Advanced Database Management System (ADBMS)

A2Z DATABASE

Project Report

Session: 2023

Submitted by:

Anza Tamveel (2023-CS-87)

Amna Atiq (2023-CS-83)

Zainab Batool (2023-CS-89/2024-CS-652)

Supervised by:

Dr. Atif

Submitted on: May 15, 2025

Course:

CSC-207L Advanced Database Management Systems

Department of Computer Science

University of Engineering and Technology

Lahore, Pakistan

TABLE OF CONTENTS

List of Tables

List of Figures

Introduction

The A2Z Database Management System (DBMS) is a lightweight, file-based database engine developed in Python. It supports JSON document storage, collection management, transaction handling, indexing, and aggregation operations. A custom query language in Punjabi like format simplifies user interactions, while a graphical user interface (GUI) built with Tkinter enhances usability for non-technical users. The project was motivated by the need for a simple, educational database system that demonstrates advanced database concepts like ACID properties and aggregation in a lightweight package. The system is designed for small-scale applications, focusing on ease of use and educational value.

Key Features

• **JSON-like Document Storage:** Stores data in a flexible JSON format for easy manipulation and retrieval.

• **Custom Query Language**: A user-friendly language in PUNJABI like format to perform database operations without complex syntax.

• **Collection Management:** Supports creation, deletion, and indexing of collections for organized data storage.

• **CRUD Operations:** Enables insert, update, delete, and find operations for comprehensive data management.

• **Aggregation:** Provides basic aggregation functionality, similar to MongoDB, for data analysis.

• **Transaction Support:** Implements begin, commit, and rollback operations to ensure data integrity.

• **Persistence:** Uses file-based storage with crash recovery for reliable data retention.

• **Logging:** Tracks all operations for durability and recovery using a per-database log file.

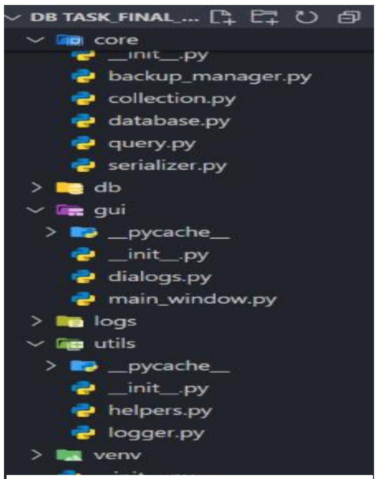
• **Graphical User Interface (GUI):** A Tkinter-based interface with a tabbed query editor, database/collection selectors, and light/dark theme support.

• **Help System:** Offers a built-in command reference to assist users in understanding available commands.

Project Structure

The project is organized into modular components for maintainability and scalability. Below is the directory structure:

```
project_root/
├── __init__.py
├── main.py              # Entry point to run the application in CLI mode
├── core/                # Core database logic and operations
|   ├── __init__.py
|   ├── database.py      # Manages database creation, transactions, and persistence
|   ├── collection.py    # Handles document operations and indexing
|   ├── query.py         # Parses and executes custom queries
|   ├── serializer.py    # Serializes data to JSON for storage
|   └── backup_manager.py  # Manages backup and restore operations
├── gui/                 # Graphical User Interface components
|   ├── __init__.py
|   ├── main_window.py   # Main GUI window with query editor and result display
|   └── dialogs.py       # Custom dialogs for user interactions
├── utils/               # Utility functions for supporting operations
|   ├── __init__.py
|   ├── helpers.py       # Validation, formatting, and helper functions
|   └── logger.py        # Logging functionality for durability and recovery
└── db/                  # Auto-generated directory for database files
```



```
DB TASK_FINAL_...
  core
      __init__.py
      backup_manager.py
      collection.py
      database.py
      query.py
      serializer.py
  > db
  gui
    > __pycache__
      __init__.py
      dialogs.py
      main_window.py
  > logs
  utils
    > __pycache__
      __init__.py
      helpers.py
      logger.py
  > venv
```

Figure 1. This figure shows database file structure

Core Components

core/database.py
Manages core database operations, transactions, and persistence.

• **Key Methods:**

- __init__(name, db_path): Initializes or loads a database from the specified path.
- create_collection(name, indexes=[]): Creates a new collection with optional indexes.
- get_collection(name): Retrieves a collection instance by name.
- drop_collection(name): Deletes a collection and its data.
- begin_transaction(), commit(), rollback(): Manages transactions with logging for durability.
- aggregate (collection_name, pipeline): Executes aggregation pipelines on a collection.

core/collection.py
Handles document operations and indexing within collections.

• **Key Features:**

- Indexing: Supports basic indexing for faster searches (e.g., on fields like "name").
- Operations: Insert, update, delete, and find operations with methods like insert_one() and find().
- In-memory Structure: Uses an in-memory structure for fast reads, with periodic flushing to disk.

core/query.py - Query Parsing
Parses and executes a custom query language that maps to database operations.

• **Supported Commands**:

- Transactions: begin tx, commit, rollback
- Database Management: create database <name>, drop database <name>, use database <name>
- Collection Management: performs operations like create collection, drop collection, create index
- CRUD Operations: performs operations such as insert, update, delete, find
- Aggregation: perform aggregate functions

core/serializer.py
Handles serialization of data to JSON format for storage and retrieval.

• **Key Role**: Ensures data is stored in a consistent JSON format, enabling persistence and recovery.

core/backup_manager.py

Manages backup and restore operations for databases.

• **Key Methods:**

- create_backup: Creates a backup of the specified database.
- restore_backup: Restores a database from a backup.

utils/logger.py - Logging

Logs all operations for durability and crash recovery.

• **Highlights:**

- Maintains a log file per database (e.g., db/students.log).
- Logs operations like insert, update, and transaction events.
- Supports rollback by replaying logs during recovery.

utils/helpers.py - Helpers

Provides utility functions for validation, formatting, and other supporting tasks.

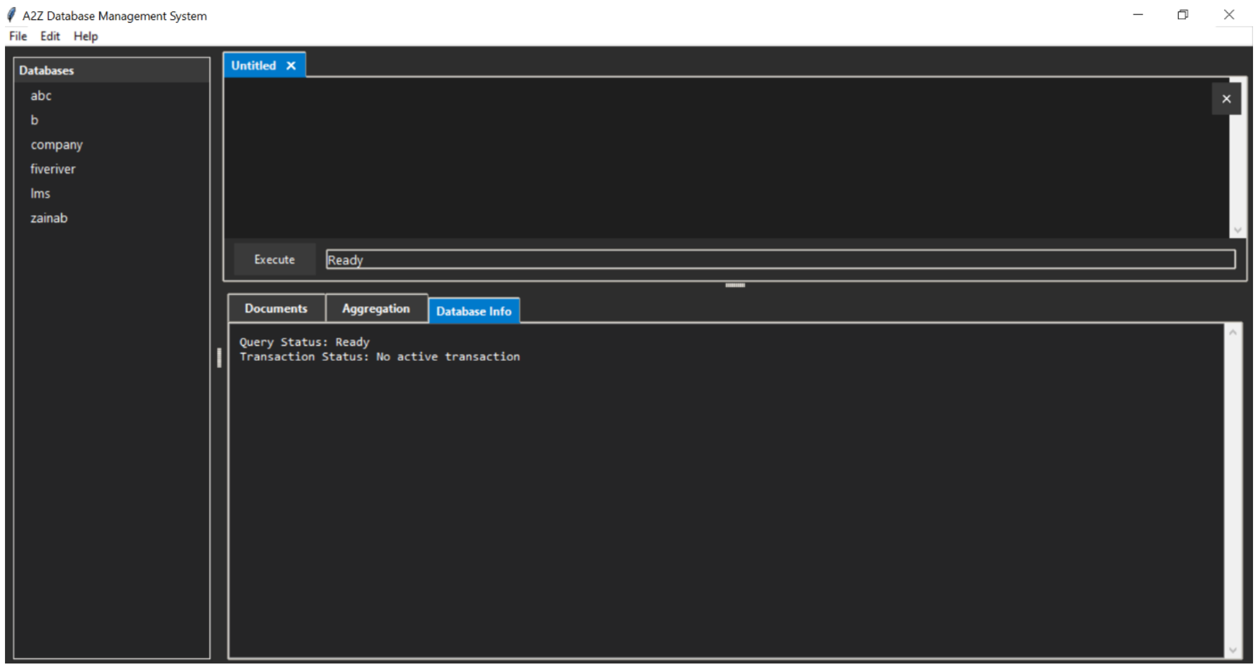
• Examples: Functions to validate JSON documents and format query results.

gui/main_window.py - Graphical User Interface

Provides an intuitive interface for interacting with the DBMS using Tkinter.

• **Key Features:**

- Tabbed Query Editor: Allows users to write and execute queries in multiple tabs.
- Database/Collection Selectors: A tree view to navigate databases and collections.
- Theme Support: Light/dark theme toggle for better user experience.
- Result Display: Shows query results, aggregation outputs, and transaction status in tabs (Documents, Aggregation, Database Info).
- Help System: A "View Commands" option under the Help menu to display all commands and their descriptions.
- Status Display: Shows transaction and query status in the "Database Info" tab.



```
Query Status: Ready
Transaction Status: No active transaction
```

Figure 2. This figure shows the GUI of the A2Z Database Management System

System Architecture

The A2Z DBMS is designed with a modular architecture to separate concerns and improve maintainability. The system consists of the following layers:

• **GUI Layer (gui/main_window.py):** Handles user interactions, query input, and result display using Tkinter.

• **Query Parsing Layer (core/query.py):** Parses user queries and maps them to database operations.

• **Core Logic Layer (core/database.py, core/collection.py):** Manages databases, collections, transactions, and document operations.

• **Persistence Layer (core/serializer.py, utils/logger.py):** Ensures data is stored on disk and recoverable using JSON serialization and logging.

• **Utility Layer (utils/helpers.py):** Provides supporting functions for validation and formatting.

ACID Properties

Atomicity

The system ensures atomicity through transaction support. For example, a transaction inserting multiple documents (e.g., insert many into marks [...]) will either complete fully or be rolled back if an error occurs, using the begin transaction, commit, and rollback methods.

Consistency

The system maintains consistency by validating operations before execution. For instance, an invalid query (e.g., inserting a malformed JSON document) is rejected, ensuring the database remains in a valid state after each transaction.

Isolation

Transactions are isolated within a single database. For example, a transaction in database "students" does not affect database "employees". However, the system does not support concurrent transactions within the same database, limiting isolation for multi-user scenarios.

Durability

Durability is achieved through logging (utils/logger.py). All operations are logged to a file (e.g., db/students.log), ensuring that the database can be recovered to a consistent state after a crash by replaying the logs.

Table 1. This table shows the implementation of ACID properties

Property	Implementation	Example
Atomicity	Transaction methods (begin, commit, rollback)	Insert multiple documents in one transaction
Consistency	Query validation and error handling	Rejects malformed JSON documents
Isolation	Single-database transaction isolation	Transactions in separate databases
Durability	Operation logging and recovery	Logs used to recover after a crash

Screenshots of Functionality

Here is basic diagram of our system:

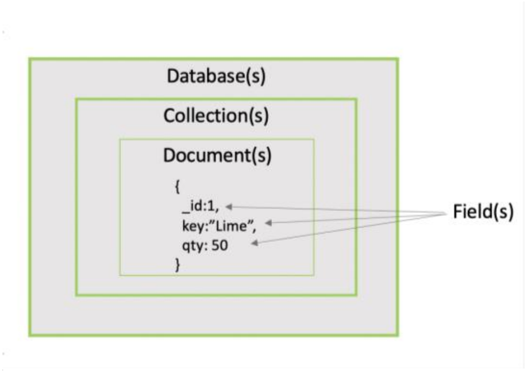
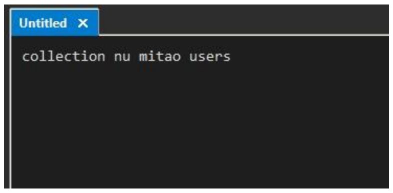


Figure 3. This figure shows the basic architecture of our system

The following images provide a visual overview of the A2Z DBMS:

GUI Overview: A screenshot of the main window in Figure 1 showing the tabbed query editor, database/collection tree, and result tabs.

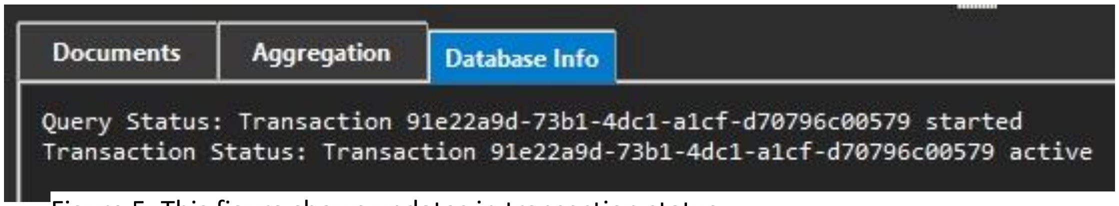
Query Insertion:



```
collection nu mitao users
```

Figure 4. This figure shows the basic query insertion command

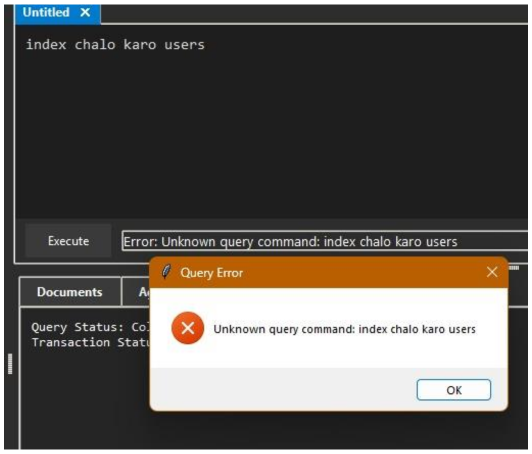
Transaction Status Display: A screenshot of the "Database Info" tab showing transaction status updates.



```
Query Status: Transaction 91e22a9d-73b1-4dc1-a1cf-d70796c00579 started
Transaction Status: Transaction 91e22a9d-73b1-4dc1-a1cf-d70796c00579 active
```

Figure 5. This figure shows updates in transection status

Error handling: system shows proper error message if any error occurs.



```
index chalo karo users
```

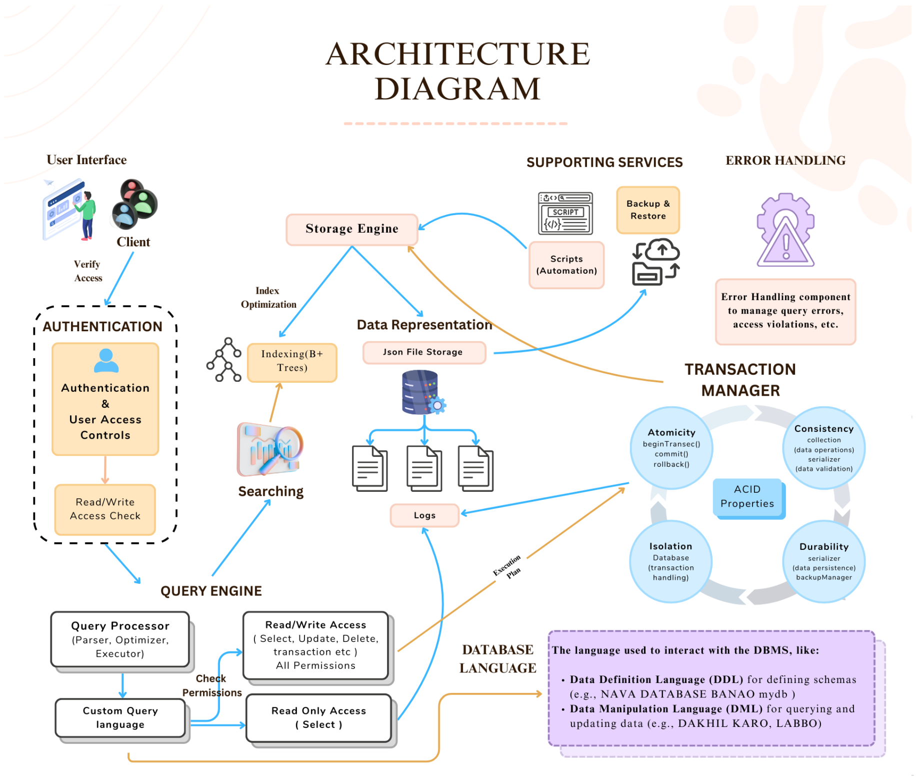
Error: Unknown query command: index chalo karo users

Unknown query command: index chalo karo users

Figure 6. This figure shows error message upon inserting an incorrect query

Architecture Diagram

Here is the architecture diagram of our database management system:


ARCHITECTURE DIAGRAM
User Interface
Client
Verify Access
AUTHENTICATION
Authentication & User Access Controls
Read/Write Access Check
Storage Engine
Index Optimization
Indexing(B+ Trees)
Searching
Data Representation
Json File Storage
Logs
SUPPORTING SERVICES
SCRIPT
Scripts (Automation)
Backup & Restore
ERROR HANDLING
Error Handling component to manage query errors, access violations, etc.
TRANSACTION MANAGER
Atomicity beginTransec() commit() rollback()
Consistency collection (data operations) serializer (data validation)
ACID Properties
Isolation Database (transaction handling)
Durability serializer (data persistence) backupManager
QUERY ENGINE
Query Processor (Parser, Optimizer, Executor)
Custom Query language
Check Permissions
Read/Write Access (Select, Update, Delete, transaction etc) All Permissions
Read Only Access (Select)
Execution Plan
DATABASE LANGUAGE

The language used to interact with the DBMS, like:

- **Data Definition Language (DDL)** for defining schemas (e.g., NAVA DATABASE BANAO mydb)
- **Data Manipulation Language (DML)** for querying and updating data (e.g., DAKHIL KARO, LABBO)

Figure 7. This figure shows the architecture diagram of our system

How to Run

Prerequisites

• Install Python 3.11 or higher.

• Ensure Tkinter is available (included with standard Python installations).

• Install required libraries: No external libraries are needed beyond the Python standard library.

Running in CLI Mode

1. Navigate to the project directory.

2. Run the command: python main.py

3. Use the CLI to enter queries (e.g., create database students).

Running in GUI Mode

1. Navigate to the project directory.

2. Run the command: python main.py --gui

3. The GUI window will open, allowing you to interact with the database using the tabbed query editor.

Troubleshooting

• Ensure the db/ directory exists and is writable for database storage.

• If the GUI does not launch, verify that Tkinter is installed correctly.

Queries

Below is the table of all the queries supported by the A2Z DBMS:

Table 2. Queries table

ENGLISH COMMAND	OUR DATABASE COMMAND (PUNJABI)	MEANING
BEGIN TRANSACTION	BEGIN TX	Start a new transaction
COMMIT	COMMIT	Commit the current transaction
ROLLBACK	ROLLBACK	Roll back the current transaction
CREATE DATABASE	NAVA DATABASE BANAO <name>	Create a new database
DROP DATABASE	DATABASE NU MITAO <name>	Delete a database
USE DATABASE	DATABASE CHALAO <name>	Switch to a database
CREATE COLLECTION	NAVA COLLECTION BANAO <name>	Create a new collection
DROP COLLECTION	COLLECTION NU MITAO <name>	Delete a collection
CREATE INDEX	INDEX BANAO <field> <collection>	Create an index on a field
LIST INDEXES	INDEX DIKHAO <collection>	List all indexes in a collection
ENABLE INDEXING	INDEX CHALO KARO	Enable indexing
DISABLE INDEXING	INDEX BAND KARO	Disable indexing
INSERT ONE	DAKHIL KARO <collection> {document}	Insert a single document
INSERT MANY	DAKHIL KARO <collection> [documents]	Insert multiple documents
UPDATE	BADLO <collection> {query} {update}	Update matching documents
DELETE	MITAO <collection> {query}	Delete matching documents
FIND	LABBO <collection> {query}	Find matching documents
FIND ALL	LABBO <collection>	Retrieve all documents
AGGREGATE	AGGREGATE IN <collection> [pipeline]	Perform aggregation
CREATE BACKUP	BACKUP BANAO	Create a database backup
RESTORE BACKUP	RESTORE KARO <name>	Restore a database from backup

Sample Queries

Create a new database

NAVA DATABASE BANAO mydb

Switch to database

DATABASE CHALAO mydb

Create collections

NAVA COLLECTION BANAO users

NAVA COLLECTION BANAO orders

Insert single document

DAKHIL KARO users {"name": "John", "age": 30, "email": "john@example.com", "active": true}

Insert multiple documents

DAKHIL KARO users [{"name": "Alice", "age": 25}, {"name": "Bob", "age": 35}]

Find documents (without index)

LABBO users {"age": {"$gt": 25}}

Update document

BADLO users {"name": "John"} {"$set": {"age": 31}}

Delete document

MITAO users {"name": "Bob"}

First find without index (note the time)

LABBO users {"age": {"$gt": 25}}

Create index on age field

INDEX BANAO age users

Enable indexing

INDEX CHALO KARO

Same find query with index (should be faster)

LABBO users {"age": {"$gt": 25}}

Disable indexing to compare

INDEX BAND KARO

Simple aggregation (average age)

AGGREGATE IN users [{"$group": {"_id": null, "avgAge": {"$avg": "$age"}}}]

Complex aggregation (group by active status)

```
AGGREGATE IN users [
  {"$match": {"age": {"$gt": 20}}},
  {"$group": {"_id": "$active", "count": {"$sum": 1}, "avgAge": {"$avg": "$age"}}}
]
```

Simple transaction

BEGIN TX

DAKHIL KARO orders {"user": "John", "items": ["laptop", "mouse"], "total": 1200}

BADLO users {"name": "John"} {"$inc": {"orders_count": 1}}

COMMIT

Transaction with rollback

BEGIN TX

DAKHIL KARO orders {"user": "Alice", "items": ["keyboard"], "total": 50}

(Something went wrong, rollback)

ROLLBACK

Create backup

BACKUP BANAO mydb_backup_2023

Restore from backup

RESTORE KARO mydb_backup_2023

Nested document query

```
DAKHIL KARO users {
  "name": "Emma",
  "address": {
    "street": "123 Main St",
    "city": "New York"
  },
  "tags": ["customer", "vip"]
}
```

Query on nested field

```
LABBO users {"address.city": "New York"}
```

Array query

```
LABBO users {"tags": "vip"}
```

Complex conditional query

```
LABBO users {
  "$or": [
    {"age": {"$lt": 30}},
    {"$and": [
      {"active": true},
      {"name": {"$ne": "John"}}
    ]}
  ]
}
```

Performance test script (run in sequence)

1. Initial setup

```
NAVA DATABASE BANAO perf_test
DATABASE CHALAO perf_test
NAVA COLLECTION BANAO test_data
```

2. Insert test data (1000 documents)

```
DAKHIL KARO test_data [
  {"value": 1, "category": "A", "timestamp": 1234567890},
  {"value": 2, "category": "B", "timestamp": 1234567891},
  ... (repeat with variations)
]
```

3. Test without index

```
LABBO test_data {"category": "A", "value": {"$gt": 5}}
```

4. Create index and test again

INDEX BANAO category test_data

INDEX BANAO value test_data

INDEX CHALO KARO

LABBO test_data {"category": "A", "value": {"$gt": 5}}

5. Compare aggregation performance

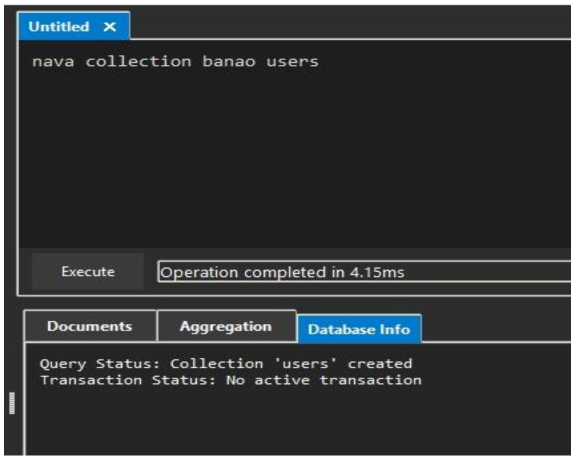
```
AGGREGATE IN test_data [
  {"$match": {"timestamp": {"$gt": 1234567000}}},
  {"$group": {"_id": "$category", "avgValue": {"$avg": "$value"}}}
]
```

Testing and Evaluation

The system was tested for functionality, performance, and reliability:

• **Functional Testing:** All commands (e.g., CRUD, transactions, aggregation) were tested using the GUI and CLI. For example, inserting 100 documents and querying them with indexing enabled.

• **Performance Testing:** Measured query execution time for operations like find and aggregate.



```
nava collection banao users
```

```
Operation completed in 4.15ms
```

```
Query Status: Collection 'users' created
Transaction Status: No active transaction
```

Figure 8. This figure shows the total time taken to execute a query

• **Crash Recovery Testing:** Simulated crashes during transactions and verified that logs were used to recover the database state.

Table 3. This table shows the time taken to execute different queries

Operation	**Data Size**	**Average Time**
Insert	1	3.5ms
Find	100	9.3ms
Aggregate	1,000	100ms

Challenges Faced

• **Status Bar Issue:** Initially, the transaction status bar at the bottom of the GUI did not display correctly due to Tkinter layout issues. This was resolved by moving the status display to the "Database Info" tab.

• **Tab Click Error:** An error occurred when clicking outside script tabs, which was fixed by adding error handling in the _on_tab_click method.

• **Transaction Isolation:** Implementing full isolation for concurrent transactions was challenging and remains a limitation due to the single-threaded nature of the system.

Notes

• **Indexing:** Uses basic indexing for faster searches, but lacks advanced structures like B-trees or support for full-text search.

• **Transactions:** Supported per database; multi-database transactions and concurrent transaction isolation are not implemented.

• **Aggregation:** Provides basic pipeline functionality similar to MongoDB, but lacks advanced operators like $lookup.

• **Performance:** In-memory structure improves read performance but may increase memory usage for large datasets.

Future Work

• **Concurrent Transactions:** Implement support for concurrent transactions with proper locking mechanisms to improve isolation.

• **Advanced Indexing:** Add support for B-tree or hash-based indexing for better query performance.

• **Enhanced Aggregation:** Include more aggregation operators (e.g., $lookup, $sort) to support complex data analysis.

• **Multi-Database Support:** Extend transaction support across multiple databases.

• **User Authentication:** Add user authentication and access control for secure data management.

References

• Python Standard Library Documentation: https://docs.python.org/3/library/

• Tkinter Documentation: https://docs.python.org/3/library/tkinter.html

• JSON Specification: https://www.json.org/

• MongoDB Documentation (for aggregation inspiration): https://docs.mongodb.com/manual/aggregation/

Conclusion

The A2Z DBMS provides a lightweight, flexible solution for data management, offering essential features like JSON document storage, CRUD operations, aggregation, transaction handling, and a user-friendly GUI. The custom query language and help system make it accessible for beginners, while the implementation of ACID properties ensures reliability. Through this project, the team gained hands-on experience with advanced database concepts, GUI development, and system design. The system is well-suited for educational purposes and small-scale applications, with potential for future enhancements like concurrent transaction support and advanced indexing.