KiteDB: A Secure and Modular NoSQL Database System

University of Engineering and Technology, Lahore

Academic Session: 2023-2027

Submitted By:

Mahad Saffi	2023-CS-59
Abdul Rehman	2023-CS-73

Supervised By:

Dr. Aatif Hussain

Course:

CSC-207: Advanced Database Management Systems

Department of Computer Science

University of Engineering and Technology, Lahore

Date: May 22, 2025

Contents

List of Figures

List of Tables

1 Introduction

1.1 Overview

KiteDB is a NoSQL database system built in Python, utilizing a JSON-based data model. It integrates a robust backend with a React-based User Management System for efficient data and user administration. Key components include:

- **KiteDB-NoSQL**: Stores JSON documents in collections, supporting complex queries, optional schemas, and secure storage.
- **User Management System**: Provides a graphical interface for user administration, integrated with the backend.

The system employs a multi-threaded TCP server for concurrency and an interactive CLI for operational control, supporting CRUD operations, indexing, transaction management, Access Control Lists (ACL), and AES encryption.

1.2 Objectives

KiteDB aims to:

1. Provide a flexible NoSQL database for efficient data management.
2. Offer intuitive CLI and GUI interfaces for administration.
3. Support JSON-based queries with operators (`$gt`, `$lt`, `$and`, `$or`).
4. Ensure security via AES-CBC encryption and ACL enforcement.
5. Enable concurrent client handling through multi-threading.
6. Serve as an educational tool for NoSQL database internals.

1.3 Scope

KiteDB is a file-based NoSQL system with encrypted local storage, suitable for small-to-medium applications. It operates on a single machine but is designed for future distributed architecture enhancements.

1.4 Target Audience

The system targets:

- Computer Science students studying NoSQL databases.
- Developers exploring database implementation.
- Instructors using KiteDB as a teaching tool.
- Administrators seeking lightweight database solutions.

1.5 Features

Key features are summarized in Table 1.

Table 1: KiteDB Features

Feature	Description
Database Management	Create, select, and delete databases and collections.
Storage Engine	Encrypted, chunked file storage for secure persistence.
CRUD Operations	Insert, retrieve, update, and delete document data.
Query Language	JSON-based queries with operators (`$eq`, `$ne`, `$gt`, etc.).
Indexing	B-Tree indexing for optimized query performance.
Transaction Support	Log-based commit and rollback for data integrity.
Interactive CLI	Authenticated console with permission validation.
Schema Validation	Optional validation for data consistency.
Security	AES-CBC encryption and inferred ACL for access control.
Multi-threading	Multi-threaded TCP server for concurrent operations.
User Management	React-based interface for user CRUD operations.

1.6 Technology Stack

The technology stack is detailed in Table 2.

Table 2: Technology Stack

Component	Description
Languages	Python 3.8+ (backend); JavaScript with React (frontend).
Libraries	PyCrypto (encryption); threading (concurrency); React, Tailwind CSS (frontend).
Development Tools	Visual Studio Code for debugging and workflows.
Storage	Local file system with chunked binary storage.
Operating Systems	Windows, adaptable to Linux and macOS.

1.7 Design Philosophy

KiteDB emphasizes modularity, security, and performance. JSON aligns with web standards, while encryption and multi-threading ensure secure and concurrent processing.

2 Architecture

KiteDB's architecture prioritizes flexibility, security, and concurrency through a modular NoSQL framework and multi-threaded server.

2.1 KiteDB-NoSQL Architecture

The system stores JSON documents with schema validation, leveraging a multi-threaded TCP server and encrypted storage.

2.1.1 Components

- **Server** (`server.py`): Uses `ThreadingTCPServer` for concurrent client connections.
- **Query Parser** (`query_parser.py`): Parses JSON-based commands.
- **Collection Manager** (`collection.py`): Handles thread-safe CRUD operations with schema validation.
- **Storage Engine** (`storage_engine.py`): Manages encrypted, chunked file storage.
- **Index Manager** (`index_manager.py`): Maintains B-Tree indexes for query optimization.
- **Transaction Manager** (`transaction.py`): Ensures atomicity via logging.
- **ACL**: Enforces permissions inferred from CLI commands.
- **Logger** (`logger.py`): Logs system events to files.
- **Database** (`database.py`): Coordinates system operations.

2.1.2 Data Flow

1. Client issues a command (e.g., `users.find{"age": {"$gt": 25}}`).
2. Server authenticates and verifies ACL permissions.
3. `QueryParser` extracts operation and parameters.
4. `Collection` executes operation, using indexing.
5. `StorageEngine` encrypts and persists data.
6. Results are returned in JSON format.

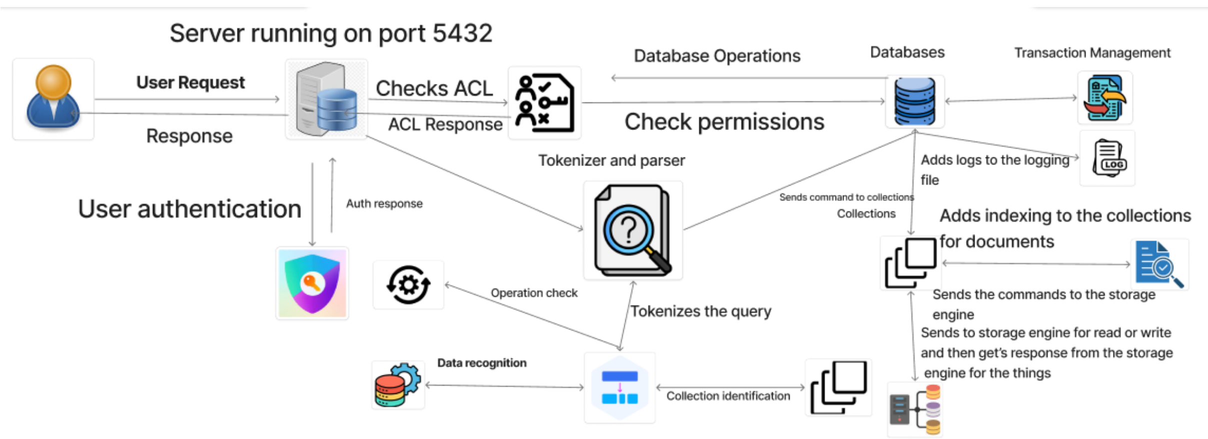


Figure 1: KiteDB Architectural Diagram

2.1.3 Design Decisions

- File-based storage for simplicity and portability.
- AES-CBC encryption for security-performance balance.
- B-Tree indexing for query efficiency.
- Multi-threading for concurrent operations.
- JSON queries for web compatibility.

3 Implementation Details

3.1 Startup Instructions

1. Open two terminals.
2. Run `python server.py` in `kitedb_nosql_json` to start the TCP server.
3. Run `python main.py` in a second terminal for the CLI.

3.2 Directory Structure

```
KiteDB/
 kitedb_nosql_json/
    src/
       core/
          collection.py
          database.py
          transaction.py
          exceptions.py
       query/
          query_parser.py
       storage/
          storage_engine.py
       index/
          index_manager.py
       logging/
          logger.py
       config.py
    server.py
    main.py
    config.yaml
```

3.3 Component Implementation

3.3.1 Storage Engine (`storage_engine.py`)

The `StorageEngine` class handles data persistence using encrypted, chunked files. It validates database names ($^[a-zA-Z0-9\_]+$ regex), creates storage directories, and uses AES-CBC encryption with a 16/24/32-byte key. Data is split into chunks (default: 500 documents) for memory efficiency. The `load()` and `save()` methods manage serialization and disk space checks.

Challenges: Thread-safe writes used `filelock`. Power failures were mitigated with journaling. **Optimizations**: In-memory chunk metadata and asynchronous I/O proposed. **Use Cases**: Secure storage for personal data or analytics.

```
def _encrypt(self, data: bytes) -> bytes:
    cipher = AES.new(self.key, AES.MODE_CBC)
    ct_bytes = cipher.encrypt(pad(data, AES.block_size))
    return cipher.iv + ct_bytes

def _decrypt(self, data: bytes) -> bytes:
    if len(data) < 16: raise StorageError("Invalid data")
    iv, ct = data[:16], data[16:]
    cipher = AES.new(self.key, AES.MODE_CBC, iv=iv)
    return unpad(cipher.decrypt(ct), AES.block_size)
```

3.3.2 Query Parser (`query_parser.py`)

The `QueryParser` interprets commands (e.g., `users.add{"name": "Alice"}`) using regex ($^{+\cdot} + *\{.*\}$). It supports `add`, `find`, `delete`, and `update` operations with operators (`$gt`, `$lt`, `$and`). Error handling addresses malformed JSON and invalid operators.

Challenges: Nested queries required recursive parsing. **Optimizations**: Cached query patterns. **Use Cases**: Dynamic data retrieval for e-commerce.

```
@staticmethod
def parse(command: str) -> Dict[str, Any]:
    if not command.strip(): raise ValidationError("Empty command")
    match = re.match(r'^(?P<collection>\w+)\.(?P<operation>\w+)\s*\{(?P<parameters>.*)\}$', command.strip())
    if not match: raise ValidationError("Invalid format")
    collection, op, payload = match.group('collection'), match.group('operation'), match.group('parameters').strip()
    if op == 'add': return {'operation': 'add', 'collection': collection, 'query': {}, 'data': [json.loads(payload)]}
```

3.3.3 Logger (`logger.py`)

The `Logger` singleton creates timestamped logs (e.g., `20250522_091200.log`) with configurable levels. It uses a rotating file handler for size management and asynchronous logging for performance.

Challenges: Thread safety via synchronized queues. **Optimizations**: Log compression. **Use Cases**: Auditing for financial systems.

```
def __new__(cls, log_dir: str, log_level: str):
    if cls._instance is None:
        cls._instance = super().__new__(cls)
        os.makedirs(log_dir, exist_ok=True)
        log_file = os.path.join(log_dir, f"{datetime.now():%Y%m%d_%H%M%S}.log")
        level = getattr(logging, log_level.upper(), logging.INFO)
        logging.basicConfig(filename=log_file, level=level, format="%(asctime)s [%(levelname)s] %(message)s")
        cls._instance.info("Logger initialized")
    return cls._instance
```

3.3.4 Index Manager (`index_manager.py`)

The `IndexManager` uses B-Tree indexing for query optimization, storing document IDs. It supports `add()`, `build()`, and `query()` with O(log n) complexity.

Challenges: In-memory limits (100,000 entries). **Optimizations**: Pre-sorted keys. **Use Cases**: Real-time analytics.

```
def add(self, field: str, value: Any, doc_id: int):
    if field not in self.index: self.index[field] = BTreeNode()
    node = self.index[field]
    if value not in node.keys: node.keys.append(value); node.values.append([doc_id]); self._size += 1
    if self._size > 100000: logger.warning(f"Index size for '{field}' exceeds 100,000")
```

3.3.5 Collection Manager (`collection.py`)

The `Collection` class manages thread-safe CRUD operations with schema validation. It supports transactions and optimizes queries with indexes.

Challenges: Concurrent inserts required fine-grained locking. **Optimizations**: Selective indexing. **Use Cases**: Content platforms.

```
def insert(self, docs: Any, apply_transaction: bool = False) -> Any:
    with self.lock:
        documents = docs if isinstance(docs, list) else [docs]
        for doc in documents: self.validate_schema(doc)
        if self.db.transaction and self.db.transaction.active and not apply_transaction:
            self.db.transaction.log({"collection": self.name, "action": "add", "params": [documents]})
            return "logged"
        docs_list = self.db.collections[self.name]
        start_id = len(docs_list)
```

```
        for i, doc in enumerate(documents): docs_list.append(doc)
            ; self.db.indexes[self.name].add_bulk(doc, start_id + i
            )
        self.db.save()
        return list(range(start_id, start_id + len(documents)))
```

3.3.6 Database (`database.py`)

The `Database` class coordinates collections, schemas, and storage with thread safety.

Challenges: Consistency with transactions. **Optimizations**: Caching. **Use Cases**: Multi-tenant databases.

```
def create_collection(self, name: str, schema: dict = None):
    if not re.match(r"^[a-zA-Z0-9_-]+$", name): raise KiteDBError
        ("Invalid name")
    if name in self.collections: raise KiteDBError("Exists")
    self.collections[name] = []; self.indexes[name] =
        IndexManager()
    if schema: self.schemas[name] = schema
    self.save()
```

3.3.7 Transaction Manager (`transaction.py`)

The `Transaction` class ensures atomicity with write-ahead logging and rollback support.

Challenges: Multi-threaded atomicity. **Optimizations**: Batching logs. **Use Cases**: Financial systems.

```
def commit(self) -> None:
    with self.lock:
        if not self.active: raise TransactionError("No
            transaction")
        try: [self.db.get_collection(op["collection"]).insert(op
            ["params"][0]) for op in self.ops if op["action"] == "
            add"]; self.active = False; self.ops.clear()
        except: self.rollback(); raise TransactionError("Failed")
```

3.3.8 Exceptions (`exceptions.py`)

Custom exceptions (`KiteDBError`, `StorageError`, etc.) ensure robust error handling with detailed logging.

3.4 Component Integration

The `Server` authenticates requests, `QueryParser` processes commands,

4 User Management System

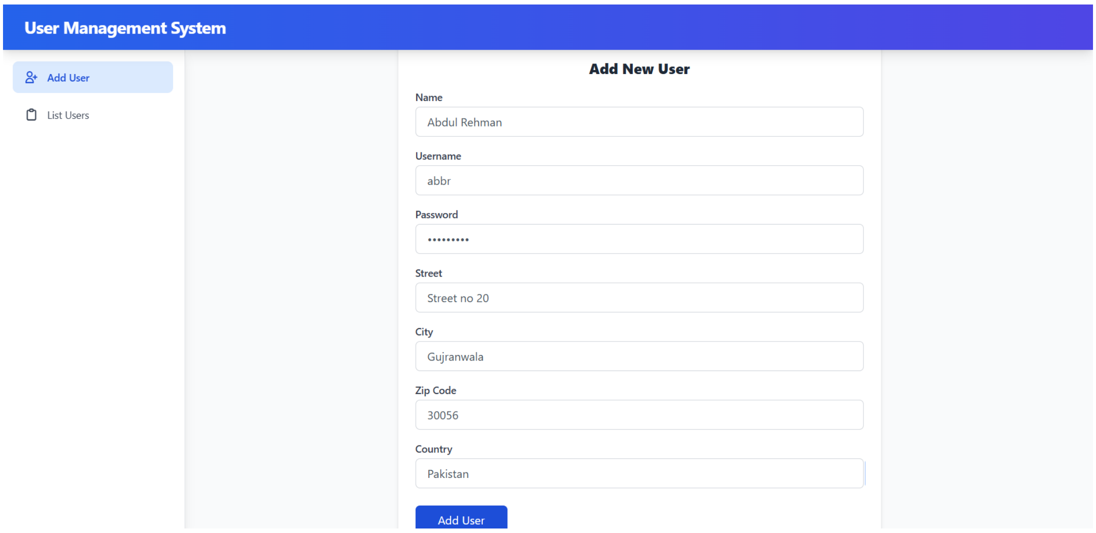
4.1 Overview

The React-based User Management System provides a graphical interface for user administration, integrated with KiteDB's backend.

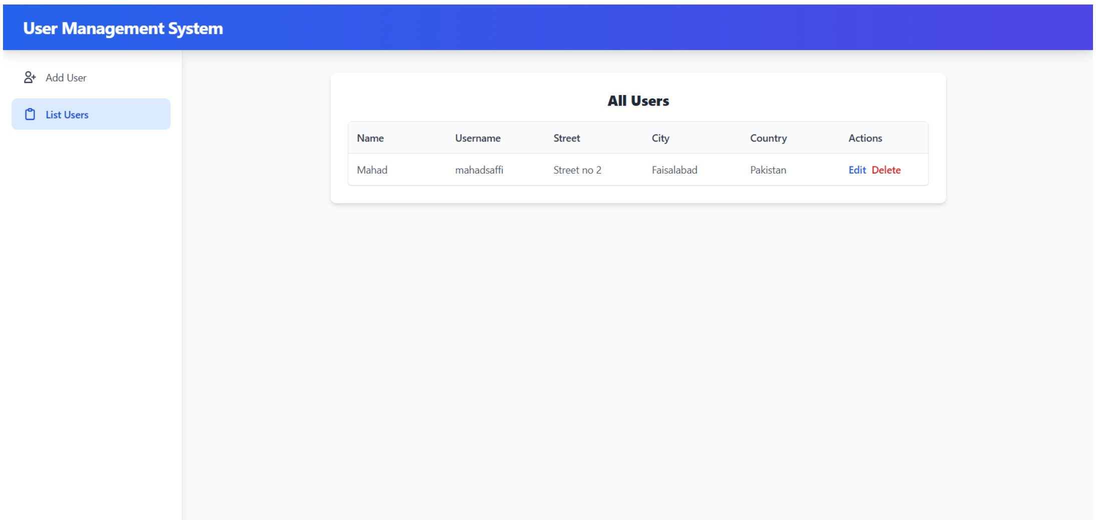
4.2 Features

- **Add User**: Form-based user creation.
- **List Users**: Table interface for viewing, editing, and deleting users.

4.3 Wireframes



(a) Add User Wireframe



(b) List Users Wireframe

Figure 2: User Interface Wireframes

4.4 Implementation

Built with React and Tailwind CSS, using `AddUser.js` and `UserList.js`, connected via a TCP server.

4.5 Design Considerations

Focuses on responsive design and secure data handling for a seamless user experience.

5 Usage and Examples

5.1 KiteDB-NoSQL Usage

1. **Launch**: Run `python server.py` and `python main.py`.
2. **Authenticate**: Use `admin/admin`.
3. **Commands**: Examples: `use <db>`, `create <coll> {<schema>}`, `<coll>.add{<doc>}`.

Example Session 1: CRUD Operations

```
kiteDB > login admin
Password: password123
kiteDB > use b
kiteDB (b) > create users {"fields": {"name": "str", "age": "int"}}
kiteDB (b) > users.add{"name": "Alice", "age": 28}
kiteDB (b) > users.find{"age": {"$gt": 25}}
kiteDB (b) > users.delete{"name": "Alice"}
```

Example Session 2: User Management

```
kiteDB > login admin
kiteDB > use b
kiteDB (b) > adduser testuser testpass
kiteDB (b) > setperm testuser b users read write
kiteDB > login testuser
kiteDB (b) > users.add{"name": "Bob", "age": 34}
```

5.2 User Management System Usage

1. **Launch**: Run `python server.py`, then `npm run dev` and `npm run server`.
2. **Add User**: Submit user details via the form.
3. **View/Edit/Delete**: Use the table interface for user management.

6 Test Cases

6.1 KiteDB-NoSQL Tests

- **Case 1**: `users.add{"name": "Alice", "age": 28}` succeeds; `users.add{"name": "Bob", "age": "30"}` fails (validation).
- **Case 2**: Concurrent `users.add{"name": "Bob"}` assigns unique IDs.
- **Case 3**: Transaction `commit` persists; `rollback` reverts.
- **Case 4**: `users.find{"$and": [{"age": {"$gt": 25}}, {"age": {"$lt": 30}}]}` returns `[{"name": "Alice", "age": 28}`.

6.2 User Management Tests

- **Case 1**: Form submission adds a user.
- **Case 2**: Delete button removes a user.

7 Repository

GitHub Repository: KiteDB

8 Conclusion

8.1 Summary

KiteDB is a secure, modular NoSQL database with a React-based user interface, supporting efficient data management and concurrency.

8.2 Challenges

Development addressed thread safety, frontend-backend integration, indexing scalability, and security-performance trade-offs.

8.3 Limitations

Current limitations include in-memory constraints, single-machine operation, basic indexing, and limited query capabilities.

8.4 Lessons Learned

Insights include concurrency management, secure design, modularity, and UI development.

9 Future Enhancements

- Row-level locking for finer concurrency.
- Client-server architecture for scalability.
- Advanced query support (e.g., joins).
- Performance optimizations (e.g., caching).

10 References

- Python Documentation: `https://www.python.org/doc/`.
- Silberschatz, A., et al., *Database System Concepts*, 7th Edition.
- MongoDB Documentation: `https://docs.mongodb.com/`.
- B-Tree: `https://en.wikipedia.org/wiki/B-tree`.
- AES: `https://https://en.wikipedia.org/wiki/Advanced_Encryption_Standard`.
- React Documentation: `https://reactjs.org/docs/`.
- Tailwind CSS: `https://tailwindcss.com/docs`.