Hash Based Database Management System

Submitted By:

Saher Riaz	2023-CS-170
Um-e-Kalsoom	2023-CS-182
Aleena Zafar	2023-CS-183
Noor-e-Hira Khan	2023-CS-184

Submitted To:

Dr. Aatif Hussain

Course:

CSC-207 Advanced Database Management System

Department of Computer Science

University of Engineering and Technology Lahore Pakistan

Table of Contents

1 Introduction 4
2 Key Concepts 4
3 Key Features 4
3.1 Transaction Management 4
3.2 Commit and Rollback Mechanism 5
3.3 Indexing 5
3.3.1 Responsibilities of the Indexer Class: 6
3.3.2 Types of Indexes Managed: 6
3.4 Constraints 6
3.5 Storing Data in JSON Documents: 7
3.6 Join Operations 7
3.7 Checkpoints 7
3.7.1 Purpose of Checkpoints: 7
3.7.2 Checkpoint Process: 7
3.8 Query Language 7
4 Basic Commands 8
4.1 Data Definition Language (DDL) 8
4.2 Data Manipulation Language (DML) 8
4.3 Query Language 8
4.4 Aggregation 8
4.5 Joins 9
5 Advanced Features 9
5.1 Transaction Management 9
5.2 Indexing Implementation 9
5.3 Constraint Enforcement 9
6 Architecture Overview 10
7 Key Components Table 11
8 Data Model 11
9 Query Lifecycle 12
10 Why Hash-Based DBMS 12
11 System Requirements 12

12 User Interface........13
13 Testing and Evaluation........16
14 Conclusion........16
15 References........17

1 INTRODUCTION

Database systems are essential in managing large-scale data in many applications. A key challenge is maintaining consistency, reliability, and performance under high transaction load. Traditional relational databases typically rely on disk-based storage and complex locking mechanisms. In contrast, this project focuses on developing a hash-based DBMS that implements efficient transaction management, including support for concurrency control, ACID properties, and checkpoints.

This system integrates a hash-based storage engine with a transaction manager, concurrency control, commit/rollback mechanisms, and a query processor. Checkpoints are used to periodically save the state of the database, making it possible to recover from crashes or failures. The system also supports a variety of SQL commands through a PyQt5-based graphical user interface, allowing users to interact with the database easily.

2 KEY CONCEPTS

Hash-Based Implementation

The database uses hash tables as its primary data structure for storing and retrieving records. Each table maintains its records in a dictionary (hash map) where keys are the primary key values and values are the complete records.

Key characteristics:

- O(1) average time complexity for insert, delete, and search operations
- Direct addressing via hash functions
- Collision handling through separate chaining

```
Table "students":
{
    "1": {"id": 1, "name": "Alice", "age": 20},
    "2": {"id": 2, "name": "Bob", "age": 21}
}
```

3 KEY FEATURES

3.1 TRANSACTION MANAGEMENT

The transaction manager is a critical component of the DBMS, ensuring the database remains in a consistent state even in the case of power outages, crashes, or unexpected failures. It supports the **ACID** properties:

- **Atomicity**: Ensures that each transaction is fully completed or fully rolled back.
- **Consistency**: Guarantees that the database transitions from one valid state to another.
- **Isolation**: Ensures that transactions do not interfere with each other.
- **Durability**: Ensures that completed transactions persist even after a failure.

Concurrency control mechanisms, such as locking and timestamp-based management, prevent data corruption and ensure that multiple transactions can execute simultaneously without violating database integrity.
The system supports ACID transactions with:

- BEGIN TRANSACTION
- COMMIT
- ROLLBACK

Example:

```
# Begin transaction
db.begin_transaction("tx1")

# Execute operations
db.create_table("students", ["id int", "name string"], {"id": ["primary_key"]}, "tx1")
db.insert("students", "1", ["1", "Alice"], "tx1")
```

3.2 COMMIT AND ROLLBACK MECHANISM

The commit and rollback mechanisms are essential for ensuring that transactions are properly completed or undone.

- **Commit**: When a transaction successfully completes all its operations, it is committed to the database, making its changes permanent.
- **Rollback**: If a transaction encounters an error or is explicitly rolled back, all changes made during the transaction are undone, restoring the database to its previous state.

These mechanisms help maintain consistency in the database, even in the event of a system failure or transaction error.

Example:

```
# Commit or rollback
db.commit_transaction("tx1")
# OR
db.rollback_transaction("tx1")
```

3.3 INDEXING

To improve data retrieval performance, the system implements indexing. Indexes are data structures that improve the speed of query processing by allowing the DBMS to locate data more efficiently..

The **Indexer Class** is responsible for managing the creation, maintenance, and utilization of indexes within the DBMS. It works closely with the transaction manager and query processor to ensure that indexed data can be quickly and efficiently retrieved.

3.3.1 Responsibilities of the Indexer Class:

- **Creating Indexes**: The Indexer class allows for the creation of both hash-based and composite indexes on one or more columns in a table. When an index is created, it stores the hashing or indexed values in memory for rapid access.
- **Index Maintenance**: The Indexer ensures that indexes are updated in real-time whenever data is modified (inserted, updated, or deleted). It ensures that changes in the data are reflected in the corresponding indexes to maintain query performance.
- **Efficient Query Execution**: By utilizing the appropriate index, the Indexer improves the performance of queries by reducing the need to perform full table scans. The system can look up records faster by referring to the indexed values, thus significantly improving SELECT query performance.
-

3.3.2 Types of Indexes Managed:

- **Hash-based Indexes**: These indexes use a hash function to map a column's values to a unique key, allowing for fast retrieval based on hashed keys.
- **Composite Indexes**: The Indexer can create composite indexes, where multiple columns are indexed together. This helps improve query performance when filtering on multiple attributes.

The **Indexer Class** is crucial for improving the overall performance and scalability of the DBMS by reducing query response times, particularly when dealing with large datasets.
The Indexer class provides:

- CREATE INDEX
- DROP INDEX
- Automatic index maintenance

Example:

```
# Create index
db.create_index("students", "name")
# Query using index
results = db.select_where("students", "name", "=", "Alice")

# Drop index
db.drop_index("students", "name")
```

3.4 CONSTRAINTS

The system enforces several constraints to ensure data integrity and consistency:

- **Primary Key**: Ensures that each record in the database has a unique identifier, preventing duplicate entries in the same table.
- **Unique Key**: Guarantees that the values in a specified column are unique across the entire table, allowing for NULL values.
- **Foreign Key**: Ensures referential integrity between tables by linking columns in one table to the primary key of another table.

3.5 STORING DATA IN JSON DOCUMENTS:

Data is stored in JSON documents, making it flexible and easily extensible. This approach supports hierarchical and semi-structured data, allowing users to store more complex data structures (e.g., lists, nested objects) within the database. JSON documents provide the added benefit of being easy to serialize and deserialize, enabling quick data exchange between the DBMS and the user interface.

Example:

```
db.create_table("students",
    ["id int", "name string", "class_id int"],
    {
        "id": ["primary_key"],
        "class_id": ["foreign_key", "classes.id"],
        "name": ["unique"]
    }
)
```

3.6 JOIN OPERATIONS

Supports INNER JOIN between tables:

```
results = db.inner_join("students", "classes",
                        "class_id", "id",
                        ["students.name", "classes.name"])
```

3.7 CHECKPOINTS

Checkpoints are a critical feature in the DBMS, providing a method for periodically saving the current state of the database to disk. The checkpointing mechanism plays an essential role in ensuring that the database can recover efficiently after a crash.

3.7.1 Purpose of Checkpoints:

Checkpoints create a snapshot of the database state at a specific point in time. This snapshot allows for fast recovery by reducing the need to reapply all transactions from the beginning after a failure.

3.7.2 Checkpoint Process:

At regular intervals, the DBMS creates a checkpoint that includes all data modifications made since the last checkpoint. Once a checkpoint is created, the system stores this information in stable storage, ensuring that the system can roll back to a consistent state if a crash occurs.

By maintaining these periodic snapshots, the system optimizes recovery times and ensures durability.

3.8 QUERY LANGUAGE

Supported SQL-like operations:

- CREATE TABLE

- INSERT
- UPDATE
- DELETE
- SELECT
- DROP TABLE
- ALTER TABLE (DROP COLUMN)
- GROUP BY
- HAVING
- DISTINCT

4 Basic Commands

4.1 Data Definition Language (DDL)

CREATE TABLE

```
CREATE TABLE students (id INT, name TEXT, age INT) CONSTRAINTS (id PRIMARY_KEY)
```

DROP TABLE

```
DROP TABLE students
```

ALTER TABLE (DROP COLUMN)

```
ALTER TABLE students DROP COLUMN age
```

4.2 Data Manipulation Language (DML)

INSERT

```
INSERT INTO students VALUES (1, 'Alice', 21)
```

UPDATE

```
UPDATE students SET name = 'Bob' WHERE id = 1
```

DELETE

```
DELETE FROM students WHERE id = 1
```

4.3 Query Language

SELECT

```
SELECT * FROM students WHERE id = 1
```

SELECT with specific columns

```
SELECT name, age FROM students WHERE id = 1
```

SELECT with conditions

```
SELECT * FROM students WHERE age > 20
```

4.4 Aggregation

GROUP BY

```
SELECT age, COUNT(*) FROM students GROUP BY age
```

HAVING

```
SELECT age, COUNT(*) FROM students GROUP BY age HAVING COUNT(*) > 1
```

DISTINCT

```
SELECT DISTINCT age FROM students
```

4.5 JOINS

INNER JOIN

```
SELECT s.name, c.course FROM students s JOIN courses c ON s.id = c.student_id
```

5 ADVANCED FEATURES

5.1 TRANSACTION MANAGEMENT

The TransactionManager class provides:

- Transaction isolation
- Locking mechanisms (read/write locks)
- Deadlock prevention
- Atomic commit/rollback

Locking hierarchy:

1. Table-level locks for schema changes
2. Row-level locks for data operations

5.2 INDEXING IMPLEMENTATION

The Indexer class maintains:

- Hash-based indexes (value → list of keys)
- Automatic updates on data modification
- Support for range queries (>, <, >=, <=, <>)

Index structure:

```
{
    "students": {
        "name": {
            "Alice": ["1", "4"],
            "Bob": ["2", "3"]
        }
    }
}
```

5.3 CONSTRAINT ENFORCEMENT

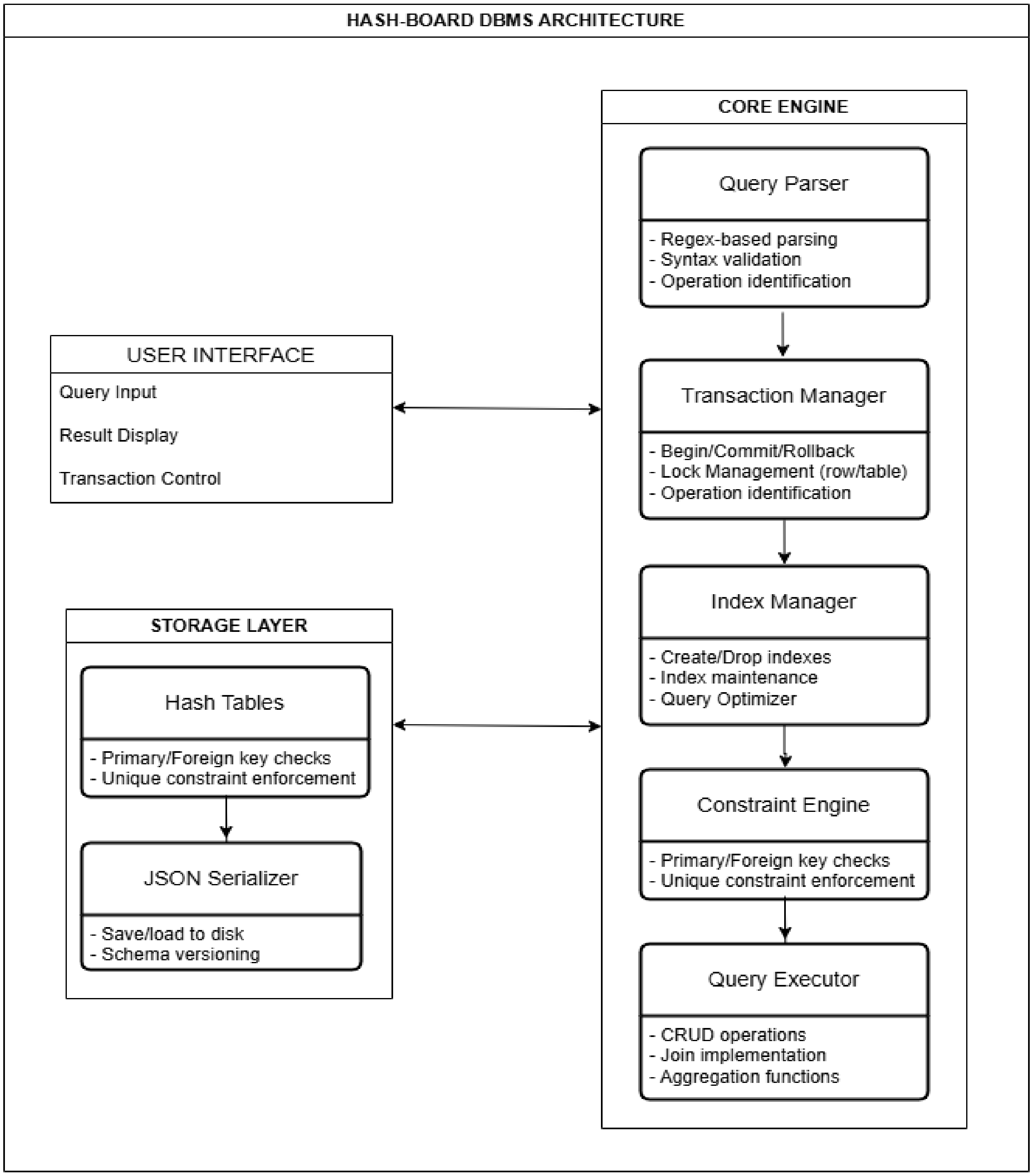
Constraints are checked during:

- INSERT operations
- UPDATE operations
- DELETE operations (for referential integrity)

Types:

1. **Primary Key**: Ensures uniqueness and non-null
2. **Foreign Key**: Maintains referential integrity
3. **Unique**: Ensures column values are unique

6 ARCHITECTURE OVERVIEW



7 KEY COMPONENTS TABLE

Component	Description	Key Methods
Database	Main database class	create_table, insert, update, delete, select
TransactionManager	Handles ACID transactions	begin_transaction, commit, rollback, acquire_lock
Indexer	Manages indexes for performance	create_index, drop_index, update_index
StorageSQLUI	Graphical user interface	execute_query, begin_transaction, create_index

8 DATA MODEL

The database uses a relational model with these characteristics:

- **Tables**: Collections of records
- **Records**: Key-value pairs (hash maps)
- **Columns**: Typed fields with constraints
- **Relationships**: Established through foreign keys

Example schema:

```
students (id PK, name, age, class_id FK → classes.id)
classes (id PK, name, teacher)
```

JSON storage format:

```
{
    "tables": {
        "students": {
            "columns": {
                "id": {"type": "int", "constraints": ["primary_key"]},
                "name": {"type": "string"},
                "age": {"type": "int"}
            },
            "records": {
                "1": {"id": 1, "name": "Alice", "age": 20}
            }
        }
    },
    "indexes": {
        "students": {
            "name": {
                "Alice": ["1"]
            }
        }
    }
}
```

9 QUERY LIFECYCLE

1. **Parsing**:
    - Query is parsed using regular expressions
    - Syntax validation occurs
2. **Planning**:
    - Appropriate execution path is selected
    - Index usage is determined
3. **Execution**:
    - Locks are acquired
    - Constraints are checked
    - Operations are performed
4. **Result Handling**:
    - Results are formatted
    - Locks are released
    - Transaction log is updated
5. **Output**:
    - Results are displayed to user
    - Status messages are shown

10 WHY HASH-BASED DBMS

Advantages of this implementation:

1. **Performance**: Constant-time O(1) operations for CRUD operations when using primary key
2. **Simplicity**: Straightforward implementation using Python dictionaries
3. **Flexibility**: Schema-less nature allows for easy modification of table structures
4. **Memory Efficiency**: Only stores actual data without empty spaces
5. **Scalability**: Handles growing datasets efficiently until hash collisions become frequent

11 SYSTEM REQUIREMENTS

- Operating System: Windows 7+, macOS 10.12+, or Linux
- Python 3.7 or later
- Required Python packages:
    - PyQt5
    - json
    - re
    - datetime
    - threading

Dependencies

```
        pip install PyQt5
```

12 USER INTERFACE

The DBMS is equipped with a user interface developed using PyQt5. This interface allows users to interact with the database through a graphical application, enabling them to:

- Execute SQL queries directly.
- View query results in a user-friendly format.
- Monitor transactions and database changes.
- Manage indexes and data operations.

The PyQt5 interface simplifies the process of managing and interacting with the database, making it accessible for non-technical users and providing a streamlined user experience.

Figure 1: Query Execution Interface

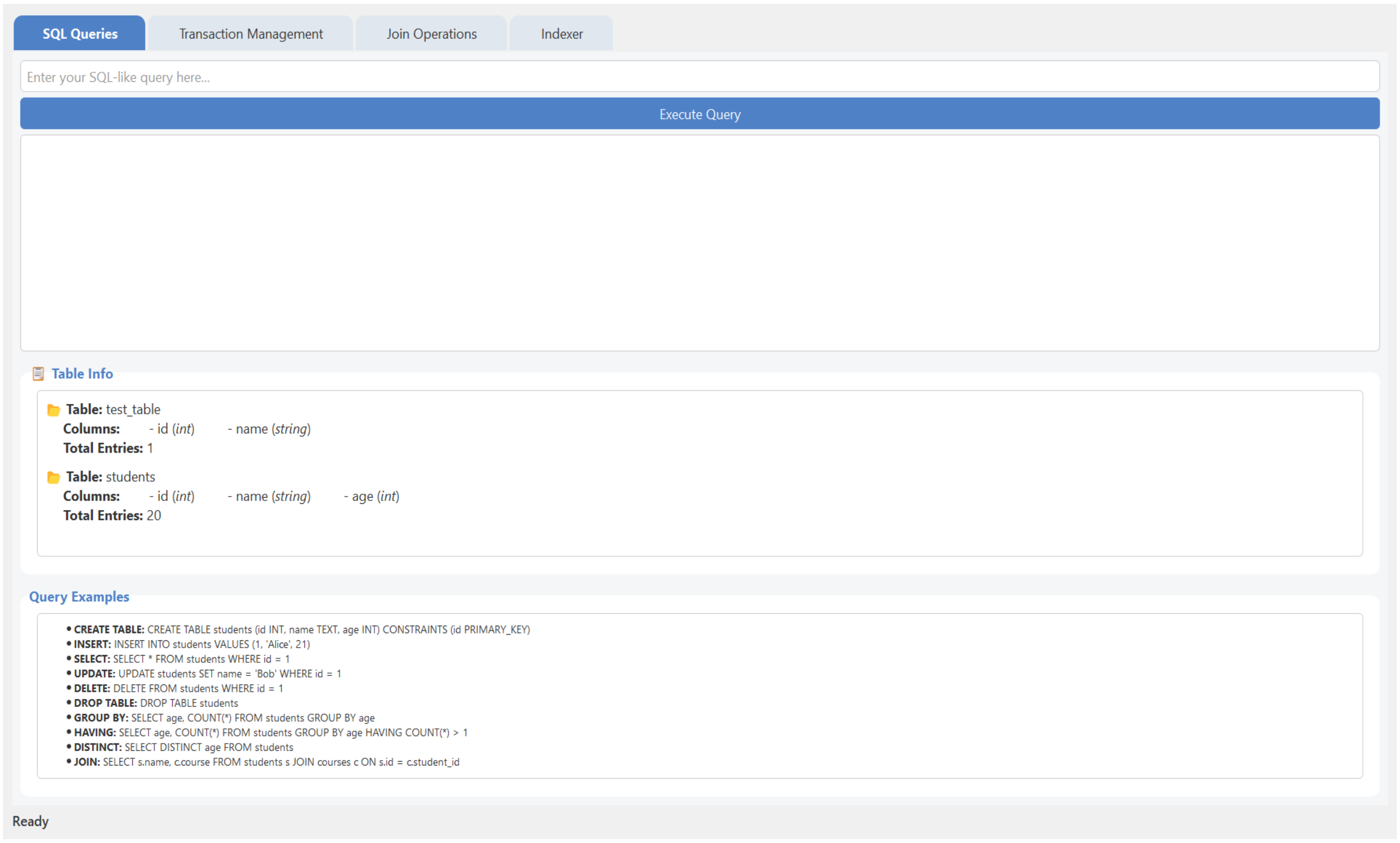


Table Info

Table: test_table
Columns: - id (*int*) - name (*string*)
Total Entries: 1

Table: students
Columns: - id (*int*) - name (*string*) - age (*int*)
Total Entries: 20

Query Examples

- **CREATE TABLE:** CREATE TABLE students (id INT, name TEXT, age INT) CONSTRAINTS (id PRIMARY_KEY)
- **INSERT:** INSERT INTO students VALUES (1, 'Alice', 21)
- **SELECT:** SELECT * FROM students WHERE id = 1
- **UPDATE:** UPDATE students SET name = 'Bob' WHERE id = 1
- **DELETE:** DELETE FROM students WHERE id = 1
- **DROP TABLE:** DROP TABLE students
- **GROUP BY:** SELECT age, COUNT(*) FROM students GROUP BY age
- **HAVING:** SELECT age, COUNT(*) FROM students GROUP BY age HAVING COUNT(*) > 1
- **DISTINCT:** SELECT DISTINCT age FROM students
- **JOIN:** SELECT s.name, c.course FROM students s JOIN courses c ON s.id = c.student_id

Ready

Figure 2: Transaction Management Interface

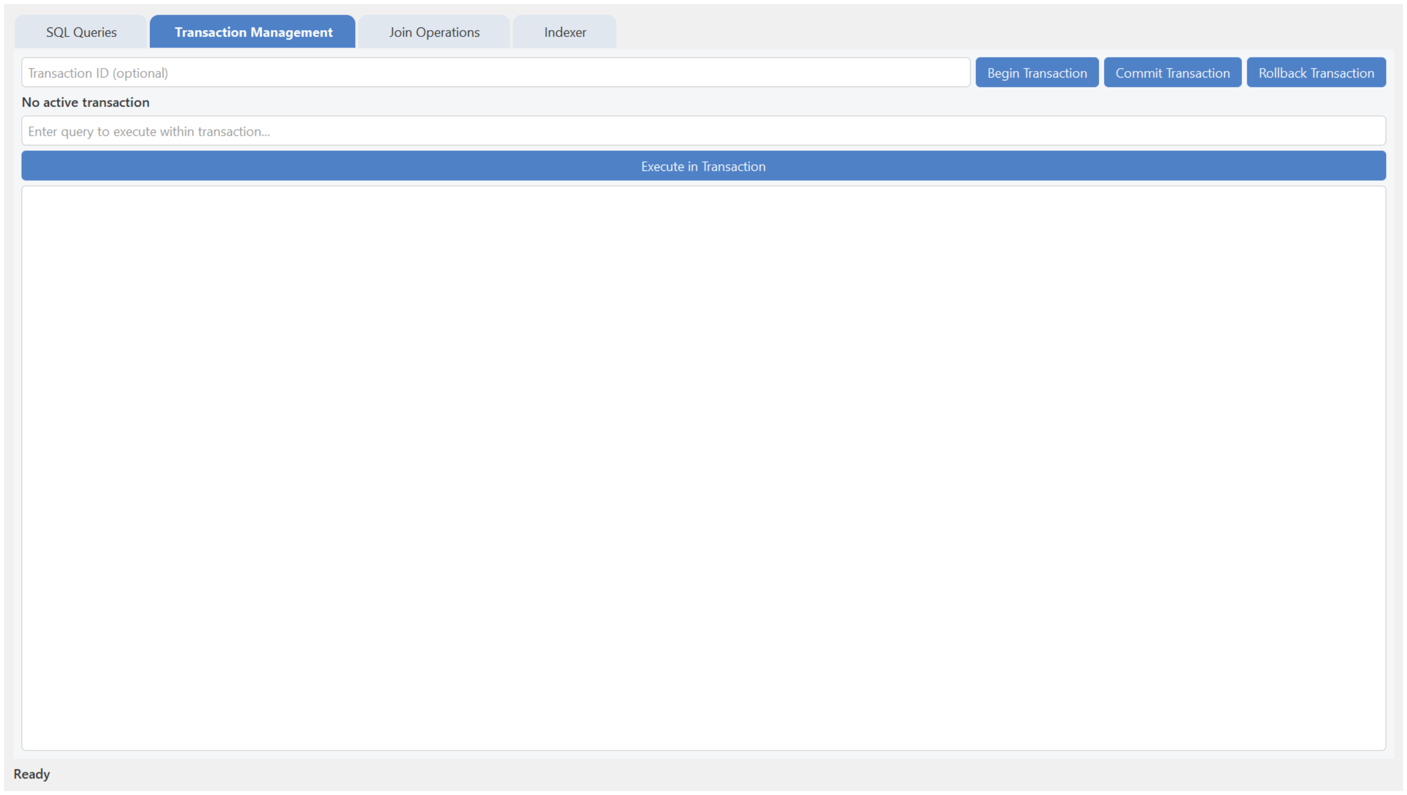


Figure 3: Join Operations Interface

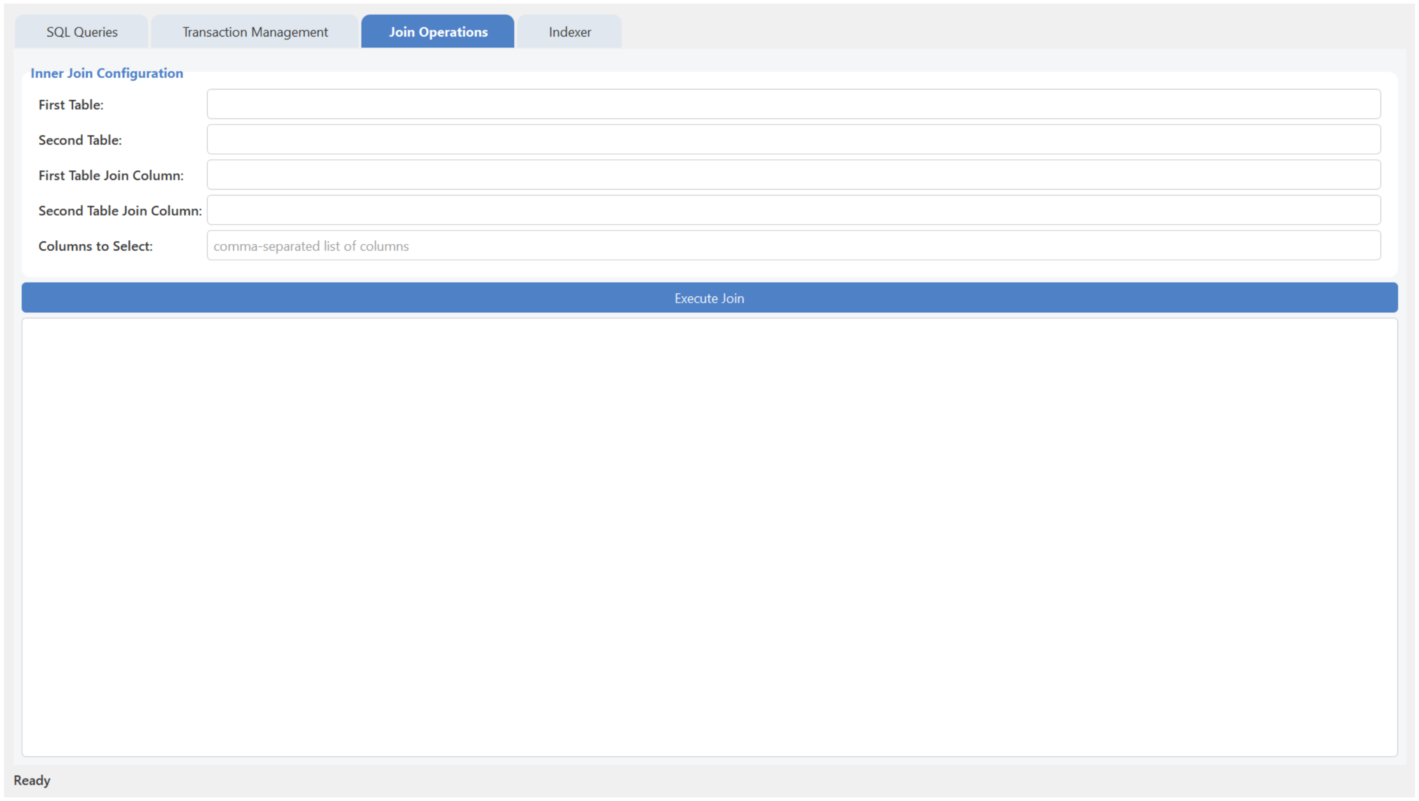
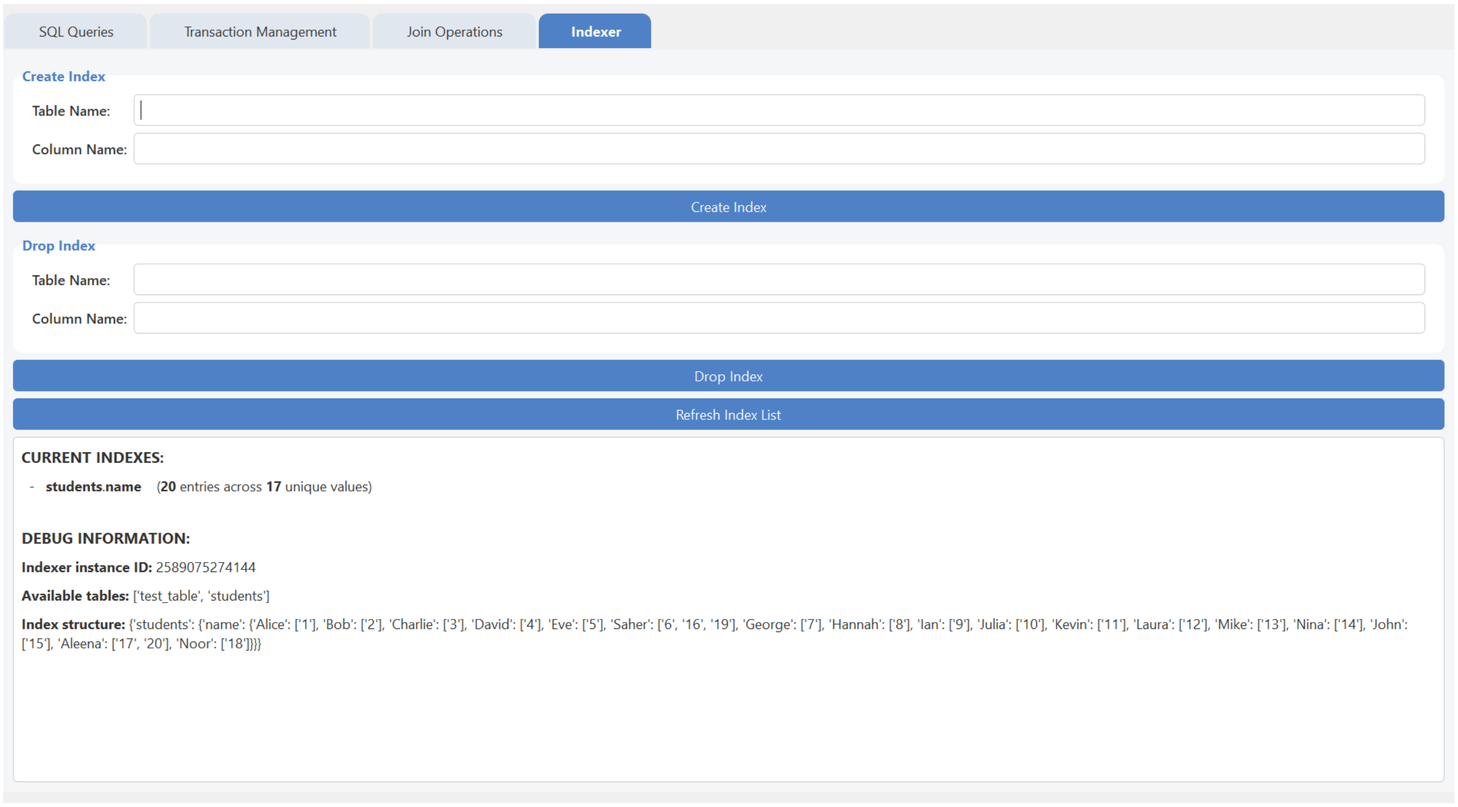


Figure 4: Indexer Interface



```
CURRENT INDEXES:
 -  students.name   (20 entries across 17 unique values)

DEBUG INFORMATION:
Indexer instance ID: 2589075274144
Available tables: ['test_table', 'students']
Index structure: {'students': {'name': {'Alice': ['1'], 'Bob': ['2'], 'Charlie': ['3'], 'David': ['4'], 'Eve': ['5'], 'Saher': ['6', '16', '19'], 'George': ['7'], 'Hannah': ['8'], 'Ian': ['9'], 'Julia': ['10'], 'Kevin': ['11'], 'Laura': ['12'], 'Mike': ['13'], 'Nina': ['14'], 'John':
['15'], 'Aleena': ['17', '20'], 'Noor': ['18']}}}
```

13 TESTING AND EVALUATION

Testing was performed to ensure the correctness, efficiency, and reliability of the DBMS. Key tests include:

- **Transaction Integrity Tests**: Ensuring commit and rollback operations work as expected under various failure scenarios.
- **Concurrency Tests**: Validating that multiple transactions can execute concurrently without causing data corruption.
- **Performance Tests**: Measuring query response times and the efficiency of indexing mechanisms.

The system passed all tests for data integrity, consistency, and performance.

14 CONCLUSION

This Hash-Based Database Management System provides a lightweight, efficient solution for applications requiring fast data access with transactional integrity. Its combination of hash-based storage, indexing,

and constraint enforcement makes it suitable for a variety of use cases while maintaining simplicity and performance.
The modular architecture allows for future expansion, and the graphical interface makes it accessible to users of varying technical skill levels. With the planned enhancements, the system has the potential to grow into a full-featured database solution.

15 REFERENCES

PyQt5 Documentation - https://www.riverbankcomputing.com/static/Docs/PyQt5/