Transaction Indexing Service HLD

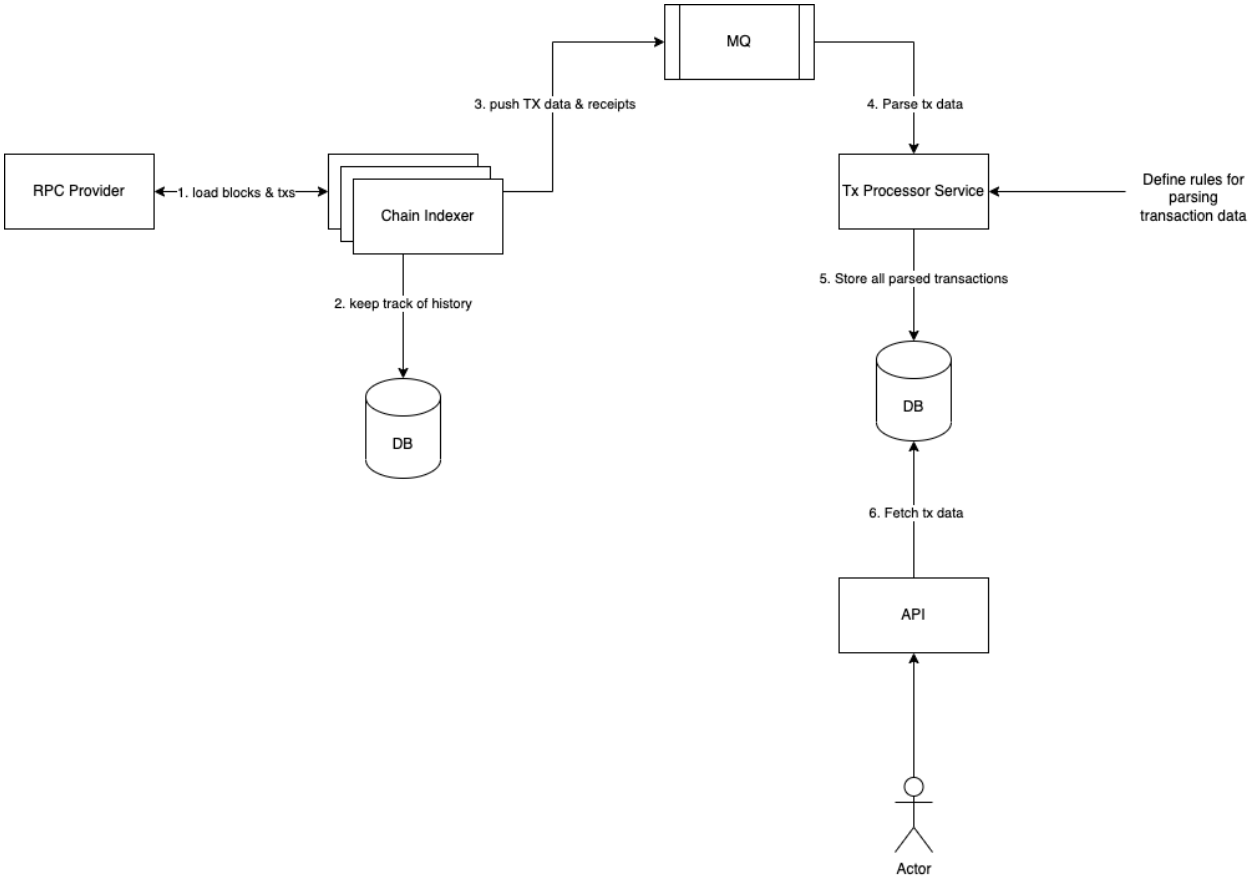
This document outlines the requirements and a proposed HLD design for a transaction indexing service.

Requirements:

- Scans multiple EVM-based chains and store all transactions in a database. The scan must be reliable and without a significant lag (though tolerable based on finality).
- Has the ability to parse native transfers and other types of transactions (ex. ERC-20 transfer or other protocol events, ex. AddLiqudity on Uniswap etc)
- Allows specifying which address activity to track via API
- Allows fetching address activity via API

Proposed architecture:

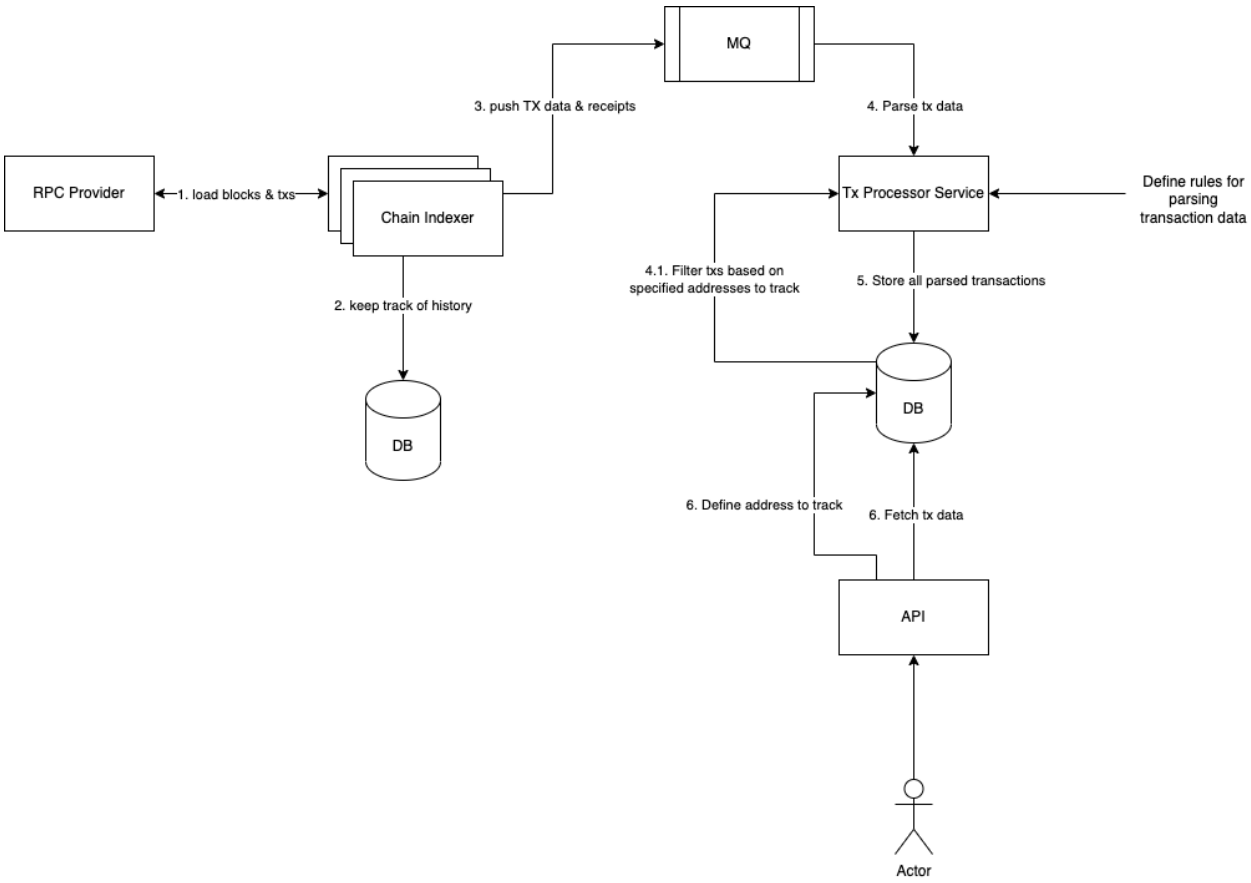
The first option of the proposed architecture includes scanning and storing all transactions in the database.



The detailed flow is given below:

1. We will use Kubernetes to deploy the service. We will also use cloud-based managed databases and queues, like AWS Aurora and Kafka/RabbitMQ, which are highly reliable.
2. The **Chain Indexer** service will be deployed as a single pod per chain (when running Nodejs). This means that the code will be the same while we will have multiple pods running for the same service, where the chain will be specified as an env variable.
    a. This service listens to new blocks, loads the transaction receipts (for chains that support it, it loads block receipt, otherwise loads transaction receipts)
    b. After loading the receipt, it is pushed to the message queue for further processing.
    c. After the block is processed, it's stored in the database. This service uses the database to keep track of the **latest processed block**.
    d. On startup, this service checks whether there is a difference between currently latest block on chain and latest processed block. If there is, it fetches those blocks while listening to new blocks in parallel.
    e. This service is highly resilient in a way that if it goes down, or if the dababase/mq go down, Kubernetes will keep it in a restart loop (as the /health endpoint is tied to MQ & DB status), and when it's stabilized, it will automatically load previous blocks.
3. The **Tx Processor Service** can be a Node service, or a data processor pipeline defined in Apache Flink or Apache Beam (which perform parallel processing), though the idea is the same:
    a. Based on the defined transaction parsing rules, this service will parse the transactions and stored the parse transactions in the database.
    b. The transactions can be grouped in two categories - native transfers & token transfers, where the first category is parsed directly by the transaction data, while the second category is parsed based on logs using business specific logic (ex. ERC-20 transfers, specific protocol interactions etc)
    c. This service will serve as an adapter where we can define (in code) the transaction parsing rules. If it's deployed as a Flink/Beam pipeline, those rules can be defined as processing functions.
        i. An example processing rule is included in the POC for ERC-20 transfers.
4. The **API Service** will serve as a user-facing application where the users can query the transactions and the portfolio for a particular address.

The second option of the proposed architecture includes scanning all transactions and storing only transactions related to tracked addresses in the database. There are small differences with the previous approach which are highlighted.



The detailed flow is given below:

5. We will use Kubernetes to deploy the service. We will also use cloud-based managed databases and queues, like AWS Aurora and Kafka/RabbitMQ, which are highly reliable.
6. The **Chain Indexer** service will be deployed as a single pod per chain (when running Nodejs). This means that the code will be the same while we will have multiple pods running for the same service, where the chain will be specified as an env variable.
    a. This service listens to new blocks, loads the transaction receipts (for chains that support it, it loads block receipt, otherwise loads transaction receipts)
    b. After loading the receipt, it is pushed to the message queue for further processing.
    c. After the block is processed, it's stored in the database. This service uses the database to keep track of the **latest processed block**.

    d. On startup, this service checks whether there is a difference between currently latest block on chain and latest processed block. If there is, it fetches those blocks while listening to new blocks in parallel.
    e. This service is highly resilient in a way that if it goes down, or if the dababase/mq go down, Kubernetes will keep it in a restart loop (as the /health endpoint is tied to MQ & DB status), and when it's stabilized, it will automatically load previous blocks.
7. The **Tx Processor Service** can be a Node service, or a data processor pipeline defined in Apache Flink or Apache Beam (which perform parallel processing), though the idea is the same:
    a. This service will first filter out the transactions only for the addresses specified in the database (by the users).
        i. This means that we will store in the DB only the transactions for the addresses that are tracked (including both from & to)
        ii. This service will load the addresses that are tracked and store them in memory. It can easily store millions of addresses in a few MBs. Another option for a more distributed approach is to store those in Redis (though it's not included in the diagram).
    b. Based on the defined transaction parsing rules, this service will parse the transactions and stored the parse transactions in the database.
    c. The transactions can be grouped in two categories - native transfers & token transfers, where the first category is parsed directly by the transaction data, while the second category is parsed based on logs using business specific logic (ex. ERC-20 transfers, specific protocol interactions etc)
    d. This service will serve as an adapter where we can define (in code) the transaction parsing rules. If it's deployed as a Flink/Beam pipeline, those rules can be defined as processing functions.
        i. An example processing rule is included in the POC for ERC-20 transfers.
8. The **API Service** will serve as a user-facing application where the users can query the transactions and the portfolio for a particular address.
    a. The users will be able to specify which addresses they want to track via the API, and those addresses will be upserted in the DB, and then the **Tx Processor Service** will refresh that data in memory.

Notes:

- The proposed design uses SQL-based database. A more preferred solution for faster queries (from the API) would be to use a database like ElasticSearch which can quickly index large amounts of data and would make search quite fast.