MongoDB CRUD Operations: A Practical Tutorial

This tutorial provides a comprehensive explanation and practical examples of the fundamental CRUD (Create, Read, Update, Delete) operations in MongoDB using the mongosh shell syntax.

We'll use a sample products collection for all our examples.

Setting Up Our Sample Data

First, let's assume we are connected to a MongoDB instance (e.g., by typing mongosh in your terminal if local, or using a connection string for Atlas).

We'll use a database called store_db and a collection called products.

```
// Switch to (or create) the 'store_db' database
use store_db;

// Optional: Drop the collection if it already exists to start fresh
db.products.drop();
```

1. Create Operations (Inserting Documents)

These operations add new documents to a collection.

insertOne(): Insert a Single Document

- **Purpose:** Inserts a single new document into a collection.
- **Syntax:** db.collection.insertOne(document)

Example:

```
db.products.insertOne({
  name: "Laptop Pro X",
  category: "Electronics",
  price: 1200.00,
  stock: 50,
  tags: ["electronics", "laptop", "high-performance"],
  details: {
    brand: "TechCorp",
    model: "ProX-2024",
    processor: "Intel i7",
    ram_gb: 16
  },
  reviews: [
    { user: "Alice", rating: 5, comment: "Amazing speed!" },
    { user: "Bob", rating: 4, comment: "Good value for money." }
  ],
  is_available: true
});
```

- MongoDB will automatically add an _id field to the document if you don't provide one.

insertMany(): Insert Multiple Documents

- **Purpose:** Inserts multiple documents into a collection in a single operation.
- **Syntax:** db.collection.insertMany([document1, document2, ...])

Example:

```
db.products.insertMany([
  {
    name: "Wireless Mouse",
    category: "Electronics",
    price: 25.50,
    stock: 200,
    tags: ["accessories", "wireless"],
    details: { brand: "ErgoGear", color: "black" },
    is_available: true
  },
  {
    name: "Mechanical Keyboard",
    category: "Electronics",
    price: 80.00,
    stock: 75,
    tags: ["accessories", "gaming", "mechanical"],
    details: { brand: "ClickyKeys", layout: "US" },
    is_available: true
  },
  {
    name: "Organic Coffee Beans",
    category: "Groceries",
    price: 15.00,
    stock: 150,
    tags: ["coffee", "organic", "beverages"],
    details: { origin: "Colombia", weight_kg: 0.5 },
    is_available: true
  },
  {
    name: "Smart Speaker Echo",
    category: "Smart Home",
    price: 99.99,
    stock: 120,
    tags: ["smart-home", "voice-assistant"],
    details: { brand: "HomeAI", assistant: "Echo" },
    is_available: false // Currently out of stock/unavailable
  },
  {
    name: "Yoga Mat Pro",
    category: "Fitness",
    price: 40.00,
    stock: 80,
    tags: ["fitness", "yoga", "accessories"],
    details: { material: "TPE", thickness_mm: 6 },
    reviews: [
      { user: "Charlie", rating: 5, comment: "Very comfortable!" }
    ],
    is_available: true
  }
]);
```

2. Read Operations (Querying Documents)

These operations retrieve documents from a collection based on specified criteria.

find(): Basic Queries

- **Purpose:** Retrieves documents that match a specified query filter.
- **Syntax:** db.collection.find(query_filter, projection)
  - query_filter: An object specifying the criteria. An empty object {} matches all documents.
  - projection (optional): An object specifying which fields to include or exclude.

Examples:

- **Find all documents:**
  db.products.find();

- **Find documents matching a specific field value:**
  db.products.find({ category: "Electronics" });

- **Find documents matching multiple field values (AND implicitly):**
  db.products.find({ category: "Electronics", stock: { $gt: 100 } });

find() with Projection

- **Purpose:** To return only specific fields from the matching documents, reducing network traffic and data processing.
- **Syntax:** db.collection.find(query_filter, { field1: 1, field2: 1, _id: 0 })
  - 1: Include the field.
  - 0: Exclude the field.
  - _id field is included by default; use _id: 0 to exclude it.

Example:

```
// Find all products, but only show 'name' and 'price'
db.products.find({}, { name: 1, price: 1, _id: 0 });
```

Query Operators

These operators are used within the query_filter to specify more complex conditions.

a. Comparison Operators

Operator	Description	Example
$eq	Equal to (default behavior)	{ price: { $eq: 25.50 } } (same

	(check the extra material)	as { price: 25.50 })
$gt	Greater than	{ stock: { $gt: 100 } }
$gte	Greater than or equal to	{ price: { $gte: 100.00 } }
$lt	Less than	{ price: { $lt: 50.00 } }
$lte	Less than or equal to	{ stock: { $lte: 75 } }
$ne	Not equal to	{ category: { $ne: "Electronics" } }
$in	Value is in an array of values	{ tags: { $in: ["gaming", "coffee"] } }
$nin	Value is NOT in an array of values	{ tags: { $nin: ["organic"] } }

Examples:

```
// Find products with price greater than 50
db.products.find({ price: { $gt: 50.00 } });

// Find products that are either 'Electronics' or 'Fitness' category
db.products.find({ category: { $in: ["Electronics", "Fitness"] } });
```

b. Logical Operators

Operator	**Description**	**Example**
$and	Joins query clauses with a logical AND.	{ $and: [{ price: { $lt: 50 } }, { stock: { $gt: 0 } }] }
$or	Joins query clauses with a logical OR.	{ $or: [{ category: "Groceries" }, { is_available: false }] }
$not	Inverts the effect of a query expression.	{ price: { $not: { $gt: 100 } } } (price not > 100, i.e., price <= 100)

$nor	Joins query clauses with a logical NOR. (Returns documents that fail to match both query clauses)	{ $nor: [{ category: "Electronics" }, { stock: { $lt: 10 } }] }

Examples:

```
// Find products that are either 'Groceries' OR are not available
db.products.find({ $or: [{ category: "Groceries" }, { is_available: false }] });

// Find products where price is NOT greater than 100 (i.e., price <= 100)
db.products.find({ price: { $not: { $gt: 100 } } });
```

c. Element Operators

Operator	Description	Example
$exists	Matches documents that have the specified field.	{ reviews: { $exists: true } } (has reviews field)
$type	Matches documents where the field is of a specified BSON type.	{ price: { $type: "double" } } (price is a double)

Examples:

```
// Find products that have a 'reviews' field
db.products.find({ reviews: { $exists: true } });

// Find documents where 'stock' field is of type 'int' (BSON type 16)
db.products.find({ stock: { $type: "int" } });
```

d. Array Operators

Operator	Description	Example

$all	Matches arrays that contain all specified elements.	{ tags: { $all: ["accessories", "wireless"] } }
$size	Matches arrays with a specified number of elements.	{ tags: { $size: 3 } }
$elemMatch	Matches documents that contain an array field with at least one element that matches all the specified query criteria.	{ reviews: { $elemMatch: { rating: { $gte: 4 }, user: "Bob" } } }

Examples:

```
// Find products that have both "accessories" AND "gaming" tags
db.products.find({ tags: { $all: ["accessories", "gaming"] } });

// Find products that have exactly 3 tags
db.products.find({ tags: { $size: 3 } });

// Find products where at least one review has a rating >= 4 and is by user "Bob"
db.products.find({ reviews: { $elemMatch: { rating: { $gte: 4 }, user: "Bob" } } });
```

e. Regular Expressions ($regex)

- **Purpose:** Allows you to match string patterns using regular expressions.
- **Syntax:** { field: { $regex: /pattern/, $options: 'options' } } or { field: { $regex: 'pattern', $options: 'options' } }
    - $options: Common options include i (case-insensitive), m (multiline), x (whitespace ignored).

Example:

```
// Find products whose name starts with "Smart" (case-insensitive)
db.products.find({ name: { $regex: /^Smart/, $options: 'i' } });

// Find products with "key" anywhere in the name (case-insensitive)
db.products.find({ name: { $regex: "key", $options: 'i' } });
```

f. Text Search ($text)

- **Purpose:** Performs a full-text search on string content.
- **Requirement:** Requires a **text index** to be created on the collection first.
- **Syntax:** db.collection.find({ $text: { $search: "search terms" } })

Example (First, create a text index):

```
db.products.createIndex({ name: "text", tags: "text", "details.brand": "text" });

// Now, perform a text search
db.products.find({ $text: { $search: "coffee organic" } });
// This will find documents where 'coffee' OR 'organic' appear in the indexed fields.
```

Chaining Query Methods (sort(), limit(), skip())

These methods are chained after find() to refine the results.

sort(): Sorting Results

- **Purpose:** Sorts the documents returned by the query.
- **Syntax:** db.collection.find(...).sort({ field1: 1, field2: -1 })
  - 1: Ascending order.
  - -1: Descending order.

Example:

```
// Find all products, sort by price in ascending order
db.products.find().sort({ price: 1 });

// Find products in 'Electronics' category, sort by price (asc) then stock (desc)
db.products.find({ category: "Electronics" }).sort({ price: 1, stock: -1 });
```

limit(): Limiting Results

- **Purpose:** Restricts the number of documents returned by the query.
- **Syntax:** db.collection.find(...).limit(number)

Example:

```
// Get the 3 cheapest products
db.products.find().sort({ price: 1 }).limit(3);
```

skip(): Skipping Results

- **Purpose:** Skips a specified number of documents from the beginning of the result set. Useful for pagination.
- **Syntax:** db.collection.find(...).skip(number)

Example (Pagination - Page 2, 3 items per page):

```
// Get the next 3 products after the first 3 (i.e., items 4, 5, 6 after sorting)
db.products.find().sort({ name: 1 }).skip(3).limit(3);
```

3. Update Operations

These operations modify existing documents in a collection.

updateOne(): Update a Single Document

- **Purpose:** Updates a single document that matches the specified filter.
- **Syntax:** db.collection.updateOne(filter, update_document, options)

Example:

```
// Update the stock of "Laptop Pro X"
db.products.updateOne(
  { name: "Laptop Pro X" },
  { $set: { stock: 45, last_updated: new Date() } }
);
```

updateMany(): Update Multiple Documents

- **Purpose:** Updates all documents that match the specified filter.
- **Syntax:** db.collection.updateMany(filter, update_document, options)

Example:

```
// Increase the price of all 'Electronics' products by 10%
db.products.updateMany(
  { category: "Electronics" },
  { $mul: { price: 1.10 } }
);
```

replaceOne(): Replace an Entire Document

- **Purpose:** Replaces a single document that matches the filter with a completely new document. The _id field must remain the same.
- **Syntax:** db.collection.replaceOne(filter, new_document, options)
- **Caution:** This replaces the *entire* document, so any fields not present in the new_document will be removed.

Example:

```
// First, find the _id of the document you want to replace (e.g., "Wireless Mouse")
// db.products.find({ name: "Wireless Mouse" }, { _id: 1 }); // Get its _id

// Let's assume its _id is ObjectId("654321abcdef...") for this example
db.products.replaceOne(
  { _id: ObjectId("654321abcdef01234567890abc") }, // Use the actual _id
  {
    name: "New Ergonomic Mouse",
    category: "Electronics",
    price: 35.00,
    is_available: true,
    // Note: 'stock', 'tags', 'details' fields are removed if not included here
  }
);
```

Update Operators

These operators are used within the update_document to specify how fields should be modified.

a. Field Update Operators

Operator	Description	Example
$set	Sets the value of a field. Creates the field if it doesn't exist.	{ $set: { status: "active", quantity: 10 } }

$unset	Removes a field from a document.	{ $unset: { "details.processor": "" } }
$inc	Increments the value of a field by a specified amount.	{ $inc: { stock: 5 } }
$mul	Multiplies the value of a field by a specified amount.	{ $mul: { price: 0.9 } }
$rename	Renames a field.	{ $rename: { "oldName": "newName" } }

Examples:

```
// Set a new field 'warranty_years' for "Laptop Pro X"
db.products.updateOne(
  { name: "Laptop Pro X" },
  { $set: { warranty_years: 2 } }
);

// Remove the 'details.processor' field from "Laptop Pro X"
db.products.updateOne(
  { name: "Laptop Pro X" },
  { $unset: { "details.processor": "" } } // Value can be anything, it's ignored
);

// Increase the stock of "Wireless Mouse" by 10
db.products.updateOne(
  { name: "Wireless Mouse" },
  { $inc: { stock: 10 } }
);

// Rename 'stock' field to 'inventory_count' for "Organic Coffee Beans"
db.products.updateOne(
  { name: "Organic Coffee Beans" },
  { $rename: { "stock": "inventory_count" } }
);
```

b. Array Update Operators

Operator	Description	Example
$push	Adds an element to an array.	{ $push: { tags: "new-tag" } }
$pull	Removes all instances of a value from an array.	{ $pull: { tags: "old-tag" } }
$addToSet	Adds an element to an array only if it's not already present.	{ $addToSet: { tags: "unique-tag" } }
$pop	Removes the first (-1) or last (1) element from an array.	{ $pop: { tags: 1 } } (removes last element)

Examples:

```
// Add a new tag to "Mechanical Keyboard"
db.products.updateOne(
  { name: "Mechanical Keyboard" },
  { $push: { tags: "rgb" } }
);

// Remove the "wireless" tag from "Wireless Mouse"
db.products.updateOne(
  { name: "Wireless Mouse" },
  { $pull: { tags: "wireless" } }
);

// Add "ergonomic" to tags of "Wireless Mouse" only if it's not already there
db.products.updateOne(
  { name: "Wireless Mouse" },
  { $addToSet: { tags: "ergonomic" } }
);

// Remove the last tag from "Yoga Mat Pro"
db.products.updateOne(
  { name: "Yoga Mat Pro" },
  { $pop: { tags: 1 } } // 1 for last, -1 for first
);
```

4. Delete Operations

These operations remove documents or entire collections/databases.

deleteOne(): Delete a Single Document

- **Purpose:** Deletes at most one document that matches the specified filter.
- **Syntax:** db.collection.deleteOne(filter)

Example:

```
// Delete the "Smart Speaker Echo" document
db.products.deleteOne({ name: "Smart Speaker Echo" });
```

deleteMany(): Delete Multiple Documents

- **Purpose:** Deletes all documents that match the specified filter.
- **Syntax:** db.collection.deleteMany(filter)

Example:

```
// Delete all products in the 'Groceries' category
db.products.deleteMany({ category: "Groceries" });

// Delete all documents in the collection (empty filter)
// db.products.deleteMany({});
```

drop(): Drop a Collection or Database

- **Purpose:**
  - db.collection.drop(): Permanently deletes an entire collection, including all its documents and indexes.
  - db.dropDatabase(): Permanently deletes the current database and all its collections.
- **Caution:** These operations are irreversible and should be used with extreme care, especially in production environments.

Examples:

```
// Drop the 'products' collection
db.products.drop(); // Returns true if successful, false if collection didn't exist

// Drop the entire 'store_db' database (make sure you are 'use store_db' first)
// db.dropDatabase();
```

This tutorial covers the essential CRUD operations in MongoDB. Practice these commands with different filters and operators to gain a solid understanding of how to manipulate data effectively in MongoDB.

EXTRA MATERIAL

Yes, the `$eq` operator in MongoDB is used to match documents where the value of a field is equal to a specified value.

While it's the default behavior when you simply specify a field-value pair (e.g., `{ category: "Electronics" }` is implicitly `category: { $eq: "Electronics" }`), there are specific use cases where explicitly using `$eq` can be beneficial or necessary:

1. **Clarity and Readability:**
   When you're writing complex queries with many operators, explicitly using $eq can sometimes make the query more readable and consistent with other operators like $gt, $lt, $ne, etc. It clearly states the intent of an equality comparison.
   - **Example:**
   - **// Explicitly stating equality for clarity**

     ```
     db.products.find({ price: { $eq: 1200.00 } });
     ```

   - **This is functionally identical to `db.products.find({ price: 1200.00 });`, but in a query with many operators, it might fit a pattern.**
2. **Programmatic Query Construction:**

When you are building MongoDB queries programmatically in your application code (e.g., using Node.js drivers), it can sometimes be easier to always use the operator syntax for consistency, even for equality.

- Example (Conceptual Node.js code):

```
const query = {};

if (someCondition) {

  query.category = { $eq: "Electronics" }; // Programmatically adding the $eq operator

} else {

  query.stock = { $gt: 100 };

}

// db.products.find(query);
```

-
-

3. When $eq is part of a larger, nested operator:
   While less common for $eq itself, sometimes you might embed it within other operators where the explicit operator syntax is required.
   - Example (Hypothetical, less practical for $eq but demonstrates nesting):
     If you had a very complex $or condition where one part was an equality check that you wanted to visually separate.

```
db.products.find({

 $or: [

  { "details.brand": { $eq: "TechCorp" } }, // Explicit $eq within $or

  { price: { $lt: 50 } }

 ]

});
```

○

○

In most simple equality queries, omitting `$eq` is perfectly fine and standard practice, as MongoDB assumes it by default. The cases above are primarily about developer preference for explicit syntax or programmatic query building.