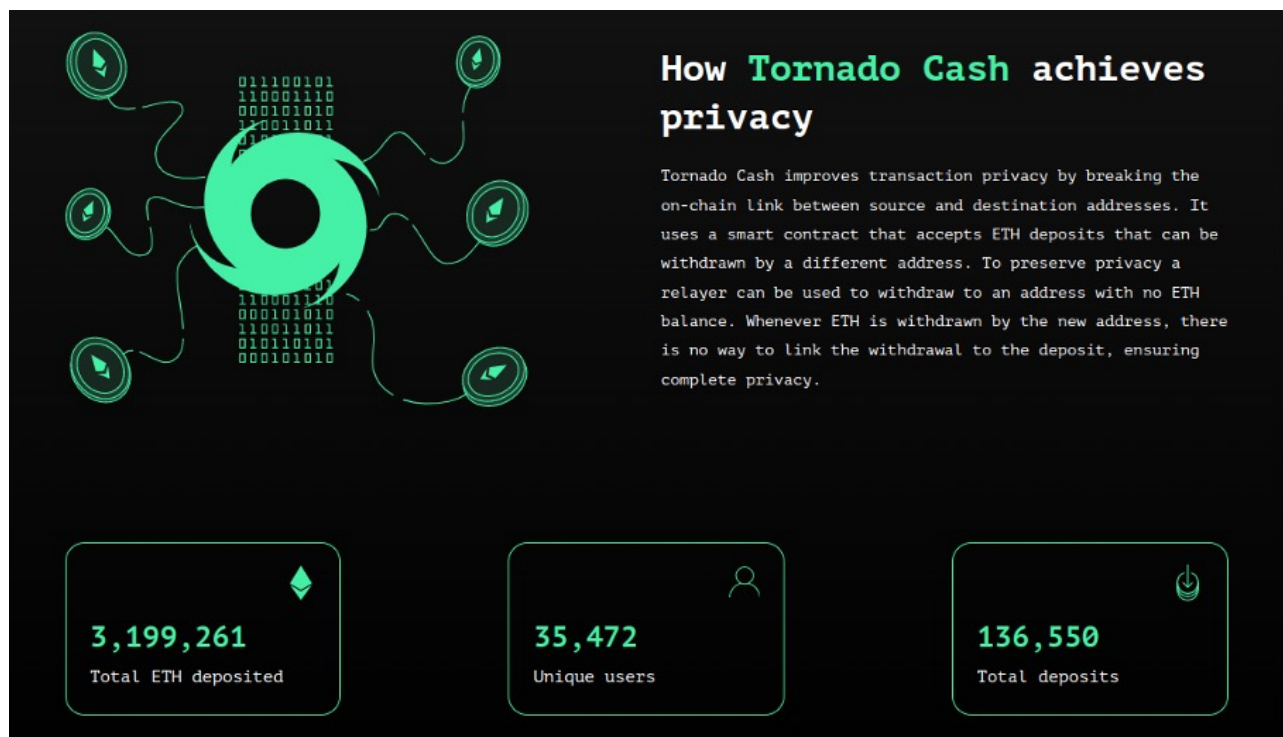

How Tornado Cash achieves privacy

Tornado Cash improves transaction privacy by breaking the on-chain link between source and destination addresses. It uses a smart contract that accepts ETH deposits that can be withdrawn by a different address. To preserve privacy a relayer can be used to withdraw to an address with no ETH balance. Whenever ETH is withdrawn by the new address, there is no way to link the withdrawal to the deposit, ensuring complete privacy.

| 3,199,261 | 35,472 | 136,550 |
| Total ETH deposited | Unique users | Total deposits |

Figure 1 - Image from the recently closed Tornado Cash website

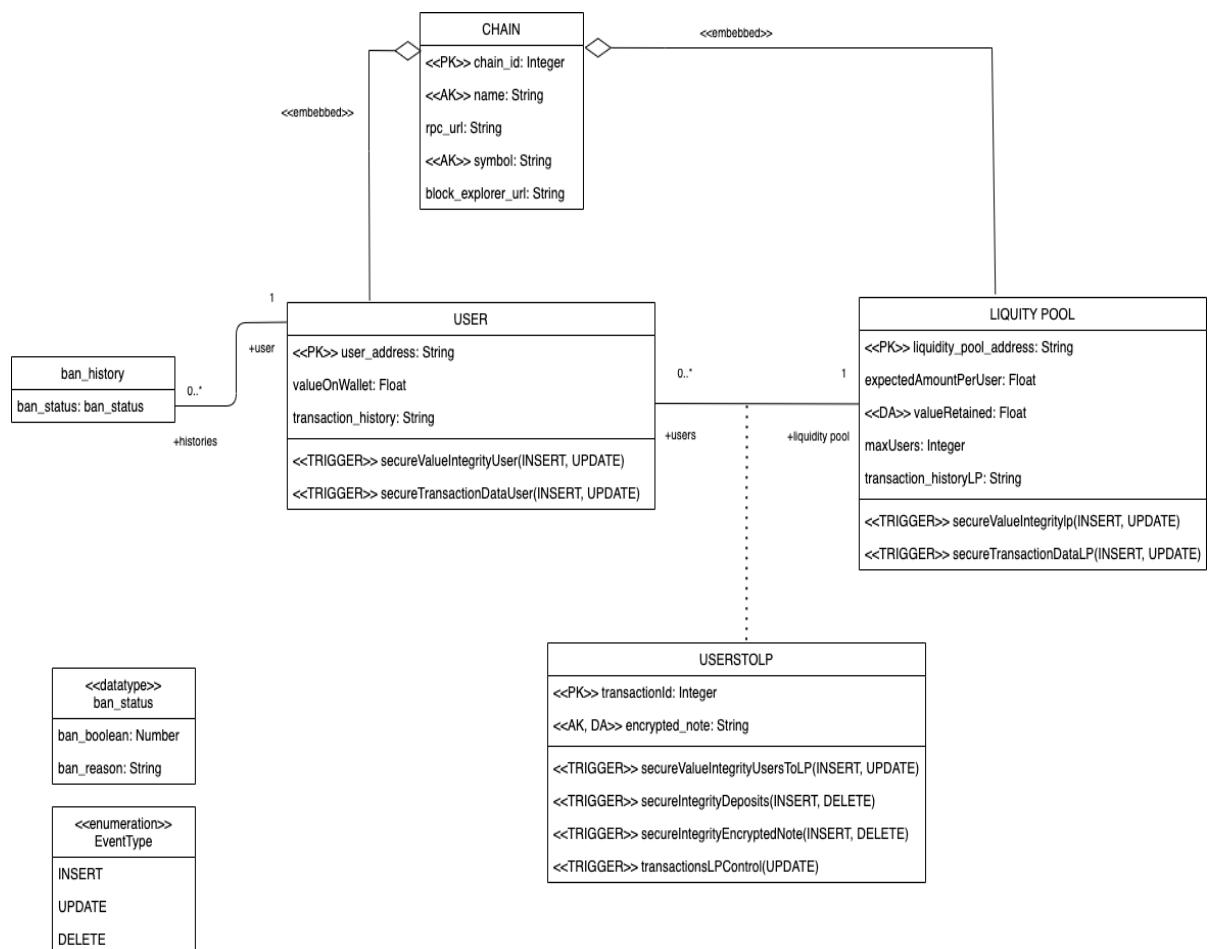$\Rightarrow$ Design of the Database

Figure 2 - UML Diagram

◊ Tables

- Chain: This table stores information about blockchain networks. It has the following fields:
  - chain_id: A unique identifier for the blockchain network.
  - chain_name: The name of the blockchain network.
  - rpc_url: The URL of the remote procedure call (RPC) endpoint for the blockchain network.
  - symbol: The symbol used to represent the cryptocurrency on the blockchain network (e.g., BTC for Bitcoin).
  - block_explorer_url: The URL of the block explorer for the blockchain network.

- LiquidityPool: This table stores information about liquidity pools. It has the following fields:
    - lp_address: The address of the liquidity pool on the blockchain network.
    - expectedAmountPerUser: The expected amount of cryptocurrency that each user will receive from the liquidity pool.
    - valueRetained: The value retained by the liquidity pool.
    - maxUsers: The maximum number of users that the liquidity pool can support.
    - transaction_history: A JSON-formatted field that stores the transaction history of the liquidity pool.
    - chain_idFK: A foreign key that references the chain_id field in the Chain table.

- Users: This table stores information about users. It has the following fields:
    - user_address: The address of the user on the blockchain network.
    - valueOnWallet: The value of cryptocurrency that the user has on their wallet.
    - transaction_history: A JSON-formatted field that stores the transaction history of the user.
    - chain_idFK: A foreign key that references the chain_id field in the *Chain* table.

- UsersToLP: This table stores information about the relationships between users and liquidity pools. It has the following fields:
    - transactionId: A unique identifier for the transaction.
    - encrypted_note: An encrypted note associated with the transaction.
    - user_addressFK: A foreign key that references the user_address field in the Users table.
    - lp_addressFK: A foreign key that references the lp_address field in the LiquidityPool table.

◊ Relations:

  ◊ *Chain-User* & *User-Liquidity Pool*
      The relations which include the table *Chain* symbolize the way transactions in real world blockchains are done. In the existing crypto world, there are a huge number of layers 1 blockchain, each with its own system of securing data, verifying transactions, and executing smart contracts.

In UML terms, it is made two one-to-many connections: one *Chain* has many *User*s, and one Chain has many *Liquidity Pools*. These relations are represented by the foreign key attribute *chain_idFK* present in each table with the "many" sides of the relation.
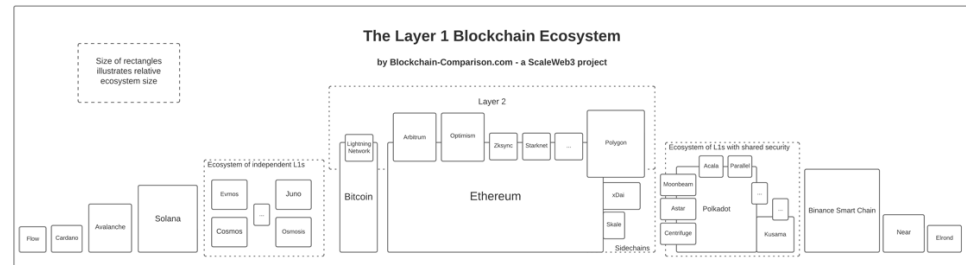

Figure 3 - Existing Layer 1 Blockchains and its ecosystems

◊ *Users-Liquidity Pool*

This relation represents the trading of assets between a user of the protocol and a certain *Liquidity Pool* with all the traits and procedures from the security policy of the protocol and the blockchain itself.

When the *Users* sends assets to the *Liquidity Pool*, an encrypted note is generated and attributed to the user for further claiming of the assets after the mixing is completed.

In the UML diagram, the complexity of the procedure justifies the creation of a new entity *UsersToLP* containing the attributes associated to the relationship: the transaction identifier and the encrypted note for the *Users* to reclaim the assets.
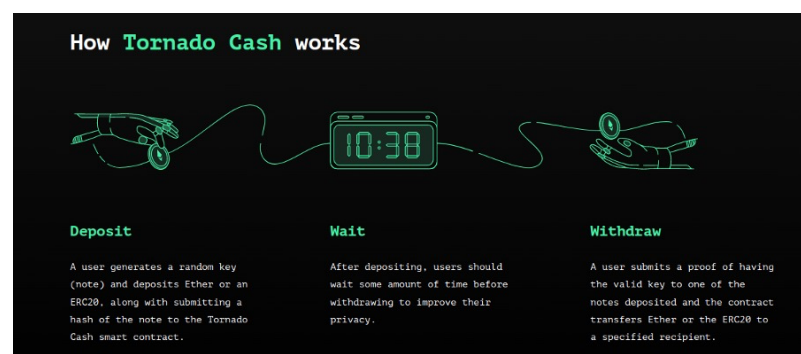

Figure 4 - User interaction with the protocol

◊ *Users-BanHistory*

Most decentralized finance and privacy protocols have implemented to monitor user activity and act if users aren't using the protocol's features accordingly. For that, there should be a mechanism that stores the banned users and prevent them from using the protocol.

In the UML diagram, there is a relation one-to-one with the *Users* and *Rating* symbolizing one user can be either unbanned, being able to use the all the features, or banned, unable to use any features of the protocol. Since it is a one-to-one relation, there's no need to add any type of foreign key or extra entity.
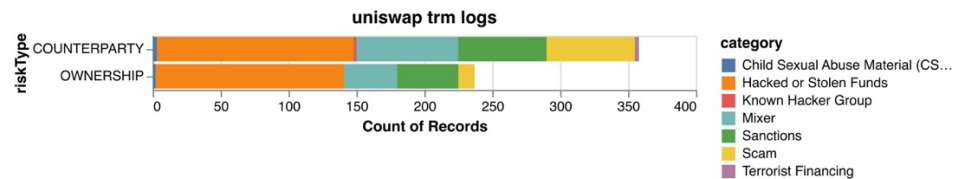

Figure 5 - Uniswap DEX control system statistics

⟹ PL/SQL Implementation

1. Persistent Stored Modules

One of the key aspects of the protocol is to assure the transactions between *Users* and *Liquidity Pool* are done smoothly and securely, despite all the complexity of the process, has said previously.

PSM in relational and SQL terms, are the best option to operate the core of this interaction due to the strong software modularity and reusage. In a real-world use scenario of the application, with hundreds of users actively sending and receiving assets, there should be a good management and reusage of resources in order to sustain the application.

PSM related methods implemented:
- *getUserWalletBalance* (Function)

   Given a *user_address* from a user, it obtains the *valueOnWallet* on a table with the *user_address* equal to the one passed as argument.

```
14    SELECT GETUSERWALLETBALANCE('0xbb6ba66A466Ef9f31cC44C8A0D9b5c84c49A4ba8') FROM DUAL;
```

Query Result | Script Output | DBMS Output | Explain Plan | Autotrace | SQL History

🗑 ⓘ ☑ Download ▾ Execution time: 0.002 seconds

	GETUSERWALLETB
1	100

Figure 6 - Execution of getUserWalletBalance

- *getNumberUsersLP* (Function)

  Given a *lp_address* from a *Liquidity Pool*, returns the number of users associated. The counting is done by checking if a table *UsersToLp* contains the address passed in the argument section as foreign key.

```
13    SELECT GETNUMBERUSERLP('0xbb6ba66A466Ef9f31cC44C8A0D9b5c84c49A4bb1') FROM DUAL;
```

Query Result | Script Output | DBMS Output | Explain Plan | Autotrace | SQL History

🗑 ⓘ ☑ Download ▾ Execution time: 0.012 seconds

	GETNUMBERUSERL
1	1

Figure 7 - Execution of getNumberUsersLP

- *joinLiquidityPool* (Procedure)

  Given a *user_address* from a user and a *lp_address* from a *Liquidity Pool* which the user pretends to join, it associated both entities if some conditions are respected:
    - If the limit of the *Liquidity Pool* is not exceeded.
    - If the given user is not already associated with another *Liquidity Pool.*

  If the conditions are met, a table *UsersToLP* suffers changes on the *user_addressFK* and *lp_addressFK* attributes to the ones passed as arguments.

```
10   BEGIN
11     JOINLIQUIDITYPOOL('0xbb6ba66A466Ef9f31cC44C8A0D9b5c84c49A4ba4', '0xbb6ba66A466Ef9f31cC44C8A0D9b5c84c49A4bb1');
12   END;
13
14   SELECT * FROM USERSTOLP;
15
16
17
```

	TRANSACTIONID	ENCRYPTED_NOTE	USER_ADDRESSFK	LP_ADDRESSFK
1	1	(null)	0xbb6ba66A466Ef9	(null)
2	2	.339823918905075!	0xbb6ba66A466Ef9	0xbb6ba66A466Ef9

Figure 8 - Execution of joinLiquidityPool

- *leaveLiquidityPool* (Procedure)

Given a *lp_adress* from a *Liquidity Pool,* after the mixing was made, the users involved should the dissociated. Before the procedure gets triggered, the number of users associated must be checked. If the number obtained doesn't match the maximum capacity, it means the mixing did not occur yet and the users cannot dissociate otherwise, the procedure gets triggered, and the dissociation happens.

```
1   UPDATE LIQUIDITYPOOL SET maxUsers = 1 WHERE LP_ADDRESS = '0xbb6ba66A466Ef9f31cC44C8A0D9b5c84c49A4bb1';
2
3   BEGIN
4     LEAVELIQUIDITYPOOL('0xbb6ba66A466Ef9f31cC44C8A0D9b5c84c49A4bb1');
5   END;
6   |
7   SELECT * FROM USERSTOLP;
8
9
```

	TRANSACTIONID	ENCRYPTED_NOTE	USER_ADDRESSFK	LP_ADDRESSFK
1	1	(null)	0xbb6ba66A466Ef9	(null)
2	2	(null)	0xbb6ba66A466Ef9	(null)

Figure 9 - Execution of leaveLiquidityPool

2. Triggers

Since the database receives data related to real-life notions such as money, there must be integrity constraints for data related to both internal application interactions and external data. An example would be that there cannot be a negative balance in a user's wallet or value held in a liquidity pool. To solve this issue, triggers are the best mechanism to deal with these types of conditions. The triggers referenced in this section are not related with other features like the explicit use of OLAP or JSON.

Triggers implemented:

- *secureValueIntegrityUser*

    Designed to prevent the insertion or update of a negative value in the *valueOnWallet* column of the *Users* table.

    Defined to fire "before" an insert or update operation occurs on the *Users* table, and it operates on a "per row" basis. This means that the trigger will be executed once for each row that is being inserted or updated.

    Contains an IF statement that checks the value of the *valueOnWallet* column in the new row being inserted or updated. If the value is less than zero, then the trigger raises an application error using the "raise_application_error" function. This will cause the insert or update operation to fail and roll back, and it will also return an error message to the user indicating that the transaction was not completed due to an insufficient balance.
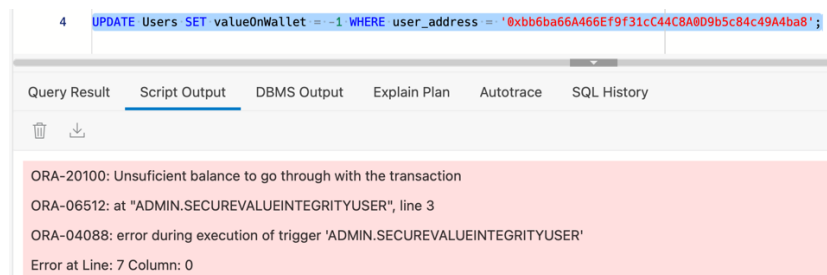

Figure 10 - Warning from secureIntegrityUser

- *secureValueIntegrityLP*

    Follows the same approach as the previous trigger (*secureValueIntegrityUser*) however prevents insertions or updates on the *valueRetained* column of the *Liquidity Pool* table.
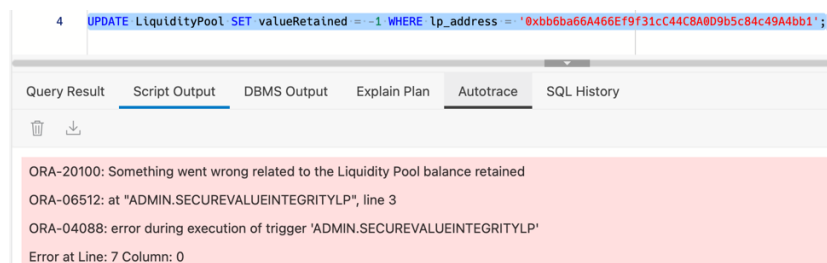

Figure 11 - Warning from secureValueIntegrityLP

- *secureIntegrityDeposists*

  Ensure that the only way possible to change the *user_addressFK* and *lp_addressFK* columns which represent the association is be done by the procedures *joinLiquidityPool* and *leaveLiquidityPool.*

  In this case, the trigger is defined to execute INSTEAD OF an insert statement on the view. This means that the trigger is executed in place of the insert statement, and it can modify the data being inserted into the underlying table directly. Without the view, it would not be possible to use an INSTEAD OF trigger to modify the data being inserted into the *UsersToLP* table.

Figure 12 - Users after the trigger of secureIntegrityDeposits

- *secureIntegrityWithdrawals*

  Following the same logical thinking of the previous trigger, the idea is to ensure that the relation between the two entities which exchange transactions do not get changed artificially. In this specific case, ensure that a user does not get diassociated artificially from a *Liquidity Pool*.

  Inside the trigger, a BEFORE statement is used to prevent the loss of integrity.

Figure 13 - Warning from secureIntegrityWithdrawals

- *secureIntegrityEncryptedNote*

    Ensure that the only way possible to change the *encrypted_note* column is done by procedures *joinLiquidityPool* and *leaveLiquidityPool.*

    Inside the trigger, there is an IF statement that checks if the trigger was fired because of an insert or delete statement. If either of these conditions is true, the trigger raises an application error with an error code of -20100. This error will stop the insert or delete from being performed, and the database will return the error message to the client that issued the insert or delete statement.
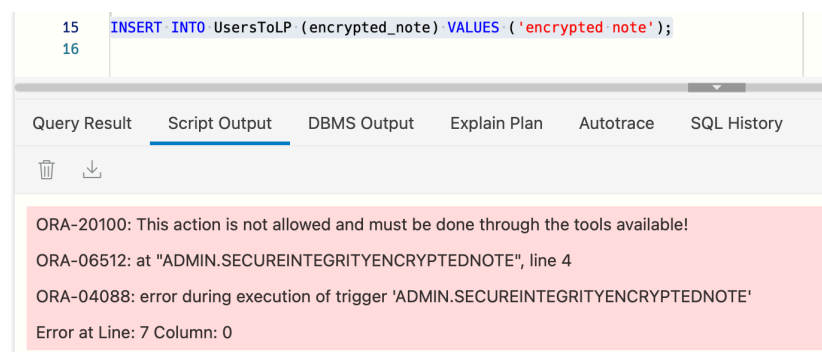

Figure 14 - Warning from secureIntegrityEncryptedNote

- *transactionsLPControl*

    Made to secure the normal inflow and outflow of money in the LiquidityPool.

    If a user join a *Liquidity Pool*, he make a deposit which means the money has to come out of his wallet and enter the value retained on the Liquidity Pool treasury. Same case as if a user leaves the *Liquidity Pool* and takes out his money. The value deposited previously must return to the user balance sheet and leave the LP treasury.

    Inside the trigger, it starts by checking if the value of the column *lp_addressFK change though* an IF statement. If the condition checked is true, then there was an interaction between the two main entities. Then there are two possible scenarios:

- If the new value in the column *lp_addressFK* is not null, it means that the user was associated with a *Liquidity Pool* and a deposit of assets happened. To complete the deposit, first, the amount deposited is obtained though the SELECT statement that retrieves the value of the *expectedAmountPerUser* column from the *Liquidity Pool* table. Second, an UPDATE statement is used to subtract the value recently obtained back to the *valueOnWallet* column on *User* table and add to the *valueRetained* on the *Liquidity Pool* table.

- If the new value in the column *lp_addressFK* is null, it means that the user was disassociated from the *Liquidity Pool* and a withdraw of assets happened. The withdraw process is like the deposit however the value obtain in the SELECT statement is used to add to the *valueOnWallet* (*User* table) and to subtract to the *valueRetained* (*Liquidity Pool*).

	USER_ADDRESS	VALUEONWALLET	TRANSACTION_HIS	CHAIN_IDFK
1	0xbb6ba66A466Ef9	4.2	{ "Hash": "0x68ea9t	1

Figure 15 - Specific user before joining the LP

	USER_ADDRESS	VALUEONWALLET	TRANSACTION_HIS	CHAIN_IDFK
1	0xbb6ba66A466Ef9	4.0	{ "Hash": "0x68ea9t	1

Figure 16 - Data from the same user after the transactionsLPControl was triggered

3. Advanced Operators (including OLAP)

◊ Analytics

- First query displays the users situated in each *Chain* ranked by the *valueOnWallet* column contained in each *Users* table, using the *RANK* operator.

```
1   SELECT user_address, valueOnWallet, chain_idFK,
2       RANK() OVER (PARTITION BY user_address ORDER BY valueOnWallet) RANK
3       FROM Users
4       ORDER BY chain_idFK, valueOnWallet DESC;
```

	USER_ADDRESS	VALUEONWALLET	CHAIN_IDFK	RANK
1	0xbb6ba66A466Ef9	100	1	1
2	0xbb6ba66A466Ef9	4.2	1	1
3	0xbb6ba66A466Ef9	35	10	1
4	0xbb6ba66A466Ef9	5	10	1

Figure 17 - Execution of the query

- Second query displays the total value of assets contained in each *Chain* (*valueOnWallet* and *valueRetained* included), using the *SUM* operator.

```
1   SELECT c.chain_name, SUM(u.valueOnWallet) AS user_value, SUM(lp.valueRetained) AS lp_value,
2       SUM(u.valueOnWallet + lp.valueRetained) AS total_value
3   FROM Chain c
4   JOIN Users u ON c.chain_id = u.chain_idFK
5   JOIN LiquidityPool lp ON c.chain_id = lp.chain_idFK
6   GROUP BY c.chain_name;
```

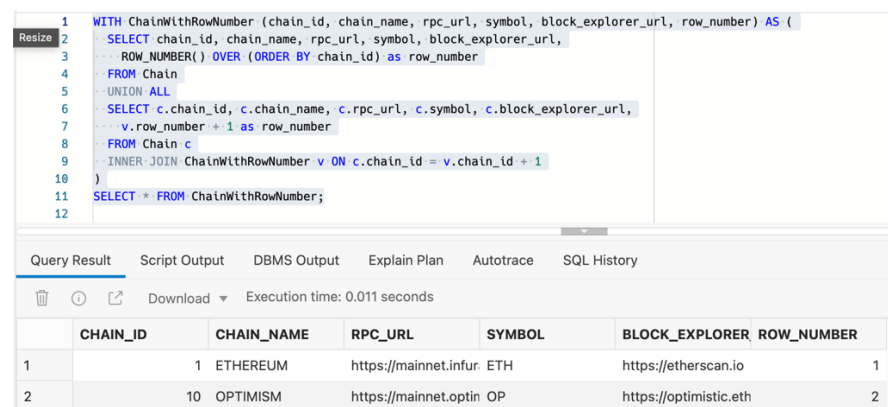|	CHAIN_NAME	USER_VALUE	LP_VALUE	TOTAL_VALUE
1	ETHEREUM	8.4	1.1	9.5

Figure 18 - Execution of the query

- Third recursive query displays the total data in each *Chain* recursively, using the *WITH* and *ROW_NUMBER* statements. The first SELECT statement retrieves all rows from the "Chain" table, and generates row numbers for each row using the ROW_NUMBER() function. The row numbers are assigned based on the order of the "chain_id" column.
The second SELECT statement retrieves all rows from the "Chain" table that have a "chain_id" value that is greater than the "chain_id" value of

the current row by 1. The row numbers for these rows are generated by adding 1 to the row number of the current row.

Normally, the *chain_id* value from each chain tends to not follow a sequence of "+1" since there are lots of chains operating now and the id picking system various a lot.

This view was made recursively only for showcase purposes. In this theme environment, there aren't many cases where recursive views would be the most efficient and correct way. The most efficient way to do the view was with a simple *SELECT* statement, even if there wasn't a counter enumerating the rows.

Figure 19 - Execution of the recursive query

◊   JSON

For JSON implementation, as shown in the tables section, the tables *Users* and *Liquidity Pool* have a transaction history column based on JSON and a constraint that checks valid JSON insertions.

In addition, two additional triggers were implemented to check the integrity of the data inserted. The triggers are:

- *secureTransactionDataUser*

The idea of the trigger is to check if the user address present in the table correspondent is the same as the address present in the "Source" section in the JSON section when inserting data.

The SELECT statement retrieves the address present in the given transaction history into a variable. Then, the variable gets compared with the user address present on the table though an IF statement.

```
16  UPDATE Users SET transaction_history = '{
17  ····"Hash": "0x68ea9bf0294e78fc85d5fb36eefc4f4885476734be834822f71463c8b4d7a7ef",
18  ····"Status": "Sucess",
19  ····"Block": 14782178,
20  ····"Timestamp": "May-15-2022 08:57:30 PM +UTC",
21  ····"Source": "0xbb6ba66a466ef9f31cc44c8a0d9b5c84c49a4bm9",
22  ····"Destination": "0x6399c842dd2be3de30bf99bc7d1bbf6fa3650e70",
23  ····"Value": 0.00,
```

Query Result | Script Output | DBMS Output | Explain Plan | Autotrace | SQL History

ORA-20100: The data being inserted is not coerent with the existing data

ORA-06512: at "ADMIN.SECURETRANSACTIONDATAUSER", line 7

ORA-04088: error during execution of trigger 'ADMIN.SECURETRANSACTIONDATAUSER'

Error at Line: 7 Column: 0

Figure 20 - Warning from JSON trigger secureTransactionDataUser

- *secureTransactionDataLP*

    This trigger follows the same idea as the previous trigger presented however it uses the *Liquidity Pool* address to assure the conditions are met, instead of the user address.

```
15  UPDATE LiquidityPool SET transaction_history = '{
16  ····"Hash": "0x5ca21bdbc9b261bc5e6e91e3d9c11c332cd2d4c331e76f192cd93ca2b0c330fh",
17  ····"Status": "Sucess",
18  ····"Block": 15528244,
19  ····"Timestamp": "Sep-13-2022 05:33:55 PM +UTC",
20  ····"Source": "0xbb6ba66A466Ef9f31cC44C8A0D9b5c84c49A4l6",
21  ····"Destination": "0xbb6ba66A466Ef9f31cC44C8A0D9b5c84c49A4ba4",
22  ····"Value": 0.45848,
23  ····"Fee": {
```

Query Result | Script Output | DBMS Output | Explain Plan | Autotrace | SQL History

ORA-20100: The data being inserted is not coerent with the existing data

ORA-06512: at "ADMIN.SECURETRANSACTIONDATALP", line 7

ORA-04088: error during execution of trigger 'ADMIN.SECURETRANSACTIONDATALP'

Error at Line: 7 Column: 0

Figure 21 - Warning from JSON trigger secureTransactionDataLP

4. Object Extension

    For this section, it was implemented a ranking system inspired in the securities systems as mentioned before.
    The users_type type represents an object that contains information about a user, including their address, the value of their wallet, their transaction history, and a foreign key referencing another table (presumably a table containing information about chains).

The rated_user type represents an object that contains information about a user who has been rated, including their address, a flag indicating whether they have been analyzed, and a reference to a users_type object containing additional information about the user.

The rated_users type represents a table of rated_user objects.

The Rating table has two columns for storing lists of rated_user objects: one for unbanned users and one for banned users. These columns are defined as nested tables, which means that they are stored as separate tables in the database. The code also creates indexes on the rated_user_address column in both of these nested tables.

Overall, it seems that the goal of the code is to create a schema for storing information about users, their ratings, and their transaction histories in an Oracle SQL database.

⇒ Conclusion

Overall, this Oracle database was designed to store and track information related to a specific protocol or application. It includes tables for storing various types of data, as well as functions and procedures for interacting with and manipulating the data. The database also includes constraints to ensure the integrity and validity of the data being stored. The specific purpose and functionality of the database can be determined by examining the specific tables, functions, and procedures that are included in the code. The aim of achieving similar features to the basic features of the *Tornado Cash* protocol was successful.

⇒ Sources

https://blockchain-comparison.com/blockchain-protocols/

https://blockworks.co/news/wallets-banned-by-uniswap-labs-for-alleged-crimes

https://tornadocash.eth.link

https://web.archive.org/web/20220201170424/https://tornadocash.eth.link/

https://www.oracle.com (DB Engine – Oracle Autonomous Database on Oracle Cloud)