سیستم تحلیل لاگ

آیدا جلالی مهدی اکبری فخرالدین عبدی

تکنولوژی کامپیوتر

هدف اصلی، ساخت سیستمی است که بتواند حجم بسیار بالایی از لاگ‌ها را دریافت، ذخیره و امکان جستجو و تحلیل آن‌ها را برای کاربران فراهم کند. طراحی این سیستم حاصل بحث و تبادل نظر تیمی و بررسی راهکارهای مختلف است.
معماری‌ای که ما به آن رسیدیم، یک Pipeline Data چندمرحله‌ای است که مسیر ورود داده را از مسیر خواندن و تحلیل داده جدا می‌کند. این جداسازی به ما اجازه می‌دهد تا هر بخش را متناسب با نیازش بهینه‌سازی کنیم و از ایجاد bottleneck در سیستم جلوگیری کنیم.
جریان کلی داده:

```
graph TD
    subgraph "Write Path (High Volume)"
        A[Log Source] -->|HTTP POST| B(Go Ingestion API)
        B -->|project_id, api_key| C(CockroachDB)
        C -->|Validate Key| B
        B -->|Push Log| D(Kafka Topic: 'logs')
        E(Go Consumer) -->|Polls Logs| D
        E -->|Write Full Log| F(Cassandra)
        E -->|Write Indexed Log| G(ClickHouse)
    end

    subgraph "Read Path (User Queries)"
        H[User via Web UI] -->|HTTP GET| I(Go Query API)
        I -->|Login/Project Info| C
        I -->|Search/Aggregate| G
        I -->|Fetch Log Details| F
        C -->|Metadata| I
        G -->|Aggregated Data| I
        F -->|Full Log Payload| I
        I -->|JSON Response| H
    end
```

ما از CockroachDB به عنوان یک پایگاه داده رابطه‌ای استفاده می‌کنیم که تراکنش‌های ACID را برای مدیریت کاربران، پروژه‌ها و کلیدهای API فراهم می‌کند. همچنین از قابلیت کلاسترینگ و Replication داخلی آن برای تحمل خطا استفاده میکنیم. استفاده از تراکنش‌های ACID تضمین می‌کند که عملیات چندمرحله‌ای (مانند ایجاد کاربر و پروژه به صورت همزمان) یا به طور کامل انجام شده یا اصلاً انجام نشود. این ماژول، جلوی ناقص ماندن داده‌ها را می‌گیرد که برای اطلاعات حیاتی ما ضروری است. همچنین، استفاده از ماژول Replication داخلی آن به این معناست که ما نیازی به پیاده‌سازی دستی مکانیزم‌های پیچیده برای تحمل خطا نداریم. استفاده از تراکنش‌ها می‌تواند کمی بار کاری را افزایش دهد و مدیریت یک

کلاستر پیچیده‌تر از یک سرور تکی است. ما این عیب را پذیرفتیم زیرا صحت و دسترس‌پذیری اطلاعات کاربران و پروژه‌ها برای ما اولویت مطلق است و این بده‌بستان کاملاً منطقی است. می‌توانستیم از سطح پایین‌تری از ایزولاسیون تراکنش‌ها استفاده کنیم که سریع‌تر است، اما ریسک بروز خطاهای ظریف در داده‌ها را افزایش می‌داد. ما تصمیم گرفتیم امنیت را به سرعت ترجیح دهیم.
انتیتی‌های ساخته شده در این دیتابیس به صورت زیر است:

```
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    username STRING UNIQUE NOT NULL,
    password_hash STRING NOT NULL
);
CREATE TABLE projects (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name STRING NOT NULL,
    api_key STRING UNIQUE NOT NULL,
    log_ttl_seconds INT NOT NULL,
    owner_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE
);
CREATE TABLE user_projects (
    user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    project_id UUID NOT NULL REFERENCES projects(id) ON DELETE CASCADE,
    PRIMARY KEY (user_id, project_id)
);
CREATE TABLE project_searchable_keys (
    project_id UUID NOT NULL REFERENCES projects(id) ON DELETE CASCADE,
    key_name STRING NOT NULL,
    PRIMARY KEY (project_id, key_name)
);
```

از Cassandra به عنوان یک پایگاه داده Store Wide-Column برای بایگانی لاگ‌های خام استفاده می‌کنیم. به طور مشخص، از قابلیت داخلی Time-To-Live یا TTL استفاده میکنیم. مدل داده‌ای که ما بر اساس کلید اصلی یعنی project-id و log-id طراحی کرده‌ایم، برای نوشتن سریع و خواندن بر اساس کلید کاملاً بهینه است. اما ماژول کلیدی که ما از آن استفاده می‌کنیم، TTL است. با تنظیم TTL برای هر ردیف در زمان درج، وظیفه زمان‌بر و پرهزینه حذف لاگ‌های قدیمی را به خود Cassandra واگذار می‌کنیم. این در حالی است که پرس‌وجوهای پیچیده و تحلیلی را دشوار می‌کند.البته چون از ابتدا وظیفه تحلیل داده‌ها را به ابزار دیگری یعنی ClickHouse سپرده‌ایم و از Cassandra فقط انتظار ذخیره و بازیابی سریع را داریم. می‌توانستیم TTL را اعمال نکنیم و خودمان یک سرویس برای پاک کردن دوره‌ای داده‌ها بنویسیم که البته این کار باعث بار اضافی روی پایگاه داده می‌شد و با اصل سادگی در مهندسی در تضاد بود.
جدول لاگ‌ها به صورت زیر است:

```
CREATE KEYSPACE log_system WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 3};

USE log_system;

CREATE TABLE logs (
    project_id uuid,
    log_id uuid,
    timestamp timestamp,
    event_name text,
    payload map<text, text>,
    PRIMARY KEY (project_id, log_id)
);
```

ما از موتور پایگاه داده MergeTree در ClickHouse برای تحلیل سریع داده‌ها استفاده می‌کنیم.ما از قابلیت Parti-tioning آن بر اساس زمان استفاده میکنیم. مزیت اصلی موتور MergeTree استفاده ذاتی آن از ذخیره‌سازی ستونی است.

این ماژول باعث می‌شود کوئری‌های GROUPBY و COUNT که تنها به چند ستون نیاز دارند، فوق‌العاده سریع باشند. ما از ماژول Partitioning برای تقسیم داده‌ها بر اساس ماه یعنی toYYYYMM(timestamp) BY PARTITION استفاده می‌کنیم که سرعت را به شدت افزایش می‌دهد. عیب این رویکرد این است که به‌روزرسانی یا حذف تکی رکوردها در موتور MergeTree عملیاتی پرهزینه و کند است ولی اهمیتی ندارد چون ماهیت داده‌های لاگ، Append-Only است و ما هیچ‌گاه یک لاگ ثبت‌شده را به‌روزرسانی نمی‌کنیم. می‌توانستیم داده‌ها را پارتیشن‌بندی نکنیم. این کار طراحی را ساده‌تر می‌کرد اما به قیمت کند شدن شدید تمام کوئری‌هایی که فیلتر زمانی داشتند تمام می‌شد که چیز خوبی نبود.

جدول مربوط به ایندکس‌های لاگ به صورت زیر است:

```
CREATE TABLE logs_index (
    project_id UUID,
    log_id UUID,
    event_name String,
    timestamp DateTime,
    searchable_key_1 String,
    searchable_key_2 String
) ENGINE = MergeTree()
PARTITION BY toYYYYMM(timestamp)
ORDER BY (project_id, event_name, timestamp);
```