
MongoDB Aggregation and Data Processing

Release 3.2.3

MongoDB, Inc.

February 17, 2016

© MongoDB, Inc. 2008 - 2016 This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 United States License](#)

1	Aggregation Pipeline	3
2	Map-Reduce	5
3	Single Purpose Aggregation Operations	7
4	Additional Features and Behaviors	9
4.1	Aggregation Pipeline	9
4.2	Map-Reduce	25
4.3	Aggregation Reference	36
5	Additional Resources	51

On this page

- [Aggregation Pipeline \(page 3\)](#)
- [Map-Reduce \(page 5\)](#)
- [Single Purpose Aggregation Operations \(page 7\)](#)
- [Additional Features and Behaviors \(page 9\)](#)
- [Additional Resources \(page 51\)](#)

Aggregation operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. MongoDB provides three ways to perform aggregation: the *aggregation pipeline* (page 3), the *map-reduce function* (page 5), and *single purpose aggregation methods* (page 7).

Aggregation Pipeline

MongoDB 2.2 introduced a new *aggregation framework* (page 9), 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.

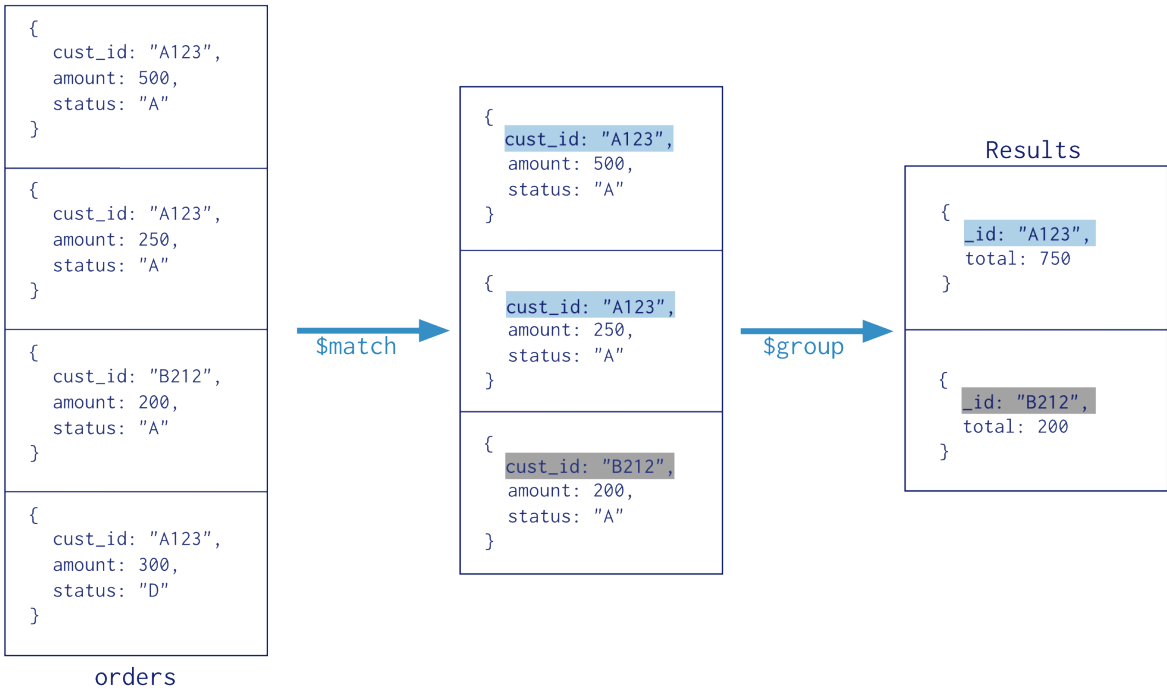
The pipeline provides efficient data aggregation using native operations within MongoDB, and is the preferred method for data aggregation in MongoDB.

The aggregation pipeline can operate on a `sharded collection`.

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 11) and *Aggregation Pipeline Optimization* (page 11) for details.


```

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



Map-Reduce

MongoDB also provides *map-reduce* (page 25) 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.

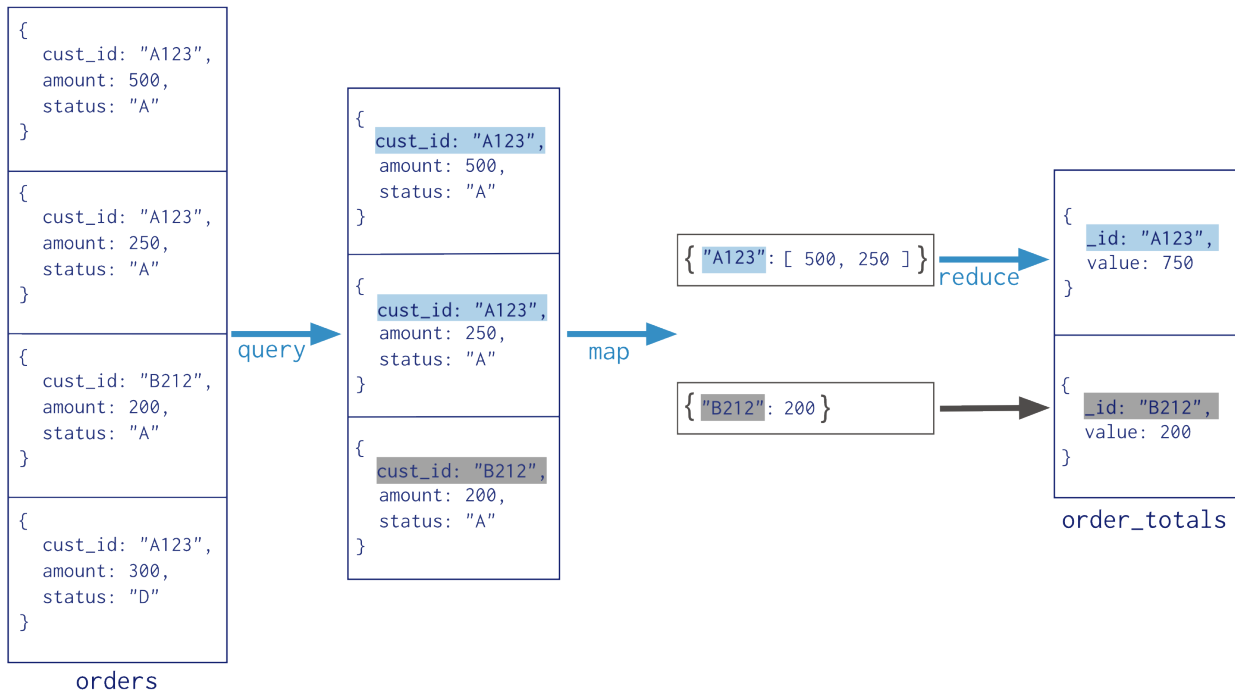
Map-reduce uses custom JavaScript functions to perform the map and reduce operations, as well as the optional *finalize* operation. While the custom JavaScript provide great flexibility compared to the aggregation pipeline, in general, map-reduce is less efficient and more complex than the aggregation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

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 16) and *Map-Reduce and Sharded Collections* (page 26) for details.

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.

```

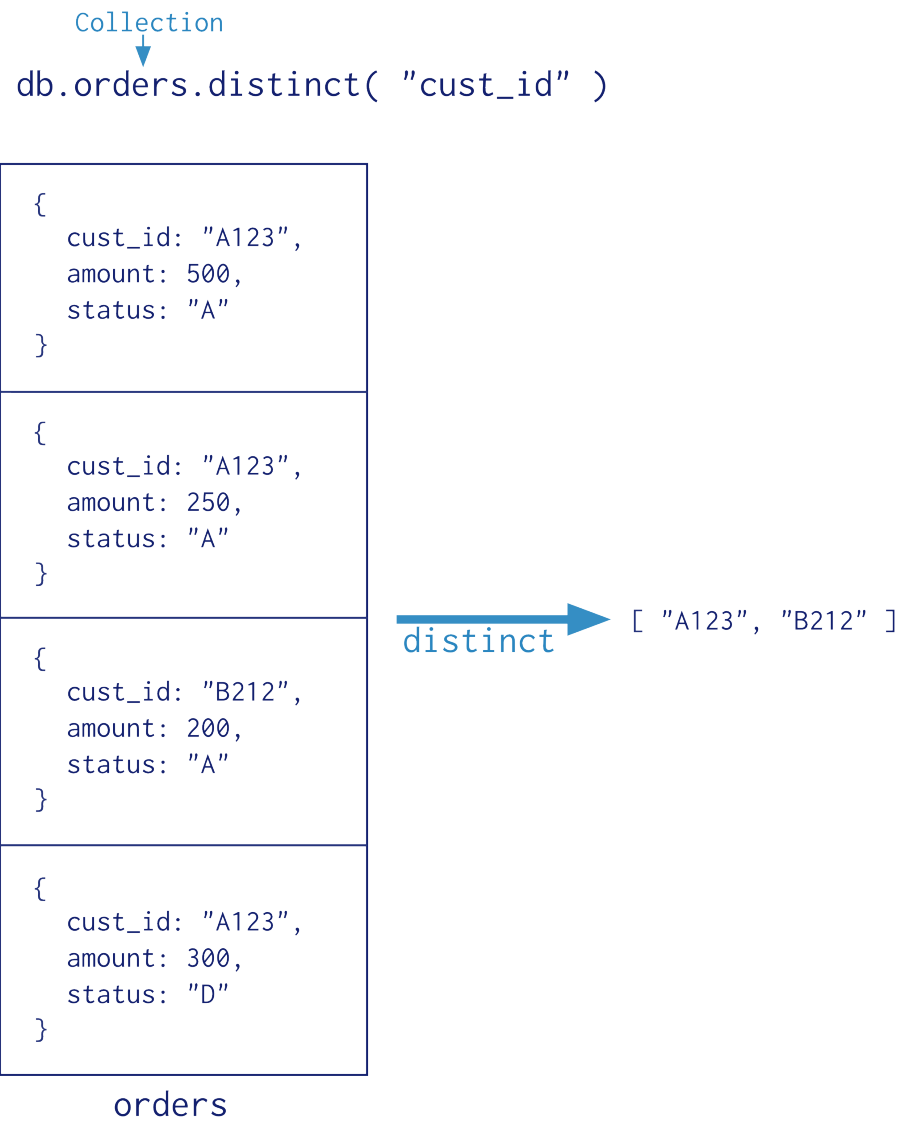
Collection
  ↓
db.orders.mapReduce(
  map   → function() { emit( this.cust_id, this.amount ); },
  reduce → function(key, values) { return Array.sum( values ) },
  {
    query: { status: "A" },
    out: "order_totals"
  }
)
    
```



Single Purpose Aggregation Operations

MongoDB also provides `db.collection.count()`, `db.collection.group()`, `db.collection.distinct()`. special purpose database commands.

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.



Additional Features and Behaviors

For a feature comparison of the aggregation pipeline, map-reduce, and the special group functionality, see *Aggregation Commands Comparison* (page 44).

4.1 Aggregation Pipeline

On this page

- [Pipeline](#) (page 9)
- [Pipeline Expressions](#) (page 9)
- [Aggregation Pipeline Behavior](#) (page 11)
- [Additional Resources](#) (page 24)

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 aggregated results.

The aggregation pipeline provides an alternative to *map-reduce* and may be the preferred solution for 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 15) for details on limits and restrictions on the aggregation pipeline.

4.1.1 Pipeline

The MongoDB aggregation pipeline consists of *stages*. Each stage transforms the documents as they pass through the pipeline. Pipeline stages do not need to produce one output document for every input document; e.g., some stages may generate new documents or filter out documents. Pipeline stages can appear multiple times in the pipeline.

MongoDB provides the `db.collection.aggregate()` method in the mongo shell and the `aggregate` command for aggregation pipeline. See *aggregation-pipeline-operator-reference* for the available stages.

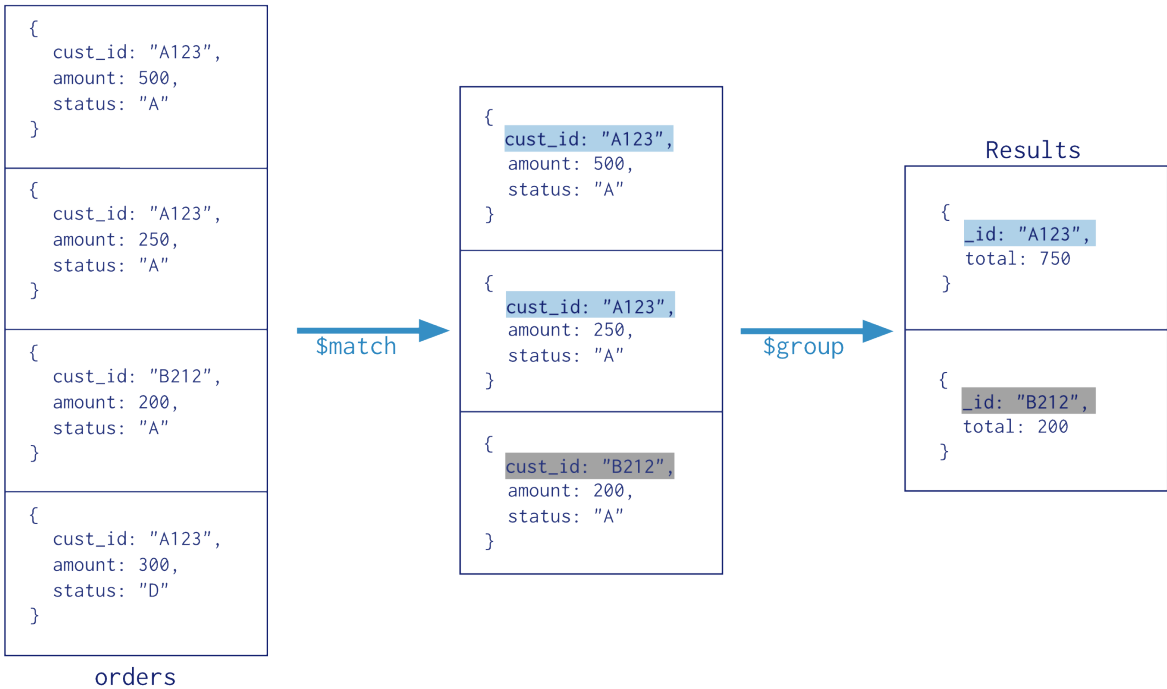
For example usage of the aggregation pipeline, consider *Aggregation with User Preference Data* (page 20) and *Aggregation with the Zip Code Data Set* (page 17).

4.1.2 Pipeline Expressions

Some pipeline stages 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 other *expression* (page 38).

```

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



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 accumulators, used in the `$group` stage, maintain their state (e.g. totals, maximums, minimums, and related data) as documents progress through the pipeline.

Changed in version 3.2: Some accumulators are available in the `$project` stage; however, when used in the `$project` stage, the accumulators do not maintain their state across documents.

For more information on expressions, see *Expressions* (page 38).

4.1.3 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.

Changed in version 3.2: Starting in MongoDB 3.2, indexes can *cover* an aggregation pipeline. In MongoDB 2.6 and 3.0, indexes could not cover an aggregation pipeline since even when the pipeline uses an index, aggregation still requires access to the actual documents.

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 11).

The aggregation pipeline supports operations on sharded collections. See *Aggregation Pipeline and Sharded Collections* (page 16).

Aggregation Pipeline Optimization

On this page

- [Projection Optimization](#) (page 12)
- [Pipeline Sequence Optimization](#) (page 12)
- [Pipeline Coalescence Optimization](#) (page 13)
- [Examples](#) (page 15)

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 13), such as with `$sort` + `$skip` + `$limit` sequences. See *`$sort` + `$limit` Coalescence* (page 13) for details on the coalescence and *`$sort` + `$skip` + `$limit` Sequence* (page 15) for an example.

For aggregation operations on *sharded collections* (page 16), 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 11) 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" } } }
```

\$project + \$skip or \$limit Sequence Optimization New in version 3.2.

When you have a sequence with `$project` followed by either `$skip` or `$limit`, the `$skip` or `$limit` moves before `$project`. For example, if the pipeline consists of the following stages:

```
{ $sort: { age : -1 } },
{ $project: { status: 1, name: 1 } },
{ $limit: 5 }
```

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

```
{ $sort: { age : -1 } },
{ $limit: 5 }
{ $project: { status: 1, name: 1 } },
```

This optimization allows for more opportunities for *\$sort + \$limit Coalescence* (page 13), such as with `$sort + $limit` sequences. See *\$sort + \$limit Coalescence* (page 13) for details on the coalescence.

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 memory¹. 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.

¹ The optimization will still apply when `allowDiskUse` is `true` and the *n* items exceed the *aggregation memory limit* (page 16).

```
{ $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 }
```

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" } ] } }
```

\$lookup + \$unwind Coalescence New in version 3.2.

When a `$unwind` immediately follows another `$lookup`, and the `$unwind` operates on the `as` field of the `$lookup`, the optimizer can coalesce the `$unwind` into the `$lookup` stage. This avoids creating large intermediate documents.

For example, a pipeline contains the following sequence:

```
{  
  $lookup: {  
    from: "otherCollection",  
    as: "resultingArray",  
    localField: "x",  
    foreignField: "y"  
  }  
},  
{ $unwind: "$resultingArray" }
```

The optimizer can coalesce the `$unwind` stage into the `$lookup` stage. If you run the aggregation with `explain` option, the `explain` output shows the coalesced stage:

```
{  
  $lookup: {  
    from: "otherCollection",  
    as: "resultingArray",  
    localField: "x",  
    foreignField: "y",  
    unwinding: { preserveNullAndEmptyArrays: false }  
  }  
}
```

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 12) to transform the sequence to the following:

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

The *`$skip + $limit Sequence Optimization`* (page 12) increases the `$limit` amount with the reordering. See *`$skip + $limit Sequence Optimization`* (page 12) 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 13) 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 12) 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 + $limit Coalescence`* (page 13) and *`$skip + $skip Coalescence`* (page 14) for details.

See also:

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

Aggregation Pipeline Limits

On this page

- [Result Size Restrictions](#) (page 16)
- [Memory Restrictions](#) (page 16)

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

Result Size Restrictions Changed in version 2.6.

Starting in MongoDB 2.6, the `aggregate` command can return a cursor or store the results in a collection. When returning a cursor or storing the results in a collection, each document in the result set is subject to the `BSON Document Size` limit, currently 16 megabytes; if any single document that exceeds the `BSON Document Size` limit, the command will produce an error. The limit only applies to the returned documents; during the pipeline processing, the documents may exceed this size. The `db.collection.aggregate()` method returns a cursor by default starting in MongoDB 2.6

If you do not specify the cursor option or store the results in a collection, the `aggregate` command returns a single BSON document that contains a field with the result set. As such, the command will produce an error if the total size of the result set exceeds the `BSON Document Size` limit.

Earlier versions of the `aggregate` command can only return a single BSON document that contains the result set and will produce an error if the if the total size of the result set exceeds the `BSON Document Size` limit.

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

On this page

- [Behavior](#) (page 16)
- [Optimization](#) (page 17)

The aggregation pipeline supports operations on *sharded* collections. This section describes behaviors specific to the *aggregation pipeline* (page 9) and sharded collections.

Behavior Changed in version 3.2.

If the pipeline starts with an exact `$match` on a shard key, the entire pipeline runs on the matching shard only. Previously, the pipeline would have been split, and the work of merging it would have to be done on the primary shard.

For aggregation operations that must run on multiple shards, if the operations do not require running on the database's primary shard, these operations will route the results to a random shard to merge the results to avoid overloading the primary shard for that database. The `$out` stage and the `$lookup` stage require running on the database's primary shard.

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.

Aggregation with the Zip Code Data Set

On this page

- [Data Model](#) (page 17)
- [aggregate\(\) Method](#) (page 17)
- [Return States with Populations above 10 Million](#) (page 18)
- [Return Average City Population by State](#) (page 18)
- [Return Largest and Smallest Cities by State](#) (page 19)

The examples in this document use the `zipcodes` collection. This collection is available at: media.mongodb.org/zips.json². Use `mongoimport` to load this data set into your `mongod` instance.

Data Model Each document in the `zipcodes` 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 name. A city can have more than one zip code associated with it as different sections of the city can each have a different zip code.
- 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.

aggregate() Method All of the following examples use the `aggregate()` helper in the `mongo` shell.

The `aggregate()` method uses the *aggregation pipeline* (page 9) to processes documents into aggregated results. An *aggregation pipeline* (page 9) consists of *stages* with each stage processing the documents as they pass along the pipeline. Documents pass through the stages in sequence.

The `aggregate()` method in the `mongo` shell provides a wrapper around the `aggregate` database command. See the documentation for your `driver` for a more idiomatic interface for data aggregation operations.

²<http://media.mongodb.org/zips.json>

Return States with Populations above 10 Million The following aggregation operation returns all states with total population greater than 10 million:

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

In this example, the *aggregation pipeline* (page 9) consists of the `$group` stage followed by the `$match` stage:

- The `$group` stage groups the documents of the `zipcode` collection by the `state` field, calculates the `totalPop` field for each state, and outputs a document for each unique state.

The new per-state documents have two fields: the `_id` field and the `totalPop` field. The `_id` field contains the value of the state; i.e. the group by field. The `totalPop` field is a calculated field that contains the total population of each state. To calculate the value, `$group` uses the `$sum` operator to add the population field (`pop`) for each state.

After the `$group` stage, the documents in the pipeline resemble the following:

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

- The `$match` stage filters these grouped documents to output only those documents whose `totalPop` value is greater than or equal to 10 million. The `$match` stage does not alter the matching documents but outputs the matching documents unmodified.

The equivalent *SQL* for this aggregation operation is:

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

See also:

`$group`, `$match`, `$sum`

Return Average City Population by State The following aggregation operation returns the average populations for cities in each state:

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

In this example, the *aggregation pipeline* (page 9) consists of the `$group` stage followed by another `$group` stage:

- The first `$group` stage groups the documents by the combination of `city` and `state`, uses the `$sum` expression to calculate the population for each combination, and outputs a document for each `city` and `state` combination.³

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

```
{
  "_id" : {
    "state" : "CO",
```

³ A city can have more than one zip code associated with it as different sections of the city can each have a different zip code.

```

    "city" : "EDGEWATER"
  },
  "pop" : 13154
}

```

- A second `$group` stage groups the documents in the pipeline by the `_id.state` field (i.e. the `state` field inside the `_id` document), uses the `$avg` expression to calculate the average city population (`avgCityPop`) for each state, and outputs a document for each state.

The documents that result from this aggregation operation resembles the following:

```

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

```

See also:

`$group`, `$sum`, `$avg`

Return Largest and Smallest Cities by State The following aggregation operation returns the smallest and largest cities by population for each state:

```

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" }
    }
  }
] )

```

In this example, the *aggregation pipeline* (page 9) consists of a `$group` stage, a `$sort` stage, another `$group` stage, and a `$project` stage:

- The first `$group` stage groups the documents by the combination of the `city` and `state`, calculates the sum of the `pop` values for each combination, and outputs a document for each `city` and `state` combination.

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

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

- The `$sort` stage orders the documents in the pipeline by the `pop` field value, from smallest to largest; i.e. by increasing order. This operation does not alter the documents.
- The next `$group` stage groups the now-sorted documents by the `_id.state` field (i.e. the `state` field inside the `_id` document) and outputs a document for each state.

The stage also calculates the following four fields for each state. 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 `$project` stage renames the `_id` field to `state` and moves the `biggestCity`, `biggestPop`, `smallestCity`, and `smallestPop` into `biggestCity` and `smallestCity` embedded documents.

The output documents of this aggregation operation resemble the following:

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

Aggregation with User Preference Data

On this page

- [Data Model \(page 21\)](#)
- [Normalize and Sort Documents \(page 21\)](#)
- [Return Usernames Ordered by Join Month \(page 21\)](#)
- [Return Total Number of Joins per Month \(page 22\)](#)
- [Return the Five Most Common “Likes” \(page 23\)](#)

Data Model Consider a hypothetical sports club with a database that contains a `users` 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
}
```

4.1.4 Additional Resources

- MongoDB Analytics: Learn Aggregation by Example: Exploratory Analytics and Visualization Using Flight Data⁴
- MongoDB for Time Series Data: Analyzing Time Series Data Using the Aggregation Framework and Hadoop⁵
- The Aggregation Framework⁶

⁴<http://www.mongodb.com/presentations/mongodb-analytics-learn-aggregation-example-exploratory-analytics-and-visualization?jmp=docs>

⁵<http://www.mongodb.com/presentations/mongodb-time-series-data-part-2-analyzing-time-series-data-using-aggregation-framework?jmp=docs>

⁶<https://www.mongodb.com/presentations/aggregation-framework-0?jmp=docs>

- Webinar: Exploring the Aggregation Framework⁷
- Quick Reference Cards⁸

4.2 Map-Reduce

On this page

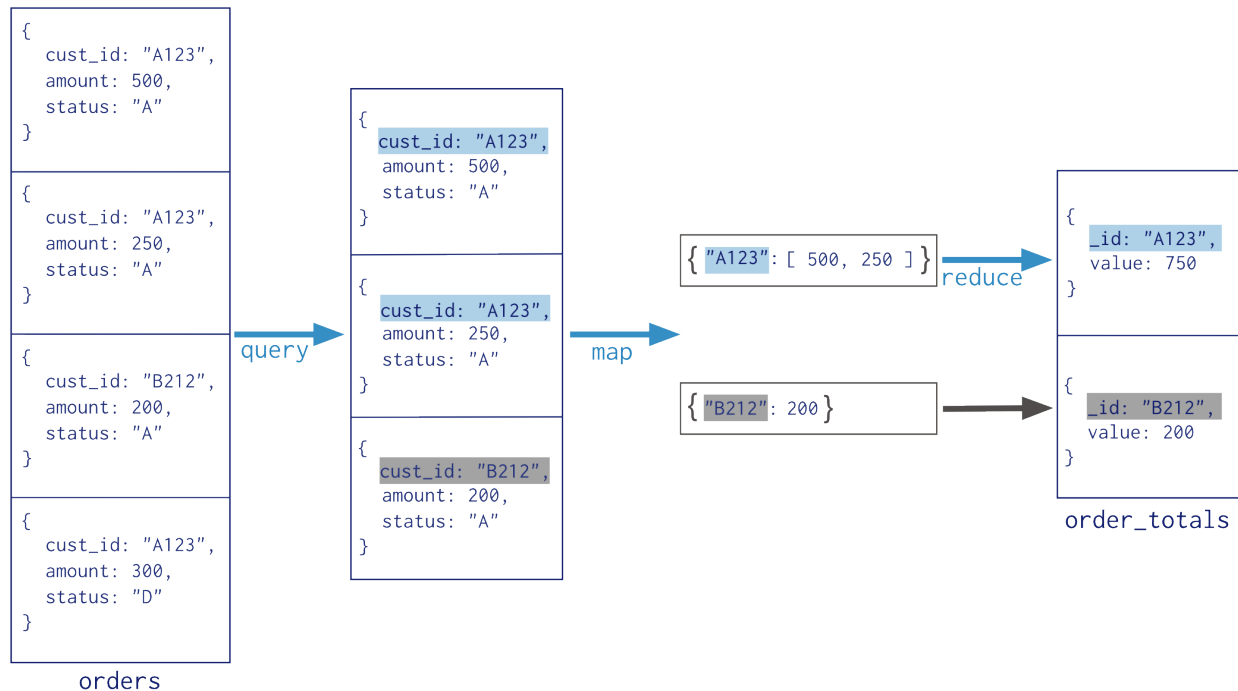
- Map-Reduce JavaScript Functions (page 26)
- Map-Reduce Behavior (page 26)

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:

```

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



⁷<https://www.mongodb.com/webinar/exploring-the-aggregation-framework?jmp=docs>

⁸<https://www.mongodb.com/lp/misc/quick-reference-cards?jmp=docs>

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 9) provides better performance and more coherent interface. However, map-reduce operations provide some flexibility that is not presently available in the aggregation pipeline.

4.2.1 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.

4.2.2 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 30) 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 <https://docs.mongodb.org/manual/reference/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 26).

Map-Reduce and Sharded Collections

On this page

- [Sharded Collection as Input](#) (page 27)
- [Sharded Collection as Output](#) (page 27)

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.
- `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. See the `db.collection.mapReduce()` reference for more information.

Map-Reduce Examples

On this page

- [Return the Total Price Per Customer \(page 28\)](#)
- [Calculate Order and Total Quantity with Average Quantity Per Item \(page 29\)](#)

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.

Perform Incremental Map-Reduce

On this page

- [Data Setup \(page 31\)](#)
- [Initial Map-Reduce of Current Collection \(page 31\)](#)
- [Subsequent Incremental Map-Reduce \(page 32\)](#)

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.count;

    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
                      }
                    );
```

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 25) 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()`.

Troubleshoot the Reduce Function

On this page

- [Confirm Output Type](#) (page 34)
- [Ensure Insensitivity to the Order of Mapped Values](#) (page 35)
- [Ensure Reduce Function Idempotence](#) (page 36)

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 25) 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 }
```

4.3 Aggregation Reference

Aggregation Pipeline Quick Reference (page 37) Quick reference card for aggregation pipeline.

Aggregation Commands (page 43) The reference for the data aggregation commands, which provide the interfaces to MongoDB's aggregation capability.

Aggregation Commands Comparison (page 44) A comparison of `group`, `mapReduce` and `aggregate` that explores the strengths and limitations of each aggregation modality.

<https://docs.mongodb.org/manual/reference/operator/aggregation> Aggregation pipeline operations have a collection of operators available to define and manipulate documents in pipeline stages.

Variables in Aggregation Expressions (page 46) Use of variables in aggregation pipeline expressions.

SQL to Aggregation Mapping Chart (page 46) An overview common aggregation operations in SQL and MongoDB using the aggregation pipeline and operators in MongoDB and common SQL statements.

4.3.1 Aggregation Pipeline Quick Reference

On this page

- [Stages](#) (page 37)
- [Expressions](#) (page 38)
- [Accumulators](#) (page 42)

Stages

In the `db.collection.aggregate` method, pipeline stages appear in an array. Documents pass through the stages in sequence. All except the `$out` and `$geoNear` stages can appear multiple times in a pipeline.

```
db.collection.aggregate( [ { <stage> }, ... ] )
```

Name	Description
\$project	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.
\$match	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. \$match uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).
\$redact	Reshapes each document in the stream by restricting the content for each document based on information stored in the documents themselves. Incorporates the functionality of \$project and \$match. Can be used to implement field level redaction. For each input document, outputs either one or zero document.
\$limit	Passes the first <i>n</i> documents unmodified to the pipeline where <i>n</i> is the specified limit. For each input document, outputs either one document (for the first <i>n</i> documents) or zero documents (after the first <i>n</i> documents).
\$skip	Skips the first <i>n</i> documents where <i>n</i> is the specified skip number and passes the remaining documents unmodified to the pipeline. For each input document, outputs either zero documents (for the first <i>n</i> documents) or one document (if after the first <i>n</i> documents).
\$unwind	Deconstructs an array field from the input documents to output a document for <i>each</i> element. Each output document replaces the array with an element value. For each input document, outputs <i>n</i> documents where <i>n</i> is the number of array elements and can be zero for an empty array.
\$group	Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.
\$sample	Randomly selects the specified number of documents from its input.
\$sort	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.
\$geoNear	Returns an ordered stream of documents based on the proximity to a geospatial point. Incorporates the functionality of \$match, \$sort, and \$limit for geospatial data. The output documents include an additional distance field and can include a location identifier field.
\$lookup	Performs a left outer join to another collection in the <i>same</i> database to filter in documents from the “joined” collection for processing.
\$out	Writes the resulting documents of the aggregation pipeline to a collection. To use the \$out stage, it must be the last stage in the pipeline.
\$indexStats	Returns statistics regarding the use of each index for the collection.

Expressions

Expressions can include *field paths and system variables* (page 38), *literals* (page 39), *expression objects* (page 39), and *expression operators* (page 39). Expressions can be nested.

Field Path and System Variables

Aggregation expressions use *field path* to access fields in the input documents. To specify a field path, use a string that prefixes with a dollar sign \$ the field name or the dotted field name, if the field is in embedded document. For example, "\$user" to specify the field path for the user field or "\$user.name" to specify the field path to "user.name" field.

"\$<field>" is equivalent to "\$CURRENT.<field>" where the CURRENT (page 46) is a system variable that defaults to the root of the current object in the most stages, unless stated otherwise in specific stages. CURRENT (page 46) can be rebound.

Along with the CURRENT (page 46) system variable, other *system variables* (page 46) are also available for use in expressions. To use user-defined variables, use \$let and \$map expressions. To access variables in expressions, use

a string that prefixes the variable name with `$$`.

Literals

Literals can be of any type. However, MongoDB parses string literals that start with a dollar sign `$` as a path to a field and numeric/boolean literals in *expression objects* (page 39) as projection flags. To avoid parsing literals, use the `$literal` expression.

Expression Objects

Expression objects have the following form:

```
{ <field1>: <expression1>, ... }
```

If the expressions are numeric or boolean literals, MongoDB treats the literals as projection flags (e.g. `1` or `true` to include the field), valid only in the `$project` stage. To avoid treating numeric or boolean literals as projection flags, use the `$literal` expression to wrap the numeric or boolean literals.

Operator Expressions

Operator expressions are similar to functions that take arguments. In general, these expressions take an array of arguments and have the following form:

```
{ <operator>: [ <argument1>, <argument2> ... ] }
```

If operator accepts a single argument, you can omit the outer array designating the argument list:

```
{ <operator>: <argument> }
```

To avoid parsing ambiguity if the argument is a literal array, you must wrap the literal array in a `$literal` expression or keep the outer array that designates the argument list.

Boolean Expressions Boolean expressions evaluate their argument expressions as booleans and return a boolean as the result.

In addition to the `false` boolean value, Boolean expression evaluates as `false` the following: `null`, `0`, and `undefined` values. The Boolean expression evaluates all other values as `true`, including non-zero numeric values and arrays.

Name	Description
<code>\$and</code>	Returns <code>true</code> only when <i>all</i> its expressions evaluate to <code>true</code> . Accepts any number of argument expressions.
<code>\$or</code>	Returns <code>true</code> when <i>any</i> of its expressions evaluates to <code>true</code> . Accepts any number of argument expressions.
<code>\$not</code>	Returns the boolean value that is the opposite of its argument expression. Accepts a single argument expression.

Set Expressions Set expressions performs set operation on arrays, treating arrays as sets. Set expressions ignores the duplicate entries in each input array and the order of the elements.

If the set operation returns a set, the operation filters out duplicates in the result to output an array that contains only unique entries. The order of the elements in the output array is unspecified.

If a set contains a nested array element, the set expression does *not* descend into the nested array but evaluates the array at top-level.

Name	Description
\$setEquals	Returns true if the input sets have the same distinct elements. Accepts two or more argument expressions.
\$setIntersect	Returns a set with elements that appear in <i>all</i> of the input sets. Accepts any number of argument expressions.
\$setUnion	Returns a set with elements that appear in <i>any</i> of the input sets. Accepts any number of argument expressions.
\$setDifference	Returns a set with elements that appear in the first set but not in the second set; i.e. performs a relative complement ⁹ of the second set relative to the first. Accepts exactly two argument expressions.
\$setIsSubset	Returns true if all elements of the first set appear in the second set, including when the first set equals the second set; i.e. not a strict subset ¹⁰ . Accepts exactly two argument expressions.
\$anyElement	Returns true if <i>any</i> elements of a set evaluate to true; otherwise, returns false. Accepts a single argument expression.
\$allElement	Returns true if <i>no</i> element of a set evaluates to false, otherwise, returns false. Accepts a single argument expression.

Comparison Expressions Comparison expressions return a boolean except for \$cmp which returns a number.

The comparison expressions take two argument expressions and compare both value and type, using the *specified BSON comparison order* for values of different types.

Name	Description
\$cmp	Returns: 0 if the two values are equivalent, 1 if the first value is greater than the second, and -1 if the first value is less than the second.
\$eq	Returns true if the values are equivalent.
\$gt	Returns true if the first value is greater than the second.
\$gte	Returns true if the first value is greater than or equal to the second.
\$lt	Returns true if the first value is less than the second.
\$lte	Returns true if the first value is less than or equal to the second.
\$ne	Returns true if the values are <i>not</i> equivalent.

Arithmetic Expressions Arithmetic expressions perform mathematic operations on numbers. Some arithmetic expressions can also support date arithmetic.

⁹[http://en.wikipedia.org/wiki/Complement_\(set_theory\)](http://en.wikipedia.org/wiki/Complement_(set_theory))

¹⁰<http://en.wikipedia.org/wiki/Subset>

Name	Description
\$abs	Returns the absolute value of a number.
\$add	Adds numbers to return the sum, or adds numbers and a date to return a new date. If adding numbers and a date, treats the numbers as milliseconds. Accepts any number of argument expressions, but at most, one expression can resolve to a date.
\$ceil	Returns the smallest integer greater than or equal to the specified number.
\$divide	Returns the result of dividing the first number by the second. Accepts two argument expressions.
\$exp	Raises e to the specified exponent.
\$floor	Returns the largest integer less than or equal to the specified number.
\$ln	Calculates the natural log of a number.
\$log	Calculates the log of a number in the specified base.
\$log10	Calculates the log base 10 of a number.
\$mod	Returns the remainder of the first number divided by the second. Accepts two argument expressions.
\$multiply	Multiplies numbers to return the product. Accepts any number of argument expressions.
\$pow	Raises a number to the specified exponent.
\$sqrt	Calculates the square root.
\$subtract	Returns the result of subtracting the second value from the first. If the two values are numbers, return the difference. If the two values are dates, return the difference in milliseconds. If the two values are a date and a number in milliseconds, return the resulting date. Accepts two argument expressions. If the two values are a date and a number, specify the date argument first as it is not meaningful to subtract a date from a number.
\$trunc	Truncates a number to its integer.

String Expressions String expressions, with the exception of \$concat, only have a well-defined behavior for strings of ASCII characters.

\$concat behavior is well-defined regardless of the characters used.

Name	Description
\$concat	Concatenates any number of strings.
\$substr	Returns a substring of a string, starting at a specified index position up to a specified length. Accepts three expressions as arguments: the first argument must resolve to a string, and the second and third arguments must resolve to integers.
\$toLower	Converts a string to lowercase. Accepts a single argument expression.
\$toUpper	Converts a string to uppercase. Accepts a single argument expression.
\$strcasecmp	Performs case-insensitive string comparison and returns: 0 if two strings are equivalent, 1 if the first string is greater than the second, and -1 if the first string is less than the second.

Text Search Expressions

Name	Description
\$meta	Access text search metadata.

Array Expressions

Name	Description
\$arrayElemAt	Returns the element at the specified array index.
\$concatArrays	Concatenates arrays to return the concatenated array.
\$filter	Selects a subset of the array to return an array with only the elements that match the filter condition.
\$isArray	Determines if the operand is an array. Returns a boolean.
\$size	Returns the number of elements in the array. Accepts a single expression as argument.
\$slice	Returns a subset of an array.

	Name	Description
Variable Expressions	\$map	Applies a subexpression to each element of an array and returns the array of resulting values in order. Accepts named parameters.
	\$let	Defines variables for use within the scope of a subexpression and returns the result of the subexpression. Accepts named parameters.

	Name	Description
Literal Expressions	\$literal	Return a value without parsing. Use for values that the aggregation pipeline may interpret as an expression. For example, use a \$literal expression to a string that starts with a \$ to avoid parsing a field path.

	Name	Description
Date Expressions	\$dayOfYear	Returns the day of the year for a date as a number between 1 and 366 (leap year).
	\$dayOfMonth	Returns the day of the month for a date as a number between 1 and 31.
	\$dayOfWeek	Returns the day of the week for a date as a number between 1 (Sunday) and 7 (Saturday).
	\$year	Returns the year for a date as a number (e.g. 2014).
	\$month	Returns the month for a date as a number between 1 (January) and 12 (December).
	\$week	Returns the week number for a date as a number between 0 (the partial week that precedes the first Sunday of the year) and 53 (leap year).
	\$hour	Returns the hour for a date as a number between 0 and 23.
	\$minute	Returns the minute for a date as a number between 0 and 59.
	\$second	Returns the seconds for a date as a number between 0 and 60 (leap seconds).
	\$millisecond	Returns the milliseconds of a date as a number between 0 and 999.
	\$dateToString	Returns the date as a formatted string.

	Name	Description
Conditional Expressions	\$cond	A ternary operator that evaluates one expression, and depending on the result, returns the value of the other two expressions. Accepts either three expressions in an ordered list or three named parameters.
	\$ifNull	Returns either the non-null result of the first expression or the result of the second expression if the first expression results in a null result. Null result encompasses instances of undefined values or missing fields. Accepts two expressions as arguments. The result of the second expression can be null.

Accumulators

Changed in version 3.2: Some accumulators are now available in the \$project stage. In previous versions of MongoDB, accumulators are available only for the \$group stage.

Accumulators, when used in the \$group stage, maintain their state (e.g. totals, maximums, minimums, and related data) as documents progress through the pipeline.

When used in the \$group stage, accumulators take as input a single expression, evaluating the expression once for each input document, and maintain their state for the group of documents that share the same group key.

When used in the \$project stage, the accumulators do not maintain their state. When used in the \$project stage, accumulators take as input either a single argument or multiple arguments.

Name	Description
\$sum	Returns a sum of numerical values. Ignores non-numeric values. Changed in version 3.2: Available in both \$group and \$project stages.
\$avg	Returns an average of numerical values. Ignores non-numeric values. Changed in version 3.2: Available in both \$group and \$project stages.
\$first	Returns a value from the first document for each group. Order is only defined if the documents are in a defined order. Available in \$group stage only.
\$last	Returns a value from the last document for each group. Order is only defined if the documents are in a defined order. Available in \$group stage only.
\$max	Returns the highest expression value for each group. Changed in version 3.2: Available in both \$group and \$project stages.
\$min	Returns the lowest expression value for each group. Changed in version 3.2: Available in both \$group and \$project stages.
\$push	Returns an array of expression values for each group. Available in \$group stage only.
\$addToSet	Returns an array of <i>unique</i> expression values for each group. Order of the array elements is undefined. Available in \$group stage only.
\$stdDevPop	Returns the population standard deviation of the input values. Changed in version 3.2: Available in both \$group and \$project stages.
\$stdDevSample	Returns the sample standard deviation of the input values. Changed in version 3.2: Available in both \$group and \$project stages.

4.3.2 Aggregation Commands

On this page

- [Aggregation Commands](#) (page 43)
- [Aggregation Methods](#) (page 43)

Aggregation Commands

Name	Description
aggregate	Performs <i>aggregation tasks</i> (page 9) 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 <i>map-reduce</i> (page 25) aggregation for large data sets.

Aggregation Methods

Name	Description
db.collection.aggregate()	(Provides access to the <i>aggregation pipeline</i> (page 9).
db.collection.group()	Groups documents in a collection by the specified key and performs simple aggregation.
db.collection.mapReduce()	(Performs <i>map-reduce</i> (page 25) aggregation for large data sets.

4.3.3 Aggregation Commands Comparison

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

	aggregate	mapReduce	group
Description	<p>New in version 2.2.</p> <p>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 <code>\$group</code>, <code>\$match</code>, and <code>\$sort</code>. See https://docs.mongodb.org/manual/reference/operator/aggregation for more information on the pipeline operators.</p>	<p>Implements the Map-Reduce aggregation for processing large data sets.</p>	<p>Provides grouping functionality. Is slower than the <code>aggregate</code> command and has less functionality than the <code>mapReduce</code> command.</p>
Key Features	<p>Pipeline operators can be repeated as needed.</p> <p>Pipeline operators need not produce one output document for every input document. Can also generate new documents or filter out documents.</p>	<p>In addition to grouping operations, can perform complex aggregation tasks as well as perform incremental aggregation on continuously growing datasets. See <i>Map-Reduce Examples</i> (page 28) and <i>Perform Incremental Map-Reduce</i> (page 30).</p>	<p>Can either group by existing fields or with a custom <code>keyf</code> JavaScript function, can group by calculated fields. See <code>group</code> for information and example using the <code>keyf</code> function.</p>
Flexibility	<p>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 <code>\$project</code> pipeline operator. See <code>\$project</code> for more information as well as https://docs.mongodb.org/manual/reference/operator/aggregation for more information on all the available pipeline operators.</p>	<p>Custom <code>map</code>, <code>reduce</code> and <code>finalize</code> JavaScript functions offer flexibility to aggregation logic. See <code>mapReduce</code> for details and restrictions on the functions.</p>	<p>Custom <code>reduce</code> and <code>finalize</code> JavaScript functions offer flexibility to grouping logic. See <code>group</code> for details and restrictions on these functions.</p>
Output Results	<p>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 <i>BSON Document size</i> 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.</p>	<p>Returns results in various options (inline, new collection, merge, replace, reduce). See <code>mapReduce</code> for details on the output options. Changed in version 2.2: Provides much better support for sharded map-reduce output than previous versions.</p>	<p>Returns results inline as an array of grouped items. The result set must fit within the <i>maximum BSON document size limit</i>. 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.</p>
Sharding Notes	<p>Supports non-sharded and sharded input collections.</p>	<p>Supports non-sharded and sharded input collections. Prior to 2.4, JavaScript code executed in a single thread. See <i>Map-Reduce</i> (page 25) and <code>mapReduce</code>.</p>	<p>Does not support sharded collection. Prior to 2.4, JavaScript code executed in a single thread. See <code>group</code>.</p>
More Information	<p>See <i>Aggregation Pipeline</i> (page 9) and <code>aggregate</code>.</p>	<p>See <i>Map-Reduce</i> (page 25) and <code>mapReduce</code>.</p>	
4.3. Aggregation Reference			45

4.3.4 Variables in Aggregation Expressions

On this page

- [User Variables](#) (page 46)
- [System Variables](#) (page 46)

Aggregation expressions (page 38) 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 46) the same as ROOT (page 46). CURRENT (page 46) is modifiable. However, since <code>\$<field></code> is equivalent to <code>\$\$CURRENT.<field></code> , rebinding CURRENT (page 46) changes the meaning of <code>\$</code> accesses.
DESCEND	One of the allowed results of a <code>\$redact</code> expression.
PRUNE	One of the allowed results of a <code>\$redact</code> expression.
KEEP	One of the allowed results of a <code>\$redact</code> expression.

See also:

`$let`, `$redact`, `$map`

4.3.5 SQL to Aggregation Mapping Chart

On this page

- [Examples](#) (page 47)
- [Additional Resources](#) (page 49)

The *aggregation pipeline* (page 9) allows MongoDB to provide native aggregation capabilities that corresponds to many common data aggregation operations in SQL.

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; <i>however</i> , 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 } ]
}
```

SQL Example	MongoDB Example	Description
<pre>SELECT COUNT(*) AS count FROM orders</pre>	<pre>db.orders.aggregate([{ \$group: { _id: null, count: { \$sum: 1 } } }])</pre>	<p>Count all records from orders</p>
<pre>SELECT SUM(price) AS total FROM orders</pre>	<pre>db.orders.aggregate([{ \$group: { _id: null, total: { \$sum: "\$price" } } }])</pre>	<p>Sum the price field from orders</p>
<pre>SELECT cust_id, SUM(price) AS total FROM orders GROUP BY cust_id</pre>	<pre>db.orders.aggregate([{ \$group: { _id: "\$cust_id", total: { \$sum: "\$price" } } }])</pre>	<p>For each unique cust_id, sum the price field.</p>
<pre>SELECT cust_id, SUM(price) AS total FROM orders GROUP BY cust_id ORDER BY total</pre>	<pre>db.orders.aggregate([{ \$group: { _id: "\$cust_id", total: { \$sum: "\$price" } } }, { \$sort: { total: 1 } }])</pre>	<p>For each unique cust_id, sum the price field, results sorted by sum.</p>
<pre>SELECT cust_id, ord_date, SUM(price) AS total FROM orders GROUP BY cust_id, ord_date</pre>	<pre>db.orders.aggregate([{ \$group: { _id: { cust_id: "\$cust_id", ord_date: { month: { \$month: "\$ord_date" }, day: { \$dayOfMonth: "\$ord_date" }, year: { \$year: "\$ord_date" } } }, total: { \$sum: "\$price" } } }])</pre>	<p>For each unique cust_id, ord_date grouping, sum the price field. Excludes the time portion of the date.</p>

```
SELECT cust_id,
       count(*)
FROM orders
```

```
db.orders.aggregate( [
  {
    $group: {
```

For cust_id with multiple records, return the cust_id and the corresponding record count.

Additional Resources

- [MongoDB and MySQL Compared](#)¹¹
- [Quick Reference Cards](#)¹²
- [MongoDB Database Modernization Consulting Package](#)¹³

¹¹<http://www.mongodb.com/mongodb-and-mysql-compared?jmp=docs>

¹²<https://www.mongodb.com/lp/misc/quick-reference-cards?jmp=docs>

¹³https://www.mongodb.com/products/consulting?jmp=docs#database_modernization

Additional Resources

- MongoDB Analytics: Learn Aggregation by Example: Exploratory Analytics and Visualization Using Flight Data¹
- MongoDB for Time Series Data: Analyzing Time Series Data Using the Aggregation Framework and Hadoop²
- The Aggregation Framework³
- Webinar: Exploring the Aggregation Framework⁴
- Quick Reference Cards⁵

¹<http://www.mongodb.com/presentations/mongodb-analytics-learn-aggregation-example-exploratory-analytics-and-visualization?jmp=docs>

²<http://www.mongodb.com/presentations/mongodb-time-series-data-part-2-analyzing-time-series-data-using-aggregation-framework?jmp=docs>

³<https://www.mongodb.com/presentations/aggregation-framework-0?jmp=docs>

⁴<https://www.mongodb.com/webinar/exploring-the-aggregation-framework?jmp=docs>

⁵<https://www.mongodb.com/lp/misc/quick-reference-cards?jmp=docs>