Data Aggregation

Objectives

- By the end of this session, you should be able to:
  - Aggregation in mongo
  - Aggregation Pipeline
  - Creating Aggregations
  - Manipulating Data
  - Working with Large Datasets
  - Tuning Pipelines

Aggregation in mongo

- In more straightforward situations, these result sets may be enough to answer your desired business question or satisfy a use case. However, more complex problems require more complex queries to answer. Solving such problems with just the find command would be highly challenging and would likely require multiple queries or some processing on the client side to organize or link the data.
- The basic limitation is where you have data contained in two separate collections. To find the correct data, you would have to run two queries instead of one, joining the data on the client or application level.
- The aggregate command in MongoDB is similar to the find command. You can provide the criteria for your query in the form of JSON documents, and it outputs a cursor containing the search result. Although aggregations can become very large and complex, at their core, they are relatively simple..

Aggregation Pipeline

- The aggregation pipeline does precisely what the name implies. It allows you to define a series of stages that filter, merge, and organize data with much more control than the standard find command. Beyond that, the pipeline structure of aggregation allows developers and database analysts to easily, iteratively, and quickly build queries on ever-changing and growing datasets.
- The key element in aggregation is called the pipeline. A pipeline is a series of instructions, where the input to each instruction is the output of the previous one. Simply put, aggregation is a method for taking a collection and, in a procedural way, filtering, transforming, and joining data from other collections to create new, meaningful datasets.
- Large multi-stage pipelines may look intimidating, but if you understand the structure of the command and the individual operations that can be performed at a given stage, then you can easily break the pipeline down into smaller parts.

Aggregation Pipeline (contd.)

- Aggregate Syntax

  There are two parameters used for aggregation.

  - The pipeline parameter contains all the logic to find, sort, project, limit, transform, and aggregate our data. The pipeline parameter itself is passed in as an array of JSON documents. You can think of this as a series of instructions to be sent to the database, and then the resulting data after the final stage is stored in a cursor to be returned to you.

  - The second parameter is the options parameter. This is optional and allows you to specify the details of the configuration, such as how the aggregation should execute or some flags that are required during debugging and building your pipelines.

Aggregation Pipeline (contd.)

- In the following diagram, the orange blocks represent the aggregation pipeline. Each of these blocks in the pipeline is referred to as an aggregation stage:

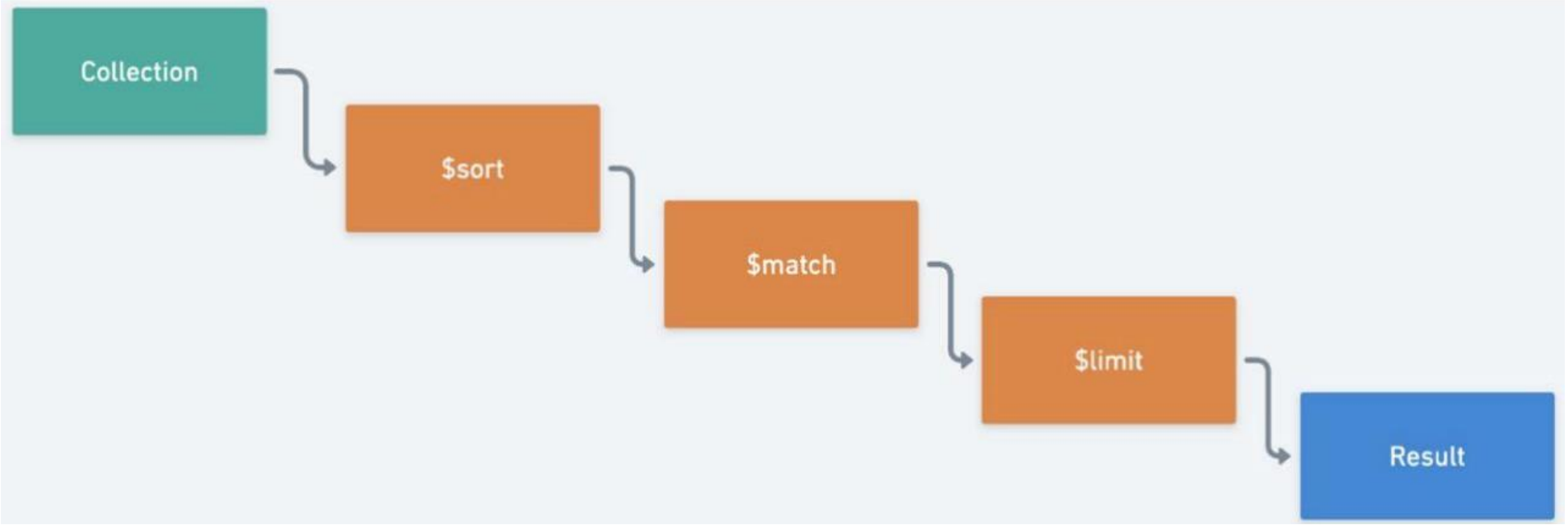

Figure 7.2: Pictorial representation of Aggregation pipeline

Creating Aggregations

- Although the pipeline structure itself is simple, there are more complex stages and operators required to accomplish advanced aggregations, as well as optimize them.
  - **Performing Simple Aggregations**

    Translate your query into sequential stages that you can map to your aggregation stages then simplify your stages wherever possible by merging duplicate stages. In this case, you can merge the two match stages
  - **Aggregation Structure**

    Think of the pipeline as a multi-tiered funnel. It starts broad at the top and becomes thinner as it approaches the bottom. As you pour documents into the top of the funnel, there are many documents, but as you move further down, this number keeps reducing at every stage, until only the documents that you want as output exit at the bottom.

Manipulating Data

- Using some of the more advanced stages and techniques in the pipeline allows us to transform our data, derive new data, and generate insights across a broader scope. This more extensive implementation of the aggregate command is more common than merely rewriting a find command as a pipeline.
- **The Group Stage**
  - As you may expect from the name, the $group stage allows you to group (or aggregate) documents based on a specific condition. Although there are many other stages and methods to accomplish various tasks with the aggregate command, the $group stage serves as the cornerstone of the most powerful queries.
- **Accumulator Expressions**
  - The $group command can accept more than just one argument. It can also accept any number of additional arguments

Manipulating Data (contd.)

- It can also accept any number of additional arguments in the following format:

```
field: { accumulator: expression},
```

Let's break this down into its three components:

• **field** will define the key of our newly computed field for each group.

• **accumulator** must be a supported accumulator operator. These are a group of operators, like other operators you may have worked with already – such as $lte – except, as the name suggests, they will accumulate their value across multiple documents belonging to the same group.

• **expression** in this context will be passed to the accumulator operator as the input of what field in each document it should be accumulating.

Working with Large Datasets

- For large production systems, you may be working on a scale of millions instead of thousands. So far, we have also been focusing strictly on a single collection at a time, but what if the scope of our aggregation grows to include multiple collections?
- How you can improve performance when working on much, much larger datasets, both while developing and for your final production-ready queries.
- Sampling with $sample
  - The first step in learning how to deal with large datasets is understanding $sample. This stage is simple yet useful. The only parameter to $sample is the desired size of your sample. This stage randomly selects documents (up to your specified size) and passes them through to the next stage.

Working with Large Datasets (contd.)

- Joining Collections with $lookup
  - Sampling may assist you when developing queries against extensive collections, but in production queries, you may sometimes need to write queries that are operating across multiple collections. In MongoDB, these collection joins are done using the $lookup aggregation step.
  - The four parameters of $lookup
    - **from**: The collection we are joining to our current aggregation
    - **localField**: The field name that we are going to use to join our documents in the local collection
    - **foreignField**: The field that links to localField in the from collection
    - **as**: This is how our new joined data will be labeled.

Working with Large Datasets (contd.)

- $unwind operator
  - $unwind is a relatively simple stage. It deconstructs an array field from an input document to output a new document for each element in the array
- combination of $lookup and $unwind is a powerful combination for answering complex questions across multiple collections in a single aggregation.

❑ Outputting Your Results with $out and $merge

- From MongoDB version 4.2 onward, we are provided with two aggregation stages:
  - $out
  - $merge
- Both stages allow us to take the output from our pipeline and write it into a collection for later use. Importantly, this whole process takes place on the server, meaning that all the data never needs to be transferred to the client across the network.

Tuning Pipelines

- We timed the execution of our pipeline by outputting the time before and after our aggregation. This is a valid technique, and you may often time your MongoDB queries on the client or application side. However, this only gives us a rough approximation of duration and only tells us the total time the response took to reach the client, not how long the server took to execute the pipeline. MongoDB provides us with a great way of learning exactly how it executed our requested query. This feature is known as Explain and is the usual way to examine and optimize our MongoDB commands.
- **Explain** does not yet support detailed execution plans for aggregations, meaning its use is limited when it comes to the optimization of pipelines.
- Since we can't rely on Explain to analyze our pipelines, it becomes even more integral to carefully construct and plan our pipeline to improve the performance of our aggregations.

Tuning Pipelines (contd.)

- MongoDB does a lot of performance optimization under the hood, but these are still good patterns to follow
  - Filter Early and Filter Often
    - Each stage of the aggregation pipeline will perform some processing on the input. That means the more significant the input, the larger the processing. If you've designed your pipeline correctly, this processing is unavoidable for the documents you are trying to return. The best you can do is to make sure you're processing only the documents you want to return.
  - Use Your Indexes
    - Indexes are another critical element in MongoDB query performance. All you need to remember when creating your aggregations is that when utilizing stages such as **$sort** and **$match**, you want to make sure that you are operating on correctly indexed fields.

Tuning Pipelines (contd.)

- Think about the Desired Output
  - One of the most important ways to improve your pipelines is to plan and evaluate them to ensure that you're getting the desired output that solves your business problem. Ask yourself the following questions if you're having trouble creating a finely tuned pipeline:
    - Am I outputting all the data to solve my problem?
    - Am I outputting only the data required to solve my problem?
    - Am I able to merge or remove any intermediate steps?
  - If you have evaluated your pipeline, tuned it, and still find it to be over-complicated or inefficient, you may need to ask some questions about the data itself.

Tuning Pipelines (contd.)

- Aggregation Options
  - **maxTimeMS:** The amount of time an operation may be processed before MongoDB kills it. Essentially a timeout for your aggregation. The default for this is 0, which means operations do not time out.
  - **allowDiskUse:** Stages in the aggregation pipeline may only use up a maximum amount of memory, making it challenging to handle massive datasets. By setting this option to true, MongoDB can write temporary files to allow the handling of more data.
  - **bypassDocumentValidation:** This option is specifically for pipelines that will be writing out to collections using **$out** or **$merge**. If this option is set to true, document validation will not occur on documents being written to the collection from this pipeline.
  - **comment:** This option is just for debugging and allows a string to be specified that helps identify this aggregation when parsing database logs.

Summary

- In this session, you learned about:
  - We have covered all the essential components that you need to understand, write, comprehend, and improve MongoDB aggregations.
  - You learned that creating multi-stage pipelines that join multiple collections, you can increase the scope of your queries to the entire database instead of a single collection.
  - We also looked at how to write the results into a new collection to enable further exploration or manipulation of the data.
  - We covered the importance of ensuring that your pipelines are written with scalability, readability, and performance in mind.
- In the next chapter, we will walk through the creation of an application in Node.js with MongoDB as a backend.