MongoDB Aggregation and Data Processing

Release 2.6.3

MongoDB Documentation Project

July 24, 2014

Contents

Aggregations 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. MongoDB provides three ways to perform aggregation: the *aggregation pipeline* (page 7), the *map-reduce function* (page 10), and *single purpose aggregation methods and commands* (page 11).

Aggregation Introduction **(page 3)** A high-level introduction to aggregation.

Aggregation Concepts **(page 7)** Introduces the use and operation of the data aggregation modalities available in MongoDB.

- ***Aggregation Pipeline*** **(page 7)** The aggregation pipeline is a framework for performing aggregation tasks, modeled on the concept of data processing pipelines. Using this framework, MongoDB passes the documents of a single collection through a pipeline. The pipeline transforms the documents into aggregated results, and is accessed through the `aggregate` database command.

- ***Map-Reduce*** **(page 10)** Map-reduce is a generic multi-phase data aggregation modality for processing quantities of data. MongoDB provides map-reduce with the `mapReduce` database command.

- ***Single Purpose Aggregation Operations*** **(page 11)** MongoDB provides a collection of specific data aggregation operations to support a number of common data aggregation functions. These operations include returning counts of documents, distinct values of a field, and simple grouping operations.

1 Aggregation Introduction

Aggregations are operations that process data records and return computed results. MongoDB provides a rich set of aggregation operations that examine and perform calculations on the data sets. Running data aggregation on the `mongod` instance simplifies application code and limits resource requirements.

Like queries, aggregation operations in MongoDB use *collections* of documents as an input and return results in the form of one or more documents.

1.1 Aggregation Modalities

Aggregation Pipelines

MongoDB 2.2 introduced a new *aggregation framework* (page 7), modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated result.

The most basic pipeline stages provide *filters* that operate like queries and *document transformations* that modify the form of the output document.

Other pipeline operations provide tools for grouping and sorting documents by specific field or fields as well as tools for aggregating the contents of arrays, including arrays of documents. In addition, pipeline stages can use *operators* for tasks such as calculating the average or concatenating a string.

The pipeline provides efficient data aggregation using native operations within MongoDB, and is the preferred method for data aggregation in MongoDB.

Map-Reduce

MongoDB also provides *map-reduce* (page 10) operations to perform aggregation. In general, map-reduce operations have two phases: a *map* stage that processes each document and *emits* one or more objects for each input document, and *reduce* phase that combines the output of the map operation. Optionally, map-reduce can have a *finalize* stage to make final modifications to the result. Like other aggregation operations, map-reduce can specify a query condition to select the input documents as well as sort and limit the results.

Map-reduce uses custom JavaScript functions to perform the map and reduce operations, as well as the optional *finalize* operation. While the custom JavaScript provide great flexibility compared to the aggregation pipeline, in general, map-reduce is less efficient and more complex than the aggregation pipeline.

Note: Starting in MongoDB 2.4, certain `mongo` shell functions and properties are inaccessible in map-reduce operations. MongoDB 2.4 also provides support for multiple JavaScript operations to run at the same time. Before MongoDB 2.4, JavaScript code executed in a single thread, raising concurrency issues for map-reduce.

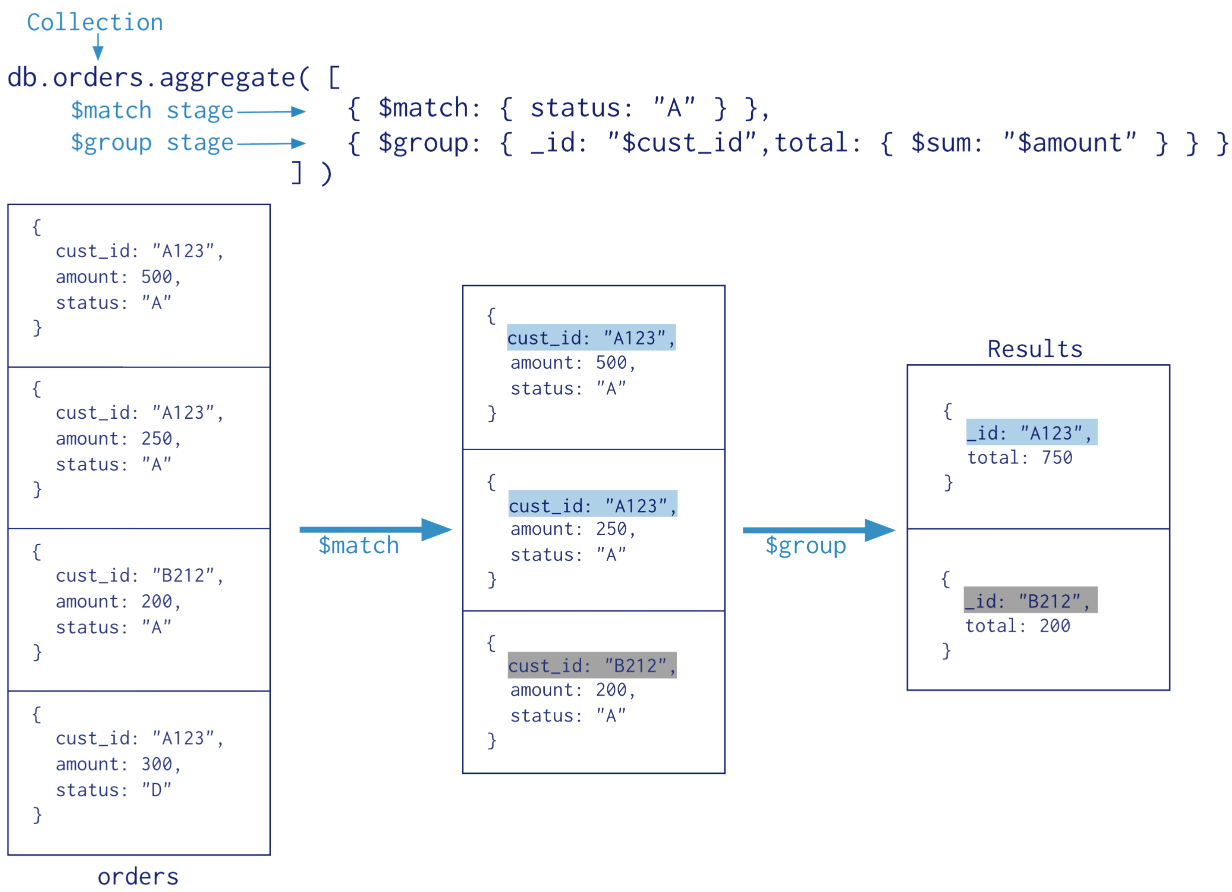

Figure 1: Diagram of the annotated aggregation pipeline operation. The aggregation pipeline has two stages: `$match` and `$group`.

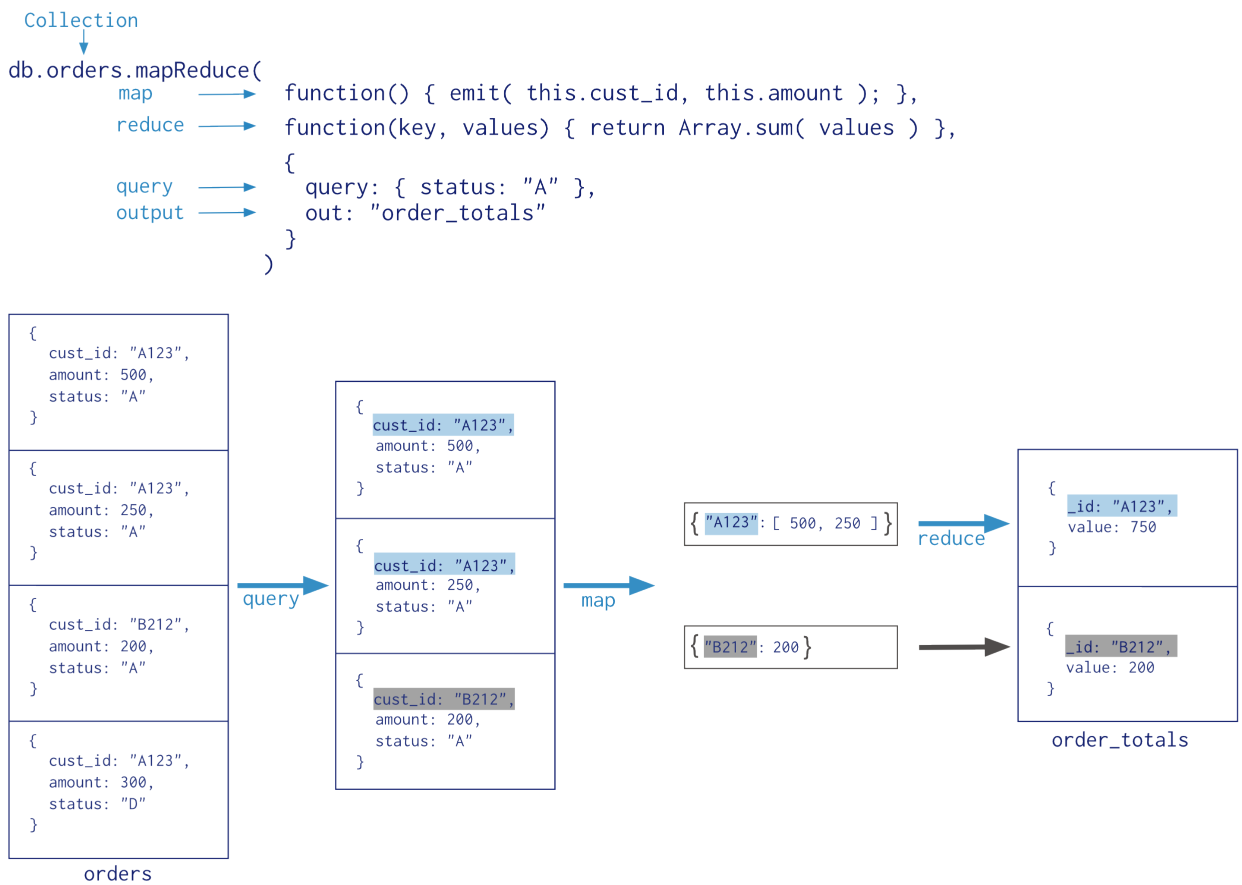

Figure 2: Diagram of the annotated map-reduce operation.

Single Purpose Aggregation Operations

For a number of common *single purpose aggregation operations* (page 11), MongoDB provides special purpose database commands. These common aggregation operations are: returning a count of matching documents, returning the distinct values for a field, and grouping data based on the values of a field. All of these operations aggregate documents from a single collection. While these operations provide simple access to common aggregation processes, they lack the flexibility and capabilities of the aggregation pipeline and map-reduce.

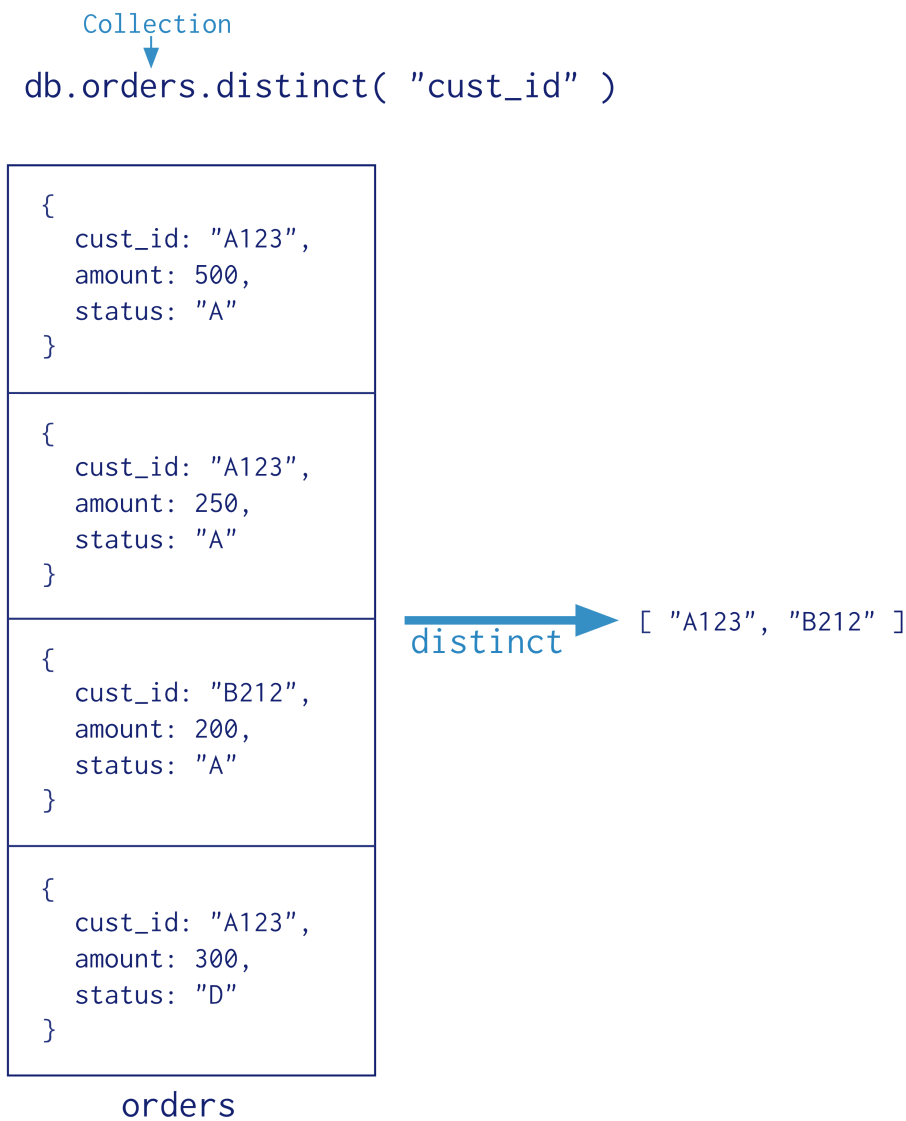

Figure 3: Diagram of the annotated distinct operation.

1.2 Additional Features and Behaviors

Both the aggregation pipeline and map-reduce can operate on a `sharded collection`. Map-reduce operations can also output to a sharded collection. See *Aggregation Pipeline and Sharded Collections* (page 18) and *Map-Reduce and Sharded Collections* (page 18) for details.

The aggregation pipeline can use indexes to improve its performance during some of its stages. In addition, the aggregation pipeline has an internal optimization phase. See *Pipeline Operators and Indexes* (page 9) and *Aggregation Pipeline Optimization* (page 14) for details.

For a feature comparison of the aggregation pipeline, map-reduce, and the special group functionality, see *Aggregation Commands Comparison* (page 39).

2 Aggregation Concepts

MongoDB provides the three approaches to aggregation, each with its own strengths and purposes for a given situation. This section describes these approaches and also describes behaviors and limitations specific to each approach. See also the *chart* (page 39) that compares the approaches.

***Aggregation Pipeline* (page 7)** The aggregation pipeline is a framework for performing aggregation tasks, modeled on the concept of data processing pipelines. Using this framework, MongoDB passes the documents of a single collection through a pipeline. The pipeline transforms the documents into aggregated results, and is accessed through the `aggregate` database command.

***Map-Reduce* (page 10)** Map-reduce is a generic multi-phase data aggregation modality for processing quantities of data. MongoDB provides map-reduce with the `mapReduce` database command.

***Single Purpose Aggregation Operations* (page 11)** MongoDB provides a collection of specific data aggregation operations to support a number of common data aggregation functions. These operations include returning counts of documents, distinct values of a field, and simple grouping operations.

***Aggregation Mechanics* (page 14)** Details internal optimization operations, limits, support for sharded collections, and concurrency concerns.

2.1 Aggregation Pipeline

New in version 2.2.

The aggregation pipeline is a framework for data aggregation modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated results.

The aggregation pipeline provides an alternative to *map-reduce* and may be the preferred solution for many aggregation tasks where the complexity of map-reduce may be unwarranted.

Aggregation pipeline have some limitations on value types and result size. See *Aggregation Pipeline Limits* (page 17) for details on limits and restrictions on the aggregation pipeline.

Pipeline

Conceptually, documents from a collection travel through an aggregation pipeline, which transforms these objects as they pass through. For those familiar with UNIX-like shells (e.g. bash), the concept is analogous to the pipe (i.e. `|`).

The MongoDB aggregation pipeline starts with the documents of a collection and streams the documents from one *pipeline operator* to the next to process the documents. Each operator in the pipeline transforms the documents as they pass through the pipeline. Pipeline operators do not need to produce one output document for every input document. Operators may generate new documents or filter out documents. Pipeline operators can be repeated in the pipeline.

Changed in version 2.6: The `db.collection.aggregate()` method returns a cursor and can return result sets of any size. Previous versions returned all results in a single document, and the result set was subject to a size limit of 16 megabytes.

See *aggregation-pipeline-operator-reference* for the list of pipeline operators that define the stages.

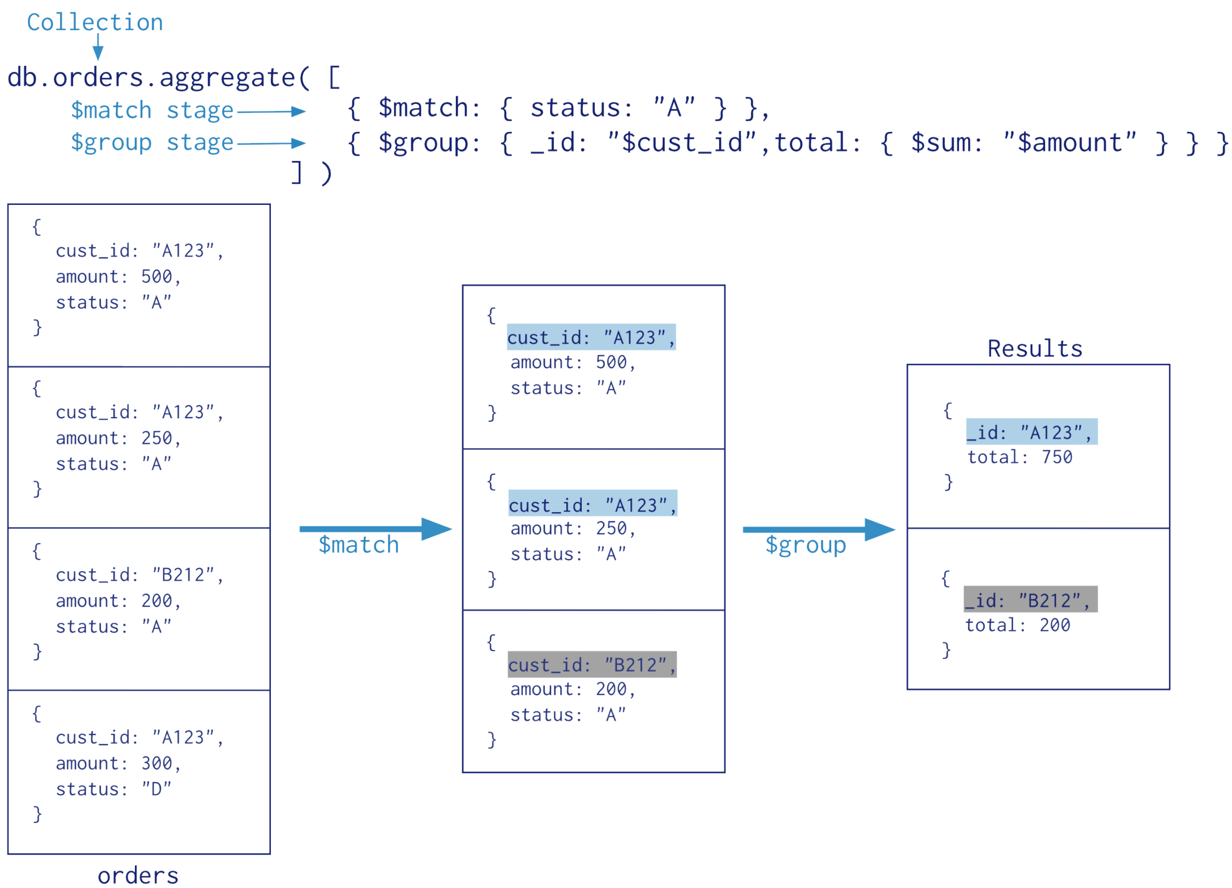

Figure 4: Diagram of the annotated aggregation pipeline operation. The aggregation pipeline has two stages: `$match` and `$group`.

For example usage of the aggregation pipeline, consider *Aggregation with User Preference Data* (page 23) and *Aggregation with the Zip Code Data Set* (page 20), as well as the `aggregate` command and the `db.collection.aggregate()` method reference pages.

Pipeline Expressions

Each pipeline operator takes a pipeline expression as its operand. Pipeline expressions specify the transformation to apply to the input documents. Expressions have a *document* structure and can contain fields, values, and *operators*.

Pipeline expressions can only operate on the current document in the pipeline and cannot refer to data from other documents: expression operations provide in-memory transformation of documents.

Generally, expressions are stateless and are only evaluated when seen by the aggregation process with one exception: *accumulator* expressions. The accumulator expressions, used with the `$group` pipeline operator, maintain their state (e.g. totals, maximums, minimums, and related data) as documents progress through the pipeline.

For the expression operators, see *aggregation-expression-operators*.

Aggregation Pipeline Behavior

In MongoDB, the `aggregate` command operates on a single collection, logically passing the *entire* collection into the aggregation pipeline. To optimize the operation, wherever possible, use the following strategies to avoid scanning the entire collection.

Pipeline Operators and Indexes

The `$match` and `$sort` pipeline operators can take advantage of an index when they occur at the **beginning** of the pipeline.

New in version 2.4: The `$geoNear` pipeline operator takes advantage of a geospatial index. When using `$geoNear`, the `$geoNear` pipeline operation must appear as the first stage in an aggregation pipeline.

Even when the pipeline uses an index, aggregation still requires access to the actual documents; i.e. indexes cannot fully cover an aggregation pipeline.

Changed in version 2.6: In previous versions, for very select use cases, an index could cover a pipeline.

Early Filtering

If your aggregation operation requires only a subset of the data in a collection, use the `$match`, `$limit`, and `$skip` stages to restrict the documents that enter at the beginning of the pipeline. When placed at the beginning of a pipeline, `$match` operations use suitable indexes to scan only the matching documents in a collection.

Placing a `$match` pipeline stage followed by a `$sort` stage at the start of the pipeline is logically equivalent to a single query with a sort and can use an index. When possible, place `$match` operators at the beginning of the pipeline.

Additional Features

The aggregation pipeline has an internal optimization phase that provides improved performance for certain sequences of operators. For details, see *Aggregation Pipeline Optimization* (page 14).

The aggregation pipeline supports operations on sharded collections. See *Aggregation Pipeline and Sharded Collections* (page 18).

2.2 Map-Reduce

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

Consider the following map-reduce operation:

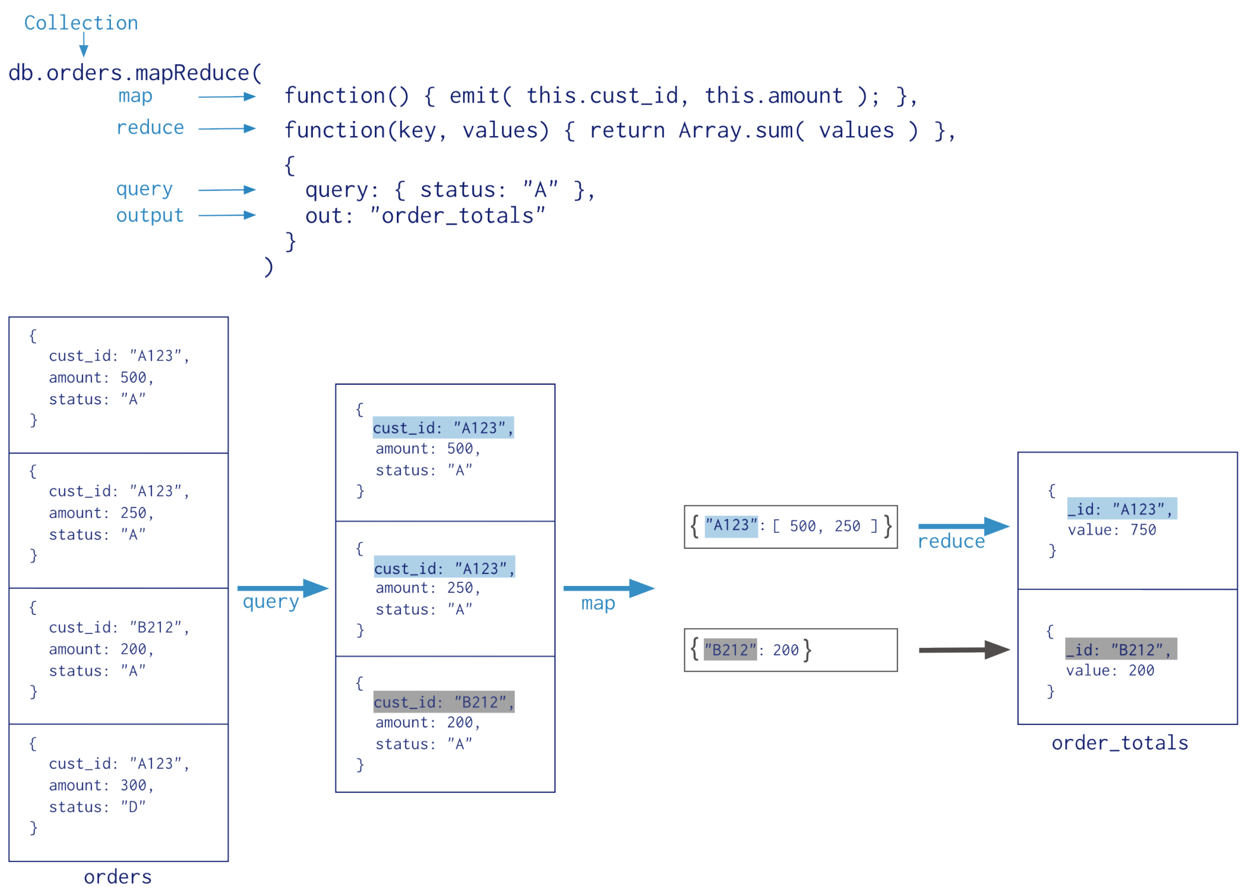

Figure 5: Diagram of the annotated map-reduce operation.

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. The input and the output collections may be sharded.

Note: For most aggregation operations, the *Aggregation Pipeline* (page 7) provides better performance and more coherent interface. However, map-reduce operations provide some flexibility that is not presently available in the aggregation pipeline.

Map-Reduce JavaScript Functions

In MongoDB, map-reduce operations use custom JavaScript functions to *map*, or associate, values to a key. If a key has multiple values mapped to it, the operation *reduces* the values for the key to a single object.

The use of custom JavaScript functions provide flexibility to map-reduce operations. For instance, when processing a document, the map function can create more than one key and value mapping or no mapping. Map-reduce operations can also use a custom JavaScript function to make final modifications to the results at the end of the map and reduce operation, such as perform additional calculations.

Map-Reduce Behavior

In MongoDB, the map-reduce operation can write results to a collection or return the results inline. If you write map-reduce output to a collection, you can perform subsequent map-reduce operations on the same input collection that merge replace, merge, or reduce new results with previous results. See `mapReduce` and *Perform Incremental Map-Reduce* (page 29) for details and examples.

When returning the results of a map reduce operation *inline*, the result documents must be within the `BSON Document Size` limit, which is currently 16 megabytes. For additional information on limits and restrictions on map-reduce operations, see the `http://docs.mongodb.org/manualreference/command/mapReduce` reference page.

MongoDB supports map-reduce operations on `sharded collections`. Map-reduce operations can also output the results to a sharded collection. See *Map-Reduce and Sharded Collections* (page 18).

2.3 Single Purpose Aggregation Operations

Aggregation refers to a broad class of data manipulation operations that compute a result based on an input *and* a specific procedure. MongoDB provides a number of aggregation operations that perform specific aggregation operations on a set of data.

Although limited in scope, particularly compared to the *aggregation pipeline* (page 7) and *map-reduce* (page 10), these operations provide straightforward semantics for common data processing options.

Count

MongoDB can return a count of the number of documents that match a query. The `count` command as well as the `count()` and `cursor.count()` methods provide access to counts in the `mongo` shell.

Example

Given a collection named `records` with *only* the following documents:

```
{ a: 1, b: 0 }
{ a: 1, b: 1 }
{ a: 1, b: 4 }
{ a: 2, b: 2 }
```

The following operation would count all documents in the collection and return the number `4`:

```
db.records.count()
```

The following operation will count only the documents where the value of the field `a` is `1` and return `3`:

```
db.records.count( { a: 1 } )
```

Distinct

The *distinct* operation takes a number of documents that match a query and returns all of the unique values for a field in the matching documents. The `distinct` command and `db.collection.distinct()` method provide this operation in the `mongo` shell. Consider the following examples of a distinct operation:

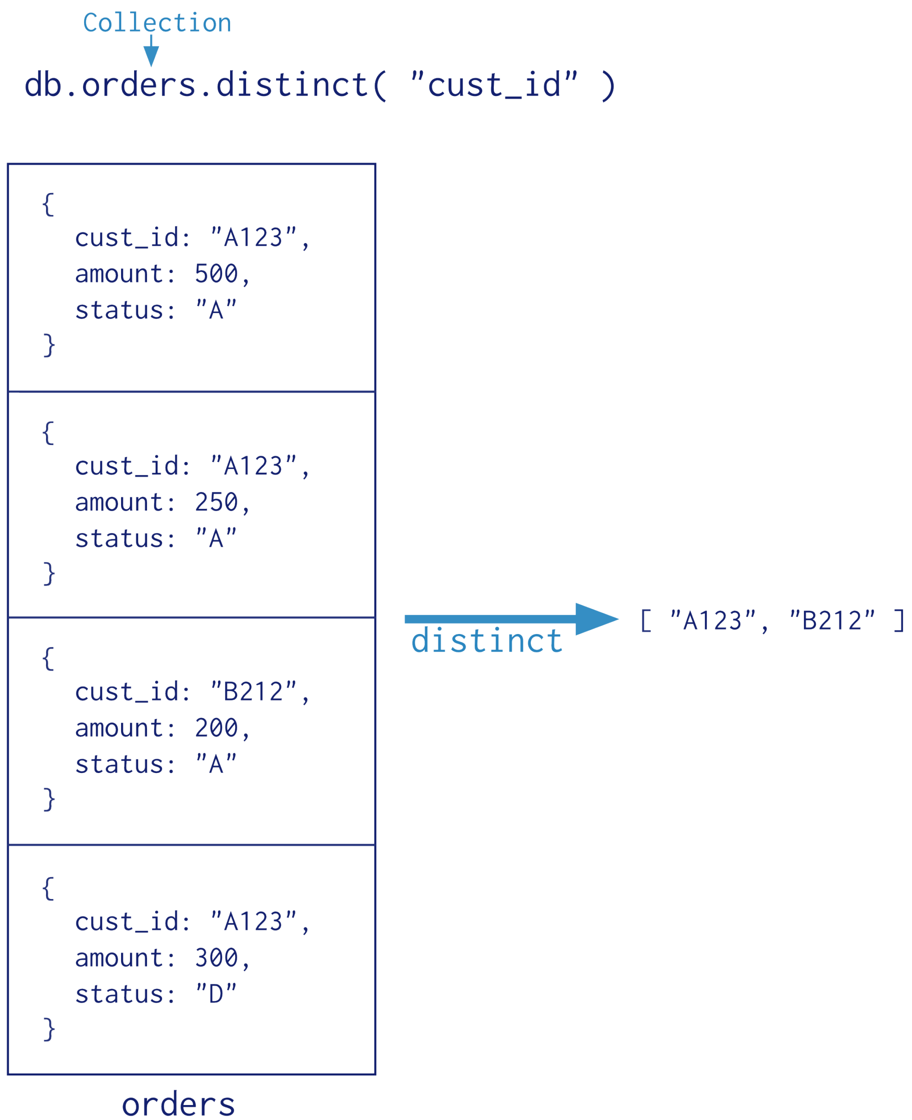

Figure 6: Diagram of the annotated distinct operation.

Example

Given a collection named `records` with *only* the following documents:

```
{ a: 1, b: 0 }
{ a: 1, b: 1 }
{ a: 1, b: 1 }
{ a: 1, b: 4 }
{ a: 2, b: 2 }
```

```
{ a: 2, b: 2 }
```

Consider the following `db.collection.distinct()` operation which returns the distinct values of the field `b`:

```
db.records.distinct( "b" )
```

The results of this operation would resemble:

```
[ 0, 1, 4, 2 ]
```

Group

The *group* operation takes a number of documents that match a query, and then collects groups of documents based on the value of a field or fields. It returns an array of documents with computed results for each group of documents.

Access the grouping functionality via the `group` command or the `db.collection.group()` method in the `mongo` shell.

> **Warning:** `group` does not support data in sharded collections. In addition, the results of the `group` operation must be no larger than 16 megabytes.

Consider the following group operation:

Example

Given a collection named `records` with the following documents:

```
{ a: 1, count: 4 }
{ a: 1, count: 2 }
{ a: 1, count: 4 }
{ a: 2, count: 3 }
{ a: 2, count: 1 }
{ a: 1, count: 5 }
{ a: 4, count: 4 }
```

Consider the following `group` operation which groups documents by the field `a`, where `a` is less than `3`, and sums the field `count` for each group:

```
db.records.group( {
   key: { a: 1 },
   cond: { a: { $lt: 3 } },
   reduce: function(cur, result) { result.count += cur.count },
   initial: { count: 0 }
} )
```

The results of this group operation would resemble the following:

```
[
  { a: 1, count: 15 },
  { a: 2, count: 4 }
]
```

See also:

The `$group` for related functionality in the *aggregation pipeline* (page 7).

2.4 Aggregation Mechanics

This section describes behaviors and limitations for the various aggregation modalities.

Aggregation Pipeline Optimization **(page 14)** Details the internal optimization of certain pipeline sequence.

Aggregation Pipeline Limits **(page 17)** Presents limitations on aggregation pipeline operations.

Aggregation Pipeline and Sharded Collections **(page 18)** Mechanics of aggregation pipeline operations on sharded collections.

Map-Reduce and Sharded Collections **(page 18)** Mechanics of map-reduce operation with sharded collections.

Map Reduce Concurrency **(page 19)** Details the locks taken during map-reduce operations.

Aggregation Pipeline Optimization

Aggregation pipeline operations have an optimization phase which attempts to reshape the pipeline for improved performance.

To see how the optimizer transforms a particular aggregation pipeline, include the `explain` option in the `db.collection.aggregate()` method.

Optimizations are subject to change between releases.

Projection Optimization

The aggregation pipeline can determine if it requires only a subset of the fields in the documents to obtain the results. If so, the pipeline will only use those required fields, reducing the amount of data passing through the pipeline.

Pipeline Sequence Optimization

`$sort` + `$match` Sequence Optimization When you have a sequence with `$sort` followed by a `$match`, the `$match` moves before the `$sort` to minimize the number of objects to sort. For example, if the pipeline consists of the following stages:

```
{ $sort: { age : -1 } },
{ $match: { status: 'A' } }
```

During the optimization phase, the optimizer transforms the sequence to the following:

```
{ $match: { status: 'A' } },
{ $sort: { age : -1 } }
```

`$skip` + `$limit` Sequence Optimization When you have a sequence with `$skip` followed by a `$limit`, the `$limit` moves before the `$skip`. With the reordering, the `$limit` value increases by the `$skip` amount.

For example, if the pipeline consists of the following stages:

```
{ $skip: 10 },
{ $limit: 5 }
```

During the optimization phase, the optimizer transforms the sequence to the following:

```
{ $limit: 15 },
{ $skip: 10 }
```

This optimization allows for more opportunities for *$sort + $limit Coalescence* (page 15), such as with `$sort` + `$skip` + `$limit` sequences. See *$sort + $limit Coalescence* (page 15) for details on the coalescence and *$sort + $skip + $limit Sequence* (page 16) for an example.

For aggregation operations on *sharded collections* (page 18), this optimization reduces the results returned from each shard.

`$redact` + `$match` Sequence Optimization When possible, when the pipeline has the `$redact` stage immediately followed by the `$match` stage, the aggregation can sometimes add a portion of the `$match` stage before the `$redact` stage. If the added `$match` stage is at the start of a pipeline, the aggregation can use an index as well as query the collection to limit the number of documents that enter the pipeline. See *Pipeline Operators and Indexes* (page 9) for more information.

For example, if the pipeline consists of the following stages:

```
{ $redact: { $cond: { if: { $eq: [ "$level", 5 ] }, then: "$$PRUNE", else: "$$DESCEND" } } },
{ $match: { year: 2014, category: { $ne: "Z" } } }
```

The optimizer can add the same `$match` stage before the `$redact` stage:

```
{ $match: { year: 2014 } },
{ $redact: { $cond: { if: { $eq: [ "$level", 5 ] }, then: "$$PRUNE", else: "$$DESCEND" } } },
{ $match: { year: 2014, category: { $ne: "Z" } } }
```

Pipeline Coalescence Optimization

When possible, the optimization phase coalesces a pipeline stage into its predecessor. Generally, coalescence occurs *after* any sequence reordering optimization.

`$sort` + `$limit` Coalescence When a `$sort` immediately precedes a `$limit`, the optimizer can coalesce the `$limit` into the `$sort`. This allows the sort operation to only maintain the top `n` results as it progresses, where `n` is the specified limit, and MongoDB only needs to store `n` items in memory [1]. See *sort-and-memory* for more information.

`$limit` + `$limit` Coalescence When a `$limit` immediately follows another `$limit`, the two stages can coalesce into a single `$limit` where the limit amount is the *smaller* of the two initial limit amounts. For example, a pipeline contains the following sequence:

```
{ $limit: 100 },
{ $limit: 10 }
```

Then the second `$limit` stage can coalesce into the first `$limit` stage and result in a single `$limit` stage where the limit amount `10` is the minimum of the two initial limits `100` and `10`.

```
{ $limit: 10 }
```

`$skip` + `$skip` Coalescence When a `$skip` immediately follows another `$skip`, the two stages can coalesce into a single `$skip` where the skip amount is the *sum* of the two initial skip amounts. For example, a pipeline contains the following sequence:

```
{ $skip: 5 },
{ $skip: 2 }
```

[1] The optimization will still apply when `allowDiskUse` is `true` and the `n` items exceed the *aggregation memory limit* (page 17).

Then the second `$skip` stage can coalesce into the first `$skip` stage and result in a single `$skip` stage where the skip amount `7` is the sum of the two initial limits `5` and `2`.

```
{ $skip: 7 }
```

`$match` + `$match` Coalescence When a `$match` immediately follows another `$match`, the two stages can coalesce into a single `$match` combining the conditions with an `$and`. For example, a pipeline contains the following sequence:

```
{ $match: { year: 2014 } },
{ $match: { status: "A" } }
```

Then the second `$match` stage can coalesce into the first `$match` stage and result in a single `$match` stage

```
{ $match: { $and: [ { "year" : 2014 }, { "status" : "A" } ] } }
```

Examples

The following examples are some sequences that can take advantage of both sequence reordering and coalescence. Generally, coalescence occurs *after* any sequence reordering optimization.

`$sort` + `$skip` + `$limit` Sequence A pipeline contains a sequence of `$sort` followed by a `$skip` followed by a `$limit`:

```
{ $sort: { age : -1 } },
{ $skip: 10 },
{ $limit: 5 }
```

First, the optimizer performs the *$skip + $limit Sequence Optimization* (page 14) to transforms the sequence to the following:

```
{ $sort: { age : -1 } },
{ $limit: 15 }
{ $skip: 10 }
```

The *$skip + $limit Sequence Optimization* (page 14) increases the `$limit` amount with the reordering. See *$skip + $limit Sequence Optimization* (page 14) for details.

The reordered sequence now has `$sort` immediately preceding the `$limit`, and the pipeline can coalesce the two stages to decrease memory usage during the sort operation. See *$sort + $limit Coalescence* (page 15) for more information.

`$limit` + `$skip` + `$limit` + `$skip` Sequence A pipeline contains a sequence of alternating `$limit` and `$skip` stages:

```
{ $limit: 100 },
{ $skip: 5 },
{ $limit: 10 },
{ $skip: 2 }
```

The *$skip + $limit Sequence Optimization* (page 14) reverses the position of the `{ $skip: 5 }` and `{ $limit: 10 }` stages and increases the limit amount:

```
{ $limit: 100 },
{ $limit: 15},
{ $skip: 5 },
{ $skip: 2 }
```

The optimizer then coalesces the two `$limit` stages into a single `$limit` stage and the two `$skip` stages into a single `$skip` stage. The resulting sequence is the following:

```
{ $limit: 15 },
{ $skip: 7 }
```

See *$limit + $limit Coalescence* (page 15) and *$skip + $skip Coalescence* (page 15) for details.

See also:

`explain` option in the `db.collection.aggregate()`

Aggregation Pipeline Limits

Aggregation operations with the `aggregate` command have the following limitations.

Type Restrictions

The *aggregation pipeline* (page 7) cannot operate on values of the following types: `Symbol`, `MinKey`, `MaxKey`, `DBRef`, `Code`, and `CodeWScope`.

Changed in version 2.4: Removed restriction on `Binary` type data. In MongoDB 2.2, the pipeline could not operate on `Binary` type data.

Result Size Restrictions

If the `aggregate` command returns a single document that contains the complete result set, the command will produce an error if the result set exceeds the `BSON Document Size` limit, which is currently 16 megabytes. To manage result sets that exceed this limit, the `aggregate` command can return result sets of *any size* if the command return a cursor or store the results to a collection.

Changed in version 2.6: The `aggregate` command can return results as a cursor or store the results in a collection, which are not subject to the size limit. The `db.collection.aggregate()` returns a cursor and can return result sets of any size.

Memory Restrictions

Changed in version 2.6.

Pipeline stages have a limit of 100 megabytes of RAM. If a stage exceeds this limit, MongoDB will produce an error. To allow for the handling of large datasets, use the `allowDiskUse` option to enable aggregation pipeline stages to write data to temporary files.

See also:

sort-memory-limit and *group-memory-limit*.

Aggregation Pipeline and Sharded Collections

The aggregation pipeline supports operations on *sharded* collections. This section describes behaviors specific to the *aggregation pipeline* (page 7) and sharded collections.

Behavior

Changed in version 2.6.

When operating on a sharded collection, the aggregation pipeline is split into two parts. The first pipeline runs on each shard, or if an early `$match` can exclude shards through the use of the shard key in the predicate, the pipeline runs on only the relevant shards.

The second pipeline consists of the remaining pipeline stages and runs on the *primary shard*. The primary shard merges the cursors from the other shards and runs the second pipeline on these results. The primary shard forwards the final results to the `mongos`. In previous versions, the second pipeline would run on the `mongos`. [2]

Optimization

When splitting the aggregation pipeline into two parts, the pipeline is split to ensure that the shards perform as many stages as possible with consideration for optimization.

To see how the pipeline was split, include the `explain` option in the `db.collection.aggregate()` method.

Optimizations are subject to change between releases.

Map-Reduce and Sharded Collections

Map-reduce supports operations on sharded collections, both as an input and as an output. This section describes the behaviors of `mapReduce` specific to sharded collections.

Sharded Collection as Input

When using sharded collection as the input for a map-reduce operation, `mongos` will automatically dispatch the map-reduce job to each shard in parallel. There is no special option required. `mongos` will wait for jobs on all shards to finish.

Sharded Collection as Output

Changed in version 2.2.

If the `out` field for `mapReduce` has the `sharded` value, MongoDB shards the output collection using the `_id` field as the shard key.

To output to a sharded collection:

- If the output collection does not exist, MongoDB creates and shards the collection on the `_id` field.
- For a new or an empty sharded collection, MongoDB uses the results of the first stage of the map-reduce operation to create the initial *chunks* distributed among the shards.

[2] Until all shards upgrade to v2.6, the second pipeline runs on the `mongos` if any shards are still running v2.4.

- `mongos` dispatches, in parallel, a map-reduce post-processing job to every shard that owns a chunk. During the post-processing, each shard will pull the results for its own chunks from the other shards, run the final reduce/finalize, and write locally to the output collection.

Note:

- During later map-reduce jobs, MongoDB splits chunks as needed.
- Balancing of chunks for the output collection is automatically prevented during post-processing to avoid concurrency issues.

In MongoDB 2.0:

- `mongos` retrieves the results from each shard, performs a merge sort to order the results, and proceeds to the reduce/finalize phase as needed. `mongos` then writes the result to the output collection in sharded mode.
- This model requires only a small amount of memory, even for large data sets.
- Shard chunks are not automatically split during insertion. This requires manual intervention until the chunks are granular and balanced.

Important: For best results, only use the sharded output options for `mapReduce` in version 2.2 or later.

Map Reduce Concurrency

The map-reduce operation is composed of many tasks, including reads from the input collection, executions of the `map` function, executions of the `reduce` function, writes to a temporary collection during processing, and writes to the output collection.

During the operation, map-reduce takes the following locks:

- The read phase takes a read lock. It yields every 100 documents.
- The insert into the temporary collection takes a write lock for a single write.
- If the output collection does not exist, the creation of the output collection takes a write lock.
- If the output collection exists, then the output actions (i.e. `merge`, `replace`, `reduce`) take a write lock. This write lock is *global*, and blocks all operations on the `mongod` instance.

Changed in version 2.4: The V8 JavaScript engine, which became the default in 2.4, allows multiple JavaScript operations to execute at the same time. Prior to 2.4, JavaScript code (i.e. `map`, `reduce`, `finalize` functions) executed in a single thread.

Note: The final write lock during post-processing makes the results appear atomically. However, output actions `merge` and `reduce` may take minutes to process. For the `merge` and `reduce`, the `nonAtomic` flag is available, which releases the lock between writing each output document. the `db.collection.mapReduce()` reference for more information.

---

3 Aggregation Examples

This document provides the practical examples that display the capabilities of *aggregation* (page 7).

***Aggregation with the Zip Code Data Set* (page 20)** Use the aggregation pipeline to group values and to calculate aggregated sums and averages for a collection of United States zip codes.

Aggregation with User Preference Data **(page 23)** Use the pipeline to sort, normalize, and sum data on a collection of user data.

Map-Reduce Examples **(page 27)** Define map-reduce operations that select ranges, group data, and calculate sums and averages.

Perform Incremental Map-Reduce **(page 29)** Run a map-reduce operations over one collection and output results to another collection.

Troubleshoot the Map Function **(page 32)** Steps to troubleshoot the `map` function.

Troubleshoot the Reduce Function **(page 33)** Steps to troubleshoot the `reduce` function.

3.1 Aggregation with the Zip Code Data Set

The examples in this document use the `zipcode` collection. This collection is available at: media.mongodb.org/zips.json[3]. Use `mongoimport` to load this data set into your `mongod` instance.

Data Model

Each document in the `zipcode` collection has the following form:

```
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [
    -74.016323,
    40.710537
  ]
}
```

The `_id` field holds the zip code as a string.

The `city` field holds the city.

The `state` field holds the two letter state abbreviation.

The `pop` field holds the population.

The `loc` field holds the location as a latitude longitude pair.

All of the following examples use the `aggregate()` helper in the `mongo` shell. `aggregate()` provides a wrapper around the `aggregate` database command. See the documentation for your `driver` for a more idiomatic interface for data aggregation operations.

Return States with Populations above 10 Million

To return all states with a population greater than 10 million, use the following aggregation operation:

```
db.zipcodes.aggregate( { $group :
                         { _id : "$state",
                           totalPop : { $sum : "$pop" } } },
                       { $match : {totalPop : { $gte : 10*1000*1000 } } } )
```

[3] http://media.mongodb.org/zips.json

Aggregations operations using the `aggregate()` helper process all documents in the `zipcodes` collection. `aggregate()` connects a number of *pipeline* (page 7) operators, which define the aggregation process.

In this example, the pipeline passes all documents in the `zipcodes` collection through the following steps:

- the `$group` operator collects all documents and creates documents for each state.

  These new per-state documents have one field in addition to the `_id` field: `totalPop` which is a generated field using the `$sum` operation to calculate the total value of all `pop` fields in the source documents.

  After the `$group` operation the documents in the pipeline resemble the following:

  ```
  {
    "_id" : "AK",
    "totalPop" : 550043
  }
  ```

- the `$match` operation filters these documents so that the only documents that remain are those where the value of `totalPop` is greater than or equal to 10 million.

  The `$match` operation does not alter the documents, which have the same format as the documents output by `$group`.

The equivalent *SQL* for this operation is:

```
SELECT state, SUM(pop) AS totalPop
       FROM zipcodes
       GROUP BY state
       HAVING totalPop >= (10*1000*1000)
```

Return Average City Population by State

To return the average populations for cities in each state, use the following aggregation operation:

```
db.zipcodes.aggregate( { $group :
                         { _id : { state : "$state", city : "$city" },
                           pop : { $sum : "$pop" } } },
                       { $group :
                       { _id : "$_id.state",
                         avgCityPop : { $avg : "$pop" } } } )
```

Aggregations operations using the `aggregate()` helper process all documents in the `zipcodes` collection. `aggregate()` connects a number of *pipeline* (page 7) operators that define the aggregation process.

In this example, the pipeline passes all documents in the `zipcodes` collection through the following steps:

- the `$group` operator collects all documents and creates new documents for every combination of the `city` and `state` fields in the source document.

  After this stage in the pipeline, the documents resemble the following:

  ```
  {
    "_id" : {
      "state" : "CO",
      "city" : "EDGEWATER"
    },
    "pop" : 13154
  }
  ```

- the second `$group` operator collects documents by the `state` field and use the `$avg` expression to compute a value for the `avgCityPop` field.

The final output of this aggregation operation is:

```
{
  "_id" : "MN",
  "avgCityPop" : 5335
},
```

Return Largest and Smallest Cities by State

To return the smallest and largest cities by population for each state, use the following aggregation operation:

```
db.zipcodes.aggregate( { $group:
                         { _id: { state: "$state", city: "$city" },
                           pop: { $sum: "$pop" } } },
                       { $sort: { pop: 1 } },
                       { $group:
                         { _id : "$_id.state",
                           biggestCity:  { $last: "$_id.city" },
                           biggestPop:   { $last: "$pop" },
                           smallestCity: { $first: "$_id.city" },
                           smallestPop:  { $first: "$pop" } } },

                       // the following $project is optional, and
                       // modifies the output format.

                       { $project:
                         { _id: 0,
                           state: "$_id",
                           biggestCity:  { name: "$biggestCity",  pop: "$biggestPop" },
                           smallestCity: { name: "$smallestCity", pop: "$smallestPop" } } } )
```

Aggregation operations using the `aggregate()` helper process all documents in the `zipcodes` collection. `aggregate()` combines a number of *pipeline* (page 7) operators that define the aggregation process.

All documents from the `zipcodes` collection pass into the pipeline, which consists of the following steps:

- the `$group` operator collects all documents and creates new documents for every combination of the `city` and `state` fields in the source documents.

  By specifying the value of `_id` as a sub-document that contains both fields, the operation preserves the `state` field for use later in the pipeline. The documents produced by this stage of the pipeline have a second field, `pop`, which uses the `$sum` operator to provide the total of the `pop` fields in the source document.

  At this stage in the pipeline, the documents resemble the following:

  ```
  {
    "_id" : {
      "state" : "CO",
      "city" : "EDGEWATER"
    },
    "pop" : 13154
  }
  ```

- `$sort` operator orders the documents in the pipeline based on the value of the `pop` field from largest to smallest. This operation does not alter the documents.

- the second `$group` operator collects the documents in the pipeline by the `state` field, which is a field inside the nested `_id` document.

Within each per-state document this `$group` operator specifies four fields: Using the `$last` expression, the `$group` operator creates the `biggestcity` and `biggestpop` fields that store the city with the largest population and that population. Using the `$first` expression, the `$group` operator creates the `smallestcity` and `smallestpop` fields that store the city with the smallest population and that population.

The documents, at this stage in the pipeline resemble the following:

```
{
  "_id" : "WA",
  "biggestCity" : "SEATTLE",
  "biggestPop" : 520096,
  "smallestCity" : "BENGE",
  "smallestPop" : 2
}
```

- The final operation is `$project`, which renames the `_id` field to `state` and moves the `biggestCity`, `biggestPop`, `smallestCity`, and `smallestPop` into `biggestCity` and `smallestCity` sub-documents.

The output of this aggregation operation is:

```
{
  "state" : "RI",
  "biggestCity" : {
    "name" : "CRANSTON",
    "pop" : 176404
  },
  "smallestCity" : {
    "name" : "CLAYVILLE",
    "pop" : 45
  }
}
```

3.2 Aggregation with User Preference Data

Data Model

Consider a hypothetical sports club with a database that contains a `user` collection that tracks the user's join dates, sport preferences, and stores these data in documents that resemble the following:

```
{
  _id : "jane",
  joined : ISODate("2011-03-02"),
  likes : ["golf", "racquetball"]
}
{
  _id : "joe",
  joined : ISODate("2012-07-02"),
  likes : ["tennis", "golf", "swimming"]
}
```

Normalize and Sort Documents

The following operation returns user names in upper case and in alphabetical order. The aggregation includes user names for all documents in the `users` collection. You might do this to normalize user names for processing.

```
db.users.aggregate(
  [
    { $project : { name:{$toUpper:"$_id"} , _id:0 } },
    { $sort : { name : 1 } }
  ]
)
```

All documents from the `users` collection pass through the pipeline, which consists of the following operations:

- The `$project` operator:
  - creates a new field called `name`.
  - converts the value of the `_id` to upper case, with the `$toUpper` operator. Then the `$project` creates a new field, named `name` to hold this value.
  - suppresses the `id` field. `$project` will pass the `_id` field by default, unless explicitly suppressed.
- The `$sort` operator orders the results by the `name` field.

The results of the aggregation would resemble the following:

```
{
  "name" : "JANE"
},
{
  "name" : "JILL"
},
{
  "name" : "JOE"
}
```

Return Usernames Ordered by Join Month

The following aggregation operation returns user names sorted by the month they joined. This kind of aggregation could help generate membership renewal notices.

```
db.users.aggregate(
  [
    { $project : { month_joined : {
                                       $month : "$joined"
                                     },
                   name : "$_id",
                   _id : 0
                 },
    { $sort : { month_joined : 1 } }
  ]
)
```

The pipeline passes all documents in the `users` collection through the following operations:

- The `$project` operator:
  - Creates two new fields: `month_joined` and `name`.
  - Suppresses the `id` from the results. The `aggregate()` method includes the `_id`, unless explicitly suppressed.
- The `$month` operator converts the values of the `joined` field to integer representations of the month. Then the `$project` operator assigns those values to the `month_joined` field.

- The `$sort` operator sorts the results by the `month_joined` field.

The operation returns results that resemble the following:

```
{
  "month_joined" : 1,
  "name" : "ruth"
},
{
  "month_joined" : 1,
  "name" : "harold"
},
{
  "month_joined" : 1,
  "name" : "kate"
}
{
  "month_joined" : 2,
  "name" : "jill"
}
```

Return Total Number of Joins per Month

The following operation shows how many people joined each month of the year. You might use this aggregated data for recruiting and marketing strategies.

```
db.users.aggregate(
  [
    { $project : { month_joined : { $month : "$joined" } } } ,
    { $group : { _id : {month_joined:"$month_joined"} , number : { $sum : 1 } } },
    { $sort : { "_id.month_joined" : 1 } }
  ]
)
```

The pipeline passes all documents in the `users` collection through the following operations:

- The `$project` operator creates a new field called `month_joined`.
- The `$month` operator converts the values of the `joined` field to integer representations of the month. Then the `$project` operator assigns the values to the `month_joined` field.
- The `$group` operator collects all documents with a given `month_joined` value and counts how many documents there are for that value. Specifically, for each unique value, `$group` creates a new "per-month" document with two fields:
  - `_id`, which contains a nested document with the `month_joined` field and its value.
  - `number`, which is a generated field. The `$sum` operator increments this field by 1 for every document containing the given `month_joined` value.
- The `$sort` operator sorts the documents created by `$group` according to the contents of the `month_joined` field.

The result of this aggregation operation would resemble the following:

```
{
  "_id" : {
    "month_joined" : 1
  },
  "number" : 3
```

```
},
{
  "_id" : {
    "month_joined" : 2
  },
  "number" : 9
},
{
  "_id" : {
    "month_joined" : 3
  },
  "number" : 5
}
```

Return the Five Most Common "Likes"

The following aggregation collects top five most "liked" activities in the data set. This type of analysis could help inform planning and future development.

```
db.users.aggregate(
  [
    { $unwind : "$likes" },
    { $group : { _id : "$likes" , number : { $sum : 1 } } },
    { $sort : { number : -1 } },
    { $limit : 5 }
  ]
)
```

The pipeline begins with all documents in the `users` collection, and passes these documents through the following operations:

- The `$unwind` operator separates each value in the `likes` array, and creates a new version of the source document for every element in the array.

Example

Given the following document from the `users` collection:

```
{
  _id : "jane",
  joined : ISODate("2011-03-02"),
  likes : ["golf", "racquetball"]
}
```

The `$unwind` operator would create the following documents:

```
{
  _id : "jane",
  joined : ISODate("2011-03-02"),
  likes : "golf"
}
{
  _id : "jane",
  joined : ISODate("2011-03-02"),
  likes : "racquetball"
}
```

- The `$group` operator collects all documents the same value for the `likes` field and counts each grouping. With this information, `$group` creates a new document with two fields:
  - `_id`, which contains the `likes` value.
  - `number`, which is a generated field. The `$sum` operator increments this field by 1 for every document containing the given `likes` value.
- The `$sort` operator sorts these documents by the `number` field in reverse order.
- The `$limit` operator only includes the first 5 result documents.

The results of aggregation would resemble the following:

```
{
  "_id" : "golf",
  "number" : 33
},
{
  "_id" : "racquetball",
  "number" : 31
},
{
  "_id" : "swimming",
  "number" : 24
},
{
  "_id" : "handball",
  "number" : 19
},
{
  "_id" : "tennis",
  "number" : 18
}
```

3.3 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.

3.4 Perform Incremental Map-Reduce

Map-reduce operations can handle complex aggregation tasks. To perform map-reduce operations, MongoDB provides the `mapReduce` command and, in the `mongo` shell, the `db.collection.mapReduce()` wrapper method.

If the map-reduce data set is constantly growing, you may want to perform an incremental map-reduce rather than performing the map-reduce operation over the entire data set each time.

To perform incremental map-reduce:

1. Run a map-reduce job over the current collection and output the result to a separate collection.
2. When you have more data to process, run subsequent map-reduce job with:
   - the `query` parameter that specifies conditions that match *only* the new documents.
   - the `out` parameter that specifies the `reduce` action to merge the new results into the existing output collection.

Consider the following example where you schedule a map-reduce operation on a `sessions` collection to run at the end of each day.

Data Setup

The `sessions` collection contains documents that log users' sessions each day, for example:

```
db.sessions.save( { userid: "a", ts: ISODate('2011-11-03 14:17:00'), length: 95 } );
db.sessions.save( { userid: "b", ts: ISODate('2011-11-03 14:23:00'), length: 110 } );
db.sessions.save( { userid: "c", ts: ISODate('2011-11-03 15:02:00'), length: 120 } );
db.sessions.save( { userid: "d", ts: ISODate('2011-11-03 16:45:00'), length: 45 } );

db.sessions.save( { userid: "a", ts: ISODate('2011-11-04 11:05:00'), length: 105 } );
db.sessions.save( { userid: "b", ts: ISODate('2011-11-04 13:14:00'), length: 120 } );
db.sessions.save( { userid: "c", ts: ISODate('2011-11-04 17:00:00'), length: 130 } );
db.sessions.save( { userid: "d", ts: ISODate('2011-11-04 15:37:00'), length: 65 } );
```

Initial Map-Reduce of Current Collection

Run the first map-reduce operation as follows:

1. Define the map function that maps the `userid` to an object that contains the fields `userid`, `total_time`, `count`, and `avg_time`:

```
var mapFunction = function() {
                      var key = this.userid;
                      var value = {
                                    userid: this.userid,
                                    total_time: this.length,
                                    count: 1,
                                    avg_time: 0
                                   };

                      emit( key, value );
                  };
```

2. Define the corresponding reduce function with two arguments `key` and `values` to calculate the total time and the count. The `key` corresponds to the `userid`, and the `values` is an array whose elements corresponds to the individual objects mapped to the `userid` in the `mapFunction`.

```
var reduceFunction = function(key, values) {

                        var reducedObject = {
                                              userid: key,
                                              total_time: 0,
                                              count:0,
                                              avg_time:0
                                            };
```

```
                    values.forEach( function(value) {
                                          reducedObject.total_time += value.total_time;
                                          reducedObject.count += value.count;
                                    }
                                  );
                    return reducedObject;
                 };
```

3. Define the finalize function with two arguments `key` and `reducedValue`. The function modifies the `reducedValue` document to add another field `average` and returns the modified document.

```
var finalizeFunction = function (key, reducedValue) {

                          if (reducedValue.count > 0)
                              reducedValue.avg_time = reducedValue.total_time / reducedValue.cou

                          return reducedValue;
                       };
```

4. Perform map-reduce on the `session` collection using the `mapFunction`, the `reduceFunction`, and the `finalizeFunction` functions. Output the results to a collection `session_stat`. If the `session_stat` collection already exists, the operation will replace the contents:

```
db.sessions.mapReduce( mapFunction,
                       reduceFunction,
                       {
                         out: "session_stat",
                         finalize: finalizeFunction
                       }
                     )
```

Subsequent Incremental Map-Reduce

Later, as the `sessions` collection grows, you can run additional map-reduce operations. For example, add new documents to the `sessions` collection:

```
db.sessions.save( { userid: "a", ts: ISODate('2011-11-05 14:17:00'), length: 100 } );
db.sessions.save( { userid: "b", ts: ISODate('2011-11-05 14:23:00'), length: 115 } );
db.sessions.save( { userid: "c", ts: ISODate('2011-11-05 15:02:00'), length: 125 } );
db.sessions.save( { userid: "d", ts: ISODate('2011-11-05 16:45:00'), length: 55 } );
```

At the end of the day, perform incremental map-reduce on the `sessions` collection, but use the `query` field to select only the new documents. Output the results to the collection `session_stat`, but `reduce` the contents with the results of the incremental map-reduce:

```
db.sessions.mapReduce( mapFunction,
                       reduceFunction,
                       {
                         query: { ts: { $gt: ISODate('2011-11-05 00:00:00') } },
                         out: { reduce: "session_stat" },
                         finalize: finalizeFunction
                       }
                     );
```

3.5 Troubleshoot the Map Function

The map function is a JavaScript function that associates or "maps" a value with a key and emits the key and value pair during a *map-reduce* (page 10) operation.

To verify the key and value pairs emitted by the map function, write your own emit function.

Consider 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: 250,
     items: [ { sku: "mmm", qty: 5, price: 2.5 },
              { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

1. Define the map function that maps the price to the cust_id for each document and emits the cust_id and price pair:

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

2. Define the emit function to print the key and value:

```
var emit = function(key, value) {
    print("emit");
    print("key: " + key + "  value: " + tojson(value));
}
```

3. Invoke the map function with a single document from the orders collection:

```
var myDoc = db.orders.findOne( { _id: ObjectId("50a8240b927d5d8b5891743c") } );
map.apply(myDoc);
```

4. Verify the key and value pair is as you expected.

```
emit
key: abc123 value:250
```

5. Invoke the map function with multiple documents from the orders collection:

```
var myCursor = db.orders.find( { cust_id: "abc123" } );

while (myCursor.hasNext()) {
    var doc = myCursor.next();
    print ("document _id= " + tojson(doc._id));
    map.apply(doc);
    print();
}
```

6. Verify the key and value pairs are as you expected.

See also:

The map function must meet various requirements. For a list of all the requirements for the map function, see mapReduce, or the mongo shell helper method db.collection.mapReduce().

3.6 Troubleshoot the Reduce Function

The `reduce` function is a JavaScript function that "reduces" to a single object all the values associated with a particular key during a *map-reduce* (page 10) operation. The `reduce` function must meet various requirements. This tutorial helps verify that the `reduce` function meets the following criteria:

- The `reduce` function must return an object whose *type* must be **identical** to the type of the `value` emitted by the `map` function.
- The order of the elements in the `valuesArray` should not affect the output of the `reduce` function.
- The `reduce` function must be *idempotent*.

For a list of all the requirements for the `reduce` function, see `mapReduce`, or the `mongo` shell helper method `db.collection.mapReduce()`.

Confirm Output Type

You can test that the `reduce` function returns a value that is the same type as the value emitted from the `map` function.

1. Define a `reduceFunction1` function that takes the arguments `keyCustId` and `valuesPrices`. `valuesPrices` is an array of integers:

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

2. Define a sample array of integers:

```
var myTestValues = [ 5, 5, 10 ];
```

3. Invoke the `reduceFunction1` with `myTestValues`:

```
reduceFunction1('myKey', myTestValues);
```

4. Verify the `reduceFunction1` returned an integer:

```
20
```

5. Define a `reduceFunction2` function that takes the arguments `keySKU` and `valuesCountObjects`. `valuesCountObjects` is an array of documents that contain two fields `count` and `qty`:

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

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

                          return reducedValue;
                      };
```

6. Define a sample array of documents:

```
var myTestObjects = [
                      { count: 1, qty: 5 },
                      { count: 2, qty: 10 },
                      { count: 3, qty: 15 }
                    ];
```

7. Invoke the `reduceFunction2` with `myTestObjects`:

```
reduceFunction2('myKey', myTestObjects);
```

8. Verify the `reduceFunction2` returned a document with exactly the `count` and the `qty` field:

```
{ "count" : 6, "qty" : 30 }
```

Ensure Insensitivity to the Order of Mapped Values

The `reduce` function takes a `key` and a `values` array as its argument. You can test that the result of the `reduce` function does not depend on the order of the elements in the `values` array.

1. Define a sample `values1` array and a sample `values2` array that only differ in the order of the array elements:

```
var values1 = [
                { count: 1, qty: 5 },
                { count: 2, qty: 10 },
                { count: 3, qty: 15 }
              ];

var values2 = [
                { count: 3, qty: 15 },
                { count: 1, qty: 5 },
                { count: 2, qty: 10 }
              ];
```

2. Define a `reduceFunction2` function that takes the arguments `keySKU` and `valuesCountObjects`. `valuesCountObjects` is an array of documents that contain two fields `count` and `qty`:

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

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

                          return reducedValue;
                      };
```

3. Invoke the `reduceFunction2` first with `values1` and then with `values2`:

```
reduceFunction2('myKey', values1);
reduceFunction2('myKey', values2);
```

4. Verify the `reduceFunction2` returned the same result:

```
{ "count" : 6, "qty" : 30 }
```

Ensure Reduce Function Idempotence

Because the map-reduce operation may call a `reduce` multiple times for the same key, and won't call a `reduce` for single instances of a key in the working set, the `reduce` function must return a value of the same type as the value emitted from the `map` function. You can test that the `reduce` function process "reduced" values without affecting the *final* value.

1. Define a `reduceFunction2` function that takes the arguments `keySKU` and `valuesCountObjects`. `valuesCountObjects` is an array of documents that contain two fields `count` and `qty`:

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

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

                          return reducedValue;
                      };
```

2. Define a sample key:

```
var myKey = 'myKey';
```

3. Define a sample `valuesIdempotent` array that contains an element that is a call to the `reduceFunction2` function:

```
var valuesIdempotent = [
                         { count: 1, qty: 5 },
                         { count: 2, qty: 10 },
                         reduceFunction2(myKey, [ { count:3, qty: 15 } ] )
                       ];
```

4. Define a sample `values1` array that combines the values passed to `reduceFunction2`:

```
var values1 = [
                { count: 1, qty: 5 },
                { count: 2, qty: 10 },
                { count: 3, qty: 15 }
              ];
```

5. Invoke the `reduceFunction2` first with `myKey` and `valuesIdempotent` and then with `myKey` and `values1`:

```
reduceFunction2(myKey, valuesIdempotent);
reduceFunction2(myKey, values1);
```

6. Verify the `reduceFunction2` returned the same result:

```
{ "count" : 6, "qty" : 30 }
```

4 Aggregation Reference

Aggregation Operator Quick Reference **(page 36)** Quick reference card for aggregation pipeline.

`http://docs.mongodb.org/manualreference/operator/aggregation` Aggregation pipeline operations have a collection of operators available to define and manipulate documents in pipeline stages.

Aggregation Commands Comparison **(page 39)** A comparison of `group`, `mapReduce` and `aggregate` that explores the strengths and limitations of each aggregation modality.

SQL to Aggregation Mapping Chart **(page 41)** An overview common aggregation operations in SQL and MongoDB using the aggregation pipeline and operators in MongoDB and common SQL statements.

Aggregation Interfaces (page 43) The data aggregation interfaces document the invocation format and output for MongoDB's aggregation commands and methods.

Variables in Aggregation (page 43) Use of variables in aggregation pipeline expressions.

4.1 Aggregation Operator Quick Reference

Pipeline Operators

Note: The *aggregation pipeline* (page 7) cannot operate on values of the following types: `Symbol`, `MinKey`, `MaxKey`, `DBRef`, `Code`, and `CodeWScope`.

Pipeline operators appear in an array. Documents pass through the operators in a sequence.

Name	Description
`$project`	Reshapes a document stream. `$project` can rename, add, or remove fields as well as create computed values and sub-documents.
`$match`	Filters the document stream, and only allows matching documents to pass into the next pipeline stage. `$match` uses standard MongoDB queries.
`$redact`	Restricts the content of a returned document on a per-field level.
`$limit`	Restricts the number of documents in an aggregation pipeline.
`$skip`	Skips over a specified number of documents from the pipeline and returns the rest.
`$unwind`	Takes an array of documents and returns them as a stream of documents.
`$group`	Groups documents together for the purpose of calculating aggregate values based on a collection of documents.
`$sort`	Takes all input documents and returns them in a stream of sorted documents.
`$geoNear`	Returns an ordered stream of documents based on proximity to a geospatial point.
`$out`	Writes documents from the pipeline to a collection. The `$out` operator must be the last stage in the pipeline.

Expression Operators

Expression operators calculate values within the *aggregation-pipeline-operator-reference*.

`$group` Operators

Name	Description
`$addToSet`	Returns an array of all the *unique* values for the selected field among for each document in that group.
`$first`	Returns the first value in a group.
`$last`	Returns the last value in a group.
`$max`	Returns the highest value in a group.
`$min`	Returns the lowest value in a group.
`$avg`	Returns an average of all the values in a group.
`$push`	Returns an array of *all* values for the selected field among for each document in that group.
`$sum`	Returns the sum of all the values in a group.

Boolean Operators

These operators accept Booleans as arguments and return Booleans as results.

The operators convert non-Booleans to Boolean values according to the BSON standards. Here, `null`, `undefined`, and `0` values become `false`, while non-zero numeric values, and all other types, such as strings, dates, objects become `true`.

Name	Description
`$and`	Returns true only when *all* values in its input array are true.
`$or`	Returns true when *any* value in its input array are true.
`$not`	Returns the boolean value that is the opposite of the input value.

Set Operators

These operators provide operations on sets.

Name	Description
`$setEquals`	Returns true if two sets have the same elements.
`$setIntersection`	Returns the common elements of the input sets.
`$setDifference`	Returns elements of a set that do not appear in a second set.
`$setUnion`	Returns a set that holds all elements of the input sets.
`$setIsSubset`	Returns true if all elements of a set appear in a second set.
`$anyElementTrue`	Returns true if *any* elements of a set evaluate to true, and false otherwise.
`$allElementsTrue`	Returns true if *all* elements of a set evaluate to true, and false otherwise.

Comparison Operators

These operators perform comparisons between two values and return a Boolean, in most cases reflecting the result of the comparison.

All comparison operators take an array with a pair of values. You may compare numbers, strings, and dates. Except for `$cmp`, all comparison operators return a Boolean value. `$cmp` returns an integer.

Name	Description
`$cmp`	Compares two values and returns the result of the comparison as an integer.
`$eq`	Takes two values and returns true if the values are equivalent.
`$gt`	Takes two values and returns true if the first is larger than the second.
`$gte`	Takes two values and returns true if the first is larger than or equal to the second.
`$lt`	Takes two values and returns true if the second value is larger than the first.
`$lte`	Takes two values and returns true if the second value is larger than or equal to the first.
`$ne`	Takes two values and returns true if the values are *not* equivalent.

Arithmetic Operators

Arithmetic operators support only numbers.

Name	Description
`$add`	Computes the sum of an array of numbers.
`$divide`	Takes two numbers and divides the first number by the second.
`$mod`	Takes two numbers and calculates the modulo of the first number divided by the second.
`$multiply`	Computes the product of an array of numbers.
`$subtract`	Takes an array that contains two numbers or two dates and subtracts the second value from the first.

String Operators

String operators that manipulate strings.

Name	Description
`$concat`	Concatenates two strings.
`$strcasecmp`	Compares two strings and returns an integer that reflects the comparison.
`$substr`	Takes a string and returns portion of that string.
`$toLower`	Converts a string to lowercase.
`$toUpper`	Converts a string to uppercase.

Text Search Operators

Operators to support text search.

Name	Description
`$meta`	Access metadata for `$sort` stage or `$project` stage.

Array Operators

Operators that manipulate arrays.

Name	Description
`$size`	Returns the size of the array.

Projection Expressions

Operators that increase the flexibility within aggregation projection and projection-like expressions. These operators are available in the `$project`, `$group`, and `$redact` pipeline stages.

Name	Description
`$map`	Applies a sub-expression to each item in an array and returns the result of the sub-expression.
`$let`	Defines variables for use within the scope of an aggregation expression.
`$literal`	Forces the aggregation pipeline to return a literal value without evaluating the expression.

Date Operators

Date operators take a “Date” typed value as a single argument and return a number.

Name	Description
`$dayOfYear`	Converts a date to a number between 1 and 366.
`$dayOfMonth`	Converts a date to a number between 1 and 31.
`$dayOfWeek`	Converts a date to a number between 1 and 7.
`$year`	Converts a date to the full year.
`$month`	Converts a date into a number between 1 and 12.
`$week`	Converts a date into a number between 0 and 53
`$hour`	Converts a date into a number between 0 and 23.
`$minute`	Converts a date into a number between 0 and 59.
`$second`	Converts a date into a number between 0 and 59. May be 60 to account for leap seconds.
`$millisecond`	Returns the millisecond portion of a date as an integer between 0 and 999.

Conditional Expressions

Name	Description
`$cond`	A ternary operator that evaluates one expression, and depending on the result returns the value of one following expressions.
`$ifNull`	Evaluates an expression and returns a value.

4.2 Aggregation Commands Comparison

The following table provides a brief overview of the features of the MongoDB aggregation commands.

	aggregate	mapReduce	group
Description	New in version 2.2. Designed with specific goals of improving performance and usability for aggregation tasks. Uses a "pipeline" approach where objects are transformed as they pass through a series of pipeline operators such as `$group`, `$match`, and `$sort`. See `http://docs.mongodb.org/manualreference/operator/aggregation` for more information on the pipeline operators.	Implements the Map-Reduce aggregation for processing large data sets.	Provides grouping functionality. Is slower than the `aggregate` command and has less functionality than the `mapReduce` command.
Key Features	Pipeline operators can be repeated as needed. Pipeline operators need not produce one output document for every input document. Can also generate new documents or filter out documents.	In addition to grouping operations, can perform complex aggregation tasks as well as perform incremental aggregation on continuously growing datasets. See *Map-Reduce Examples* (page 27) and *Perform Incremental Map-Reduce* (page 29).	Can either group by existing fields or with a custom `keyf` JavaScript function, can group by calculated fields. See `group` for information and example using the `keyf` function.
Flexibility	Limited to the operators and expressions supported by the aggregation pipeline. However, can add computed fields, create new virtual sub-objects, and extract sub-fields into the top-level of results by using the `$project` pipeline operator. See `$project` for more information as well as `http://docs.mongodb.org/manualreference/operator/aggregation` for more information on all the available pipeline operators.	Custom `map`, `reduce` and `finalize` JavaScript functions offer flexibility to aggregation logic. See `mapReduce` for details and restrictions on the functions.	Custom `reduce` and `finalize` JavaScript functions offer flexibility to grouping logic. See `group` for details and restrictions on these functions.
Output Results	Returns results in various options (inline as a document that contains the result set, a cursor to the result set) or stores the results in a collection. The result is subject to the *BSON Document size* limit if returned inline as a document that contains the result set. Changed in version 2.6: Can return results as a cursor or store the results to a collection.	Returns results in various options (inline, new collection, merge, replace, reduce). See `mapReduce` for details on the output options. Changed in version 2.2: Provides much better support for sharded map-reduce output than previous versions.	Returns results inline as an array of grouped items. The result set must fit within the *maximum BSON document size limit*. Changed in version 2.2: The returned array can contain at most 20,000 elements; i.e. at most 20,000 unique groupings. Previous versions had a limit of 10,000 elements.
Sharding Notes	Supports non-sharded and sharded input collections.	Supports non-sharded and sharded input collections. Prior to 2.4, JavaScript code executed in a single thread.	Does **not** support sharded collection. Prior to 2.4, JavaScript code executed in a single thread.
More Information	See *Aggregation Pipeline* (page 7) and `aggregate`.	See *Map-Reduce* (page 10) and `mapReduce`.	See `group`.

4.3 SQL to Aggregation Mapping Chart

The *aggregation pipeline* (page 7) allows MongoDB to provide native aggregation capabilities that corresponds to many common data aggregation operations in SQL. If you're new to MongoDB you might want to consider the `http://docs.mongodb.org/manualfaq` section for a selection of common questions.

The following table provides an overview of common SQL aggregation terms, functions, and concepts and the corresponding MongoDB *aggregation operators*:

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	`$match`
GROUP BY	`$group`
HAVING	`$match`
SELECT	`$project`
ORDER BY	`$sort`
LIMIT	`$limit`
SUM()	`$sum`
COUNT()	`$sum`
join	No direct corresponding operator; *however*, the `$unwind` operator allows for somewhat similar functionality, but with fields embedded within the document.

Examples

The following table presents a quick reference of SQL aggregation statements and the corresponding MongoDB statements. The examples in the table assume the following conditions:

- The SQL examples assume *two* tables, `orders` and `order_lineitem` that join by the `order_lineitem.order_id` and the `orders.id` columns.
- The MongoDB examples assume *one* collection `orders` that contain documents of the following prototype:

```
{
  cust_id: "abc123",
  ord_date: ISODate("2012-11-02T17:04:11.102Z"),
  status: 'A',
  price: 50,
  items: [ { sku: "xxx", qty: 25, price: 1 },
           { sku: "yyy", qty: 25, price: 1 } ]
}
```

- The MongoDB statements prefix the names of the fields from the *documents* in the collection `orders` with a `$` character when they appear as operands to the aggregation operations.

SQL Example	MongoDB Example	Description

SELECT COUNT(*) AS count
FROM orders
	db.orders.aggregate([
	{
	$group: {
	_id: null,
	count: { $sum: 1 }
	}
	}
	])
		Count all records from orders

SELECT SUM(price) AS total
FROM orders
	db.orders.aggregate([
	{
	$group: {
	_id: null,
	total: { $sum: "$price" }
	}
	}
	])
		Sum the price field from orders

SELECT cust_id,
SUM(price) AS total
FROM orders
GROUP BY cust_id
	db.orders.aggregate([
	{
	$group: {
	_id: "$cust_id",
	total: { $sum: "$price" }
	}
	}
	])
		For each unique cust_id, sum the price field.

SELECT cust_id,
SUM(price) AS total
FROM orders
GROUP BY cust_id
ORDER BY total
	db.orders.aggregate([
	{
	$group: {
	_id: "$cust_id",
	total: { $sum: "$price" }
	}
	},
	{ $sort: { total: 1 } }
	])
		For each unique cust_id, sum the price field, results sorted by sum.

SELECT cust_id,
ord_date,
SUM(price) AS total
FROM orders
GROUP BY cust_id,
ord_date
	db.orders.aggregate([
	{
	$group: {
	_id: {
	cust_id: "$cust_id",
	ord_date: "$ord_date"
	},
	total: { $sum: "$price" }
	}
	}
	])
		For each unique cust_id, ord_date grouping, sum the price field.

SELECT cust_id,
count(*)
FROM orders
GROUP BY cust_id
HAVING count(*) > 1
	db.orders.aggregate([
	{
	$group: {
	_id: "$cust_id",
	count: { $sum: 1 }
	}
	},
		For cust_id with multiple records, return the cust_id and the corresponding record count.

4.4 Aggregation Interfaces

Aggregation Commands

Name	Description
aggregate	Performs *aggregation tasks* (page 7) such as group using the aggregation framework.
count	Counts the number of documents in a collection.
distinct	Displays the distinct values found for a specified key in a collection.
group	Groups documents in a collection by the specified key and performs simple aggregation.
mapReduce	Performs *map-reduce* (page 10) aggregation for large data sets.

Aggregation Methods

Name	Description
db.collection.aggregate()	Provides access to the *aggregation pipeline* (page 7).
db.collection.group()	Groups documents in a collection by the specified key and performs simple aggregation.
db.collection.mapReduce()	Performs *map-reduce* (page 10) aggregation for large data sets.

4.5 Variables in Aggregation

Aggregation expressions can use both user-defined and system variables.

Variables can hold any `BSON type data`. To access the value of the variable, use a string with the variable name prefixed with double dollar signs (`$$`).

If the variable references an object, to access a specific field in the object, use the dot notation; i.e. `"$$<variable>.<field>"`.

User Variables

User variable names can contain the ascii characters `[_a-zA-Z0-9]` and any non-ascii character.

User variable names must begin with a lowercase ascii letter `[a-z]` or a non-ascii character.

System Variables

MongoDB offers the following system variables:

Variable	Description
ROOT	References the root document, i.e. the top-level document, currently being processed in the aggregation pipeline stage.
CURRENT	References the start of the field path being processed in the aggregation pipeline stage. Unless documented otherwise, all stages start with CURRENT (page 44) the same as ROOT (page 44). CURRENT (page 44) is modifiable. However, since `$<field>` is equivalent to `$$CURRENT.<field>`, rebinding CURRENT (page 44) changes the meaning of `$` accesses.
DESCEND	One of the allowed results of a `$redact` expression.
PRUNE	One of the allowed results of a `$redact` expression.
KEEP	One of the allowed results of a `$redact` expression.

See also:

`$let`, `$redact`

Index