*[University Logo]*

[University Name]

[Department]

## **[Course code]**: [Course name]

# [assignment]

### Report

|  |  |
| --- | --- |
| NAME: |  |
| ID: |  |
| Class/Section: |  |
| Semester: |  |

Instructor: **[Instructor name]**

### Date of Submission: **[date]**

Table of Contents:

# Introduction

## Background

## Objectives of the Report

## Overview of NoSQL Systems

# Data Models / Type of Schema

## Document Data Model (MongoDB)

### JSON Document Structures

### Schema Flexibility

## Key-Value Data Model (Berkeley DB)

### Key-Value Pair Organization

### Data Modeling Considerations

# Query Mechanisms

## MongoDB Query Language and Interfaces

### Query Syntax and Operators

### Code Examples for Queries

## Java/DPL API for Berkeley DB

### Code Examples for Data Operations

### Comparison of Query Expressiveness

# Management of NoSQL

## Security

## Indexing

## Concurrency Control

## Backup and Recovery

# Redesign of Training Centre Database

## Original Relational Schema

## Redesigned Data Models

### MongoDB Document Structures

### Berkeley DB Key-Value Organization

## Code Implementations

## Demonstration of Analogous Queries

# Advantages and Disadvantages of NoSQL

## Benefits of NoSQL

## Drawbacks of NoSQL

# Conclusion

## Key Findings and Insights

# INTRODUCTION**:**

## **BACKGROUND**

The term NoSQL originally referred to “non-SQL” or “non-relational” databases, but the term has since evolved to mean “not only SQL,” as NoSQL databases have expanded to include a wide range of different database architectures and data models.

With the advent of big data, we need to use NoSQL databases for storing and managing unstructured and semi-structured data because unlike traditional relational databases that use tables with pre-defined schemas to store data, NoSQL databases use flexible data models that can adapt to changes in data structures and are capable of scaling horizontally to handle growing amounts of data. This data is modeled in means other than the tabular relations used in relational databases. Such databases came into existence in the late **1960s,** but did not obtain the NoSQL moniker until a surge of popularity in the early **twenty-first century**. NoSQL databases are used in real-time web applications and big data and their use are increasing over time.

## **Objectives of the Report**

This report aims to research the methodology and technology *of* document stores and key-value stores by considering a re-design *of* the existing training Centre database so that it may operate using MongoDB and Berkeley DB respectively. Additionally, it will also discuss MongoDB and Berkeley DB data models, query mechanisms, security mechanisms, indexing and the tools for designing them in details and their comparison.

## **Overview of NoSQL Systems**

NoSQL, also referred to as “not only SQL”, “non-SQL”, is an approach to database design that enables the storage and querying of data outside the traditional structures found in relational databases. While it can still store data found within relational database management systems (RDBMS), it just stores it differently compared to an RDBMS. The decision to use a relational database versus a non-relational database is largely contextual, and it varies depending on the use case.

NoSQL databases, house data within one data structure, such as **JSON document**. Since this non-relational database design does not require a schema, it offers rapid scalability to manage large and typically unstructured data sets.

NoSQL is also type of distributed database, which means that information is copied and stored on various servers, which can be remote or local. This ensures availability and reliability of data. If some of the data goes offline, the rest of the database can continue to run

# Data Models / Type of Schema

## Document Data Model (**MongoDB**)

A document database (also known as a document-oriented database or a document store) is a database that stores information in documents



Document databases are considered to be non-relational (or NoSQL) databases. Instead of storing data in fixed rows and columns, document databases use flexible documents. Document databases are the most popular alternative to tabular, relational databases such as in JSON format. In JSON format, data is stored in the form of objects instead of tables as displayed in above picture.

A document is a record in a document database. A document typically stores information about one object and any of its related metadata.

Documents store data in field-value pairs. The values can be a variety of types and structures, including strings, numbers, dates, arrays, or objects. Documents can also be stored in formats like BSON and XML.

**Collections:**

A collection is a group of documents. Collections typically store documents that have similar contents.

Not all documents in a collection are required to have the same fields, because document databases have a flexible schema. Note that some document databases provide schema validation, so the schema can optionally be locked down when needed.

Example:

The document below that stores information about Donna could be added to the users collection.



**CRUD operations**

Document databases typically have an API or query language that allows developers to execute the CRUD (create, read, update, and delete) operations.

* **Create**: Documents can be created in the database. Each document has a unique identifier.
* **Read**: Documents can be read from the database. The API or query language allows developers to query for documents using their unique identifiers or field values. Indexes can be added to the database in order to increase read performance.
* **Update**: Existing documents can be updated either in whole or in part.
* **Delete**: Documents can be deleted from the database.

**Key Features of Document Databases:**

* Document databases have the following key features:
* Document model: Data is stored in documents (unlike other databases that store data in structures like tables or graphs). Documents map to objects in most popular programming languages, which allows developers to rapidly develop their applications.
* Flexible schema: Document databases have a flexible schema, meaning that not all documents in a collection need to have the same fields. Note that some document databases support schema validation, so the schema can be optionally locked down.
* Distributed and resilient: Document databases are distributed, which allows for horizontal scaling (typically cheaper than vertical scaling) and data distribution. Document databases provide resiliency through replication.
* Querying through an API or query language: Document databases have an API or query language that allows developers to execute the CRUD operations on the database. Developers have the ability to query for documents based on unique identifiers or field values.
  + 1. **JSON DOCUMENT STRUCTURES**
* A **JSON document**, at its core, is a structured representation of data objects. It makes use of attribute-value pairs and arrays to store information in an organized manner. The structure of a JSON document enables it to be easily read by both humans and machines, making it a preferred choice for data storage and exchange.
* The **structure of a JSON** document is simple yet powerful. It consists of attribute-value pairs and arrays, which essentially act as the building blocks for data representation. An attribute-value pair, in the context of JSON documents, comprises a key (attribute) and its corresponding value. This could be anything – numbers, strings, Boolean or even other JSON objects.
* JSON documents are **frequently used** in various software development and data interchange scenarios. They function as a bridge between servers and web applications, enabling efficient data transmission that can be easily interpreted and utilized.

**Example:**

{

"user\_id": 101,

"name": "Jane Doe",

"email": "jane.doe@example.com",

"active": true,

"roles": ["user", "admin"],

"last\_login": {

"date": "2023-12-31T13:45:30Z",

"ip": "192.168.1.25"

},

"preferences": {

"language": "English",

"notifications": {

"email": true,

"sms": false

}

}

}

The above example stores the data of user in JSON structure and also uses nested object for preferences.

* + 1. **SCHEMA FLEXIBILITY**

Document databases come with dynamic and self-describing schemas (implementation of a data model in a specific database) that offer you the flexibility to have documents with different fields in a collection. This ability to accommodate varying fields across documents eliminates the need for pre-defining schemas in a database.

When developers don’t have to pre-define schemas, they can easily modify structures without causing disruptions during schema migration. Some document databases come with a schema validation feature that allows you to enforce document structure rules and optionally lock down schemas.

## . Key-Value Data Model (Berkeley DB)

A key-value data model or database is also referred to as a key-value store. It is a non-relational type of database. In this, an associative array is used as a basic database in which an individual key is linked with just one value in a collection. For the values, keys are special identifiers. Any kind of entity can be valued. The collection of key-value pairs stored on separate records is called key-value databases and they do not have an already defined structure.

**How do key-value databases work?**

A number of easy strings or even a complicated entity is referred to as a value that is associated with a key by a key-value database, which is utilized to monitor the entity. Like in many programming paradigms, a key-value database resembles a map object or array, or dictionary, however, which is put away in a tenacious manner and controlled by a DBMS. An efficient and compact structure of the index is used by the key-value store to have the option to rapidly and dependably find value using its key.

**For example:**

Redis is a key-value store used to tracklists, maps, heaps, and primitive types (which are simple data structures) in a constant database. Redis can uncover a very basic point of interaction to query and manipulate value types, just by supporting a predetermined number of value types and when arranged, is prepared to do high throughput.

**Features:**

* One of the most un-complex kinds of NoSQL data models.
* For storing, getting, and removing data, key-value databases utilize simple functions.
* Querying language is not present in key-value databases.
* Built-in redundancy makes this database more reliable.

**2.2.1. KEY- VALUE PAIR ORGANIZATION**

* **Creating Key-Value Pairs**: Creating key-value pairs in JavaScript is straightforward. In objects, you define keys and associate them with values using colons. In arrays, each element has an index that can be considered a key, with the corresponding value**.**

// Creating key-value pairs in an object

const person = {

name: "John",

age: 30,

city: "New York"

};

// Creating key-value pairs in an array

const fruits = ["apple", "banana", "cherry"];

* **Accessing Values**: To access values associated with specific keys in JavaScript objects, you use dot notation or bracket notation.

// Accessing values using dot notation

console.log(person.name); // Output: "John"

// Accessing values using bracket notation

console.log(person["age"]); // Output: 30

This how key-value pairs are organized.

**2.2.2. DATA MODELING CONSIDERATIONS**

There are several key considerations in Berkeley DB. These are the following:

* It is primarily a key value store which means that data was stored in key value pairs that was useful in indexing and cashing, etc.
* Berkeley DB is also the schema less architecture just like MongoDB. It does not enforces any predefined schema but each key value pair have different schema which provide the flexibility to the application and also ensures data consistency.
* It also supports different data types.
* It also implements the error handling methods that ensures that your application can respond appropriately in failures.
* It also supports the creation of secondary indexes for efficient retrieval of data that will be based on primary indexes.
* Berkeley DB ensures data consistency as it provides proper mechanisms for data concurrency control as it is very important for an application to run properly.

# Query Mechanisms

## MongoDB Query Language and Interfaces

* **MongoDB Query Language:**

MQL, short for MongoDB Query Language, is Mongo DB’s proprietary language designed for data retrieval. It empowers users to fetch documents that align with precise criteria, execute aggregations, make updates to documents, and perform document deletions. Mongo DB’s decision to develop its own query language is a result of their commitment to tailoring querying functionality to their unique requirements and the diverse needs of their customers.

* **Interfaces:**

There are several interfaces to interact with mongo DB query language. These are as following:

MongoDB Shell

MongoDB Compass

MongoDB Shell

MongoDB Atlas

MongoDB Drivers

**3.1.1. QUERY SYNTAX AND OPERATORS AND CODE EXAMPLES**

* **Operators**
* **Comparison**

The following operators can be used in queries to compare values:

$eq: Values are equal

$ne: Values are not equal

$gt: Value is greater than another value

$gte: Value is greater than or equal to another value

$lt: Value is less than another value

$lte: Value is less than or equal to another value

$in: Value is matched within an array

* **Logical**

The following operators can logically compare multiple queries.

$and: Returns documents where both queries match

$or: Returns documents where either query matches

$nor: Returns documents where both queries fail to match

$not: Returns documents where the query does not match

* Evaluation

The following operators assist in evaluating documents.

$regex: Allows the use of regular expressions when evaluating field values

$text: Performs a text search

$where: Uses a JavaScript expression to match documents

* **Fields**

The following operators can be used to update fields:

$currentDate: Sets the field value to the current date

$inc: Increments the field value

$rename: Renames the field

$set: Sets the value of a field

$unset: Removes the field from the document

* **Array**

The following operators assist with updating arrays.

$addToSet: Adds distinct elements to an array

$pop: Removes the first or last element of an array

$pull: Removes all elements from an array that match the query

$push: Adds an element to an array

* **Query Syntax And Examples:**
* **Create Collection:**

We can create collection by using **createCollection()** or during **insert** process.

* + Example:

db.createCollection("posts")

db.posts.insertOne(object)

* **Insert:**

To insert a single document, insertOne() is used.

* + Example:

db.posts.insertOne({

title: "Post Title 1",

body: "Body of post.",

category: "News",

likes: 1,

tags: ["news", "events"],

date: Date()

})

* **Find:**

To select data from a collection in MongoDB, we can use the **find()** method. This method accepts a query object. If left empty, all documents will be returned and To select only one document, we can use the **findOne()** method. This method accepts a query object. If left empty, it will return the first document it finds.

* + Example:

db.posts.find()

db.posts.findOne()

db.posts.find( {category: "News"} )

* **Update:**

To update an existing document we can use the **updateOne**() or **updateMany**() methods.

* + Example:

db.posts.updateOne(

{ title: "Post Title 5" },

{

$set:

{

title: "Post Title 5",

body: "Body of post.",

category: "Event",

likes: 5,

tags: ["news", "events"],

date: Date()

}

},

{ upsert: true }

)

* **Delete:**

We can delete documents by using the methods **deleteOne**() or **deleteMany**(). These methods accept a query object. The matching documents will be deleted.

* + Example:

db.posts.deleteOne({ title: "Post Title 5" })

db.posts.deleteMany({ category: "Technology" })

## Java/DPL API for Berkeley DB

**Data Storage**

Berkeley DB Java Edition stores data quickly and easily without much of the overhead found in other databases. Berkeley DB Java Edition is a single JAR file that runs in the same JVM as your application, so there is no remote server. A local cache keeps the most active data in memory, avoiding costly disk access, and bounds the usage of JVM memory to a predictable amount.

* Local, in-process data storage
* Schema-neutral, application native data storage
* Keyed and sequential data retrieval
* Easy-to-use Java Collections API
* Direct Persistence Layer (DPL) for accessing Java objects
* Schema evolution of DPL classes
* Single process, multi-threading model
* Record level locking for high concurrency
* Support for secondary indexes
* In-memory, on disk or both
* Configurable background cleaner threads re-organize data and optimize disk use

**Transactions**

Berkeley DB Java Edition stores data reliably and ensures data integrity. In the event of a system failure, Berkeley DB Java Edition will recover transactional data and reset the system to a functional and consistent state from log and database information.

* Full ACID compliance
* Selectable isolation levels and durability guarantees, configurable on a per-transaction basis
* Managed transactions using the Java Transaction API (JTA)
* J2EE application server integration using J2EE Connector Architecture(JCA)
* Auditing, monitoring, and administration using the Java Management Extensions (JMX)
* Catastrophic and routine failure recovery modes
* Timeout based deadlock detection
* Hot and cold backups, log file compaction, and full database dumps

**Deployment**

* Berkeley DB Java Edition is highly portable, very flexible and easy to integrate. It was designed from day one as a pure Java product taking full advantage of the Java environment. As a single Java Archive (JAR) file, it runs within the JVM running your application. Berkeley DB Java Edition was designed to serve the large and growing Java community with enterprise-grade, pure Java, data storage solution.
* 100% pure Java for portability and ease of development
* Single JAR file - easy to install, runs in the same JVM as the application
* Java 1.5 or later Standard Edition JVM required
* Programmatic administration and management
* Zero human administration
* API for routine administrative functions
* Small footprint 820KB
* Scalable to terabytes of data, millions of records
* Source code, test suite included

**3.2.1. CODE EXAMPLES FOR DATA OPERATIONS**

* **Example 1:**

Following code is the example of how we can use java/DPL(Direct Persistence Layer) API for Berkeley DB. First of all, we will define the environment and database configurations. Then open database connection and insert and read key value pairs in the database and then finally close the connection.

import com.sleepycat.je.\*;

public class BerkeleyDBExample {

public static void main(String[] args) {

// Specify the environment and database location

String envHome = "./myEnv";

String dbName = "myDatabase";

EnvironmentConfig envConfig = new EnvironmentConfig();

envConfig.setAllowCreate(true);

// Open the environment

Environment myEnv = new Environment(envHome, envConfig);

// Specify database configuration

DatabaseConfig dbConfig = new DatabaseConfig();

dbConfig.setAllowCreate(true);

// Open the database

Database myDatabase = myEnv.openDatabase(null, dbName, dbConfig);

try {

// Insert data

String key = "name";

String value = "Herry";

DatabaseEntry keyEntry = new DatabaseEntry(key.getBytes("UTF-8"));

DatabaseEntry dataEntry = new DatabaseEntry(value.getBytes("UTF-8"));

myDatabase.put(null, keyEntry, dataEntry);

// Read data

DatabaseEntry retrievedData = new DatabaseEntry();

myDatabase.get(null, keyEntry, retrievedData, LockMode.DEFAULT);

String retrievedValue = new String(retrievedData.getData(), "UTF-8");

System.out.println("Retrieved value: " + retrievedValue);

} catch (Exception e) {

e.printStackTrace();

} finally {

// Close the database and environment

if (myDatabase != null) {

myDatabase.close();

}

if (myEnv != null) {

myEnv.close();

}

}

}

}

* **Example 2:**

The following example code deals all the CRUD operations in Berkeley DB.

import com.sleepycat.je.\*;

public class BerkeleyDB\_CRUD {

public static void main(String[] args) {

// Specify the environment and database location

String envHome = "./myEnv";

String dbName = "myDatabase";

EnvironmentConfig envConfig = new EnvironmentConfig();

envConfig.setAllowCreate(true);

// Open the environment

Environment myEnv = new Environment(envHome, envConfig);

// Specify database configuration

DatabaseConfig dbConfig = new DatabaseConfig();

dbConfig.setAllowCreate(true);

**// Open the database**

Database myDatabase = myEnv.openDatabase(null, dbName, dbConfig);

try {

**// Insert records**

insertData(myDatabase, "Name", "Herry");

insertData(myDatabase, "Phone", "2345");

insertData(myDatabase, "CGPA", "3.6");

**// Find and print a specific record**

findAndPrintRecord(myDatabase, "Phone");

**// Update a record**

updateData(myDatabase, "Phone", "3457");

// Find and print the updated record

findAndPrintRecord(myDatabase, "Phone");

**// Delete a record**

deleteData(myDatabase, "name");

// Find and print all remaining records

findAllRecords(myDatabase);

} catch (Exception e) {

e.printStackTrace();

} finally {

// Close the database and environment

if (myDatabase != null) {

myDatabase.close();

}

if (myEnv != null) {

myEnv.close();

}

}

}

private static void insertData(Database database, String key, String value) throws Exception {

DatabaseEntry keyEntry = new DatabaseEntry(key.getBytes("UTF-8"));

DatabaseEntry dataEntry = new DatabaseEntry(value.getBytes("UTF-8"));

database.put(null, keyEntry, dataEntry);

System.out.println("Inserted: Key - " + key + ", Value - " + value);

}

private static void findAndPrintRecord(Database database, String key) throws Exception {

DatabaseEntry keyEntry = new DatabaseEntry(key.getBytes("UTF-8"));

DatabaseEntry dataEntry = new DatabaseEntry();

if (database.get(null, keyEntry, dataEntry, LockMode.DEFAULT) == OperationStatus.SUCCESS) {

String value = new String(dataEntry.getData(), "UTF-8");

System.out.println("Found: Key - " + key + ", Value - " + value);

} else {

System.out.println("Record not found for key: " + key);

}

}

private static void updateData(Database database, String key, String updatedValue) throws Exception {

DatabaseEntry keyEntry = new DatabaseEntry(key.getBytes("UTF-8"));

DatabaseEntry dataEntry = new DatabaseEntry(updatedValue.getBytes("UTF-8"));

if (database.put(null, keyEntry, dataEntry) == OperationStatus.SUCCESS) {

System.out.println("Updated: Key - " + key + ", Value - " + updatedValue);

} else {

System.out.println("Update failed for key: " + key);

}

}

private static void deleteData(Database database, String key) throws Exception {

DatabaseEntry keyEntry = new DatabaseEntry(key.getBytes("UTF-8"));

if (database.delete(null, keyEntry) == OperationStatus.SUCCESS) {

System.out.println("Deleted record for key: " + key);

} else {

System.out.println("Deletion failed for key: " + key);

}

}

private static void findAllRecords(Database database) throws Exception {

try (Cursor cursor = database.openCursor(null, null)) {

DatabaseEntry keyEntry = new DatabaseEntry();

DatabaseEntry dataEntry = new DatabaseEntry();

while (cursor.getNext(keyEntry, dataEntry, LockMode.DEFAULT) == OperationStatus.SUCCESS) {

String key = new String(keyEntry.getData(), "UTF-8");

String value = new String(dataEntry.getData(), "UTF-8");

System.out.println("Key: " + key + ", Value: " + value);

}

}

}

}

**3.2.1. COMPARISON OF QUERY EXPRESSIVENESS**

* Berkeley DB with java edition with java API’s for data access unlike MongoDB.
* In Berkeley DB, queries are written programmatically DPL API in java either by using classes as mentioned in above example2.
* It also guarantee strong consistency in ACID transactions as data will be stored in the form of objects of classes.

# Management of NoSQL

## Security

NoSQL covers the following aspects to ensure security:

* **Authorization:**

For authorization, two techniques are used. The first one is **Role-Based Access Control (RBAC)** in which users are assigned with specific roles and permissions for regulating their access to certain data operations. It define roles and assign permissions to these roles on job responsibilities. The second one is **Fine-Grained Access Control** users have assigned with least privileges to perform their specific tasks to restrict access at document level.

* **Authentication:**

Authentication requires users to authenticate themselves before accessing the database such as login, etc. NoSQL use strong authentication mechanisms such as username, email, passwords. It often uses biometric authentication, security tokens for **Multi-Factor Authorization**.

* **Encryption:**

NoSQL store data at rest in encrypted form to prevent it from unauthorized user’s access. NoSQL prevents eavesdropping by using Transport Layer Security (TLS) and Secure Sockets Layer (SSL) to encrypt data during transmission between SQL database and the application.

## INDEXING

MongoDB uses indexing in order to make the query processing more efficient. If there is no indexing, then the MongoDB must scan every document in the collection and retrieve only those documents that match the query. Indexes are special data structures that stores some information related to the documents such that it becomes easy for MongoDB to find the right data file. The indexes are order by the value of the field specified in the index.

**Syntax:**

MongoDB provides a method called **createIndex**() that allows user to create an index.

db.COLLECTION\_NAME.createIndex({KEY:1})

**Example:**

db.mycol.createIndex({“age”:1})

{

“createdCollectionAutomatically” : false,

“numIndexesBefore” : 1,

“numIndexesAfter” : 2,

“ok” : 1

}

The **createIndex**() method also has a number of optional parameters.

These include:

* + background (Boolean)
  + unique (Boolean)
  + name (string)
  + sparse (Boolean)
  + expireAfterSeconds (integer)
  + hidden (Boolean)
  + storageEngine (Document)

**Drop index:**

In order to drop an index, MongoDB provides the **dropIndex**() method.

**Syntax:**

db.NAME\_OF\_COLLECTION.dropIndex({KEY:1})

The dropIndex() methods can only delete one index at a time. In order to delete (or drop) multiple indexes from the collection, MongoDB provides the **dropIndexes**() method that takes multiple indexes as its parameters.

db.NAME\_OF\_COLLECTION.dropIndexes({KEY1:1, KEY2: 1})

## CONCURRENCY CONTROL

Database transactions provide a safe and predictable programming model to deal with concurrent changes to the data. Traditional relational databases, like SQL Server, allow you to write the business logic using stored-procedures and/or triggers, send it to the server for execution directly within the database engine. With traditional relational databases, you are required to deal with two different programming languages the (non-transactional) application programming language such as JavaScript, Python, C#, Java, etc. and the transactional programming language (such as T-SQL) that is natively executed by the database

**Optimistic concurrency control:**

Optimistic concurrency control allows you to prevent lost updates and deletes. Concurrent, conflicting operations are subjected to the regular pessimistic locking of the database engine hosted by the logical partition that owns the item. When two concurrent operations attempt to update the latest version of an item within a logical partition, one of them will win and the other will fail. However, if one or two operations attempting to concurrently update the same item had previously read an older value of the item, the database doesn’t know if the previously read value by either or both the conflicting operations was indeed the latest value of the item. Fortunately, this situation can be detected with the **Optimistic Concurrency Control (OCC)** before letting the two operations enter the transaction boundary inside the database engine. **OCC** protects your data from accidentally overwriting changes that were made by others. It also prevents others from accidentally overwriting your own changes.

## Backup and Recovery

NoSQL databases are popular for their scalability, flexibility, and performance, but they also pose some challenges when it comes to backup and recovery. Unlike relational databases, NoSQL databases have different data models, architectures, and consistency levels, which affect how you can protect and restore your data in case of failure or disaster. Following are the steps to implement the backup and recovery strategies in NoSQL:

1. **Backup frequency and retention**
2. **Backup methods and formats**
3. **Backup tools and storage**
4. **Recovery methods and procedures**
5. **Recovery tools and testing**
6. **Backup and recovery best practices**

# Redesign of Training Centre Database

## Original Relational Schema

NoSQL databases do not have a schema in the same rigid way that relational databases have a schema. Each of the four main types of NoSQL database has an underlying structure that is used to store the data. But the details of how the data is organized are very flexible, sometimes even to the point of being called “schema-less,” which is actually misleading. A fundamental property of NoSQL databases is the need to optimize data access, which puts the focus on query patterns and business workflows. The first step is to establish business requirements, and work out the specific needs of the people who will use the application. The goal for schema design is to plan keys and indexes that are fast and effective for application queries and that complement workflow pattern.

## Redesigned Data Models

NoSQL databases are designed to break away from the rows and columns of the relational database model. But it’s a common mistake to think that NoSQL databases don’t have any sort of data model. A useful description of how the data will be organized is the beginning of a schema.

Relational databases have had generations of users and developers to work out standard design methods. Various formal tools exist for describing the relationships between the main objects in a business domain, and these formal descriptions can then be used to dictate how the data will be stored.

The same types of standard data modeling tools are not available for NoSQL data modeling. One recommendation is to begin with a business domain model expressed in a form that can be incorporated in an application, such as a JSON document.

**5.2.1.** **MONGODB DOCUMENT STRUCTURES**

**Different Types of Data Models**

The three types of data models that are typically classified as follows:

1. **Conceptual data model:**

Conceptual Data Models are rough sketches that provide the big picture, detailing where data/information from various business processes will be stored in the database system and the relationships they will be involved with. A conceptual data model typically includes the entity class, attributes, constraints, and the relationship between security and data integrity requirements.

1. **Logical data model:**

Logical data models provide more detailed, subjective information about data set relationships. At this stage, we can clearly connect what data types and relations are used. Logical data models are generally missed in rapid business contexts, having their utility in data-driven initiatives requiring important procedure execution.

1. **Physical data model:**

The schema/layout for data storage routines within a database is defined by the physical data model. A physical data model is a ready-to-implement plan that can be stored in a relational database.

* + 1. **BERKELEY DB KEY-VALUE ORGANIZATION**

Berkeley DB offers different methods for organizing the data. Some are discussed below:

* **Hash Tables:**  Supports key/value pairs and only page-level locking. It goes well with large scale data and predictable search and update.
* **BTree:**  Supports key/value pairs and only page-level locking. Allows to retrieve data with keys between a start and end value. This is not possible in hash tables.
* **Record-Number-based Storage:** Supports variable-length values. The key of the database will have the value of the number of the record.
* **Queues**: Supports fixed-length values and record-level locking. Deals with the data in the order that they are inserted.

## **Code Implementations**

Already discussed in section 3.1.1. and 3.2.1.

## **Demonstration of Analogous Queries**

Already discussed in section 3.1.1. and 3.2.1.

# **Advantages and Disadvantages of NoSQL**

## **Benefits of NoSQL**

Following are some key advantages of NoSQL:

* **High Performance:**

NoSQL ensures high read and write performance in parallel and distributed environment. It is very useful in real time analytics and in content delivery systems.

* **Scalability:**

As NoSQL databases are designed horizontally so, it makes it easier to handle high traffic loads and big data. It is very helpful in handle data and traffic loads on social media apps. NoSQL is also helpful for applications with dynamic workloads that require scaling depending upon the storage.

* **Flexible Schema:**

NoSQL offers flexible schema that helps developer to manage the unstructured and semi structured data.

* **Schema-less design:**

As we have discussed earlier that NoSQL is the schema less architecture unlike SQL so it allows developers to add new fields in the data without affecting or changing the existing one. It makes it helpful for the applications where data structure is dynamic.

* **Supports for large data sets:**

Data grows dynamically in NoSQL that makes it well suited for large data sets especially for big data.

* **Cost Effectiveness:**

NoSQL database can run on commodity hardware and scale horizontally so it is useful in projects with budget constraints.

## **Drawbacks of NoSQL**

Although NoSQL is very beneficial for most of the projects and applications with dynamic data but it also has some drawbacks:

* **Inconsistency:**

Sometimes NoSQL sacrifices the consistency to achieve high performance and then make it consistent eventually because of this it cannot be useful in financial systems like banking transactions where consistency is very important part.

* **Suitability:**

NoSQL database is not suitable for all types of data bases. Using it for unsuitable application often results in suboptimal performance.

* **Maturity Support:**

NoSQL databases are designed for specific applications so does not have predefined features that results community support and less features as compared to relational databases.

* **Learning NoSQL:**

For developing applications in NoSQL is challenging. Developers need to learn query languages and new concepts. It also has limited querying capability as compared to traditional databases**.** It is not well suited for the applications that rely on complex queries.

# **7.Conclusion**

## **7.1. Key Findings and Insights**

The massive increase of data and the existence of several distinct methods in which data is generated, accumulated, stored, and utilized has significantly changed with time. Additionally, the nature of data has changed throughout the years transforming from structured to more unstructured data. This brings about a need for efficient storage and management of such data which cannot be handled by traditional RDBMS methods. Hence, NoSQL databases have gained popularity and have become pivotal in database management. Finance is one domain where dynamic and large amounts of data are produced on a daily basis, thereby making NoSQL databases an ideal choice for data management. These papers compares these types of NoSQL databases based on certain metrics like data model, indexing methods, atomicity, integrity and several more and demonstrate implementation of three NoSQL databases namely, MongoDB, Cassandra and Redis, using financial data. Experiments were performed to compare the performance of the aforementioned databases when using fundamental READ queries to retrieve the complete dataset and complex READ queries to retrieve a specific section. Aggregation operations were also implemented on the data. Fundamental WRITE queries to load the entire dataset and complex WRITE queries to update particular parts of it were also performed.

References

1. INTRODUCTION
   1. BACKGROUND

<https://www.geeksforgeeks.org/introduction-to-nosql/>

* 1. OBJECTIVES OF THE REPORT
  2. OVERVIEW OF NOSQL SYSTEMS

<https://www.ibm.com/topics/nosql-databases>

1. DATA MODELS/ TYPES OF SCHEMA
   1. DOCUMENT DATBASES(MONGO DB)

<https://www.mongodb.com/document-databases>

* + 1. JSON DOCUMENT STRUCTURES

<https://www.sanity.io/glossary/json-document>

* + 1. SCHEME FLEXIBILITY

<https://www.g2.com/articles/document-databases>

* 1. KEY-VALUE DATA MODEL (BERKELEY DB)

<https://www.geeksforgeeks.org/key-value-data-model-in-nosql/>

* + 1. KEY-VALUE PAIR ORGANIZATION

<https://bootcamp.uxdesign.cc/understanding-key-value-%EF%B8%8Fpairs-and-object-entries-%EF%B8%8Ffor-mapping-data-in-javascript-54c988bb33df>

* + 1. DATA MODELING CONSIDERATIONS

1. QUERY MECHANISMS
   1. MongoDB Query Language and Interfaces

<https://www.knowi.com/blog/the-best-introduction-to-mongodb-query-language-mql/#:~:text=MQL%20is%20typically%20used%20in%20conjunction%20with%20programming,with%20the%20MongoDB%20database%20and%20execute%20MQL%20queries>.

* + 1. QUERY SYNTAX AND OPERATORS AND CODE EXAMPLES

<https://www.w3schools.com/mongodb/mongodb_query_operators.php>

<https://www.w3schools.com/mongodb/mongodb_mongosh_create_collection.php>

<https://www.w3schools.com/mongodb/mongodb_mongosh_insert.php>

<https://www.w3schools.com/mongodb/mongodb_mongosh_find.php>

<https://www.w3schools.com/mongodb/mongodb_mongosh_update.php>

<https://www.w3schools.com/mongodb/mongodb_mongosh_delete.php>

* 1. Java/DPL API for Berkeley DB

<https://www.oracle.com/partners/campaign/index-093405.html>

* + 1. CODE EXAMPLES FOR DATA OPERATIONS
    2. COMPARISON OF QUERY EXPRESSIVENESS

1. MANAGEMENT OF NOSQL
   1. Security
   2. Indexing

<https://www.geeksforgeeks.org/indexing-in-mongodb/>

* 1. CONCURRENCY CONTROL

<https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/database-transactions-optimistic-concurrency>

* 1. BACKUP AND RECOVERY

<https://www.linkedin.com/advice/3/how-do-you-implement-backup-recovery-strategies-1e>

1. REDESIGN OF TRAINING CENTRE DATABASE
   1. ORIGINAL RELATIONAL SCHEMA

<https://www.mongodb.com/nosql-explained/data-modeling>

* 1. REDESIGNED DATA MODELS

<https://www.mongodb.com/nosql-explained/data-modeling>

* + 1. MONGODB DOCUMENT STRUCTURES

<https://www.geeksforgeeks.org/mongodb-data-modelling/>

* + 1. BERKELEY DB KEY-VALUE ORGANIZATION

<https://cs.ulb.ac.be/public/_media/teaching/infoh415/student_projects/2019/berkeleydb.pdf>

* 1. CODE IMPLEMENTATIONS
  2. DEMONSTRATION OF ANALOGOUS QUERIES

1. ADVANTAGES AND DISADVANTAGES OF NOSQL
   1. BENEFITS OF NOSQL
   2. DRAWBACKS OF NOSQL
2. CONCLUSION

<https://www.sciencedirect.com/science/article/pii/S1877050922020737#abs0001>