Introduction to Mongo DB

pm_jat @ daiict

Document Databases - Recap

- Based on Key-Value strategy, where Value is a Document
- A collection of documents is referred as “Collection”
- A document database is collection of “document **collections**”
- Here is RDB and Document DB correspondences

RDB	Doc DB
Database/Schema	Database
Table	“SET” of similar Documents
Row/Tuple	“Document”
Row ID	_id

Mongo DB "Data Models"

- Document as a basic unit
- "Collection" – collection of related documents
- We can have a collection level "schema"
- MongoDB database design uses following strategies
  - Collections with "embedded documents" – less normalized
  - Separated document Collections – more normalized
  - Typically mongo DB uses combined strategy.
- So how does mongo db look like?
  - **"MongoDB Database is collection of document collections"** vis-à-vis "relational database is collection of tables"

Why do we have “Schema” for databases?

- Schema information helps DBMS in **ensuring Database Integrity Constraints**, that is
  - data are valid in terms of types, cardinality, and other existential constraints like not null, unique, referential integrity etc.
- Schema information acts as assertion that can be used for
  - Efficient storage and access paths (File organization and indexes).
  - Query Optimization and efficient execution of queries

Motivation for Schema-less-ness/Flexible Schema

- Why Schema-less/Flexible Schema is becoming important?
- In Big Data era: figuring out schema in advance is becoming impractical
- More and more column may get added with the time
- Some columns may be becoming irrelevant with the time
- Creating databases in “fixed schema” approach have problems of
  - Too many columns, and every row having values only for few of them.
  - Database becomes sparse with lots of null values

Problems with schema-less-ness

- Totally schema-less can be disastrous?
  - Two data objects having two names quantity attribute - qty for quantity; how do we reconcile this?
  - “3” and “three” as value for an attributes
- DBMS can not do any kind of validation for such anomalies in absence of schema?
- So, we require having a tradeoff between strict schema and schema less.
- Sometime schema is “**Implicit**” only. That is DBMS does not know anything about schema. Where as “application programmer” can assume some schema and access accordingly.
- Again, DBMS can not perform any kind of validation for implicit schema.
- Also DBMS performance of database operation gets affected in absence of schema.

MongoDB and Schema-less-ness

- MongoDB's collections, by default, does not require its documents to have the same schema, i.e. documents in a collection
  - Field names can different
  - data type for a field can differ across documents within a collection
- This means each document in a collection can have its own schema?
- However we can specify some partial schema at collection level? i.e. required fields for every document, their "global data type", and so on
- "Schema of a collection" (or document) can be altered later also?

MongoDB and Schema

- We can define certain (Schema) rules at Collection level.
- Mongo DB calls them “Document Validation Rules”.
  - This can be read as Rule that every document in a collection should comply.
- Document Validation rules can be specified at Collection Creation time or can be specified later.
- Mongo DB provides “**JSON Schema**” as the means of defining schema validation rules.
- Here is an JSON Schema example from Mongo Doc Pages

Source: https://docs.mongodb.com/manual/core/schema-validation/

Example: JSON Schema

```
db.createCollection("students", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         required: [ "name", "year", "major", "address" ],
         properties: {
            name: {
               bsonType: "string",
               description: "must be a string and is required"
            },
            year: {
               bsonType: "int",
               minimum: 2017,
               maximum: 3017,
               description: "must be an integer in [ 2017, 3017 ] and is required"
            },
```

Source: https://docs.mongodb.com/manual/core/schema-validation/

```
major: {
   enum: [ "Math", "English", "Computer Science", "History", null ],
   description: "can only be one of the enum values and is required"
},
gpa: {
   bsonType: [ "double" ],
   description: "must be a double if the field exists"
},
address: {
   bsonType: "object",
   required: [ "city" ],
   properties: {
      street: {
         bsonType: "string",
         description: "must be a string if the field exists"
      },
      city: {
         bsonType: "string",
         "description": "must be a string and is required"
```

Create Collection

```
db.createCollection( <name>,
   { //options
     capped: <boolean>,
     autoIndexId: <boolean>,
     size: <number>,
     max: <number>,
     storageEngine: <document>,
     validator: <document>,
     validationLevel: <string>,
     validationAction: <string>,
     indexOptionDefaults: <document>,
     viewOn: <string>,                    /
     pipeline: <pipeline>,                /
     collation: <document>,               /
     writeConcern: <document>
   }
```

Three options related to Schema Validations:

- validator
- validationLevel
- validationAction

JSON Schema validation options

- `validator` – Used to specify validation rules for the collection. Rules are specified as JSON document containing rules in the form of mongo db query expressions.
- `validationLevel` – specifies "how strictly" MongoDB applies validation rules to existing documents during an update, and
- `validationAction` - specifies whether MongoDB should error and **reject documents** that violate the validation rules or warn about the **violations in the log** but allow invalid documents.

Source: https://docs.mongodb.com/manual/core/schema-validation/

Validation Level option

- `validationLevel` option values: `off`, `strict`, `moderate`
- It determines on what operations MongoDB applies the validation rules
- If the validationLevel is `strict` (the default), MongoDB applies validation rules to all inserts and updates.
- If the validationLevel is `moderate`, MongoDB applies validation rules to inserts and to updates to existing documents that already fulfill the validation criteria.
- With the moderate level, updates to existing documents that do not fulfill the validation criteria are not checked for validity.

Source: https://docs.mongodb.com/manual/core/schema-validation/

Validation Action options

- `validationAction` option values: `error`, `warn`
- It determines how MongoDB handles documents that violate the validation rules:
  - If the validationAction is `error` (the default), MongoDB rejects any insert or update that violates the validation criteria.
  - If the validationAction is `warn`, MongoDB logs any violations but allows the insertion or update to proceed.

Source: https://docs.mongodb.com/manual/core/schema-validation/

_id field of Mongo DB documents

- In MongoDB, each document stored in a collection requires a unique _id field that acts as a primary key.
- If an inserted document omits the _id field, the MongoDB driver automatically generates an ObjectId for the _id field.
- The _id field has the following behavior and constraints:
  - By default, MongoDB creates a unique index on the _id field during the creation of a collection.
  - The _id field is always the first field in the documents.
  - The _id field may contain values of any BSON data type, other than an array.

BSON data types

- BSON is a binary serialization format used to store documents and make remote procedure calls in MongoDB.
- The BSON specification is available at http://bsonspec.org/
- These are the data types that can be used for specifying schema and field values

Mongo DB

- You may not require creating a collection at all.

Creating Mongo DB Collection

- MongoDB may not require explicit creation of a empty collection.
- First use makes one (implicit creation); for example following will create one: `booksDB.books.insertOne(…)`
- Or can be explicitly created using `booksDB.createCollection(..)`

Mongo DB Operations

- Mongo DB is very compressive No SQL systems, and it provides very rich set of “Transactional” and “Analytical operations”.
- Typically what operations, a database should have
  - CRUD
  - Transactional Support for concurrent access – ACID?
  - More here: dealing with “shards” and “replicas”

Mongo DB Operations

- Mongo DB is very compressive No SQL systems, and it provides very rich set of “Transactional” and “Analytical operations”.
- Write operations
  - INSERT, UDATE, DELETE
  - All write operations in MongoDB are “**atomic**” on the level of a single document.
- Read operation
  - Find
  - JOIN
  - Aggregate and Aggregation Pipeline (we can create customizing sequence of analytical tasks)
  - map-reduce (can also specify map and reduce codes for customized processing) however aggregation pipeline s supposed to be doing more efficient job)

Read/Write Concerns

- Read/Write Concerns are specified with every read write operations
  - Individually for each operation or Setting for a “Transaction”
- This is the mechanism of setting “Consistency Level”
- For write operations, we typically specify do following in r/w concerns-
  - For **write**, we specify, **how many replicas** a write should be performed before returning success
  - For **read**, we specify, **which replica** do we read from?

INSERT operation

- Provide following commands
  - db.collection.insertOne //one
  - db.collection.insertMany //many
  - db.collection.insert //one or many
- This **inserts specified documents if not already there in the collection**
- It also creates the collection, if collection not already there
- Value for **_id field is automatically supplied** if not specified
- All write operations in MongoDB are atomic on the level of a single document.
- We can also specify the "**write concerns**", i.e. the level of acknowledgement?

INSERT Examples

```
db.inventory.insertOne(
   { item: "canvas", qty: 100, tags: ["cotton"], size: { h: 28, w: 35.5, uom: "cm" } }
)
```

```
db.inventory.insertMany([
   { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
   { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "A" },
   { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
   { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
   { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" }
]);
```

Source: https://docs.mongodb.com/manual/tutorial/insert-documents/

Insert – more methods

- The following methods can also add new documents to a collection:

- `db.collection.update()` when used with the **`upsert: true`** option.
- `db.collection.updateOne()` when used with the **`upsert: true`** option.
- `db.collection.updateMany()` when used with the **`upsert: true`** option.
- `db.collection.findAndModify()` when used with the **`upsert: true`** option.
- `db.collection.findOneAndUpdate()` when used with the **`upsert: true`** option.
- `db.collection.findOneAndReplace()` when used with the **`upsert: true`** option.
- `db.collection.save()`.
- `db.collection.bulkWrite()`.

Query Documents - FIND

- Mongo FIND is very comprehensive like **SQL SELECT**
- It is used as: "`db.collection-name.find(...)`"
- Wee specify query selection criteria as parameter to find in the form of "json document"
- To get all documents we leave criteria document as empty. Example below:

```
db.inventory.find( {} )     | SELECT * FROM inventory
```

Source: https://docs.mongodb.com/manual/tutorial/query-documents/

FIND: specify equality check

- Specified as attribute value pairs. Examples below

```
db.inventory.find( { status: "D" } )
```

```
SELECT * FROM inventory WHERE status = "D"
```

- We can AND by simply using comma, as following:
  `db.inventory.find( {status: "A", category: 5 })`
- For OR, we require little extra work

```
db.inventory.find(
    {$or:[{status:"A"}, {status:"D"}]}
)
```

Source: https://docs.mongodb.com/manual/tutorial/query-documents/

FIND: more examples

- More conditional operators

```
db.inventory.find( { status: "A", qty: { $lt: 30 } } )
```

```
SELECT * FROM inventory WHERE status = "A" AND qty < 30
```

```
db.inventory.find( { $or: [ { status: "A" }, { qty: { $lt: 30 } } ] } )
```

```
SELECT * FROM inventory WHERE status = "A" OR qty < 30
```

Source: https://docs.mongodb.com/manual/tutorial/query-documents/

FIND: more examples

- More conditional operators

```
db.inventory.find( { status: { $in: [ "A", "D" ] } } )
```

```
SELECT * FROM inventory WHERE status in ("A", "D")
```

```
db.inventory.find( {
     status: "A",
     $or: [ { qty: { $lt: 30 } }, { item: /^p/ } ]  //regular expression
} )
```

```
SELECT * FROM inventory WHERE status = "A" AND ( qty < 30 OR item LIKE "p%")
```

Source: https://docs.mongodb.com/manual/tutorial/query-documents/

Embedded/Nested Documents fields in FIND

- Example:

```
db.inventory.find(
    { "size.h": { $lt: 15 }, "size.uom": "in",
        status: "D" }
)
```

- Predicate expressed here is:
  size.h < 15 AND size.uom = “IN” AND status=“D”

Array Field in FIND

- Consider following collection with Array Field tags, dim_cm

```
db.inventory.insertMany([
   { item: "journal", qty: 25, tags: ["blank", "red"], dim_cm: [ 14, 21 ] },
   { item: "notebook", qty: 50, tags: ["red", "blank"], dim_cm: [ 14, 21 ] },
   { item: "paper", qty: 100, tags: ["red", "blank", "plain"], dim_cm: [ 14, 21 ] },
   { item: "planner", qty: 75, tags: ["blank", "red"], dim_cm: [ 22.85, 30 ] },
   { item: "postcard", qty: 45, tags: ["blue"], dim_cm: [ 10, 15.25 ] }
]);
```

- Query “db.inventory.find({ tags: ["red", "blank"] })” shall look for document that have tags as exactly this two element array, i.e. ["red", "blank"]

Array Field in FIND

- Consider following collection with Array Field tags, dim_cm

```
db.inventory.insertMany([
   { item: "journal", qty: 25, tags: ["blank", "red"], dim_cm: [ 14, 21 ] },
   { item: "notebook", qty: 50, tags: ["red", "blank"], dim_cm: [ 14, 21 ] },
   { item: "paper", qty: 100, tags: ["red", "blank", "plain"], dim_cm: [ 14, 21 ] },
   { item: "planner", qty: 75, tags: ["blank", "red"], dim_cm: [ 22.85, 30 ] },
   { item: "postcard", qty: 45, tags: ["blue"], dim_cm: [ 10, 15.25 ] }
]);
```

- Query "db.inventory.find({ tags: { $all: ["red", "blank"] } })" shall look for document that have this array (i.e. ["red", "blank"]) as subset of elements in tags attributes.

Array Field in FIND

```
db.inventory.insertMany([
   { item: "journal", qty: 25, tags: ["blank", "red"], dim_cm: [ 14, 21 ] },
   { item: "notebook", qty: 50, tags: ["red", "blank"], dim_cm: [ 14, 21 ] },
   { item: "paper", qty: 100, tags: ["red", "blank", "plain"], dim_cm: [ 14, 21 ] },
   { item: "planner", qty: 75, tags: ["blank", "red"], dim_cm: [ 22.85, 30 ] },
   { item: "postcard", qty: 45, tags: ["blue"], dim_cm: [ 10, 15.25 ] }
]);
```

- Query “db.inventory.find({ tags: "red" })” returns the documents that have “red” as an element in tags attribute
- Query “db.inventory.find({ dim_cm: { $gt: 25 } })” returns the documents that have at-least one element > 25 in dim_cm array attribute.

Find returns “Cursor”

- And you can iterate through as following.
  Note: selection criteria here is “type=2”

```
var myCursor = db.users.find( { type: 2 } );

while (myCursor.hasNext()) {
   print(tojson(myCursor.next()));
}
```

Find: Null or Missing Fields

- If attribute (field) item has null value

```
db.inventory.find( { item: null } )
```

- If item type (BSON) is 10 (i.e. Null)

```
db.inventory.find( { item : { $type: 10 } } )
```

- If attribute item is non existent in a document

```
db.inventory.find( { item : { $exists: false } } )
```

"read-concern" with FIND

- We can also specify "**read concern**" for **choosing isolation level** while reading from replica sets and replica set shards!
- Typically specifies how many nodes, read should consult before returning a value because, replicate sets can be in middle of update, and all replicas are not consistent?
- We shall talk later about this.

Projection (specifying fields)

- By default all fields are projected
- We can either be field inclusive mode, or exclusive mode
- Id:0, name:1, qty: 1, will show name field only (exclude _id, and include name)
- salary:0, email:0; will exclude these two fields (rest will be projected)
- 0 and 1 can not be mixed in a project expression except with _id field

UPDATE operation

- Either we use db.collection.update(<filter>, <update>, <options>)
- Or one of following variations

  db.collection.updateOne(<filter>, <update>, <options>)

  db.collection.updateMany(<filter>, <update>, <options>)

  db.collection.replaceOne(<filter>, <update>, <options>)
- The method uses operators such as **$set** and **$unset** for specifying the updates
- updateOne() finds the first document that matches the filter and applies the specified update modifications.
- By default update() also updates single document. Has various options to do the same done by all other variations.

Source: https://docs.mongodb.com/manual/tutorial/update-documents/

Update Example

```
db.books.update(
    { _id: 1 },
    {
      $inc: { stock: 5 },
      $set: {
        item: "ABC123",
        "info.publisher": "2222",
        tags: [ "software" ],
        "ratings.1": { by: "xyz", rating: 3 }
      }
    }
)
```

```
UPDATE books
SET    stock = stock + 5
       item = "ABC123"
       publisher = 2222
       pages = 430
       tags = "software"
       rating_authors = "ijk,xyz"
       rating_values = "4,3"
WHERE  _id = 1
```

Source: https://docs.mongodb.com/manual/reference/method/db.collection.update

option with Update operation

- `multi`: boolean. by default updates one.
- `upsert`: boolean. Value true creates a new document when no document matches the query criteria.
- `writeConcern`: A document expressing the write concern. Default write concern "`w: 1`".
- `hint`: a document or string that specifies the index to use to support the query predicate.

Update – replace entire document

```
db.books.update(
   { _id: 2 },
   {
     item: "XYZ123",
     stock: 10,
     info: { publisher: "2255", pages: 150 },
     tags: [ "baking", "cooking" ]
   }
)
```

```
DELETE from books WHERE _id = 2
INSERT INTO books
            (_id,
             item,
             stock,
             publisher,
             pages,
             tags)
VALUES      (2,
             "xyz123",
             10,
             "2255",
             150,
             "baking,cooking")
```

Source: https://docs.mongodb.com/manual/reference/method/db.collection.update

Update Example: document before and after update

```
"_id" : 1,
"item" : "TBD",
"stock" : 0,
"info" : { "publisher" : "1111", "pages" : 430 },
"tags" : [ "technology", "computer" ],
"ratings" : [ { "by" : "ijk", "rating" : 4 }, { "by" : "lmn", "rating" : 5 } ]
"reorder" : false
},
```

```
"_id" : 1,
"item" : "ABC123",
"stock" : 5,
"info" : { "publisher" : "2222", "pages" : 430 },
"tags" : [ "software" ],
"ratings" : [ { "by" : "ijk", "rating" : 4 }, { "by" : "xyz", "rating" : 3 } ],
"reorder" : false
```

Source: https://docs.mongodb.com/manual/reference/method/db.collection.update

Update Example with “upsert”

```
db.books.update(
    { item: "ZZZ135" },    // Query parameter
    {                      // Replacement document
      item: "ZZZ135",
      stock: 5,
      tags: [ "database" ]
    },
    { upsert: true }       // Options
)
```

Source: https://docs.mongodb.com/manual/reference/method/db.collection.update

Use of Index while updating!

- `hint` option specifies the index (available since ver 4.2)

```
db.members.update(
    { points: { $lte: 20 }, status: "P" },      // Query parameter
    { $set: { misc1: "Need to activate" } },    // Update document
    { multi: true, hint: { status: 1 } }        // Options
)
```

- Creating Index

```
db.members.createIndex( { status: 1 } )
db.members.createIndex( { points: 1 } )
```

Source: https://docs.mongodb.com/manual/reference/method/db.collection.update

Update: more examples

```
db.inventory.updateOne(
   { item: "paper" },
   {
     $set: { "size.uom": "cm", status: "P" },
     $currentDate: { lastModified: true }
   }
)
```

```
db.inventory.updateMany(
   { "qty": { $lt: 50 } },
   {
     $set: { "size.uom": "in", status: "P" },
     $currentDate: { lastModified: true }
   }
)
```

```
db.inventory.replaceOne(
   { item: "paper" },
   { item: "paper", instock: [ { warehouse: "A", qty: 60 }, { warehouse: "B", qty: 40 } ]
)
```

Delete Documents

- Provide following methods

  db.collection.deleteMany()

  db.collection.deleteOne()

- Examples:

```
db.inventory.deleteMany({}) //delete all
db.inventory.deleteMany({ status : "A" })
                    //delete all with status=“A”
db.inventory.deleteOne( { status: "D" } )
                    //delete one with status=“A”
```

References

[1] Chapter 9, NoSQL Distilled

[2] https://docs.mongodb.com/manual/