A Lightweight and Scalable NoSQL Database using BSON

Nikhil Parbat[1], Rishi Mishra[2], Sarvesh Shinde[3]

[1]Student, School of Computing, MIT ADT University, Pune
nikhilparbat908@gmail.com,
[2]Student, Yashwantrao Chavan Maharashtra Open University, Nashik
[3]Student,M.C.T. Rajiv Gandhi Institute of Technology, Mumbai
Domain: Databases and Backend Development

Abstract **- In modern application development, Lightweight and self-contained databases are crucial for offline-first, embedded, and edge computing settings in contemporary application development. In contrast to SQLite-based solutions like PocketBase, this article presents a file-based NoSQL database that stores documents efficiently using a single BSON. It provides a high-performance, schema-less storage engine that includes integrated file management, authentication, and real-time synchronization. It can be developed in Golang as it promises portability, low resource use, and seamless integration with modern applications. This article examines its design, indexing methods, and potential advantages over traditional relational and NoSQL databases.**

Keywords **– Database, Backend Systems, NoSQL, Golang, BSON, JSON**

I.INTRODUCTION

As applications evolve, so does the need for lightweight, effective, and adaptive databases. Traditional relational databases are reliable, but because of their performance overhead and potential for complex schemas, they are less suitable for modern development needs. SQLite[1] is the foundation of Backend-as-a-Service (BaaS) systems like PocketBase, which inherit its transactional design and schema requirements.

Unfortunately, many of the systems in use today are either very resource-intensive or do not include the features that are required for the best read/write speed, such as file management, authentication, real-time synchronisation, and caching. In contrast, NoSQL databases offer schema flexibility.

This article proposes the idea of an open-source database that is packaged in a single file and designed to serve as a primary database for small to medium-sized applications as well as a lightweight solution for embedded and offline-first use cases. In contrast to relational databases, it will store documents in a single BSON, enabling high-performance, schema-less data management.

Developed in Go, it will prioritize portability, minimal resource consumption, and seamless integration into modern applications. This paper explores its architecture, indexing strategies, caching mechanisms, and advantages over traditional relational and NoSQL databases.

In the next section we are going to talk about the Background and the deeper insights of Golang and BSON[3].

II.BACKGROUND

1. Golang:

The statically typed, compiled language Go (Golang)[2] is made with scalability, simplicity, and efficiency in mind. It is a great option for backend development because of its lightweight design and effective implementation, particularly for applications like this database that demand high speed, little resource usage, and smooth deployment.

Go's ability to generate self-contained binaries without requiring external dependencies or runtimes is one of its main features. In contrast to Python, JavaScript, or Java, which need interpreters or virtual machines, this database can be a single-file backend. Furthermore, Go uses less memory than Java and C++ and runs much quicker than Python and JavaScript because of its built-in nature.

Concurrency is another major factor in Go's selection. Unlike Python and JavaScript, which rely on threads and event loops, Go's goroutines provide a highly efficient concurrency model. This enables real-time synchronization and caching without excessive resource consumption. The language's garbage collection is also more optimized compared to Java, reducing memory overhead while avoiding the complexities of manual memory management seen in C++.

Simplicity and maintainability are additional reasons why Go is well-suited for this database. Unlike TypeScript and Java, which introduce layers of abstraction, Go follows a minimalist syntax that keeps the codebase clean and easy to manage. This ensures long-term maintainability without sacrificing readability or performance.

Lastly, Go's cross-platform compatibility allows it to be deployed effortlessly across different environments. Unlike Python, which requires an interpreter, or Java, which depends on the JVM, Go compiles directly into native executables. This ensures that it can be run with minimal setup, making it a truly portable and efficient database solution.

BSON

BSON (Binary JSON)[3] is a binary-encoded serialization format designed to efficiently store and traverse JSON-like documents. Originally developed for MongoDB, BSON provides several optimizations over JSON[4], including faster parsing and traversal due to its structured binary format. Unlike JSON, which represents data as text, BSON stores data in a compact binary form, reducing the overhead associated with parsing textual representations. BSON also supports additional data types, such as 32-bit and 64-bit integers, floating-point numbers, and binary data, which are not inherently present in standard JSON. Additionally, BSON includes length-prefixed string encoding and embedded document structures, allowing for more efficient querying and retrieval of deeply nested fields without the need for full document scanning. While BSON may not always be more space-efficient than JSON due to metadata storage, it significantly improves performance in NoSQL databases where high-speed read and write operations are crucial.

SQLite

In the next section we are going to talk about the methodologies we use in this system

III.METHODOLOGIES

1. System Architecture

Our Database is a lightweight, embedded NoSQL alternative to SQLite3 also known as SQLite[1], designed for developers who need a self-contained, high-performance solution without a dedicated server. It stores data in a BSON based Database instead of relational tables. Unlike most NoSQL databases that need a dedicated server, this one works as a single-file, self-contained database, making it perfect for offline applications, edge computing, and lightweight deployments. Built in Go(Golang), it provides high performance and cross-platform compatibility.

At its core, the system consists of a storage engine, an indexing system, a query processor, concurrency management and language bindings for integration with different programming environments. Instead of storing data in multiple files or requiring a background process, everything is packed into a single {appname.bsondb} file. This keeps things simple and efficient while allowing direct file access without the requirement for a dedicated database server. To keep data safe and durable, the system uses an append-only structure or Write-Ahead Logging (WAL) to prevent data loss in extreme situations like crashing. Moreover , since the entire database is in a single file, scenarios like server stopping or crashing all together won't stop us from using the database

Indexing and Querying

Implementation of B-Trees, Hash Indexing or Log-Structured Merge Trees (LSM-Trees) can be implemented according to workloads which will enable fast lookups. Storing the indexes within the database file allows it to be self contained and boost query speeds. Query execution is document based and inspired by MongoDB[5] which can be optimized for embedded databases.

Concurrency & Transactions

Unlike SQLite's strict ACID model, this system offers optional Multi-Version Concurrency Control (MVCC) or lightweight locking for managing multiple reads and writes. For those who need transaction safety, there's a simple logging mechanism that ensures durability with minimal performance overhead typically associated with strict transactional models. Referenced from [9]

Security and Integration

Using AES and ChaCha20 encryption which will ensure privacy so that even if the database files get accessed they won't be readable. For development API support in languages such as C, Python, Rust, etc will enable easy integration into various applications. Moreover, inclusion of a CLI for managing the database, running queries and optimising storage can also be used.

Basic Syntax:

The proposed query syntax is designed to be simple, readable, and intuitive, making it accessible to developers with varying levels of experience. It follows a declarative style similar to SQL but is optimized for a NoSQL document-based database. This set of syntax provides certain advantages like ease of understanding as it uses clear, self-explanatory keywords. Minimal complexity as it reduces unnecessary symbols and verbose expressions. As the inspiration comes from both SQL and NoSQL syntax, it makes it easier to learn for almost all levels of seniority. CRUD operations follow a predictable and uniform pattern.

```
CREATE DB mydb;
USE mydb;
CREATE COLLECTION users
 { name: string,
email: string,
age: int | float
 };
ADD users [ { id: "user123",
 name: "John Doe",
email: "john@example.com",
age: 30 } ];
GET users["user123"];
UPDATE users["user123"]
SET age = 31;
DELETE users["user123"];
```

Figure 1.0
Basic representation of the query syntax in Database

From figure 1.0 , we can see the basic representation of the query syntax which can be used for this database.

CREATE DB mydb; – Creates a new database named mydb. USE mydb; – Selects the database mydb for further operations. CREATE COLLECTION users {...}; – Defines a collection (users) with structured fields (name, email, age). ADD users [{...}]; – Inserts documents into the users collection. GET users["user123"]; – Retrieves a document by its unique identifier. UPDATE users["user123"] SET age = 31; – Modifies specific fields within a document. DELETE users["user123"]; – Removes a document from the collection.

This approach ensures a structured yet flexible way to manage NoSQL data while maintaining clarity and efficiency.

IV.PROPOSED BLOCK DIAGRAM

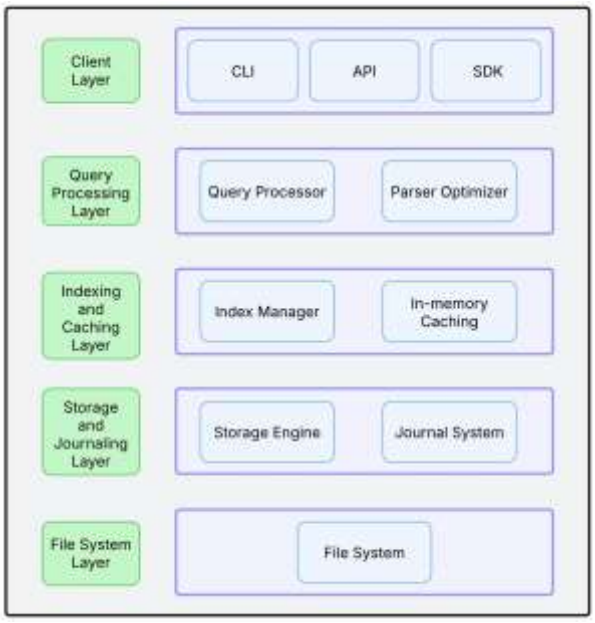


Figure 2.0
Different Layers of Database

In This Proposed Block Diagram, it showcases the different layers involved and a basic process flow of the database. Users can interact with the database using commands, API calls or using the SDK. Next, the queries are parsed and optimized for the execution Engine which then sends them to the relevant storage components. The indexing and caching layer is responsible for managing the indexes which can be in the form of B-Tree or Hash Maps.It makes use of frequent caching to reduce disk reads. Moving on to the next later of Storage and journaling which handles the input and output with relation to the BSON files and logs changes before committing to prevent data corruption. This layer also handles transactions and recovery mechanisms. The final layer is the File System Layer which handles raw BSON data in the form of {appname}.bson.

In the next section we will be discussing about the results and basic insights of the system

RESULT

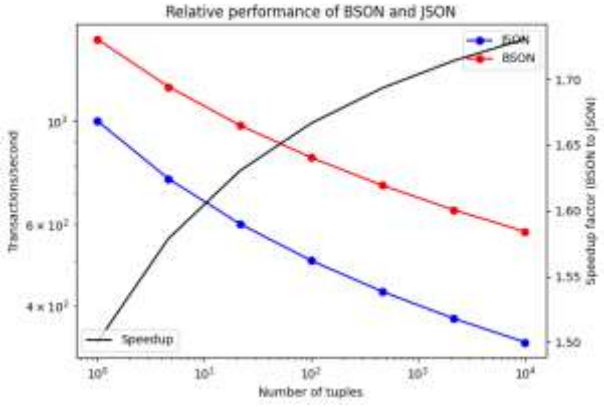


Figure 3.0

The graph represented BSON vs JSON performance based on number of tuples adapted from [8]

The Figure illustrates the comparative performance of BSON & JSON in terms of transactions per second across varying numbers of tuples that is plotted on a logarithmic scale. The x-axis represents the number of tuples ranging from 10^0 to 10^4, while the y-axis on the left denotes transactions per second and the right y-axis indicates the speedup factor of BSON relative to JSON.

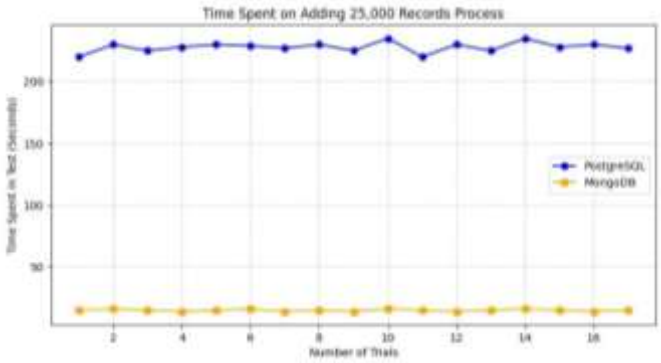
The blue curve represents JSON performance and the red curve represents BSON performance. It can be seen that BSON consistently scores higher transactional throughput than json. The black curve line shows the speedup factor (BSON to JSON), which increases with the number of tuples showcasing that BSON exhibits better efficiency than JSON when it comes to larger dataset sizes. Initially for smaller datasets both JSON & BSON have close performance but as the size increases, BSON's advantage becomes more visible.

This analysis makes BSON's efficiency clearer when handling large-scale data transactions making it a preferable choice for high-performance applications requiring structured data storage and retrieval.

Factor	**BSON**	**JSON**
Serialization Speed	Fast (Binary format)	Slower (String parsing)
Deserialization Speed	Fast (Pre-typed data)	Slower (Type inference)
Storage Size	Larger (Metadata overhead)	Smaller (Compact format)
Indexing Speed	Efficient (Pre-typed fields)	Slower (Parsing required)
Query Execution	Faster (Direct access)	Slower (Needs parsing)
Scalability with Tuples	Better	Degrades over time

Table 1.0

BSON (Binary JSON) is designed for speed and efficiency in databases such as MongoDB, while JSON is lighter and more human-readable but more processing-intensive.BSON becomes better with tuples while performance of JSON deteriorates with time in such a scenario. As a binary format, BSON is much faster in serialization as it converts data directly into a structured binary format without additional parsing overhead, whereas JSON employs string-based encoding that needs additional processing to convert structured data into text format, making serialization slower than BSON.Stores data in pre-typed format, i.e., values already have a specified type (e.g., integers, strings, dates), which results in quicker deserialization as no additional type inference is required whereas JSON Stores data in plain text form, so the system has to parse and identify data types during deserialization, thus taking more time.BSON occupies more space because of additional metadata, e.g., field lengths and data types, which aid in quicker query execution but contribute to more storage usage, whereas JSON is More lightweight as it does not include additional metadata, thus occupying less space and being efficient for storage but taking more processing time during queries.BSON Indexing is quicker because data is already structured with pre-specified types, so databases such as MongoDB can easily find indexed fields, whereas Slower in indexing as parsing each record to identify field values is required before creating an index, which takes computational overhead.BSON Offers quicker query execution as the database engine can access and process fields directly without additional parsing, whereas JSON Queries are slower as JSON data has to be read and parsed before execution, thus making retrieval operations less efficient. BSON It handles big datasets and tuples well and keeps performance with increasing database size, hence being more scalable in big applications while JSON Performance decreases over time as parsing big JSON documents is computationally costly, thus less appropriate for huge datasets



25,000 records were added to both databases 16 times in a row,the duration and average duration of these trials in seconds are given

Figure 4.0

Another major advantage of using such a NoSQL database is the speed and efficiency of it in comparison to SQL databases. Figure 4.0 shows the test results done by [13] which highlights the speed of a NoSQL database (MongoDB) in comparison to a SQL database(Postgres). The test involved a series of trials where 25,000 records were added to each database and the time taken to complete the process is recorded. As visible from Figure 4.0 the time taken by MongoDB is 20 seconds whereas the time taken by Postgres exceeds 200 seconds.We can see from this that MongoDB (NoSQL) is faster than Postgres (SQL). For

applications that need to enter data quickly, this notable performance difference can be essential. Furthermore, MongoDB's architecture is significantly favored by the particular workload utilized in the test, which involves bulk inserts.

In the next section finally we will conclude and discuss what we learn from this research.

CONCLUSION

This paper has proposed the implementation of a BSON based NoSQL database similar to SQLite[1] which significantly improves performance in comparison to using JSON based databases. It highlights the difference in read-write speeds, query processing and transaction speed with reference to scale as BSON's advantages outweigh those of JSON.

In the future, we plan to implement this theoretical database as an actual database which can be integrated with different languages. Alongside that, benchmarking the performance against existing NoSQL databases like MongoDB[5] and CouchDB[6] will help validate our findings. Moreover, addition of more optimization techniques will further enhance BSON's efficiency.

REFERENCES

[1] SQLite *sqlite.org*
[2] Golang *go.dev*
[3] BSON *bsonspec.org*
[4] JSON *json.org*
[5] MongoDB *mongodb.com*
[6] CouchDB *couchdb.com*
[7]Avrilia Floratou, Nikhil Teletia, David J. DeWitt, Jignesh M. Patel, Donghui Zhang Can the Elephants Handle the NoSQL Onslaught? In *ARXIV*
[8] Geoffrey Litt, Seth Thompson, John Whittaker Improving performance of schemaless document storage in PostgreSQL using BSON
[9]https://en.wikipedia.org/wiki/Lightning_Memory-Mapped_Database
[10]"Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service" by AWS
[11]"An Effective Scalable SQL Engine for NoSQL Databases" by Damien Collé, Philippe Cudré-Mauroux, and Anastasia Ailamaki
[12]"Evaluating NoSQL Databases for OLAP Workloads: A Benchmarking Study of MongoDB, Redis, Kudu, and ArangoDB" by Rishi Kesav Mohan
[13] "Response Times Comparison of MongoDBand PostgreSQL Databases in Specific Test Scenarios" Emin Güney and Nurdogan Ceylan