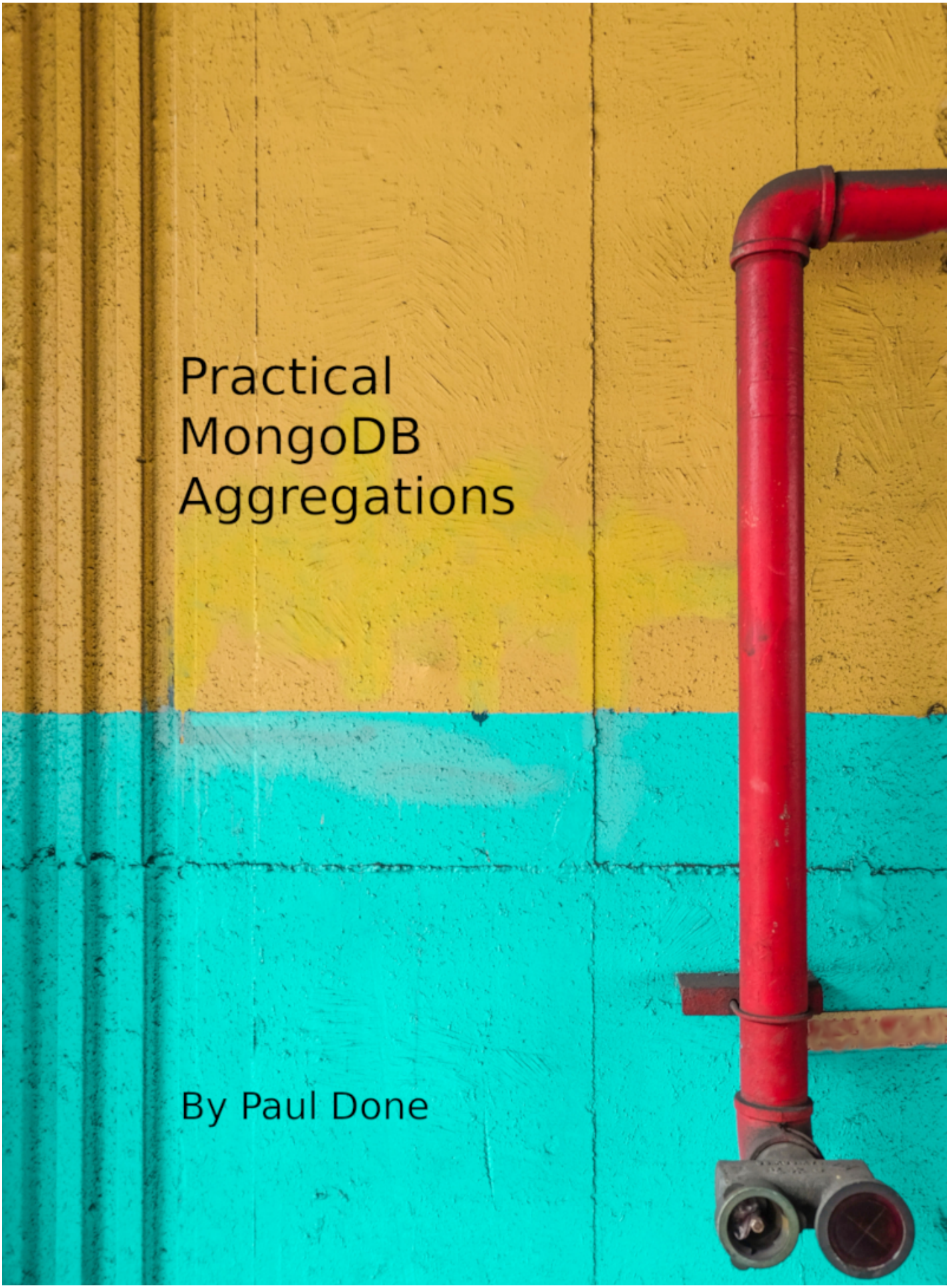

Practical
MongoDB
Aggregations
By Paul Done

Practical MongoDB Aggregations

Author: Paul Done (@TheDonester)

Version: 3.03

Last Updated: February 2022

MongoDB Versions Supported: 4.2 → 5.2

Book published at: www.practical-mongodb-aggregations.com

Content created & assembled at: github.com/pkdone/practical-mongodb-aggregations-book

---

Acknowledgements - many thanks to the following people for their valuable feedback:

- Jake McInteer
- John Page
- Asya Kamsky
- Mat Keep
- Brian Leonard

Front cover image adapted from a Photo by Henry & Co. from Pexels under the Pexels License *(free to use & modify)*

Foreword

By Asya Kamsky (@asya999)

I've been involved with databases since the early 1990's when I "accidentally" got a job with a small database company. For the next two decades, databases were synonymous with SQL in my mind, until someone asked me what I thought about these new "No SQL" databases, and MongoDB in particular. I tried MongoDB for a small project I was doing on the side, and the rest, as they say, is history.

When I joined the company that created MongoDB in early 2012, the query language was simple and straightforward but didn't include options for easy data aggregation, because the general advice was "store the data the way you expect to access the data", which was a fantastic approach for fast point queries. As time went on, though, it became clear that sometimes you want to answer questions that you didn't know you'd have when you were first designing the application, and the options for that within the database itself were limited. Map-Reduce was complicated to understand and get right, and required writing and running JavaScript, which was inefficient. This led to a new way to aggregate data natively in the server, which was called "The Aggregation Framework". Since the stages of data processes were organized as a pipeline (familiarly evoking processing files on the Unix command line, for those of us who did such things a lot) we also referred to it as "The Aggregation Pipeline". Very quickly "Agg" became my favorite feature for its flexibility, power and ease of debugging.

We've come a long away in the last nine years, starting with just seven stages and three dozen expressions operating on a single collection, to where we are now: over thirty stages, including special stages providing input to the pipeline, allowing powerful output from the pipeline, including data from other collections in a pipeline, and over one hundred and fifty expressions, available not just in the aggregation command but also in queries and updates.

The nature of data is such that we will never know up-front all the questions we will have about it in the future, so being able to construct complex queries (aka aggregations) about it is critical to success. While complex data processing can be performed in any programming language you are comfortable with, being able to analyze your data without having to move it from where it's currently stored provides a tremendous advantage over exporting and loading the data elsewhere just to be able to use it for your analytics.

For years, I've given talks about the power of the Aggregation Pipeline, answered questions from users about how to do complex analysis with it, and frequently fielded requests for a comprehensive "Aggregation Cookbook". Of course it would be great to have a repository of "recipes" with which to solve common data tasks that involve more than a single stage or expression combination, but it's hard to find the time to sit down and write something like that. This is why I was so stoked to see that my colleague, Paul Done, had just written this book and laid the foundation for that cookbook.

I hope you find this collection of suggestions, general principles, and specific pipeline examples useful in your own application development and I look forward to seeing it grow over time to become the cookbook that will help everyone realize the full power of their data.

Who This Book Is For

This book is for developers, architects, data analysts, data engineers and data scientists who have some familiarity with MongoDB and have already acquired a small amount of rudimentary experience using the MongoDB Aggregation Framework. If you do not yet have this entry level knowledge, don't worry because there are plenty of getting started guides out there. If you've never used the Aggregation Framework before, you should start with one or more of the following resources before using this book:

- The MongoDB Manual, and specifically its Aggregation section
- The MongoDB University free online courses, and specifically The MongoDB Aggregation Framework (M121) introduction course
- The MongoDB: The Definitive Guide book by Bradshaw, Brazil & Chodorow, and specifically its section *7. Introduction to the Aggregation Framework*

This book is not for complete novices, explaining how you should get started on your first MongoDB aggregation pipeline. Neither is this book a comprehensive programming language guide detailing every nuance of the Aggregation Framework and its syntax. This book intends to assist you with two key aspects:

1. Providing a set of opinionated yet easy to digest principles and approaches for increasing your effectiveness in using the Aggregation Framework
2. Providing a set of examples for building aggregation pipelines to solve various types of data manipulation and analysis challenges

Introduction

This introduction section of the book helps you understand what MongoDB Aggregations are, the philosophy of Aggregations, what people use them for, and the history of the framework.

Introducing MongoDB Aggregations

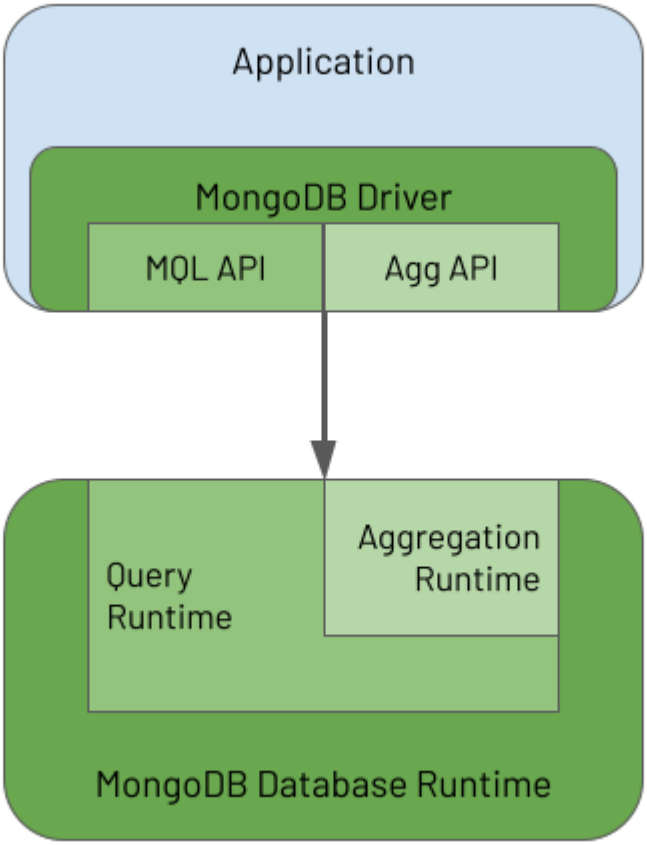
What Is MongoDB's Aggregation Framework?

MongoDB's Aggregation Framework enables users to send an analytics or data processing workload, written using an aggregation language, to the database to execute the workload against the data it holds. You can think of the Aggregation Framework as having two parts:

1. The Aggregations API provided by the MongoDB Driver embedded in each application to enable the application to define an aggregation task called a pipeline and send it to the database for the database to process

2. The Aggregation Runtime running in the database to receive the pipeline request from the application and execute the pipeline against the persisted data

The following diagram illustrates these two elements and their inter-relationship:



The driver provides APIs to enable an application to use both the MongoDB Query Language (MQL) and the Aggregation Framework. In the database, the Aggregation Runtime re-uses the Query Runtime to efficiently execute the query part of an aggregation workload that typically appears at the start of an aggregation pipeline.

What Is MongoDB's Aggregations Language?

MongoDB's aggregation pipeline language is somewhat of a paradox. It can appear daunting, yet it is straightforward. It can seem verbose, yet it is lean and to the point. It is Turing

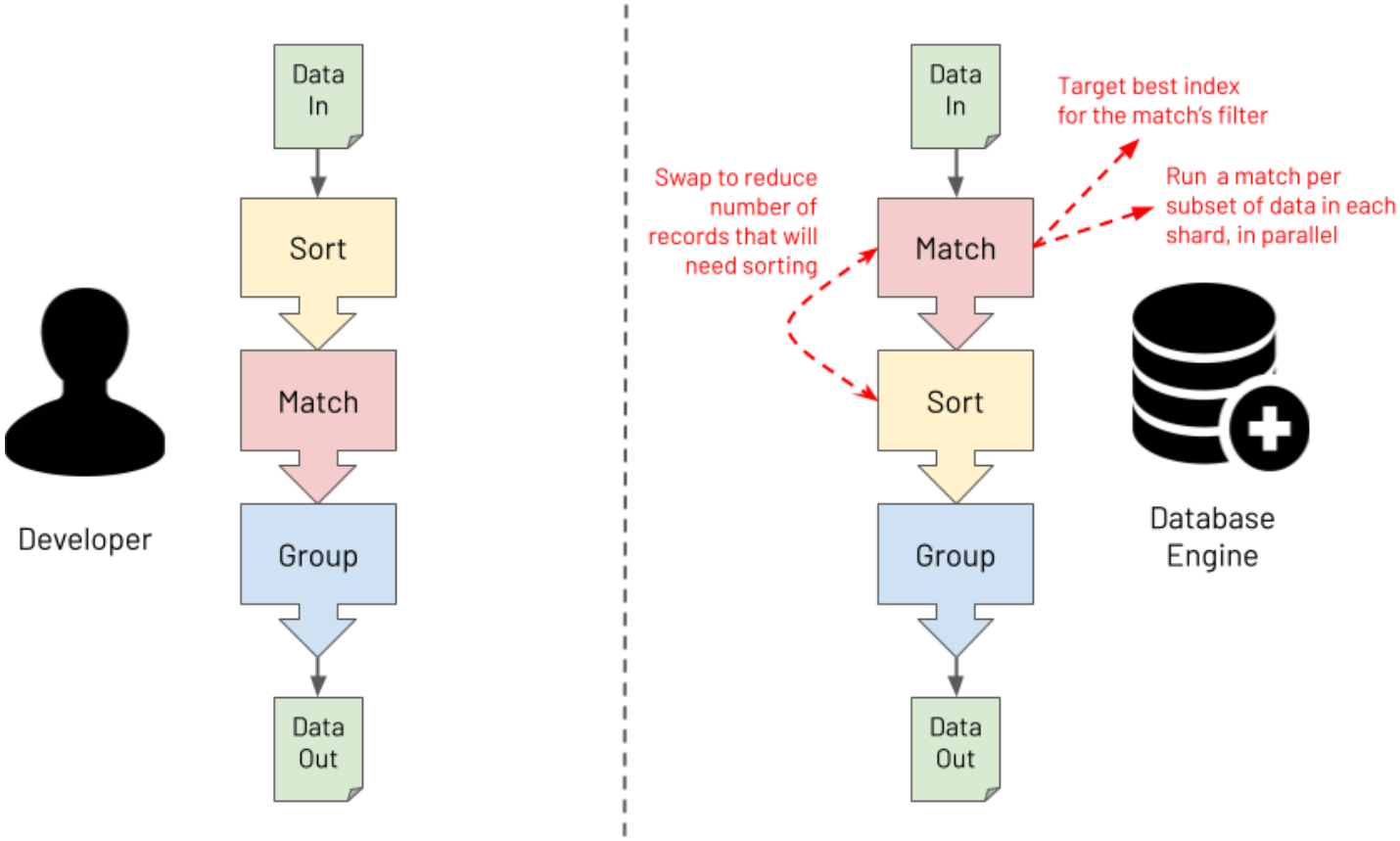
complete and able to solve any business problem *****. Conversely, it is a strongly opinionated Domain Specific Language (DSL), where, if you attempt to veer away from its core purpose of mass data manipulation, it will try its best to resist you.

> ***** *As John Page once showed, you can even code a Bitcoin miner using MongoDB aggregations, not that he (or hopefully anyone for that matter) would ever recommend you do this for real, for both the sake of your bank balance and the environment!*

Invariably, for beginners, the Aggregation Framework seems difficult to understand and comes with an initially steep learning curve that you must overcome to become productive. In some programming languages, you only need to master a small set of the language's aspects to be largely effective. With MongoDB aggregations, the initial effort you must invest is slightly greater. However, once mastered, users find it provides an elegant, natural and efficient solution to breaking down a complex set of data manipulations into a series of simple easy to understand steps. This is the point when users achieve the Zen of MongoDB Aggregations, and it is a lovely place to be.

MongoDB's aggregation pipeline language is focused on data-oriented problem-solving rather than business process problem-solving. Depending on how you squint, it can be regarded as a functional programming language rather than a procedural programming language. Why? Well, an aggregation pipeline is an ordered series of statements, called stages, where the entire output of one stage forms the entire input of the next stage, and so on, with no side effects. This functional nature is probably why many users regard the Aggregation Framework as having a steeper learning curve than many languages. Not because it is inherently more difficult to understand but because most developers come from a procedural programming background and not a functional one. Most developers also have to learn how to think like a functional programmer to learn the Aggregation Framework.

The Aggregation Framework's functional characteristics ultimately make it especially powerful for processing massive data sets. Users focus more on defining "the what" in terms of the required outcome. Users focus less on "the how" of specifying the exact logic to apply to achieve each transformation. You provide one specific and clear advertised purpose for each stage in the pipeline. At runtime, the database engine can then understand the exact intent of each stage. For example, the database engine can obtain clear answers to the questions it asks, such as, "is this stage for performing a filter or is this stage for grouping on some fields?". With this knowledge, the database engine has the opportunity to optimise the pipeline at runtime. The diagram below shows an example of the database performing a pipeline optimisation. It may decide to re-order stages to optimally leverage an index whilst ensuring that the output isn't changed. Or, it may choose to execute some steps in parallel against subsets of the data in different shards, reducing response time whilst again ensuring the output is never changed.



Last and by far least in terms of importance is a discussion about syntax. So far, MongoDB aggregations have been described here as a programming language, which it is (a Domain Specific Language). However, with what syntax is a MongoDB aggregation pipeline constructed? The answer is "it depends", and the answer is mostly irrelevant. This book will highlight pipeline examples using MongoDB's Shell and the JavaScript interpreter it runs in. The book will express aggregation pipelines using a JSON based syntax. However, if you are using one of the many programming language drivers that MongoDB provides, you will be using that language to construct an aggregation pipeline, not JSON. An aggregation is specified as an array of objects, regardless of how the programming language may facilitate this. This programmatic rather than textual format has a couple of advantages compared to querying with a string. It has a low vulnerability to injection attacks, and it is highly composable.

What's In A Name?

You might have realised by now that there doesn't seem to be one single name for the subject of this book. You will often hear:

- Aggregation
- Aggregations
- Aggregation Framework
- Aggregation Pipeline

- Aggregation Pipelines
- Aggregation Language
- Agg
- *...and so on*

The reality is that any of these names are acceptable, and it doesn't matter which you use. This book uses most of these terms at some point. Just take it as a positive sign that this MongoDB capability (and its title) was not born in a marketing boardroom. It was built by database engineers, for data engineers, where the branding was an afterthought at best!

What Do People Use The Aggregation Framework For?

The Aggregation Framework is versatile and used for many different data processing and manipulation tasks. Some typical example uses are for:

- Real-time analytics
- Report generation with roll-ups, sums & averages
- Real-time dashboards
- Redacting data to present via views
- Joining data together from different collections on the "server-side"
- Data science, including data discovery and data wrangling
- Mass data analysis at scale (a la "big data")
- Real-time queries where deeper "server-side" data post-processing is required than provided by the MongoDB Query Language (MQL)
- Copying and transforming subsets of data from one collection to another
- Navigating relationships between records, looking for patterns
- Data masking to obfuscate sensitive data
- Performing the Transform (T) part of an Extract-Load-Transform (ELT) workload
- Data quality reporting and cleansing
- Updating a materialised view with the results of the most recent source data changes
- Representing data ready to be exposed via SQL/ODBC/JDBC (using MongoDB's BI Connector)
- Supporting machine learning frameworks for efficient data analysis (e.g. via MongoDB's Spark Connector)
- *...and many more*

History Of MongoDB Aggregations

The Emergence Of Aggregations

MongoDB's developers released the first major version of the database (version 1.0) in February 2009. Back then, both users and the predominant company behind the database, MongoDB Inc. (called *10gen* at the time) were still establishing the sort of use cases that the database would excel at and where the critical gaps were. Within half a year of this first major release, MongoDB's engineering team had identified a need to enable materialised views to be generated on-demand. Users needed this capability to maintain counts, sums, and averages for their real-time client applications to query. In December 2009, in time for the following major release (1.2), the database engineers introduced a quick tactical solution to address this gap. This solution involved embedding a JavaScript engine in the database and allowing client applications to submit and execute "server-side" logic using a simple Map-Reduce API.

A Map-Reduce workload essentially does two things. Firstly it scans the entire data set, looking for the matching subset of records required for the given scenario. This phase may also transform or exclude the fields of each record. This is the "map" action. Secondly, it condenses the subset of matched data into grouped, totalled, and averaged result summaries. This is the "reduce" action. Functionally, MongoDB's *Map-Reduce* capability provides a solution to users' typical data processing requirements, but it comes with the following drawbacks:

1. The database has to bolt in an inherently slow JavaScript engine to execute users' Map-Reduce code.
2. Users have to provide two sets of JavaScript logic, a *map* (or matching) function and a *reduce* (or grouping) function. Neither is very intuitive to develop, lacking a solid data-oriented bias.
3. At runtime, the lack of ability to explicitly associate a specific intent to an arbitrary piece of logic means that the database engine has no opportunity to identify and apply optimisations. It is hard for it to target indexes or re-order some logic for more efficient processing. The database has to be conservative, executing the workload with minimal concurrency and employing locks at various times to prevent race conditions and inconsistent results.
4. Poor scalability because the monolithic and opaque nature of Map-Reduce logic means the database engine can't break parts of it up and execute these parts in parallel across multiple shards.

Over the following two years, as user behaviour with Map-Reduce became more understood, MongoDB engineers started to envision a better solution. Also, users were increasingly trying to use Map-Reduce to perform mass data processing given MongoDB's ability to hold large data sets. They were hitting the same Map-Reduce limitations. Users desired a more targeted capability leveraging a data-oriented Domain Specific Language (DSL). The engineers saw how to deliver a framework enabling a developer to define a series of data manipulation steps with valuable composability characteristics. Each step would have a clear advertised intent, allowing

the database engine to apply optimisations at runtime. The engineers could also design a framework that would execute "natively" in the database and not require a JavaScript engine. In August 2012, this solution, called the Aggregation Framework, was introduced in the 2.2 version of MongoDB. MongoDB's Aggregation Framework provided a far more powerful, efficient, scalable and easy to use replacement to Map-Reduce.

Within its first year, the Aggregation Framework rapidly became the go-to tool for processing large volumes of data in MongoDB. Now, nearly a decade on, it is like the Aggregation Framework has always been part of MongoDB. It feels like part of the database's core DNA. MongoDB still supports Map-Reduce, but developers rarely use it nowadays. MongoDB's aggregations are always the correct answer for processing data in the database!

> *It is not widely known, but MongoDB's engineering team re-implemented the Map-Reduce "back-end" in MongoDB 4.4 to execute within the aggregations runtime. They had to develop some additional aggregation stages and operators to fill some gaps. For the most part, these are internal-only stages or operators that the Aggregation Framework does not surface for developers to use in regular aggregations. The two exceptions are the new $function and $accumulator 4.4 operators, which the refactoring work influenced and which now serve as two useful operators for use in any aggregation pipeline. In MongoDB 4.4, each Map-Reduce "aggregation" still uses JavaScript for certain phases, and so it will not achieve the performance of a native aggregation for an equivalent workload. Nor does this change magically address the other drawbacks of Map-Reduce workloads concerning composability, concurrency, scalability and opportunities for runtime optimisation. The primary purpose of the change was for the database engineers to eliminate redundancy and promote resiliency in the database's codebase. With MongoDB 5.0, Map-Reduce is now deprecated, and it is likely to be removed in the subsequent major version of MongoDB.*

Key Releases & Capabilities

Below is a summary of the evolution of the Aggregation Framework in terms of significant capabilities added in each major release:

- **MongoDB 2.2 (August 2012)**: Initial Release
- **MongoDB 2.4 (March 2013)**: Efficiency improvements (especially for sorts), a concat operator
- **MongoDB 2.6 (April 2014)**: Unlimited size result sets, explain plans, spill to disk for large sorts, an option to output to a new collection, a redact stage
- **MongoDB 3.0 (March 2015)**: Date-to-string operators

- **MongoDB 3.2 (December 2015)**: Sharded cluster optimisations, lookup (join) & sample stages, many new arithmetic & array operators
- **MongoDB 3.4 (November 2016)**: Graph-lookup, bucketing & facets stages, many new array & string operators
- **MongoDB 3.6 (November 2017)**: Array to/from object operators, more extensive date to/from string operators, a REMOVE variable
- **MongoDB 4.0 (July 2018)**: Number to/from string operators, string trimming operators
- **MongoDB 4.2 (August 2019)**: A merge stage to insert/update/replace records in existing non-sharded & sharded collections, set & unset stages to address the verbosity/rigidity of project stages, trigonometry operators, regular expression operators
- **MongoDB 4.4 (July 2020)**: A union stage, custom JavaScript operator expressions (function & accumulator), first & last array element operators, string replacement operators, a random number operator
- **MongoDB 5.0 (July 2021)**: A setWindowFields stage, time-series/window operators, date manipulation operators
- **MongoDB 5.1 (November 2021)**: Support for lookup & graph-lookup stages joining to sharded collections, documents and densify stages
- **MongoDB 5.2 (January 2021)**: An array sorting operator, operators to get a subset of ordered arrays and a subset of ordered grouped documents

Getting Started

For developing aggregation pipelines effectively, and also to try the examples in the second half of this book, you need the following two elements:

1. A **MongoDB database**, **version 4.2 or greater**, running somewhere which is network accessible from your workstation
2. A **MongoDB client tool** running on your workstation with which to submit aggregation execution requests and to view the results

Note that each example aggregation pipeline shown in the second major part of this book is marked with the minimum version of MongoDB that you must use to execute the provided pipeline. Some example pipelines use aggregation features that MongoDB introduced in versions greater than 4.2. Where this is the case, the exact version number required is called out. For MongoDB versions 4.0 and earlier, some examples may work unchanged. Some examples may work with minor alterations, and some may not work at all due to fundamental dependencies on features added in MongoDB versions after 4.0.

Database

The database deployment for you to connect to can be a single server, a replica set or a sharded cluster. You can run this deployment locally on your workstation or remotely on-prem or in the cloud. It doesn't matter which. You need to know the MongoDB URL for connecting to the database and, if authentication is enabled, the credentials required for full read and write access.

If you don't already have access to a MongoDB database, the two most accessible options for running a database for free are:

1. Provision a Free Tier MongoDB Cluster in MongoDB Atlas, which is MongoDB Inc.'s cloud-based Database-as-a-Service (once deployed, in the Atlas Console, there is a button you can click to copy the URL of the cluster)
2. Install and run a MongoDB single server locally on your workstation

Client Tool

There are many options for the client tool, four of which are:

1. ***Modern* Shell**. Install the modern version of MongoDB's command-line tool, the MongoDB Shell: `mongosh`
2. ***Legacy* Shell**. Install the legacy version of MongoDB's command-line tool, the Mongo Shell: `mongo` (you will often find this binary bundled with a MongoDB database installation or you can download it from the Atlas console)
3. **Compass**. Install the *official* MongoDB Inc. provided graphical user interface (GUI) tool, MongoDB Compass
4. **Studio 3T**. Install the *3rd party* 3T Software Labs provided graphical user interface (GUI) tool, Studio 3T

The book's examples present code in such a way to make it easy to copy and paste into MongoDB's Shell (`mongosh` or `mongo`) to execute. All subsequent instructions in this book assume you are using the Shell. However, you will find it straightforward to use one of the mentioned GUI tools instead to consume the code examples. Of the two Shell versions, is it is easier to use and view results with the *modern* Shell.

MongoDB Shell With Atlas Database

Here is an example of how you can start the *modern* Shell to connect to an Atlas Free Tier MongoDB Cluster (change the text `mongosh` to `mongo` if you are using the *legacy* Shell):

```
mongosh "mongodb+srv://mycluster.a123b.mongodb.net/test" --username myuser
```

Note before running the command above, ensure

1. You have added your workstation's IP address to the Atlas Access List
2. You have created a database user for the deployed Atlas cluster, with rights to create, read and write to any database
3. You have changed the dummy URL and username text, shown in the above example command, to match your real cluster's details (these details are accessible via the cluster's `Connect` button in the Atlas Console)

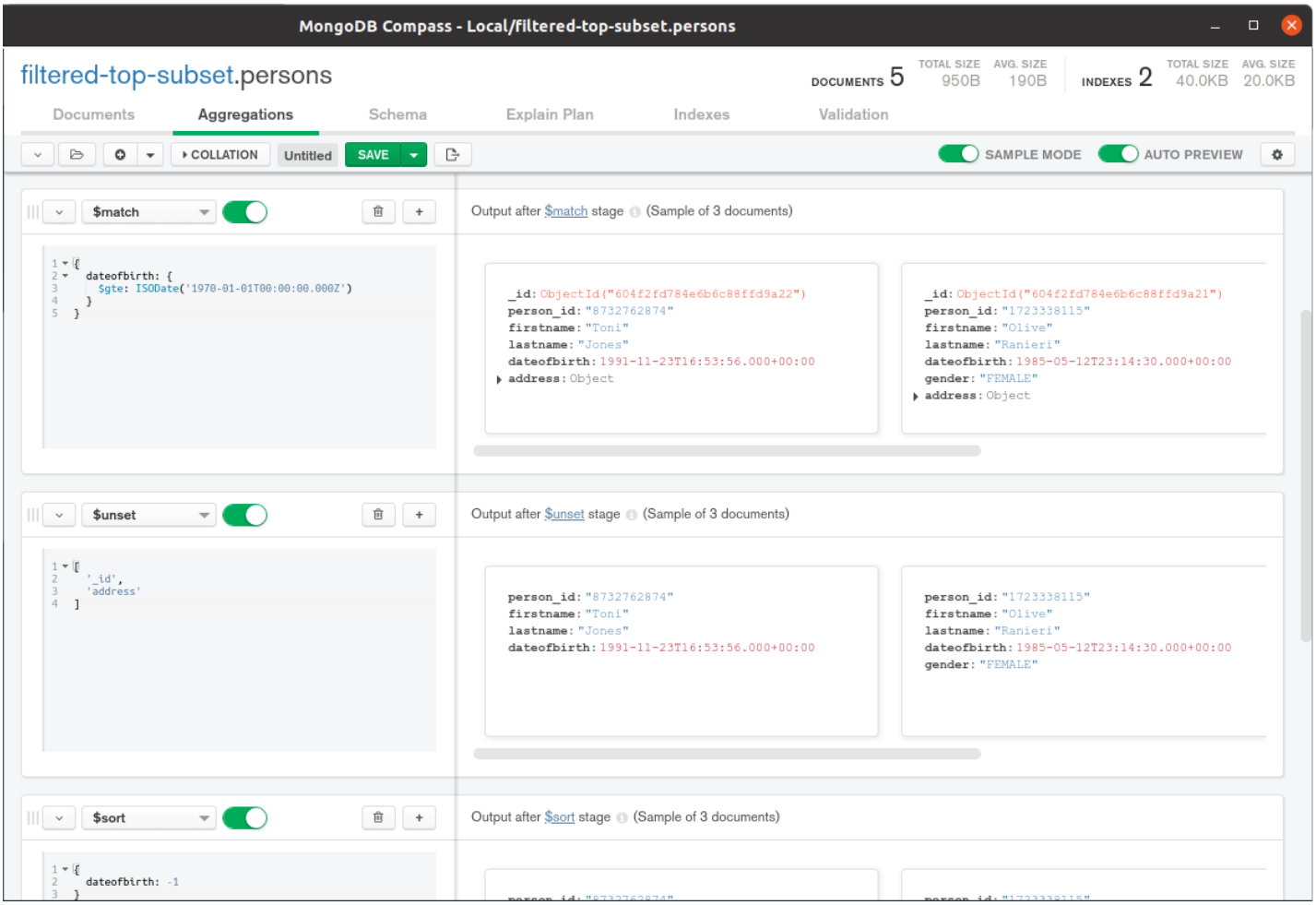
MongoDB Shell With Local Database

Here is an example of how you can start the *modern* Shell to connect to a MongoDB single server database if you've installed one locally on your workstation (change the text `mongosh` to `mongo` if you are using the *legacy* Shell):

```
mongosh "mongodb://localhost:27017"
```

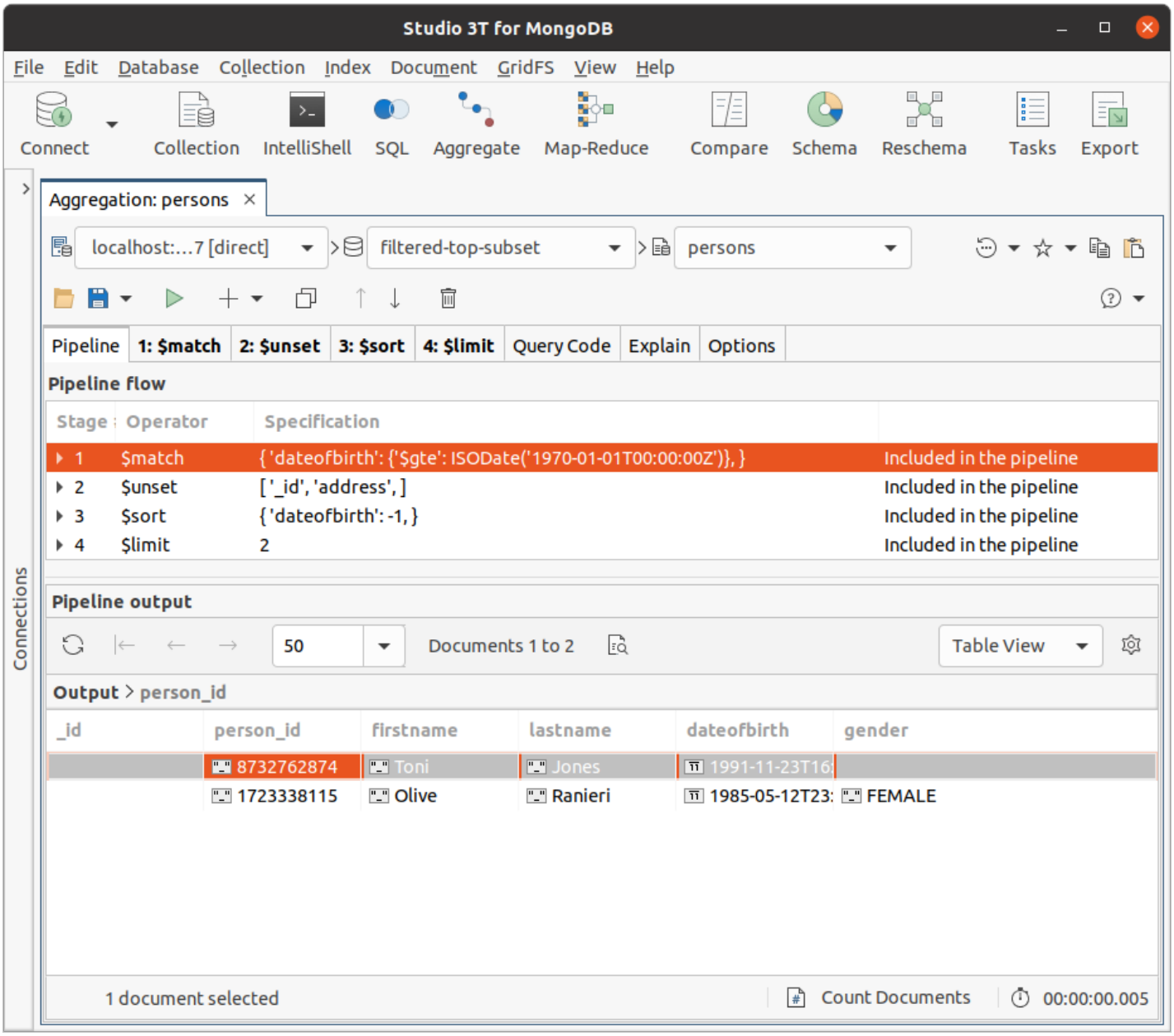
MongoDB Compass GUI

MongoDB Compass provides an *Aggregation Pipeline Builder* tool to assist users in prototyping and debugging aggregation pipelines and exporting them to different programming languages. Below is a screenshot of the aggregation tool in Compass:



Studio 3T GUI

Studio 3T provides an *Aggregation Editor* tool to help users prototype and debug aggregation pipelines and translate them to different programming languages. Below is a screenshot of the aggregation tool in Studio 3T:



Getting Help

No one can hold the names and syntax of all the different aggregation stages and operators in their heads. I'd bet even *MongoDB Aggregations Royalty* (Asya Kamsky) couldn't, although I'm sure she would give it a good go!

The good news is there is no need for you to try to remember all the stages & operators. The MongoDB online documentation provides you with a set of excellent *references* here:

- MongoDB Aggregation Pipeline Stages reference
- MongoDB Aggregation Pipeline Operators reference

To help you get started with the purpose of each stage in the MongoDB Framework, consult the "cheatsheets" in the appendix of this book:

- MongoDB Aggregation Stages Cheatsheet
- MongoDB Aggregation Stages Cheatsheet Source Code

If you are getting stuck with an aggregation pipeline and want some help, an active online community will almost always have the answer. So pose your questions at either:

- The MongoDB Developer Hub - Community Forums
- Stack Overflow - MongoDB Questions

You may be asking for just general advice. However, suppose you want to ask for help on a specific aggregation pipeline under development. In that case, you should provide a sample input document, a copy of your current pipeline code (in its JSON syntax format and not a programming language specific format) and an example of the output that you are trying to achieve. If you provide this extra information, you will have a far greater chance of receiving a timely and optimal response.

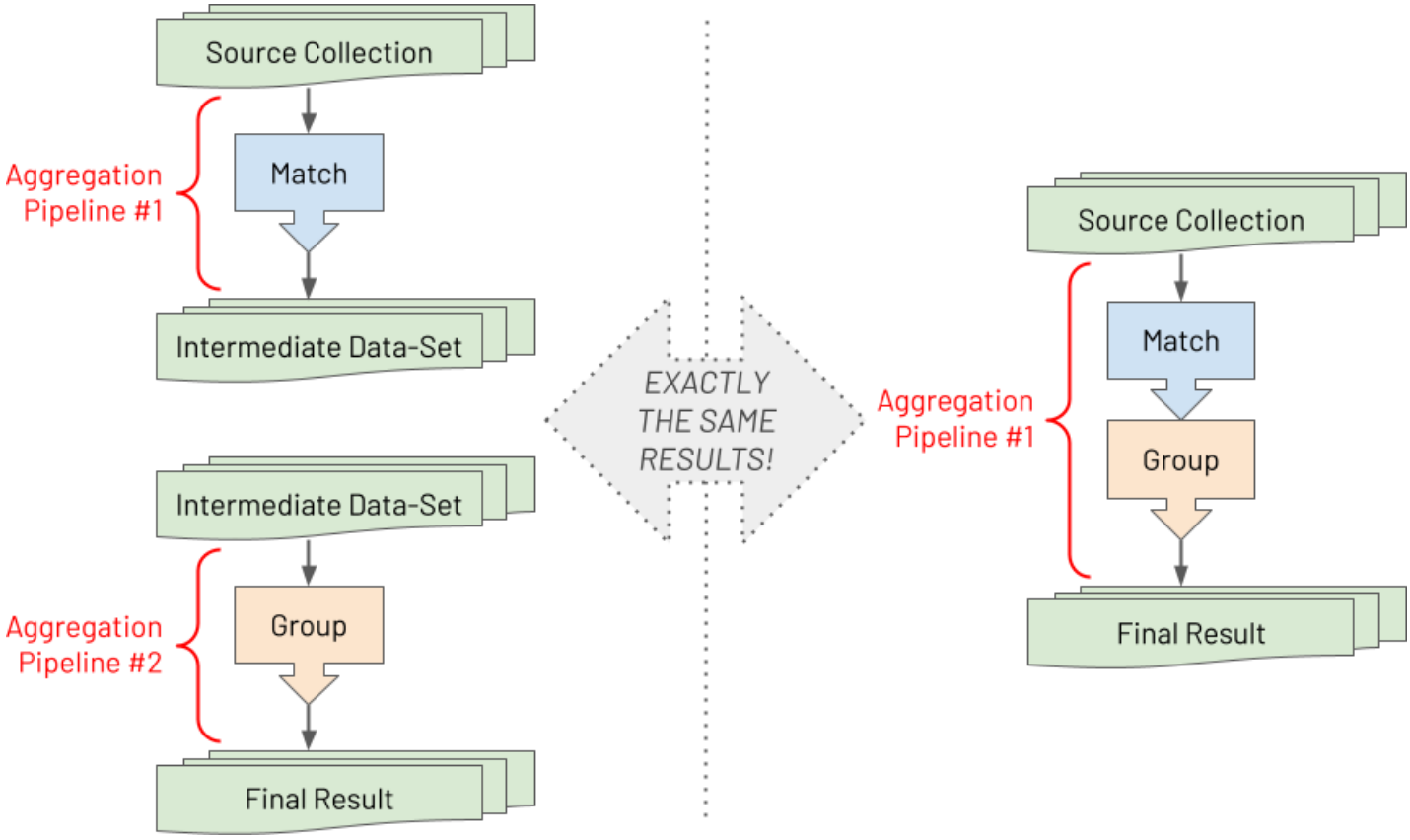
Guiding Tips & Principles

The following set of chapters provide opinionated yet easy-to-digest principles and approaches for increasing effectiveness, productivity, and performance when developing aggregation pipelines.

Embrace Composability For Increased Productivity

An aggregation pipeline is an ordered series of instructions, called stages. The entire output of one stage forms the whole input of the next stage, and so on, with no side effects. Pipelines exhibit high composability where stages are stateless self-contained components selected and assembled in various combinations (pipelines) to satisfy specific requirements. This composability promotes iterative prototyping, with straightforward testing after each increment.

With MongoDB's aggregations, you can take a complex problem, requiring a complex aggregation pipeline, and break it down into straightforward individual stages, where each step can be developed and tested in isolation first. To better comprehend this composability, it may be helpful to internalise the following visual model.



Suppose you have two pipelines with one stage in each. After saving the intermediate results by running the first pipeline, you run the second pipeline against the saved intermediate data set. The final result is the same as running a single pipeline containing both stages serially. There is no difference between the two. As a developer, you can reduce the cognitive load by understanding how a problem can be broken down in this way when building aggregation pipelines. Aggregation pipelines enable you to decompose a big challenge into lots of minor challenges. By embracing this approach of first developing each stage separately, you will find even the most complex challenges become surmountable.

Specific Tips To Promote Composability

In reality, once most developers become adept at using the Aggregation Framework, they tend not to rely on temporary intermediate data sets whilst prototyping each stage. However, it is still a reasonable development approach if you prefer it. Instead, seasoned aggregation pipeline developers typically comment out one or more stages of an aggregation pipeline when using MongoDB's Shell (or they use the "disable stage" capability provided by the GUI tools for MongoDB).

To encourage composability and hence productivity, some of the principles to strive for are:

- Easy disabling of subsets of stages, whilst prototyping or debugging

- Easy addition of new fields to a stage or new stages to a pipeline by performing a copy, a paste and then a modification without hitting cryptic error messages resulting from issues like missing a comma before the added element
- Easy appreciation of each distinct stage's purpose, at a glance

With these principles in mind, the following is an opinionated list of guidelines for how you should textually craft your pipelines in JavaScript to improve your pipeline development pace:

1. Don't start or end a stage on the same line as another stage
2. For every field in a stage, and stage in a pipeline, include a trailing comma even if it is currently the last item
3. Include an empty newline between every stage
4. For complex stages include a `//` comment with an explanation on a newline before the stage
5. To "disable" some stages of a pipeline whilst prototyping or debugging another stage, use the multi-line comment `/*` prefix and `*/` suffix

Below is an example of a poor pipeline layout if you have followed none of the guiding principles:

```
// BAD

var pipeline = [
  {"$unset": [
    "_id",
    "address"
  ]}, {"$match": {
    "dateofbirth": {"$gte": ISODate("1970-01-01T00:00:00Z")}
  }}//, {"$sort": {
  //  "dateofbirth": -1
  //}}, {"$limit": 2}
];
```

Whereas the following is an example of a far better pipeline layout, where you meet all of the guiding principles:

```
// GOOD

var pipeline = [
  {"$unset": [
    "_id",
    "address",
  ]},

  // Only match people born on or after 1st January 1970
  {"$match": {
    "dateofbirth": {"$gte": ISODate("1970-01-01T00:00:00Z")},
  }},

  /*
  {"$sort": {
    "dateofbirth": -1,
  }},

  {"$limit": 2},
  */
];
```

Notice trailing commas are included in the code snippet, at both the end of stage level and end of field level.

It is worth mentioning that some (but not all) developers take an alternative but an equally valid approach to constructing a pipeline. They decompose each stage in the pipeline into different JavaScript variables, where each stage's variable is defined separately, as shown in the example below:

```
// GOOD

var unsetStage = {
  "$unset": [
    "_id",
    "address",
  ]};

var matchStage = {
  "$match": {
    "dateofbirth": {"$gte": ISODate("1970-01-01T00:00:00Z")},
  }};

var sortStage = {
   "$sort": {
    "dateofbirth": -1,
  }};


var limitStage = {"$limit": 2};

var pipeline = [
  unsetStage,
  matchStage,
  sortStage,
  limitStage,
];
```

Furthermore, some developers may take additional steps if they do not intend to transfer the prototyped pipeline to a different programming language:

- They may choose to decompose elements inside a stage into additional JavaScript variables to avoid code "typos". For instance, to prevent one part of a pipeline incorrectly referencing a field computed earlier in the pipeline due to a misspelling.
- They may choose to factor out the generation of some boilerplate code, representing a complex set of expressions, from part of a pipeline into a separate JavaScript function. This new function is essentially a macro. They can then re-use this function from multiple places within the main pipeline's code. Whenever the pipeline invokes this function, the pipeline's body directly embeds the returned boilerplate code. The Array Sorting & Percentiles chapter, later in this book, provides an example of this approach.

In summary, this book is not advocating a multi-variable approach over a single-variable approach when you define a pipeline. The book is just highlighting two highly composable options. Ultimately it is a personal choice concerning which you find most comfortable and productive.

Better Alternatives To A Project Stage

The quintessential tool used in MongoDB's Query Language (MQL) to define or restrict fields to return is a *projection*. In the MongoDB Aggregation Framework, the analogous facility for specifying fields to include or exclude is the $project stage. For many earlier versions of MongoDB, this was the only tool to define which fields to keep or omit. However, `$project` comes with a few usability challenges:

1. ***$project* is confusing and non-intuitive**. You can only choose to include fields or exclude fields in a single stage, but not both. There is one exception, though, where you can exclude the _id field yet still define other fields to include (this only applies to the _id field). It's as if `$project` has an identity crisis.

2. ***$project* is verbose and inflexible**. If you want to define one new field or revise one field, you will have to name all other fields in the projection to include. If each input record has 100 fields and the pipeline needs to employ a `$project` stage for the first time, things become tiresome. To include a new 101st field, you now also have to name all the original 100 fields in this new `$project` stage too. You will find this irritating if you have an evolving data model, where additional new fields appear in some records over time. Because you use a `$project` for inclusion, then each time a new field appears in the data set, you must go back to the old aggregation pipeline to modify it to name the new field explicitly for inclusion in the results. This is the antithesis of flexibility and agility.

In MongoDB version 4.2, the $set and $unset stages were introduced, which, in most cases, are preferable to using `$project` for declaring field inclusion and exclusion. They make the code's intent much clearer, lead to less verbose pipelines, and, critically, they reduce the need to refactor a pipeline whenever the data model evolves. How this works and guidance on when to use `$set` & `$unset` stages is described in the section *When To Use Set & Unset*, further below.

Despite the challenges, though, there are some specific situations where using `$project` is advantageous over `$set` / `$unset`. These situations are described in the section *When To Use Project* further below.

MongoDB version 3.4 addressed some of the disadvantages of `$project` by introducing a new $addFields stage, which has the same behaviour as `$set`. `$set` came later than `$addFields` and `$set` is actually just an alias for `$addFields`. However, back then, the Aggregation Framework provided no direct equivalent to `$unset`. Both `$set` and `$unset` stages are available in modern versions of MongoDB, and their counter purposes are obvious to deduce by their names (`$set` Vs `$unset`). The name `$addFields` doesn't fully reflect that you can modify existing fields rather than just adding new fields. This book prefers `$set` over `$addFields` to help promote consistency and avoid any confusion of intent. However, if you are wedded to `$addFields`, use that instead, as there is no behavioural difference.

When To Use *$set* & *$unset*

You should use `$set` & `$unset` stages when you need to retain most of the fields in the input records, and you want to add, modify or remove a minority subset of fields. This is the case for most uses of aggregation pipelines.

For example, imagine there is a collection of credit card payment documents similar to the following:

```
// INPUT  (a record from the source collection to be operated on by an
aggregation)
{
  _id: ObjectId("6044faa70b2c21f8705d8954"),
  card_name: "Mrs. Jane A. Doe",
  card_num: "1234567890123456",
  card_expiry: "2023-08-31T23:59:59.736Z",
  card_sec_code: "123",
  card_provider_name: "Credit MasterCard Gold",
  transaction_id: "eb1bd77836e8713656d9bf2debba8900",
  transaction_date: ISODate("2021-01-13T09:32:07.000Z"),
  transaction_curncy_code: "GBP",
  transaction_amount: NumberDecimal("501.98"),
  reported: true
}
```

Then imagine an aggregation pipeline is required to produce modified versions of the documents, as shown below:

```
// OUTPUT  (a record in the results of the executed aggregation)
{
  card_name: "Mrs. Jane A. Doe",
  card_num: "1234567890123456",
  card_expiry: ISODate("2023-08-31T23:59:59.736Z"), // Field type converted from
text
  card_sec_code: "123",
  card_provider_name: "Credit MasterCard Gold",
  transaction_id: "eb1bd77836e8713656d9bf2debba8900",
  transaction_date: ISODate("2021-01-13T09:32:07.000Z"),
  transaction_curncy_code: "GBP",
  transaction_amount: NumberDecimal("501.98"),
  reported: true,
  card_type: "CREDIT"                                  // New added literal value
field
}
```

Here, shown by the `//` comments, there was a requirement to modify each document's structure slightly, to convert the `card_expiry` text field into a proper date field, and add a new `card_type` field, set to the value "CREDIT", for every record.

Naively you might decide to build an aggregation pipeline using a `$project` stage to achieve this transformation, which would probably look similar to the following:

```
// BAD
[
  {"$project": {
    // Modify a field + add a new field
    "card_expiry": {"$dateFromString": {"dateString": "$card_expiry"}},
    "card_type": "CREDIT",

    // Must now name all the other fields for those fields to be retained
    "card_name": 1,
    "card_num": 1,
    "card_sec_code": 1,
    "card_provider_name": 1,
    "transaction_id": 1,
    "transaction_date": 1,
    "transaction_curncy_code": 1,
    "transaction_amount": 1,
    "reported": 1,

    // Remove _id field
    "_id": 0,
  }},
]
```

As you can see, the pipeline's stage is quite lengthy, and because you use a `$project` stage to modify/add two fields, you must also explicitly name each other existing field from the source records for inclusion. Otherwise, you will lose those fields during the transformation. Imagine if each payment document has hundreds of possible fields, rather than just ten!

A better approach to building the aggregation pipeline, to achieve the same results, would be to use `$set` and `$unset` instead, as shown below:

```
// GOOD
[
  {"$set": {
    // Modified + new field
    "card_expiry": {"$dateFromString": {"dateString": "$card_expiry"}},
    "card_type": "CREDIT",
  }},

  {"$unset": [
    // Remove _id field
    "_id",
  ]},
]
```

This time, when you need to add new documents to the collection of existing payments, which include additional new fields, e.g. `settlement_date` & `settlement_curncy_code`, no changes

are required. The existing aggregation pipeline allows these new fields to appear in the results automatically. However, when using `$project`, each time the possibility of a new field arises, a developer must first refactor the pipeline to incorporate an additional inclusion declaration (e.g. `"settlement_date": 1`, or `"settlement_curncy_code": 1`).

When To Use *$project*

It is best to use a `$project` stage when the required shape of output documents is very different from the input documents' shape. This situation often arises when you do not need to include most of the original fields.

This time for the same input payments collection, let us imagine you require a new aggregation pipeline to produce result documents. You need each output document's structure to be very different from the input structure, and you need to retain far fewer original fields, similar to the following:

```
// OUTPUT  (a record in the results of the executed aggregation)
{
  transaction_info: {
    date: ISODate("2021-01-13T09:32:07.000Z"),
    amount: NumberDecimal("501.98")
  },
  status: "REPORTED"
}
```

Using `$set` / `$unset` in the pipeline to achieve this output structure would be verbose and would require naming all the fields (for exclusion this time), as shown below:

```
// BAD
[
  {"$set": {
    // Add some fields
    "transaction_info.date": "$transaction_date",
    "transaction_info.amount": "$transaction_amount",
    "status": {"$cond": {"if": "$reported", "then": "REPORTED", "else":
"UNREPORTED"}},
  }},

  {"$unset": [
    // Remove _id field
    "_id",

    // Must name all other existing fields to be omitted
    "card_name",
    "card_num",
    "card_expiry",
    "card_sec_code",
    "card_provider_name",
    "transaction_id",
    "transaction_date",
    "transaction_curncy_code",
    "transaction_amount",
    "reported",
  ]},
]
```

However, by using `$project` for this specific aggregation, as shown below, to achieve the same results, the pipeline will be less verbose. The pipeline will have the flexibility of not requiring modification if you ever make subsequent additions to the data model, with new previously unknown fields:

```
// GOOD
[
  {"$project": {
    // Add some fields
    "transaction_info.date": "$transaction_date",
    "transaction_info.amount": "$transaction_amount",
    "status": {"$cond": {"if": "$reported", "then": "REPORTED", "else":
"UNREPORTED"}},

    // Remove _id field
    "_id": 0,
  }},
]
```

Another potential downside can occur when using `$project` to define field inclusion, rather than using `$set` (or `$addFields`). When using `$project` to declare all required fields for

inclusion, it can be easy for you to carelessly specify more fields from the source data than intended. Later on, if the pipeline contains something like a `$group` stage, this will cover up your mistake. The final aggregation's output will not include the erroneous field in the output. You might ask, "Why is this a problem?". Well, what happens if you intended for the aggregation to take advantage of a covered index query for the few fields it requires, to avoid unnecessarily accessing the raw documents. In most cases, MongoDB's aggregation engine can track fields' dependencies throughout a pipeline and, left to its own devices, can understand which fields are not required. However, you would be overriding this capability by explicitly asking for the extra field. A common error is to forget to exclude the `_id` field in the projection inclusion stage, and so it will be included by default. This mistake will silently kill the potential optimisation. If you must use a `$project` stage, try to use it as late as possible in the pipeline because it is then clear to you precisely what you are asking for as the aggregation's final output. Also, unnecessary fields like `_id` may already have been identified by the aggregation engine as no longer required, due to the occurrence of an earlier `$group` stage, for example.

Main Takeaway

In summary, you should always look to use `$set` (or `$addFields`) and `$unset` for field inclusion and exclusion, rather than `$project`. The main exception is if you have an obvious requirement for a very different structure for result documents, where you only need to retain a small subset of the input fields.

Using Explain Plans

When using the MongoDB Query Language (MQL) to develop queries, it is important to view the explain plan for a query to determine if you've used the appropriate index and if you need to optimise other aspects of the query or the data model. An explain plan allows you to understand fully the performance implications of the query you have created.

The same applies to aggregation pipelines and the ability to view an *explain plan* for the executed pipeline. However, with aggregations, an explain plan tends to be even more critical because considerably more complex logic can be assembled and run in the database. There are far more opportunities for performance bottlenecks to occur, requiring optimisation.

The MongoDB database engine will do its best to apply its own aggregation pipeline optimisations at runtime. Nevertheless, there could be some optimisations that only you can make. A database engine should never optimise a pipeline in such a way as to risk changing the functional behaviour and outcome of the pipeline. The database engine doesn't always have the extra context that your brain has, relating to the actual business problem at hand. It may

not be able to make some types of judgment calls about what pipeline changes to apply to make it run faster. The availability of an explain plan for aggregations enables you to bridge this gap. It allows you to understand the database engine's applied optimisations and detect further potential optimisations you can manually implement in the pipeline.

Viewing An Explain Plan

To view the explain plan for an aggregation pipeline, you can execute commands such as the following:

```
db.coll.explain().aggregate([{"$match": {"name": "Jo"}}]);
```

In this book, you will already have seen the convention used to firstly define a separate variable for the pipeline, followed by the call to the `aggregate()` function, passing in the pipeline argument, as shown here:

```
db.coll.aggregate(pipeline);
```

By adopting this approach, it's easier for you to use the same pipeline definition interchangeably with different commands. Whilst prototyping and debugging a pipeline, it is handy for you to be able to quickly switch from executing the pipeline to instead generating the explain plan for the same defined pipeline, as follows:

```
db.coll.explain().aggregate(pipeline);
```

As with MQL, there are three different verbosity modes that you can generate an explain plan with, as shown below:

```
// QueryPlanner verbosity  (default if no verbosity parameter provided)
db.coll.explain("queryPlanner").aggregate(pipeline);
```

```
// ExecutionStats verbosity
db.coll.explain("executionStats").aggregate(pipeline);
```

```
// AllPlansExecution verbosity
db.coll.explain("allPlansExecution").aggregate(pipeline);
```

In most cases, you will find that running the `executionStats` variant is the most informative mode. Rather than showing just the query planner's thought process, it also provides actual statistics on the "winning" execution plan (e.g. the total keys examined, the total docs examined, etc.). However, this isn't the default because it actually executes the aggregation in

addition to formulating the query plan. If the source collection is large or the pipeline is suboptimal, it will take a while to return the explain plan result.

Note, the aggregate() function also provides a vestigial `explain` option to ask for an explain plan to be generated and returned. Nonetheless, this is more limited and cumbersome to use, so you should avoid it.

Understanding The Explain Plan

To provide an example, let us assume a shop's data set includes information on each customer and what retail orders the customer has made over the years. The *customer orders* collection contains documents similar to the following example:

```
{
  "customer_id": "elise_smith@myemail.com",
  "orders": [
    {
      "orderdate": ISODate("2020-01-13T09:32:07Z"),
      "product_type": "GARDEN",
      "value": NumberDecimal("99.99")
    },
    {
      "orderdate": ISODate("2020-05-30T08:35:52Z"),
      "product_type": "ELECTRONICS",
      "value": NumberDecimal("231.43")
    }
  ]
}
```

You've defined an index on the `customer_id` field. You create the following aggregation pipeline to show the three most expensive orders made by a customer whose ID is `tonijones@myemail.com`, as shown below:

```
var pipeline = [
  // Unpack each order from customer orders array as a new separate record
  {"$unwind": {
    "path": "$orders",
  }},

  // Match on only one customer
  {"$match": {
    "customer_id": "tonijones@myemail.com",
  }},

  // Sort customer's purchases by most expensive first
  {"$sort" : {
    "orders.value" : -1,
  }},

  // Show only the top 3 most expensive purchases
  {"$limit" : 3},

  // Use the order's value as a top level field
  {"$set": {
    "order_value": "$orders.value",
  }},

  // Drop the document's id and orders sub-document from the results
  {"$unset" : [
    "_id",
    "orders",
  ]},
];
```

Upon executing this aggregation against an extensive sample data set, you receive the following result:

```
[
  {
    customer_id: 'tonijones@myemail.com',
    order_value: NumberDecimal("1024.89")
  },
  {
    customer_id: 'tonijones@myemail.com',
    order_value: NumberDecimal("187.99")
  },
  {
    customer_id: 'tonijones@myemail.com',
    order_value: NumberDecimal("4.59")
  }
]
```

You then request the *query planner* part of the explain plan:

```
db.customer_orders.explain("queryPlanner").aggregate(pipeline);
```

The query plan output for this pipeline shows the following (excluding some information for brevity):

```
stages: [
  {
    '$cursor': {
      queryPlanner: {
        parsedQuery: { customer_id: { '$eq': 'tonijones@myemail.com' } },
        winningPlan: {
          stage: 'FETCH',
          inputStage: {
            stage: 'IXSCAN',
            keyPattern: { customer_id: 1 },
            indexName: 'customer_id_1',
            direction: 'forward',
            indexBounds: {
              customer_id: [
                '["tonijones@myemail.com", "tonijones@myemail.com"]'
              ]
            }
          }
        },
      }
    }
  },

  { '$unwind': { path: '$orders' } },

  { '$sort': { sortKey: { 'orders.value': -1 }, limit: 3 } },

  { '$set': { order_value: '$orders.value' } },

  { '$project': { _id: false, orders: false } }
]
```

You can deduce some illuminating insights from this query plan:

- To optimise the aggregation, the database engine has reordered the pipeline positioning the filter belonging to the `$match` to the top of the pipeline. The database engine moves the content of `$match` ahead of the `$unwind` stage without changing the aggregation's functional behaviour or outcome.
- The first stage of the database optimised version of the pipeline is an *internal* `$cursor` stage, regardless of the order you placed the pipeline stages in. The `$cursor` *runtime* stage is always the first action executed for any aggregation. Under the covers, the aggregation engine re-uses the MQL query engine to perform a "regular" query against the collection, with a filter based on the aggregation's `$match` contents. The aggregation

runtime uses the resulting query cursor to pull batches of records. This is similar to how a client application with a MongoDB driver uses a query cursor when remotely invoking an MQL query to pull batches. As with a normal MQL query, the regular database query engine will try to use an index if it makes sense. In this case an index is indeed leveraged, as is visible in the embedded `$queryPlanner` metadata, showing the `"stage" : "IXSCAN"` element and the index used, `"indexName" : "customer_id_1"`.

- To further optimise the aggregation, the database engine has collapsed the `$sort` and `$limit` into a single *special internal sort stage* which can perform both actions in one go. In this situation, during the sorting process, the aggregation engine only has to track the current three most expensive orders in memory. It does not have to hold the whole data set in memory when sorting, which may otherwise be resource prohibitive in many scenarios, requiring more RAM than is available.

You might also want to see the *execution stats* part of the explain plan. The specific new information shown in `executionStats`, versus the default of `queryPlanner`, is identical to the normal MQL explain plan returned for a regular `find()` operation. Consequently, for aggregations, similar principles to MQL apply for spotting things like "have I used the optimal index?" and "does my data model lend itself to efficiently processing this query?".

You ask for the *execution stats* part of the explain plan:

```
db.customer_orders.explain("executionStats").aggregate(pipeline);
```

Below is a redacted example of the output you will see, highlighting some of the most relevant metadata elements you should generally focus on.

```
executionStats: {
  nReturned: 1,
  totalKeysExamined: 1,
  totalDocsExamined: 1,
  executionStages: {
    stage: 'FETCH',
    nReturned: 1,
    works: 2,
    advanced: 1,
    docsExamined: 1,
    inputStage: {
      stage: 'IXSCAN',
      nReturned: 1,
      works: 2,
      advanced: 1,
      keyPattern: { customer_id: 1 },
      indexName: 'customer_id_1',
      direction: 'forward',
      indexBounds: {
        customer_id: [
          '["tonijones@myemail.com", "tonijones@myemail.com"]'
        ]
      },
      keysExamined: 1,
    }
  }
}
```

Here, this part of the plan also shows that the aggregation uses the existing index. Because `totalKeysExamined` and `totalDocsExamined` match, the aggregation fully leverages this index to identify the required records, which is good news. Nevertheless, the targeted index doesn't necessarily mean the aggregation's query part is fully optimised. For example, if there is the need to reduce latency further, you can do some analysis to determine if the index can completely cover the query. Suppose the *cursor query* part of the aggregation is satisfied entirely using the index and does not have to examine any raw documents. In that case, you will see `totalDocsExamined: 0` in the explain plan.

Pipeline Performance Considerations

Similar to any programming language, there is a downside if you prematurely optimise an aggregation pipeline. You risk producing an over-complicated solution that doesn't address the performance challenges that will manifest. As described in the previous chapter, Using Explain Plans, the tool you should use to identify opportunities for optimisation is the *explain plan*. You will typically use the explain plan during the final stages of your pipeline's development once it is functionally correct.

With all that said, it can still help you to be aware of some guiding principles regarding performance whilst you are prototyping a pipeline. Critically, such guiding principles will be invaluable to you once the aggregation's explain plan is analysed and if it shows that the current pipeline is sub-optimal.

This chapter outlines three crucial tips to assist you when creating and tuning an aggregation pipeline. For sizeable data sets, adopting these principles may mean the difference between aggregations completing in a few seconds versus minutes, hours or even longer.

1. Be Cognizant Of Streaming Vs Blocking Stages Ordering

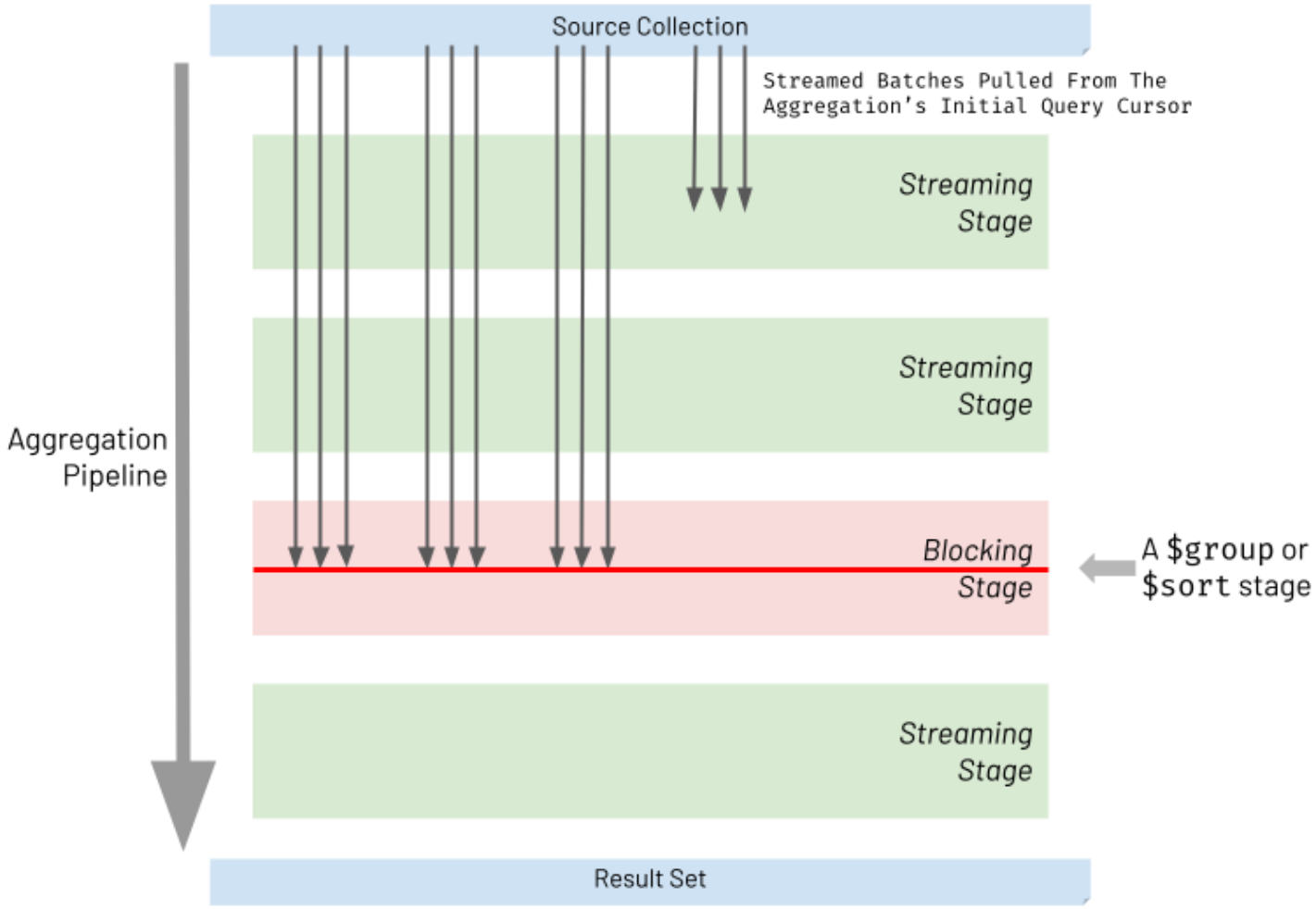
When executing an aggregation pipeline, the database engine pulls batches of records from the initial query cursor generated against the source collection. The database engine then attempts to stream each batch through the aggregation pipeline stages. For most types of stages, referred to as *streaming stages*, the database engine will take the processed batch from one stage and immediately stream it into the next part of the pipeline. It will do this without waiting for all the other batches to arrive at the prior stage. However, two types of stages must block and wait for all batches to arrive and accumulate together at that stage. These two stages are referred to as *blocking stages* and specifically, the two types of stages that block are:

- `$sort`
- `$group` *

---

* *actually when stating* `$group`*, this also includes other less frequently used "grouping" stages too, specifically:* `$bucket`, `$bucketAuto`, `$count`, `$sortByCount` & `$facet` *(it's a stretch to call* `$facet` *a group stage, but in the context of this topic, it's best to think of it that way)*

The diagram below highlights the nature of streaming and blocking stages. Streaming stages allow batches to be processed and then passed through without waiting. Blocking stages wait for the whole of the input data set to arrive and accumulate before processing all this data together.



When considering `$sort` and `$group` stages, it becomes evident why they have to block. The following examples illustrate why this is the case:

1. ***$sort* blocking example**: A pipeline must sort *people* in ascending order of *age*. If the stage only sorts each batch's content before passing the batch on to the pipeline's result, only individual batches of output records are sorted by age but not the whole result set.

2. ***$group* blocking example**: A pipeline must group *employees* by one of two *work departments* (either the *sales* or *manufacturing* departments). If the stage only groups employees for a batch, before passing it on, the final result contains the work departments repeated multiple times. Each duplicate department consists of some but not all of its employees.

These often unavoidable blocking stages don't just increase aggregation execution time by reducing concurrency. If used without careful forethought, the throughput and latency of a pipeline will slow dramatically due to significantly increased memory consumption. The following sub-sections explore why this occurs and tactics to mitigate this.

$sort Memory Consumption And Mitigation

Used naïvely, a `$sort` stage will need to see all the input records at once, and so the host server must have enough capacity to hold all the input data in memory. The amount of memory required depends heavily on the initial data size and the degree to which the prior stages can reduce the size. Also, multiple instances of the aggregation pipeline may be in-flight at any one time, in addition to other database workloads. These all compete for the same finite memory. Suppose the source data set is many gigabytes or even terabytes in size, and earlier pipeline stages have not reduced this size significantly. It will be unlikely that the host machine has sufficient memory to support the pipeline's blocking `$sort` stage. Therefore, MongoDB enforces every stage is limited to 100 MB of consumed RAM. The database throws an error if it exceeds this limit.

To avoid the memory limit obstacle, you can set the `allowDiskUse:true` option for the overall aggregation for handling large result data sets. Consequently, the pipeline's *sort* operation spills to disk if required, and the 100 MB limit no longer constrains the pipeline. However, the sacrifice here is significantly higher latency, and the execution time is likely to increase by orders of magnitude.

To circumvent the aggregation needing to manifest the whole data set in memory or overspill to disk, attempt to refactor your pipeline to incorporate one of the following approaches (in order of most effective first):

1. **Use Index Sort**. If the `$sort` stage does not depend on a `$unwind`, `$group` or `$project` stage preceding it, move the `$sort` stage to near the start of your pipeline to target an index for the sort. The aggregation runtime does not need to perform an expensive in-memory sort operation as a result. The `$sort` stage won't necessarily be the first stage in your pipeline because there may also be a `$match` stage that takes advantage of the same index. Always inspect the explain plan to ensure you are inducing the intended behaviour.

2. **Use Limit With Sort**. If you only need the first subset of records from the sorted set of data, add a `$limit` stage directly after the `$sort` stage, limiting the results to the fixed amount you require (e.g. 10). At runtime, the aggregation engine will collapse the `$sort` and `$limit` into a single special internal sort stage which performs both actions together. The in-flight sort process only has to track the ten records in memory, which currently satisfy the executing sort/limit rule. It does not have to hold the whole data set in memory to execute the sort successfully.

3. **Reduce Records To Sort**. Move the `$sort` stage to as late as possible in your pipeline and ensure earlier stages significantly reduce the number of records streaming into this late blocking `$sort` stage. This blocking stage will have fewer records to process and less thirst for RAM.

$group Memory Consumption And Mitigation

Like the `$sort` stage, the `$group` stage has the potential to consume a large amount of memory. The aggregation pipeline's 100 MB RAM limit equally applies to the `$group` stage. As with sorting, you can use the pipeline's `allowDiskUse:true` option to avoid this limit for heavyweight grouping operations, but with the same downsides.

In reality, most grouping scenarios focus on accumulating summary data such as totals, counts, averages, highs and lows, and not itemised data. In these situations, considerably reduced result data sets are produced, requiring far less processing memory than a `$sort` stage. Contrary to many sorting scenarios, grouping operations will typically demand a fraction of the host's RAM.

To ensure you avoid excessive memory consumption when you are looking to use a `$group` stage, adopt the following principles:

1. **Avoid Unnecessary Grouping**. This chapter covers this recommendation in far greater detail in the section *2. Avoid Unwinding & Regrouping Documents Just To Process Array Elements*.

2. **Group Summary Data Only**. If the use case permits it, use the group stage to accumulate things like totals, counts and summary roll-ups only, rather than holding all the raw data of each record belonging to a group. The Aggregation Framework provides a robust set of accumulator operators to help you achieve this inside a `$group` stage.

2. Avoid Unwinding & Regrouping Documents Just To Process Array Elements

Sometimes, you need an aggregation pipeline to mutate or reduce an array field's content for each record. For example:

- You may need to add together all the values in the array into a total field
- You may need to retain the first and last elements of the array only
- You may need to retain only one recurring field for each sub-document in the array
- *...or numerous other array "reduction" scenarios*

To bring this to life, imagine a retail `orders` collection where each document contains an array of products purchased as part of the order, as shown in the example below:

```
[
  {
    _id: 1197372932325,
    products: [
      {
        prod_id: 'abc12345',
        name: 'Asus Laptop',
        price: NumberDecimal('429.99')
      }
    ]
  },
  {
    _id: 4433997244387,
    products: [
      {
        prod_id: 'def45678',
        name: 'Karcher Hose Set',
        price: NumberDecimal('23.43')
      },
      {
        prod_id: 'jkl77336',
        name: 'Picky Pencil Sharpener',
        price: NumberDecimal('0.67')
      },
      {
        prod_id: 'xyz11228',
        name: 'Russell Hobbs Chrome Kettle',
        price: NumberDecimal('15.76')
      }
    ]
  }
]
```

The retailer wants to see a report of all the orders but only containing the expensive products purchased by customers (e.g. having just products priced greater than 15 dollars). Consequently, an aggregation is required to filter out the inexpensive product items of each order's array. The desired aggregation output might be:

```
[
  {
    _id: 1197372932325,
    products: [
      {
        prod_id: 'abc12345',
        name: 'Asus Laptop',
        price: NumberDecimal('429.99')
      }
    ]
  },
  {
    _id: 4433997244387,
    products: [
      {
        prod_id: 'def45678',
        name: 'Karcher Hose Set',
        price: NumberDecimal('23.43')
      },
      {
        prod_id: 'xyz11228',
        name: 'Russell Hobbs Chrome Kettle',
        price: NumberDecimal('15.76')
      }
    ]
  }
]
```

Notice order `4433997244387` now only shows two products and is missing the inexpensive product.

One naïve way of achieving this transformation is to *unwind* the *products* array of each order document to produce an intermediate set of individual product records. These records can then be *matched* to retain products priced greater than 15 dollars. Finally, the products can be *grouped* back together again by each order's `_id` field. The required pipeline to achieve this is below:

```
// SUBOPTIMAL

var pipeline = [
  // Unpack each product from the each order's product as a new separate record
  {"$unwind": {
    "path": "$products",
  }},

  // Match only products valued over 15.00
  {"$match": {
    "products.price": {
      "$gt": NumberDecimal("15.00"),
    },
  }},

  // Group by product type
  {"$group": {
    "_id": "$_id",
    "products": {"$push": "$products"},
  }},
];
```

This pipeline is suboptimal because a `$group` stage has been introduced, which is a blocking stage, as outlined earlier in this chapter. Both memory consumption and execution time will increase significantly, which could be fatal for a large input data set. There is a far better alternative by using one of the Array Operators instead. Array Operators are sometimes less intuitive to code, but they avoid introducing a blocking stage into the pipeline. Consequently, they are significantly more efficient, especially for large data sets. Shown below is a far more economical pipeline, using the `$filter` array operator, rather than the `$unwind/$match/$group` combination, to produce the same outcome:

```
// OPTIMAL

var pipeline = [
  // Filter out products valued 15.00 or less
  {"$set": {
    "products": {
      "$filter": {
        "input": "$products",
        "as": "product",
        "cond": {"$gt": ["$$product.price", NumberDecimal("15.00")]},
      }
    },
  }},
];
```

Unlike the suboptimal pipeline, the optimal pipeline will include "empty orders" in the results for those orders that contained only inexpensive items. If this is a problem, you can include a

simple `$match` stage at the start of the optimal pipeline with the same content as the `$match` stage shown in the suboptimal example.

To reiterate, there should never be the need to use an `$unwind/$group` combination in an aggregation pipeline to transform an array field's elements for each document in isolation. One way to recognize if you have this anti-pattern is if your pipeline contains a `$group` on a `$_id` field. Instead, use *Array Operators* to avoid introducing a blocking stage. Otherwise, you will suffer a magnitude of increase in execution time when your pipeline handles more than 100MB of in-flight data. Adopting this best practice may mean the difference between achieving the required business outcome and abandoning the whole task as unachievable.

The primary use of an `$unwind/$group` combination is to correlate patterns across many records' arrays rather than transforming the content within each input record's array only. For an illustration of an appropriate use of `$unwind/$group` refer to this book's Unpack Array & Group Differently example.

3. Encourage Match Filters To Appear Early In The Pipeline

Explore If Bringing Forward A Full Match Is Possible

As discussed, the database engine will do its best to optimise the aggregation pipeline at runtime, with a particular focus on attempting to move the `$match` stages to the top of the pipeline. Top-level `$match` content will form part of the filter that the engine first executes as the initial query. The aggregation then has the best chance of leveraging an index. However, it may not always be possible to promote `$match` filters in such a way without changing the meaning and resulting output of an aggregation.

Sometimes, a `$match` stage is defined later in a pipeline to perform a filter on a field that the pipeline computed in an earlier stage. The computed field isn't present in the pipeline's original input collection. Some examples are:

- A pipeline where a `$group` stage creates a new `total` field based on an accumulator operator. Later in the pipeline, a `$match` stage filters groups where each group's `total` is greater than `1000`.
- A pipeline where a `$set` stage computes a new `total` field value based on adding up all the elements of an array field in each document. Later in the pipeline, a `$match` stage filters documents where the `total` is less than `50`.

At first glance, it may seem like the match on the computed field is irreversibly trapped behind an earlier stage that computed the field's value. Indeed the aggregation engine cannot automatically optimise this further. In some situations, though, there may be a missed opportunity where beneficial refactoring is possible by you, the developer.

Take the following trivial example of a collection of *customer order* documents:

```
[
  {
    customer_id: 'elise_smith@myemail.com',
    orderdate: ISODate('2020-05-30T08:35:52.000Z'),
    value: NumberDecimal('9999')
  }
  {
    customer_id: 'elise_smith@myemail.com',
    orderdate: ISODate('2020-01-13T09:32:07.000Z'),
    value: NumberDecimal('10101')
  }
]
```

Let's assume the orders are in a *Dollars* currency, and each `value` field shows the order's value in *cents*. You may have built a pipeline to display all orders where the value is greater than 100 dollars like below:

```
// SUBOPTIMAL

var pipeline = [
  {"$set": {
    "value_dollars": {"$multiply": [0.01, "$value"]}, // Converts cents to dollars
  }},

  {"$unset": [
    "_id",
    "value",
  ]},

  {"$match": {
    "value_dollars": {"$gte": 100},  // Peforms a dollar check
  }},
];
```

The collection has an index defined for the `value` field (in *cents*). However, the `$match` filter uses a computed field, `value_dollars`. When you view the explain plan, you will see the pipeline does not leverage the index. The `$match` is trapped behind the `$set` stage (which computes the field) and cannot be moved to the pipeline's start. MongoDB's aggregation engine tracks a field's dependencies across multiple stages in a pipeline. It can establish how far up the pipeline it can promote fields without risking a change in the aggregation's

behaviour. In this case, it knows that if it moves the `$match` stage ahead of the `$set` stage, it depends on, things will not work correctly.

In this example, as a developer, you can easily make a pipeline modification that will enable this pipeline to be more optimal without changing the pipeline's intended outcome. Change the `$match` filter to be based on the source field `value` instead (greater than `10000` cents), rather than the computed field (greater than `100` dollars). Also, ensure the `$match` stage appears before the `$unset` stage (which removes the `value` field). This change is enough to allow the pipeline to run efficiently. Below is how the pipeline looks after you have made this change:

```
// OPTIMAL

var pipeline = [
  {"$set": {
    "value_dollars": {"$multiply": [0.01, "$value"]},
  }},

  {"$match": {                // Moved to before the $unset
    "value": {"$gte": 10000},   // Changed to perform a cents check
  }},

  {"$unset": [
    "_id",
    "value",
  ]},
];
```

This pipeline produces the same data output. However, when you look at its explain plan, it shows the database engine has pushed the `$match` filter to the top of the pipeline and used an index on the `value` field. The aggregation is now optimal because the `$match` stage is no longer "blocked" by its dependency on the computed field.

Explore If Bringing Forward A Partial Match Is Possible

There may be some cases where you can't unravel a computed value in such a manner. However, it may still be possible for you to include an additional `$match` stage, to perform a *partial match* targeting the aggregation's query cursor. Suppose you have a pipeline that masks the values of sensitive `date_of_birth` fields (replaced with computed `masked_date` fields). The computed field adds a random number of days (one to seven) to each current date. The pipeline already contains a `$match` stage with the filter `masked_date > 01-Jan-2020`. The runtime cannot optimise this to the top of the pipeline due to the dependency on a computed value. Nevertheless, you can manually add an extra `$match` stage at the top of the pipeline, with the filter `date_of_birth > 25-Dec-2019`. This new `$match` leverages an index and filters records seven days earlier than the existing `$match`, but the aggregation's final output is the

same. The new `$match` may pass on a few more records than intended. However, later on, the pipeline applies the existing filter `masked_date > 01-Jan-2020` that will naturally remove surviving surplus records before the pipeline completes.

Pipeline Match Summary

In summary, if you have a pipeline leveraging a `$match` stage and the explain plan shows this is not moving to the start of the pipeline, explore whether manually refactoring will help. If the `$match` filter depends on a computed value, examine if you can alter this or add an extra `$match` to yield a more efficient pipeline.

Expressions Explained

Summarising Aggregation Expressions

Expressions give aggregation pipelines their data manipulation power. However, they tend to be something that developers start using by just copying examples from the MongoDB Manual and then refactoring these without thinking enough about what they are. Proficiency in aggregation pipelines demands a deeper understanding of expressions.

Aggregation expressions come in one of three primary flavours:

- **Operators.** Accessed as an object with a `$` prefix followed by the operator function name. The "*dollar-operator-name*" is used as the main key for the object. Examples: `{$arrayElemAt: ...}`, `{$cond: ...}`, `{$dateToString: ...}`
- **Field Paths.** Accessed as a string with a `$` prefix followed by the field's path in each record being processed. Examples: `"$account.sortcode"`, `"$addresses.address.city"`
- **Variables.** Accessed as a string with a `$$` prefix followed by the fixed name and falling into three sub-categories:
    - **Context System Variables.** With values coming from the system environment rather than each input record an aggregation stage is processing. Examples: `"$$NOW"`, `"$$CLUSTER_TIME"`
    - **Marker Flag System Variables.** To indicate desired behaviour to pass back to the aggregation runtime. Examples: `"$$ROOT"`, `"$$REMOVE"`, `"$$PRUNE"`

- **Bind User Variables.** For storing values you declare with a `$let` operator (or with the `let` option of a `$lookup` stage, or `as` option of a `$map` or `$filter` stage). Examples: `"$$product_name_var"`, `"$$orderIdVal"`

You can combine these three categories of aggregation expressions when operating on input records, enabling you to perform complex comparisons and transformations of data. To highlight this, the code snippet below is an excerpt from this book's Mask Sensitive Fields example, which combines all three expressions.

```
"customer_info": {"$cond": {
                    "if":   {"$eq": ["$customer_info.category", "SENSITIVE"]},
                    "then": "$$REMOVE",
                    "else": "$customer_info",
                  }}
```

The pipeline retains an embedded sub-document (`customer_info`) in each resulting record unless a field in the original sub-document has a specific value (`category=SENSITIVE`). `{$cond: ...}` is one of the operator expressions used in the excerpt (a "conditional" operator expression which takes three arguments: `if`, `then` & `else`). `{$eq: ...}` is another operator expression (a "comparison" operator expression). `"$$REMOVE"` is a "marker flag" variable expression instructing the pipeline to exclude the field. Both `"$customer_info.category"` and `"$customer_info"` elements are field path expressions referencing each incoming record's fields.

What Do Expressions Produce?

As described above, an expression can be an Operator (e.g. `{$concat: ...}`), a Variable (e.g. `"$$ROOT"`) or a Field Path (e.g. `"$address"`). In all these cases, an expression is just something that dynamically populates and returns a new JSON/BSON data type element, which can be one of:

- a Number *(including integer, long, float, double, decimal128)*
- a String *(UTF-8)*
- a Boolean
- a DateTime *(UTC)*
- an Array
- an Object

However, a specific expression can restrict you to returning just one or a few of these types. For example, the `{$concat: ...}` Operator, which combines multiple strings, can only

produce a *String* data type (or null). The Variable `"$$ROOT"` can only return an *Object* which refers to the root document currently being processed in the pipeline stage.

A Field Path (e.g. `"$address"`) is different and can return an element of any data type, depending on what the field refers to in the current input document. For example, suppose `"$address"` references a sub-document. In this case, it will return an *Object*. However, if it references a list of elements, it will return an *Array*. As a human, you can guess that the Field Path `"$address"` won't return a *DateTime*, but the aggregation runtime does not know this ahead of time. There could be even more dynamics at play. Due to MongoDB's flexible data model, `"$address"` could yield a different type for each record processed in a pipeline stage. The first record's `address` may be an *Object* sub-document with street name and city fields. The second record's `address` might represent the full address as a single *String*.

In summary, *Field Paths* and *Bind User Variables* are expressions that can return any JSON/BSON data type at runtime depending on their context. For the other kinds of expressions (*Operators*, *Context System Variables* and *Marker Flag System Variables*), the data type each can return is fixed to one or a set number of documented types. To establish the exact data type produced by these specific operators, you need to consult the Aggregation Pipeline Quick Reference documentation.

For the Operator category of expressions, an expression can also take other expressions as parameters, making them composable. Suppose you need to determine the day of the week for a given date, for example:

```
{"$dayOfWeek": ISODate("2021-04-24T00:00:00Z")}
```

Here the `$dayOfWeek` Operator expression can only return an element of type *Number* and takes a single parameter, an element of type *DateTime*. However, rather than using a hardcoded date-time for the parameter, you could have provided an expression. This could be a *Field Path* expression, for example:

```
{"$dayOfWeek": "$person_details.data_of_birth"}
```

Alternatively, you could have defined the parameter using a *Context System Variable* expression, for example:

```
{"$dayOfWeek": "$$NOW"}
```

Or you could even have defined the parameter using yet another *Operator* expression, for example:

```
{"$dayOfWeek": {"$dateFromParts": {"year" : 2021, "month" : 4, "day": 24}}}
```

Furthermore, you could have defined `year`, `month` and `day` parameters for `$dateFromParts` to be dynamically generated using expressions rather than literal values. The ability to chain expressions together in this way gives your pipelines a lot of power and flexibility when you need it.

Can All Stages Use Expressions?

The following question is something you may not have asked yourself before, but asking this question and considering why the answer is what it is can help reveal more about what aggregation expressions are and why you use them.

Question: Can aggregation expressions be used within any type of pipeline stage?

Answer: No

There are many types of stages in the Aggregation Framework that don't allow expressions to be embedded. Examples of some of the most commonly used of these stages are:

- `$match`
- `$limit`
- `$skip`
- `$sort`
- `$count`
- `$lookup`
- `$out`

Some of these stages may be a surprise to you if you've never really thought about it before. You might well consider `$match` to be the most surprising item in this list. The content of a `$match` stage is just a set of query conditions with the same syntax as MQL rather than an aggregation expression. There is a good reason for this. The aggregation engine re-uses the MQL query engine to perform a "regular" query against the collection, enabling the query engine to use all its usual optimisations. The query conditions are taken as-is from the `$match` stage at the top of the pipeline. Therefore, the `$match` filter must use the same syntax as MQL.

In most of the stages that are unable to leverage expressions, it doesn't usually make sense for their behaviour to be dynamic, based on the pipeline data entering the stage. For a client application that paginates results, you might define a value of `20` for the `$limit` stage. However, maybe you want to dynamically bind a value to the `$limit` stage, sourced by a `$lookup` stage earlier in the pipeline. The lookup operation might pull in the user's preferred "page list size" value from a "user preferences" collection. Nonetheless, the Aggregation

Framework does not support this today for the listed stage types to avoid the overhead of the extra checks it would need to perform for what are essentially rare cases.

In most cases, only one of the listed stages needs to be more expressive: the `$match` stage, but this stage is already flexible by being based on MQL query conditions. However, sometimes, even MQL isn't expressive enough to sufficiently define a rule to identify records to retain in an aggregation. The remainder of this chapter explores these challenges and how they are solved.

What Is Using *$expr* Inside *$match* All About?

The previously stated generalisation about `$match` not supporting expressions is actually inaccurate. Version 3.6 of MongoDB introduced the $expr operator, which you can embed within a `$match` stage (or in MQL) to leverage aggregation expressions when filtering records. Essentially, this enables MongoDB's query runtime (which executes an aggregation's `$match`) to re-use expressions provided by MongoDB's aggregation runtime.

Inside an `$expr` operator, you can include any composite expression fashioned from `$` operator functions, `$` field paths and `$$` variables. A few situations demand having to use `$expr` from inside a `$match` stage. Examples include:

- A requirement to compare two fields from the same record to determine whether to keep the record based on the comparison's outcome
- A requirement to perform a calculation based on values from multiple existing fields in each record and then comparing the calculation to a constant

These are impossible in an aggregation (or MQL `find()`) if you use regular `$match` query conditions.

Take the example of a collection holding information on different instances of rectangles (capturing their width and height), similar to the following:

```
[
  { _id: 1, width: 2, height: 8 },
  { _id: 2, width: 3, height: 4 },
  { _id: 3, width: 20, height: 1 }
]
```

What if you wanted to run an aggregation pipeline to only return rectangles with an `area` greater than `12`? This comparison isn't possible in a conventional aggregation when using a single `$match` query condition. However, with `$expr`, you can analyse a combination of fields in-situ using expressions. You can implement the requirement with the following pipeline:

```
var pipeline = [
  {"$match": {
    "$expr": {"$gt": [{"$multiply": ["$width", "$height"]}, 12]},
  }},
];
```

The result of executing an aggregation with this pipeline is:

```
[
  { _id: 1, width: 2, height: 8 },
  { _id: 3, width: 20, height: 1 }
]
```

As you can see, the second of the three shapes is not output because its area is only `12` (`3 x 4`).

Restrictions When Using Expressions with *$match*

You should be aware that there are restrictions on when the runtime can benefit from an index when using a `$expr` operator inside a `$match` stage. This partly depends on the version of MongoDB you are running. Using `$expr`, you can leverage a `$eq` comparison operator with some constraints, including an inability to use a multi-key index. For MongoDB versions before 5.0, if you use a "range" comparison operator (`$gt`, `$gte`, `$lt` and `$lte`), an index cannot be employed to match the field, but this works fine in version 5.0 and greater.

There are also subtle differences when ordering values for a specific field across multiple documents when some values have different types. MongoDB's query runtime (which executes regular MQL and `$match` filters) and MongoDB's aggregation runtime (which implements `$expr`) can apply different ordering rules when filtering, referred to as "type bracketing". Consequently, a range query may not yield the same result with `$expr` as it does with MQL if some values have different types.

Due to the potential challenges outlined, only use a `$expr` operator in a `$match` stage if there is no other way of assembling the filter criteria using regular MQL syntax.

Sharding Considerations

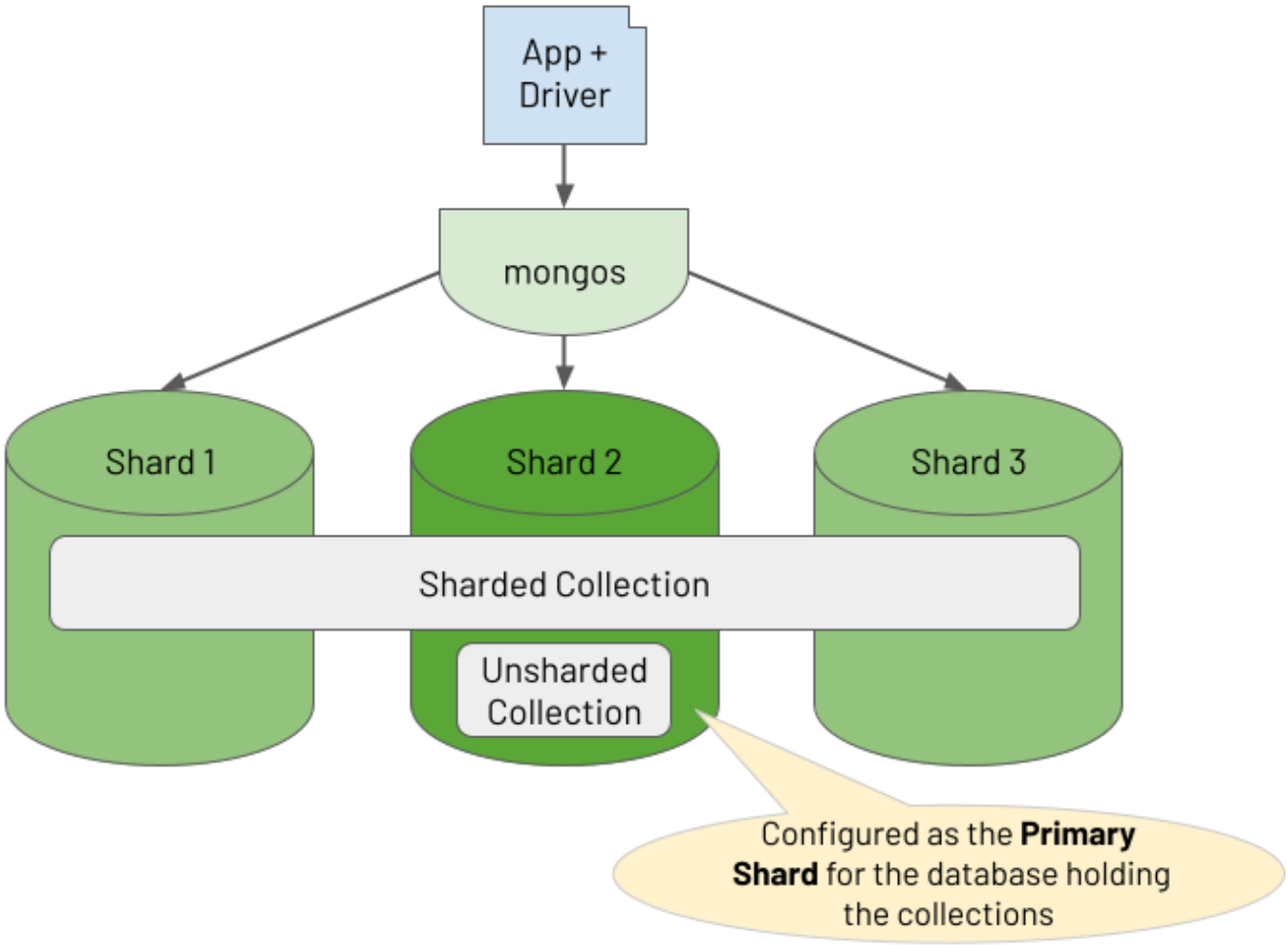
MongoDB Sharding isn't just an effective way to scale out your database to hold more data and support higher transactional throughput. Sharding also helps you scale out your analytical workloads, potentially enabling aggregations to complete far quicker. Depending on the nature of your aggregation and some adherence to best practices, the cluster may execute parts of the aggregation in parallel over multiple shards for faster completion.

There is no difference between a replica set and a sharded cluster regarding the functional capabilities of the aggregations you build, except for a minimal set of constraints. This chapter's *Sharded Aggregation Constraints* section outlines these constraints. When it comes to optimising your aggregations, in most cases, there will be little to no difference in the structure of a pipeline when refactoring for performance on a sharded cluster compared to a simple replica set. You should always adhere to the advice outlined in the chapter Pipeline Performance Considerations. The aggregation runtime takes care of distributing the appropriate parts of your pipeline to each shard that holds the required data. The runtime then transparently coalesces the results from these shards in the most optimal way possible. Nevertheless, it is worth understanding how the aggregation engine distributes work and applies its sharded optimisations in case you ever suffer a performance problem and need to dig deeper into why.

Brief Summary Of Sharded Clusters

In a sharded cluster, you partition a collection of data across multiple shards, where each shard runs on a separate set of host machines. You control how the system distributes the data by defining a shard key rule. Based on the shard key of each document, the system groups subsets of documents together into "chunks", where a range of shard key values identifies each chunk. The cluster balances these chunks across its shards.

In addition to holding sharded collections in a database, you may also be storing unsharded collections in the same database. All of a database's unsharded collections live on one specific shard in the cluster, designated as the "primary shard" for the database (not to be confused with a replica set's "primary replica"). The diagram below shows the relationship between a database's collections and the shards in the cluster.



One or more deployed mongos processes act as a reverse proxy, routing read and write operations from the client application to the appropriate shards. For document write operations (i.e. create, update, delete), a mongos router knows which shard the document lives on and routes the operation to that specific shard. For read operations, if the query includes the shard key, the mongos knows which shards hold the required documents to route the query to (called "targeting"). If the query does not include the shard key, it sends the query to all shards using a "scatter/gather" pattern (called "broadcasting"). These are the rules for sharded reads and writes, but the approach for sharded aggregations requires a deeper explanation. Consequently, the rest of this chapter outlines how a sharded cluster handles the routing and execution of aggregations.

Sharded Aggregation Constraints

Some of MongoDB's stages only partly support sharded aggregations depending on which version of MongoDB you are running. These stages all happen to reference a second collection in addition to the pipeline's source input collection. In each case, the pipeline can use a

sharded collection as its source, but the second collection referenced must be unsharded (for earlier MongoDB versions, at least). The affected stages and versions are:

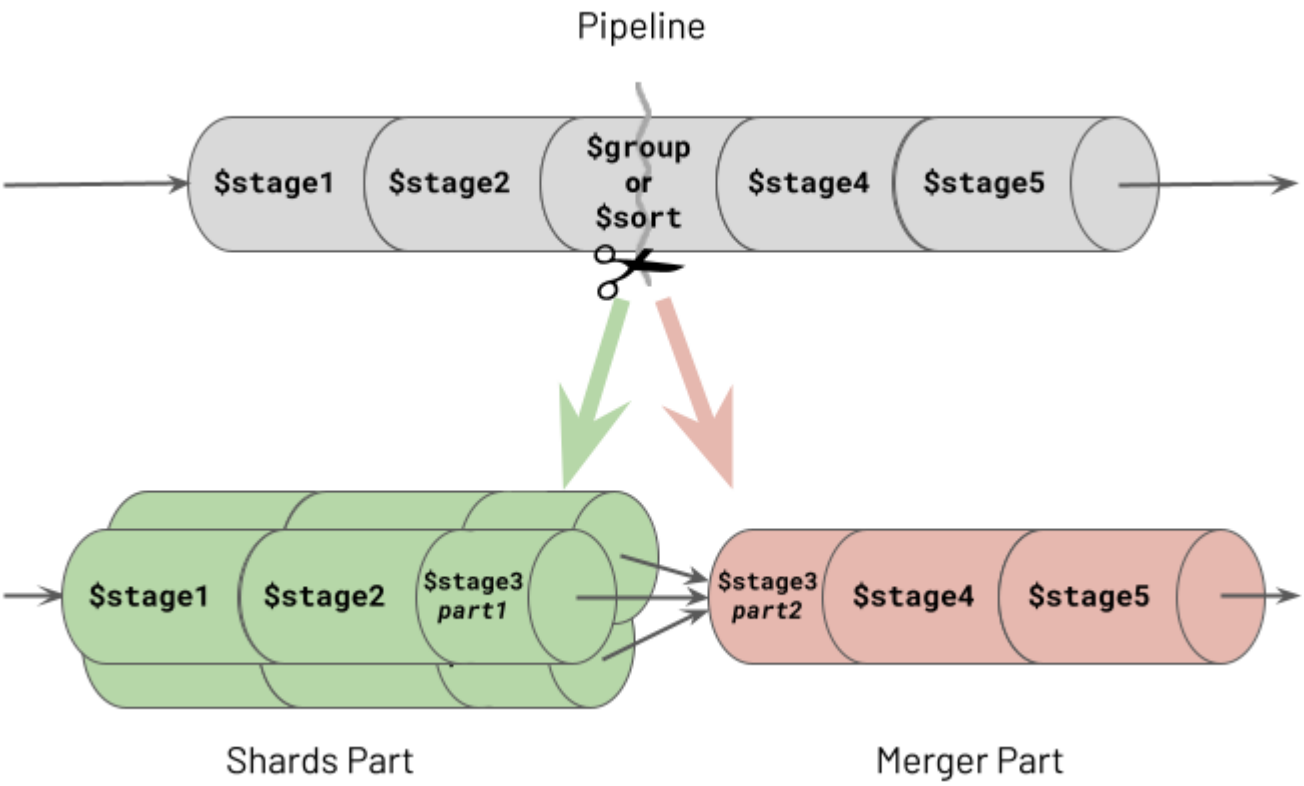
- `$lookup`. In MongoDB versions prior to 5.1, the other referenced collection to join with must be unsharded.
- `$graphLookup`. In MongoDB versions prior to 5.1, the other referenced collection to recursively traverse must be unsharded.
- `$out`. In all MongoDB versions, the other referenced collection used as the destination of the aggregation's output must be unsharded. However, you can use a `$merge` stage instead to output the aggregation result to a sharded collection.

Where Does A Sharded Aggregation Run?

Sharded clusters provide the opportunity to reduce the response times of aggregations. For example, there may be an unsharded collection containing billions of documents where it takes 60 seconds for an aggregation pipeline to process all this data. Instead, suppose a cluster of four shards is hosting this same collection of evenly balanced data. Depending on the nature of the aggregation, it may be possible for the cluster to execute the aggregation's pipeline concurrently on each shard. Consequently, the same aggregation's total data processing time may be closer to 15 seconds. However, this won't always be the case because certain types of pipelines will demand combining substantial amounts of data from multiple shards for further processing. The aggregation's response time could go in the opposite direction in such circumstances, completing in far longer than 60 seconds due to the significant network transfer and marshalling overhead.

Pipeline Splitting At Runtime

A sharded cluster will attempt to execute as many of a pipeline's stages as possible, in parallel, on each shard containing the required data. However, certain types of stages must operate on all the data in one place. Specifically, these are the sorting and grouping stages, collectively referred to as the "blocking stages" (described in the chapter Pipeline Performance Considerations). Upon the first occurrence of a blocking stage in the pipeline, the aggregation engine will split the pipeline into two parts at the point where the blocking stage occurs. The Aggregation Framework refers to the first section of the divided pipeline as the "Shards Part", which can run concurrently on multiple shards. The remaining portion of the split pipeline is called the "Merger Part", which executes in one location. The following illustration shows how this pipeline division occurs.



One of the two stages which causes a split, shown as stage 3, is a `$group` stage. The same behaviour actually occurs with all grouping stages, specifically `$bucket`, `$bucketAuto`, `$count` and `$sortByCount`. Therefore any mention of the `$group` stage in this chapter is synonymous with all of these grouping stages.

You can see two examples of aggregation pipeline splitting in action in the MongoDB Shell screenshots displayed below, showing each pipeline and its explain plan. The cluster contains four shards ("*s0*", "*s1*", "*s2*" and "*s3*") which hold the distributed collection. The two example aggregations perform the following actions respectively:

1. Sharded sort, matching on shard key values and limiting the number of results
2. Sharded group, matching on non-shard key values with `allowDiskUse:true` and showing the total number of records per group

Sharded Sort

```
> pipeline;
[
  {
    '$match': {
      mykey: {
        '$in': [ 0, 1, 3, 7 ]
      }
    }
  },
  {
    '$sort': { favColour: 1 }
  },
  { '$limit': 5 }
]


> explain;
{
  mergeType: 'mongos',
  splitPipeline: {
    shardsPart: [
      {
        '$match': {
          mykey: {
            '$in': [ 0, 1, 3, 7 ]
          }
        }
      },
      {
        '$sort': {
          sortKey: { favColour: 1 },
          limit: 5
        }
      }
    ],
    mergerPart: [
      {
        '$sort': {
          sortKey: { favColour: 1 },
          mergePresorted: true,
          limit: 5
        }
      }
    ]
  },
  shards: { s0: {}, s2: {}, s3: {} }
}
```

Sharded Group (allow-disk)

```
> pipeline;
[
  {
    '$match': {
      randValue: { '$gte': 0.5 }
    }
  },
  {
    '$group': {
      _id: '$favColour',
      total_rcs: { '$sum': 1 }
    }
  }
]


> explain;
{
  mergeType: 'anyShard',
  splitPipeline: {
    shardsPart: [
      {
        '$match': {
          randValue: { '$gte': 0.5 }
        }
      },
      {
        '$group': {
          _id: '$favColour',
          total_rcs: {
            '$sum': { '$const': 1 }
          }
        }
      }
    ],
    mergerPart: [
      {
        '$group': {
          _id: '$$ROOT._id',
          total_rcs: { '$sum': '$$ROOT.total_rcs'},
          '$doingMerge': true
        }
      }
    ]
  },
  shards: { s0: {}, s1: {}, s2: {}, s3: {} }
}
```

You can observe some interesting behaviours from these two explain plans:

- **Shards Part Of Pipeline Running In Parallel**. In both cases, the pipeline's `shardsPart` executes on multiple shards, as indicated in the shards array field at the base of the explain plan. In the first example, the aggregation runtime targets only three shards. However, in the second example, the runtime must broadcast the pipeline's `shardsPart` to run on all shards - the section *Execution Of The Shards Part Of The Split Pipeline* in this chapter discusses why.

- **Optimisations Applied In Shards Part**. For the `$sort` or `$group` blocking stages where the pipeline splits, the blocking stage divides into two. The runtime executes the first phase of the blocking stage as the last stage of the `shardsPart` of the divided pipeline. It then completes the stage's remaining work as the first stage of the `mergerPart`. For a `$sort` stage, this means the cluster conducts a large portion of the sorting work in parallel on all shards, with a remaining "merge sort" occurring at the final location. For a `$group` stage, the cluster performs the grouping in parallel on every shard, accumulating partial sums and totals ahead of its final merge phase. Consequently, the runtime does not have to ship masses of raw ungrouped data from the source shards to where the runtime merges the partially formed groups.

- **Merger Part Running From A Single Location**. The specific location where the runtime executes the pipeline's `mergerPart` stages depends on several variables. The explain plan shows the location chosen by the runtime in the `mergeType` field of its output. In these two examples, the locations are `mongos` and `anyShard`, respectively. This chapter's *Execution Of The Merger Part Of The Split Pipeline* section outlines the rules that the aggregation runtime uses to decide this location.

- **Final Merge Sorting When The Sort Stage Is Split**. The `$sort`'s final phase shown in the `mergerPart` of the first pipeline is not a blocking operation, whereas, with `$group` shown in the second pipeline, `$group`'s final phase is blocking. This chapter's *Difference In Merging Behaviour For Grouping Vs Sorting* section discusses why.

Unfortunately, if you are running your aggregations in MongoDB versions 4.2 to 5.2, the explain plan generated by the aggregation runtime erroneously neglects to log the final phase of the `$sort` stage in the pipeline's `mergerPart`. This is caused by an explain plan bug but rest assured that the final phase of the `$sort` stage (the "merge sort") does indeed happen in the pipeline's `mergerPart`.

Execution Of The Shards Part Of The Split Pipeline

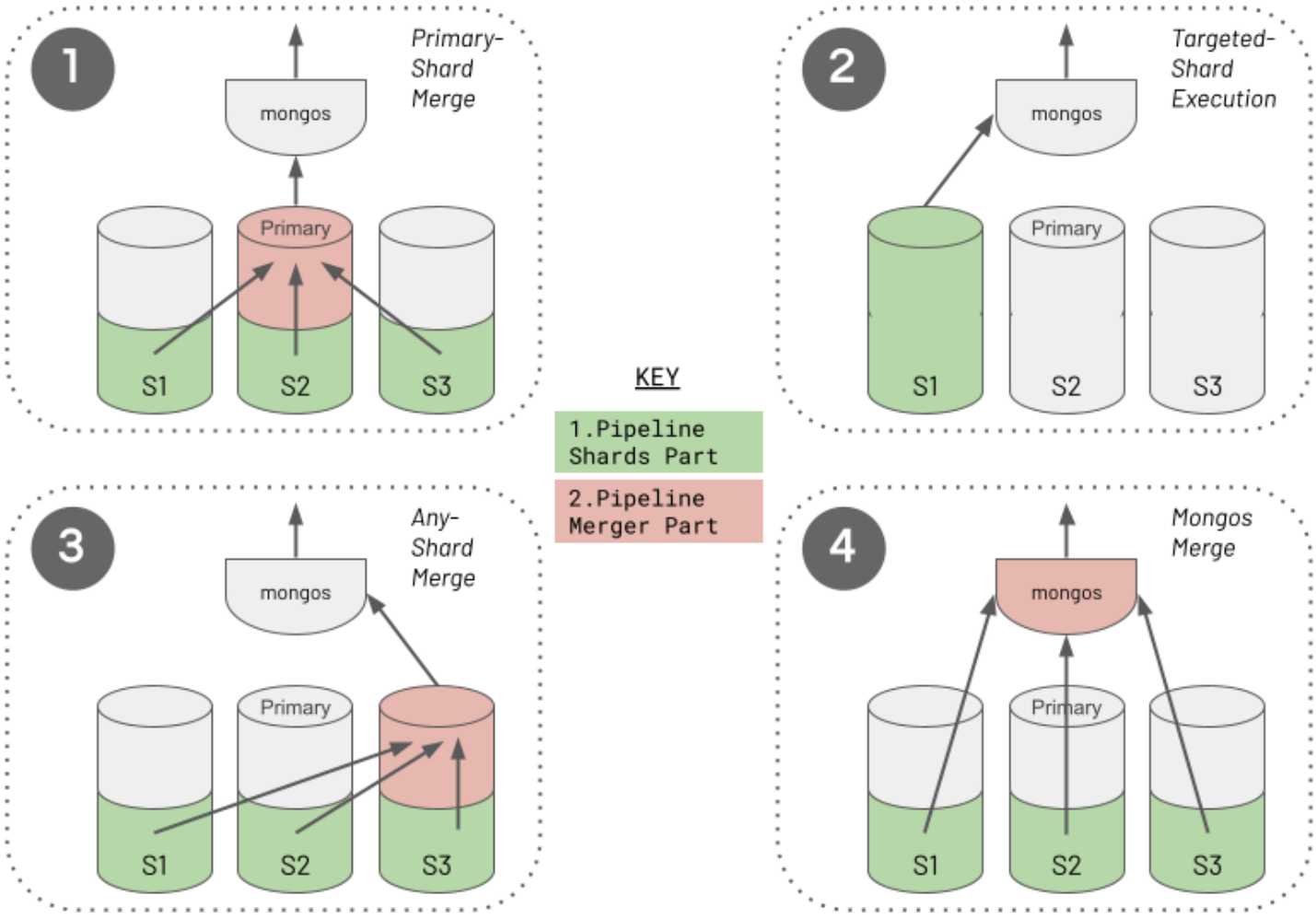
When a mongos receives a request to execute an aggregation pipeline, it needs to determine where to target the *shards part* of the pipeline. It will endeavour to run this on the relevant subset of shards rather than broadcasting the work to all.

Suppose there is a `$match` stage occurring at the start of the pipeline. If the filter for this `$match` includes the shard key or a prefix of the shard key, the mongos can perform a targeted operation. It routes the *shards part* of the split pipeline to execute on the applicable shards only.

Furthermore, suppose the runtime establishes that the `$match`'s filter contains an exact match on a shard key value for the source collection. In that case, the pipeline can target a single shard only, and doesn't even need to split the pipeline into two. The entire pipeline runs in one place, on the one shard where the data it needs lives. Even if the `$match`'s filter only has a partial match on the first part of the shard key (the "prefix"), if this spans a range of documents encapsulated within a single chunk, or multiple chunks on the same shard only, the runtime will just target the single shard.

Execution Of The Merger Part Of The Split Pipeline (If Any)

The aggregation runtime applies a set of rules to determine where to execute the *merger part* of an aggregation pipeline for a sharded cluster and whether a split is even necessary. The following diagram captures the four different approaches the runtime will choose from.



The aggregation runtime selects the *merger part* location (if any) by following a decision tree, with four possible outcomes. The list below outlines the ordered decisions the runtime takes. However, it is crucial to understand that this order does not reflect precedence or preference. Achieving either the *Targeted-Shard Execution* (2) or *Mongos Merge* (4) is usually the preferred outcome for optimum performance.

1. **Primary-Shard Merge**. When the pipeline contains a stage referencing a second unsharded collection, the aggregation runtime will place this stage in the *merger part* of the split pipeline. It executes this *merger part* on the designated primary shard, which holds the referenced unsharded collection. This is always the case for the stages that can only reference unsharded collections (i.e. for `$out` generally or for `$lookup` and `$graphLookup` in MongoDB versions before 5.1). This is also the situation if the collection happens to be unsharded and you reference it from a `$merge` stage or, in MongoDB 5.1 or greater, from a `$lookup` or `$graphLookup` stage.

2. **Targeted-Shard Execution**. As discussed earlier, if the runtime can ensure the pipeline matches the required subset of the source collection data to just one shard, it does not split the pipeline, and there is no *merger part*. Instead, the runtime executes the entire pipeline on the one matched shard, just like it would for non-sharded deployments. This optimisation avoids unnecessarily breaking the pipeline into two parts, where intermediate data then has to move from the *shards part(s)* to the *merger part*. The behaviour of pinning to a single shard occurs even if the pipeline contains a `$merge`, `$lookup` or `$graphLookup` stage referencing a second sharded collection containing records dispersed across multiple shards.

3. **Any-Shard Merge**. Suppose you've configured `allowDiskUse:true` for the aggregation to avoid the 100 MB memory consumption limit per stage. If one of the following two situations is also true, the aggregation runtime must run the *merger part* of the split pipeline on a randomly chosen shard (a.k.a. "any shard"):

   - The pipeline contains a grouping stage (which is where the split occurs), *or*
   - The pipeline contains a `$sort` stage (which is where the split occurs), and a subsequent blocking stage (a grouping or `$sort` stage) occurs later.

   For these cases, the runtime picks a shard to execute the merger, rather than merging on the mongos, to maximise the likelihood that the host machine has enough storage space to spill to disk. Invariably, each shard's host machine will have greater storage capacity than the host machine of a mongos. Consequently, the runtime must take this caution because, with `allowDiskUse:true`, you are indicating the likelihood that your pipeline will cause memory capacity pressure. Notably, the aggregation runtime does not need to mitigate the same risk by merging on a shard for the other type of blocking stage (a `$sort`) when `$sort` is the only blocking stage in the pipeline. You can read why a single `$sort` stage can be treated differently and does not need the same host storage capacity for merging in this chapter's *Difference In Merging Behaviour For Grouping Vs Sorting* section.

4. **Mongos Merge**. This is the default approach and location. The aggregation runtime will perform the *merger part* of the split pipeline on the mongos that instigated the aggregation in all the remaining situations. If the pipeline's *merger part* only contains

streaming stages (described in the chapter Pipeline Performance Considerations), the runtime assumes it is safe for the mongos to run the remaining pipeline. A mongos has no concept of local storage to hold data. However, it doesn't matter in this situation because the runtime won't need to write to disk as RAM pressure will be minimal. The category of streaming tasks that supports a Mongos Merge also includes the final phase of a split `$sort` stage, which processes data in a streaming fashion without needing to block to see all the data together. Additionally, suppose you have defined `allowDiskUse:false` (the default). In that case, you are signalling that even if the pipeline has a `$group` stage (or a `$sort` stage followed by another blocking stage), these blocking activities will not need to overspill to disk. Performing the final merge on the mongos is the default because fewer network data transfer hops are required to fulfil the aggregation, thus reducing latency compared with merging on "any shard".

Regardless of where the *merger part* runs, the mongos is always responsible for streaming the aggregation's final batches of results back to the client application.

It is worth considering when no blocking stages exist in a pipeline. In this case, the runtime executes the entire pipeline in parallel on the relevant shards and the runtime streams each shard's output directly to the mongos. You can regard this as just another variation of the default behaviour (*4 - Mongos Merge*). All the stages in the aggregation constitute just the *shards part* of the pipeline, and the mongos "stream merges" the final data through to the client.

Difference In Merging Behaviour For Grouping Vs Sorting

You will have read in the Pipeline Performance Considerations chapter about `$sort` and `$group` stages being blocking stages and potentially consuming copious RAM. Consequently, you may be confused by the statement that, unlike a `$group` stage, when the pipeline splits, the aggregation runtime will finalise a `$sort` stage on a mongos even if you specify `allowDiskUse:true`. This is because the final phase of a split `$sort` stage is not a blocking activity, whereas the final phase of a split `$group` stage is. For `$group`, the pipeline's *merger part* must wait for all the data coming out of all the targeted shards. For `$sort`, the runtime executes a streaming merge sort operation, only waiting for the next batch of records coming out of each shard. As long as it can see the first of the sorted documents in the next batch from every shard, it knows which documents it can immediately process and stream on to the rest of the pipeline. It doesn't have to block waiting to see all of the records to guarantee correct ordering.

This optimisation doesn't mean that MongoDB has magically found a way to avoid a `$sort` stage being a blocking stage in a sharded cluster. It hasn't. The first phase of the `$sort` stage, run on each shard in parallel, is still blocking, waiting to see all the matched input data for that

shard. However, the final phase of the same `$sort` stage, executed at the merge location, does not need to block.

Summarising Sharded Pipeline Execution Approaches

In summary, the aggregation runtime seeks to execute a pipeline on the subset of shards containing the required data only. If the runtime must split the pipeline to perform grouping or sorting, it completes the final merge work on a mongos, when possible. Merging on a mongos helps to reduce the number of required network hops and the execution time.

Performance Tips For Sharded Aggregations

All the recommended aggregation optimisations outlined in the Pipeline Performance Considerations chapter equally apply to a sharded cluster. In fact, in most cases, these same recommendations, repeated below, become even more critical when executing aggregations on sharded clusters:

1. **Sorting - Use Index Sort**. When the runtime has to split on a `$sort` stage, the *shards part* of the split pipeline running on each shard in parallel will avoid an expensive in-memory sort operation.

2. **Sorting - Use Limit With Sort**. The runtime has to transfer fewer intermediate records over the network, from each shard performing the *shards part* of a split pipeline to the location that executes the pipeline's *merger part*.

3. **Sorting - Reduce Records To Sort**. If you cannot adopt point 1 or 2, moving a `$sort` stage to as late as possible in a pipeline will typically benefit performance in a sharded cluster. Wherever the `$sort` stage appears in a pipeline, the aggregation runtime will split the pipeline at this point (unless preceded by a `$group` stage which would cause the split earlier). By promoting other activities to occur in the pipeline first, the hope is these reduce the number of records entering the blocking `$sort` stage. This sorting operation, executing in parallel at the end of *shards part* of the split pipeline, will exhibit less memory pressure. The runtime will also stream fewer records over the network to the split pipeline's *merger part* location.

4. **Grouping - Avoid Unnecessary Grouping**. Using array operators where possible instead of `$unwind` and `$group` stages will mean that the runtime does not need to split the pipeline due to an unnecessarily introduced `$group` stage. Consequently, the aggregation can efficiently process and stream data directly to the mongos rather than flowing through an intermediary shard first.

5. **Grouping - Group Summary Data Only**. The runtime has to move fewer computed records over the network from each shard performing the *shards part* of a split pipeline to the *merger part*'s location.

6. **Encourage Match Filters To Appear Early In The Pipeline**. By filtering out a large subset of records on each shard when performing the *shards part* of the split pipeline, the runtime needs to stream fewer records to the *merger part* location.

Specifically for sharded clusters, there are two further performance optimisations you should aspire to achieve:

1. **Look For Opportunities To Target Aggregations To One Shard Only**. If possible, include a `$match` stage with a filter on a shard key value (or shard key prefix value).

2. **Look For Opportunities For A Split Pipeline To Merge On A Mongos**. If the pipeline has a `$group` stage (or a `$sort` stage followed by a `$group` / `$sort` stage) which causes the pipeline to divide, avoid specifying `allowDiskUse:true` if possible. This reduces the amount of intermediate data transferred over the network, thus reducing latency.

Advanced Use Of Expressions For Array Processing

One of the most compelling aspects of MongoDB is the ability to embed arrays within documents. Unlike relational databases, this characteristic typically allows each entity's entire data structure to exist in one place as a document. Documents better represent "real-world" objects and how developers think about such entities. When writing code to interact with the stored data, this intuitive data representation reduces the cognitive load on developers, enabling them to deliver new application capabilities quicker.

The Aggregation Framework provides a rich set of aggregation operator expressions for analysing and manipulating arrays. When optimising for performance, these array expressions are critical to avoid unwinding and regrouping documents where you only need to process each document's array in isolation. For most situations when you need to manipulate an array, there is usually a single array operator expression that you can turn to solve your requirement.

Occasionally, you may still need to assemble a composite of multiple lower-level expressions to handle a challenging array manipulation task. These situations are the most difficult aspect for anyone using the Aggregation Framework. As a result, this chapter endeavours to bootstrap the knowledge you will require to fulfil such undertakings. Like aggregation pipelines in general, a large part of the challenge relates to adapting your mindset to a Functional programming paradigm rather than a Procedural one. As this book discusses in its introductory

chapter, the functional aspect of aggregations is essential for the database's aggregation engine to process data at scale efficiently.

Comparing with procedural approaches can help to bring clarity when describing array manipulation pipeline logic. Therefore, the first few explanations in this chapter include examples of equivalent JavaScript code snippets you would use to achieve comparable outcomes in regular client-side applications.

Lastly, if you haven't read this book's Expressions Explained chapter yet, you should do so before continuing with this chapter.

"If-Else" Conditional Comparison

Even though performing conditional comparisons is more of a general principle than specific to array manipulation, it is first worth touching upon it to introduce the topic of advanced expressions. Consider the trivialised scenario of a retailer wanting to calculate the total cost of a customer's shopping order. The customer might order multiple of the same product, and the vendor applies a discount if more than 5 of the product items are in the order.

In a procedural style of JavaScript, you might write the following code to calculate the total order cost:

```
let order = {"product" : "WizzyWidget", "price": 25.99, "qty": 8};

// Procedural style JavaScript
if (order.qty > 5) {
  order.cost = order.price * order.qty * 0.9;
} else {
  order.cost = order.price * order.qty;
}
```

This code modifies the customer's order to the following, to include the total cost:

```
{product: 'WizzyWidget', qty: 8, price: 25.99, cost: 187.128}
```

To achieve a similar outcome in an aggregation pipeline, you might use the following:

```
db.customer_orders.insertOne(order);

var pipeline = [
  {"$set": {
    "cost": {
      "$cond": {
        "if":   {"$gte": ["$qty", 5 ]},
        "then": {"$multiply": ["$price", "$qty", 0.9]},
        "else": {"$multiply": ["$price", "$qty"]},
      }
    },
  }},
];

db.customer_orders.aggregate(pipeline);
```

This pipeline produces the following output with the customer order document transformed to:

```
{product: 'WizzyWidget', price: 25.99, qty: 8, cost: 187.128}
```

If you were going to use a functional programming approach in JavaScript, the code would be more like the following to achieve the same outcome:

```
// Functional style JavaScript
order.cost = (
              (order.qty > 5) ?
              (order.price * order.qty * 0.9) :
              (order.price * order.qty)
             );
```

Here, you can see that the JavaScript code's construction in a functional style more closely resembles the aggregation pipeline's structure. This comparison highlights why some people may find composing aggregation expressions foreboding. The challenge is predominantly due to the less familiar paradigm of functional programming rather than the intricacies of MongoDB's aggregation language per se.

The other difference in this comparison and the rest of the comparisons in this chapter is the pipeline will work unchanged when run against a collection of many records, which could feasibly be many billions. The sample JavaScript code only works against one document at a time and would need to be modified to loop through a list of records. This JavaScript code would need to fetch each document from the database back to a client, apply the modifications and then write the result back to the database. Instead, the aggregation pipeline's logic operates against each document in-situ within the database for far superior performance and efficiency.

The "Power" Array Operators

When you want to transform or extract data from an array field, and a single high-level array operator (e.g. `$avg`, `$max`, `$filter`) does not give you what you need, the tools to turn to are the `$map` and `$reduce` array operators. These two "power" operators enable you to iterate through an array, perform whatever complexity of logic you need against each array element and collect together the result for inclusion in a stage's output.

The `$map` and `$reduce` operators are the "swiss army knives" of the Aggregation Framework. Do not confuse these two array operators with MongoDB's old Map-Reduce API, which was essentially made redundant and obsolete by the emergence of the superior Aggregation Framework in MongoDB. In the old Map-Reduce API, you combine a `map()` function and a `reduce()` function to generate a result. In the Aggregation Framework, the `$map` and `$reduce` operators are independent of each other. Depending on your specific requirements, you would use one or the other to process an array's field, but not both together. Here's an explanation of these two "power" operators:

- `$map`. Allows you to specify some logic to perform against each element in the array that the operator iterates, returning an array as the final result. Typically you use `$map` to mutate each array member and then return this transformed array. The `$map` operator exposes the current array element's content to your logic via a special variable, with the default name of `$$this`.
- `$reduce`. Similarly, you can specify some logic to execute for each element in an array that the operator iterates but instead returning a single value (rather than an array) as the final result. You typically use `$reduce` to compute a summary having analysed each array element. For example, you might want to return a number by multiplying together a specific field value from each array's element. Like the `$map` operator, the `$reduce` operator provides your logic with access to the current array element via the variable `$$this`. The operator also provides a second variable, called `$$value`, for your logic to update when accumulating the single result (e.g. the multiplication result).

The rest of this chapter explores how these two "power" operators are used to manipulate arrays.

"For-Each" Looping To Transform An Array

Imagine you wanted to process a list of the products ordered by a customer and convert the array of product names to upper case. In a procedural style of JavaScript, you might write the following code to loop through each product in the array and convert its name to upper case:

```
let order = {
  "orderId": "AB12345",
  "products": ["Laptop", "Kettle", "Phone", "Microwave"]
};

// Procedural style JavaScript
for (let pos in order.products) {
  order.products[pos] = order.products[pos].toUpperCase();
}
```

This code modifies the order's product names to the following, with the product names now in uppercase:

```
{orderId: 'AB12345', products: ['LAPTOP', 'KETTLE', 'PHONE', 'MICROWAVE']}
```

To achieve a similar outcome in an aggregation pipeline, you might use the following:

```
db.orders.insertOne(order);

var pipeline = [
  {"$set": {
    "products": {
      "$map": {
        "input": "$products",
        "as": "product",
        "in": {"$toUpper": "$$product"}
      }
    }
  }}
];

db.orders.aggregate(pipeline);
```

Here, a `$map` operator expression is applied to loop through each product name in the input products array and add the upper case version of the product name to the replacement output array.

This pipeline produces the following output with the order document transformed to:

```
{orderId: 'AB12345', products: ['LAPTOP', 'KETTLE', 'PHONE', 'MICROWAVE']}
```

Using functional style in JavaScript, your looping code would more closely resemble the following to achieve the same outcome:

```
// Functional style JavaScript
order.products = order.products.map(
  product => {
    return product.toUpperCase();
  }
);
```

Comparing an aggregation `$map` operator expression to a JavaScript `map()` array function is far more illuminating to help explain how the operator works.

"For-Each" Looping To Compute A Summary Value From An Array

Suppose you wanted to process a list of the products ordered by a customer but produce a single summary string field from this array by concatenating all the product names from the array. In a procedural JavaScript style, you could code the following to produce the product names summary field:

```
let order = {
  "orderId": "AB12345",
  "products": ["Laptop", "Kettle", "Phone", "Microwave"]
};

order.productList = "";

// Procedural style JavaScript
for (const pos in order.products) {
  order.productList += order.products[pos] + "; ";
}
```

This code yields the following output with a new `productList` string field produced, which contains the names of all the products in the order, delimited by semi-colons:

```
{
  orderId: 'AB12345',
  products: [ 'Laptop', 'Kettle', 'Phone', 'Microwave' ],
  productList: 'Laptop; Kettle; Phone; Microwave; '
}
```

You can use the following pipeline to achieve a similar outcome:

```
db.orders.insertOne(order);

var pipeline = [
  {"$set": {
    "productList": {
      "$reduce": {
        "input": "$products",
        "initialValue": "",
        "in": {
          "$concat": ["$$value", "$$this", "; "]
        }
      }
    }
  }}
];

db.orders.aggregate(pipeline);
```

Here, a `$reduce` operator expression loops through each product in the input array and concatenates each product's name into an accumulating string. You use the `$$this` expression to access the current array element's value during each iteration. For each iteration, you employ the `$$value` expression to reference the final output value, to which you append the current product string (+ delimiter).

This pipeline produces the following output where it transforms the order document to:

```
{
  orderId: 'AB12345',
  products: [ 'Laptop', 'Kettle', 'Phone', 'Microwave' ],
  productList: 'Laptop; Kettle; Phone; Microwave; '
}
```

Using a functional approach in JavaScript, you could have used the following code to achieve the same result:

```
// Functional style JavaScript
order.productList = order.products.reduce(
  (previousValue, currentValue) => {
    return previousValue + currentValue + "; ";
  },
  ""
);
```

Once more, by comparing the use of the aggregation operator expression (`$reduce`) to the equivalent JavaScript array function (`reduce()`), the similarity is more pronounced.

"For-Each" Looping To Locate An Array Element

Imagine storing data about buildings on a campus where each building document contains an array of rooms with their sizes (width and length). A room reservation system may require finding the first room in the building with sufficient floor space for a particular number of meeting attendees. Below is an example of one building's data you might load into the database, with its array of rooms and their dimensions in metres:

```
db.buildings.insertOne({
  "building": "WestAnnex-1",
  "room_sizes": [
    {"width": 9, "length": 5},
    {"width": 8, "length": 7},
    {"width": 7, "length": 9},
    {"width": 9, "length": 8},
  ]
});
```

You want to create a pipeline to locate an appropriate meeting room that produces an output similar to the following. The result should contain a newly added field, `firstLargeEnoughRoomArrayIndex`, to indicate the array position of the first room found to have enough capacity.

```
{
  building: 'WestAnnex-1',
  room_sizes: [
    { width: 9, length: 5 },
    { width: 8, length: 7 },
    { width: 7, length: 9 },
    { width: 9, length: 8 }
  ],
  firstLargeEnoughRoomArrayIndex: 2
}
```

Below is a suitable pipeline that iterates through the room array elements capturing the position of the first one with a calculated area greater than 60m²:

```
var pipeline = [
  {"$set": {
    "firstLargeEnoughRoomArrayIndex": {
      "$reduce": {
        "input": {"$range": [0, {"$size": "$room_sizes"}]},
        "initialValue": -1,
        "in": {
          "$cond": {
            "if": {
              "$and": [
                // IF ALREADY FOUND DON'T CONSIDER SUBSEQUENT ELEMENTS
                {"$lt": ["$$value", 0]},
                // IF WIDTH x LENGTH > 60
                {"$gt": [
                  {"$multiply": [
                    {"$getField": {"input": {"$arrayElemAt": ["$room_sizes",
"$$this"]}, "field": "width"}},
                    {"$getField": {"input": {"$arrayElemAt": ["$room_sizes",
"$$this"]}, "field": "length"}},
                  ]},
                  60
                ]}
              ]
            },
            // IF ROOM SIZE IS BIG ENOUGH CAPTURE ITS ARRAY POSITION
            "then": "$$this",
            // IF ROOM SIZE NOT BIG ENOUGH RETAIN EXISTING VALUE (-1)
            "else": "$$value"
          }
        }
      }
    }
  }}
];

db.buildings.aggregate(pipeline);
```

Here the `$reduce` operator is again used to loop and eventually return a single value. However, the pipeline uses a generated sequence of incrementing numbers for its input rather than the existing array field in each source document. The `$range` operator is used to create this sequence which has the same size as the rooms array field of each document. The pipeline uses this approach to track the array position of the matching room using the `$$this` variable. For each iteration, the pipeline calculates the array room element's area. If the size is greater than 60, the pipeline assigns the current array position (represented by `$$this`) to the final result (represented by `$$value`).

The "iterator" array expressions have no concept of a *break* command that procedural programming languages typically provide. Therefore, even though the executing logic may have already located a room of sufficient size, the looping process will continue through the remaining array elements. Consequently, the pipeline logic must include a check during each

iteration to avoid overriding the final value (the `$$value` variable) if it already has a value. Naturally, for massive arrays containing a few hundred or more elements, an aggregation pipeline will incur a noticeable latency impact when iterating the remaining array members even though the logic has already identified the required element.

Reproducing *$map* Behaviour Using *$reduce*

It is possible to implement the `$map` behaviour using `$reduce` to transform an array. This method is more complex, but you may need to use it in some rare circumstances. Before looking at an example of why let's first compare a more basic example of using `$map` and then `$reduce` to achieve the same thing.

Suppose you have captured some sensor readings for a device:

```
db.deviceReadings.insertOne({
  "device": "A1",
  "readings": [27, 282, 38, 22, 187]
});
```

Imagine you want to produce a transformed version of the `readings` array, with the device's ID concatenated with each reading in the array. You want the pipeline to produce an output similar to the following, with the newly included array field:

```
{
  device: 'A1',
  readings: [ 27, 282, 38, 22, 187 ],
  deviceReadings: [ 'A1:27', 'A1:282', 'A1:38', 'A1:22', 'A1:187' ]
}
```

You can achieve this using the `$map` operator expression in the following pipeline:

```
var pipeline = [
  {"$set": {
    "deviceReadings": {
      "$map": {
        "input": "$readings",
        "as": "reading",
        "in": {
          "$concat": ["$device", ":", {"$toString": "$$reading"}]
        }
      }
    }
  }}
];

db.deviceReadings.aggregate(pipeline);
```

You can also accomplish the same with the `$reduce` operator expression in the following pipeline:

```
var pipeline = [
  {"$set": {
    "deviceReadings": {
      "$reduce": {
        "input": "$readings",
        "initialValue": [],
        "in": {
          "$concatArrays": [
            "$$value",
            [{"$concat": ["$device", ":", {"$toString": "$$this"}]}]
          ]
        }
      }
    }
  }}
];

db.deviceReadings.aggregate(pipeline);
```

You will see the pipeline has to do more work here, holding the transformed element in a new array and then concatenating this with the "final value" array the logic is accumulating in the `$$value` variable.

So why would you ever want to use `$reduce` for this requirement and take on this extra complexity? Well, suppose the aggregation output containing the transformed array needed to include the element's array position number in the concatenated string value to yield a result similar to the following:

```
{
  device: 'A1',
  readings: [ 27, 282, 38, 22, 187 ],
  deviceReadings: [ 'A1-0:27', 'A1-1:282', 'A1-2:38', 'A1-3:22', 'A1-4:187' ]
}
```

You cannot achieve this using `$map` because the logic in each iteration does not know the current loop count or input array position. However, you can solve this by using `$reduce` if adopting the pattern where the input is a `$range` generated sequence of incrementing numbers (as covered earlier in this chapter). Consequently, the string concatenating code can include the array position in each transformed element's value (accessed via the `$$this` variable), as shown in the pipeline below:

```
var pipeline = [
  {"$set": {
    "deviceReadings": {
      "$reduce": {
        "input": {"$range": [0, {"$size": "$readings"}]},
        "initialValue": [],
        "in": {
          "$concatArrays": [
            "$$value",
            [{"$concat": [
              "$device",
              "-",
              {"$toString": "$$this"},
              ":",
              {"$toString": {"$arrayElemAt": ["$readings", "$$this"]}},
            ]}]
          ]
        }
      }
    }
  }}
];

db.deviceReadings.aggregate(pipeline);
```

Adding New Fields To Existing Objects In An Array

One of the primary uses of the `$map` operator expression is to add more data to each existing object in an array. Suppose you've persisted a set of retail orders, where each order document contains an array of order items. Each order item in the array captures the product's name, unit price, and quantity purchased, as shown in the example below:

```
db.orders.insertOne({
    "custid": "jdoe@acme.com",
    "items": [
      {
        "product" : "WizzyWidget",
        "unitPrice": 25.99,
        "qty": 8,
      },
      {
        "product" : "HighEndGizmo",
        "unitPrice": 33.24,
        "qty": 3,
      }
    ]
});
```

You now need to calculate the total cost for each product item (`quantity` x `unitPrice`) and add that cost to the corresponding order item in the array. You can use a pipeline similar to the following to achieve this:

```
var pipeline = [
  {"$set": {
    "items": {
      "$map": {
        "input": "$items",
        "as": "item",
        "in": {
          "product": "$$item.product",
          "unitPrice": "$$item.unitPrice",
          "qty": "$$item.qty",
          "cost": {"$multiply": ["$$item.unitPrice", "$$item.qty"]}},
        }
      }
    }
  }
];

db.orders.aggregate(pipeline);
```

Here, for each element in the source array, the pipeline creates an element in the new array by explicitly pulling in the three fields from the old element (`product` , `unitPrice` and `quantity`) and adding one new computed field (`cost`). The pipeline produces the following output:

```
{
  custid: 'jdoe@acme.com',
  items: [
    {
      product: 'WizzyWidget',
      unitPrice: 25.99,
      qty: 8,
      cost: 187.128
    },
    {
      product: 'HighEndGizmo',
      unitPrice: 33.24,
      qty: 3,
      cost: 99.72
    }
  ]
}
```

Similar to the disadvantages of using a `$project` stage in a pipeline, outlined in an earlier chapter, the `$map` code is burdened by explicitly naming every field in the array element to retain. You will find this tiresome if each array element has lots of fields. In addition, if your data model evolves and new types of fields appear in the array's items over time, you will be forced to return to your pipeline and refactor it each time to include these newly introduced fields.

Just like using `$set` instead of `$project` for a pipeline stage, there is a better solution to allow you to retain all existing array item fields and add new ones when you process arrays. A good solution is to employ the `$mergeObjects` operator expression to combine all existing fields plus the newly computed fields into each new array element. `$mergeObjects` takes an array of objects and combines the fields from all the array's objects into one single object. To use `$mergeObjects` in this situation, you provide the current array element as the first parameter to `$mergeObjects`. The second parameter you provide is a new object containing each computed field. In the example below, the code adds only one generated field, but if you require it, you can include multiple generated fields in this new object:

```
var pipeline = [
  {"$set": {
    "items": {
      "$map": {
        "input": "$items",
        "as": "item",
        "in": {
          "$mergeObjects": [
            "$$item",
            {"cost": {"$multiply": ["$$item.unitPrice", "$$item.qty"]}},
          ]
        }
      }
    }
  }}
];

db.orders.aggregate(pipeline);
```

This pipeline produces the same output as the previous "hardcoded field names" pipeline, but with the advantage of being sympathetic to new types of fields appearing in the source array in the future.

Instead of using `$mergeObjects`, there is an alternative and slightly more verbose combination of three different array operator expressions that you can similarly employ to retain all existing array item fields and add new ones. These three operators are:

- `$objectToArray`. This converts an object containing different field key/value pairs into an array of objects where each object has two fields: `k`, holding the field's name, and `v`, holding the field's value. For example: `{height: 170, weight: 60}` becomes `[{k: 'height', v: 170}, {k: 'weight', v: 60}]`
- `$concatArrays`. This combines the contents of multiple arrays into one single array result.
- `$arrayToObject`. This converts an array into an object by performing the reverse of the `$objectToArray` operator. For example: `{k: 'height', v: 170}, {k: 'weight', v: 60}, {k: 'shoeSize', v: 10}]` becomes `{height: 170, weight: 60, shoeSize: 10}`

The pipeline below shows the combination in action for the same retail orders data set as before, adding the newly computed total cost for each product:

```
var pipeline = [
  {"$set": {
    "items": {
      "$map": {
        "input": "$items",
        "as": "item",
        "in": {
          "$arrayToObject": {
            "$concatArrays": [
              {"$objectToArray": "$$item"},
              [{
                "k": "cost",
                "v": {"$multiply": ["$$item.unitPrice", "$$item.qty"]},
              }]
            ]
          }
        }
      }
    }}
  }
];

db.orders.aggregate(pipeline);
```

If this achieves the same as using `$mergeObjects` but is more verbose, why bother using this pattern? Well, in most cases, you wouldn't. One situation where you would use the more verbose combination is if you need to dynamically set the name of an array item's field, in addition to its value. Rather than naming the computed total field as `cost`, suppose you want the field's name also to reflect the product's name (e.g. `costForWizzyWidget`, `costForHighEndGizmo`). You can achieve this by using the `$arrayToObject` / `$concatArrays` / `$objectToArray` approach rather than the `$mergeObjects` method, as follows:

```
var pipeline = [
  {"$set": {
    "items": {
      "$map": {
        "input": "$items",
        "as": "item",
        "in": {
          "$arrayToObject": {
            "$concatArrays": [
              {"$objectToArray": "$$item"},
              [{
                "k": {"$concat": ["costFor", "$$item.product"]},
                "v": {"$multiply": ["$$item.unitPrice", "$$item.qty"]},
              }]
            ]
          }
        }
      }
    }}
  }
];

db.orders.aggregate(pipeline);
```

Below you can see the new pipeline's output. The pipeline has retained all existing array item's fields and added a new field to each item with a dynamically generated name.

```
{
  custid: 'jdoe@acme.com',
  items: [
    {
      product: 'WizzyWidget',
      unitPrice: 25.99,
      qty: 8,
      costForWizzyWidget: 207.92
    },
    {
      product: 'HighEndGizmo',
      unitPrice: 33.24,
      qty: 3,
      costForHighEndGizmo: 99.72
    }
  ]
}
```

When retaining existing items from an array, plus adding new fields, you can use either approach to override an existing item's field with a new value. For example, you may want to modify the current `unitPrice` field to incorporate a discount. For both `$mergeObjects` and `$arrayToObject` expressions, to achieve this, you provide a re-definition of the field as a subsequent parameter after first providing the reference to the source array item. This tactic

works because the last definition wins if the same field is defined more than once with different values.

Rudimentary Schema Reflection Using Arrays

As a final "fun" example, let's see how to employ an `$objectToArray` operator expression to use reflection to analyse the shape of a collection of documents as part of a custom schema analysis tool. Such reflection capabilities are vital in databases that provide a flexible data model, such as MongoDB, where the included fields may vary from document to document.

Imagine you have a collection of customer documents, similar to the following:

```
db.customers.insertMany([
  {
    "_id": ObjectId('6064381b7aa89666258201fd'),
    "email": 'elsie_smith@myemail.com',
    "dateOfBirth": ISODate('1991-05-30T08:35:52.000Z'),
    "accNnumber": 123456,
    "balance": NumberDecimal("9.99"),
    "address": {
      "firstLine": "1 High Street",
      "city": "Newtown",
      "postcode": "NW1 1AB",
    },
    "telNums": ["07664883721", "01027483028"],
    "optedOutOfMarketing": true,
  },
  {
    "_id": ObjectId('734947394bb73732923293ed'),
    "email": 'jon.jones@coolemail.com',
    "dateOfBirth": ISODate('1993-07-11T22:01:47.000Z'),
    "accNnumber": 567890,
    "balance": NumberDecimal("299.22"),
    "telNums": "07836226281",
    "contactPrefernece": "email",
  },
]);
```

In your schema analysis pipeline, you use `$objectToArray` to capture the name and type of each top-level field in the document as follows:

```
var pipeline = [
  {"$project": {
    "_id": 0,
    "schema": {
      "$map": {
        "input": {"$objectToArray": "$$ROOT"},
        "as": "field",
        "in": {
          "fieldname": "$$field.k",
          "type": {"$type": "$$field.v"},
        }
      }
    }
  }}
];

db.customers.aggregate(pipeline);
```

For the two example documents in the collection, the pipeline outputs the following:

```
{
  schema: [
    {fieldname: '_id', type: 'objectId'},
    {fieldname: 'email', type: 'string'},
    {fieldname: 'dateOfBirth', type: 'date'},
    {fieldname: 'accNnumber', type: 'int'},
    {fieldname: 'balance', type: 'decimal'},
    {fieldname: 'address', type: 'object'},
    {fieldname: 'telNums', type: 'array'},
    {fieldname: 'optedOutOfMarketing', type: 'bool'}
  ]
},
{
  schema: [
    {fieldname: '_id', type: 'objectId'},
    {fieldname: 'email', type: 'string'},
    {fieldname: 'dateOfBirth', type: 'date'},
    {fieldname: 'accNnumber', type: 'int'},
    {fieldname: 'balance', type: 'decimal'},
    {fieldname: 'telNums', type: 'string'},
    {fieldname: 'contactPrefernece', type: 'string'}
}
```

The difficulty with this basic pipeline approach is once there are many documents in the collection, the output will be too lengthy and complex for you to detect common schema patterns. Instead, you will want to add an `$unwind` and `$group` stage combination to accumulate recurring fields that match. The generated result should also highlight if the same field name appears in multiple documents but with different data types. Here is the improved pipeline:

```
var pipeline = [
  {"$project": {
    "_id": 0,
    "schema": {
      "$map": {
        "input": {"$objectToArray": "$$ROOT"},
        "as": "field",
        "in": {
          "fieldname": "$$field.k",
          "type": {"$type": "$$field.v"},
        }
      }
    }
  }},

  {"$unwind": "$schema"},

  {"$group": {
    "_id": "$schema.fieldname",
    "types": {"$addToSet": "$schema.type"},
  }},

  {"$set": {
    "fieldname": "$_id",
    "_id": "$$REMOVE",
  }},
];

db.customers.aggregate(pipeline);
```

This pipeline's output now provides a far more comprehensible summary, as shown below:

```
{fieldname: '_id', types: ['objectId']},
{fieldname: 'address', types: ['object']},
{fieldname: 'email', types: ['string']},
{fieldname: 'telNums', types: ['string', 'array']},
{fieldname: 'contactPrefernece', types: ['string']},
{fieldname: 'accNnumber', types: ['int']},
{fieldname: 'balance', types: ['decimal']},
{fieldname: 'dateOfBirth', types: ['date']},
{fieldname: 'optedOutOfMarketing', types: ['bool']}
```

This result highlights that the `telNums` field can have one of two different data types within documents.

The main drawback of this rudimentary schema analysis pipeline is its inability to descend through layers of arrays and sub-documents hanging off each top-level document. This challenge is indeed solvable using a pure aggregation pipeline, but the code involved is far more complex and beyond the scope of this chapter. If you are interested in exploring this further, you can customise the "graphDescend" GitHub example project. That project shows

you how to traverse through hierarchically structured documents using a single aggregation pipeline.

Further Array Manipulation Examples

This book's Array Manipulation Examples section contains more examples of using expressions to process arrays.

Aggregations By Example

The following set of chapters provide examples to solve common data manipulation challenges grouped by different data processing requirements. The best way to use these examples is to try them out yourself as you read each example (see Getting Started for advice on how to execute these examples).

Foundational Examples

This section provides examples for common data manipulation patterns used in many aggregation pipelines, which are relatively straightforward to understand and adapt.

Filtered Top Subset

Minimum MongoDB Version: 4.2

Scenario

You want to query a collection of people to find the three youngest people who have a job in engineering, sorted by the youngest person first.

This example is the only one in the book that you can also achieve entirely using MQL and serves as a helpful comparison between MQL and Aggregation Pipelines.

Sample Data Population

Drop any old version of the database (if it exists) and then populate a new `persons` collection with 5 person documents:

```
use book-filtered-top-subset;
db.dropDatabase();

// Create an index for a persons collection
db.persons.createIndex({"vocation": 1, "dateofbirth": 1});

// Insert records into the persons collection
db.persons.insertMany([
  {
    "person_id": "6392529400",
    "firstname": "Elise",
    "lastname": "Smith",
    "dateofbirth": ISODate("1972-01-13T09:32:07Z"),
    "vocation": "ENGINEER",
    "address": {
        "number": 5625,
        "street": "Tipa Circle",
        "city": "Wojzinmoj",
    },
  },
  {
    "person_id": "1723338115",
    "firstname": "Olive",
    "lastname": "Ranieri",
    "dateofbirth": ISODate("1985-05-12T23:14:30Z"),
    "gender": "FEMALE",
    "vocation": "ENGINEER",
    "address": {
        "number": 9303,
        "street": "Mele Circle",
        "city": "Tobihbo",
    },
  },
  {
    "person_id": "8732762874",
    "firstname": "Toni",
    "lastname": "Jones",
    "dateofbirth": ISODate("1991-11-23T16:53:56Z"),
    "vocation": "POLITICIAN",
    "address": {
        "number": 1,
        "street": "High Street",
        "city": "Upper Abbeywoodington",
    },
  },
  {
    "person_id": "7363629563",
    "firstname": "Bert",
    "lastname": "Gooding",
    "dateofbirth": ISODate("1941-04-07T22:11:52Z"),
    "vocation": "FLORIST",
    "address": {
        "number": 13,
        "street": "Upper Bold Road",
```

```
        "city": "Redringtonville",
    },
  },
  {
    "person_id": "1029648329",
    "firstname": "Sophie",
    "lastname": "Celements",
    "dateofbirth": ISODate("1959-07-06T17:35:45Z"),
    "vocation": "ENGINEER",
    "address": {
        "number": 5,
        "street": "Innings Close",
        "city": "Basilbridge",
    },
  },
  {
    "person_id": "7363626383",
    "firstname": "Carl",
    "lastname": "Simmons",
    "dateofbirth": ISODate("1998-12-26T13:13:55Z"),
    "vocation": "ENGINEER",
    "address": {
        "number": 187,
        "street": "Hillside Road",
        "city": "Kenningford",
    },
  },
]);
```

Aggregation Pipeline

Define a single pipeline ready to perform the aggregation:

```
var pipeline = [
  // Match engineers only
  {"$match": {
    "vocation": "ENGINEER",
  }},

  // Sort by youngest person first
  {"$sort": {
    "dateofbirth": -1,
  }},

  // Only include the first 3 youngest people
  {"$limit": 3},

  // Exclude unrequired fields from each person record
  {"$unset": [
    "_id",
    "vocation",
    "address",
  ]},
];
```

Execution

Execute the aggregation using the defined pipeline and also view its explain plan:

```
db.persons.aggregate(pipeline);
```

```
db.persons.explain("executionStats").aggregate(pipeline);
```

Expected Results

Three documents should be returned, representing the three youngest people who are engineers (ordered by youngest first), omitting the `_id` or `address` attributes of each person, as shown below:

```
[
  {
    person_id: '7363626383',
    firstname: 'Carl',
    lastname: 'Simmons',
    dateofbirth: ISODate('1998-12-26T13:13:55.000Z')
  },
  {
    person_id: '1723338115',
    firstname: 'Olive',
    lastname: 'Ranieri',
    dateofbirth: ISODate('1985-05-12T23:14:30.000Z'),
    gender: 'FEMALE'
  },
  {
    person_id: '6392529400',
    firstname: 'Elise',
    lastname: 'Smith',
    dateofbirth: ISODate('1972-01-13T09:32:07.000Z')
  }
]
```

Observations

- **Index Use.** A basic aggregation pipeline, where if many records belong to the collection, a compound index for `vocation + dateofbirth` should exist to enable the database to fully optimise the execution of the pipeline combining the filter of the `$match` stage with the sort from the `sort` stage and the limit of the `limit` stage.
- **Unset Use.** An `$unset` stage is used rather than a `$project` stage. This enables the pipeline to avoid being verbose. More importantly, it means the pipeline does not have to be modified if a new field appears in documents added in the future (for example, see the `gender` field that appears in only *Olive's* record).
- **MQL Similarity.** For reference, the MQL equivalent for you to achieve the same result is shown below (you can try this in the *Shell*):

```
db.persons.find(
    {"vocation": "ENGINEER"},
    {"_id": 0, "vocation": 0, "address": 0},
  ).sort(
    {"dateofbirth": -1}
  ).limit(3);
```

Group & Total

Minimum MongoDB Version: 4.2

Scenario

You want to generate a report to show what each shop customer purchased in 2020. You will group the individual order records by customer, capturing each customer's first purchase date, the number of orders they made, the total value of all their orders and a list of their order items sorted by date.

Sample Data Population

Drop any old version of the database (if it exists) and then populate a new `orders` collection with 9 order documents spanning 2019-2021, for 3 different unique customers:

```
use book-group-and-total;
db.dropDatabase();

// Create index for an orders collection
db.orders.createIndex({"orderdate": -1});

// Insert records into the orders collection
db.orders.insertMany([
  {
    "customer_id": "elise_smith@myemail.com",
    "orderdate": ISODate("2020-05-30T08:35:52Z"),
    "value": NumberDecimal("231.43"),
  },
  {
    "customer_id": "elise_smith@myemail.com",
    "orderdate": ISODate("2020-01-13T09:32:07Z"),
    "value": NumberDecimal("99.99"),
  },
  {
    "customer_id": "oranieri@warmmail.com",
    "orderdate": ISODate("2020-01-01T08:25:37Z"),
    "value": NumberDecimal("63.13"),
  },
  {
    "customer_id": "tj@wheresmyemail.com",
    "orderdate": ISODate("2019-05-28T19:13:32Z"),
    "value": NumberDecimal("2.01"),
  },
  {
    "customer_id": "tj@wheresmyemail.com",
    "orderdate": ISODate("2020-11-23T22:56:53Z"),
    "value": NumberDecimal("187.99"),
  },
  {
    "customer_id": "tj@wheresmyemail.com",
    "orderdate": ISODate("2020-08-18T23:04:48Z"),
    "value": NumberDecimal("4.59"),
  },
  {
    "customer_id": "elise_smith@myemail.com",
    "orderdate": ISODate("2020-12-26T08:55:46Z"),
    "value": NumberDecimal("48.50"),
  },
  {
    "customer_id": "tj@wheresmyemail.com",
    "orderdate": ISODate("2021-02-29T07:49:32Z"),
    "value": NumberDecimal("1024.89"),
  },
  {
    "customer_id": "elise_smith@myemail.com",
    "orderdate": ISODate("2020-10-03T13:49:44Z"),
    "value": NumberDecimal("102.24"),
  },
]);
```

Aggregation Pipeline

Define a single pipeline ready to perform the aggregation:

```
var pipeline = [
  // Match only orders made in 2020
  {"$match": {
    "orderdate": {
      "$gte": ISODate("2020-01-01T00:00:00Z"),
      "$lt": ISODate("2021-01-01T00:00:00Z"),
    },
  }},

  // Sort by order date ascending (required to pick out 'first_purchase_date'
below)
  {"$sort": {
    "orderdate": 1,
  }},

  // Group by customer
  {"$group": {
    "_id": "$customer_id",
    "first_purchase_date": {"$first": "$orderdate"},
    "total_value": {"$sum": "$value"},
    "total_orders": {"$sum": 1},
    "orders": {"$push": {"orderdate": "$orderdate", "value": "$value"}},
  }},

  // Sort by each customer's first purchase date
  {"$sort": {
    "first_purchase_date": 1,
  }},

  // Set customer's ID to be value of the field that was grouped on
  {"$set": {
    "customer_id": "$_id",
  }},

  // Omit unwanted fields
  {"$unset": [
    "_id",
  ]},
];
```

Execution

Execute the aggregation using the defined pipeline and also view its explain plan:

```
db.orders.aggregate(pipeline);
```

```
db.orders.explain("executionStats").aggregate(pipeline);
```

Expected Results

Three documents should be returned, representing the three customers, each showing the customer's first purchase date, the total value of all their orders, the number of orders they made and a list of each order's detail, for 2020 only, as shown below:

```
[
  {
    customer_id: 'oranieri@warmmail.com',
    first_purchase_date: ISODate('2020-01-01T08:25:37.000Z'),
    total_value: NumberDecimal('63.13'),
    total_orders: 1,
    orders: [
      {orderdate: ISODate('2020-01-01T08:25:37.000Z'), value:
NumberDecimal('63.13')}
    ]
  },
  {
    customer_id: 'elise_smith@myemail.com',
    first_purchase_date: ISODate('2020-01-13T09:32:07.000Z'),
    total_value: NumberDecimal('482.16'),
    total_orders: 4,
    orders: [
      {orderdate: ISODate('2020-01-13T09:32:07.000Z'), value:
NumberDecimal('99.99')},
      {orderdate: ISODate('2020-05-30T08:35:52.000Z'), value:
NumberDecimal('231.43')},
      {orderdate: ISODate('2020-10-03T13:49:44.000Z'), value:
NumberDecimal('102.24')},
      {orderdate: ISODate('2020-12-26T08:55:46.000Z'), value:
NumberDecimal('48.50')}
    ]
  },
  {
    customer_id: 'tj@wheresmyemail.com',
    first_purchase_date: ISODate('2020-08-18T23:04:48.000Z'),
    total_value: NumberDecimal('192.58'),
    total_orders: 2,
    orders: [
      {orderdate: ISODate('2020-08-18T23:04:48.000Z'), value:
NumberDecimal('4.59')},
      {orderdate: ISODate('2020-11-23T22:56:53.000Z'), value:
NumberDecimal('187.99')}
    ]
  }
]
```

Note, the order of fields shown for each document may vary.

Observations

- **Double Sort Use.** It is necessary to perform a `$sort` on the order date both before and after the `$group` stage. The `$sort` before the `$group` is required because the `$group` stage uses a `$first` group accumulator to capture just the first order's `orderdate` value

for each grouped customer. The `$sort` after the `$group` is required because the act of having just grouped on customer ID will mean that the records are no longer sorted by purchase date for the records coming out of the `$group` stage.

- **Renaming Group.** Towards the end of the pipeline, you will see what is a typical pattern for pipelines that use `$group`, consisting of a combination of `$set` + `$unset` stages, to essentially take the group's key (which is always called `_id`) and substitute it with a more meaningful name (`customer_id`).

- **Lossless Decimals.** You may notice the pipeline uses a `NumberDecimal()` function to ensure the order amounts in the inserted records are using a lossless decimal type, IEEE 754 decimal128. In this example, if you use a JSON *float* or *double* type instead, the order totals will suffer from a loss of precision. For instance, for the customer `elise_smith@myemail.com`, if you use a *double* type, the `total_value` result will have the value shown in the second line below, rather than the first line:

```
// Desired result achieved by using decimal128 types
total_value: NumberDecimal('482.16')

// Result that occurs if using float or double types instead
total_value: 482.15999999999997
```

Unpack Arrays & Group Differently

Minimum MongoDB Version: 4.2

Scenario

You want to generate a retail report to list the total value and quantity of expensive products sold (valued over 15 dollars). The source data is a list of shop orders, where each order contains the set of products purchased as part of the order.

Sample Data Population

Drop any old version of the database (if it exists) and then populate a new `orders` collection where each document contains an array of products purchased:

```
use book-unpack-array-group-differently;
db.dropDatabase();

// Insert 4 records into the orders collection each with 1+ product items
db.orders.insertMany([
  {
    "order_id": 6363763262239,
    "products": [
      {
        "prod_id": "abc12345",
        "name": "Asus Laptop",
        "price": NumberDecimal("431.43"),
      },
      {
        "prod_id": "def45678",
        "name": "Karcher Hose Set",
        "price": NumberDecimal("22.13"),
      },
    ],
  },
  {
    "order_id": 1197372932325,
    "products": [
      {
        "prod_id": "abc12345",
        "name": "Asus Laptop",
        "price": NumberDecimal("429.99"),
      },
    ],
  },
  {
    "order_id": 9812343774839,
    "products": [
      {
        "prod_id": "pqr88223",
        "name": "Morphy Richardds Food Mixer",
        "price": NumberDecimal("431.43"),
      },
      {
        "prod_id": "def45678",
        "name": "Karcher Hose Set",
        "price": NumberDecimal("21.78"),
      },
    ],
  },
  {
    "order_id": 4433997244387,
    "products": [
      {
        "prod_id": "def45678",
        "name": "Karcher Hose Set",
        "price": NumberDecimal("23.43"),
      },
      {
```

```
        "prod_id": "jkl77336",
        "name": "Picky Pencil Sharpener",
        "price": NumberDecimal("0.67"),
      },
      {
        "prod_id": "xyz11228",
        "name": "Russell Hobbs Chrome Kettle",
        "price": NumberDecimal("15.76"),
      },
    ],
  },
]);
```

Aggregation Pipeline

Define a single pipeline ready to perform the aggregation:

```
var pipeline = [
  // Unpack each product from the each order's product as a new separate record
  {"$unwind": {
    "path": "$products",
  }},

  // Match only products valued greater than 15.00
  {"$match": {
    "products.price": {
      "$gt": NumberDecimal("15.00"),
    },
  }},

  // Group by product type, capturing each product's total value + quantity
  {"$group": {
    "_id": "$products.prod_id",
    "product": {"$first": "$products.name"},
    "total_value": {"$sum": "$products.price"},
    "quantity": {"$sum": 1},
  }},

  // Set product id to be the value of the field that was grouped on
  {"$set": {
    "product_id": "$_id",
  }},

  // Omit unwanted fields
  {"$unset": [
    "_id",
  ]},
];
```

Execution

Execute the aggregation using the defined pipeline and also view its explain plan:

```
db.orders.aggregate(pipeline);
```

```
db.orders.explain("executionStats").aggregate(pipeline);
```

Expected Results

Four documents should be returned, representing only the four expensive products that were referenced multiple times in the customer orders, each showing the product's total order value and amount sold as shown below:

```
[
  {
    product_id: 'pqr88223',
    product: 'Morphy Richardds Food Mixer',
    total_value: NumberDecimal('431.43'),
    quantity: 1
  },
  {
    product_id: 'abc12345',
    product: 'Asus Laptop',
    total_value: NumberDecimal('861.42'),
    quantity: 2
  },
  {
    product_id: 'def45678',
    product: 'Karcher Hose Set',
    total_value: NumberDecimal('67.34'),
    quantity: 3
  },
  {
    product_id: 'xyz11228',
    product: 'Russell Hobbs Chrome Kettle',
    total_value: NumberDecimal('15.76'),
    quantity: 1
  }
]
```

Note, the order of fields shown for each document may vary.

Observations

- **Unwinding Arrays.** The `$unwind` stage is a powerful concept, although often unfamiliar to many developers initially. Distilled down, it does one simple thing: it generates a new record for each element in an array field of every input document. If a source collection has 3 documents and each document contains an array of 4 elements, then performing an `$unwind` on each record's array field produces 12 records (3 x 4).
- **Introducing A Partial Match**. The current example pipeline scans all documents in the collection and then filters out unpacked products where `price > 15.00`. If the pipeline executed this filter as the first stage, it would incorrectly produce some result product records with a value of 15 dollars or less. This would be the case for an order composed of both inexpensive and expensive products. However, you can still improve the pipeline by including an additional "partial match" filter at the start of the pipeline for products valued at over 15 dollars. The aggregation could leverage an index (on `products.price`), resulting in a partial rather than full collection scan. This extra filter stage is beneficial if the input data set is large and many customer orders are for inexpensive items only. This approach is described in the chapter Pipeline Performance Considerations.

Distinct List Of Values

Minimum MongoDB Version: 4.2

Scenario

You want to query a collection of persons where each document contains data on one or more languages spoken by the person. The query result should be an alphabetically sorted list of unique languages that a developer can subsequently use to populate a list of values in a user interface's "drop-down" widget.

This example is the equivalent of a *SELECT DISTINCT* statement in SQL.

Sample Data Population

Drop any old version of the database (if it exists) and then populate a new `persons` collection with 9 documents:

```
use book-distinct-values;
db.dropDatabase();

// Insert records into the persons collection
db.persons.insertMany([
  {
    "firstname": "Elise",
    "lastname": "Smith",
    "vocation": "ENGINEER",
    "language": "English",
  },
  {
    "firstname": "Olive",
    "lastname": "Ranieri",
    "vocation": "ENGINEER",
    "language": ["Italian", "English"],
  },
  {
    "firstname": "Toni",
    "lastname": "Jones",
    "vocation": "POLITICIAN",
    "language": ["English", "Welsh"],
  },
  {
    "firstname": "Bert",
    "lastname": "Gooding",
    "vocation": "FLORIST",
    "language": "English",
  },
  {
    "firstname": "Sophie",
    "lastname": "Celements",
    "vocation": "ENGINEER",
    "language": ["Gaelic", "English"],
  },
  {
    "firstname": "Carl",
    "lastname": "Simmons",
    "vocation": "ENGINEER",
    "language": "English",
  },
  {
    "firstname": "Diego",
    "lastname": "Lopez",
    "vocation": "CHEF",
    "language": "Spanish",
  },
  {
    "firstname": "Helmut",
    "lastname": "Schneider",
    "vocation": "NURSE",
    "language": "German",
  },
  {
```

```
    "firstname": "Valerie",
    "lastname": "Dubois",
    "vocation": "SCIENTIST",
    "language": "French",
  },
]);
```

Aggregation Pipeline

Define a single pipeline ready to perform the aggregation:

```
var pipeline = [
  // Unpack each language field which may be an array or a single value
  {"$unwind": {
    "path": "$language",
  }},

  // Group by language
  {"$group": {
    "_id": "$language",
  }},

  // Sort languages alphabetically
  {"$sort": {
    "_id": 1,
  }},

  // Change _id field's name to 'language'
  {"$set": {
    "language": "$_id",
    "_id": "$$REMOVE",
  }},
];
```

Execution

Execute the aggregation using the defined pipeline and also view its explain plan:

```
db.persons.aggregate(pipeline);
```

```
db.persons.explain("executionStats").aggregate(pipeline);
```

Expected Results

Seven unique language names should be returned sorted in alphabetical order, as shown below:

```
[
  {language: 'English'},
  {language: 'French'},
  {language: 'Gaelic'},
  {language: 'German'},
  {language: 'Italian'},
  {language: 'Spanish'},
  {language: 'Welsh'}
]
```

Observations

- **Unwinding Non-Arrays.** In some of the example's documents, the `language` field is an array, whilst in others, the field is a simple string value. The `$unwind` stage can seamlessly deal with both field types and does not throw an error if it encounters a non-array value. Instead, if the field is not an array, the stage outputs a single record using the field's string value in the same way it would if the field was an array containing just one element. If you are sure the field in every document will only ever be a simple field rather than an array, you can omit this first stage (`$unwind`) from the pipeline.

- **Group ID Provides Unique Values.** By grouping on a single field and not accumulating other fields such as total or count, the output of a `$group` stage is just every unique group's ID, which in this case is every unique language.

- **Unset Alternative.** For the pipeline to be consistent with earlier examples in this book, it could have included an additional `$unset` stage to exclude the `_id` field. However, partly to show another way, the example pipeline used here marks the `_id` field for exclusion in the `$set` stage by being assigned the `$$REMOVE` variable.

Joining Data Examples

This section provides examples for joining a source collection used by an aggregation pipeline with another collection using different approaches.

One-to-One Join

Minimum MongoDB Version: 4.4 *(due to use of $first array operator)*

Scenario

You want to generate a report to list all shop purchases for 2020, showing the product's name and category for each order, rather than the product's id. To achieve this, you need to take the customer *orders* collection and join each order record to the corresponding product record in the *products* collection. There is a many:1 relationship between both collections, resulting in a 1:1 join when matching an order to a product. The join will use a single field comparison between both sides, based on the product's id.

Sample Data Population

Drop any old version of the database (if it exists) and then populate new `products` and `orders` collections with documents spanning 2019-2021 (the database commands have been split in two to enable your clipboard to hold all the text - ensure you copy and execute each of the two sections):

-Part 1-

```
use book-one-to-one-join;
db.dropDatabase();

// Create index for a products collection
db.products.createIndex({"id": 1});

// Insert 4 records into the products collection
db.products.insertMany([
  {
    "id": "a1b2c3d4",
    "name": "Asus Laptop",
    "category": "ELECTRONICS",
    "description": "Good value laptop for students",
  },
  {
    "id": "z9y8x7w6",
    "name": "The Day Of The Triffids",
    "category": "BOOKS",
    "description": "Classic post-apocalyptic novel",
  },
  {
    "id": "ff11gg22hh33",
    "name": "Morphy Richardds Food Mixer",
    "category": "KITCHENWARE",
    "description": "Luxury mixer turning good cakes into great",
  },
  {
    "id": "pqr678st",
    "name": "Karcher Hose Set",
    "category": "GARDEN",
    "description": "Hose + nosels + winder for tidy storage",
  },
]);
```

-Part 2-

```
// Create index for a orders collection
db.orders.createIndex({"orderdate": -1});

// Insert 4 records into the orders collection
db.orders.insertMany([
  {
    "customer_id": "elise_smith@myemail.com",
    "orderdate": ISODate("2020-05-30T08:35:52Z"),
    "product_id": "a1b2c3d4",
    "value": NumberDecimal("431.43"),
  },
  {
    "customer_id": "tj@wheresmyemail.com",
    "orderdate": ISODate("2019-05-28T19:13:32Z"),
    "product_id": "z9y8x7w6",
    "value": NumberDecimal("5.01"),
  },
  {
    "customer_id": "oranieri@warmmail.com",
    "orderdate": ISODate("2020-01-01T08:25:37Z"),
    "product_id": "ff11gg22hh33",
    "value": NumberDecimal("63.13"),
  },
  {
    "customer_id": "jjones@tepidmail.com",
    "orderdate": ISODate("2020-12-26T08:55:46Z"),
    "product_id": "a1b2c3d4",
    "value": NumberDecimal("429.65"),
  },
]);
```

Aggregation Pipeline

Define a single pipeline ready to perform the aggregation:

```
var pipeline = [
  // Match only orders made in 2020
  {"$match": {
    "orderdate": {
      "$gte": ISODate("2020-01-01T00:00:00Z"),
      "$lt": ISODate("2021-01-01T00:00:00Z"),
    }
  }},

  // Join "product_id" in orders collection to "id" in products" collection
  {"$lookup": {
    "from": "products",
    "localField": "product_id",
    "foreignField": "id",
    "as": "product_mapping",
  }},

  // For this data model, will always be 1 record in right-side
  // of join, so take 1st joined array element
  {"$set": {
    "product_mapping": {"$first": "$product_mapping"},
  }},

  // Extract the joined embeded fields into top level fields
  {"$set": {
    "product_name": "$product_mapping.name",
    "product_category": "$product_mapping.category",
  }},

  // Omit unwanted fields
  {"$unset": [
    "_id",
    "product_id",
    "product_mapping",
  ]},
];
```

Execution

Execute the aggregation using the defined pipeline and also view its explain plan:

```
db.orders.aggregate(pipeline);
```

```
db.orders.explain("executionStats").aggregate(pipeline);
```

Expected Results

Three documents should be returned, representing the three customers orders that occurred in 2020, but with each order's `product_id` field replaced by two new looked up fields, `product_name` and `product_category`, as shown below:

```
[
  {
    customer_id: 'elise_smith@myemail.com',
    orderdate: ISODate('2020-05-30T08:35:52.000Z'),
    value: NumberDecimal('431.43'),
    product_name: 'Asus Laptop',
    product_category: 'ELECTRONICS'
  },
  {
    customer_id: 'oranieri@warmmail.com',
    orderdate: ISODate('2020-01-01T08:25:37.000Z'),
    value: NumberDecimal('63.13'),
    product_name: 'Morphy Richardds Food Mixer',
    product_category: 'KITCHENWARE'
  },
  {
    customer_id: 'jjones@tepidmail.com',
    orderdate: ISODate('2020-12-26T08:55:46.000Z'),
    value: NumberDecimal('429.65'),
    product_name: 'Asus Laptop',
    product_category: 'ELECTRONICS'
  }
]
```

Observations

- **Single Field Match.** The pipeline includes a `$lookup` join between a single field from each collection. For an illustration of performing a join based on two or more matching fields, see the Multi-Field Join & One-to-Many example.
- **First Element Assumption.** In this particular data model example, the join between the two collections is 1:1. Therefore the returned array of joined elements coming out of the `$lookup` stage always contains precisely one array element. As a result, the pipeline extracts the data from this first array element only, using the `$first` operator. For an illustration of performing a 1:many join instead, see the Multi-Field Join & One-to-Many example.
- **First Element For Earlier MongoDB Versions.** MongoDB only introduced the `$first` array operator expression in version 4.4. However, it is straightforward for you to replace

its use in the pipeline with an equivalent solution, using the `$arrayElemAt` operator, to then allow the pipeline to work in MongoDB versions before 4.4:

```
// $first equivalent
"product_mapping": {"$arrayElemAt": ["$product_mapping", 0]},
```

Multi-Field Join & One-to-Many

Minimum MongoDB Version: 4.2

Scenario

You want to generate a report to list all the orders made for each product in 2020. To achieve this, you need to take a shop's *products* collection and join each product record to all its orders stored in an *orders* collection. There is a 1:many relationship between both collections, based on a match of two fields on each side. Rather than joining on a single field like `product_id` (which doesn't exist in this data set), you need to use two common fields to join (`product_name` and `product_variation`).

Note that the requirement to perform a 1:many join does not mandate the need to join by multiple fields on each side of the join. However, in this example, it has been deemed beneficial to show both of these aspects in one place.

Sample Data Population

Drop any old version of the database (if it exists) and then populate new `products` and `orders` collections with documents spanning 2019-2021 (the database commands have been split in two to enable your clipboard to hold all the text - ensure you copy and execute each of the two sections):

-Part 1-

```
use book-multi-one-to-many;
db.dropDatabase();

// Insert 6 records into the products collection
db.products.insertMany([
  {
    "name": "Asus Laptop",
    "variation": "Ultra HD",
    "category": "ELECTRONICS",
    "description": "Great for watching movies",
  },
  {
    "name": "Asus Laptop",
    "variation": "Normal Display",
    "category": "ELECTRONICS",
    "description": "Good value laptop for students",
  },
  {
    "name": "The Day Of The Triffids",
    "variation": "1st Edition",
    "category": "BOOKS",
    "description": "Classic post-apocalyptic novel",
  },
  {
    "name": "The Day Of The Triffids",
    "variation": "2nd Edition",
    "category": "BOOKS",
    "description": "Classic post-apocalyptic novel",
  },
  {
    "name": "Morphy Richards Food Mixer",
    "variation": "Deluxe",
    "category": "KITCHENWARE",
    "description": "Luxury mixer turning good cakes into great",
  },
  {
    "name": "Karcher Hose Set",
    "variation": "Full Monty",
    "category": "GARDEN",
    "description": "Hose + nosels + winder for tidy storage",
  },
]);
```

-Part 2-

```
// Create index for the orders collection
db.orders.createIndex({"product_name": 1, "product_variation": 1});

// Insert 4 records into the orders collection
db.orders.insertMany([
  {
    "customer_id": "elise_smith@myemail.com",
    "orderdate": ISODate("2020-05-30T08:35:52Z"),
    "product_name": "Asus Laptop",
    "product_variation": "Normal Display",
    "value": NumberDecimal("431.43"),
  },
  {
    "customer_id": "tj@wheresmyemail.com",
    "orderdate": ISODate("2019-05-28T19:13:32Z"),
    "product_name": "The Day Of The Triffids",
    "product_variation": "2nd Edition",
    "value": NumberDecimal("5.01"),
  },
  {
    "customer_id": "oranieri@warmmail.com",
    "orderdate": ISODate("2020-01-01T08:25:37Z"),
    "product_name": "Morphy Richards Food Mixer",
    "product_variation": "Deluxe",
    "value": NumberDecimal("63.13"),
  },
  {
    "customer_id": "jjones@tepidmail.com",
    "orderdate": ISODate("2020-12-26T08:55:46Z"),
    "product_name": "Asus Laptop",
    "product_variation": "Normal Display",
    "value": NumberDecimal("429.65"),
  },
]);
```

Aggregation Pipeline

Define a single pipeline ready to perform the aggregation:

```
var pipeline = [
  // Join by 2 fields in in products collection to 2 fields in orders collection
  {"$lookup": {
    "from": "orders",
    "let": {
      "prdname": "$name",
      "prdvartn": "$variation",
    },
    // Embedded pipeline to control how the join is matched
    "pipeline": [
      // Join by two fields in each side
      {"$match":
        {"$expr":
          {"$and": [
            {"$eq": ["$product_name",  "$$prdname"]},
            {"$eq": ["$product_variation",  "$$prdvartn"]},
          ]},
        },
      },

      // Match only orders made in 2020
      {"$match": {
        "orderdate": {
          "$gte": ISODate("2020-01-01T00:00:00Z"),
          "$lt": ISODate("2021-01-01T00:00:00Z"),
        }
      }},

      // Exclude some unwanted fields from the right side of the join
      {"$unset": [
        "_id",
        "product_name",
        "product_variation",
      ]},
    ],
    as: "orders",
  }},

  // Only show products that have at least one order
  {"$match": {
    "orders": {"$ne": []},
  }},

  // Omit unwanted fields
  {"$unset": [
    "_id",
  ]},
];
```

Execution

Execute the aggregation using the defined pipeline and also view its explain plan:

```
db.products.aggregate(pipeline);
```

```
db.products.explain("executionStats").aggregate(pipeline);
```

Expected Results

Two documents should be returned, representing the two products that had one or more orders in 2020, with the orders embedded in an array against each product, as shown below:

```
[
  {
    name: 'Asus Laptop',
    variation: 'Normal Display',
    category: 'ELECTRONICS',
    description: 'Good value laptop for students',
    orders: [
      {
        customer_id: 'elise_smith@myemail.com',
        orderdate: ISODate('2020-05-30T08:35:52.000Z'),
        value: NumberDecimal('431.43')
      },
      {
        customer_id: 'jjones@tepidmail.com',
        orderdate: ISODate('2020-12-26T08:55:46.000Z'),
        value: NumberDecimal('429.65')
      }
    ]
  },
  {
    name: 'Morphy Richards Food Mixer',
    variation: 'Deluxe',
    category: 'KITCHENWARE',
    description: 'Luxury mixer turning good cakes into great',
    orders: [
      {
        customer_id: 'oranieri@warmmail.com',
        orderdate: ISODate('2020-01-01T08:25:37.000Z'),
        value: NumberDecimal('63.13')
      }
    ]
  }
]
```

Observations

- **Multiple Join Fields.** To perform a join of two or more fields between the two collections, you need to use a let parameter rather than specifying the `localField` and `foreignField` parameters used in a single field join. With a `let` parameter, you bind multiple fields from the first collection into variables ready to be used in the joining process. You use an embedded `pipeline` inside the `$lookup` stage to match the *bind* variables with fields in the second collection's records. In this instance, because the `$expr` operator performs an equality comparison specifically (as opposed to a range comparison), the aggregation runtime can employ an appropriate index for this match.
- **Reducing Array Content.** The presence of an embedded pipeline in the `$lookup` stage provides an opportunity to filter out three unwanted fields brought in from the second collection. Instead, you could use an `$unset` stage later in the top-level pipeline to project out these unwanted array elements. If you need to perform more complex array content filtering rules, you can use the approach described in section *2. Avoid Unwinding & Regrouping Documents Just To Process Array Elements*" of the *Pipeline Performance Considerations* chapter.

Data Types Conversion Examples

This section provides examples for converting weakly typed fields, represented as strings in documents, to strongly typed fields that are easier for applications to query and use.

Strongly-Typed Conversion

Minimum MongoDB Version: 4.2

Scenario

A 3rd party has imported a set of *retail orders* into a MongoDB collection but with all data typing lost (it stored all field values as strings). You want to re-establish correct typing for all the documents and copy them into a new "cleaned" collection. You can incorporate such type transformation logic in the aggregation pipeline because you know the type each field had in the original record structure.

Unlike most examples in this book, the aggregation pipeline writes its output to a collection rather than streaming the results back to the calling application.

Sample Data Population

Drop any old version of the database (if it exists) and then populate a new `orders` collection with three orders documents, where each order has text fields only (note, the second document is intentionality missing the field `reported` in the sub-document `further_info`):

```
use book-convert-to-strongly-typed;
db.dropDatabase();

// Insert orders documents
db.orders.insertMany([
  {
    "customer_id": "elise_smith@myemail.com",
    "order_date": "2020-05-30T08:35:52",
    "value": "231.43",
    "further_info": {
      "item_qty": "3",
      "reported": "false",
    },
  },
  {
    "customer_id": "oranieri@warmmail.com",
    "order_date": "2020-01-01T08:25:37",
    "value": "63.13",
    "further_info": {
      "item_qty": "2",
    },
  },
  {
    "customer_id": "tj@wheresmyemail.com",
    "order_date": "2019-05-28T19:13:32",
    "value": "2.01",
    "further_info": {
      "item_qty": "1",
      "reported": "true",
    },
  },
]);
```

Aggregation Pipeline

Define a single pipeline ready to perform the aggregation:

```
var pipeline = [
  // Convert strings to required types
  {"$set": {
    "order_date": {"$toDate": "$order_date"},
    "value": {"$toDecimal": "$value"},
    "further_info.item_qty": {"$toInt": "$further_info.item_qty"},
    "further_info.reported": {"$switch": {
      "branches": [
        {"case": {"$eq": [{"$toLower": "$further_info.reported"}, "true"]},
"then": true},
        {"case": {"$eq": [{"$toLower": "$further_info.reported"}, "false"]},
"then": false},
      ],
      "default": {"$ifNull": ["$further_info.reported", "$$REMOVE"]},
    }},
  }},

  // Output to an unsharded or sharded collection
  {"$merge": {
    "into": "orders_typed",
  }},
];
```

Execution

Execute the aggregation using the defined pipeline to generate and populate a new collection called `orders_typed`:

```
db.orders.aggregate(pipeline);
```

Check the contents of the new `orders_typed` collection to ensure the relevant fields are now appropriately typed:

```
db.orders_typed.find();
```

View the explain plan for the aggregation pipeline:

```
db.orders.explain("executionStats").aggregate(pipeline);
```

Expected Results

The same number of documents should appear in the new `orders_typed` collection as the source collection had, with the same field structure and fields names, but now using strongly-

typed boolean/date/integer/decimal values where appropriate, as shown below:

```
[
  {
    _id: ObjectId('6064381b7aa89666258201fd'),
    customer_id: 'elise_smith@myemail.com',
    further_info: {
      item_qty: 3,
      reported: false
    },
    order_date: ISODate('2020-05-30T08:35:52.000Z'),
    value: NumberDecimal('231.43')
  },
  {
    _id: ObjectId('6064381b7aa89666258201fe'),
    customer_id: 'oranieri@warmmail.com',
    further_info: {
      item_qty: 2
    },
    order_date: ISODate('2020-01-01T08:25:37.000Z'),
    value: NumberDecimal('63.13')
  },
  {
    _id: ObjectId('6064381b7aa89666258201ff'),
    customer_id: 'tj@wheresmyemail.com',\
    further_info: {
      item_qty: 1,
      reported: true
    },
    order_date: ISODate('2019-05-28T19:13:32.000Z'),
    value: NumberDecimal('2.01')
  }
]
```

Observations

- **Boolean Conversion.** The pipeline's conversions for integers, decimals, and dates are straightforward using the corresponding operator expressions, `$toInt`, `$toDecimal` and `$toDate`. However, the operator expression `$toBool` is not used for the boolean conversion. This is because `$toBool` will convert any non-empty string to *true* regardless of its value. As a result, the pipeline uses a `$switch` operator to compare the lowercase version of strings with the text `'true'` and `'false'`, returning the matching boolean.

- **Preserving Non-Existence.** The field `further_info.reported` is an optional field in this scenario. The field may not always appear in a document, as illustrated by one of the three documents in the example. If a field is not present in a document, this potentially significant fact should never be lost. The pipeline includes additional logic for the

`further_info.reported` field to preserve this information. The pipeline ensures the field is not included in the output document if it didn't exist in the source document. A `$ifNull` conditional operator is used, which returns the `$$REMOVE` marker flag if the field is missing, instructing the aggregation engine to omit it.

- **Output To A Collection.** The pipeline uses a `$merge` stage to instruct the aggregation engine to write the output to a collection rather than returning a stream of results. For this example, the default settings for `$merge` are sufficient. Each transformed record coming out of the aggregation pipeline becomes a new record in the target collection. The pipeline could have used a `$out` rather than a `$merge` stage. However, because `$merge` supports both unsharded and sharded collections, whereas `$out` only supports the former, `$merge` provides a more universally applicable example. If your aggregation needs to create a brand new unsharded collection, `$out` may be a little faster because the aggregation will completely replace the existing collection if it exists. Using `$merge`, the system has to perform more checks for every record the aggregation inserts (even though, in this case, it will be to a new collection).

- **Trickier Date Conversions.** In this example, the date strings contain all the date parts required by the `$toDate` operator to perform a conversion correctly. In some situations, this may not be the case, and a date string may be missing some valuable information (e.g. which century a 2-character year string is for, such as the century `19` or `21`). To understand how to deal with these cases, see the Convert Incomplete Date Strings example chapter.

Convert Incomplete Date Strings

Minimum MongoDB Version: 4.2

Scenario

An application is ingesting *payment* documents into a MongoDB collection where each document's *payment date* field contains a string looking vaguely like a date-time, such as `"01-JAN-20 01.01.01.123000000"`. You want to convert each *payment date* into a valid BSON date type when aggregating the payments. However, the payment date fields do not contain all the information required for you to determine the exact date-time accurately. Therefore you cannot use just the MongoDB's Date Operator Expressions directly to perform the text-to-date conversion. Each of these text fields is missing the following information:

- The specific **century** (1900s?, 2000s, other?)
- The specific **time-zone** (GMT?, IST?, PST?, other?)

- The specific **language** that the three-letter month abbreviation represents (is "JAN" in French? in English? other?)

You subsequently learn that all the payment records are for the **21st century** only, the time-zone used when ingesting the data is **UTC**, and the language used is **English**. Armed with this information, you build an aggregation pipeline to transform these text fields into date fields.

Sample Data Population

Drop any old version of the database (if it exists) and then populate a new *payments* collection with 12 sample payments documents, providing coverage across all 12 months for the year 2020, with random time elements.

```
use book-convert-incomplete-dates;
db.dropDatabase();

// Insert records into the payments collection
db.payments.insertMany([
  {"account": "010101", "payment_date": "01-JAN-20 01.01.01.123000000", "amount":
1.01},
  {"account": "020202", "payment_date": "02-FEB-20 02.02.02.456000000", "amount":
2.02},
  {"account": "030303", "payment_date": "03-MAR-20 03.03.03.789000000", "amount":
3.03},
  {"account": "040404", "payment_date": "04-APR-20 04.04.04.012000000", "amount":
4.04},
  {"account": "050505", "payment_date": "05-MAY-20 05.05.05.345000000", "amount":
5.05},
  {"account": "060606", "payment_date": "06-JUN-20 06.06.06.678000000", "amount":
6.06},
  {"account": "070707", "payment_date": "07-JUL-20 07.07.07.901000000", "amount":
7.07},
  {"account": "080808", "payment_date": "08-AUG-20 08.08.08.234000000", "amount":
8.08},
  {"account": "090909", "payment_date": "09-SEP-20 09.09.09.567000000", "amount":
9.09},
  {"account": "101010", "payment_date": "10-OCT-20 10.10.10.890000000", "amount":
10.10},
  {"account": "111111", "payment_date": "11-NOV-20 11.11.11.111000000", "amount":
11.11},
  {"account": "121212", "payment_date": "12-DEC-20 12.12.12.999000000", "amount":
12.12}
]);
```

Aggregation Pipeline

Define a single pipeline ready to perform the aggregation:

```
var pipeline = [
  // Change field from a string to a date, filling in the missing gaps
  {"$set": {
    "payment_date": {
      "$let": {
        "vars": {
          "txt": "$payment_date",  // Assign "payment_date" field to variable
"txt",
          "month": {"$substrCP": ["$payment_date", 3, 3]},  // Extract month text
        },
        "in": {
          "$dateFromString": {"format": "%d-%m-%Y %H.%M.%S.%L", "dateString":
            {"$concat": [
              {"$substrCP": ["$$txt", 0, 3]},  // Use 1st 3 chars in string
              {"$switch": {"branches": [  // Replace month 3 chars with month
number
                {"case": {"$eq": ["$$month", "JAN"]}, "then": "01"},
                {"case": {"$eq": ["$$month", "FEB"]}, "then": "02"},
                {"case": {"$eq": ["$$month", "MAR"]}, "then": "03"},
                {"case": {"$eq": ["$$month", "APR"]}, "then": "04"},
                {"case": {"$eq": ["$$month", "MAY"]}, "then": "05"},
                {"case": {"$eq": ["$$month", "JUN"]}, "then": "06"},
                {"case": {"$eq": ["$$month", "JUL"]}, "then": "07"},
                {"case": {"$eq": ["$$month", "AUG"]}, "then": "08"},
                {"case": {"$eq": ["$$month", "SEP"]}, "then": "09"},
                {"case": {"$eq": ["$$month", "OCT"]}, "then": "10"},
                {"case": {"$eq": ["$$month", "NOV"]}, "then": "11"},
                {"case": {"$eq": ["$$month", "DEC"]}, "then": "12"},
              ], "default": "ERROR"}},
              "-20",  // Add hyphen + hardcoded century 2 digits
              {"$substrCP": ["$$txt", 7, 15]}  // Use remaining 3 millis (ignore
last 6 nanosecs)
            ]
          }}
        }
      }
    },
  }},

  // Omit unwanted fields
  {"$unset": [
    "_id",
  ]},
];
```

Execution

Execute the aggregation using the defined pipeline and also view its explain plan:

```
db.payments.aggregate(pipeline);
```

```
db.payments.explain("executionStats").aggregate(pipeline);
```

Expected Results

Twelve documents should be returned, corresponding to the original twelve source documents, but this time with the `payment_date` field converted from text values to proper date typed values, as shown below:

```
[
  {
    account: '010101',
    payment_date: ISODate('2020-01-01T01:01:01.123Z'),
    amount: 1.01
  },
  {
    account: '020202',
    payment_date: ISODate('2020-02-02T02:02:02.456Z'),
    amount: 2.02
  },
  {
    account: '030303',
    payment_date: ISODate('2020-03-03T03:03:03.789Z'),
    amount: 3.03
  },
  {
    account: '040404',
    payment_date: ISODate('2020-04-04T04:04:04.012Z'),
    amount: 4.04
  },
  {
    account: '050505',
    payment_date: ISODate('2020-05-05T05:05:05.345Z'),
    amount: 5.05
  },
  {
    account: '060606',
    payment_date: ISODate('2020-06-06T06:06:06.678Z'),
    amount: 6.06
  },
  {
    account: '070707',
    payment_date: ISODate('2020-07-07T07:07:07.901Z'),
    amount: 7.07
  },
  {
    account: '080808',
    payment_date: ISODate('2020-08-08T08:08:08.234Z'),
    amount: 8.08
  },
  {
    account: '090909',
    payment_date: ISODate('2020-09-09T09:09:09.567Z'),
    amount: 9.09
  },
  {
    account: '101010',
    payment_date: ISODate('2020-10-10T10:10:10.890Z'),
    amount: 10.1
  },
  {
    account: '111111',
    payment_date: ISODate('2020-11-11T11:11:11.111Z'),
```

```
    amount: 11.11
  },
  {
    account: '121212',
    payment_date: ISODate('2020-12-12T12:12:12.999Z'),
    amount: 12.12
  }
]
```

Observations

- **Concatenation Explanation.** In this pipeline, the text fields (e.g. `'12-DEC-20 12.12.12.999000000'`) are each converted to date fields (e.g. `2020-12-12T12:12:12.999Z`). This is achieved by concatenating together the following four example elements before passing them to the `$dateFromString` operator to convert to a date type:
  - `'12-'` *(day of the month from the input string + the hyphen suffix already present in the text)*
  - `'12'` *(replacing 'DEC')*
  - `'-20'` *(hard-coded hyphen + hardcoded century)*
  - `'20 12.12.12.999'` *(the rest of input string apart from the last 6 nanosecond digits)*
- **Further Reading.** This example is based on the output of the blog post: Converting Gnarly Date Strings to Proper Date Types Using a MongoDB Aggregation Pipeline.

Trend Analysis Examples

This section provides examples of analysing data sets to identify different trends, categorisations and relationships.

Faceted Classification

Minimum MongoDB Version: 4.2

Scenario

You want to provide a faceted search capability on your retail website to enable customers to refine their product search by selecting specific characteristics against the product results listed in the web page. It is beneficial to classify the products by different dimensions, where each dimension, or facet, corresponds to a particular field in a product record (e.g. *product rating*, *product price*). Each facet should be broken down into sub-ranges so that a customer can select a specific sub-range (*4 - 5 stars*) for a particular facet (e.g. *rating*). The aggregation pipeline will analyse the *products* collection by each facet's field (*rating* and *price*) to determine each facet's spread of values.

Sample Data Population

Drop any old version of the database (if it exists) and then populate a new `products` collection with 16 documents (the database commands have been split in two to enable your clipboard to hold all the text - ensure you copy and execute each of the two sections):

-Part 1-

```
use book-faceted-classfctn;
db.dropDatabase();

// Insert first 8 records into the collection
db.products.insertMany([
  {
    "name": "Asus Laptop",
    "category": "ELECTRONICS",
    "description": "Good value laptop for students",
    "price": NumberDecimal("431.43"),
    "rating": NumberDecimal("4.2"),
  },
  {
    "name": "The Day Of The Triffids",
    "category": "BOOKS",
    "description": "Classic post-apocalyptic novel",
    "price": NumberDecimal("5.01"),
    "rating": NumberDecimal("4.8"),
  },
  {
    "name": "Morphy Richardds Food Mixer",
    "category": "KITCHENWARE",
    "description": "Luxury mixer turning good cakes into great",
    "price": NumberDecimal("63.13"),
    "rating": NumberDecimal("3.8"),
  },
  {
    "name": "Karcher Hose Set",
    "category": "GARDEN",
    "description": "Hose + nosels + winder for tidy storage",
    "price": NumberDecimal("22.13"),
    "rating": NumberDecimal("4.3"),
  },
  {
    "name": "Oak Coffee Table",
    "category": "HOME",
    "description": "size is 2m x 0.5m x 0.4m",
    "price": NumberDecimal("22.13"),
    "rating": NumberDecimal("3.8"),
  },
  {
    "name": "Lenovo Laptop",
    "category": "ELECTRONICS",
    "description": "High spec good for gaming",
    "price": NumberDecimal("1299.99"),
    "rating": NumberDecimal("4.1"),
  },
  {
    "name": "One Day in the Life of Ivan Denisovich",
    "category": "BOOKS",
    "description": "Brutal life in a labour camp",
    "price": NumberDecimal("4.29"),
    "rating": NumberDecimal("4.9"),
  },
```

```
  {
    "name": "Russell Hobbs Chrome Kettle",
    "category": "KITCHENWARE",
    "description": "Nice looking budget kettle",
    "price": NumberDecimal("15.76"),
    "rating": NumberDecimal("3.9"),
  },
]);
```

-Part 2-

```
// Insert second 8 records into the collection
db.products.insertMany([
  {
    "name": "Tiffany Gold Chain",
    "category": "JEWELERY",
    "description": "Looks great for any age and gender",
    "price": NumberDecimal("582.22"),
    "rating": NumberDecimal("4.0"),
  },
  {
    "name": "Raleigh Racer 21st Century Classic",
    "category": "BICYCLES",
    "description": "Modern update to a classic 70s bike design",
    "price": NumberDecimal("523.00"),
    "rating": NumberDecimal("4.5"),
  },
  {
    "name": "Diesel Flare Jeans",
    "category": "CLOTHES",
    "description": "Top end casual look",
    "price": NumberDecimal("129.89"),
    "rating": NumberDecimal("4.3"),
  },
  {
    "name": "Jazz Silk Scarf",
    "category": "CLOTHES",
    "description": "Style for the winder months",
    "price": NumberDecimal("28.39"),
    "rating": NumberDecimal("3.7"),
  },
  {
    "name": "Dell XPS 13 Laptop",
    "category": "ELECTRONICS",
    "description": "Developer edition",
    "price": NumberDecimal("1399.89"),
    "rating": NumberDecimal("4.4"),
  },
  {
    "name": "NY Baseball Cap",
    "category": "CLOTHES",
    "description": "Blue & white",
    "price": NumberDecimal("18.99"),
    "rating": NumberDecimal("4.0"),
  },
  {
    "name": "Tots Flower Pots",
    "category": "GARDEN",
    "description": "Set of three",
    "price": NumberDecimal("9.78"),
    "rating": NumberDecimal("4.1"),
  },
  {
    "name": "Picky Pencil Sharpener",
    "category": "Stationery",
```

```
    "description": "Ultra budget",
    "price": NumberDecimal("0.67"),
    "rating": NumberDecimal("1.2"),
  },
]);
```

Aggregation Pipeline

Define a single pipeline ready to perform the aggregation:

```
var pipeline = [
  // Group products by 2 facets: 1) by price ranges, 2) by rating ranges
  {"$facet": {

    // Group by price ranges
    "by_price": [
      // Group into 3 ranges: inexpensive small price range to expensive large
price range
      {"$bucketAuto": {
        "groupBy": "$price",
        "buckets": 3,
        "granularity": "1-2-5",
        "output": {
          "count": {"$sum": 1},
          "products": {"$push": "$name"},
        },
      }},

      // Tag range info as "price_range"
      {"$set": {
        "price_range": "$_id",
      }},

      // Omit unwanted fields
      {"$unset": [
        "_id",
      ]},
    ],

    // Group by rating ranges
    "by_rating": [
      // Group products evenly across 5 rating ranges from low to high
      {"$bucketAuto": {
        "groupBy": "$rating",
        "buckets": 5,
        "output": {
          "count": {"$sum": 1},
          "products": {"$push": "$name"},
        },
      }},

      // Tag range info as "rating_range"
      {"$set": {
        "rating_range": "$_id",
      }},

      // Omit unwanted fields
      {"$unset": [
        "_id",
      ]},
    ],
  }},
];
```

Execution

Execute the aggregation using the defined pipeline and also view its explain plan:

```
db.products.aggregate(pipeline);
```

```
db.products.explain("executionStats").aggregate(pipeline);
```

Expected Results

A single document should be returned, which contains 2 facets (keyed off `by_price` and `by_rating` respectively), where each facet shows its sub-ranges of values and the products belonging to each sub-range, as shown below:

```
[
  {
    by_price: [
      {
        count: 6,
        products: [
          'Picky Pencil Sharpener', 'One Day in the Life of Ivan Denisovich',
          'The Day Of The Triffids', 'Tots Flower Pots', 'Russell Hobbs Chrome
Kettle',
          'NY Baseball Cap'
        ],
        price_range: {
          min: NumberDecimal('0.500000000000000'), max:
NumberDecimal('20.0000000000000')
        }
      },
      {
        count: 5,
        products: [
          'Karcher Hose Set', 'Oak Coffee Table', 'Jazz Silk Scarf',
          'Morphy Richardds Food Mixer', 'Diesel Flare Jeans'
        ],
        price_range: {
          min: NumberDecimal('20.0000000000000'), max:
NumberDecimal('200.0000000000000')
        }
      },
      {
        count: 5,
        products: [
          'Asus Laptop', 'Raleigh Racer 21st Century Classic', 'Tiffany Gold
Chain',
          'Lenovo Laptop', 'Dell XPS 13 Laptop'
        ],
        price_range: {
          min: NumberDecimal('200.0000000000000'), max:
NumberDecimal('2000.0000000000000')
        }
      }
    ],
    by_rating: [
      {
        count: 4,
        products: [
          'Picky Pencil Sharpener', 'Jazz Silk Scarf', 'Morphy Richardds Food
Mixer',
          'Oak Coffee Table'
        ],
        rating_range: { min: NumberDecimal('1.2'), max: NumberDecimal('3.9') }
      },
      {
        count: 3,
        products: [
          'Russell Hobbs Chrome Kettle', 'Tiffany Gold Chain', 'NY Baseball Cap'
```

```
        ],
        rating_range: { min: NumberDecimal('3.9'), max: NumberDecimal('4.1') }
      },
      {
        count: 3,
        products: [ 'Lenovo Laptop', 'Tots Flower Pots', 'Asus Laptop' ],
        rating_range: { min: NumberDecimal('4.1'), max: NumberDecimal('4.3') }
      },
      {
        count: 3,
        products: [
          'Karcher Hose Set', 'Diesel Flare Jeans', 'Dell XPS 13 Laptop'
        ],
        rating_range: { min: NumberDecimal('4.3'), max: NumberDecimal('4.5') }
      },
      {
        count: 3,
        products: [
          'Raleigh Racer 21st Century Classic', 'The Day Of The Triffids',
          'One Day in the Life of Ivan Denisovich'
        ],
        rating_range: { min: NumberDecimal('4.5'), max: NumberDecimal('4.9') }
      }
    ]
  }
]
```

Observations

- **Multiple Pipelines.** The `$facet` stage doesn't have to be employed for you to use the `$bucketAuto` stage. In most *faceted search* scenarios, you will want to understand a collection by multiple dimensions at once (*price* & *rating* in this case). The `$facet` stage is convenient because it allows you to define various `$bucketAuto` dimensions in one go in a single pipeline. Otherwise, a client application must invoke an aggregation multiple times, each using a new `$bucketAuto` stage to process a different field. In fact, each section of a `$facet` stage is just a regular aggregation [sub-]pipeline, able to contain any type of stage (with a few specific documented exceptions) and may not even contain `$bucketAuto` or `$bucket` stages at all.

- **Single Document Result.** If the result of a `$facet` based aggregation is allowed to be multiple documents, this will cause a problem. The results will contain a mix of records originating from different facets but with no way of ascertaining the facet each result record belongs to. Consequently, when using `$facet`, a single document is always returned, containing top-level fields identifying each facet. Having only a single result record is not usually a problem. A typical requirement for faceted search is to return a

small amount of grouped summary data about a collection rather than large amounts of raw data from the collection. Therefore the 16MB document size limit should not be an issue.

- **Spread Of Ranges.** In this example, each of the two employed bucketing facets uses a different granularity number scheme for spreading out the sub-ranges of values. You choose a numbering scheme based on what you know about the nature of the facet. For instance, most of the *ratings* values in the sample collection have scores bunched between late 3s and early 4s. If a numbering scheme is defined to reflect an even spread of ratings, most products will appear in the same sub-range bucket and some sub-ranges would contain no products (e.g. ratings 2 to 3 in this example). This wouldn't provide website customers with much selectivity on product ratings.

Largest Graph Network

Minimum MongoDB Version: 4.2

Scenario

Your organisation wants to know the best targets for a new marketing campaign based on a social network database similar to *Twitter*. You want to search the collection of social network users, each holding a user's name and the names of other people who follow them. You will execute an aggregation pipeline that walks each user record's `followed_by` array to determine which user has the largest *network reach*.

Note this example uses a simple data model for brevity. However, this is unlikely to be an optimum data model for using `$graphLookup` at scale for social network users with many followers or running in a sharded environment. For more guidance on such matters, see this reference application: Socialite

Sample Data Population

Drop any old version of the database (if it exists) and then populate a new `users` collection with 10 social network users documents, plus an index to help optimise the *graph traversal*:

```
use book-largest-graph-network;
db.dropDatabase();

// Create index on field which each graph traversal hop will connect to
db.users.createIndex({"name": 1});

// Insert records into the users collection
db.users.insertMany([
  {"name": "Paul", "followed_by": []},
  {"name": "Toni", "followed_by": ["Paul"]},
  {"name": "Janet", "followed_by": ["Paul", "Toni"]},
  {"name": "David", "followed_by": ["Janet", "Paul", "Toni"]},
  {"name": "Fiona", "followed_by": ["David", "Paul"]},
  {"name": "Bob", "followed_by": ["Janet"]},
  {"name": "Carl", "followed_by": ["Fiona"]},
  {"name": "Sarah", "followed_by": ["Carl", "Paul"]},
  {"name": "Carol", "followed_by": ["Helen", "Sarah"]},
  {"name": "Helen", "followed_by": ["Paul"]},
]);
```

Aggregation Pipeline

Define a single pipeline ready to perform the aggregation:

```
var pipeline = [
  // For each social network user, graph traverse their 'followed_by' list of
people
  {"$graphLookup": {
    "from": "users",
    "startWith": "$followed_by",
    "connectFromField": "followed_by",
    "connectToField": "name",
    "depthField": "depth",
    "as": "extended_network",
  }},

  // Add new accumulating fields
  {"$set": {
    // Count the extended connection reach
    "network_reach": {
      "$size": "$extended_network"
    },

    // Gather the list of the extended connections' names
    "extended_connections": {
      "$map": {
        "input": "$extended_network",
        "as": "connection",
        "in": "$$connection.name", // Just get name field from each array element
      }
    },
  }},

  // Omit unwanted fields
  {"$unset": [
    "_id",
    "followed_by",
    "extended_network",
  ]},

  // Sort by person with greatest network reach first, in descending order
  {"$sort": {
    "network_reach": -1,
  }},
];
```

Execution

Execute the aggregation using the defined pipeline and also view its explain plan:

```
db.users.aggregate(pipeline);
```

```
db.users.explain("executionStats").aggregate(pipeline);
```

Expected Results

Ten documents should be returned, corresponding to the original ten source social network users, with each one including a count of the user's *network reach*, and the names of their *extended connections*, sorted by the user with the most extensive network reach first, as shown below:

```
[
  {
    name: 'Carol',
    network_reach: 8,
    extended_connections: [ 'David', 'Toni', 'Fiona', 'Sarah', 'Helen', 'Carl',
'Paul',  'Janet' ]
  },
  {
    name: 'Sarah',
    network_reach: 6,
    extended_connections: [ 'David', 'Toni', 'Fiona', 'Carl', 'Paul', 'Janet' ]
  },
  {
    name: 'Carl',
    network_reach: 5,
    extended_connections: [ 'David', 'Toni', 'Fiona', 'Paul', 'Janet' ]
  },
  {
    name: 'Fiona',
    network_reach: 4,
    extended_connections: [ 'David', 'Toni', 'Paul', 'Janet' ]
  },
  {
    name: 'David',
    network_reach: 3,
    extended_connections: [ 'Toni', 'Paul', 'Janet' ]
  },
  {
    name: 'Bob',
    network_reach: 3,
    extended_connections: [ 'Toni', 'Paul', 'Janet' ]
  },
  {
    name: 'Janet',
    network_reach: 2,
    extended_connections: [ 'Toni', 'Paul' ]
  },
  {
    name: 'Toni',
    network_reach: 1,
    extended_connections: [ 'Paul']
  },
  {
    name: 'Helen',
    network_reach: 1,
    extended_connections: [ 'Paul' ]
  },
  { name: 'Paul',
    network_reach: 0,
    extended_connections: []
  }
]
```

Observations

- **Following Graphs.** The `$graphLookup` stage helps you traverse relationships between records, looking for patterns that aren't necessarily evident from looking at each record in isolation. In this example, by looking at *Paul's* record in isolation, it is evident that *Paul* has no *friends* and thus has the lowest network reach. However, it is not obvious that *Carol* has the greatest network reach just by looking at the number of people *Carol* is directly followed by, which is two. *David*, for example, is followed by three people (one more than *Carol*). However, the executed aggregation pipeline can deduce that *Carol* has the most extensive network reach.
- **Index Use.** The `$graphLookup` stage can leverage the index on the field `name` for each of its `connectToField` hops.
- **Extracting One Field From Each Array Element.** The pipeline uses the `$map` array operator to only take one field from each *user* element matched by the `$graphLookup` stage. The `$map` logic loops through each matched *user*, adding the value of the user's `name` field to the `$map`'s array of results and ignoring the other field (`followed_by`). For more information about using the `$map` operator, see the Advanced Use Of Expressions For Array Processing chapter.

Incremental Analytics

Minimum MongoDB Version: 4.2

Scenario

You have a set of *shop orders* accumulated over many years, with the retail channel adding new order records continuously to the *orders* collection throughout each trading day. You want to frequently generate a summary report so management can understand the state of the business and react to changing business trends. Over the years, it takes increasingly longer to generate the report of all daily sums and averages because there is increasingly more days' worth of data to process. From now on, to address this problem, you will only generate each new day's summary analysis at the end of the day and store it in a different collection which accumulates the daily summary records over time.

Unlike most examples in this book, the aggregation pipeline writes its output to a collection rather than streaming the results back to the calling application.

Sample Data Population

Drop any old version of the database (if it exists) and then add 9 documents to the `orders` collection representing 5 orders for 01-Feb-2021 and 4 orders for 02-Feb-2021:

```
use book-incremental-analytics;
db.dropDatabase();

// Create index for a daily_orders_summary collection
db.daily_orders_summary.createIndex({"day": 1}, {"unique": true});

// Create index for a orders collection
db.orders.createIndex({"orderdate": 1});

// Insert records into the orders collection
// (5 orders for 1st Feb, 4 orders for 2nd Feb)
db.orders.insertMany([
  {
    "orderdate": ISODate("2021-02-01T08:35:52Z"),
    "value": NumberDecimal("231.43"),
  },
  {
    "orderdate": ISODate("2021-02-01T09:32:07Z"),
    "value": NumberDecimal("99.99"),
  },
  {
    "orderdate": ISODate("2021-02-01T08:25:37Z"),
    "value": NumberDecimal("63.13"),
  },
  {
    "orderdate": ISODate("2021-02-01T19:13:32Z"),
    "value": NumberDecimal("2.01"),
  },
  {
    "orderdate": ISODate("2021-02-01T22:56:53Z"),
    "value": NumberDecimal("187.99"),
  },
  {
    "orderdate": ISODate("2021-02-02T23:04:48Z"),
    "value": NumberDecimal("4.59"),
  },
  {
    "orderdate": ISODate("2021-02-02T08:55:46Z"),
    "value": NumberDecimal("48.50"),
  },
  {
    "orderdate": ISODate("2021-02-02T07:49:32Z"),
    "value": NumberDecimal("1024.89"),
  },
  {
    "orderdate": ISODate("2021-02-02T13:49:44Z"),
    "value": NumberDecimal("102.24"),
  },
]);
```

Aggregation Pipeline

Define a function to create a pipeline, but with the start and end date parameterised, ready to be used to perform the aggregation multiple times, for different days:

```
function getDayAggPipeline(startDay, endDay) {
  return [
    // Match orders for one day only
    {"$match": {
      "orderdate": {
        "$gte": ISODate(startDay),
        "$lt": ISODate(endDay),
      }
    }},

    // Group all orders together into one summary record for the day
    {"$group": {
      "_id": null,
      "date_parts": {"$first": {"$dateToParts": {"date": "$orderdate"}}},
      "total_value": {"$sum": "$value"},
      "total_orders": {"$sum": 1},
    }},

    // Get date parts from 1 order (need year+month+day, for UTC)
    {"$set": {
      "day": {
        "$dateFromParts": {
          "year": "$date_parts.year",
          "month": "$date_parts.month",
          "day":"$date_parts.day"
       }
     },
    }},

    // Omit unwanted field
    {"$unset": [
      "_id",
      "date_parts",
    ]},

    // Add day summary to summary collection (overwrite if already exists)
    {"$merge": {
      "into": "daily_orders_summary",
      "on": "day",
      "whenMatched": "replace",
      "whenNotMatched": "insert"
    }},
  ];
}
```

Execution

For 01-Feb-2021 orders only, build the pipeline and execute the aggregation:

```
// Get the pipeline for the 1st day
var pipeline = getDayAggPipeline("2021-02-01T00:00:00Z", "2021-02-02T00:00:00Z");

// Run aggregation for 01-Feb-2021 orders & put result in summary collection
db.orders.aggregate(pipeline);

// View the summary collection content (should be 1 record only)
db.daily_orders_summary.find()
```

From the results, you can see that only a single order summary has been generated, for 01-Feb-2021, containing the total value and number of orders for that day.

Now for the next day only (for 02-Feb-2021 orders), build the pipeline and execute the aggregation:

```
// Get the pipeline for the 2nd day
var pipeline = getDayAggPipeline("2021-02-02T00:00:00Z", "2021-02-03T00:00:00Z");

// Run aggregation for 02-Feb-2021 orders & put result in summary collection
db.orders.aggregate(pipeline);

// View the summary collection content (should be 2 record now)
db.daily_orders_summary.find()
```

From the results, you can see that order summaries exist for both days.

To simulate the organisation's occasional need to correct an old order retrospectively, go back and add a new "high value" order for the first day. Then re-run the aggregation for the first day only (01-Feb-2021) to show that you can safely and correctly recalculate the summary for just one day:

```
// Retrospectively add an order to an older day (01-Feb-2021)
db.orders.insertOne(
  {
    "orderdate": ISODate("2021-02-01T09:32:07Z"),
    "value": NumberDecimal("11111.11"),
  },
)

// Get the pipeline for the 1st day again
var pipeline = getDayAggPipeline("2021-02-01T00:00:00Z", "2021-02-02T00:00:00Z");

// Re-run aggregation for 01-Feb-2021 overwriting 1st record in summary
collections
db.orders.aggregate(pipeline);

// View the summary collection content (should still be 2 records but 1st changed)
db.daily_orders_summary.find()
```

From the results, you can see that two order summaries still exist, one for each of the two trading days, but the total value and order count for the first day has changed.

For completeness, also view the explain plan for the aggregation pipeline:

```
db.products.explain("executionStats").aggregate(pipeline);
```

Expected Results

The content of the `daily_orders_summary` collection after running the aggregation for just the 1st day should be similar to below:

```
[
  {
    _id: ObjectId('6062102e7eeb772e6ca96bc7'),
    total_value: NumberDecimal('584.55'),
    total_orders: 5,
    day: ISODate('2021-02-01T00:00:00.000Z')
  }
]
```

The content of the `daily_orders_summary` collection after running the aggregation for the 2nd day should be similar to below:

```
[
  {
    _id: ObjectId('6062102e7eeb772e6ca96bc7'),
    total_value: NumberDecimal('584.55'),
    total_orders: 5,
    day: ISODate('2021-02-01T00:00:00.000Z')
  },
  {
    _id: ObjectId('606210377eeb772e6ca96bcc'),
    total_value: NumberDecimal('1180.22'),
    total_orders: 4,
    day: ISODate('2021-02-02T00:00:00.000Z')
  }
]
```

After re-running the aggregation for the 1st day following the addition of the missed order, the content of the `daily_orders_summary` collection should be similar to below (notice the first record now shows a value of one greater than before for `total_orders`, and for `total_value` the value is now significantly higher):

```
[
  {
    _id: ObjectId('6062102e7eeb772e6ca96bc7'),
    total_value: NumberDecimal('11695.66'),
    total_orders: 6,
    day: ISODate('2021-02-01T00:00:00.000Z')
  },
  {
    _id: ObjectId('606210377eeb772e6ca96bcc'),
    total_value: NumberDecimal('1180.22'),
    total_orders: 4,
    day: ISODate('2021-02-02T00:00:00.000Z')
  }
]
```

Observations

- **Merging Results.** The pipeline uses a `$merge` stage to instruct the aggregation engine to write the output to a collection rather than returning a stream of results. In this example, with the options you provide to `$merge`, the aggregation inserts a new record in the destination collection if a matching one doesn't already exist. If a matching record already exists, it replaces the previous version.

- **Incremental Updates.** The example illustrates just two days of shop orders, albeit with only a few orders, to keep the example simple. At the end of each new trading day, you run the aggregation pipeline to generate the current day's summary only. Even after the

source collection has increased in size over many years, the time it takes you to bring the summary collection up to date again stays constant. In a real-world scenario, the business might expose a graphical chart showing the changing daily orders trend over the last rolling year. This charting dashboard is not burdened by the cost of periodically regenerating values for all days in the year. There could be hundreds of thousands of orders received per day for real-world retailers, especially large ones. A day's summary may take many seconds to generate in that situation. Without an *incremental analytics* approach, if you need to generate a year's worth of daily summaries every time, it would take hours to refresh the business dashboard.

- **Idempotency.** If a retailer is aggregating tens of thousands of orders per day, then during end-of-day processing, it may choose to generate 24 hourly summary records rather than a single daily record. This provides the business with finer granularity to understand trends better. As with any software process, when generating hourly results into the summary collection, there is the risk of not fully completing if a system failure occurs. If an in-flight aggregation terminates abnormally, it may not have written all 24 summary collection records. The failure leaves the summary collection in an indeterminate and incomplete state for one of its days. However, this isn't a problem because of the way the aggregation pipeline uses the `$merge` stage. When an aggregation fails to complete, it can just be re-run. When re-run, it will regenerate all the results for the day, replacing existing summary records and filling in the missing ones. The aggregation pipeline is idempotent, and you can run it repeatedly without damaging the summary collection. The overall solution is self-healing and naturally tolerant of inadvertently aborted aggregation jobs.

- **Retrospective Changes.** Sometimes, an organisation may need to go back and correct records from the past, as illustrated in this example. For instance, a bank may need to fix a past payment record due to a settlement issue that only comes to light weeks later. With the approach used in this example, it is straightforward to re-execute the aggregation pipeline for a prior date, using the updated historical data. This will correctly update the specific day's summary data only, to reflect the business's current state.

Securing Data Examples

This section provides examples of how aggregation pipelines can assist with the way data is accessed and distributed securely.

Restricted View

Minimum MongoDB Version: 4.2

Scenario

You have a *persons* collection, where a particular client application shouldn't be allowed to see sensitive information. Consequently, you will provide a read-only view of a filtered subset of peoples' data only. In a real-world situation, you would also use MongoDB's Role-Based Access Control (RBAC) to limit the client application to only be able to access the view and not the original collection. You will use the view (named *adults*) to restrict the personal data for the client application in two ways:

1. Only show people aged 18 and over (by checking each person's `dateofbirth` field)
2. Exclude each person's `social_security_num` field from results

Essentially, this is an illustration of achieving "record-level" access control in MongoDB.

Sample Data Population

Drop any old version of the database (if it exists), create an index and populate the new `persons` collections with 5 records:

```
use book-restricted-view;
db.dropDatabase();

// Create index for a persons collection
db.persons.createIndex({"dateofbirth": -1});

// Create index for non-$expr part of filter in MongoDB version < 5.0
db.persons.createIndex({"gender": 1});

// Create index for combination of $expr & non-$expr filter in MongoDB version >=
5.0
db.persons.createIndex({"gender": 1, "dateofbirth": -1});

// Insert records into the persons collection
db.persons.insertMany([
  {
    "person_id": "6392529400",
    "firstname": "Elise",
    "lastname": "Smith",
    "dateofbirth": ISODate("1972-01-13T09:32:07Z"),
    "gender": "FEMALE",
    "email": "elise_smith@myemail.com",
    "social_security_num": "507-28-9805",
    "address": {
        "number": 5625,
        "street": "Tipa Circle",
        "city": "Wojzinmoj",
    },
  },
  {
    "person_id": "1723338115",
    "firstname": "Olive",
    "lastname": "Ranieri",
    "dateofbirth": ISODate("1985-05-12T23:14:30Z"),
    "gender": "FEMALE",
    "email": "oranieri@warmmail.com",
    "social_security_num": "618-71-2912",
    "address": {
        "number": 9303,
        "street": "Mele Circle",
        "city": "Tobihbo",
    },
  },
  {
    "person_id": "8732762874",
    "firstname": "Toni",
    "lastname": "Jones",
    "dateofbirth": ISODate("2014-11-23T16:53:56Z"),
    "gender": "FEMALE",
    "email": "tj@wheresmyemail.com",
    "social_security_num": "001-10-3488",
    "address": {
        "number": 1,
        "street": "High Street",
```

```
        "city": "Upper Abbeywoodington",
    },
  },
  {
    "person_id": "7363629563",
    "firstname": "Bert",
    "lastname": "Gooding",
    "dateofbirth": ISODate("1941-04-07T22:11:52Z"),
    "gender": "MALE",
    "email": "bgooding@tepidmail.com",
    "social_security_num": "230-43-7633",
    "address": {
        "number": 13,
        "street": "Upper Bold Road",
        "city": "Redringtonville",
    },
  },
  {
    "person_id": "1029648329",
    "firstname": "Sophie",
    "lastname": "Celements",
    "dateofbirth": ISODate("2013-07-06T17:35:45Z"),
    "gender": "FEMALE",
    "email": "sophe@celements.net",
    "social_security_num": "377-30-5364",
    "address": {
        "number": 5,
        "street": "Innings Close",
        "city": "Basilbridge",
    },
  },
]);
```

Aggregation Pipeline

Define a single pipeline ready to perform the aggregation:

```
var pipeline = [
  // Filter out any persons aged under 18 ($expr required to reference '$$NOW')
  {"$match":
    {"$expr":{
      "$lt": ["$dateofbirth", {"$subtract": ["$$NOW", 18*365.25*24*60*60*1000]}]
    }},
  },

  // Exclude fields to be filtered out by the view
  {"$unset": [
    "_id",
    "social_security_num",
  ]},
];
```

Execution

Firstly, to test the defined aggregation pipeline (before using it to create a view), execute the aggregation for the pipeline and also observe its explain plan:

```
db.persons.aggregate(pipeline);
```

```
db.persons.explain("executionStats").aggregate(pipeline);
```

Now create the new *adults* view, which will automatically apply the aggregation pipeline whenever anyone queries the view:

```
db.createView("adults", "persons", pipeline);
```

Execute a regular MQL query against the view, without any filter criteria, and also observe its explain plan:

```
db.adults.find();
```

```
db.adults.explain("executionStats").find();
```

Execute a MQL query against the view, but this time with a filter to return only adults who are female, and again observe its explain plan to see how the `gender` filter affects the plan:

```
db.adults.find({"gender": "FEMALE"});
```

```
db.adults.explain("executionStats").find({"gender": "FEMALE"});
```

Expected Results

The result for both the `aggregate()` command and the `find()` executed on the *view* should be the same, with three documents returned, representing the three persons who are over 18 but not showing their actual dates of birth, as shown below:

```
[
  {
    person_id: '6392529400',
    firstname: 'Elise',
    lastname: 'Smith',
    dateofbirth: ISODate('1972-01-13T09:32:07.000Z'),
    gender: 'FEMALE',
    email: 'elise_smith@myemail.com',
    address: { number: 5625, street: 'Tipa Circle', city: 'Wojzinmoj' }
  },
  {
    person_id: '1723338115',
    firstname: 'Olive',
    lastname: 'Ranieri',
    dateofbirth: ISODate('1985-05-12T23:14:30.000Z'),
    gender: 'FEMALE',
    email: 'oranieri@warmmail.com',
    address: { number: 9303, street: 'Mele Circle', city: 'Tobihbo' }
  },
  {
    person_id: '7363629563',
    firstname: 'Bert',
    lastname: 'Gooding',
    dateofbirth: ISODate('1941-04-07T22:11:52.000Z'),
    gender: 'MALE',
    email: 'bgooding@tepidmail.com',
    address: { number: 13, street: 'Upper Bold Road', city: 'Redringtonville' }
  }
]
```

The result of running the `find()` against the *view* with the filter `"gender": "FEMALE"` should be two females records only because the male record has been excluded, as shown below:

```
[
  {
    person_id: '6392529400',
    firstname: 'Elise',
    lastname: 'Smith',
    dateofbirth: ISODate('1972-01-13T09:32:07.000Z'),
    gender: 'FEMALE',
    email: 'elise_smith@myemail.com',
    address: { number: 5625, street: 'Tipa Circle', city: 'Wojzinmoj' }
  },
  {
    person_id: '1723338115',
    firstname: 'Olive',
    lastname: 'Ranieri',
    dateofbirth: ISODate('1985-05-12T23:14:30.000Z'),
    gender: 'FEMALE',
    email: 'oranieri@warmmail.com',
    address: { number: 9303, street: 'Mele Circle', city: 'Tobihbo' }
  }
]
```

Observations

- **Expr & Indexes.** The "NOW" system variable used here returns the current system date-time. However, you can only access this system variable via an aggregation expression and not directly via the regular MongoDB query syntax used by MQL and `$match`. You must wrap an expression using `$$NOW` inside an `$expr` operator. As described in the section *Restrictions When Using Expressions* in an earlier chapter, if you use an $expr query operator to perform a range comparison, you can't make use of an index in versions of MongoDB earlier then 5.0. Therefore, in this example, unless you use MongoDB 5.0 or greater, the aggregation will not take advantage of an index on `dateofbirth`. For a view, because you specify the pipeline earlier than it is ever run, you cannot obtain the current date-time at runtime by other means.

- **View Finds & Indexes.** Even for versions of MongoDB before 5.0, the explain plan for the *gender query* run against the view shows an index has been used (the index defined for the `gender` field). At runtime, a view is essentially just an aggregation pipeline defined "ahead of time". When `db.adults.find({"gender": "FEMALE"})` is executed, the database engine dynamically appends a new `$match` stage to the end of the pipeline for the gender match. It then optimises the pipeline by moving the new `$match` stage to the pipeline's start. Finally, it adds the filter extracted from the new `$match` stage to the aggregation's initial query, and hence it can then leverage an index containing the `gender` field. The following two excerpts, from an explain plan from a MongoDB version before

5.0, illustrate how the filter on `gender` and the filter on `dateofbirth` combine at runtime and how the index for `gender` is used to avoid a full collection scan:

```
'$cursor': {
  queryPlanner: {
    plannerVersion: 1,
    namespace: 'book-restricted-view.persons',
    indexFilterSet: false,
    parsedQuery: {
      '$and': [
        { gender: { '$eq': 'FEMALE' } },
        {
          '$expr': {
            '$lt': [
              '$dateofbirth',
              {
                '$subtract': [ '$$NOW', { '$const': 568036800000 } ]
                ...
```

```
inputStage: {
  stage: 'IXSCAN',
  keyPattern: { gender: 1 },
  indexName: 'gender_1',
  direction: 'forward',
  indexBounds: { gender: [ '["FEMALE", "FEMALE"]' ] }
}
```

In MongoDB 5.0 and greater, the explain plan will show the aggregation runtime executing the pipeline more optimally by entirely using the compound index based on both the fields `gender` and `dateofbirth`.

- **Further Reading.** The ability for *find* operations on a view to automatically push filters into the view's aggregation pipeline, and then be further optimised, is described in the blog post: Is Querying A MongoDB View Optimised?

Mask Sensitive Fields

Minimum MongoDB Version: 4.4 *(due to use of $rand operator)*

Scenario

You want to perform irreversible masking on the sensitive fields of a collection of *credit card payments*, ready to provide the output data set to a 3rd party for analysis, without exposing sensitive information to the 3rd party. The specific changes that you need to make to the payments' fields are:

- Partially obfuscate the carder holder's name
- Obfuscate the first 12 digits of the card's number, retaining only the final 4 digits
- Adjust the card's expiry date-time by adding or subtracting a random amount up to a maximum of 30 days (~1 month)
- Replace the card's 3 digit security code with a random set of 3 digits
- Adjust the transaction's amount by adding or subtracting a random amount up to a maximum of 10% of the original amount
- Change the `reported` field's boolean value to the opposite value for roughly 20% of the records
- If the embedded `customer_info` sub-document's `category` field is set to *RESTRICTED*, exclude the whole `customer_info` sub-document

Sample Data Population

Drop any old version of the database (if it exists) and then populate a new `payments` collection with 2 credit card payment documents, containing sensitive data:

```
use book-mask-sensitive-fields;
db.dropDatabase();

// Insert records into the payments collection
db.payments.insertMany([
  {
    "card_name": "Mrs. Jane A. Doe",
    "card_num": "1234567890123456",
    "card_expiry": ISODate("2023-08-31T23:59:59Z"),
    "card_sec_code": "123",
    "card_type": "CREDIT",
    "transaction_id": "eb1bd77836e8713656d9bf2debba8900",
    "transaction_date": ISODate("2021-01-13T09:32:07Z"),
    "transaction_amount": NumberDecimal("501.98"),
    "reported": false,
    "customer_info": {
      "category": "RESTRICTED",
      "rating": 89,
      "risk": 3,
    },
  },
  {
    "card_name": "Jim Smith",
    "card_num": "9876543210987654",
    "card_expiry": ISODate("2022-12-31T23:59:59Z"),
    "card_sec_code": "987",
    "card_type": "DEBIT",
    "transaction_id": "634c416a6fbcf060bb0ba90c4ad94f60",
    "transaction_date": ISODate("2020-11-24T19:25:57Z"),
    "transaction_amount": NumberDecimal("64.01"),
    "reported": true,
    "customer_info": {
      "category": "NORMAL",
      "rating": 78,
      "risk": 55,
    },
  },
]);
```

Aggregation Pipeline

Define a single pipeline ready to perform the aggregation:

```
var pipeline = [
  // Replace a subset of fields with new values
  {"$set": {
    // Extract the last word from the name , eg: 'Doe' from 'Mrs. Jane A. Doe'
    "card_name": {"$regexFind": {"input": "$card_name", "regex": /(\S+)$/}},

    // Mask card num 1st part retaining last 4 chars, eg: '1234567890123456' ->
'XXXXXXXXXXXX3456'
    "card_num": {"$concat": [
                  "XXXXXXXXXXXX",
                  {"$substrCP": ["$card_num", 12, 4]},
                ]},

    // Add/subtract a random time amount of a maximum of 30 days (~1 month) each-
way
    "card_expiry": {"$add": [
                     "$card_expiry",
                     {"$floor": {"$multiply": [{"$subtract": [{"$rand": {}},
0.5]}, 2*30*24*60*60*1000]}},
                   ]},

    // Replace each digit with random digit, eg: '133' -> '472'
    "card_sec_code": {"$concat": [
                       {"$toString": {"$floor": {"$multiply": [{"$rand": {}},
10]}}},
                       {"$toString": {"$floor": {"$multiply": [{"$rand": {}},
10]}}},
                       {"$toString": {"$floor": {"$multiply": [{"$rand": {}},
10]}}},
                     ]},

    // Add/subtract a random percent of the amount's value up to 10% maximum each-
way
    "transaction_amount": {"$add": [
                            "$transaction_amount",
                            {"$multiply": [{"$subtract": [{"$rand": {}}, 0.5]},
0.2, "$transaction_amount"]},
                           ]},

    // Retain field's bool value 80% of time on average, setting to the opposite
value 20% of time
    "reported": {"$cond": {
                   "if":   {"$lte": [{"$rand": {}}, 0.8]},
                   "then": "$reported",
                   "else": {"$not": ["$reported"]},
                 }},

    // Exclude sub-doc if the sub-doc's category field's value is 'RESTRICTED'
    "customer_info": {"$cond": {
                        "if":   {"$eq": ["$customer_info.category",
"RESTRICTED"]},
                        "then": "$$REMOVE",
                        "else": "$customer_info",
                      }},
```

```
    // Mark _id field to excluded from results
    "_id": "$$REMOVE",
  }},

  // Take regex matched last word from the card name and prefix it with hardcoded
value
  {"$set": {
    "card_name": {"$concat": ["Mx. Xxx ", {"$ifNull": ["$card_name.match",
"Anonymous"]}]},
  }},
];
```

Execution

Execute the aggregation using the defined pipeline and also view its explain plan:

```
db.payments.aggregate(pipeline);
```

```
db.payments.explain("executionStats").aggregate(pipeline);
```

Expected Results

Two documents should be returned, corresponding to the original two source documents, but this time with many of their fields redacted and obfuscated, plus the `customer_info` embedded document omitted for one record due to it having been marked as `RESTRICTED`, as shown below:

```
[
  {
    card_name: 'Mx. Xxx Doe',
    card_num: 'XXXXXXXXXXXX3456',
    card_expiry: ISODate('2023-08-31T23:29:46.460Z'),
    card_sec_code: '295',
    card_type: 'CREDIT',
    transaction_id: 'eb1bd77836e8713656d9bf2debba8900',
    transaction_date: ISODate('2021-01-13T09:32:07.000Z'),
    transaction_amount: NumberDecimal('492.4016988351474881660000000000000'),
    reported: false
  },
  {
    card_name: 'Mx. Xxx Smith',
    card_num: 'XXXXXXXXXXXX7654',
    card_expiry: ISODate('2023-01-01T00:34:49.330Z'),
    card_sec_code: '437',
    card_type: 'DEBIT',
    transaction_id: '634c416a6fbcf060bb0ba90c4ad94f60',
    transaction_date: ISODate('2020-11-24T19:25:57.000Z'),
    transaction_amount: NumberDecimal('58.360813374867622236000000000000000'),
    reported: false,
    customer_info: { category: 'NORMAL', rating: 78, risk: 55 }
  }
]
```

Observations

- **Targeted Redaction.** The pipeline uses a `$cond` operator to return the `$$REMOVE` marker variable if the `category` field is equal to `RESTRICTED`. This informs the aggregation engine to exclude the whole `customer_info` sub-document from the stage's output for the record. Alternatively, the pipeline could have used a `$redact` stage to achieve the same. However, `$redact` typically has to perform more processing work due to needing to check every field in the document. Hence, if a pipeline is only to redact out one specific sub-document, use the approach outlined in this example.

- **Regular Expression.** For masking the `card_name` field, a regular expression operator is used to extract the last word of the field's original value. `$regexFind` returns metadata into the stage's output records, indicating if the match succeeded and what the matched value is. Therefore, an additional `$set` stage is required later in the pipeline to extract the actual matched word from this metadata and prefix it with some hard-coded text.

- **Meaningful Insight.** Even though the pipeline is irreversibly obfuscating fields, it doesn't mean that the masked data is useless for performing analytics to gain insight. The pipeline masks some fields by fluctuating the original values by a small but limited

random percentage (e.g. `card_expiry`, `transaction_amount`), rather than replacing them with completely random values (e.g. `card_sec_code`). In such cases, if the input data set is sufficiently large, then minor variances will be equalled out. For the fields that are only varied slightly, users can derive similar trends and patterns from analysing the masked data as they would the original data.

- **Further Reading.** This example is based on the output of two blog posts: 1) MongoDB Irreversible Data Masking, and 2) MongoDB Reversible Data Masking.

Time-Series Examples

This section provides examples of aggregating time-series data, common in use cases involving financial data sets and Internet-of-Things (IoT).

IoT Power Consumption

Minimum MongoDB Version: 5.0 *(due to use of time series collections, $setWindowFields stage & $integral operator)*

Scenario

You are monitoring various air-conditioning units running in two buildings on an industrial campus. Every 30 minutes, a device in each unit sends the unit's current power consumption reading back to base, which a central database persists. You want to analyse this data to see how much energy in kilowatt-hours (kWh) each air-conditioning unit has consumed over the last hour for each reading received. Furthermore, you want to compute the total energy consumed by all the air-conditioning units combined in each building for every hour.

Sample Data Population

Drop any old version of the database (if it exists) and then populate a new `device_readings` collection with device readings spanning 3 hours of a day for air-conditioning units in two different buildings.

```
use book-iot-power-consumption;
db.dropDatabase();

// Use a time-series collection for optimal processing
// NOTE: This command can be commented out and the full example will still work
db.createCollection("device_readings", {
  "timeseries": {
    "timeField": "timestamp",
    "metaField": "deviceID",
    "granularity": "minutes"
  }
});

// Create compound index to aid performance for partitionBy & sortBy of
setWindowFields
db.device_readings.createIndex({"deviceID": 1, "timestamp": 1});

// Insert 18 records into the device readings collection
db.device_readings.insertMany([
  // 11:29am device readings
  {
    "buildingID": "Building-ABC",
    "deviceID": "UltraAirCon-111",
    "timestamp": ISODate("2021-07-03T11:29:59Z"),
    "powerKilowatts": 8,
  },
  {
    "buildingID": "Building-ABC",
    "deviceID": "UltraAirCon-222",
    "timestamp": ISODate("2021-07-03T11:29:59Z"),
    "powerKilowatts": 7,
  },
  {
    "buildingID": "Building-XYZ",
    "deviceID": "UltraAirCon-666",
    "timestamp": ISODate("2021-07-03T11:29:59Z"),
    "powerKilowatts": 10,
  },

  // 11:59am device readings
  {
    "buildingID": "Building-ABC",
    "deviceID": "UltraAirCon-222",
    "timestamp": ISODate("2021-07-03T11:59:59Z"),
    "powerKilowatts": 9,
  },
  {
    "buildingID": "Building-ABC",
    "deviceID": "UltraAirCon-111",
    "timestamp": ISODate("2021-07-03T11:59:59Z"),
    "powerKilowatts": 8,
  },
  {
    "buildingID": "Building-XYZ",
```

```
    "deviceID": "UltraAirCon-666",
    "timestamp": ISODate("2021-07-03T11:59:59Z"),
    "powerKilowatts": 11,
  },

  // 12:29pm device readings
  {
    "buildingID": "Building-ABC",
    "deviceID": "UltraAirCon-222",
    "timestamp": ISODate("2021-07-03T12:29:59Z"),
    "powerKilowatts": 9,
  },
  {
    "buildingID": "Building-ABC",
    "deviceID": "UltraAirCon-111",
    "timestamp": ISODate("2021-07-03T12:29:59Z"),
    "powerKilowatts": 9,
  },
  {
    "buildingID": "Building-XYZ",
    "deviceID": "UltraAirCon-666",
    "timestamp": ISODate("2021-07-03T12:29:59Z"),
    "powerKilowatts": 10,
  },

  // 12:59pm device readings
  {
    "buildingID": "Building-ABC",
    "deviceID": "UltraAirCon-222",
    "timestamp": ISODate("2021-07-03T12:59:59Z"),
    "powerKilowatts": 8,
  },
  {
    "buildingID": "Building-ABC",
    "deviceID": "UltraAirCon-111",
    "timestamp": ISODate("2021-07-03T12:59:59Z"),
    "powerKilowatts": 8,
  },
  {
    "buildingID": "Building-XYZ",
    "deviceID": "UltraAirCon-666",
    "timestamp": ISODate("2021-07-03T12:59:59Z"),
    "powerKilowatts": 11,
  },

  // 13:29pm device readings
  {
    "buildingID": "Building-ABC",
    "deviceID": "UltraAirCon-222",
    "timestamp": ISODate("2021-07-03T13:29:59Z"),
    "powerKilowatts": 9,
  },
  {
    "buildingID": "Building-ABC",
```

```
    "deviceID": "UltraAirCon-111",
    "timestamp": ISODate("2021-07-03T13:29:59Z"),
    "powerKilowatts": 9,
  },
  {
    "buildingID": "Building-XYZ",
    "deviceID": "UltraAirCon-666",
    "timestamp": ISODate("2021-07-03T13:29:59Z"),
    "powerKilowatts": 10,
  },

  // 13:59pm device readings
  {
    "buildingID": "Building-ABC",
    "deviceID": "UltraAirCon-222",
    "timestamp": ISODate("2021-07-03T13:59:59Z"),
    "powerKilowatts": 8,
  },
  {
    "buildingID": "Building-ABC",
    "deviceID": "UltraAirCon-111",
    "timestamp": ISODate("2021-07-03T13:59:59Z"),
    "powerKilowatts": 8,
  },
  {
    "buildingID": "Building-XYZ",
    "deviceID": "UltraAirCon-666",
    "timestamp": ISODate("2021-07-03T13:59:59Z"),
    "powerKilowatts": 11,
  },
]);
```

Aggregation Pipeline

Define a pipeline ready to perform an aggregation to calculate the energy an air-conditioning unit has consumed over the last hour for each reading received:

```
var pipelineRawReadings = [
  // Calculate each unit's energy consumed over the last hour for each reading
  {"$setWindowFields": {
    "partitionBy": "$deviceID",
    "sortBy": {"timestamp": 1},
    "output": {
      "consumedKilowattHours": {
        "$integral": {
          "input": "$powerKilowatts",
          "unit": "hour",
        },
        "window": {
          "range": [-1, "current"],
          "unit": "hour",
        },
      },
    },
  }},
];
```

Define a pipeline ready to compute the total energy consumed by all the air-conditioning units combined in each building for every hour:

```
var pipelineBuildingsSummary = [
  // Calculate each unit's energy consumed over the last hour for each reading
  {"$setWindowFields": {
    "partitionBy": "$deviceID",
    "sortBy": {"timestamp": 1},
    "output": {
      "consumedKilowattHours": {
        "$integral": {
          "input": "$powerKilowatts",
          "unit": "hour",
        },
        "window": {
          "range": [-1, "current"],
          "unit": "hour",
        },
      },
    },
  }},

  // Sort each reading by unit/device and then by timestamp
  {"$sort": {
    "deviceID": 1,
    "timestamp": 1,
  }},

  // Group readings together for each hour for each device using
  // the last calculated energy consumption field for each hour
  {"$group": {
    "_id": {
      "deviceID": "$deviceID",
      "date": {
          "$dateTrunc": {
            "date": "$timestamp",
            "unit": "hour",
          }
      },
    },
    "buildingID": {"$last": "$buildingID"},
    "consumedKilowattHours": {"$last": "$consumedKilowattHours"},
  }},

  // Sum together the energy consumption for the whole building
  // for each hour across all the units in the building
  {"$group": {
    "_id": {
      "buildingID": "$buildingID",
      "dayHour": {"$dateToString": {"format": "%Y-%m-%d  %H", "date":
"$_id.date"}},
    },
    "consumedKilowattHours": {"$sum": "$consumedKilowattHours"},
  }},

  // Sort the results by each building and then by each hourly summary
  {"$sort": {
```

```
    "_id.buildingID": 1,
    "_id.dayHour": 1,
  }},

  // Make the results more presentable with meaningful field names
  {"$set": {
    "buildingID": "$_id.buildingID",
    "dayHour": "$_id.dayHour",
    "_id": "$$REMOVE",
  }},
];
```

Execution

Execute an aggregation using the pipeline to calculate the energy an air-conditioning unit has consumed over the last hour for each reading received and also view its explain plan:

```
db.device_readings.aggregate(pipelineRawReadings);
```

```
db.device_readings.explain("executionStats").aggregate(pipelineRawReadings);
```

Execute an aggregation using the pipeline to compute the total energy consumed by all the air-conditioning units combined in each building for every hour and also view its explain plan:

```
db.device_readings.aggregate(pipelineBuildingsSummary);
```

```
db.device_readings.explain("executionStats").aggregate(pipelineBuildingsSummary);
```

Expected Results

For the pipeline to calculate the energy an air-conditioning unit has consumed over the last hour for each reading received, results like the following should be returned (redacted for brevity - only showing the first few records):

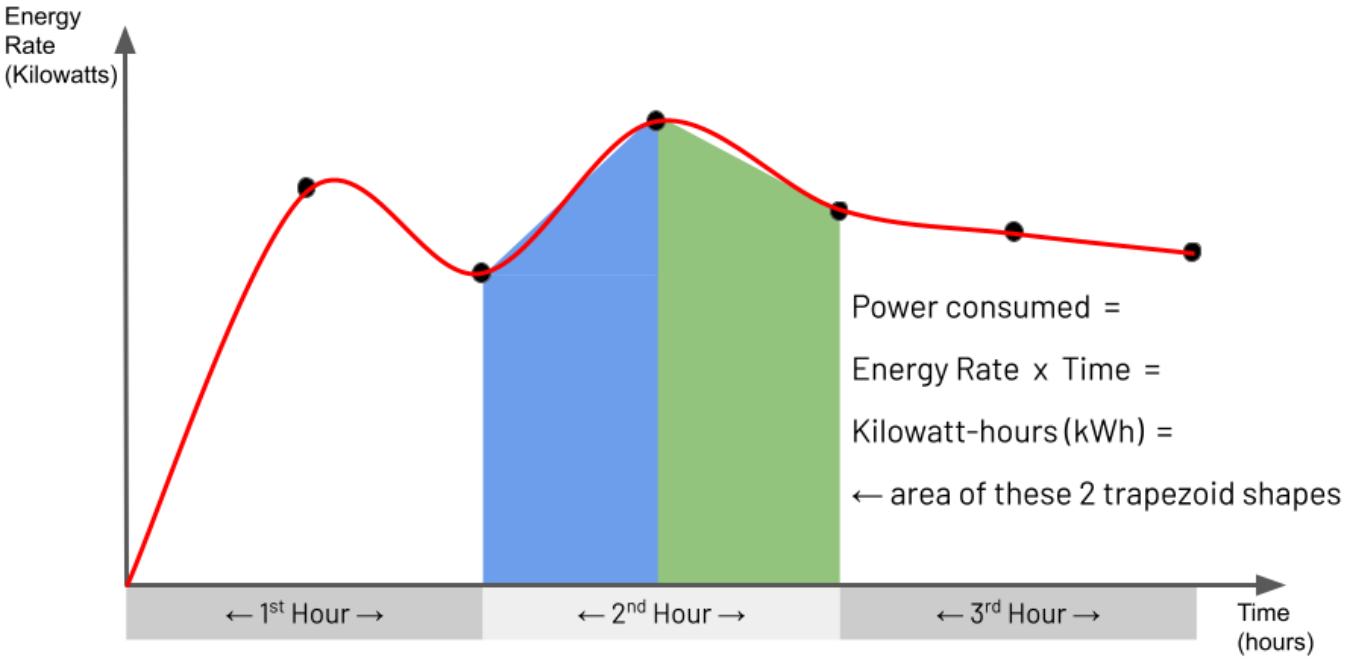
```
[
  {
    _id: ObjectId("60ed5e679ea1f9f74814ca2b"),
    buildingID: 'Building-ABC',
    deviceID: 'UltraAirCon-111',
    timestamp: ISODate("2021-07-03T11:29:59.000Z"),
    powerKilowatts: 8,
    consumedKilowattHours: 0
  },
  {
    _id: ObjectId("60ed5e679ea1f9f74814ca2f"),
    buildingID: 'Building-ABC',
    deviceID: 'UltraAirCon-111',
    timestamp: ISODate("2021-07-03T11:59:59.000Z"),
    powerKilowatts: 8,
    consumedKilowattHours: 4
  },
  {
    _id: ObjectId("60ed5e679ea1f9f74814ca32"),
    buildingID: 'Building-ABC',
    deviceID: 'UltraAirCon-111',
    timestamp: ISODate("2021-07-03T12:29:59.000Z"),
    powerKilowatts: 9,
    consumedKilowattHours: 8.25
  },
  {
    _id: ObjectId("60ed5e679ea1f9f74814ca35"),
    buildingID: 'Building-ABC',
    deviceID: 'UltraAirCon-111',
    timestamp: ISODate("2021-07-03T12:59:59.000Z"),
    powerKilowatts: 8,
    consumedKilowattHours: 8.5
  },
  {
    _id: ObjectId("60ed5e679ea1f9f74814ca38"),
    buildingID: 'Building-ABC',
    deviceID: 'UltraAirCon-111',
    timestamp: ISODate("2021-07-03T13:29:59.000Z"),
    powerKilowatts: 9,
    consumedKilowattHours: 8.5
  },
  ...
  ...
]
```

For the pipeline to compute the total energy consumed by all the air-conditioning units combined in each building for every hour, the following results should be returned:

```
[
  {
    buildingID: 'Building-ABC',
    dayHour: '2021-07-03  11',
    consumedKilowattHours: 8
  },
  {
    buildingID: 'Building-ABC',
    dayHour: '2021-07-03  12',
    consumedKilowattHours: 17.25
  },
  {
    buildingID: 'Building-ABC',
    dayHour: '2021-07-03  13',
    consumedKilowattHours: 17
  },
  {
    buildingID: 'Building-XYZ',
    dayHour: '2021-07-03  11',
    consumedKilowattHours: 5.25
  },
  {
    buildingID: 'Building-XYZ',
    dayHour: '2021-07-03  12',
    consumedKilowattHours: 10.5
  },
  {
    buildingID: 'Building-XYZ',
    dayHour: '2021-07-03  13',
    consumedKilowattHours: 10.5
  }
]
```

Observations

- **Integral Trapezoidal Rule.** As documented in the MongoDB Manual, `$integral` *"returns an approximation for the mathematical integral value, which is calculated using the trapezoidal rule"*. For non-mathematicians, this explanation may be hard to understand. You may find it easier to comprehend the behaviour of the `$integral` operator by studying the illustration below and the explanation that follows:



Essentially the trapezoidal rule determines the area of a region between two points under a graph by matching the region with a trapezoid shape that approximately fits this region and then calculating the area of this trapezoid. You can see a set of points on the illustrated graph with the matched trapezoid shape underneath each pair of points. For this IoT Power Consumption example, the points on the graph represent an air-conditioning unit's power readings captured every 30 minutes. The Y-axis is the *power rate* in Kilowatts, and the X-axis is *time* to indicate when the device captured each reading. Consequently, for this example, the energy consumed by the air-conditioning unit for a given hour's span is the area of the hour's specific section under the graph. This section's area is approximately the area of the two trapezoids shown. Using the `$integral` operator for the window of time you define in the `$setWindowFields` stage, you are asking for this approximate area to be calculated, which is the Kilowatt-hours consumed by the air-conditioning unit in one hour.

- **Window Range Definition.** For every captured document representing a device reading, this example's pipeline identifies a window of *1-hour* of previous documents relative to this *current* document. The pipeline uses this set of documents as the input for the `$integral` operator. It defines this window range in the setting `range: [-1, "current"], unit: "hour"`. The pipeline assigns the output of the `$integral` calculation to a new field called `consumedKilowattHours`.

- **One Hour Range Vs Hours Output.** The fact that the `$setWindowFields` stage in the pipeline defines `unit: "hour"` in two places may appear redundant at face value. However, this is not the case, and each serves a different purpose. As described in the previous observation, `unit: "hour"` for the `"window"` option helps dictate the size of the window of the previous number of documents to analyse. However, `unit: "hour"` for

the `$integral` operator defines that the output should be in hours ("Kilowatt-hours" in this example), yielding the result `consumedKilowattHours: 8.5` for one of the processed device readings. However, if the pipeline defined this `$integral` parameter to be `"unit": "minute"` instead, which is perfectly valid, the output value would be `510` Kilowatt-minutes (i.e. 8.5 x 60 minutes).

- **Optional Time Series Collection.** This example uses a time series collection to store sequences of device measurements over time efficiently. Employing a time series collection is optional, as shown in the `NOTE` Javascript comment in the example code. The aggregation pipeline does not need to be changed and achieves the same output if you use a regular collection instead. However, when dealing with large data sets, the aggregation will complete quicker by employing a time series collection.
- **Index for Partition By & Sort By.** In this example, you define the index `{deviceID: 1, timestamp: 1}` to optimise the use of the combination of the `partitionBy` and `sortBy` parameters in the `$setWindowFields` stage. This means that the aggregation runtime does not have to perform a slow in-memory sort based on these two fields, and it also avoids the pipeline stage memory limit of 100 MB. It is beneficial to use this index regardless of whether you employ a regular collection or adopt a time series collection.

Array Manipulation Examples

This section provides examples for processing and manipulating array fields contained in documents, without having to *unwind* and *re-group* the content of each array.

You should ensure you have read the Advanced Use Of Expressions For Array Processing chapter first before digesting the examples in this book's section.

Summarising Arrays For First, Last, Minimum, Maximum & Average Values

Minimum MongoDB Version: 4.4 *(due to use of $first and $last array operators)*

Scenario

You want to generate daily summaries for the exchange rates of foreign currency "pairs" (e.g. "Euro-to-USDollar"). You need to analyse an array of persisted hourly rates for each currency

pair for each day. You will output a daily summary of the open (first), close (last), low (minimum), high (maximum) and average exchange rate values for each currency pair.

Sample Data Population

Drop any old version of the database (if it exists) and then populate new *currency-pair daily* records:

```
use book-array-high-low-avg;
db.dropDatabase();

// Inserts records into the currency_pair_values collection
db.currency_pair_values.insertMany([
  {
    "currencyPair": "USD/GBP",
    "day": ISODate("2021-07-05T00:00:00.000Z"),
    "hour_values": [
      NumberDecimal("0.71903411"), NumberDecimal("0.72741832"),
NumberDecimal("0.71997271"),
      NumberDecimal("0.73837282"), NumberDecimal("0.75262621"),
NumberDecimal("0.74739202"),
      NumberDecimal("0.72972612"), NumberDecimal("0.73837292"),
NumberDecimal("0.72393721"),
      NumberDecimal("0.72746837"), NumberDecimal("0.73787372"),
NumberDecimal("0.73746483"),
      NumberDecimal("0.73373632"), NumberDecimal("0.75737372"),
NumberDecimal("0.76783263"),
      NumberDecimal("0.75632828"), NumberDecimal("0.75362823"),
NumberDecimal("0.74682282"),
      NumberDecimal("0.74628263"), NumberDecimal("0.74726262"),
NumberDecimal("0.75376722"),
      NumberDecimal("0.75799222"), NumberDecimal("0.75545352"),
NumberDecimal("0.74998835"),
    ],
  },
  {
    "currencyPair": "EUR/GBP",
    "day": ISODate("2021-07-05T00:00:00.000Z"),
    "hour_values": [
      NumberDecimal("0.86739394"), NumberDecimal("0.86763782"),
NumberDecimal("0.87362937"),
      NumberDecimal("0.87373652"), NumberDecimal("0.88002736"),
NumberDecimal("0.87866372"),
      NumberDecimal("0.87862628"), NumberDecimal("0.87374621"),
NumberDecimal("0.87182626"),
      NumberDecimal("0.86892723"), NumberDecimal("0.86373732"),
NumberDecimal("0.86017236"),
      NumberDecimal("0.85873636"), NumberDecimal("0.85762283"),
NumberDecimal("0.85362373"),
      NumberDecimal("0.85306218"), NumberDecimal("0.85346632"),
NumberDecimal("0.84647462"),
      NumberDecimal("0.84694720"), NumberDecimal("0.84723232"),
NumberDecimal("0.85002222"),
      NumberDecimal("0.85468322"), NumberDecimal("0.85675656"),
NumberDecimal("0.84811122"),
    ],
  },
]);
```

Aggregation Pipeline

Define a single pipeline ready to perform the aggregation:

```
var pipeline = [
  // Generate day summaries from the hourly array values
  {"$set": {
    "summary.open": {"$first": "$hour_values"},
    "summary.low": {"$min": "$hour_values"},
    "summary.high": {"$max": "$hour_values"},
    "summary.close": {"$last": "$hour_values"},
    "summary.average": {"$avg": "$hour_values"},
  }},

  // Exclude unrequired fields from each daily currency pair record
  {"$unset": [
    "_id",
    "hour_values",
  ]},
];
```

Execution

Execute the aggregation using the defined pipeline and also view its explain plan:

```
db.currency_pair_values.aggregate(pipeline);

db.currency_pair_values.explain("executionStats").aggregate(pipeline);
```

Expected Results

Two documents should be returned, now showing the daily summary open, low, high, close and average prices for each currency pair:

```
[
  {
    currencyPair: 'USD/GBP',
    day: ISODate("2021-07-05T00:00:00.000Z"),
    summary: {
      open: Decimal128("0.71903411"),
      low: Decimal128("0.71903411"),
      high: Decimal128("0.76783263"),
      close: Decimal128("0.74998835"),
      average: Decimal128("0.74275533")
    }
  },
  {
    currencyPair: 'EUR/GBP',
    day: ISODate("2021-07-05T00:00:00.000Z"),
    summary: {
      open: Decimal128("0.86739394"),
      low: Decimal128("0.84647462"),
      high: Decimal128("0.88002736"),
      close: Decimal128("0.84811122"),
      average: Decimal128("0.86186929875")
    }
  }
]
```

Observations

- ***$first* & *$last* For Earlier MongoDB Versions.** MongoDB only introduced the `$first` and `$last` array operator expressions in version 4.4. However, it is straightforward for you to replace each one in the pipeline with an equivalent solution, using the `$arrayElemAt` operator. Below are the alternatives you can use instead of `$first` and `$last` to operator correctly in MongoDB versions before 4.4:

```
// $first equivalent
"summary.open": {"$arrayElemAt": ["$hour_values", 0]},

// $last equivalent
"summary.close": {"$arrayElemAt": ["$hour_values", -1]},
```

Pivot Array Items By A Key

Minimum MongoDB Version: 4.2

Scenario

You have a set of geographically dispersed weather station zones where each zone has multiple sensor devices collecting readings such as temperature, humidity and pressure. Each weather station assembles readings from its devices and once per hour transmits this set of measurements to a central database to store. The set of persisted readings are randomly ordered measurements for different devices in the zone. You need to take the mix of readings and group these by device, so the weather data is easier to consume by downstream dashboards and applications.

> *This example's pipeline relies on some of the more difficult to understand array operator expressions, like* `$map`, `$mergeObjects` *and* `$filter`*. Consequently, ensure you have digested the Advanced Use Of Expressions For Array Processing chapter first, which explains how to use these operators. The pipeline also uses the* `$setUnion` *operator for finding unique values in an array. The Observations part of this chapter explains this in more detail.*

Sample Data Population

Drop any old version of the database (if it exists) and then populate new hourly weather station measurement records:

```
use book-pivot-array-by-key;
db.dropDatabase();

// Inserts records into the weather_measurements collection
db.weather_measurements.insertMany([
  {
    "weatherStationsZone": "FieldZone-ABCD",
    "dayHour": ISODate("2021-07-05T15:00:00.000Z"),
    "readings": [
      {"device": "ABCD-Device-123", "tempCelsius": 18},
      {"device": "ABCD-Device-789", "pressureMBar": 1004},
      {"device": "ABCD-Device-123", "humidityPercent": 31},
      {"device": "ABCD-Device-123", "tempCelsius": 19},
      {"device": "ABCD-Device-123", "pressureMBar": 1005},
      {"device": "ABCD-Device-789", "humidityPercent": 31},
      {"device": "ABCD-Device-123", "humidityPercent": 30},
      {"device": "ABCD-Device-789", "tempCelsius": 20},
      {"device": "ABCD-Device-789", "pressureMBar": 1003},
    ],
  },
  {
    "weatherStationsZone": "FieldZone-ABCD",
    "dayHour": ISODate("2021-07-05T16:00:00.000Z"),
    "readings": [
      {"device": "ABCD-Device-789", "humidityPercent": 33},
      {"device": "ABCD-Device-123", "humidityPercent": 32},
      {"device": "ABCD-Device-123", "tempCelsius": 22},
      {"device": "ABCD-Device-123", "pressureMBar": 1007},
      {"device": "ABCD-Device-789", "pressureMBar": 1008},
      {"device": "ABCD-Device-789", "tempCelsius": 22},
      {"device": "ABCD-Device-789", "humidityPercent": 34},
    ],
  },
]);
```

Aggregation Pipeline

Define a single pipeline ready to perform the aggregation that groups the weather readings by device.

```
var pipeline = [
  // Loop for each unique device, to accumulate an array of devices and their
readings
  {"$set": {
    "readings_device_summary": {
      "$map": {
        "input": {
          "$setUnion": "$readings.device"  // Get only unique device ids from the
array
        },
        "as": "device",
        "in": {
          "$mergeObjects": {  // Merge array of key:values elements into single
object
            "$filter": {
              "input": "$readings",  // Iterate the "readings" array field
              "as": "reading",  // Name the current array element "reading"
              "cond": {  // Only include device properties matching the current
dervice
                "$eq": ["$$reading.device", "$$device"]
              }
            }
          }
        }
      }
    },
  }},

  // Exclude unrequired fields from each record
  {"$unset": [
    "_id",
    "readings",
  ]},
];
```

Execution

Execute the aggregation using the defined pipeline and also view its explain plan:

```
db.weather_measurements.aggregate(pipeline);

db.weather_measurements.explain("executionStats").aggregate(pipeline);
```

Expected Results

Two documents should be returned, with the weather station hourly records containing a new array field of elements representing each device and its measurements, as shown below:

```
[
  {
    weatherStationsZone: 'FieldZone-ABCD',
    dayHour: ISODate("2021-07-05T15:00:00.000Z"),
    readings_device_summary: [
      {
        device: 'ABCD-Device-123',
        tempCelsius: 19,
        humidityPercent: 30,
        pressureMBar: 1005
      },
      {
        device: 'ABCD-Device-789',
        pressureMBar: 1003,
        humidityPercent: 31,
        tempCelsius: 20
      }
    ]
  },
  {
    weatherStationsZone: 'FieldZone-ABCD',
    dayHour: ISODate("2021-07-05T16:00:00.000Z"),
    readings_device_summary: [
      {
        device: 'ABCD-Device-123',
        humidityPercent: 32,
        tempCelsius: 22,
        pressureMBar: 1007
      },
      {
        device: 'ABCD-Device-789',
        humidityPercent: 34,
        pressureMBar: 1008,
        tempCelsius: 22
      }
    ]
  }
]
```

Observations

- **Pivoting Items By A Key.** The pipeline does not use the source array field directly to provide the initial list of items for the `$map` operator to loop through. Instead, it uses the `$setUnion` operator to capture each unique device name from the array of readings. This approach essentially allows you to group subsets of array items by a key. The array

processing and grouping work is self-contained within each document for optimum aggregation performance.

- **Merging Subset Of Array Elements Into One Item.** For each `$map` iteration, a `$filter` operator collects the subset of readings from the original array which match the unique device's name. The `$mergeObjects` operator then takes this subset of readings and turns it into an object, with the measurement type (e.g. `temperature`) as the key and the measurement (e.g. `21℃`) as the value. Suppose more than one reading of the same type exists for a device (e.g. `temporature=22`, `temperature=23`). In that case, the `$mergeObject` operator retains the last value only (e.g. `23℃`), which is the desired behaviour for this example scenario.

- **Potentially Adopt A Better Data Model.** In this example, the weather station hourly data is just persisted directly into the database in the exact structure that the system receives it. However, if it is possible for you to take control of exactly what structure you persist the data in initially, you should take this opportunity. You want to land the data in the database using a model that lends itself to the optimum way for consuming applications to access it. For the Internet of Things (IoT) type uses case, where time-series data is collected and then analysed downstream, you should be adopting the Bucketing pattern. However, if you are fortunate enough to be using MongoDB version 5.0 or greater, you can instead use MongoDB's *time series* collection feature. This particular type of collection efficiently stores sequences of measurements over time to improve subsequent query efficiency. It automatically adopts a *bucketing pattern* internally, meaning that you don't have to design your data model for this explicitly.

Array Sorting & Percentiles

Minimum MongoDB Version: 4.2

Scenario

You've conducted performance testing of an application with the results of each "test run" captured in a database. Each record contains a set of response times for the test run. You want to analyse the data from multiple runs to identify the slowest ones. You calculate the median (50th percentile) and 90th percentile response times for each test run and only keep results where the 90th percentile response time is greater than 100 milliseconds.

For MongoDB version 5.1 and earlier, the example will use a modified version of an "expression generator" function for inline sorting of arrays created by Asya Kamsky. Adopting this approach

avoids the need for you to use the combination of `$unwind`, `$sort`, and `$group` stages. Instead, you process each document's array in isolation for optimum performance. You can re-use this chapter's custom `sortArray()` function as-is for your own situations where you need to sort an array field's contents. For MongoDB version 5.2 and greater, you can instead use MongoDB's new $sortArray operator.

Sample Data Population

Drop any old version of the database (if it exists) and then populate the test run results documents:

```
use book-inline-array-sort-percentile;
db.dropDatabase();

// Insert 7 records into the performance_test_results collection
db.performance_test_results.insertMany([
  {
    "testRun": 1,
    "datetime": ISODate("2021-08-01T22:51:27.638Z"),
    "responseTimesMillis": [
      62, 97, 59, 104, 97, 71, 62, 115, 82, 87,
    ],
  },
  {
    "testRun": 2,
    "datetime": ISODate("2021-08-01T22:56:32.272Z"),
    "responseTimesMillis": [
      34, 63, 51, 104, 87, 63, 64, 86, 105, 51, 73,
      78, 59, 108, 65, 58, 69, 106, 87, 93, 65,
    ],
  },
  {
    "testRun": 3,
    "datetime": ISODate("2021-08-01T23:01:08.908Z"),
    "responseTimesMillis": [
      56, 72, 83, 95, 107, 83, 85,
    ],
  },
  {
    "testRun": 4,
    "datetime": ISODate("2021-08-01T23:17:33.526Z"),
    "responseTimesMillis": [
      78, 67, 107, 110,
    ],
  },
  {
    "testRun": 5,
    "datetime": ISODate("2021-08-01T23:24:39.998Z"),
    "responseTimesMillis": [
      75, 91, 75, 87, 99, 88, 55, 72, 99, 102,
    ],
  },
  {
    "testRun": 6,
    "datetime": ISODate("2021-08-01T23:27:52.272Z"),
    "responseTimesMillis": [
      88, 89,
    ],
  },
  {
    "testRun": 7,
    "datetime": ISODate("2021-08-01T23:31:59.917Z"),
    "responseTimesMillis": [
      101,
    ],
```

```
  },
]);
```

Aggregation Pipeline

If you are using version 5.1 or earlier of MongoDB, you need to define the following custom `sortArray()` function for inline sorting of the contents of an array field, ready for you to use in a pipeline:

```
// Macro function to generate a complex aggregation expression for sorting an
array
// This function isn't required for MongoDB version 5.2+ due to the new $sortArray
operator
function sortArray(sourceArrayField) {
  return {
    // GENERATE BRAND NEW ARRAY TO CONTAIN THE ELEMENTS FROM SOURCE ARRAY BUT NOW
SORTED
    "$reduce": {
      "input": sourceArrayField,
      "initialValue": [],  // THE FIRST VERSION OF TEMP SORTED ARRAY WILL BE EMPTY
      "in": {
        "$let": {
          "vars": {  // CAPTURE $$this & $$value FROM OUTER $reduce BEFORE
OVERRIDDEN
            "resultArray": "$$value",
            "currentSourceArrayElement": "$$this"
          },
          "in": {
            "$let": {
              "vars": {
                // FIND EACH SOURCE ARRAY'S CURRENT ELEMENT POSITION IN NEW SORTED
ARRAY
                "targetArrayPosition": {
                  "$reduce": {
                    "input": {"$range": [0, {"$size": "$$resultArray"}]},  //
"0,1,2.."
                    "initialValue": {  // INITIALISE SORTED POSITION TO BE LAST
ARRAY ELEMENT
                      "$size": "$$resultArray"
                    },
                    "in": {  // LOOP THRU "0,1,2..."
                      "$cond": [
                        {"$lt": [
                          "$$currentSourceArrayElement",
                          {"$arrayElemAt": ["$$resultArray", "$$this"]}
                        ]},
                        {"$min": ["$$value", "$$this"]},  // ONLY USE IF LOW VAL
NOT YET FOUND
                        "$$value"  // RETAIN INITIAL VAL AGAIN AS NOT YET FOUND
CORRECT POSTN
                      ]
                    }
                  }
                }
              },
              "in": {
                // BUILD NEW SORTED ARRAY BY SLICING OLDER ONE & INSERTING NEW
ELEMENT BETWEEN
                "$concatArrays": [
                  {"$cond": [  // RETAIN THE EXISTING FIRST PART OF THE NEW ARRAY
                    {"$eq": [0, "$$targetArrayPosition"]},
                    [],
                    {"$slice": ["$$resultArray", 0, "$$targetArrayPosition"]},
```

```
                    ]},
                    ["$$currentSourceArrayElement"],  // PULL IN THE NEW POSITIONED
ELEMENT
                    {"$cond": [  // RETAIN THE EXISTING LAST PART OF THE NEW ARRAY
                      {"$gt": [{"$size": "$$resultArray"}, 0]},
                      {"$slice": [
                        "$$resultArray",
                        "$$targetArrayPosition",
                        {"$size": "$$resultArray"}
                      ]},
                      [],
                    ]},
                  ]
                }
              }
            }
          }
        }
      }
    };
}
```

Define the new `arrayElemAtPercentile()` function for capturing the element of a sorted array at the nth percentile position:

```
// Macro function to find nth percentile element of a sorted version of an array
function arrayElemAtPercentile(sourceArrayField, percentile) {
  return {
    "$let": {
      "vars": {
        "sortedArray": sortArray(sourceArrayField),
        // Comment out the line above and uncomment the line below if running MDB
5.2 or greater
        // "sortedArray": {"$sortArray": {"input": sourceArrayField, "sortBy":
1}},
      },
      "in": {
        "$arrayElemAt": [  // FIND ELEMENT OF ARRAY AT NTH PERCENTILE POSITION
          "$$sortedArray",
          {"$subtract": [  // ARRAY IS 0-INDEX BASED SO SUBTRACT 1 TO GET POSITION
            {"$ceil":  // FIND NTH ELEMENT IN THE ARRAY, ROUNDED UP TO NEAREST
integer
              {"$multiply": [
                {"$divide": [percentile, 100]},
                {"$size": "$$sortedArray"}
              ]}
            },
            1
          ]}
        ]
      }
    }
  };
}
```

If running MongoDB version 5.2 or greater, you can instead comment/uncomment the specific lines indicated in the code above to leverage MongoDB's new `$sortArray` operator

Define the pipeline ready to perform the aggregation:

```
var pipeline = [
  // Capture new fields for the ordered array + various percentiles
  {"$set": {
    "sortedResponseTimesMillis": sortArray("$responseTimesMillis"),
    // Comment out the line above and uncomment the line below if running MDB 5.2
or greater
    // "sortedResponseTimesMillis": {"$sortArray": {"input":
"$responseTimesMillis", "sortBy": 1}},
    "medianTimeMillis": arrayElemAtPercentile("$responseTimesMillis", 50),
    "ninetiethPercentileTimeMillis": arrayElemAtPercentile("$responseTimesMillis",
90),
  }},

  // Only show results for tests with slow latencies (i.e. 90th%-ile responses
>100ms)
  {"$match": {
    "ninetiethPercentileTimeMillis": {"$gt": 100},
  }},

  // Exclude unrequired fields from each record
  {"$unset": [
    "_id",
    "datetime",
    "responseTimesMillis",
  ]},
];
```

If running MongoDB version 5.2 or greater, you can instead comment/uncomment the specific lines indicated in the code above to leverage MongoDB's new `$sortArray` operator

Execution

Execute the aggregation using the defined pipeline and also view its explain plan:

```
db.performance_test_results.aggregate(pipeline);

db.performance_test_results.explain("executionStats").aggregate(pipeline);
```

Expected Results

Five documents should be returned, representing the subset of documents with a 90th percentile response time greater than 100 milliseconds, as shown below:

```
[
  {
    testRun: 7,
    sortedResponseTimesMillis: [ 101 ],
    medianTimeMillis: 101,
    ninetiethPercentileTimeMillis: 101
  },
  {
    testRun: 1,
    sortedResponseTimesMillis: [
      59, 62, 62,  71,  82,
      87, 97, 97, 104, 115
    ],
    medianTimeMillis: 82,
    ninetiethPercentileTimeMillis: 104
  },
  {
    testRun: 2,
    sortedResponseTimesMillis: [
      34, 51, 51,  58,  59,  63,  63,
      64, 65, 65,  69,  73,  78,  86,
      87, 87, 93, 104, 105, 106, 108
    ],
    medianTimeMillis: 69,
    ninetiethPercentileTimeMillis: 105
  },
  {
    testRun: 3,
    sortedResponseTimesMillis: [
      56, 72,  83, 83,
      85, 95, 107
    ],
    medianTimeMillis: 83,
    ninetiethPercentileTimeMillis: 107
  },
  {
    testRun: 4,
    sortedResponseTimesMillis: [ 67, 78, 107, 110 ],
    medianTimeMillis: 78,
    ninetiethPercentileTimeMillis: 110
  }
]
```

Observations

- **Macro Functions.** In this chapter, you employ two functions, `sortArray()` and `arrayElemAtPercentile()`, to generate portions of aggregation boilerplate code. These functions are essentially macros. You invoke these functions from within the pipeline you create in the MongoDB Shell. Each function you invoke embeds the returned boilerplate

code into the pipeline's code. You can see this in action by typing the text `pipeline` into the Shell and pressing *enter*. Note, you may first have to increase the depth displayed in *mongosh* by issuing the *mongosh* command `config.set("inspectDepth", 100)`. This action will display a single large piece of code representing the whole pipeline, including the macro-generated code. The aggregation runtime never sees or runs the custom functions `sortArray()` and `arrayElemAtPercentile()` directly. Of course, you won't use JavaScript functions to generate composite expressions if you use a different programming language and MongoDB Driver. You will use the relevant features of your specific programming language to assemble composite expression objects.

- **Sorting On Primitives Only.** The custom `sortArray()` function used for MongoDB versions 5.1 and earlier will sort arrays containing just primitive values, such as integers, floats, date-times and strings. However, if an array's members are objects (i.e. each has its own fields and values), the code will not sort the array correctly. It is possible to enhance the function to enable sorting an array of objects, but this enhancement is not covered here. For MongoDB versions 5.2 and greater, the new `$sortArray` operator provides options to easily sort an array of objects.

- **Comparison With Classic Sorting Algorithms.** Despite being more optimal than unwinding and re-grouping arrays to bring them back into the same documents, the custom sorting code will be slower than commonly recognised computer science sorting algorithms. This situation is due to the limitations of the aggregation domain language compared with a general-purpose programming language. The performance difference will be negligible for arrays with few elements (probably up to a few tens of members). For larger arrays containing hundreds of members or more, the degradation in performance will be more profound. For MongoDB versions 5.2 and greater, the new `$sortArray` operator leverages a fully optimised sorting algorithm under the covers to avoid this issue.

Appendices

The following sections contain reference material you may find useful on your MongoDB Aggregations journey.

Stages Cheatsheet

A simple example for each stage in the MongoDB Aggregation Framework.

Stages:

Query	Mutate	Summarise/Itemise	Join	Input/
$geoNear	$addFields	$bucket	$graphLookup	$docur
$limit	$densify	$bucketAuto	$lookup	$merge
$match	$project	$count	$unionWith	$out
$sample	$redact	$facet		
$skip	$replaceRoot	$group		
$sort	$replaceWith	$sortByCount		
	$set	$unwind		
	$setWindowFields			
	$unset			

The following stages are not included because they are unrelated to aggregating business data or rely on external systems: `$collStats`, `$indexStats`, `$listSessions`, `$planCacheStats`, `$currentOp`, `$listLocalSessions`, `$search`, `$searchMeta`

Input Collections:

```
// shapes
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}

// lists
{_id: "▤", a: "●", b: ["◰", "◱"]}
{_id: "▥", a: "▲", b: ["◲"]}
{_id: "▦", a: "▲", b: ["◰", "◳", "◱"]}
{_id: "▧", a: "●", b: ["◰"]}
{_id: "▨", a: "■", b: ["◳", "◱"]}

// places
{_id: "◧", loc: {type: "Point", coordinates: [1,1]}}
{_id: "◨", loc: {type: "Point", coordinates: [3,3]}}
{_id: "◩", loc: {type: "Point", coordinates: [5,5]}}
{_id: "◪", loc: {type: "LineString", coordinates: [[7,7],[8,8]]}}
```

$addFields

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$addFields: {z: "●"}
   ⬇
{_id: '◐', x: '■', y: '▲', val: 10, z: '●'}
{_id: '◑', x: '■', y: '■', val: 60, z: '●'}
{_id: '◒', x: '●', y: '■', val: 80, z: '●'}
{_id: '◓', x: '▲', y: '▲', val: 85, z: '●'}
{_id: '◔', x: '■', y: '▲', val: 90, z: '●'}
{_id: '◕', x: '●', y: '■', val: 95, z: '●'}
```

$bucket

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$bucket: {
  groupBy: "$val",
  boundaries: [0, 25, 50, 75, 100],
  default: "Other"
}
   ⬇
{_id: 0, count: 1}
{_id: 50, count: 1}
{_id: 75, count: 4}
```

$bucketAuto

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$bucketAuto: {groupBy: "$val", buckets: 3}
   ⬇
{_id: {min: 10, max: 80}, count: 2}
{_id: {min: 80, max: 90}, count: 2}
{_id: {min: 90, max: 95}, count: 2}
```

$count

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$count: "amount"
   ⬇
{amount: 6}
```

$densify

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$densify: {
  field: "val",
  partitionByFields: ["x"],
  range: {bounds: "full", step: 25}
}
   ⬇
{_id: '◐', x: '■', y: '▲', val: 10}
{x: '■', val: 35}
{_id: '◑', x: '■', y: '■', val: 60}
{x: '●', val: 10}
{x: '●', val: 35}
{x: '●', val: 60}
{_id: '◒', x: '●', y: '■', val: 80}
{x: '▲', val: 10}
{x: '▲', val: 35}
{x: '▲', val: 60}
{_id: '◓', x: '▲', y: '▲', val: 85}
{x: '■', val: 85}
{_id: '◔', x: '■', y: '▲', val: 90}
{x: '●', val: 85}
{_id: '◕', x: '●', y: '■', val: 95}
```

$documents

```
[     ]
   ⬇
$documents: {
  {p: "▭", q: "▯"},
  {p: "▯", q: "▭"}
}
   ⬇
{p: '▭', q: '▯'}
{p: '▯', q: '▭'}
```

$facet

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$facet: {
  X_CIRCLE_FACET: [{$match: {x: "●"}}],
  FIRST_TWO_FACET: [{$limit: 2}]
}
   ⬇
{
  X_CIRCLE_FACET: [
    {_id: '◒', x: '●', y: '■', val: 80},
    {_id: '◕', x: '●', y: '■', val: 95}
  ],
  FIRST_TWO_FACET: [
    {_id: '◐', x: '■', y: '▲', val: 10},
    {_id: '◑', x: '■', y: '■', val: 60}
  ]
}
```

$geoNear

```
{_id: "◧", loc: {type: "Point", coordinates: [1,1]}}
{_id: "◨", loc: {type: "Point", coordinates: [3,3]}}
{_id: "◩", loc: {type: "Point", coordinates: [5,5]}}
{_id: "◪", loc: {type: "LineString", coordinates: [[7,7],[8,8]]}}
   ⬇
$geoNear: {
  near: {type: "Point", coordinates: [9,9]},
  distanceField: "distance"
}
   ⬇
{_id: '◪', loc: { type: 'LineString', coordinates: [[7,7], [8,8]]}
      distance: 156565.32902203742}
{_id: '◩', loc: { type: 'Point', coordinates: [5,5]}
      distance: 627304.9320885336}
{_id: '◨', loc: { type: 'Point', coordinates: [3,3]}
      distance: 941764.4675092621}
{_id: '◧', loc: { type: 'Point', coordinates: [1,1]}
      distance: 1256510.3666236876}
```

$graphLookup

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$graphLookup: {
  from: "shapes",
  startWith: "$x",
  connectFromField: "x",
  connectToField: "y",
  depthField: "depth",
  as: "connections"
}
$project: {connections_count: {$size: "$connections"}}
   ⬇
{_id: '◐', connections_count: 3}
{_id: '◑', connections_count: 3}
{_id: '◒', connections_count: 0}
{_id: '◓', connections_count: 6}
{_id: '◔', connections_count: 3}
{_id: '◕', connections_count: 0}
```

$group

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$group: {_id: "$x", ylist: {$push: "$y"}}
   ⬇
{_id: '●', ylist: ['■', '■']}
{_id: '■', ylist: ['▲', '■', '▲']}
{_id: '▲', ylist: ['▲']}
```

$limit

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$limit: 2
   ⬇
{_id: '◐', x: '■', y: '▲', val: 10}
{_id: '◑', x: '■', y: '■', val: 60}
```

$lookup

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   +
{_id: "▤", a: "●", b: ["◰", "◱"]}
{_id: "▥", a: "▲", b: ["◲"]}
{_id: "▦", a: "▲", b: ["◰", "◳", "◱"]}
{_id: "▧", a: "●", b: ["◰"]}
{_id: "▨", a: "■", b: ["◳", "◱"]}
   ⬇
$lookup: {
  from: "lists",
  localField: "y",
  foreignField: "a",
  as: "refs"
}
   ⬇
{_id: '◐', x: '■', y: '▲', val: 10, refs: [
  {_id: '▥', a: '▲', b: ['◲']},
  {_id: '▦', a: '▲', b: ['◰', '◳', '◱']}
]}
{_id: '◑', x: '■', y: '■', val: 60, refs: [
  {_id: '▨', a: '■', b: ['◳', '◱']}
]}
{_id: '◒', x: '●', y: '■', val: 80, refs: [
  {_id: '▨', a: '■', b: ['◳', '◱']}
]}
{_id: '◓', x: '▲', y: '▲', val: 85, refs: [
  {_id: '▥', a: '▲', b: ['◲']},
  {_id: '▦', a: '▲', b: ['◰', '◳', '◱']}
]}
{_id: '◔', x: '■', y: '▲', val: 90, refs: [
  {_id: '▥', a: '▲', b: ['◲']},
  {_id: '▦', a: '▲', b: ['◰', '◳', '◱']}
]}
{_id: '◕', x: '●', y: '■', val: 95, refs: [
  {_id: '▨', a: '■', b: ['◳', '◱']}
]}
```

$match

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$match: {y: "▲"}
   ⬇
{_id: '◐', x: '■', y: '▲', val: 10}
{_id: '◓', x: '▲', y: '▲', val: 85}
{_id: '◔', x: '■', y: '▲', val: 90}
```

$merge

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$merge: {into: "results"}
   ⬇
db.results.find()
{_id: '◐', x: '■', y: '▲', val: 10}
{_id: '◑', x: '■', y: '■', val: 60}
{_id: '◒', x: '●', y: '■', val: 80}
{_id: '◓', x: '▲', y: '▲', val: 85}
{_id: '◔', x: '■', y: '▲', val: 90}
{_id: '◕', x: '●', y: '■', val: 95}
```

$out

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$out: "results"
   ⬇
db.results.find()
{_id: '◐', x: '■', y: '▲', val: 10}
{_id: '◑', x: '■', y: '■', val: 60}
{_id: '◒', x: '●', y: '■', val: 80}
{_id: '◓', x: '▲', y: '▲', val: 85}
{_id: '◔', x: '■', y: '▲', val: 90}
{_id: '◕', x: '●', y: '■', val: 95}
```

$project

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$project: {x: 1}
   ⬇
{_id: '◐', x: '■'}
{_id: '◑', x: '■'}
{_id: '◒', x: '●'}
{_id: '◓', x: '▲'}
{_id: '◔', x: '■'}
{_id: '◕', x: '●'}
```

$redact

```
{_id: "◧", loc: {type: "Point", coordinates: [1,1]}}
{_id: "◨", loc: {type: "Point", coordinates: [3,3]}}
{_id: "◩", loc: {type: "Point", coordinates: [5,5]}}
{_id: "◪", loc: {type: "LineString", coordinates: [[7,7],[8,8]]}}
   ⬇
$redact: {$cond: {
  if  : {$eq: ["$type", "LineString"]},
  then: "$$PRUNE",
  else: "$$DESCEND"
}}
   ⬇
{_id: '◧', loc: { type: 'Point', coordinates: [1,1]}}
{_id: '◨', loc: { type: 'Point', coordinates: [3,3]}}
{_id: '◩', loc: { type: 'Point', coordinates: [5,5]}}
{_id: '◪'}
```

$replaceRoot

```
{_id: "▤", a: "●", b: ["◰", "◱"]}
{_id: "▥", a: "▲", b: ["◲"]}
{_id: "▦", a: "▲", b: ["◰", "◳", "◱"]}
{_id: "▧", a: "●", b: ["◰"]}
{_id: "▨", a: "■", b: ["◳", "◱"]}
   ⬇
$replaceRoot: {
  newRoot: {first: {$first: "$b"}, last: {$last: "$b"}}
}
   ⬇
{first: '◰', last: '◱'}
{first: '◲', last: '◲'}
{first: '◰', last: '◱'}
{first: '◰', last: '◰'}
{first: '◳', last: '◱'}
```

$replaceWith

```
{_id: "▤", a: "●", b: ["◰", "◱"]}
{_id: "▥", a: "▲", b: ["◲"]}
{_id: "▦", a: "▲", b: ["◰", "◳", "◱"]}
{_id: "▧", a: "●", b: ["◰"]}
{_id: "▨", a: "■", b: ["◳", "◱"]}
   ⬇
$replaceWith: {
  first: {$first: "$b"}, last: {$last: "$b"}
}
   ⬇
{first: '◰', last: '◱'}
{first: '◲', last: '◲'}
{first: '◰', last: '◱'}
{first: '◰', last: '◰'}
{first: '◳', last: '◱'}
```

$sample

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$sample: {size: 3}
   ⬇
{_id: '◔', x: '■', y: '▲', val: 90}
{_id: '◓', x: '▲', y: '▲', val: 85}
{_id: '◑', x: '■', y: '■', val: 60}
```

$set

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$set: {y: "▲"}
   ⬇
{_id: '◐', x: '■', y: '▲', val: 10}
{_id: '◑', x: '■', y: '▲', val: 60}
{_id: '◒', x: '●', y: '▲', val: 80}
{_id: '◓', x: '▲', y: '▲', val: 85}
{_id: '◔', x: '■', y: '▲', val: 90}
{_id: '◕', x: '●', y: '▲', val: 95}
```

$setWindowFields

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$setWindowFields: {
  partitionBy: "$x",
  sortBy: {"_id": 1},
  output: {
    cumulativeValShapeX: {
      $sum: "$val",
      window: {
        documents: ["unbounded", "current"]
      }
    }
 }
}
   ⬇
{_id: '◐', x: '■', y: '▲', val: 10, cumulativeValShapeX: 10}
{_id: '◑', x: '■', y: '■', val: 60, cumulativeValShapeX: 70}
{_id: '◔', x: '■', y: '▲', val: 90, cumulativeValShapeX: 160}
{_id: '◓', x: '▲', y: '▲', val: 85, cumulativeValShapeX: 85}
{_id: '◒', x: '●', y: '■', val: 80, cumulativeValShapeX: 80}
{_id: '◕', x: '●', y: '■', val: 95, cumulativeValShapeX: 175}
```

$skip

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$skip: 5
   ⬇
{_id: '◕', x: '●', y: '■', val: 95}
```

$sort

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$sort: {x: 1, y: 1}
   ⬇
{_id: '◑', x: '■', y: '■', val: 60}
{_id: '◐', x: '■', y: '▲', val: 10}
{_id: '◔', x: '■', y: '▲', val: 90}
{_id: '◓', x: '▲', y: '▲', val: 85}
{_id: '◒', x: '●', y: '■', val: 80}
{_id: '◕', x: '●', y: '■', val: 95}
```

$sortByCount

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$sortByCount: "$x"
   ⬇
{_id: '■', count: 3}
{_id: '●', count: 2}
{_id: '▲', count: 1}
```

$unionWith

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   +
{_id: "▤", a: "●", b: ["◰", "◱"]}
{_id: "▥", a: "▲", b: ["◲"]}
{_id: "▦", a: "▲", b: ["◰", "◳", "◱"]}
{_id: "▧", a: "●", b: ["◰"]}
{_id: "▨", a: "■", b: ["◳", "◱"]}
   ⬇
$unionWith: {coll: "lists"}
   ⬇
{_id: '◐', x: '■', y: '▲', val: 10}
{_id: '◑', x: '■', y: '■', val: 60}
{_id: '◒', x: '●', y: '■', val: 80}
{_id: '◓', x: '▲', y: '▲', val: 85}
{_id: '◔', x: '■', y: '▲', val: 90}
{_id: '◕', x: '●', y: '■', val: 95}
{_id: '▤', a: '●', b: ['◰', '◱']}
{_id: '▥', a: '▲', b: ['◲']}
{_id: '▦', a: '▲', b: ['◰', '◳', '◱']}
{_id: '▧', a: '●', b: ['◰']}
{_id: '▨', a: '■', b: ['◳', '◱']}
```

$unset

```
{_id: "◐", x: "■", y: "▲", val: 10}
{_id: "◑", x: "■", y: "■", val: 60}
{_id: "◒", x: "●", y: "■", val: 80}
{_id: "◓", x: "▲", y: "▲", val: 85}
{_id: "◔", x: "■", y: "▲", val: 90}
{_id: "◕", x: "●", y: "■", val: 95}
   ⬇
$unset: ["x"]
   ⬇
{_id: '◐', y: '▲', val: 10}
{_id: '◑', y: '■', val: 60}
{_id: '◒', y: '■', val: 80}
{_id: '◓', y: '▲', val: 85}
{_id: '◔', y: '▲', val: 90}
{_id: '◕', y: '■', val: 95}
```

$unwind

```
{_id: "▤", a: "●", b: ["◰", "◱"]}
{_id: "▥", a: "▲", b: ["◲"]}
{_id: "▦", a: "▲", b: ["◰", "◳", "◱"]}
{_id: "▧", a: "●", b: ["◰"]}
{_id: "▨", a: "■", b: ["◳", "◱"]}
   ⬇
$unwind: {path: "$b"}
   ⬇
{_id: '▤', a: '●', b: '◰'}
{_id: '▤', a: '●', b: '◱'}
{_id: '▥', a: '▲', b: '◲'}
{_id: '▦', a: '▲', b: '◰'}
{_id: '▦', a: '▲', b: '◳'}
{_id: '▦', a: '▲', b: '◱'}
{_id: '▧', a: '●', b: '◰'}
{_id: '▨', a: '■', b: '◳'}
{_id: '▨', a: '■', b: '◱'}
```

Stages Cheatsheet Source

To test all the aggregation stage examples shown in the Cheatsheet, run the following JavaScript from the MongoDB Shell connected to a MongoDB database.

Collections Configuration & Data Population

```
// DB configuration
use cheatsheet;
db.dropDatabase();
db.places.createIndex({loc: "2dsphere"});

// 'shapes' collection
db.shapes.insertMany([
  {_id: "◐", x: "■", y: "▲", val: 10},
  {_id: "◑", x: "■", y: "■", val: 60},
  {_id: "◒", x: "●", y: "■", val: 80},
  {_id: "◓", x: "▲", y: "▲", val: 85},
  {_id: "◔", x: "■", y: "▲", val: 90},
  {_id: "◕", x: "●", y: "■", val: 95},
]);

// 'lists' collection
db.lists.insertMany([
  {_id: "▤", a: "●", b: ["◰", "◱"]},
  {_id: "▥", a: "▲", b: ["◲"]},
  {_id: "▦", a: "▲", b: ["◰", "◳", "◱"]},
  {_id: "▧", a: "●", b: ["◰"]},
  {_id: "▨", a: "■", b: ["◳", "◱"]},
]);

// 'places' collection
db.places.insertMany([
  {_id: "◧", loc: {type: "Point", coordinates: [1,1]}},
  {_id: "◨", loc: {type: "Point", coordinates: [3,3]}},
  {_id: "◩", loc: {type: "Point", coordinates: [5,5]}},
  {_id: "◪", loc: {type: "LineString", coordinates: [[7,7],[8,8]]}},
]);
```

Aggregation Stage Examples

Some of these stages apply to later versions of MongoDB only (e.g. 4.4, 5.0). Each stage is marked with the minimum version of MongoDB (shown in brackets). Therefore if you are running an earlier version, first comment out the appropriate "JavaScript stage sections" before executing the code.

```
// $addFields  (v3.4)
db.shapes.aggregate([
  {"$addFields": {"z": "●"}}
]);


// $bucket  (v3.4)
db.shapes.aggregate([
  {"$bucket": {
    "groupBy": "$val", "boundaries": [0, 25, 50, 75, 100], "default": "Other"
  }}
]);


// $bucketAuto  (v3.4)
db.shapes.aggregate([
  {"$bucketAuto": {"groupBy": "$val", "buckets": 3}}
]);


// $count  (v3.4)
db.shapes.aggregate([
  {"$count": "amount"}
]);


// $densify  (v5.1)
db.shapes.aggregate([
  {"$densify": {
    "field": "val",
    "partitionByFields": ["x"],
    "range": {"bounds": "full", "step": 25}
  }}
]);


// $documents  (v5.1)
db.aggregate([
  {"$documents": [
    {"p": "▭", "q": "▯"},
    {"p": "▯", "q": "▭"},
  ]}
]);


// $facet  (v3.4)
db.shapes.aggregate([
  {"$facet": {
    "X_CIRCLE_FACET": [{"$match": {"x": "●"}}],
    "FIRST_TWO_FACET" : [{"$limit": 2}],
  }}
]);
```

```
// $geoNear  (v2.2)
db.places.aggregate([
  {"$geoNear": {
    "near": {"type": "Point", "coordinates": [9,9]}, "distanceField": "distance"
  }}
]);


// $graphLookup  (v3.4)
db.shapes.aggregate([
  {"$graphLookup": {
    "from": "shapes",
    "startWith": "$x",
    "connectFromField": "x",
    "connectToField": "y",
    "depthField": "depth",
    "as": "connections",
  }},
  {$project: {"connections_count": {"$size": "$connections"}}}
]);


// $group  (v2.2)
db.shapes.aggregate([
  {"$group": {"_id": "$x", "ylist": {"$push": "$y"}}}
]);


// $limit  (v2.2)
db.shapes.aggregate([
  {"$limit": 2}
]);


// $lookup  (v3.2)
db.shapes.aggregate([
  {"$lookup": {
    "from": "lists",
    "localField": "y",
    "foreignField": "a",
    "as": "refs",
  }}
]);


// $match  (v2.2)
db.shapes.aggregate([
  {"$match": {"y": "▲"}
}]);


// $merge  (v4.2)
db.results.drop();
db.shapes.aggregate([
```

```
  {"$merge": {"into": "results"}}
]);
db.results.find();


// $out  (v2.6)
db.results.drop();
db.shapes.aggregate([
  {"$out": "results"}
]);
db.results.find();


// $project  (v2.2)
db.shapes.aggregate([
  {"$project": {"x": 1}}
]);


// $redact  (v2.6)
db.places.aggregate([
  {"$redact": {"$cond": {
    "if"  : {"$eq": ["$type", "LineString"]},
    "then": "$$PRUNE",
    "else": "$$DESCEND"
  }}}
]);


// $replaceRoot  (v3.4)
db.lists.aggregate([
  {"$replaceRoot": {"newRoot": {"first": {"$first": "$b"}, "last": {"$last":
"$b"}}}}
]);


// $replaceWith  (v4.2)
db.lists.aggregate([
  {"$replaceWith": {"first": {"$first": "$b"}, "last": {"$last": "$b"}}}
]);


// $sample  (v3.2)
db.shapes.aggregate([
  {"$sample": {"size": 3}}
]);


// $set  (v4.2)
db.shapes.aggregate([
  {"$set": {"y": "▲"}}
]);
```

```
// $setWindowFields  (v5.0)
db.shapes.aggregate([
  {"$setWindowFields": {
    "partitionBy": "$x",
    "sortBy": {"_id": 1},
    "output": {
      "cumulativeValShapeX": {
        "$sum": "$val",
        "window": {
          "documents": ["unbounded", "current"]
        }
      }
    }
  }}
]);


// $skip  (v2.2)
db.shapes.aggregate([
  {"$skip": 5}
]);


// $sort  (v2.2)
db.shapes.aggregate([
  {"$sort": {"x": 1, "y": 1}}
]);


// $sortByCount  (v3.4)
db.shapes.aggregate([
  {"$sortByCount": "$x"}
]);


// $unionWith  (v4.4)
db.shapes.aggregate([
  {"$unionWith": {"coll": "lists"}}
]);


// $unset  (v4.2)
db.shapes.aggregate([
  {"$unset": ["x"]}
]);


// $unwind  (v2.2)
db.lists.aggregate([
  {"$unwind": {"path": "$b"}}
]);
```

Book Version History

A summary of the significant additions in each *major* version of this book.

Version 3.0 (released in October 2021)

- Advanced Use Of Expressions For Array Processing guide chapter
- Summarising Arrays For First, Last, Min, Max & Average example chapter
- Pivot Array Items By A Key example chapter
- Array Sorting & Percentiles example chapter

Version 2.0 (released in July 2021)

- Sharding Considerations guide chapter
- Distinct List Of Values example chapter
- IoT Power Consumption example chapter
- Stages Cheatsheet appendices
- Book Version History appendix

Version 1.0 (released in May 2021)

- Introducing MongoDB Aggregations chapter
- History Of MongoDB Aggregations chapter
- Getting Started chapter
- Getting Help chapter
- Embrace Composability For Increased Productivity guide chapter
- Better Alternatives To A Project Stage guide chapter
- Using Explain Plans guide chapter
- Pipeline Performance Considerations guide chapter
- Expressions Explained guide chapter
- Filtered Top Subset example chapter
- Group & Total example chapter
- Unpack Arrays & Group Differently example chapter
- One-to-One Join example chapter
- Multi-Field Join & One-to-Many example chapter
- Strongly-Typed Conversion example chapter
- Convert Incomplete Date Strings example chapter
- Faceted Classification example chapter
- Largest Graph Network example chapter
- Incremental Analytics example chapter
- Restricted View example chapter
- Mask Sensitive Fields example chapter

End