SE 3XA3: Test Plan
GoDBMS

Team #7, Databased
Faiq Ahmed, ahmedf46
Eesha Qureshi, qureshe
Kevin Kannammalil, kannammk

March 11, 2022

Contents

List of Tables

List of Figures

Table 1: **Revision History**

Date	**Version**	**Notes**
March 9, 2022	1.0	Initial Document
March 10, 2022	1.2	Started Tests for Functional Requirements
March 10, 2022	1.2	Finished Introduction
March 11, 2022	1.2	Finished Functional Requirement Tests
March 11, 2022	1.2	Finished Non-Functional Requirement Tests
March 11, 2022	1.3	Final Draft

This document outlines the testing plan for the GoDBMS system.

1 General Information

1.1 Purpose

The purpose of this document is to provide in depth details of the testing plans and methods that will be used to verify the functionality of the GoDBMS software.

1.2 Scope

The test suites that comprise this document will be used to ensure that the software meets its functional and non-functional requirements that were outlined in the Software Requirements Specification (SRS). It will cover both the specific tests, as well as the testing tools and environments used to execute them. The tests will additionally cover verification procedures for the Proof of Concept (POC), as well as internal unit testing plans.

1.3 Acronyms, Abbreviations, and Symbols

Table 2: **Table of Abbreviations**

Abbreviation	**Definition**
SRS	Software Requirements Specification; a document that outlines important information about the software project such as stakeholders, functional requirements, non-functional requirements and project scope
POC	Proof of Concept; a prototype, demo or design that provides evidence of the project's feasibility
DBMS	**Database** Management System; software that allows a user to perform **database** transactions and operations
CLI	Command Line Interface; the point of communication between the software and the user
Go	GoLang

Table 3: **Table of Definitions**

Term	**Definition**
Database	A set of **record**s that store information organized by keys
Command Line	The point of communication between user and internal computer instructions
Table	An organized collection of **record**s in a **database**
Record	A single row of data in a **database**
Primary Key	A unique identifier for a **record** in a **database**
Struct	A modular collection of fields in GoLang
Schema	An outline of fields, table names, and relationships in a **database**
Columns	Vertical aggregations of data in a table
Catalog	A list storing the names of all the tables in the database. This list can be encoded and decoded from a file to allow persistent storage.
Tuple	An immutable finite sequence of elements
Message Passing	The method of invoking a process in a computer
Concurrency	The use of threads to run multiple computer processes at the same time
GoLang	A compiled programming language

1.4 Overview of Document

This document will outline the testing team and tools, test schedule, functional and non functional tests, POC testing, and unit testing.

2 Plan

2.1 Software Description

GoDBMS is a **database** management system based off of SimpleDB, which is an open source Java based DBMS. GoDBMS changes the implementation of this DBMS to be simpler and more focused on the **message passing concurrency** model of **Go** programming language to reduce the complexity of the design and improve its concurrent performance.

2.2 Test Team

The test team for this project consists of the members of the Databased group: Faiq Ahmed, Kevin Kannammalil, and Eesha Qureshi.

These members will be responsible for writing the test cases that encompass this project as the project is being developed, as well as create and distribute the necessary interviews and surveys needed to get feedback on the non functional requirements of this project.

2.3 Automated Testing Approach

Automated testing will be a large part of our system, since the system can easily be broken down into a set of basic queries that can prove correctness of the program. The user facing portion of the system is very minimal and only encompasses a basic **command line** interface which would need to be tested with manual testing. However, a large portion of the actual system from the queries to the storage of the **database** can be tested with unit tests as individual components.

2.4 Testing Tools

This system will be tested using **Go**'s built in testing library which has support for unit, integration, and fuzz testing. For testing of non functional requirements where interviews or surveys may be needed, we will be using google forms to create and receive feedback on the specified topics.

2.5 Testing Schedule

See Gantt Chart at the following url ...

3 System Test Description

3.1 Tests for Functional Requirements

3.1.1 Input Parsing

1. IP-T1

   Type: Functional, Dynamic, Manual

   Initial State: The **command line** interface has been started and is waiting for a user input

   Input: User enters an input that is not a create **table**, insert, delete, list or search statement

   Output: The system must print an error back to the user stating that their query syntax is invalid

   How test will be performed: The **command line** interface program will be ran manually and the input would be typed in. The tester will validate that it returns the correct error statement.

2. IP-BE1-T1

   Type: Functional, Dynamic, Automated

   Initial State: The parser has been initialized

   Input: A string query to create a **table** will be passed in to the parser testing function with a missing **table** name

   Output: The function must return an error stating that the **table** name is missing

How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax. The unit test will ensure that the function returns the correct error statement for the missing **table** name.

3. IP-BE1-T2

   Type: Functional, Dynamic, Automated

   Initial State: The parser has been initialized

   Input: A string query to create a **table** will be passed in to the parser testing function with missing **column**s

   Output: The function must return an error stating that the **table column**s are missing

   How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax. The unit test will ensure that the function returns the correct error statement for the missing **table column**s.

4. IP-BE1-T3

   Type: Functional, Dynamic, Automated

   Initial State: The parser has been initialized

   Input: A string query to create a **table** will be passed in to the parser testing function with a missing **primary key**

   Output: The function must return an error stating that the **table primary key** is missing

   How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax. The unit test will ensure that the function returns the correct error statement for the missing **primary key**.

5. IP-BE1-T4

   Type: Functional, Dynamic, Automated

   Initial State: The parser has been initialized

   Input: A string query to create a **table** will be passed in to the parser testing function with the correct syntax

Output: The function must return a **struct** that contains the **schema** of the create **table** statement

How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax before creating and returning a **struct** with the passed in information. The unit test must validate that the fields in the **struct** match the **schema** that was passed in the input string.

6. IP-BE2-T1

   Type: Functional, Dynamic, Automated

   Initial State: The parser has been initialized

   Input: A string query to delete a **table** will be passed in to the parser testing function with a missing **table** name

   Output: The function must return an error stating that the **table** name is missing

   How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax. The unit test will ensure that the function returns the correct error statement for the missing **table** name.

7. IP-BE2-T2

   Type: Functional, Dynamic, Automated

   Initial State: The parser has been initialized

   Input: A string query to delete a **table** will be passed in to the parser testing function with the correct syntax

   Output: The function must return a **struct** that contains the **schema** of the delete **table** statement

   How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax before creating and returning a **struct** with the passed in information. The unit test must validate that the fields in the **struct** match the **schema** that was passed in the input string.

8. IP-BE4-T1

   Type: Functional, Dynamic, Automated

Initial State: The parser has been initialized

Input: A string query to insert a **record** will be passed in to the parser testing function with a missing **table** name

Output: The function must return an error stating that the **table** name is missing

How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax. The unit test will ensure that the function returns the correct error statement for the missing **table** name.

9. IP-BE4-T2

   Type: Functional, Dynamic, Automated

   Initial State: The parser has been initialized

   Input: A string query to insert a **tuple** will be passed in to the parser testing function with missing **column**s

   Output: The function must return an error stating that the **record column**s are missing

   How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntaxd. The unit test will ensure that the function returns the correct error statement for the missing **record column**s.

10. IP-BE4-T3

    Type: Functional, Dynamic, Automated

    Initial State: The parser has been initialized

    Input: A string query to insert a **tuple**s will be passed in to the parser testing function with missing **record** data

    Output: The function must return an error stating that the **record** data is missing

    How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax. The unit test will ensure that the function returns the correct error statement for the missing **record** data.

11. IP-BE4-T4

    Type: Functional, Dynamic, Automated

    Initial State: The parser has been initialized

    Input: A string query to insert a **record** will be passed in to the parser testing function with the correct syntax

    Output: The function must return a **struct** that contains the **schema** of the insert **record** statement

    How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax before creating and returning a **struct** with the passed in information. The unit test must validate that the fields in the **struct** match the **schema** that was passed in the input string.

12. IP-BE5-T1

    Type: Functional, Dynamic, Automated

    Initial State: The parser has been initialized

    Input: A string query to modify a **record** will be passed in to the parser testing function with a missing **table** name

    Output: The function must return an error stating that the **table** name is missing

    How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax. The unit test will ensure that the function returns the correct error statement for the missing **table** name.

13. IP-BE5-T2

    Type: Functional, Dynamic, Automated

    Initial State: The parser has been initialized

    Input: A string query to modify a **record** will be passed in to the parser testing function with missing **primary key** to uniquely identify the **tuple**

    Output: The function must return an error stating that the **record primary key** is missing

How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax. The unit test will ensure that the function returns the correct error statement for the missing **primary key**.

14. IP-BE5-T3

    Type: Functional, Dynamic, Automated

    Initial State: The parser has been initialized

    Input: A string query to insert a **tuple**s will be passed in to the parser testing function with missing data to update

    Output: The function must return an error stating that the **record** data to update is missing

    How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax. The unit test will ensure that the function returns the correct error statement for the missing **record** data to update.

15. IP-BE5-T4

    Type: Functional, Dynamic, Automated

    Initial State: The parser has been initialized

    Input: A string query to modify a **record** will be passed in to the parser testing function with the correct syntax

    Output: The function must return a **struct** that contains the **schema** of the modify **record** statement

    How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax before creating and returning a **struct** with the passed in information. The unit test must validate that the fields in the **struct** match the **schema** that was passed in the input string.

16. IP-BE6-T1

    Type: Functional, Dynamic, Automated

    Initial State: The parser has been initialized

    Input: A string query to delete a **record** will be passed in to the parser testing function with a missing **table** name

Output: The function must return an error stating that the **table** name is missing

How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax. The unit test will ensure that the function returns the correct error statement for the missing **table** name.

17. IP-BE6-T2

    Type: Functional, Dynamic, Automated

    Initial State: The parser has been initialized

    Input: A string query to delete a **record** will be passed in to the parser testing function with missing **primary key** to uniquely identify the **tuple**

    Output: The function must return an error stating that the **record primary key** is missing

    How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax. The unit test will ensure that the function returns the correct error statement for the missing **primary key**.

18. IP-BE6-T3

    Type: Functional, Dynamic, Automated

    Initial State: The parser has been initialized

    Input: A string query to delete a **record** will be passed in to the parser testing function with the correct syntax

    Output: The function must return a **struct** that contains the **schema** of the delete **record** statement

    How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax before creating and returning a **struct** with the passed in information. The unit test must validate that the fields in the **struct** match the **schema** that was passed in the input string.

19. IP-BE7-T1

    Type: Functional, Dynamic, Automated

Initial State: The parser has been initialized

Input: A string query to search for **record**s will be passed in to the parser testing function with missing **table** names

Output: The function must return an error stating that the **table** names to search from is missing

How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax. The unit test will ensure that the function returns the correct error statement for the missing **table** names.

20. IP-BE7-T2

    Type: Functional, Dynamic, Automated

    Initial State: The parser has been initialized

    Input: A string query to search for **record**s will be passed in to the parser testing function with missing **table column**s to return

    Output: The function must return an error stating that the **table column**s to return are missing

    How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax. The unit test will ensure that the function returns the correct error statement for the missing **table column**s to return.

21. IP-BE7-T3

    Type: Functional, Dynamic, Automated

    Initial State: The parser has been initialized

    Input: A string query to search for **record**s will be passed in to the parser testing function with the correct syntax

    Output: The function must return a **struct** that contains the **schema** of the search **record** statement

    How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax before creating and returning a **struct** with the passed in information. The unit test must validate that the fields in the **struct** match the **schema** that was passed in the input string.

3.1.2 Saving Data

1. SD-BE1-T1

   Type: Functional, Dynamic, Manual

   Initial State: The **command line** interface has been started and is waiting for a user input, and the **database** has been initialized

   Input: User enters an input to create a **table** with the correct syntax and a unique **table** name

   Output: The system must print a success message back to the user informing them that their **table** has been created

   How test will be performed: The **command line** interface program will be ran manually and the input would be typed in. The tester will validate that it returns the correct success statement.

2. SD-BE1-T2

   Type: Functional, Dynamic, Automated

   Initial State: The **database** has been initialized

   Input: A **struct** to create a **table** has been passed in with the correct syntax and a unique **table** name

   Output: The system must create the specified **table** in the **catalog**

   How test will be performed: A unit test will pass the **struct** into the create **table** controller function which will validate the **struct** before inserting the **table** into the **catalog**. The unit test will then ensure that the **catalog** contains the specified **table** with the correct **schema**.

3. SD-BE1-T3

   Type: Functional, Dynamic, Automated

   Initial State: The **database** has been initialized and already has **table**s added to it

   Input: A **struct** to create a **table** has been passed in with the correct syntax and a **table** name that already exists

   Output: The system must return an error stating that a **table** by the specified name already exists

How test will be performed: A unit test will pass the **struct** into the create **table** controller function which will validate the **struct** and ensure that the **table** name doesn't already exist in the **catalog**. The unit test will then ensure that the function returns the correct error statement for **table** name already exists.

4. SD-BE2-T1

   Type: Functional, Dynamic, Manual

   Initial State: The **command line** interface has been started, is waiting for a user input, and the **database** already has **table**s added to it

   Input: User enters an input to delete a **table** with the correct syntax and an existing **table** name

   Output: The system must print a success message back to the user informing them that their **table** has been deleted

   How test will be performed: The **command line** interface program will be ran manually and the input would be typed in. The tester will validate that it returns the correct success statement.

5. SD-BE2-T2

   Type: Functional, Dynamic, Automated

   Initial State: The **database** has been initialized and already has **table**s added to it

   Input: A **struct** to delete a **table** has been passed in with the correct syntax and an existing **table** name

   Output: The system must delete the specified **table** in the **catalog**

   How test will be performed: A unit test will pass the **struct** into the delete **table** controller function which will validate the **struct** before deleting the **table** from the **catalog**. The unit test will then ensure that the **catalog** no longer contains that **table** name.

6. SD-BE2-T3

   Type: Functional, Dynamic, Automated

   Initial State: The **database** has been initialized

   Input: A **struct** to delete a **table** has been passed in with the correct syntax and a **table** name that does not exist

Output: The system must return an error stating that a **table** by the specified name does not exist

How test will be performed: A unit test will pass the **struct** into the delete **table** controller function which will validate the **struct** and ensure that the **table** name exists in the **catalog**. The unit test will then ensure that the function returns the correct error statement for **table** name does not exist.

7. SD-BE3-T1

   Type: Functional, Dynamic, Automated

   Initial State: The **database** has been initialized

   Input: A string query to list all **table**s has been passed in to the parser testing function with the correct syntax

   Output: The function must return all **table**s in the **catalog**

   How test will be performed: A unit test will pass the input string into the parser testing function which would process the string and validate its syntax. The unit test must validate that the returned strings contains all the **table**s that the **catalog** also contains. This can be obtained by calling the get **catalog** function directly.

8. SD-BE4-T1

   Type: Functional, Dynamic, Manual

   Initial State: The **command line** interface has been started, is waiting for a user input, and the **database** already has **table**s added to it

   Input: User enters an input to insert a **record** with the correct syntax, an existing **table** name, and a **primary key** that does not already exist

   Output: The system must print a success message back to the user informing them that their **record** has been inserted

   How test will be performed: The **command line** interface program will be ran manually and the input would be typed in. The tester will validate that it returns the correct success statement.

9. SD-BE4-T2

   Type: Functional, Dynamic, Automated

Initial State: The **database** has been initialized and already has **table**s added to it

Input: A **struct** to insert a **record** has been passed in with the correct syntax, an existing **table** name, and a **primary key** that does not already exist

Output: The system must insert the specified **tuple** in the **database**

How test will be performed: A unit test will pass the **struct** into the insert **record** controller function which will validate the **struct** before inserting the **record** into the **database**. The unit test will then ensure that the **database** contains the specified **record**.

10. SD-BE4-T3

    Type: Functional, Dynamic, Automated

    Initial State: The **database** has been initialized

    Input: A **struct** to insert a **record** has been passed in with the correct syntax and a **table** name that does not exist

    Output: The system must return an error stating that a **table** by the specified name does not exist

    How test will be performed: A unit test will pass the **struct** into the insert **record** controller function which will validate the **struct** and ensure that the **table** name exists in the **catalog**. The unit test will then ensure that the function returns the correct error statement stating that the **table** name does not exist.

11. SD-BE5-T1

    Type: Functional, Dynamic, Manual

    Initial State: The **command line** interface has been started, is waiting for a user input, and the **database** already has **table**s and **record**s added to it

    Input: User enters an input to modify a **record** with the correct syntax, an existing **table** name, and a **primary key** to uniquely identify the **record**

    Output: The system must print a success message back to the user informing them that their **record** has been modified

How test will be performed: The **command line** interface program will be ran manually and the input would be typed in. The tester will validate that it returns the correct success statement.

12. SD-BE5-T2

    Type: Functional, Dynamic, Automated

    Initial State: The **database** has been initialized and already has **table**s and **record**s added to it

    Input: A **struct** to modify a **record** has been passed in with the correct syntax, an existing **table** name, and a **primary key** to uniquely identify the **record**

    Output: The system must modify the specified **tuple** in the **database**

    How test will be performed: A unit test will pass the **struct** into the insert **record** controller function which will validate the **struct** before modifying the **record** in the **database**. The unit test will then ensure that the **database** contains the modified **record**.

13. SD-BE5-T3

    Type: Functional, Dynamic, Automated

    Initial State: The **database** has been initialized and already has **table**s and **record**s added to it

    Input: A **struct** to modify a **record** has been passed in with the correct syntax and a **table** name that does not exist

    Output: The system must return an error stating that a **table** by the specified name does not exist

    How test will be performed: A unit test will pass the **struct** into the modify **record** controller function which will validate the **struct** and ensure that the **table** name exists in the **catalog**. The unit test will then ensure that the function returns the correct error statement stating that the **table** name does not exist.

14. SD-BE5-T4

    Type: Functional, Dynamic, Automated

    Initial State: The **database** has been initialized and already has **table**s and **record**s added to it

Input: A **struct** to modify a **record** has been passed in with the correct syntax, an existing **table** name, and a **primary key** that does not exist

Output: The system must return an error stating that a **record** with the specified **primary key** does not exist

How test will be performed: A unit test will pass the **struct** into the modify **record** controller function which will validate the **struct** and ensure that the **primary key** exists in the **table**. The unit test will then ensure that the function returns the correct error statement stating that the **primary key** does not exist.

15. SD-BE6-T1

    Type: Functional, Dynamic, Manual

    Initial State: The **command line** interface has been started, is waiting for a user input, and the **database** already has **table**s and **record**s added to it

    Input: User enters an input to delete a **record** with the correct syntax, an existing **table** name, and a **primary key** to uniquely identify the **record**

    Output: The system must print a success message back to the user informing them that their **record** has been deleted

    How test will be performed: The **command line** interface program will be ran manually and the input would be typed in. The tester will validate that it returns the correct success statement.

16. SD-BE6-T2

    Type: Functional, Dynamic, Automated

    Initial State: The **database** has been initialized and already has **table**s and **record**s added to it

    Input: A **struct** to delete a **record** has been passed in with the correct syntax, an existing **table** name, and a **primary key** to uniquely identify the **record**

    Output: The system must delete the specified **tuple** in the **database**

    How test will be performed: A unit test will pass the **struct** into the delete **record** controller function which will validate the **struct** before delete the **record** from the **database**. The unit test will then ensure that the **database** no longer contains the deleted **record**.

17. SD-BE6-T3

    Type: Functional, Dynamic, Automated

    Initial State: The **database** has been initialized and already has **table**s and **record**s added to it

    Input: A **struct** to delete a **record** has been passed in with the correct syntax and a **table** name that does not exist

    Output: The system must return an error stating that a **table** by the specified name does not exist

    How test will be performed: A unit test will pass the **struct** into the delete **record** controller function which will validate the **struct** and ensure that the **table** name exists in the **catalog**. The unit test will then ensure that the function returns the correct error statement stating that the **table** name does not exist.

18. SD-BE7-T1

    Type: Functional, Dynamic, Manual

    Initial State: The **command line** interface has been started, is waiting for a user input, and the **database** already has **table**s and **record**s added to it

    Input: User enters an input to search for **record**s with the correct syntax, and an existing **table** name

    Output: The system must print back all **record**s find from their search query back to the user

    How test will be performed: The **command line** interface program will be ran manually and the input would be typed in. The tester will validate that it returns the correct search results.

19. SD-BE7-T2

    Type: Functional, Dynamic, Automated

    Initial State: The **database** has been initialized and already has **table**s and **record**s added to it

Input: A **struct** to search for **record**s has been passed in with the correct syntax and a **table** name that does not exist

Output: The system must return an error stating that a **table** by the specified name does not exist

How test will be performed: A unit test will pass the **struct** into the search **record** controller function which will validate the **struct** and ensure that the **table** name exists in the **catalog**. The unit test will then ensure that the function returns the correct error statement stating that the **table** name does not exist.

20. SD-BE8-T1

    Type: Structural, Static, Manual

    Initial State: Testers have access to the source code for the transactions in the **database** management system.

    Input: The testers do a code walk through to ensure that the system correctly locks the **record** during a write.

    Output: We are able to determine whether **struct**urally the code will prevent writes and other transactions at the same time.

    How test will be performed: This test will be performed completely manually and would be a code walk through to determine if the locking mechanism for the system is programmed correctly.

21. SD-BE8-T2

    Type: Functional, Dynamic, Automated

    Initial State: The **database** has been initialized and already has **table**s and **record**s added to it

    Input: A **struct** to modify a **tuple** will be passed in with the correct syntax, and an existing **table** name

    Output: The system will ensure that the **record** has been locked

    How test will be performed: A function to check if the **record** has been locked will be called immediately after the **struct** to modify a **tuple** has been passed in to ensure that the **record** is locked during a write.

22. SD-BE8-T3

Type: Functional, Dynamic, Manual

Initial State: The **database** has been initialized and already has **table**s and **record**s added to it

Input: The tester will manually pass in multiple read, and write queries to the **database** through multiple CLIs

Output: The system will output the result of all passed in queries

How test will be performed: A tester will manually pass in multiple queries at the same time and ensure the result of the queries is correct.

3.2 Tests for Nonfunctional Requirements

3.2.1 Look and Feel Testing

1. NFR-1-LF1

   Type: Dynamic, Manual

   Initial State: The **database** is installed locally and the **command line** interface for the **database**

   Condition: There is already preexisting data in the **database**

   Input: The tester enters multiple search queries

   Output: The **command line** interface returns the corresponding results of the queries

   How test will be performed: The tester will create a search query for a preexisting **table** and wait for an output, whether that be an error or a successful output. The tester will answer Question 1 of the Usability Survey to determine whether the existing User Interface fails the appearance requirements.

2. NFR-2-LF2

   Type: Dynamic, Manual

   Initial State: The **database command line** interface is installed.

   Input: The tester runs the main source file.

Output: The tester is greeted with a user interface that matches the layout and style of the preexisting **command line**.

How test will be performed: The tester will download the **database** files and run the source file on the **command line**. The tester will then answer Question 2 of the Usability Survey so the developers can gather feedback on how to change the styling of the User Interface.

3.2.2 Usability and Humanity Testing

1. NFR-3-UH1

   Type: Dynamic, Manual

   Initial State: The **database command line** interface is installed.

   Input: The tester runs the main source file.

   Output: The tester is greeted with a user interface and prompts to start.

   How test will be performed: An amateur tester which can also be a student with a basic understand of **database** will run the **database** and answer Question 2 of the Usability Survey to provide feedback. This feedback will be crucial considering the **database** is meant to be designed in a way for beginners to use and learn. Based on the feedback, the developers may simplify the language and user experience further.

2. NFR-4-UH2

   Type: Dynamic, Manual

   Initial State: The **database command line** interface is open to use.

   Input: The tester runs a faulty query to produce an error.

   Output: The tester receives an error that they can use to deduce how comprehensible the **database** prompts and errors are.

   How test will be performed: A tester familiar with the English language would run the **database** for the first time and try out queries to see how much they understood. Then they will respond to Question 3 in the Usability Survey to receive feedback about their experience to review the language the developers used to communicate with the users.

3.2.3 Performance Testing

1. NFR-5-P1

   Type: Dynamic, Manual

   Initial State: The **database** is ready to use with **table**s created.

   Condition: There is a reasonable amount of data already stored into the **database**.

   Input: The tester runs multiple search queries on the data and **record**s the time taken for the output.

   Output: The **database** retrieves the data and returns it to the user on the UI.

   How test will be performed: The tester keeps track of the time taken for output of the multiple search queries and then calculates the average time for the transactions. The average time is then compared to the expected *RESPONSE_TIME* to figure out if there are any bottlenecks.

2. NFR-6-P2

   Type: Structural, Dynamic, Manual

   Initial State: The **database command line** interface is open to use.

   Condition: There is a **table** with the correct **schema** to accept numbers as values.

   Input: The tester creates and runs an insert statement query with numbers that have decimals as values. The tester then searches for the that **record** in the **database**.

   Output: The tester receives a result of the search query with the correct precision.

   How test will be performed: A tester will create a new **database table** with a CREATE TABLE query specifying one of the **column**s to contain numbers. They will then insert mock data that contain decimals into the **table** and run search queries onto it to figure out whether it maintains the precision of decimals for the values compared to when it was inserted.

3. NFR-7-P3

   Type: Structural, Dynamic, Manual

   Initial State: The **database command line** interface is open to use.

   Condition: There is no data currently stored in the **database**.

   Input: The tester creates a new **table** and runs multiple insert statement.

   Output: A *.data* file is generated for that **table**.

   How test will be performed: A tester will create a new **database table** with a CREATE TABLE query and will then insert mock data. The tester should the *.data* file in the Data folder of their local directory.

3.2.4 Operational and Environmental Testing

1. NFR-8-OE1

   Type: Dynamic, Manual

   Initial State: There is currently an existing internet connection and the **database** is not running.

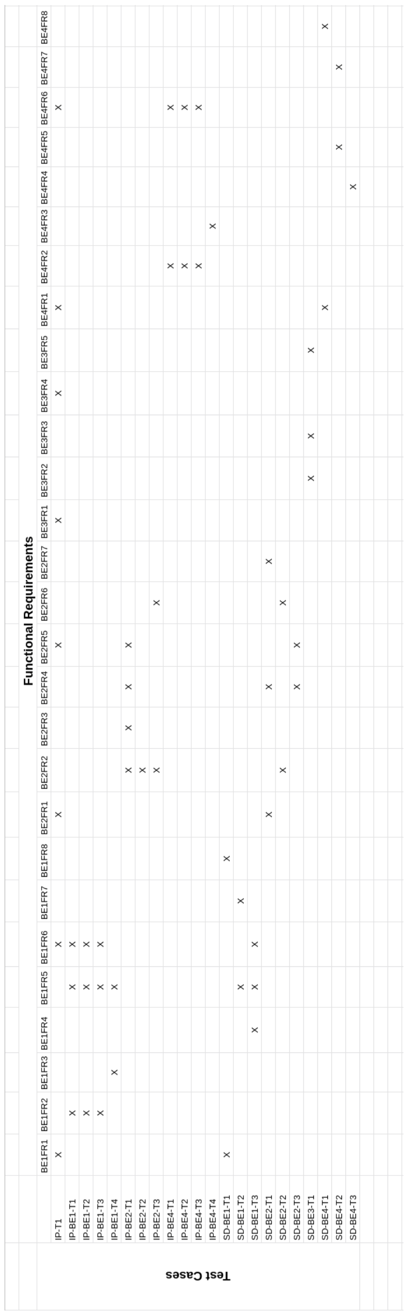
   Condition: The tester disconnects from the internet.

   Input: The tester starts the **database**.

   Output: The **database command line** interface runs as usual.

   How test will be performed: A tester will compare the response of the **database** from when there was an existing internet connection to when there isn't. All the functionalities of the **database** should still be working as expected to confirm that there is no dependence of the **database** on the internet.

3.3 Traceability Between Test Cases and Requirements

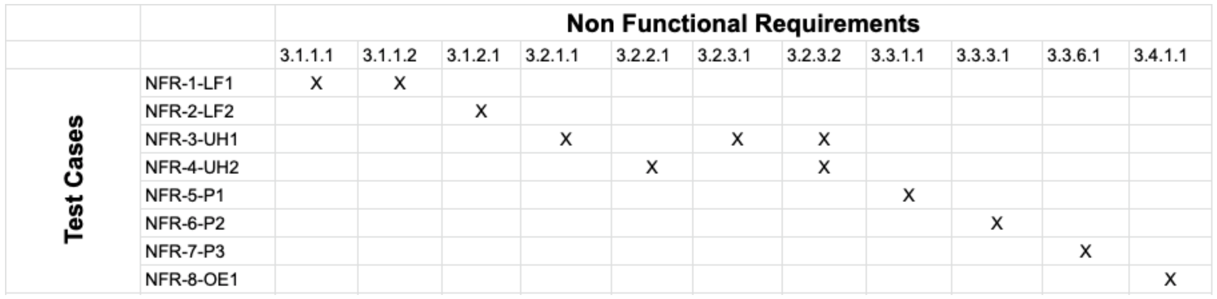


	Functional Requirements																											
Test Cases	BE1FR1	BE1FR2	BE1FR3	BE1FR4	BE1FR5	BE1FR6	BE1FR7	BE1FR8	BE2FR1	BE2FR2	BE2FR3	BE2FR4	BE2FR5	BE2FR6	BE2FR7	BE3FR1	BE3FR2	BE3FR3	BE3FR4	BE3FR5	BE4FR1	BE4FR2	BE4FR3	BE4FR4	BE4FR5	BE4FR6	BE4FR7	BE4FR8
IP-T1	X					X			X				X			X			X		X					X		
IP-BE1-T1		X			X	X																						
IP-BE1-T2		X			X	X																						
IP-BE1-T3		X			X	X																						
IP-BE1-T4			X		X																							
IP-BE2-T1										X	X	X	X															
IP-BE2-T2										X																		
IP-BE2-T3										X				X														
IP-BE4-T1																						X				X		
IP-BE4-T2																						X				X		
IP-BE4-T3																						X				X		
IP-BE4-T4																							X					
SD-BE1-T1	X							X																				
SD-BE1-T2					X		X																					
SD-BE1-T3				X	X	X																						
SD-BE2-T1									X			X			X													
SD-BE2-T2										X				X														
SD-BE2-T3												X	X															
SD-BE3-T1																	X	X		X								
SD-BE4-T1																					X							X
SD-BE4-T2																									X		X	
SD-BE4-T3																								X				

Figure 1: Traceability Matrix for Functional Requirements in BE1 to BE4

		Functional Requirements																										
		BE5FR1	BE5FR2	BE5FR3	BE5FR4	BE5FR5	BE5FR6	BE5FR7	BE5FR8	BE5FR9	BE6FR1	BE6FR2	BE6FR3	BE6FR4	BE6FR5	BE6FR6	BE6FR7	BE7FR1	BE7FR2	BE7FR3	BE7FR4	BE7FR5	BE7FR6	BE8FR1	BE8FR2	BE8FR3	BE8FR4	BE8FR5
Test Cases	IP-T1	X						X			X				X			X				X		X				
	IP-BE5-T1		X					X																				
	IP-BE5-T2		X					X																				
	IP-BE5-T3		X					X																				
	IP-BE5-T4			X																								
	IP-BE6-T1											X			X													
	IP-BE6-T2											X			X													
	IP-BE6-T3												X															
	IP-BE7-T1																		X			X						
	IP-BE7-T2																		X			X						
	IP-BE7-T3																			X								
	SD-BE5-T1	X								X																		
	SD-BE5-T2						X		X																			
	SD-BE5-T3				X																							
	SD-BE5-T4					X																						
	SE-BE6-T1										X						X											
	SE-BE6-T2															X												
	SE-BE6-T3													X														
	SE-BE7-T1																	X					X					
	SE-BE7-T2																				X							
	SE-BE8-T1																								X	X	X	X
	SE-BE8-T2																								X			
	SE-BE8-T3																									X	X	X

Figure 2: Traceability Matrix for Functional Requirements in BE5 to BE8



		Non Functional Requirements										
		3.1.1.1	3.1.1.2	3.1.2.1	3.2.1.1	3.2.2.1	3.2.3.1	3.2.3.2	3.3.1.1	3.3.3.1	3.3.6.1	3.4.1.1
Test Cases	NFR-1-LF1	X	X									
	NFR-2-LF2			X								
	NFR-3-UH1				X		X	X				
	NFR-4-UH2					X		X				
	NFR-5-P1								X			
	NFR-6-P2									X		
	NFR-7-P3										X	
	NFR-8-OE1											X

Figure 3: Traceability Matrix for Nonfunctional Requirements in NFR-1 to NFR-8

4 Tests for Proof of Concept

Our proof of concept demo implemented a good amount of the functional requirements related to creating a **table**, inserting **record**s, searching for **record**s and listing all **table**s. Most of the tests for the proof of concept were manual integration tests as follows:

4.1 System-Wide Testing

1. POC-T1

   Type: Functional, Dynamic, Manual

   Initial State: The **command line** interface is initialized and waiting for user input

   Input: A create **table** statement with missing fields

   Output: Appropriate error for missing fields is returned to the user

   How test will be performed: This test was performed manually by supplying incorrect create **table** statements missing **primary key**s or **table** names and the output was manually verified to return the correct error statement.

2. POC-T2

   Type: Functional, Dynamic, Manual

   Initial State: The **command line** interface is initialized and waiting for user input

   Input: A insert tuple statement with missing fields

   Output: Appropriate error for missing fields is returned to the user

   How test will be performed: This test was performed manually by supplying incorrect insert tuple statements missing **table** name or **column** data and the output was manually verified to return the correct error statement.

3. POC-T3

Type: Functional, Dynamic, Manual

Initial State: The **command line** interface is initialized and waiting for user input

Input: A create **table** statement without syntactic errors and a unique **table** name, followed by a list **table**s statement

Output: The newly created **table** should be displayed in the output of the list **table**s statement

How test will be performed: This test was performed manually a correct create **table** statement and then supplying a list **table** statement to observe the newly created **table** in the output.

4. POC-T4

   Type: Functional, Dynamic, Manual

   Initial State: The **command line** interface is initialized, waiting for user input and the **database** has **table**s inserted

   Input: A insert **record** statement without syntactic errors and a unique **table** name, followed by a select statement

   Output: The newly inserted **record** should be displayed in the output of the select statement statement

   How test will be performed: This test was performed manually a correct insert **record** statement and then running a select query on the **table** to observe the newly created **record** in the output.

5 Comparison to Existing Implementation

GoDBMS is a simpler implementation of a DBMS as compared to SimpleDB. It is implemented in Go rather than Java. Memory usage has so far been optimized by using pointers instead of implicit objects. Only basic data types string and integer are supported in GoDBMS as compared to the larger range of types allowed in SimpleDB. A simple CLI has also been implemented instead of requiring a third party software to pass queries. Concurrency and message passing remains to be implemented. Overall however, GoDBMS preserves the basic functionality of SimpleDB.

6 Unit Testing Plan

6.1 Unit testing of internal functions

Unit testing on internal functions will be performed using Go's built-in standard library for testing called testing package. Tests for each unit/function are written in the form of individual testing functions, in which the result assigned the value of a function call with predefined test case parameters. The result is then compared to the manually determined expected result with comparator logic. Using the Go testing package library, all the test functions in the test driver will execute when the go test command is ran, and the **command line** will display a PASS or FAIL status for each test as well as its runtime.

6.2 Unit testing of output files

Output and log files are not testable in this implementation.

7 Appendix

7.1 Symbolic Parameters

The definition of the test cases will call for SYMBOLIC_CONSTANTS. Their values are defined in this section for easy maintenance.

$RESPONSE\_TIME$ = 5 seconds

7.2 Usability Survey Questions?

1. On a scale of 1 to 10, how comfortable were you using the **database** on the **command line** interface compared to other **database**s on the **command line** interface like IBM DB2?

2. Were they any visual disturbances or inconveniences when starting up the **database**?

3. On a scale of 1 to 10, how intuitive was the user experience when you first started using the **database**

4. Were there any confusions with the prompts or errors that were given by the **database**?