

NOSQL DATABASE: OPERATIONAL CONCEPT DOCUMENT

CIS 681 – Software Modeling and Analysis
Project 1

Vishtasp Jokhi
vpjokhi@syr.edu

TABLE OF CONTENTS

EXECUTIVE SUMMARY	2
INTRODUCTION	3
CONCEPTS	3
1. KEY-VALUE DATABASE	3
2. IN-MEMORY DATABASE	4
REQUIRED FUNCTIONALITITES	4
ORGANISING PRINCIPLES	5
USES	6
USERS	6
1. COURSE INSTRUCTOR AND TEACHING ASSISTANTS	6
2. DEVELOPERS	6
POSSIBLE EXTENSIONS	7
USER INTERFACES	8
STRUCTURE AND PARTITIONING	10
KEY APPLICATION ACTIVITIES	13
QUERY PROCESSING	13
1. WRITE TYPE ACTIVITIES	13
2. READ TYPE ACTIVITIES	13
3. SAVE ACTIVITIES	14
SHARDING	16
CRITICAL ISSUES	18
CONCLUSION	19

LIST OF FIGURES

Figure 1: Input User Interface	8
Figure 2: Output User Interface	9
Figure 3: Package Diagram	10
Figure 4: Query Processing Activity Diagram	15
Figure 5: Sharding Activity Diagram	16

EXECUTIVE SUMMARY

SQL (Structured Query Language) or relational databases have some drawbacks when dealing with certain types of applications, for which they are not at all well-suited. They perform very inefficiently when dealing with requirements for some modern day applications, like handling large streams of data, to pick one example. This has led to the rise in popularity of NoSQL databases, which offer higher flexibility and performance. This document presents the concept and a possible architecture for implementing a key-value type of NoSQL database.

The functioning of the NoSQL database is divided into various smaller, logically grouped packages, depending on the tasks that need to be carried out or functionalities that need to be provided. The operation of the database can be divided into the following high-level activities: Testing, Query Processing, Sharding and Displaying. Each of these activities comprise of multiple tasks.

In its current form, the NoSQL database will primarily be used by the course Instructor and Teaching Assistants to test and evaluate its functionalities. Other users will be developers, who will look at extending the capability and functionalities of the database to support more features. Correspondingly, the main use will be to demonstrate the functionalities of the NoSQL database, and to explore ways in which the design choices affect these functionalities. Extensions and possible use cases will also be discussed briefly.

There are certain significant issues that need to be addressed while developing a NoSQL database. Some of these are:

- Sharding
- Query Processing
- Variable sharding strategies for applications involving multiple NoSQL databases with different key and value types
- Enforcing security in databases accessed over distributed systems
- Incorporating ACID properties

INTRODUCTION

CONCEPTS

1. KEY-VALUE DATABASE

The goal is to develop a key-value type of database. This type of database holds a set of key and value pairs, and can be considered analogous to a dictionary. Each key is unique, and has a corresponding value associated with it.

A key could be of any data type. Each key is passed through a hash function to generate a hash value, which can be thought of as a numeric index. This is, in turn, associated with the value corresponding to that particular key. It is important to note that though a key is unique, the hash generated by the key may not be unique. This issue will be addressed momentarily.

The value part of each pair also stores metadata, or data about the data that is stored. This could include a short text description, a time-date stamp, keys pointing to related values, and so on. The rest of the value part contains the actual data to be stored, which we will call *payload*, to avoid confusion.

One of the most interesting features of such a database is that, the payload could hold any object or data type, and possibly multiple objects. This grants the database a lot more flexibility than relational databases. Thus, we can say that the payload can hold data of a generic type.

This type of structure is the heart of the key-value type of NoSQL database. We will develop the key-value database using the in-built Dictionary functionality provided by the C# Standard Library. The Dictionary has its own hash function which will be responsible for mapping the keys to their values. As mentioned above, two keys may generate the same hash. Such cases are handled by the Dictionary, and developer would not need to provide for such eventualities. However, it is important to keep in mind that when dealing with extremely large data sets, the likelihood of multiple keys generating the same hash increases, and hence the time taken to process queries will also increase.

Complex graph relationships or dependencies can also be mapped quite easily in such databases, by storing the keys that point to the related elements as part of the element metadata.

2. IN-MEMORY DATABASE

The database to be developed is an In-Memory database. This means that the entire database is stored in the RAM, which is volatile memory. This has some major advantages and disadvantages, as follows:

Advantages:

Since the database exists in the RAM, also known as the main memory, its performance is much faster and more predictable than a database stored on a storage disk. This is because executing the same task would require fewer CPU operations. Also, the seek time is eliminated when querying data in memory, as opposed to data on a disk.

Disadvantages:

Since the database is stored in volatile memory, the data can be permanently lost in case there is a power loss, or if the application hangs and needs to be reset. Permanent storage of the data has to be done explicitly. Effective means of doing that need to be developed into the application.

REQUIRED FUNCTIONALITIES

Its apparent lack of structure make NoSQL databases very flexible. They do not enforce a fixed database schema, unlike relational databases. However, this flexibility comes at a cost, in that it places a much greater burden on the developer to enforce certain features that are inherent in SQL databases. ACID properties of transactions in SQL databases are a great example of this.

That being said, NoSQL databases have huge advantages over SQL databases when it comes to applications that have the following requirements:

- Supporting heterogeneous collections of data
- Handling extremely large data sets
- Handling large streams of data
- Processing high volumes of queries received over a distributed system
- Effectively managing data with complex dependencies, relationships or graph structures

The above requirements shall be implemented, along with certain others that could be easily implemented with relational databases. However, the developer must explicitly take care to implement these functionalities:

- Maintaining data integrity
- Ability to support compound queries of a complex nature
- Ability to query data by its key, metadata and payload
- Augmenting an existing database with data from an external source
- Saving the database to persistent memory
- Ability to rollback to a previously saved state
- In case the application is run over a network, providing secure levels of access depending on the type of user

ORGANISING PRINCIPLES

Most of the functionalities mentioned above will be achieved by splitting them into smaller tasks. Each task, or a group of related tasks, will be put into separate packages that make it easier to understand the operation the overall application. The various packages that make up the application and will include packages that control the program flow, test the various functionalities, process queries, perform writing and modification of data, display results to the console, implement sharding and save the database to persistent memory on command as well as at regular intervals.

USES

The inherent characteristics of NoSQL databases make them particularly well suited for applications that involve high throughput from streams, deal with very large sets of data, or data that has a complex relationship structure. Some examples of such applications are Big Data analysis, real-time analysis of social media networks, etc.

The current project focuses on designing and implementing a key-value database, and as such, its primary uses will be restricted to testing and evaluating the functionalities, their performance and the implications of design choices.

USERS

1. COURSE INSTRUCTOR AND TEACHING ASSISTANTS

These users will focus on testing the implemented database. This includes checking the various required functionalities and evaluating the overall performance of the application.

IMPACT ON DESIGN:

Ideally, testing should be designed such that it takes minimum input from the user, and displays the results of the test effectively in a lucid, concise manner.

Keeping this in mind, the program should include a Test package that will be responsible for demonstrating each of the requirements by running step-by-step through a series of tests. The results should be displayed concisely to facilitate easy comprehension. Processes involving a change in the state of the database should display the original state of the database, state the nature of the change and then display the altered state of the database.

2. DEVELOPERS

Developers would typically use the database in the following two ways:

a. Extending the capability of the existing program: This entails incorporating functionalities that will improve the usefulness of the application. For example,

providing the capability to grant remote access over a distributed system. Such extensions could be implemented via plug-ins or dynamic link libraries.

b. Incorporating the program into a specific application: Developers could take the generic key-value database that will be developed, and adapt it towards a specific application.

IMPACT ON DESIGN

While developing the program, care should be taken to ensure that the design doesn't restrict possible expansion and adaptation. Due consideration should be given to the possibility of accepting plug-ins and working with dynamic link libraries.

POSSIBLE EXTENSIONS

Using the database as is would not provide many real world applications. However, there are scores of possible applications that a key-value type of database could be suitable for, with appropriate extensions. Some examples are:

- Granting access to remote users over a distributed system. Different levels of access could be provided depending on the class of user.
- Expanding the application for use in a data management service for a large collaboration system.
- Real-time analysis of complex social media networks
- Accepting, storing and analysing high volumes of data from projects like the Large Hadron Collider, Sloan Sky Survey, etc.

Another point worth noting is that a NoSQL database can also be used for applications that currently use an SQL database quite effectively. So while there may be no great incentive to change to a NoSQL database in the near future, it is possible that such applications may start using NoSQL databases at some point down the line. Some examples of these applications are:

- Library management software
- Hospital patient records
- Inventory management software
- Banking applications

In these cases, it will be developers who adapt the existing database to suit these applications, which will ultimately be used by the end users via an appropriate GUI.

USER INTERFACES

The application will be run from the command prompt. Further extensions may implement a GUI.

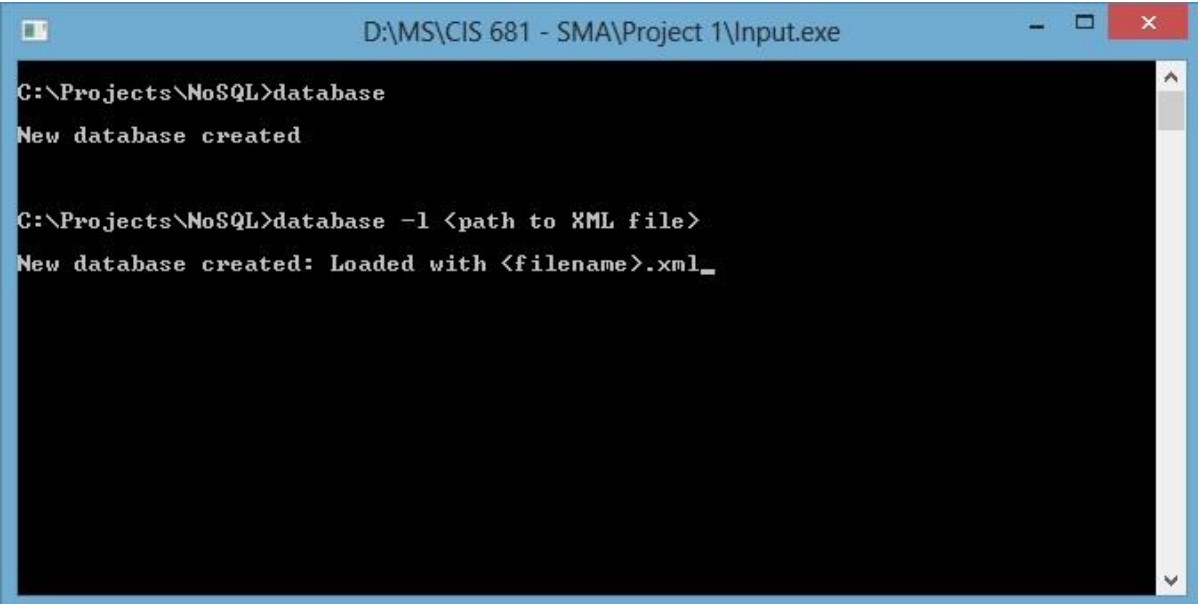
On the command line, the NoSQL database application will be run via the following command:

```
database <-l filename.xml>
```

The default command (-l option not specified) will create the database without writing any values into it.

If the <-l> option followed by the file name of the XML file containing the input will create the database and write the contents of the XML file into it.

A sample image of the interface with the input command is shown in Figure 1 below:



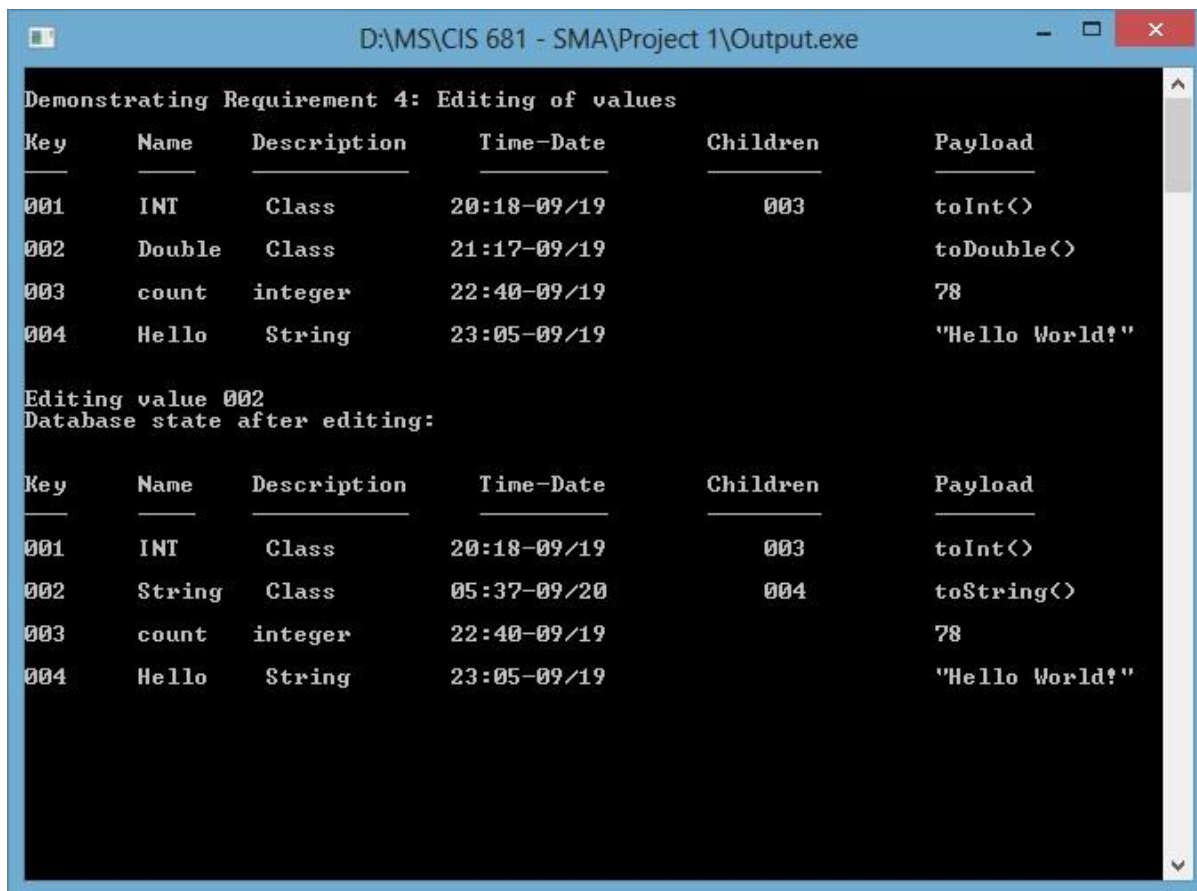
```
D:\MS\CIS 681 - SMA\Project 1\Input.exe
C:\Projects\NoSQL>database
New database created

C:\Projects\NoSQL>database -l <path to XML file>
New database created: Loaded with <filename>.xml_
```

Figure 1

As mentioned previously, when running tests, a concise view is desired in the output.

Figure 2 shows a sample output window, with the desired functionality clearly demonstrated.



```
Demonstrating Requirement 4: Editing of values
Key      Name      Description      Time-Date      Children      Payload
-----
001      INT       Class            20:18-09/19    003           toInt()
002      Double    Class            21:17-09/19    004           toDouble()
003      count     integer          22:40-09/19    78            78
004      Hello     String           23:05-09/19    "Hello World!"

Editing value 002
Database state after editing:
Key      Name      Description      Time-Date      Children      Payload
-----
001      INT       Class            20:18-09/19    003           toInt()
002      String    Class            05:37-09/20    004           toString()
003      count     integer          22:40-09/19    78            78
004      Hello     String           23:05-09/19    "Hello World!"
```

Figure 2

STRUCTURE AND PARTITIONING

Most modern software applications are so large, that it is impractical to develop the program as one single source file. Instead, the application is developed as a collection of packages. The high level functionalities of the application are broken down into smaller tasks that must be carried out, in order to implement those functionalities. Logically similar sets of tasks are grouped together, into what we call packages. The various packages interact with each other, to provide the fully functional software. The chief advantages of such an approach are easier development, testing and debugging.

The Package structure of the NoSQL database application is shown in Figure 3. The Package Diagram is followed by a brief description of each individual package, and its interactions with other packages.

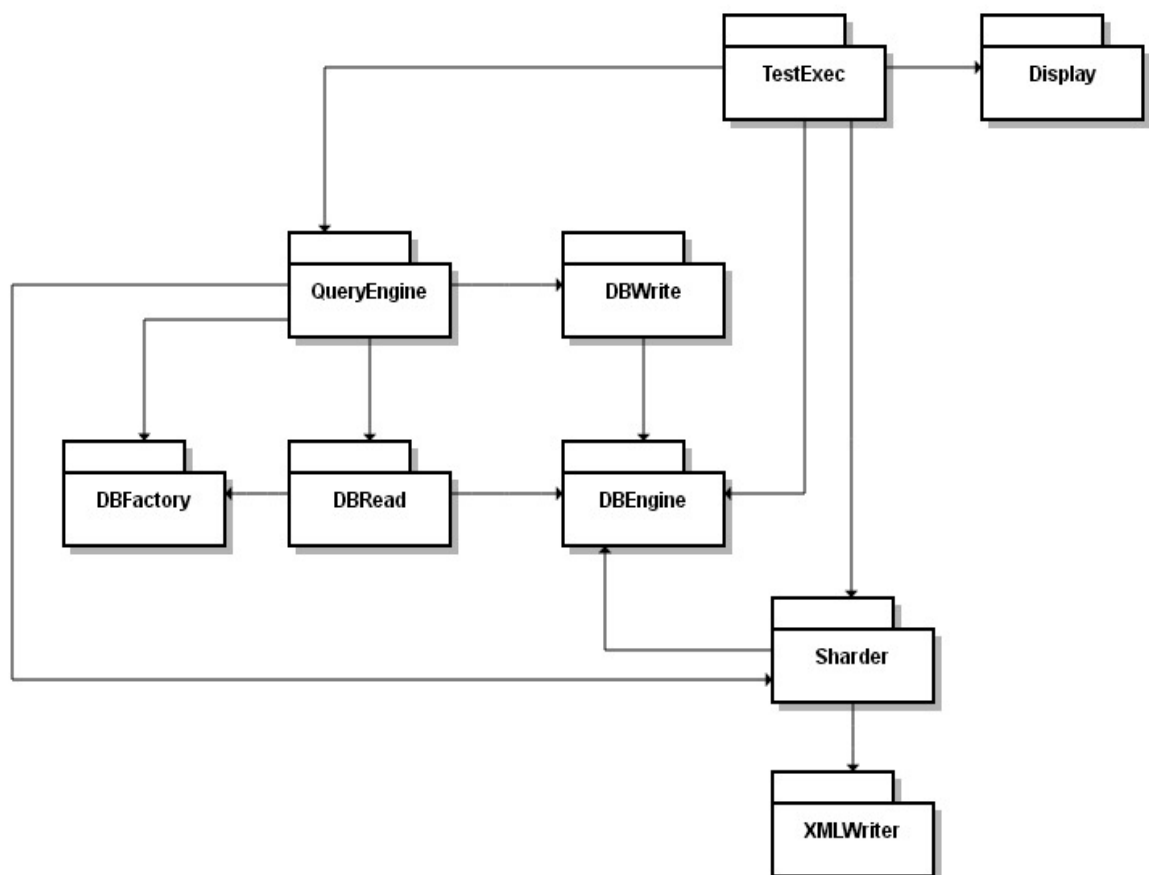


Figure 3

TestExec: This is the package that controls the program flow, and contains the main method. The series of tests that must be performed to demonstrate the database functionalities will be run by this package. It also acts as an entry point into the program. The TestExec package will be responsible for creating the database. We could think of it as a constructor for the database.

TestExec passes queries on to the QueryEngine package. The results returned by each query are passed on to the Display package. It also interacts with the Sharder package to implement periodic saving of the database to persistent memory.

DBEngine: This package contains the heart of the database. It holds the database instance and hence, all its key-value pairs that are loaded in-memory at runtime. It is created by the TestExec package.

DBEngine responds to read queries from the DBRead package. All queries that modify the database state are received from the DBWrite package. The Sharder package calls on the DBEngine to save its state to persistent memory.

QueryEngine: It is the responsibility of this package to parse the queries it receives from the TestExec. Depending on the type of query, it will decide on a course of further action. It is also responsible for processing compound queries.

The QueryEngine interacts with the DBRead package to perform all read queries. In addition to DBRead, it also communicates with the DBFactory package in case a compound query is received. All commands to change the database state are executed by interacting with the DBWrite package. Save commands are processed through the Sharder package.

DBRead: All read queries on the database are processed by the DBRead package. It is responsible for returning the set of keys that correspond to the query results, to the QueryEngine package.

DBRead receives parsed queries from the QueryEngine package. It executes the query by interacting with the DBEngine. In case of compound queries,

DBFactory: The DBFactory can be thought of as a database in and of itself. It is used to hold intermediate results while processing compound queries. This process has been explained in the subsequent section.

All the write operations on the DBFactory are performed by the QueryEngine package. The DBRead package executes each query segment of the compound query on the DBFactory.

DBWrite: All operations that involve modifying the state of the database are performed by the DBWrite package. This is responsible for providing the addition, deletion, modification and rollback functionalities.

The DBWrite package receives parsed write commands from the QueryEngine. Correspondingly, it performs the required modifications on the DBEngine.

Sharder: The Sharder package is responsible for implementing sharding. This is a principle used to save the entire database state into a set of smaller data collections, or shards.

This package can be called on by either the TestExec (to perform periodic saving) or by the QueryEngine (to perform saving on-command). The Sharder also interacts with the XMLWriter.

XMLWriter: The individual shards developed by the Sharder are converted to XML files for persistent storing by this package.

It is utilised by the Sharder.

Display: The Display package is responsible for printing all outputs to the console.

It is called on by the TestExec package.

KEY APPLICATION ACTIVITIES

The flow of the main program is fairly straightforward. The TestExec package will create the database, and instruct the QueryEngine to load it with values from an XML input file. Then it will proceed to demonstrate all the program functionalities one after another. A big part of this involves processing the various types of queries that the program is required to handle. Another important program activity is sharding, which plays a key role in saving the database state to persistent memory.

QUERY PROCESSING

This high-level activity is mainly handled by the QueryEngine package. It encompasses all the possible commands that could be sent to the application. Namely – read, add, edit, delete, rollback, augment and save.

The QueryEngine calls on other dependant packages depending upon the input it receives from the TestExec package. The activities to be performed will depend on the type of query. These can broadly be classified as:

1. WRITE TYPE ACTIVITIES

1.1 Perform appropriate database modification: All commands that change the state of the database (add, delete, edit, augment, rollback) will be performed by corresponding method calls to the DBWrite package.

1.2 Return acknowledgement of change: Once the requested modification has been performed, an acknowledgement indicating successful modification will be returned.

1.3 Display acknowledgement: A confirmation message will be printed to the console to indicate the successful completion of the operation to the user.

2. READ TYPE ACTIVITIES

Depending on whether the query is a simple query or a compound query, we have the following two possible sets of activities:

2.1 Simple query: The query is executed via the DBRead package, which returns the appropriate set of keys from the DBEngine. The results are passed to the TestExec package, which then prints the results to the user interface using the Display package.

2.2 Compound query: The compound query is broken down into a series of simple queries, or query segments. Each segment will be performed on the results of the preceding one. The results of the first segment will be taken from the DBEngine and will then be stored in the DBFactory. The second segment will be performed on the results of the first (the contents of DBFactory), and this new set is now stored in the DBFactory, overwriting the results of the previous segment. The third segment is then performed on the results of the second, and so on until the entire compound query has been executed. The final result will then be displayed to the user.

3. SAVE ACTIVITIES

When a save command is received, or when periodic saving has to be performed, the following two activities will take place:

3.1 Implement sharding: The contents of the database will be split into smaller files, or shards, for more convenient storing. Each shard will be sent from the Sharder package to the XMLWriter for further processing.

3.2 Writing to XML File: Each shard developed by the Sharder package will be written to an XML file for permanent storage onto the hard drive.

An activity diagram representing the entire Query Processing functionality is shown below in Figure 4.

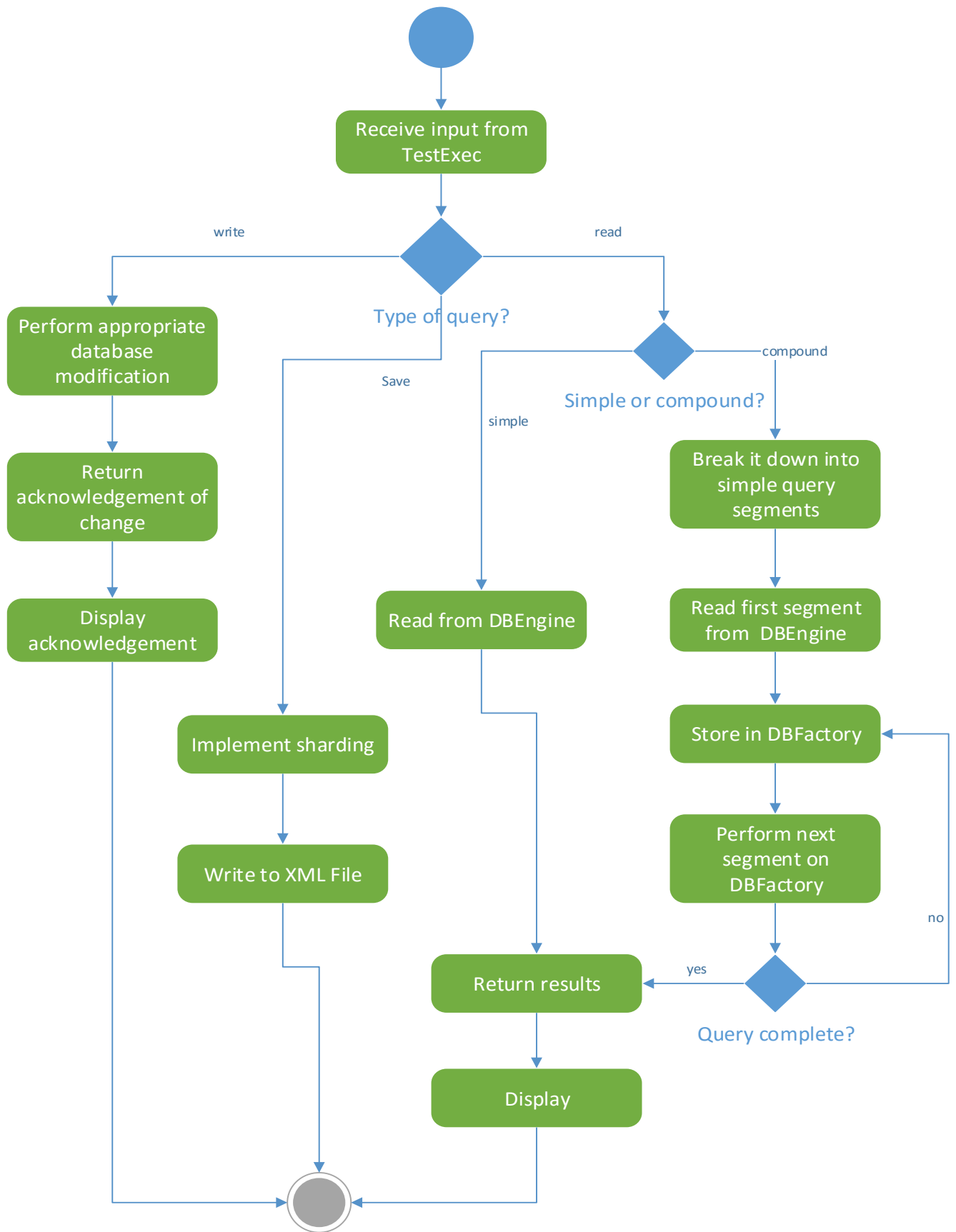


Figure 4

SHARDING

A high level activity diagram of the sharding process is shown below in Figure 5.

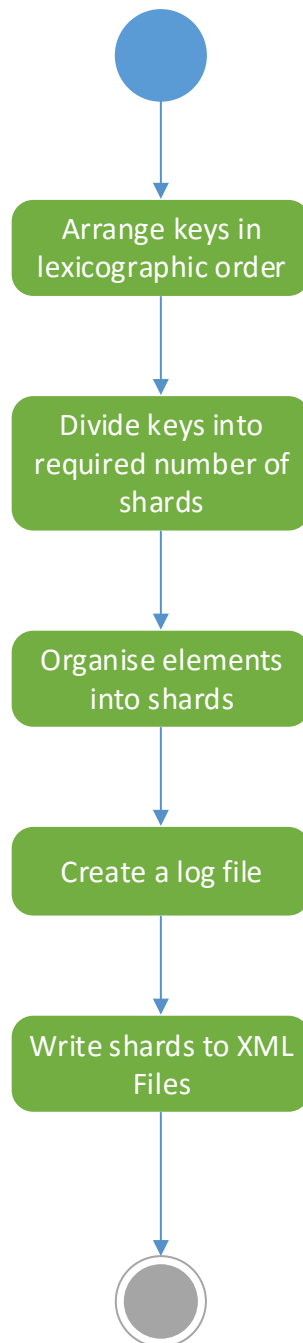


Figure 5

The various activities involved in this process are:

1. Arrange keys in lexicographic order: The entire set of keys stored in the database is arranged into lexicographic order. This serves to sort the keys into a logical order.

2. Divide keys into required number of shards: Now that the keys are in order, they can be divided equally into the required number of shards. A suitable number of shards can be taken, to provide optimum performance.

3. Organise elements into shards: The keys and their corresponding values will be assigned to their respective shards.

4. Create a log file: A log file containing data about the various keys held in each shard along with their corresponding metadata is created, for referencing purposes. This will be useful for looking up data contained in shards, for future queries.

5. Write to XML Files: All the shards will be written to XML files for permanent storage.

The method proposed above has one drawback: Look-up in the log file can only be performed by keys or metadata. Since the payload is not stored, direct look-up by payload is not possible. Hence, to search by payload, all the individual shards must be scanned for the specific payload value.

It is important to note that this is one possible method for sharding. Other methods can be used, depending on application suitability. For example, in applications involving frequent write operations, data can be sorted by its Time-Date stamp for easy look-up.

Ideally, sharding strategies should be tailored to suit specific applications.

CRITICAL ISSUES

1. Sharding: Improperly implemented sharding can cause serious issues like data replication and consequently, data inconsistency. The strategy to implement sharding which has been described above has a restriction; queries on payload cannot be supported. Also, many different strategies for sharding can be implemented, apart from the one suggested above. Each one has its own drawbacks and limitations.

Solution: The entire process of sharding should be carefully considered while designing, keeping in mind the possible drawbacks. Extensive testing should be carried out to ensure successful implementation.

2. Variable sharding strategies for applications involving multiple NoSQL databases with different key and value types: In applications that require multiple databases to handle different kinds of data sets (for example: data management application for a large collaborative system), one fixed sharding strategy for all the databases may prove to be inefficient.

Solution: The different usage information for each database should be carefully considered and sharding strategies suitable for each database should be implemented to optimise performance.

3. Query Processing: In applications involving multiple databases, the structure of each database and the type of data stored in them may be different. The set of queries developed for one type may not work well with another type.

Solution: Care should be taken to develop queries that are as flexible and generic as possible, while still being effective.

4. Incorporating ACID properties: Atomicity, Consistency, Isolation and Durability are properties that are not inherent in NoSQL databases. This could especially pose problems in applications that are run over networks, and service multiple users simultaneously.

Solution: These properties should be developed into the application by using a robust design.

5. Security in databases accessed over distributed systems: In applications run over networks, there is a high probability that multiple user segments will use the application. Security is a serious issue that needs to be addressed while developing such applications. Breaches of security could include damages to the system, loss or theft of data, compromise of data integrity, among other issues.

Solution: Authorization of users must be implemented. Multiple levels of access should be developed. Users should only be allowed to access as much functionality as they are authorised.

CONCLUSION

NoSQL databases have many useful advantages over relational databases that make them well suited for modern applications that require high performance and flexibility. However, there are critical issues and nuances that must be considered and accounted for carefully in order to develop useful, stable and robust applications.