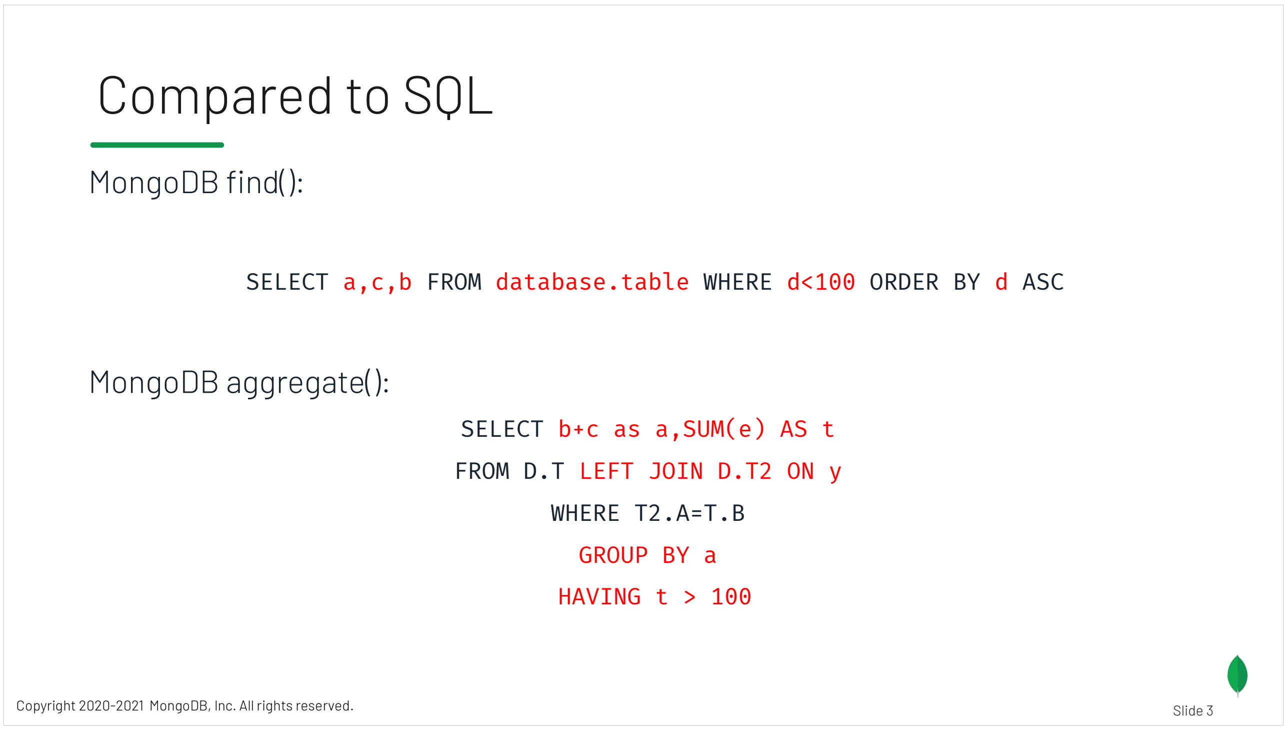
Aggregation

DF200 - Optimizing Storage and Retrieval

Release: 20211217

Aggregation Basics

- Until now, we have only filtered data
  - What Documents
  - What Fields
  - What Order
- Aggregation allows us to compute new data
  - Calculated fields
  - Summarised and grouped values
  - Reshape documents



Compared to SQL

MongoDB find():

```
SELECT a,c,b FROM database.table WHERE d<100 ORDER BY d ASC
```

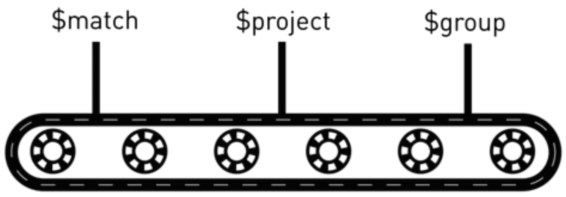
MongoDB aggregate():

```
SELECT b+c as a,SUM(e) AS t
FROM D.T LEFT JOIN D.T2 ON y
WHERE T2.A=T.B
GROUP BY a
HAVING t > 100
```

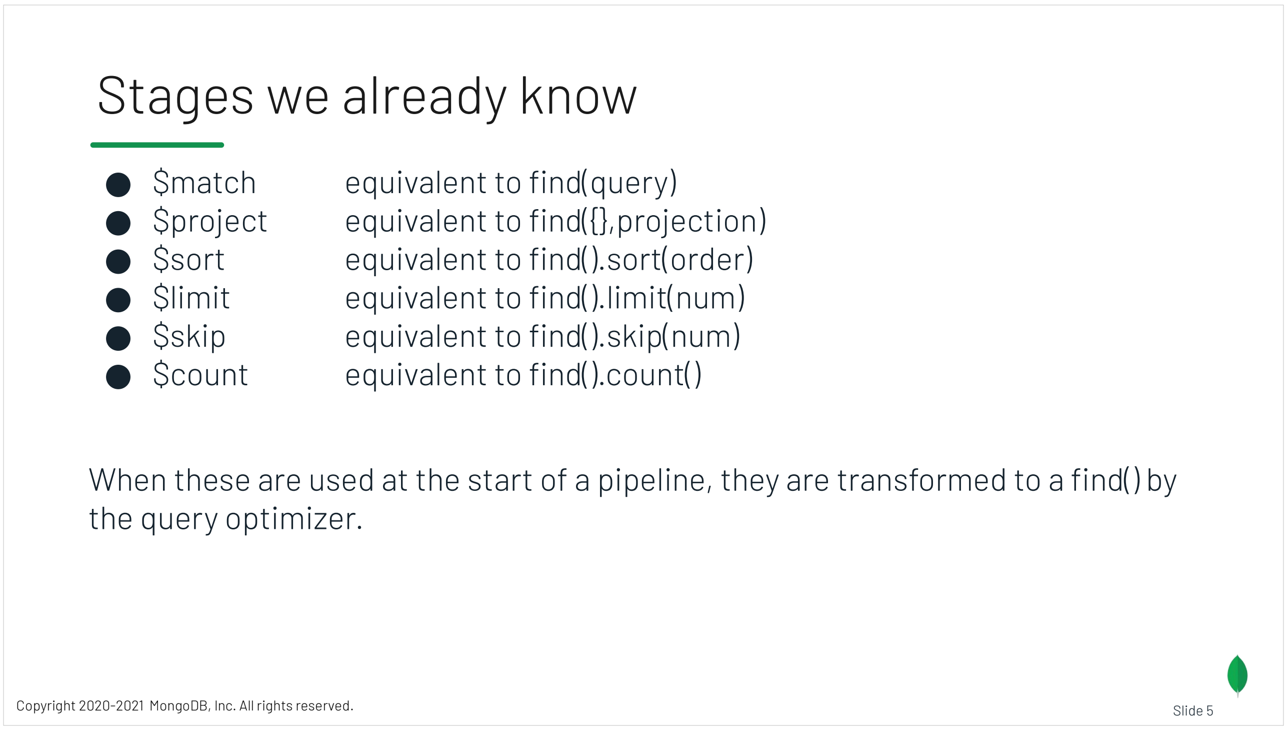
MongoDB's find is equivalent of SQL SELECT and WHERE statements
Aggregate stages such as $GROUP and $LOOKUP can be equivalent of SQL join and group
Find() and Aggregate are slowly converging in MongoDB so this distinction gets less and less with each release.

Aggregation is a pipeline

- Each transformation is a single step known as a stage.
- Compared to one huge SQL style statement this is
  - Easier to understand
  - Easier to debug
  - Easier for MongoDB to rewrite and optimize



- Aggregation is a linear process with stages
- It makes it easier to understand and follow what queries are doing

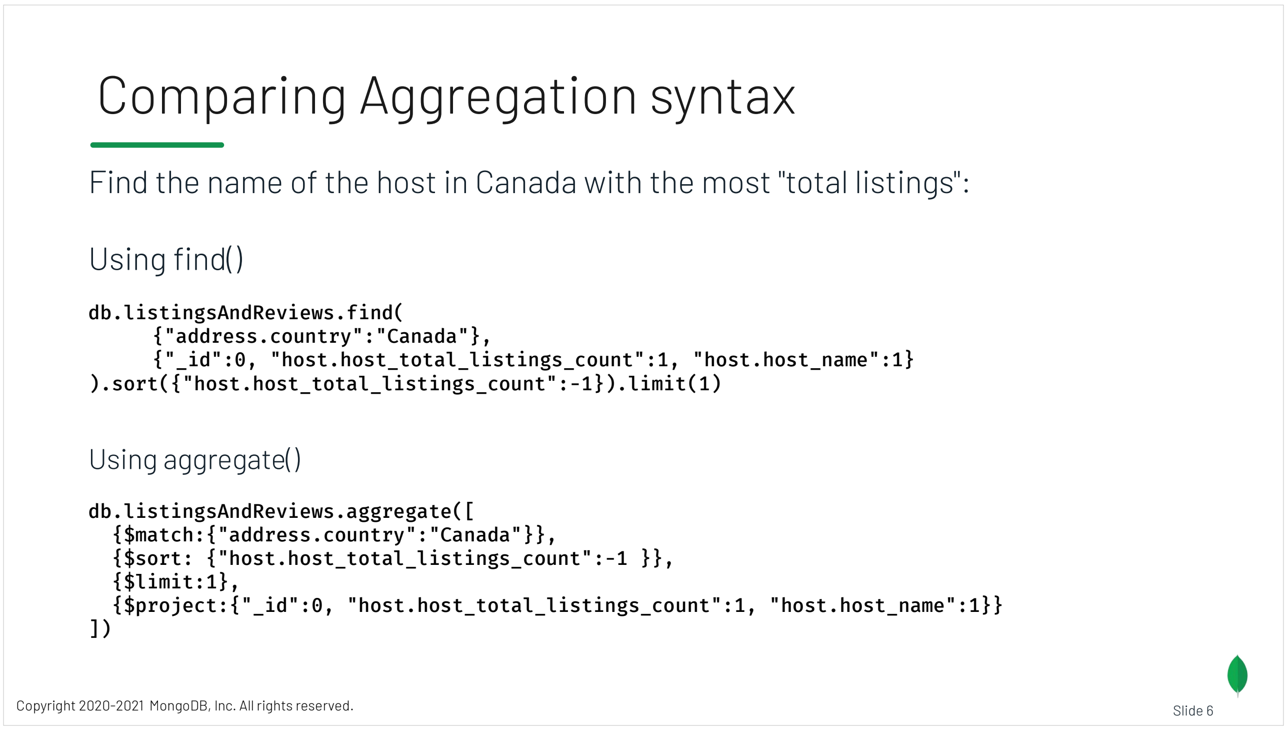


Stages we already know

● $match	equivalent to find(query)
● $project	equivalent to find({},projection)
● $sort	equivalent to find().sort(order)
● $limit	equivalent to find().limit(num)
● $skip	equivalent to find().skip(num)
● $count	equivalent to find().count()

When these are used at the start of a pipeline, they are transformed to a find() by the query optimizer.

The most common stages are $match, $project, $sort, $limit, $skip, $count and $group (which we will learn about later)



Comparing Aggregation syntax

Find the name of the host in Canada with the most "total listings":

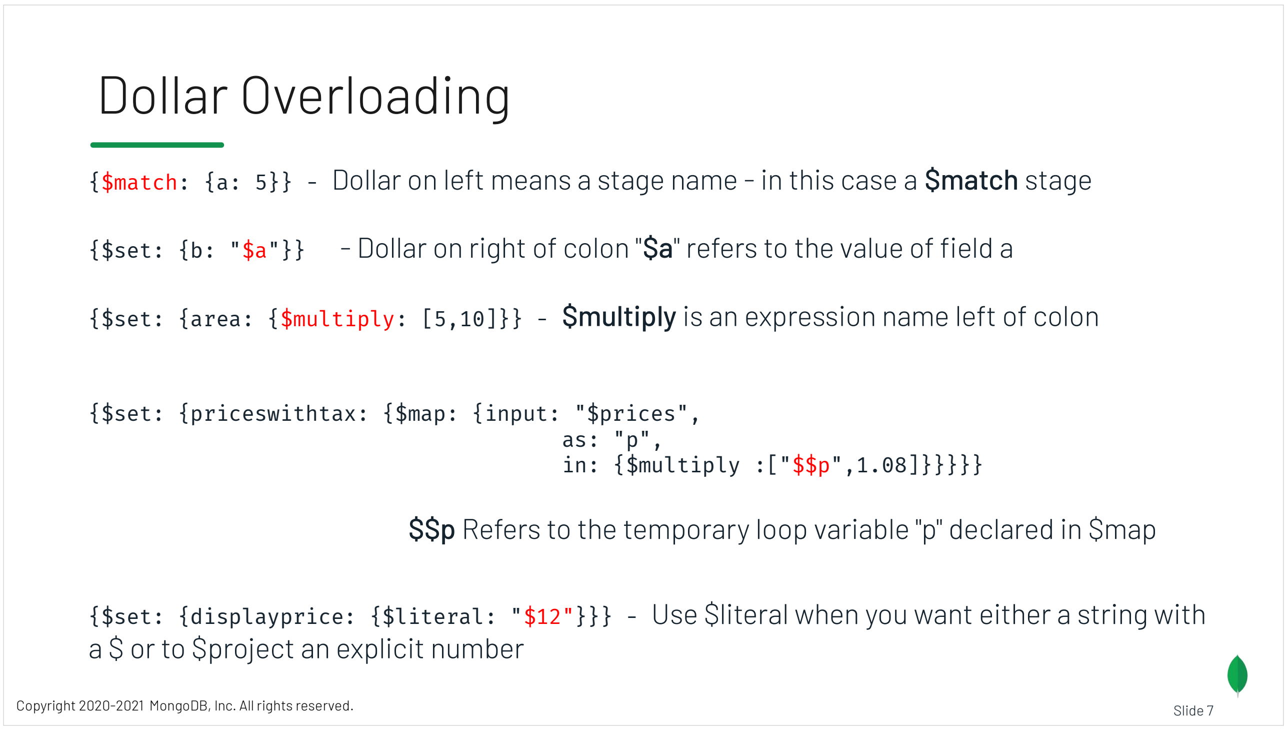
Using find()

```
db.listingsAndReviews.find(
     {"address.country":"Canada"},
     {"_id":0, "host.host_total_listings_count":1, "host.host_name":1}
).sort({"host.host_total_listings_count":-1}).limit(1)
```

Using aggregate()

```
db.listingsAndReviews.aggregate([
  {$match:{"address.country":"Canada"}},
  {$sort: {"host.host_total_listings_count":-1 }},
  {$limit:1},
  {$project:{"_id":0, "host.host_total_listings_count":1, "host.host_name":1}}
])
```

Example



Dollar Overloading

`{$match: {a: 5}}` - Dollar on left means a stage name - in this case a **$match** stage

`{$set: {b: "$a"}}` - Dollar on right of colon "**$a**" refers to the value of field a

`{$set: {area: {$multiply: [5,10]}}` - **$multiply** is an expression name left of colon

```
{$set: {priceswithtax: {$map: {input: "$prices",
                                     as: "p",
                                     in: {$multiply :["$$p",1.08]}}}}}
```

$$p Refers to the temporary loop variable "p" declared in $map

`{$set: {displayprice: {$literal: "$12"}}}` - Use $literal when you want either a string with a $ or to $project an explicit number

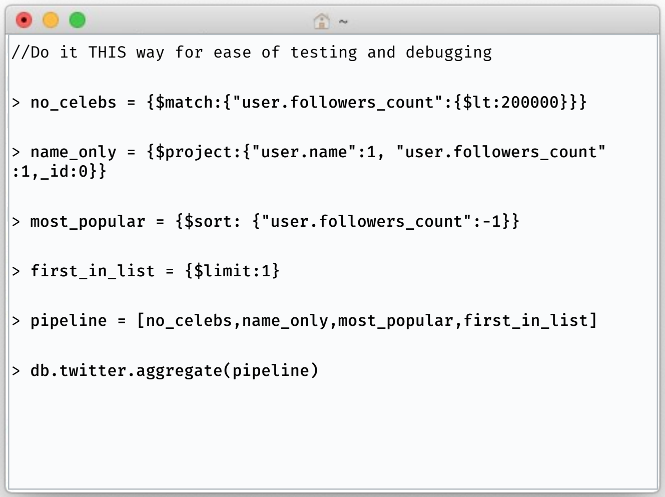
- Dollars are used in a number of ways in aggregation syntax, this can be confusing for a beginner
  - On the Left They refer to a stage or expression name
  - On the Right they refer to a field value
  - Double dollars refer to a temporary variable or constant
  - You can also use $literal if you need a dollar symbol as a variable.

How to write Aggregations

Think as a programmer - not as a database shell.

Define variables. Doing this helps you keep track of brackets.

It also helps you really understand the Object concept.



```
//Do it THIS way for ease of testing and debugging

> no_celebs = {$match:{"user.followers_count":{$lt:200000}}}

> name_only = {$project:{"user.name":1, "user.followers_count"
:1,_id:0}}

> most_popular = {$sort: {"user.followers_count":-1}}

> first_in_list = {$limit:1}

> pipeline = [no_celebs,name_only,most_popular,first_in_list]

> db.twitter.aggregate(pipeline)
```

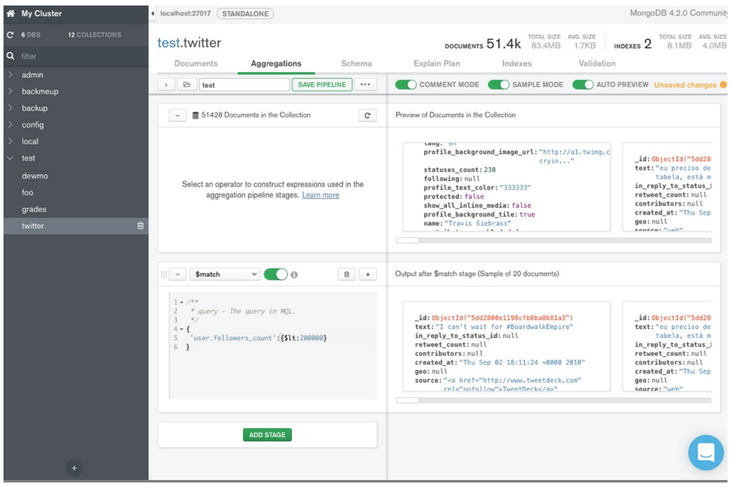
Write aggregations in variables and add these to your pipeline to make testing and debugging easier.

GUI Aggregation Builder

Compass is MongoDB's GUI query tool.

It has an Aggregation Builder and visualizer.

It is helpful when learning but limits your thinking - be careful.



Select an operator to construct expressions used in the aggregation pipeline stages. Learn more

```
/**
 * query - The query in MQL.
 */
{
  'user.followers_count':{$lt:200000}
}
```

```
profile_background_image_url:"http://a1.twimg.c
                             cryin..."
statuses_count:238
following:null
profile_text_color:"333333"
protected:false
show_all_inline_media:false
profile_background_tile:true
name:"Travis Siebrass"
```

```
_id:ObjectId("5dd28
text:"eu preciso de
     tabela, está m
in_reply_to_status_i
retweet_count:null
contributors:null
created_at:"Thu Sep
geo:null
```

```
_id:ObjectId("5dd2800e1198cfb8ba8b81a3")
text:"I can't wait for #BoardwalkEmpire"
in_reply_to_status_id:null
retweet_count:null
contributors:null
created_at:"Thu Sep 02 18:11:24 +0000 2010"
geo:null
source:"<a href="http://www.tweetdeck.com"
       rel="nofollow">TweetDeck</a>"
```

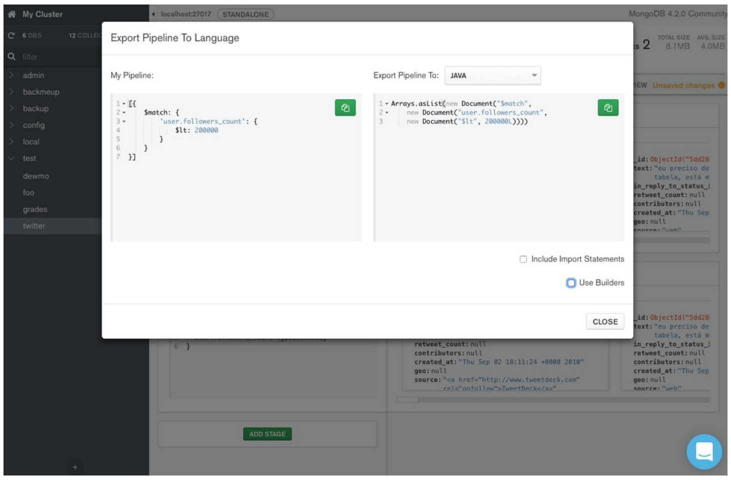
```
_id:ObjectId("5dd28
text:"eu preciso de
     tabela, está m
in_reply_to_status_i
retweet_count:null
contributors:null
created_at:"Thu Sep
geo:null
source:"web"
```

Compass is MongoDB’s GUI tool, but Atlas also offers some aggregation functionality.

Compass - Code export

Once you write a pipeline in compass - it can generate code for you.

Multiple languages are supported.



```
[{
    $match: {
        'user.followers_count': {
            $lt: 200000
        }
    }
}]
```

```
Arrays.asList(new Document("$match",
    new Document("user.followers_count",
    new Document("$lt", 200000L))))
```

Compass/Atlas can generate code from aggregation pipelines for some programming languages.

Aggregations are programs

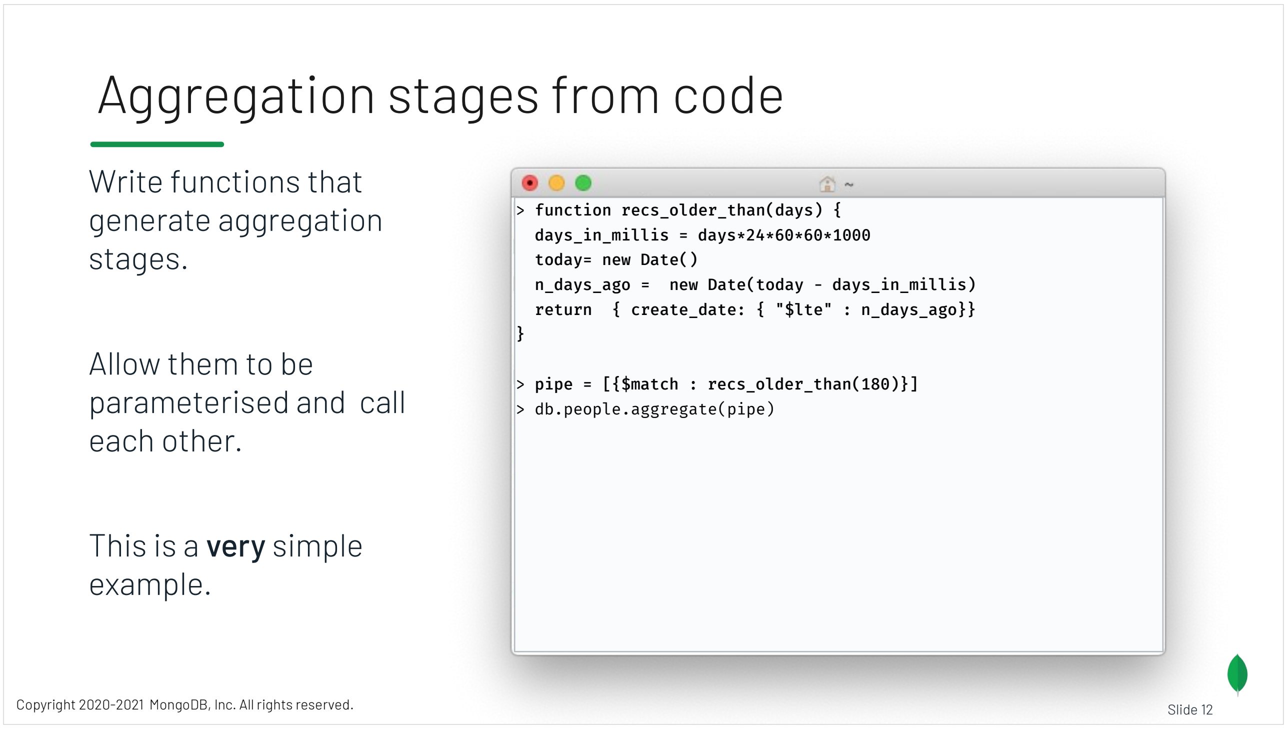
- Initially, you might write simple ones by hand
- You don't often write them ad-hoc (one-off)
- Sometimes you write code to generate them
- Very complex aggregations cannot be written by hand

You might train a neural network outside MongoDB to recognise fraud - then compile it into an aggregation to use in real-time fraud detection. You can do this because aggregation is essentially a programming language

Aggregations are objects forming a metaprogram. They can be written by code as well as by hand.

Many people will only ever have simple - or perhaps slightly complex handwritten aggregations.

However really powerful aggregations can almost not be written by hand, you have to generate them from functions
that then call other functions to generate them. It's difficult to express this idea well but some of the most complex and
powerful aggregation can be >1MB in size - but are created from a combination of input data and machine learning and
recursive functions.



Aggregation stages from code

Write functions that generate aggregation stages.

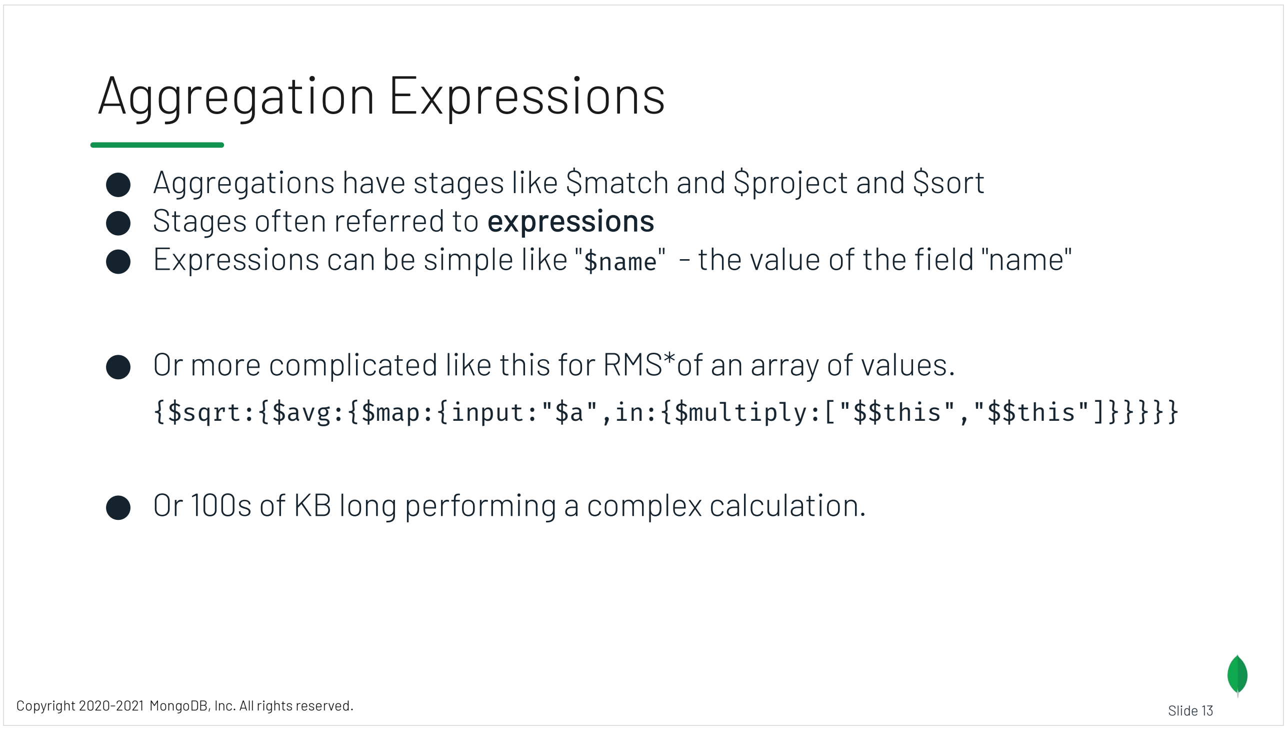
Allow them to be parameterised and call each other.

This is a **very** simple example.

```
> function recs_older_than(days) {
  days_in_millis = days*24*60*60*1000
  today= new Date()
  n_days_ago =  new Date(today - days_in_millis)
  return  { create_date: { "$lte" : n_days_ago}}
}

> pipe = [{$match : recs_older_than(180)}]
> db.people.aggregate(pipe)
```

Normally when you are writing aggregations you are doing it in code - not the shell, because aggregations are objects you can
easily create functions to output part of your aggregation to allow you to easily parameterise them.

This is an important technique in creating complex and powerful aggregations.



Aggregation Expressions

- Aggregations have stages like $match and $project and $sort
- Stages often referred to **expressions**
- Expressions can be simple like "`$name`" - the value of the field "name"

- Or more complicated like this for RMS*of an array of values.

```
{$sqrt:{$avg:{$map:{input:"$a",in:{$multiply:["$$this","$$this"]}}}}}
```

- Or 100s of KB long performing a complex calculation.

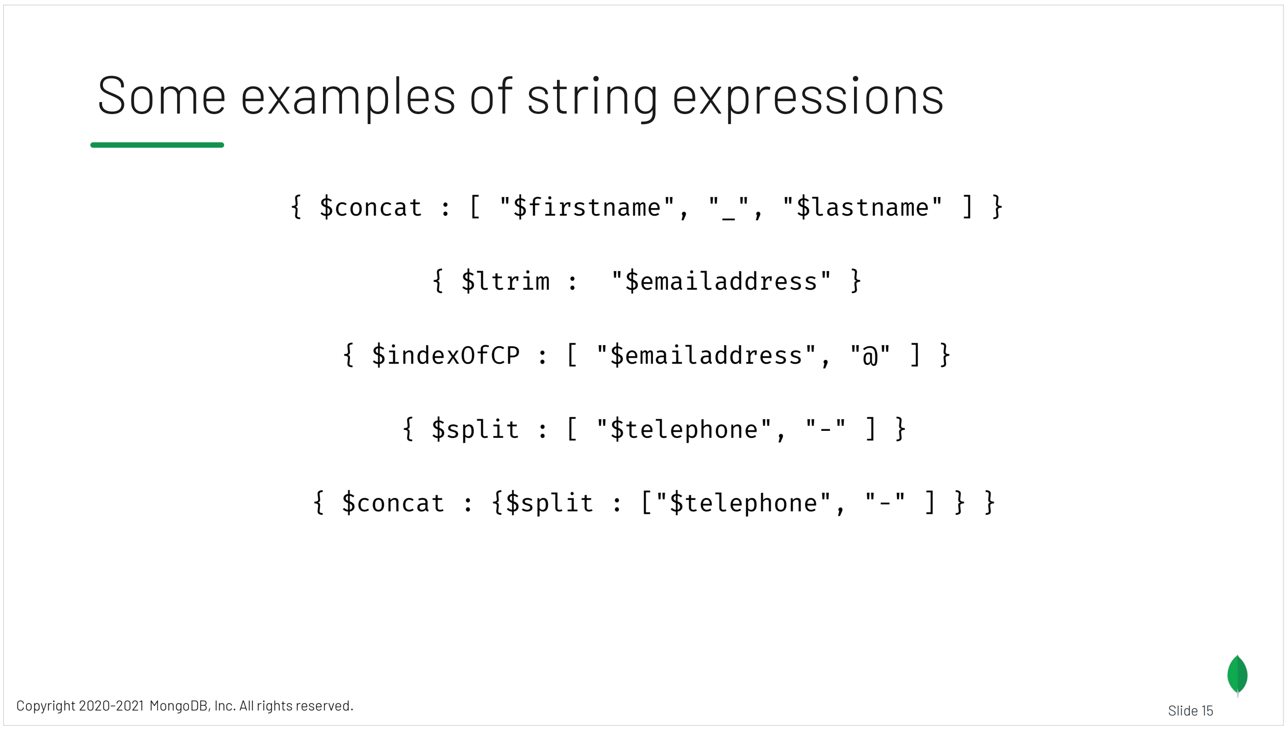
*Root Mean Square is the Square root of the average of the squares of values and is often used in signal processing.

Some examples of arithmetic expressions

```
{ $subtract : [ "$cost", "$price" ] }

{ $add : [ "$price", "$shipping" ] }

{ $add : [ "$price", "$shipping", "$tax" ] }

{ $multiply : [ "$price", "$taxrate" ] }

{ $divide : [ "$quantity", "$price" ] }

{ $add : [ "$price" , { $multiply : ["$price", "$taxrate" ] } ] }
```

There are many type of expressions - if you think of a common function in a programming language then there is likely a matching mongodb expressions. Even simple things like basic maths are expressed like this, one advantage over normal inline operators like + and * is that you can
$add a list of things.

Expression arguments are always an array.
If there is only one argument you can give it as a scalar and MongoDB converts it to an Array of one element behind the scenes



Some examples of string expressions

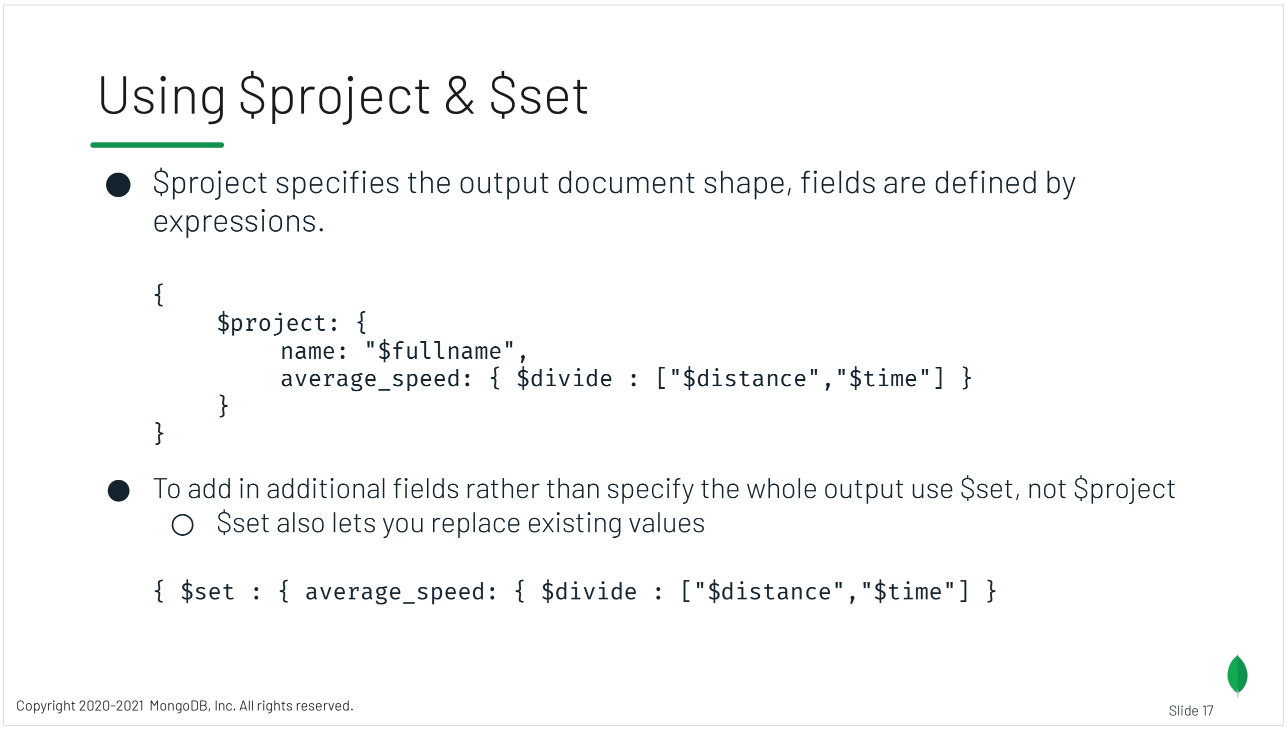
```
{ $concat : [ "$firstname", "_", "$lastname" ] }

{ $ltrim :  "$emailaddress" }

{ $indexOfCP : [ "$emailaddress", "@" ] }

{ $split : [ "$telephone", "-" ] }

{ $concat : {$split : ["$telephone", "-" ] } }
```

Another common thing you would expect in a language are string expressions.
You can concatenate strings together, remove leading space, search for strings in strings - typical developer things.

Expression Categories

- Arithmetic Expression Operators
- Array Expression Operators
- Boolean Expression Operators
- Comparison Expression Operators
- Conditional Expression Operators
- Date Expression Operators
- Literal Expression Operator
- Object Expression Operators
- Set Expression Operators
- String Expression Operators
- Text Expression Operator
- Trigonometry Expression Operators
- Type Expression Operators
- Accumulators ($group)

There are hundreds of different expressions - like built-in functions in a programming language that you can use to express any data transformation. These are the categories and links to them Some are for specific data types, some, that we see later will let you iterate over arrays some even implement the idea of a mathematical set of unique values.



Using $project & $set

- $project specifies the output document shape, fields are defined by expressions.

```
{
     $project: {
          name: "$fullname",
          average_speed: { $divide : ["$distance","$time"] }
     }
}
```

- To add in additional fields rather than specify the whole output use $set, not $project
  - $set also lets you replace existing values

```
{ $set : { average_speed: { $divide : ["$distance","$time"] }
```

From MongoDB 4.4 - you can use aggregation style projections in find() as well as with aggregate() in a $project stage.
AddFields let's use add or modify a field without having to know all the other fields that are there. It's much less brittle.
A common anti-pattern is to use $project early in your pipeline to 'limit what fields are being processed' - you should not do this because MongoDB can and does work out what fields it needs to take from the database itself and doesn't pass anything into the pipeline that won't help compute the final output. It also limits several of MongoDB's optimization options if you do this.

- $addFields is an older name for $set
- Only use $project as a final stage to format output in production.
  - It useful to project just the fields you want early when developing a query.
  - Never use it early to 'optimize' the data in the pipeline, that's an anti-pattern

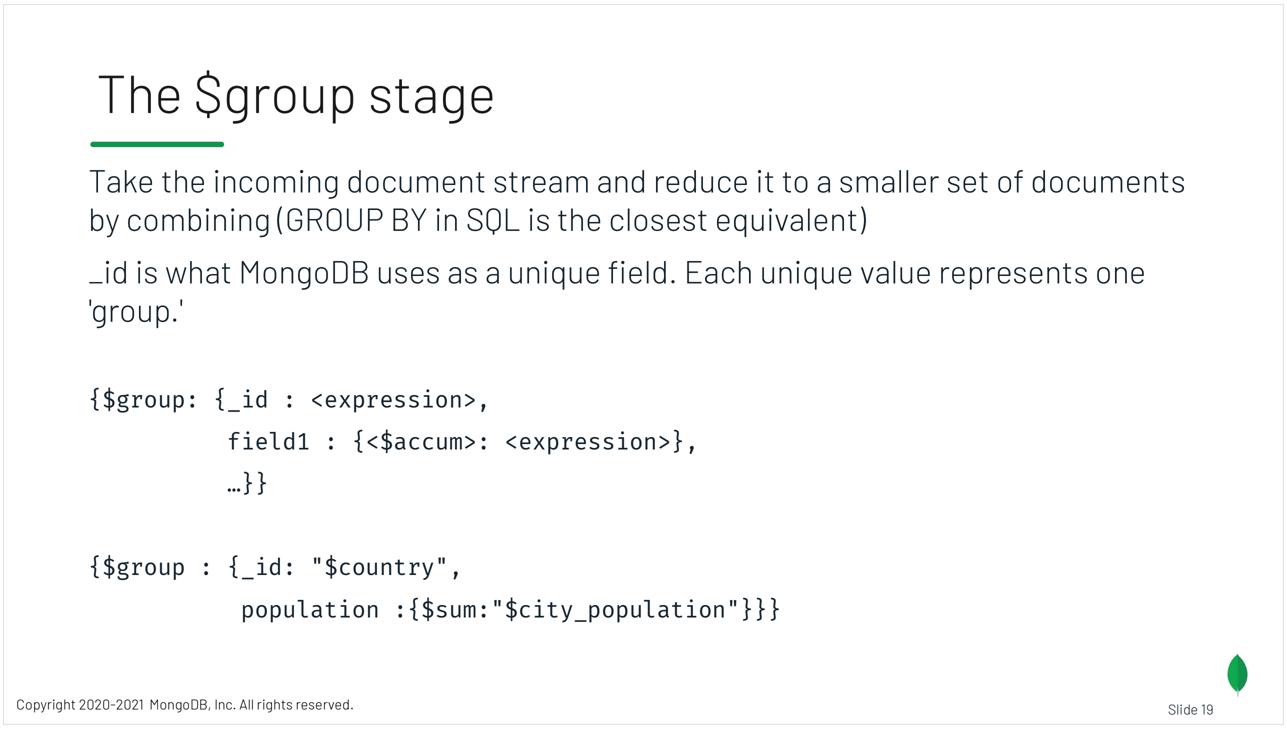
Exercise

Using the Shell or Compass aggregation builder and the airbnb listings data (`sample_airbnb.listingsAndReviews`) read the following question and output the answer. You will need to use a `$set` stage and a `$project` stage with expressions

1. The price for the basic rental is in the `$price` field; for that price, you can have the number of guests provided in `$guests_included` field.
2. The property may take more guests in total (maximum guests is in `$accommodates` property).
3. You have to pay `$extra_people` dollars per person for every person more than `$guests_included`

- Calculate how many extra guests you can have for each property and add that as a field using `$set`
- Calculate how much it would cost with these extra guests. `$project` the basic price and the price if fully occupied with `$accommodates` people.

If you finish this easily you could test out using `$sort` and `$limit` to find out the most expensive property.
We could also use the same techniques to compute the cheapest place to stay per person.
Instead of $set we can also go for $addFields and it would provide us similar results.



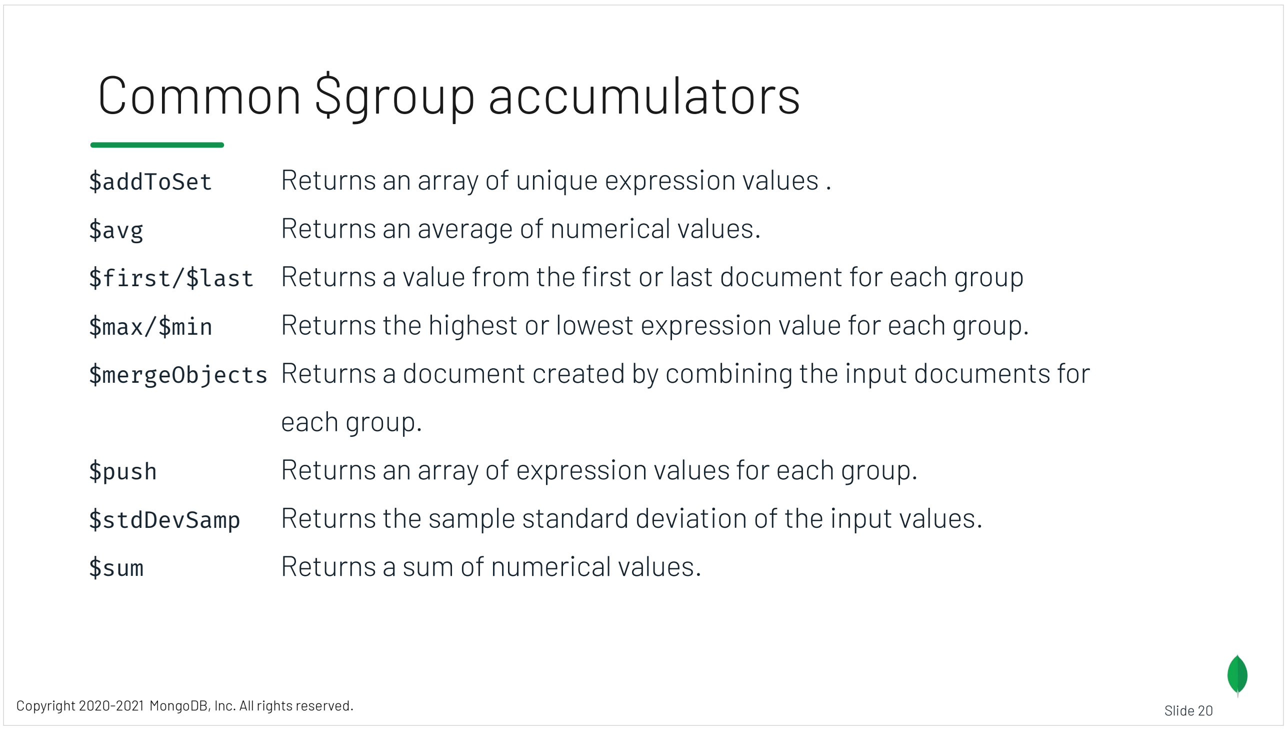
The $group stage

Take the incoming document stream and reduce it to a smaller set of documents by combining (GROUP BY in SQL is the closest equivalent)

_id is what MongoDB uses as a unique field. Each unique value represents one 'group.'

```
{$group: {_id : <expression>,
          field1 : {<$accum>: <expression>},
          …}}

{$group : {_id: "$country",
           population :{$sum:"$city_population"}}}
```

$group with an _id of a constant (like null) to group everything into a total.
Grouping does not use an index - unless the index is used for covering.
If you want the highest (or lowest) value in each group you normally use the $max and $min accumulators
However, if you use $sort and then $first or $last instead and this is at the start of an aggregation MongoDB can optimise picking these values from the index
This is a common case to "Get the largest in each group" and so is worth knowing this optimization.



Common $group accumulators

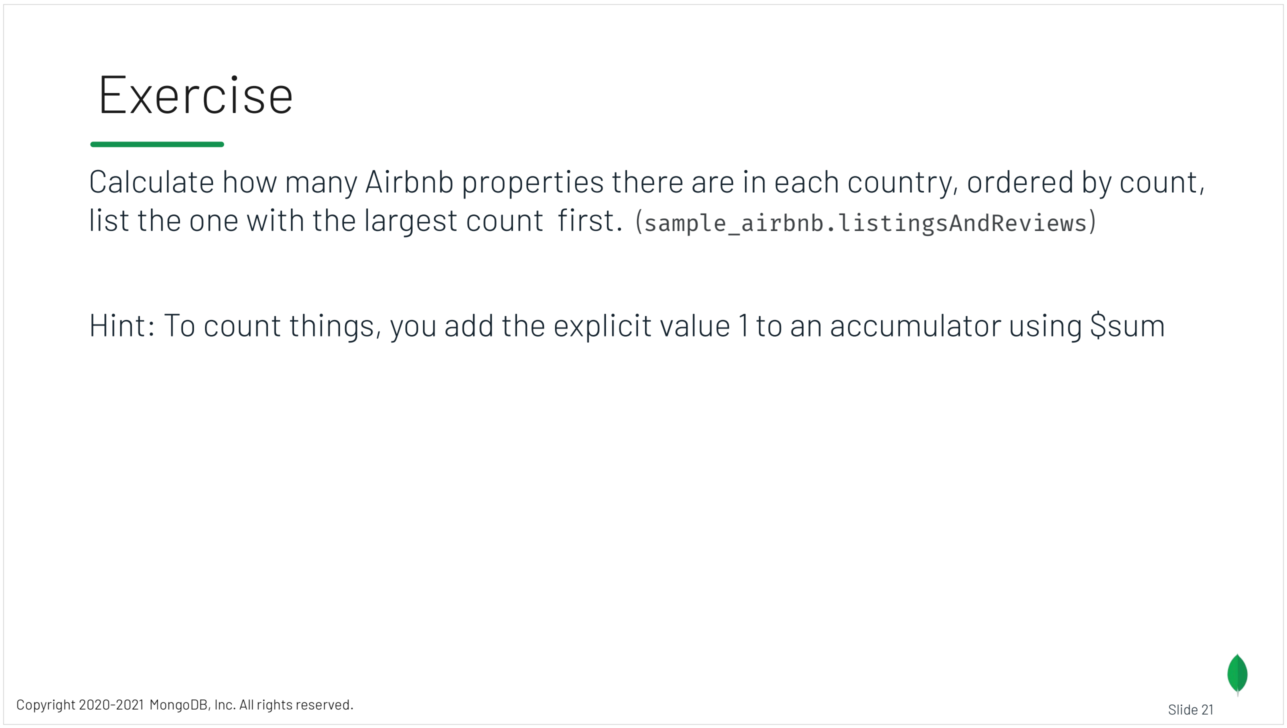
`$addToSet`	Returns an array of unique expression values .
`$avg`	Returns an average of numerical values.
`$first/$last`	Returns a value from the first or last document for each group
`$max/$min`	Returns the highest or lowest expression value for each group.
`$mergeObjects`	Returns a document created by combining the input documents for each group.
`$push`	Returns an array of expression values for each group.
`$stdDevSamp`	Returns the sample standard deviation of the input values.
`$sum`	Returns a sum of numerical values.

As well as being usable in $group - some of these can simply be applied to an array in a $project or $set.

Also, as of 4.4, there is the ability to write your own javascript accumulator expressions - DO NOT DO THIS - Javascript is much much slower than native.

$first and $last imply data is sorted.

https://docs.mongodb.com/manual/reference/operator/aggregation/group/#accumulators-group

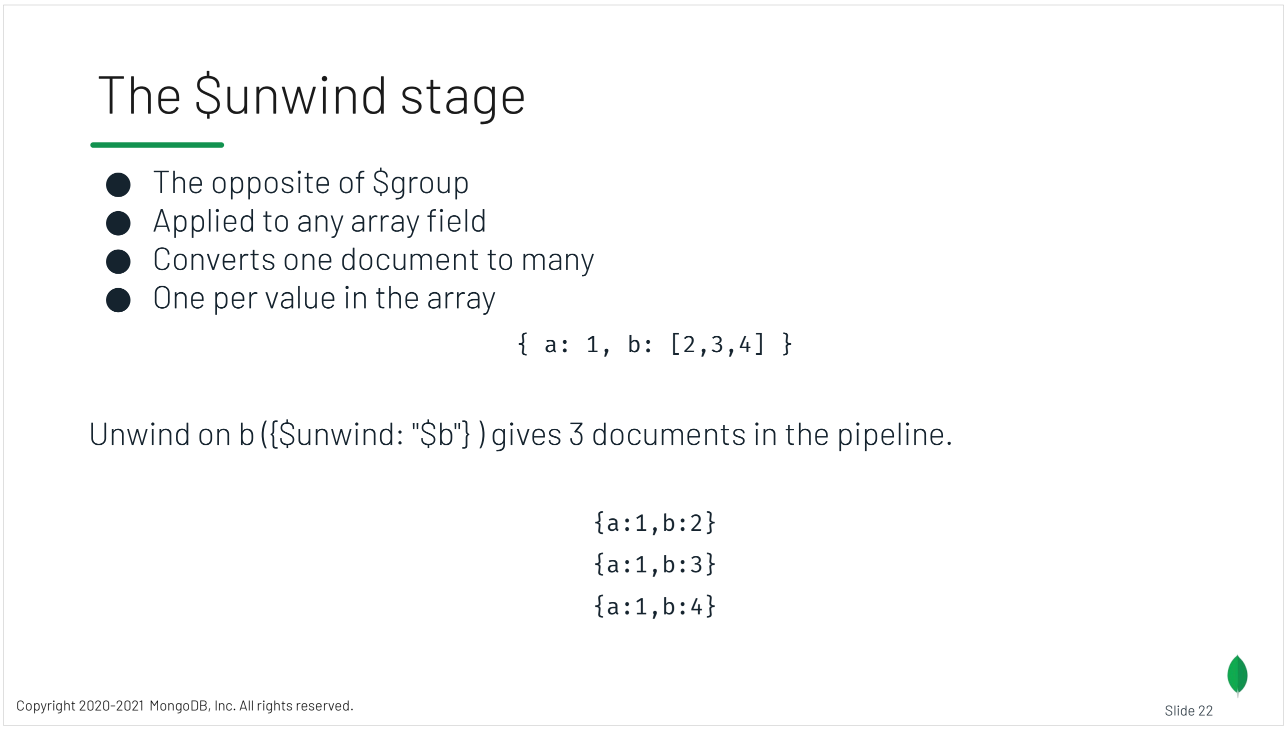


Exercise

Calculate how many Airbnb properties there are in each country, ordered by count, list the one with the largest count first. (`sample_airbnb.listingsAndReviews`)

Hint: To count things, you add the explicit value 1 to an accumulator using $sum

Exercise



The $unwind stage

- The opposite of $group
- Applied to any array field
- Converts one document to many
- One per value in the array

```
{ a: 1, b: [2,3,4] }
```

Unwind on b ({$unwind: "$b"}) gives 3 documents in the pipeline.

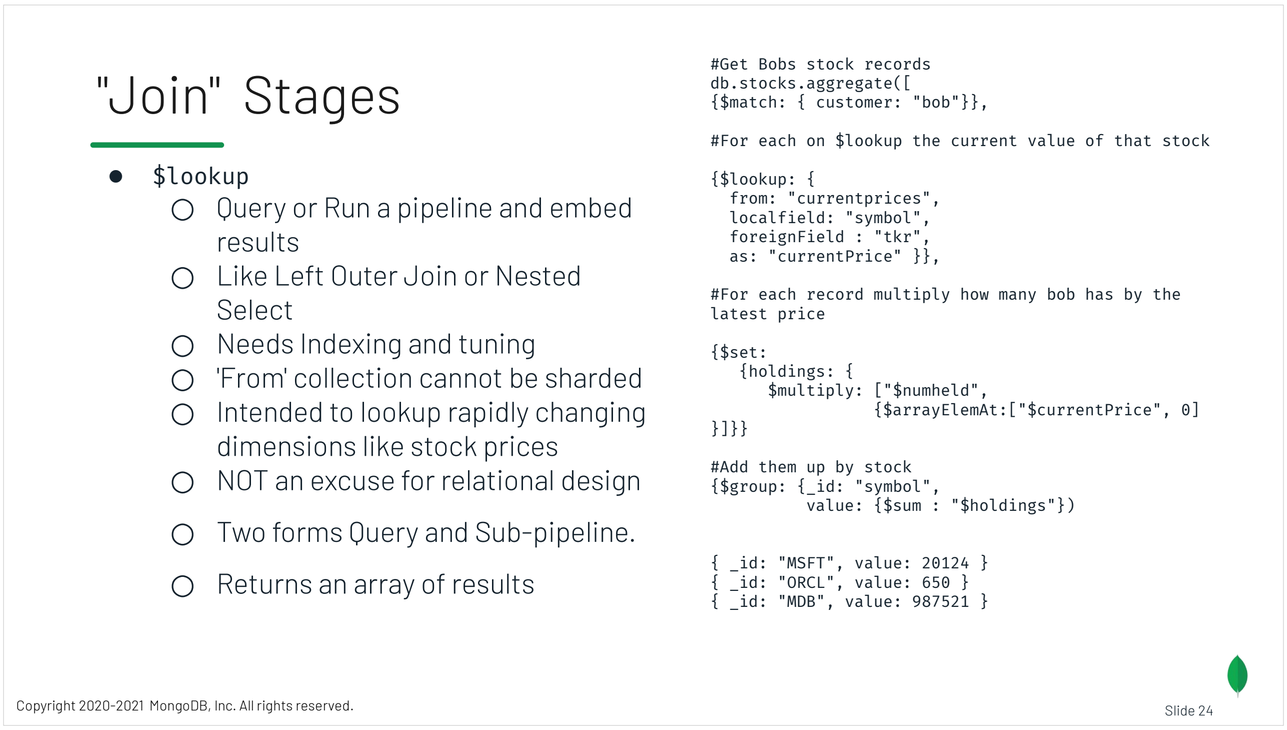
```
{a:1,b:2}
{a:1,b:3}
{a:1,b:4}
```

$unwind is opposite of group and allows you to expand arrays out to separate documents

Exercise with $unwind

What amenities are offered in the smallest number of countries?

Exercise



"Join" Stages

- `$lookup`
  - Query or Run a pipeline and embed results
  - Like Left Outer Join or Nested Select
  - Needs Indexing and tuning
  - 'From' collection cannot be sharded
  - Intended to lookup rapidly changing dimensions like stock prices
  - NOT an excuse for relational design
  - Two forms Query and Sub-pipeline.
  - Returns an array of results

```
#Get Bobs stock records
db.stocks.aggregate([
{$match: { customer: "bob"}},

#For each on $lookup the current value of that stock

{$lookup: {
  from: "currentprices",
  localfield: "symbol",
  foreignField : "tkr",
  as: "currentPrice" }},

#For each record multiply how many bob has by the
latest price

{$set:
   {holdings: {
      $multiply: ["$numheld",
                 {$arrayElemAt:["$currentPrice", 0]
}]}}

#Add them up by stock
{$group: {_id: "symbol",
          value: {$sum : "$holdings"})


{ _id: "MSFT", value: 20124 }
{ _id: "ORCL", value: 650 }
{ _id: "MDB", value: 987521 }
```

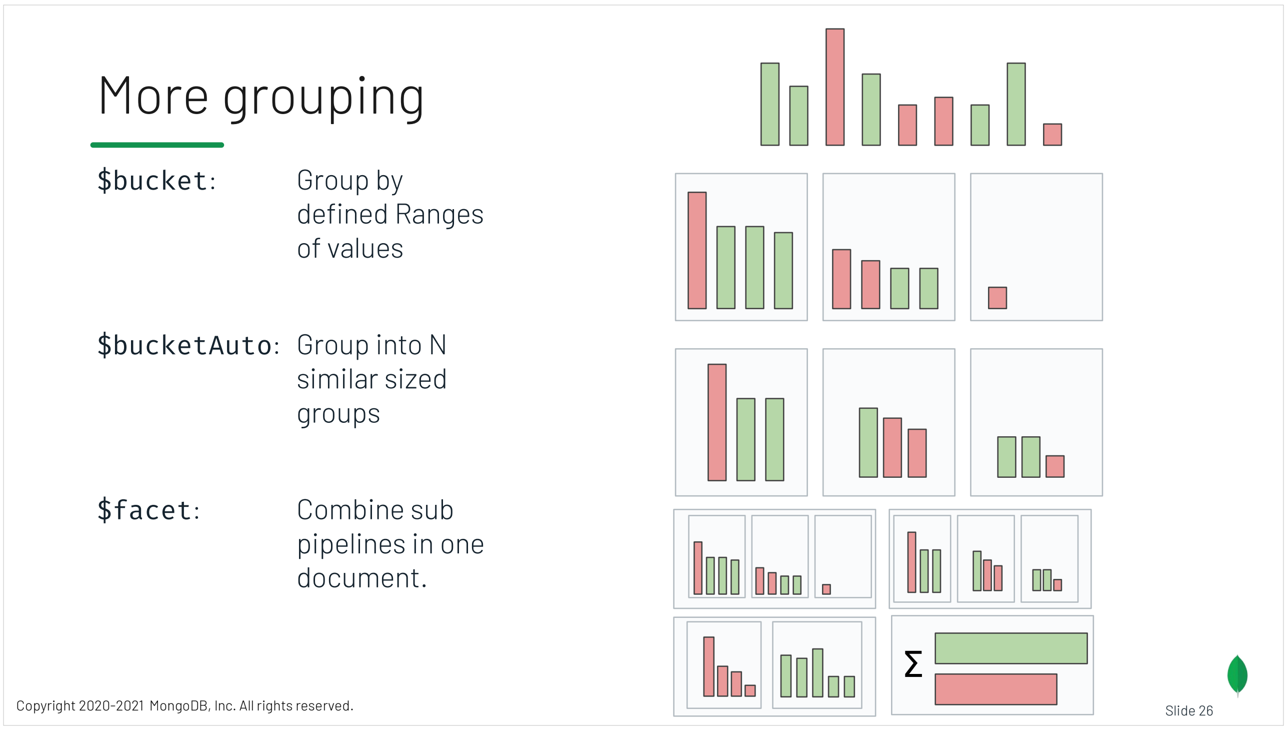
`$lookup` is like a Left outer Join, however, it is not a reason to use MongoDB like a relational database. There is always a cost when joining data so good document schema design is more important.
Sometimes you want to pull in data that changes very rapidly and you don't want to make huge edits all the time - in that case $lookup is good.

"Join" Stages

- `$graphLookup`
  - Recursively lookup on same collection
  - A way to traverse trees or graphs
  - Rarely useful in practice as both limited in functionality and relatively slow.
  - There are better schema design patterns for most use cases.

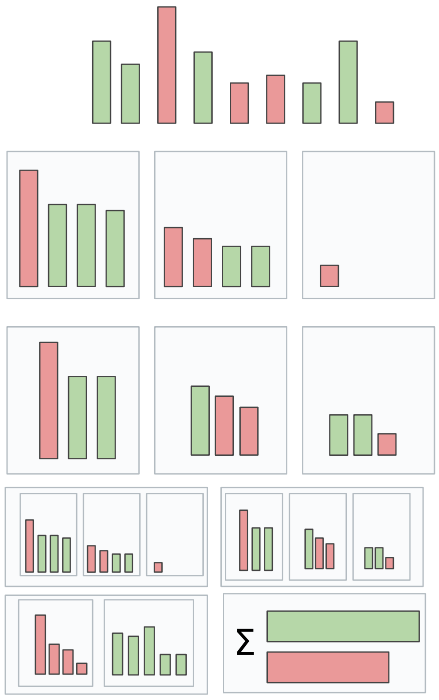
```
db.landmarks.aggregate([
{ $match: { _id: "Brooklyn Bridge" }},
{ $graphLookup: {
         from: "places",
         startWith: "$location",
         connectFromField: "isIn",
         connectToField: "_id",
         as: "address"
      }}])
```

```
Returns:

{ _id: "Brooklyn Bridge",
  location: "Brooklyn",
  address : [ { _id: "Brooklyn", isIn: "NYC"},
              { _id: "NYC", isIn: "New York" },
              { _id: "New York", isIn: "USA" }]


}
```



More grouping

`$bucket`:	Group by defined Ranges of values
`$bucketAuto`:	Group into N similar sized groups
`$facet`:	Combine sub pipelines in one document.



- $bucket is a more specialized form of $group
  - Group into ranges of values without additional expression to round them
  - For example group ages 0-10,10-20,20-30
- $bucketAuto
  - Determine boundaries of buckets automatically
  - Segment data set into same-sized groups
  - Lots of statistical options
- $facet
  - Run multiple sub aggregations and include in a single document
  - Efficient way to run multiple final stages
  - Limited by document size
  - Often used with $sortByCount to find largest group.

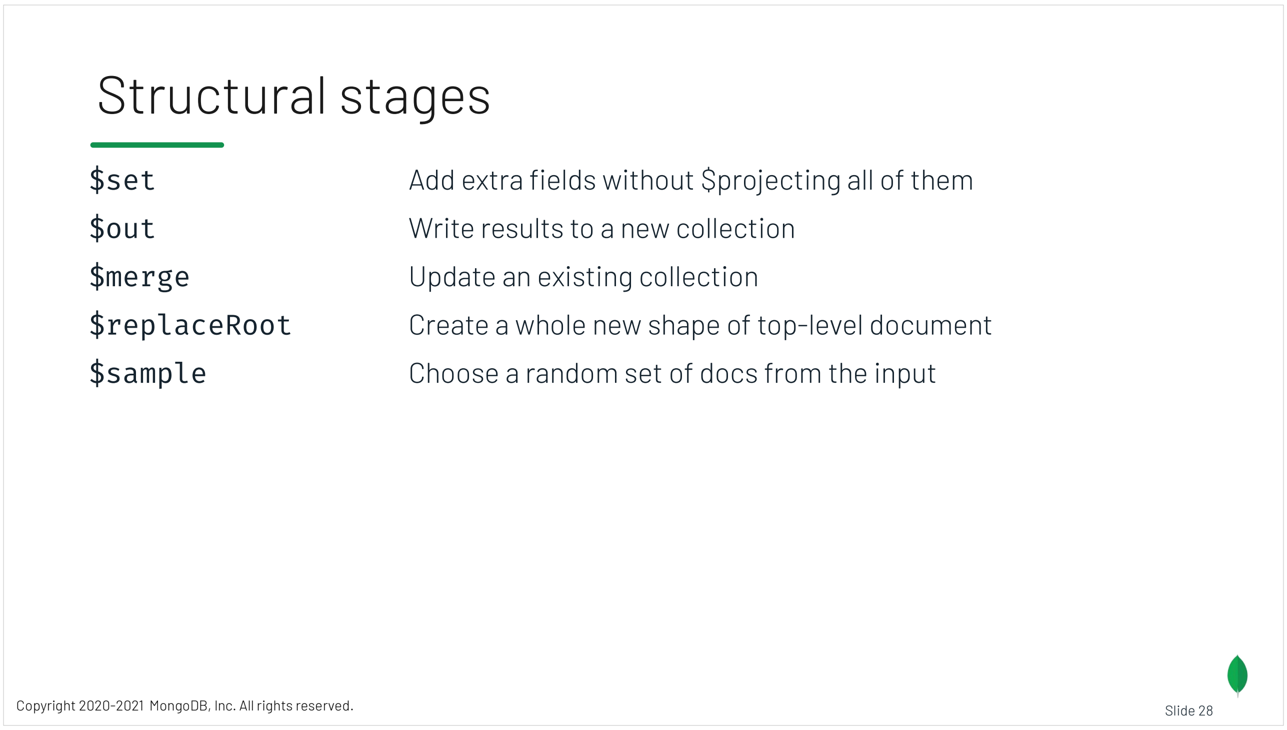
Grouping: $sortByCount

Shortcut for one of the most popular groupings to do

See what the most common values are of a field.

```
[{
        $group: {
            id: "$city" ,
            count: {
                $sum: 1
            }
        }
    },
    {
        $sort: {
            count: -1
        }
    }
]
```

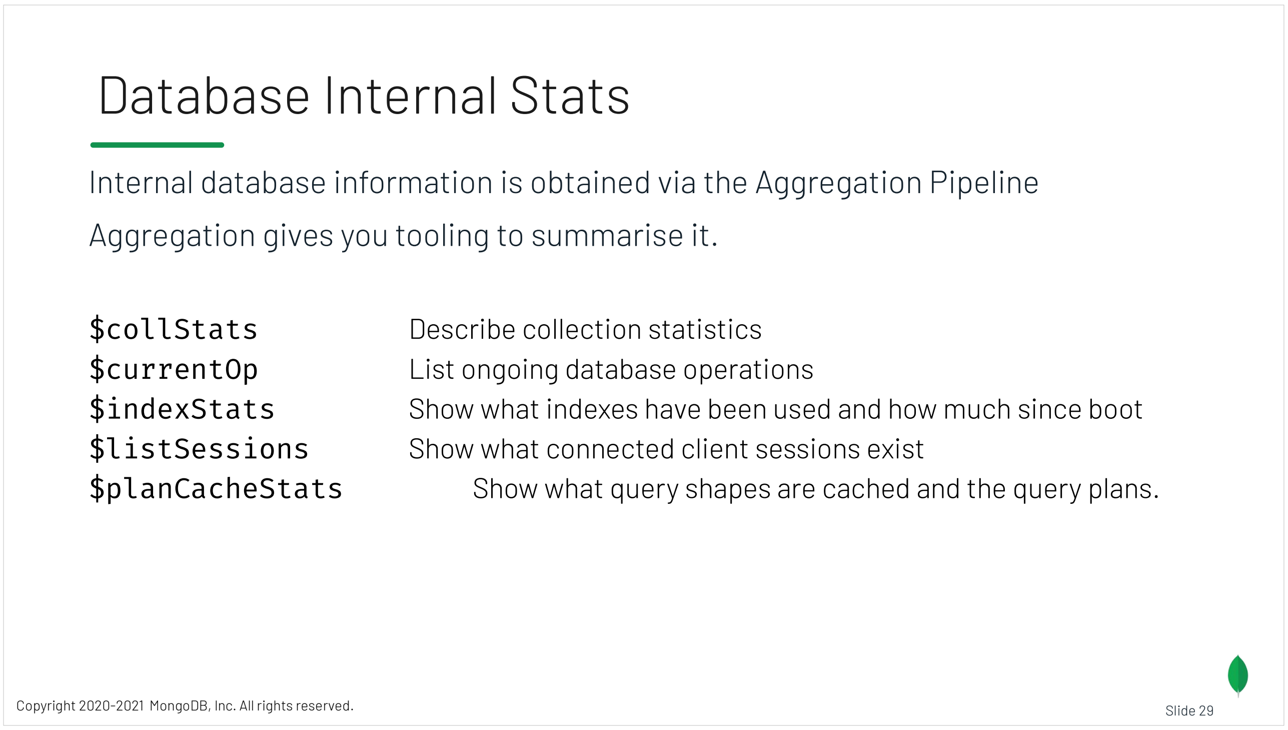
```
{ $sortByCount:  "$city" }
```



Structural stages

`$set`	Add extra fields without $projecting all of them
`$out`	Write results to a new collection
`$merge`	Update an existing collection
`$replaceRoot`	Create a whole new shape of top-level document
`$sample`	Choose a random set of docs from the input

$set is really useful as we can change an existing field in the pipeline or add one without needing to know all the others - before we had this we had to use $project.
$out writes to a new collection, it can overwrite an existing one but not the one you are reading from. It writes to a temporary collection then renames, so is never see half written.
$merge lets you create a stream of insert/update/delete operations from a pipeline and apply them to a collection - similar to materialized views.
$replaceRoot is used when you want to take an object you have created and make it be the whole document rather than a field in it.
$sample uses an efficient method to grab random, non repeating documents. You can set the sample size,.



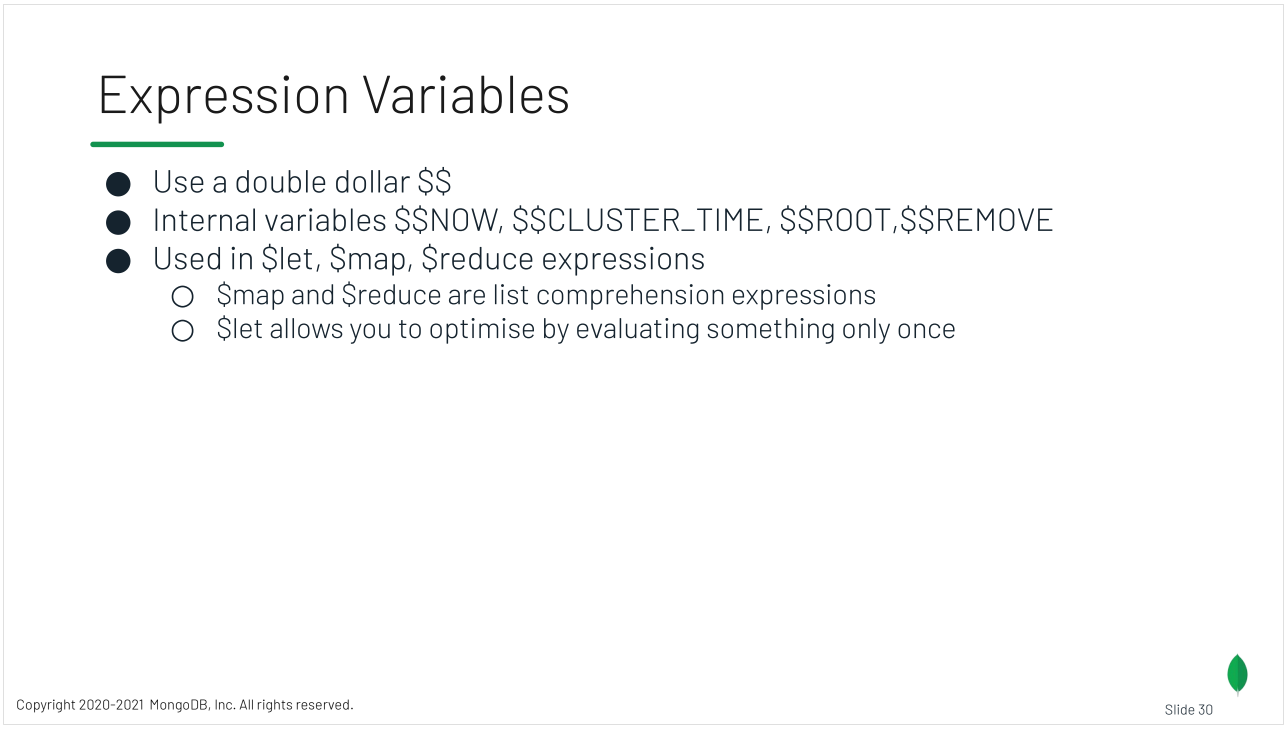
Database Internal Stats

Internal database information is obtained via the Aggregation Pipeline

Aggregation gives you tooling to summarise it.

`$collStats`	Describe collection statistics
`$currentOp`	List ongoing database operations
`$indexStats`	Show what indexes have been used and how much since boot
`$listSessions`	Show what connected client sessions exist
`$planCacheStats`	Show what query shapes are cached and the query plans.

- $collStats shows us information about collection sizes and number of records.
- $currentOp is used to see what is happening on the database instance
- $indexStats shows size and usage of indexes
- $listSessions shows logged in users
- $planCacheStats shows how the query planner is working..



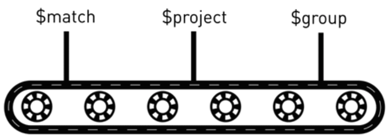
Expression Variables

- Use a double dollar $$
- Internal variables $$NOW, $$CLUSTER_TIME, $$ROOT,$$REMOVE
- Used in $let, $map, $reduce expressions
  - $map and $reduce are list comprehension expressions
  - $let allows you to optimise by evaluating something only once

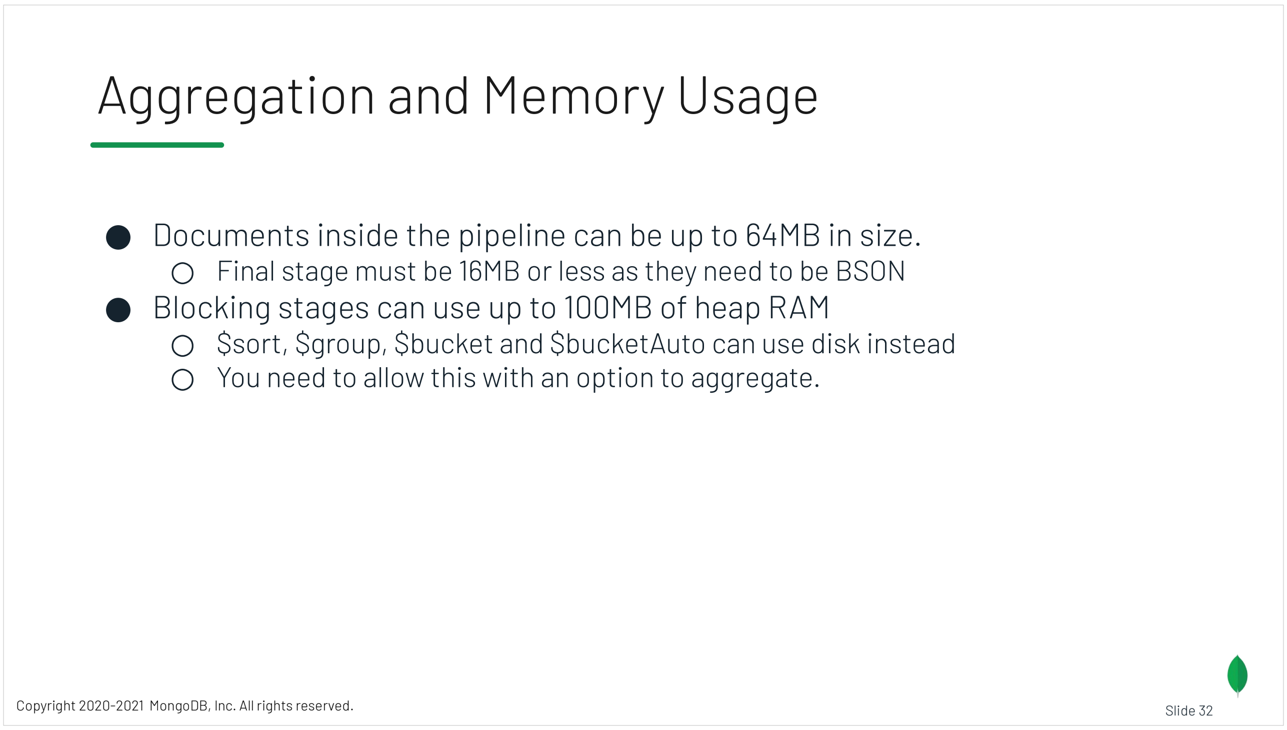
Expression variables use $$ syntax

Aggregation Optimization

- The pipeline will merge stages and reorder as needed
- The pipeline will work out the required early projection; **do not $project to optimize.**
- $unwind->$group(_id:"$_id") is an anti-pattern, use projection accumulators
- Stages can be Streaming or Blocking
  - $sort (no index) ,$group,$bucket,$facet block next step until complete
  - a blocking stage removes parallelism
  - Try to run in parallel as much as possible.



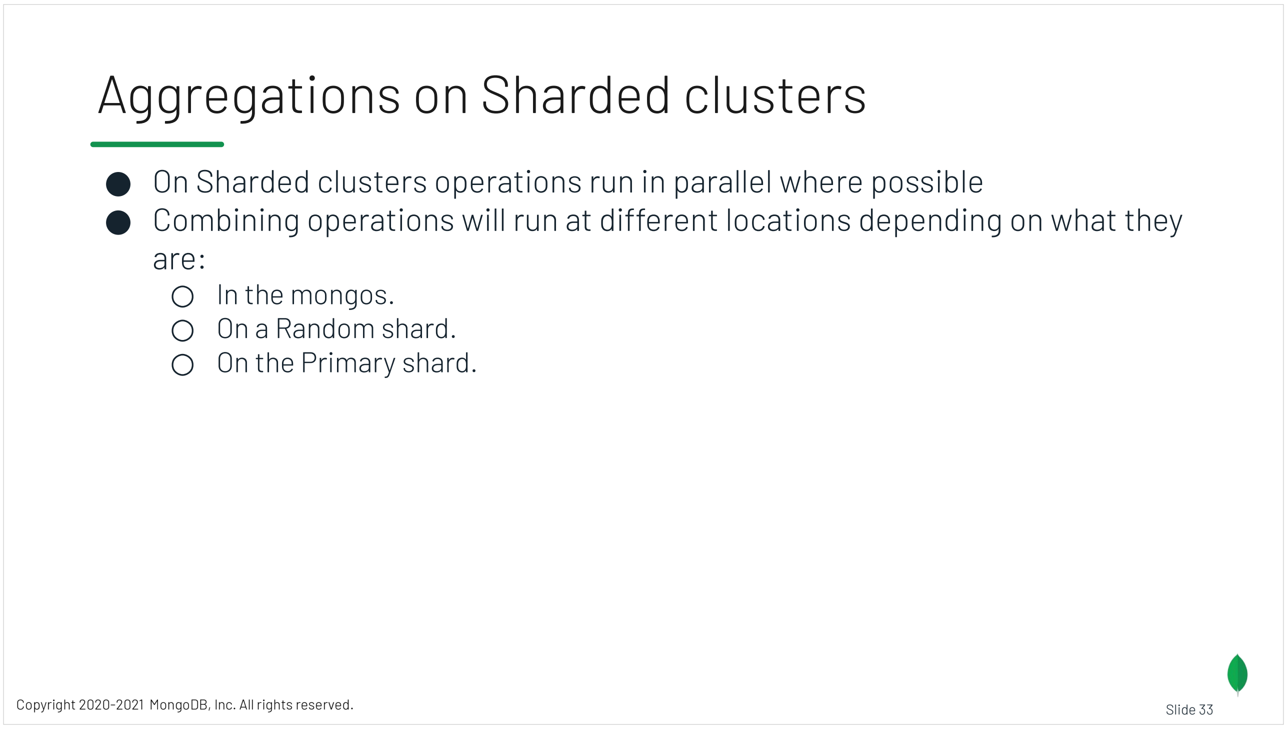
The MDB engine will attempt to optimize aggregations and reorder stages where possible.
Do not unwind an array to work on it, then regroup it - apply the operators to the array directly.



Aggregation and Memory Usage

- Documents inside the pipeline can be up to 64MB in size.
  - Final stage must be 16MB or less as they need to be BSON
- Blocking stages can use up to 100MB of heap RAM
  - $sort, $group, $bucket and $bucketAuto can use disk instead
  - You need to allow this with an option to aggregate.

Aggregation uses Heap Memory - separate from the Database Cache.
Internally Aggregation does not use BSON and so internal temporary documents can be more then 16MB
Documents at the end of the pipeline must be 16MB or less though.
There is a 100MB limit for RAM usage in blocking stages like Group, Sort and GraphLookup
You can get round this by allowing Aggregation to use the Disk - passing the option
`allowDiskUse: true`

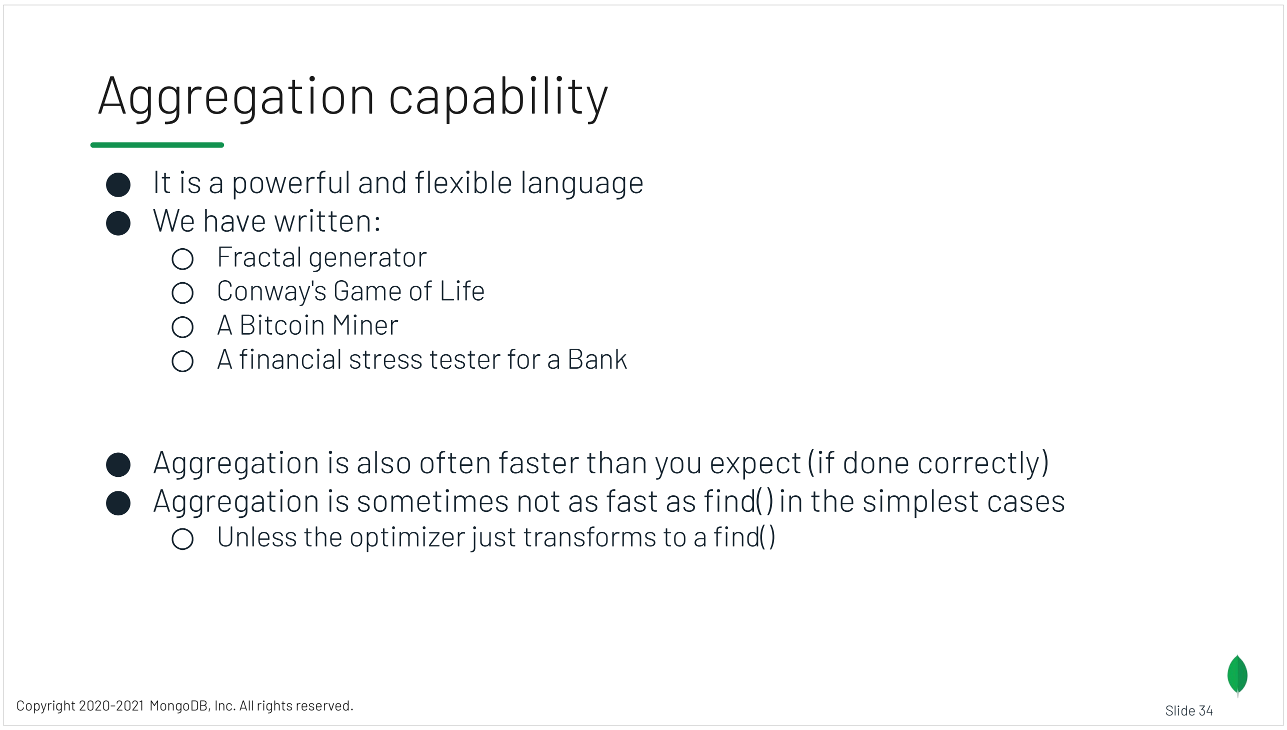


Aggregations on Sharded clusters

- On Sharded clusters operations run in parallel where possible
- Combining operations will run at different locations depending on what they are:
  - In the mongos.
  - On a Random shard.
  - On the Primary shard.

When aggregation operations run on multiple shards, the results are routed to the mongos to be merged, except in the following cases:

- If the pipeline includes the $out or $lookup stages, the merge runs on the primary shard.
- If the pipeline includes a sorting or grouping stage, and the allowDiskUse setting is enabled, the merge runs on a randomly-selected shard.



Aggregation capability

- It is a powerful and flexible language
- We have written:
  - Fractal generator
  - Conway's Game of Life
  - A Bitcoin Miner
  - A financial stress tester for a Bank

- Aggregation is also often faster than you expect (if done correctly)
- Aggregation is sometimes not as fast as find() in the simplest cases
  - Unless the optimizer just transforms to a find()

- The aggregation framework is a powerful tool.
- It has been used to write a Bitcoin Miner.

Recap

- Aggregation is a way to manipulate records inside the server before returning them.
- It is a full programming language in a mostly functional paradigm.
- It can filter, summarise and calculate almost anything.
- It does take time to learn and master - but it's worth it.

Final Exercise

- Aggregation is a large topic with many stages and expressions
- Today we have introduced you to it but, like any other language it takes time and practice to become good.
- Here is a final, more challenging exercise to begin that journey.

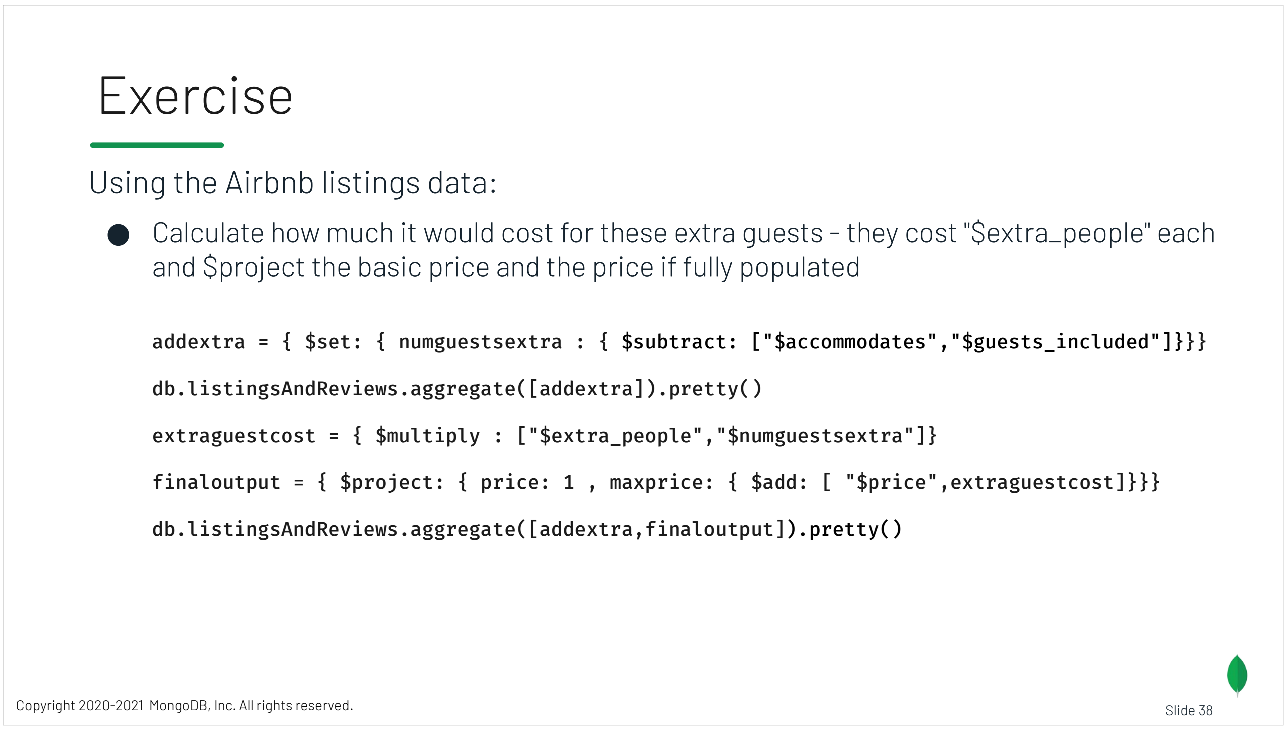
Using the Airbnb listings data:

- What are the 10 most common words in review comments that have more than 6 characters? A word is the same word regardless of case.

- How many unique reviews does each word appear in?
- https://docs.mongodb.com/manual/meta/aggregation-quick-reference/#aggregation-expressions

Exercise

Answers



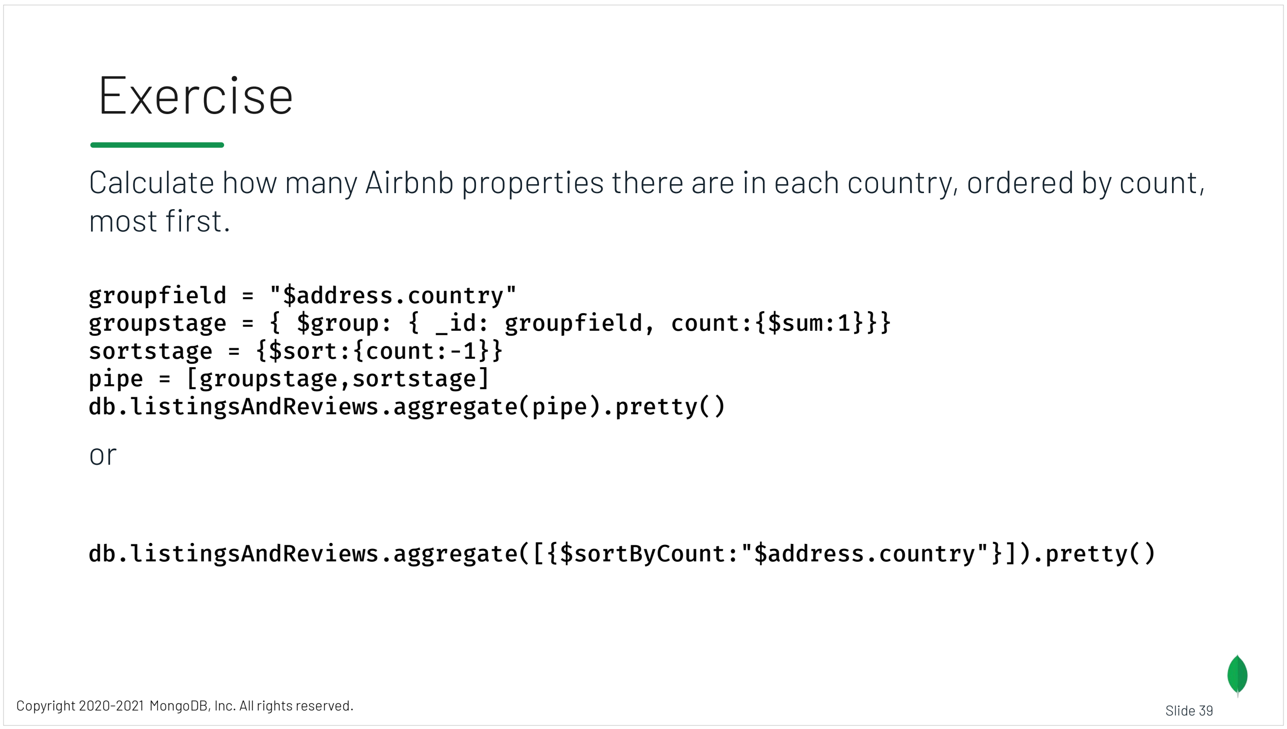
Exercise

Using the Airbnb listings data:

- Calculate how much it would cost for these extra guests - they cost "$extra_people" each and $project the basic price and the price if fully populated

```
addextra = { $set: { numguestsextra : { $subtract: ["$accommodates","$guests_included"]}}}

db.listingsAndReviews.aggregate([addextra]).pretty()

extraguestcost = { $multiply : ["$extra_people","$numguestsextra"]}

finaloutput = { $project: { price: 1 , maxprice: { $add: [ "$price",extraguestcost]}}}

db.listingsAndReviews.aggregate([addextra,finaloutput]).pretty()
```

Data quality is an issue, so you get some wacky answers.



Exercise

Calculate how many Airbnb properties there are in each country, ordered by count, most first.

```
groupfield = "$address.country"
groupstage = { $group: { _id: groupfield, count:{$sum:1}}}
sortstage = {$sort:{count:-1}}
pipe = [groupstage,sortstage]
db.listingsAndReviews.aggregate(pipe).pretty()
```

or

```
db.listingsAndReviews.aggregate([{$sortByCount:"$address.country"}]).pretty()
```

Exercise Answer

Exercise with $unwind

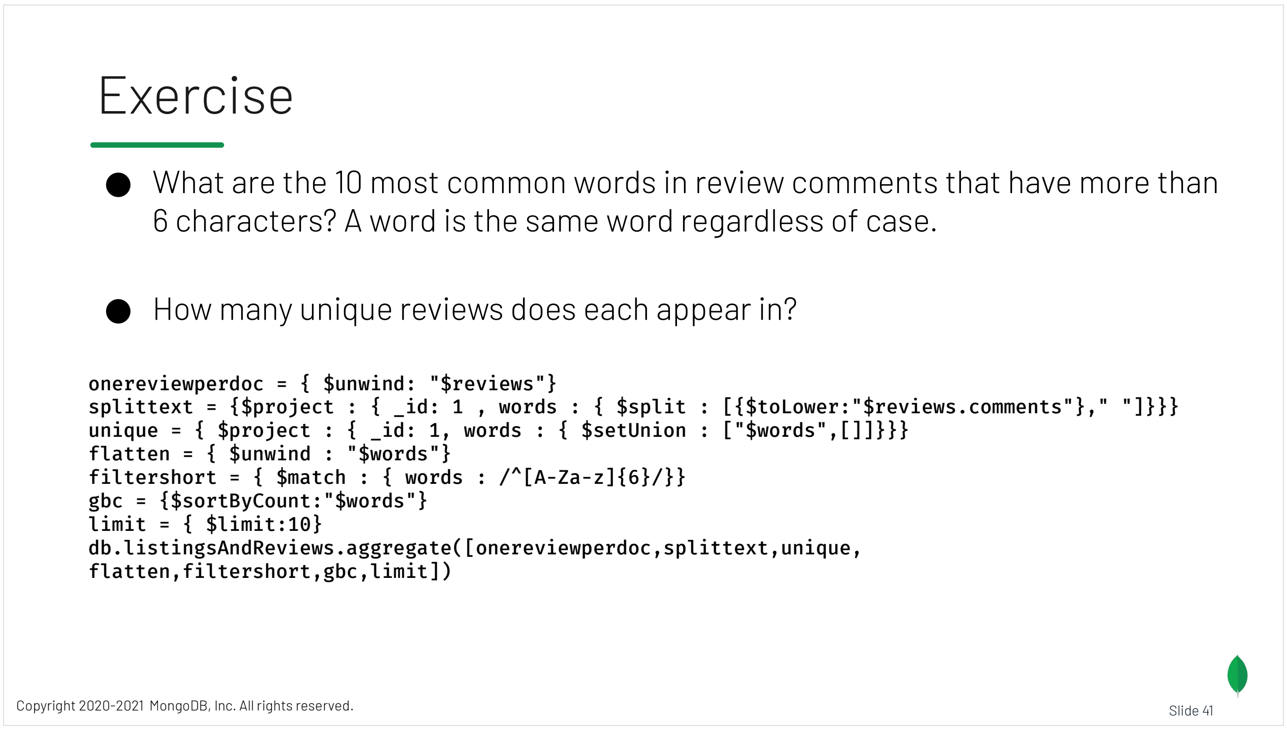
What amenities are offered in the smallest number of countries?

```
unwindstage = { $unwind:"$amenities"}
groupcountry = {$group :{_id: "$amenities",countries: {$addToSet:"$address.country"}}}
unwind2 = { $unwind : "$countries" }
groupcount = {$group : { "_id" : "$_id", count : {$sum : 1}}}
sortstage = {$sort:{count:1}}
pipe=[unwindstage,groupcountry,unwind2,groupcount,sortstage]
db.listingsAndReviews.aggregate( pipe ).pretty()
```

The second $unwind and $group is in one record so is better to be the expression

```
{$set: {count : {$size:"$countries"}}}
```

Exercise Answer



Exercise

- What are the 10 most common words in review comments that have more than 6 characters? A word is the same word regardless of case.
- How many unique reviews does each appear in?

```
onereviewperdoc = { $unwind: "$reviews"}
splittext = {$project : { _id: 1 , words : { $split : [{$toLower:"$reviews.comments"}," "]}}}
unique = { $project : { _id: 1, words : { $setUnion : ["$words",[]]}}}
flatten = { $unwind : "$words"}
filtershort = { $match : { words : /^[A-Za-z]{6}/}}
gbc = {$sortByCount:"$words"}
limit = { $limit:10}
db.listingsAndReviews.aggregate([onereviewperdoc,splittext,unique,
flatten,filtershort,gbc,limit])
```

Exercise Answer