



# DBS101 Database Systems Fundamentals



Royal University of Bhutan

## Lesson 15

## Learning Outcomes

1. Create indexes to support queries.
2. Understand query plans.
3. Implement aggregation on queries
4. Understand aggregation stages.
5. Execute transactions in mongodb.

## Indexes

Indexes are special data structures that store a small portion of the data to support ordered and efficient querying.

In document databases, indexes point to document identity.

Indexes in MongoDB support equality matches, range based and sorted results.

## Indexes

What happens in when you execute a `find()` query without creating a index in MongoDB?

## Indexes

What happens in when you execute a `findOne()` query without creating a index in MongoDB?

- Mongoddb reads all documents in the collection **in memory** to retrieve documents that fit the filter condition.

## Indexes

- “\_id” is a default index per collection.

Trade-offs:

- Indexes come at a write cost; they need to be updated when documents associated with them are updated.

So, redundant and unused indexes need to be deleted.

## Indexes

### Common Index Types:

1. Single field Index: Supports one field.
2. Compound Indexes: Supports more than one field.
  - Both Indexes can be multifield if they operate in an array field.

## Indexes: Single Field Index

Example:

Consider the following query:

**use analytics**

```
db.customers.find( {birthdate: { $lte: ISODate( '1973-01-01' ) } } )
```



## Indexes: Single Field Index

Example:

Index to support querying:

```
db.customers.createIndex( {birthdate:1} )
```

## Indexes: Single Field Index

Example:

Index to support unique values in a field:

```
db.customers.createIndex({email:1}, unique:true)
```

## Indexes: Single Field Index

Show all indexes created in a database

Syntax:

```
db.collections.getIndexes()
```

```
db.customers.getIndexes()
```

## Indexes: Single Field Index

`explain()` operation: Retrieves the query plan of a query.

```
db.customers.explain().find( {birthdate: { $lte: ISODate( '1973-01-01' ) } } )
```

## Indexes: Single Field Index

`explain()` operation: Retrieves the query plan of a query.

```
db.customers.explain().find( {birthdate: { $lte: ISODate( '1973-01-01' ) } } )
```

## Indexes: Single Field Index

In a collection when running a query to see the Execution plan, the '**Winning plan**' is listed. This plan provides the details of the execution stages (IXSCAN , COLLSCAN, FETCH, SORT, etc.).

- The **IXSCAN**: stage indicates the query is using an index and what index is being selected.
- The **COLLSCAN** stage indicates a collection scan is perform, not using any indexes.
- The **FETCH** stage indicates documents are being read from the collection.
- The **SORT** stage indicates documents are being sorted in memory.

## Indexes: Single Field Index

In the example:

**IXSCAN** is listed because bithdate\_1 index is used.

**FETCH** - Reads only the documents the index has specified.

Note: FETCH won't be necessary if only the birthdate was retrieved through projection.

## Indexes: Multikey Indexes

Indexes on an array field can index primitives, subdocuments or subarrays.

Multikey indexes can be single field or compound indexes.

**Important!:** Mongodb forces a limitation of 1 array per index. Hence, for a compound index only one index can be an array.



## Indexes: Multikey Indexes

### Interesting fact:

- Internally MongoDB decomposes the array and stores each unique value found within it as an individual entry.
- The multikey indexes need to fetch the documents after IXSCAN stage because the index entries have each of the array values stored separately.

## Indexes: Compound Indexes

- Index on multiple fields.
- Supports queries that match on the prefix of the index fields.

```
db.account.createIndex({limit:9000,products:1})
```

```
db.account.explain().find({account_id:{$gte:557378}})
```

```
db.account.explain().find({limit:9000,account_id:{$gte:557378}})
```

## Indexes: Compound Indexes

- The order of the fields in a compound index matters.
- Recommended order for creation:  
Equality, Sort, Range

**Equality** predicate: testing for exact matches should be placed first. Example: `limit:9000`

**Sort** predicate: Determines the order of the results.  
Example: `products:1`

## Indexes: Compound Indexes

**Range filter:** Use projections to return only the fields included in index to avoid fetching all documents.

```
db.accounts.find({limit:9000},{products:1})
```

```
db.accounts.explain().find({limit:9000},{products:1})
```

## Deleting Indexes

- Indexes have a write cost so we should delete redundant or non-frequently used indexes.

**Is this completely true?**

## Deleting Indexes

- In production we hide indexes to test whether they affect queries, this is because creating indexes again takes time.

```
db.customers.getIndexes()
```

```
db.customers.hideIndex({birthdate:1})
```

```
db.customers.explain().find({birthdate:{$lte:ISODate('1973-01-01')}})
```

```
db.customers.unhideIndex({birthdate:1})
```

## Deleting Indexes

- The `dropIndex` command deletes the index from the system.

```
db.customers.dropIndex({birthdate:1})
```

## Aggregation

- Aggregation provides us summary data.

Stage: An aggregation operation performed on the data.

Aggregation pipeline: A series of stages completed one at a time in order.

- Aggregations can be performed individually but can be made into a string of operations called a pipeline.



## Aggregation

### Syntax:

```
db.collection.aggregate([  
    {$stage_name: {<Expression>}},  
    {$stage_name: {<Expression>}}  
])
```

## Aggregation: \$match stage

- Filters for data that matches criteria

```
db.accounts.aggregate([{$match:{ limit: {$gte:  
11000}}}] )
```

- \$match stage is placed at the beginning of the aggregation pipeline to make use of indexes.

## Aggregation: \$group stage

- Creates a single document for each distinct value.
- Uses a group key to group output.

```
db.transactions.aggregate([
  {
    $group: {
      _id: "$limit",    // Group by the "limit" field
      count: { $sum: 1 } // Count the number of
documents for each group
    }
  }
])
```

## Aggregation: \$sort and \$limit stage

- \$sort: Sorts all input data passed through the pipeline in sorted order.
- 1: Ascending order, -1: Descending order

```
db.customers.aggregate([ { $sort: {name:1} } ])
```

- \$limit: Limits the number of documents passed down from the pipeline.

```
db.customers.aggregate([ { $sort:{name:1} }, { $limit:3 } ])
```

## Aggregation: \$sort and \$limit stage

```
db.customers.aggregate([ { $sort:{name:1} } , {$limit:3} ] )
```

**In this example what will happen if i pass \$limit before sort?**

## Aggregation: \$project stage

- Determines the output document stage.
- Allows one to determine the fields that can be returned by the aggregation.
- Should be the last stage as it specifies the output.
- Follows inclusion(1) and exclusion(0).
- Can also create a new field if it does not exist.

```
db.customers.aggregate([{$project: {name:1, nickname:  
'$username', _id:0}}])
```

## Aggregation: \$set stage

- Adds or modifies fields in the pipeline.
- Useful when one wants to change value of existing fields in the pipeline or add new fields in the upcoming pipeline stage

```
db.accounts.aggregate([{$set: {balance: '$limit'}}])
```

## Aggregation: \$out stage

- Writes documents that are returned by an aggregation pipeline into a collection.
- Must be the **last stage**.
- Creates a new collection if it does not already exist.
- If collection exists, \$out replaces the existing collection with new data.

Syntax:

```
$out: { db: "<db_name>", col: "<collection_name>" }
```



## Aggregation: \$out stage

```
db.customers.aggregate([{$match:{username:"valenciajennifer"},{$group:{_id:"$username",name:{$first:"$name"}}},{$limit:1},{$sort:{name:1}},{$project:{_id:0,name:1}},{$set:{nickname:"$name"}},{$out:"new_collection_demo"}])
```

## Transactions in MongoDB

ACID Transactions: A group of DB operations that will be completed as a unit or not at all.

What is the full form of ACID?

What does each quality represent in a transaction?

## Transactions in MongoDB

- Single document operations are atomic in nature.

**Is this true?Why?**

## Transactions in MongoDB

- Multi-document operations are not atomic in nature.
- MongoDB locks resources when involved in a transaction so multi-document operations should only be performed in special cases.
- It is more common to do transactions through **drivers**(Through programming languages).

**Important!:** By default transactions in MongoDB have a maximum runtime or less than 1 minute after the first write.

## Transactions in MongoDB

### Sessions

- Are used to group database operations that are related to each other and should be run together.

**Note:** If commands in a transaction take more than 1 minute a mongodb server error will be thrown when the transaction timeouts.

## Transactions in MongoDB

Mongodb only allows transactions on replica sets:

Workaround: Using run-rs

```
npm install run-rs -g
```

```
run-rs -v 4.0.0 --shell
```

Note: keep run-rs, running in the background

## Transactions in MongoDB

Open mongodb compass:

Use the following connection string to connect :

**mongodb://localhost:27017,localhost:27018,localhost:27019/test?replicaSet=rs**

Create database "analytics" and Create collection "accounts"

Load data from previously downloaded **accounts.js** into collection.

# Transactions in MongoDB

## New Connection

Connect to a MongoDB deployment



FAVORITE

URI 

Edit Connection String ☒

```
mongodb://localhost:27017,localhost:27018,localhost:27019/test?replicaSet=rs
```

➤ Advanced Connection Options

Save

Save & Connect

Connect



## Transactions in MongoDB

Let us do a transaction on accounts where the limit of one account is deducted and then incremented to another account. (Money transfer simulation)

```
db.accounts.find({account_id: 328304})
```

```
db.accounts.find({account_id: 487188})
```

## Transactions in MongoDB

```
const session = db.getMongo().startSession()
session.startTransaction()

const accounts =
session.getDatabase('analytics').getCollection('accounts')

accounts.updateOne({account_id: 328304}, {$inc:
{limit:-100}})

accounts.updateOne({account_id: 487188}, {$inc: {limit:
100}})

session.commitTransaction()
```

## Transactions in MongoDB

```
db.accounts.find({account_id: 328304})
```

```
db.accounts.find({account_id: 487188})
```

To abort transaction:

```
session.abortTransaction()
```

- Does not have feedback but aborts transaction

Recap:

Indexing

Aggregation

Transactions

DB connection: Guided Session