CHAPTER 7

Tuning Aggregation Pipelines

When getting started with MongoDB, most developers will begin with the basic CRUD operations (Create-Read-Update-Delete) that they are familiar with from other databases. `insert`, `find`, `update`, and `delete` operations will indeed form the backbone of most applications. However, in almost all applications, complex data retrieval and manipulation requirements will exist that exceed what is possible with the basic MongoDB commands.

The MongoDB `find()` command is versatile and easy to use, but the aggregation framework allows you to take it to the next level. Aggregation pipelines can do anything a `find()` operation can do and much more. As MongoDB themselves like to say in blogs, marketing materials, and even on T-shirts: *aggregate is the new find.*

Aggregation pipelines allow you to simplify your application code by reducing logic that might otherwise require multiple find operations and complex data manipulation. When leveraged correctly, a single aggregation can replace many queries and their associated network round trip times.

As you may recall from earlier chapters, an essential part of tuning your application is to ensure that as much work as possible takes place on the database. Aggregation allows you to take data transformation logic that would usually sit on the application and move it to the database. A properly tuned aggregation pipeline can consequently massively outperform alternative solutions.

However, with all the added benefits that using aggregations brings, it also creates a new set of tuning challenges. In this chapter, we will make sure you have all the knowledge required to leverage and tune aggregation pipelines.

© Guy Harrison, Michael Harrison 2021
G. Harrison and M. Harrison, *MongoDB Performance Tuning*, https://doi.org/10.1007/978-1-4842-6879-7_7

Tuning Aggregation Pipelines

In order to effectively tune aggregation pipelines, we must first be able to effectively identify which aggregations are in need of tuning and what aspects can be improved. As with `find()` queries, the `explain()` command is our best friend here. As you may recall from earlier chapters, to examine the execution plan of a query, we add the `.explain()` method after the collection name. For example, to explain a `find()`, we might use the following command:

```
db.customers.
  explain().
  find(
    { Country: 'Japan', LastName: 'Smith' },
    { _id: 0, FirstName: 1, LastName: 1 }
  ).
  sort({ FirstName: -1 }).
  limit(3);
```

We can explain an aggregation pipeline in the same way:

```
db.customers.explain().aggregate([
  { $match: {
      Country: 'Japan',
      LastName: 'Smith',
    } },
  {  $project: {
      _id: 0,
      FirstName: 1,
      LastName: 1,
    } },
  { $sort: {
      FirstName: -1,
    } },
  { $limit: 3 } ] );
```

However, there are significant differences between the execution plan from a `find()` command and one from an `aggregate()`.

When running the `explain()` against a standard find command, we can see information about the execution by looking at the `queryPlanner.winningPlan` object.

The `explain()` output for an aggregation pipeline is similar but also critically different. Firstly, the `queryPlanner` object we are previously used to now resides within a new object, which resides within an array called `stages`. The `stages` array contains each of the aggregation stages as individual objects. For example, the aggregation we looked at earlier would have the following simplified explain output:

```
{
  "stages": [
    {"$cursor": {
        "queryPlanner": {
          // . . .
          "winningPlan": {
            "stage": "PROJECTION_SIMPLE",
            //      . . .
            "inputStage": {
              "stage": "FETCH",
              //    . . .
              "inputStage": {
                "stage": "IXSCAN",
                . . .
              } } },
          "rejectedPlans": []
        } } },
    { "$sort": {
        "sortKey": {
          "FirstName": -1
        },
        "limit": 3
      } } ],
 . . .
}
```

Inside the execution plan for an aggregation pipeline, the `queryPlanner` stage reveals the initial data access operations required to bring data into the pipeline. This will usually represent the operation that supports an initial `$match` operation or – if there is no `$match` condition specified – a collection scan to retrieve all data from the collection.

The `stages` array shows information about each subsequent step in the aggregation pipeline. Note that MongoDB can merge and reorder aggregation stages during execution so these stages might not match the stages you have in your raw pipeline definition – more on this in the next section.

We've written a helper script to simplify the interpretation of aggregation execution plans within our tuning script package.[1] The method `mongoTuning.aggregationExecutionStats()` will provide a top-level summary of the time taken by each step. Here's an example of using `aggregationExecutionSteps`:

```
mongo> var exp = db.customers.explain('executionStats').aggregate([
...   { $match:{
...         "Country":{ $eq:"Japan" }}
...   },
...   { $group:{ _id:{ "City":"$City"  },
...              "count":{$sum:1} }
...   },
...   { $sort:{  "_id.City":-1 }},
...   { $limit:  10 },
... ] );

mongo>  mongoTuning.aggregationExecutionStats(exp);

1  IXSCAN ( Country_1_LastName_1 ms:0 keys:21368 nReturned:21368)
2  FETCH ( ms:13 docsExamined:21368 nReturned:21368)
3  PROJECTION_SIMPLE ( ms:15 nReturned:21368)
4  $GROUP ( ms:70 returned:31)
5  $SORT ( ms:70 returned:10)

Totals:  ms: 72  keys: 21368  Docs: 21368
```

[1]See the introduction for information on how to use the tuning package.

Optimizing Aggregation Ordering

An aggregation is constructed from a series of stages, represented by an array of documents which are executed in order from first to last. The output from each stage is passed to the next stage for processing, with the initial input being an entire collection.

The sequential nature of these stages is the reason aggregations are referred to as pipelines: data flows through the pipes, being filtered and transformed at each stage until finally exiting the pipeline as a result. The easiest way to optimize these pipelines is to reduce the amount of data as early as possible; this will reduce the amount of work done by each successive step. It logically follows that the stages in your aggregation that will perform the most work should operate on as little data as possible, with as much filtering as possible being performed in earlier stages.

Tip When constructed aggregation pipelines, filter early and filter often! The earlier data is removed from a pipeline, the lower the overall data processing load for MongoDB.

MongoDB will automatically resequence the order of operations in a pipeline to optimize performance – we'll see an example of some optimizations in the next section. However, for complex pipelines, you may need to set the order yourself.

One such case where automatic reordering is not possible is aggregations using `$lookup`. The `$lookup` stage allows you to join two collections. If you're joining two collections, you may have a choice between filtering before or after the join, and in this case, it is very important to try and reduce the size of the data *before* the join operation, because for each document that is passed into a lookup operation, MongoDB must attempt to find a matching document in a separate collection. Every document we can filter out before the lookout will reduce the number of lookups that need to occur. It's an obvious but critical optimization.

Let's look at an example aggregation which generates a "top 5" list of product purchases:

```
db.lineitems.aggregate([
  { $group:{ _id:{ "orderId":"$orderId" ,"prodId":"$prodId"  },
             "itemCount-sum":{$sum:"$itemCount"} } },
  { $lookup:
    { from:         "orders",  localField:"_id.orderId",
      foreignField: "_id",             as:"orders"
    }  },
  { $lookup:
    { from:         "customers", localField:"orders.customerId",
      foreignField: "_id",                as:"customers"
    }  },
  { $lookup:
    { from:         "products",  localField:"_id.prodId",
      foreignField: "_id",                as:"products"
    }  },
  { $sort:{  "count":-1 }},
  { $limit:  5 },
],{allowDiskUse: true});
```

This is quite a big aggregation pipeline. In fact, without the `allowDiskUse:true` flag, it will generate an out of memory error; we will cover why this error occurs later in the chapter.

Note that we join `orders`, `customers`, and `products` *before* we sort the results and limit the output. As a result, we have to perform all three join lookups for each lineItem. We could – and should – position the `$sort` and `$limit` directly after the `$group` operation:

```
db.lineitems.aggregate([
  { $group:{ _id:{ "orderId":"$orderId" ,"prodId":"$prodId"  },
             "itemCount-sum":{$sum:"$itemCount"} } },
  { $sort:{  "count":-1 }},
  { $limit:  5 },
  { $lookup:
    { from:         "orders",  localField:"_id.orderId",
      foreignField: "_id",             as:"orders"
    }  },
  { $lookup:
    { from:         "customers", localField:"orders.customerId",
      foreignField: "_id",                as:"customers"
    }  },
  { $lookup:
    { from:          "products",  localField:"_id.prodId",
      foreignField: "_id",               as:"products"
    }  }
],{allowDiskUse: true});
```

The difference in performance is striking. By moving the `$sort` and `$limit` a few lines earlier, we have created a much more efficient and scalable solution. Figure 7-1 illustrates the performance improvement obtained by moving `$limit` earlier in the pipeline.

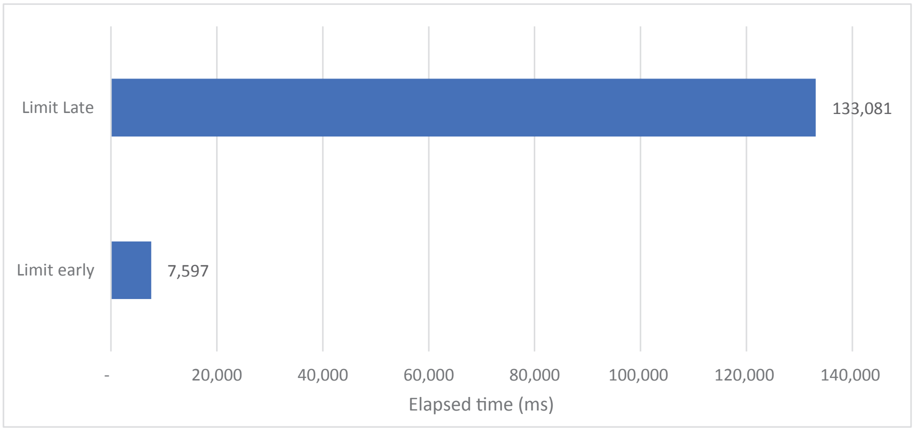

Figure 7-1. *Effect of moving limit clause earlier in a `$lookup` pipeline*

Tip Take care to sequence aggregation pipelines to eliminate documents earlier rather than later. The earlier data is eliminated from a pipeline, the less work will be required in later pipelines.

Automatic Pipeline Optimizations

MongoDB will perform some optimizations on aggregation pipelines to improve performance. The exact optimizations change from release to release, and when running an aggregation through a driver or the MongoDB shell, there is no obvious sign that optimization has occurred. In fact, the only way for you to be certain is to check the query plan using `explain()`. If you are surprised to see that your aggregation explain does not match what you just sent to MongoDB, do not be alarmed. That's the optimizer doing its job.

Let's run a few aggregations and observe how MongoDB decides to improve the pipeline using explain(). Here is a very badly constructed aggregation pipeline:

```
> var explain = db.listingsAndReviews.explain("executionStats").
   aggregate([
    {$match: {"property_type" : "House"}},
    {$match: {"bedrooms" : 3}},
    {$limit: 100},
    {$limit: 5},
    {$skip: 3},
    {$skip: 2}
]);
```

You can probably guess what is going to happen here. The multiple $match, $limit, and $skip stages, when placed one after another, can be merged into a single stage without altering the result. The two $match stages can be merged using $and. The result of two $limit stages is always the smaller limit value, and the effect of two $skips is the sum of $skip values. Although the results from the pipeline are unchanged, we can observe the effect of optimization in the query plan. Here is a simplified view of our merged stages outputted from our mongoTuning.aggregationExecutionStats command:

```
1  COLLSCAN ( ms:0 docsExamined:525 nReturned:5)
2  LIMIT ( ms:0 nReturned:5)
3  $SKIP ( ms:0 returned:0)

Totals:  ms: 1  keys: 0  Docs: 525
```

As you can see, MongoDB has merged the six steps from our pipeline into just three operations.

There are a few other smart merges that MongoDB can perform on your behalf. If you have a $lookup stage where you immediately $unwind the joined documents, MongoDB will merge the $unwind into the $lookup. For example, this aggregation joins a user with their blog comments:

```
> var explain = db.users.explain("executionStats").aggregate([
 { $lookup: {
     from: "comments",
      as: "comments",
      localField: "email",
      foreignField: "email"
 }},
 { $unwind: "$comments"}
]);
```

The `$lookup` and `$unwind` will become a single stage in the execution, which will eliminate the creation of large joined documents that will immediately be unwound into smaller documents. The execution plan will look like the following snippet:

```
> mongoTuning.aggregationExecutionStats(explain);


1  COLLSCAN ( ms:9 docsExamined:183 nReturned:183)
2  $LOOKUP ( ms:4470 returned:50146)

Totals:  ms: 4479  keys: 0  Docs: 183
```

Similarly, `$sort` and `$limit` stages will be merged, allowing the `$sort` to only maintain the limited number of documents instead of its entire input. Here is an example of such an optimization. The query

```
> var explain = db.users.explain("executionStats").
 aggregate([
  { $sort: {year: -1}},
  { $limit: 1}
 ]);
> mongoTuning.aggregationExecutionStats(explain);
```

will result in a single stage in the explain output:

```
1  COLLSCAN ( ms:0 docsExamined:183 nReturned:183)
2  SORT ( ms:0 nReturned:1)

Totals:  ms: 0  keys: 0  Docs: 183
```

There is another important optimization that does not involve merging or moving stages in your pipeline. If your aggregation only requires a subset of document attributes, MongoDB may add a projection to remove all unused fields. This reduces the size of the datasets passing through the pipeline. For example, the following aggregation only actually uses two fields – Country and City:

```
mongo> var exp = db.customers.
...   explain('executionStats').
...   aggregate([
...     { $match: { Country: 'Japan' } },
...     { $group: { _id: { City: '$City' } } }
...   ]);
```

MongoDB inserts a projection into the execution plan to eliminate all unneeded attributes:

```
mongo> mongoTuning.aggregationExecutionStats(exp);

1  IXSCAN ( Country_1_LastName_1 ms:4 keys:21368 nReturned:21368)
2  FETCH ( ms:12 docsExamined:21368 nReturned:21368)
3  PROJECTION_SIMPLE ( ms:12 nReturned:21368)
4  $GROUP ( ms:61 returned:31)

Totals:  ms: 68  keys: 21368  Docs: 21368
```

So, we now know MongoDB will effectively add and merge stages to improve your pipeline. There are also some cases where the optimizer will reorder your stages. The most important of these is reordering of `$match` operations.

If a pipeline contains a `$match` after a stage that will project new fields into the document (such as `$group`, `$project`, `$unset`, `$addFields`, or `$set`) and if the `$match` stage does not require the projected fields, MongoDB will move that `$match` stage earlier in the pipeline. This reduces the number of documents that must be processed in later stages.

For instance, consider this aggregation:

```
var exp=db.customers.explain("executionStats").aggregate([
  { '$group': {
      '_id': '$Country',
      'numCustomers': {
        '$sum': 1
      } } },
  { '$match': {
      '$or': [
        { '_id': 'Netherlands' },
        { '_id': 'Sudan' },
        { '_id': 'Argentina' } ] } }
]);
```

Prior to MongoDB 4.0, MongoDB would perform the exact steps specified in the pipeline – perform a `$group` operation, and then use `$match` to eliminate countries other than those specified. This is wasteful, especially since we have an index on Country which could be used to rapidly find the documents required.

However, in modern versions of MongoDB, the `$match` operation will be relocated prior to the `$group` operation, reducing the number of documents that need to be grouped and allowing the index to be used. Here is the resulting execution plan:

```
mongo> mongoTuning.aggregationExecutionStats(exp);

1   IXSCAN ( Country_1_LastName_1 ms:1 keys:13720 nReturned:13717)
2   PROJECTION_COVERED ( ms:1 nReturned:13717)
3   SUBPLAN ( ms:1 nReturned:13717)
4   $GROUP ( ms:20 returned:3)
```

The MongoDB automatic optimizations are one of the unsung heroes of recent MongoDB releases, improving performance without requiring any work on your part. Understanding how these optimizations work should empower you to make good decisions when creating your aggregations as well as understand anomalies in the execution plans

For more information on exactly what will occur in the optimization for any given MongoDB release, refer to the official documentation at `http://bit.ly/MongoAggregatePerf`.

Optimizing Multi-collection Joins

One of the really significant capabilities provided only by the aggregation framework is the ability to merge data from multiple collections. The most significant and mature capability is found in the $lookup operator, which allows a join between two collections.

In Chapter 4, we experimented with some alternative schema designs, some of which would frequently require joins to assemble information. For instance, we created a schema in which customers and orders were held in separate collections. In this case, we'd use $lookup to join the customer data and order data like this:

```
db.customers.aggregate([
  { $lookup:
     { from:         "orders",
       localField:   "_id",
       foreignField: "customerId",
       as:           "orders"
     }
  },
]);
```

This statement embeds an array of orders within each customer document. The _id attribute in the customer document is matched to the customerId attribute in the orders collection.

It's not too challenging to construct a join using $lookup, but there are some definite potential issues concerning join performance. Because the $lookup function is executed once for each document in the source data, it's essential that the $lookup be quick. In practice, this means that the $lookup should be supported by an index. In the preceding case, we would need to be sure that there is an index on the customerId attribute within the orders collection.

Unfortunately, the explain() command doesn't help us to determine if the join is efficient or that an index has been used. For instance, here is the explain output (using mongoTuning.aggregationExecutionStats) from the preceding operation:

```
1  COLLSCAN ( ms:10 docsExamined:411121 nReturned:411121)
2  $LOOKUP ( ms:5475 returned:411121)
```

The explain output tells us that we used a collection scan to perform the initial retrieval of customers, but doesn't show us if we used an index within the $lookup stage.

However, if you don't have a supporting index, you will almost certainly notice the performance degradation that results. Figure 7-2 shows how performance degrades as more and more documents are involved in a join. With an index, join performance is efficient and predictable. Without an index, join performance degrades steeply as more documents are added to the join.

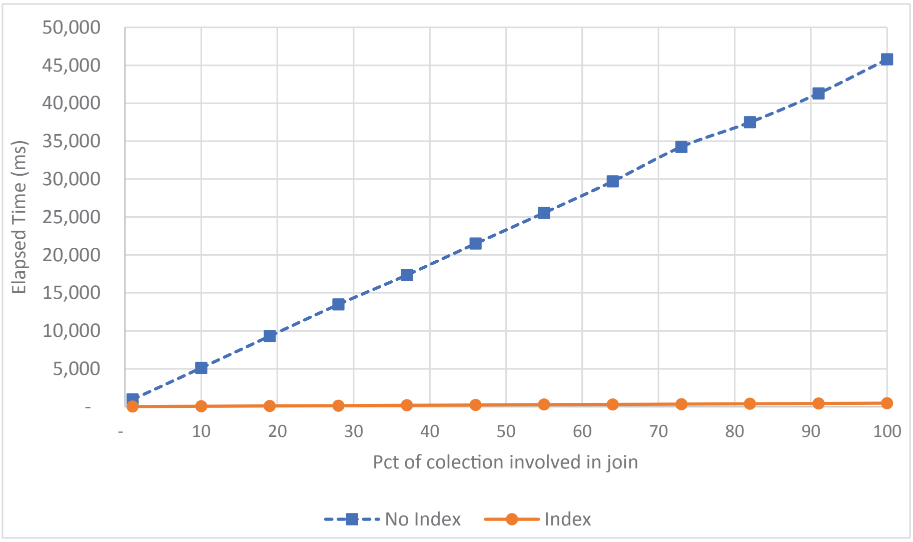

Figure 7-2. *$lookup performance – indexed vs. non-indexed*

Tip Always create an index on the `foreignField` attributes in a `$lookup`, unless the collections are of trivial size.

Join Order

When joining collections, we sometimes have a choice of the order in which we join. For instance, this query joins *from* customers *to* orders:

```
db.customers.aggregate([
  { $lookup:
     { from:         "orders",
       localField:   "_id",
       foreignField: "customerId",
       as:           "orders"
     }
  },
  { $unwind:  "$orders" },
  { $count: "count" },
]);
```

The following query returns the same data, but joins *from* orders *to* customers:

```
db.orders.aggregate([
  { $lookup:
     { from:         "customers",
       localField:   "customerId",
       foreignField: "_id",
       as:           "customer"
     }
  },
  { $count: "count" },
] );
```

These two queries have very different performance characteristics. Although there are indexes to support the $lookup operations in each query, the join from orders to customers results in far more $lookup calls – simply because there are more orders than customers. Consequently, joining from orders to customers takes much longer than the reverse. Figure 7-3 shows the relative performance.

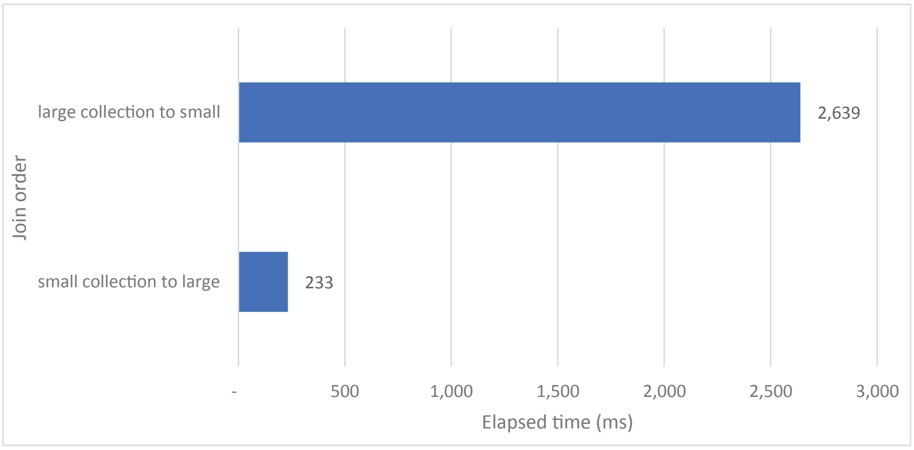

Figure 7-3. *Join order and $lookup performance*

When deciding upon the join order, follow these guidelines:

1. You should try to reduce the amount of data to be joined as much as possible before the join. So if one of the collections is to be filtered, that collection should come first in the join order.
2. If you only have an index to support one of the two join orders, then you should use the join order that has the supporting index.
3. If the preceding two criteria are met for both join orders, then you should try to join from the smallest collection to the largest collection.

Tip When all else is equal, join from a small collection to a large collection, rather than from a large collection to a smaller one.

Optimizing Graph Lookups

Graph databases such as Neo4J specialize in traversing graphs of relationships – such as those you might find in a social network. Many non-graph databases have been incorporating Graph Compute Engines to perform similar tasks. Using older versions of MongoDB, you may have been forced to fetch large amounts of graph data across the network and run some computation on the application level. The process would have been slow and cumbersome. Luckily for us, since MongoDB 3.4, we can perform simple graph traversal using the `$graphLookup` aggregation framework stage.

Imagine you have data stored in MongoDB that represents a social network. In this network, a single user is connected to a large number of other users as friends. These sorts of networks are a common use of graph database. Let's run through an example using the following sample data:

```
db.getSiblingDB("GraphTest").socialGraph.findOne();
{
      "_id" : ObjectId("5a739cda0c31c5f5afcff87f"),
      "person" : 561596,
      "name" : "User# 561596",
      "friends" : [
            94230,
            224410,
            387968,
            406744,
            707890,
            965522,
            1189677,
            1208173
      ]
}
```

Using an aggregation pipeline with the $graphLookup stage, we can expand our social network for an individual user. Here's an example pipeline:

```
db.socialGraph.aggregate([
  {$match:{person:1476767}},
  {$graphLookup: {
      from: "socialGraph",
      startWith: [1476767],
      connectFromField: "friends",
      connectToField: "person",
      maxDepth: 2,
      depthField: "Depth",
      as: "GraphOutput"
    }
  },{$unwind:"$GraphOutput"}
], {allowDiskUse: true});
```

What we are doing here is starting with person 1476767 and then following the elements of the friends array out to two levels, essentially finding "friends of friends."

Increasing the value of the maxDepth field exponentially increases the amount of data we must cope with. You can think of each level of depth as requiring a sort of self-join into the collection. For each document in the initial dataset, we read the collection to find a friend; then for each document in that dataset, read the collection to find those friends; and so on. We stop once we have hit maxDepth connections.

It's clear that if each self-join requires a collection scan, the performance is going to degrade rapidly as we increase the depth of the network. Consequently, it is important to ensure there is an index available for MongoDB to use while traversing the connections. That index should be on the `connectToField` attribute.

Figure 7-4 illustrates the performance of a `$graphLookup` operation with and without an index. Without an index, performance degrades rapidly as we increase the depth of the operation. With an index, the graph lookup is far more scalable and efficient.

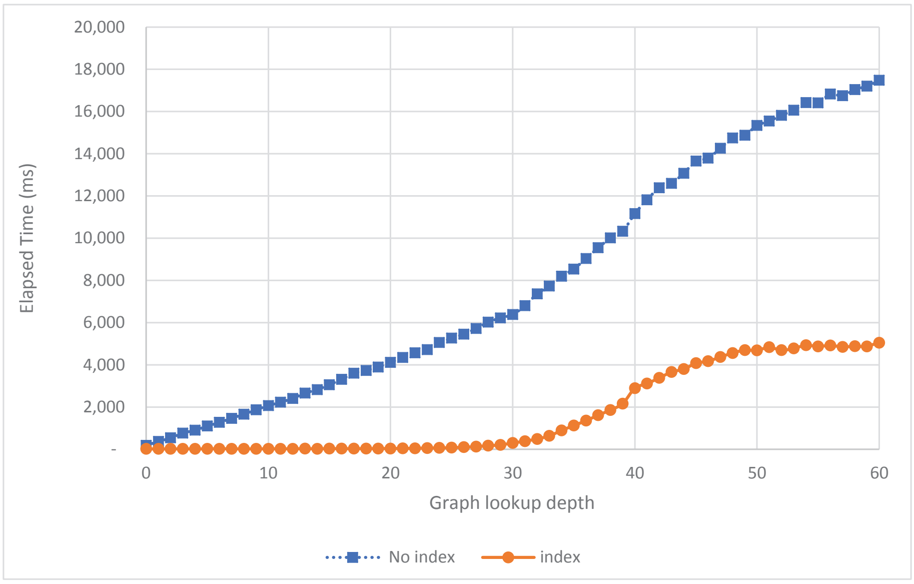

***Figure 7-4.** `$graphLookup` performance with or without indexing*

Tip When performing `$graphLookup` operations, ensure you have an index on the `connectToField` attribute.

Aggregation Memory Utilization

When performing aggregations in MongoDB, there are two important limits to remember that apply to all aggregations regardless of what stages the pipeline is constructed from. Alongside these are some specific limitations that will need to be considered when tuning your application. The two limits that you will always have to keep in mind are the document size limit and the memory usage limit.

In MongoDB, the size limit for a single document is 16MB. This is true for aggregations as well. When performing aggregations, if any of the output documents can exceed this limit, then an error will be thrown. This may not be a problem when performing simple aggregations. However, when grouping, manipulating, unwinding, and joining documents across multiple collections, you will have to consider the growing size of the output documents. An important distinction here is that this limit only applies to documents in the result. For example, if a document exceeds this limit during the pipeline, but is reduced below the limit before the end, no error will be thrown. In addition, MongoDB combines some operations internally to avoid the limit. For instance, if a `$lookup` returns an array that is larger than the limit, but that `$lookup` is followed immediately by an `$unwind`, a document size error will not be issued.

The second limit to keep in mind is the memory usage limit. At each stage in the aggregation pipeline, there is a memory limit by default of 100MB. If this limit is exceeded, MongoDB will produce an error.

MongoDB does provide a way for getting around this limit during aggregations. The `allowDiskUse` option can be used to remove this limit for almost all stages. As you may have guessed, when set to true, this allows MongoDB to create a temporary file on disk to hold some data while aggregating, bypassing the memory limit. You may have noticed this in some of the previous examples. Here is an example of setting this limit to true in one of our previous aggregations:

```
db.customers.aggregate([
  { '$group': {
      '_id': '$Country',
      'numCustomers': {
        '$sum': 1
      } } },
  { '$match': {
      '$or': [
        { '_id': 'Netherlands' },
        { '_id': 'Sudan' },
        { '_id': 'Argentina' } ] } }
],{allowDiskUse:true});
```

As we said, the `allowDiskUse` option will bypass the limit for *almost* all stages. Unfortunately, there are still a few stages that are limited to 100MB even with `allowDiskUse` set to true. The two accumulators `$addToSet` and `$push` will not spill to disk, because these accumulators could add huge amounts of data into the next stage if not properly optimized.

There is currently no obvious work around for these three limited stages, meaning you will have to optimize the query and pipeline itself to ensure you do not run into this limit and receive an error from MongoDB.

To avoid hitting these memory limits, you should think about how much data you actually need to fetch. Ask yourself if you're using all the fields being returned from the query, and could the data be represented more succinctly? Removing unnecessary attributes from intermediary documents is an easy and powerful way to reduce memory use.

If all else fails or if you want to avoid the performance drag when data spills to disk, you could try increasing the internal memory limits for these operations. These memory limits are controlled by undocumented parameters of the form "`internal*Bytes`". The three most important of these are

- `internalQueryMaxBlockingSortMemoryUsageBytes`: The maximum memory available to a `$sort` (see the next section for more details)
- `internalLookupStageIntermediateDocumentMaxSizeBytes`: The maximum memory available to a `$lookup` operation
- `internalDocumentSourceGroupMaxMemoryBytes`: The maximum memory available to a `$group` operation

You can adjust these parameters using the `setParameter` command. For instance, to increase sort memory, you could issue this command:

```
db.getSiblingDB("admin").
    runCommand({ setParameter: 1,
    internalQueryMaxBlockingSortMemoryUsageBytes: 1048576000 });
```

We'll discuss this further in the next section in the context of sort optimization. Be very careful when adjusting memory limits, however, since you might hurt the overall performance of your MongoDB cluster if you exceed the memory capacity of its server.

Sorting in Aggregation Pipelines

We looked at optimizing sorting within `find()` operations in Chapter 6. Sorts in aggregation pipelines differ from sorts in a couple of significant ways:

1. An aggregation can exceed the memory limit for a blocking sort by performing a "disk sort." In a disk sort, excess data is written to and from disk during the sort operation.
2. Aggregations might not be able to take advantage of indexed sorting options unless the sort is very early in the pipeline.

Indexed Aggregation Sorts

Similarly to `find()`, aggregations are able to use an index to resolve a sort and thus avoid high memory utilization or a disk sort. However, this usually can only occur if the `$sort` occurs early enough the pipeline to be rolled into the initial data access operation.

For instance, consider this operation in which we sort some data and add a field:

```
mongo> var exp=db.baseCollection.explain('executionStats').
...     aggregate([
...        { $sort:{  d:1 }},
...        {$addFields:{x:0}}
...     ],{allowDiskUse: true});
mongo> mongoTuning.aggregationExecutionStats(exp);

1  IXSCAN ( d_1 ms:97 keys:1000000 nReturned:1000000)
2  FETCH ( ms:500 docsExamined:1000000 nReturned:1000000)
3  $ADDFIELDS ( ms:3316 returned:1000000)

Totals:  ms: 3358  keys: 1000000  Docs: 1000000
```

There is an index on the sorted attribute, and we are able to use that index to optimize the sort. However, if we move the `$addFields` operation before the sort, then the aggregation is unable to utilize the index, and a costly "disk sort" occurs:

```
mongo> var exp=db.baseCollection.explain('executionStats').
...     aggregate([
...        {$addFields:{x:0}},
...         { $sort:{  d:1 }},
... ],{allowDiskUse: true});
```

```
mongo> mongoTuning.aggregationExecutionStats(exp);

1  COLLSCAN ( ms:16 docsExamined:1000000 nReturned:1000000)
2  $ADDFIELDS ( ms:1164 returned:1000000)
3  $SORT ( ms:12125 returned:1000000)

Totals:  ms: 12498  keys: 0  Docs: 1000000
```

Figure 7-5 compares the performance of the two aggregations. By moving the sort to the start of the aggregation pipeline, a costly disk sort was avoided and elapsed time was significantly reduced.

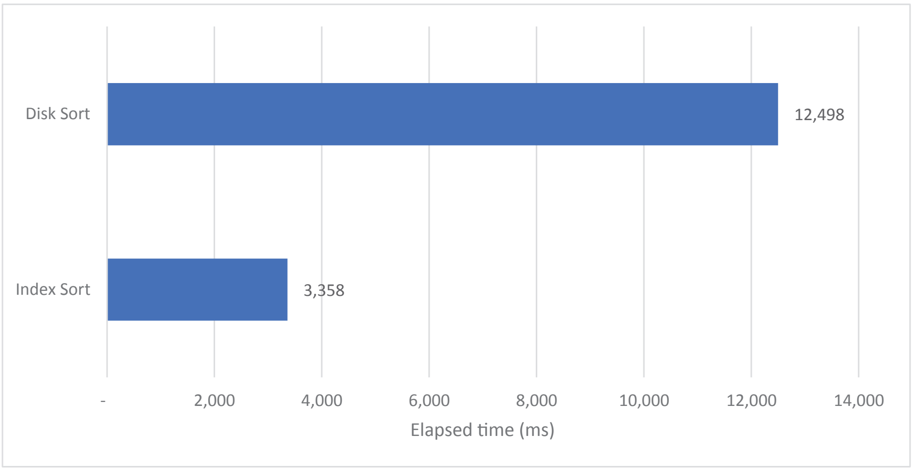

Figure 7-5. *Disk sort vs. indexed sort in an aggregation pipeline*

Tip Move sorts that have a supporting index as early in the aggregation pipeline as possible to avoid an expensive disk sort.

Disk Sorts

If there is no index supporting a sort and the sort exceeds the 100MB limit, then you will receive a `QueryExceededMemoryLimitNoDiskUseAllowed` error:

```
mongo>var exp=db.baseCollection.
...     aggregate([
...        { $sort:{  d:1 }},
...        {$addFields:{x:0}}
...     ],{allowDiskUse: false});
```

```
2020-08-22T15:36:01.890+1000 E  QUERY    [js] uncaught exception: Error:
command failed: {
        "operationTime" : Timestamp(1598074560, 3),
        "ok" : 0,
        "errmsg" : "Error in $cursor stage :: caused by :: Sort exceeded
memory limit of 104857600 bytes, but did not opt in to external sorting.",
        "code" : 292,
        "codeName" : "QueryExceededMemoryLimitNoDiskUseAllowed",
```

If it's possible to use an index to support this sort as outlined in the previous section, then that is usually the best solution. However, in complex aggregation pipelines, this won't always be possible since the data to be sorted may be the result of previous pipeline stages. In this case, we have two options:

1. Use a "disk sort" by specifying `allowDiskUse:true`.
2. Increase the global limit for blocking sorts by changing the `internalQueryMaxBlockingSortMemoryUsageBytes` parameter.

Changing MongoDB default memory parameters should only be undertaken with extreme care, since there is a risk of causing memory starvation on the server, which could make global performance worse. However, 100MB is not a lot of memory in today's world, and so increasing the parameter may be the best option. Here, we increase the maximum sort memory to 1GB:

```
mongo>db.getSiblingDB("admin").
...    runCommand({ setParameter: 1,
internalQueryMaxBlockingSortMemoryUsageBytes: 1048576000 });
```

```
{
        "was" : 104857600,
        "ok" : 1,
…
```

Figure 7-6 shows how performance is improved for the example query when we increased `internalQueryMaxBlockingSortMemoryUsageBytes` to avoid a disk sort.

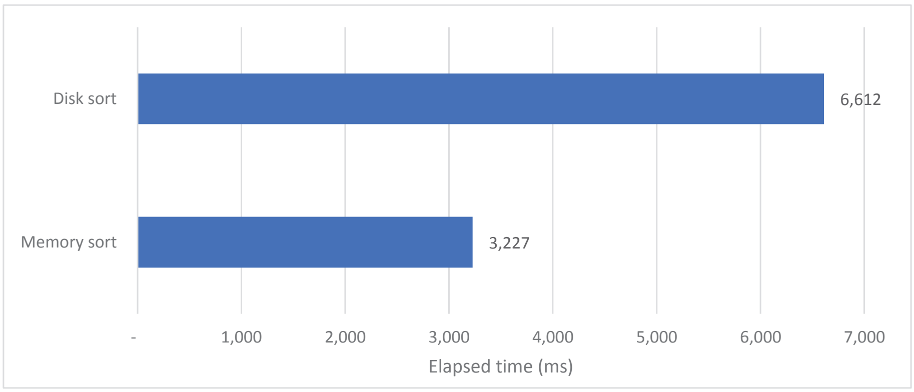

Figure 7-6. *Disk sort vs. memory sort in aggregation*

Another thing to think about with disk sorts is scalability. If you set `diskUsage:true`, then you can rest assured that your query will run even if there is not enough memory to complete the sort. However, when the query switches from memory sort to disk sort, performance will suddenly degrade. In a production context, it might seem like your application suddenly "hits a wall."

Figure 7-7 shows how the switch to a disk sort results in a sudden jump in execution time, compared with the relatively linear trend when there is sufficient memory to support the sort.

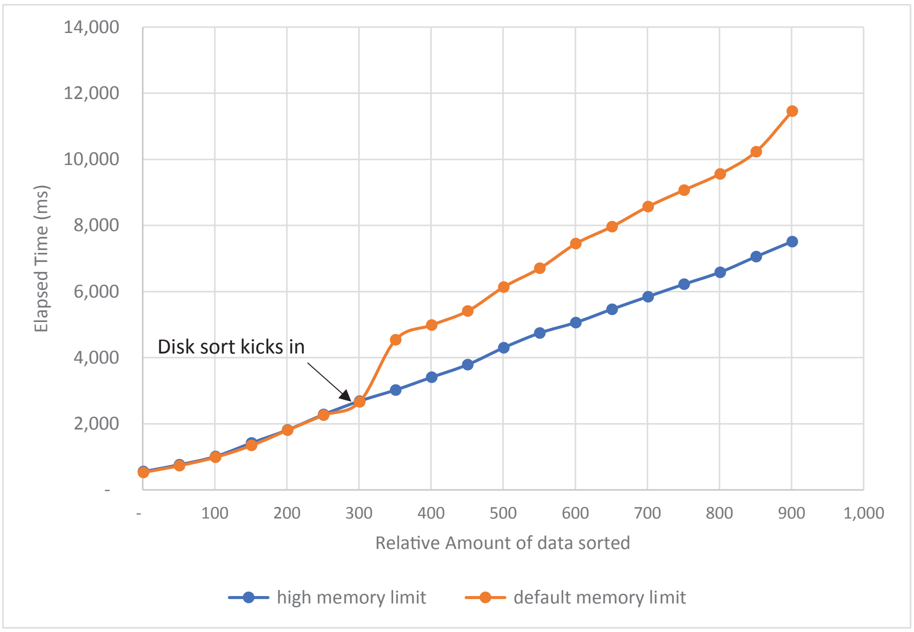

***Figure 7-7.** How disk sorts in aggregation affect scalability*

Tip Disk sorts in aggregation pipelines are expensive and slow. You may wish to increase the default memory limit for aggregation sorting if you want to improve the performance of large aggregation sorts.

Optimizing Views

If you have worked with SQL databases before, you are probably familiar with the concept of *views*. In MongoDB, a view is a sort of synthetic collection that contains the results of an aggregation pipeline. From a query perspective, views look and feel just like a normal collection, except that they are read-only.

The main advantage of creating a view is to simplify and unify application logic by storing complex pipeline definitions in the database.

When it comes to performance, it is important to understand that when a view is created, the result is not stored in memory or copied into a new collection. When you query a view, you are still querying the original collection. MongoDB will take the aggregation pipeline defined for the view and then append your additional query parameters, creating a new pipeline. This gives the appearance of querying the view, but under the hood, a complex aggregation pipeline is still being issued.

Consequently creating a view will not give you a performance advantage when compared to executing the pipeline that defines the view.

Because a view is essentially just an aggregation against a collection, our methods for optimizing a view are the same as for any aggregation. If your view is performing poorly, optimize the pipeline that defines the view using the techniques described earlier in this chapter.

When writing queries against a view, bear in mind that indexes on the underlying collection cannot generally be exploited when executing queries against the view. For instance, consider this view which aggregates product codes by order count:

```
db.createView('productTotals', 'lineitems', [
  { $group: {
      _id: { prodId: '$prodId' },
      'itemCount-sum': { $sum: '$itemCount' }
    }
  },
  { $project: {
      ProdId: '$_id.prodId',
      OrderCount: '$itemCount-sum',
      _id: 0
    }
  }]);
```

We can use this view to find totals for a particular product code:

```
mongo> db.productTotals.find({ ProdId: 83 });
{
  "ProdId": 83,
  "OrderCount": 460051
}
```

However, even if there were an index on `prodId` within the `lineItems` collection, the index will not be used when querying from the view. Even though we are only asking for a single product code, MongoDB will aggregate data from all products before returning results.

Although it's far more tedious, this aggregation pipeline will use an index on `ProdId` and consequently will return data much faster:

```
db.lineitems.aggregate(
  [ {  $match: { prodId: 83 }},
    {  $group: {
         _id: { prodId: '$prodId' },
         'itemCount-sum': { $sum: '$itemCount' }  }
    },
    { $project: {
         ProdId: '$_id.prodId',
         OrderCount: '$itemCount-sum',
         _id: 0
      }
    }
  ] );
```

Tip Views can't always take advantage of indexes on the underlying collection when resolving queries. If you are querying from a view for attributes that are indexed in the underlying collection, you might get better performance from bypassing the view and querying the underlying collection directly.

Materialized Views

As we have discussed, MongoDB views do not improve query performance and, in some cases, might actually hurt performance by suppressing indexes. Even if the view only contains a few documents, it can still take a long time to query because the data needs to be reconstructed every time the view is queried.

Materialized views offer a solution here – especially when a view returns small amounts of aggregated information from large source collections. A materialized view is a collection that contains the documents that would be returned by a view definition, but stores the view results in the database so that the view doesn't have to be executed every time you read the data.

In MongoDB, we can use the $merge or $out aggregation operator to create a materialized view. $out completely replaces a target collection with the results of an aggregation. $merge provides a sort of "upsert" into an existing collection, allowing for incremental changes to the target. We'll look a bit more at $merge in Chapter 8.

To create a materialized view, we simply run an aggregation pipeline that might usually be used to define a view, but, as the final step of that aggregation, we use $merge to output the resulting documents into a collection. By running this aggregation pipeline, we can create a new collection that reflects an aggregation of the data in another collection at the time of execution. Unlike a view, however, this collection may be much smaller, allowing for improved performance.

Let's look at an example of this. Here's a complex pipeline that creates a summary of sales by product and city:

```
db.customers.aggregate([
  { $lookup:
     { from:         "orders",
       localField:   "_id",
       foreignField: "customerId",
       as:           "orders"  } },
  { $unwind:  "$orders" },
  { $lookup:
     { from:         "lineitems",
       localField:   "orders._id",
       foreignField: "orderId",
       as:           "lineItems"  }  },
  { $unwind:  "$lineItems" },
  { $group:{ _id:{ "City":"$City" ,
                 "lineItems_prodId":"$lineItems.prodId"  },
           "count":{$sum:1},
           "lineItems_itemCount-sum":{$sum:"$lineItems.itemCount"} } },
  { $project: {
        "CityName": "$_id.City"  ,
          "ProductId": "$_id.lineItems_prodId"   ,
          "OrderCount": "$lineItems_itemCount-sum"   ,
          "_id": 0
         }  } ] );
```

If we add the following $merge operation to that pipeline, then we'll create a collection salesByCityMV that contains the outputs of the aggregation:[2]

```
{$merge:
    {       into:"salesByCityMV"}}
```

Figure 7-8 shows the execution time for querying from the materialized view as compared to querying from a normal view. As you can see, the performance of the materialized view is far superior. This is because the bulk of the work has already been completed by the time the final find query was sent.

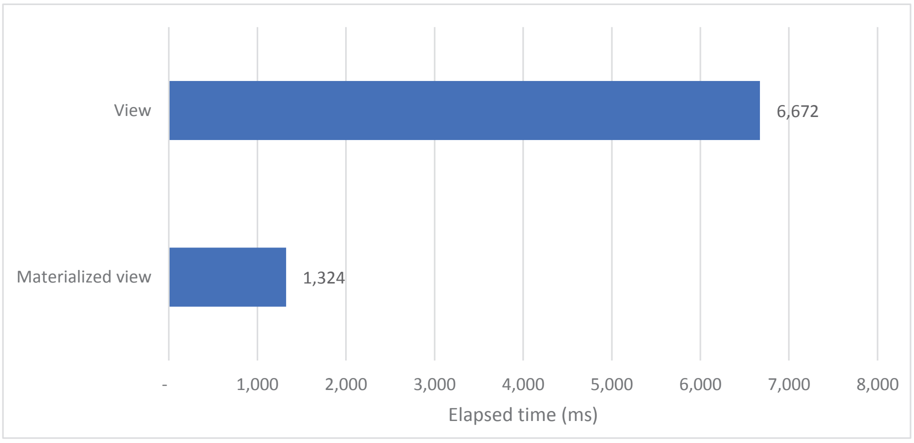

Figure 7-8. *Materialized view vs. direct view*

There is one obvious weakness to this method: the materialized view becomes out of date the second data changes in the original collection. It becomes the responsibility of the application or database administrator to ensure the materialized view is refreshed at intervals that make sense. For example, a materialized view might contain access sales records for the previous day. The aggregation could be run at midnight each night, ensuring that the data was correct for each new day.

[2]Merge has a lot of extra options for dealing with existing data that we'll discuss in Chapter 8.

Tip For complex aggregations where speed of the query is more important that absolute point in time accuracy, a materialized view offers a powerful way to provide quick access to aggregation output.

When creating materialized views, ensure that the refresh of the view is not going to be run more often than the queries on that view. The database still has to use resources to create the view, so there is little reason to refresh a materialized view each hour that may only be queried once a day.

If the source table is updated infrequently, you could arrange for the refresh of the materialized view to occur automatically whenever an update is detected. The MongoDB *Change Stream* facility allows you to listen for changes in a collection. When the change notification is received, you could trigger the rebuild of the materialized view.

We'll look at some more uses of the `$merge` operator in Chapter 8.

Summary

MongoDB created an incredibly powerful method to construct complex queries with the aggregation framework. Over the years, they have expanded this framework to support a wider range of use cases and even to take responsibility for some data transformation that may have previously occurred on the application level. If the past is any indication, the `aggregate` command will grow over time to accommodate more and more complex functionality. With all this in mind, if you wish to create an advanced and performant MongoDB application, you should be leveraging everything that `aggregate` has to offer.

But with the great power of aggregation pipelines comes the great responsibility to ensure that the pipelines are optimized. In this chapter, we have outlined some of the key performance concerns you will want to keep in mind as you create your aggregations.

Filtering and stage order will allow you to minimize data flowing through your pipeline. Indexing relevant fields for `$lookup` and `$graphLookup` will ensure quick retrieval of the relevant documents. You will also need to ensure you use the `allowDiskUse` option when fetching large results to avoid hitting memory limits or alter those memory limits to avoid expensive "disk sorts."

In the next chapter, we will cover the C, U, and D of CRUD – Create, Update, and Delete – and consider the optimization of data manipulation statements such as `insert`, `update`, and `delete`.