MongoDB NoSQL Database

MongoDB is a document database designed for ease of development and scaling. The Manual introduces key concepts in MongoDB, presents the query language, and provides operational and administrative considerations and procedures as well as a comprehensive reference section.

MongoDB offers both a *Community* and an *Enterprise* version of the database:

- MongoDB Community is the source available and free to use edition of MongoDB.
- MongoDB Enterprise is available as part of the MongoDB Enterprise Advanced subscription and includes comprehensive support for your MongoDB deployment. MongoDB Enterprise also adds enterprise-focused features such as LDAP and Kerberos support, on-disk encryption, and auditing.

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 are:

- 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.

Collections/Views/On-Demand Materialized Views

MongoDB stores documents in collections. Collections are analogous to tables in relational databases.

In addition to collections, MongoDB supports:

- Read-only Views (Starting in MongoDB 3.4)
- On-Demand Materialized Views (Starting in MongoDB 4.2).

Key Features

- High Performance

MongoDB provides high performance data persistence. In particular,

- 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

MongoDB supports a rich query language to support read and write operations (CRUD) as well as:

- Data Aggregation
- Text Search and Geospatial Queries.

- **High Availability**

MongoDB's replication facility, called replica set, provides:

- *automatic* failover
- data redundancy.

A replica set is a group of MongoDB servers that maintain the same data set, providing redundancy and increasing data availability.

- **Horizontal Scalability**

MongoDB provides horizontal scalability as part of its *core* functionality:

- Sharding distributes data across a cluster of machines.

Define Your Data Set

When setting up a data store, your first task is to answer the question: “What data would I like to store and how do the fields relate to each other?”.

This guide uses a hypothetical inventory database to track items and their quantities, sizes, tags, and ratings.

Here is an example of the types of fields you might wish to capture:

name	**quantity**	**Size**	**status**	**tags**	**rating**
journal	25	14x21,cm	A	brown, lined	9
notebook	50	8.5x11,in	A	college-ruled,perforated	8
paper	100	8.5x11,in	D	watercolor	10
planner	75	22.85x30,cm	D	2019	10
postcard	45	10x,cm	D	double-sided,white	2

Start Thinking in JSON

While a table might seem like a good place to store data, as you can see from the example above, there are fields in this data set that require multiple values and would not be easy to search or display if modeled in a single column (for example – size and tags).

In a SQL database you might solve this problem by creating a relational table.

In MongoDB, data is stored as documents. These documents are stored in MongoDB in JSON (JavaScript Object Notation) format. JSON documents support embedded

fields, so related data and lists of data can be stored with the document instead of an external table.

JSON is formatted as name/value pairs. In JSON documents, fieldnames and values are separated by a colon, fieldname and value pairs are separated by commas, and sets of fields are encapsulated in “curly braces” ({}).

If you wanted to begin to model one of the rows above, for example this one:

name	**Quantity**	**Size**	**status**	**tags**	**rating**
notebook	50	8.5x11,in	A	college-ruled,perforated	8

You might start with the name and quantity fields. In JSON these fields would look like:

{"name": "notebook", "qty": 50}

Identify Candidates for Embedded Data and Model Your Data

Next you will decide which fields require multiple values. These fields will be candidates for embedded documents or lists/arrays of embedded documents within the document.

For example, in the data above, size might consist of three fields:

```
{ "h": 11, "w": 8.5, "uom": "in" }
```

And some items have multiple ratings, so ratings might be represented as a list of documents containing the field scores:

```
[ { "score": 8 }, { "score": 9 } ]
```

And you might need to handle multiple tags per item. So you might store them in a list too.

```
[ "college-ruled", "perforated" ]
```

Finally, a JSON document that stores an inventory item might look like this:

```
{
 "name": "notebook",
 "qty": 50,
 "rating": [ { "score": 8 }, { "score": 9 } ],
 "size": { "height": 11, "width": 8.5, "unit": "in" },
 "status": "A",
 "tags": [ "college-ruled", "perforated"]
}
```

MongoDB stores data records as documents (specifically BSON documents) which are gathered together in collections. A database stores one or more collections of documents.

Databases

In MongoDB, databases hold one or more collections of documents. To select a database to use, in the mongo shell, issue the use <db> statement, as in the following example:

use myDB

Create a Database

If a database does not exist, MongoDB creates the database when you first store data for that database. As such, you can switch to a non-existent database and perform the following operation in the mongo shell:

use myNewDB

db.myNewCollection1.insertOne({ x: 1 })

The insertOne() operation creates both the database myNewDB and the collection myNewCollection1 if they do not already exist. Be sure that both the database and collection names follow MongoDB Naming Restrictions.

Collections

MongoDB stores documents in collections. Collections are analogous to tables in relational databases.

Create a Collection

If a collection does not exist, MongoDB creates the collection when you first store data for that collection.

db.myNewCollection2.insertOne({ x: 1 })

db.myNewCollection3.createIndex({ y: 1 })

Both the insertOne() and the createIndex() operations create their respective collection if they do not already exist. Be sure that the collection name follows MongoDB Naming Restrictions.

Explicit Creation

MongoDB provides the db.createCollection() method to explicitly create a collection with various options, such as setting the maximum size or the documentation validation rules. If you are not specifying these options, you do not need to explicitly create the collection since MongoDB creates new collections when you first store data for the collections.

Document Validation

New in version 3.2.

By default, a collection does not require its documents to have the same schema; i.e. the documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection.

Starting in MongoDB 3.2, however, you can enforce document validation rules for a collection during update and insert operations.

MongoDB CRUD Operations

CRUD operations *create*, *read*, *update*, and *delete* documents.

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.

e.g.

```
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" },
] );
```

Read Operations

Read operations retrieve documents from a collection; i.e. query 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.

Select All Documents in a Collection

To select all documents in the collection, pass an empty document as the query filter parameter to the find method. The query filter parameter determines the select criteria:

copy

copied

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

This operation corresponds to the following SQL statement:

```
SELECT * FROM inventory
```

Specify Equality Condition

To specify equality conditions, use <field>:<value> expressions in the query filter document:

```
{ <field1>: <value1>, ... }
```

The following example selects from the inventory collection all documents where the status equals "D":

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

This operation corresponds to the following SQL statement:

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

Specify Conditions Using Query Operators

A query filter document can use the query operators to specify conditions in the following form:

```
{ <field1>: { <operator1>: <value1> }, ... }
```

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

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

NOTE
Although you can express this query using the $or operator, use the $in operator rather than the $or operator when performing equality checks on the same field.

The operation corresponds to the following SQL statement:

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

Refer to the Query and Projection Operators document for the complete list of MongoDB query operators.

Specify AND Conditions

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 } } )
```

The operation corresponds to the following SQL statement:

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

See comparison operators for other MongoDB comparison operators.

Specify OR Conditions

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 } } ] } )
```

The operation corresponds to the following SQL statement:

copy

copied

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

NOTE
Queries which use comparison operators are subject to Type Bracketing.

Specify AND as well as OR Conditions

In the following example, the compound query document selects all documents in the collection where the status equals "A" **and** *either* qty is less than ($lt) 30 *or* item starts with the character p:

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

The operation corresponds to the following SQL statement:

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

NOTE
MongoDB supports regular expressions $regex queries to perform string pattern matches.

RDBMS Where Clause Equivalents in MongoDB

To query the document on the basis of some condition, you can use following operations.

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:{$eg;<value>}}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'xyz'
Less Than	{<key>:{$lt:<value>}}	db.mycol.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>:{$lte:<value>}}	db.mycol.find({"likes":{$lte:50}}).pretty()	where likes <= 50
Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{<key>:{$gte:<value>}}	db.mycol.find({"likes":{$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>:{$ne:<value>}}	db.mycol.find({"likes":{$ne:50}}).pretty()	where likes != 50
Values in an array	{<key>:{$in:[<value1>, <value2>,……<valueN>]}}	db.mycol.find({"name":{$in:["Raj", "Ram", "Raghu"]}}).pretty()	Where name matches any of the value in

			:["Raj", "Ram", "Raghu"]
Values not in an array	{<key>:{$nin:<value>}}	db.mycol.find({"name":{$nin:["Ramu", "Raghav"]}}).pretty()	Where name values is not in the array :["Ramu", "Raghav"] or, doesn't exist at all

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.

MongoDB's **update()** and **save()** methods are used to update document into a collection.

The update() method update values in the existing document while the save() method replaces the existing document with the document passed in save() method.

MongoDB Update() method

The update() method updates values in the existing document.

Syntax:

Basic syntax of **update()** method is as follows

>db.COLLECTION_NAME.update(SELECTIOIN_CRITERIA, UPDATED_DATA)

Example

Consider the mycol collectioin has following data.

{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}

{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}

{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}

Following example will set the new title 'New MongoDB Tutorial' of the documents whose title is 'MongoDB Overview'

>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB Tutorial'}}) >db.mycol.find()

{ "_id" : ObjectId(5983548781331adf45ec5), "title":"New MongoDB Tutorial"}

{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}

{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}

>

MongoDB Save() Method

The **save()** method replaces the existing document with the new document passed in save() method Syntax

Basic syntax of mongodb **save()** method is shown below:

>db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})

Example

Following example will replace the document with the _id

'5983548781331adf45ec7' >db.mycol.save(

{

"_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point New Topic", "by":"Tutorials Point"

}

)

>db.mycol.find()

{ "_id" : ObjectId(5983548781331adf45ec5), "title":"Tutorials Point New Topic", "by":"Tutorials Point"}

{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}

{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials

Point Overview"} >

. MongoDB Delete Document

The remove() Method

MongoDB's **remove()** method is used to remove document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag

1. **deletion criteria :** (Optional) deletion criteria according to documents will be removed.
2. **justOne :** (Optional) if set to true or 1, then remove only one document.

Syntax:

Basic syntax of **remove()** method is as follows

```
>db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)
```

Example

Consider the mycol collectioin has following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

Following example will remove all the documents whose title is 'MongoDB Overview'

```
>db.mycol.remove({'title':'MongoDB Overview'})
>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
>
```

Remove only one

if there are multiple records and you want to delete only first record, then set **justOne** parameter in **remove()** method

```
>db.COLLECTION_NAME.remove(DELETION_CRITERIA,1)
```

Remove All documents

If you don't specify deletion criteria, then mongodb will delete whole documents from the collection. **This is equivalent of SQL's truncate command.**

>db.mycol.remove()

>db.mycol.find()

MongoDB Projection

In mongodb projection meaning is selecting only necessary data rather than selecting whole of the data of a document. If a document has 5 fields and you need to show only 3, then select only 3 fields from them.

The find() Method

MongoDB's **find()** method, explained in previously accepts second optional parameter that is list of fields that you want to retrieve. In MongoDB when you execute **find()** method, then it displays all fields of a document. To limit this you need to set list of fields with value 1 or 0. 1 is used to show the field while 0 is used to hide the field.

Syntax:

Basic syntax of **find()** method with projection is as follows

>db.COLLECTION_NAME.find({},{KEY:1})

Example

Consider the collection myycol has the following data

{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}

{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}

{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}

Following example will display the title of the document while quering the document.

>db.mycol.find({},{"title":1,_id:0})

{"title":"MongoDB Overview"}

{"title":"NoSQL Overview"}

{"title":"Tutorials Point Overview"}

>

Please note **_id** field is always displayed while executing **find()** method, if you don't want this field, then you need to set it as 0

>

MongoDB Sort Documents The sort() Method

To sort documents in MongoDB, you need to use sort() method. sort() method accepts a document containing list of fields along with their sorting order. To specify sorting order 1 and -1 are used. 1 is used for ascending order while -1 is used for descending order.

Syntax:

Basic syntax of sort() method is as follows

```
>db.COLLECTION_NAME.find().sort({KEY:1})
```

Example

Consider the collection myycol has the following data

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}

{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}

{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

Following example will display the documents sorted by title in descending order.

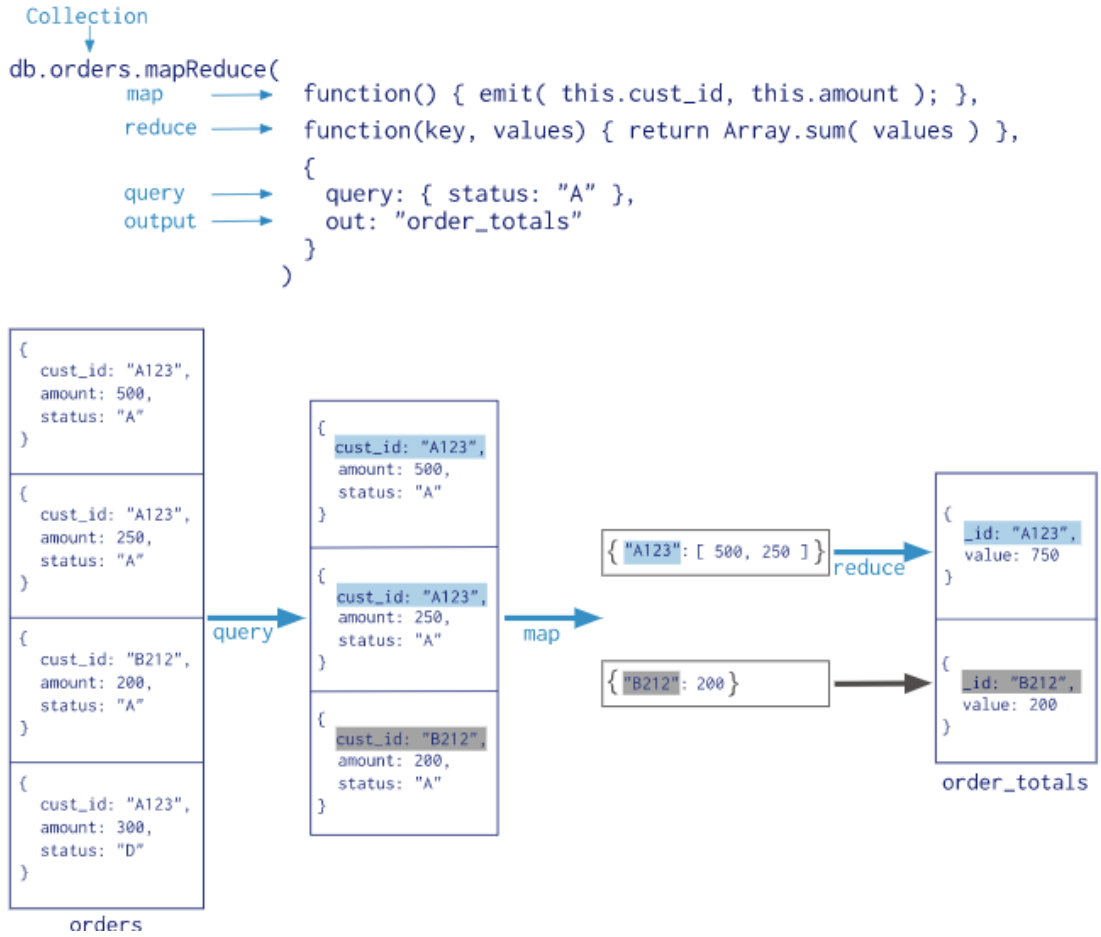
```
>db.mycol.find({},{"title":1,_id:0}).sort({"title":-1})

{"title":"Tutorials Point Overview"}

{"title":"NoSQL Overview"}

{"title":"MongoDB Overview"}

>
```

Please note if you don't specify the sorting preference, then sort() method will display documents in ascending order.

Map Reduce

Map-reduce is a data processing paradigm for condensing large volumes of data into useful *aggregated* results. To perform map-reduce operations, MongoDB provides the mapReduce database command.

Consider the following map-reduce operation:



In this map-reduce operation, MongoDB applies the *map* phase to each input document (i.e. the documents in the collection that match the query condition). The map function emits key-value pairs. For those keys that have multiple values, MongoDB applies the *reduce* phase, which collects and condenses the aggregated data. MongoDB then stores the results in a collection. Optionally, the output of the reduce function may pass through a *finalize* function to further condense or process the results of the aggregation.

All map-reduce functions in MongoDB are JavaScript and run within the mongod process. Map-reduce operations take the documents of a single collection as the *input* and can perform any arbitrary sorting and limiting before beginning the map stage. mapReduce can return the results of a map-reduce operation as a document, or may write the results to collections.

Consider the below orders dataset

```
db.orders.insertMany([

   { _id: 1, cust_id: "Ant O. Knee", ord_date: new Date("2020-03-01"), price: 25, items: [ { sku: "oranges", qty: 5, price: 2.5 }, { sku: "apples", qty: 5, price: 2.5 } ], status: "A" },

   { _id: 2, cust_id: "Ant O. Knee", ord_date: new Date("2020-03-08"), price: 70, items: [ { sku: "oranges", qty: 8, price: 2.5 }, { sku: "chocolates", qty: 5, price: 10 } ], status: "A" },

   { _id: 3, cust_id: "Busby Bee", ord_date: new Date("2020-03-08"), price: 50, items: [ { sku: "oranges", qty: 10, price: 2.5 }, { sku: "pears", qty: 10, price: 2.5 } ], status: "A" },

   { _id: 4, cust_id: "Busby Bee", ord_date: new Date("2020-03-18"), price: 25, items: [ { sku: "oranges", qty: 10, price: 2.5 } ], status: "A" },

   { _id: 5, cust_id: "Busby Bee", ord_date: new Date("2020-03-19"), price: 50, items: [ { sku: "chocolates", qty: 5, price: 10 } ], status: "A"},

   { _id: 6, cust_id: "Cam Elot", ord_date: new Date("2020-03-19"), price: 35, items: [ { sku: "carrots", qty: 10, price: 1.0 }, { sku: "apples", qty: 10, price: 2.5 } ], status: "A" },

   { _id: 7, cust_id: "Cam Elot", ord_date: new Date("2020-03-20"), price: 25, items: [ { sku: "oranges", qty: 10, price: 2.5 } ], status: "A" },

   { _id: 8, cust_id: "Don Quis", ord_date: new Date("2020-03-20"), price: 75, items: [ { sku: "chocolates", qty: 5, price: 10 }, { sku: "apples", qty: 10, price: 2.5 } ], status: "A" },

   { _id: 9, cust_id: "Don Quis", ord_date: new Date("2020-03-20"), price: 55, items: [ { sku: "carrots", qty: 5, price: 1.0 }, { sku: "apples", qty: 10, price: 2.5 }, { sku: "oranges", qty: 10, price: 2.5 } ], status: "A" },

   { _id: 10, cust_id: "Don Quis", ord_date: new Date("2020-03-23"), price: 25, items: [ { sku: "oranges", qty: 10, price: 2.5 } ], status: "A" }

])
```

Return the Total Price Per Customer

Perform the map-reduce operation on the orders collection to group by the cust_id, and calculate the sum of the price for each cust_id:

1. Define the map function to process each input document:

- In the function, this refers to the document that the map-reduce operation is processing.
- The function maps the price to the cust_id for each document and emits the cust_id and price pair.

```
var mapFunction1 = function() {
  emit(this.cust_id, this.price);
};
```

2. Define the corresponding reduce function with two arguments keyCustId and valuesPrices:

- The valuesPrices is an array whose elements are the price values emitted by the map function and grouped by keyCustId.
- The function reduces the valuesPrice array to the sum of its elements.

```
var reduceFunction1 = function(keyCustId, valuesPrices) {
  return Array.sum(valuesPrices);
};
```

3. Perform map-reduce on all documents in the orders collection using the mapFunction1 map function and the reduceFunction1 reduce function.

```
db.orders.mapReduce(
  mapFunction1,
  reduceFunction1,
  { out: "map_reduce_example" }
)
```

This operation outputs the results to a collection named map_reduce_example. If the map_reduce_example collection already exists, the operation will replace the contents with the results of this map-reduce operation.

4. Query the map_reduce_example collection to verify the results:
   db.map_reduce_example.find().sort({ _id: 1 })

The operation returns the following documents:

```
{ "_id" : "Ant O. Knee", "value" : 95 }
{ "_id" : "Busby Bee", "value" : 125 }
{ "_id" : "Cam Elot", "value" : 60 }
{ "_id" : "Don Quis", "value" : 155 }
```