

Article

A Model for a Serialized Set-Oriented NoSQL Database Management System

Alexandru-George Șerban *  and Alexandru Boicea 

Computer Science Department, National University of Science and Technology Politehnica of Bucharest, 060042 Bucharest, Romania; alexandru.boicea@upb.ro

* Correspondence: alexandru.serban00@stud.acs.upb.ro

Abstract

Recent advancements in data management highlight the increasing focus on large-scale integration and analytics, with the management of duplicate information becoming a more resource-intensive and costly task. Existing SQL and NoSQL systems inadequately address the semantic constraints of set-based data, either by compromising relational fidelity or through inefficient deduplication mechanisms. This paper presents a set-oriented centralized NoSQL database management system (DBMS) that enforces uniqueness by construction, thereby reducing downstream deduplication and enhancing result determinism. The system utilizes in-memory execution with binary serialized persistence, achieving $\mathcal{O}(1)$ time complexity for exact-match CRUD operations while maintaining ACID-compliant transactional semantics through explicit commit operations. A comparative performance evaluation against Redis and MongoDB highlights the trade-offs between consistency guarantees and latency. The results reveal that enforced set uniqueness completely eliminates duplicates, incurring only moderate latency trade-offs compared to in-memory performance measures. The model can be extended for fuzzy queries and imprecise data by retrieving the membership function information. This work demonstrates that the set-oriented DBMS design represents a distinct architectural paradigm that addresses data integrity constraints inadequately handled by contemporary database systems.

Keywords: set data structures; serialized database management systems; data uniqueness constraints; SQL semantics; NoSQL systems

1. Introduction

A limited number of database engines, within the broader constellation of real-world implementations, adopt set-oriented semantics as a foundational architectural principle. In the proposed model, operations for updating, inserting, deleting, and performing exact searches on individual elements are designed to exhibit $\mathcal{O}(1)$ time complexity, thereby ensuring high performance. At the core of this approach are hash-based sets, which are lightweight data structures that are practical to implement and serve as a focal point for efficient data representation and manipulation. The objective is to design and evaluate a database model that guarantees uniqueness by construction, rather than relying on user-defined constraints or query-level deduplication.

Proper big data analysis requires unique entries. Sets are appropriate structures for working with large volumes of data [1]. General performance metrics for a big data system [2,3] highlight: scalability, real execution time, effective use of resources, and lower energy consumption for the server hosting the database engine.



Received: 4 December 2025

Revised: 7 January 2026

Accepted: 9 January 2026

Published: 13 January 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and

conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

Contemporary NoSQL systems abandon relational constraints in favor of schema flexibility and horizontal scalability, yet they do not address the central problem of enforcing data uniqueness as a core architectural principle. Document-oriented systems such as MongoDB and key-value stores like Redis offer substantial flexibility; nevertheless, they neither treat set-oriented semantics as a primary architectural concern nor systematically guarantee the elimination of duplicate data without explicit application-level handling.

The research question addressed in this paper is as follows: *To what extent is a DBMS architecture that enforces set semantics at the system level—by storing committed data in serialized binary files and managing the temporary state of database objects in the server’s main memory—competitive with other NoSQL systems, such as Redis and MongoDB?*

The research contributions as part of this work highlight:

- Feasibility of designing and implementing a DBMS based on set-theoretic specifications with explicit consistency semantics, along with an evaluation of system stability during incremental insertion and commitment of a medium-sized dataset;
- A comparative workload performance analysis against two NoSQL systems, namely Redis and MongoDB.

2. Background and State of the Art

The purpose of this paper is to model a set-oriented NoSQL DBMS, ensuring the removal of redundancies and deduplicated lines. An early multi-paradigm programming language called SET Language (SETL) conceptualized operations such as union or intersection [4,5].

One critique of the Structured Query Language (SQL) is that it deviates from Codd’s relational model in that, following the execution of a SELECT command, duplicate rows can be fetched [6]. A table, which corresponds to a relation, is defined as a subset of the Cartesian product of the attributes’ domain, i.e., the set of values for columns [7]. SQL initially had no support for set and bag operations. Additionally, it illustrates ambiguous semantics, which can produce different results depending on the dialect [8].

A table is not simply considered a list of results, but a set of tuples. SQL deviates from the relational model [9] by permitting the existence of repeated elements. These changes amplify downstream complications: implementation of additional deduplication mechanisms and multiplication of anomalous data from redundant storage.

Incongruencies between SQL and the relational model led to calls for alternative design approaches [10]. Set operations follow a declarative paradigm. For a dataset comprising unique elements, the redundancies are removed, eliminating the need to delete deduplicated information. This approach resolves insertion anomalies while also achieving consistency, as updating the value of an element in a set or a related data structure occurs only once. The proposed model stores relevant committed information as serialized data in compressed binary datafiles.

Redundancy-free databases are flexible to modifications, leading to the removal of data anomalies and inconsistencies [11,12]. A set-oriented database employs flexible data structures, as changes to data are localized and operations such as inserting, updating, and deleting an element occur only once. Similar normalized systems are easier to maintain than repositories with redundancies and deduplicated elements [12,13]. By definition, a set data structure ensures unique members.

The methodology prioritizes correctness in duplicate elimination and result set stability, rather than optimizing solely for throughput. The architecture and implementation choices emphasize simplicity and transparency, ensuring that the effects of enforcing set semantics can be isolated. A large body of research addresses duplicate issues through data cleaning and integration by means of ETL processes [14], CDC data deduplication using

sliding-window technique [15], and record linkage [16], particularly in heterogeneous domains for information systems [17]. These approaches normally function as preprocessing or postprocessing stages that depend on heuristics instead of enforcing strict conditions. While they enhance data quality, they do not fundamentally change the semantics of the underlying database systems.

In the context of modern data storage challenges for deep learning applications, a benchmark study compares the download times and disk usage of three data formats—raw files, Python’s `pickle` 2.1, and binary large objects (BLOBs)—across three NoSQL key-value storage databases: Redis, MongoDB, and Cassandra [18].

3. Related Work

This section situates the proposed set-oriented model within the wider context of relational theory, SQL semantics, data quality, and contemporary NoSQL database systems.

The gap between theory and practice becomes evident for modern data-intensive systems, such as data lakes and machine learning analytics workflows [19], in various ways:

1. Compromised statistical viability when models assume unique records but encounter duplicate information;
2. Increased computational overhead due to the need for deduplication steps during data preprocessing and ingestion;
3. Inconsistent and non-deterministic result sets and different cardinality across SQL dialects;
4. Additional deduplication logic handling, leading to downstream complexity and potential errors. It further requests functional indexes, cleaning operations, application-level logic to ensure data integrity, or additional queries.

This work is placed at the intersection of set-oriented database design, NoSQL storage models, and query language semantics. Query loads with a high rate of replicated information tend to incur growing costs due to index maintenance and constraint checking.

3.1. Set-Oriented Databases vs. SQL Semantics

Relational database theory is based on set semantics, whereas SQL systems primarily adhere to multiset logic (bag semantics). This discrepancy enables the presence of duplicate tuples as a fundamental feature, which complicates obtaining deterministic results, subsequent data processing, and query reasoning. Although contemporary SQL systems offer tools such as primary and unique constraints, deduplication queries, and conflict-resolution operators, maintaining global uniqueness continues to be a complex and error-prone challenge for users. The differences between the strict set semantics of the relational model and SQL are historically grounded [20].

The proposed set-oriented system’s design ensures that no duplicate records are returned or stored, thereby eliminating the need for explicit deduplication queries. In contrast, achieving comparable correctness guarantees in SQL requires the use of additional operators or constraints, which consequently increases the complexity of the queries.

The conflict between SQL’s permissive multiset logic and the relational model’s set-based semantics creates challenges in data management: additional computational processing power is requested for replicated information, duplicate data propagates through ETL pipelines [21], and result sets become inconsistent across different SQL configurations.

SQL attempts to provide mechanisms for duplicate data related issues, including:

- UNIQUE and PRIMARY KEY constraints imposed during insertion;
- DISTINCT operator and GROUP BY statement to eliminate duplicates in query results;

- UPSERT operators to handle conflicts during data insertion or updates, such as ON CONFLICT in PostgreSQL, MERGE in both SQL Server and Oracle.

Yet, these solutions ultimately delegate the responsibility of ensuring data uniqueness to the schema designer and application logic, rather than embedding it as a core architectural principle of the DBMS.

3.2. NoSQL Systems and Data Uniqueness

Although NoSQL systems are designed to meet specific scalability and flexibility needs, they do not address the uniqueness constraint issue at the architectural level in a systematic way.

Key-value stores such as **Redis** offer in-memory performance, but lack an adequate native query language. Additionally, it does not natively support data encryption in transit, relying instead on TLS. Compared to most of the relational database models, transactional support is limited, as Redis implements the MULTI/EXEC commands for grouping multiple operations into a single transaction, but it does not provide full ACID compliance. Support for rolling back transactions is non-existent, as performing this operation would have a significant impact on the simplicity and performance of Redis [22].

Document databases such as **MongoDB** provide flexible storage for semi-structured data, but they ensure uniqueness only through explicit indexing on specific fields. Duplicate detection and deletion across entire documents remain the responsibility of the application, and NoSQL query languages do not enforce semantic guardrails to guarantee that result sets are unique collections.

4. Methods

The client-server architecture for the proposed set-oriented NoSQL DBMS model involves software components that allow for the transmission/reception of variable-length messages, representing commands, via TCP sockets. A proof-of-concept implementation in Python3 utilizes a two-tier client-server architecture.

4.1. General Architecture

This subsection outlines the architectural design principles, detailing the specifications of the three primary organizational levels—internal, conceptual, and external—and the implementation of the database engine's software components.

A proposed set-oriented database engine demonstrates characteristics of an in-memory database by maintaining the temporary state of objects in the server host's RAM. At the same time, it provides transactional support, with committed user data persistently stored in a binary datafile that retains a serialized dictionary.

In the general case, let $t_1, t_2, \dots, t_n$ represent n unique identifiers for the database objects, and let $o_{(t_i)_j}$ denote the j -th name for an object of type t_i with the associated value $v[t_i][o_{(t_i)_j}]$. The memorized dictionary D is defined as follows:

$$D = \{t_1 : \{o_{(t_1)_1} : v[t_1][o_{(t_1)_1}], \dots\}, \dots, t_n : \{o_{(t_n)_1} : v[t_n][o_{(t_n)_1}], \dots\}\} \quad (1)$$

An interaction of user X with the set-oriented database engine is formalized as the mapping between the set of commands $\mathcal{C}$ and the corresponding data store, denoted as $\phi : \mathcal{C} \rightarrow D_X$. The last accessed or modified state of the objects in the dictionary D_X is held in RAM, while the committed information is saved in a non-volatile manner in the user's binary datafile.

The set of commands $\mathcal{C}$ specifies the external level of the proposed DBMS, whereas the function ϕ describes the conceptual level. The entire stored data for user X , representing the

internal level, is encapsulated in the dictionary D_X . A set $\mathcal{U} = \{D_1, D_2, \dots, D_x\}$ represents the collection of schemas for x users. When the database engine commits the serialized dictionary D_X , it performs a DUMP operation—labeled by the function $d : D_X \rightarrow \{0, 1\}^*$ —to write it into the associated binary datafile.

Therefore, the composition $d(\phi)$ occurs, where the codomain of ϕ corresponds to the domain of D_X . The set $\mathcal{C}$ consists of commands which operate on the types $t_1, \dots, t_n$, as defined in the dictionary D_X . An inverse function of $d : D_X \rightarrow \{0, 1\}^*$ is the operation LOAD, represented by the mapping $l : \{0, 1\}^* \rightarrow D_X$, which reads the binary datafile and reconstructs the serialized dictionary D_X in the server's main memory. If D_{X_1} is an instance of D_X , then $l(d(D_{X_1})) = D_{X_1}$.

Maintaining the previous m instances or snapshots of D_X : $D_{X_1}, D_{X_2}, \dots, D_{X_m}$, is inefficient from a storage perspective, especially in RAM. A different structure is proposed to facilitate executing transactional operations, as exemplified by COMMIT and ROLLBACK. The previous states of the database objects are stored in a stack S , with values pushed following three DML commands: INSERT, UPDATE, and DELETE.

Only the prior snapshot of the modified object of type t_i is added to S , not the entire dictionary instance of D_X , referred to as D_X^{previous} . If object $o_{(t_i)_j}$ is modified, then $D_X^{\text{previous}}[t_i][o_{(t_i)_j}]$ is pushed onto S .

When the user invokes COMMIT, the current state of the dictionary instance D_X^{current} is permanently saved in the binary datafile by performing the DUMP operation. The stack S is then cleared, as the temporary states of the database objects are no longer needed.

When the user invokes ROLLBACK, the last state of $o_{(t_i)_j}$ is popped from S and restored in the current dictionary instance of D_X^{current} . Rolling back to defined savepoints is achieved by associating labels with specific positions in the stack S . The user can revert to a savepoint by popping elements from S until reaching the designated label, restoring the corresponding previous states of the modified objects in D_X^{current} .

The advantage of this approach is that only the modified objects are stored in S , leading to reduced memory usage compared to storing entire snapshots of D_X .

Semantic enforcements for object $o_{(t_i)_j}$ include: $\forall x, y \in v[t_i][o_{(t_i)_j}], x \neq y$ and $\forall x \in v[t_i][o_{(t_i)_j}], x \neq \text{null}$. Database records containing null values can introduce query complexity, ambiguity, integrity issues, an increased risk of errors, as well as performance overhead in indexed searching [23]. The proposed model requires that all elements in a set be unique, non-null, and of a defined data type.

4.1.1. Data Organization

The modelling of the data storage and interaction within the DBMS levels is illustrated in Figure 1.

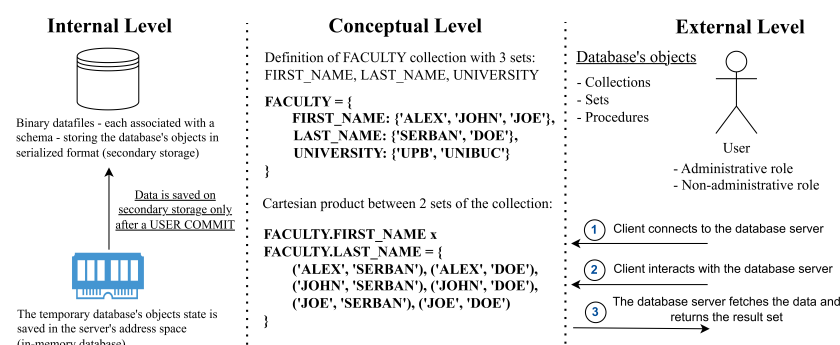


Figure 1. The internal, conceptual, and external levels of the proposed set-oriented DBMS.

The **internal level** is described by binary datafiles, each associated with one user's schema. The temporary state of the database schema objects, created and modified by clients, is memorized locally in a stack. Uncommitted data is retained in the server process's address space, which underpins a working in-memory database. In addition, the server's database engine writes on secondary storage only after the user explicitly commits. As a drawback, the access time for a single overloaded serialized binary datafile increases. In such cases, multiple datafiles can be created, with one control file pointing to the location of the stored queried database objects. For the simplified DBMS model, where users do not access large quantities of data, no control file is created.

An exemplified structure for the **conceptual level** is a collection named FACULTY, comprising three sets—FIRST_NAME LAST_NAME and UNIVERSITY. One datafile stores a serialized dictionary data structure, whose three keys correspond to the database's objects: collections, sets, and procedures. Members belonging to two sets can be correlated by performing Cartesian product. Unconforming tuples can then be removed individually from the resulting data. The difference between sets marks another operation that leads to the elimination of elements. Database procedures are viewed as a stored list of compiled commands in advance that are sequentially executed.

Externally, the client can assume either an administrative or non-administrative role, enabling access to and querying of database objects from one or more schemas. Users authenticate before submitting queries to the server, which then responds either with an error message or by returning the result set.

4.1.2. Software Components Implementation Principles

The software components within the two-tier client-server architecture implement distinct functions:

- **Server side:** interprets the commands received from users. The processing of the received queries entails parsing, compiling, executing, and fetching of the results. The server functions as the database engine. Parsing includes the syntactic analysis, whereas the compilation completes the semantic analysis. The temporary state of a database object is saved in a stack and memorized in the server process's address space. At the beginning of a session, the server reads the binary datafile corresponding to a single user schema, in order to fetch the serialized dictionary with three keys associating the database's objects. The binary datafile is written by the server process only when the user chooses to commit the data, at which point the temporary objects' state from the main memory is permanently saved.
- **Client side:** sends commands to the centralized database server following authentication. At this stage, the client performs basic syntactic checking to ensure that the query is properly formatted. One user interacts with the centralized database server through a GUI with multiple window layouts: authentication, registration, and terminal—a multiline graphical element from which commands can be sent.

Loosely coupled DBMS software components provide increased scalability, modularity, and flexibility [24]. The server's main class—named DatabaseServer—constructor implementation in Python3 is presented in code Listing 1.

It inherits from another class, ProcessCommands, whose methods `send_server_response`, `receive_user_cmd`, `parse_user_cmd`, `compile_user_cmd`, and `execute_user_cmd`—are invoked by the DBMS server to receive and process the commands and send the fetched result set to the user. Errors during the execution of either the client or the server processes are marked by the logger in a log file.

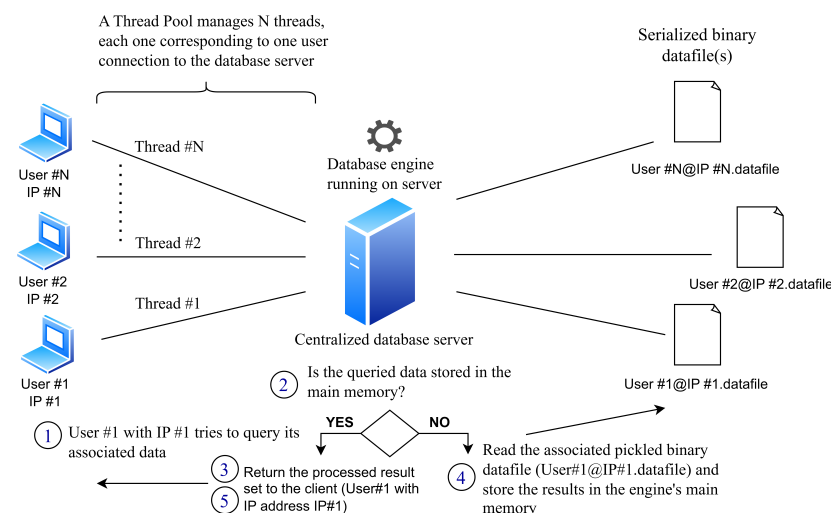
Listing 1. DBMS server's main class constructor implementation and initialization in Python3.

```

1 class DatabaseServer(ProcessCommands):
2     logger = logging.getLogger(__name__)
3
4     def __init__(self, serverIP = 'localhost', serverPort = 20000):
5         super().__init__()
6         # Define and initialize server TCP/IPv4 endpoint
7         with socket.create_server((serverIP, serverPort), family = socket.
8             AF_INET) as self.serverSocket:
9             try:
10                 self.serverSocket.setsockopt(socket.SOL_SOCKET, socket.
11                     SO_REUSEADDR, 1)
12                 self.dbmsClients, self.exitServer = set(), False
13                 logging.basicConfig(filename = 'dbms-server.log', level =
14                     logging.INFO, format = '%(asctime)s %(levelname)s %(module)s %(funcName)s: %(message)s')
15                 self.logger.info('Info: DBMS Server started IP address: {},
16                     port: {}'.format(serverIP, serverPort))
17                 self.exec_server()
18             except OSError:
19                 self.logger.error('Error: Could not create TCP/IPv4 socket
20                     for the DBMS server')
21                 exit(1)

```

The diagram presenting the centralized database server managing more users is reflected in Figure 2.

**Figure 2.** The two-tier client-server architecture of the proposed system which handles multiple connections. The clients are numbered from #1 to #n.

A singular connection to the set-oriented database server within the model is recognized by: username, client's IP address and port. The `SO_REUSEADDR` option is used by the database system's sockets for higher availability and to avoid remaining in a `TIME_WAIT` state [25].

Multiple client connections are managed by a Thread Pool. To achieve optimal performance, the maximum number of threads handled by the Thread Pool is determined by the number of CPU cores available on the database server host, as implemented in code Listing 2.

Listing 2. Handling of multiple connections using the DBMS server’s main class method `exec_server`.

```

1  def exec_server(self):
2      # Handle multiple user connections using a ThreadPool
3      with concurrent.futures.ThreadPoolExecutor(max_workers =
4          multiprocessing.cpu_count()) as client_exec:
5          try:
6              while True:
7                  try:
8                      if self.exitServer:
9                          self.logger.info('Info: DBMS Server will be
10                             stopped')
11                          os.kill(os.getpid(), signal.SIGKILL)
12
13                          clientConn, clientAddr = self.serverSocket.accept()
14                          # Check if the number of connected clients does not
15                          exceed CPU cores count
16                          if len(self.dbmsClients) < client_exec._max_workers
17                          : client_exec.submit(self.handle_client_conn, clientConn, clientAddr)
18                          else:
19                              clientConn.sendall(b'END')
20                              clientConn.close()
21                      except Exception as e:
22                          self.logger.critical('Critical: DBMS server process
23                             stopped running normally, error: {}'.format(e))
24                          sys.exit(1)
25                      finally: self.logger.info('DBMS server has stopped')

```

Each user is identified by their associated schema, which provides access exclusively to the objects within that schema. If the queried data is not found in the main memory, the database engine reads the serialized binary datafile. At the start of each session, the server process updates the in-memory database with the most recently committed data from the user’s datafile. The corresponding datafile for User#i with IP address IP#i follows the naming convention: User#i@IP#i.datafile.

The proposed model optionally supports a single account per client with an administrative role, allowing for access to view and manipulate objects beyond the current schema. In this simplified version, database privileges cannot be transferred between users. The credentials of registered users are stored in a binary file, which the server process accesses when a client attempts to log in.

4.1.3. System Trade-Offs

Database architecture designers can better understand system resource usage by correctly identifying the relevant trade-offs.

The committed user information is stored in a serialized dictionary on secondary storage. A lower load factor for a hash table results in fewer collisions. However, the current uncommitted data is retained in RAM, and handled by the server’s host process.

A classic time-memory trade-off appears between the access time and the storage size of the serialized dictionary. The relation between the two metrics is detailed in Section 4.3. Each COMMIT operation necessitates the complete serialization of the in-memory dictionary D_X , resulting in a time complexity of $\mathcal{O}(n)$, which is directly proportional to the size of the dataset. Deserialization during server startup exhibits similar scaling behavior, potentially creating bottlenecks during session initialization, particularly when dealing with large schemas. The size of the binary datafile increases linearly with the number of entries.

The trade-off between strong consistency guarantees and latency revealed unique characteristics of the proposed set-oriented system when compared to SQL, Redis, and Mon-

goDB. Mainly, it has underlined a subtle transactional architectural design principle for a NoSQL key-value store. Enforcing strong consistency guarantees through explicit commit operations introduces latency compared to in-memory databases that automatically persist changes. This trade-off is particularly pronounced in scenarios involving frequent write operations, where the overhead of committing data to secondary storage can lead to increased response times. Frequent UPDATE and DELETE operations push multiple snapshots of the previous database state onto stack S , thereby consuming additional RAM and degrading the performance of ROLLBACK operations. Write amplification becomes increasingly significant in workloads with selective updates to large datasets, as the entire serialized structure is rewritten to secondary storage. Additionally, achieving interconnectivity with business intelligence platforms such as Tableau or Power BI necessitates the development of custom adapters.

4.2. Database Operations

This subsection highlights the novel communication principles for a simplified database model, which involves transmitting generalized data types as a bytestream of pickled and encrypted data between the client and server. Furthermore, the processing of commands is presented as a pipeline with three stages:

- parsing;
- compiling;
- executing, and fetching of the results.

Let $c \in \mathcal{C}$ —as defined in the Section 4.1—represent a command sent by the client to the set-oriented database server, $o_{(t_i)_j}$ the queried object, and P an optional predicate. The specifications of c include the tuple $(o_{(t_i)_j}, P)$.

Selecting data from object $o_{(t_i)_j}$ that satisfies predicate P is denoted as follows: $\text{SELECT}(o_{(t_i)_j}, P)$. The performed operation entails fetching the appropriate result set from the current instance of the serialized dictionary D_X^{current} . Searching occurs over the values of object $o_{(t_i)_j}$ in D_X^{current} , returning only those that satisfy P . The queried codomain is denoted as follows: $v[t_i][o_{(t_i)_j}]$.

Let $c \in \mathcal{C}$ represent a command defined by the tuple $(o_{(t_i)_j}, P)$, which can be expressed as $c(o_{(t_i)_j}, P)$, and let $\phi : \mathcal{C} \rightarrow D_X$ be the mapping that associates commands in $\mathcal{C}$ with the corresponding data store in the proposed DBMS. Executing $\phi(c(o_{(t_i)_j}, P))$ involves two steps:

1. Identifying the queried values $v[t_i][o_{(t_i)_j}]$ for the object $o_{(t_i)_j}$ of type t_i in the serialized dictionary D_X ;
2. Filtering the results according to the predicate P , i.e., selecting those values for which $P(v[t_i][o_{(t_i)_j}])$ holds true.

The processing of c involves the composition of the three functions in this order: executing, compiling, and parsing.

4.2.1. Client-Server Communication

The communication methodology which involves two TCP endpoints and the transmission of the public keys is detailed in Figure 3.

The dataflow between the client and the database server is viewed as sending/receiving a bytestream encrypted data, with serialization handled using Python's pickle module [26].

During each server and client process run, a public-private key pair is generated to ensure secure communication between the TCP endpoints in each direction. The proposed DBMS model guarantees encryption in transit.

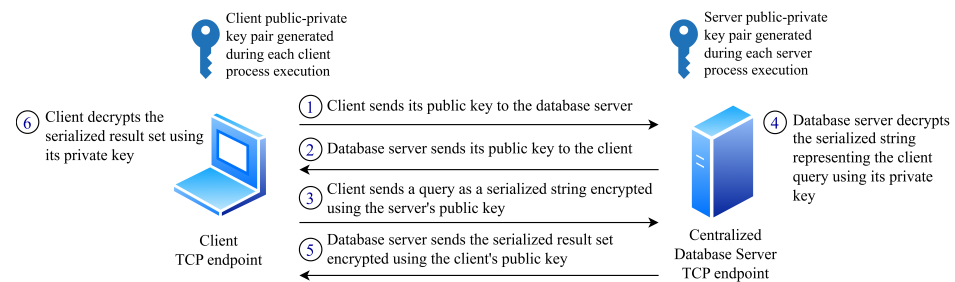


Figure 3. Principles of the encrypted communication between the client and database server.

A case study provides a benchmark platform for three asymmetric keys algorithms used in database domain: RSA, ElGamal, and ECIES [27]. A satisfying average time for decryption of strings with 30 characters was obtained using RSA with a 1024-bit key size.

Types of exchanged serialized messages include query strings and the returned result sets.

The following functions are involved in the client-server communication:

- **s**: send message;
- **r**: receive message;
- **l**: load the unserialized bytestream message;
- **d**: dump the serialized message as a bytestream;
- **enc**: encrypt message using either the client or server public key;
- **dec**: decrypt message using either the client or server private key;
- **pubClientKey**, **pubServerKey**: client/server public key;
- **privClientKey**, **privServerKey**: client/server private key.

The involved communication operations are denoted as follows:

- Sending an encrypted serialized message m from client to server:
 $s(\text{enc}(d(m), \text{pubServerKey}))$
- Sending an encrypted serialized message m from server to client:
 $s(\text{enc}(d(m), \text{pubClientKey}))$
- Loading and decrypting the bytestream b corresponding to the serialized message m from client to server:
 $l(\text{dec}(b, \text{privServerKey}))$
- Loading and decrypting the bytestream b corresponding to the serialized message m from server to client:
 $l(\text{dec}(b, \text{privClientKey}))$
- Loading and decrypting the bytestream b corresponding to the received serialized message m from client to server:
 $l(\text{dec}(r(b), \text{privServerKey}))$
- Loading and decrypting the bytestream b corresponding to the received serialized message m from server to client:
 $l(\text{dec}(r(b), \text{privClientKey}))$

Sending or receiving messages longer than 1024 bits over TCP/IPv4 sockets requires splitting the payload. In the proof-of-concept set-oriented DBMS implementation, the TCP payload is logically delimited by two frames, referred to as START and STOP.

Loading and decrypting a bytestream b , and sending an encrypted serialized message m , are symmetric operations.

4.2.2. Query Language Design Principles

The set-oriented NoSQL DBMS model's commands support CRUD operations. Syntactically, a query is defined in the same way as a dictionary's structure, presenting attributes,

each associated with a value. The processing of one command consists of a three-stage pipeline: parsing, compiling, executing, and fetching the result set. Queries have an imposed maximum depth, each with a specific set of valid, case insensitive attributes.

For example, creating a collection named mycollection made of one set denoted x is defined as follows:

```
CREATE {
  COLLECTION: {
    NAME: {mycollection};
  };
  ATTR: {
    SET: {x};
  };
};
```

The preceding command involves first level attributes—COLLECTION, ATTR—and second level attributes—NAME, SET.

Fetching the returned selected data from collection mycollection satisfying a condition marked by a WHERE clause entails a query wherein an alias must be set:

```
SELECT {
  ATTR: {
    COLLECTION: {mycollection};
    SET: {x};
    AS: {xalias};
    WHERE: {xalias > 10};
  };
};
```

The stages of the pipeline that handle the processing of commands in the proposed DBMS model involve the inherited methods of the main class, as presented in code Listing 3.

Listing 3. Stages of the command processing pipeline in the database engine.

```
1  while True:
2      try:
3          user_cmd = self.receive_user_cmd()
4          if not user_cmd: break
5
6          parsed_user_cmd = self.parse_user_cmd(user_cmd)
7          compiled_user_cmd = self.compile_user_cmd(parsed_user_cmd)
8          self.send_server_response(self.execute_user_cmd(
9              compiled_user_cmd, userObjects))
10         except Exception as e:
11             return('Error processing the client command: {}'.format(e))
```

Attribute value pairs in a command can optionally be separated from one another using the ; character. Reliable and consistent transactions generally follow ACID rules. The attributes can be chosen in any order since commands are serialized dictionaries. The syntactic rules for the CREATE, SELECT, UPDATE, and DELETE commands, expressed in Backus-Naur form, are provided in Appendix A.

The parser's scope is signified by the syntactic analysis, symbolizing a deterministic finite automaton. A user must ensure that the command is properly formatted before sending it to the database server. The DBMS server's workload can be simplified if the

parsing task is delegated to the client. In the context of the proposed system's query language, compilation and thus semantic analysis entails checking the correct correspondence between a command and its allowed set of attributes.

4.3. Evaluation Metrics

Performance metrics for the proposed set-oriented DBMS model focus on query response time and the size of the committed binary datafile. The execution of a large number of queries indicates the stability of the database system and can further serve as a method for determining the optimal workload of the component [28]. A natural technique for evaluating the query response time of a DBMS implementation in Python3 is to use the `time` method from the `time` module [29]. More accurate results can be obtained using a packet analyzer, which measures the elapsed time from when the user sends the command to when the database responds with the fetched result set.

Additional tests, as shown in Figures 4 and 5, demonstrate the temporal and spatial performance of selecting and committing data from a set in the proposed DBMS. Each of the 2000 entries is a randomly generated UUID string representation consisting of exactly 5 characters. The final size of the serialized binary datafile wherein the committed set data is permanently saved is 16KB. The graphs indicate that the execution time of the `SELECT` command on the queried set is linearly correlated with the number of inserted entries, as well as the size of the binary datafile where the committed data is permanently saved. The serialization of the data is performed using Python's `pickle` module, which efficiently converts the data into a binary format for storage [18]. The complete data used for generating the two plots is provided in [30].

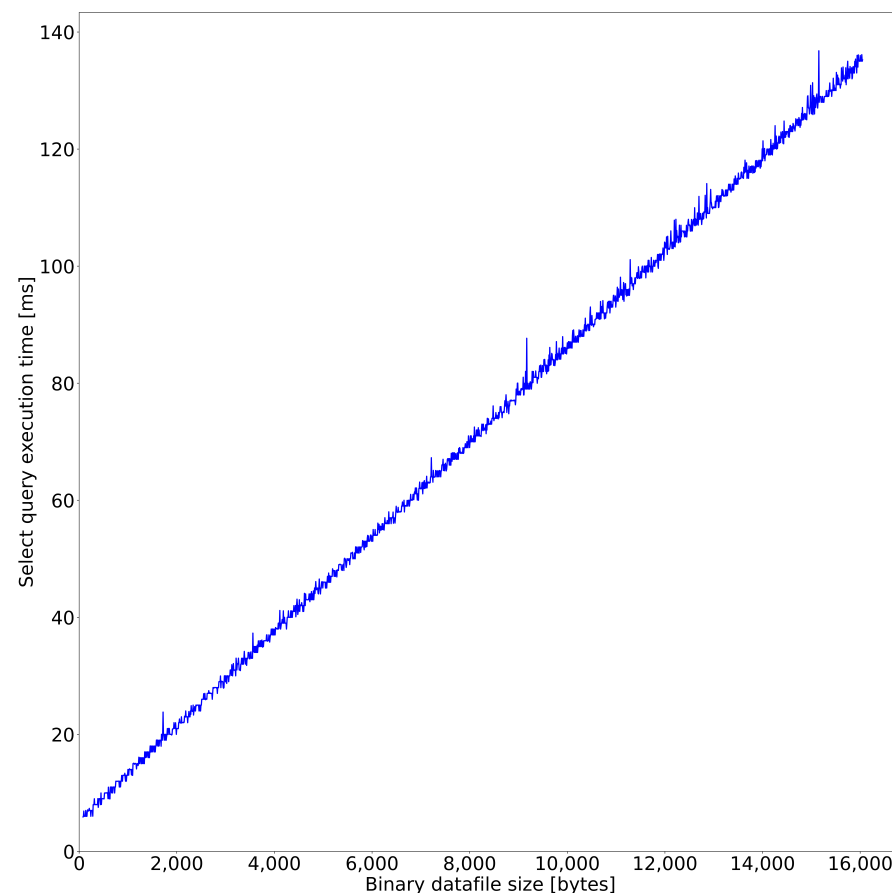


Figure 4. Execution time of the `SELECT` command on a queried set with an increasing number of entries.

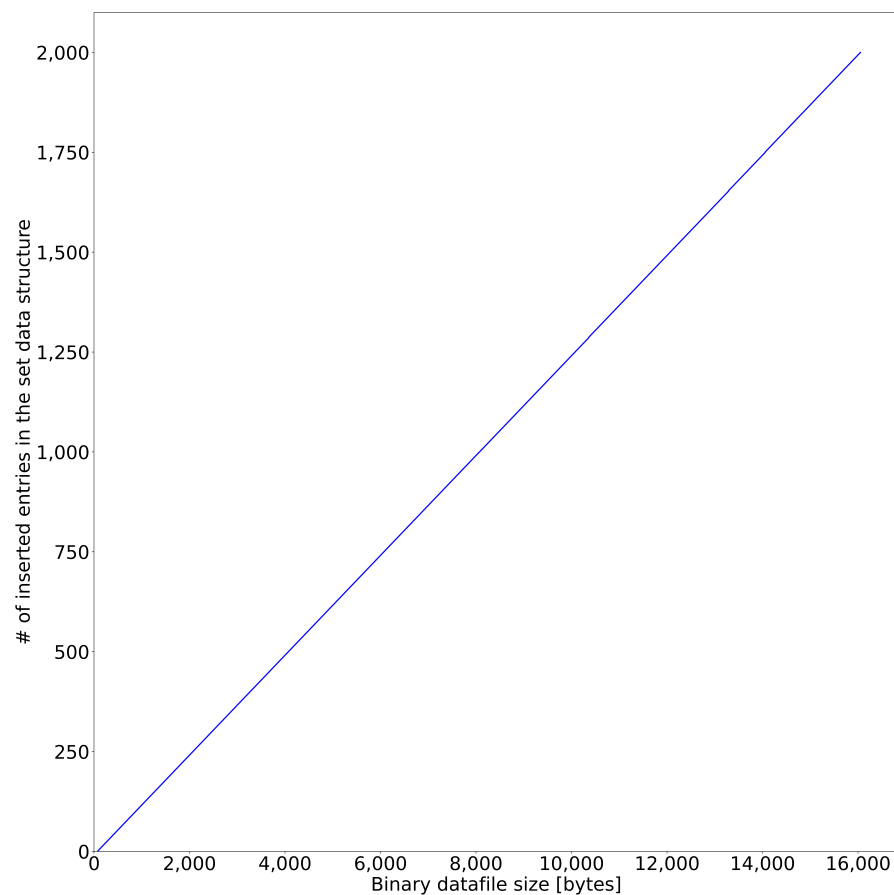


Figure 5. Correlation between the number of entries in the set and the size of the committed binary datafile.

5. Results and Further Improvement

A key performance metric of the proposed set-oriented DBMS is the query response time expressed in milliseconds. The evaluation tests implicate client and server hosts, both with 32GB of RAM and a 2.20 GHz Intel(R) Core(TM) i9-14900HX x64-based processor on multi-threaded Windows 11 Pro operating system.

The results obtained from running 16 commands in order are listed in Table 1.

The executed tests were performed by a single client connected to the server, which ran the exact set of ordered DBMS operations in 1000 batches during a single session. The query response time column denotes the average results after the trials for three NoSQL engines and one SQL DBMS: the proposed set-oriented system, Redis, MongoDB, and Oracle Database 21c Enterprise Edition.

The workload represents a realistic session combining creation, insertion, selection with predicates, updates, and deletion. For these tests, the client and server processes were executed on the same host, so that the query response primarily reflects the processing time. Redis is purely an in-memory database, which does not explicitly support data commit, whereas MongoDB ensures atomicity for CRUD operations at the document level.

The empty string value present in the sixth command within Oracle RDBMS is interpreted as NULL. The collection MY_COLLECTION is represented as a one-column table with the UNIQUE constraint applied.

The results indicate the strict set-related semantics of the proposed NoSQL DBMS model. The query response time is higher than that of Redis and MongoDB, primarily due to the additional overhead introduced by the parsing, compiling, and executing stages of the command processing pipeline. Nonetheless, the proposed system offers strong

consistency guarantees through explicit commit operations, which is a trade-off for the increased latency. The exhibited SELECT response time complexity for the committed set is $\mathcal{O}(n)$, where n represents the size or the number of inserted entries.

Table 1. Benchmark for evaluating 16 DBMS CRUD operations and one data commit.

Command Text	Query Response Time [ms]			
	Proposed DBMS	Redis 7.0	MongoDB 8.0	Oracle DBMS 21c
create { collection: { name: {my_collection}; }; attr: { set: {x};};};	9.1987	0.0849	4.8206	13.6306
insert { attr: { value: 'string12 34# '}; collection: { my_collection }; set: {x};};};	9.1232	0.0720	0.2620	1.4671
insert { attr: { value: {-16}; collection: {my_collection}; set: {x};};};	9.1965	0.0602	0.2882	1.0991
insert { attr: { value: {''}; collection: {my_collection}; set: {x};};};	9.0709	0.0532	0.2288	0.9792
select { attr: { set: {x}; collection: {my_collection}; as:{x};};};	9.3135	0.0492	0.0050	3.1364
delete { attr: { where: {x == ''}; as: {x}; set: {x}; collection: {my_collection};};};	9.5891	0.0523	0.2286	3.0212
update { attr: { where: {x == 'string12 34#'}; value: {1234}; as: {x}; set: {x}; collection: {my_collection};};};	9.4857	0.0505	0.1985	1.5817
update { attr: { where: {x == -16}; value: {16}; as: {x}; set: {x}; collection: {my_collection};};};	9.4697	0.0502	0.1823	3.1619
select { attr: { set: {x}; collection: {my_collection}; as: {x}; where: {x > 100};};};	9.4842	0.0980	0.2065	2.6276
delete { attr: { as: {x}; set: {x}; collection: {my_collection};};};	9.2470	0.0502	0.2624	1.2402
create { set: { name: {x};};};	7.9932	0	4.7694	9.2077
insert { attr: { value: {10}; set: {x};};};	8.0337	0.0502	0.2346	1.3914
insert { attr: { value: {20}; set: {x};};};	8.0067	0.0476	0.3254	0.9595
select { attr: { set: {x}; as: {x}; where: {x > 10};};};	9.4766	0.0553	0.2624	4.5545
drop { collection: { name: {my_collection};};};	7.9428	0.0458	0.3260	23.2416
commit {};	8.4013	-	-	0.6323

The formula for the query response time is the sum of components:

$$T_{\text{query response time}} = T_{\text{query parsing}} + T_{\text{send query to server}} + T_{\text{compile query}} + T_{\text{execute query and fetch results}} + T_{\text{receive query from server}} \quad (2)$$

The round-trip time (RTT) between the client and server, measured via the TCP/IPv4 sockets, consists of:

$$T_{\text{send query to server}} + T_{\text{receive query from server}} \quad (3)$$

If the client and server programs run on the same machine, the localhost interface is addressed, and the RTT is negligible.

The processing time of the DBMS query is given by:

$$T_{\text{query parsing}} + T_{\text{compile query}} + T_{\text{receive query from server}} \quad (4)$$

For each session, the server process fetches the most recently saved information from the binary datafile and loads it into its address space, thus establishing an in-memory database. Consequently, the uncommitted temporary state of the user's objects is stored in RAM. As a result, the time required to access the datafile does not directly impact query response performance.

5.1. Concurrency Analysis and Resource Contention

In the case of multiple simultaneous clients, the centralized DBMS architecture uses thread-pool-based request handling, with the maximum number of concurrent connections limited by the available CPU cores. The stability of the system is contingent upon its ability to efficiently manage the resources associated with multiple client connections within an adequate time frame.

The binary datafile is read at the start of the session, and it is written by the server process upon user commitment. While the proposed set-oriented model optimizes the use of the in-memory database, proper locking mechanisms must be implemented when the user requests read or write operations on secondary storage. A shared lock allows multiple threads to concurrently read a datafile while restricting write access. Conversely, a mutual lock prevents concurrent access by multiple threads, ensuring that only a single thread can read or write to the datafile at a time.

Table 2 illustrates the behavior of the proposed set-oriented DBMS model with varying numbers of connected clients: 8, 16, and 32. The evaluation is based on 10 batches of 18 commands, executed concurrently by multiple connections for a single user.

Table 2. Performance evaluation of a set-oriented DBMS model for concurrent workloads.

No. of Connected Clients	Total Execution Time [s]	Throughput [Commands/s]
8	3.3013	436.1918
16	7.4329	387.4665
32	15.9023	362.2117

In addition to the 16 commands listed in Table 1, the collection MY_COLLECTION is dropped, followed by the execution of another COMMIT operation for the concurrent workloads. The total number of processed commands is $B \times C \times \text{\#connections}$, where B is the total count of batches, C is the number of commands per batch, and $\text{\#connections}$ denotes the number of active clients. The evaluation is conducted for $B = 10$, $C = 18$, and $\text{\#connections}$ in 8, 16, and 32.

Throughput refers to the number of commands processed concurrently by the database server per second. Let $T_{\text{\#connections}}$ denote the real execution time for the system with $\text{\#connections}$ active clients. In this paper, database throughput is defined as follows:

$$\text{Throughput} = \frac{B \times C \times \text{\#connections}}{T_{\text{\#connections}}} \quad (5)$$

The average query response time for the 16 commands executed sequentially over 1000 batches is 8.9395 ms. In comparison, the same single-threaded workload for $B = 10$ batches, each consisting of 16 commands, yields an average execution time of 1.4303 s.

The multi-threaded architecture specifications included the use of RAM to store the temporary state of database objects and the implementation of locking mechanisms to control access to the binary datafile whenever it is accessed during a connection. To achieve better response times, the DBMS designer should aim to minimize the frequency of datafile reads and writes. In the proposed set-oriented system, interactions with secondary storage occur only at the beginning of the session and when the user commits.

5.2. Comparative Analysis of SQL Performance

One of the motivating factors for the proposed set-oriented NoSQL system construction arises from the discrepancies between the theoretical foundations of the relational model and its practical implementations in SQL-based DBMSs. An analysis of key system parameters during a user's Oracle Database 21c Enterprise session is provided. The queried schema view is `V$SYSSTAT`, and the name of the invoked statistics are available in `V$STATNAME` [31]. The used metrics present Oracle's capabilities, revealing memory, I/O, and CPU usage.

The statistics serve the following purposes:

1. Memory oriented:

- `session uga memory`—tracks the total amount of memory allocated in the user global area (UGA) for the session;
- `session pga memory`—tracks the total amount of memory allocated in the process global area (PGA) for the session;
- `db block gets`—the number of logical reads of database blocks from the buffer cache performed during the session;

2. I/O oriented:

- `physical reads`—the number of physical disk/secondary storage reads performed during the session;
- `physical writes`—the number of physical disk/secondary storage writes performed during the session;

3. CPU oriented:

- `CPU used by this session`—the total CPU time used by the session, measured in microseconds.

A synthetic dataset of 2000 entries was generated by running the script `sql-benchmark.py` [30]. The testing scenario's workload involved defining a table called `FACULTY`, which contains three columns: `FIRST_NAME`, `LAST_NAME`, and `UNIVERSITY`. Integrity constraints were enforced on the columns to ensure that all entries are unique, with `UNIVERSITY` serving as a unique key that can accept null values with a probability of `null_probability`, which is defined as a parameter in the script. For proper session-level analysis, the initial system parameter values—before running the queries—were also included.

The table `FACULTY` presents the subsequent structure:

```
FACULTY {
  FIRST_NAME VARCHAR2(5),
  LAST_NAME  VARCHAR2(5),
  UNIVERSITY VARCHAR2(5) NULL,
  CONSTRAINT PK_FACULTY PRIMARY KEY (FIRST_NAME, LAST_NAME),
  CONSTRAINT UNQ_UNIVERSITY UNIQUE (UNIVERSITY)
}
```

In total, session-level metrics were collected to assess the execution of 8 SQL commands: 6 SELECT queries, which involved operations such as collation, NULL values, UNION and UNION ALL statements, the DISTINCT keyword, and a functional index search; and 2 MERGE commands, one for a tuple not yet inserted and another for an existing record.

The results in Table 3 present, in the first row, the system specifications for the user's session prior to executing the SELECT and MERGE commands.

Table 3. System metrics of the Oracle DBMS during a session with an assigned workload.

SQL Command	CPU	Memory		I/O		
	CPU Used by This Session [μs]	Session UGA Memory [Bytes]	Session PGA Memory [Bytes]	DB Block Gets	Physical Reads	Physical Writes
Initial	41,163	180,375,592	340,809,896	13,439,240	43,247	81,731
CMD1	+8	+130,960	+393,216	0	0	0
CMD2	+8	−24	+196,608	0	0	0
CMD3	+11	0	−458,752	0	0	0
CMD4	+8	0	+655,360	0	0	0
CMD5	+17	+655,456	0	0	0	0
CMD6	+14	+655,456	+655,360	+10	0	0
CMD7	+12	0	0	+1	0	0
CMD8	+14	+130,912	+262,144	+655	0	+4

The values for the six metrics in the subsequent rows represent the variation relative to the immediately preceding configuration. The corresponding SQL ORACLE command texts, denoted as CMD1 to CMD8, are provided in Appendix B.

The UNIVERSITY column is constrained to hold unique values, which are either random UUID strings of precisely five characters in length or NULL values. CMD2 performs a collation operation that replaces NULL values with a predefined string value. In CMD5, the DISTINCT keyword is used, which ensures that the retrieved data from the table is treated as a set of unique tuples, eliminating any duplicate rows. Commands 3, 4, and 8 execute SELECT queries that incorporate a predicate on the UNIVERSITY field. This predicate evaluates whether the value in the UNIVERSITY column begins with the letter 'A' or 'B'. The MERGE commands 6 and 7 are structurally identical; however, the difference in their execution lies in the operation performed on the UNIVERSITY field. Command 6 updates an existing value in the UNIVERSITY column, while command 7 inserts a new value into the same column.

The invariance in the physical reads column during a session in which the COMMIT operation does not occur is noteworthy. Disk/secondary storage writes occur exclusively during CMD8, which involves a functionally indexed search. These aspects highlight the efforts of relational systems to store data in main memory and the application of complex heuristics to achieve progressively shorter response times.

5.3. Extension for Improved Handling of Larger Datasets

The system can present architectural enhancements to better handle larger quantities of data. The simplified version of the DBMS model currently supports a single serialized binary datafile per user, which may degrade performance as the number of entries increases. One potential solution to this issue is the creation of a control file to locate the queried data. Each time a size limit is reached, the server process saves an additional binary datafile to secondary storage.

Section 4.3 shows a linear relationship between the count of inserted records in the set stored in the serialized dictionary D_X for user X 's schema—as defined in Section 4.1—and

the observed response time. Based on the presented metrics, a cut-off can be established for the maximum number of entries allowed in a single datafile. When the size reaches around 10.5 KB, more than 1300 entries were inserted in the set, leading to a query response time of approximately 90 ms for the SELECT command.

The control file is a binary file that stores a serialized dictionary, with the keys representing the names of the user's objects, and the corresponding values indicating the locations of the datafiles where the information is permanently stored.

The modified data retrieval system organization is illustrated in Figure 6.

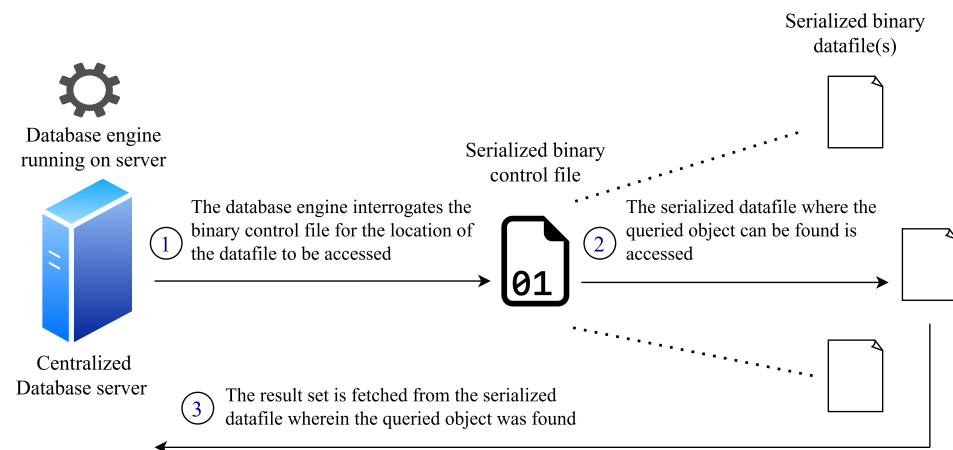


Figure 6. Big data NoSQL environment wherein multiple datafiles are created.

The binary control file stores a serialized dictionary mapping the names of the database objects to the location of the associated datafile from where the information can be accessed.

A leading principle of the proposed DBMS model in this paper is the storing of the uncommitted and temporary user's objects state in the server's address space. In the case of big data systems, fully pursuing an in-memory database implementation becomes a more difficult task. The query response evaluation could also add the time to access the control file, and then the pointed datafile.

The control file-centric extension provides organizational clarity over ensuring reduced latency.

The system supports creating and performing of various related operations on sets whose members are known and inserted/deleted by users. In a newer design iteration of the DBMS models, general sets can be defined, whose members follow a predicate rule. One implementation principle for evaluating the predicate rules involves lambda functions.

6. Future Research on Integrating the Proposed DBMS with Fuzzy Sets

Fuzzy sets are suitable for symbolizing vague or unclear information [32]. The resulting methodology, called Computing with Words, is expressive [33], introducing concepts involving linguistic variables and granularity. The intersection of fuzzy set theory and relational databases led to the creation of the SQLf query language [34], which is an extension of SQL.

Extension for Fuzzy Queries

A fundamental fuzzy set-oriented DBMS model implementation concept involves storing the membership function data in the serialized binary datafile, alongside the universe of discourse. The paradigm sways from the defined set structures to pairs of elements with their associated degree in the $[0, 1]$ interval.

Formally, if U symbolizes the universe of discourse and $\mu_{\tilde{S}}(e) \in [0, 1]$ the associated membership function, a fuzzy set is denoted as follows:

$$\tilde{S} = \{(e, \mu_{\tilde{S}}(e)) : e \in U, \mu_{\tilde{S}}(e) > 0\} \quad (6)$$

The extension scope is represented by applying the modelling specifications of the proposed set-oriented NoSQL systems to fuzzy data. Sets, in general, can be viewed as constrained fuzzy structures which adhere to a strict rule of belonging. Implicitly, the inserted values within such systems inherently hold a membership degree equal to 1. In this regard, set-oriented databases are a subset of fuzzy set-oriented databases. The proposed DBMS model deals with precise data defined by users. In practice, human applications or real life situations entail vague information, and related database queries should fetch the result sets accordingly.

The proposed construction principles for the fuzzy store suppose storing the imprecise information in binary datafiles, each representing a serialized dictionary D_i , whose abstract structure is defined in the Section 4.1. Within this framework, query complexity is decreased by returning the appropriate result set based on the details of the provided membership function. A related research direction for fuzzy database implementation explores the use of associative arrays [35]. The core tenet of fuzzy stores is to minimize the access time for the data based on the degree of membership specified in the query.

To illustrate the methodology for extending the principles described in this paper, only one object type is defined in D_i , namely the fuzzy set, labeled as t_1 . The set of fuzzy commands is $\mathcal{C}_{\mathcal{F}}$. The specifications of a command $c \in \mathcal{C}_{\mathcal{F}}$ include the tuple $(o_{(t_1)_j}, P, \mu_{o_{(t_1)_j}})$, where $o_{(t_1)_j}$ is a fuzzy object instance of $\tilde{S}$ and P an optional predicate. A suitable representation of the dictionary D_i has its keys consisting of the membership degrees for the elements associated with the universe of discourse.

The differences in approaches between the proposed set-centric DBMS model and the fuzzy set store are shown in Table 4.

Table 4. Comparison of system suitability based on use cases. A checkmark (✓) indicates suitability; an X indicates non-suitability.

Use Case	Proposed Set-Oriented DBMS Model	Extension for Fuzzy Sets
Clear, uniquely defined, and non-ambiguous information	✓	X
Ambiguous information	X	✓
Duplicate information	X	X
Transactionally consistent information	✓	✓

Fuzzy sets have applicability in the management of complex modern data [36], presenting an improved query result time and manipulation of the result sets [37,38].

7. Discussion

The response time expresses 2 components: command processing and client-server communication. Performing basic CRUD operations involving hashable set data structures implicates $\mathcal{O}(1)$ time complexity. Range queries and filtering operations with general predicates exhibit $\mathcal{O}(n)$ complexity, where n is the cardinality of the queried set. If the client and server programs are running on different hosts, the encrypted information is transferred, thus network latency becomes a factor.

Server overload is dependent upon the number of connected clients. Each user is handled by a thread within a Thread Pool. If there are more connections than available CPUs to the server machine, then performance bottlenecks can appear. For the proposed set-oriented DBMS model, the maximum number of accepted threads is limited.

The focal point for the evaluation tests of the proof of concept database engine symbolizing the proposed model was marked by the commands' average processing time trials. In order to obtain reliable results, network latency and overload of the server's resources had to become neglectable influences. Accordingly, the testing environment was configured as follows:

- both the client and server running on the same network host;
- a singular client being connected to the server at the time of the commands' execution.

In the case of serialized NoSQL systems, the trade-off between the access time and storage size of the binary datafiles marks another important metric. System availability and uptime can be increased if multiple servers would handle client connections. The principles underlined in this paper for the centralized DBMS are simply replicated one by one.

The results highlight the semantic implications of the proposed model. The system ensures by design that no duplicates are stored or fetched, removing the need for additional mechanisms. These results indicate that the proposed model trades modest performance overhead for stronger set-oriented semantic guarantees.

In addition, Redis does not explicitly provide some fundamental transactional features, and the defined set of commands is not as expressive as the one in the proposed model. Even though it presents important architectural differences, MongoDB implements a system of weak consistency guarantees as did Redis.

8. Conclusions

This paper introduces a formal model for a set-oriented NoSQL DBMS architecture, designed to address the fundamental challenge of enforcing data uniqueness at the architectural level. Information systems which manage modern and complex data should present the best traits of both SQL and NoSQL engines.

Through comparative performance analysis, it has been demonstrated that the set-oriented model trades latency for consistency guarantees, with the proposed system operating at approximately 8–9 ms per operation, compared to sub-millisecond response times in existing systems. This trade-off is well-suited for applications where uniqueness and data integrity enforcements are non-negotiable requirements.

The importance of properly designing a database system which manages modern and complex data must be underlined. Classic and renowned query languages such as SQL involve bottlenecks, indicating inadaptability to certain situations from one dialect to another. Deduplicated entries within a relation prove to be problematic for proper big data analysis, whereas sets are naturally fitting in the context of various modern applications.

The extended application of set-theoretic principles to fuzzy databases represents a natural progression toward the better management of imprecise data.

Set-oriented DBMS models constitute a distinct architectural paradigm within the constellation of NoSQL engines, one that prioritizes semantic clarity and consistency guarantees over raw latency optimization. In the future, distributed architectures in the context of big data should be explored, as well as an investigation of the fuzzy stores design trade-offs.

Author Contributions: Conceptualization, A.-G.S.; methodology, A.-G.S.; software, A.-G.S.; validation, A.B.; formal analysis, A.B. and A.-G.S.; data curation, A.-G.S.; writing—original draft preparation, A.-G.S.; writing—review and editing, A.B. and A.-G.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The dataset is openly available at <https://doi.org/10.5281/zenodo.18079195> under the CC BY 4.0 license (accessed on 28 December 2025).

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CDC	Change Data Capture
CRUD	Create, Read, Update and Delete
DBMS	Database Management System
ETL	Extract, Transform, Load
IP	Internet Protocol
NoSQL	Not Only SQL
RDBMS	Relational Database Management System
RTT	Round-trip Time
SQL	Structured Query Language
SETL	Set Language
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UUID	Universally Unique Identifier

Appendix A

The fundamental syntactic rules for the proposed DBMS commands CREATE, SELECT, UPDATE and DELETE, marking CRUD operations, are presented in Table A1. The syntax is described in Backus-Naur form. The columns **Nonterminal symbols** (left-hand side) and **Production Rule** (right-hand side) are associated in the rule definition.

Table A1. Proposed set-oriented DBMS formal grammar rules for CRUD operations on sets and collections.

Nonterminal Symbols	Production Rule
<var>	::= [a-zA-Z_] [a-zA-Z0-9_]*
<name>	::= '<var>' 0 [-] [1-9] [0-9]* [.] [0-9]*
<expr>	::= <var> == <name> <var> != <name> <var> > <name> <var> >= <name> <var> < <name> <var> <= <name>
<set>	::= set : {<var>}[;]
<collection>	::= collection : { name : {<var>}[;]}[;]
<schema>	::= schema : {<var>}[;]
<as>	::= as : {<var>}[;]
<where>	::= where : {<expr>}[;]
<value>	::= value : {<name>}[;]

Table A1. *Cont.*

Nonterminal Symbols	Production Rule
<create>	::= create {<set>}[;] create { [{ <collection>, attr : {<set>}}][;];
<select>	::= select {attr : [{<set>, [collection : {<var>}], [schema], <as>, [where]]}[;];
<update>	::= update {attr : [{<set>, [collection : {<var>}], [schema], <as>, [where], [value]]}[;];
<delete>	::= delete {attr : [{<set>, [collection : {<var>}], [schema], <as>, [where]]}[;];

Appendix B

The SQL commands—SELECT and MERGE—executed during the ORACLE Database 21c Enterprise session are listed in Table A2. The function `insert_synthetic_data_with_nulls` in the script `sql-benchmark.py` includes a parameter called `null_probability`, which takes values between 0 and 1. This parameter determines the probability of inserting a NULL value into the UNIVERSITY column. The commands are executed sequentially, from CMD1 to CMD8. The UNIVERSITY column enforces a unique key constraint and stores random UUID string values.

Table A2. The corresponding labeled Oracle SQL commands during the session testing of the relational database system.

Label	Command Text
CMD1	SELECT * FROM FACULTY WHERE UNIVERSITY IS NULL;
CMD2	SELECT FIRST_NAME, LAST_NAME, COALESCE(UNIVERSITY, 'Unknown University') AS UNIVERSITY FROM FACULTY;
CMD3	SELECT FIRST_NAME, LAST_NAME, UNIVERSITY FROM FACULTY WHERE UNIVERSITY LIKE 'A%' UNION SELECT FIRST_NAME, LAST_NAME, UNIVERSITY FROM FACULTY WHERE UNIVERSITY LIKE 'B%';
CMD4	SELECT FIRST_NAME, LAST_NAME, UNIVERSITY FROM FACULTY WHERE UNIVERSITY LIKE 'A%' UNION ALL SELECT FIRST_NAME, LAST_NAME, UNIVERSITY FROM FACULTY WHERE UNIVERSITY LIKE 'B%';
CMD5	SELECT DISTINCT UNIVERSITY FROM FACULTY
CMD6	MERGE INTO FACULTY F USING (SELECT :1 AS FIRST_NAME, :2 AS LAST_NAME, :3 AS UNIVERSITY FROM DUAL) SRC ON (F.FIRST_NAME = SRC.FIRST_NAME AND F.LAST_NAME = SRC.LAST_NAME) WHEN MATCHED THEN UPDATE SET F.UNIVERSITY = SRC.UNIVERSITY WHEN NOT MATCHED THEN INSERT (FIRST_NAME, LAST_NAME, UNIVERSITY) VALUES (SRC.FIRST_NAME, SRC.LAST_NAME, SRC.UNIVERSITY)
CMD7	MERGE INTO FACULTY F USING (SELECT :1 AS FIRST_NAME, :2 AS LAST_NAME, :3 AS UNIVERSITY FROM DUAL) SRC ON (F.FIRST_NAME = SRC.FIRST_NAME AND F.LAST_NAME = SRC.LAST_NAME) WHEN MATCHED THEN UPDATE SET F.UNIVERSITY = SRC.UNIVERSITY WHEN NOT MATCHED THEN INSERT (FIRST_NAME, LAST_NAME, UNIVERSITY) VALUES (SRC.FIRST_NAME, SRC.LAST_NAME, SRC.UNIVERSITY)
CMD8	SELECT * FROM FACULTY WHERE UNIVERSITY LIKE 'A%';

References

- Gadepally, V.; Kepner, J. Big data dimensional analysis. In Proceedings of the 2014 IEEE High Performance Extreme Computing Conference, Waltham, MA, USA, 9–11 September 2014; pp. 1–6. [\[CrossRef\]](#)
- Chen, M.; Chen, W.; Cai, L. Testing of big data analytics systems by benchmark. In Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Västerås, Sweden, 9–13 April 2018. [\[CrossRef\]](#)

3. Ivanov, T.; Rabl, T.; Poess, M.; Queral, A.; Poelman, J.; Poggi, N.; Buell, J. Big Data Benchmark Compendium. In *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things, Proceedings of the 7th TPC Technology Conference, TPCTC 2015, Kohala Coast, HI, USA, 31 August–4 September 2015*; Springer: Cham, Switzerland, 2016. [CrossRef]
4. Cantone, D.; Omodeo, O.; Policriti, A. *Set Theory for Computing: From Decision Procedures to Declarative Programming with Sets*; Springer: New York, NY, USA, 2001. [CrossRef]
5. Schwartz, J.; Dewar, R.; Dubinsky, E.; Schonberg, E. *Programming with Sets: An Introduction to SETL*; Springer: New York, NY, USA, 1986. [CrossRef]
6. Date, C. *The Relational Model for Database Management Version 2—A Critical Analysis: Deconstructing RM/V2*; Technics Publications: Basking Ridge, NJ, USA, 2024. Available online: <https://www.isbnsearch.org/isbn/9781634624220> (accessed on 28 December 2025).
7. Codd, E. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* **1970**, *13*, 377–387. [CrossRef]
8. Ricciotti, W.; Cheney, J. A Formalization of SQL with Nulls. *J. Autom. Reason.* **2022**, *66*, 989–1030. [CrossRef] [PubMed]
9. Eessaar, E. Using Relational Databases in the Engineering Repository Systems. In *Proceedings of the Eighth International Conference on Enterprise Information Systems—DISI, Paphos, Cyprus, 23–27 May 2006*; pp. 30–37. [CrossRef]
10. Date, C.J.; Darwen, H. *Foundation for Future Database Systems: The Third Manifesto*, 2nd ed.; Addison-Wesley: Reading, MA, USA, 2000. Available online: <https://dl.acm.org/doi/abs/10.5555/556540> (accessed on 28 December 2025). [CrossRef]
11. Silberschatz, A.; Korth, H.F.; Sudarshan, S. *Database System Concepts*, 6th ed.; McGraw-Hill: New York, NY, USA, 2010. Available online: <https://isbnsearch.org/isbn/9780073523323> (accessed on 28 December 2025).
12. Garcia-Molina, H.; Ullman, J.D.; Widom, J. *Database Systems: The Complete Book*, 2nd ed.; Pearson Prentice Hall: Upper Saddle River, NJ, USA, 2008. Available online: <https://isbnsearch.org/isbn/9780131873254> (accessed on 28 December 2025).
13. Date, C.J. *An Introduction to Database Systems*, 8th ed.; Pearson Education: Upper Saddle River, NJ, USA, 2004. Available online: <https://isbnsearch.org/isbn/0321189566> (accessed on 28 December 2025).
14. Wrembel, R. Data Integration, Cleaning, and Deduplication: Research Versus Industrial Projects. In *Proceedings of the International Conference on Information Integration and Web, Virtual Event, 28–30 November 2022*; pp. 3–17. [CrossRef]
15. Xia, W.; Jiang, H.; Feng, D.; Douglass, F.; Shilane, P.; Hua, Y. A Comprehensive Study of the Past, Present, and Future of Data Deduplication. *Proc. IEEE* **2016**, *104*, 1681–1710. [CrossRef]
16. Azeroual, O.; Jha, M.; Nikiforova, A.; Sha, K.; Alsmirat, M.; Jha, S. A Record Linkage-Based Data Deduplication Framework with DataCleaner Extension. *Multimodal Technol. Interact.* **2022**, *6*, 27. [CrossRef]
17. Costa, G.; Cuzzocrea, A.; Manco, G.; Ortale, R. Data De-duplication: A Review. In *Learning Structure and Schemas from Documents*; Biba, M., Xhafa, F., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 385–412. [CrossRef]
18. Cheng, P.; Gunawi, H.S. *Storage Benchmarking with Deep Learning Workloads*; Technical Report; University of Chicago: Chicago, IL, USA, 2021. Available online: https://newtraell.cs.uchicago.edu/files/tr_authentic/TR-2021-01.pdf (accessed on 28 December 2025).
19. Muvva, S.M. Standardizing Open Table Formats for Big Data Analysis: Implications for Machine Learning and AI Applications. *J. Artif. Intell. Cloud Comput.* **2023**, *2*, 1–3. [CrossRef]
20. Ardeleanu, S. *Relational Database Programming: A Set-Oriented Approach*, 1st ed.; Apress: Bucharest, Romania, 2016. Available online: <https://isbnsearch.org/isbn/9781484220795> (accessed on 28 December 2025).
21. Machireddy, J.R. Research Data Quality Management and Performance Optimization for Enterprise-Scale ETL Pipelines in Modern Analytical Ecosystems. *J. Data Sci. Predict. Anal. Big Data Appl.* **2023**, *8*, 1–26.
22. Redis. Transactions. Available online: <https://redis.io/docs/latest/develop/using-commands/transactions/> (accessed on 28 December 2025).
23. Kvet, M. Identifying and Treating NULL Values in the Oracle Database—Performance Case Study. In *Proceedings of the 33rd Conference of Open Innovations Association (FRUCT), Helsinki, Finland, 24–26 May 2023*; pp. 161–168. [CrossRef]
24. Irmert, F.; Daum, M.; Wegener, K.M. Modularization of Database Management Systems. In *Proceedings of the 2008 EDBT Workshop on Software Engineering for Tailor-Made Data Management, SETMDM '08, Nantes, France, 25 March 2008*; pp. 40–44. [CrossRef]
25. Python Software Foundation. *socket—Low-level Networking Interface*. Available online: <https://docs.python.org/3/library/socket.html> (accessed on 2 December 2025).
26. Python Software Foundation. *pickle—Python Object Serialization*. Available online: <https://docs.python.org/3/library/pickle.html> (accessed on 27 December 2025).
27. Boicea, A.; Rădulescu, F.; Truică, C.; Costea, C. Database encryption using asymmetric keys: A case study. In *Proceedings of the 21st International Conference on Control Systems and Computer Science (CSCS), Bucharest, Romania, 29–31 May 2017*. [CrossRef]
28. Popeangă, D.; Mocanu, M.; Boicea, A.; Rădulescu, F.; Ciolofan, S. A Case Study On DBMS Stability Performance Evaluation. *UPB Sci. Bull. Ser. C* **2024**, *86*, 141–150.

29. Python Software Foundation. time—Time Access and Conversions. Available online: <https://docs.python.org/3/library/time.html> (accessed on 4 December 2025).
30. Zenodo. Performance Metrics for a Set-Oriented DBMS Model and System Parameters Analysis for Oracle. Available online: <https://zenodo.org/records/18079195> (accessed on 28 December 2025).
31. Oracle Corporation. Oracle Database Reference Release 26. Available online: <https://docs.oracle.com/en/database/oracle/oracle-database/26/refrn/V-SYSSTAT.html> (accessed on 28 December 2025).
32. Zadeh, L.A. Fuzzy Sets. *Inf. Control* **1965**, *8*, 338–353. [[CrossRef](#)]
33. Zadeh, L.A. Fuzzy Logic = Computing with Words. *IEEE Trans. Fuzzy Syst.* **1996**, *4*, 103–111. [[CrossRef](#)]
34. Bosc, P.; Pivert, O. SQLf: A relational database language for fuzzy querying. *IEEE Trans. Fuzzy Syst.* **1995**, *3*, 1–17. [[CrossRef](#)] [[PubMed](#)]
35. Min, K.; Jananthan, H.; Kepner, J. Fuzzy Relational Databases via Associative Arrays. In Proceedings of the IEEE MIT Undergraduate Research Technology Conference (URTC), Cambridge, MA, USA, 6–8 October 2023; pp. 1–5. [[CrossRef](#)]
36. Zongmin, M.; Li, Y. Data modeling and querying with fuzzy sets: A systematic survey. *Fuzzy Sets Syst.* **2022**, *445*, 147–183. [[CrossRef](#)]
37. Suharjito, S. Query Optimization Using Fuzzy Logic in Integrated. *Indones. J. Electr. Eng. Comput. Sci.* **2016**, *4*, 637–642. [[CrossRef](#)]
38. Sharma, P. Retrieval of Information Using Fuzzy Queries. *Int. J. Eng. Tech.* **2016**, *2*, 118–122.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.