UNIT – 2

Advanced Database Techniques: Structured verses unstructured data, Concept of NOSQL database, Comparative study of SQL and NOSQL, Databases types, NOSQL Data Modeling, Benefits of NOSQL, NOSQL using MongoDB- MongoDB shell, data types, manipulation (insert, update, delete documents), querying, aggregation, indexing, crowd-sourcing.

1. Compare Structured Data, SQL and Unstructured Data
   • 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).
   • Structured data was a huge improvement over strictly paper-based unstructured systems, but life doesn’t always fit into neat little boxes. As a result, the structured data always had to be supplemented by paper or microfilm storage. As technology performance has continued to improve, and prices have dropped, it was possible to bring into computing systems unstructured and semi-structured data.
   • Unstructured data is all those things that can’t be so readily classified and fit into a neat box: photos and graphic images, videos, streaming instrument data, webpages, pdf files, PowerPoint presentations, emails, blog entries, wikis and word processing documents.
   • 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 rigid 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.

2. What is NoSQL?
   • NoSQL databases differ from the traditional relational database management system as they do not require data to fit a schema. This is especially useful when you are working with large amounts of data that don’t necessarily fit a structure, as data can be stored without a schema meaning there is no need for a fixed data model. NoSQL databases also differ from relational models as they have the ability to scale out, and take advantage of new nodes which is of particular importance presently as transaction rates and availability requirements are increasing.

3. What are the benefits of NoSQL?
   • NoSQL database differs from the traditional RDBMS as they do not require data to fit in a schema. Utilizing the NoSQL DB gives organizations access to range of benefits including the following:
   • **Elastic Scaling:** organizations are able to scale out and take advantage of new nodes according to their data storage needs.
• **No need for data to fit a schema:** both structured and unstructured data can be stored as there is no fixed data model. This flexibility gives organizations access to much larger quantities of data.

• **Ability to cope with hardware failure:** accepting that hardware failures will occur meant the NoSQL database was designed with redundancy in mind.

• **Quick and easy development:** it is easy to change how data is stored using refactoring or batch processing.

• These benefits mean the NoSQL database is ideally suited to those organizations that need a database which can cope with large amounts of disparate data.

4. **State and explain the Advantages and Disadvantages of NoSQL?**
   - **Advantages:**
     1) **Scalable**
        - NoSQL’s Elastic Scaling is precisely what makes it so well-suited for big data. Relational databases tend often to ‘scale up’: they add larger, more powerful servers as the database load begins to increase. In the case of big data – which is likely to grow at a breakneck pace- this simply isn’t a viable choice. It’s thus far better to ‘scale out’ instead; distributing the database across multiple hosts in order to efficiently manage server load.

     2) **Flexible**
        - A NoSQL database is considerably less restricted than an SQL database, mainly because it’s not locked into any one specific data model (this also forms the crux of one of its chief disadvantages, but more on that in a moment). Applications can store data in virtually any structure or format necessary, making change management a breeze. Ultimately, this means more up-time and better reliability. Contrast this against relational databases, which must be strictly and attentively managed; where even a minor change may result in downtime or a reduction of service.

     3) **Administrator-Friendly**
        - NoSQL databases tend more often than not to be considerably less complex and considerably simpler to deploy than their relational cousins. This is because, as noted by Tech Nirvana, they’re “designed from the ground up to require less management, with automatic repair, data distribution, and simple data models.” All these factors together ultimately lead to a database which requires considerably less overhead management.

     4) **Cost-Efficient and Open Source**
        - The servers utilized in a NoSQL implementation are typically cheap, low-grade commodity devices, as opposed to the oft-expensive servers and storage systems required in relational databases. That’s not the only thing that drives down the cost, either. NoSQL is entirely open-source, meaning generally higher reliability security, and speed of deployment.

     5) **Limitless**
        - NoSQL meshes naturally with cloud computing. This is due to a couple of factors. Foremost among these is that NoSQL’s horizontal scaling meshes extremely well with the cloud, allowing them to take full advantage of cloud computing’s elastic scaling. In addition, the ease of deployment and
management within a NoSQL database (and its focus on big data) make it a prime partner for cloud computing, allowing administrators to focus more on the software side of things rather than having to worry about what hardware they’re using.

- **Disadvantages:**
  
  1. **Narrow Focus**
  
  One of the primary reasons that NoSQL will never wholly replace SQL is that it was never meant to do so. NoSQL databases have a very narrow focus: they are designed primarily for storage, and offer very little functionality beyond. When transactions enter the equation, relational databases are still the better choice. Further, NoSQL doesn’t really do so well with data backup on its own.

  2. **Standardization**
  
  That NoSQL is open-source could at once be considered its greatest strength and its greatest weakness. The truth is, there really aren’t many reliable standards for NoSQL databases quite yet, meaning that no two databases are likely to be created equal. Getting a particular implementation to play nice with existing infrastructure can thus be something of a crap-shoot, while support could end up being spotty when compared against a more traditional database implementation.

  3. **Performance and Scaling**
  
  Because of the way data is stored and managed in a NoSQL database, data consistency might well end up being a concern. NoSQL puts performance and scalability first; consistency isn’t really a consideration. Depending on what you’re using it for, this could actually be either a crippling weakness or a powerful strength. In certain situations – such as when you’re dealing with a massive onslaught of unstructured data – this is completely acceptable. In other situations, such as management of financial records, it most certainly is not.

  4. **Lack of Maturity**
  
  While it’s certainly true that NoSQL isn’t exactly the new kid on the block (the underlying technology has existed for at least ten years now), widespread acceptance of NoSQL still lags; compared to traditional relational databases the technology is still relatively immature. This is reflected also in a lack of developers and administrators with the right knowledge and skills: NoSQL may be administrator and developer-friendly, true, but that means nothing if neither administrator nor developer have the tools or understanding to address it.

  Relational databases are much better established in the enterprise world, and thus enjoy more functionality, greater acceptance, and a wealth of professionals who actually understand how to manage them.

  5. **Doesn’t play nice with Analytics**
  
  Admittedly, this is a weakness which has been addressed in recent years, with the emergence of startups like Prelog. Even so, NoSQL doesn’t necessarily mesh well with traditional BI applications and platforms. It might well be less complex than SQL in many areas, but where analytics is concerned, it has the very real potential to become a complicated, difficult-to-decipher behemoth.
5. What is Querying?
   - Query is a statement that performs operations on a database. There are various types of queries provided by NoSQL some of them are as follows:

1) Find
   - Find Query returns a subset of document in a collection. This query finds the document with the help of matching the arguments with the document.
   - Syntax: `>db.user.find({ "username" : "ABC" , "age" : "20" })`
   - This query will search for document with username ABC and age 20.

2) OR
   - There are two ways to do an OR query in MongoDB. "$in" can be used to query for a variety of values for a single key. "$or" is more general; it can be used to query for any of the given values across multiple keys.
   - If you have more than one possible value to match for a single key, use an array of criteria with "$in". For instance, suppose we were running a raffle and the winning ticket numbers were 725, 542, and 390. To find all three of these documents, we can construct the following query:
   - `>db.raffle.find({"ticket_no" : {"$in" : [725, 542, 390]}})`
   - The opposite of "$in" is "$nin", which returns documents that don’t match any of the criteria in the array. If we want to return all of the people who didn’t win anything in the raffle, we can query for them with this:
   - `>db.raffle.find({"ticket_no" : {"$nin" : [725, 542, 390]}})`
   - "$in" gives you an OR query for a single key, but what if we need to find documents where "ticket_no" is 725 or "winner" is true? For this type of query, we’ll need to use the "$or" conditional. "$or" takes an array of possible criteria. In the raffle case, using "$or" would look like this:
   - `>db.raffle.find({"$or" : [{"ticket_no" : 725}, {"winner" : true}]}]

3) Where
   - Key/value pairs are a fairly expressive way to query, but there are some queries that they cannot represent. For queries that cannot be done any other way, there are "$where" clauses, which allow you to execute arbitrary JavaScript as part of your query. This allows you to do (almost) anything within a query.
   - The most common case for this is wanting to compare the values for two keys in a document, for instance, if we had a list of items and wanted to return documents where any two of the values are equal. Here’s an example:
   - `>db.foo.insert({"apple" : 1, "banana" : 6, "peach" : 3})`
   - `>db.foo.insert({"apple" : 8, "spinach" : 4, "watermelon" : 4})`
   - In the second document, "spinach" and "watermelon" have the same value, so we’d like that document returned. It’s unlikely MongoDB will ever have a $ conditional for this, so we can use a "$where" clause to do it with JavaScript:
   - `>db.foo.find({"$where" : function () { ... for (var current in this) {`
... for (var other in this) {
  ... if (current != other && this[current] == this[other]) {
    ... return true;
  ...
  }
  ...
  ...
  return false;
  ...
});

- If the function returns true, the document will be part of the result set; if it returns false, it won’t be.

We used a function earlier, but you can also use strings to specify a "$where" query; the following two "$where" queries are equivalent:

  > db.foo.find({"$where" : "this.x + this.y == 10"})
  > db.foo.find({"$where" : "function() { return this.x + this.y == 10; }"})

- "$where" queries should not be used unless strictly necessary: they are much slower than regular queries. Each document has to be converted from BSON to a JavaScript object and then run through the "$where" expression. Indexes cannot be used to satisfy a "$where", either. Hence, you should use "$where" only when there is no other way of doing the query. You can cut down on the penalty by using other query filters in combination with "$where". If possible, an index will be used to filter based on the non- $where clauses; the "$where" expression will be used only to fine-tune the results.

6. Explain Crowd-Sourcing.

- Crowd sourcing was introduced by Jeff Howe and Mark Robinson in Wired Magazine in June 2006.

- In crowd Sourcing model organizations get access to a larger talent crowd and data scientists get access to a larger and wider diversity of data to occur up with better business analytics solutions.

- Crowd sourcing addresses problem solving by considering
  - Predictive analysis
  - Descriptive Analysis
  - Estimations
  - Hypothesis Validations

- PROBLEMS WITH CROWDSOURCING
  - Quality
  - Intellectual property leakage
  - No time constraint
  - Not much control over development or ultimate product
  - Ill-will with own employees
  - Choosing what to crowdsource & what to keep in-house

- SOME APPLICATIONS OF CROWDSOURCING
  - Testing & Refining a Product
    - Netflix
    - SellaBand
  - Market Research
    - Threadless
  - Knowledge Management
7. Explain MongoDB Shell.

- MongoDB comes with a JavaScript shell that allows interaction with a MongoDB instance from the command line. The shell is very useful for performing administrative functions, inspecting a running instance, or just playing around. The mongo shell is a crucial tool for using MongoDB.
- Running the Shell
  To start the shell, run the mongo executable:
  ```
  $ ./mongo
  MongoDB shell version: 1.6.0
  url: test
  connecting to: test
type "help" for help
  >
  ```
  - The shell automatically attempts to connect to a MongoDB server on startup, so make sure you start mongod before starting the shell.
  - The shell is a full-featured JavaScript interpreter, capable of running arbitrary JavaScript programs.
  To illustrate this, let's perform some basic math:
  ```
  > x = 200
  200
  > x / 5;
  40
  ```
  - We can also leverage all of the standard JavaScript libraries:
    ```
    > Math.sin(Math.PI / 2);
    1
    > new Date("2010/1/1");
    "Fri Jan 01 2010 00:00:00 GMT-0500 (EST)"
    > "Hello, World!".replace("World", "MongoDB");
    Hello, MongoDB!
    ```
  - We can even define and call JavaScript functions:
    ```
    > function factorial (n) {
    ... if (n <= 1) return 1;
    ... return n * factorial(n - 1);
    ... }
    > factorial(5);
    120
    ```
  - Note that you can create multiline commands. The shell will detect whether the JavaScript statement is complete when you press Enter and, if it is not, will allow you to continue writing it on the next line.
A MongoDB Client

Although the ability to execute arbitrary JavaScript is cool, the real power of the shell lies in the fact that it is also a stand-alone MongoDB client. On startup, the shell connects to the test database on a MongoDB server and assigns this database connection to the global variable `db`. This variable is the primary access point to MongoDB through the shell.

The shell contains some add-ons that are not valid JavaScript syntax but were implemented because of their familiarity to users of SQL shells. The add-ons do not provide any extra functionality, but they are nice syntactic sugar. For instance, one of the most important operations is selecting which database to use:

```
> use foobar
switched to dbfoo
```

Now if you look at the `db` variable, you can see that it refers to the `foobar` database:

```
>db
foobar
```

Because this is a JavaScript shell, typing a variable will convert the variable to a string (in this case, the database name) and print it.

Collections can be accessed from the `db` variable. For example, `db.baz` returns the `baz` collection in the current database. Now that we can access a collection in the shell, we can perform almost any database operation.

Basic Operations with the Shell

We can use the four basic operations, create, read, update, and delete (CRUD), to manipulate and view data in the shell.

8. Explain NoSQL/MongoDB datatypes.

MongoDB adds support for a number of additional data types while keeping JSON’s essential key/value pair nature. Exactly how values of each type are represented varies by language, but this is a list of the commonly supported types and how they are represented as part of a document in the shell:

- **Null**
  Null can be used to represent both a null value and a nonexistent field:
  
  ```json
  {"x" : null}
  ```

- **Boolean**
  There is a Boolean type, which will be used for the values 'true' and 'false':
  
  ```json
  {"x" : true}
  ```

- **64-bit floating point number**
  All numbers in the shell will be of this type. Thus, this will be a floating-point number:
  
  ```json
  {"x" : 3.14}
  ```
  As will this:
  
  ```json
  {"x" : 3}
  ```

- **String**
  Any string of UTF-8 characters can be represented using the string type:
  
  ```json
  {"x" : "foobar"}
  ```

- **Object id**
  An object id is a unique 12-byte ID for documents.
  
  ```json
  {"x" : ObjectId()}
  ```
- **Date**
  Dates are stored as milliseconds since the epoch. The time zone is not stored:
  ```javascript
  {"x" : new Date()}
  ```

- **Regular expression**
  Documents can contain regular expressions, using JavaScript’s regular expression Syntax:
  ```javascript
  {"x" : /foobar/i}
  ```

- **Code**
  Documents can also contain JavaScript code:
  ```javascript
  {"x" : function() { /* ... */ }}
  ```

- **Undefined**
  Undefined can be used in documents as well (JavaScript has distinct types for null and undefined):
  ```javascript
  {"x" : undefined}
  ```

- **Array**
  Sets or lists of values can be represented as arrays:
  ```javascript
  {"x" : ["a", "b", "c"]}
  ```

- **Embedded document**
  Documents can contain entire documents, embedded as values in a parent document:
  ```javascript
  {"x" : {"foo" : "bar"}}
  ```

9. **Comparison between SQL and NoSQL.**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>SQL</th>
<th>NoSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Storage Model</td>
<td>Relational DBMS</td>
<td>Document-oriented</td>
</tr>
<tr>
<td>JOINs</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Transaction</td>
<td>ACID</td>
<td>No</td>
</tr>
<tr>
<td>Support agile practices</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Data schema</td>
<td>Fixed</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Scalability</td>
<td>Vertical</td>
<td>Horizontal</td>
</tr>
<tr>
<td>Replication</td>
<td>Yes (depending on software edition)</td>
<td>Primary-Secondary</td>
</tr>
<tr>
<td>MapReduce</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Query Language</td>
<td>SQL query language</td>
<td>JSON query language</td>
</tr>
<tr>
<td>Secondary Indexes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Triggers</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Foreign keys</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Company</td>
<td>Microsoft</td>
<td>MongoDB, Inc</td>
</tr>
<tr>
<td>Licence</td>
<td>Commercial</td>
<td>Open Source</td>
</tr>
<tr>
<td>Implementation language</td>
<td>C++</td>
<td>C++</td>
</tr>
<tr>
<td>OS support</td>
<td>Windows</td>
<td>Windows, Linux, OS X, Solaris</td>
</tr>
</tbody>
</table>

10. Explain CURD operation?
We can use the four basic operations, create, read, update, and delete (CRUD), to manipulate and view data in the shell.

**Create:**
The insert function adds a document to a collection. For example, suppose we want to store a blog post. First, we'll create a local variable called `post` that is a JavaScript object representing our document. It will have the keys "title", "content", and "date" (the date that it was published):

```javascript
post = {
  "title" : "My Blog Post",
  "content" : "Here's my blog post.",
  "date" : new Date()
}
```

This object is a valid MongoDB document, so we can save it to the blog collection using the insert method:

```javascript
> db.blog.insert(post)
```

The blog post has been saved to the database. We can see it by calling `find` on the collection:

```javascript
> db.blog.find()
```

**Read:**
Find returns all of the documents in a collection. If we just want to see one document from a collection, we can use `findOne`:

```javascript
> db.blog.findOne()
```
Find and findOne can also be passed criteria in the form of a query document. This will restrict the
documents matched by the query. The shell will automatically display up to 20 documents
matching a find, but more can be fetched. See Chapter 4 for more information on querying.

**Update:**
If we would like to modify our post, we can use update. Update takes (at least) two parameters:
the first is the criteria to find which document to update, and the second is the new document.
Suppose we decide to enable comments on the blog post we created earlier. We’ll need to add an
array of comments as the value for a new key in our document.
The first step is to modify the variable post and add a "comments" key:
> post.comments = []
[]
Then we perform the update, replacing the post titled “My Blog Post” with our new version of the
document:
> db.blog.update({title : "My Blog Post"}, post)
Now the document has a "comments" key. If we call find again, we can see the new key:
> db.blog.find()
{
  "_id" :ObjectId("4b23c3ca7525f35f94b60a2d"),
  "title" : "My Blog Post",
  "content" : "Here's my blog post.",
  "date" : "Sat Dec 12 2009 11:23:21 GMT-0500 (EST)"
  "comments" : [ ]
}

**Delete:**
Remove deletes documents permanently from the database. Called with no parameters, it
removes all documents from a collection. It can also take a document specifying criteria for
removal. For example, this would remove the post we just created:
> db.blog.remove({title : "My Blog Post"})

11. Explain the feature of MongoDB/NoSQL?
- MongoDB is a powerful, flexible, and scalable data store. It combines the ability to scale out with
many of the most useful features of relational databases, such as secondary indexes, range
queries, and sorting. MongoDB is also incredibly featureful: it has tons of useful features such as
built-in support for Map-Reduce-style aggregation and geospatial indexes.

1) Rich Data Model
2) Scaling
3) Speed
4) Stored JavaScript
5) Aggregation
6) Fixed-Size Collection
7) **File Storage**

1. **A Rich Data Model**
   - MongoDB is a *document-oriented database*, not a relational one. The primary reason for moving away from the relational model is to make scaling out easier.
   - The basic idea is to replace the concept of a “row” with a more flexible model, the “document.” By allowing embedded documents and arrays, the document-oriented approach makes it possible to represent complex hierarchical relationships with a single record.
   - MongoDB is also schema-free: a document’s keys are not predefined or fixed in any way. Without a schema to change, massive data migrations are usually unnecessary.
   - New or missing keys can be dealt with at the application level, instead of forcing all data to have the same shape. This gives developers a lot of flexibility in how they work with evolving data models.

2. **Scaling**
   - Data set sizes for applications are growing at an incredible pace. Advances in sensor technology, increases in available bandwidth, and the popularity of handheld devices that can be connected to the Internet have created an environment where even small-scale applications need to store more data than many databases were meant to handle.
   - A terabyte of data, once an unheard-of amount of information, is now commonplace.
   - As the amount of data that developers need to store grows, developers face a difficult decision: how should they scale their databases? Scaling a database comes down to the choice between scaling up (getting a bigger machine) or scaling out (partitioning data across more machines). Scaling up is often the path of least resistance, but it has drawbacks: large machines are often very expensive, and eventually a physical limit is reached where a more powerful machine cannot be purchased at any cost. For the type of large web application that most people aspire to build, it is either impossible or not cost-effective to run off of one machine. Alternatively, it is both extensible and economical to scale out: to add storage space or increase performance, you can buy another commodity server and add it to your cluster.
   - MongoDB was designed from the beginning to scale out. Its document-oriented data model allows it to automatically split up data across multiple servers. It can balance data and load across a cluster, redistributing documents automatically. This allows developers to focus on programming the application, not scaling it. When they need more capacity, they can just add new machines to the cluster and let the database figure out how to organize everything.

3. **Speed**
   - Incredible performance is a major goal for MongoDB and has shaped many design decisions. MongoDB uses a binary wire protocol as the primary mode of interaction with the server (as opposed to a protocol with more overhead, like HTTP/REST). It adds dynamic padding to documents and pre-allocates data files to trade extra space usage for consistent performance. It uses memory-mapped files in the default storage engine, which pushes the responsibility for memory management to the operating system.
   - It also features a dynamic query optimizer that “remembers” the fastest way to perform a query. In short, almost every aspect of MongoDB was designed to maintain high performance.
Although MongoDB is powerful and attempts to keep many features from relational systems, it is not intended to do everything that a relational database does. Whenever possible, the database server offloads processing and logic to the client side (handled either by the drivers or by a user’s application code). Maintaining this streamlined design is one of the reasons MongoDB can achieve such high performance.

**Simple Administration**

MongoDB tries to simplify database administration by making servers administrate themselves as much as possible. Aside from starting the database server, very little administration is necessary. If a master server goes down, MongoDB can automatically failover to a backup slave and promote the slave to a master. In a distributed environment, the cluster needs to be told only that a new node exists to automatically integrate and configure it.

MongoDB’s administration philosophy is that the server should handle as much of the configuration as possible automatically, allowing (but not requiring) users to tweak their setups if needed.

4. **Stored JavaScript**
   - Instead of stored procedures, developers can store and use JavaScript functions and values on the server side.

5. **Aggregation**
   - MongoDB supports Map Reduce and other aggregation tools.

6. **Fixed-size collections**
   - Capped collections are fixed in size and are useful for certain types of data, such as logs.

7. **File storage**
   - MongoDB supports an easy-to-use protocol for storing large files and file metadata. Some features common to relational databases are not present in MongoDB, notably joins and complex multirow transactions. These are architectural decisions to allow for scalability, because both of those features are difficult to provide efficiently in a distributed system.