Analysis of Aerospike as a competitive NoSQL database solution for the modern age

GUILHERME ALMEIDA, JOSÉ OSÓRIO, and NUNO PEREIRA

In the modern days, there is an increasing number of solutions for data storage, either the classic Relational Databases, dominated by competitors such as **MySQL** and **PostgreSQL** or novel *NoSQL* which aim to drive a paradigm shift into more varied solutions, tailored to a product's needs. One such *NoSQL* database is **Aerospike**[1].

This paper provides an in-depth analysis of **Aerospike**, its use cases and strong features, and how it positions itself against competitors in the industry.

CCS Concepts: • **Information systems** → **Database design and models**; **Storage architectures**; **Information storage technologies**; • **Computer systems organization** → *Peer-to-peer architectures.*

Additional Key Words and Phrases: Aerospike, Relational, NoSQL, Key-Value, Document, Storage, Schemaless

1 INTRODUCTION

For a long time, Relational Database Management Systems (henceforth called *RDBMS*'s) have dominated the data storage landscape, with big competitors such as **MySQL** and **PostgreSQL** standing above others. The guarantees that these types of systems provide are compelling enough to make their users think again before migrating to another solution.

However, as the times come and go, there has been an increasing shift into distributed computing and *"High Availability"* systems. Using this systems alongside traditional RDBMS systems surfaced an issue: RDBMS systems were not able to keep up with the evolution of the software landscape. This led to the big boom of **NoSQL**, which comprises several technologies providing a different set of guarantees for distributed workloads. One such technology is Aerospike [1].

Aerospike is an open-source distributed NoSQL database designed for high-throughput, real-time applications with "blazing-fast reads/writes and unmatched up-time". Its features will be further explored in the following sections of this document.

2 GENERAL INFO

The history of Aerospike is best explained with a transcript from the Wikipedia page[8]:

> Aerospike was first known as Citrusleaf. In August 2012, the company - which had been providing its database since 2010 - rebranded both the company and software name to Aerospike. The name "Aerospike" is derived from the aerospike engine, a type of rocket nozzle that can maintain its output efficiency over a large range of altitudes, and is intended to refer to the

Authors' Contact Information: Guilherme Almeida, up202006137@fe.up.pt; José Osório, up202004653@fe.up.pt; Nuno Pereira, up202007865@fe.up.pt.

> software's ability to scale up. In 2012, Aerospike acquired AlchemyDB and integrated the two databases' functions, including the addition of a relational data management system. On June 24, 2014, Aerospike was open-sourced under the AGPL 3.0 license for the Aerospike database server and the Apache License Version 2.0 for its Aerospike client software development kits (SDKs).

Aerospike is open-source, boasting a generous free tier titled "Community Edition". There are other licenses available[4], namely:

- Standard Edition
- Enterprise Edition
- Aerospike Cloud Managed Service

The main differences between these are the features/services that each license unlocks. Pricing is defined on a per-production workload basis for the Aerospike Enterprise edition and has fixed rates for the Cloud Managed Service Edition.[2].

Due to its open-source nature, there are a lot of community contributions made to the project and surrounding tools, available in Aerospike's Github page[5].

Documentation-wise, Aerospike boasts an extensive library detailing its architecture and design choices, as well as important features and how to use them. There are some examples available but we feel they are lacking.

3 MAIN FEATURES

3.1 Multi-Model Support

Aerospike has a "row-oriented" data model, with each row representing a single record inside the database. Each record has a *Primary-Key* (PK) and a set of "bins", each holding a particular value. These "bins" can be thought of as "columns" of a **RDBMS**.

By default, the PK is the only way to retrieve records from an instance. This is similar to other **Key-Value** stores, such as Redis. However, unlike Redis, with the help of *Secondary Indices* (more on that later), queries can be made using specific bins other than the PK. This means that each key's value is not completely opaque to the data store.

Bins can hold scalar values or more complex data structures, such as JSON data. This allows Aerospike to store entire documents inside a single record, similar to a **Document store**. There are APIs specifically designed to interface with the document features of Aerospike, as mentioned later in section 4.1.

With the help of another piece of software that would sit in front of the Aerospike server, applications can make use of the **Graph** data modeling features provided to better represent their domain-specific data.

Even though Aerospike has, to some degree, some form of support for multiple data-modeling paradigms, the main one is the **Key-Value** paradigm.

3.2 Schemaless Data-Model

As mentioned, Aerospike stores record data inside "bins", which are just containers for arbitrary values.

Aerospike boasts a "schema-less" data model in the sense that the schema used by the database is not enforced by the database technology and is determined by its use. For example, 2 separate records can both have a "bin" with the same name but one record holds an integer in this bin and the other record holds a string.

This flexibility allows applications to evolve their data structures effortlessly without users having to experience database downtime usually associated with schema migrations in RDBMSs, something which has been a staple characteristic of Aerospike.

3.3 Hybrid Memory Architecture

Records in Aerospike are stored inside **Namespace**s.

A Namespace is an object in Aerospike that holds a "data-storage configuration" that is shared by all its records. Aerospike supports the following storage options:

- Flash
- DRAM
- PMEM (Intel Octane)

This technology is called Hybrid Memory Architecture (HMA)[6] and is one of the features that set Aerospike apart from its competition by allowing database administrators to use a single instance of the Aerospike server to serve both low-latency and persistent-storage data access needs.

Each namespace has both a data storage and an index storage configuration. By default, index (meta)data is stored in memory but this is configurable per Namespace. Different combinations of index and data storage options allow Aerospike to suit many different use cases:

(1) It can serve as an in-memory data store (if all data is stored in memory)
(2) Frequently accessed data items can be stored in memory while "colder" data items are persisted in flash storage.
(3) Etc.

By operating close to the physical hardware, and in a fashion similar to the physical operations happening in the physical storage components of a computer (obtained by coding design), Aerospike's HMA is well-suited for real-time applications that require high throughput, low latency, and scalable storage capabilities.

3.4 Cross-Data-center Replication

Aerospike's cross-data-center replication (XDR)[9] feature enables seamless data synchronization and disaster recovery across geographically distributed clusters, enabling applications to maintain uninterrupted operations in the event of localized outages or disasters.

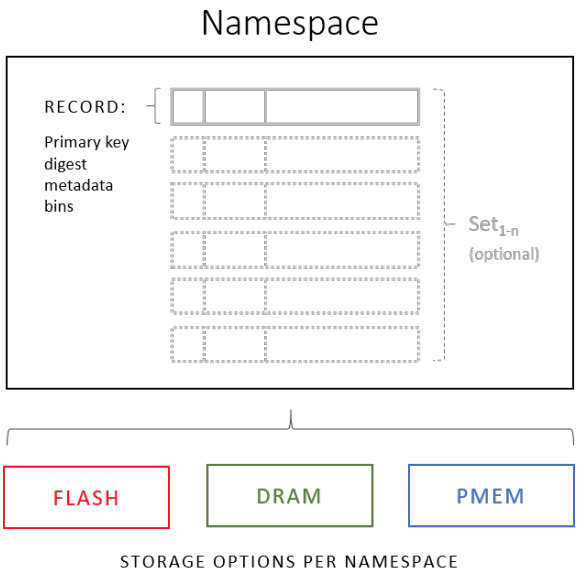

Fig. 1. Conceptual components of Aerospike data model

XDR ensures data consistency and availability by replicating data asynchronously or synchronously between multiple data centers or regions. XDR supports configurable replication policies, allowing developers to prioritize data consistency, latency, or throughput based on their application's requirements.

4 APPLICATION PROGRAMMING INTERFACES AND CLIENT LIBRARIES

4.1 JSON API

Aerospike's JSON API provides a convenient and efficient way for developers to interact with the database using JSON documents.

With these APIs, developers can store, retrieve, update, and query JSON documents using simple and intuitive commands. Additionally, the JSON API supports nested data structures, arrays, and complex data types, providing flexibility for representing diverse types of data within the database.

This API unfortunately only works for the **Aerospike Java client**.

4.2 Smart Client Libraries

Aerospike's smart client libraries are designed to simplify the process of integrating Aerospike into applications written in various programming languages. These client libraries abstract away the complexities of connecting to and interacting with the Aerospike database, allowing developers to focus on building their applications rather than managing database connections and optimizations.

One key feature of Aerospike's smart client libraries is their ability to encapsulate knowledge about the Aerospike cluster topology and perform intelligent routing of requests. This means that the client libraries maintain an up-to-date understanding of the structure of the Aerospike cluster, including information about individual nodes, partitions, and data distribution.

With this knowledge, smart client libraries can automatically route requests to the appropriate nodes in the cluster, balancing the workload evenly and minimizing network latency. Additionally, smart client libraries implement sophisticated load-balancing algorithms to distribute requests across cluster nodes efficiently.

Furthermore, smart client libraries incorporate fail-over mechanisms to handle node failures gracefully. In the event of a node failure, the client libraries can detect the failure and redirect requests to healthy nodes, ensuring uninterrupted service and maintaining high availability.

Aerospike has a wide variety of these libraries as we can see in their documentation[7]

5 CONSISTENCY FEATURES

Aerospike's database can be configured as ***Available and Partition-tolerant (AP)*** or ***Consistent and Partition-tolerant (CP)***.[3]

By default, it is configured to prioritize **availability** and **partition tolerance**, with its **Strong Consistency** mode being only available to the Enterprise Edition and the Aerospike Cloud Managed Service.

As a natural consequence of using the Community Version of Aerospike, this means that we cannot take advantage of its **Strong Consistency** features but it should be mentioned that Aerospike defaulting to a high availability configuration is better, given the nature of the project at hand.

5.1 Strong Consistency

Aerospike's strong consistency guarantee states that all writes to a single record will be applied in a specific order (sequentially), and writes will not be re-ordered or skipped.[3]

This is done per record, with atomic and isolated writes/updates and utilizing a hybrid clock for ordering guarantees and optimized by employing a roster of nodes, defined as the nodes that are part of the cluster in a steady state, this being, a state where all roster nodes are present and where all partitions are current.

This configuration is modifiable to allow for:

- **linearizing reads**, guaranteeing a single linear view of data
- **session consistency**, ensuring sequential updates for individual processes
- **relaxed consistency** which allows reads from replicas or unavailable partitions

6 REPLICATION FEATURES

Aerospike utilizes a combination of the Paxos[14] consensus protocol and gossip-based communication[12] for distributed replication. Paxos ensures strong consistency guarantees, while gossip protocols facilitate

efficient and scalable communication between cluster nodes. This replication strategy enables Aerospike to achieve high availability and fault tolerance while maintaining low latency and high throughput.

7 DATA PROCESSING FEATURES

One of the features Aerospike provides that allows for powerful data processing capabilities is the presence of **User Defined Functions**, (UDF). These are Lua scripts that can run on the Aerospike server.

They come in 2 variants:

- Record UDFs
- Stream UDFs

At first glance these look like *Stored Procedures* in classic RDBMS systems. However, UDFs are more limited in what they can do inside the database when compared to a normal Stored Procedure.

7.1 Record UDFs

Record UDFs run on a single record. They can implement all CRUD operations.

These are usually used for:

- Implementing features not yet present in the Aerospike server itself
- Executing many operations inside a transaction
- Etc.

7.2 Stream UDFs

Stream UDFs, in contrast with Record UDFs, execute on a stream of Records in a read-only fashion. They are useful for aggregating and transforming query results and are the closest concept to *Map-Reduce* in Aerospike. Stream UDFs that run on the client must be present on both the server and the client since its execution is split between these 2 environments: the servers/nodes filter and map results, which are then aggregated and reduced by the smart client.

8 ADEQUATE VS. INADEQUATE SCENARIOS

As mentioned, Aerospike is designed to provide uninterrupted, low-latency, high-throughput service to client applications. As such, adequate use of Aerospike can involve contexts that require frequent data access. According to reports, the first commercial uses of Aerospike were aimed at collecting ad metrics.

In scenarios where an application relies heavily on relational data models, with intricate relationships, joins, and foreign keys, Aerospike does not deliver schema flexibility enough to make it a better option over a traditional RDBMS. The limited support for transactional operations can also pose a problem in scenarios where stricter requirements are at play, such as financial and healthcare businesses.

9 ADVANTAGES/DISADVANTAGES

One of the main advantages when using Aerospike is the fact that, due to its Hybrid Memory Architecture, and depending on the database configuration, it can serve as both a database and a cache. All these features sit on top of the extensive replication features that allow Aerospike to provide nearly uninterrupted service across any cluster size.

On the other hand, developing using Aerospike can be quite bothersome, both due to the large number of tools needed to manage the database (*aql* is used to inspect and modify database contents, *asadm* allows management operations on a cluster, like creating indices) but also to the lackluster documentation regarding some important aspects of its architecture and features.

10 PROTOTYPE PRESENTATION

10.1 Topic and Dataset

Our prototype focuses on the development of an instant-messaging platform, aiming to provide instantaneous delivery of messages, and support for high concurrency, scalability, and fault tolerance. The dataset used in our prototype includes messaging logs generated by a handmade script that takes advantage of the ***Faker.js***[10] library and its data mocking capabilities to output a readable data file by the AQL tool. Additionally, the dataset contains channels, servers, and messages, forming the foundation of our distributed chat application.

10.2 Conceptual Data Model

We have developed a conceptual data model to illustrate the relationships and entities within our chat application. For this purpose, we opted for a class diagram[13] which provides a high-level overview of the entities such as users, channels, servers, and messages, along with their associations and attributes.

The data schema of the prototype eventually shrunk down from this current one for simplicity's sake.

10.2.1 User. The User entity encapsulates essential user information within the system, including the username, email address, and login passwords. Additionally, it stores user status indicators and profile pictures for personalized user experiences.

10.2.2 Channel. The Channel entity serves as a conduit for messaging between users within the platform. Each channel is uniquely named and functions as a repository for messages exchanged among users.

10.2.3 Server. The Server entity acts as a container for organizing channels in a semantically meaningful way. Similar to channels, servers possess distinct names, icons, and regions, and they facilitate the aggregation of users and channels associated with them.

10.2.4 Message. The Message entity embodies the content exchanged between users within the platform. Additionally, each message is linked to the user who authored it.

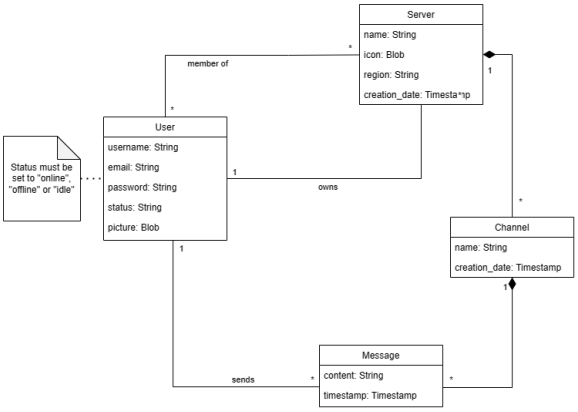

Fig. 2. Prototype Application's Class Diagram

10.3 Physical Data Model

The physical data model we arrived at can be summarized by the following picture:

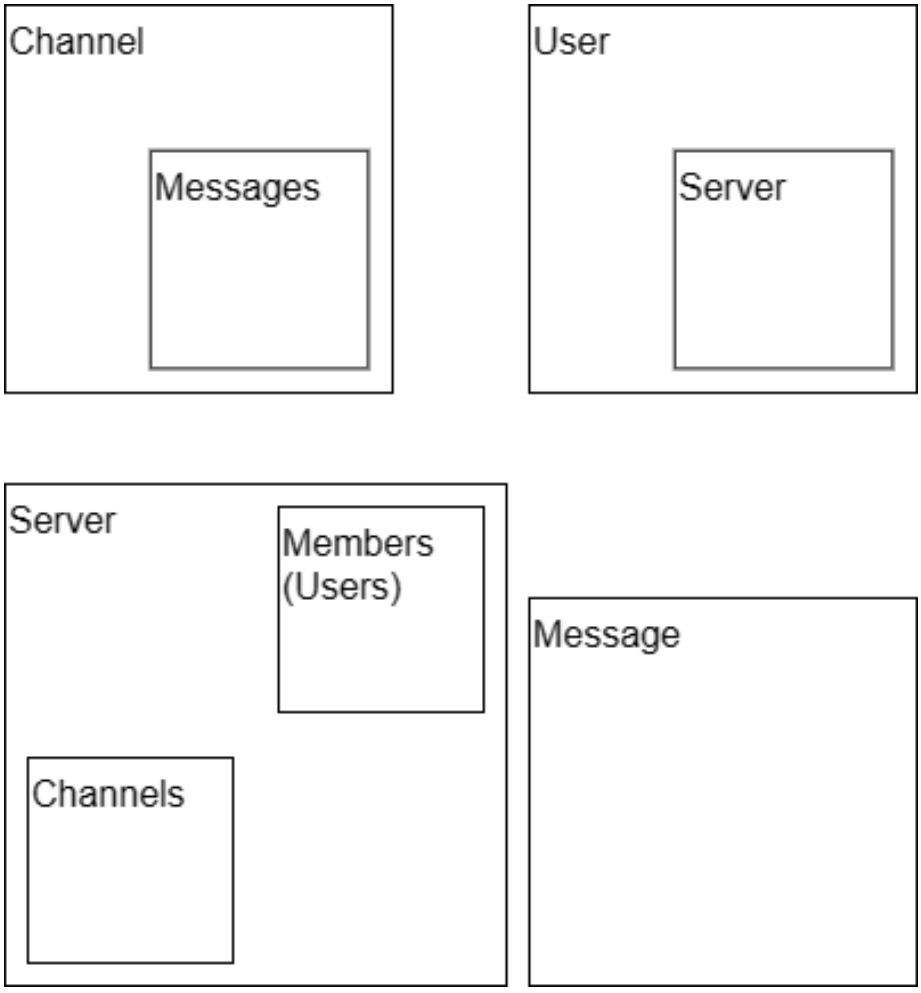

Fig. 3. Physical Data Model

The main feature of our data modeling is the presence of "previews" inside aggregates. These are used in place of the actual object to reduce the number of hits to the database made by the application. This also improves load times since all data needed is loaded right away.

However, this structure has one major drawback that needs to be handled with care: synchronizing previews with their respective object. This can be solved partially with some of the operations Aerospike provides but still requires some intervention on the application side.

Despite this drawback, we believe this modeling to achieve our goals for the prototype.

10.4 Implemented Data Structures with Illustrative Values

The data structures implemented rely on both the *Key-Value* and *Document* capabilities of Aerospike: most records are fetched using only their PK but some of the bins in them contain JSON documents. As already mentioned, these JSON documents serve as "previews" of the actual data they represent. For example, as you can see in Appendix A.1 *Users* have a preview of the *Servers* they are a member of to remove many possible database calls (fetching the server information for every server in the user record). This introduces data redundancy, leading to possible exhaustion of storage space, but reduces application load times significantly, something crucial in the type of application that is our prototype.

The same rationale can be applied to the other data types.

10.5 Examples of Data Processing Features

Since Aerospike features Stream UDFs, suitable for stream processing, we tried to implement a rudimentary form of record counting. Even though we managed to load the script into our Aerospike instance, we did not manage to correctly invoke it when performing operations: the existing documentation is lackluster and the source code for the various pieces of what makes a full Aerospike installation also provided little help in this matter. As such, the processing steps that could happen inside the database were moved to the application.

10.6 Architecture of the Prototype

Our prototype adopts a distributed and scalable **Client-Server** architecture to accommodate the requirements of our instant messaging platform. At its core, the architecture revolves around three main components: the frontend, backend, and database.

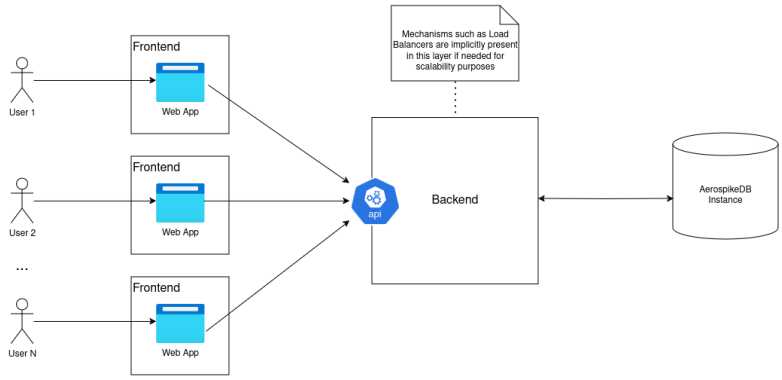

Fig. 4. Prototype Application's Architectural Overview

10.6.1 Frontend. Each user is served by an individual web-app instance, implemented using SvelteKit[11]. This approach ensures a personalized and responsive user experience, with each web-app instance tailored

to the preferences and interactions of its respective user. The web-app handles user interface rendering and user input processing, providing seamless communication with the backend module.

10.6.2 Backend. The backend module, implemented in Node.js through SvelteKit, contains the main business logic of our chat application. It serves as the intermediary between the frontend and the database, handling user authentication, message processing, channel management, and other core functionalities.

Load balancers and such mechanisms are implicitly present in this layer to ensure scalability and fault tolerance. They distribute incoming requests across multiple instances of the backend module, optimizing resource utilization and improving system performance. Additionally, load balancers monitor the health of backend instances and automatically route traffic away from unhealthy or overloaded nodes, enhancing system reliability and availability.

Note: In our prototype load balancing mechanisms were not implemented, since it wasn't the main focus of the project and we did not consider it as a priority. Having this said, we consider it was worth mentioning, because in a real-world scenario they would be crucial to the availability of the system as a whole.

10.6.3 Database. The database stores and manages the persistent data used by our chat application, including user profiles, messages and channels. Aerospike DB serves as the underlying database technology, providing high throughput, low latency, and seamless scalability.

The architecture of our prototype is designed to scale horizontally, allowing us to handle increasing user loads and data volumes without sacrificing performance or reliability. By leveraging distributed computing principles and modern technologies, our architecture enables seamless communication and collaboration among users in our instant messaging platform.

REFERENCES

[1] Aerospike 2024. *Aerospike Database.* Aerospike. https://aerospike.com/ [Online; accessed 29-April-2024].

[2] Aerospike 2024. *Aerospike Database's Pricing for its Cloud Managed Service Edition.* Aerospike. https://aerospike.com/products/aerospike-cloud/aerospike-cloud-pricing/ [Online; accessed 29-April-2024].

[3] Aerospike 2024. *Aerospike Database's Consistency.* Aerospike. https://aerospike.com/docs/server/architecture/consistency [Online; accessed 30-April-2024].

[4] Aerospike 2024. *Aerospike Database's Different Editions and Licensing models.* Aerospike. https://aerospike.com/products/features-and-editions/ [Online; accessed 29-April-2024].

[5] Aerospike 2024. *Aerospike Database's GitHub Repository.* Aerospike. https://github.com/aerospike/aerospike-server [Online; accessed 29-April-2024].

[6] Aerospike 2024. *Aerospike Database's Hybrid Memory Architecture.* Aerospike. https://aerospike.com/products/features/hybrid-memory-architecture/ [Online; accessed 30-April-2024].

[7] Aerospike 2024. *Aerospike Database's Hybrid Memory Architecture.* Aerospike. https://aerospike.com/developer/client [Online; accessed 30-April-2024].

[8] Wikipedia 2024. *Aerospike Database's Wikipedia Entry.* Wikipedia. https://en.wikipedia.org/wiki/Aerospike_(database) [Online; accessed 29-April-2024].

[9] Aerospike 2024. *Aerospike Database's Cross-Datacenter Replication (XDR) Architecture.* Aerospike. https://aerospike.com/docs/server/architecture/xdr [Online; accessed 30-April-2024].

[10] Faker 2024. *Faker Website.* Faker. https://fakerjs.dev/ [Online; accessed 13-Maio-2024].

[11] Aerospike 2024. *SvelteKit.* Aerospike. https://kit.svelte.dev/ [Online; accessed 30-April-2024].

[12] Wikipedia contributors. 2023. Gossip protocol — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Gossip_protocol&oldid=1187077867 [Online; accessed 30-April-2024].

[13] Wikipedia contributors. 2024. Class diagram — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Class_diagram&oldid=1216657280 [Online; accessed 30-April-2024].

[14] Wikipedia contributors. 2024. Paxos (computer science) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Paxos_(computer_science)&oldid=1213846427 [Online; accessed 30-April-2024].

A APPENDIX I

A.1 AQL Script Example

```
-- Users
INSERT INTO test.users (PK, username, email, password, image, servers)
VALUES ('user1', "testuser", 'testuser@mail.com', "password",
"https://picsum.photos/300/300",
JSON('{"server1": { "id": "server1", "name": "testserver", "image": "https://picsum.photos/300/300"},
"server2": { "id": "server2", "name": "testserver2", "image": "https://picsum.photos/300/300"}}'));

-- Servers
INSERT INTO test.servers (PK, name, image, channels,members)
VALUES ('server1', "testserver", "https://picsum.photos/300/300",
    JSON('{"channel1": { "id": "channel1", "name": "testchannel", "server": "server1"},
    "channel2": { "id": "channel2", "name": "testchannel2", "server": "server1"}}'),
    LIST('["user1"]'));

-- Channels
INSERT INTO test.channels (PK, name, server, messages)
VALUES ('channel1', "testchannel", "server1", LIST('["message1", "message2"]'));

INSERT INTO test.channels (PK, name, server, messages)
VALUES ('channel2', "testchannel2", "server1", LIST('["message3", "message4"]'));

-- Messages
INSERT INTO test.messages (PK, senderId, senderName, senderImage, content, timestamp)
VALUES ('message1', "user1", "testuser",
"https://picsum.photos/300/300","Conteúdo da mensagem", 1715278308000);
```

```
INSERT INTO test.messages (PK, senderId, senderName, senderImage, content, timestamp)
VALUES ('message2', "user2", "testuser2",
"https://picsum.photos/300/300","Conteúdo da mensagem1", 1715278308000);

INSERT INTO test.messages (PK, senderId, senderName, senderImage, content, timestamp)
VALUES ('message3', "user1", "testuser",
"https://picsum.photos/300/300","Conteúdo da mensagem2", 1715278308000);

INSERT INTO test.messages (PK, senderId, senderName, senderImage, content, timestamp)
VALUES ('message4', "user2", "testuser2",
"https://picsum.photos/300/300","Conteúdo da mensagem3", 1715278308000);

-- Keywords
INSERT INTO test.keywords (PK, messageIds)
VALUES ('mensagem', LIST('["message1", "message2", "message4", "message7"]'));
```