# Semi-structured Data Study: Audit Log Generation, Storage, Management, and Retrieval

Fei Gu

May 28, 2025

## 1 Introduction

In our current project development, my company is designing a middle-tier management system. One of its core requirements is to build a comprehensive **audit log module** to record user operational behavior and device status changes. Given the existence of multiple user roles (e.g., administrators, maintenance personnel, general users) and various types of devices within the system, the fields required for each audit log entry significantly vary in structure.

Under traditional relational database modeling, adapting to different user behaviors and device interaction patterns often necessitates designing multiple data tables and complex entity relationship structures. This approach not only leads to a bloated data model and increased redundant fields but also introduces significant performance overhead during both write and query phases. Especially in an audit log system, **high-frequency read-write operations** are standard: users need to retrieve logs instantly after logging in, and every device operation must be recorded in real-time. This subjects the database to high concurrent read-write pressure, leading to high I/O bandwidth usage and slower system response times.

Furthermore, log data inherently possesses structural uncertainty and dynamic extensibility. For instance, some device events record GPS coordinates, while certain user behaviors include browser information. These fields are difficult to flexibly adapt within traditional relational tables. Therefore, to enhance system scalability, data processing efficiency, and development flexibility, this study attempts to adopt a **semi-structured data model**, combining MongoDB and Redis to build a log management system. This aims to explore the capabilities of semi-structured data in database modeling and query optimization.

This project will serve as a practical case study for semi-structured data research. By building a data system with realistic business scenarios, we will evaluate its advantages and disadvantages compared to traditional solutions in terms of write efficiency, query performance, and system scalability.

## 2 Problem Statement

In traditional database design, structured data models rely on fixed table schemas and field constraints, which prove to be inadequate when handling log-like data. As business complexity increases, the variety of user roles and device types involved in the system grows daily, leading to inconsistent log data structures. Such data often includes nested structures, optional fields, and dynamic key-value pairs, making it difficult to express flexibly using traditional relational models. This, in turn, results in redundant table designs, decreased query efficiency, and increased maintenance costs.

Especially in high-frequency read-write audit log scenarios, relational databases struggle to simultaneously meet the dual demands of data write throughput and complex structured queries, and they face significant bottlenecks during system expansion.

Therefore, the problem addressed by this study is: **How can a semi-structured data model be used to optimize the modeling, storage, and query efficiency of log-like data, while maintaining high system performance without sacrificing scalability?**

# 3 Methodology

## 3.1 Semi-structured data

**Semi-structured data**[1] is a form of structured data that does not obey the tabular structure of data models associated with relational databases or other forms of data tables, but nonetheless contains tags or other markers to separate semantic elements and enforce hierarchies of records and fields within the data. Therefore, it is also known as self-describing structure. In semi-structured data, the entities belonging to the same class may have different attributes even though they are grouped together, and the attributes' order is not important.

Semi-structured data lies between structured and unstructured data. Unlike the structured data used by relational databases, it does not have strict data structure requirements where every piece of data must contain an entity name (table), attributes (columns), and tuples (rows), and where the data structure cannot be arbitrarily modified once defined. Nor is it like unstructured data such as this document or audio/video files, which record and store data randomly and disparately.

Semi-structured data exhibits a concrete structure that can be predefined. However, in practical applications, the data structure can be modified according to specific needs. It can also reference or embed other data structures, allowing for flexible data storage. Simultaneously, semi-structured data also has certain organizational rules; it is not like unstructured data, which is difficult to index, filter, search, and sort.

### 3.1.1 Types of semi-structured data

**JSON/BSON/YAML** JSON-formatted semi-structured storage is currently the most popular semi-structured storage model. It describes data using key-value pairs. It allows for flexible definition and modification of data structures without prior definition. Although its data format storage is not as diverse as relational databases, all data is saved as strings, which facilitates data storage space allocation and data storage optimization. Conversely, JSON file data types are expressed through data type conversion and inference during parsing (or by relying on JSON schema).

Common databases that use JSON format for data storage include MongoDB, ElasticSearch, etc. The relational database PostgreSQL also natively supports the JSON data type.

**XML/HTML** XML-type unstructured data is typically used for system property settings in software development projects and for saving web page structures. It is characterized by using tags to mark metadata, adding different tag attributes to annotate and reference metadata, and using tag bodies to store data values. Data embedding is achieved by embedding other tags within the tag body.

Cases using XML format for data storage partly represent historical data storage forms, for example, application cases that required unstructured storage before the popularity of the JSON format. Therefore, currently common relational databases with some history generally support XML formatted data storage.

Another part of the use cases is when there is a higher requirement for expressing complex hierarchical data structures, and when each piece of data needs to contain more complex information, including adding attributes to a data item, or defining namespaces. Examples include electronic medical records, regulations/standards, and credential information. Implementing these in JSON-formatted data structures would be more complex.

Currently popular relational databases such as SQL Server, MySQL, and PostgreSQL all support it, and there are also some natively supporting XML format databases, such as BaseX and MarkLogic.

HTML format is generally only used for web page structure storage, and its nature is basically similar to XML format. This project will temporarily omit it.

**Logfile** Log files are file formats used for storing system logs or event logs, and generally, there is no unified standard. More common are plain text file data records separated by spaces, tabs, or certain specific symbols (colons, quotes) on a line-by-line basis. Or data is recorded using key-value pairs (similar to '.sh' shell scripts), JSON format, or CSV (columns separated by semicolons, rows by newlines) format.

[1]WikiPedia: `https://en.wikipedia.org/wiki/Semi-structured_data`

**Email/HTTP message/ MQTT message** Email storage is a special form of semi-structured storage. The structured part is represented by the header of each email declaring the email's metadata attributes and their values. The body contains the plain text content of the email. The unstructured part is that the header attributes are not fixed and follow different standards for specific attribute names and values. HTTP and MQTT are similar.

### 3.1.2 Advantages and disadvantages of semi-structured data

**Flexible data model**

- No need to pre-define table schemas.
- Suitable for storing dynamic fields or diverse objects.
- Examples: user configurations, product attributes, log data, etc.

```
{
   "feix0033": {
      "name": "Fei",
      "creation-time" : "2025-05-25T16:26:00"
      "semanster": 1
   },
   "feix0033@easv365.dk": {
      "name": "Gu",
      "language": "chinese"
   }
}
```

**Natural nested hierarchy** Compared to the entity-relationship definitions of relational databases and the connections between tables that cannot be intuitively observed (only relying on abstract diagrams like ERDs), semi-structured database data itself can directly show relationships through nesting.

```
      <User>
            <Name lang="en">Fei</Name>
            <Name lang="cn">Gu</Name>
            <Create-time lo="dk">2025-05-25T16:26:00</Create-time>
      </User>
```

**Suitable for agile development or rapid iteration** As business develops and changes, unstructured databases can flexibly extend data structures. For example, when new device features are developed and applied, the audit log event records need to add new feature records. At this point, if traditional structured data were used, new data structures would need to be created and existing data migrated. Unstructured data, however, can directly add corresponding fields to new data.

```
2023-10-27 10:00:00 INFO User logged in: user123
2023-10-27 10:00:05 DEBUG Processing request for /api/data
2023-10-27 10:00:10 ERROR Database connection failed: 192.168.1.100
2023-10-27 10:00:15 WARNING Disk space low on /var/log

# Add new field: *status*
2023-10-28 10:00:00 INFO SUCCESS User logged in: user123
2023-10-28 10:00:05 DEBUG START Processing request for /api/data
2023-10-28 10:00:10 ERROR STOP Database connection failed: 192.168.1.100
2023-10-28 10:00:15 WARNING STOP Disk space low on /var/log
```

**Stronger data expressiveness** Compared to a single table with only row and column representation, unstructured data can include metadata, attributes, hierarchical structures, and other diverse characteristics.

## 3.2 System Design and Implementation Plan

### 3.2.1 System Design

This project adopts a modular service architecture based on NestJS, combined with MongoDB for storing semi-structured data (JSON) and Redis for implementing the caching layer. The system is divided into the following main modules:

**Audit Log Module** The audit log module is responsible for receiving log inputs such as user behavior and device status changes. It handles the service part of the audit logs, including:

- `controller`: Receives HTTP requests and calls services.
- `service`: Processes obtained data and writes it to the database.
- `interface`: Database implementation interface for connecting to different databases.
- `dto`: Data structure conversion corresponding to different databases.
- `entities/schemas`: Different database structures.

**Cache Module** Uses Redis to cache log query results, improving query performance.

**Database Module (MongoDB / PostgreSQL)** Implements CRUD (Create, Read, Update, Delete) functionality for the database.

- `database.module.ts` Database module
- `mongodb/`
- `postgresql/`

### 3.2.2 Technology Stack Explanation

- `Nestjs`: A Typescript-based backend modular framework that provides support for operations on unstructured data.
- `MongoDB`: A document-oriented database that natively supports JSON format for data storage.
- `Redis`: An in-memory database used to implement data cache queries.
- `PostgreSQL`: A powerful relational database that, in addition to supporting structured data, also natively supports some semi-structured data.
- `Postman`: Used to test the storage of different semi-structured data structures and varied metadata structures.
- `k6`: An API and high-concurrency simulator, used to test multi-user simulated concurrent writes and cache hits.

### 3.2.3 Implementation Plan

1. Set up the basic structure of the audit log service using Nest.js, introduce and implement multiple databases, and realize CRUD services for different databases. Use Swagger to simulate frontend requests.
2. Introduce k6 and implement basic functions for API testing and high-concurrency testing.
3. Build test data structures. Design different data structures and metadata structures based on actual requirements.
4. Use the test data structures to send identical requests to the read/write interfaces of different databases via Postman, comparing read/write operations for various semi-structured and structured data.

5. Use the test data structures to send data write requests with different metadata structures to the same database's read/write interfaces via Postman, testing and comparing changes in metadata structures. Since structured data requires defining many auxiliary tables, this test is not practically meaningful and does not need to be performed.
6. Use k6 to simulate concurrent read/write operations on different databases, comparing the read-/write efficiency of different semi-structured data under high concurrency.
7. Introduce a caching service and again use k6 to simulate high concurrency, observing the new response times.

## 3.3 Evaluation Method and Metrics

### 3.3.1 Measurement Goals

1. **Comparison of Storage and Read Performance for Different Data Types**
   - Compare the performance differences of JSON, XML, and traditional structured data in log writing and querying.
2. **Support for Metadata Structure Diversity**
   - Verify whether metadata differences generated by different users/devices can be correctly stored and read by the system.
3. **Write Performance Under High Concurrency**
   - Simulate multiple users concurrently writing a large volume of audit logs.
4. **Effect of Cache Hits on Query Efficiency Improvement**

### 3.3.2 Measurement Content

| Measurement Metric | Description |
|---|---|
| Response Time | Average time required for writes and queries (milliseconds) |
| System Throughput | Number of requests processed per second (req/s) |
| CPU / Memory Usage | Resource consumption under high concurrency |
| Redis Hit Rate | Whether the cache is correctly hit (hit or miss, number of hits) |
| Log Accuracy | Whether diverse metadata structures are accurately stored in the database |

### 3.3.3 Evaluation Objects

| Object/Dimension | Description |
|---|---|
| Data Type Support | JSON / XML / Structured Data |
| Write Performance | Performance under different data types and concurrency |
| Query Performance | Impact of cache hits/misses on query time |
| Data Consistency | Accuracy of log data under concurrent writes |
| Cache Mechanism Effectiveness | Whether Redis correctly caches data, and if it's hit |

### 3.3.4 Evaluation Methods

| Method | Purpose |
|---|---|
| **Benchmark** | Compare write/read performance of different data structures |
| **Simulated Metadata Variant Writing** | Simulate writing and reading of various metadata structures for verificati |
| **Stress Test** | Simulate concurrent user writes using k6 tool |
| **Cache Hit Observation** | Observe Redis logs or use Vitest unit tests to monitor get/set calls |
| **Response Time Comparison** | Use Postman or k6 to compare latency changes with and without cache h |

#### 3.3.5 Evaluation Tools

| Tool | Purpose |
|---|---|
| **Postman** | Manually or automatically run API tests, record response times |
| **Vitest** | Unit and integration tests, verify data structure accuracy and cache logic |
| **k6** | High-concurrency write simulation tool, generate pressure test reports |
| **RedisInsight / CLI** | View cache status and key hits |
| **MongoDB Compass / Robo 3T** | View the actual storage structure of semi-structured data |
| **Docker Compose** | Quickly rebuild test environment, simulate production deployment |

#### 3.3.6 Evaluation Criteria

| Hypothesis | Verification Me |
|---|---|
| Semi-structured data is more suitable for storing audit logs with inconsistent metadata structures | Diverse structured |
| Caching can effectively reduce database read pressure | Redis cache hit te |
| Both JSON and XML can be used as semi-structured data formats | Store using Mongo |
| The system can maintain high write performance and data consistency under high concurrency | Use k6 for concurr |

—

## 4 Analysis & Results

### 4.1 Description of Achievements

This is the meat of your synopsis. You have to outline what you have done in this section. Document the effort you've put into analyzing the problem and also the work you've done building and implementing a solution.

- First, describe what was done.
- How well was it done, and how good is it?
- How much was implemented, and what were the results?

### 4.2 Analysis of Achievements

This isn't your GitHub repository. You should not include all the code you've written. Instead, you should include the most important parts of your work. What is important? Parts of the code that are a result of your analysis and the choices you've made. Connect theory and analysis to how the code is written.

Demonstrate that you understand the theory and that you can apply it to solve a problem. Don't expect and plan for creating a "product". Write only enough code, outside your main objective and focus on your problem.

- Highlight significant aspects.
- For example, where the code implemented a certain requirement.
- What evaluation was conducted, and what results were obtained?
- What is its value?
- (Include some pseudocode to illustrate its utility.)

—

## 5 Conclusion

Summarize.

This is your chance to briefly summarize what you've done throughout the project. Focus on the most important parts and takeaways from your work. And finally, what did you find out? Did you confirm or reject your hypothesis?