MongoDB - Document-based NoSQL System

Week 10

COS60009: Data Management for the Big Data Age

SWINBURNE UNIVERSITY OF TECHNOLOGY

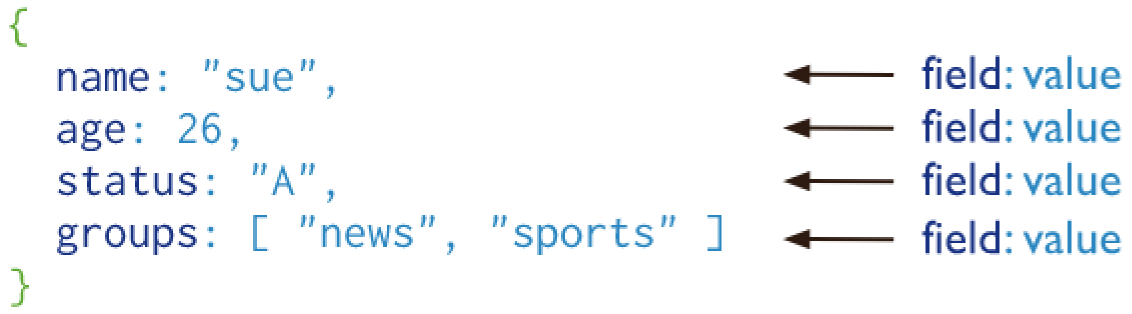
Learning Objectives

- Introduction to MongoDB
- MongoDB CRUD Operations
  - Insert Documents
  - Query Documents
  - Update Documents
  - Delete Documents
  - Bulk Write
- SQL to MongoDB Mapping
- Aggregation and Map-Reduce
- Transactions
- Other MongoDB Features

Refer to **https://docs.mongodb.com/manual/**

Introduction to MongoDB

- MongoDB is a document database designed for ease of development and scaling
- Document Database
  - A record in MongoDB is a document, which is a data structure composed of field and value pairs
  - MongoDB documents are similar to JSON objects.
  - The values of fields may include other documents, arrays, and arrays of documents.
- The advantages of using documents
  - Documents (i.e. objects) correspond to native data types in many programming languages.
  - Embedded documents and arrays reduce need for expensive joins.
  - Dynamic schema supports fluent polymorphism.



```
{
   name: "sue",
   age: 26,
   status: "A",
   groups: [ "news", "sports" ]
}
```

Key Features

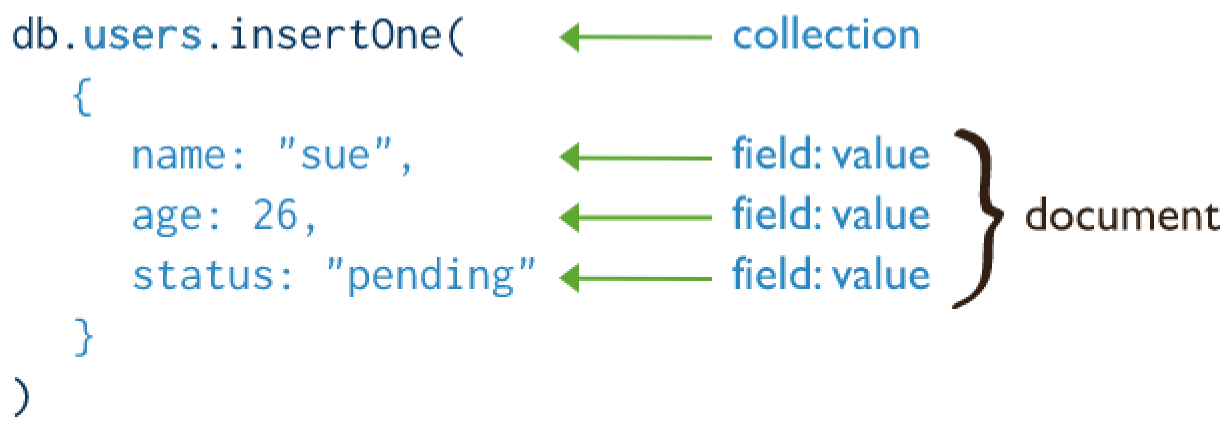
- High Performance
  - Support for embedded data models reduces I/O activity on database system.
  - Indexes support faster queries and can include keys from embedded documents and arrays.
- Rich Query Language
  - Support read and write operations (CRUD)
  - Data Aggregation
  - Text Search and Geospatial Queries.
- High Availability
  - replication facility, called replica set, provides: automatic failover and data redundancy
- Horizontal Scalability
  - Sharding distributes data across a cluster of machines.
  - Starting in 3.4, MongoDB supports creating zones of data based on the shard key.
- Support for Multiple Storage Engines

MongoDB CRUD Operations

- Create Operations
- Read Operations
- Update Operations
- Delete Operations
- Bulk Write

Create Operations

- Create or insert operations add new documents to a collection. If the collection does not currently exist, insert operations will create the collection.
- MongoDB provides the following methods to insert documents into a collection:
  - db.collection.insertOne() New in version 3.2
  - db.collection.insertMany() New in version 3.2
- In MongoDB, insert operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.



```
db.users.insertOne(
   {
      name: "sue",
      age: 26,
      status: "pending"
   }
)
```

Insert a Single Document

- db.collection.insertOne() inserts a single document into a collection.
- The following example inserts a new document into the inventory collection. If the document does not specify an _id field, MongoDB adds the _id field with an ObjectId value to the new document.

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

- insertOne() returns a document that includes the newly inserted document's _id field value.

Insert Multiple Documents

- db.collection.insertMany() can insert multiple documents into a collection. Pass an array of documents to the method.
- The following example inserts new documents into the inventory collection. If the documents do not specify an _id field, MongoDB adds the _id field with an ObjectId value to each document.

```
db.inventory.insertMany([
    // MongoDB adds the _id field with an ObjectId if _id is not present
    { item: "journal", qty: 25, status: "A",
      size: {h: 14, w: 21, uom: "cm"}, tags: ["black", "red"]},
    { item: "notebook", qty: 50, status: "A",
      size: {h: 8.5, w: 11, uom: "in"}, tags: ["red", "black"]},
    { item: "paper", qty: 100, status: "D",
      size: {h: 8.5, w: 11, uom: "in"}, tags: ["red", "black", "plain"]},
    { item: "planner", qty: 75, status: "D",
      size: {h: 22.85, w: 30, uom: "cm"}, tags: ["black", "red"]},
    { item: "postcard", qty: 45, status: "A",
      size: {h: 10, w: 15.25, uom: "cm"}, tags: ["blue"]}
]);
```

- insertMany() returns a document that includes the newly inserted documents _id field values

Insert Behavior

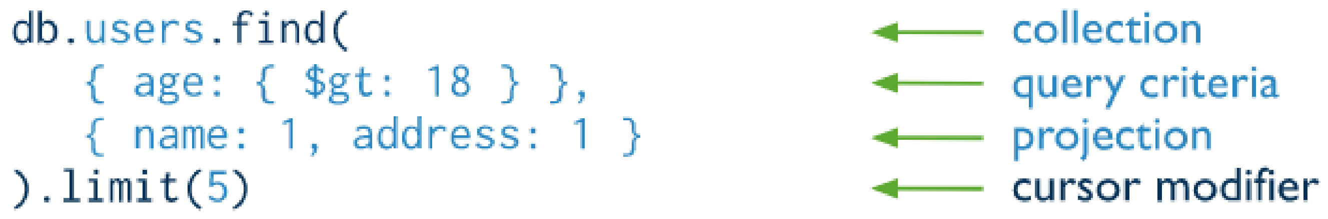
- Collection Creation
  - If the collection does not currently exist, insert operations will create the collection.
- _id Field
  - In MongoDB, each document stored in a collection requires a unique _id field that acts as a primary key. If an inserted document omits the _id field, the MongoDB driver automatically generates an ObjectId for the _id field.
  - This also applies to documents inserted through update operations with upsert: true.
- Atomicity
  - All write operations in MongoDB are atomic on the level of a single document. For more information on MongoDB and atomicity, see Atomicity and Transactions
- Write Acknowledgement
  - With write concerns, you can specify the level of acknowledgement requested from MongoDB for write operations. For details, see Write Concern

Read Operations

- Read operations retrieves documents from a collection; i.e. queries a collection for documents. MongoDB provides the following methods to read documents from a collection

```
db.collection.find( )
```

- You can specify query filters or criteria that identify the documents to return.



```
db.users.find(
    { age: { $gt: 18 } },
    { name: 1, address: 1 }
).limit(5)
```

Query Documents

- To select all documents in the collection, pass an empty document as the query filter document to the db.collection.find() method.

```
db.inventory.find( {} )
```

- To query for documents that match specific equality conditions, pass the find() method a query filter document with the <field>: <value> of the desired documents.

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

Query Documents with Operators/AND/OR

- The following example retrieves all documents from the inventory collection where status equals either "A" or "D"

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

- A compound query can specify conditions for more than one field in the collection’s documents. *Implicitly*, a logical AND conjunction connects the clauses of a compound query so that the query selects the documents in the collection that match all the conditions.
- The following example retrieves all documents in the inventory collection where the status equals "A" and qty is less than ($lt) 30.

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

- Using the $or operator, you can specify a compound query that joins each clause with a logical OR conjunction so that the query selects the documents in the collection that match at least one condition.
- The following example retrieves all documents in the collection where the status equals "A" or qty is less than ($lt) 30

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

Query on Embedded/Nested Documents

- Match an Embedded/Nested Document
  - The following query selects all documents where the field size equals the document { h: 14, w: 21, uom: "cm" }, here the field order is sensitive.

```
db.inventory.find( {size: { h: 14, w: 21, uom: "cm" }
} )
```

- Specify Equality Match on a Nested Field
  - The following example selects all documents where the field uom nested in the size field equals the string value "in"

```
db.inventory.find( { “size.uom”: “in”} )
```

- Specify Match using Query Operator
  - The following query uses the less than operator ($lt) on the field h embedded in the size field

```
db.inventory.find( { "size.h": { $lt: 15 } } )
```

- Specify AND Condition
  - The following query selects all documents where the nested field h is less than 15, the nested field uom equals "in", and the status field equals "D".

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

Query an Array

- Match an Element in an Array
  - The following example queries for all documents where tags is an array that contains the string "red" as one of its elements

```
db.inventory.find( { tags: “red”} )
```

- Match an Array Exactly
  - The following example queries for all documents where the field tags value is an array with exactly two elements, "red" and "black", in the specified order.

```
db.inventory.find( {tags: [“red”, “black”]
} )
```

- Query for an Element by the Array Index Position
  - The following example queries for all documents where the second element in the array tags is “black”

```
db.inventory.find( { “tags.1": “black” } )
```

- Query an Array by Array Length
  - Use the $size operator to query for arrays by number of elements. For example, the following selects documents where the array tags has 3 elements

```
db.inventory.find( { "tags": { $size: 3 } }
)
```

Query an Array of Embedded Documents

```
db.inventory.insertMany([
   {item:"journal",instock:[{warehouse:"A",qty:5},{warehouse:"C“, qty:15}]},
   {item:"notebook",instock:[{warehouse:"C",qty:5}]},
   {item:"paper",instock:[{warehouse:"A",qty:60},{warehouse:"B",qty:15}]},
   {item:"planner",instock:[{warehouse:"A",qty:40},{warehouse:"B",qty:5}]},
   {item:"postcard",instock:[{warehouse:"B",qty:15},{warehouse:"C",qty:35}]}
]);
```

- Query for a Document Nested in an Array
  - The following example selects all documents where an element in the instock array matches the specified document. Equality matches on the whole embedded/nested document require an exact match of the specified document, including the field order.

```
db.inventory.find({ "instock": { warehouse: "A", qty: 5 }})
```

- Specify a Query Condition on a Field Embedded in an Array of Documents
  - The following example selects all documents where the instock array has at least one embedded document that contains the field qty whose value is less than or equal to 20.

```
db.inventory.find({ 'instock.qty': { $lte: 20 }})
```

Query an Array of Embedded Documents (cont’d)

- Use the Array Index to Query for a Field in the Embedded Document
  - The following example selects all documents where the instock array has as its first element a document that contains the field qty whose value is less than or equal to 20.

```
db.inventory.find({'instock.0.qty':{ $lte:20 }})
```

- A Single Nested Document Meets Multiple Query Conditions on Nested Fields
  - Use $elemMatch operator to specify multiple criteria on an array of embedded documents such that at least one embedded document satisfies all the specified criteria.
  - The following example queries for documents where the instock array has at least one embedded document that contains both the field qty equal to 5 and the field warehouse equal to A.

```
db.inventory.find({"instock":{$elemMatch:{qty:5,warehouse:"A"}
```

  - The following example queries for documents where the instock array has at least one embedded document that contains the field qty that is greater than 10 and less than or equal to 20.

```
db.inventory.find({"instock":{$elemMatch:{qty:{$gt:10,$lte:20
}}}})
```

Query an Array of Embedded Documents (cont'd)

- Combination of Elements Satisfies the Criteria
  - If the compound query conditions on an array field do not use the $elemMatch operator, the query selects those documents whose array contains any combination of elements that satisfies the conditions.
  - The following query matches documents where any document nested in the instock array has the qty field greater than 10 and any document (but not necessarily the same embedded document) in the array has the qty field less than or equal to 20.

```
db.inventory.find( { "instock.qty": { $gt: 10,  $lte: 20 } } )
```

  - The following example queries for documents where the instock array has at least one embedded document that contains the field qty equal to 5 and at least one embedded document (but not necessarily the same embedded document) that contains the field warehouse equal to A.

```
db.inventory.find( { "instock.qty": 5, "instock.warehouse": "A" } )
```

Project Fields to Return from Query

```
db.inventory.insertMany( [
  { item: "journal", status: "A", size: { h: 14, w: 21, uom: "cm" }, instock: [ { warehouse: "A", qty: 5 } ] },
  { item: "notebook", status: "A",  size: { h: 8.5, w: 11, uom: "in" }, instock: [ { warehouse: "C", qty: 5 } ] },
  { item: "paper", status: "D", size: { h: 8.5, w: 11, uom: "in" }, instock: [ { warehouse: "A", qty: 60 } ] },
  { item: "planner", status: "D", size: { h: 22.85, w: 30, uom: "cm" }, instock: [ { warehouse: "A", qty: 40 } ] },
  { item: "postcard", status: "A", size: { h: 10, w: 15.25, uom: "cm" }, instock: [ { warehouse: "B", qty: 15 }, {
warehouse: "C", qty: 35 } ] }
]);
```

- By default, queries in MongoDB return all fields in matching documents. To limit the amount of data that MongoDB sends to applications, you can include a projection document to specify or restrict fields to return.
- Return the Specified Fields and the _id Field Only
  - The following operation returns all documents that match the query. In the result set, only the item, status and, by default, the _id fields return in the matching documents

```
db.inventory.find({status: "A"}, {item: 1, status: 1})
```

- Suppress _id Field

```
db.inventory.find({status: "A"}, {item: 1, status: 1, _id:
0})
```

Project Fields to Return from Query (Cont'd)

- Return All But the Excluded Fields
  - The following example which returns all fields except for the status and the instock fields in the matching documents

```
db.inventory.find({status: "A"}, {status: 0, instock: 0})
```

- Return Specific Fields in Embedded Documents
  - The following example returns: the _id field (returned by default), the item field, the status field, the uom field in the size document.

```
db.inventory.find(
   {status: "A"},
   {item: 1, status: 1, "size.uom":
1}
)
```

- Projection on Embedded Documents in an Array
  - The following example specifies a projection to return: the _id field (returned by default), the item field, the status field, the qty field in the documents embedded in the instock array

```
db.inventory.find({status:"A"},{item:1,status:1,"instock.qty
":1})
```

Query for Null or Missing Fields

```
db.inventory.insertMany([
   { _id: 1, item: null },
   { _id: 2 }
])
```

- Equality Filter
  - The { item : null } query matches documents that either contain the item field whose value is null or that do not contain the item field

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

- Type Check
  - The { item : { $type: 10 } } query matches only documents that contain the item field whose value is null; i.e. the value of the item field is of BSON Type Null (type number 10)

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

- Existence Check
  - The { item : { $exists: false } } query matches documents that do not contain the item field

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

Iterate a Cursor in the mongo Shell

- The db.collection.find() method returns a cursor. To access the documents, you need to iterate the cursor. However, in the mongo shell, if the returned cursor is not assigned to a variable using the var keyword, then the cursor is automatically iterated up to 20 times to print up to the first 20 documents in the results
- You can call the cursor variable in the shell to iterate up to 20 times and print the matching documents, or use the cursor method next() to access the documents, or use iterator index

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

```
var myCursor = db.users.find( { type: 2 } );
while (myCursor.hasNext()) {
  print(tojson(myCursor.next()));
}
```

```
var myCursor = db.inventory.find( { type: 2 } );
var documentArray = myCursor.toArray();
var myDocument = documentArray[3];
```

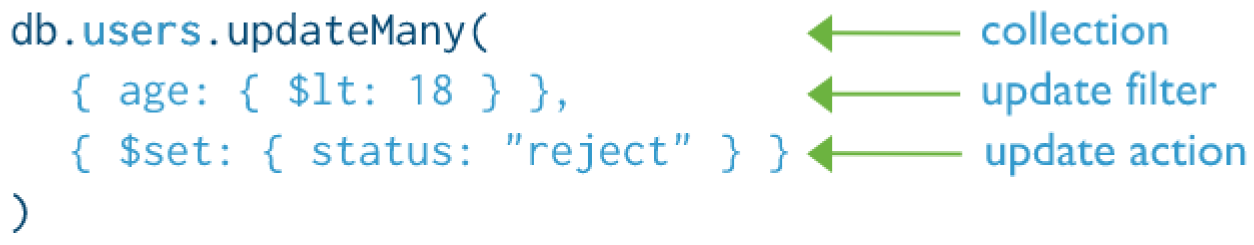
```
var myCursor = db.users.find( { type: 2 } );
var myDocument = myCursor[1];
```

- Use the limit() method on a cursor to specify the maximum number of documents the cursor will return

```
db.collection.find(<query>).limit(<number>)
```

Update Operations

- Update operations modify existing documents in a collection. MongoDB provides the following methods to update documents of a collection:
  - db.collection.updateOne() New in version 3.2
  - db.collection.updateMany() New in version 3.2
  - db.collection.replaceOne() New in version 3.2
- In MongoDB, update operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.
- You can specify criteria, or filters, that identify the documents to update. These filters use the same syntax as read operations.



```
db.users.updateMany(
   { age: { $lt: 18 } },
   { $set: { status: "reject" } }
)
```

Update Documents

```
db.inventory.insertMany( [
   { item: "canvas", qty: 100, size: { h: 28, w: 35.5, uom: "cm" }, status: "A" },
   { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
   { item: "mat", qty: 85, size: { h: 27.9, w: 35.5, uom: "cm" }, status: "A" },
   { item: "mousepad", qty: 25, size: { h: 19, w: 22.85, uom: "cm" }, status: "P" },
   { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "P" },
   { 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" },
   { item: "sketchbook", qty: 80, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
   { item: "sketch pad", qty: 95, size: { h: 22.85, w: 30.5, uom: "cm" }, status: "A" }
] );
```

- To use the update operators, pass to the update methods an update document of the form. Some update operators, such as $set, will create the field if the field does not exist.

```
{
  <update operator>: { <field1>: <value1>, ... },
  <update operator>: { <field2>: <value2>, ... },
  ...
}
```

Update Documents in a Collection

- Update a Single Document.
  - The following example uses the **db.collection.updateOne**() method on the inventory collection to update the first document where item equals "paper”. It uses the $set operator to update the value of the size.uom field to "cm" and the value of the status field to "P", and the $currentDate operator to update the value of the lastModified field to the current date. If lastModified field does not exist, $currentDate will create the field.
- Update Multiple Documents.
  - The following example uses the **db.collection.updateMany**() method on the inventory collection to update all documents where qty is less than 50.It uses the $set operator to update the value of the size.uom field to "in" and the value of the status field to "P", and the $currentDate operator to update the value of the lastModified field to the current date. If lastModified field does not exist, $currentDate will create the field. See $currentDate for details

```
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 }
   }
)
```

Update Documents in a Collection (Cont'd)

- Replace a Document
  - To replace the entire content of a document except for the _id field, pass an entirely new document as the second argument to db.collection.replaceOne().
  - When replacing a document, the replacement document must consist of only field/value pairs; i.e. do not include update operators expressions.
  - The replacement document can have different fields from the original document. In the replacement document, you can omit the _id field since the _id field is immutable; however, if you do include the _id field, it must have the same value as the current value.
  - The following example replaces the first document from the inventory collection where item: "paper"

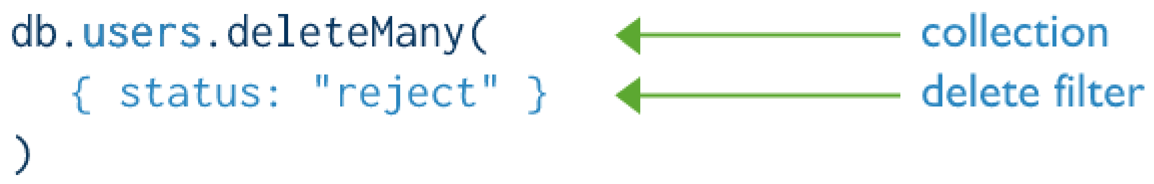
```
db.inventory.replaceOne(
   { item: "paper" },
   { item: "paper", instock: [ { warehouse: "A", qty: 60 }, { warehouse: "B", qty: 40 } ] }
)
```

Behavior of Update Operation

- Atomicity
  - All write operations in MongoDB are atomic on the level of a single document.
- _id Field
  - Once set, you cannot update the value of the _id field nor can you replace an existing document with a replacement document that has a different _id field value.
- Field Order
  - MongoDB preserves the order of the document fields following write operations.
- Upsert Option
  - If updateOne(), updateMany(), or replaceOne() includes upsert : true and no documents match the specified filter, then the operation creates a new document and inserts it. If there are matching documents, then the operation modifies or replaces the matching document or documents.
- Write Acknowledgement
  - With write concerns, you can specify the level of acknowledgement requested from MongoDB for write operations.

Delete Operations

- Delete operations remove documents from a collection. MongoDB provides the following methods to delete documents of a collection:
  - ☐ db.collection.deleteOne() New in version 3.2
  - ☐ db.collection.deleteMany() New in version 3.2
- In MongoDB, delete operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.
- You can specify criteria, or filters, that identify the documents to remove. These filters use the same syntax as read operations.



```
db.users.deleteMany(
   { status: "reject" }
)
```

Delete Documents

```
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: "P" },
   { 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" },
] );
```

- Delete All Documents that Match a Condition
  - The following example removes all documents from the inventory collection where the status field equals "A“

    ```
    db.inventory.deleteMany({ status : "A" })
    ```

- Delete Only One Document that Matches a Condition
  - The following example deletes the **first document** where status is "D“

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

- Delete Behavior
  - Indexes: Delete operations do not drop indexes, even if deleting all documents from a collection.
  - Atomicity: All write operations in MongoDB are atomic on the level of a single document.
  - Write Acknowledgement: With write concerns, you can specify the level of acknowledgement requested from MongoDB for write operations.

Bulk Write Operations

- MongoDB provides clients the ability to perform write operations in bulk. Bulk write operations affect a single collection.
- The db.collection.bulkWrite() method provides the ability to perform bulk insert, update, and remove operations. MongoDB also supports bulk insert through the db.collection.insertMany().
- bulkWrite() supports the following write operations: insertOne, updateOne, updateMany, replaceOne, deleteOne, deleteMany. Each write operation is passed to bulkWrite() as a document in an array.
- For example, the following performs multiple write operations. The characters collection contains the following documents

```
{ "_id" : 1, "char" : "Brisbane", "class" : "monk", "lvl" : 4 },
{ "_id" : 2, "char" : "Eldon", "class" : "alchemist", "lvl" : 3 },
{ "_id" : 3, "char" : "Meldane", "class" : "ranger", "lvl" : 3 }
```

Bulk Write Example

```
try { db.characters.bulkWrite(
      [  { insertOne : { "document" :  { "_id" : 4, "char" : "Dithras", "class" : "barbarian", "lvl" : 4 } }
         },
         { insertOne : { "document" :  { "_id" : 5, "char" : "Taeln", "class" : "fighter", "lvl" : 3 } }
         },
         { updateOne : { "filter" : { "char" : "Eldon" },
                           "update" : { $set : { "status" : "Critical Injury" } }
                         }
         },
         { deleteOne : { "filter" : { "char" : "Brisbane"} }
         },
         { replaceOne : { "filter" : { "char" : "Meldane" },
                            "replacement" : { "char" : "Tanys", "class" : "oracle", "lvl" : 4 }
                          }
         }
      ]
   );
}
catch (e) {
   print(e);
}
```

SQL to MongoDB Mapping Chart

SQL Terms/Concepts	MongoDB Terms/Concepts
database	database
table	collection
row	document or BSON document
column	field
index	index
table joins	`$lookup`, embedded documents
primary key Specify any unique column or column combination as primary key.	primary key In MongoDB, the primary key is automatically set to the _idfield.
aggregation (e.g. group by)	aggregation pipeline See the SQL to Aggregation Mapping Chart
transactions	transactions For many scenarios, the denormalized data model (embedded documents and arrays) will continue to be optimal for your data and use cases instead of multi-document transactions. That is, for many scenarios, modeling your data appropriately will minimize the need for multi-document transactions.

Aggregation

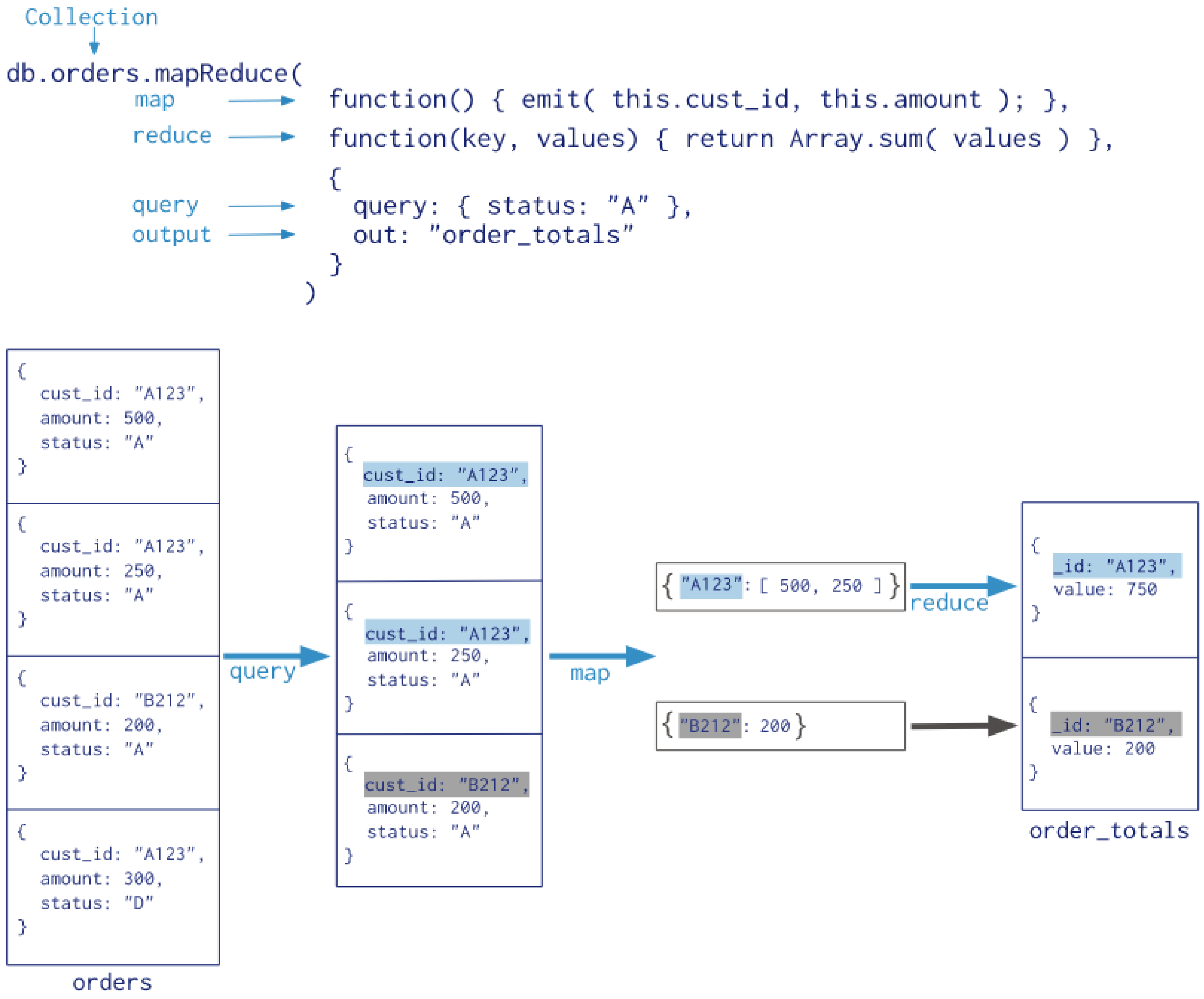
- Aggregation operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.
- Aggregation Pipeline

```
db.orders.aggregate([
   { $match: { status: "A" } },
   { $group: { _id: "$cust_id", total: { $sum: "$amount" } } }
])
```

- First Stage: The $match stage filters the documents by the status field and passes to the next stage those documents that have status equal to "A".
- Second Stage: The $group stage groups the documents by the cust_id field to calculate the sum of the amount for each unique cust_id.

Map-Reduce for Aggregation

- Map-reduce is a data processing paradigm for condensing large volumes of data into useful aggregated results



```
db.orders.mapReduce(
    function() { emit( this.cust_id, this.amount ); },
    function(key, values) { return Array.sum( values ) },
    {
      query: { status: "A" },
      out: "order_totals"
    }
)
```

orders:

```
{
   cust_id: "A123",
   amount: 500,
   status: "A"
}
{
   cust_id: "A123",
   amount: 250,
   status: "A"
}
{
   cust_id: "B212",
   amount: 200,
   status: "A"
}
{
   cust_id: "A123",
   amount: 300,
   status: "D"
}
```

query →

```
{
   cust_id: "A123",
   amount: 500,
   status: "A"
}
{
   cust_id: "A123",
   amount: 250,
   status: "A"
}
{
   cust_id: "B212",
   amount: 200,
   status: "A"
}
```

map →

```
{ "A123": [ 500, 250 ] }
{ "B212": 200 }
```

reduce →

order_totals:

```
{
   _id: "A123",
   value: 750
}
{
   _id: "B212",
   value: 200
}
```

Read Concern and Write Acknowledgement

- The readConcern option allows you to control the consistency and isolation properties of the data read from replica sets and replica set shards
- Write concern describes the level of acknowledgment requested from MongoDB for write operations to a standalone mongod or to replica sets or to sharded clusters.
- Through the effective use of write concerns and read concerns, you can adjust the level of consistency and availability guarantees as appropriate, such as waiting for stronger consistency guarantees, or loosening consistency requirements to provide higher availability
- For multi-document transactions, you set the read or write concern at the transaction level, not at the individual operation level.

Other Features of MongoDB

- Flexible Schema
  - Schema validation using JSON schema is also provided but not compulsory
- Atomicity
  - Single Document Atomicity
  - Multi-Document Transactions
- Indexes: default _id Index, single field index, compound index, multikey index for arrays, text index, hashed index
- Security: authentication, authorization, TLS/SSL transport encryption
- Replication
  - Redundancy for data availability
  - Primary and secondary replicas
- Sharding:
  - A method for distributing data across multiple machines
  - Sharded clusters and zones