MSCCS 1.4

M. SC. COMPUTER SCIENCE

SEMESTER - I

REVISED SYLLABUS AS PER NEP 2020

NoSQL TECHNOLOGIES

Programe Co-ordinator : **Mandar L. Bhanushe**
Head, Faculty of Science and Technology,
CODE, University of Mumbai – 400098.

Course Coordinator : **Sumedh Sejole**
Asst. Professor,
CDOE, University of Mumbai

Editor : **Dr. Shraddha Sable**
Asst. Professor
S. K. College of sci & comm, Nerul, Navi Mumbai

Course Writers : **Dr. Rajeshri Pravin Shinkar**
Assistant Professor,
SIES.

: **Rani Podichetty**
Assistant Professor,
K.B. College of. Arts and Commerce for Women.

: **Vijay Kothawade**
Assr. Professor,
SJRS College, Kalwa West.

: **Milind Thorat**
Lecturer,
KJSIT.

August 2024, Print I,

Published by
Director
Centre for Distance and Online Education, University of Mumbai,
Vidyanagari, Mumbai - 400 098.

DTP COMPOSED AND PRINTED BY
Mumbai University Press,
Vidyanagari, Santacruz (E), Mumbai - 400098.

CONTENTS

*****

ELECTIVES

Programme Name: M.Sc. Computer Science (Semester I) **Total Credits:** 02 **College assessment:** 25	**Course Name:** NoSQL Technologies **Total Marks:** 50 **University assessment:** 25

Prerequisite: Basic understanding of databases, SQL concepts, and familiarity with programming languages like Java or Python.

Course Outcome:

Upon the successful completion of this course, students will be able to:

- Understand NoSQL characteristics, storage types, and advantages/drawbacks.
- Interface and interact with MongoDB, Redis, HBase, and Apache Cassandra effectively.
- Comprehend storage architecture in NoSQL, including column-oriented, document stores, and key/value stores.
- Perform CRUD operations proficiently, including data creation, access, update, and deletion.
- Query NoSQL stores using MongoDB features, accessing HBase data, and querying Redis.
- Apply indexing and ordering concepts in NoSQL databases like MongoDB, CouchDB, and Cassandra.
- Manage transactions and ensure data integrity in NoSQL, including distributed ACID systems.
- Utilize NoSQL effectively in the cloud, such as Google App Engine Data Store and Amazon SimpleDB.

Course Code	Course Title	Total Credits
PSCS506a	**NoSQL Technologies**	**02**
MODULE - I **Unit 1: Introduction to NoSQL and Interfacing with NoSQL Data Stores** **Basics Introduction to NoSQL:** Characteristics of NoSQL, NoSQL Storage types, Advantages and Drawbacks, NoSQL Products Interfacing and interacting with NoSQL: Storing Data in and Accessing Data from MongoDB, Redis, HBase and Apache Cassandra, Language Bindings for NoSQL Data Stores **Understanding the storage architecture:** Working with ColumnOriented Databases, HBase Distributed Storage Architecture, Document Store Internals, Understanding Key/Value Stores in Memcached and Redis, Eventually Consistent Non-relational **Databases Performing CRUD operations:** Creating Records, Accessing Data, Updating and Deleting Data **Unit 2: Querying, Indexing, and Data Management in NoSQL Databases** **Querying NoSQL Stores:** Similarities Between SQL and MongoDB Query Features, Accessing Data from Column-Oriented Databases Like HBase, Querying		**02**

Redis Data Stores Indexing and Ordering Data Sets: Essential Concepts Behind a Database Index, Indexing and Ordering in MongoDB, ouchDB and Apache Cassandra **Managing Transactions and Data Integrity:** RDBMS and ACID, Distributed ACID Systems, Upholding CAP, Consistency Implementations Using NoSQL in The Cloud: Google App Engine Data Store, Amazon SimpleDB	

Text Books:

1. QL & NoSQL Databases, Andreas Meier · Michael Kaufmann, Springer Vieweg, 2019
2. Professional NoSQL by Shashank Tiwari, Wrox-John Wiley & Sons, Inc, 2011
3. SQL & NoSQL Databases, Andreas Meier · Michael Kaufmann, Springer Vieweg, 2019
4. NoSQL: Database for Storage and Retrieval of Data in Cloud, Ganesh Chandra Deka, CRC Press, 2017
5. Demystifying NoSQL by Seema Acharya, Wiley, 2020

*****

1

NoSQL TECHNOLOGIES

Unit Structure

1.0 OBJECTIVE

- To study the NoSQL characteristics, storage types and advantages/drawbacks.
- To study MongoDB, HBase, Apache Cassandra.
- To understand data storage architecture of NoSQL including column-oriented, document stores, and key/value pairs.

1.1 INTRODUCTION TO NOSQL AND INTERFACING WITH NOSQL DATA STORES

“NoSQL” is “nonSQL” or “not only SQL”, It stores the databases in the format other than traditional format of RDBMS like relational type of tables. It is useful for managing and accessing various types of databases for large volume of data.

Basics Introduction to NoSQL:

- A non-relational database which stores data in a non-tabular manner.
- NoSQL database can store data in traditional as well as non-traditional structural way.
- Relational Databases have been only one choice or the default choice for data storage.
- After relational databases, current excitement about NoSQL databases has come.
- The value of relational databases are for two areas of memory a. fast, small, volatile main memory b. Larger, slower, non - volatile backing store.
- As main memory is volatile to keep data around, for backing store (File system, Database).
- The database allows more flexibility than a file system in storing large amounts of data in a way that allows an application program to get information quickly and easily.
- A NoSQL database provides a mechanism for storage and retrieval of data that employs less constrained consistency models than traditional relational database.
- NoSQL systems are also referred to as “NotonlySQL” to emphasize that they do in fact allow SQL-like query languages to be used.

1.2 CHARACTERISTICS OF NOSQL

1. High Scalability:

NoSQL have higher scalability for the large database.

2. Independent of Schema:

NoSQL have more efficiency to work with the independent of schema feature i.e. large volume of heterogeneous type of data which requires no schemas for structuring it.

3. Complex with free working:

NoSQL is very easy to handle than the SQL databases, for storing data in an semi-structured, unstructured form that requires no tabular format or arrangement.

4. Flexible to accommodate:

NoSQL have heterogeneous data that does not require any the of structure format, they are very flexible in terms of their reliability and use.

1.3 NOSQL STORAGE TYPES

A database is an easily accessible collection of organised data or information kept in a computer system. A Database Management System often oversees a database (DBMS).

The nontabular data is stored in a non-relational database called NoSQL. NoSQL is an acronym for Not Only SQL. Document, key-value, column-oriented, and graphs are the primary types.

It is divided into four different types:

1. Document Database.

2. Key-Value Database.

3. Column-oriented Database.

4. Graph Database.

1. Document Database:

In document database, it stores the data in the form of document. The data is grouped into the specified files where it is useful for building any application software.

The most important benefit of document database is it allows to the use to store the database in a particular format i.e. document format.

It is hierarchical and semi-structured format of NoSQL database it allows efficient storage for the data. For example user profile it works very well for storing the data. MongoDb is very good example of NoSQL database.

2. Key -Value Database:

Key-Value database is a type of NoSQL database where it stores the data in a schema-less manner.

It store the data in key-value format. One data point is assign as a key while another data point is assign as value for key-value allotment.

Example of Key-Value is the term ‘age’ is assign as key data point while ‘45’ can be termed as value.

3. Column-oriented Database:

It stores the data in the form of columns where it segregates the data into homogenous categories.

User can access the data very easily without retrieving unnecessary information.

Column-oriented databases works efficiently for data analytics in many social media networking sites.

This type of databases can accommodate large volume of data, For filtering the data or information, column-oriented databases are used. Apache HBase is an example of column-oriented database.

4. Graph Database:

In Graph Database we can store the data in the form of graphical knowledge and its related element like nodes, edges etc.

Data points are placed very well so that nodes are easily related to the edges and thus, a connection or network can easily establish.

Graph-based databases focus on the relationship between the elements. It stores the data in the form of nodes in the database. The connections between the nodes are called links or relationships.

Key features of graph database:

- In a graph-based database, it is easy to identify the relationship between the data by using the links.
- The Query's output is real-time results.
- The speed depends upon the number of relationships among the database elements.
- Updating data is also easy, as adding a new node or edge to a graph database is a straightforward task that does not require significant schema changes.
- For software development Graph database is useful.
- Good example for NoSQL database is Amazon Neptune where it makes high effective and organized functioning of software. Amazon Neptune is reliable, fast and graph database service that runs or build various applications with highly connected databases.

1.4 ADVANTAGES OF NOSQL

- No constraint on the structure of the data to be stored.
- Integration with cloud computing.

- It can store large volume of data.
- Flexible data model.
- High performance.
- Open Source.

1.5 DRAWBACKS OF NOSQL

- Less developed as compared to traditional SQL.
- Improvements are required for cross-platform support.
- In NoSQL data inconsistency may occur.
- Large document size
- GUI is not available.
- It mainly designed for storage but it has very less functionality.
- Backup is one drawback of NoSQL database as some NoSQL databases like MongoDB, it has no approach for the backup of data in a consistent manner.

1.6 NOSQL PRODUCTS INTERFACING AND INTERACTING WITH NOSQL

- MongoDB is an open-source document-oriented database where it is designed to store a large volume of data and it allows user to work with the data efficiently. Storage and retrieval of data in MongoDB is not in the form of tables. It supports the languages like C, C++, C# and .Net, Java, Node.js, Perl, PHP, Python, Scala etc. User can easily create an application using any of these languages.
- Examples: There are many companies that uses MongoDB like Facebook, eBay, Google etc to store large volume of respective data.

1.7 SQL AND NOSQL

SQL	NoSQL
It is called as RDBMS or Relational Database.	It is called as Non-Relational or Distributed Database.
Table-based databases.	It can be document based, key-value pairs, graph databases.
Vertical Scalability.	Horizontal Scaliability.
Fixed or Predefined schema.	Flexible schema.
It is not suitable for hierarchical data storage.	It is suitable for hierarchical data storage.

Example: Oracle, Microsoft SQL server, MySQL, PostgreSQL etc.	Example: Document: MongoDB, CouchDB Key-value: Redis and DynamoDB Column based: Cassandra and Hbase Graph: Neo4j and Amazon Neptune.

1. 8 STORING DATA IN AND ACCESSING DATA FROM MONGODB

MongoDB is a cross platform document-oriented database it provides high performance and availability as well as easy scalability.

MongoDB works on the concept of collection and document.

Database: It is a physical container for the collections. Each database gets its own set of files on the file system. A single MongoDB server has multiple interfaces for databases.

Collection: It is a group of MongoDB document. It is an equivalent to relational database (RDBMS) table. A collection exists within a single database. Collections do not enforce a schema document. Within a collection can have different fields?

Document: It is a set of key-value pair. Document have dynamic schema. Dynamic schema is a document that has same collection, it does not require to have the same set of fields and structure and common fields in a collection document may hold different types of data.

1.9 SQL SERVER AND MONGODB

SQL Server	MongoDB
Database	Database
Table	Collection
Index	Index
Row	Document
Column	Field
Joining	Linking & Embedding

1.10 USING THE MONGODB SHELL

MongoDB shell is a great tool for navigation, inspection and for manipulation document data.

User can connect to MongoDB at localhost. Following are the shell commands.

To create a database

Use aj

To check in which database we are working with

Db

Display list of all databases

Show dbs

[here you will not able to see database aj which you have created as we have not yet stored anythong]

To insert data in a document

db.user.insert({name:"amol",age:30,address:"thane"})

to drop database

db.dropDatabase()   // will delete current database which is selected; here aj was used so it will remove aj and all data present in the same.

Create Collection in MongoDB

the data in MongoDB is stored in form of documents. These documents are stored in Collection and Collection is stored in Database.

Method 1: Creating the Collection in MongoDB on the fly

```
> db.aj.insert({
... id:1,
... name:"amol",
... sal:55000
... })
WriteResult({ "nInserted" : 1 })
```

To check whether the document is successfully inserted

db.aj.find()

to display the list of collections

show collections

Method 2: Creating collection with options before inserting the documents

db.createCollection("students")

the options that we can provide while creating a collection: **capped**: type: boolean.

This parameter takes only true and false. This specifies a cap on the max entries a collection can have. Once the collection reaches that limit, it starts overwriting old entries.

The point to note here is that when you set the capped option to true you also have to specify the size parameter.

size: type: number.

This specifies the max size of collection (capped collection) in bytes.

max: type: number.

This specifies the max number of documents a collection can hold.

autoIndexId: type: Boolean

The default value of this parameter is false. If you set it true then it automatically creates index field _id for each document.

```
> db.createCollection("teacher", { capped : true, size : 9232768} )
```

This command will create a collection named "teacher" with the max size of 9232768 bytes. Once this collection reaches that limit it will start overwriting old entries.

```
> db.createCollection("emp",{capped:true,size:44554444})
{ "ok" : 1 }
```

In order to remove collection

```
> db.emp.drop()
true
```

Insert Multiple Documents in collection

To insert multiple documents in collection, we define an array of documents and later we use the insert() method on the array variable as shown in the example below.

```
var test=
[
 {
 "sid" : 1,
 "sname" : "juhi",
 "mks" : 88
 },
 {
 "sid" : 2,
 "sname" : "kajol",
 "mks" : 87
 },
 {
 "sid" : 3,
 "sname" : "aamir",
 "mks" : 78
 }
 ];
db.student.insert(test);
```

to see the inserted records

```
db.student.find()
```

To print the data in JSON format

```
db.student.find().forEach(printjson)
```

or

```
db.student.find().pretty()
```

To fetch the data of “juhi” from students collection

```
> db.student.find({sname:"juhi"}).pretty()
```

To fetch the details of students having mks > 80

```
db.student.find({"mks":{$gt:80}}).pretty()
```

gt—greater than

ne—not equal

lt—less than

gte—greater than equals

lte—less than equals

to update details

(change kajol to raveena)

```
> db.student.update({"sname":"kajol"},{$set:{"sname":"raveena"}})
```

Updating Document using save() method

Here in order to use save() method we need to know unique id of each document.

If you do not use id then it will consider it as new document and that gets inserted in the collection.

To get the unique Id for a particular document…

db.student.find({"sname":"aamir"}).pretty()

will provide id for aamir

now lets change his name to salman using save method

db.student.save({"_id":ObjectId("5c318a49310a8bf25f38dc87"),"sid name":"salman", "mks":76})

To remove certain document from collection

db.student.remove({“sid”:3})

here we can have multiple documents containing the same information ; as id allocation is done by system uniqueness is maintained…

lets replicate multiple document code here. Create same variable test … and copy paste the code.

How to remove only one document matching your criteria?

When there are more than one documents present in collection that matches the criteria then all those documents will be deleted if you run the remove command. However there is a way to limit the deletion to only one document so that even if there are more documents matching the deletion criteria, only one document will be deleted.

db.student.remove({"mks":87},1)

here assume that there are 2 entries with mks =87; but we want to remove only 1 entry out of it so by writing 1 we are removing only 1 entry. (this parameter takes only Boolean value 0 or 1)

to display a particular column only

suppose that we need to display only sid

db.student.find({},{"_id":0,"sid":1})

Value 1 means show that field and 0 means do not show that field. When we set a field to 1 in Projection other fields are automatically set to 0, except _id, so to avoid the _id we need to specifically set it to 0 in projection. The vice versa is also true when we set few fields to 0, other fields set to 1 automatically.

db.student.find({},{"_id":0,"sname":0,"mks":0})

here we are keeping id , sname, mks as 0 means do not display those columns; rest columns which are not mentioned are treated /default as 1

```
> db.student.find({}, {"_id": 0, "sid" : 0, "mks" : 1})

Error: error: {
        "ok" : 0,
        "errmsg" : "Projection cannot have a mix of inclusion and exclusion.",
        "code" : 2,
        "codeName" : "BadValue"
}
>
```

Mixing of 0 and 1 is not allowed…

The limit() method in MongoDB

This method limits the number of documents returned in response to a particular query.

db.student.find({mks:{$gt:80}}).limit(1).pretty()

(It managed to get only one document, which is the first document that matched the given criteria.)

Here in the example there are 3 students with marks > 80; by using limit() method we get the 1st record that matches the criteria.

But instead of 1st record we want 2nd record then we can use skip() method.

MongoDB Skip() Method

```
db.student.find({mks:{$gt:80}}).limit(1).skip(1).pretty()
{
        "_id" : ObjectId("5c31947c310a8bf25f38dc88"),
        "sid" : 1,
        "sname" : "juhi",
        "mks" : 88
}
> db.student.find({mks:{$gt:80}}).limit(1).skip(2).pretty()
{
        "_id" : ObjectId("5c31947c310a8bf25f38dc89"),
        "sid" : 2,
        "sname" : "kajol",
        "mks" : 87
}
```

Sorting of records

```
db.student.find().sort({"sname":-1})
```

1.11 REDIS

It is Remote Dictionary Server. It is an open source.

It is used for NoSQL key-value storage for the primary purpose of an application cache or quick response of the database.

- Redis link resides outside ibm.com which stores data in memory, rather the our traditional storage like disk of SSD (solid-state drive).
- It helps in delivering the data with high speed, reliable and performance as well as high availability.
- It supports the multiple data structures.

1.12 HBASE

- HBase is a distributed column-oriented database built on top of the Hadoop file system. It is an open-source project and is horizontally scalable.
- HBase is a data model that is similar to Google's big table designed to provide quick random access to huge amounts of structured data. It leverages the fault tolerance provided by the Hadoop File System (HDFS).
- It is a part of the Hadoop ecosystem that provides random real-time read/write access to data in the Hadoop File System.
- One can store the data in HDFS either directly or through HBase. Data consumer reads/accesses the data in HDFS randomly using HBase. HBase sits on top of the Hadoop File System and provides read and write access.

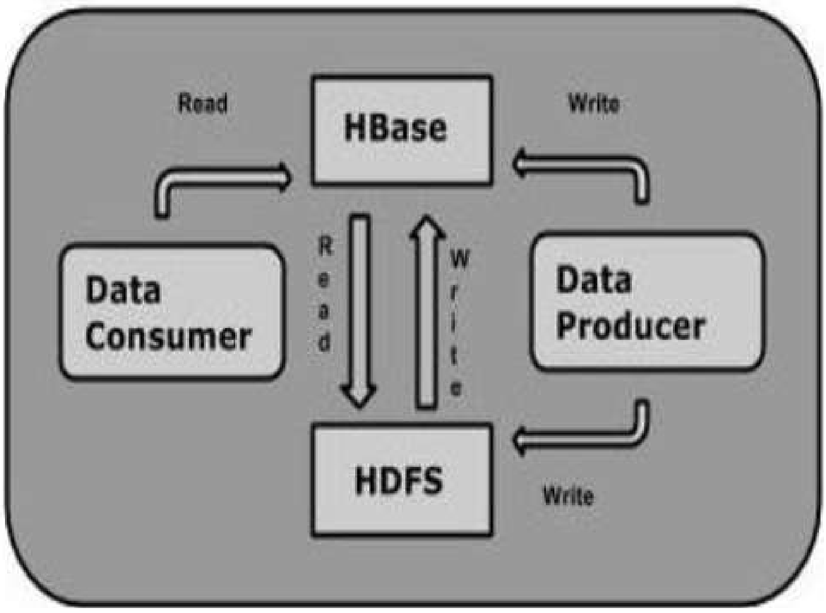


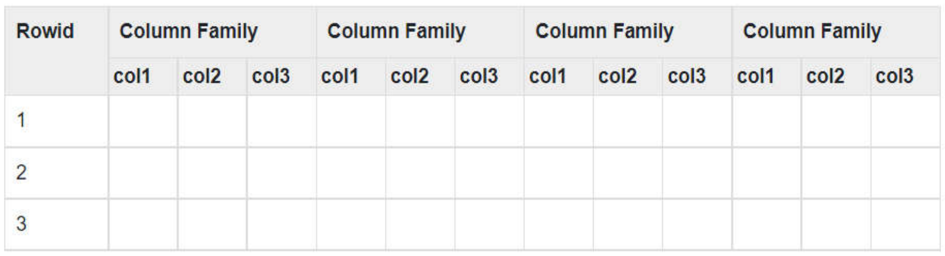
Fig: 1.12 HBase

1.12.1 Storage Mechanism in HBase:

- HBase is a column-oriented database and the tables in it are sorted by row. The table schema defines only column families, which are the key value pairs. A table have multiple column families and each column family can have any number of columns. Subsequent column values are stored contiguously on the disk. Each cell value of the table has a timestamp. In short, in an HBase:
- Table is a collection of rows.
- Row is a collection of column families.

- Column family is a collection of columns.
- Column is a collection of key value pairs.

Given below is an example schema of table in HBase.



Rowid	Column Family			Column Family			Column Family			Column Family		
	col1	col2	col3	col1	col2	col3	col1	col2	col3	col1	col2	col3
1												
2												
3												

1.12.2 HBase and RDBMS:

HBase	RDBMS
It is schema-less.	It is governed by schema.
It doesn't have the concept of fixed columns schema, it defines only column families.	It has structure of tables consist of rows and columns.
Built for wide tables.	It is useful for small tables.
It is horizontally scalable.	It is hard to scale.
No transactions are there in HBase.	RDBMS is transactional.
De-normalized data.	Normalized data.
It is good for semi-structured as well as structured data.	It is good for structured data.

1.12.3 Features of HBase:

- It is linearly scalable.
- It has automatic failure support.
- It provides consistent read and writes.
- It provides data replication across cluster.

1.12.4 Applications of HBase:

- It is used whenever there is a need to write heavy applications.
- HBase is used whenever we need to provide fast random access to available data.
- Companies such as Facebook, Twitter, Yahoo, and Adobe use HBase internally.

1.13 APACHE CASSANDRA

- Apache Cassandra is a open-source, free, distributed, column-oriented, NoSQL database management system used to designed for handling large volume of data on commodity servers.
- It provides high availability with no single point of failure.
- It offers robust support for clusters with multiple datacentres.

1.13.1 Data Model :

- **Cluster:** Cassandra database is a distributed across several machines which operate together. The outermost container is known as Cluster.
- In Cassandra for failure handling, every node contains a replica and in case of a node failure, the replica works very well.
- Cassandra arranges the nodes in a cluster in the form of ring and assigns the data to each node.
- **Key-space:** In Cassandra the outermost container of data. The basic attributes of a key-space in Cassandra are:
- **Replication factor:** It is the number of machines in the cluster.
- **Replica placement strategy:** To place the machines for replica place in the form of ring.
- **Column families:** Key-space is a container for a list of one or more column families. A column family is a container of a collection of rows. Each row contains ordered columns. Column families represent the structure of your data. It has at least one or more column families.
- **Column Families:** A column family is a container for an ordered collection of rows. Each row, in turn, is an ordered collection of columns.

1.13.2 Relational Table and Cassandra Column Family:

The following table lists the points that differentiate a column family from a table of relational databases.

Relational Table	Cassandra column Family
A Schema in a relational model is fixed. Once we define certain columns for a table, while inserting data, in every row all the columns must be filled at least with a null value.	In Cassandra, although the column families are defined, the columns are not. User can freely add any column to any column family at any time.
Relational tables define only columns and the user fills in the table with values.	In Cassandra, a table contains columns, or can be defined as a super column family.

1.14 LANGUAGE BINDINGS OR NOSQL DATA STORES

- A NoSQL is a database that provides a mechanism for storage and retrieval of data, they are used in real-time web applications and big data and their use are increasing over time.
- Many NoSQL stores compromise consistency in favor of availability, speed and partition tolerance.

When should NoSQL be used

- When huge amount of data need to be stored and retrieved.
- The relationship between data you store is not that important.
- The data changing over time and is not structured.
- Support of constraint and joins is not required at database level.
- The data is growing continuously and you need to scale the database regular to handle the data.
- In graph type of NoSQL databases the nodes are navigate as per the relationships using by language bindings.

1.15 SUMMARY

NoSQL database can store data in traditional as well as non-traditional structural way. Relational Databases have been only one choice or the default choice for data storage. After relational databases, current excitement about NoSQL databases has come. MongoDB is a cross platform document-oriented database it provides high performance and availability as well as easy scalability. MongoDB works on the concept of collection and document. HBase is a distributed column-oriented database built on top of the Hadoop file system. It is an open-source project and is horizontally scalable. HBase is a part of the Hadoop ecosystem that provides random real-time read/write access to data in the Hadoop File System. In Cassandra, a table contains columns, or can be defined as a super column family.

Example:

Document: MongoDB, CouchDB

Key-value: Redis and DynamoDB

Column based: Cassandra and Hbase

Graph: Neo4j and Amazon Neptune.

1.16 QUESTIONS

Q.1) Define NoSQL and Discuss in detail storage types of NoSQL

Q.2) Discuss about characteristics of NoSQL.

Q.3) Give Advantages and Drawbacks of NoSQL

Q.3) Describe MongoDB in detail.

Q.4) Compare

1. SQL and NoSQL.
2. SQL Server and MongoDB
3. HBase and RDBMS
4. Relational Table and Cassandra column Family.

Q.5) Define Collection and Document

Q.6) Elaborate the concept of HBase.

Q.7) Discuss features and applications of HBase.

Q.8) Explain Apache Cassandra.

1.17 REFERENCES

- AL & NoSQL Databases, Andreas Meier – Michael Kaufmann, Springer Vieweg, 2019
- Professional NoSQL by Shashank Tiwari, Wrox-John Wiley & Sons, Inc, 2011
- SQL & NoSQL Databases, Andreas Meier – Michael Kaufmann, Springer Vieweg- 2019
- NoSQL : Database for Storage and Retrieval of Data in Cloud, Ganesh Chandra Deka, CRC Press, 2017
- Demystifying NoSQL by Seema Acharya, Wiley, 2020

*****

2

UNDERSTANDING THE STORAGE ARCHITECTURE

Unit Structure

2.0 OBJECTIVES

The main goal of learning this unit is:

- To understand how data is stored, organized and retrieved.
- To understand the concepts like distributed systems, sharding, replication, and consistency models.
- To understand the data distribution strategies for designing and enhancing databases scalability, performance, and fault tolerance.

2.1 INTRODUCTION

In NoSQL technology, the storage architecture spilt from the traditional relational databases. It is design to handle or we can say manage large volumes of unstructured or semi-structured data making the data storage architecture more flexible and scalable. Instead of using immutable tables with redefined schemas, NoSQL databases commonly use key-value stores, document stores, wide-column stores, or graph databases. Let us have a overview on this technology.

2.2 OVERVIEW

NoSQL, or "Not Only SQL," refers to a broad class of database management systems (DBMS) that don't adhere strictly to the traditional relational database management system (RDBMS) model. While relational databases store data in tables with predefined schemas and support SQL queries, NoSQL databases offer more flexibility in terms of data models, scalability.

2.2.1 Understanding the storage architecture:

The storage architecture in NoSQL databases always call for distributed systems, where the data is spread at many multiple nodes to ensure scalability and fault tolerance. In this the each node in the cluster may store a fragment of the data, and replication techniques are used to maintain data consistency and availability.

In addition the NoSQL technology databases always prioritize horizontal scalability, allowing for easy expansion by adding more nodes to the cluster. This is achieved via sharding concepts in which the data is partitioned across multiple servers based on a certain criteria, such as key range or hashing algorithms.

Sharding concepts-it is a method in which the data records are stored at multiple servers' instances. It takes place through the concept of storage area networks in which it makes the hardware to perform like a single server. The NoSQL framework is basically deigned to support automatic distribution of the data across multiple servers which includes the query load.

2.2.2 What is storage architecture?

The storage architecture in NoSQL technology spins around the principles of flexibility, scalability, and performance to manage or handle large volumes of data which might be unstructured or semi-structured data whereas in traditional relational databases, which uses structured tables and schemas, the NoSQL databases engages various data models such as key-value, document, column-family, or graph-based.

The various concepts used are:

2.2.2. i: Key-Value Stores:

In this the databases stores the data as a collection of key-value pairs. In this each key is unique in nature and it is associated with a value which can be a simple string or a complex data structure. Examples: Redis and Amazon DynamoDB.

2.2.2. ii: Document Stores:

In this the data is stored as a document typically in JSON and BSON format, and it also allows nested and flexible schemas. This type of model is suitable for semi-structured data. MongoDB and Couchbase are the most popular document store databases.

2.2.2. iii: Column-family Stores:

In this the data is manage into columns grouped by the column families which helps in enabling efficient retrieval of the fragment of columns. This model is suitable for analytics and time-series data. Examples: Apache Cassandra and HBase.

2.2.2. iv: Graph Databases:

In this the databases represent the data as nodes, edges, and properties which helps in allowing for efficient querying of complex relationships.

Examples: Neo4j and Amazon Neptune.

2.2.3 Working with ColumnOriented Databases:

As working with column oriented databases in NoSQL technology it involves a unique architecture and features which helps to store and query data, especially in the scenarios where it requires analytics and aggregation are common tasks. Some of the concepts used for the interaction with column-oriented databases are:

2.2.3. i: Understanding the Column Model:

In column oriented databases it stores the data in columns rather than rows which means that all the values for a particular column are stored together, it allows for efficient and fast access to specific column during the query process.

2.2.3. ii: Data Modelling:

When you are working with a column-oriented database, we need to design the data model with a focus on column families or column groups rather than tables.

This process involves identifying the columns that are used frequently during the query process and organizing them into logical groups.

2.2.3. iii: Loading Data:

For executing any query data in column-oriented databases, we need to load it into the databases first. This process or stage involves importing the data from various sources, such files or other databases. For this type process NoSQL column-oriented databases provide tools or APIs.

2.2.3. iv: Querying Data:

While querying data in column-oriented databases, we typically write the queries which normally target the specific columns or groups of columns. This actually boosts the query performance especially when we are working on aggregations or analytics on large datasets.

Some of the column-oriented databases support SQL query languages, while others make use of APIs

2.2.3. v: Optimizing Performance:

In order to get the best performance out of a column-oriented database, you may need to optimize your data model, queries and indexing strategies. For this we need renormalizing the data so to reduce the need for joins, creating appropriate and tuning database configuration settings.

2.2.3.vi: Scaling Out:

The main goal for designing column-oriented databases was scale out horizontally across multiple nodes in a cluster. This plays a vital role in handling larger datasets and higher query loads by adding more nodes to the cluster.

2.3 HBASE DISTRIBUTED STORAGE ARCHITECTURE

HBase: it is a distributed, column-oriented database which is built on top of Hadoop Distributed File System (HDFS). This architecture consists of three main components that is shown with the help of a diagram given below figure.1

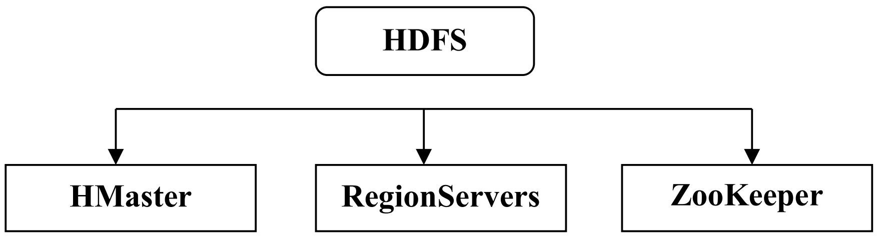


Figure. 1

2.3.1 HMaster:

HMaster: This component manages the metadata and coordinates the administrative operations. This is also responsible for coordinating and managing regions across the cluster, including assigning regions to RegionServers and handling failover.

2.3.2 RegionServers:

This component handle the data storage and processing. These are responsible for serving data to clients. Each RegionServers manages multiple regions, which are horizontal partitions of data stored in HDFS.

2.3.3 ZooKeeper:

This component assists in distributed coordination and management of the cluster. It helps in maintaining configuration information, providing synchronization, and handling failover for HBase. The HBase distributes the data across multiple RegionServers, by allowing for horizontal scalability and fault tolerance.

This model is actually inspired by Google's Bigtable, where data is stored in rows with columns, and rows can have a variable number of columns. It is highly scalable, fault tolerant, and suitable for real-time read/write operations on large database.

2.3.4 Applications of HBase Distributed Storage Architecture:

It is often used in the scenarios requiring real-time read/write access to large datasets such as follows:

2.3.4. i. Big Data Analytics:

HBase is utilized for storing and processing large volumes of data generated by various sources, enabling real-time analytics and insights.

2.3.4. ii. Time Series Data:

It's suitable for storing time-series data like logs, sensor data, financial data, etc., where data needs to be appended continuously and queried efficiently.

2.3.4. iii. Social Media Analytics:

HBase can store and process social media data such as user interactions, posts, comments, and likes, enabling real-time analysis for targeted advertising, sentiment analysis, etc.

2.3.4. iv. Internet of Things (IoT):

HBase can handle the massive volume of data generated by IoT devices, providing a scalable and reliable storage solution for sensor data, telemetry, and device logs.

2.3.4. v. Content Management Systems:

HBase can be used as a backend storage for content management systems handling large volumes of structured and unstructured data, providing high availability and scalability.

2.3.4. vi. Recommendation Systems:

HBase can store user profiles, preferences, and historical interactions, facilitating real-time recommendation generation for e-commerce platforms, streaming services, etc.

2.3.4. vii. Fraud Detection:

It's utilized for storing and analyzing transactional data in real-time to detect fraudulent activities and patterns, providing immediate alerts for proactive measures.

2.3.4. viii. Online Gaming:

HBase can store player profiles, game states, and interactions in online gaming platforms, enabling real-time updates and personalized gaming experiences.

2.4 DOCUMENT STORE INTERNALS

The Document Store Internals in NoSQL technology refers to the mechanisms and structures which is used to manage and data in a document-oriented database. In document-oriented databases, data is stored in flexible, semi-structured documents, typically in formats like JSON or BSON (Binary JSON).

Some of the key aspects of document store internals:

2.4.1 Document Structure:

Documents are the basic unit of data storage. Each document contains key-value pairs or key-array pairs where the values can be simple data types, arrays, or nested documents.

2.4.2 Indexing:

Document stores usually utilize indexes to efficiently query and retrieve data. Indexes can be created on various fields within the documents to optimize query performance.

2.4.3 Storage Engine:

The storage engine is responsible for managing the storage and retrieval of documents on disk. Different document stores may use different storage engines optimized for various use cases, such as memory-mapped storage, log-structured storage, or LSM (Log-Structured Merge-tree) storage.

2.4.4 Concurrency Control:

Document stores often implement concurrency control mechanisms to handle multiple concurrent read and write operations. This may involve techniques such as multi-version concurrency control (MVCC) to ensure consistency and isolation of transactions.

2.4.5 Replication and Sharing:

To achieve high availability and scalability, document stores typically support replication and sharding. Replication involves maintaining multiple copies of data across different nodes for fault tolerance, while sharding partitions data across multiple nodes to distribute the workload.

2.4.6 Query Processing:

Document stores provide query languages or APIs to query and manipulate data. Query processing involves parsing, optimizing, and executing queries efficiently, often leveraging indexes and other optimization techniques.

2.4.7 Durability and ACID Properties

Document stores may provide durability guarantees to ensure that committed data is not lost in the event of failures. They may also support ACID (Atomicity, Consistency, Isolation, Durability) properties to ensure data consistency and transactional integrity.

2.5 UNDERSTANDING KEY/VALUE STORES IN MEMCACHED AND REDIS

Understanding key/value stores in Memcached and Redis involves grasping the fundamental concepts and features of these popular NoSQL databases.

2.5.1 Memcached:

- Memcached is an in-memory key/value store primarily used for caching frequently accessed data to improve application performance.
- It operates as a distributed caching system, allowing multiple

instances to be deployed across a network.

- Data is stored in the form of key/value pairs, where keys are unique identifiers and values are arbitrary data.
- Memcached does not support persistence; data is stored only in memory and is lost when the server restarts or if it runs out of memory.
- It employs a simple protocol for client-server communication, making it lightweight and efficient.
- Memcached is often used in web applications to cache database query results, session data, and frequently accessed objects.

2.5.2 Redis:

- Redis is an in-memory data store that supports a wide range of data structures beyond key/value pairs, including strings, hashes, lists, sets, and sorted sets.
- Like Memcached, Redis operates primarily in memory, but it also offers optional persistence mechanisms for durability.
- Redis can be used as a caching solution, a message broker, a data structure server, and more, due to its rich set of features.
- It supports advanced data manipulation operations, such as atomic increments/decrements, range queries, and server-side scripting with Lua.
- Redis can be used as a caching solution, a message broker, a data structure server, and more, due to its rich set of features.
- It offers different persistence options, including snapshots (RDB) and append-only logs (AOF), allowing users to choose the level of durability they need.
- Redis is often used in scenarios requiring fast data access, real-time analytics, and pub/sub messaging, and distributed locking.

2.5.3 Key -Value Stores:

In this the databases stores the data as a collection of key-value pairs. In this each key is unique in nature and it is associated with a value which can be a simple string or a complex data structure. Examples: Redis and Amazon DynamoDB.

In summary, while both Memcached and Redis are key/value stores used for caching and fast data access, Redis offers a broader range of features and data structures, making it suitable for a wider variety of use cases beyond simple caching. Understanding the strengths and limitations of each system is crucial for selecting the appropriate solution for specific application requirements.

2.6 EVENTUALLY CONSISTENT NON-RELATIONAL

Eventually consistent" refers to a consistency model used in distributed systems, including many NoSQL databases. In an eventually consistent system, updates to data will propagate through the system asynchronously, and eventually, all replicas of the data will converge to the same state. This model prioritizes availability and partition tolerance over strict consistency at all times.

In the context of non-relational (NoSQL) databases, eventually consistent systems often use replication and distribution techniques to achieve scalability and fault tolerance. Here's how it typically works:

1. Replication:

Data is replicated across multiple nodes in the database cluster to ensure fault tolerance and high availability. When a write operation occurs, it is propagated to all replicas asynchronously.

2. Consistently Models:

a. Read your Writes Consistency:

In many eventually consistent systems, a client reading data after performing a write operation will always see its own writes (read-your-writes consistency).

b. Monotonic Reads and Writes:

Guarantees that if a client reads a particular version of a data item, any subsequent reads will not return an older version.

c. Monotonic Writes:

Guarantees that if a client writes a sequence of data values to a data item, those writes will be observed in the same order by all replicas.

d. Causal Consistency:

Ensures that if one operation causally precedes another, all replicas will see them in the same order.

3. Conflict Resolution:

In an eventually consistent system, conflicts may arise when concurrent writes occur to the same data item on different replicas. Conflict resolution strategies vary depending on the database system and may involve techniques such as last-write-wins, vector clocks, or application-level conflict resolution.

Examples of NoSQL databases that implement an eventually consistent model include:

1) Amazon DynamoDB:

Offers configurable consistency levels, including eventual consistency and strong consistency.

2) Cassandra:

Provides tunable consistency, allowing users to choose between eventual consistency and various levels of strong consistency.

3) Riak:

Implements eventual consistency with support for conflict resolution strategies.

2.7 SUMMARY

NoSQL (Not Only SQL) is a broad term used to describe a category of database management systems that differ from traditional relational databases in their data models, scalability, and flexibility. Here's a summary of key points about NoSQL technology:

1. Flexible Data Models:

Unlike relational databases, which use structured schemas and tables, NoSQL databases can handle semi-structured and unstructured data. They are well-suited for applications with rapidly changing data requirements.

2. Scalability:

NoSQL databases are designed to scale horizontally across multiple servers, making them suitable for large-scale distributed systems. They can handle high volumes of data and traffic more efficiently than traditional databases.

3. High Performance:

Many NoSQL databases are optimized for specific use cases, such as real-time analytics, content management, or caching. They often offer high-performance features like in-memory processing, asynchronous replication, and automatic sharding.

4. Types of NoSQL Databases:

NoSQL databases are categorized into four main types:

a. **Document-Oriented Databases:**

Store data in flexible, semi-structured documents (e.g, MongoDB, Couchbase).

b. Key-value stores:

Store data as key-value pairs, providing fast retrieval based on keys (e.g., Redis, Amazon DynamoDB).

c. Column-Family Stores:

Store data in columns rather than rows, suitable for large-scale distributed systems (e.g., Apache Cassandra, HBase).

d. Graph Databses:

Optimize for managing relationships between data entities (e.g., Neo4j, Amazon Neptune).

e. Challenges:

While NoSQL databases offer advantages in scalability and flexibility, they also pose challenges such as data consistency, lack of standardization, and potentially steep learning curves for developers accustomed to relational databases.

2.8 REFERENCES AND BIBLIOGRAPHY

- https://chatgpt.com/
- https://www.google.co.in/
- Youtube

2.9 QUESTIONS FOR PRACTICE

Q1. What is NoSQL?

Q2. What is NoSQL used for?

Q3. Explain HBase

Q4. Explain the use of Hadoop

*****

3

DATABASES PERFORMING CRUD OPERATIONS

Unit Structure

3.0 OBJECTIVES

- Detailing the operations of create, read, update, and delete within the context of data sets in a NoSQL database.
- Elaborating on the significance placed on create operations compared to updates, accompanied by illustrative examples.
- Investigating the atomicity and integrity aspects concerning updates in NoSQL databases.
- Clarifying the methods utilized for persisting interconnected data in NoSQL databases.

3.1 INTRODUCTION

CRUD operations—Create, Read, Update, and Delete—are essential for interacting with data in any database. These operations are particularly significant in the realm of NoSQL databases, which encompass a diverse array of database types rather than a single product or technology.

NoSQL databases vary in how they implement CRUD operations, largely based on their structure, whether they are document stores, key-value stores, or column-oriented databases. A common characteristic among NoSQL databases is the emphasis on create and read operations over update and delete operations. Sometimes, only create and read operations are supported.

The upcoming sections will explore how CRUD operations are implemented in NoSQL databases, focusing on creating records. The discussion will be structured around different NoSQL database types: column-oriented, document-centric, and key-value stores.

3.2 CREATING RECORDS

Creating a record involves saving a new entry in the database. It is crucial to have a unique identifier to distinguish each record and ensure that it does not already exist in the database. In relational databases, this identifier is known as the primary key, which uniquely identifies each record in a table. If a primary key already exists, the record should be updated rather than recreated.

Relational databases use normalization principles, introduced by E.F. Codd and refined into the Boyce-Codd Normal Form (BCNF). These principles aim to minimize data redundancy and ensure data integrity by organizing data so that each piece of information is stored only once and referenced as needed.

In a normalized relational database schema, two records with identical values are considered the same, enforced through primary keys. In object-oriented programming languages, this concept is often replaced by reference-based identification, where a unique record is identified by its memory address.

NoSQL databases, which may resemble traditional tables or object stores, use either value-based or reference-based semantics for record identification. Despite these differences, the concept of a unique primary key remains important across all types of databases.

Many databases provide tools for generating primary keys to ensure their uniqueness and relevance. For example, MongoDB uses a 12-byte BSON object ID as the default primary key, which includes a timestamp, machine ID, process ID, and a counter to ensure uniqueness.

Similarly, HBase, a column-oriented database, uses row keys that are byte arrays. These keys should be logically meaningful for the application and are ordered by their byte sequence, affecting how data is accessed and stored.

In this section, we will cover how to create records in specific NoSQL databases, using MongoDB for document stores, HBase for column-oriented databases, and Redis for key-value stores.

MongoDB:

MongoDB, a document-centric database, uses BSON object IDs for record identification. When creating a new document, MongoDB assigns a unique BSON object ID, which includes a timestamp, machine ID, process ID, and a counter to ensure uniqueness.

HBase:

HBase, a column-oriented database, uses row keys to identify records. These row keys are byte arrays that should be logically significant for the application's data model. HBase rows are ordered by these keys, influencing how data is accessed and queried.

Redis:

Redis, a key-value store, uses simple keys to store values. Each key in Redis is unique and can be a string or binary data. Redis keys are typically generated by the application, ensuring they are unique and relevant to the stored data.

By understanding these different approaches to creating records in NoSQL databases, we can better appreciate the flexibility and challenges associated with each model. This knowledge will be further expanded as we explore read, update, and delete operations in the subsequent sections

3.2.1 Creating Records in a Document-Centric Database:

A typical example used in many relational database examples is that of a simplified retail system, which creates and manages order records. Each person’s purchase at this fictitious store is an order. An order consists of a bunch of line items. Each order line item includes a product (an item) and number of units of that product purchased. A line item also has a price attribute, which is calculated by multiplying the unit price of the product by the number of units purchased. Each order table has an associated product table that stores the product description and a few other attributes about the product. Figure 3-1 depicts order, product, and their relationship table in a traditional entity-relationship diagram.

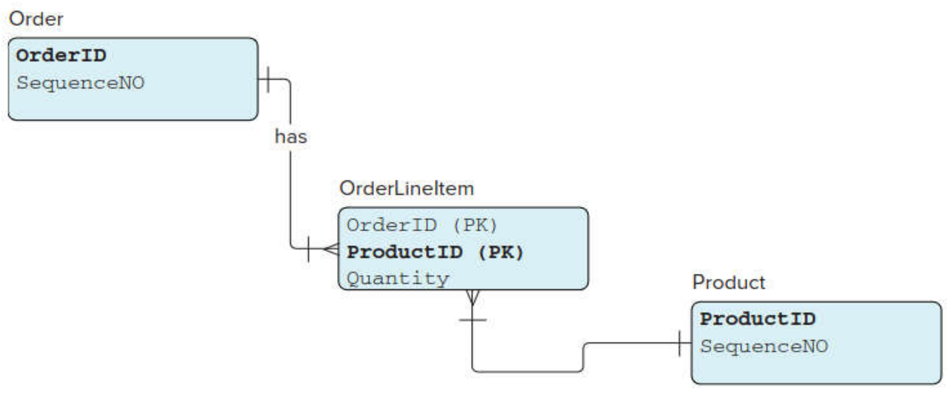


Figure 3-1

To store this same data in MongoDB, a document store, you would de-normalize the structure and store each order line item detail with the order record itself. As a specific case, consider an order of four coffees: one latte, one cappuccino, and two regular. This coffee order would be stored in MongoDB as a graph of nested JSON-like documents as follows:

```
{
        order_date: new Date(),
        “line_items”: [
                {
                        item : {
                name: “latte”,
                unit_price: 4.00
},
quantity: 1
                },
                {
                        item: {
                                name: “cappuccino”,
                                unit_price: 4.25
                },
                quantity: 1
                },
                {
                        item: {
                                name: “regular”,
                                unit_price: 2.00
                },
                quantity: 2
                }
        ]
}
```

Open a command-line window, change to the root of the MongoDB folder, and start the MongoDB server as follows: bin/mongod --dbpath ~/data/db

Now, in a separate command window, start a command-line client to interact with the server: bin/mongo. Use the command-line client to store the coffee order in the orders collection, within the mydb database.

Although storing the entire nested document collection is advised, sometimes it’s necessary to store the nested objects separately. When nested documents are stored separately, it’s your responsibility to join the record sets together. There is no notion of a database join in MongoDB so you must either manually implement the join operation by using the object id on the client side or leverage the concept of DBRef.

You can restructure this example in a way that doesn't store the unit price data for a product in the nested document but keeps it separately in another collection, which stores information on products. In the new format, the item name serves as the key to link between the two collections. Therefore, the restructured orders data is stored in a collection called orders2 as follows:

```
> t2 = {
...     order_date: new Date(),
...     “line_items”: [
...         {
...             “item_name”:”latte”,
...             “quantity”:1
...         },
...         {
...             “item_name”:”cappuccino”,
...             “quantity”:1
...         },
...         {
...             “item_name”:”regular”,
...             “quantity”:2
...         }
...     ]
... };


{
    “order_date” : “Sat Oct 30 2010 23:03:31 GMT-0700 (PDT)”,
    “line_items” : [
        {
            “item_name” : “latte”,
            “quantity” : 1
        },
        {
            “item_name” : “cappuccino”,
            “quantity” : 1
        },
        {
```

```
                    "item_name" : "regular",
                    "quantity" : 2
            }
    ]
}
> db.orders2.save(t2);
```

To verify that the data is stored correctly, you can return the contents of the orders2 collection as follows:

```
> db.orders2.find();
{ "_id" : ObjectId("4ccd06e8d3c7ab3d1941b104"), "order_date" : "Sat Oct 30 2010
23:03:31 GMT-0700 (PDT)", "line_items" : [
    {
            "item_name" : "latte",
            "quantity" : 1
    },
...
] }
```

Next, save the product data, wherein item name and unit price are stored, as follows:

```
> p1 = {
...     "_id": "latte",
...     "unit_price":4
... };
{ "_id" : "latte", "unit_price" : 4 }
> db.products.save(p1);
```

Again, you can verify the record in the products collection with the help of the find method:

```
> db.products.find();
    { "_id" : "latte", "unit_price" : 4 }
```

Now, you could manually link the two collections and retrieve related data sets like this:

```
> order1 = db.orders2.findOne();
{
    "_id" : ObjectId("4ccd06e8d3c7ab3d1941b104"),
    "order_date" : "Sat Oct 30 2010 23:03:31 GMT-0700 (PDT)",
    "line_items" : [
                {
                        “item_name” : “latte”,
                        “quantity” : 1
                },
                {
                        “item_name” : “cappuccino”,
                        “quantity” : 1
                },
                {
                        “item_name” : “regular”,
                        “quantity” : 2
                }
        ]
}
> db.products.findOne( { _id: order1.line_items[0].item_name } );
                        { “_id” : “latte”, “unit_price” : 4 }
```

Alternatively, part of this manual process can be automated with the help of DBRef, which is a more formal specification for relating two document collections in MongoDB. To illustrate DBRef, you rehash the orders example and establish the relationship by first defining the products and then setting up a DBRef to products from within the orders collection.

Add latte, cappuccino, and regular, with their respective unit prices, to the product2 collection as follows:

```
> p4 = {“name”:”latte”, “unit_price”:4};
        { “name” : “latte”, “unit_price” : 4 }
> p5 = {
... “name”: “cappuccino”,
... “unit_price”:4.25
... };
{ “_id” : “cappuccino”, “unit_price” : 4.25 }
> p6 = {
... “name”: “regular”,
... “unit_price”:2
... };
{ “_id” : “regular”, “unit_price” : 2 }
> db.products2.save(p4);
> db.products2.save(p5);
> db.products2.save(p6);
```

Verify that all the three products are in the collection:

```
> db.products.find();
{ “_id” : ObjectId(“4ccd1209d3c7ab3d1941b105”), “name” : “latte”,
    “unit_price” : 4 }
{ “_id” : ObjectId(“4ccd1373d3c7ab3d1941b106”), “name” : “cappuccino”,
    “unit_price” : 4.25 }
{ “_id” : ObjectId(“4ccd1377d3c7ab3d1941b107”), “name” : “regular”,
    “unit_price” : 2 }
```

Next, define a new orders collection, called orders3, and use DBRef to establish the relationship between orders3 and products. The orders3 collection can be defined as follows:

```
t3 = {
    ... order_date: new Date(),
    ... “line_items”: [
... {
    ... “item_name”: new DBRef(‘products2’, p4._id),
    ... “quantity:1
... },
... {
    ... “item_name”: new DBRef(‘products2’, p5._id),
    ... “quantity”:1
... },
... {
    ... “item_name”: new DBRef(‘products2’, p6._id),
    ... “quantity”:2
... }
    ... ]
... };
db.orders3.save(t3);
```

The MongoDB creation process is fairly simple and as you saw, some aspects of the relationship can also be formally established using DBRef. Next, the create operation is viewed in the context of column-oriented databases.

3.3.2 Using the Create Operation in Column-Oriented Databases:

Column-oriented databases, unlike MongoDB, do not incorporate relational references such as foreign keys or constraints across multiple collections. These databases store data in a de-normalized manner, similar to a data warehouse fact table that contains extensive transactional records. In this structure, a row-key uniquely identifies each record, and all columns within a column-family are stored together.

In column-oriented databases like HBase, data storage includes a time dimension, making the create or data insert operation crucial while effectively eliminating the concept of updating records. For example, consider maintaining a large catalog of various products, with varying amounts of information on the type, category, characteristics, price, and source of each product. You might create a table with column-families for type, characteristics, and source. Each column-family would then contain individual attributes or fields (referred to as columns).

To start the HBase server, open a command-line window or terminal, navigate to the HBase installation directory, and start the server in local standalone mode with the following command:

bin/start-hbase.sh

This command initializes the HBase environment, allowing you to create and manage your product catalog efficiently within a column-oriented database structure.

Open another command-line window and connect to the HBase server using the HBase shell: *bin/hbase shell*

Next, create the products table:

```
hbase(main):001:0> create ‘products’, ‘type’, ‘characteristics’, ‘source’

> 0 row(s) in 1.1570 seconds
```

Once the table is created, you can save data in it. HBase uses the put keyword to denote a data creation operation. The word “put” connotes a hash map-like operation for data insertion and because HBase under the hood is like a nested hash map, it’s probably more appropriate than the create keyword.

To create a record with the following fields:

```
type:category = “coffee beans”
type:name = “arabica”
type:genus = “Coffea”
characteristics: cultivation_method = “organic”

characteristics: acidity = “low”
```

you can put it into the products table like so:

```
hbase(main):001:0> put ‘products’, ‘product1’, ‘type:category’, ‘coffee beans’
>  0 row(s) in 0.0710 seconds
```

```
hbase(main):002:0> put ‘products’, ‘product1’, ‘type:name’, ‘arabica’
> 0 row(s) in 0.0020 seconds
hbase(main):003:0> put ‘products’, ‘product1’, ‘type:genus’, ‘Coffea’
> 0 row(s) in 0.0050 seconds
hbase(main):004:0> put ‘products’, ‘product1’,

‘characteristics: cultivation_method’, ‘organic’
> 0 row(s) in 0.0060 seconds
hbase(main):005:0> put ‘products’, ‘product1’, ‘characteristics: acidity’,
‘low’
> 0 row(s) in 0.0030 seconds
```

Now you can query for the same record to make sure it’s in the data store. To get the record do the following:

```
hbase(main):008:0> get ‘products’, ‘product1’

COLUMN                     CELL
characteristics: acidity   timestamp=1288555025970, value=lo
characteristics: cultivatio       timestamp=1288554998029,
value=organic
n_method
source: country            timestamp=1288555050543, value=yemen
source: terrain            timestamp=1288555088136,
value=mountainous
type:category              timestamp=1288554892522,    value=coffee
beans
type:genus           timestamp=1288554961942, value=Coffea
type:name            timestamp=1288554934169, value=Arabica
7 row(s) in 0.0190 seconds
```

What if you put in a value for “type:category” a second time stored as “beans” instead of its original value of “coffee beans” as follows?

```
hbase(main):009:0> put ‘products’, ‘product1’, ‘type:category’, ‘beans’

> 0 row(s) in 0.0050 seconds
```

Now, if you get the record again, the output is as follows:

```
hbase(main):010:0> get ‘products’,       ‘product1’
COLUMN                     CELL
characteristics: acidity   timestamp=1288555025970, value=low
characteristics: cultivatio       timestamp=1288554998029,
value=organic
n_method
source: country      timestamp=1288555050543, value=yemen
source: terrain      timestamp=1288555088136, value=mountainous
```

```
type:category                 timestamp=1288555272656, value=beans
type:genus             timestamp=1288554961942, value=Coffea
type:name              timestamp=1288554934169, value=Arabica
7 row(s) in 0.0370 seconds
```

You may notice that the value for type: category is now beans instead of coffee beans. In reality, both values are still stored as different versions of the same field value and only the latest one of these is returned by default. To look at the last four versions of the type:category field, run the following command:

```
hbase(main):011:0>   get   ‘products’,   ‘product1’,   {   COLUMN   =>
‘type:category’,
VERSIONS => 4 }
COLUMN                        CELL
type:category                         timestamp=1288555272656,
value=beans
type:category                 timestamp=1288554892522,      value=coffee
beans
```

Currently, there are only two versions available, so those are returned.

When dealing with highly structured, limited, and relational data, HBase might not be the most suitable solution. HBase requires a flattened data structure, creating a hierarchy only between column-families and their constituent columns. Additionally, each cell's data is stored along a time dimension, necessitating the flattening of nested data sets for storage.

Consider a retail order system. In HBase, retail order data can be stored in a couple of ways:

Flatten All Data Sets:

Store all fields of an order, including product data, in a single row.

Maintain Order Line Items in a Single Row: Store product information in a separate table and reference the product row-key within the order line item information.

If you choose the first option of flattening the order data, you might make the following choices:

Create one column-family for regular line items and another for additional line items like discounts or rebates.

Within the regular line item column-family, include columns for item or product name, description, quantity, and price.

When flattening everything, ensure each line item has a unique key to prevent them from being stored as versions of the same key/value pair. For instance, instead of naming all product name columns as product_name, use unique identifiers like product_name_1, product_name_2, etc. This approach helps maintain data integrity and allows efficient data retrieval within HBase's flattened structure.

The next example uses Redis to illustrate creating data in a key/value map.

3.2.3 Using the Create Operation in Key/Value Maps:

Redis is a simple, yet powerful, data structure server that lets you store values as a simple key/value pair or as a member of a collection. Each key/value pair can be a standalone map of strings or reside in a collection. A collection could be any of the following types: list, set, sorted set, or hash.

A standalone key/value string pair is like a variable that can take string values. You can create a Redis string key/value map like so:

```
./redis-cli set akey avalue
```

You can confirm that the value is created successfully with the help of the get command as follows:

```
./redis-cli get akey
```

The response, as expected, is avalue. The set method is the same as the create or the put method. If you invoke the set method again but this time set another value for the key, akey, the original value is replaced with the new one. Try out the following:

```
./redis-cli set akey another value
./redis-cli get akey
```

The response, as expected, would be the new value: *another value.*

The familiar set and get commands for a string can't be used for Redis collections, though. For example, using lpush and rpush creates and populates a list. A nonexistent list can be created along with its first member as follows:

```
./redis-cli lpush list_of_books 'MongoDB: The Definitive Guide'
```

You can use the range operation to verify and see the first few members of the list — list_of_ books — like so:

```
./redis-cli lrange list_of_books 0 -1
1. "MongoDB: The Definitive Guide"
```

The range operation uses the index of the first element, 0, and the index of the last element, -1, to get all elements in the list. In Redis, when you query a nonexistent list, it returns an empty list and doesn't throw an exception.

You run a range query for a nonexistent list — mylist — like so: *./redis-cli lrange mylist 0 -1*

Redis returns a message: empty list or set. You can use lpush much as you use rpush to add a member to mylist like so:

```
./redis-cli rpush mylist 'a member'
```

Now, of course mylist isn't empty and repeating a range query reveals the presence of a member. Members can be added to a list, either on the left or on the right, and can be popped from either direction as well. This allows you to leverage lists as queues or stacks.

For a set data structure, a member can be added using the SADD operation. Therefore, you can add 'a set member' to aset like so:

```
./redis-cli sadd aset 'a set member'
```

The command-line program would respond with an integral value of 1 confirming that it's added to the set. When you rerun the same SADD command, the member is not added again. You may recall that a set, by definition, holds a value only once and so once present it doesn't make sense to add it again. You will also notice that the program responds with a 0, which indicates that nothing was added. Like sets, sorted sets store a member only once but they also have a sense of order like a list.

You can easily add 'asset member' to a sorted set, called azset, like so:

```
./redis-cli zadd azset 1 'a sset member'
```

The value 1 is the position or score of the sorted set member. You can add another member, 'sset member 2', to this sorted set as follows:

```
./redis-cli zadd azset 4 'sset member 2'
```

You could verify that the values are stored by running a range operation, similar to the one you used for a list. The sorted set range command is called zrange and you can ask for a range containing the first five values as follows:

```
./redis-cli zrange azset 0 4
1. "a sset member"
2. "sset member 2"
```

What happens when you now add a value at position or score 3 and what happens when you try and add another value to position or score 4, which already has a value?

Adding a value to azset at score 3 like so:

```
./redis-cli zadd azset 3 'member 3'
```

and running the zrange query like so:

```
./redis-cli zrange azset 0 4
```

reveals:

```
1. "a sset member"
2. "member 3"
3. "sset member 2"
```

Adding a value at position or score 3 again, like so:

./redis-cli zadd azset 3 'member 3 again'

and running the zrange query like so:

./redis-cli zrange azset 0 4

reveals that the members have been re-positioned to accommodate the new member, like so:

1. "a sset member"

2. "member 3"

3. "member 3 again"

4. "sset member 2"

Therefore, adding a new member to a sorted set does not replace existing values but instead re-orders the members as required.

Redis also defines the concept of a hash, in which members could be added like so:

./redis-cli hset bank account1 2350

./redis-cli hset bank account2 4300

You can verify the presence of the member using the hget, or its variant hgetall, command:

./redis-cli hgetall bank

To store a complicated nested hash, you could create a hierarchical hash key like so:

./redis-cli hset product:fruits apple 1.35

./redis-cli hset product:fruits banana 2.20

Once data is stored in any of the NoSQL data stores, you need to access and retrieve it. After all, the entire idea of saving data is to retrieve it and use it later

3.3 ACCESSING DATA

You have already seen some of the ways to access data. In an attempt to verify whether records were created, some of the simplest get commands have already been explored.

3.3.1 Accessing Documents from MongoDB:

MongoDB allows for document queries using syntax and semantics that closely resemble SQL. Ironic as it may be, the similarity to SQL in a NoSQL world makes querying for documents easy and powerful in MongoDB.

You can dive right in to accessing a few nested MongoDB documents. Once again, you use the orders collection in the database mydb, which was created earlier in this chapter.

Start the MongoDB server and connect to it using the mongo JavaScript shell. Change to the mydb database with the use mydb command. First, get all the documents in the orders collection like so:

db.orders.find()

Now, start filtering the collection. Get all the orders after October 25, 2010, that is, with order_date greater than October 25, 2010. Start by creating a date object. In the JavaScript shell it would be: var refdate = new Date(2010, 9, 25);

JavaScript dates have months starting at 0 instead of 1, so the number 9 represents October. In Python the same variable creation could be like so:

from datetime import datetime

refdate = datetime(2010, 10, 25)

and in Ruby it would be like so:

require 'date'

refdate = Date.new(2010, 10, 25)

Then, pass refdate in a comparator that compares the order_date field values against refdate.

The query is as follows:

db.orders.find({"order_date": {$gt: refdate}});

MongoDB supports a rich variety of comparators, including less than, greater than, less than or equal to, greater than or equal to, equal to, and not equal to. In addition, it supports set inclusion and exclusion logic operators like contained in and not contained in a given set. The data set is a nested document so it can be beneficial to query on the basis of a value of a nested property. In Mongo, doing that is easy. Traversing through the tree using dot notation could access any nested field. To get all documents from the orders collection where line item name is latte, you write the following query:

db.orders.find({ "line_items.item.name" : "latte" })

The dot notation works whether there are single nested values or a list of them as was the case in the orders collection. MongoDB expression matching supports regular expressions. Regular expressions can be used in nested documents the same way they are used with top-level fields. In relational databases, indexes are the smart way of making queries faster. In general, the way that works is simple. Indexes provide an efficient lookup mechanism based on a B-tree-like structure that avoids complete table scans. Because less data is searched through to find the relevant records, the queries are faster and more efficient. MongoDB supports the notion of indexes to speed up queries. By default, all collections are indexed on the basis of the _id value. In addition to this default index, MongoDB allows you to create secondary indexes. Secondary indexes can be created at the top field level or at the nested field levels. For example, you could create an index on the quantity value of a line item as follows:

db.orders.ensureIndex({ "line_items.quantity" : 1 });

Now, querying for all documents where quantity of a line item is 2 can be fairly fast. Try running the following query:

db.orders.find({ "line_items.quantity" : 2 });

Indexes are stored separate from the table and they use up a namespace.

3.3.2 Accessing Data from HBase:

The most efficient way to query HBase is by using the row-key. Row-keys in HBase are ordered, and ranges of contiguous row-keys are stored together. Therefore, querying a row-key involves finding the range where the starting row-key is less than or equal to the given row-key.

Designing the row-key correctly is crucial for an application's performance. It's beneficial to relate the row-key semantically to the data. For instance, the Google Bigtable research paper suggests using inverted domain names for row-keys to group related content. Similarly, for an orders table, you might design row-keys using a combination of item or product name, order date, and category. The sequence depends on the primary access pattern. For chronological access, use:

<date> + <timestamp> + <category> + <product>

For access by category and product names, use:

<category> + <product> + <date> + <timestamp>

While row-keys provide efficient lookup for vast data amounts, HBase lacks built-in support for secondary indexes. Queries not leveraging row-keys result in table scans, which are costly and slow.

3.4 UPDATING AND DELETING DATA

In the relational database world, ACID (Atomicity, Consistency, Isolation, Durability) semantics ensure database integrity. These principles enforce various levels of data isolation and modification control. However, NoSQL databases often deprioritize or even disregard ACID transactions.

Understanding ACID:

Atomicity: A transaction is fully completed or fully rolled back.

Consistency: Every transaction brings the database from one valid state to another, maintaining database invariants.

Isolation: Transactions do not interfere with each other; the result is as if transactions were serially executed.

Durability: Once a transaction is committed, it remains so, even in the event of a system failure.

NoSQL databases, like MongoDB, HBase, and Redis, handle updates and deletions differently compared to traditional relational databases.

3.4.1 Updating and Modifying Data in MongoDB, HBase, and Redis

Unlike relational databases, NoSQL databases do not typically use locking mechanisms. This design choice facilitates sharding and scalability. In distributed systems, locking can complicate data updates and degrade performance.

Despite the absence of locking, you can perform atomic updates using specific techniques. For instance, update the entire document rather than individual fields to maintain atomicity. Utilize the atomic methods provided by the database. For MongoDB, available atomic methods include:

$set: Updates the value of a field.

$inc: Increments the value of a field.

$push: Appends a value to an array field.

$pull: Removes instances of a value from an array field.

$addToSet: Adds a value to an array field if it does not already exist.

These methods help ensure that updates are atomic and consistent, even without traditional locking mechanisms. By understanding and leveraging these methods, you can effectively manage updates and deletions in NoSQL databases, maintaining data integrity and performance.

For example, { $set : { "order_date" : new Date(2010, 10, 01) } } updates the order_ date in the orders collection in an atomic manner. An alternative strategy to using atomic operations is to use the update if current principle. Essentially this involves three steps:

1. Fetch the object.
2. Modify the object locally.
3. Send an update request that says "update the object to this new value if it still matches its old value."

The document or row-level locking and atomicity also applies to HBase. HBase supports a row-level read-write lock. This means rows are locked when any column in that row is being modified, updated, or created. In HBase terms the distinction between create and update is not clear. Both operations perform similar logic. If the value is not present, it's inserted or else updated

Therefore, row-level locking is a great idea, unless a lock is acquired on an empty row and then it's unavailable until it times out. Redis has a limited concept of a transaction and an operation can be performed within the confines of such a transaction. Redis MULTI command initiates a transactional unit. Calling EXEC after a MULTI executes all the commands and calling DISCARD rolls back the operations. A simple example of atomic increment of two keys: key1 and key2 could be as follows:

```
> MULTI
OK

> INCR key1
QUEUED

> INCR key2
QUEUED

> EXEC
1) (integer) 1
2) (integer) 1
```

3.4.2 Limited Atomicity and Transactional Integrity:

While the specifics of minimal atomic support may differ among databases, they share many common characteristics. This section delves into some prevalent concepts surrounding the CAP Theorem and eventual consistency.

The CAP Theorem posits that at any given time, it's only possible to maximize two out of three factors:

Consistency – Ensuring that all clients have the same data view.

Availability – Guaranteeing that all clients can read and write.

Partition tolerance – Maintaining system functionality across distributed networks.

Another important concept often discussed is eventual consistency, which can be perplexing and frequently misunderstood.

Eventual consistency serves as a consistency model within parallel and distributed programming domains. It can be interpreted in two primary ways:

Over a sufficiently extended period without updates, it's anticipated that all updates will eventually propagate through the system, resulting in consistency across all replicas.

In the presence of ongoing updates, a given update will eventually reach a replica or the replica will be retired from service. Eventual consistency aligns with the principles of Basically Available, Soft State, Eventual consistency (BASE), contrasting with the principles of ACID discussed earlier.

3.5 SUMMARY

This chapter introduced the fundamental operations of create, read, update, and delete within the framework of NoSQL databases. It delved into these operations within the context of three types of NoSQL data stores: document stores, column-oriented databases, and key/value hash maps. MongoDB serves as an example of a document store, HBase as a column store, and Redis as a key/value hash map.

Throughout the discussion, it became evident that across all data stores, the emphasis lies more on data creation or insertion rather than updates. In certain scenarios, updates are constrained. Towards the conclusion of the chapter, the discussion extended to cover topics such as updates, transactional integrity, and consistency.

3.6 REVIEW QUESTIONS

1. How do create, read, update, and delete operations differ in NoSQL databases compared to traditional relational databases?
2. What are the distinctive characteristics of document stores, column-oriented databases, and key/value hash maps in the context of NoSQL databases?
3. How do NoSQL databases reconcile the principles of ACID with the emphasis on data creation or insertion over updates?

*****

4

QUERYING NOSQL STORES

Unit Structure

4.0 OBJECTIVES

- Demonstrating various query mechanisms within NoSQL databases through examples with sample datasets.
- Exploring querying scenarios specific to MongoDB, HBase, and Redis.
- Crafting sophisticated and intricate queries in NoSQL environments.
- Leveraging non-SQL alternatives to achieve robust querying functionalities.

4.1 INTRODUCTION

SQL stands out as one of the simplest yet most potent domain-specific languages ever devised. Its learning curve is gentle due to a limited vocabulary, clear grammar, and straightforward syntax. Despite its brevity and narrow focus, SQL excels at its intended purpose, allowing users to manipulate structured datasets with precision akin to a skilled ninja. Through SQL, users effortlessly filter, sort, dissect, and segment data, harnessing the power of relations to join datasets and create intersections and unions.

However, SQL's reliance on relational algebra restricts its utility to relational databases exclusively, as implied by its name — SQL lacks compatibility with NoSQL databases. Nevertheless, the absence of SQL doesn't impede querying of datasets, as data storage inherently implies the potential for retrieval and manipulation. NoSQL databases offer their own mechanisms for accessing and manipulating data, often straying from relational constraints.

While proponents of NoSQL databases sought alternatives to relational databases due to structural constraints and the rigidity of ACID transactions, they didn't necessarily reject SQL outright. Indeed, some still yearn for SQL's familiarity in the realm of NoSQL, leading to the creation of query languages bearing resemblance to SQL syntax and style.

In this chapter, you'll delve into numerous tips and tricks for querying NoSQL stores, exploring various products and technologies under the broad umbrella of NoSQL.

4.2 EXPLORING SIMILARITIES BETWEEN SQL AND MONGODB QUERY FEATURES

Despite MongoDB's identity as a document database, its query language exhibits striking similarities to SQL. With preliminary examples already presented, the SQL-like query capabilities of MongoDB are self-evident.

To grasp the capabilities of MongoDB's query language and witness its functionality firsthand, let's load a more substantial dataset into a MongoDB database. While previous datasets in this book have been modest in scale to emphasize MongoDB's core features, this chapter introduces the MovieLens dataset, comprising millions of movie-rating records.

To begin, visit grouplens.org/node/73 and download the dataset containing one million movie-rating records. The dataset is available in tar.gz and .zip formats; choose the appropriate format for your platform. After downloading, extract the contents to a folder in your file system. Upon extraction, you should have three files:

- movies.dat
- ratings.dat
- users.dat

The movies.dat data file contains data on the movies themselves. This file contains 3,952 records, and each line in that file contains one record. The record is saved in the following format:

<MovieID>::<Title>::<Genres>

The MovieId is a simple integral sequence of numbers. The movie title is a string, which includes the year the movie was released, specified in brackets appended to its name. The movie titles are the same as those in IMDB (www.imdb.com). Each movie may be classified under multiple genres, which are specified in a pipe-delimited format. A sample line from the file is like so:

1::Toy Story (1995)::Animation|Children's|Comedy

The ratings.dat file contains the ratings of the 3,952 movies by more than 6,000 users. The ratings file has more than 1 million records. Each line is a different record that contains data in the following format:

UserID::MovieID::Rating::Timestamp

UserID and MovieID identify and establish a relationship with the user and the movie, respectively. The rating is a measure on a 5-point (5-star) scale. Timestamp captures the time when the ratings were recorded.

The users.dat file contains data on the users who rated the movies. The information on more than 6,000 users is recorded in the following format:

UserID::Gender::Age::Occupation::Zip-code

4.2.1 Loading the MovieLens Data:

To simplify the process, let's upload the data into three MongoDB collections: movies, ratings, and users, with each collection mapping to a corresponding .dat data file. Unfortunately, the mongoimport utility, which is typically used for this task, doesn't support the double-colon (::) delimiter used in the MovieLens data. As an alternative, we'll utilize a programming language along with a MongoDB driver to parse the text files and load the dataset into MongoDB collections.

For brevity, we'll use Ruby for this task. However, you could also opt for Python, Java, PHP, C, or any other supported language. The following code snippet (Listing 4-1) demonstrates how to extract and load data from the users, movies, and ratings data files into their respective MongoDB collections. This code employs basic file-reading and string-splitting functionalities, coupled with the MongoDB driver to accomplish the task. While not the most elegant solution and lacking exception handling, it serves our immediate purpose.

LISTING 4-1: movielens_dataloader.rb

```
require ‘rubygems’ #can skip this line in Ruby 1.9
require ‘mongo’

field_map = {
        “users” => %w(_id gender age occupation zip_code),
        “movies” => %w(_id title genres),
        “ratings” => %w(user_id movie_id rating timestamp)
}

db = Mongo::Connection.new.db(“mydb”)
collection_map = {
        “users” => db.collection(“users”),
        “movies” => db.collection(“movies”),
        “ratings” => db.collection(“ratings”)
```

```
}
unless ARGV.length == 1
        puts “Usage: movielens_dataloader data_filename”
        exit(0)
end

class Array
    def to_h(key_definition)
        result_hash = Hash.new()
        counter = 0
        key_definition.each do |definition|
            if not self[counter] == nil then
                if self[counter].is_a? Array or self[counter].is_a? Integer then
                        result_hash[definition] = self[counter]
                else
                        result_hash[definition] = self[counter].strip
                end
            else
                # Insert the key definition with an empty value.
                # Because we probably still want the hash to contain the key.
                result_hash[definition] = “”
            end

            # For some reason counter.next didn’t work here....
            counter = counter + 1
        end

        return result_hash
    end
end

if File.exists?(ARGV[0])
        file = File.open(ARGV[0], ‘r’)
        data_set = ARGV[0].chomp.split(“.”)[0]
                file.each { |line|
                field_names = field_map[data_set]
                field_values = line.split(“::”).map { |item|
                if item.to_i.to_s == item
                        item = item.to_i

                else

                        item
            end
        }
        puts "field_values: #{field_values}"
        #last_field_value = line.split("::").last
        last_field_value = field_values.last
        puts "last_field_value: #{last_field_value}"
        if last_field_value.split("|").length > 1
            field_values.pop
            field_values.push(last_field_value.split().join('\n').split("|"))
        end
        field_values_doc = field_values.to_h(field_names)
        collection_map[data_set].insert(field_values_doc)
        }
        puts "inserted #{collection_map[data_set].count()} records into the
#{collection_map[data_set].to_s} collection"
end
```

Once the data is loaded into the MongoDB collections, you're ready to execute queries to manipulate and analyze it. Queries can be executed from the JavaScript shell or from any supported programming language. In this example, most queries will be executed using the JavaScript shell, with a few select queries demonstrated using different programming languages and their respective drivers. The inclusion of programming language examples serves to illustrate that nearly all functionalities available in the JavaScript shell are accessible through the various language drivers.

To begin querying the MongoDB collections, ensure that the MongoDB server is running and connect to it using the Mongo shell, which can be found in the bin folder of your MongoDB installation directory.

In the Mongo JavaScript shell, start by retrieving a count of all the values in the ratings collection with the following command:

db.ratings.count();

You should receive a response of 1000209, confirming that over a million ratings were successfully uploaded.

Next, retrieve a sample set of the ratings data using the following command:

db.ratings.find();

In the shell, you don't need to explicitly use a cursor to display values from a collection. The shell automatically limits the display to a maximum of 20 rows at a time. To view more data, simply type "it" (short for iterate) in the shell. If more records exist beyond the ones already displayed, you'll see 20 more records along with a label indicating "has more."

The ratings data, for example, { “_id” : ObjectId(“4cdcf1ea5a918708b0000001”), “user_ id” : 1, “movie_id” : 1193, “rating” : 5, “timestamp” : “978300760” }, makes little intuitive sense about the movie it relates to because it’s linked to the movie id and not its name. You can get around this problem by answering the following questions:

- How can I get all the ratings data for a given movie?
- How do I get the movie information for a given rating?
- How do I put together a list all the movies with the ratings data grouped by the movies they relate to?

MongoDB, relational data is correlated explicitly outside the server's scope. MongoDB introduces the concept of a DBRef to establish relationships between fields in different collections, but this feature has certain limitations and doesn't offer the same level of functionality as explicit ID-based linking.

To retrieve all ratings data for a specific movie in MongoDB, you filter the dataset using the movie ID as the criterion. For instance, to view all ratings for the renowned Academy Award-winning movie "Titanic," you first need to find its ID and then use that to filter the ratings collection. If you're unsure about the exact title string for "Titanic" but confident that the word "titanic" appears in it, you can perform an approximate, rather than exact, match with the title strings in the movies collection.

In a relational database management system (RDBMS), under such circumstances, you might rely on the LIKE expression in a SQL WHERE clause to retrieve a list of all potential candidates. In MongoDB, although there's no LIKE expression, there's a more robust feature available: the ability to define patterns using regular expressions. Thus, to obtain a list of all records in the movies collection that contain "Titanic" or "titanic" in their title, you can execute the following query:

db.movies.find({ title: /titanic/i});

This query returns the following set of documents:

{ “_id” : 1721, “title” : “Titanic (1997)”, “genres” : [“Drama”, “Romance”] }

{ “_id” : 2157, “title” : “Chambermaid on the Titanic, The (1998)”, “genres” : “Romance” }

{ “_id” : 3403, “title” : “Raise the Titanic (1980)”, “genres” : [“Drama”, “Thriller”] }

{ “_id” : 3404, “title” : “Titanic (1953)”, “genres” : [“Action”, “Drama”] }

The title field in the MovieLens data set includes the year the movie was released. Within the title field, the release year is included in parentheses. So, if you remembered or happen to know that Titanic was released in the year 1997, you can write a more tuned query expression as follows: *db.movies.find({ title: /titanic.*\(1997\).*/i});*

This returns just one document:

{ "_id" : 1721, "title" : "Titanic (1997)", "genres" : ["Drama", "Romance"] }

The expression essentially looks for all title strings that have Titanic, titanic, TitaniC, or TiTAnic in it. In short, it ignores case. In addition, it looks for the string (1997). It also states that there may be 0 or more characters between titanic and (1997) and after (1997). The support for regular expressions is a powerful feature and it is always worthwhile to gain mastery over them. The range of values for the movie_id field of the ratings collection is defined by the _id of the movies collection. So to get all ratings for the movie Titanic, which has an id of 1721, you could query like so:

db.ratings.find({ movie_id: 1721 });

To find out the number of available ratings for Titanic, you can count them as follows:

db.ratings.find({ movie_id: 1721 }).count();

The response to the count is 1546. The ratings are on a 5-point scale. To get a list and count of only the 5-star ratings for the movie Titanic you can further filter the record set like so:

db.ratings.find({ movie_id: 1721, rating: 5 });

db.ratings.find({ movie_id: 1721, rating: 5 }).count();

DATA-TYPE SENSITIVITY IN QUERY DOCUMENTS:

MongoDB query documents are data-type sensitive. That is, { movie_id: "1721" } and { movie_id: 1721 } are not the same, the first one matches a string and the second one considers the value as a number. When specifying documents, be sure to use the correct data type. To illustrate further, the movie_id is stored as a number (integer) in the ratings and the movies collections, so querying for a string match doesn't return correct results.

Therefore, the response to *db.ratings .find({ movie_id: 1721 });* returns up to a total of 1,546 documents, but the response to *db.ratings.find({ movie_id: "1721" });* returns none.

If you browse Listing 6-1 carefully, you will notice the following line:

```
field_values = line.split("::").map { |item|
if item.to_i.to_s == item
        item = item.to_i
else
    item
end
}
```

This bit of code checks to see if the split string holds an integer value and saves it as an integer, if that's the case. Making this little extra effort to save numerical values as numbers has its benefits. Indexing and querying on numerical records is usually faster and more efficient than on character-based (string) records.

Next, you may want to get some statistics of all the ratings for Titanic. To find out the distinct set of ratings by users (from the possible set of integers between 1 and 5, both inclusive), you could query as follows:

```
db.runCommand({ distinct: 'ratings', key: 'rating', query: { movie_id: 1721} });
```

Ratings for Titanic include all possible cases between 1 and 5 (both inclusive) so the response is like so:

```
{ "values" : [ 1, 2, 3, 4, 5 ], "ok" : 1 }
```

runCommand takes the following arguments:

Collection name for the field labeled distinct

Field name for key, whose distinct values would be listed

Query to optionally filter the collection

runCommand is slightly different in pattern than the query style you have seen so far because the collection is filtered before the distinct values are searched for. Distinct values for all ratings in the collection can be listed in a way that you have seen so far, as follows:

```
db.ratings.distinct("rating");
```

You know from the distinct values that Titanic has all possible ratings from 1 to 5. To see how these ratings break down by each rating value on the 5-point scale, you could group the counts like so:

```
db.ratings.group(
... { key: { rating:true },
... initial: { count:0 },
... cond: { movie_id:1721 },
... reduce: function(obj, prev) { prev.count++; }
... }
... );
```

The output of this grouping query is an array as follows:

```
[
{
        "rating" : 4,
        "count" : 500
},
{
        "rating" : 1,
        "count" : 100
},
{
        "rating" : 5,
        "count" : 389
},
{

        "rating" : 3,
        "count" : 381
},
{
        "rating" : 2,
        "count" : 176
}
]
```

This group by function is quite handy for single MongoDB instances but doesn't work in sharded deployments. Use MongoDB's MapReduce facility to run grouping functions in a sharded MongoDB setup. A MapReduce version of the grouping function is included right after the group operation is explained. The group operation takes an object as an input. This group operation object includes the following fields: key — The document field to group by. The preceding example has only one field: rating.

Additional group by fields can be included in a comma-separated list and assigned as the value of the key field. A possible configuration could be

– key: { fieldA: true, fieldB: true}.

initial — Initial value of the aggregation statistic.

In the previous example the initial count is set to 0.

cond — The query document to filter the collection.

reduce — The aggregation function.

keyf (optional) — An alternative derived key if the desired key is not an existing document field.

finalize (optional) — A function that can run on every item that the reduce function iterates through. This could be used to modify existing items.

Theoretically, the example could easily be morphed into a case where ratings for each movie are grouped by the rating points by simply using the following group operation:

```
db.ratings.group(
... { key: { movie_id:true, rating:true },
... initial: { count:0 },
... reduce: function(obj, prev) { prev.count++; }
... }
... );
```

In real cases, though, this wouldn't work for the ratings collection of 1 million items. You would be greeted instead with the following error message:

```
Fri Nov 12 14:27:03 uncaught exception: group command failed: {
"errmsg" : "exception: group() can't handle more than 10000 unique keys",
"code" : 10043,
"ok" : 0
}
```

The result is returned as a single BSON object and therefore the collection over which the group operation is applied should not have more than 10,000 keys. This limitation can also be overcome with the MapReduce facility.

4.2.2 MapReduce in MongoDB:

MapReduce is a patented software framework developed by Google, designed to facilitate distributed computing across large clusters of computers. You can learn more about Google's MapReduce framework by referring to the research paper titled "MapReduce: Simplified Data Processing on Large Clusters," available online at http://labs.google.com/papers/mapreduce.html. This framework has inspired numerous clones and distributed computing frameworks within the open-source community, including MongoDB's implementation.

Both Google's and MongoDB's MapReduce features draw inspiration from similar constructs found in the realm of functional programming. In functional programming, a map function applies a function to each member of a collection, while a reduce function (or fold function) aggregates results across the collection.

MongoDB's MapReduce functionality differs from Google's original framework and is not merely a clone. Hadoop's MapReduce, on the other hand, serves as an open-source implementation of Google's distributed computing concepts, incorporating infrastructure for both column databases (like HBase) and MapReduce-based computing.

While grasping the concept of MapReduce can initially seem daunting, understanding its structure and flow reveals it to be a potent tool for executing large-scale computations across distributed datasets. Starting with simple examples and gradually progressing to more complex ones is an effective approach to mastering this topic.

A basic example of aggregation using MapReduce could involve counting the occurrences of each item type in a collection. To utilize MapReduce, you must define both a map function and a reduce function, and then apply these functions to the collection. The map function applies a specified function to every member of the collection, emitting a key/value pair for each member. The output of the map function, in the form of key/value pairs, is then processed by the reduce function, which aggregates the results across all key/value pairs to produce an output.

For instance, consider the following map function designed to count the number of female (F) and male (M) respondents in the users collection:

```
> var map = function() {
... emit({ gender:this.gender }, { count:1 });
... };
```

This map function emits a key/value pair for each item in the collection that has a gender property. It counts 1 for each such occurrence. The reduce function for counting the number of total occurrences of male and female types among all users is as follows:

```
> var reduce = function(key, values) {
... var count = 0;
... values.forEach(function(v) {
... count += v['count'];
... });
...
... return { count:count };
... };
```

A reduce function takes a key/value pair emitted by the map function. In this particular reduce function, each value in the key/value pair is passed through a function that counts the number of occurrences of a particular type. The line count += v['count'] could also be written as count += v.count because of JavaScript's ability to access object members and their values as a hash data structure.

Finally, running this map and reduce function pair against the users collection leads to an output of the total count of female and male members in the users collection. The mapReduce run and result extraction commands are as follows:

```
> var ratings_respondents_by_gender = db.users.mapReduce(map, reduce);
> ratings_respondents_by_gender
{
"result" : "tmp.mr.mapreduce_1290399924_2",
"timeMillis" : 538,
"counts" : {
"input" : 6040,
"emit" : 6040,
"output" : 2
},
"ok" : 1,
}
> db[ratings_respondents_by_gender.result].find();
{ "_id" : { "gender" : "F" }, "value" : { "count" : 1709 } }
{ "_id" : { "gender" : "M" }, "value" : { "count" : 4331 } }
```

To verify the output, filter the users collection for gender values "F" and "M" and count the number of documents in each filtered sub-collection. The commands for filtering and counting the users collection for gender values "F" and "M" is like so:

```
> db.users.find({ "gender":"F" }).count();
1709
> db.users.find({ "gender":"M" }).count();
4331
```

Next, you can modify the map function slightly and run the map and reduce functions against the ratings collection to count the number of each type of rating (1, 2, 3, 4 or 5) for each movie. In other words, you are counting the collection grouped by rating value for each movie. Here are the complete map and reduce function definitions run against the ratings collection:

```
> var map = function() {
... emit({ movie_id:this.movie_id, rating:this.rating }, { count:1 });
... };


> var reduce = function(key, values) {
```

```
... var count = 0;
... values.forEach(function(v) {
... count += v['count'];
... });
...
... return { count: count };
... };
> var group_by_movies_by_rating = db.ratings.mapReduce(map, reduce);
> db[group_by_movies_by_rating.result].find();
```

To get a count of each type of rating for the movie Titanic, identified by movie_id 1721, you simply filter the MapReduce output using nested property access method like so:

```
> db[group_by_movies_by_rating.result].find({ "_id.movie_id":1721 });
{ "_id" : { "movie_id" : 1721, "rating" : 1 }, "value" : { "count" : 100 } }
{ "_id" : { "movie_id" : 1721, "rating" : 2 }, "value" : { "count" : 176 } }
{ "_id" : { "movie_id" : 1721, "rating" : 3 }, "value" : { "count" : 381 } }
{ "_id" : { "movie_id" : 1721, "rating" : 4 }, "value" : { "count" : 500 } }
{ "_id" : { "movie_id" : 1721, "rating" : 5 }, "value" : { "count" : 389 } }
```

In the two examples of MapReduce so far, the reduce function is identical but the map function is different. In each case a count of 1 is established for a different emitted key/value pair. In one a key/ value pair is emitted for each document that has a gender property, whereas in the other a key/value pair is emitted for each document identified by the combination of a movie id and a rating id. Next, you could calculate the average rating for each movie in the ratings collection as follows:

```
> var map = function() {
... emit({ movie_id:this.movie_id }, { rating:this.rating, count:1 });
... };
> var reduce = function(key, values) {
... var sum = 0;
... var count = 0;
... values.forEach(function(v) {
... sum += v['rating'];
... count += v['count'];
... });
...
... return { average:(sum/count) };
... };
> var average_rating_per_movie = db.ratings.mapReduce(map, reduce);
> db[average_rating_per_movie.result].find();
```

MapReduce allows you to write many types of sophisticated aggregation algorithms, some of which were presented in this section. A few others are introduced later in the book. By now you have had a chance to understand many ways of querying MongoDB collections. Next, you get a chance to familiarize yourself with querying tabular databases. HBase is used to illustrate the querying mechanism.

4.3 ACCESSING DATA FROM COLUMN-ORIENTED

4.3.1 Databases Like Hbase:

Before delving into querying an HBase data store, it's necessary to first populate it with data. Similar to MongoDB, you've already gained some experience with storing and accessing data in HBase and its underlying file system, often Hadoop Distributed FileSystem (HDFS).

This familiarity with HBase and Hadoop basics will serve as a foundation for this section. As a practical example, historical daily stock market data from the New York Stock Exchange (NYSE) spanning from the 1970s to February 2010 will be loaded into an HBase instance. This dataset, sourced from original data providers by Infochimp.org, is available for access at www.infochimps.com/datasets/nyse-daily-1970-2010-open-close-high-low-and-volume.

4.3.2 The Historical Daily Market Data:

The zipped download of the entire dataset is substantial at 199 MB, yet relatively small compared to HDFS and HBase standards. The robust infrastructures of HBase and Hadoop are capable of handling petabytes of data spread across multiple physical machines. For the purpose of this example, a manageable dataset was chosen intentionally to avoid distraction from the complexities of preparing and loading large datasets.

This chapter focuses on query methods in NoSQL stores, particularly on column-oriented databases. The principles demonstrated with smaller datasets remain applicable to larger datasets.

The dataset is logically partitioned into three types of fields:

1. Combination of exchange, stock symbol, and date serving as the unique identifier.

2. Price-related metrics including open, high, low, close, and adjusted close.

3. Daily trading volume.

The row key can be constructed using a combination of the exchange, stock symbol, and date. For example, 'NYSE, AA, 2008-02-27' could be structured as 'NYSEAA20080227' to serve as a row key. All price-related information can be stored in a column family named 'price', while volume data can reside in a column family named 'volume'.

The table itself is named 'historical_daily_stock_price'. To retrieve the row data for 'NYSE, AA, 2008-02-27', you can issue the following query:

get 'historical_daily_stock_price', 'NYSEAA20080227'

To retrieve the open price:

get 'historical_daily_stock_price', 'NYSEAA20080227', 'price:open'

You could also use a programming language to query for the data. A sample Java program to get the

open and high price data could be as follows:

```
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.io.RowResult;
import java.util.HashMap;
import java.util.Map;
import java.io.IOException;

public class HBaseConnector {
    public static Map retrievePriceData(String rowKey) throws
IOException {
            HTable table = new HTable(new HBaseConfiguration(),
                  "historical_daily_stock_price");
Map stockData = new HashMap();
RowResult result = table.getRow(rowKey);
for (byte[] column : result.keySet()) {
    stockData.put(new String(column), new
    String(result.get(column).getValue()));
            }
    return stockData;
    }
public static void main(String[] args) throws IOException {
        Map                          stock_data                    =
        HBaseConnector.retrievePriceData("NYSEAA20080227");
        System.out.println(stock_data.get("price:open"));
        System.out.println(stock_data.get("price:high"));
        }
}
```

HBaseConnector.java

HBase includes very few advanced querying techniques beyond what is illustrated, but its capability to index and query can be extended with the help of Lucene and Hive

4.4 QUERYING REDIS DATA STORES

You've gained insights into the fundamentals of data storage and access with Redis. This section delves a bit deeper into querying data within Redis.

Consistent with the examples provided thus far in this chapter, a sample dataset is first loaded into a Redis instance. For demonstration purposes, the NYC Data Mine public raw data on parking spaces, available online at www.nyc.gov/data, is utilized. The dataset can be downloaded in a comma-separated text format, named parking_facilities.csv.

Refer to Listing 4-2 for a straightforward Python program that parses this CSV dataset and loads it into a local Redis store. Remember to initiate your local Redis server prior to executing the Python script to load the data. Running the Redis-server program, accessible in the Redis installation directory, initiates a Redis server instance that, by default, listens for client connections on port 6379.

LISTING 4-2: Python program to extract NYC parking facilities data

```
import csv
import redis


f = open("parking_facilities.csv", "r")
parking_facilities = csv.DictReader(f, delimiter=',')
r = redis.Redis(host='localhost', port=6379, db=0)
def add_parking_facility(license_number,
   facility_type,
   entity_name,
   camis_trade_name,
   address_bldg,
    address_street_name,
    address_location,
    address_city,
    address_state,
    address_zip_code,
    telephone_number,
    number_of_spaces):

if r.sadd("parking_facilities_set", license_number):
r.hset("parking_facility:%s" % license_number, "facility_type",
facility_type)
r.hset("parking_facility:%s" % license_number, "entity_name",
entity_name)
r.hset("parking_facility:%s" % license_number, "camis_trade_name",
camis_trade_name)
r.hset("parking_facility:%s" % license_number, "address_bldg",
address_bldg)
r.hset("parking_facility:%s" % license_number, "address_street_name",
address_street_name)
r.hset("parking_facility:%s" % license_number, "address_location",
address_location)
r.hset("parking_facility:%s"          %          license_number,
"address_city",address_city)
r.hset("parking_facility:%s" % license_number, "address_state",
address_state)
r.hset("parking_facility:%s" % license_number, "address_zip_code",
address_zip_code)
r.hset("parking_facility:%s" % license_number, "telephone_number",
telephone_number)
r.hset("parking_facility:%s" % license_number, "number_of_spaces",
number_of_spaces)
    return True
else:
    return False

if __name__ == "__main__":
      for parking_facility_hash in parking_facilities:
            add_parking_facility(parking_facility_hash['License
Number'],
parking_facility_hash['Facility Type'],
parking_facility_hash['Entity Name'],
parking_facility_hash['Camis Trade Name'],
parking_facility_hash['Address Bldg'],
parking_facility_hash['Address Street Name'],
parking_facility_hash['Address Location'],
parking_facility_hash['Address City'],
parking_facility_hash['Address State'],
parking_facility_hash['Address Zip Code'],
parking_facility_hash['Telephone Number'],
parking_facility_hash['Number of Spaces'])
    print "added parking_facility with %s" % parking_facility_hash['License Number']
```

nyc_parking_data_loader.py

The Python program loops through a list of extracted hash records and saves the values to a Redis instance. Each hash record is keyed using the license number. All license numbers themselves are saved in a set named parking_facilities_set.

To get a list of all license numbers in the set named parking_facilities_list, connect via another program or simply the command-line client and use the following command: SMEMBERS parking_facilities_set All 1,912 license numbers in the set would be printed out. You can run *wc –l paking_facilities.csv* to verify if this number is correct. Each line in the CSV corresponds to a parking facility so the two numbers should reconcile.

For each parking facility the attributes are stored in a hash, which is identified by the key of the form parking_facility:<license_number>. Thus, to see all keys in the hash associated with license number 1105006 you can use the following command:

HKEYS parking_facility:1105006

The response is as follows:

1. "facility_type"
2. "entity_name"
3. "camis_trade_name"
4. "address_bldg"
5. "address_street_name"
6. "address_location"

7. "address_city"

8. "address_state"

9. "address_zip_code"

10. "telephone_number"

11. "number_of_spaces"

The license number 1105006 was first on the list returned by the SMEMBERS parking_facilities_ set command. However, sets are not ordered, so rerunning this command may not result in the same license number on top. If you need the list of members to appear in a certain order, use the sorted sets instead of the set. All you may need to do to use a sorted set is to replace the line

if r.sadd("parking_facilities_set", license_number): with the following:
if r.zadd("parking_facilities_set", license_number):

Now, you can query for specific values in the hash, say facility type, as follows:
HGET parking_facility:1105006 facility_type

The response is "Parking Lot". You can also print out all values using the HVALS command as follows:
HVALS parking_facility:1105006

The response is:

1. "Parking Lot"

2. "CENTRAL PARKING SYSTEM OF NEW YORK, INC"

3. ""

4. "41-61"

5. "KISSENA BOULEVARD"

6. ""

7. "QUEENS"

8. "NY"

9. "11355"

10. "2126296602"

11. "808"

Of course, it would be much nicer if you could print out all the keys and the corresponding values in a hash. You can do that using the HGETALL command as follows:

HGETALL parking_facility:1105006

The response is as follows:

```
1. "facility_type"

2. "Parking Lot"

3. "entity_name"

4. "CENTRAL PARKING SYSTEM OF NEW YORK, INC"

5. "camis_trade_name"

6. ""

7. "address_bldg"

8. "41-61"

9. "address_street_name"

10. "KISSENA BOULEVARD"

11. "address_location"

12. ""

13. "address_city"

14. "QUEENS"

15. "address_state"

16. "NY"

17. "address_zip_code"

18. "11355"

19. "telephone_number"

20. "2126296602"

21. "number_of_spaces"

22. "808"
```

Sometimes, you may not need all the key/value pairs but just want to print out the values for a specific set of fields. For example, you may want to print out only the address_city and the address_zip_code as follows:

HMGET parking_facility:1105006 address_city address_zip_code

The response is:

1. "QUEENS"

2. "11355"

You could similarly set values for a set of fields using the HMSET command. To get a count of the number of keys, you can use the HLEN command as follows:
HLEN parking_facility:1105006

The response is 11. If you wanted to check if address_city was one of these, you can use the

HEXISTS command to verify if it exists as a key. The command is used as follows:

HEXISTS parking_facility:1105006 address_city

The response is 1 if the field exists and 0 if it doesn't.

Going back to the set parking_facilities_set, you may just want to count the number of members instead of listing them all using the SCARD command as follows: SCARD parking_facilities_set As expected, the response is 1912. You could verify if a specific member exists in the set using the SISMEMBER command. To verify if 1005006 is a member of the set, you could use the following command:
SISMEMBER parking_facilities_set 1105006

Integral values of 0 and 1 are returned to depict false and true for this query that verifies if a member exists in a set.

4.5 SUMMARY

In this chapter, several advanced query mechanisms were introduced, surpassing the complexity of those previously discussed. Querying concepts were elucidated through practical examples. MongoDB's querying intricacies were explored using a sample dataset containing movie ratings. The HBase illustration utilized historical stock market data, while Redis's querying capabilities were showcased using sample NYC government data.

It's important to note that the coverage of querying capabilities in this chapter is not exhaustive and does not encompass all possible use cases. The examples provided serve as just a glimpse into the myriad possibilities. However, navigating through these examples should help you grasp the style and mechanics of querying within NoSQL data stores.

4.6 REVIEW QUESTIONS

1. How do SQL and MongoDB query features compare and what similarities can be identified between them?
2. What steps are involved in loading the MovieLens dataset into MongoDB?
3. How does MongoDB implement MapReduce functionality, and what are its advantages?

*****

5

INDEXING AND ORDERING DATA SETS

Unit Structure

5.0 OBJECTIVES

- Develop indexes to improve query performance.
- Create and maintain indexes in document databases and column-family databases.
- Organize NoSQL data sets efficiently.
- Make informed design choices to create optimal indexes and ordering patterns.

5.1 INTRODUCTION

In this chapter, we'll take steps to ensure that your queries are optimized for speed and efficiency. In relational databases, using indexes is a common way to enhance query performance. The same concept applies to NoSQL databases.

Indexes are designed to boost data access performance. They function similarly to a book's index. When you need to find a specific term or word in a book, you have two options:

Scan the entire book page by page.

Use the index at the end to locate the pages where the term or word appears and go directly to those pages.

Clearly, using the index is the more efficient choice, saving time and effort. Similarly, in a database, you can either:

Search through the entire collection or dataset item by item.

Use the index to quickly locate the relevant data.

Again, the index lookup is the preferred method. While the analogy between book indexes and database indexes is useful, it's important not to stretch the similarity too far. Book indexes cover a limited subset of terms based on free text, while database indexes apply to all data sets within a collection, created on item identifiers or specific properties.

5.2 ESSENTIAL CONCEPTS BEHIND A DATABASE INDEX

There is no universal formula for creating an index, but the most effective methods are based on a few common principles. These principles often involve hash functions and B-tree or B+-tree data structures. This section explores these concepts to provide a theoretical foundation.

A hash function is a well-defined mathematical function that converts a large, variable-sized, and complex data value into a single integer or set of bytes. The output of a hash function is known by various names, such as hash code, hash value, hash sum, and checksum. Hash codes are often used as keys for associative arrays, also known as hash maps. Hash functions are particularly useful for mapping complex database property values to hash codes for index creation.

A tree data structure organizes values in a hierarchical, tree-like manner, with links or pointers between certain nodes. A binary tree is a specific type of tree where each node has at most two children: one on the left and one on the right. A node can either be a parent, with up to two children, or a leaf, being the last node in the chain. At the base of the tree is the root node. Figure 5-1 illustrates a binary tree data structure.

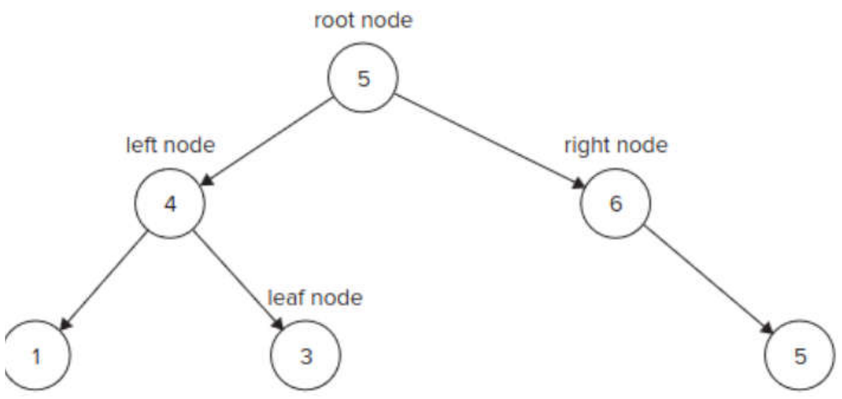


Figure 5.1

A B-tree is an extension of a binary tree, allowing each parent node to have more than two child nodes. This structure keeps the data sorted, enabling efficient search and data access. A B+-tree is a specific variant of the B-tree, where all records are stored in the leaf nodes, which are sequentially linked. B+-trees are the most commonly used tree structures for storing database indexes.

For those interested in learning more about B-trees and B+-trees, consider the following resources available online:

Wikipedia - B-tree

Semaphore Corporation - B-tree Algorithm

Wikipedia - B+ tree

For a more structured tutorial, you might refer to "Introduction to Algorithms" by Cormen, Leiserson, Rivest, and Stein, ISBN 0-262-03384-4.

Although the basic principles of indexing are similar, the implementation and application vary across different NoSQL products. In the following sections, we will explore indexing in MongoDB, CouchDB, and Apache Cassandra. We will also cover effective data sorting, which is closely related to indexing.

5.3 INDEXING AND ORDERING IN MONGO DB

MongoDB offers a wide array of options for indexing collections to improve query performance.

By default, it creates an index on the _id property for all collections. Indexing is best understood through examples.

Once set up, you should have three collections: movies, ratings, and users. To grasp the significance and impact of an index, it's essential to measure query performance with and without an index. MongoDB provides built-in tools to explain query plans and identify slow-running queries. A query plan outlines what the database server needs to do to execute a given query. To begin, use the explain plan utility to analyze query performance. For example, to retrieve all items in the ratings collection, you can run a query like this:

frustrating();

movie lens_indexation

To run explain plan for this query you can run this query:

frustrating().explain();

movielens_indexes.txt

The output of the explain plan would be something like this:

{

```
        “cursor” : “BasicCursor”,
        “nscanned” : 1000209,

        “nscannedObjects” : 1000209,
        “n” : 1000209,
        “millis” : 1549,
        “indexBounds” : {
        }
}
```

The output indicates that it took 1,549 milliseconds to return 1,000,209 documents, examining 1,000,209 items in the process. It also mentions that a BasicCursor was used.

The explain function's output is a document with several properties:

cursor: The type of cursor used to return the query result sets. It can be a basic cursor (indicating a table scan) or a B-tree cursor (indicating an index was used).

nscanned: The number of entries scanned. When an index is used, this corresponds to the number of index entries.

nscannedObjects: The number of documents scanned.

n: The number of documents returned.

millis: The time, in milliseconds, taken to execute the query.

indexBounds: The minimum and maximum index keys within which the query was matched. This field is only relevant when an index is used.

The next example demonstrates how to query a subset of the ratings collection. This collection contains rankings (on a scale of 1 to 5) for various movies by different users. To filter the ratings collection to a subset related to a specific movie, we need to correlate movie IDs in the ratings collection with names in the movies collection. We'll use the original Toy Story (Toy Story 1) as an example, but you can choose any movie.

To retrieve the document related to Toy Story, we can use a regular expression. To query all documents related to Toy Story in the movies collection, use the following approach:

```
db.movies.find({ title: /Toy Story/i })
```

This query uses a regular expression to match the title field in the movies collection, ensuring all documents related to Toy Story are retrieved

```
db.movies.find({title: /Toy Story/i});
```

movielens_indexes.txt

The output should be as follows:

```
{ "_id" : 1, "title" : "Toy Story (1995)", "genres" : [ "Animation",
"Children's", "Comedy" ] }

{ "_id" : 3114, "title" : "Toy Story 2 (1999)", "genres" : [ "Animation",
"Children's", "Comedy" ] }
```

I guess Toy Story 3 wasn't released when these ratings were compiled. That's why you don't see that in the list. Next, take the movie ID for "Toy Story", which happens to be 1, and use that to find all the relevant ratings from all the users. Before you do that, though, run the explain plan function to view how the database ran the regular expression query to find Toy Story in the movies collection. You can run the explain plan like so:

```
db.movies.find({title: /Toy Story/i}).explain();
```

movielens_indexes.txt

The output should be as follows:

```
{
        "cursor" : "BasicCursor",
        "nscanned" : 3883,
        "nscannedObjects" : 3883,
        "n" : 2,
        "millis" : 6,
        "indexBounds" : {
        }
}
```

Run a count, using db.movies.count();, on the movies collection to verify the number of documents and you will observe that it matches with the nscanned and nscannedObjects value of the query explanation. This means the regular expression query led to a table scan, which isn't efficient. The number of documents was limited to 3,883 so the query still ran fast enough and took only 6 milliseconds. In a short bit you will see how you could leverage indexes to make this query more efficient, but for now return to the ratings collection to get a subset that relates to Toy Story.

To list all ratings that relate to Toy Story (more accurately Toy Story (1995)) you can query as follows:

```
db.ratings.find({movie_id: 1});
```

movielens_indexes.txt

To see the query plan for the previous query run explain as follows:

```
db.ratings.find({movie_id: 1}).explain();
```

movielens_indexes.txt

The output should be as follows:

```
{
“cursor” : “BasicCursor”,
“nscanned” : 1000209,
“nscannedObjects” : 1000209,
“n” : 2077,
“millis” : 484,
“indexBounds” : {
}
}
```

At this stage it’s evident that the query is not running optimally because the nscanned and nscannedObjects count reads 1,000,209, which is all the documents in the collection. This is a good point to introduce indexes and optimize things.

5.4 CREATING AND USING INDEXES IN MONGODB

The ensureIndex keyword does most of the index creation magic in MongoDB. The last query filtered the ratings collection based on the movie_id so creating an index on that property should transform the lookup from table scan to B-tree index traversal. First, verify if the theory does hold good.

Create the index by running the following command:

```
db.ratings.ensureIndex({ movie_id:1 });
```

movielens_indexes.txt

This creates an index on movie_id and sorts the keys in the index in an ascending order. To create an index with keys sorted in descending order use the following:

```
db.ratings.ensureIndex({ movie_id:-1 });
```

movielens_indexes.txt

Then rerun the earlier query as follows:

```
db.ratings.find({movie_id: 1});
```

movielens_indexes.txt

Verify the query plan after that as follows:

```
db.ratings.find({movie_id: 1}).explain();
```

movielens_indexes.txt

The output should be:

```
{
"cursor" : "BtreeCursor movie_id_1",
"nscanned" : 2077,
"nscannedObjects" : 2077,
"n" : 2077,
"millis" : 2,
"indexBounds" : {
        "movie_id" : [
                [
                        1,
                        1
                ]
        ]
}
}
```

At first glance, it's evident that the number of items (and documents) looked up has drastically reduced from 1,000,209 (the total number of documents in the collection) to 2,077 (the number of documents matching the filter criteria). This improvement signifies a substantial performance boost. In algorithmic terms, the document search has transitioned from linear time complexity to constant time complexity. Consequently, the total time to run the query dropped from 484 milliseconds to just 2 milliseconds, resulting in a reduction of over 99 percent in query execution time.

The query plan's cursor value indicates that the movie_id_1 index was used. You can experiment by creating an index with keys sorted in descending order and rerunning the query and the query plan. However, before executing the query, analyze the list of indexes in the ratings collection to determine how to force a particular index.

Retrieving a list (or array) of all indexes is straightforward. You can use the following query

```
db.ratings.getIndexes();
```

Assuming there are two indexes on movie_id (one in ascending order and one in descending order), along with the default _id index, the list of indexes should include these three. The output of getIndexes is as follows:

```
[
        {
                “name” : “_id_”,
                “ns” : “mydb.ratings”,
                “key” : {
                        “_id” : 1
                        }
                },
        {
                “_id” : ObjectId(“4d02ef30e63c3e677005636f”),
                “ns” : “mydb.ratings”,
                “key” : {
                        “movie_id” : -1
                },
                “name” : “movie_id_-1”
        },
        {
                “_id” : ObjectId(“4d032faee63c3e6770056370”),
                “ns” : “mydb.ratings”,
                “key” : {
                        “movie_id” : 1
                },
                “name” : “movie_id_1”
        }
]
```

You have already created an index on movie_id using a descending order sort using the

following command:

```
db.ratings.ensureIndex({ movie_id:-1 });
```

movielens_indexes.txt

If required, you could force a query to use a particular index using the hint method. To force the descending order index on movie_id to get ratings related to “Toy Story (1995)” you can query as follows:

```
db.ratings.find({ movie_id:1 }).hint({ movie_id:-1 });
```

movielens_indexes.txt

Soon after running this query, you can verify the query plan to see which index was used and how it performed. A query plan for the last query using the descending order index on movie_id can be accessed as follows:

```
db.ratings.find({ movie_id:1 }).hint({ movie_id:-1 }).explain();
```

movielens_indexes.txt

The output of the query explain plan is as follows:

```
{
    "cursor" : "BtreeCursor movie_id_-1",
    "nscanned" : 2077,
    "nscannedObjects" : 2077,
    "n" : 2077,
    "millis" : 17,
    "indexBounds" : {
        "movie_id" : [
            [
                1,
                1
            ]
        ]
    }
}
```

The explain plan output confirms that the descending order index on movie_id, identified by movie_id_-1, was utilized. It also shows that, similar to the ascending order index, the descending order index accessed only 2,077 items.

However, there's an interesting detail in the output. Despite using an index and scanning only a limited number of documents, it took 17 milliseconds to return the result set. This is significantly less than the 484 milliseconds required for a table scan but notably more than the 2 milliseconds taken by the ascending order index. This discrepancy might be because the movie_id 1 is at the beginning of the ascending order list, and the results might have been cached from a previous query.

Ascending order indexes do not always outperform descending order indexes when accessing documents at the beginning of the list, nor do descending order indexes consistently outperform ascending order indexes when accessing documents at the end of the list. Typically, both index types perform similarly for items near the middle of the list. To verify this, you can use both indexes to search for ratings for a movie whose movie_id is at the other end.

The movie_id field in the ratings collection corresponds to the _id field in the movies collection. The _id field, like the movie_id field, has integer values. Therefore, finding the movie_id at the top of the descending order sort is equivalent to finding the maximum value for the _id field in the movies collection. One way to determine the maximum _id value in the movies collection is to sort it in descending order as follows:

```
db.movies.find().sort({ _id: -1 }).limit(1);
```

This query sorts the _id field in descending order and returns the first document, which will have the highest _id value. You can then use this movie_id to test the performance of both the ascending and descending order indexes.

```
db.movies.find().sort({ _id:-1 });
```

movielens_indexes.txt

The JavaScript console returns only 20 documents at a time so it's easy to find the maximum value, which is 3,952, at a quick glance. If you are running this query using a language API or any other mechanism you may want to limit the number of items in the result. Because only one item is required, you could simply run the query like so:

```
db.movies.find().sort({ _id:-1 }).limit(1);
```

movielens_indexes.txt

The movie_id 3952 corresponds to Contender, The (2000). To get ratings for the movie The Contender, you could use either the ascending or the descending ordered index on movie_id. Because the objective here is to analyze how both of these indexes perform for an item that satisfies boundary conditions, you can use both of them one after the other. In both cases you can also run the query plans. The query and query plan commands for the ascending order movie_id index are as follows:

```
db.ratings.find({ movie_id:3952 }).hint({ movie_id:1 });

db.ratings.find({ movie_id:3952 }).hint({ movie_id:1 }).explain();
```

movielens_indexes.txt

The output of the query plan is like so:

```
{
        "cursor" : "BtreeCursor movie_id_1",
        "nscanned" : 388,
        "nscannedObjects" : 388,
        "n" : 388,
        "millis" : 2,
        "indexBounds" : {
                "movie_id" : [
                        [
                                3952,
                                3952
                        ]
                ]
        }
}
```

The query and query plan commands for the descending order movie_id index is as follows:

```
db.ratings.find({ movie_id:3952 }).hint({ movie_id:-1 });

db.ratings.find({ movie_id:3952 }).hint({ movie_id:-1 }).explain();

{
        "cursor" : "BtreeCursor movie_id_-1",
        "nscanned" : 388,
        "nscannedObjects" : 388,
        "n" : 388,
        "millis" : 0,
        "indexBounds" : {
                "movie_id" : [
                        [
                                3952,
                                3952
                        ]
                ]
        }
}
```

movielens_indexes.txt

From multiple runs of these queries, it seems that values at the extremes don't consistently benefit from indexes that start at the corresponding end. However, it's important to remember that the query plan output is not idempotent. Each execution can yield a different result. For instance, values might be cached, so the underlying data structures may not be accessed on subsequent runs. Additionally, for smaller data sets like the movies collection, the difference is negligible, and extraneous overheads such as I/O lag can significantly affect response time. Generally, for large data sets, a sort order that aligns with the queried item should be used.

Occasionally, after making numerous modifications to a collection, it can be beneficial to rebuild indexes. To rebuild all indexes for the ratings collection, you can run the following command:

db.ratings.reIndex();

This command will rebuild all indexes on the ratings collection, ensuring they are optimized and up-to-date.

:

db.ratings.reIndex();

movielens_indexes.txt

You can alternatively use the runCommand to reindex:

db.runCommand({ reIndex:'ratings' });

movielens_indexes.txt

Rebuilding indexes is not required in most cases unless the size of the collection has changed in a considerable way or the index seems to be occupying an unusually large amount of disk space Sometimes, you may want to drop and create new indexes instead of rebuilding the ones that exist. Indexes can be dropped with the dropIndex command:

db.ratings.dropIndex({ movie_id:-1 });

movielens_indexes.txt

This command drops the descending order movie_id index. You can also drop all indexes if need

be. All indexes (except the one of the _id field) can be dropped as follows:

db.ratings.dropIndexes();

movielens_indexes.txt

5.4.1 Compound and Embedded Keys:

You have created indexes on only a single field or property. It is also possible to create compound indexes, for example, to create an index on movie_id and ratings fields together. The command to create such an index is:

db.ratings.ensureIndex({ movie_id:1, rating:-1 });

movielens_indexes.txt

This creates a compound index on movie_id (ascending order) and rating (descending order). You can create three more indexes out of the four possible compound indexes involving movie_id and rating. The four possibilities arise due to the potential combinations of ascending and descending order sorts for the two keys. The order of the sort can impact queries involving sorting and range queries, so it's important to consider the order when defining compound indexes for your collection.

A compound index involving movie_id and rating can be used to query documents that match both these keys, as well as for queries that match on movie_id alone. When using this index to filter documents based on movie_id alone, the behavior is similar to using a single-field index on movie_id. Compound keys are not limited to two keys; you can include as many keys as needed. For example, to create a compound index for movie_id, rating, and user_id, you would use the following command:

```
db.ratings.ensureIndex({ movie_id: 1, rating: -1, user_id: 1 });
```

This index can be used to query for the following combinations:

movie_id, rating, and user_id

movie_id and rating

movie_id

Compound indexes can also include nested (or embedded) fields. Before exploring how compound indexes involve nested fields, let's cover how to create a single index involving a nested field. To illustrate, let's use a collection of people (named people2). An element of the people2 collection is as follows:

```
{
        "_id" : ObjectId("4d0688c6851e434340b173b7"),
        "name" : "joe",
        "age" : 27,
        "address" : {
                "city" : "palo alto",
                "state" : "ca",
                "zip" : "94303",
                "country" : "us"
        }
}
```

You can create an index on the zip field of the address field as follows:

```
db.people2.ensureIndex({ "address.zip":1 });
```

movielens_indexes.txt

Next, you can create a compound index for the name and address.zip fields:

```
db.people2.ensureIndex({ name:1, "address.zip":1 });
```

movielens_indexes.txt

You can also choose the entire sub-document as the key of an index so you can create a single index

for the address field:

```
db.people2.ensureIndex({ address:1 });
```

movielens_indexes.txt

This indexes the entire document and not just the zip field of the document. Such an index can be used if an entire document is passed as a query document to get a subset of the collection. A MongoDB collection field can also contain an array instead of a document. You can index such fields as well. Now consider another example of an orders collection to illustrate how array properties can be indexed. An element of the orders collection is as follows:

```
{
"_id" : ObjectId("4cccff35d3c7ab3d1941b103"),
"order_date" : "Sat Oct 30 2010 22:30:12 GMT-0700 (PDT)",
        "line_items" : [
                {
                        "item" : {
                                "name" : "latte",
                                "unit_price" : 4
                        },
                        "quantity" : 1
                },
                {
                        "item" : {
                                "name" : "cappuccino",
                                "unit_price" : 4.25
                        },
                        "quantity" : 1
                },
                {
                        "item" : {
                                "name" : "regular",
                                "unit_price" : 2
                        },
                "quantity" : 2
            }
        ]
}
```

You could index with line_items:

db.orders.ensureIndex({ line_items:1 });

movielens_indexes.txt

When an indexed field contains an array, each element of the array is added to the index.

In addition, you could index by the item property of the line_items array:

db.orders.ensureIndex({ "line_items.item":1 });

movielens_indexes.txt

You could go one level further and index it by the name property of the item document contained in

the line_items array as follows:

db.orders.ensureIndex({ "line_items.item.name":1 });

movielens_indexes.txt

So, you could query by this nested name field as follows:

db.orders.find({ "line_items.item.name":"latte" });

movielens_indexes.txt

Run the query plan to confi rm that the cursor value used for the query is BtreeCursor line_items.item.name_1, which as you know indicates the use of the nested index.

5.4.2 Creating Unique and Sparse Indexes:

MongoDB offers various options to index documents for efficient query performance. Indexes can also serve the purpose of imposing constraints. To create a sparse index, you can explicitly specify it as follows:

db.ratings.ensureIndex({ movie_id: 1 }, { sparse: true });

A sparse index means that documents with a missing indexed field are completely ignored and left out of the index. While this can be desirable, it's important to note that a sparse index may not reference all documents in the collection.

MongoDB also supports creating unique indexes. For example, to create a unique index on the title field of the movies collection, you can use:

db.movies.ensureIndex({ title: 1 }, { unique: true });

If two items in the movies collection had the same title, a unique index would not be created unless you explicitly specified that all duplicates after the first entry be dropped. This can be done as follows:

db.movies.ensureIndex({ title: 1 }, { unique: true, dropDups: true });

If a document in the collection contains a missing value for the indexed field, a null value will be inserted in place of the missing value. Unlike a sparse index, the document will not be skipped. Additionally, if two documents are missing the indexed field, only the first one is saved; the rest would be ignored in the collection.

5.4.3 Keyword-based Search and Multikeys:

Keyword-based search and multikeys are other important aspects of MongoDB indexes. To enhance the query performance of a regular expression-based search in a text field, you can create an index like so:

db.movies.ensureIndex({ title: "text" });

This index enables a text search on the title field, improving the efficiency of queries that use regular expressions to search for specific patterns in the text

db.movies.ensureIndex({ title:1 });

In some cases, though, creating a traditional index may not be enough, especially when you don't want to rely on regular expressions and need to do a full text search. You have already seen that a field that contains an array of values can be indexed. In such instances, MongoDB creates multikeys: one for each unique value in the array. For example, you could save a set of blogposts in a collection, named blogposts, where each element could be as follows:

```
{

"_id" : ObjectId("4d06bf4c851e434340b173c3"),

"title" : "NoSQL Sessions at Silicon Valley Cloud Computing Meetup in
January

2011",

"creation_date" : "2010-12-06",

"tags" : [

        "amazon dynamo",

        "big data",

        "cassandra",
        “cloud”,
        “couchdb”,
        “google bigtable”,
        “hbase”,
        “memcached”,
        “mongodb”,
        “nosql”,
        “redis”,
        “web scale”
]
}
```

Now, you could easily create a multikey index on the tags field as follows:

```
db.blogposts.ensureIndex({ tags:1 });
```

So far it’s like any other index but next you could search by any one of the tag values like so:

```
db.blogposts.find({ tags:”nosql” });
```

This feature can be used to build out a complete keyword-based search. As with tags, you would need to save the keywords in an array that could be saved as a value of a field. The extraction of the keywords itself is not done automatically by MongoDB. You need to build that part of the system yourself. Maintaining a large array and querying through numerous documents that each hold a large array could impose a performance drag on the database. To identify and preemptively correct some of the slow queries you can leverage the MongoDB database profiler. In fact, you can use the profiler to log all the operations.

The profiler lets you define three levels:

0 — Profiler is off

1 — Slow operations (greater than 100 ms) are logged

2 — All operations are logged

To log all operations you can set the profiler level to 2 like so:

```
db.setProfilingLevel(2);
```

The profiler logs themselves are available as a MongoDB collection, which you can view using a query as follows:

```
db.system.profile.find();
```

If you have been following along until now, you have theoretically learned almost everything there is to learn about indexes and sorting in MongoDB. Next, you use the available tools to tune the query to optimal performance as you access data from your collections.

5.5 INDEXING AND ORDERING IN COUCHDB

In CouchDB, indexing is automatic and triggered for all changed data sets when they are first read after the change. This indexing mechanism is different from MongoDB's, where indexes need to be explicitly created. CouchDB follows the MapReduce style data manipulation. The map function emits key/value pairs based on the collection data, which leads to view results. When these views are accessed for the first time, a B-tree index is built from this data. Subsequent queries return data from the B-tree, and the underlying data remains untouched. This means that queries after the first one benefit from the B-tree index.

5.5.1 The B-tree Index in CouchDB:

A B-tree index scales well for large amounts of data. Despite significant data growth, the height of a B-tree remains in single digits, enabling fast data retrieval. In CouchDB, the B-tree implementation has specialized features such as MultiVersion Concurrency Control (MVCC) and an append-only design. MVCC allows multiple reads and writes to occur in parallel without the need for exclusive locking. This is similar to distributed software version control systems like GitHub, where all writes are sequenced and reads are not impacted by writes. CouchDB uses a _rev property to hold the most current revision value. Like optimistic locking, writes and reads are coordinated based on the _rev value. Therefore, each version is the latest one at the time a client starts reading the data. As documents are modified or deleted, the index in the view results is updated.

5.6 INDEXING IN APACHE CASSANDRA

Apache Cassandra is a hybrid between a column-oriented database and a pure key/value data store, incorporating ideas from Google Bigtable and Amazon Dynamo. Like column-oriented databases, Cassandra supports row-key-based order and index by default. In addition, Cassandra also supports secondary indexes. Secondary indexes support in Cassandra is explained using a simple example. The same example is revisited to explain support for secondary indexes.

To follow along, start the Cassandra server using the cassandra program in the bin directory of the Cassandra distribution. Then connect to Cassandra using the CLI as follows:

```
PS C:\applications\apache-cassandra-0.7.4> .\bin\cassandra-cli -host localhost

Starting Cassandra Client

Connected to: “Test Cluster” on localhost/9160

Welcome to cassandra CLI.

Type ‘help;’ or ‘?’ for help. Type ‘quit;’ or ‘exit;’ to quit.
```

When your setup is complete, make CarDataStore the current keyspace as follows:

```
[default@unknown] use CarDataStore;

Authenticated to keyspace: CarDataStore
```

Use the following command to verify that the data you added earlier exists in your local Cassandra data store:

```
[default@CarDataStore] get Cars[‘Prius’];

=> (column=make, value=746f796f7461, timestamp=1301824068109000)

=> (column=model, value=70726975732033, timestamp=1301824129807000)

Returned 2 results.
```

The Cars column-family has two columns: make and model. To make querying by values in the make column more efficient, create a secondary index on the values in that column. Since the column already exists, modify the definition to include an index. You can update the column-family and column definition as follows:

```
[default@CarDataStore] update column family Cars with comparator=UTF8Type

... and column_metadata=[{column_name: make, validation_class: UTF8Type,

index_type: KEYS},

... {column_name: model, validation_class: UTF8Type}];

9f03d6cb-7923-11e0-aa26-e700f669bcfc

Waiting for schema agreement...

... schemas agree across the cluster
```

cassandra_secondary_index.txt

The update command created an index on the column make. The type of index created is of type KEYS. Cassandra defi nes a KEYS type index, which resembles a simple hash of key/value pairs. Now, query for all values that have a make value of toyota. Use the familiar SQL-like syntax as follows:

```
[default@CarDataStore] get Cars where make = ‘toyota’;

-------------------

RowKey: Prius

=> (column=make, value=toyota, timestamp=1301824068109000)

=> (column=model, value=prius 3, timestamp=1301824129807000)

RowKey: Corolla

=> (column=make, value=toyota, timestamp=1301824154174000)

=> (column=model, value=le, timestamp=1301824173253000)

2 Rows Returned.
```

cassandra_secondary_index.txt

Try another query, but this time fi lter the Cars data by model value of prius 3 as follows:

```
[default@CarDataStore] get Cars where model = ‘prius 3’;

No indexed columns present in index clause with operator EQ
```

cassandra_secondary_index.txt

The query that filters by make works smoothly but the one that fi lters by model fails. This is because there is an index on make but not on model. Try another query where you combine both make and model as follows:

```
[default@CarDataStore] get Cars where model = ‘prius 3’ and make =
‘toyota’;

-------------------

RowKey: Prius

=> (column=make, value=toyota, timestamp=1301824068109000)

=> (column=model, value=prius 3, timestamp=1301824129807000)

1 Row Returned.
```

cassandra_secondary_index.txt

The index works again because at least one of the filter criteria has an indexed set of values. The example at hand doesn’t have any numerical values in its columns so showing a greater-than or less- than filter is not possible. However, if you did want to leverage a filter for such an inequality comparator- based query then you are going to be out of luck. Currently, the KEYS index does not have the capability to perform range queries. Range queries via indexes may be supported in the future if Cassandra includes a B-tree, or a similar index type. The rudimentary KEYS index isn’t sufficient for range queries.

5.7 SUMMARY

In this chapter, you explored the details of indexing documents and their fields in MongoDB. You also learned about the automatic view indexing in CouchDB. A prominent theme that emerged was that both databases support indexes, and these indexes aren't drastically different from those in relational databases.

You also gained insights into special features, such as how arrays in MongoDB are indexed as multi-keys, and how CouchDB automatically indexes all documents that have changed since the last read.

In addition to indexes in document databases, you learned about indexing capabilities in Apache Cassandra, a popular column-family database.

5.8 REVIEW QUESTIONS

Q1. How does MongoDB index arrays, and what benefit does this provide?

Q2. Explain the automatic view indexing mechanism in CouchDB and its impact on query performance.

Q3. What distinguishes Apache Cassandra's indexing capabilities from other databases, and how does it handle secondary indexes?

*****

6

MANAGING TRANSACTIONS AND DATA INTEGRITY

Unit Structure

6.0 OBJECTIVE

- To understand the concept of RDBMS
- To study the Distributed ACID Systems
- Exploring ready-to-use NoSQL databases in the cloud
- Leveraging Google AppEngine and its scalable data store
- Using Amazon SimpleDB

6.1 INTRODUCTION

- The NoSQL databases in compact not only SQL are not arranged in tabular format and store data differently relative to relational tables.
- NoSQL databases of various types based on their data model. The main types are document, key & value, wide & column, and graph.
- They give flexible schemas and scale easily with substantial amounts of data and high user loads.
- NoSQL databases are broadly used in real-time web applications and big data, because their main advantages are high scalability and high availability.

- NoSQL databases are also the selected choice of developers, as they naturally lend themselves to an agile development paradigm by fast adapting to changing requirements.
- NoSQL databases enable the data to be stored in ways that are more intuitive and easier to understand, or closer to the use of the data is used by applications with little transformations required when storing or retrieving using NoSQL-style APIs.
- Moreover, NoSQL databases can take full benefit of the cloud to deliver zero downtime

6.2 MANAGING TRANSACTIONS AND DATA INTEGRITY

- Transaction management focuses on guaranteeing that transactions are correctly stored in the database.
- The **transaction manager** is the member of a DBMS that processes transactions. A **transaction** is a sequence of behaviors to be taken on the database such that they must be entirely completed or entirely aborted.
- A transaction is a **logical part of work**. All its elements must be processed else the database will be inconsistent.
- For example, with a sale of a product, the transaction consists of at least two parts: an update to the inventory on hand, and an update to the customer data for the items sold in order to bill the customer after.
- Updating only the inventory or only the customer information would create a database absence of integrity and an inconsistent database.

Transaction managers are designed to achieve the **ACID** (atomicity, consistency, isolation, and durability) concept. These attributes are:

1. **Atomicity:** If a transaction has two or more single pieces of information, either all of the pieces are committed or none are.
2. **Consistency:** Either a transaction creates a reasonable new database state or, if any failure occurs, the transaction manager returns the database to its earlier state.
3. **Isolation:** A transaction in process and not now committed must remain isolated from any other transaction.
4. **Durability:** Committed data are saved by the DBMS so that, at the time of a failure and system recovery, these data are available in their correct state.
5. Transaction atomicity needs all transactions to be processed on an **all or nothing** basis and that any group of transactions is **serializable**.

6. When a transaction is executed, either all its changes to the database are completed, else none of the changes are committed.

7. The entire unit of work must be processed. If a transaction is terminated before it is completed, the transaction manager must undo the executed actions to restore the database to its previous state before the transaction commences.

8. If a transaction is successfully completed, it does not require to be undone. For efficiency, transactions should be no bigger than necessary to ensure the integrity of the database.

- For example, in accounting management, a debit and credit would be a comparable transaction, because this is the minimum amount of work needed to hold the books in balance.
- Serializability connects to the execution of a set of transactions. An interleaved execution schedule is serializable if its result is equivalent to a non interleaved schedule.
- Interleaved operations are frequently used to increase the efficiency of computing resources, so it is not unusual for the components of numerous transactions to be interleaved.
- Interleaved transactions cause problems when they involve each other and, as a result, knows the correctness of the database.
- The ACID concept is essential to concurrent update control and recovery after a transaction failure.

6.3 RDBMS AND ACID

- RDBMS stands for **R**elational **D**atabase **M**anagement **S**ystem. RDBMS is the justification for SQL, and for all latest database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.
- A Relational database management system (RDBMS) is a database management system (DBMS) that depends on the relational model as introduced by E. F. Codd in 1970.
- The data in an RDBMS is stored in database is called the tables. This table is mainly a collection of associated data entries and its collection of numerous columns and rows.
- Every table is divided up into smaller entities called fields. A field is a column in a table that is developed to maintain specific information about every record in the table.
- Example CUSTOMERS table consists of different fields like id, name, age, Salary, City and Country.

- A record means a row of data is each individual entry that found in a table. For example, there are seven records in the above CUSTOMERS table.
- A column is a vertical unit or entity in a table that contains all information related with a specific field in a table.
- A NULL value in a table is a value in a field that appears to be blank or does not exist, which means a field with a NULL value is the same as a field with no value.
- Constraints are the rules mandatory on data columns on a table. These are used to restrict the type of data that can go into a table. This makes sure the accuracy and reliability of the data in the database.

S.No.	**Constraints**
1.	**NOT NULL Constraint** Ensures that a column cannot have a NULL value.
2.	**DEFAULT Constraint** Provides a default value for a column when none is specified.
3.	**UNIQUE Key** Ensures that all the values in a column are different.
4.	**PRIMARY Key** Uniquely identifies each row/record in a database table.
5.	**FOREIGN Key** Uniquely identifies a row/record in any another database table.
6.	**CHECK Constraint** Ensures that all values in a column satisfy certain conditions.
7.	**INDEX Constraint** Used to create and retrieve data from the database very quickly.

Data Integrity:

The following categories of data integrity exist with each RDBMS −

Entity Integrity	This ensures that there are no duplicate rows in a table.
Domain Integrity	Enforces valid entries for a given column by restricting the type, the format, or the range of values.

Referential integrity	Rows cannot be deleted, which are used by other records
User-Defined Integrity	Enforces some specific business rules that do not fall into entity, domain or referential integrity.

Database Normalization:

- Database normalization is the process of competently organizing data in a database. There are two explanations for this normalization process.
- Removing redundant data, for example, storing the same data on multiple tables.
- It means that data dependencies make sense.
- Both these explanations are worthy objective as they reduce the amount of space or size a database consumes and ensures that data is logically stored.
- Normalization collection of a series of guidelines that help direct you in creating a good database structure.
- Normalization guidelines are separated into normal forms, think of a form as the format or the way a database structure is planned.
- The main aim of normal forms is to organize or store the database structure, so that it complies with the rules of first normal form, then second normal form and in the end the third normal form.
- It is our choice to take it ahead and go to the Fourth Normal Form, Fifth Normal Form and so on, but in general, the Third Normal Form is more than sufficient for a normal database Application.

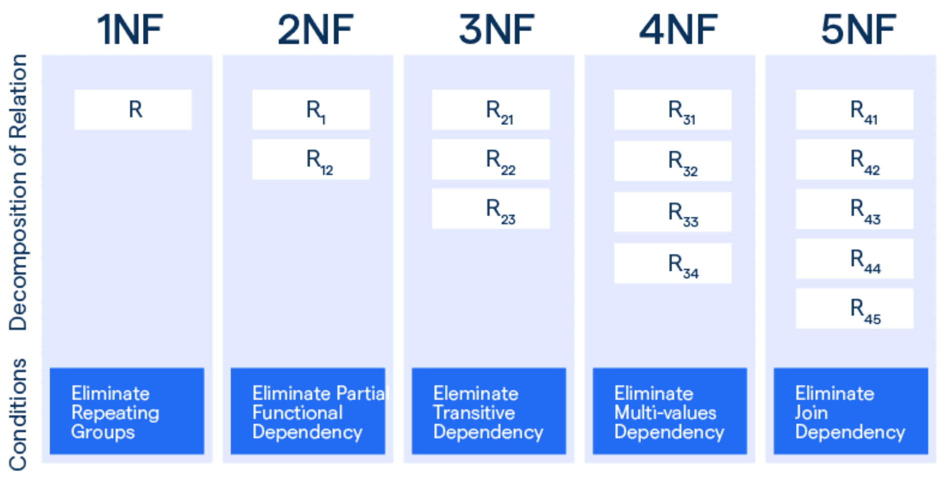

1NF
2NF
3NF
4NF
5NF
Decomposition of Relation
Conditions
R
R1
R12
R21
R22
R23
R31
R32
R33
R34
R41
R42
R43
R44
R45
Eliminate Repeating Groups
Eliminate Partial Functional Dependency
Eleminate Transitive Dependency
Eliminate Multi-values Dependency
Eliminate Join Dependency

Fig. 1 Normalization

ACID Properties:

A transaction is a very small part of a program and it may aim to contain several low level tasks. A transaction in a database system must preserve Atomicity, Consistency, Isolation, and Durability jointly known as ACID properties in order to ensure accuracy, completeness, and data integrity.

1. Atomicity:

- This property defines that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none.
- There must be no state in a database where a transaction remains partially completed.
- States should be defined either advance the execution of the transaction or after the execution or abortion or failure of the transaction.

2. Consistency:

- The database must remain in a consistent state after a particular transaction.
- No transaction should have any negative effect on the data residing in the database.
- If the database was in a compatible state before the execution of any transaction, it must also remain consistent after the execution of the transaction.

3. Durability:

- The database should be durable enough to hold all its latest updates even if the system is unsuccessful or restarts.
- If a transaction updates a chunk of data in a database and commits, then the database will hold the altered data.
- If a transaction commits / succeeds but the system fails / aborts before the data could be written on to the disk, then that data will be updated once the system springs back into action.

4. Isolation:

- In a database system where multiple transactions are being executed simultaneously and in parallel, the property of isolation defines that all the transactions will be carried out and executed as if it is the only transaction in the system.
- No transaction will influence the existence of any other transaction.

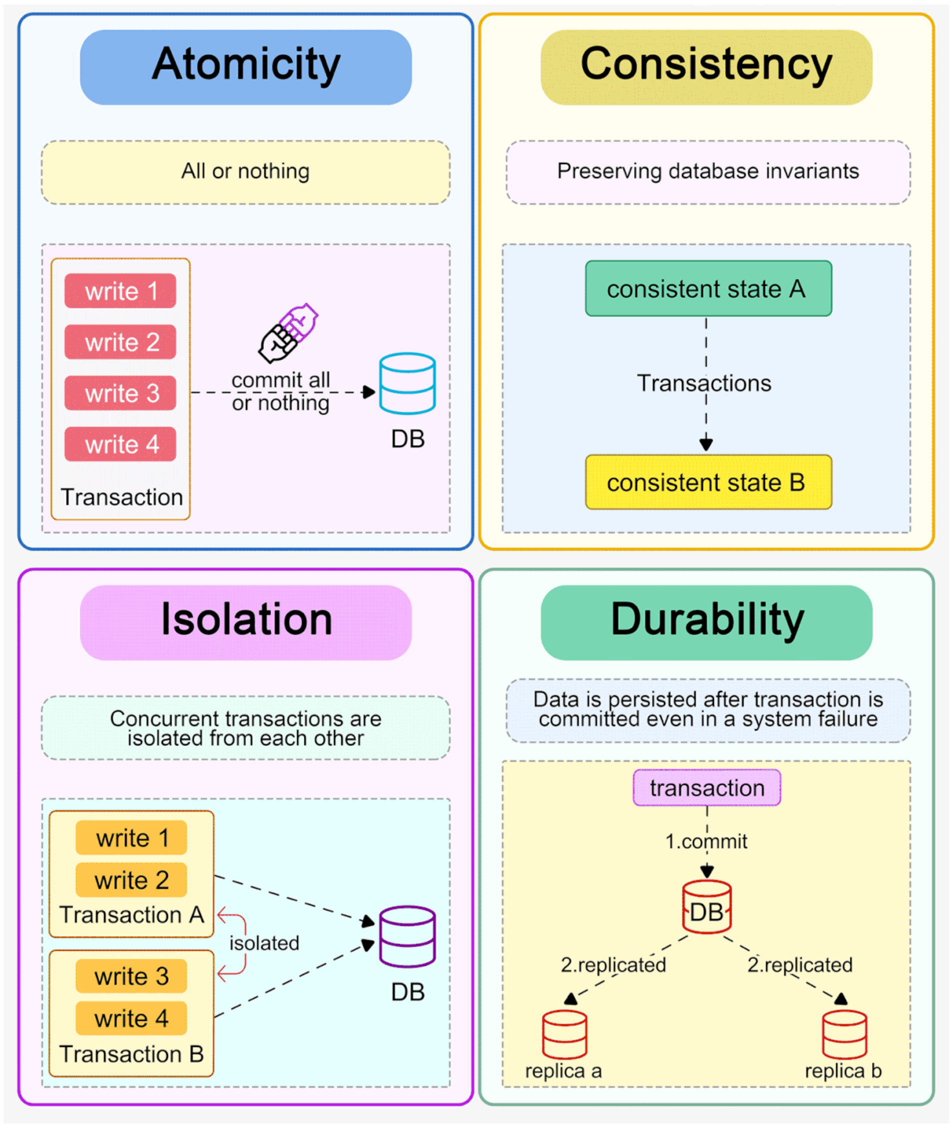


Fig. 2 ACID

6.4 DISTRIBUTED ACID SYSTEMS

- Distributed ACID transactions are ACID compliant transactions that alter multiple rows in more than one fragment usually distributed across multiple nodes.
- A transaction is a unit of work performed within a part of a database, often consisting of multiple operations.
- Like all forms of ACID, a distributed ACID transaction has four key properties:
  - **Atomicity:** All operations in a transaction are consider as a one atomic unit. All are performed or none of them are performed.
  - **Consistency:** The database is always in a consistent state or an internal state.

- **Isolation:** find how and when changes made by one transaction become visible to others.
- **Durability:** Enables all transaction results to permanently remain in the system. Any modifications or changes must continue even in case of power loss or system failures.

- There are various types of distributed databases. The most commonly used distributed databases consist of common NoSQL databases like Apache Cassandra, and distributed SQL databases, like YugabyteDB.
- Few distributed databases support ACID transactions in a limited trend, while others fully assist distributed ACID transactions. ACID transactions can be classified into three types:

1. Single row ACID
2. Single shard ACID
3. Distributed ACID transactions

- Only distributed ACID transactions are fully distributed and are the default transaction for distributed SQL databases

6.5 UPHOLDING CAP

- CAP Theorem is a idea that a distributed database system can only have 2 of the 3: Consistency, Availability and Partition Tolerance.

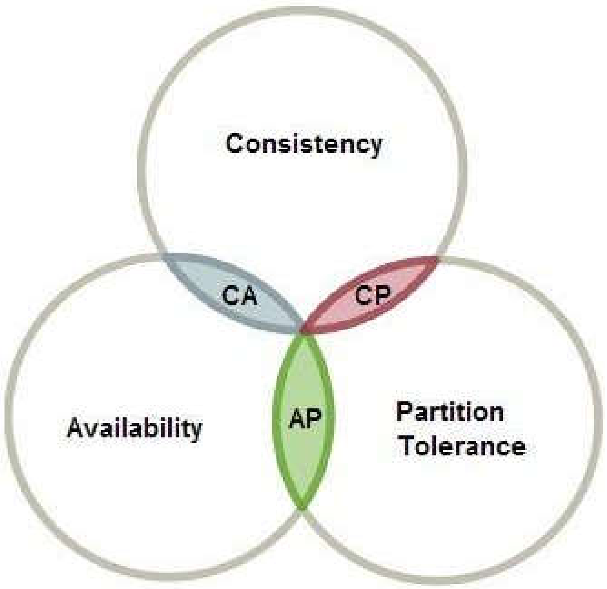


Fig. 3 CAP

- CAP Theorem is very essential in the Big Data world, mainly when we need to make trade offs between the three, based on our special use case.

Partition Tolerance:

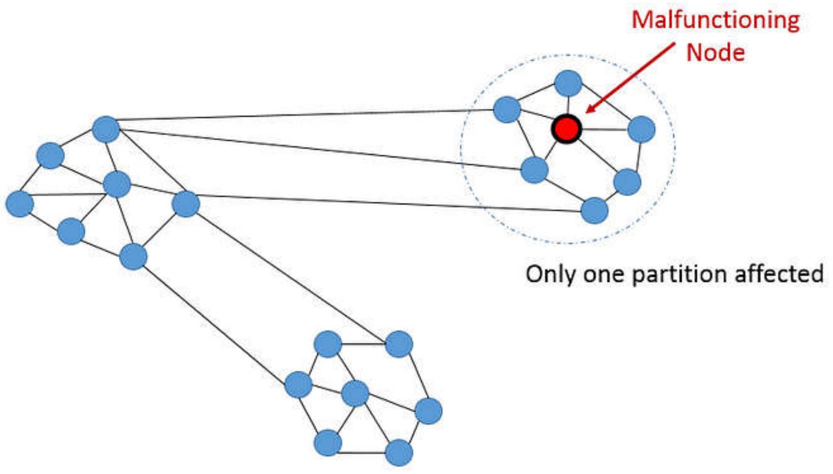


Fig. 4 Partition Tolerance

- This condition shows that the system continues to run, regardless of the number of messages being delayed by the network between two nodes.
- A system that is partition tolerant can encourage any amount of network failure that doesn't result in a failure of the entire network.
- Data records are properly replicated across combinations of nodes and networks to keep the system up through fitful outages.
- When trading with modern distributed systems, Partition Tolerance is not an option. It's a necessity. For this we have to trade between Consistency and Availability.

High Consistency:

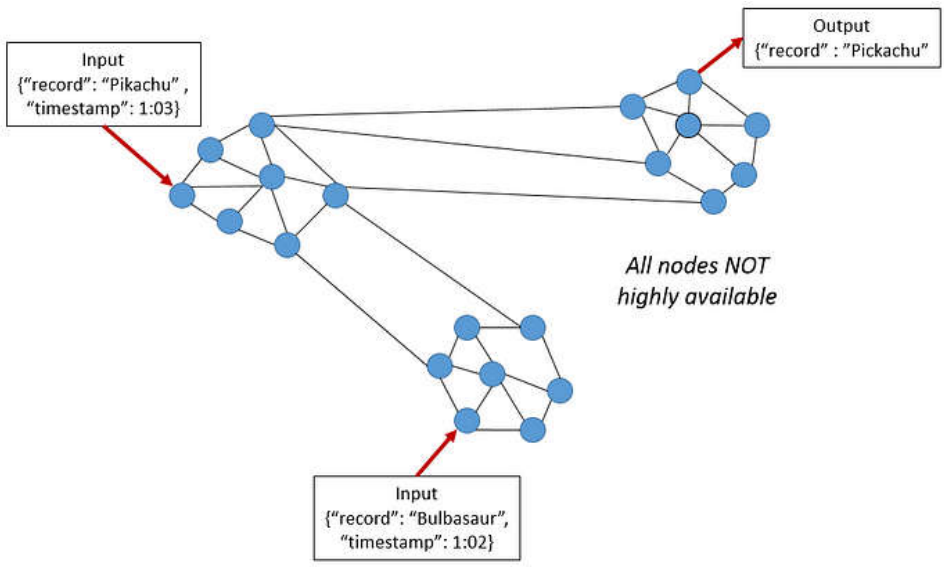

Input
{"record": "Pikachu" , "timestamp": 1:03}
Output
{"record" : "Pickachu"
All nodes NOT highly available
Input
{"record": "Bulbasaur", "timestamp": 1:02}

Fig. 5 High Consistency

- This condition defines that all nodes show the same data at the same time.
- Simply put, performing a read operation will return the value of the most recent write operation affecting all nodes to return the same data.
- A system has stability if a transaction starts with the system in a consistent state, and ends with the system in a consistent state.
- In this model, a system can shift into an inconsistent state during a transaction, but the entire transaction gets rolled back if there is a problem or error during any stage in the process.
- In Fig.5 , we have 2 different records ("Bulbasaur" and "Pikachu") at different timestamps.
- The output on the third partition is "Pikachu", the latest input. but, the nodes will need time to update and will not be Available on the network as frequently.

High Availability:

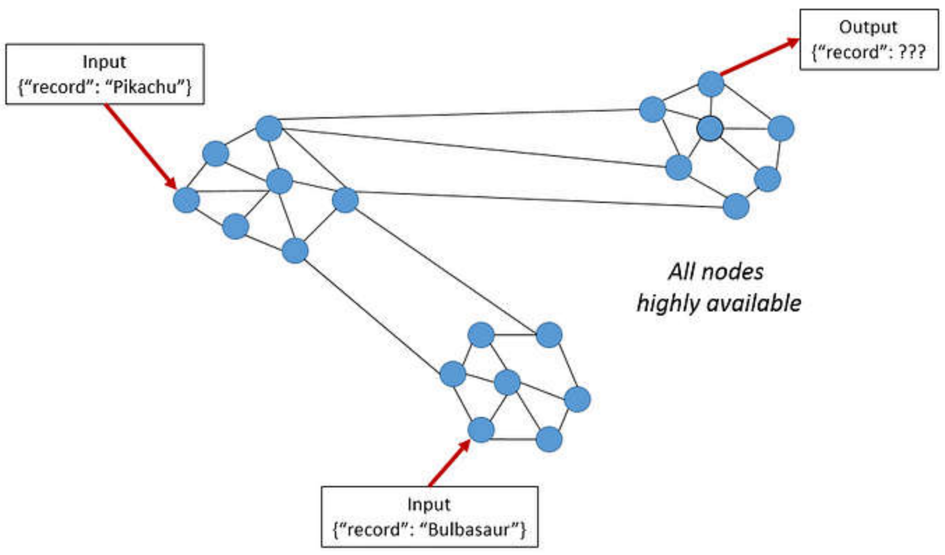


Fig. 6 High Availability

- This condition defines that every request gets a response on success or failure.
- Achieving availability in a distributed system requires that the system remains working 100% of the time.
- Every client gots a response, regardless of the state of any individual node in the system.
- This metric is trivial to calculate, either can submit read/write commands, or cannot.

- Hence, the databases are time free as the nodes need to be present online at all times.
- This means that, dissimilar to the previous example, we do not know if "Pikachu" or "Bulbasaur" was added first.
- The output could be one. Hence why, high availability isn't viable when analyzing streaming data at high frequency.

6.6 CONSISTENCY IMPLEMENTATIONS USING NOSQL IN THE CLOUD

- The Most current age popular applications, like Google and Amazon, have achieved high availability and the ability to concurrently service millions of users by scaling out horizontally among multiple machines, lay out across multiple data centers.
- Success stories of large-scale web applications like those from Google and Amazon have manifested that in horizontally scaled environments, NoSQL solutions tend to shine over their relational counterparts.
- Horizontally scaled environments available on-demand / needs and provisioned as required have been christened as the "cloud."
- If scalability and availability is a priority, NoSQL in the cloud is maybe the ideal setup.
- Many cloud service enablers exist and multiple NoSQL products are available. In many instances, like Amazon EC2 (Elastic Compute Cloud), you have the choice to install any NoSQL product you want to use.
- Google revolutionized the cloud computing landscape by developing a services-ready, easy-to-use infrastructure. However, Google wasn't the first to launch cloud contributions. Amazon EC2 was already an established player in the market when Google first made its service public.
- Google's model was so suitable, though, that its cloud platform, the Google App Engine (GAE), has seen universal and rapid adoption in a short time frame. The app engine isn't without its share of limitations.
- Its sandboxed environment and lack of support for long-running processes are surrounded by a few of its aspects that are much execrable.

6.7 GOOGLE APP ENGINE DATA STORE

- The Google App Engine (GAE) provides a sandboxed deployment environment for applications, which are written using either the

Python programming language or a language that can run on a Java Virtual Machine (JVM).

- Google provides developers with a set of rich APIs and an SDK to build applications for the app engine.
- To explain the data store features and the available APIs for data modeling, I first cover all that relates to the Python SDK for the app engine.

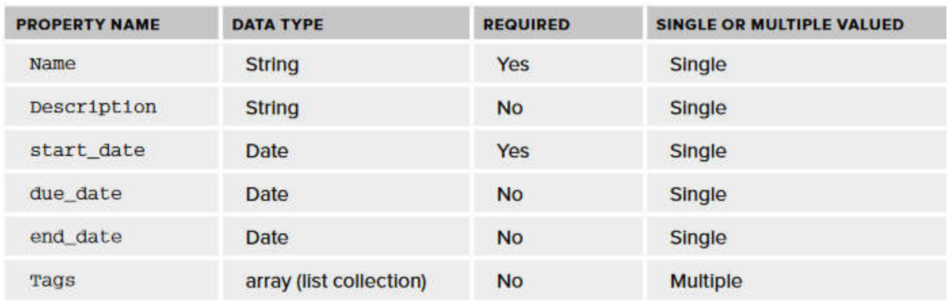
GAE Python SDK: Installation, Setup, and Getting Started:

- To get started you need to install Python and the GAE Python SDK. You can download Python from python.org and the GAE Python SDK is available online at http://code.google.com/appengine/downloads.html#Google_App_Engine_SDK_for_Python. Detailed installation instructions are beyond the scope of this chapter but installation of both Python and GAE Python SDK on all supported environments is fairly easy and straightforward.
- If you still run into trouble while setting up your environment, just Google for a solution to your problem and like most developers you won't be disappointed.
- Although this chapter exclusively focuses on the GAE data store, you will benefit from understanding the essentials of application development on the app engine.
- For the Python SDK, spend a little while reading through the tutorial titled "Getting Started: Python," which is available online at http://code.google.com/appengine/docs/python/gettingstarted/. Applications built on GAE are web applications. The getting started tutorial explains the following:

Task Manager: A Sample Application:

- Consider a simple task management application in which a user can define a task, track its status, and check it as done once completed.
- To define a task, the user needs to give it a name and a description.
- Tags can be added to categorize it and start, and expected due dates could be specified. Once completed, the end date can be recorded.
- Tasks belong to a user and in the first version of the application they are not shared with anyone other than the owner.
- To model a task, it would be helpful to list the properties, specify the data type for each property, state whether it's required or optional, and mention whether it is single or multiple valued.

Table 1 lists a task's properties and its characteristics.



PROPERTY NAME	DATA TYPE	REQUIRED	SINGLE OR MULTIPLE VALUED
Name	String	Yes	Single
Description	String	No	Single
start_date	Date	Yes	Single
due_date	Date	No	Single
end_date	Date	No	Single
Tags	array (list collection)	No	Multiple

Table. 1 Properties of a Task

Here, the Task class is modified to specify constraints:

```
import datetime
from google.appengine.ext import db
class Task(db.Model):
name = db.StringProperty(required=True)
description = db.StringProperty()
start_date = db.DateProperty(required=True)
due_date = db.DateProperty()
end_date = db.DateProperty()
tags = db.StringListProperty()
taskmanager GAE project
Available for
download on
Wrox.com
Available for
download on
Wrox.com
```

- ORM, or Object-Relational Mapping, provides a bridge between the object- oriented programming and the relational database worlds. A number of validation options are available. For example, required=True makes a property value mandatory. The argument choices=set([“choice1”,“choice2”, “choice3”, “choice4”]) restricts the value to members of the defined set. Custom validation logic defined in a function can be passed as a value to the validator argument of a particular property class.

- GAE uses Google's Bigtable as the data store. Bigtable is a sorted, ordered, distributed sparse column-family-oriented map, which imposes little restrictions on the number or types of columns in a column-family or the data type of the values stored in these columns. Also, Bigtable allows sparse data sets to be saved effectively, thereby allowing two rows in a table to have completely different sets of columns. It also permits different value types for the same columns. In other words, in a single data store, two entities of the same kind (for example, Task) can have different sets of properties or two entities of the same kind can have a property (identified by the same name) that can contain different types of data. The data modeling API provides a level of structure on top of the more accommodating Bigtable. The data modeling API provides an application-level restriction on the property data types, its values sets, and the relationship among them. In the simple example that depicts a "Task" entity, a Python class named Task defines the data model

- The GAE data store can be thought of as an object store where each entity is an object. That means data store entities or members could be instances of a Python class, like Task. The class name, Task, translates to an entity kind.

6.8 AMAZON SIMPLEDB

- Amazon SimpleDB is a ready-to-run database alternative to the app engine data store. It's elastic and is a fully managed database in the cloud.
- The two data stores app engine data store and SimpleDB are quite different in their API as well as the internal fabric but both provide you a highly scalable and grow-as-you-use model to a data store.

Enabling SimpleDB service for AWS account:

Once you have successfully set up an AWS account, you must follow these steps to enable the SimpleDB service for your account:

1. Log in to your AWS account.
2. Navigate to the SimpleDB home page—http://aws.amazon.com/simpledb/.
3. Click on the Sign Up For Amazon SimpleDB button on the right side of the page.
4. Provide the requested credit card information and complete the signup process.

You have now successfully set up your AWS account and enabled it for SimpleDB.

- All communication with SimpleDB or any of the Amazon web services must be through either the SOAP interface or the Query/ReST interface. The request messages sent through either of these interfaces are digitally signed by the sending user in order to ensure that the messages have not been tampered within transit, and that they really originate from the sending user. Requests that use the Query/ReST interface will use the access keys for signing the request, whereas requests to the SOAP interface will use the x.509 certificates.

Your new AWS account is associated with the following items:

- A unique 12-digit AWS account number for identifying your account.
- AWS Access Credentials are used for the purpose of authenticating requests made by you through the ReST Request API to any of the web services provided by AWS. An initial set of keys is automatically generated for you by default. You can regenerate the Secret Access Key at any time if you like. Keep in mind that when you generate a new access key, all requests made using the old key will be rejected.
    - An Access Key ID identifies you as the person making requests to a web service.
    - A Secret Access Key is used to calculate the digital signature when you make requests to the web service.
    - Be careful with your Secret Access Key, as it provides full access to the account, including the ability to delete all of your data.
- All requests made to any of the web services provided by AWS using the SOAP protocol use the X.509 security certificate for authentication. There are no default certificates generated automatically for you by AWS. You must generate the certificate by clicking on the Create a new Certificate link, then download them to your computer and make them available to the machine that will be making requests to AWS.
    - Public and private key for the x.509 certificate. You can either upload your own x.509 certificate if you already have one, or you can just generate a new certificate and then download it to your computer.

Query API and authentication:

- There are two interfaces to SimpleDB. The SOAP interface uses the SOAP protocol for the messages, while the ReST Requests uses HTTP requests with request parameters to describe the various SimpleDB methods and operations.
- In this book, we will be focusing on using the ReST Requests for talking to SimpleDB, as it is a much simpler protocol and utilizes straightforward HTTP-based requests and responses for communication, and the requests are sent to SimpleDB using either a HTTP GET or POST method.
- The ReST Requests need to be authenticated in order to establish that they are originating from a valid SimpleDB user, and also for accounting and billing purposes.

- This authentication is performed using your access key identifiers.
- Every request to SimpleDB must contain a request signature calculated by constructing a string based on the Query API and then calculating an RFC 2104-compliant HMAC-SHA1 hash, using the Secret Access Key.

The basic steps in the authentication of a request by SimpleDB are:

- You construct a request to SimpleDB.
- You use your Secret Access Key to calculate the request signature, a Keyed-Hashing for Message Authentication code (HMAC) with an SHA1 hash function.
- You send the request data, the request signature, timestamp, and your Access Key ID to AWS.
- AWS uses the Access Key ID in the request to look up the associated Secret Access Key.
- AWS generates a request signature from the request data using the retrieved Secret Access Key and the same algorithm you used to calculate the signature in the request.
- If the signature generated by AWS matches the one you sent in the request, the request is considered to be authentic. If the signatures are different, the request is discarded, and AWS returns an error response. If the timestamp is older than 15 minutes, the request is rejected.

6.9 SUMMARY

- Transaction management focuses on ensuring that transactions are correctly recorded in the database.
- Transaction atomicity requires that all transactions are processed on an all-or-nothing basis and that any collection of transactions is serializable.
- RDBMS stands for Relational Database Management System. RDBMS is the basis for SQL, and for all modern database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.
- Database normalization is the process of efficiently organizing data in a database. There are two reasons for this normalization process.A transaction is a unit of work performed within a database, often composed of multiple operations.
- Most current-generation popular applications, like Google and Amazon, have achieved high availability and the ability to concurrently service millions of users by scaling out horizontally among multiple machines, spread across multiple data centers.

6.10 REFERENCE FOR FURTHER READING

- QL & NoSQL Databases, Andreas Meier · Michael Kaufmann, Springer Vieweg, 2019
- Professional NoSQL by Shashank Tiwari, Wrox-John Wiley & Sons, Inc, 2011

6.11 UNIT END EXERCISES

1. Write a short note on RDBMS and ACID.
2. What do you understand about NoSQL?
3. Explain the Distributed ACID Systems?
4. Write a short note on Upholding CAP?

*****