MONGODB

Content

- Part 1: Introduction & Basics
- 2: CRUD
- 3: Schema Design
- 4: Indexes
- 5: Aggregation
- 6: Replication & Sharding

History

mongoDB = “Hu**mongo**us DB”

- Open-source
- The port 27017 is used for mongoDB server.
- Document-based
- “High performance, high availability”
- Automatic scaling
- C-P on CAP

-blog.mongodb.org/post/475279604/on-distributed-consistency-part-1

-mongodb.org/manual

Other NoSQL Types

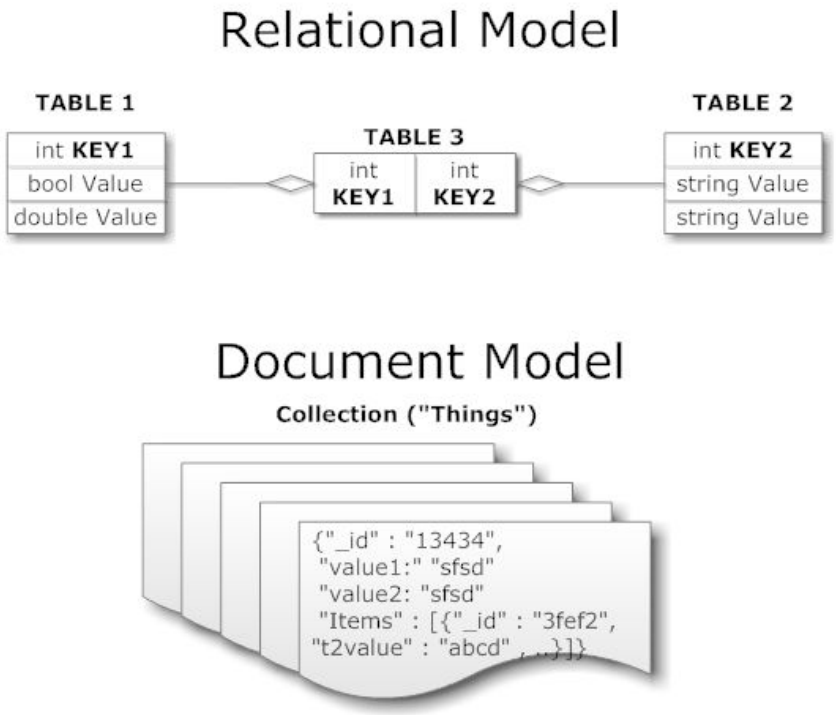
Key/value (Dynamo)

Columnar/tabular (HBase)

Document (mongoDB)

Graph (Neo4j)

http://www.aaronstannard.com/post/2011/06/30/MongoDB-vs-SQL-Server.aspx



Motivations

Problems with SQL

- Rigid schema
- Not easily scalable (designed for 90's technology or worse)
- Requires unintuitive joins

Perks of mongoDB

- Easy interface with common languages (Java, Javascript, PHP, etc.)
- DB tech should run anywhere (VM's, cloud, etc.)
- Keeps essential features of RDBMS's while learning from key-value noSQL systems

Company Using mongoDB

"MongoDB powers Under Armour's online store, and was chosen for its dynamic schema, ability to scale horizontally and perform multi-data center replication."

http://www.mongodb.org/about/production-deployments/

In Good Company

mongoDB

-Steve Francia, http://www.slideshare.net/spf13/mongodb-9794741?v=qf1&b=&from_search=13

The _id Field

- By default, each document contains an _id field. This field has a number of special characteristics:
  - Value serves as primary key for collection.
  - Value is unique, immutable, and may be any non-array type.
  - Default data type is ObjectId, which is “small, likely unique, fast to generate, and ordered.” Sorting on an ObjectId value is roughly equivalent to sorting on creation time.

http://docs.mongodb.org/manual/reference/bson-types/

mongoDB vs. SQL

mongoDB	SQL
Document	Tuple
Collection	Table/View
PK: _id Field	PK: Any Attribute(s)
Uniformity not Required	Uniform Relation Schema
Index	Index
Embedded Structure	Joins
Shard	Partition

CRUD

Create, Read, Update, Delete

Getting Started with mongoDB

To install mongoDB, go to this link and click on the appropriate OS and architecture: http://www.mongodb.org/downloads

First, extract the files (preferrably to the C drive).

Finally, create a data directory on C:\ for mongoDB to use
i.e. “md data” followed by “md data\db”

http://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows/

Getting Started with mongoDB

Open your mongodb/bin directory and run mongod.exe to start the database server.

To establish a connection to the server, open another command prompt window and go to the same directory, entering in mongo.exe. This engages the mongodb shell—it's that easy!

http://docs.mongodb.org/manual/tutorial/getting-started/

CRUD: Using the Shell

To check which db you're using	db
Show all databases	show dbs
Switch db's/make a new one	use <name>
See what collections exist	show collections

Note: db's are not actually created until you insert data!

CRUD: Using the Shell (cont.)

To insert documents into a collection/make a new collection:

db.<collection>.insert(<document>)
<=>
INSERT INTO <table>
VALUES(<attributevalues>);

CRUD: Inserting Data

Insert one document

```
db.<collection>.insert({<field>:<value>})
```

Inserting a document with a field name new to the collection is inherently supported by the BSON model.

To insert multiple documents, use an array.

MongoDB – Bulk.insert() Method

MongoDB, the Bulk.insert() method is used to perform insert operations in bulk.

The Bulk.insert() method is used to insert multiple documents in one go.

Unordered Insertion of documents:

```
var bulk = db.students.initializeUnorderedBulkOp();
bulk.insert( { first_name: "Sachin", last_name: "Tendulkar" } );
bulk.insert( { first_name: "Virender", last_name: "Sehwag" } );
bulk.insert( { first_name: "Shikhar", last_name: "Dhawan" } );
bulk.insert( { first_name: "Mohammed", last_name: "Shami" } );
bulk.insert( { first_name: "Shreyas", last_name: "Iyer" } );
bulk.execute();
```

db.students.find().sort({'_id':-1}).limit(5).pretty()

Schema Validation in MongoDB

- **Schema validation** in **MongoDB** provides a structured approach to define and enforce rules for document structures within collections.
- By specifying validation criteria such as data types, required fields, and custom expressions using **JSON** Schema syntax, MongoDB ensures data integrity and consistency.
- **Mongodb schema validation**
- **Schema validation** in **MongoDB** is a feature that allows us to set the structure for the data in the documents of a collection.
- We follow some set of rules, and **validation rules**, which ensure that the data we insert or update follows a **specific predefined schema** and ensures the data must have only specific datatypes, required fields, and validation expressions mentioned in the predefined schema.
- When we create a collection for the first time and we want it to meet specific criteria then we can define the collection with the schema validation rules.
- These validation rules can include specifying the required fields we want, and the datatype for those fields, and also allow the user's custom expressions. We use the command **$jsonSchema** for specifying the rules.

When to use Schema Validation

- **Schema Validation** is like setting rules for **how your document must look in our database.**
- When we are experimenting on a new application in which we think that the incoming data might change the application's fields and we are unsure about the structure we might not want to use schema validation.

- **Step 1**: We create a collection named of 'students' using the createCollection() command.

- Step 2: With the **'$jsonShema'** command inside the validator we specify the schema validation rules.
- Here with the required property we give a list of fields that every document must have when inserted into the collection.

- **Step 3**: Give all the fields and their datatypes inside the properties.

```
db.createCollection("Students", {
    validator: {
        $jsonSchema: {
            bsonType: "object",
            required: ["name", "id"],
            properties: {
                name: {
                    bsonType: "string",
                    description: "Name must be a string."
                },
                id: {
                       bsonType: "int",
                    description: "id must be an integer."
                },

            }
        }
    }
});
show collections;
db.Students.insert({name:"s",id:1})
db.Students.find()
```

CRUD: Querying

- Done on collections.
- Get all docs: db.<collection>.find()
  - Returns a cursor, which is iterated over shell to display first 20 results.
  - Add .limit(<number>) to limit results
  - SELECT * FROM <table>;

- Get one doc: db.<collection>.findOne()

CRUD: Querying

To match a specific value:

db.<collection>.find({<field>:<value>})

“AND”

db.<collection>.find({<field1>:<value1>, <field2>:<value2>
})

SELECT *

FROM <table>

WHERE <field1> = <value1> AND <field2> = <value2>;

CRUD: Querying

OR
db.<collection>.find({ $or: [<field>:<value1><field>:<value2>]})

SELECT *
FROM <table>
WHERE <field> = <value1> OR <field> = <value2>;

Checking for multiple values of same field

db.<collection>.find({<field>: {$in [<value>, <value>]}})

CRUD: Querying

Including/excluding document fields

db.<collection>.find({<field1>:<value>}, {<field2>: 0})

SELECT field1
FROM <table>;

db.<collection>.find({<field>:<value>}, {<field2>: 1})

Find documents with or w/o field

db.<collection>.find({<field>: { $exists: true}})

Find Query Options

•The sort() method orders the documents in the result set.

• The following operation returns documents in the collection sorted in ascending order by the name field:

•sort() corresponds to the ORDER BY statement in SQL.

```
db.bios.find().sort( { name: 1 } )
```

- Limit the Number of Documents to Return
- The limit() method limits the number of documents in the result set. The following operation returns at most 5 documents in the collection:
- db.bios.find().limit(5)
- limit() corresponds to the LIMIT statement in SQL.

- The skip() method controls the starting point of the results set. The following operation skips the first 5 documents in the bios collection and returns all remaining documents:
- db.bios.find().skip(5)

- **Combine Methods**
- The following statements chain cursor methods limit() and sort():
- db.bios.find().sort({ name: 1 }).limit(5)
- db.bios.find().limit(5).sort({ name: 1 })

- >db.mycol.find({ $and: [{<key1>:<value1>}, { <key2>:<value2>}] })

NOT in MongoDB

Syntax
To query documents based on the NOT condition, you need to use $not keyword following is the basic syntax of **NOT −**

```
>db.COLLECTION_NAME.find(
        {
                $NOT: [
                        {key1: value1}, {key2:value2}
                ]
        }
).pretty()
```

MongoDB – Comparison Query Operators

- Comparison Query Operators in MongoDB are used to filter documents based on some specific criteria within their fields.
- MongoDB uses various comparison query operators to compare the values of the documents. The following table contains the comparison query operators:

Operators	Description
$eq	Matches the values of the fields that are equal to a specified value.
$ne	Matches all values of the field that are not equal to a specified value.
$gt	Matches values of the fields that are greater than a specified value.
$gte	Matches values of the fields that are greater than equal to the specified value.
$lt	Matches values of the fields that are less than a specified value
$lte	Matches values of the fields that are less than equal to the specified value
$in	Matches any of the values specified in an array.
$nin	Matches none of the values specified in an array.

Matching values using $or operator:

- In this example, we are retrieving only those employee's documents whose branch is ECE or joiningYear is 2017.

- db.contributor.find({$or: [{branch: "ECE"}, {joiningYear: 2017}]}).pretty()

Example 1: Query for “not null” in Specific Field

- Consider a MongoDB collection named “students” containing documents representing student profiles.
- Each document includes a field named “name” that stores the student’s name.
- To retrieve documents where the “name” field is not null, we can use the $ne operator as follows

- db.students.find({ name: { $ne: null } });

Example 2

- Filtering Results Based on Non-Null Values in the “name” and “age” Fields
- In some cases, we may need to filter results based on multiple fields, with each field having non-null values. Let’s reuse the previous example by considering a scenario where we also want to filter students based on their “age.” We can combine multiple $ne operators to achieve this.

- db.students.find({ name: { $ne: null }, age: { $ne: null } });

Query an Array in MongoDB

- MongoDB's flexible structure makes it a popular choice for storing and managing diverse data.
- In MongoDB, data is stored in collections and collections have documents that support data types like strings, numbers, objects, and most important arrays.
- **Arrays in MongoDB allow the users to store data in an ordered form.**
- Efficiently querying array elements is crucial for developers to extract meaningful information from the databases.

Different Methods in MongoDB to Query Array Elements

Method	Definition	Syntax
Query using dot notation	dot notation to access an element by its index in the array.	db.collection.find({“arrayName.index”: “value”})
Query using $elemMatch	$elemMatch operator matches documents that contain an array with at least one element that matches the specified query criteria.	db.collection.find({ <arrayField>: {$elemMatch: {<query>}})
Query using $slice	The $slice is a projection operator in MongoDB that limits the number of elements from an array to return in the results.	db.collection.find({}, {arrayName: { $slice: ! }})
Unwinding	Unwinding allows users in MongoDB to output a document for each element in the array. This makes it easier for the developers to run aggregation queries on the array data.	db.collection.aggregate([{$unwind: “$arrayName”}])

Creating an Array in MongoDB

- Insert a Document with an Array FieldTo insert a document that contains an array, you can use the **insertOne or insertMany** methods. Here's an example using insertOne:

- db.hobby.insertOne({ name: "Jiya", hobbies: ["reading", "traveling", "swimming"]})

- db.hobby.find()
- Insert Multiple Documents with Arrays
- You can insert multiple documents into the hobby collection using insertMany:

- db.hobby.insertMany([{ name: “Jiya", hobbies: ["cooking", "dancing"] }, { name: "Amit", hobbies: ["cycling", "hiking", "photography"] }])

Query for Documents with an Array Containing All Specified Elements

- find documents where the hobbies array contains all specified values, use the $all operator:
- **db.hobby.find({ hobbies: { $all: ["reading", "swimming"] }})**

- Query for Documents with an Array Containing at Least One of the Specified Elements:
- To find documents where the hobbies array contains at least one of the specified values, **use the $in operator:**

- **db.hobby.find({ hobbies: { $in: ["dancing", "photography"] } })**

- **Query for the Length of an Array**
- To query for documents based on the length of the hobbies array, use the $size operator:

- db.hobby.find({ hobbies: { $size: 3 } })

MongoDB Save() Method

- The **save()** method replaces the existing document with the new document passed in the save() method.
- Syntax
- The basic syntax of MongoDB **save()** method is shown below −
- >db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})

MongoDB findOneAndUpdate() method

- The **findOneAndUpdate()** method updates the values in the existing document.
- Syntax
- The basic syntax of **findOneAndUpdate()** method is as follows −
- >db.COLLECTION_NAME.findOneAndUpdate(SELECTIOIN_CRITERIA, UPDATED_DATA)

Update All Documents

- In MongoDB, you can update all the documents in the collection using db.collection.updateMany() method.

- Syntax: db.collection.updateMany({}, {update})

- **Updating all documents:**
- In this example, we are updating all the documents of employee collection. Or in other words, we are updating the salary of all the employees.

- db.employee.updateMany({}, {$set: {salary: 50000}})

CRUD: Updating

db.<collection>.update(
{<field1>:<value1>}, //all docs in which field = value
{$set: {<field2>:<value2>}}, //set field to value
{multi:true}) //update multiple docs

upsert: if true, creates a new doc when none matches search criteria.

UPDATE <table>
SET <field2> = <value2>
WHERE <field1> = <value1>;

CRUD: Updating

Updating first document:

In this example, we are updating the value of department field in the first document of the employee collection using db.collection.updateOne() method.

Or

in other words, we update the department of riya. So, before updating the department of riya is “Development” and now, after updating the department of riya is “HR”.

```
db.employee.update({}, { $set: {department: "HR"}})
```

Updating single document that matches the filter:

```
db.employee.updateOne({branch: "ECE"}, {$set: {department: "Development"}})
```

In this example, we are updating a single document in the employee collection that match the given filter, i.e., branch: “ECE”.

In this example, we are updating a document in the employee collection.
Here multiple documents **matches the given filter, so this method will update the first document** among these documents like as shown in the below

```
db.employee.updateOne({department: "HR"}, {$set: {salary: 30000}})
```

CRUD: Updating

To remove a field

db.<collection>.update({<field>:<value>},
{ $unset: { <field>: 1}})

Replace all field-value pairs

db.<collection>.update({<field>:<value>}, { <field>:<value>,
<field>:<value>})

*NOTE: This overwrites ALL the contents of a document, even removing fields.

CRUD: Upsert in MongoDB

- In MongoDB, upsert is an option that is used for update operation e.g. update(), findAndModify(), etc.
- In upsert is a combination of update and insert (update + insert = upsert). If the value of this option is set to true and the document or documents found that match the specified query, **then the update operation will update the matched document or documents.**
- if the value of this option is set to true **and no document or documents matches** the specified document, then **this option inserts a new document** in the collection and this new document have the fields that indicate in the operation.
- By default, the value of the upsert option is false. If the value of upsert in a sharded collection is true then you have to include the full shard key in the filter.

Upsert with findAndModify() method:

- If a document or documents found that matches the given query criteria, then the findAndModify() method updates the document/documents.
- If no document/documents match the given query criteria, then the findAndModify() method inserts a new document in the collection.

- db.employee.findAndModify({query:{name:"Ram"},
  update:{$set:{department:"Development"}},
  upsert:true})

Remove All Documents from a Collection

- To remove all documents in a collection, call the remove method with an empty query document {}.
- The following operation deletes all documents from the bios collection:

- db.student.remove({ })

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

- The following operation removes all the documents from the collection products where qty is greater than 20:

- db.products.remove({ qty: { $gt: 20 } })

CRUD: Removal

Remove all records where field = value

- db.<collection>.remove({<field>:<value>})
- DELETE FROM <table> WHERE <field> = <value>;

As above, but only remove first document

- db.<collection>.remove({<field>:<value>}, true)

CRUD: Isolation

- By default, all writes are atomic **only** on the level of a single document.
- This means that, by default, all writes can be interleaved with other operations.
- You can isolate writes on an **unsharded** collection by adding $isolated:1 in the query area:

db.<collection>.remove({<field>:<value>, $isolated: 1})

Thanks