**Main data modelling principle:**

What you use together, store them together!

**Data Modelling methods in MongoDB**

1. Embedding: Storing additional information as a nested object / list of nested objects in the same document
2. Referencing / Linking / Data Normalization: Saves the id field of one document into another document.

|  |  |
| --- | --- |
| **Embedding** | **Referencing** |
| + easy access / modification of the nested data in just 1 query | + smaller documents, avoid duplication of data |
| - possible to exceed the size limit of a single document (16mb)  - slower query? Need to load the entire document into memory | - needs to query data from multiple documents, more joins between documents |

Generally, embedding is an easier solution at the start, referencing should be used for scaling up once the amount of data increases. Because we want to create new collections.

**Inserting documents**

|  |  |
| --- | --- |
| **insertOne** | **insertMany** |
| **db.grades.insertOne({**  **student\_id: 654321,**  **products: [**  **{**  **type: "exam",**  **score: 90,**  **},**  **{**  **type: "homework",**  **score: 59,**  **},**  **{**  **type: "quiz",**  **score: 75,**  **},**  **{**  **type: "homework",**  **score: 88,**  **},**  **],**  **class\_id: 550,**  **})** | **db.grades.insertMany([**  **{**  **student\_id: 546789,**  **products: [**  **{**  **type: "quiz",**  **score: 50,**  **},**  **{**  **type: "homework",**  **score: 70,**  **},**  **{**  **type: "quiz",**  **score: 66,**  **},**  **{**  **type: "exam",**  **score: 70,**  **},**  **],**  **class\_id: 551,**  **},**  **{**  **student\_id: 777777,**  **products: [**  **{**  **type: "exam",**  **score: 83,**  **},**  **{**  **type: "quiz",**  **score: 59,**  **},**  **{**  **type: "quiz",**  **score: 72,**  **},**  **{**  **type: "quiz",**  **score: 67,**  **},**  **],**  **class\_id: 550,**  **},**  **{**  **student\_id: 223344,**  **products: [**  **{**  **type: "exam",**  **score: 45,**  **},**  **{**  **type: "homework",**  **score: 39,**  **},**  **{**  **type: "quiz",**  **score: 40,**  **},**  **{**  **type: "homework",**  **score: 88,**  **},**  **],**  **class\_id: 551,**  **},**  **])** |

**Finding documents**

db.<collection>.find()

db.<collection>.find({name: “john”})

db.<collection>.find({name: {$in: [“john”, “sam”]} } )

**Query in Array Elements**

1. To find an item in the product array with matching field:

db.<collection>.find({

<documents>: {

$elemMatch: { amount: { $lte: 4500 }, transaction\_code: "sell" },

},

})

1. To find product array / product item with the same match:

db.<collection>.find({products: “InvestmentStock” } } )

**Query using $AND / $OR operators**

**Find a Document by Using Implicit $and**

Use implicit $and to select documents that match multiple expressions. For example:

db.routes.find({ "airline.name": "Southwest Airlines", stops: { $gte: 1 } })

**Find a Document by Using the $or Operator**

Use the $or operator to select documents that match at least one of the included expressions. For example:

db.routes.find({

$or: [{ dst\_airport: "SEA" }, { src\_airport: "SEA" }],

})

**Find a Document by Using the $and Operator**

Use the $and operator to use multiple $or expressions in your query.

db.routes.find({

$and: [

{ $or: [{ dst\_airport: "SEA" }, { src\_airport: "SEA" }] },

{ $or: [{ "airline.name": "American Airlines" }, { airplane: 320 }] },

]

})

**Updating MongoDB Documents by Using updateOne()**

The updateOne() method accepts a filter document, an update document, and an optional options object. MongoDB provides update operators and options to help you update documents. In this section, we'll cover three of them: $set, upsert, and $push.

**$set**

**The $set operator replaces the value of a field with the specified value, as shown in the following code:**

db.podcasts.updateOne(

{

\_id: ObjectId("5e8f8f8f8f8f8f8f8f8f8f8"),

},

{

$set: {

subscribers: 98562,

},

}

)

**upsert**

**The upsert option creates a new document if no documents match the filtered criteria.** Here's an example:

db.podcasts.updateOne(

{ title: "The Developer Hub" },

{ $set: { topics: ["databases", "MongoDB"] } },

{ upsert: true }

)

**$push**

**The $push operator adds a new value to the hosts array field. Here's an example:**

db.podcasts.updateOne(

{ \_id: ObjectId("5e8f8f8f8f8f8f8f8f8f8f8") },

{ $push: { hosts: "Nic Raboy" } }

)

**The findAndModify() method is used to find and replace a single document in MongoDB. It accepts a filter document, a replacement document, and an optional options object. The following code shows an example:**

db.podcasts.findAndModify({

query: { \_id: ObjectId("6261a92dfee1ff300dc80bf1") },

update: { $inc: { subscribers: 1 } },

new: true,

})

**updateMany()**

To update multiple documents, use the updateMany() method. This method accepts a filter document, an update document, and an optional options object. The following code shows an example:

db.books.updateMany(

{ publishedDate: { $lt: new Date("2019-01-01") } },

{ $set: { status: "LEGACY" } }

)

**Delete One Document**

The following code shows an example of the deleteOne() method:

db.podcasts.deleteOne({ \_id: Objectid("6282c9862acb966e76bbf20a") })

**Delete Many Documents**

The following code shows an example of the deleteMany() method:

db.podcasts.deleteMany({category: “crime”})

**Sorting Results**

Use cursor.sort() to return query results in a specified order. Within the parentheses of sort(), include an object that specifies the field(s) to sort by and the order of the sort. Use 1 for ascending order, and -1 for descending order.

**Syntax:**

db.collection.find(<query>).sort(<sort>)

**Example:**

// Return data on all music companies, sorted alphabetically from A to Z.

db.companies.find({ category\_code: "music" }).sort({ name: 1 });

To ensure documents are returned in a consistent order, include a field that contains unique values in the sort. An easy way to do this is to include the \_id field in the sort. Here's an example:

// Return data on all music companies, sorted alphabetically from A to Z. Ensure consistent sort order

db.companies.find({ category\_code: "music" }).sort({ name: 1, \_id: 1 });

**Limiting Results**

Use cursor.limit() to specify the maximum number of documents the cursor will return. Within the parentheses of limit(), specify the maximum number of documents to return.

**Syntax:**

db.companies.find(<query>).limit(<number>)

**Example:**

// Return the three music companies with the highest number of employees. Ensure consistent sort order.

db.companies

.find({ category\_code: "music" })

.sort({ number\_of\_employees: -1, \_id: 1 })

.limit(3);

**Add a Projection Document**

To specify fields to include or exclude in the result set, add a projection document as the second parameter in the call to db.collection.find().

Syntax:

db.collection.find( <query>, <projection> )

**Include a Field**

To include a field, set its value to 1 in the projection document.

Syntax:

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

Example:

// Return all restaurant inspections - business name, result, and \_id fields only

db.inspections.find(

{ sector: "Restaurant - 818" },

{ business\_name: 1, result: 1 }

)

**Exclude a Field**

To exclude a field, set its value to 0 in the projection document.

Syntax:

db.collection.find(query, { <field> : 0, <field>: 0 })

Example:

// Return all inspections with result of "Pass" or "Warning" - exclude date and zip code

db.inspections.find(

{ result: { $in: ["Pass", "Warning"] } },

{ date: 0, "address.zip": 0 }

)

**While the \_id field is included by default, it can be suppressed by setting its value to 0 in any projection.**

// Return all restaurant inspections - business name and result fields only

db.inspections.find(

{ sector: "Restaurant - 818" },

{ business\_name: 1, result: 1, \_id: 0 }

)

**Count Documents**

Use db.collection.countDocuments() to count the number of documents that match a query. countDocuments() takes two parameters: a query document and an options document.

**Syntax:**

db.collection.countDocuments( <query>, <options> )

The query selects the documents to be counted.

**Examples:**

// Count number of docs in trip collection

db.trips.countDocuments({})

// Count number of trips over 120 minutes by subscribers

db.trips.countDocuments({ tripduration: { $gt: 120 }, usertype: "Subscriber" })

**Using Transactions: All of the operation has to succeed for them to all take effect**

1. Create variables used in the transaction.

// Collections

const accounts = client.db("bank").collection("accounts")

const transfers = client.db("bank").collection("transfers")

// Account information

let account\_id\_sender = "MDB574189300"

let account\_id\_receiver = "MDB343652528"

let transaction\_amount = 100

1. Start a new session.

const session = client.startSession()

1. Begin a transaction with the WithTransaction() method on the session.

const transactionResults = await session.withTransaction(async () => {

// Operations will go here

})

1. Update the balance field of the sender’s account by decrementing the transaction\_amount from the balance field.

const senderUpdate = await accounts.updateOne(

{ account\_id: account\_id\_sender },

{ $inc: { balance: -transaction\_amount } },

{ session }

)

1. Update the balance field of the receiver’s account by incrementing the transaction\_amount to the balance field.

const receiverUpdate = await accounts.updateOne(

{ account\_id: account\_id\_receiver },

{ $inc: { balance: transaction\_amount } },

{ session }

)

1. Create a transfer document and insert it into the transfers collection.

const transfer = {

transfer\_id: "TR21872187",

amount: 100,

from\_account: account\_id\_sender,

to\_account: account\_id\_receiver,

}

const insertTransferResults = await transfers.insertOne(transfer, { session })

1. Update the transfers\_complete array of the sender’s account by adding the transfer\_id to the array.

const updateSenderTransferResults = await accounts.updateOne(

{ account\_id: account\_id\_sender },

{ $push: { transfers\_complete: transfer.transfer\_id } },

{ session }

)

1. Update the transfers\_complete array of the receiver’s account by adding the transfer\_id to the array.

const updateReceiverTransferResults = await accounts.updateOne(

{ account\_id: account\_id\_receiver },

{ $push: { transfers\_complete: transfer.transfer\_id } },

{ session }

)

1. Log a message regarding the success or failure of the transaction.

if (transactionResults) {

console.log("Transaction completed successfully.")

} else {

console.log("Transaction failed.")

}

1. Catch any errors and close the session.

} catch (err) {

console.error(`Transaction aborted: ${err}`)

process.exit(1)

} finally {

await session.endSession()

await client.close()

}

**Aggregation Pipelines**

**$match**

The $match stage filters for documents that match specified conditions. Here's the code for $match:

{

$match: {

"field\_name": "value"

}

}

**$group**

The $group stage groups documents by a group key.

{

$group:

{

\_id: <expression>, // Group key

<field>: { <accumulator> : <expression> }

}

}

**$match and $group in an Aggregation Pipeline**

The following aggregation pipeline finds the documents with a field named "state" that matches a value "CA" and then groups those documents by the group key "$city" and shows the total number of zip codes in the state of California.

db.zips.aggregate([

{

$match: {

state: "CA"

}

},

{

$group: {

\_id: "$city",

totalZips: { $count : { } }

}

}

])

**$project**

**The $project stage specifies the fields of the output documents. 1 means that the field should be included, and 0 means that the field should be supressed. The field can also be assigned a new value.**

{

$project: {

state:1,

zip:1,

population:"$pop",

\_id:0

}

}

**$set**

**The $set stage creates new fields or changes the value of existing fields, and then outputs the documents with the new fields.**

{

$set: {

place: {

$concat:["$city",",","$state"]

},

pop:10000

}

}

**$count**

**The $count stage creates a new document, with the number of documents at that stage in the aggregation pipeline assigned to the specified field name.**

{

$count: "total\_zips"

}

**Aggregation for node.js**

const client = new MongoClient(uri)

const dbname = "bank";

const collection\_name = "accounts";

const accountsCollection = client.db(dbname).collection(collection\_name);

**Next, we build an aggregation pipeline that uses $match and $group and that will find accounts with a balance of less than $1,000. Then we group the results by the account\_type, and calculate the total\_balance and avg\_balance for each type.**

const pipeline = [

// Stage 1: match the accounts with a balance less than $1,000

{ $match: { balance: { $lt: 1000 } } },

// Stage 2: Calculate average balance and total balance

{

$group: {

\_id: "$account\_type",

total\_balance: { $sum: "$balance" },

avg\_balance: { $avg: "$balance" },

},

},

]

**To run an aggregation pipeline, we append the aggregate method to the collection. The aggregate method takes an array of stages as an argument, which is stored in a variable. The aggregate method returns a cursor that we can iterate over to get the results.**

const main = async () => {

try {

await client.connect()

console.log(`Connected to the database 🌍. \nFull connection string: ${safeURI}`)

let result = await accountsCollection.aggregate(pipeline)

for await (const doc of result) {

console.log(doc)

}

} catch (err) {

console.error(`Error connecting to the database: ${err}`)

} finally {

await client.close()

}

}

main()