Sieve: A lightweight real-time data streaming engine with an expressive DSL for Ethereum & The Superchain.

Ayodeji Akinola
akinayodeji4all@gmail.com

December 30, 2024

1 Motivation

Ethereum blockchain ecosystem has evolved beyond single-chain architectures, with Ethereum and its layer 2 solutions (the Superchain) processing substantial data volumes across multiple networks. Ethereum processes blocks approximately every 12 seconds, with each block containing up to 10MB of data. Layer 2 solutions like Unchain, Ink, and Optimism process additional data at varying frequencies. While existing blockchain data solutions offer various querying options, they often face challenges with real-time data processing and cross-chain compatibility. The complexity lies not in the query languages themselves, but in efficiently handling high-frequency updates, maintaining low latency across multiple networks, and correlating events across different chain architectures.

Key Reasons:

1. **Real-Time Processing:** Applications like MEV searchers, trading systems, and block builders require immediate access to specific blockchain events. The streaming architecture processes blocks as they arrive, delivering relevant data with minimal latency.

2. **Memory Efficiency:** By processing data streams in-memory without persistence, Sieve minimizes resource usage while maintaining high performance. The filter engine performs efficient property-based filtering using configurable conditions to process transaction data in real-time.

3. **Cross-Chain Compatibility:** With the growing ecosystem of L2s and sidechains, Sieve provides a unified interface for filtering data across multiple chains, abstracting the complexity of different protocols and endpoints.

4. **Developer Experience:** The ORM-like DSL provides an intuitive interface for defining complex filter conditions, making it accessible.

2 System Architecture

Sieve is a high-performance streaming engine that connects directly to Ethereum & multiple L2s called the Superchain (like *Base*, *Unchain*, *Ink*, and *Optimism*). Unlike traditional blockchain data solutions that store and index data, Sieve operates purely as a streaming service, acting as a real-time filter between blockchain networks and applications.

Core Principles:

1. Pure streaming with no persistence layer
2. Configurable filtering at the ingestion point
3. Unified interface across multiple chains

Critical Use Cases:

- *MEV searchers* can instantly detect profitable opportunities across multiple chains
- *Trading firms* can monitor specific contract events without processing irrelevant data
- *Block builders* can efficiently track and analyze cross-chain transaction patterns

By filtering at ingestion, unnecessary data never reaches the consumer, providing these stakeholders with exactly the data they need, when they need it, without the overhead of processing irrelevant information or managing complex infrastructure.

Architecture:

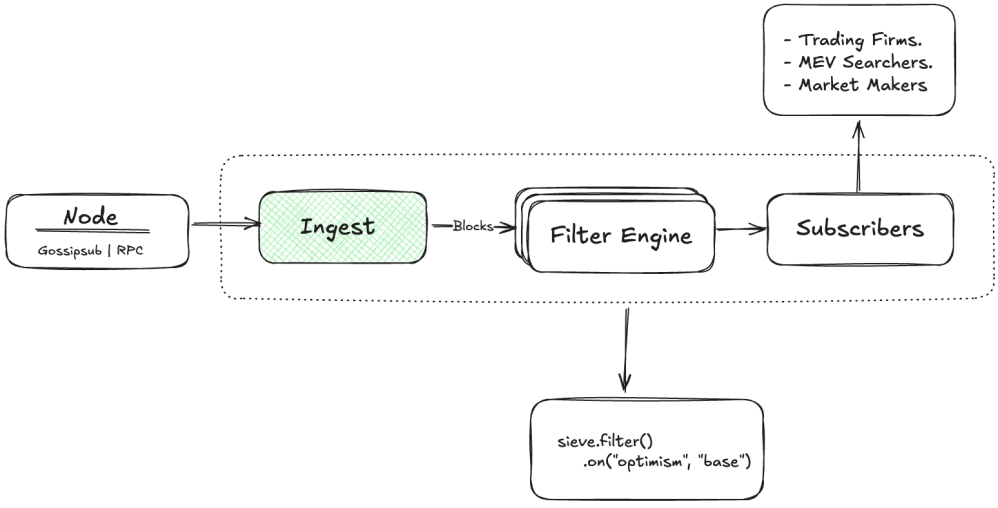

Figure 1: High-level Sieve Architecture

3 Streaming Layer

The streaming layer forms the foundation of Sieve, responsible for ingesting blockchain data from multiple chains through both RPC and Gossipsub protocols. This layer ensures reliable, ordered delivery of block and transaction data to the filtering system.

It is composed of three main components that work together to provide a reliable block & transaction stream:

- Network layer
- Connection Orchestrator
- Ingestion Pipeline

Data Flow:
[Network Layer] → [Connection Orchestrator] → [Ingestion Layer] → [Filtering Engine]

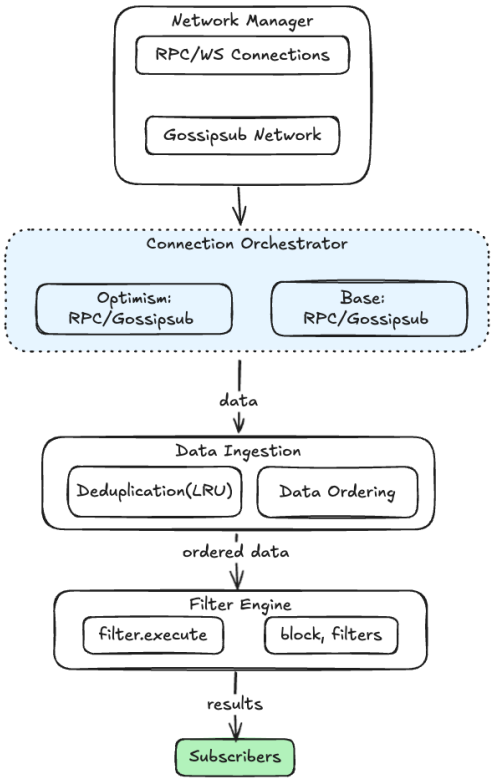

Figure 2: Streaming Pipeline to subscribers.

3.1 Network Layer

This foundational layer provides the core implementation for connecting to blockchain nodes via both RPC and Gossipsub protocols. It's a pure connection handler that takes URLs and peer information as input and manages the connection lifecycle. The layer is protocol-aware

but chain-agnostic, focusing solely on maintaining stable connections and providing raw data streams. It handles connection establishment, reconnection logic, and basic error handling for both RPC (HTTP/WebSocket) and Gossipsub protocols.

3.2 Connection Orchestrator

The Connection Orchestrator serves as the coordination layer between raw node connections and data processing. It maintains a registry of chain configurations and creates appropriate Network instances for each chain. This layer abstracts away the complexity of managing multiple chains and their connections, providing a unified interface for the Ingestion Layer. It handles the lifecycle of Connections, coordinates data flow from multiple chains, and provides the first level of stream aggregation.

3.3 Ingestion Layer

Sitting at the top of the stack, the Ingestion Layer receives aggregated data streams from the Connection Orchestrator and transforms them into a clean, unified data feed. It handles cross-chain block deduplication and manages any gaps in block sequences. This layer provides a simple `subscribe_stream()` interface that consumers can use to specify which chains they want to monitor (e.g., `ingest.subscribe_stream(Chain.Optimism | Chain.Ethereum)`), abstracting away all the complexity of multi-chain data ingestion and processing.

4 Filter Engine

The filter engine is a high-performance, memory-efficient system processing concurrent filters across blockchain data streams. It combines predicate matching with optimized execution strategies for handling thousands of filters efficiently.

5 Logical Operations

Logical operations combine different conditions to filter events. They enable complex filtering patterns through basic and advanced operators.

Core Operators

- `AND`: All conditions must be true
- `OR`: At least one condition must be true
- `NOT`: Negate the condition

Advanced Operators

- `ANY_OF`: Group-based OR operations
- `ALL_OF`: Group-based AND operations
- `NONE_OF`: No conditions should be true

6 Implementation Examples

In our implementation we made use of builder pattern where we followed a callback-style because it's more flexible for dynamic filter creation, better for complex conditional logic, memory efficient (no vector allocation) and finally it allows for stateful filter building.

OR Operations

```
.or(|tx| {
     tx.value().gt(U256::from(1000));
     tx.gas_price().lt(50000);
})
```

AND Operations

```
.and(|tx| {
      tx.value().gt(U256::from(1000));
      tx.gas_price().lt(50000);
})
```

7 Comparison Operations

Value-based filtering operations:

- Basic: `eq`, `neq`, `gt`, `gte`, `lt`, `lte`
- Range: `between`, `outside`
- Pattern: `contains`, `matches`
- Existence: `exists`

Examples

eq - Equal to a specific value

```
.filter(|f| {
    f.event("Transfer")
     .value().eq(eth(100))  // Equal to 100 ETH
})
```

neq - Not equal to a specified value

```
.filter(|f| {
    f.value().neq(eth(1))  // Not equal to 1 ETH
})
```

8 Filter Tree Design

The filter is structured as a tree that represents logical conditions for matching blockchain data (transactions, blocks, events). The engine evaluates this tree structure to determine matches.

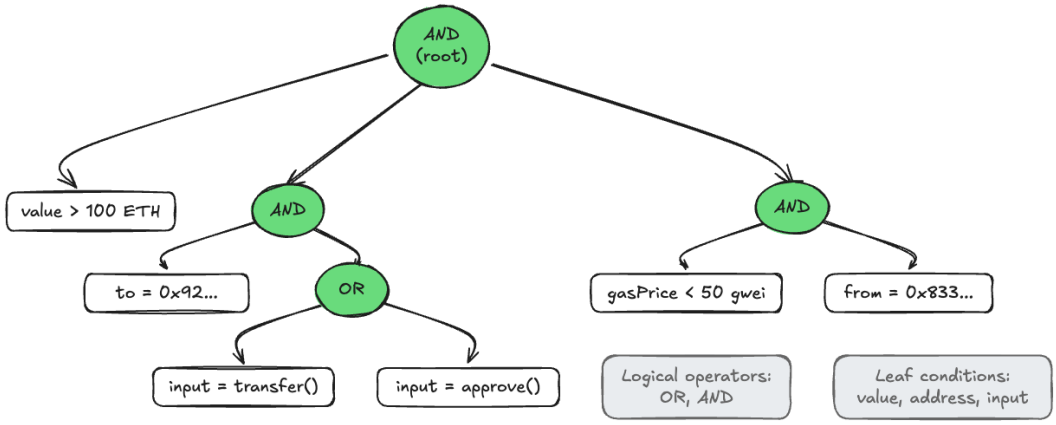

Figure 3: Complex logical Operation Tree

The diagram above demonstrates a hierarchical filter tree optimized for blockchain data evaluation. Each node represents either a logical operator (AND, OR) or a condition predicate (value comparisons, address matching, function signatures). The tree structure enables efficient evaluation through short-circuiting & parallel processing of independent subtrees. Conditions are organized to minimize computational overhead, with gas-efficient checks performed first and more expensive contract interactions deferred when possible.

9 Execution Model

A parallel evaluation system for filter conditions using Rayon workers. The engine evaluates two node types:

Leaf Node Evaluation

Direct evaluation of single conditions through a shared context. Conditions include value checks, address matching, and pattern validation against blockchain data.

Logical Node Evaluation

- AND: Parallel evaluation of all children, succeeds if all match
- OR: Parallel evaluation with early return on first match
- NOT: Inverts aggregate result of child evaluations
- XOR: Counts successful matches, ensures exactly one match

10 Optimization Techniques

The filter engine employs two key optimization techniques: **cost-based ordering** and **short-circuit evaluation**. These optimizations aim to minimize computation by performing cheaper operations first and terminating the evaluation as early as possible.

Cost-Based Ordering

The engine orders operations based on their computational cost and likelihood of execution. Think of checking a high-value DEX trade: we have two operations: checking the transaction value and analyzing the input data. Value checking is a simple numeric comparison that takes microseconds, while input analysis requires expensive decoding and pattern matching that might take hundreds of microseconds.

By checking value first, we can quickly eliminate transactions that do not meet the threshold without ever touching the input data. This is particularly effective since most blockchain transactions won't match our criteria.

Short-Circuit Evaluation

The engine optimizes OR conditions by prioritizing operations that are more likely to succeed and cheaper to compute. Consider a filter looking for either simple token transfers or complex decoding of call-data:

By prioritizing simple token transfer checks, which are both common and cheap to verify, we can often avoid the expensive data decoding entirely. This reordering preserves the logical meaning while minimizing the average computational cost. In practice, this means that a simple token transfer can be matched with a single quick check, while complex call-data analysis only occurs when necessary.

The combined effect of these optimizations means that in the best case, we only perform a single cheap comparison, and even in average cases, we rarely need to execute our most expensive operations.

Filter Node Execution Priority:

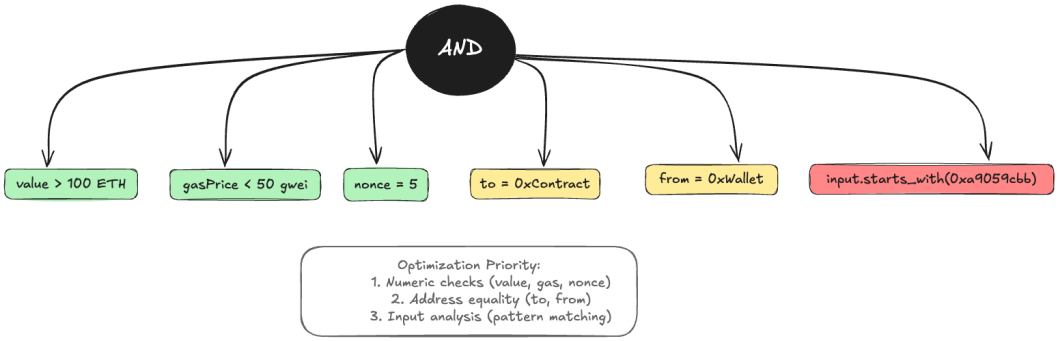

Figure 4: Transaction Filter Evaluation Priority and Ordering.

11 Caching Strategy

The engine maintains a global cache for decoded blockchain data in a dash-map (thread-safe) that persists across evaluation contexts. This single-layer caching system focuses on preventing redundant decoding operations.

When processing transactions, the engine:

- Decodes transaction data, call-data, and input data once
- Stores decoded results in the global cache
- Shares cached results across all active filter evaluations
- Prevents duplicate decoding of frequently accessed data

12 Future Implementation

- **Dynamic Event Listeners**: Enable the creation of short-lived listeners that are dynamically instantiated to monitor specific transactions, events, etc. For instance,

when a user sends 100 ETH on the base chain, a listener can be created instantly to track this event on the base or optimism network and react accordingly in real time.

- **Optimized Performance**: To support thousands of listeners across multiple chains, it's crucial to optimize how events are processed. This can be done by ensuring that each listener operates with minimal overhead, efficiently analyzing filter trees and filtering events, without introducing performance bottlenecks.

13 Known Challenges in Call Data Analysis

While analyzing ethereum call data, multiple approaches were tested to optimize data processing and improve developer experience. The following challenges emerged as significant pain points requiring careful consideration.

Calldata Decoding Complexity

- Runtime lacks parameter names, which requires developers to work with raw positional values.
- Nested structures intensify the issue, making it un-intuitive and error prone to work with low-level data.

Developer Experience Pain Points

- A deep understanding of ABI encoding is necessary to handle raw call data.
- Developers must manually track positions in nested structures, which adds to the cognitive load just to write filters.

Trade-offs in ABI Usage

- Full ABIs simplify development but are memory intensive.
- Omitting ABIs saves resources but significantly complicates the developer experience.
- Even middle-ground approaches require understanding low-level details, which can be challenging.

14 Conclusion

The system strikes a balance between I/O and CPU-bound operations through a carefully designed processing pipeline. Blocks are ingested through RPC endpoints and Gossipsub networks in an I/O-bound phase, transitioning to a CPU-intensive stage where transactions and logs are decoded and cached in a shared **Data View**. This caching strategy minimizes redundant decoding and ensures efficient data access for multiple filter threads.

To address the challenges of filtering, the execution engine employs a tree-structured approach, optimizing for early termination by evaluating filters from the least to the most computationally expensive. Using in-memory processing, property-based filtering, and a unified interface for cross-chain data, the system meets the growing demands of real-time applications in the Ethereum ecosystem. This solution bridges the gap between usability for developers and performance optimization, addressing the complexities of high-frequency updates and cross-chain compatibility.