Unit III

Structured/Unstructured Data

Structured Data

- Data that resides in a fixed field within a record or file is called structured data. This includes data contained in relational databases.

- Structured data first depends on creating a data model – a model of the types of business data that will be recorded and how they will be stored, processed and accessed.

- This includes defining what fields of data will be stored and how that data will be stored: data type (numeric, alphabetic, name, date, address) and any restrictions on the data input (number of characters; restricted to certain terms such as Mr., Ms. or Dr.; M or F).

- Structured data has the advantage of being easily entered, stored, queried and analyzed.

- **Structured data is** often managed using Structured Query Language (SQL) – a programming language created for managing and querying data in relational database management systems. Originally developed by IBM in the early 1970s and later developed commercially by Relational Software, Inc. (now Oracle Corporation).

- **Unstructured data** is all those things that can't be so readily classified : photos and graphic images, videos, streaming instrument data, webpages, pdf files, PowerPoint presentations, emails, blog entries, wikis and word processing documents, books, journals, documents, metadata, health records, audio, analog **data**, images, files, body of an e-mail message.

- **Semi-Structured data** is a cross between the two. It is a type of structured data, but **lacks the strict data model structure.** With semi-structured data, tags or other types of markers are used to identify certain elements within the data, but the data doesn’t have a strict structure.

- For example, word processing software now can include metadata showing the author's name and the date created, with the bulk of the document just being unstructured text.

- **Emails** have the sender, recipient, date, time and other fixed fields added to the unstructured data of the email message content and any attachments.

- Photos or other graphics can be tagged with keywords such as the creator, date, location and keywords, making it possible to organize and locate graphics. **XML and other markup languages** are often used to manage semi-structured data.

Structured, Semi-structured, and Unstructured data

- **Structured data**
  - Information stored DB
  - Strict format
  - **Example-Databases, DataWarehouse,Enterprise systems(ERP)**

- **Semi-structured data**
  - Data may have certain structure but not all information collected has identical structure
  - Some attributes may exist in some of the entities of a particular type but not in others
  - **Example:** XML,E-Mail

- **Unstructured data**
  - Very limited indication of data type
    - **Example** a simple text document, Analog data,GPS Tracking information,Audio/video data.

- **Limitations for SQL database**
- **Scalability**: Users have to scale relational database on powerful servers that are expensive and difficult to handle. To scale relational database it has to be distributed on to multiple servers.

- **Complexity**: In SQL server's data has to fit into tables anyhow. If your data doesn't fit into tables, then you need to design your database structure that will be complex and again difficult to handle.

NOSQL

- A **NoSQL** or **Not Only SQL** database provides a mechanism for storage and retrieval of data that is modeled other than the tabular relations used in relational databases.

- NoSQL = "Not Only SQL"

  Not every data management/analysis problem
  is best solved exclusively using a traditional DBMS

- The data structure (e.g. key-value, graph, or document) differs from the RDBMS, and therefore some operations are faster in NoSQL and some in RDBMS.

- Motivations for this approach include simplicity of design, horizontal scaling and finer control over availability.

What is NoSQL database

- Relational databases, on the other hand, were not designed to cope with the scale and the challenges that face modern applications.

- The basic quality of NoSQL is that, it **may not require fixed table schemas, usually avoid join operations**, and typically scale horizontally.

- NoSQL includes a wide variety of different database technologies and were developed in response to a rise in the volume of data stored about users, objects and products, the frequency in which this data is accessed, and performance and processing needs.

❑ Four main types of NoSQL databases/Data Models

1)Key / Value databases:

- the model is reduced to a simple hash table which consists of key / value pairs.
- It is often easily distributed across multiple servers. As the name implies, a key-value store is a system that stores values indexed for retrieval by keys.
- These systems can hold structured or unstructured data.
- The most famous products of this group include Redis, Dynamo, and Riak,BerkeleyDB.
- **Key-value stores** are the simplest NoSQL databases. Every single item in the database is stored as an attribute name (or "key"), together with its value.

2)Column-oriented databases:

- The data are stored in sections of columns which offers more flexibility and easy aggregation.
- Rather than store sets of information in a heavily structured table of columns and rows with uniform sized fields for each record, as is the case with relational databases, column-oriented databases contain one extendable column of closely related data.
- i.e. columns are logically grouped into column families.
- RDBMS stores a single column as a continous disk entry.
- Different rows are stored in different places on disk while columnar databases store all the cells corresponding to a column as a continuous disk entry thus makes search/access faster.
- Facebook's Cassandra, BigTable from Google, and Amazon's SimpleDB ,HBase are the examples which belongs to this group.

3)Document databases:

- The data model consists of document collections where individual documents can have multiple fields, without necessarily defining a schema.
- A document store is similar to a key value store in that stored objects are character string keys.
- The difference is that the values being stored are referred to as **documents** provide some encodings like XML, JSON,BSON
- **Document databases** pair each key with a complex data structure known as a document.
- Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.
- The best known and used are **MongoDB** and CouchDB.

4)Graph databases:

- The domain model consists of vertices interconnected by edges which creates a rich graph structure.
- **Graph stores** are used to store information about networks, such as social connections. Graph stores include Neo4J, OrientDB and HyperGraphDB.
- Scalability concerns are perfectly address by Graph store.

✔ There is a large number of companies using NoSQL. To name a few :

- Google
- Facebook
- Mozilla
- Adobe
- Foursquare
- LinkedIn
- McGraw-Hill Education
- Vermont Public Radio

❖ The Benefits of NoSQL

- When compared to relational databases, NoSQL databases are more scalable and provide superior performance, and their data model addresses several issues that the relational model is not designed to address.
- Large volumes of structured, semi-structured, and unstructured data.
- Object-oriented programming that is easy to use and flexible .
- Efficient, scale-out architecture.

- NoSQL database also trades off “ACID” (atomicity, consistency, isolation and durability).

- **No schema required**: Data can be inserted in a NoSQL database without first defining a rigid database schema. This provides immense application flexibility.

- **Auto elasticity:** NoSQL automatically spreads your data onto multiple servers without requiring application assistance. Servers can be added or removed from the data layer automatically.

❖ Advantages of NoSQL database

1.) NoSQL databases generally process data faster than relational databases.

2.) NoSQL databases are also often faster because their data models are simpler.

3.) Major NoSQL systems are flexible enough to better enable developers to use the applications in ways that meet their needs.

❖ SQL vs NoSQL: High-Level Differences

- SQL databases are primarily called as Relational Databases (RDBMS); whereas NoSQL database are primarily called as non-relational or distributed database.

- SQL databases are table based databases whereas NoSQL databases are **document based, key-value pairs, graph databases or wide-column stores.**

- This means that SQL databases represent data in form of tables which consists of n number of rows of data whereas NoSQL databases are the collection of **key-value pair, documents, graph databases or wide-column stores** which do not have standard schema definitions.

- SQL databases have predefined schema whereas NoSQL databases have dynamic schema for unstructured data.

- SQL databases are **vertically scalable** whereas the NoSQL databases are **horizontally scalable**.

- SQL databases uses SQL (structured query language) for defining and manipulating the data, which is very powerful.

- In NoSQL database, queries are focused on collection of documents. Sometimes it is also called as UnQL (Unstructured Query Language). The syntax of using UnQL varies from database to database.

- **SQL database examples:** MySql, Oracle,Postgres,Sqlite and MS-SQL.

- **NoSQL database examples:** MongoDB, BigTable, Redis, RavenDb, Cassandra, Hbase, Neo4j and CouchDb

- **For the type of data to be stored:** SQL databases are not best fit for **hierarchical data storage.** But, NoSQL database fits better for the hierarchical data storage as it follows the key-value pair way of storing data similar to JSON data.

- NoSQL database are highly preferred for large data set (i.e for big data). HBase is an example.

- **For scalability:** In most typical situations, SQL databases are vertically scalable. You can manage increasing load by increasing the CPU, RAM, etc, on a single server.

- On the other hand, NoSQL databases are horizontally scalable. You can just add few more servers easily in your NoSQL database infrastructure to handle the large traffic.

- **For properties:** SQL databases emphasizes on ACID properties (Atomicity, Consistency, Isolation and Durability) whereas the NoSQL database follows the Brewers CAP theorem (Consistency, Availability and Partition tolerance)

- **For DB types:** On a high-level, we can classify SQL databases as either open-source or close-sourced from commercial vendors. NoSQL databases can be classified on the basis of way of storing data as graph databases, key-value store databases, document store databases, column store databases and XML databases.

Advantages of NoSQL

1: Elastic scaling

2: Big data

Today, the volumes of "big data" that can be handled by NoSQL systems, such as Hadoop.

3: Economics

NoSQL databases typically use clusters of cheap commodity servers to manage the exploding data and transaction volumes, while RDBMS tends to rely on expensive proprietary servers and storage systems.

The result is that the cost per gigabyte or transaction/second for NoSQL can be many times less than the cost for RDBMS, allowing you to store and process more data at a much lower price point.

4 Flexible data models

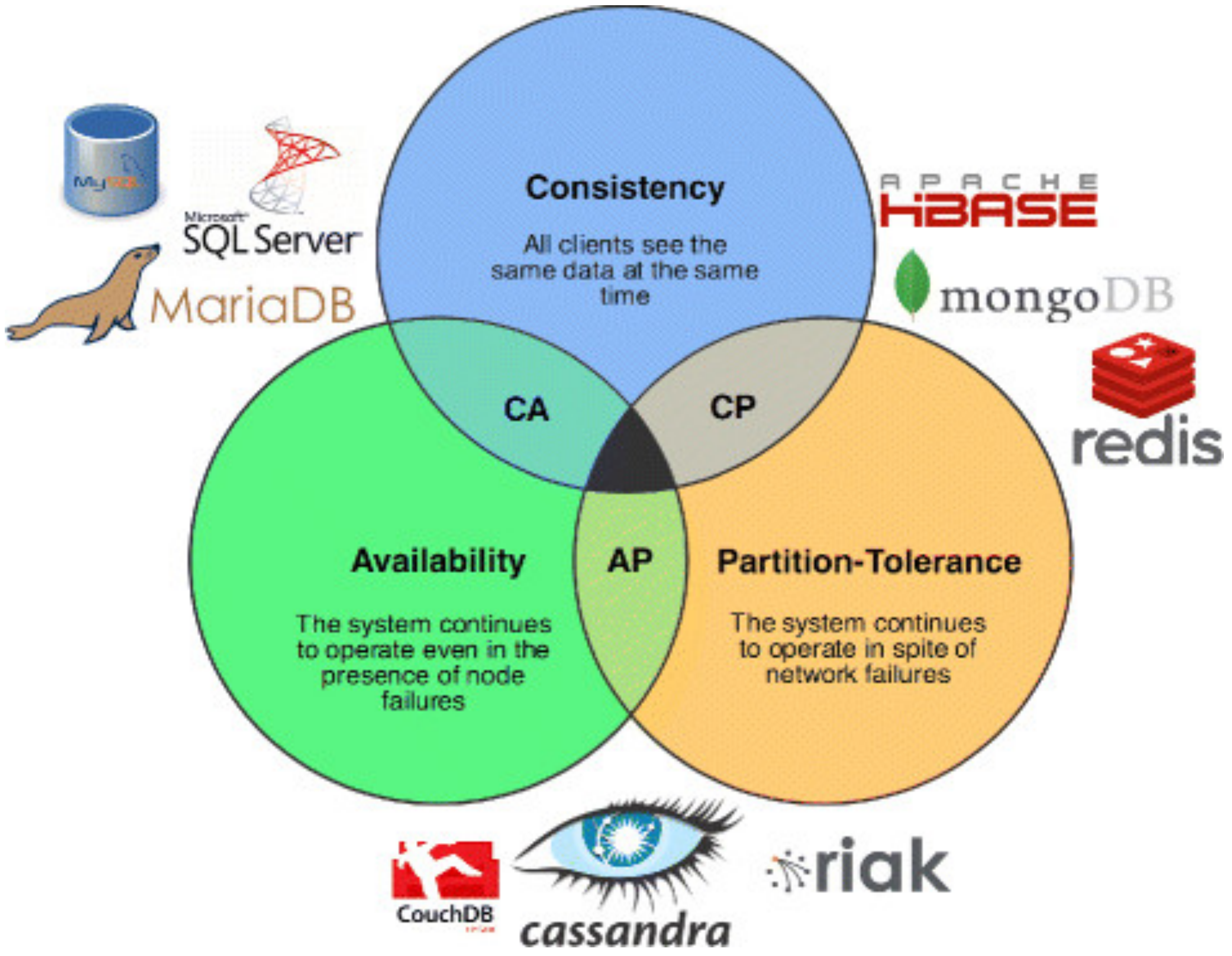
- Minor changes to the data model of an RDBMS have to be carefully managed.
- NoSQL databases have far more relaxed -- or even nonexistent -- data model restrictions.

CAP Theorem

- While designing applications for a distributed architecture, some basic requirements should be present in relations.

- **C-**Consistency
- **A-**Availability
- **P-**Partition tolerance

CAP Theorem (Brewer's Theorem)


Consistency
All clients see the same data at the same time
Availability
The system continues to operate even in the presence of node failures
Partition-Tolerance
The system continues to operate in spite of network failures
CA
CP
AP
MySQL
Microsoft SQL Server
MariaDB
APACHE HBASE
mongoDB
redis
CouchDB
cassandra
riak

CAP Theorem (Brewer's Theorem)

- it is impossible for a ***distributed*** computer system to simultaneously provide all three of the following guarantees:
  - *Consistency*: all nodes see the same data at the same time
  - *Availability*: Node failures do not prevent other survivors from continuing to operate (a guarantee that every request receives a response about whether it succeeded or failed)
  - *Partition tolerance*: the system continues to operate despite arbitrary partitioning due to network failures (e.g., message loss)
- A distributed system can satisfy any two of these guarantees at the same time but not all three.

- **C-**Consistency:-After update operation every client should see the same data- **data consistency.**
- **A-**Availability:-system should always on so service guarantee availability.
- **P-**Partition tolerance:-The system continues to function even the communication among the servers is unreliable.

- **CA**-RDBMS
- **CP-**MongoDB,HBase,Redis
- **AP-**Cassandra,CouchDB,DynamoDB,Riak

BASE

(Basically Available, Soft-State, Eventually Consistent)

- **Basic Availability:** fulfill request, even in partial consistency. Database should appears to work.
- **Soft State:** data storage may not contain the write consistent state.

Different replicas on different shards have to be mutually consistent at all the time.

- **Eventual Consistency:** at some point in the future, data will converge to a consistent state; delayed consistency, as opposed to immediate consistency of the ACID properties.
- A BASE model normally focuses on availability as it is important for scaling.
- But it does not guranteed for data consistency.

END

MongoDB

- **MongoDB** is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of **collection and document.**

- ✔ **Database -**Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

- ✔ **Collection-**Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

✔ Document

- A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects.

- The values of fields may include other documents, arrays, and arrays of documents.

- A document is a set of key-value pairs. Documents have dynamic schema.

  Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

❑ The advantages of using documents are:

- Documents (i.e. objects) correspond to native data types in many programming languages.

- Embedded documents and arrays reduce need for expensive joins.

The relationship of RDBMS terminology with MongoDB.

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
Column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by mongodb itself)

Database Server and Client	
Mysqld/Oracle	mongod
mysql/sqlplus	mongo

Features

✔ **High Performance**

- MongoDB provides high performance data persistence. In particular,Support for embedded data models reduces I/O activity on database system.
- Indexes support faster queries and can include keys from embedded documents and arrays.

✔ **High Availability**

- To provide high availability, MongoDB's replication facility, called replica sets, provide:

  -automatic failover.

  -data redundancy.
- A replica set is a group of MongoDB servers that maintain the same data set, providing redundancy and increasing data availability.

MongoDB - Replication

- Replication is the process of synchronizing data across multiple servers.

- Replication provides redundancy and increases data availability with multiple copies of data on different database servers, replication protects a database from the loss of a single server.

- Replication also allows you to recover from hardware failure and service interruptions.

- With additional copies of the data, you can dedicate one to disaster recovery, reporting, or backup.

❖ MongoDB Create Database

- **The use Command**
- MongoDB **use DATABASE_NAME** is used to create database. The command will create a new database, if it doesn't exist otherwise it will return the existing database.

- **SYNTAX:**

```
use DATABASE_NAME
```

- **EXAMPLE:**
- If you want to create a database with name **<mydb>**, then **use DATABASE** statement would be as follows:

```
>use mydb
```

- switched to db mydb.

- To check your currently selected database use the command db

```
>db
mydb
```

- If you want to check your databases list, then use the command **show dbs**.

```
>show dbs
local     0.78125GB
test     0.23012GB
```

- Your created database (mydb) is not present in list. To display database you need to insert atleast one document into it.

```
>db.tab1.insert({name:"tutorials point"})
>show dbs
local      0.78125GB
mydb        0.23012GB
test       0.23012GB
```

- In mongodb default database is **test.** If you didn't create any database then collections will be stored in test database.

MongoDB Drop Database

- **The dropDatabase() Method**
- MongoDB **db.dropDatabase()** command is used to drop an existing database.
- **SYNTAX:**

```
db.dropDatabase()
```

- This will delete the selected database. If you have not selected any database, then it will delete default 'test' database

- **EXAMPLE:**
- First, check the list available databases by using the command **show dbs**

```
>show dbs
local      0.78125GB
mydb       0.23012GB
test       0.23012GB
```

- If you want to delete new database **<mydb>**, then **dropDatabase()** command would be as follows:

```
>use mydb
switched to db mydb
>db.dropDatabase()
>{ "dropped" : "mydb", "ok" : 1 }
>
```

- **Now check list of databases**

```
>show dbs
local      0.78125GB
test      0.23012GB
```

MongoDB Create Collection

- **The createCollection() Method**
- MongoDB **db.createCollection(name, options)** is used to create collection.
- **SYNTAX:**

db.createCollection(name, options)

- In the command, **name** is name of collection to be created. **Options** is a document and used to specify configuration of collection.

Parameter	**Type**	**Description**
Name	String	Name of the collection to be created
Options	Document	(Optional) Specify options about memory size and indexing

Options parameter is optional, so you need to specify only name of the collection. Following is the list of options you can use:

Field	Type	Description
Capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a collection fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. **If you specify true, you need to specify size parameter also.**
autoIndexID	Boolean	(Optional) If true, automatically create index on _id field.s Default value is false.
Size	number	(Optional) Specifies a maximum size in bytes for a capped collection. **If capped is true, then you need to specify this field also.**
Max	number	(Optional) Specifies the maximum number of documents allowed in the capped collection.

While inserting the document, MongoDB first checks size field of capped collection, then it checks max field.

- **EXAMPLES:**
- Basic syntax of **createCollection()** method without options is as follows

```
>use test
    switched to db test
>db.createCollection("mycollection")
                { "ok" : 1 }
```

- You can check the created collection by using the command **show collections**

```
>show collections
    mycollection
    system.indexes
```

- Following example shows the syntax **of createCollection()** method with few important options:

```
>db.createCollection("mycol",
                { capped : true,
                  autoIndexID : true,
                  size : 6142800,
                  max : 10000 } )
{ "ok" : 1 }
>
```

- In mongodb you don't need to create collection. MongoDB creates collection automatically, when you insert some document.

```
>db.demo.insert({"name" :"tutorials"})
>show collections
mycol
mycollection
system.indexes
demo
>
```

- **MongoDB Drop Collection**
- **The drop() Method**

MongoDB's **db.collection.drop()** is used to drop a collection from the database.

- **SYNTAX:**

db.COLLECTION_NAME.drop()

- **EXAMPLE:**
- First, check the available collections into your database **mydb**

```
>use mydb
```

- switched to db mydb

```
>show collections
 mycol
mycollection
system.indexes
demo
>
```

- Now drop the collection with the name **mycollection**

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

- **Again check the list of collections into database**

```
>show collections
mycol
system.indexes
demo
>
```

- drop() method will return true, if the selected collection is dropped successfully otherwise it will return false.

❖ MongoDB - Insert Document

- **The insert() Method**
- To insert data into MongoDB collection, you need to use MongoDB's **insert()** or **save()**method.

- In MongoDB, the db.collection.insert() method adds new documents into a collection.

- In addition, both the db.collection.update() method and the db.collection.save() method can also add new documents through an operation called an **upsert.**

- An **upsert** is an operation that performs either an update of an existing document or an insert of a new document if the document to modify does not exist.

- **SYNTAX**

```
>db.COLLECTION_NAME.insert(document)
```

- **EXAMPLE**

```
1)db.inventory.insert( { _id: 10, type: "misc", item: "card", qty: 15 } )

2)db.mycol.insert(
        {       _id: ObjectId(7df78ad8902c),
    title: 'MongoDB Overview',
    description: 'MongoDB is no sql database',
            by: 'tutorials point',
      url: 'http://www.tutorialspoint.com',
      tags: ['mongodb', 'database', 'NoSQL'],
            likes: 100
        }
    )
```

- Here **mycol** is our collection name. If the collection doesn't exist in the database, then MongoDB will create this collection and then insert document into it.
- In the inserted document if we don't specify the _id parameter, then MongoDB assigns an unique ObjectId for this document.
- _id is 12 bytes hexadecimal number unique for every document in a collection. 12 bytes are divided as follows:
- _id: ObjectId(4 bytes timestamp, 3 bytes machine id,
  2 bytes process id, 3 bytes incrementer)
- To insert multiple documents in single query, you can pass an array of documents in insert() command.

- **Insert Multiple Documents**
- The following example performs a bulk insert of three documents by passing an array of documents to the insert() method.

- **The documents in the array do not need to have the same fields.**

- For instance, the first document in the array has an _id field and a type field. Because the second and third documents do not contain an _id field,mongod will create the _id field for the second and third documents during the insert:

```
db.products.insert(
        [
         { _id: 11, item: "pencil",qty: 50, type: "no.2" },
         { item: "pen", qty: 20 },
                  { item: "eraser", qty: 25 }
        ]      )
```

- The operation inserted the following three documents:

```
{ "_id" : 11, "item" : "pencil", "qty" : 50, "type" : "no.2" }

{ "_id" : ObjectId("51e0373c6f35bd826f47e9a0"), "item" : "pen",
"qty" : 20 }

{ "_id" : ObjectId("51e0373c6f35bd826f47e9a1"), "item" :
"eraser", "qty" : 25 }
```

- **Insert a Document with update() Method**
- The following example creates a new document if no document in the inventory collection contains

  {type: "book", item : "journal" }:

- db.inventory.update(

```
                              { type: "book", item : "journal" },
                               { $set : { qty: 10 } },
                               { upsert : true } )
```

- MongoDB adds the _id field and assigns as its value a unique ObjectId. The new document includes the item and type fields from the <query> criteria and the qty field from the <update> parameter.

- { "_id" : ObjectId("51e8636953dbe31d5f34a38a"),

```
   "item" :"journal", "qty" : 10, "type" : "book"
}
```

❖ MongoDB Update Document

- MongoDB's **update()** and **save()** methods are used to update document into a collection. The update() method update values in the existing document while the save() method replaces the existing document with the document passed in save() method.

- **MongoDB Update() method**
- The update() method updates values in the existing document.

- **SYNTAX:**

```
>db.COLLECTION_NAME.update(SELECTIOIN_CRITERIA,
                           UPDATED_DATA)
```

- **EXAMPLE**
- Consider the mycol collectioin has following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB
                    Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL
                    Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials
                    Point Overview"}
```

- Following example will set the new title **'New MongoDB Tutorial'** of the documents whose title is 'MongoDB Overview'

```
>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New
               MongoDB Tutorial'}})
>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"New MongoDB
    Tutorial"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point
    Overview"}
>
```

- **By default mongodb will update only single document, to update multiple you need to set a paramter 'multi' to true.**

```
  >db.mycol.update({'title':'MongoDB Overview'},
                        {$set:{'title':'New MongoDB Tutorial'}},
                         {multi:true}
                       )
```

- **MongoDB Save() Method**
- The **save()** method replaces the existing document with the new document passed in save() method.

- **SYNTAX**

```
>db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})
```

- **EXAMPLE**
- Following example will replace the document with the _id '5983548781331adf45ec5'

```
>db.mycol.save(        { "_id" : ObjectId(5983548781331adf45ec5),
             "title":"Tutorials Point New Topic",
                  "by":"Tutorials Point"
                }
            )
>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec7),
       "title":"Tutorials Point New Topic",
       "by":"Tutorials Point“
}
{ "_id" : ObjectId(5983548781331adf45ec6),
    "title":"NoSQL Overview“
}
{ "_id" : ObjectId(5983548781331adf45ec7),
    "title":"Tutorials Point Overview“
}
>
```

- **Insert a Document with save() Method**
- The following example creates a new document in the inventory collection:

```
db.inventory.save( { type: "book", item: "notebook", qty: 40 } )
```

- MongoDB adds the _id field and assigns as its value a unique ObjectId.

-
  ```
  { "_id" : ObjectId("51e866e48737f72b32ae4fbc"),
    "type" : "book",
    "item" : "notebook",
     "qty" : 40
   }
  ```

- **Replace an Existing Document**
- The products collection contains the following document:

```
{ "_id" : 100, "item" : "water", "qty" : 30 }
```

- The **save()** method performs an update with upsert since the document contains an _id field:

```
db.products.save( { _id : 100, item : "juice" } )
```

- Because the _id field holds a value that exists in the collection, the operation performs an update to replace the document and results in the following document:

```
{ "_id" : 100, "item" : "juice" }
```

❖ MongoDB - Query Document

- **The find() Method**
- To query data from MongoDB collection, you need to use MongoDB's **find()** method.
- **SYNTAX**

```
>db.COLLECTION_NAME.find()
```

- **find()** method will display all the documents in a non structured way.

- **The pretty() Method**
- To display the results in a formatted way, you can use **pretty()** method.
- **SYNTAX:**

```
>db.mycol.find().pretty()
```

- **Example**

```
>db.mycol.find().pretty()

{   "_id": ObjectId(7df78ad8902c),
    "title": "MongoDB Overview",
    "description": "MongoDB is no sql database",
    "by": "tutorials point",
     "url": "http://www.tutorialspoint.com",
     "tags": ["mongodb", "database", "NoSQL"],
     "likes": "100"
}
>
```

- **Specify Equality Condition**

- To specify equality condition, use the query document { <field>: <value> } to select all documents that contain the <field> with the specified <value>.

- The following example retrieves from the inventory collection all documents where the type field has the value snacks:

```
db.inventory.find( { type: "snacks" } )
```

- **Specify Conditions Using Query Operators**
- A query document can use the query operators to specify conditions in a MongoDB query.

- The following example selects all documents in the inventory collection where the value of the type field is either 'food' or 'snacks':

```
db.inventory.find( { type: { $in: [ 'food', 'snacks' ] } } )
```

- Although you can express this query using the $or operator, use the $in operator rather than the $or operator when performing equality checks on the **same field**.

- The $or operator performs a logical OR operation on an array of two or more <expressions> and selects the documents that satisfy at least one of the <expressions>.

- **The $or has the following syntax:**
- { $or: [{ <expression1> }, { <expression2> }, ... , { <expressionN> }] }

- **Example:**

db.inventory.find({ $or: [{ quantity: { $lt: 20 } } , { price: 10 }] })

- This query will select all documents in the inventory collection where either the quantity field value is less than 20 **or** the price field value equals 10.

- **RDBMS Where Clause Equivalents in MongoDB**
- To query the document on the basis of some condition, you can use following operations

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:<value>}	db.mycol.find({"by":"tutorials ”}).pretty()	where by = 'tutorials point'
Less Than	{<key>:{$lt:<value>}}	db.mycol.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>:{$lte:<value>}}	db.mycol.find({"likes":{$lte:50}}).pretty()	where likes <= 50

Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{<key>:{$gte:<value>}}	db.mycol.find({"likes":{$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>:{$ne:<value>}}	db.mycol.find({"likes":{$ne:50}}).pretty()	where likes != 50
Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50

- **Specify AND Conditions**
- A compound query can specify conditions for more than one field in the collection’s documents.
- Implicitly, a logical AND conjunction connects the clauses of a compound query so that the query selects the documents in the collection that match all the conditions.

- In the following example, the query document specifies an equality match on the field food **and** a less than ($lt) comparison match on the field price:

```
db.inventory.find( { type: 'food', price: { $lt: 9.95 } } )
```

- This query selects all documents where the type field has the value 'food' **and** the value of the price field is less than 9.95.

- **Specify OR Conditions**
- Using the $or operator, you can specify a compound query that joins each clause with a logical OR conjunction so that the query selects the documents in the collection that match at least one condition.

- In the following example, the query document selects all documents in the collection where the field qty has a value greater than ($gt) 100 **or** the value of the price field is less than ($lt) 9.95:

```
db.inventory.find( { $or: [ { qty: { $gt: 100 } },
                            { price: { $lt: 9.95 } }
                          ] } )
```

- **Specify AND as well as OR Conditions**
- With additional clauses, you can specify precise conditions for matching documents.
- In the following example, the compound query document selects all documents in the collection where the value of the type field is 'food' **and** *either* the qty has a value greater than ($gt) 100 *or* the value of the price field is less than ($lt) 9.95:

```
db.inventory.find( { type: 'food',
                 $or: [ { qty: { $gt: 100 } },
                        { price: { $lt: 9.95 } }
                             ]
                       }
                     )
```

❖ MongoDB Delete Document

- **The remove() Method**
- MongoDB's **remove()** method is used to remove document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag
- **deletion criteria :** (Optional) deletion criteria according to documents will be removed.
- **justOne :** (Optional) if set to true or 1, then remove only one document.

- **SYNTAX:**
- >db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)

- **EXAMPLE**
- Consider the mycol collection has following data.

```
{ "_id" : ObjectId(59835487813 31adf45ec5),
   "title":"MongoDB Overview"}

{ "_id" : ObjectId(59835487813 31adf45ec6),
    "title":"NoSQL Overview"}

{ "_id" : ObjectId(59835487813 31adf45ec7),
   "title":"Tutorials Point Overview"}
```

- Following example will remove all the documents whose title is 'MongoDB Overview‘

```
>db.mycol.remove({'title':'MongoDB Overview'})
>db.mycol.find()

{ "_id" : ObjectId(598354878l33ladf45ec6), "title":"NoSQL
                              Overview"}

{ "_id" : ObjectId(598354878l33ladf45ec7), "title":"Tutorials Point
                                   Overview"}
```

❖ Remove All documents

- If you don't specify deletion criteria, then mongodb will delete whole documents from the collection.
- This is equivalent of **SQL's truncate command.**
- To remove all documents from a collection, pass an empty query document {} to the remove() method.
- The remove() method does not remove the indexes.
- The following example removes all documents from the inventory collection:

db.inventory.remove({})

- To remove all documents from a collection, it may be more efficient to use the drop() method to drop the entire collection, including the indexes, and then recreate the collection and rebuild the indexes.

- **Remove Documents that Match a Condition**
- To remove the documents that match a deletion criteria, call the remove() method with the <query>parameter.

- The following example removes all documents from the inventory collection where the type field equals food:

```
db.inventory.remove( { type : "food" } )
```

- For large deletion operations, it may be more efficient to copy the documents that you want to keep to a new collection and then use drop() on the original collection.

- **Remove a Single Document that Matches a Condition**
- To remove a single document, call the remove() method with the justOne parameter set to true or 1.

- The following example removes one document from the inventory collection where the type field equals food:

```
db.inventory.remove( { type : "food" }, 1 )
```

MongoDB Projection

- In **mongodb,** projection meaning is selecting only necessary data rather than selecting whole of the data of a document. If a document has 5 fields and you need to show only 3, then select only 3 fields from them.

- **The find() Method**
- In MongoDB when you execute **find()** method, then it displays all fields of a document. To limit this you need to set list of fields with value 1 or 0. 1 is used to show the field while 0 is used to hide the field.

- **SYNTAX:**
- >db.COLLECTION_NAME.find({},{KEY:1})

- **EXAMPLE**
- Consider the collection myycol has the following data

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

- Following example will display the title of the document while quering the document.

```
>db.mycol.find({},{"title":1,_id:0})
{"title":"MongoDB Overview"}
{"title":"NoSQL Overview"}
{"title":"Tutorials Point Overview"}
>
```

_id field is always displayed while executing **find()** method, if you don't want this field, then you need to set it as 0

- **MongoDB Limit Records**
- **The Limit() Method**
- To limit the records in MongoDB, you need to use **limit()** method. **limit()** method accepts one number type argument, which is number of documents that you want to displayed.

- **SYNTAX:**

```
>db.COLLECTION_NAME.find().limit(NUMBER)
```

- **EXAMPLE**
- Consider the collection mycol has the following data

```
{ "_id" :1, "title":"MongoDB Overview"}
{ "_id" : 2, "title":"NoSQL Overview"}
{ "_id" : 3, "title":"Tutorials Point  Overview"}
```

- Following example will display only 2 documents while querying the document.

```
db.mycol.find({},{"title":1,_id:0}).limit(2)
```

```
{"title":"MongoDB Overview"}
{"title":"NoSQL Overview"}
```

- If you don't specify number argument in **limit()** method then it will display all documents from the collection.

- **MongoDB Skip() Method**
- Apart from limit() method there is one more method **skip()** which also accepts number type argument and used to skip number of documents.

- **SYNTAX:**

```
>db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)
```

- **EXAMPLE:**
- Following example will only display only second document.

```
>db.mycol.find({},{"title":1,_id:0}).limit(1).skip(1)

{"title":"NoSQL Overview"}
>
```

- Default value in **skip()** method is 0

MongoDB Sort Documents

- **The sort() Method**
- To sort documents in MongoDB, use **sort()** method. **sort()** method accepts a document containing list of fields along with their sorting order. To specify sorting order 1 and -1 are used. 1 is used for ascending order while -1 is used for descending order.

- **SYNTAX:**

```
>db.COLLECTION_NAME.find().sort({KEY:1})
```

- **EXAMPLE**
- Consider the collection mycol has the following data

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}

{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}

{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

- Following example will display the documents sorted by title in descending order.

```
>db.mycol.find({},{"title":1,_id:0}).sort({"title":-1})
```

```
{"title":"Tutorials Point Overview"}
 {"title":"NoSQL Overview"}
{"title":"MongoDB Overview"}
```

If you don't specify the sorting preference, then **sort()** method will display documents in ascending order.

MongoDB Aggregation

- Aggregations operations process data records and return computed results.

- Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.

- MongoDB provides a rich set of aggregation operations that examine and perform calculations on the data sets.

- Like queries, aggregation operations in MongoDB use collections of documents as an input and return results in the form of one or more documents.

- In **sql** count(*) and with group by is an equivalent of mongodb aggregation.

- **The aggregate() Method**

For the aggregation in mongodb
use **aggregate()** method.

- **SYNTAX:**

```
db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)
```

- **EXAMPLE:**
- In the collection you have the following data:

```
{ _id: ObjectId(7df78ad8902c)
   title: 'MongoDB Overview',
  description: 'MongoDB is no sql database',
   by_user: 'tutorials point',
    url: 'http://www.tutorialspoint.com',
   tags: ['mongodb', 'database', 'NoSQL'],
   likes: 100
},
{ _id: ObjectId(7df78ad8902d)
title: 'NoSQL Overview',
 description: 'No sql database is very fast',
 by_user: 'tutorials point',
 url: 'http://www.tutorialspoint.com',
tags: ['mongodb', 'database', 'NoSQL'], likes: 10 },

{ _id: ObjectId(7df78ad8902e)
 title: 'Neo4j Overview',
 description: 'Neo4j is no sql database',
 by_user: 'Neo4j',
 url: 'http://www.neo4j.com',
 tags: ['neo4j', 'database', 'NoSQL'],
 likes: 750
},
```

- Now from the above collection if you want to display a list that how many tutorials are written by each user then use following **aggregate()** method :

```
db.mycol.aggregate(
     [
      {
            $group :
                      {_id : "$by_user", num_tutorial : {$sum : 1}}
         }
     ]              )
```

Output:

```
{
   "result" : [
           {
                "_id" : "tutorials point",
              "num_tutorial" : 2
           },
           {
              "_id" : “Neo4j",
                "num_tutorial" : 1
           }
            ],
 "ok" : 1 }
```

- Sql equivalent query for the above use case will be

```
select by_user, count(*)
from mycol
group by by_user
```

- In the above example we have grouped documents by field **by_user** and on each occurence of **by_user** previous value of sum is incremented.

- A list available for aggregation expressions.

Expression	Description	Example
$sum	Sums up the defined value from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : "$likes"}}}])
$avg	Calculates the average of all given values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$avg : "$likes"}}}])
$min	Gets the minimum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$min : "$likes"}}}])
$max	Gets the maximum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$max : "$likes"}}}])

$push	Inserts the value to an array in the resulting document.	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$push: "$url"}}}])
$addToSet	Inserts the value to an array in the resulting document but does not create duplicates.	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$addToSet : "$url"}}}])
$first	Gets the first document from the source documents according to the grouping.	db.mycol.aggregate([{$group : {_id : "$by_user", first_url : {$first : "$url"}}}])
$last	Gets the last document from the source documents according to the grouping.	db.mycol.aggregate([{$group : {_id : "$by_user", last_url : {$last : "$url"}}}])

Example

- **A collection books contains the following documents:**
- { "_id" : 8751, "title" : "The Banquet", "author" : "Dante", "copies" : 2 }
- { "_id" : 8752, "title" : "Divine Comedy", "author" : "Dante", "copies" : 1 }
- { "_id" : 8645, "title" : "Eclogues", "author" : "Dante", "copies" : 2 }
- { "_id" : 7000, "title" : "The Odyssey", "author" : "Homer", "copies" : 10 }
- { "_id" : 7020, "title" : "Iliad", "author" : "Homer", "copies" : 10 }

- **Group title by author**
- The following aggregation operation pivots the data in the books collection to have titles grouped by authors.

```
db.books.aggregate (
    [
        { $group :
            { _id : "$author", books: { $push: "$title" } }
        }
    ]
                    )
```

- The operation returns the following documents:

```
{ "_id" : "Homer", "books" : [ "The Odyssey", "Iliad" ] }
{ "_id" : "Dante", "books" : [ "The Banquet", "Divine Comedy",
   "Eclogues" ] }
```

- **Group Documents by author**
- The following aggregation operation uses the $$ROOT system variable to group the documents by authors. The resulting documents must not exceed the BSON Document Size limit.

```
db.books.aggregate(
        [
            {
            $group :
            { _id : "$author", books: { $push:
        "$$ROOT" } }
            }
        ]
            )
```

- **The operation returns the following documents:**

- { "_id" : "Homer", "books" :
- [
- { "_id" : 7000, "title" : "The Odyssey", "author" : "Homer", "copies" : 10 },
- { "_id" : 7020, "title" : "Iliad", "author" : "Homer", "copies" : 10 }
-]
- }
- { "_id" : "Dante", "books" :
- [
- { "_id" : 8751, "title" : "The Banquet", "author" : "Dante", "copies" : 2 },
- { "_id" : 8752, "title" : "Divine Comedy", "author" : "Dante", "copies" : 1 },
- { "_id" : 8645, "title" : "Eclogues", "author" : "Dante", "copies" : 2 }
-]
- }

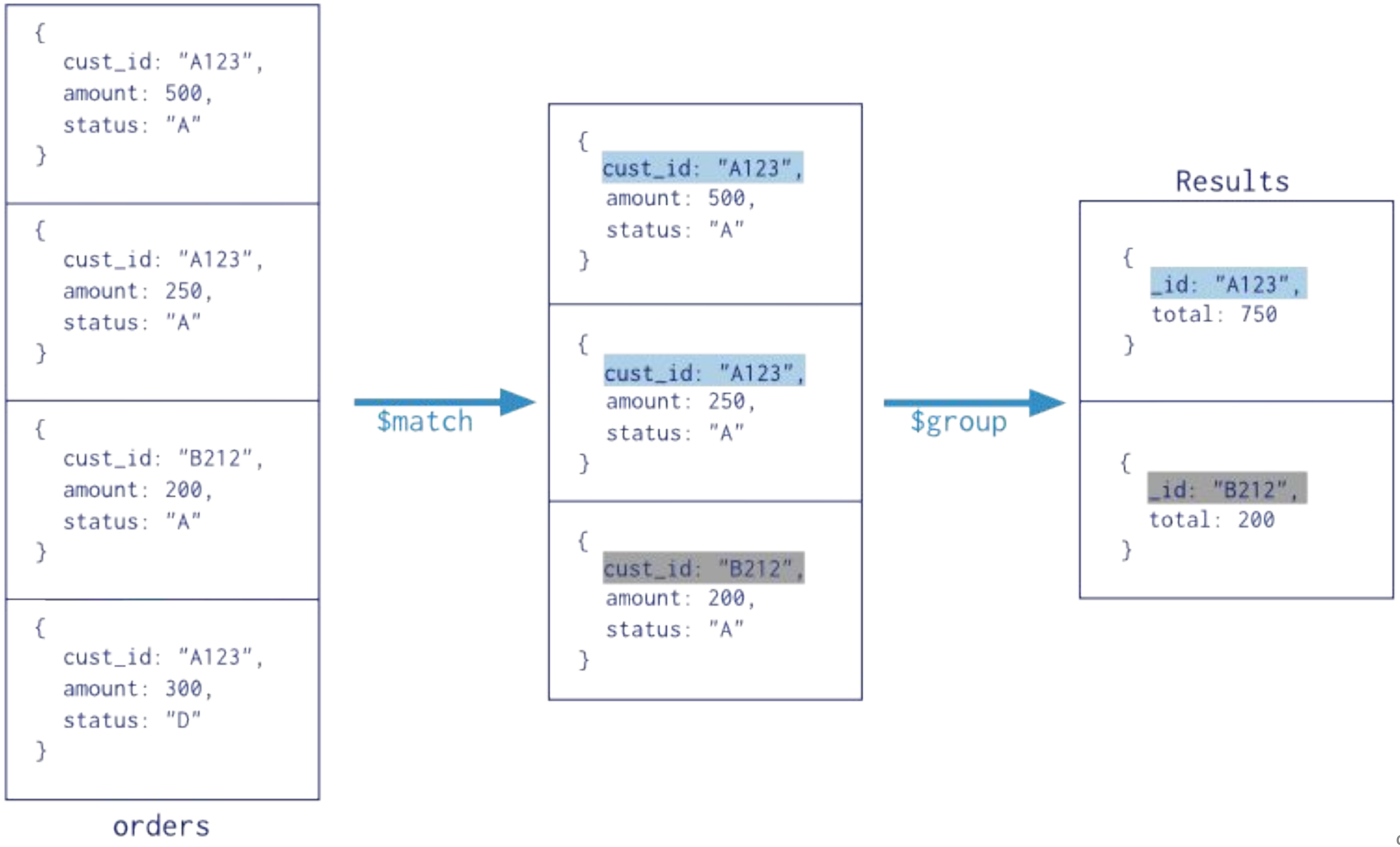
- **Aggregation Pipeline**

- The aggregation pipeline is a framework for data aggregation modeled on the concept of data processing pipelines.

- Documents enter a multi-stage pipeline that transforms the documents into an aggregated results.

- The aggregation pipeline provides an alternative to *map-reduce* and may be the preferred solution for many aggregation tasks.

```
  Collection
     ↓
db.orders.aggregate( [
    $match phase ——→  { $match: { status: "A" } },
    $group phase ——→  { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
                   ] )
```



```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
{
  cust_id: "A123",
  amount: 300,
  status: "D"
}
```

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
```

```
{
  _id: "A123",
  total: 750
}
{
  _id: "B212",
  total: 200
}
```

❖ Map-Reduce

- Map-reduce is a data processing paradigm for condensing large volumes of data into useful *aggregated* results.

- For map-reduce operations, **MongoDB** provides the mapReduce database command.

```
db.collection.mapReduce(
                         <map>,
                          <reduce>,
                          { out: <collection>,
                            query: <document>,
                            sort: <document>,
                            limit: <number>,
                            finalize: <function>,
                            scope: <document>,
                                 jsMode: <boolean>,
                         verbose: <boolean> }
                         )
```

- **MapReduce Command:**
- Syntax:

```
 db.collection.mapReduce(
    function()
                  {emit(key,value);},         //map function
function(key,values){ return reduceFunction},
            //reduce function
     {   out: collection,
         query: document,
         sort: document,
         limit: number
     }
                )
```

- The map-reduce function first queries the collection, then maps the result documents to emit key-value pairs which is then reduced based on the keys that have multiple values.

- **In the above syntax:**
- **map** is a javascript function that maps a value with a key and emits a key-value pair
- **reduce** is a javscript function that reduces or groups all the documents having the same key
- **out** specifies the location of the map-reduce query result
- **query** specifies the optional selection criteria for selecting documents
- **sort** specifies the optional sort criteria

- **limit** specifies the optional maximum number of documents to be returned

- **finalize f**ollows the reduce method and modifies the output.

- **Scope** specifies global variables that are accessible in the map ,reduce and finalize functions.

- **jsMode** Specifies whether to convert intermediate data into BSON format between the execution of the map and reduce functions. Defaults to false.

- **verbose s**pecifies whether to include the timing information in the result information. The verbose defaults to true to include the timing information.

- **Requirements for the map Function**

  function() { ... emit(key, value); }

- In the map function, reference the current document as **this** within the function.
- The **emit(key,value)** function associates the key with a value.
  - The map function can call emit(key,value) any number of times, including 0, per each input document.

  - Consider the map function may call emit(key,value) either 0 or 1 times depending on the value of the input document's status field:

```
function()
        {
          if (this.status == 'A')
               emit(this.cust_id, 1);
}
```

- **Requirements for the reduce Function**
- The reduce function has the following prototype:

```
function(key, values)
    { ... return result; }
```

- MongoDB will **not** call the reduce function for a key that has only a single value.

- MongoDB can invoke the reduce function more than once for the same key.

- The reduce function can access the variables defined in the scope parameter.

- **Requirements for the finalize Function**
- The finalize function has the following prototype:

```
function(key, reducedValue)
                    { ... return modifiedObject; }
```

- The finalize function receives as its arguments a key value and the **reducedValue** from the reduce function.

- The finalize function can access the variables defined in the scope parameter.

Using MapReduce:

- Consider the following document structure storing user posts.

- The document stores user_name of the user and the status of post.

```
   {
      "post_text": "tutorialspoint is an awesome website”,
"user_name": "mark",
       "status":"active"
    }
```

- Now, we will use a mapReduce function on our **posts** collection to select all the active posts, group them on the basis of user_name and then count the number of posts by each user using the following code:

```
db.posts.mapReduce(
      function()
                  { emit(this.user_name,1); },
      function(key, values) {return Array.sum(values)},
             {
                  query:{status:"active"},
                  out:"post_total"
        }
                 )
```

- **Output:**

```
{
  "result" : "post_total",
   "timeMillis" : 9,
   "counts" :
         {
          "input" : 4,
         "emit" : 4,
         "reduce" : 2,
         "output" : 2
         },
"ok" : 1, }
```

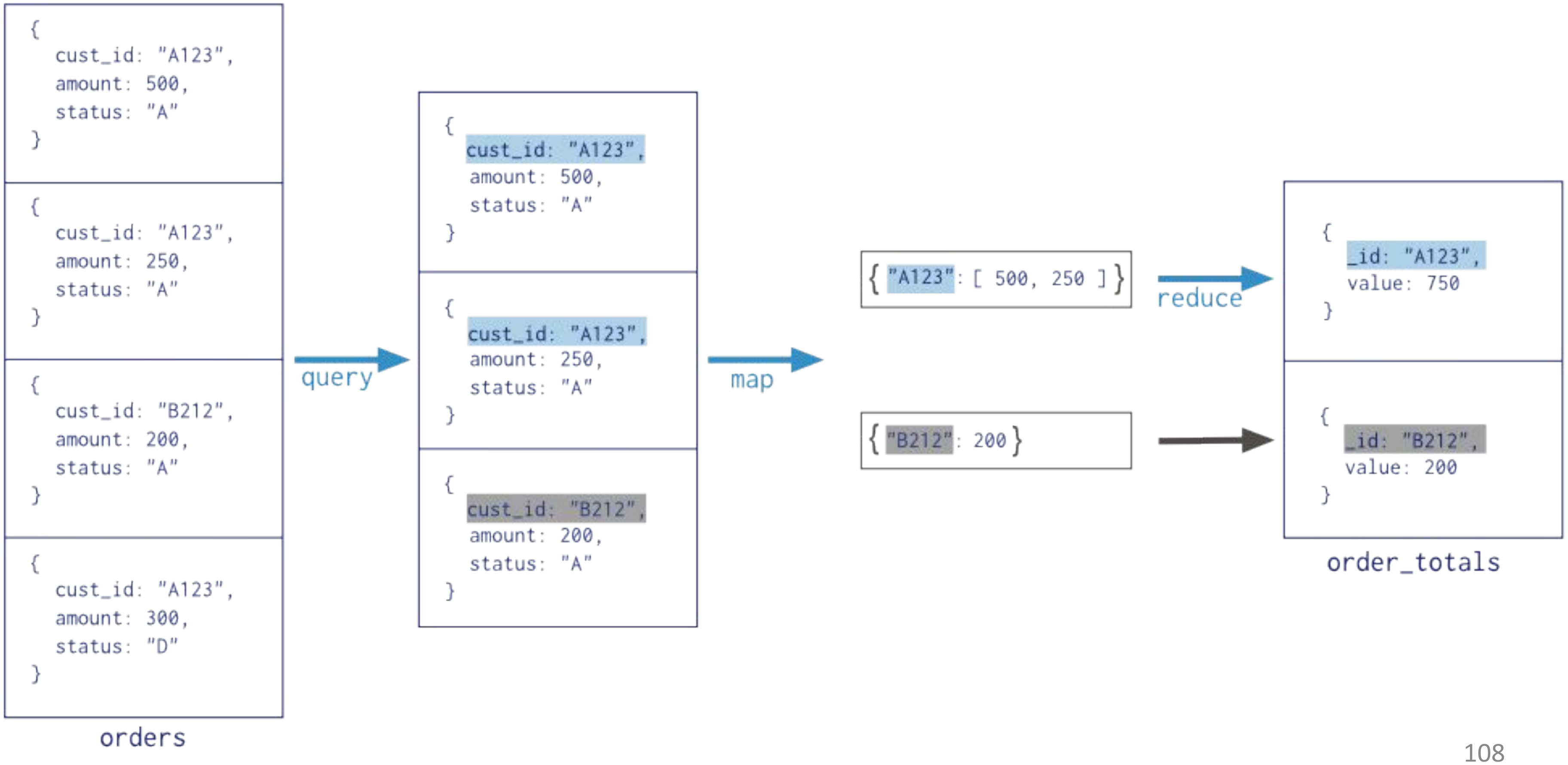
- The result shows that a total of 4 documents matched the query (status:"active"), the map function emitted 4 documents with key-value pairs and finally the reduce function grouped mapped documents having the same keys into 2.

- To see the result of this mapReduce query use the **find** operator:

```
db.posts.mapReduce( function() { emit(this.user_id,1); },
function(key, values) {return Array.sum(values)},
      {
            query:{status:"active"},
                  out:"post_total"
            }
).find()
```

- Consider the following map-reduce operation:

```
  Collection
     |
     v
db.orders.mapReduce(
          map    ──►  function() { emit( this.cust_id, this.amount ); },
          reduce ──►  function(key, values) { return Array.sum( values ) },
                      {
          query  ──►    query: { status: "A" },
          output ──►    out: "order_totals"
                      }
                    )
```



```
orders:
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
{
  cust_id: "A123",
  amount: 300,
  status: "D"
}
```

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
```

```
{ "A123": [ 500, 250 ] }
{ "B212": 200 }
```

```
order_totals:
{
  _id: "A123",
  value: 750
}
{
  _id: "B212",
  value: 200
}
```

- In this map-reduce operation, MongoDB applies the ***map* phase** to each input document (i.e. the documents in the collection that match the query condition).

- The map function emits key-value pairs. For those keys that have multiple values, MongoDB applies the ***reduce* phase,** which collects and condenses the aggregated data. MongoDB then stores the results in a collection.

- All map-reduce functions in MongoDB are JavaScript and run within the mongod process.

- Map-reduce operations take the documents of a single *collection* as the *input* and can perform any arbitrary sorting and limiting before beginning the map stage.

- mapReduce can return the results of a map-reduce operation as a document, or may write the results to collections.

- The input and the output collections may be sharded.

- **Map-Reduce Examples**

- Consider the following map-reduce operations on a collection **orders** that contains documents of the following prototype:

```
{ _id: ObjectId("50a8240b927d5d8b5891743c"),
  cust_id: "abc123",
  ord_date: new Date("Oct 04, 2012"),
  status: 'A', price: 25,
  items: [ { sku: "mmm", qty: 5, price: 2.5 },
           { sku: "nnn", qty: 5, price: 2.5 }
         ]
}
```

❖ Return the Total Price Per Customer

- Perform the map-reduce operation on the **orders** collection to group by the cust_id, and calculate the sum of the price for each cust_id:

- **1)Define the map function to process each input document:**
  - In the function, this refers to the document that the map-reduce operation is processing.
  - The function maps the price to the cust_id for each document and emits the cust_id and price pair.

```
var mapFunction1 = function()
{
            emit(this.cust_id, this.price);
};
```

- 2)Define the corresponding reduce function with two arguments keyCustId and valuesPrices:
- The **valuesPrices** is an array whose elements are the price values emitted by the map function and grouped by **keyCustId.**
- The function reduces the **valuesPrice** array to the sum of its elements.

```
var reduceFunction1 = function(keyCustId, valuesPrices)
{
        return Array.sum(valuesPrices);
};
```

3)Perform the map-reduce on all documents in the orders collection using the mapFunction1 map function and the reduceFunction1 reduce function.

```
db.orders.mapReduce( mapFunction1,
                     reduceFunction1,
                      {
                                   out: "map_reduce_example"
                               } )
```

- This operation outputs the results to a collection named **map_reduce_example.**
- If the **map_reduce_example** collection already exists, the operation will replace the contents with the results of this map-reduce operation.

- **Single Purpose Aggregation Operations**
- Aggregation refers to a broad class of data manipulation operations that compute a result based on an input *and* a specific procedure.

- MongoDB provides a number of aggregation operations that perform specific aggregation operations on a set of data.

✔ Count

- MongoDB can return a count of the number of documents that match a query. The count command as well as the count() and cursor.count() methods provide access to counts in the mongo shell.
- **Example**
- Given a collection named **records** with *only* the following documents:

```
{ a: 1, b: 0 }
 { a: 1, b: 1 }
 { a: 1, b: 4 }
 { a: 2, b: 2 }
```

- The following operation would count all documents in the collection and return the number 4:

```
db.records.count()
```

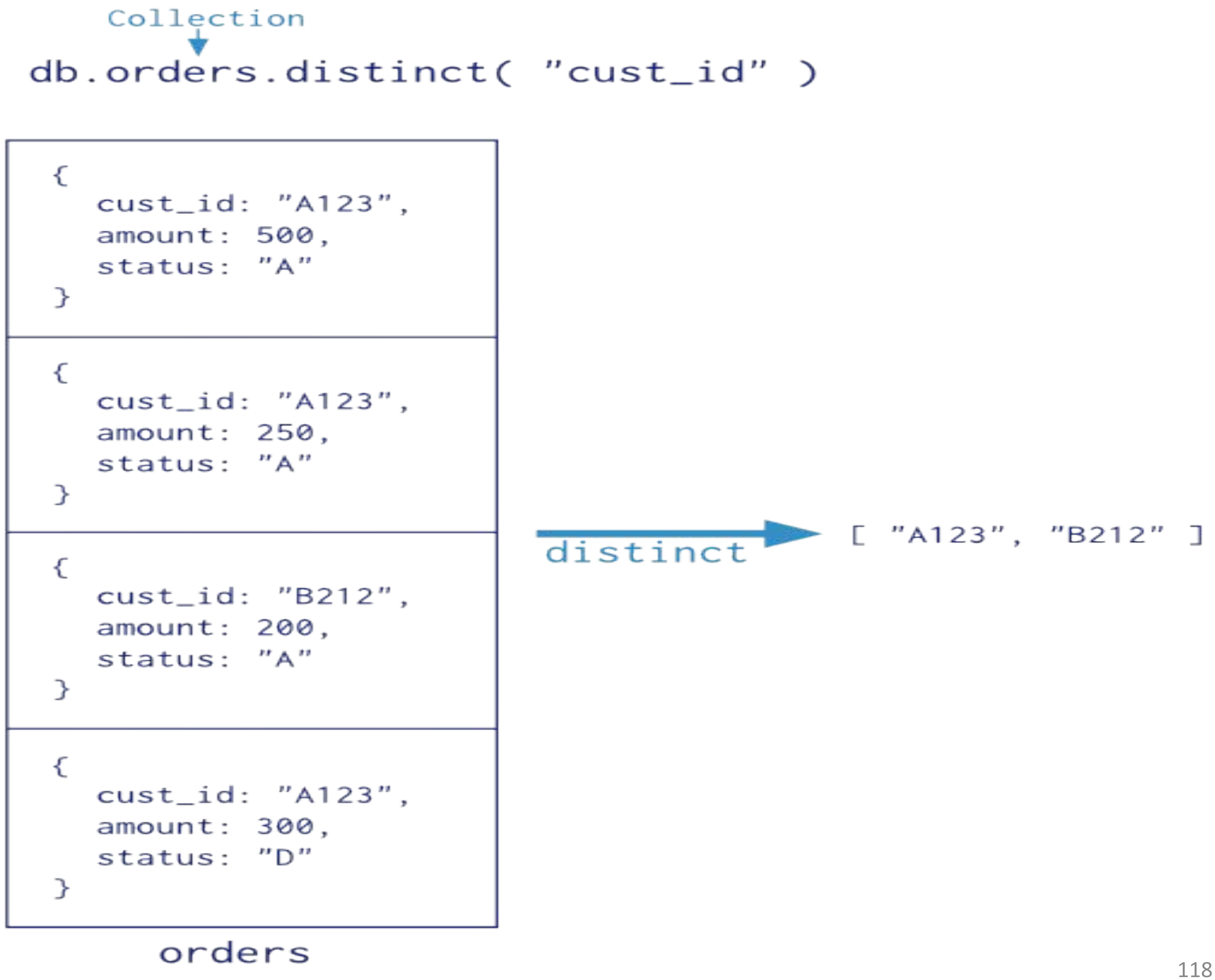
- The following operation will count only the documents where the value of the field a is 1 and return 3:

```
db.records.count( { a: 1 } )
```

- **Distinct**
- The *distinct* operation takes a number of documents that match a query and returns all of the unique values for a field in the matching documents.

- The distinct command and db.collection.distinct() method provide this operation in the mongo shell.

- Example of a distinct operation:



```
db.orders.distinct( "cust_id" )
```

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
```

```
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
```

```
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
```

```
{
  cust_id: "A123",
  amount: 300,
  status: "D"
}
```

```
[ "A123", "B212" ]
```

- **Example**
- Given a collection named records with *only* the following documents:

```
{ a: 1, b: 0 }
 { a: 1, b: 1 }
{ a: 1, b: 1 }
{ a: 1, b: 4 }
{ a: 2, b: 2 }
 { a: 2, b: 2 }
```

db.records.distinct("b")

- **Output:**[0, 1, 4, 2]

Group

- The *group* operation takes a number of documents that match a query, and then collects groups of documents based on the value of a field or fields.

- It returns an array of documents with computed results for each group of documents.

- Access the grouping functionality via the group command or the db.collection.group() method in the mongo shell.

- Groups documents in a collection by the specified keys and performs simple aggregation functions such as computing counts and sums.
- The method is analogous to a SELECT <...> GROUP BY statement in SQL.
- The group() method returns an array.
- **Definition**

db.collection.group({ *key, reduce, initial, [keyf,] [cond,] finalize* })

Field	Type	Description
key	document	The field or fields to group. Returns a “key object” for use as the grouping key.
reduce	function	An aggregation function that operates on the documents during the grouping operation. These functions may return a sum or a count. The function takes two arguments: the current document and an aggregation result document for that group.
initial	document	Initializes the aggregation result document.

keyf	function	Optional. Alternative to the key field. Specifies a function that creates a “key object” for use as the grouping key. Use keyf instead of key to group by calculated fields rather than existing document fields.
cond	documen t	Optional. The selection criteria to determine which documents in the collection to process. If you omit the cond field, db.collection.group() processes all the documents in the collection for the group operation.
finalize	function	Optional. A function that runs each item in the result set before db.collection.group() returns the final value. This function can either modify the result document or replace the result document as a whole.
ns	string	The collection from which to perform the group by operation.

- The db.collection.group() method is a shell wrapper for the group command.

- However, the db.collection.group() method takes **the** keyf field and the reduce field whereas the group command takes the $keyf field and the $reduce field.

- **Example**
- Given a collection named **records** with the following documents:

```
{ a: 1, count: 4 }
{ a: 1, count: 2 }
 { a: 1, count: 4 }
 { a: 2, count: 3 }
 { a: 2, count: 1 }
{ a: 1, count: 5 }
{ a: 4, count: 4 }
```

- Following group operation groups documents by the field a, where a is less than 3, and sums the field count for each group:

```
db.records.group(
{
     key: { a: 1 },
     cond: { a: { $lt: 3 } },
     reduce: function(cur, result)
                      {
                           result.count += cur.count },
                 initial: { count: 0 }
          }
               )
```

- **Output:** [{ a: 1, count: 15 }, { a: 2, count: 4 }]

Group command:

- Groups documents in a collection by the specified key and performs simple aggregation functions, such as computing counts and sums.

- The command is analogous to a SELECT <...> GROUP BY statement in SQL.

- The command returns a document with the grouped records.

- **Syntax:**

```
{ group:
      { ns: <namespace>,
        key: <key>,
        $reduce: <reduce function>,
        $keyf: <key function>,
        cond: <query>,
        finalize: <finalize function>
      }
}
```

❖ Group by Two Fields

- The following example groups by the ord_dt and item.sku fields those documents that have ord_dt greater than 01/07/2015:
- db.runCommand(

```
                        { group:
                {
                 ns: 'orders',
                 key: { ord_dt: 1, 'item.sku': 1 },
                      cond: { ord_dt:
                        { $gt: **new** Date(
 '01/07/2015' ) }},                               $reduce: **function** (
curr, result ) { },
                                     initial: { }
                                    }
                            }
                        )
```

- **db.runCommand()** runs the command in the context of the current database. Some commands are only applicable in the context of the admin database, and you must change your db object to before running these commands.

- The method call is analogous to the SQL statement:

```
SELECT ord_dt, item_sku
FROM orders
 WHERE ord_dt > ‘01/07/2014’
 GROUP BY ord_dt, item_sku
```

```
{ "_id" : 1, "domainName" : "test1.com", "hosting" : "hostgator.com" }

{ "_id" : 2, "domainName" : "test2.com", "hosting" : "aws.amazon.com"}

 { "_id" : 3, "domainName" : "test3.com", "hosting" : "aws.amazon.com" }

{ "_id" : 4, "domainName" : "test4.com", "hosting" : "hostgator.com" }

{ "_id" : 5, "domainName" : "test5.com", "hosting" : "aws.amazon.com" }

 { "_id" : 6, "domainName" : "test6.com", "hosting" : "cloud.google.com" }

{ "_id" : 7, "domainName" : "test7.com", "hosting" : "aws.amazon.com" }

 { "_id" : 8, "domainName" : "test8.com", "hosting" : "hostgator.com" }

 { "_id" : 9, "domainName" : "test9.com", "hosting" : "cloud.google.com" }

{ "_id" : 10, "domainName" : "test10.com", "hosting" : "godaddy.com" }
```

The following example groups by the “hosting” field, and display the total sum of each hosting.

```
> db.website.aggregate(
    {
     $group : {_id : "$hosting", total : { $sum : 1 }  }
    }
                  );
```

 Equivalent query in SQL:-

```
SELECT hosting, SUM(hosting) AS total
FROM website
GROUP BY hosting
```

```
{ "result" :
[
{ "_id" : "godaddy.com", "total" : 1 },
 { "_id" : "cloud.google.com", "total" : 2 },
 { "_id" : "aws.amazon.com", "total" : 4 },
{ "_id" : "hostgator.com", "total" : 3 }
],
"ok" : 1
 }
```

❖ MongoDB Indexing

- Indexes support the efficient execution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. This scan is highly inefficient and require the **mongod** to process a large volume of data.

- Indexes are special data structures.

- The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in index.

- Indexes in MongoDB are similar to indexes in other database systems. MongoDB defines indexes at the collection level and supports indexes on any field or sub-field of the documents in a MongoDB collection.

- **Index Types**
- MongoDB provides a number of different index types.

- You can create indexes on any field or embedded field within a document or sub-document.

- MongoDB also supports indexes of arrays, called multi-key indexes, single field indexes or compound indexes .

- In the mongo shell, you can create an index by calling the ensureIndex() method.

- MongoDB indexes may be ascending, (i.e. 1) or descending (i.e. -1) in their ordering.

- MongoDB indexes use a B-tree data structure.

Single Field Indexes

- **Example**
- Consider **friends** collection:

```
{ "_id" : ObjectId(...), "name" : "Alice" "age" : 27 }
```

- The following command creates an index on the name field:

```
db.friends.ensureIndex( { "name" : 1 } )
```

❑ Indexes on Embedded Fields

- You can create indexes on fields embedded in sub-documents.
- Indexes on embedded fields differ from *indexes on sub-documents*, which include the full content up to the maximum index size of the sub-document in the index.
- Consider a collection named **people** that holds documents that resemble the following example document:

```
{
              "_id": ObjectId(...)
          "name": "John "
              "address":
                { "street": "Main",
                           "zipcode": "53511",
                        "state": "WI"
                }
 }
```

- You can create an index on the address.zipcode field, using the following specification:

    db.people.ensureIndex({ "address.zipcode": 1 })

❑ Indexes on Subdocuments

- For example, the **factories** collection contains documents that contain a metro field, such as:

```
{
  _id: ObjectId(...),
  metro: { city: "New York",
           state: "NY" },
  name: "Giant Factory"
}
```

- The **metro field** is a subdocument, containing the **embedded fields city and state.** The following command creates an index on the metro field as a whole:

```
db.factories.ensureIndex( { metro: 1 } )
```

- The following query can use the index on the metro field:

```
db.factories.find(
                  { metro:
                      { city: "New York",
                          state: "NY"
                      }
                   } )
```

- This query returns the above document. When performing equality matches on subdocuments, field order matters and the subdocuments must match exactly. For example, the following query does not match the above document:

```
db.factories.find( { metro: { state: "NY", city: "New York" } } )
```

- **The ensureIndex() Method**
- **SYNTAX:**

```
>db.COLLECTION_NAME.ensureIndex({KEY:1})
```

- Here key is the name of field on which you want to create index and 1 is for ascending order. To create index in descending order you need to use -1.

- **EXAMPLE**

```
>db.mycol.ensureIndex({"title":1})
```

- In **ensureIndex()** method you can pass multiple fields, to create index on multiple fields.

```
>db.mycol.ensureIndex({"title":1,"description":-1})
```

Compound Indexes

- MongoDB supports ***compound indexes***, where a single index structure holds references to multiple fields within a collection's documents.

- MongoDB supports indexes that include content on a single field, as well as compound indexes that include content from multiple fields.

- **Build a Compound Index**

```
db.collection.ensureIndex( { a: 1, b: 1, c: 1 } )
```

- The value of the field in the index specification describes the kind of index for that field.

- For example, a value of 1 specifies an index that orders items in ascending order. A value of -1 specifies an index that orders items in descending order.

- **Example**
- The following operation will create an index on the item, category, and price fields of the products collection:

```
db.products.ensureIndex( { item: 1, category: 1, price: 1 } )
```

- Compound indexes can support queries that match on multiple fields.

- **Sort Order**
- Indexes store references to fields in either ascending (1) or descending (-1) sort order.

- For single-field indexes, the sort order of keys doesn't matter because MongoDB can traverse the index in either direction. However, for *compound indexes*, sort order can matter in determining whether the index can support a sort operation.

- Consider a collection **events** that contains documents with the fields username and date. Applications can issue queries that return results sorted first by ascending username values and then by descending (i.e. more recent to last) date values, such as:

db.events.find().sort({ username: 1, date: -1 })

- or queries that return results sorted first by descending username values and then by ascending date values, such as:

```
db.events.find().sort( { username: -1, date: 1 } )
```

- The following index can support both these sort operations:

```
db.events.ensureIndex( { "username" : 1, "date" : -1 } )
```

- However, the above index **cannot** support sorting by ascending username values and then by ascending date values, such as the following:

```
db.events.find().sort( { username: 1, date: 1 } )
```

Multikey Indexes

- To index a field that holds an array value, MongoDB adds index items for each item in the array.
- These ***multikey*** indexes allow MongoDB to return documents from queries using the value of an array.
- MongoDB automatically determines whether to create a **multikey** index if the indexed field contains an array value; you do not need to explicitly specify the multikey type.

❖ Limitations

- **Interactions between Compound and Multikey Indexes**
- While you can create multikey *compound indexes*, at most one field in a compound index may hold an array.

- For example, given an index on { a: 1, b: 1 }, the following documents are permissible:

{a: [1, 2], b: 1} {a: 1, b: [1, 2]}

- However, the following document is impermissible, and MongoDB cannot insert such a document into a collection with the {a: 1, b: 1 } index:

```
{a: [1, 2], b: [1, 2]}
```

- If you attempt to insert such a document, MongoDB will reject the insertion, and produce an error that says cannot **index parallel arrays.**

- MongoDB does not index parallel arrays because they require the index to include each value in the Cartesian product of the compound keys, which could quickly result in incredibly large and difficult to maintain indexes.

- **Examples**
- **Index Basic Arrays**
- Given the following document:

```
{ "_id" : ObjectId("..."),
"name" : "Warm Weather",
 "author" : "Steve",
"tags" : [ "weather", "hot", "record", "april" ]
 }
```

- Then an index on the tags field, { tags: 1 }, would be a multikey index and would include these four separate entries for that document:

"weather",

"hot",

"record", and

"april".

- Queries could use the multikey index to return queries for any of the above values.

❖ Index Arrays with Embedded Documents

- You can create multikey indexes on fields in objects embedded in arrays, as in the following example:
- Consider a **feedback** collection with documents in the following form:

```
{ "_id": ObjectId(...),
  "title": "Grocery Quality",
  "comments": [ { author_id: ObjectId(...),
                  date: Date(...),
                  text: "Please expand the selection."
                  },
                { author_id: ObjectId(...),
                  date: Date(...),
          text: "Please expand the mustard selection."
          },
              { author_id: ObjectId(...),
          date: Date(...),
          text: "Please expand the olive selection."
                    }
                ]
 }
```

- An index on the **comments.text** field would be a multikey index and would add items to the index for all embedded documents in the array.

- With the index { "comments.text": 1 } on the feedback collection, consider the following query:

```
db.feedback.find(
    { "comments.text": "Please expand the olive selection." } )
```

- The query would select the documents in the collection that contain the following embedded document in the comments array:

```
{
        author_id: ObjectId(...),
        date: Date(...),
         text: "Please expand the olive selection."
}
```

❖ Array of Embedded Documents

- Consider that the inventory collection includes the following documents:

```
{
_id: 100,
type: "food",
item: "xyz",
qty: 25,
price: 2.5,
ratings: [ 5, 8, 9 ],
memos: [ { memo: "on time", by: "shipping" },
         { memo: "approved", by: "billing" } ]
 }
```

```
{
_id: 101,
type: "fruit",
item: "jkl",
qty: 10,
price: 4.25,
ratings: [ 5, 9 ],
memos: [ { memo: "on time", by: "payment" },
         { memo: "delayed", by: "shipping" }
       ]
}
```

❖ Match a Field in the Embedded Document Using the Array Index

- If you know the array index of the embedded document, you can specify the document using the subdocument's position using the dot notation.

- The following example selects all documents where the **memos** contains an array whose first element (i.e. index is 0) is a document that contains the field **by** whose value is **'shipping'**:

```
db.inventory.find( { 'memos.0.by': 'shipping' } )
```

- The operation returns the following document:

```
{
_id: 100,
type: "food",
item: "xyz",
qty: 25,
price: 2.5,
ratings: [ 5, 8, 9 ],
memos: [ { memo: "on time", by: "shipping" },
            { memo: "approved", by: "billing" }
          ]
}
```

❖ Match a Field Without Specifying Array Index

- If you do not know the index position of the document in the array, concatenate the name of the field that contains the array, with a dot (.) and the name of the field in the subdocument.

- The following example selects all documents where the memos field contains an array that contains at least one embedded document that contains the field by with the value ‘shipping’:

```
db.inventory.find( { 'memos.by': 'shipping' } )
```

- **The operation returns the following documents:**

```
{
_id: 100,
type: "food",
item: "xyz",
qty: 25,
price: 2.5,
ratings: [ 5, 8, 9 ],
memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing"
    } ]
}
{
_id: 101,
type: "fruit",
item: "jkl",
qty: 10,
price: 4.25,
ratings: [ 5, 9 ],
memos: [ { memo: "on time", by: "payment" }, { memo: "delayed", by: "shipping"
    } ]
}
```

Create a Unique Index

- MongoDB allows you to specify a unique constraint on an index. These constraints prevent applications from inserting documents that have duplicate values for the inserted fields.

- Additionally, if you want to create an index on a collection that has existing data that might have duplicate values for the indexed field, then use duplicate dropping.

- **Unique Indexes**
- To create a unique index, consider the following prototype:

```
db.collection.ensureIndex( { a: 1 }, { unique: true } )
```

- **For example,** you may want to create a unique index on the "tax-id": of the accounts collection to prevent storing multiple account records for the same legal entity:

```
db.accounts.ensureIndex( { "tax-id": 1 }, { unique: true } )
```

- **Unique Constraint Across Separate Documents**
- The unique constraint applies to separate documents in the collection.

- That is, the unique index prevents *separate* documents from having the same value for the indexed key, but the index does not prevent a document from having multiple elements or embedded documents in an indexed array from having the same value.

- In the case of a single document with repeating values, the repeated value is inserted into the index only once.

- For example, a collection has a unique index on a.b:

```
db.collection.ensureIndex( { "a.b": 1 }, { unique: true } )
```

- The unique index permits the insertion of the following document into the collection if no other document in the collection has the a.b value of 5:

```
db.collection.insert( { a: [ { b: 5 }, { b: 5 } ] } )
```

- **Drop Duplicates**
- MongoDB cannot create a *unique index* on a field that has duplicate values.

- To force the creation of a unique index, you can specify the dropDups option, which will only index the first occurrence of a value for the key, and delete all subsequent values.

- To create an unique index that drops duplicates on the username field of the accounts collection, use a command in the following form:

```
db.accounts.ensureIndex(
                        { username: 1 },
                        { unique: true,
                          dropDups: true }
                        )
```

- Specifying { dropDups: true } will delete data from your database.

- By default, dropDups is false.

Index Names

- The default name for an index is the concatenation of the indexed keys and each key's direction in the index, 1 or -1.

- **Example**
- Consider the following command to create an index on item and quantity:

```
db.products.ensureIndex( { item: 1, quantity: -1 } )
```

- The resulting index is named: item_1_quantity_-1.

- Optionally, you can specify a name for an index instead of using the default name.

- **Example**
- Issue the following command to create an index on item and quantity and specify inventory as the index name:

```
db.products.ensureIndex(
                    { item: 1, quantity: -1 } ,
                    { name: "inventory" }
                                  )
```

- The resulting index has the name inventory.

- To view the name of an index, use the getIndexes() method.

Create a Sparse Index

- Sparse omit references to documents that do not include the indexed field. For fields that are only present in some documents sparse indexes may provide a significant space savings.

- To create a *sparse index* on a field, use following operation

```
db.collection.ensureIndex( { a: 1 }, { sparse: true } )
```

- **Example**
- The following operation, creates a sparse index on the users collection that *only* includes a document in the index if the twitter_name field exists in a document.

```
db.users.ensureIndex(
                    { twitter_name: 1 }, { sparse: true } )
```

- The index excludes all documents that do not include the twitter_name field.

- **Example:Create a Sparse Index On A Collection**
- Consider a collection scores that contains the following documents:

```
{    "_id" : ObjectId("523b6e32fb408eea0eec2647"),
     "userid" : “abc”
 }
 {       "_id" : ObjectId("523b6e61fb408eea0eec2648"),
       "userid" : “xyz",
        "score" : 82 }
{     "_id" : ObjectId("523b6e6ffb408eea0eec2649"),
      "userid" : “lmn",
      "score" : 90
 }
```

- The collection has a sparse index on the field score:

db.scores.ensureIndex({ score: 1 } , { sparse: true })

- Then, the following query on the scores collection uses the sparse index to return the documents that have the score field less than ($lt) 90:

db.scores.find({ score: { $lt: 90 } })

- Because the document for the userid “abc" does not contain the score field and thus does not meet the query criteria, the query can use the sparse index to return the results:

-
  ```
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"),
  "userid" : “xyz",
   "score" : 82 }
```

- **Sparse Index On A Collection Cannot Return Complete Results**
- Consider a collection scores that contains the following documents:

{ "_id" : ObjectId("523b6e32fb408eea0eec2647"), "userid" : “abc”
}

{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : “xyz", "score" : 82 }

{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid" : “lmn", "score" : 90 }

- The collection has a sparse index on the field score:

```
db.scores.ensureIndex( { score: 1 } , { sparse: true } )
```

- Because the document for the userid “abc" does not contain the score field, the sparse index does not contain an entry for that document.

- To use the sparse index, explicitly specify the index with hint():

```
db.scores.find().sort( { score: -1 } ).hint( { score: 1 } )
```

- The use of the index results in the return of only those documents with the score field:

```
{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid" : “lmn",
   "score" : 90
}

{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : “xyz”,
   "score" : 82
}
```

- Consider the following operation:

```
db.users.find().hint( { score: 1 } )
```

- This operation returns all documents in the collection named **users** using the index on the **score** field.

- **Sparse Index with Unique Constraint**
- Consider a collection scores that contains the following documents:

```
{ "_id" : ObjectId("523b6e32fb408eea0eec2647"),
  "userid" : "newbie"
 }
 { "_id" : ObjectId("523b6e61fb408eea0eec2648"),
   "userid" : "abby", "score" : 82
  }
 { "_id" : ObjectId("523b6e6ffb408eea0eec2649"),
  "userid" : "nina", "score" : 90 }
```

- You could create an index with a unique constraint and sparse filter on the score field using the following operation:

```
db.scores.ensureIndex( { score: 1 } , { sparse: true, unique: true } )
```

- This index *would permit* the insertion of documents that had unique values for the score field *or* did not include a score field.

- Consider the following *insert operation*:

```
db.scores.insert( { "userid": "AAAAAAA", "score": 43 } )
db.scores.insert( { "userid": "BBBBBBB", "score": 34 } )
db.scores.insert( { "userid": "CCCCCCC" } )
db.scores.insert( { "userid": "DDDDDDD" } )
```

- However, the index *would not permit* the addition of the following documents since documents already exists with score value of 82 and 90:

```
db.scores.insert( { "userid": "AAAAAAA", "score": 82 } )
db.scores.insert( { "userid": "BBBBBBB", "score": 90 } )
```

Create a Hashed Index

- Hashed indexes compute a hash of the value of a field in a collection and index the hashed value.
- MongoDB automatically computes the hashes when resolving queries using hashed indexes. Applications do **not** need to compute hashes.
- MongoDB supports hashed indexes of any single field.
- The hashing function collapses embedded documents and computes the hash for the entire value, but does not support multi-key (i.e. arrays) indexes.

- **Hashed indexes** maintain entries with hashes of the values of the indexed field.

- To create a *hashed index*, specify hashed as the value of the index key, as in the following example:

- **Example**
- Specify a hashed index on _id

```
db.collection.ensureIndex( { _id: "hashed" } )
```

- **MongoDB Java**

✔ **Installation**

- Before we start using MongoDB in our Java programs, we need to make sure that we have MongoDB JDBC Driver and Java set up on the machine.
- You need to download the jar file(**Download mongo.jar**). Make sure to download latest release of it.
- You need to include the **mongo.jar** into your classpath.

✔ **Connect to database**

- To connect database, you need to specify database name, if database doesn't exist then mongodb creates it automatically.

```
import com.mongodb.MongoClient;
 import com.mongodb.MongoException;
import com.mongodb.WriteConcern;
 import com.mongodb.DB;
 import com.mongodb.DBCollection;
 import com.mongodb.BasicDBObject;
 import com.mongodb.DBObject;
 import com.mongodb.DBCursor;
import com.mongodb.ServerAddress;
 import java.util.Arrays;
public class MongoDBJDBC
{
 public static void main( String args[] )
{
   try
        { // To connect to mongodb server
  MongoClient mongoClient = new MongoClient( "localhost" , 27017 );
// Now connect to your databases
                    DB db = mongoClient.getDB( "test" );
System.out.println("Connect to database successfully");
 boolean auth = db.authenticate(myUserName, myPassword);
System.out.println("Authentication: "+auth); }
catch(Exception e)
    {
        System.err.println( e.getClass().getName() + ": " +
                    e.getMessage() );
    }
}
}
```

- Now, let's compile and run above program to create our database test. You can change your path as per your requirement.

```
$javac MongoDBJDBC.java
$java -classpath ".:mongo-2.10.1.jar" MongoDBJDBC
Connect to database successfully
Authentication: true
```

References

- http://docs.mongodb.org/manual/ or
- SQL/XML/MongoDB (https://www.w3schools.com/)
- https://www.tutorialspoint.com/mongodb/
- https://www.json.org/
- https://www.tutorialspoint.com/json/

END

BIG DATA

"***Big Data***" is data whose scale, diversity, and complexity require new architecture, techniques, algorithms, and analytics to manage it and extract value and hidden knowledge from it…

- **Big data** is a term for data sets that are so large or complex that traditional data processing applications are inadequate. Challenges include analysis, capture, data curation, search, sharing, storage, transfer, visualization, querying, updating and information privacy.

5 V's of Big data:

- Velocity, Volume, Value, Variety, and Veracity.
- **1)Velocity-**velocity refers to the speed at which vast amounts of data are being generated, collected and analyzed. Every day the number of emails, twitter messages, photos, video clips, etc. increases at lighting speeds around the world. Every second of every day data is increasing. Not only must it be analyzed, but the speed of transmission, and access to the data must also remain instantaneous to allow for real-time access to website, credit card verification and instant messaging. Big data technology allows us now to analyze the data while it is being generated, without ever putting it into databases.
- **2)Volume**-Volume refers to the incredible amounts of data generated each second from social media, cell phones, cars, credit cards, M2M sensors, photographs, video, etc. The vast amounts of data have become so large in fact that we can no longer store and analyze data using traditional database technology.

- **3)Value-**When we talk about value, we're referring to the worth of the data being extracted. Having endless amounts of data is one thing, but unless it can be turned into value it is useless. While there is a clear link between data and insights, this does not always mean there is value in Big Data.
- **4)Variety-**Variety is defined as the different types of data we can now use. Today's data is unstructured. In fact, 80% of all the world's data fits into this category, including photos, video sequences, social media updates, etc. New and innovative big data technology is now allowing structured and unstructured data to be harvested, stored, and used simultaneously.
- **5)Veracity-**Veracity is the quality or trustworthiness of the data. Just how accurate is all this data? For example, think about all the Twitter posts with hash tags, abbreviations, typos, etc., and the reliability and accuracy of all that content.

References

- https://www.xsnet.com/blog/bid/205405/the-v-s-of-big-data-velocity-volume-value-variety-and-veracity

END