mapReduce

The `mapReduce` command allows you to run map-reduce aggregation operations over a collection. The `mapReduce` command has the following prototype form:

```
db.runCommand(
               {
                 mapReduce: <collection>,
                 map: <function>,
                 reduce: <function>,
                 finalize: <function>,
                 out: <output>,
                 query: <document>,
                 sort: <document>,
                 limit: <number>,
                 scope: <document>,
                 jsMode: <boolean>,
                 verbose: <boolean>
               }
             )
```

Pass the name of the collection to the `mapReduce` command (i.e. `<collection>`) to use as the source documents to perform the map reduce operation. The command also accepts the following parameters:

Field	Type	Description
`mapReduce`	collection	The name of the collection on which you want to perform map-reduce. This collection will be filtered using `query` before being processed by the `map` function.
`map`	Javascript function	A JavaScript function that associates or "maps" a `value` with a `key` and emits the `key` and value `pair`. See Requirements for the map Function for more information.
`reduce`	JavaScript function	A JavaScript function that "reduces" to a single object all the `values` associated with a particular `key`. See Requirements for the reduce Function for more information.
`out`	string or document	Specifies where to output the result of the map-reduce operation. You can either output to a collection or return the result inline. On a primary member of a replica set you can output either to a collection or inline, but on a secondary, only inline output is possible. See out Options for more information.

`query`	document	Optional. Specifies the selection criteria using query operators for determining the documents input to the `map` function.
`sort`	document	Optional. Sorts the *input* documents. This option is useful for optimization. For example, specify the sort key to be the same as the emit key so that there are fewer reduce operations. The sort key must be in an existing index for this collection.
`limit`	number	Optional. Specifies a maximum number of documents for the input into the `map` function.
`finalize`	Javascript function	Optional. Follows the `reduce` method and modifies the output. See Requirements for the finalize Function for more information.
`scope`	document	Optional. Specifies global variables that are accessible in the `map`, `reduce` and `finalize` functions.
`jsMode`	Boolean	Optional. Specifies whether to convert intermediate data into BSON format between the execution of the `map` and `reduce` functions. Defaults to `false`. If `false`: • Internally, MongoDB converts the JavaScript objects emitted by the `map` function to BSON objects. These BSON objects are then converted back to JavaScript objects when calling the `reduce` function. • The map-reduce operation places the intermediate BSON objects in temporary, on-disk storage. This allows the map-reduce operation to execute over arbitrarily large data sets. If `true`: • Internally, the JavaScript objects emitted during `map` function remain as JavaScript objects. There is no need to convert the objects for the `reduce` function, which can result in faster execution. • You can only use `jsMode` for result sets with fewer than 500,000 distinct `key` arguments to the mapper's `emit()` function. The `jsMode` defaults to false.
`verbose`	Boolean	Optional. Specifies whether to include the `timing` information in the result information. The `verbose` defaults to `true` to include the `timing` information.

The following is a prototype usage of the `mapReduce` command:

```
var mapFunction = function() { ... };
var reduceFunction = function(key, values) { ... };

db.runCommand(
               {
                 mapReduce: <input-collection>,
                 map: mapFunction,
                 reduce: reduceFunction,
                 out: { merge: <output-collection> },
                 query: <query>
               }
             )
```

JAVASCRIPT IN MONGODB:

Although `mapReduce` uses JavaScript, most interactions with MongoDB do not use JavaScript but use an idiomatic driver in the language of the interacting application.

NOTE:

Changed in version 2.4.

In MongoDB 2.4, `map-reduce operations`, the `group` command, and `$where` operator expressions **cannot** access certain global functions or properties, such as `db`, that are available in the `mongo` shell.

When upgrading to MongoDB 2.4, you will need to refactor your code if your `map-reduce operations`, `group` commands, or `$where` operator expressions include any global shell functions or properties that are no longer available, such as `db`.

The following JavaScript functions and properties **are available** to `map-reduce operations`, the `group` command, and `$where` operator expressions in MongoDB 2.4:

Available Properties	Available Functions	
`args`	`assert()`	`Map()`
`MaxKey`	`BinData()`	`MD5()`
`MinKey`	`DBPointer()`	`NumberInt()`
	`DBRef()`	`NumberLong()`
	`doassert()`	`ObjectId()`
	`emit()`	`print()`
	`gc()`	`printjson()`
	`HexData()`	`printjsononeline()`
	`hex_md5()`	`sleep()`
	`isNumber()`	`Timestamp()`
	`isObject()`	`tojson()`
	`ISODate()`	`tojsononeline()`
	`isString()`	`tojsonObject()`
		`UUID()`
		`version()`

Requirements for the `map` Function

The `map` function is responsible for transforming each input document into zero or more documents. It can access the variables defined in the `scope` parameter, and has the following prototype:

```
function() {
   ...
   emit(key, value);
}
```

The `map` function has the following requirements:

- In the `map` function, reference the current document as `this` within the function.
- The `map` function should *not* access the database for any reason.
- The `map` function should be pure, or have *no* impact outside of the function (i.e. side effects.)
- A single emit can only hold half of MongoDB's maximum BSON document size.
- The `map` function may optionally call `emit(key,value)` any number of times to create an output document associating `key` with `value`.

The following `map` function will call `emit(key,value)` either 0 or 1 times depending on the value of the input document's `status` field:

```
function() {
    if (this.status == 'A')
        emit(this.cust_id, 1);
}
```

The following `map` function may call `emit(key,value)` multiple times depending on the number of elements in the input document's `items` field:

```
function() {
    this.items.forEach(function(item){ emit(item.sku, 1); });
}
```

Requirements for the `reduce` Function

The `reduce` function has the following prototype:

```
function(key, values) {
   ...
   return result;
}
```

The `reduce` function exhibits the following behaviors:

- The `reduce` function should *not* access the database, even to perform read operations.
- The `reduce` function should *not* affect the outside system.
- MongoDB will **not** call the `reduce` function for a key that has only a single value. The `values` argument is an array whose elements are the `value` objects that are "mapped" to the `key`.
- MongoDB can invoke the `reduce` function more than once for the same key. In this case, the previous output from the `reduce` function for that key will become one of the input values to the next `reduce` function invocation for that key.
- The `reduce` function can access the variables defined in the `scope` parameter.
- The inputs to `reduce` must not be larger than half of MongoDB's maximum BSON document size. This requirement may be violated when large documents are returned and then joined together in subsequent `reduce` steps.

Because it is possible to invoke the `reduce` function more than once for the same key, the following properties need to be true:

- the *type* of the return object must be **identical** to the type of the `value` emitted by the `map` function.
- the `reduce` function must be *associative*. The following statement must be true:

```
reduce(key, [ C, reduce(key, [ A, B ]) ] ) == reduce( key, [ C, A, B ] )
```

- the `reduce` function must be *idempotent*. Ensure that the following statement is true:

```
reduce( key, [ reduce(key, valuesArray) ] ) == reduce( key, valuesArray )
```

- the `reduce` function should be *commutative*: that is, the order of the elements in the `valuesArray` should not affect the output of the `reduce` function, so that the following statement is true:

```
reduce( key, [ A, B ] ) == reduce( key, [ B, A ] )
```

Requirements for the `finalize` Function

The `finalize` function has the following prototype:

```
function(key, reducedValue) {
   ...
   return modifiedObject;
}
```

The `finalize` function receives as its arguments a `key` value and the `reducedValue` from the `reduce` function. Be aware that:

- The `finalize` function should *not* access the database for any reason.
- The `finalize` function should be pure, or have *no* impact outside of the function (i.e. side effects.)
- The `finalize` function can access the variables defined in the `scope` parameter.

`out` Options

You can specify the following options for the `out` parameter:

Output to a Collection

This option outputs to a new collection, and is not available on secondary members of replica sets.

```
out: <collectionName>
```

Output to a Collection with an Action

This option is only available when passing a collection that already exists to `out`. It is not available on secondary members of replica sets.

```
out: { <action>: <collectionName>
        [, db: <dbName>]
        [, sharded: <boolean> ]
        [, nonAtomic: <boolean> ] }
```

When you output to a collection with an action, the `out` has the following parameters:

- `<action>`: Specify one of the following actions:
    - `replace`
      Replace the contents of the `<collectionName>` if the collection with the `<collectionName>` exists.
    - `merge`
      Merge the new result with the existing result if the output collection already exists. If an existing document has the same key as the new result, *overwrite* that existing document.
    - `reduce`
      Merge the new result with the existing result if the output collection already exists. If an existing document has the same key as the new result, apply the `reduce` function to both the new and the existing documents and overwrite the existing document with the result.
- `db`:
  Optional. The name of the database that you want the map-reduce operation to write its output. By default this will be the same database as the input collection.
- `sharded`:
  Optional. If `true` *and* you have enabled sharding on output database, the map-reduce operation will shard the output collection using the `_id` field as the shard key.
- `nonAtomic`:

  New in version 2.2.

Optional. Specify output operation as non-atomic. This applies **only** to the `merge` and `reduce` output modes, which may take minutes to execute.
By default `nonAtomic` is `false`, and the map-reduce operation locks the database during post-processing.
If `nonAtomic` is `true`, the post-processing step prevents MongoDB from locking the database: during this time, other clients will be able to read intermediate states of the output collection.

Output Inline

Perform the map-reduce operation in memory and return the result. This option is the only available option for `out` on secondary members of replica sets.

```
out: { inline: 1 }
```

The result must fit within the maximum size of a BSON document.

Map-Reduce Examples

In the `mongo` shell, the `db.collection.mapReduce()` method is a wrapper around the `mapReduce` command. The following examples use the `db.collection.mapReduce()` method:

Consider the following map-reduce operations on a collection `orders` that contains documents of the following prototype:

```
{
     _id: ObjectId("50a8240b927d5d8b5891743c"),
     cust_id: "abc123",
     ord_date: new Date("Oct 04, 2012"),
     status: 'A',
     price: 25,
     items: [ { sku: "mmm", qty: 5, price: 2.5 },
              { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

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 the 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:

Calculate Order and Total Quantity with Average Quantity Per Item

In this example, you will perform a map-reduce operation on the `orders` collection for all documents that have an `ord_date` value greater than `01/01/2012`. The operation groups by the `item.sku` field, and calculates the number of orders and the total quantity ordered for each `sku`. The operation concludes by calculating the average quantity per order for each `sku` value:

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.
   - For each item, the function associates the `sku` with a new object `value` that contains the `count` of `1` and the item `qty` for the order and emits the `sku` and `value` pair.

```
var mapFunction2 = function() {
                       for (var idx = 0; idx < this.items.length; idx++) {
                           var key = this.items[idx].sku;
                           var value = {
                                         count: 1,
                                         qty: this.items[idx].qty
                                       };
                           emit(key, value);
                       }
                    };
```

2. Define the corresponding reduce function with two arguments `keySKU` and `countObjVals`:
   - `countObjVals` is an array whose elements are the objects mapped to the grouped `keySKU` values passed by map function to the reducer function.
   - The function reduces the `countObjVals` array to a single object `reducedValue` that contains the `count` and the `qty` fields.
   - In `reducedVal`, the `count` field contains the sum of the `count` fields from the individual array elements, and the `qty` field contains the sum of the `qty` fields from the individual array elements.

```
var reduceFunction2 = function(keySKU, countObjVals) {
                     reducedVal = { count: 0, qty: 0 };

                     for (var idx = 0; idx < countObjVals.length; idx++) {
                         reducedVal.count += countObjVals[idx].count;
                         reducedVal.qty += countObjVals[idx].qty;
                     }

                     return reducedVal;
                  };
```

3. Define a finalize function with two arguments `key` and `reducedVal`. The function modifies the `reducedVal` object to add a computed field named `avg` and returns the modified object:

```
var finalizeFunction2 = function (key, reducedVal) {

                       reducedVal.avg = reducedVal.qty/reducedVal.count;

                       return reducedVal;

                    };
```

4. Perform the map-reduce operation on the `orders` collection using the `mapFunction2`, `reduceFunction2`, and `finalizeFunction2` functions.

```
db.orders.mapReduce( mapFunction2,
                     reduceFunction2,
                     {
                       out: { merge: "map_reduce_example" },
                       query: { ord_date:
                                  { $gt: new Date('01/01/2012') }
                              },
                       finalize: finalizeFunction2
                     }
                   )
```

This operation uses the `query` field to select only those documents with `ord_date` greater than `new Date(01/01/2012)`. Then it output the results to a collection `map_reduce_example`. If the `map_reduce_example` collection already exists, the operation will merge the existing contents with the results of this map-reduce operation.

For more information and examples, see the Map-Reduce page and Perform Incremental Map-Reduce.

Output

If you set the out parameter to write the results to a collection, the `mapReduce` command returns a document in the following form:

```
{
    "result" : <string or document>,
    "timeMillis" : <int>,
    "counts" : {
        "input" : <int>,
        "emit" : <int>,
        "reduce" : <int>,
        "output" : <int>
    },
    "ok" : <int>,
}
```

If you set the out parameter to output the results inline, the `mapReduce` command returns a document in the following form:

```
{
    "results" : [
       {
          "_id" : <key>,
          "value" :<reduced or finalizedValue for key>
       },
       ...
    ],
    "timeMillis" : <int>,
    "counts" : {
       "input" : <int>,
       "emit" : <int>,
       "reduce" : <int>,
       "output" : <int>
    },
    "ok" : <int>
}
```

`mapReduce.result`

For output sent to a collection, this value is either:

- a string for the collection name if out did not specify the database name, or
- a document with both `db` and `collection` fields if out specified both a database and collection name.

`mapReduce.results`

For output written inline, an array of resulting documents. Each resulting document contains two fields:

- `_id` field contains the `key` value,
- `value` field contains the reduced or finalized value for the associated `key`.

`mapReduce.timeMillis`

The command execution time in milliseconds.

`mapReduce.counts`

Various count statistics from the `mapReduce` command.

`mapReduce.counts.input`

The number of documents the `mapReduce` command called the `map` function.

`mapReduce.counts.emit`

The number of times the `mapReduce` command called the `emit` function.

`mapReduce.counts.reduce`

The number of times the `mapReduce` command called the `reduce` function.

`mapReduce.counts.output`

The number of output values produced.

`mapReduce.ok`

A value of `1` indicates the `mapReduce` command ran successfully. A value of `0` indicates an error.

Additional Information

- Troubleshoot the Map Function
- Troubleshoot the Reduce Function
- `db.collection.mapReduce()`
- Aggregation Concepts

Was this page helpful? Yes No