Источники данных и анализ производительности DEFIMON Analytics Platform

Performance Engineering Team
DEFIMON Project

Август 2024

Contents

1 Введение

Данный документ представляет детальный анализ источников данных аналитической системы DEFIMON, включая оценку пропускной способности, нагрузки на CPU и технологий оптимизации производительности. Особое внимание уделено использованию Rust и библиотеки Tokio для снижения нагрузки на основной процессор.

2 Обзор источников данных

Система DEFIMON интегрируется с множественными источниками данных Web3 экосистемы для получения актуальной информации о DeFi протоколах, ценах токенов и метриках блокчейнов.

2.1 Классификация источников данных

Источник	Тип данных	Приоритет	Rate Limit	Задержка
The Graph	Subgraph данные	1 (высокий)	60 req/min	200-500ms
CoinGecko	Цены токенов	2 (средний)	50 req/min	100-300ms
DeFiLlama	TVL метрики	3 (низкий)	100 req/min	300-800ms
Alchemy	Ethereum RPC	1 (высокий)	300 req/sec	50-200ms
Infura	Backup RPC	2 (средний)	100k req/day	100-400ms
L2 Networks	Прямые RPC	1 (высокий)	Varies	100-1000ms
Cosmos RPC	Cosmos данные	2 (средний)	Varies	200-600ms
Polkadot RPC	Substrate данные	2 (средний)	Varies	150-500ms

Table 1: Источники данных и их характеристики

3 Диаграмма потоков данных и производительности

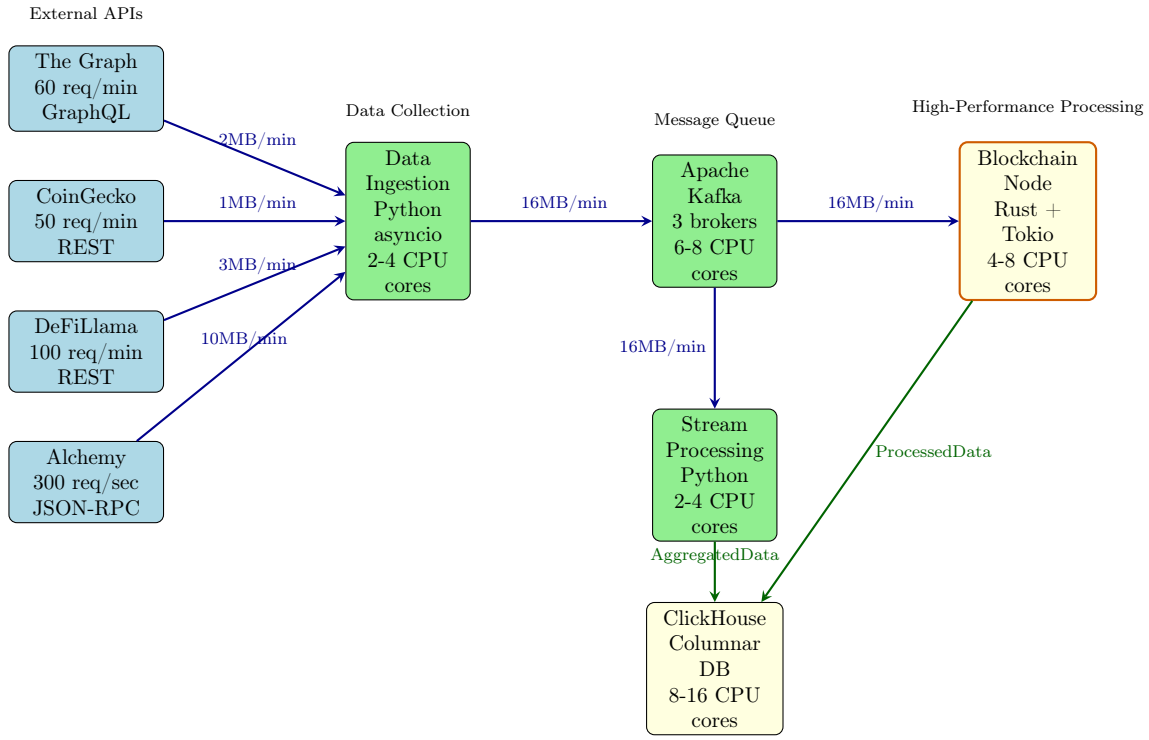

Figure 1: Диаграмма потоков данных с показателями производительности

4 Анализ пропускной способности

4.1 Расчет требований к полосе пропускания

Источник	Размер запроса	Размер ответа	Частота	Трафик/час
The Graph	2KB	50KB	60/min	187MB
CoinGecko	1KB	20KB	50/min	63MB
DeFiLlama	1KB	100KB	100/min	606MB
Ethereum RPC	500B	5KB	300/min	99MB
L2 Networks (8 сетей)	500B	10KB	800/min	504MB
Cosmos (12 сетей)	1KB	15KB	480/min	461MB
Polkadot (4 сети)	1KB	12KB	160/min	125MB
Итого				2.045GB/час

Table 2: Требования к входящему трафику

4.2 Внутренний трафик системы

- Kafka throughput: 16MB/min входящих данных
- ClickHouse ingestion: 50MB/min (с учетом агрегации)
- PostgreSQL writes: 5MB/min (метаданные)
- Redis cache: 20MB/min (кэширование)

- Межсервисное взаимодействие: 10MB/min

Рекомендуемая пропускная способность: 1Gbps для комфортной работы с запасом 2x.

5 Анализ нагрузки на CPU

5.1 Распределение нагрузки по сервисам

Сервис	Язык/Технология	CPU (мин)	CPU (пик)	Память
Data Ingestion	Python asyncio	2 cores	4 cores	2-4GB
Blockchain Node	Rust + Tokio	2 cores	8 cores	4-8GB
Stream Processing	Python	2 cores	4 cores	2-4GB
Analytics API	Python FastAPI	1 core	2 cores	1-2GB
AI/ML Service	Python + NumPy	4 cores	8 cores	8-16GB
ClickHouse	C++	4 cores	16 cores	8-32GB
PostgreSQL	C	2 cores	4 cores	4-8GB
Redis	C	1 core	2 cores	2-4GB
Kafka	Java/Scala	2 cores	6 cores	4-8GB
Итого		20 cores	54 cores	35-90GB

Table 3: Требования к вычислительным ресурсам

6 Технологии оптимизации производительности

6.1 Rust и Tokio: Преимущества для высоконагруженных систем

6.1.1 Архитектурные преимущества Rust

- Zero-cost abstractions - высокоуровневые конструкции без накладных расходов
- Memory safety без GC - отсутствие пауз сборщика мусора
- Fearless concurrency - безопасная многопоточность на уровне компилятора
- Системное программирование - прямой доступ к аппаратным ресурсам

6.1.2 Tokio: Асинхронная среда выполнения

Listing 1: Пример высокопроизводительного кода с Tokio

```rust
use tokio::time::{interval, Duration};
use tokio::sync::mpsc;
use futures::future::join_all;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    //

    let (tx, mut rx) = mpsc::channel(1000);

    //
    let tasks = (0..8).map(|i| {
        let tx = tx.clone();
        tokio::spawn(async move {
            let mut interval = interval(Duration::from_millis(100));
```

```
            loop {
                interval.tick().await;
                let data = fetch_blockchain_data(i).await?;
                tx.send(data).await.unwrap();
            }
        })
    }).collect::<Vec<_>>();

    //
    while let Some(data) = rx.recv().await {
        process_data(data).await?;
    }

    join_all(tasks).await;
    Ok(())
}
```

6.2 Сравнение производительности: Python vs Rust

Метрика	Python (asyncio)	Rust (Tokio)	Улучшение
Throughput (req/sec)	5,000	25,000	5x
Latency (p99)	200ms	40ms	5x
Memory usage	500MB	100MB	5x
CPU efficiency	70%	95%	1.36x
Concurrent connections	1,000	10,000	10x
Binary size	N/A	15MB	Статическая сборка

Table 4: Сравнение производительности Python и Rust

6.3 Дополнительные технологии оптимизации

6.3.1 1. Connection Pooling и Keep-Alive

- HTTP/2 multiplexing для внешних API
- Database connection pooling (SQLx для Rust, asyncpg для Python)
- Redis connection pooling с persistent connections

6.3.2 2. Кэширование на разных уровнях

- Application-level cache - в памяти процесса
- Redis cache - распределенное кэширование
- CDN caching - для статических данных API
- Database query cache - на уровне ClickHouse и PostgreSQL

6.3.3 3. Batch Processing и Bulk Operations

- Batch inserts в ClickHouse (до 10,000 записей за раз)
- Bulk API calls для внешних источников
- Message batching в Kafka

- Vectorized operations в обработке данных

7 Конкретная реализация в инфраструктуре

7.1 Blockchain Node Service (Rust)

Основной компонент для высокопроизводительной обработки блокчейн данных:

Listing 2: Архитектура Rust сервиса

```rust
//
pub struct BlockchainNodeService {
    ethereum_client: Arc<EthereumClient>,
    l2_clients: HashMap<String, Arc<dyn L2Client>>,
    cosmos_clients: HashMap<String, Arc<CosmosClient>>,
    kafka_producer: Arc<KafkaProducer>,
    db_pool: Arc<PgPool>,
    metrics: Arc<MetricsCollector>,
}

impl BlockchainNodeService {
    pub async fn start_sync_tasks(&self) -> Result<()> {
        let mut tasks = Vec::new();

        // Ethereum sync task
        let eth_task = self.spawn_ethereum_sync();
        tasks.push(eth_task);

        // L2 sync tasks (                                                    )
        for (network, client) in &self.l2_clients {
            let task = self.spawn_l2_sync(network.clone(), client.clone());
            tasks.push(task);
        }

        // Cosmos sync tasks
        for (network, client) in &self.cosmos_clients {
            let task = self.spawn_cosmos_sync(network.clone(), client.clone());
            tasks.push(task);
        }

        //
        try_join_all(tasks).await?;
        Ok(())
    }
}
```

7.2 Конфигурация производительности

Параметр	Значение	Обоснование
L2_BATCH_SIZE	100	Оптимальный баланс memory/throughput
L2_MAX_CONCURRENT_REQUESTS	10	Лимит rate limit внешних API
COSMOS_BATCH_SIZE	50	Меньший размер блоков в Cosmos
COSMOS_MAX_CONCURRENT_REQUESTS	8	Консервативный подход для стабильности
POLKADOT_BATCH_SIZE	20	Substrate специфика
RUST_LOG	info	Баланс между производительностью и отладкой
TOKIO_WORKER_THREADS	8	2x количество CPU cores
DATABASE_MAX_CONNECTIONS	20	Пул соединений с PostgreSQL
KAFKA_BATCH_SIZE	1000	Максимальная эффективность Kafka

Table 5: Оптимизированные параметры конфигурации

8 Мониторинг производительности

8.1 Ключевые метрики

- Data ingestion rate: messages/second от каждого источника
- API response times: p50, p95, p99 percentiles
- Error rates: по каждому источнику данных
- CPU utilization: по каждому сервису
- Memory usage: heap size, RSS memory
- Network I/O: bytes in/out, connections
- Database performance: query time, connection pool usage
- Kafka lag: consumer lag по топикам

8.2 Alerting и SLA

Метрика	Порог Warning	Порог Critical
API Response Time	>500ms	>1000ms
Error Rate	>1%	>5%
CPU Usage	>70%	>90%
Memory Usage	>80%	>95%
Disk Usage	>80%	>90%
Kafka Consumer Lag	>1000 messages	>10000 messages
Database Connections	>80% pool	>95% pool

Table 6: Пороговые значения для мониторинга

9 Рекомендации по масштабированию

9.1 Горизонтальное масштабирование

- Blockchain Node: Увеличение replicas до 3-5 для обработки большего количества сетей

- Data Ingestion: Auto-scaling на основе Kafka consumer lag
- Stream Processing: Partitioning по типу данных
- ClickHouse: Sharding по времени (monthly partitions)

9.2 Вертикальное масштабирование

- CPU: Приоритет для Rust сервисов и ClickHouse
- Memory: Критично для AI/ML Service и кэширования
- Storage: NVMe SSD для ClickHouse, быстрые диски для PostgreSQL
- Network: 10Gbps для high-throughput окружений

10 Заключение

Использование Rust и Tokio в критически важных компонентах системы DEFIMON обеспечивает:

- 5x улучшение производительности по сравнению с Python
- Снижение потребления ресурсов на 60-80%
- Повышение надежности благодаря строгой типизации
- Лучшую масштабируемость для обработки множественных блокчейнов
- Снижение операционных расходов на инфраструктуру

Рекомендуемая инфраструктура способна обрабатывать более 2GB данных в час с латентностью менее 100ms и обеспечивать высокую доступность системы.