

THE MAGIC OF MARYSES OF MONGO DB



Example Application

The screenshot shows a web browser window with a dark purple header bar containing standard icons like back, forward, and search. The main content area has a dark purple background with white text. At the top, it says "Potion Reviews". Below that, there are three entries, each featuring a small icon of a potion bottle and the name of the potion in a large serif font. To the right of each name are its score, taste, and strength values, along with a "More Info" button.

Potion Name	Score	Taste	Strength
Invisibility	70	4	1
Love	84	3	5
Shrinking	94	2	3

THE
MAGICAL MARVELS
OF MONGO DB

Course Outline

1. Conjuring MongoDB
2. Mystical Modifications
3. Materializing Potions
4. Morphing Models
5. Aggregation Apparitions

THE
MAGICAL MARVELS
OF MONGODB

Conjuring MongoDB

Level 1 - Section 1

Introducing MongoDB

What Is MongoDB?

- Open-source **NoSQL** database
- Document-oriented
- Great for unstructured data, especially when you have a lot of it

Began developing MongoDB as part of their stack



2007



2009

Catch-all term for databases that generally aren't relational and don't have a query language like SQL

Open-sourced

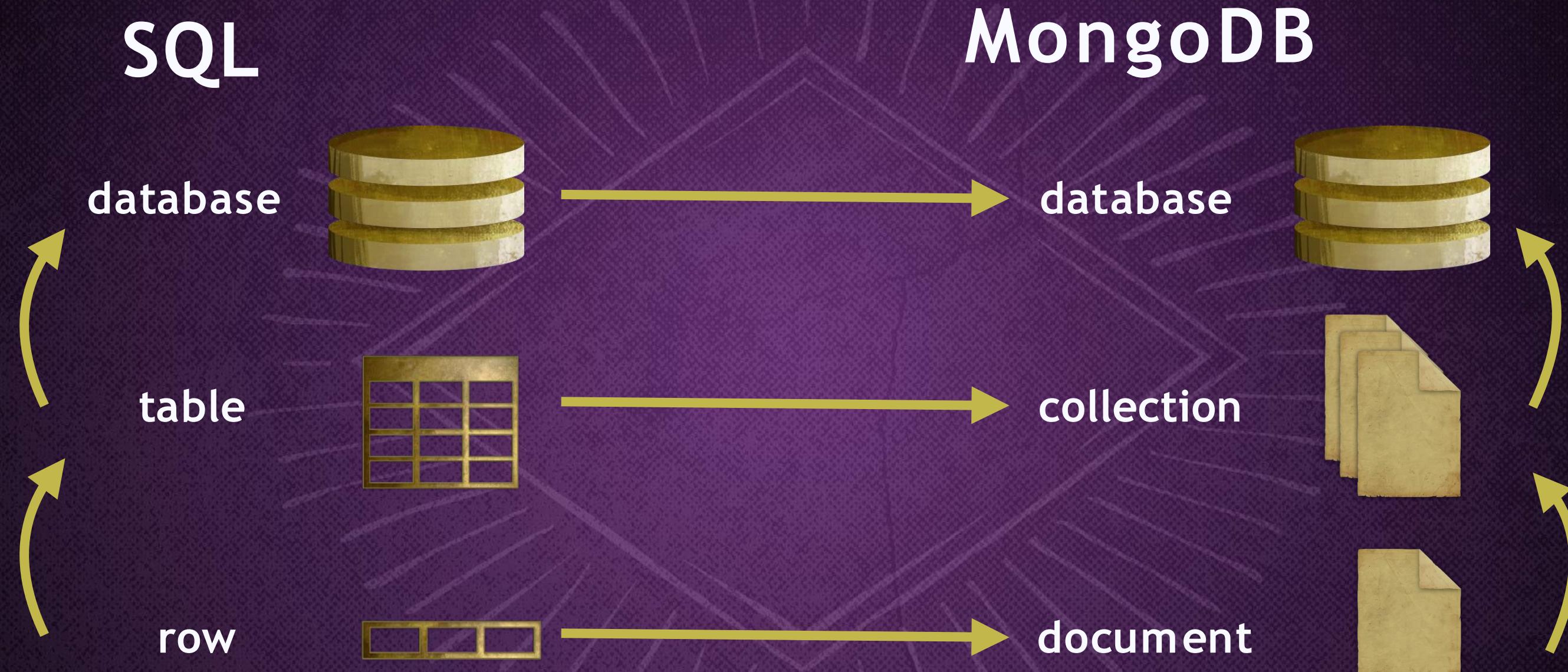
Name comes from the word
“humongous”



2013

Renamed to MongoDB

MongoDB Comparison to SQL



The main difference? SQL is *relational* and MongoDB is *document-oriented*.

THE
MAGICAL MARVELS
of MONGODB

Relational vs. Document-oriented

Relational database management systems save data in rows within tables. MongoDB saves data as documents within collections.

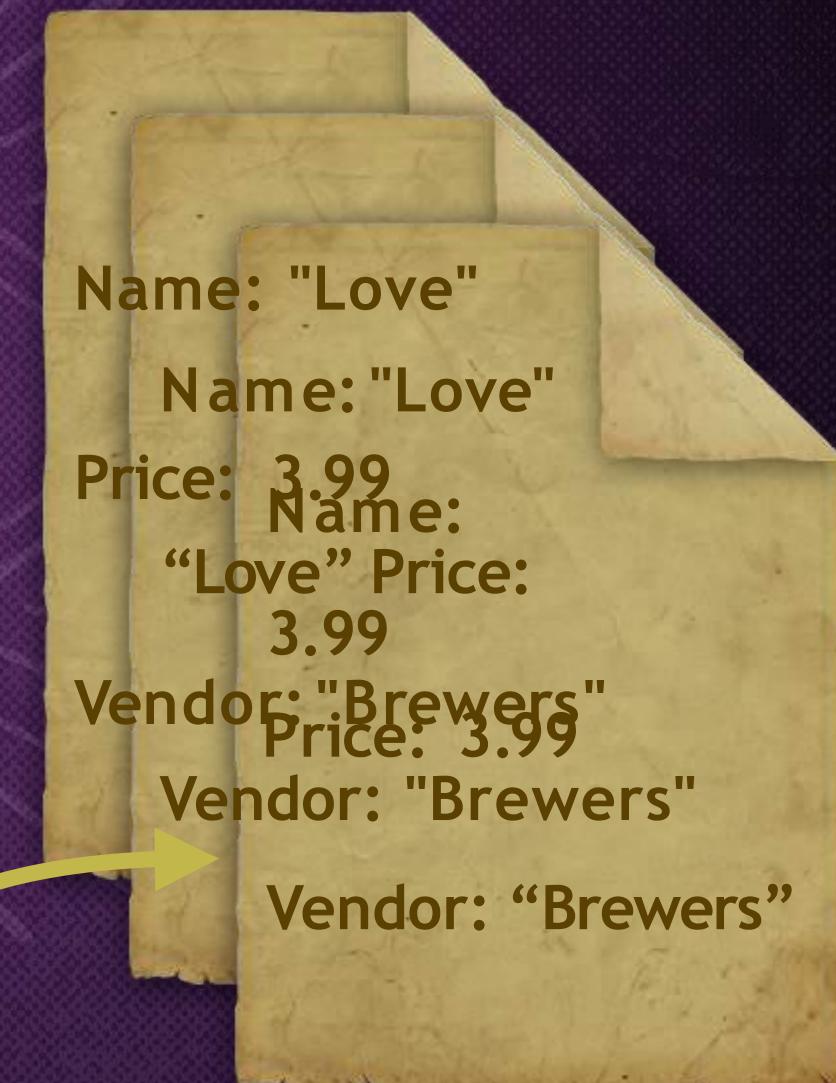
Potions Table

potion_id	name	price	vendor_id
1	“Love”	3.99	2
2	“Invisibility”	15.99	1
3	“Shrinking”	9.99	1

Vendors Table

vendor_id	name
1	“Kettlecooked”
2	“Brewers”

Potions Collection



All data is grouped
within documents

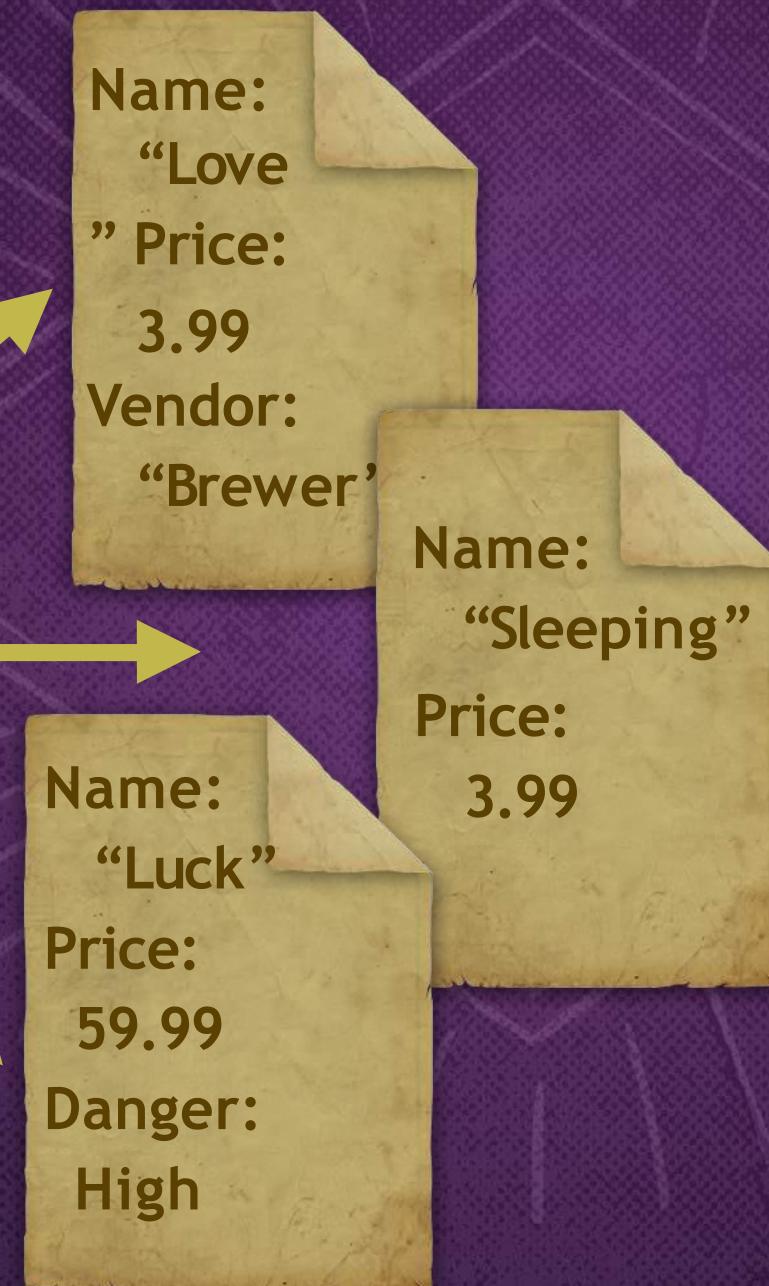
Collections Group Documents

Collections are simply groups of documents. Since documents exist independently, they can have different fields.

Potions Collection



Potions can have
different data!



This is referred to as a
“dynamic schema.”

Starting the Shell

We can access MongoDB through the terminal application. If you want try this out locally, follow the link below for MongoDB installation instructions.



Mongo
commands go
after the >

How Do We Interact With MongoDB?

All instances of MongoDB come with a command line program we can use to interact with our database using JavaScript.

Regular JavaScript
variable assignment

Access the variable
to see the
contents

Get a response back

SHELL

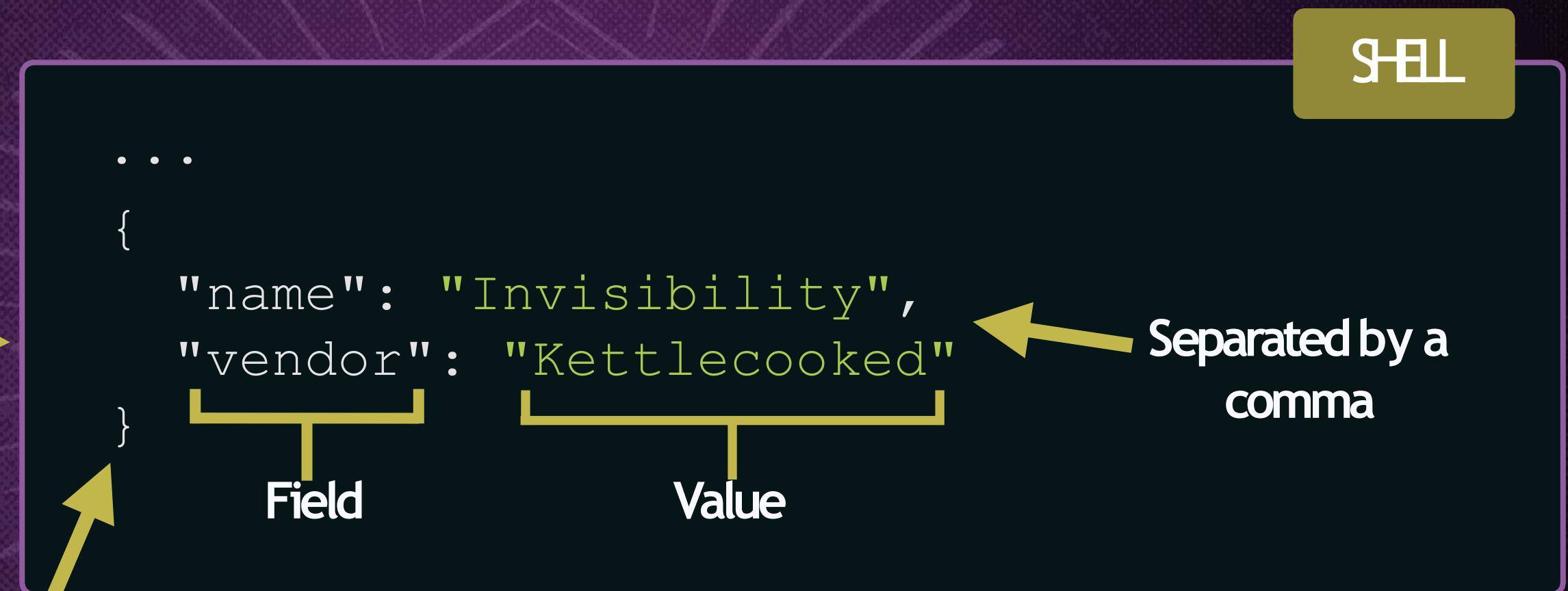
```
> var potion = {  
    "name": "Invisibility",  
    "vendor": "Kettlecooked"  
}
```

```
> potion  
{  
    "name": "Invisibility",  
    "vendor": "Kettlecooked"  
}
```

This is all just normal JavaScript!

Documents Are Just JSON-like Objects

Here's what a simple document looks like.



Using the Shell

MongoDB comes with helper methods to make it easy to interact with the database.

Switches to use the database and creates it if doesn't exist when we write to it

SHELL

```
> use reviews  
switched to db reviews  
>db  
reviews
```

Returns the current database name

Show list of commands

SHELL

```
>help  
db.help()  
...  
show dbs  
>show dbs  
test  
reviews
```

Show list of databases

Name and size of databases

THE MAGICAL MARVELS OF MONGODB

Documents Need to Be Stored in Collections

Documents are always stored in collections within a database.

Potion Document

```
{  
  "name": "Invisibility",  
  "vendor": "Kettlecooked"  
}
```

Document must be
placed in a
collection



Potions Collection



Inserting a Document Into a Collection

We can use the *insert()* collection method to save a potion document to the potions collection.

This collection doesn't exist yet, so it will automatically be created

```
> db.potions.insert(  
{  
  "name": "Invisibility",  
  "vendor": "Kettlecooked"  
}  
)  
  
WriteResult({ "nInserted": 1 })
```

SHELL

To write to the database, we specify the collection and the operation to perform

Potion document as a parameter of the insert method

What's a WriteResult?

Whenever we write to the database, we'll always be returned a WriteResult object that tells us if the operation was successful or not.

SHELL

```
> db.potions.insert(  
  {  
    "name": "Invisibility",  
    "vendor": "Kettlecooked"  
  }  
)  
WriteResult({ "nInserted": 1 })
```



1 document
successfully inserted

Finding All Potions

We can use the `find()` collection method to retrieve the potion from the inventory collection.

All collection methods must end with parentheses

SHELL

```
> db.potions.find()  
{  
  "_id": ObjectId("559f07d741894edebdd8aa6d") ,  
  "name": "Invisibility" ,  
  "vendor": "Kettlecooked"  
}
```

Unique id that gets automatically generated

Using Find to Return All Documents in a Collection

SHELL

```
> db.potions.insert(...)  
WriteResult({ "nInserted": 1 })  
  
> db.potions.insert(...)  
WriteResult({ "nInserted": 1 })  
  
> db.potions.find()  
{ "name": "Invisibility" ... }  
{ "name": "Love" ... }  
{ "name": "Shrinking" ... }
```

Let's add 2 more
potions

Now find returns a
total of 3 potions

Conjuring MongoDB

Level 1 - Section 2

Queries and Data Types

ObjectIds Make Documents Unique

Every document is required to have a unique `_id` field. If we don't specify one when inserting a document, MongoDB will generate one using the ObjectId data type.

SHELL

```
> db.potions.find()  
{  
  "_id": ObjectId("559f07d741894edebdd8aa6d") ,  
  "name": "Invisibility",  
  "vendor": "Kettlecooked"  
}
```



It's common to let MongoDB handle `_id` generation.

THE
MAGICAL MARVELS
OF MONGODB

Finding a Specific Potion With a Query

We can perform a query of equality by specifying a field to query and the value we'd like.

Queries are field/value pairs

SHELL

```
> db.potions.find( { "name": "Invisibility" } )  
{  
  "_id": ObjectId("559f07d741894edebdd8aa6d") ,  
  "name": "Invisibility" ,  
  "vendor": "Kettlecooked"  
}
```

Queries will return all the fields of matching documents

Queries That Return Multiple Values

More than 1 document matches the query

SHELL

```
> db.potions.find( { "vendor": "Kettlecooked" } )  
{  
    "_id": ObjectId("55d232a5819aa726..."),  
    "name": "Invisibility",  
    "vendor": "Kettlecooked"  
}  
{  
    "_id": ObjectId("55c3b9501aad0cb0..."),  
    "name": "Shrinking",  
    "vendor": "Kettlecooked"  
}
```

Two separate documents
are returned



Queries are case sensitive.

What Else Can We Store?

Documents are persisted in a format called BSON.

BSON is like JSON, so you can store:

Strings

```
"Invisibility"
```

Numbers

```
1400
```

```
3.14
```

Booleans

```
true
```

```
false
```

Arrays

```
[ "newt toes", "pickles" ]
```

Objects

```
{ "type" : "potion" }
```

Null

```
null
```

BSON comes with some extras.

ObjectID

```
ObjectId(...)
```

Date

```
ISODate(...)
```

Building Out Our Potions

Now that we have a better grasp on documents, let's build out our potion document with all the necessary details.



Adding Price and Score

We can store both integers and floats in a document.

```
{  
  "name": "Invisibility",  
  "vendor": "Kettlecooked",  
  "price": 10.99,  
  "score": 59  
}
```

MongoDB will preserve the precision of both floats and integers

Adding a tryDate

Dates can be added using the JavaScript Date object and get saved in the database as an ISODate object.

```
{  
  "name": "Invisibility",  
  "vendor": "Kettlecooked",  
  "price": 10.99,  
  "score": 59,  
  "tryDate": new Date(2012, 8, 13)  
}
```

Reads as September 13, 2012, since JavaScript months begin at 0

Dates get converted to an ISO format when saved to the database

```
"tryDate": ISODate("2012-09-13T04:00:00Z")
```

Adding a List of Ingredients

Arrays are a great option for storing lists of data.

```
{  
  "name": "Invisibility",  
  "vendor": "Kettlecooked",  
  "price": 10.99,  
  "score": 59,  
  "tryDate": new Date(2012, 8, 13),  
  "ingredients": ["newt toes", 42, "laughter"]  
}
```

We can store any data type
within an array

Adding a Potion's Ratings

Each potion has 2 different ratings, which are scores based on a scale of 1 to 5.

```
{  
  "name": "Invisibility",  
  "vendor": "Kettlecooked",  
  "price": 10.99,  
  "score": 59,  
  ...  
}
```

Each rating will have 2 fields

```
{  
  "strength": 2,  
  "flavor": 5  
}
```

MongoDB supports embedded documents so we can simply add this to our potion document

Embedded Documents

We embed documents simply by adding the document as a value for a given field.

```
{  
  "name": "Invisibility",  
  "vendor": "Kettlecooked",  
  "price": 10.99,  
  "score": 59,  
  "tryDate": new Date(2012, 8, 13),  
  "ingredients": ["newt toes", 42, "laughter"],  
  "ratings": {"strength": 2, "flavor": 5}  
}
```

An embedded document doesn't require an id
since it's a child of the main document

Inserting Our New Potion

We've cleared out the inventory collection — now let's add our newly constructed potion!

SHELL

```
> db.potions.insert(  
{  
  "name": "Invisibility",  
  "vendor": "Kettlecooked",  
  "price": 10.99,  
  "score": 59,  
  "tryDate": new Date(2012, 8, 13),  
  "ingredients": ["newt toes", 42, "laughter"],  
  "ratings": {"strength": 2, "flavor": 5}  
}  
)  
WriteResult({ "nInserted": 1 })
```

Document successfully inserted!

Finding Potions by Ingredients

Array values are treated individually, which means we can query them by specifying the field of the array and the value we'd like to find.

Same format as basic query for equality

SHELL

```
> db.potions.find({ "ingredients": "laughter" })  
{  
  "_id": "ObjectId(...)",  
  "name": "Invisibility",  
  ...  
  "ingredients": ["newt toes", "secret", "laughter"]  
}
```

Potion contains the right
ingredient

Finding a Potion Based on the Flavor

We can search for potions by their ratings using dot notation to specify the embedded field we'd like to search.

```
{  
  "_id": "ObjectId(...)",  
  "name": "Invisibility",  
  ...  
  "ratings": { "strength": 2, "flavor": 5 }  
}
```



“ratings.strength”

“ratings.flavor”

We can easily query
embedded documents

```
db.potions.find( {"ratings.flavor": 5} )
```

What About Insert Validations?

If we were to insert a new potion but accidentally set the price value to a string, the potion would still get saved despite all other potions having integer values.

SHELL

```
> db.potions.insert({  
  "name": "Invisibility",  
  "vendor": "Kettlecooked",  
  "price": "Ten dollars", ←  
  "score": 59  
})  
  
WriteResult({ "nInserted": 1 })
```

Data we might consider to be invalid but MongoDB will think is fine

The document still got saved to the database!



Validations Supported by MongoDB

MongoDB will only enforce a few rules, which means we'll need to make sure data is valid client-side before saving it.

```
{  
  "_id": ObjectId("55c3b9561..."),  
  "name": "Invisibility",  
  "vendor": "Kettlecooked",  
  "price": 10.99  
}  
  
{  
  "_id": ObjectId("55d232a51..."),  
  "name": "Shrinking",  
  "vendor": "Kettlecooked",  
  "price": 9.99  
}
```



No other document shares same `_id`



No syntax errors



Document is less than 16mb

Validations Supported by MongoDB

MongoDB will only enforce a few rules, which means we'll need to make sure data is valid client-side before saving it.

```
{  
  "_id": ObjectId("55c3b9561..."),  
  "name": "Invisibility",  
  "vendor": "Kettlecooked",  
  "price": 10.99  
, ← Missing end bracket  
{  
  "_id": ObjectId("55d232a51..."),  
  "name": "Shrinking",  
  "vendor": "Kettlecooked",  
  "price": 9.99  
}
```



No other document shares same `_id`



No syntax errors



Document is less than 16mb

Validations Supported by MongoDB

MongoDB will only enforce a few rules, which means we'll need to make sure data is valid client-side before saving it.

```
{  
  "_id": 1,  
  "name": "Invisibility",  
  "vendor": "Kettlecooked",  
  "price": 10.99  
},  
{  
  "_id": 1, ← Duplicate _id  
  "name": "Shrinking",  
  "vendor": "Kettlecooked",  
  "price": 9.99  
}
```



No other document shares same _id



No syntax errors



Document is less than 16mb

Mystical Modifications

Level 2 - Section 1

Removing and Modifying Documents

Potion Catastrophe

Uh-oh — we sneezed while performing a spell and ruined some potions in our database!

PotionReviews

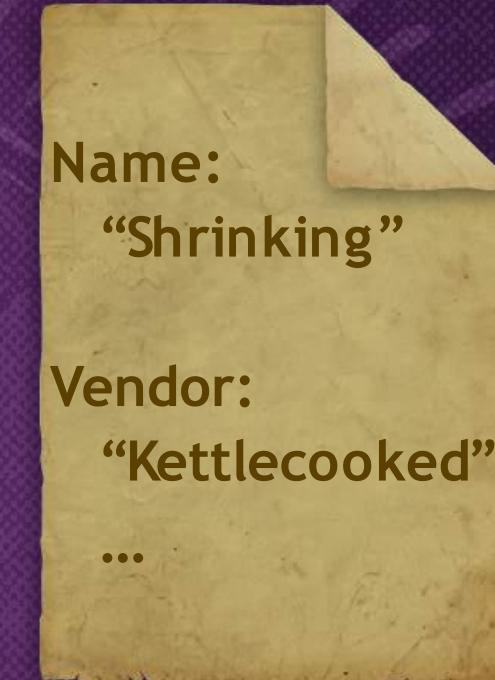
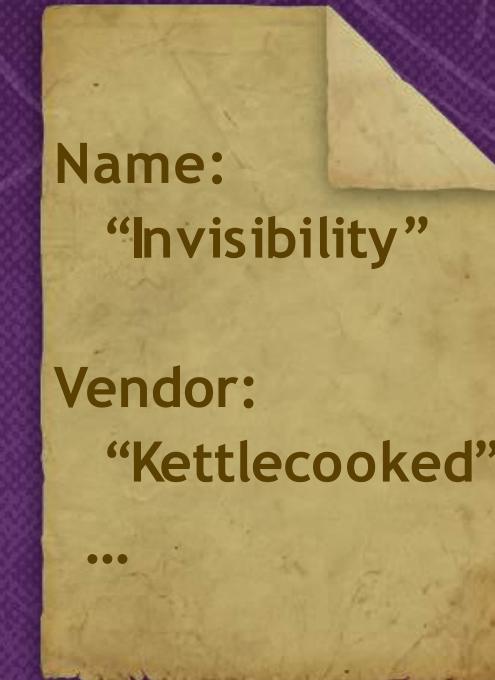
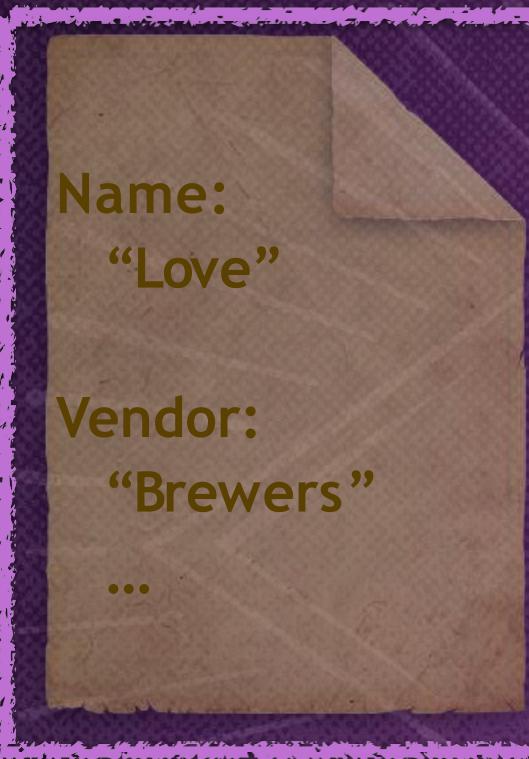
Potion	Taste	Score	Effect	Strength	Action
Invisibility	4	10		1	More Info
Love	3	84	Shrinking	5	More Info
Love	2	94	Strength	3	More Info

Need a way to remove the affected potions from our collection

Delete a Single Document

The `remove()` collection method will delete documents that match the provided query.

Ruined
Potions



1 document
successfully removed

```
> db.potions.remove(  
  { "name": "Love" }  
)  
WriteResult({ "nRemoved": 1 })
```

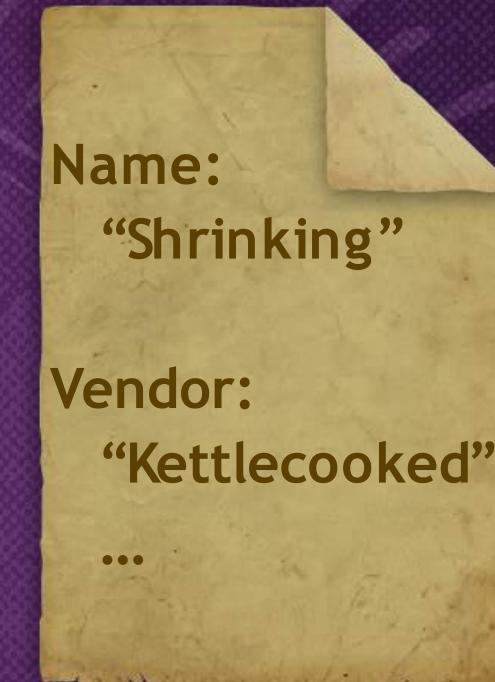
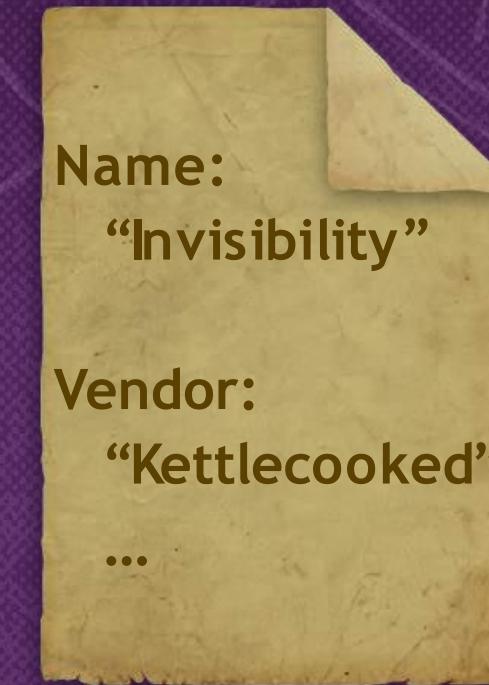
SHELL

Query matches
single document

Delete a Single Document

The `remove()` collection method will delete documents that match the provided query.

Ruined
Potions



1 document
successfully removed

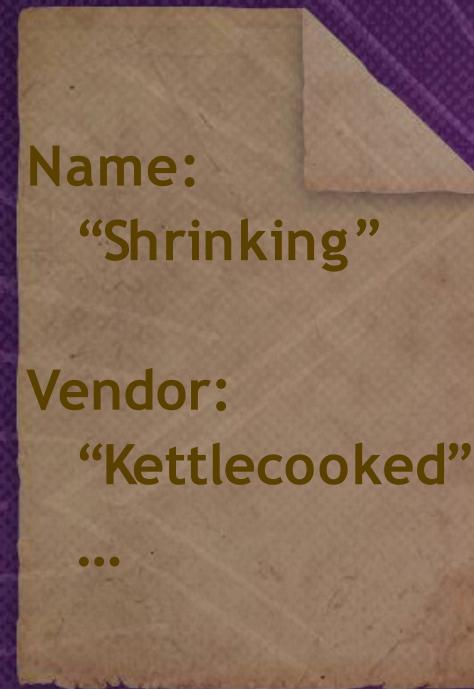
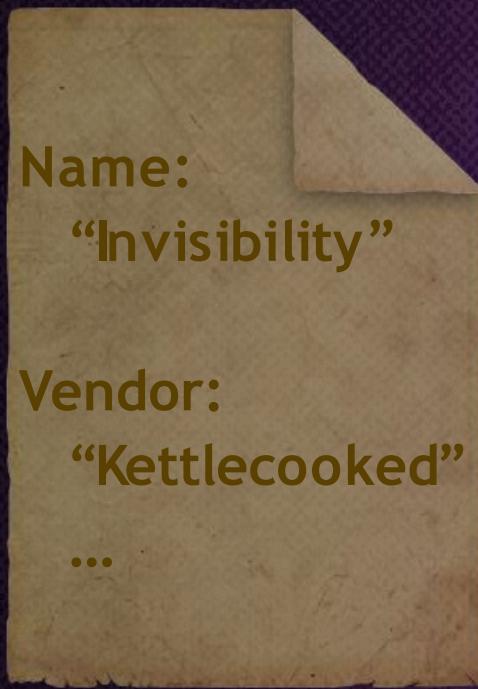
```
> db.potions.remove(  
  { "name": "Love" }  
)  
WriteResult({ "nRemoved": 1 })
```

SHELL

Query matches
single document

Delete Multiple Documents

If our query matches multiple documents, then `remove()` will delete all of them.



*Passing {} as the query would
delete all documents in the
collection.*

SHELL

```
> db.potions.remove(  
  { "vendor": "Kettlecooked" }  
)
```

```
WriteResult({ "nRemoved": 2 })
```

*Query matches
both
documents*

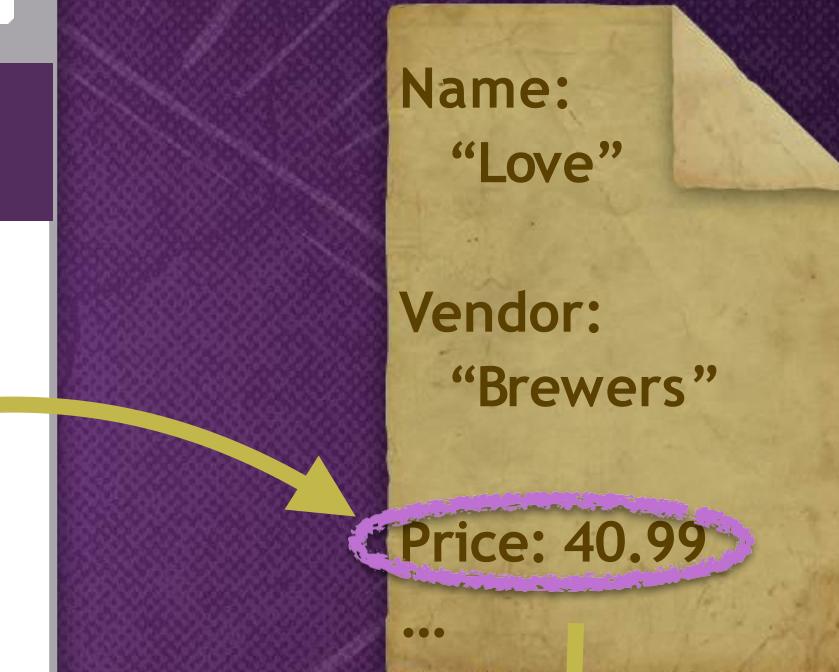
Removed 2 documents

Update the Price of a Single Potion

We made a typo while inserting our love potion, so let's update the price.

The screenshot shows a web browser window with a title bar and a header titled "Potion Reviews". Below the header are three potion reviews:

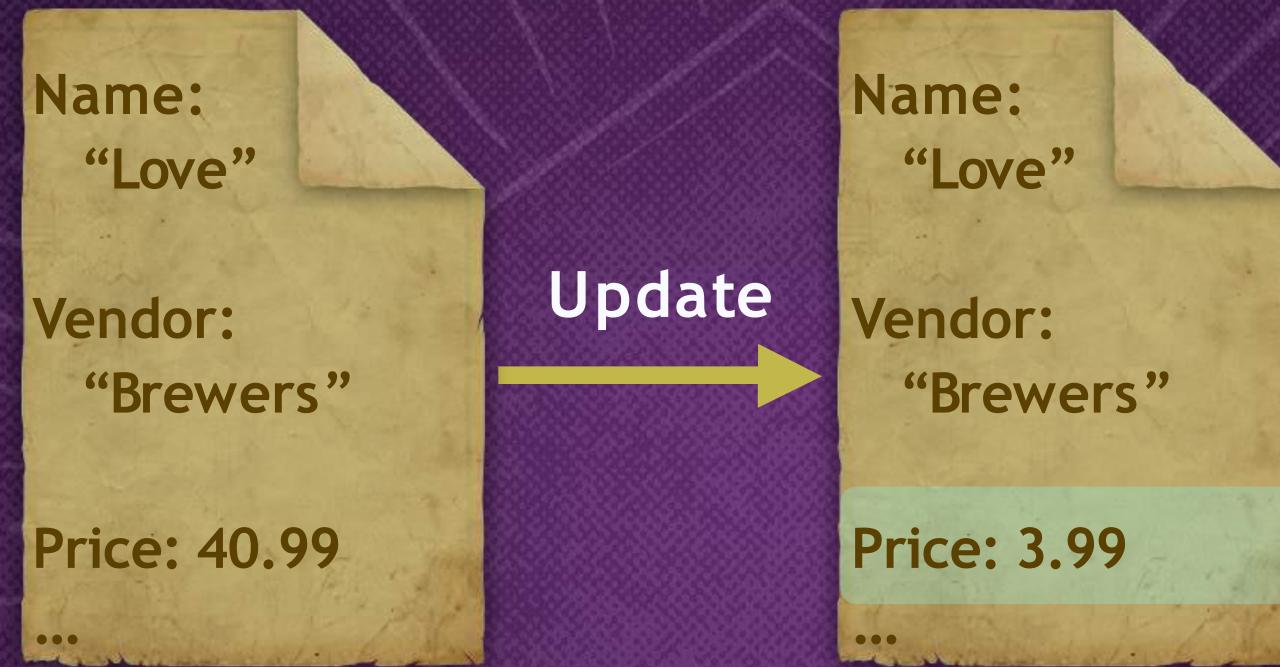
- Invisibility**: Score: 70, Taste: 4, Strength: 1. A "More Info" button is to the right.
- Love**: Score: 84, Taste: 3, Strength: 5. This review is highlighted with a purple border and a "More Info" button.
- Shrinking**: Score: 94, Taste: 2, Strength: 3. A "More Info" button is to the right.



*Needs to be updated
with the correct
price*

Updating a Document

We can use the *update()* collection method to modify existing documents.



```
> db.potions.update(  
  { "name": "Love" },  
  { "$set": { "price": 3.99 } })
```

SHELL

Query parameter

Update parameter

Update operators

always begin with a \$

THE
MAGICAL MARVELS
of MONGODB

Understanding the Update WriteResult

The WriteResult gives a summary of what the *update()* method did.

SHELL

```
> db.potions.update(  
  { "name": "Love" },  
  { "$set": { "price": 3.99 } }  
)  
WriteResult ( {  
  "nMatched": 1,  
  "nUpserted": 0,  
  "nModified": 1  
} )
```

Number of documents matched

Number of documents that were created

Number of documents modified

Update Without an Operator

If the update parameter consists of only field/value pairs, then everything but the `_id` is replaced in the matching document.



```
> db.potions.update(  
  { "name": "Love" },  
  { "price": 3.99 } )
```

Useful for
importing data

SHELL

THE
MAGICAL MARVELS
OF MONGODB

Updating Multiple Documents

The update method can take a third parameter for options.

Notice

WE ARE NOT
CALLED KC

4 documents
matched and
modified

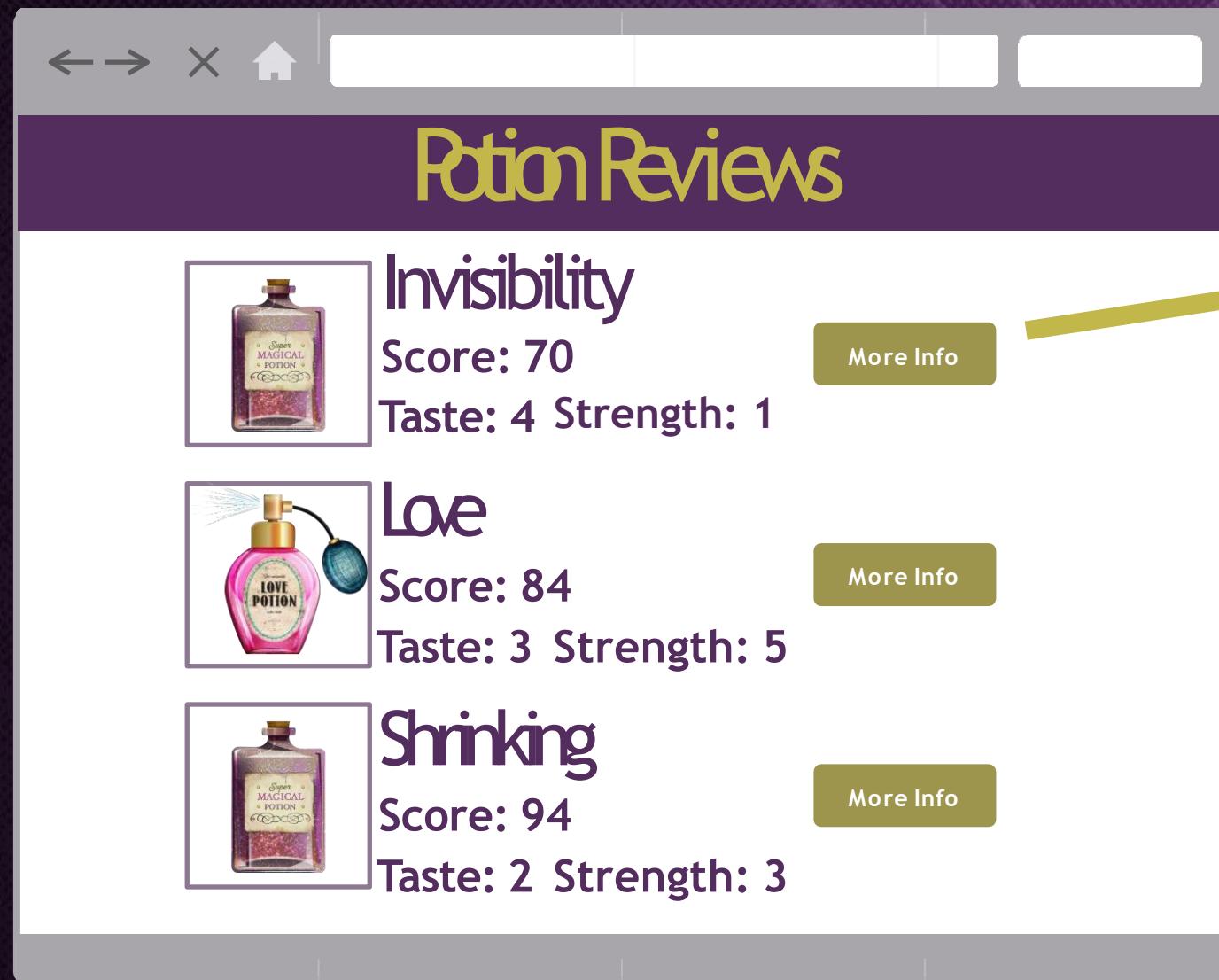
SHELL

```
> db.potions.update(  
  { "vendor": "KC"},  
  { "$set": { "vendor": "Kettlecooked" } },  
  { "multi": true }) ← When multi is true, the update  
                      modifies all matching documents
```

WriteResult ({
 "nMatched": 4,
 "nUpserted": 0,
 "nModified": 4
 })

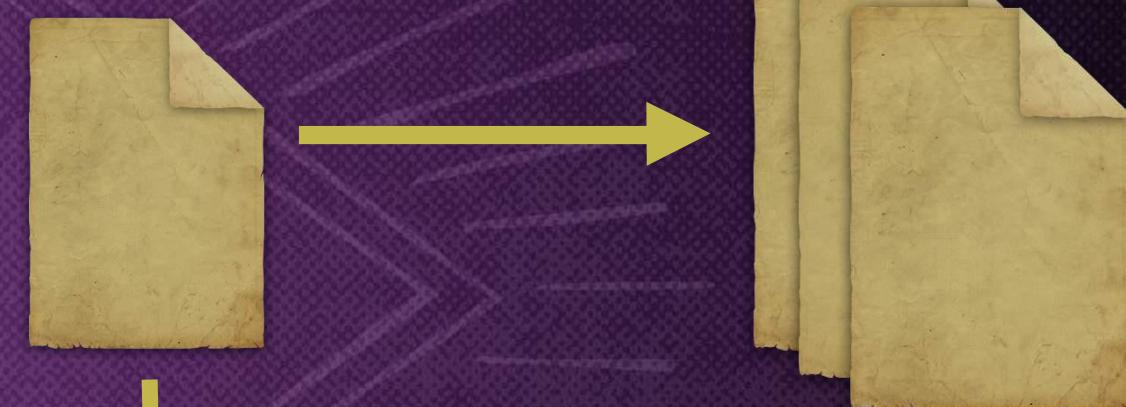
Recording Potion Views

Time to start analyzing which potions are viewed the most. To do this, we need to record each potion page view.



*Create or update
existing log document*

Logs Collection

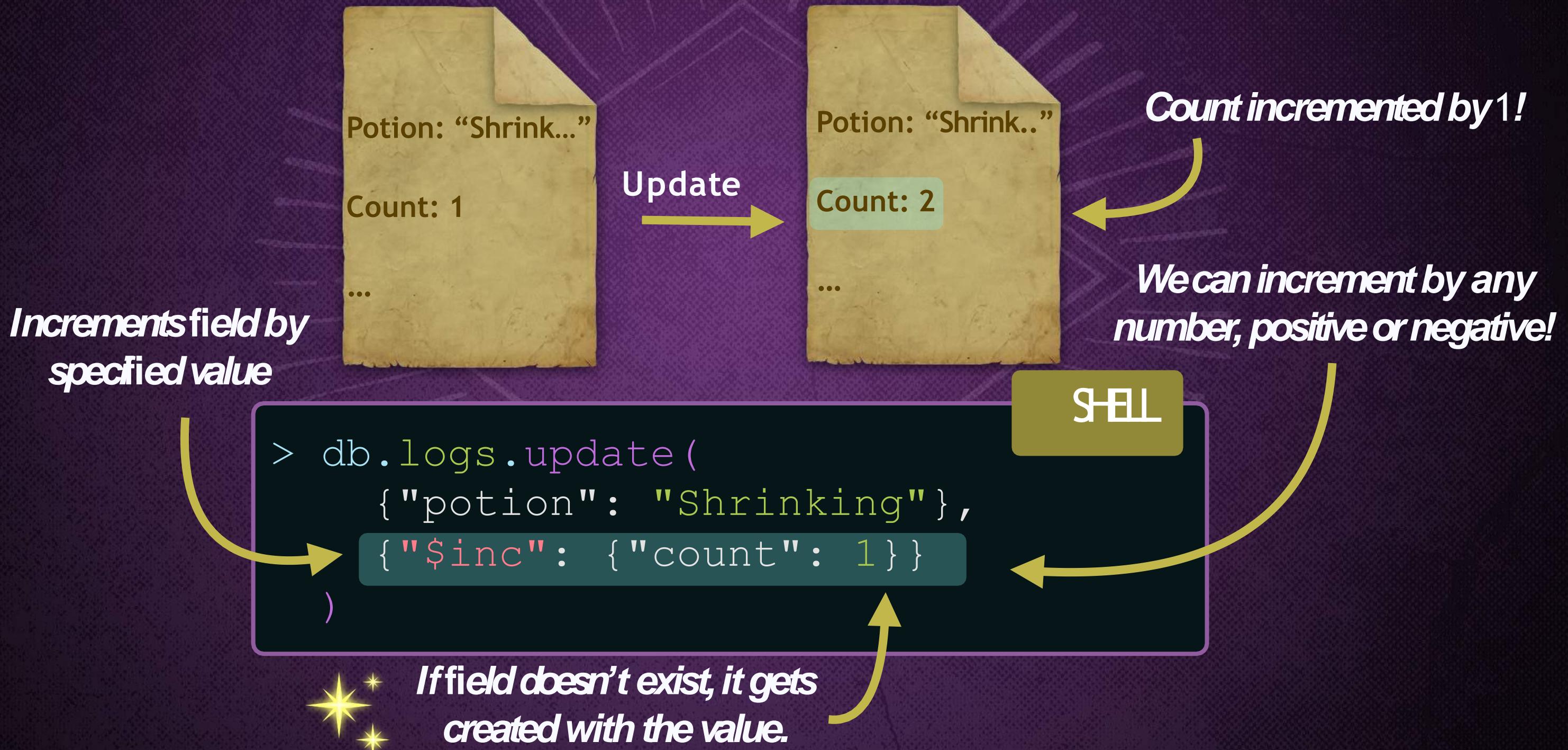


*We'll update count
with each click*

```
{  
  "_id": ObjectId(...),  
  "potion": "Frog Tonic",  
  "count": 1  
}
```

Update a Document's Count

We can use the `$inc` operator to increment the count of an existing log document.



Update a Non-existing Potion

If we run the update on a potion that doesn't exist, then nothing will happen.

```
> db.logs.update(  
  { "potion": "Love" } , ◀  
  { "$inc": { "count": 1 } } ,  
)  
WriteResult({  
  "nMatched": 0 ,  
  "nUpserted": 0 ,  
  "nModified": 0  
})
```

SHELL

Potion log doesn't exist yet



No potions matched or modified

Find or Create With Upsert

The upsert option either updates an existing document or creates a new one.

If the field doesn't exist, it gets created with the value

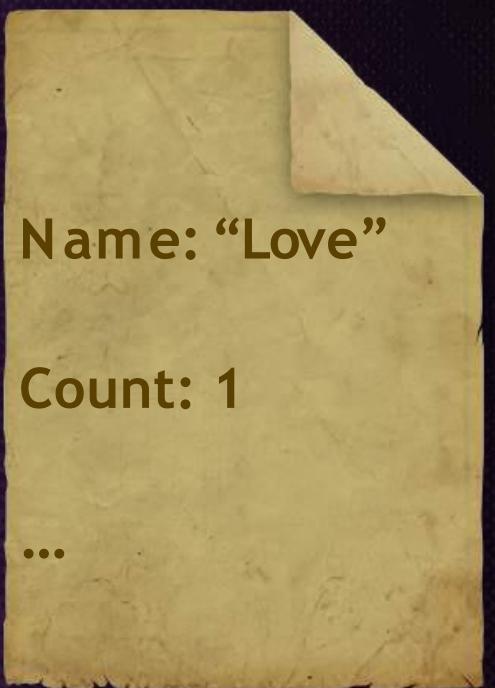
```
> db.logs.update (   
    { "potion": "Love" },   
    { "$inc": { "count": 1 } },   
    { "upsert": true }   
 )
```

```
WriteResult ( {
```

```
    "nMatched": 0,   
    "nUpserted": 1,   
    "nModified": 0  
} )
```

SHELL

Results in new document



Creates a document using the values from the query and update parameter

1 document created

Updating Once More

If we run the same update again, the update will act normally and *upsert* won't create another document.

```
> db.logs.update (   
    { "potion": "Love" },   
    { "$inc": { "count": 1 } },   
    { "upsert": true }   
 )
```

```
WriteResult ( {
```

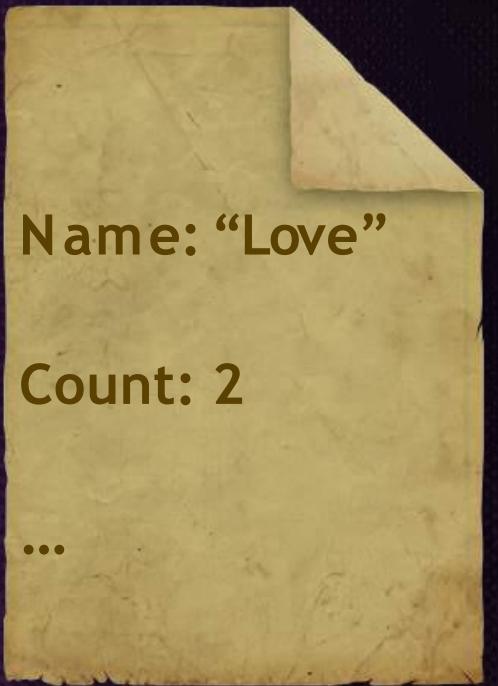
```
    "nMatched": 1,   
    "nUpserted": 0,   
    "nModified": 1
```

```
} )
```

SHELL

Result

Count of 2



Document found and modified but nothing created

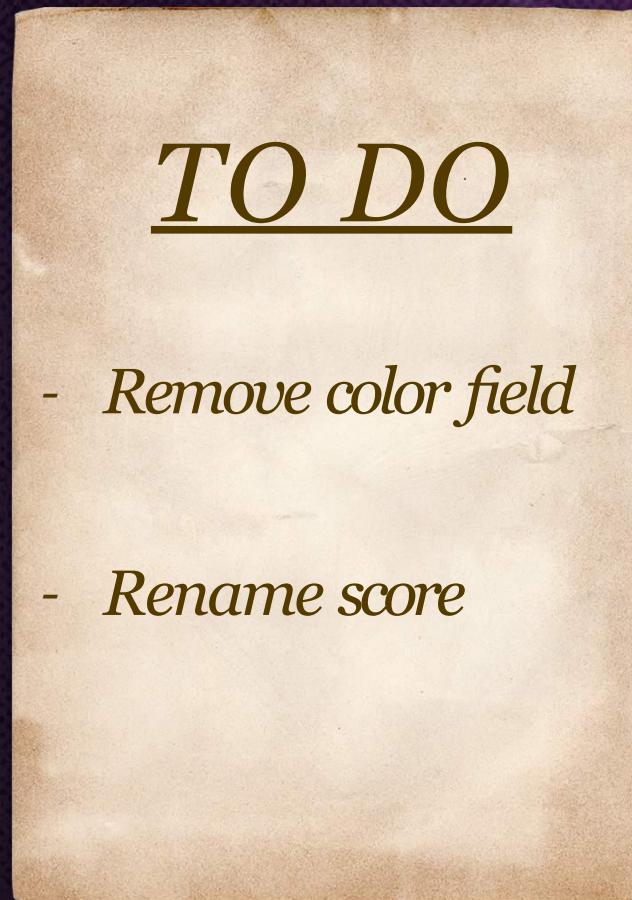
Mystical Modifications

Level 2 - Section 2

Advanced Modification

Improving Potions

We rushed a bit during development and need to fix up our potions. Luckily, we keep a to-do list for what to work on.

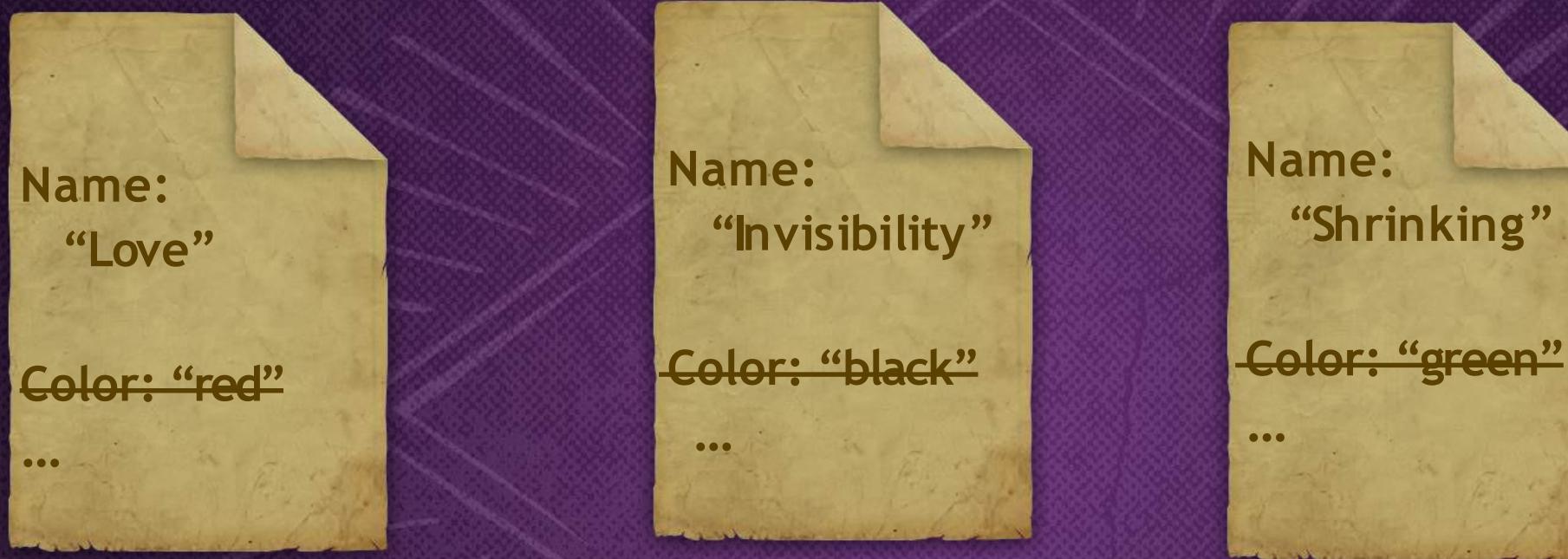


We can accomplish these tasks with update operators!



Removing Fields From Documents

We initially thought we'd need a potion's color, but we never use it. The `$unset` operator can be used to remove specified fields.



**Colorfield
removed
from
documents**

**Query for
all potions**

**Update all
potions**

```
> db.potions.update(  
  {},  
  { "$unset": { "color": "" } },  
  { "multi": true })
```

SHELL

**The value we pass
doesn't impact the
operation**

Updating a Field Name With \$rename

We can use `$rename` to change field names.

```
{  
  "_id": ObjectId(...),  
  "name": "Love",  
  "score": 84,  
  ...  
}
```



```
{  
  "_id": ObjectId(...),  
  "name": "Love",  
  "grade": 84,  
  ...  
}
```

*Renamed
to grade!*

```
> db.potions.update(  
  {},  
  { "$rename": { "score": "grade" } },  
  { "multi": true })
```

*Rename
s
specified
field*

SHELL

*New
field
name*

*Field to
rename*

*THE
MAGICAL MARVELS
of MONGODB*

Potion Ingredient Regulation

The Magical Council has passed a new regulation requiring us to list all ingredients in a potion. No more secret ingredients!

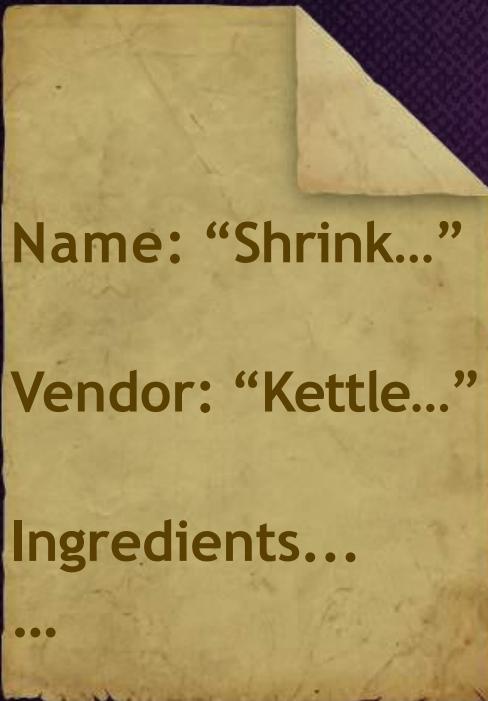
Notice

All *secret ingredients*
must be listed!

```
{  
  "_id": ObjectId(...),  
  "name": "Shrinking",  
  ...  
  "ingredients": ["hippo", "secret", "mouse feet"]  
}
```

Need to update with
actual ingredient:

The Dilemma of Updating an Array



```
"ingredients": ["hippo", "secret", "mouse feet"]
```

```
> db.potions.update(  
  { "ingredients": "secret" },  
  { "$set": { "ingredients": "42" } })
```

SHELL



Would overwrite the entire array and set it as 42

```
"ingredients": 42
```

Updating Array Values by Location

Since array values are treated individually, we can update a single value by specifying its location in the array using **dot notation**.

```
{  
  "_id": ObjectId(...),  
  "name": "Shrinking",  
  "vendor": "Kettlecooked",  
  "score": 94,  
  ...  
  "ingredients": ["hippo", "secret", "mouse feet"]  
}
```

ingredients.0

ingredients.1

ingredients.2

BSON arrays start with an index of 0

Updating Single Array Value

The **\$set** operator will update the value of a specified field.

The secret ingredient!

SHELL

```
> db.potions.update(  
  { "name": "Shrinking" },  
  { "$set": { "ingredients.1" : 42 } }  
)
```

```
WriteResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
```

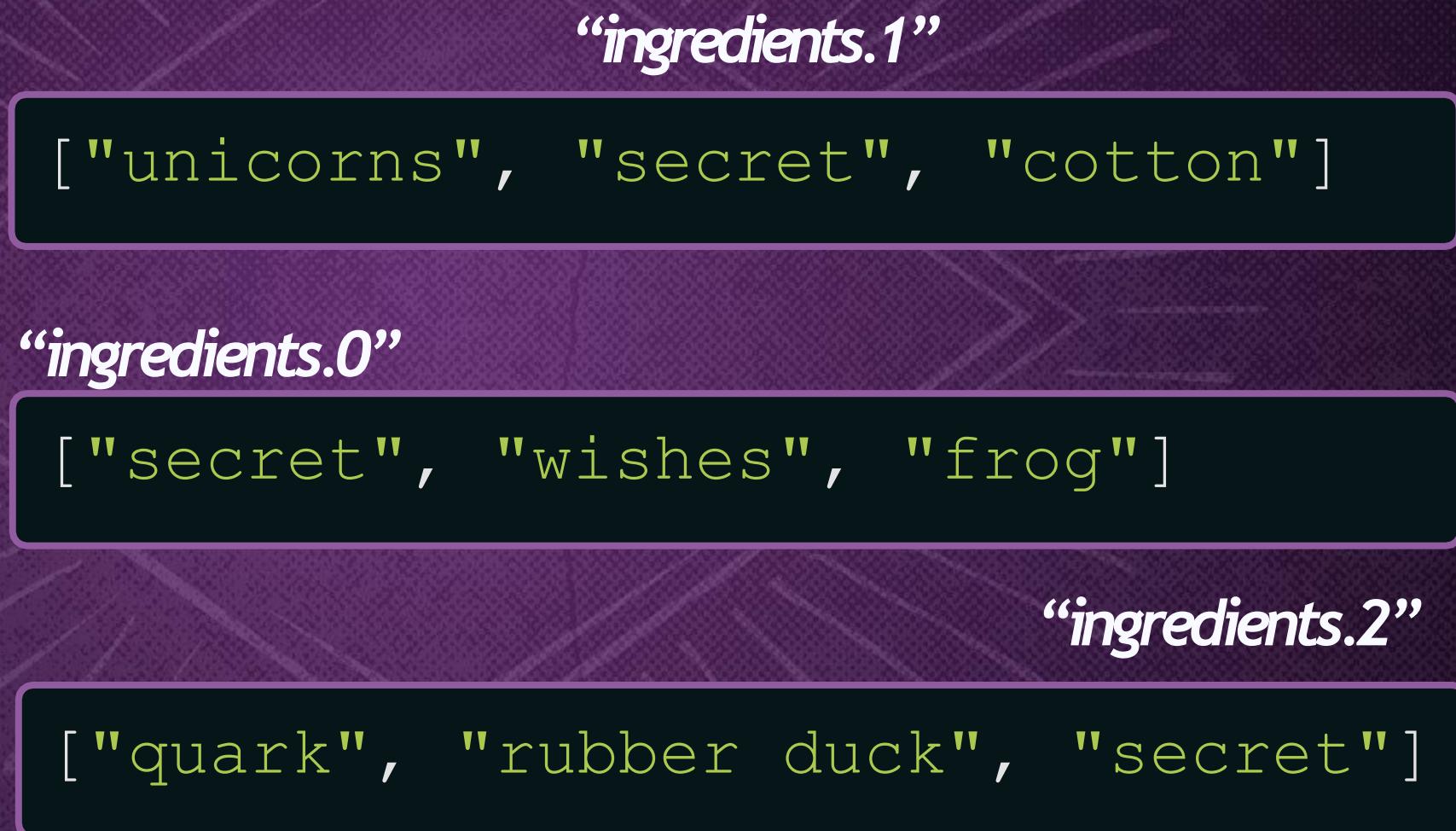
```
{  
  "_id": ObjectId(...),  
  "name": "Shrinking",  
  ...  
  "ingredients": ["hippo", 42, "mouse feet"]  
}
```

*Successful
update*

Updating Multiple Arrays

We need to change “secret” in multiple documents, but the location isn’t always the same for every potion.

Potions Collection



Updating Values Without Knowing Position

The positional operator is a placeholder that will set the proper position for the value specified in the query parameter.

```
> db.potions.update(  
  { "ingredients": "secret"}, ,  
  { "$set": { "ingredients.$" : 42 } },  
  { "multi": true}  
)
```

SHELL

*Query for the value we
want to change*

*Multi is true to make
the change to all
documents*

*The \$ is a placeholder for
the matched value*

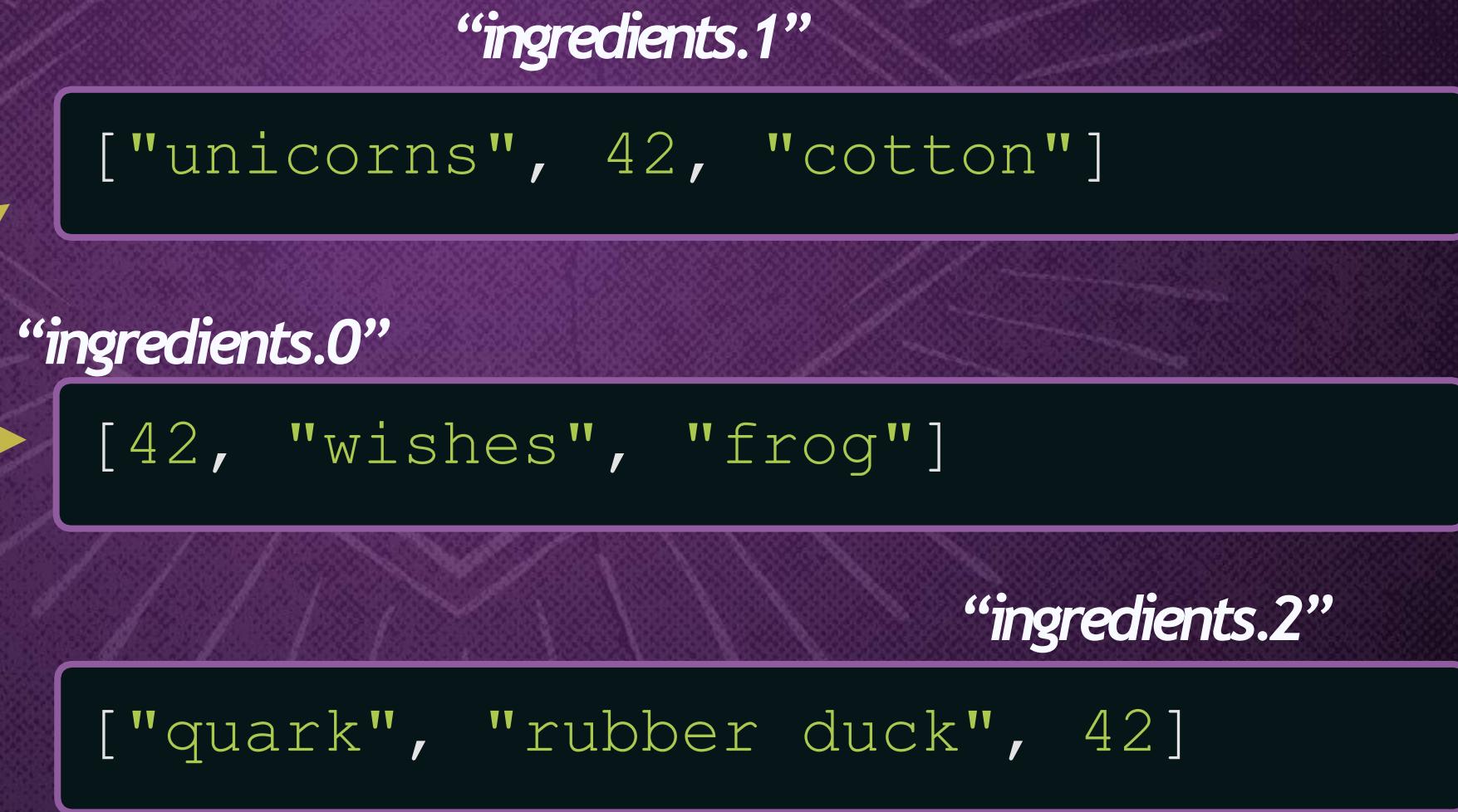


Only updates the first match per document

The Result of Using the Positional Operator to Update

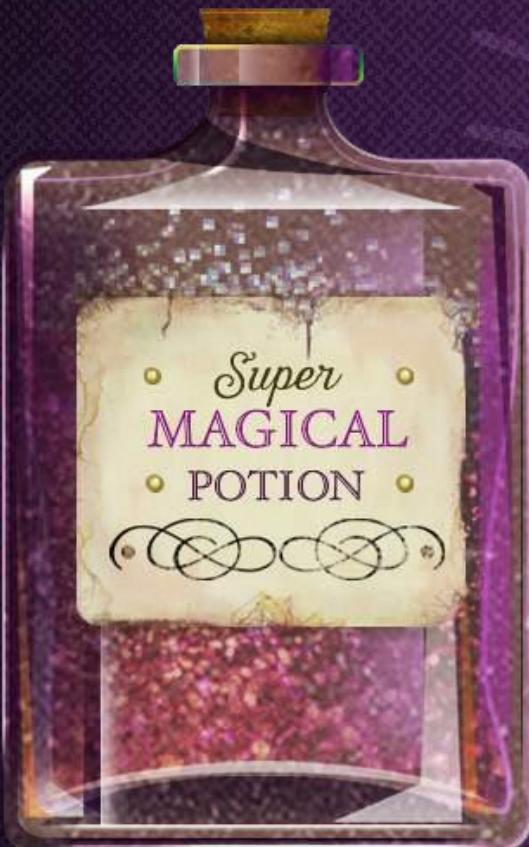
```
...  
  { "ingredients": "secret" },  
  { "$set": { "ingredients.$" : 42 } },  
...
```

Potions Collection



Shrunken Conundrum

Uh-oh – the shrinking potion hasn't worn off, and we keep on shrinking! We better update that strength rating.

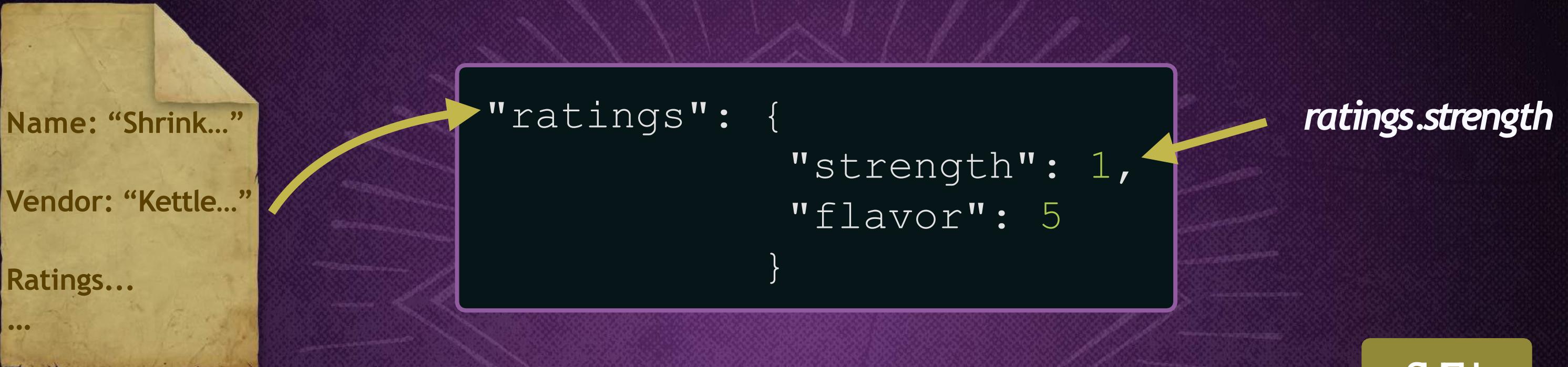


```
{  
  "_id": ObjectId(...),  
  "name": "Shrinking",  
  ...  
  "ratings": {  
    "strength": 1,  
    "flavor": 5  
  }  
}
```

*Update
strength
to 5*

Updating an Embedded Value

We can update using the dot notation to specify the field we want to update.



```
WriteResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
```

Useful Update Operators

MongoDB provides a variety of ways to modify the values of fields.

\$max

Updates if new value is greater than current or inserts if empty

\$min

Updates if new value is less than current or inserts if empty

\$mul

Multiplies current field value by specified value. If empty, it inserts 0.

Reference > Operators > Update Operators > Field Update Operators > \$max

\$max

Definition

\$max

The `$max` operator updates the value of the field to a specified value *if* the specified value is **greater than** the current value of the field. The `$max` operator can compare values of different types, using the [BSON comparison order](#).

The `$max` operator expression has the form:

```
{ $max: { <field1>: <value1>, ... } }
```

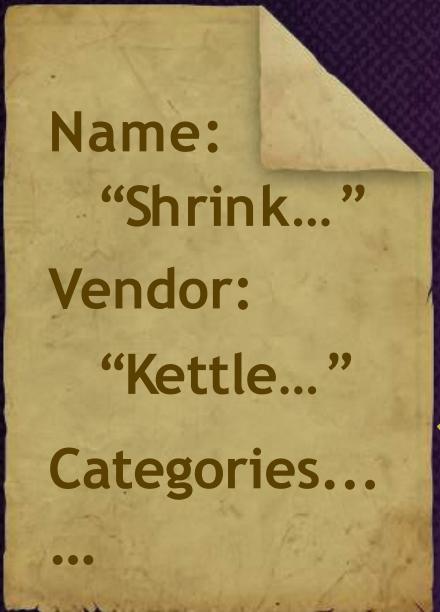
To specify a `<field>` in an embedded document or in an array, use [dot notation](#).

MongoDB's documentation is great

THE
MAGICAL MARVELS
OF MONGODB

Modifying Arrays

We've added categories to the potions but need a way to easily manipulate the values.

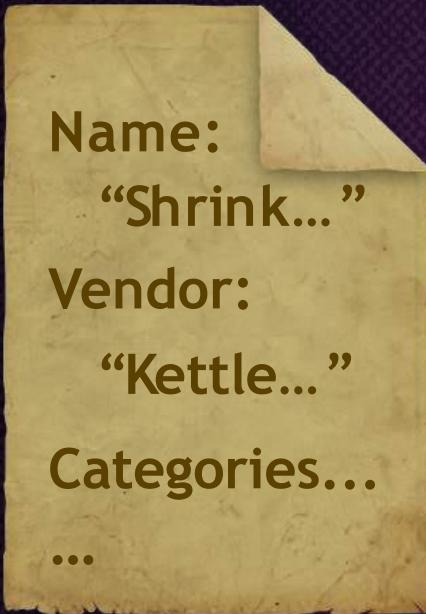


List of categories for the potion

```
"categories": ["tasty", "effective"]
```

Removing the First or Last Value of an Array

The `$pop` operator will remove either the first or last value of an array.



"categories": ["tasty", "effective"]

```
> db.potions.update(  
  { "name": "Shrinking" },  
  { "$pop": { "categories": 1 } })
```



*Doesn't return the value –
only modifies the array*

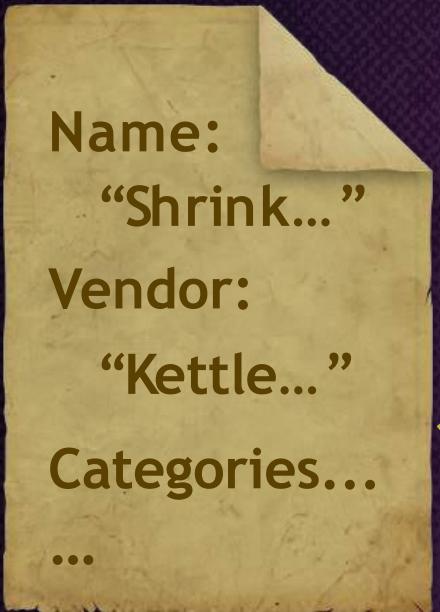
- 1 *Removes the first element*
- 1 *Removes the last element*

Result

"categories": ["tasty"]

Adding Values to the End of an Array

The `$push` operator will add a value to the end of an array.



"categories": ["tasty"]

```
> db.potions.update(  
  { "name": "Shrinking" },  
  { "$push": { "categories": "budget" } })
```

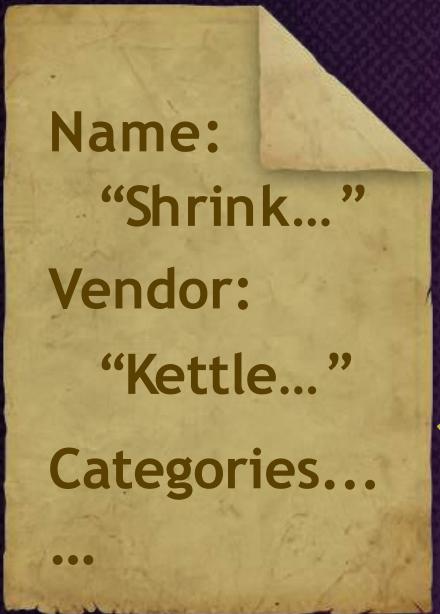
Added to the end

Result

"categories": ["tasty", "budget"]

Adding Unique Values to an Array

The `$addToSet` operator will add a value to the end of an array unless it is already present.



```
"categories": ["tasty", "budget"]
```

```
> db.potions.update(  
  { "name": "Shrinking" },  
  { "$addToSet": { "categories": "budget" } })
```

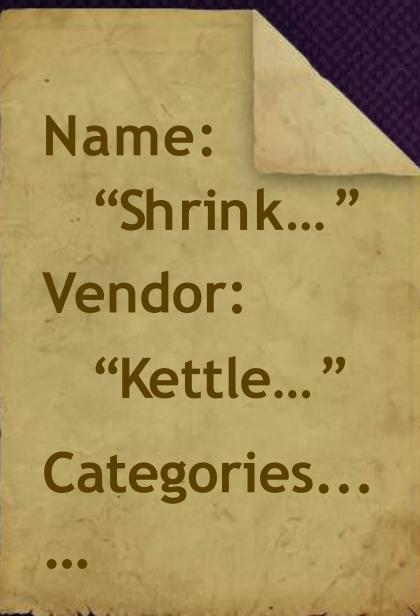
Result

```
"categories": ["tasty", "budget"]
```

*Value already exists, so it
doesn't get added again*

Removing Values From an Array

The `$pull` operator will remove any instance of a value from an array.



```
"categories": ["tasty", "budget"]
```

```
> db.potions.update(  
  { "name": "Shrinking" },  
  { "$pull": { "categories": "tasty" } })
```

Result

```
"categories": ["budget"]
```



If value isn't unique, then all instances will be removed from the array.

Materializing Potions

Level 3 - Section 1

Query Operators

Adding a Filter for Potions

We've received a new feature request to allow users to filter potions based on multiple criteria.

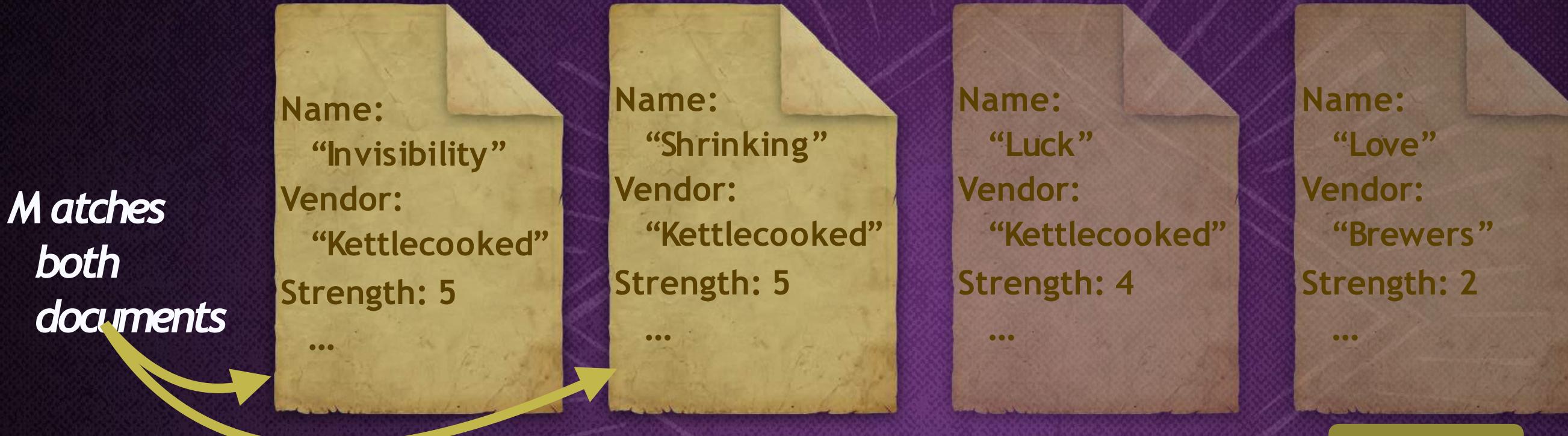
The screenshot shows a web browser window with a title bar and a main content area. The content area has a header 'Potion Reviews'. On the left, there are two filter sections: 'Vendor' and 'Strength'. The 'Vendor' section has a checked checkbox for 'Kettlecooked' and an unchecked checkbox for 'Brewers'. The 'Strength' section has a checked checkbox for '5' and an unchecked checkbox for '4'. Below these filters, there are three potion cards: 'Invisibility', 'Love', and 'Shrinking'. Each card includes an image of the potion bottle, its name, its score, taste rating, strength rating, and a 'More Info' button.

Potion	Score	Taste	Strength
Invisibility	70	4	1
Love	84	3	5
Shrinking	94	2	3

*Only show potions made by Kettlecooked
that have a strength of 5*

Querying With Multiple Criteria

We can query based on multiple criteria by passing in comma-separated queries.



> db.potions.find(
> {
→ "vendor": "Kettlecooked",
→ "ratings.strength": 5
}
)

We can pass in more than 1 query

Finding Potions Based on Conditions

Queries of equality are great, but sometimes we'll need to query based on conditions.

Potion Reviews

Ingredients

- Laughter
- Unicorn
- +more

Vendor

- Kettlecooked
- Brewers
- +more

Price

- Under \$10
- Under \$20

Invisibility
Score: 70
Taste: 4 Strength: 1
[More Info](#)

Love
Score: 84
Taste: 3 Strength: 5
[More Info](#)

Shrinking
Score: 94
Taste: 2 Strength: 3
[More Info](#)

Search for potions with a price less than 20

Comparison Query Operators

We can use comparison query operators to match documents based on the comparison of a specified value.

Common Comparisons

`$gt`

greater than

`$gte`

greater than or equal to

`$ne`

not equal to

`$lt`

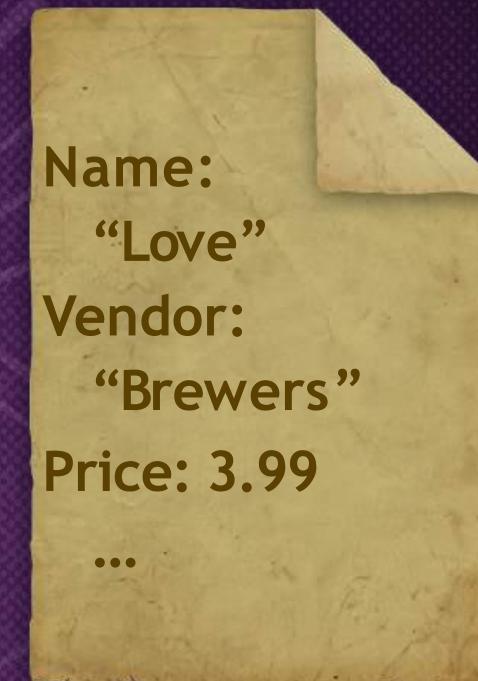
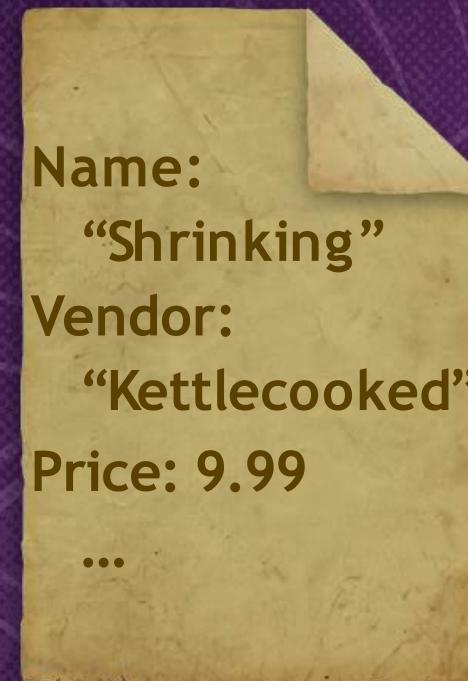
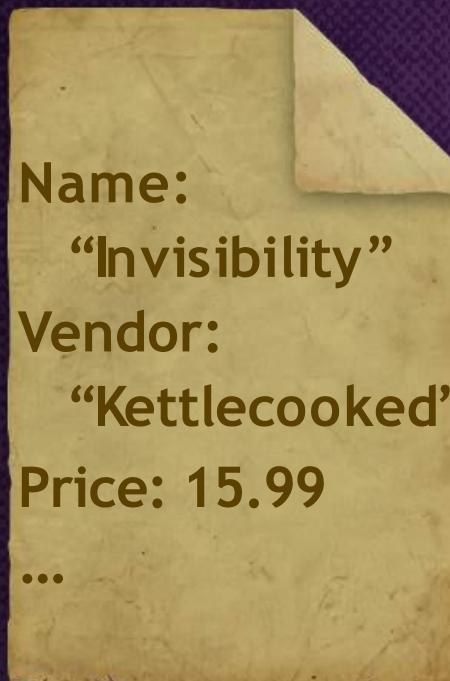
less than

`$lte`

less than or equal to

Finding Potions That Are Less Than \$20

We can match the appropriate documents by using the `$lt` comparison operator.



```
> db.potions.find( { "price": { "$lt": 20 } } )
```

SHELL

Price less than 20

Finding Potions Between Prices

We can query with a range by combining comparison operators.

Name:
“Invisibility”
Vendor:
“Kettlecooked”
Price: 15.99
...

Name:
“Shrinking”
Vendor:
“Kettlecooked”
Price: 9.99
...

Name:
“Luck”
Vendor:
“Kettlecooked”
Price: 59.99
...

Name:
“Love”
Vendor:
“Brewers”
Price: 3.99
...

```
> db.potions.find( { "price": { "$gt": 10, "$lt": 20 } } )
```

Price greater than 10 and
less than 20

SHELL

Queries of Non-equality

We can use the `$ne` operator to find potions with fields that don't equal the specified value.

Name:
“Invisibility”
Vendor:
“Kettlecooked”
Price: 15.99
...

Name:
“Shrinking”
Vendor:
“Kettlecooked”
Price: 9.99
...

Name:
“Luck”
Vendor:
“Kettlecooked”
Price: 59.99
...

Name:
“Love”
Vendor:
“Brewers”
Price: 3.99
...

```
> db.potions.find( { "vendor": { "$ne": "Brewers" } } )
```

SHELL

Vendor not equal to “Brewers”

Range Queries on an Array

Each potion has a size field that contains an array of available sizes. We can use **\$elemMatch** to make sure at least 1 element matches all criteria.

Potion sizes

Name: “Invisibility”
Price: 15.99
Sizes: [34,64,80]
...

Name: “Shrinking”
Price: 9.99
Sizes:[32,64,112]
...

Name: “Luck”
Price: 59.99
Sizes: [10,16,32]
...

Name: “Love”
Price: 3.99
Sizes: [2,8,16]
...

At least 1 value in an array MUST be greater than 8 and less than 16

The value 10 matches!

```
> db.potions.find(  
  { "sizes" : { "$elemMatch": { "$gt": 8, "$lt": 16 } } }  
)
```

SHELL

Be Careful When Querying Arrays With Ranges

What happens when we try to perform a normal range query on an array?

Name:
“Invisibility”
Price: 15.99
Sizes: [34,64,80]
...

Name:
“Shrinking”
Price: 9.99
Sizes:[32,64,112]
...

Name:
“Luck”
Price: 59.99
Sizes: [10,16,32]
...

Name:
“Love”
Price: 3.99
Sizes: [2,8,16]
...

SHELL

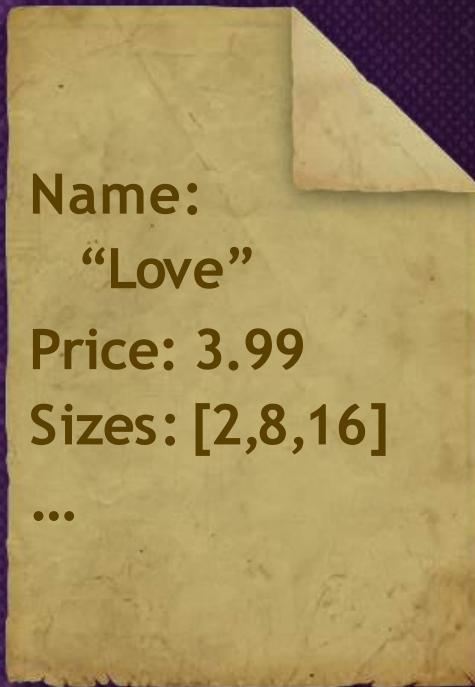
```
> db.potions.find(  
  { "sizes" : { "$gt" : 8, "$lt" : 16 } }  
)
```



Doesn't contain any matching sizes, so why did it match?

Be Careful When Querying Arrays With Ranges

What happens when we try to perform a normal range query on an array?

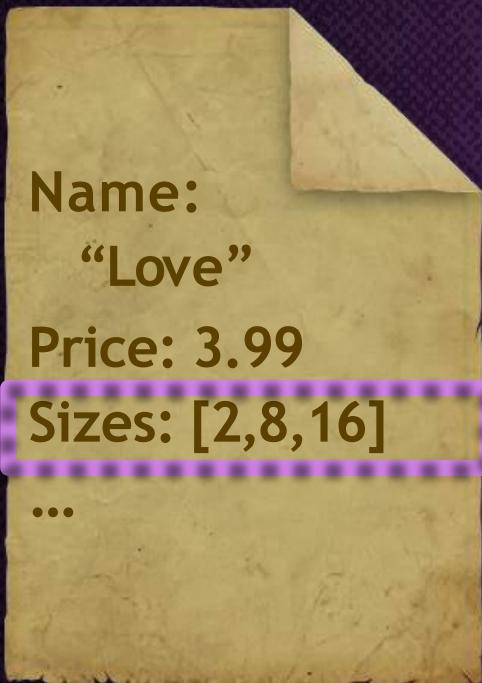


SHELL

```
> db.potions.find(  
  { "sizes" : { "$gt": 8, "$lt": 16 } }  
)
```

Why Did the Document Match?

Each value in the array is checked individually. If at least 1 array value is true for each criteria, the entire document matches.



Range Query

```
{ "sizes": { "$gt": 8, "$lt": 16 } }
```



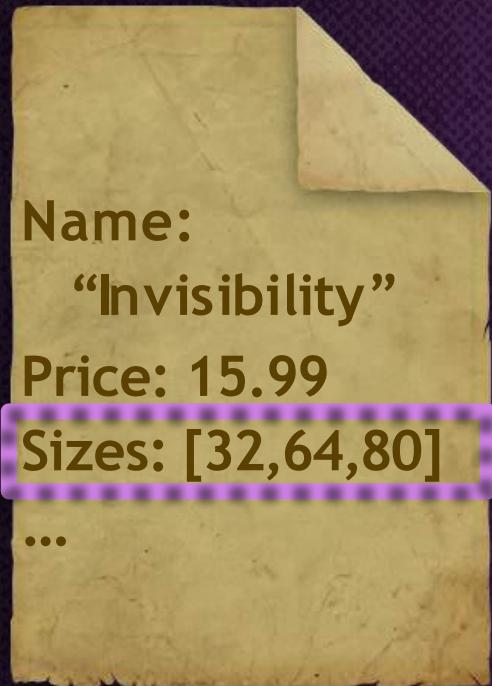
Both criteria are met by at least 1 value

```
"sizes": [2, 8, 16]
```

Not Matching a Document

Conversely, the document will not match if only 1 criteria is met.

Only 1 criteria is met, so the document doesn't match



Range Query

```
{ "sizes": { "$gt": 8, "$lt": 16 } }
```



```
"sizes": [ 32, 64, 80 ]
```

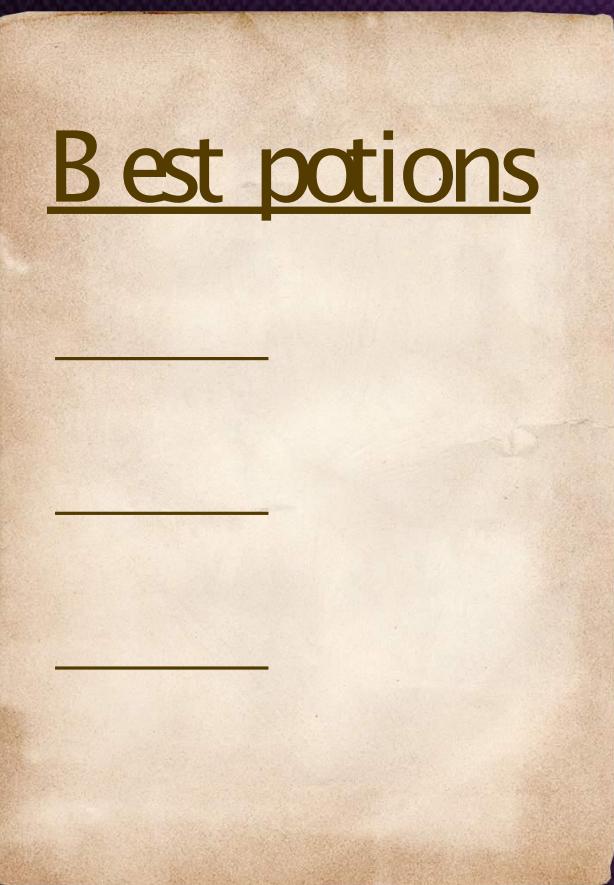
Materializing Potions

Level 3 - Section 2

Customizing Queries

Listing Our Best Potions

We're putting together a list of the best potions we've used. Let's find potions with a grade equal to or greater than 80.



*Need the name and vendor
of potions with a high
grade*



Potions Collection



Introducing Projections

`find()` takes a second parameter called a “projection” that we can use to specify the exact fields we want back by setting their value to true.

SHELL

```
> db.potions.find(  
  { "grade": { "$gte": 80 } },  
  { "vendor": true, "name": true } )  
{  
  "_id": ObjectId(...),  
  "vendor": "Kettlecooked",  
  "name": "Shrinking"  
}  
...
```

When selecting fields, all other fields but the `_id` are automatically set to false



Only retrieve what's needed

Excluding Fields

Sometimes you want all the fields except for a few. In that case, we can exclude specific fields.

SHELL

```
> db.potions.find(  
  { "grade": { "$gte": 80 } },  
  { "vendor": false, "price": false } )  
{  
  "_id": ObjectId(...),  
  "name": "Shrinking",  
  "grade": 94,  
  "ingredients": [...],  
  ...  
}
```

When excluding fields, all fields but those set to false are defaulted to true

★ *Great for removing sensitive data*

Excluding the `_id`

The `_id` field is always returned whenever selecting or excluding fields. It's the only field that can be set to false when selecting other fields.

SHELL

```
> db.potions.find(  
  { "grade": { "$gte": 80 } },  
  { "vendor": true, "price": true, "_id": false }  
)  
{  
  "vendor": "Homebrewed",  
  "price": 9.99  
}
```

*The only time we can mix
an exclusion with selections*



Removing the id is common when preparing data reports for non-developers.

Either Select or Exclude Fields

Whenever projecting, we either select or exclude the fields we want — we don't do both.

```
> db.potions.find(  
  { "grade": { "$gte": 80 } },  
  { "name": true, "vendor": false }  
)
```

SHELL

Causes an error to be raised

ERROR

```
"$err": "Can't canonicalize query: BadValue  
Projection cannot have a mix of inclusion  
and exclusion."
```

Counting Our Potions

Time to advertise our expertise and list the total number of potions we've reviewed.

The screenshot shows a web browser window with a title bar and navigation icons. The main content area displays three potion reviews:

- Invisibility**: Score: 70, Taste: 4, Strength: 1. Includes a "More Info" button.
- Love**: Score: 84, Taste: 3, Strength: 5. Includes a "More Info" button.
- Shrinking**: Score: 94, Taste: 2, Strength: 3. Includes a "More Info" button.

On the left side, there are filters for Ingredients (Laughter, Unicorn, +more), Vendor (Kettlecooked, Brewers, +more), and Price (Under \$10, Under \$20). A prominent banner at the top states "Over 10,000 Potion Reviews!" with a large yellow arrow pointing to it.

*Need to count the total number
of potions in the potions
collection*

Introducing the Cursor

Whenever we search for documents, an object is returned from the `find` method called a "cursor object."

SHELL

```
> db.potions.find( { "vendor": "Kettlecooked" } )  
{ "_id": ObjectId(...), ... }  
{ "_id": ObjectId(...), ... }  
{ "_id": ObjectId(...), ... }  
...
```

First 20
documents

By default, the first 20 documents
are printed out

Iterating Through the Cursor

When there are more than 20 documents, the cursor will iterate through them 20 at a time.

```
db.potions.find()
```



*Sends 20
documents*

SHELL

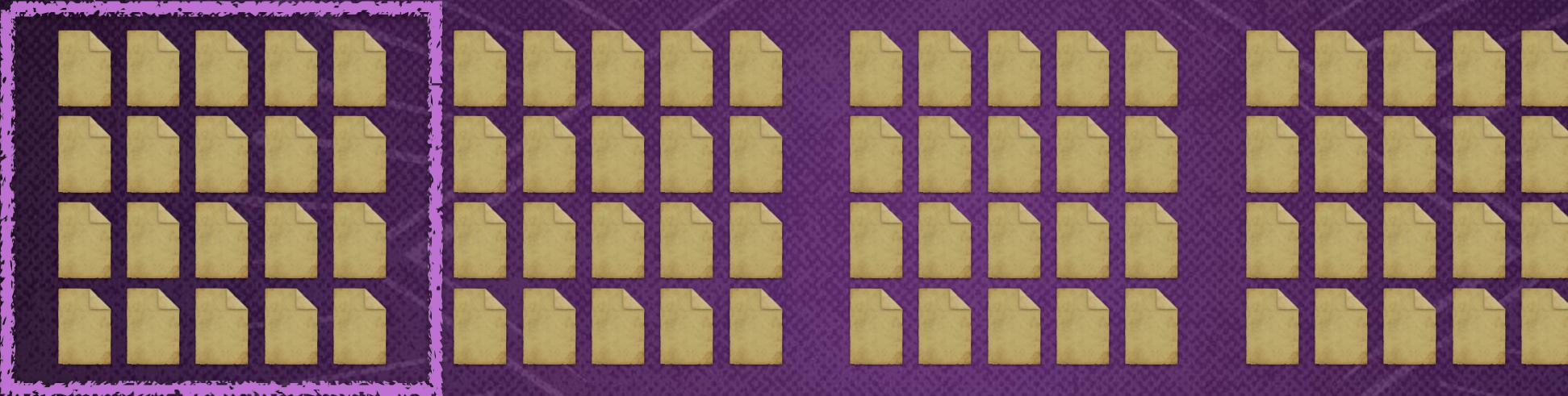
```
...  
{ "_id": ObjectId(...), "name": ... }  
{ "_id": ObjectId(...), "name": ... }  
{ "_id": ObjectId(...), "name": ... }
```

type "it" for more

Continuing to Iterate Through the Cursor

Typing “it” will display the next 20 documents in the cursor.

```
db.potions.find()
```



SHELL

Next
batch
sent

```
> it
```

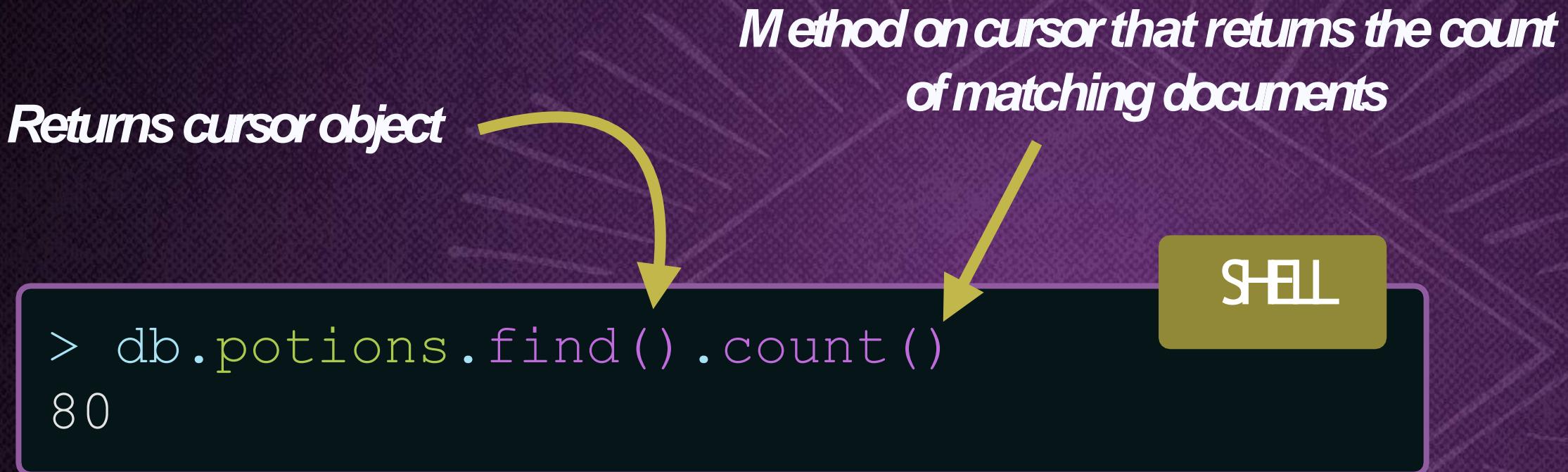
Iterates the cursor

```
{ "_id": ObjectId(...), "name": ... }  
{ "_id": ObjectId(...), "name": ... }  
...  
type "it" for more
```

We'll continue being prompted until no documents are left

Cursor Methods

Since the cursor is actually an object, we can chain methods on it.



Cursor methods always come after `find()` since it returns the cursor object.

Sort by Price

We want to implement a way for users to sort potions by price.

Potion Reviews

Ingredients + more

Vendor + more

Price + more

Sort

Price high

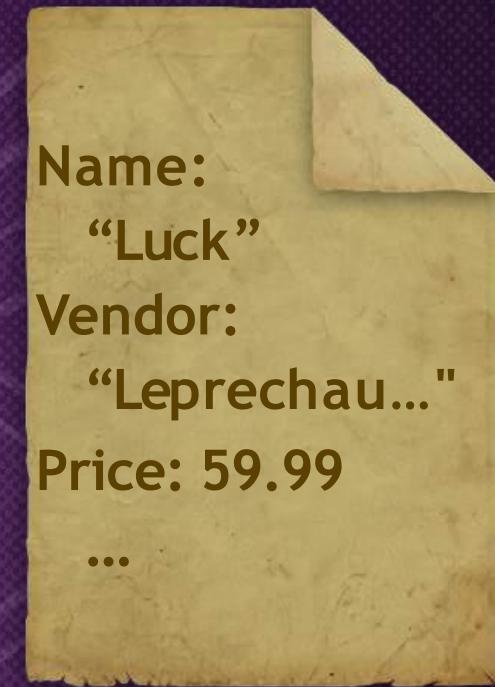
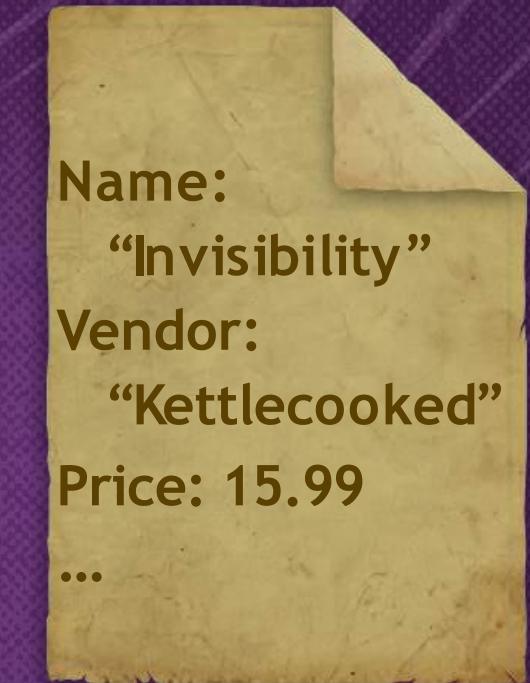
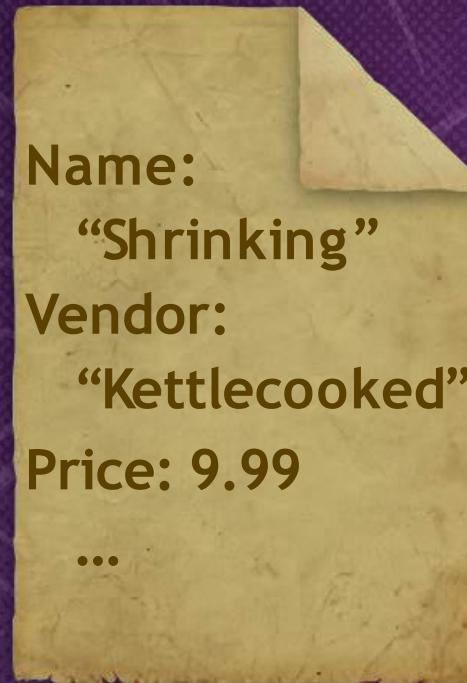
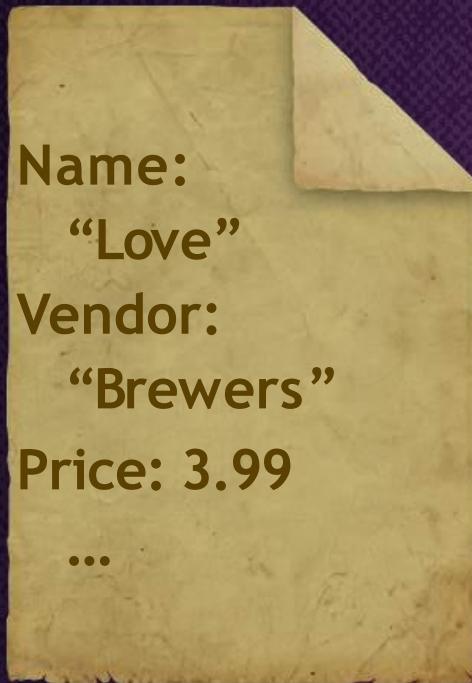
Price low

Potion	Score	Taste	Strength
Invisibility	70	4	1
Love	84	3	5
Shrinking	94	2	3

*Sort potions with the
lowest price first*

Sorting Potions

We can use the `sort()` cursor method to sort documents.



SHELL

```
> db.potions.find().sort({ "price": 1 })
```

Field to sort

-1 *to order descending*
1 *to order ascending*

Paginating the Potions Page

We only want to show 3 potions per page. Time to implement pagination!

Invisibility

Score: 70

Taste: 4 Strength: 1

More Info

Love

Score: 84

Taste: 3 Strength: 5

More Info

Shrinking

Score: 94

Taste: 2 Strength: 3

More Info

< Back

Next >

Paginate results so we only see 3 potions on each page

Basic Pagination

We can implement basic pagination by limiting and skipping over documents. To do this, we'll use the `skip()` and `limit()` cursor methods.

Page 1



Skip 0, Limit 3

```
> db.potions.find().limit(3)
```

SHELL

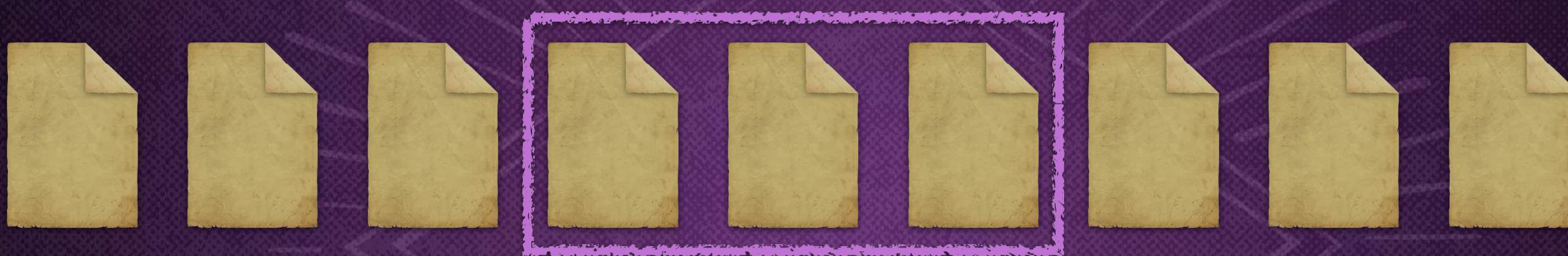
Since we're not skipping, we
can leave off the skip
method and just limit 3



Basic Pagination

We can implement basic pagination by limiting and skipping over documents.

Page 2



Skip 3, Limit 3

SHELL

```
> db.potions.find().skip(3).limit(3)
```

Basic Pagination

We can implement basic pagination by limiting and skipping over documents.

Page 3



Skip 6, Limit 3

SHELL

```
> db.potions.find().skip(6).limit(3)
```



This approach can become really
expensive with large collections.

Morphing Models

Level 4 - Section 1

Data Modeling

Introducing User Profiles

We've added a user profile page to the site and want to allow users to enter up to 3 of their favorite potions.

The screenshot shows a web browser window with a title bar containing icons for back, forward, and home. The main content area has a header 'Potion Reviews' and a sub-header 'User Profile'. It contains two input fields: 'Username' with the value 'Mukthar' and 'E-mail' with the value 'myself@example.com'. Below these is a button labeled 'Update'. To the right of the form, there is a placeholder for a 'Favorite Potion' field with the value 'Luck' and two empty input fields below it.

User Document

```
{  
  "_id": ObjectId(...),  
  "username": "Mukthar",  
  "email": "myself@example.com",  
  "favorites": "Luck"  
}
```



Need a way to store up
to 3 potions

Storing Favorites Within a User

Users and their favorites are strongly related and will be used together often.



Favorites

User Document

```
{  
  "_id": ObjectId(...),  
  "username": "Mukthar",  
  "email": "myself@example.com",  
  "favorites": [  
    "Newt Tonic",  
    "Sleeping",  
    "Love"  
  ]  
}
```

*Use an array to hold
multiple values*

Adding Vendor Information

We'd like to add more vendor information to our potions so users can be more informed about where they get their potions.

Potion Document

Vendor



Inserting the data as an embedded document

```
{  
  "_id": ObjectId(...),  
  "name": "Invisibility",  
  ...  
  "vendor": {  
    "name": "Kettlecooked",  
    "phone": 5555555555,  
    "organic": true  
  }  
}
```

Storing Vendor Information

Each potion now contains its vendor information, which means we have vendor information repeated throughout our documents.

Name:
“Invisi...”

Vendor:
“Kettlecooked”,
“Organic”

...

Name:
“Shrink...”

Vendor:
“Kettlecooked”,
“Organic”

...

Name:
“Love”

Vendor:
VendorPoof
“Brewers”,
“Non-Organic”

“Non-Organic”

...

Name:
“Luck”

Vendor:
“Leprechau...”,
“Organic”

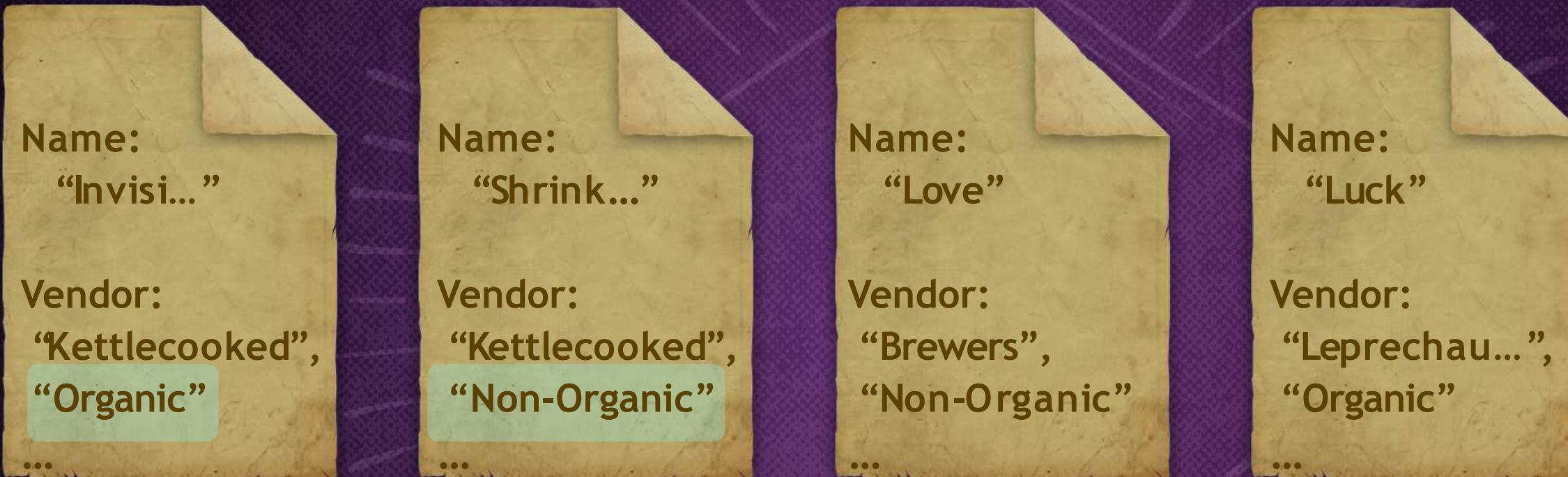
...



*We're going to have duplicate
vendor information for each
potion.*

Dangers of Duplication

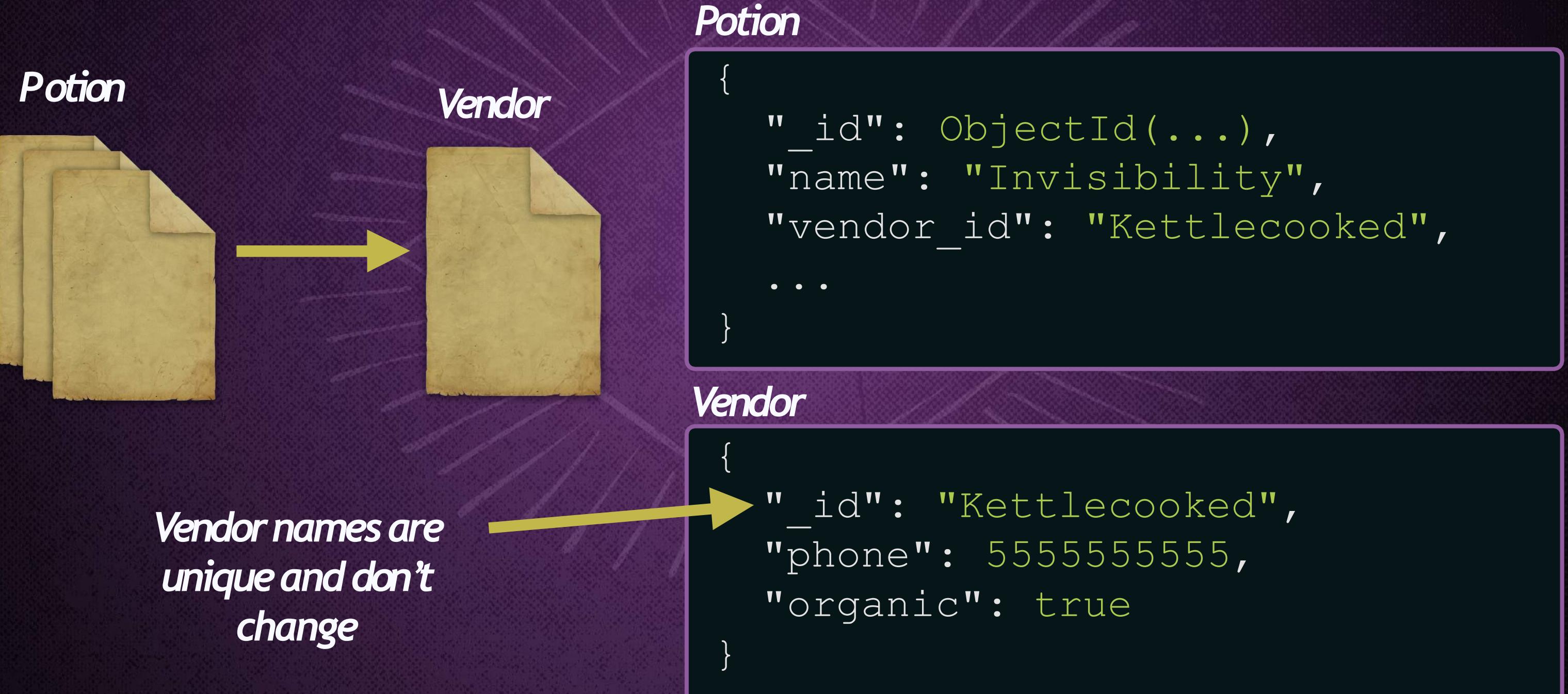
Duplicate data can be hard to keep consistent throughout the database.



*If 1 potion gets updated with new information and
the rest don't, our data is no longer correct*

Referencing Vendor Information

Instead of embedding the vendor information, we can create a vendors collection and reference the vendor document in each potion document.



Inserting Referenced Documents

```
> db.vendors.insert({  
  "_id": "Kettlecooked",  
  "phone": 5555555555,  
  "organic": true  
})
```

SHELL

*We can specify the unique
_id of document*

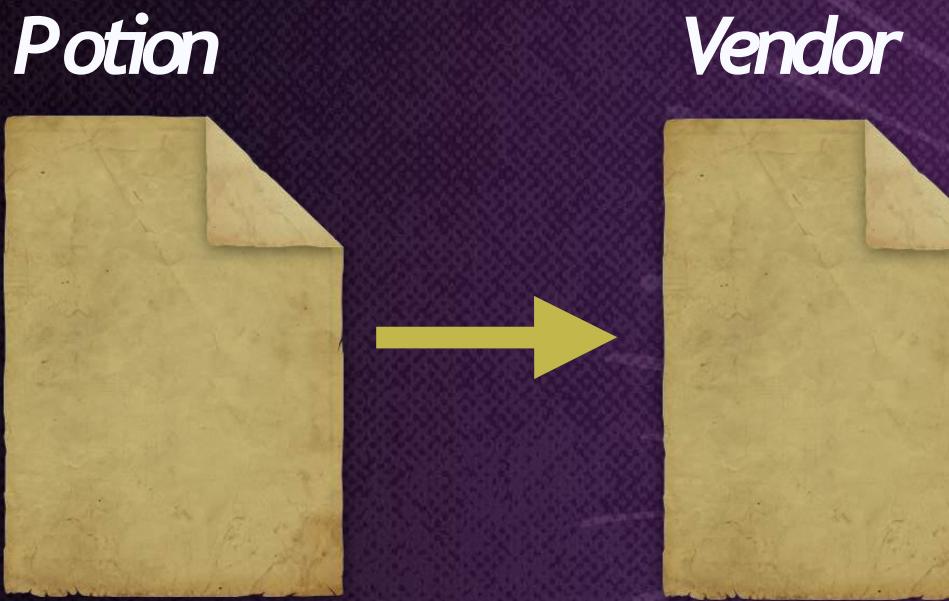
```
> db.potions.insert({  
  "name": "Invisibility",  
  "vendor_id": "Kettlecooked"  
  ...  
})
```

SHELL

Referenced document

Querying a Referenced Document

In order to pull a potion document and the vendor information, we must first query for the potion to get the vendor_id and then query once more to get their vendor information.



```
{  
  "_id": ObjectId(...),  
  "name": "Invisibility",  
  "vendor_id": "Kettlecooked",  
  ...  
}
```

First, query to retrieve potion information

```
db.potions.find( { "name": "Invisibility" } )
```

Second, query to retrieve vendor information

```
db.vendors.find( { "_id": "Kettlecooked" } )
```

We get the vendor_id
from the first query

Some Features of Embedded Documents



Data readily available

With a single query, we can grab a user's email and their addresses

SHELL

```
db.users.find( { } , { "email": true, "favorites": true } )
```

```
{  
  "_id": ObjectId(...),  
  "email": "azrius@example.com"  
  "favorites": [  
    "Newt Tonic",  
    "Sleeping",  
    "Love"  
  ]  
}
```

More Features of Embedded Documents



Data readily available

SHELL

```
db.users.find( { } , { "email": true, "favorites": true } )
```

Atomic write operations

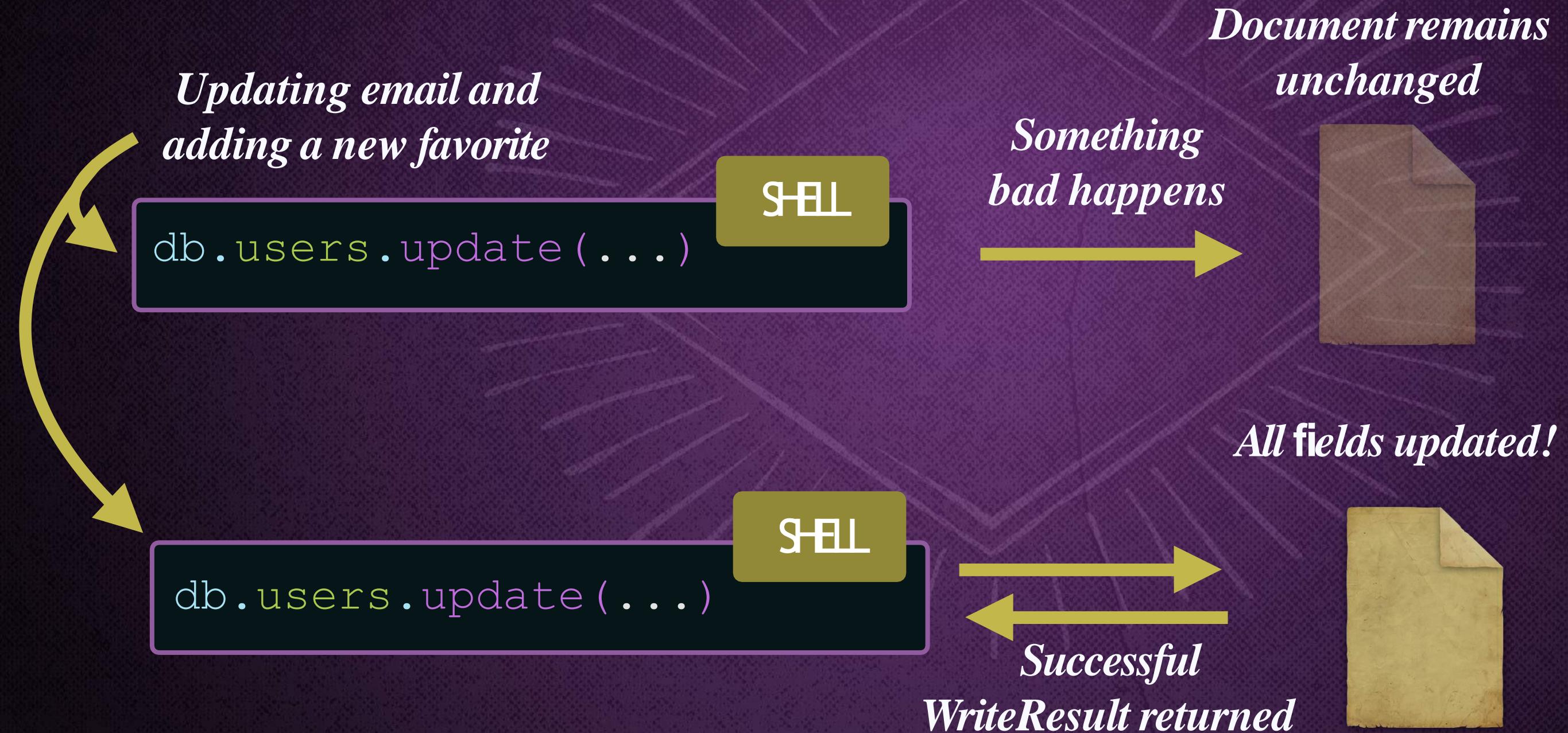
SHELL

```
db.users.insert({  
  "username": "Azrius",  
  "email": "azrius@example.com",  
  "favorites": ["Newt...", "Sleeping", "Love"]  
})
```

*Guarantee that the document write
completely happens or doesn't at all*

Why Does Atomicity Matter?

If we update a user's email and add a favorite potion, but an error occurs in the favorites portion of the update, then none of the update will occur.



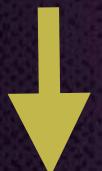
Referenced Documents Exist Independently

Since we've referenced vendor documents, they now exist entirely on their own outside of potion documents.

One place to edit vendor information



```
db.vendors.update({ "_id": ObjectId(...) }, { ... })
```



Multi-document writes not supported

New Potion



```
db.potions.insert({ ... })
```

SHELL

New Potion's Vendor

```
db.vendors.insert({ ... })
```

SHELL

No guarantee that both write operations will occur

Multi-document Write Operations

Currently, MongoDB doesn't support multi-document writes. We aren't able to guarantee write operations on an atomic level.

Adding new potion

```
db.potions.insert(...)
```

SHELL

Successful write



New potion created!

Adding the new vendor

```
db.vendors.insert(...)
```

SHELL

Error occurs



*No vendor
document
created*

Dangers of Not Having Transactions

MongoDB doesn't recognize document relationships, so we now have a potion with a nonexistent vendor.



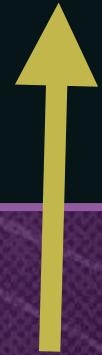
New potion created!



*No vendor
document
created*

Potion document

```
{  
  "_id": ObjectId(...),  
  "name": "Time Travel",  
  "vendor_id": "Magical Inc.",  
  ...  
}
```



The references still exist!

*Vendor never got
created!*

Morphing Models

Level 4 - Section 2

Data Modeling Decisions

Choosing Which Route to Take

When deciding how to model a one-to-many relationship, we must carefully consider how our data will be used in order to decide between embedding and referencing.

Embedding

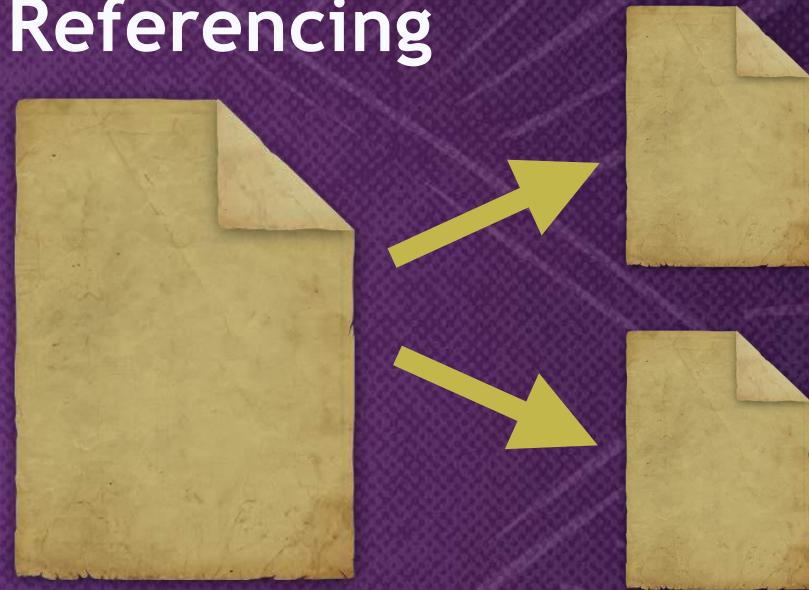


Single query

Documents accessed through parent

Atomic writes

Referencing



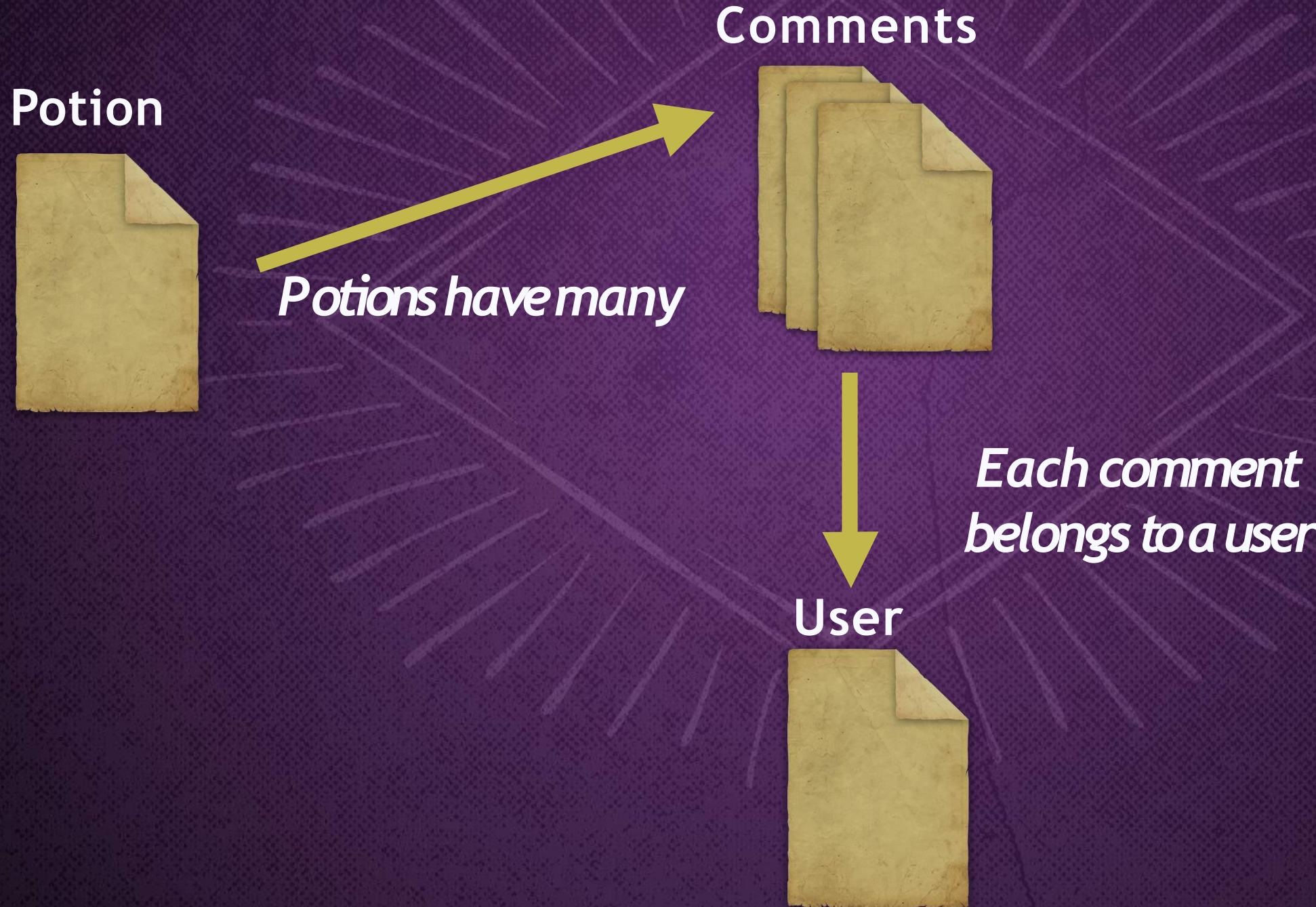
Requires 2 queries

Documents exist independently

Doesn't support multi-document writes

Adding Comments to Potions

We'd like to allow users to comment on the potions and need to determine the best route.



First Question: How Will the Data Be Used?

Data that is frequently used together will benefit from being embedded while data that's rarely used can afford the cost of referencing.

How often is the data used together?	Always	Sometimes	Rarely
Embed	✓	✓	✓
Reference		✓	✓

Embedding will work most of the time

Either option can work effectively

Data Usage: Potion Comments

Potion Reviews

Invisibility

Score: 70

Taste: 4 Strength: 1

Comments

I don't agree. You are wrong.
-Sauron

✓ Embedding

✗ Referencing

Whenever we display potions, we'll always want to display comments

Username displayed for each comment

We would be forced to perform multiple queries

Second Question: What's the Size of the Data?

The size of the data in a relationship has a significant impact on data modeling. It's crucial to think ahead!

Expected Size	Less than 100	More than a few hundred	Thousands
Embed	✓	✓	
Reference		✓	✓

Might start to see a decline in read performance when embedded

Data Size: Potion Reviews

Potion Reviews

Invisibility

Score: 70
Taste: 4 Strength: 1

Comments

I don't agree. You are wrong.
-Sauron

✓ Embedding

✗ Referencing

When the data size is over 100, we can consider referencing

Most potions won't get more than 50 comments, and each comment has a single author

Third Question: Will the Data Change Often?

Sometimes embedding data can lead to data duplication. Depending on whether or not the duplicate data changes a lot can factor into our decision making.

Frequency of Change	Never/Rarely	Occasionally	Constantly
Embed	✓	✓	
Reference		✓	✓

Data duplication is okay if we don't expect change

Depends on whether or not we want the overhead of managing duplication

Prevents inconsistencies from duplication

Data Change: Potion Comments



Embedding



Referencing



Embedding



Referencing

Requires updating all of a user's comments if their information changes

Embedding Comments in Potions

We can confidently embed comments within potions. We know we'll have less than 100 comments per potion, they're used together often, and the data doesn't change often.

```
{  
  "name": "Invisibility",  
  ...  
  "comments": [  
    {  
      "title": "The best potion!",  
      "body": "Lorem ipsum abra cadabra"  
    },  
    ...  
  ]  
}
```



*Comments readily
available*

Referencing Users in Comments

We only need the username from each user, and embedding user documents could lead to duplication issues. Let's reference a user by using his or her unique username.

```
{  
  "name": "Invisibility",  
  ...  
  "comments": [  
    {  
      "title": "The best potion!",  
      "body": "Lorem ipsum abra cadabra",  
      "user_id": "Mukthar"  
    },  
    ...  
  ]  
}
```

Usernames can't be changed and are unique

Data Modeling Guidelines and Takeaways



Generally, embedding is the best starting point



Reference data when you need to access document independently

Consider referencing when you have large data sizes



Focus on how your data will be used



If you find you need complex references, consider a relational database

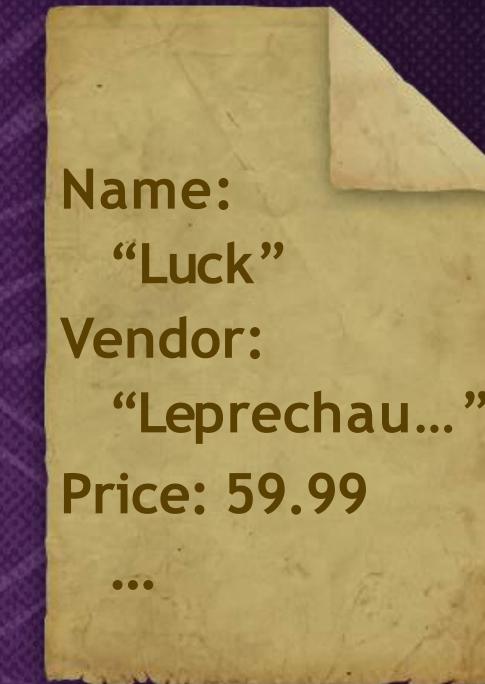
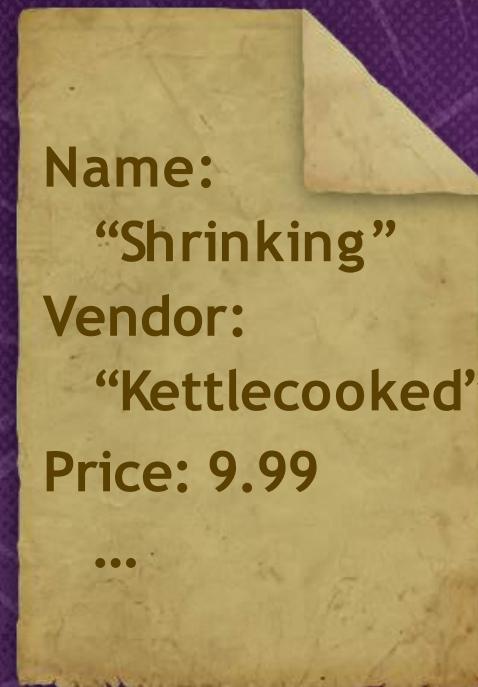
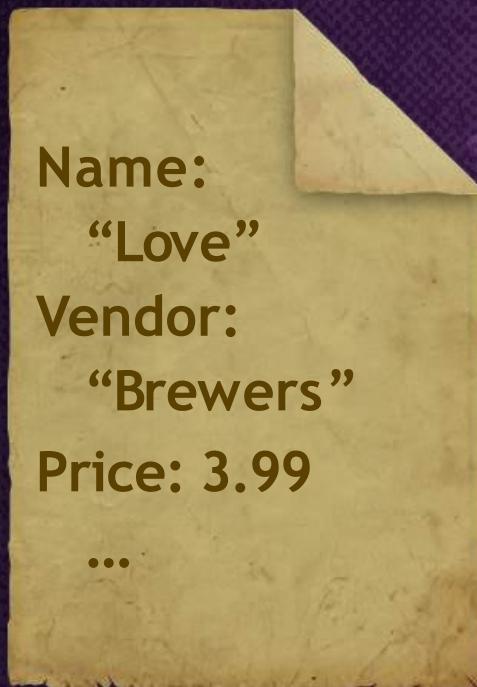
Aggregation Apparitions

Level 5 - Section 1

Common Aggregations

Finding the Number of Potions Per Vendor

Time for an audit! We need to know how many potions we have per vendor.



```
> db.potions.find( {}, { "name": true, "vendor": true } )
```

We could manually pull all the data and count everything,
but it's better to have MongoDB handle that for us!

Introducing the Aggregation Framework

The aggregation framework allows for advanced computations.

A screenshot of a MongoDB shell window titled "SHELL". Inside the window, a command is entered:

```
> db.potions.aggregate(  
  [ { "$group": { "_id": "$vendor_id" } } ]  
)
```

Annotations with yellow arrows point to specific parts of the code:

- An arrow points to the word "aggregate" with the text *Takes stage operators as parameters*.
- An arrow points to the first brace of the array with the text *Stage operator that's used to group data by any field we specify*.
- An arrow points to the "\$group" stage with the text *Field names that begin with a "\$" are called "field paths" and are links to a field in a document*.
- An arrow points to the closing brace of the array with the text *Go within an array*.



"Aggregate" is a fancy word for combining data.

Using the Aggregation Framework to Group Data

```
> db.potions.aggregate(  
  [ { "$group": { "_id": "$vendor_id" } } ]  
)
```

SHELL

This is known as the ‘group key’ and is required

Potions Collection



\$group
→

Results

```
{ "_id": "Kettlecooked" },  
{ "_id": "Brewers" },  
{ "_id": "Leprechaun Inc" }
```

*Returns result object
containing the unique vendors
in the inventory collection*

Using Accumulators

Anything specified after the group key is considered an accumulator. Accumulators take a single expression and compute the expression for grouped documents.

SHELL

```
> db.potions.aggregate([  
  { "$group": { "_id": "$vendor_id", "total": { "$sum": 1 } } }  
])
```

Group key

Will add 1 for each matching document

Accumulator

Inventory



\$group

Results

Total number of documents per vendor!

```
{ "_id": "Kettlecooked", "total": 2 },  
{ "_id": "Brewers", "total": 1, },  
{ "_id": "Leprechaun Inc", "total": 1 }
```

Field Paths Vs. Operators

When values begin with a '\$', they represent field paths that point to the value

```
> db.potions.aggregate([  
  { "$group": { "_id": "$vendor_id", "total": { "$sum": 1 } } }  
])
```

SHELL

When fields begin with a '\$', they are operators that perform a task

Summing the Grade Per Vendor

```
> db.potions.aggregate([
  { "$group": {
    "_id": "$vendor_id",
    "total": { "$sum": 1 },
    "grade_total": { "$sum": "$grade" }
  }}
])
```

SHELL

Field path

Sums the grade values for potions in their group

Results

```
{"_id": "Kettlecooked", "total": 2, "grade_total": 400},
{"_id": "Brewers", "total": 1, "grade_total": 340},
{"_id": "Leprechaun Inc", "total": 1, "grade_total": 92}
```

Averaging Potion Grade Per Vendor

Name: "Invisibility"
Vendor: "Kettlecooked"
Grade: 70
...

Name: "Shrinking"
Vendor: "Kettlecooked"
Grade: 94
...

Name: "Love"
Vendor: "Brewers"
Grade: 84
...

Name: "Sleep"
Vendor: "Brewers"
Grade: 30
...

> db.potions.aggregate([
 { "\$group": {
 "_id": "\$vendor_id",
 "avg_grade": { "\$avg": "\$grade" }
 } }])

Results

{ "_id": "Kettlecooked", "avg_grade": 82 },
{ "_id": "Brewers", "avg_grade": 57 }

Returning the Max Grade Per Vendor

Name: "Invisibility"
Vendor: "Kettlecooked"
Grade: 70
...

Name: "Shrinking"
Vendor: "Kettlecooked"
Grade: 94
...

Name: "Love"
Vendor: "Brewers"
Grade: 84
...

Name: "Sleep"
Vendor: "Brewers"
Grade: 30
...

```
> db.potions.aggregate([
  { "$group": {
    "_id": "$vendor_id",
    "max_grade": { "$max": "$grade" }
  }
])
```

Results

```
{ "_id": "Kettlecooked", "max_grade": 94 },
{ "_id": "Brewers", "max_grade": 84 }
```

Using \$max and \$min Together

Name: "Invisibility"
Vendor: "Kettlecooked"
Grade: 70
...

Name: "Love"
Vendor: "Brewers"
Grade: 84
...

Name: "Shrinking"
Vendor: "Kettlecooked"
Grade: 94
...

Name: "Sleep"
Vendor: "Brewers"
Grade: 30
...

Results

```
{ "_id": "Kettlecooked", "max_grade": 94, "min_grade": 70 },  
{ "_id": "Brewers", "max_grade": 84, "min_grade": 30 }
```

```
> db.potions.aggregate([  
  { "$group": {  
    "_id": "$vendor_id",  
    "max_grade": { "$max": "$grade" },  
    "min_grade": { "$min": "$grade" }  
  }}  
])
```

SHELL

We can use the same field in multiple accumulators

Aggregation Apparitions

Level 5 - Section 2

The Aggregation Pipeline

Pulling Conditional Vendor Information

It turns out that potions made with unicorn aren't permitted, so we need to count the number of potions per vendor that contain it.

Notice

*All potions containing
unicorn are strictly
forbidden*

Steps to find potions with unicorns:

- 1) *Query potions*
- 2) *Group by vendor*
- 3) *Sum the number of potions per vendor*

Introducing the Aggregation Pipeline

The aggregate method acts like a pipeline, where we can pass data through many stages in order to change it along the way.

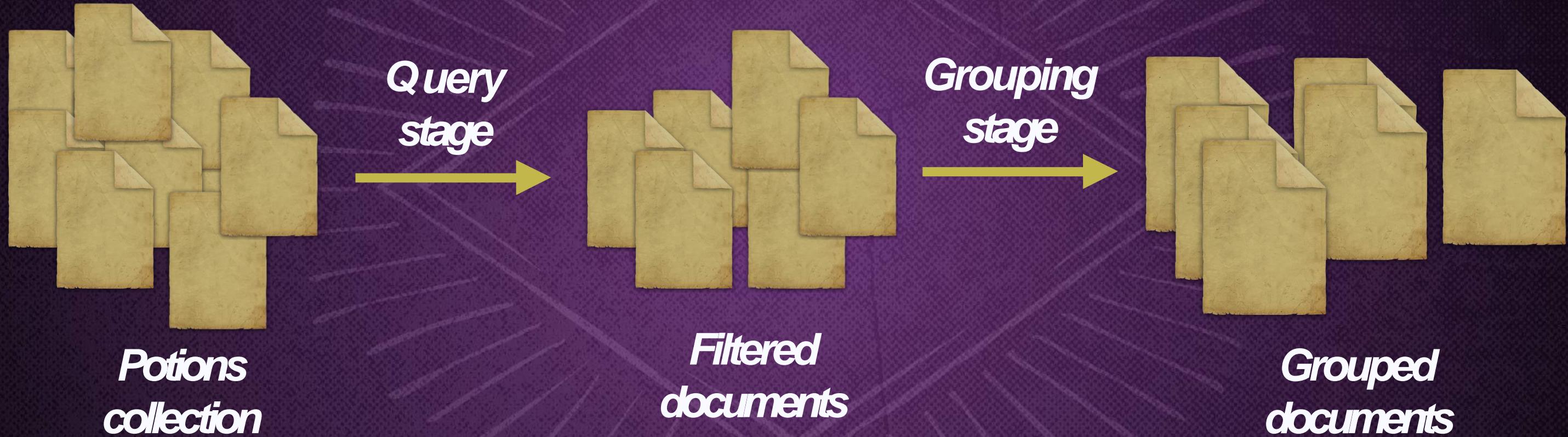
```
db.potions.aggregate( [stage, stage, stage] )
```

We've already seen what a stage looks like in the last section

```
db.potions.aggregate([  
  { "$group": {"_id": "$vendor_id", "total": {"$sum": 1} } }  
])
```

How the Pipeline Works

Each stage modifies the working data set and then passes the altered documents to the next stage until we get our desired result.



Using the \$match Stage Operator

\$match is just like a normal query and will only pass documents to the next stage if they meet the specified condition(s).

```
db.potions.aggregate([
  { "$match": { "ingredients": "unicorn" } }
])
```

SHELL

We can use the same query
we would use with find()

Use match early to reduce the number of documents for
better performance.



Grouping Potions With Data

SHELL

```
db.potions.aggregate ( [  
  { "$match": { "ingredients": "unicorn" } },  
  { "$group":  
    {  
      "_id": "$vendor_id",  
      "potion_count": { "$sum": 1 }  
    }  
  }  
] )
```

*2 stages separated by
comma within an array*

Result

```
{ "_id": "Poof", "potion_count": 20 },  
{ "_id": "Kettlecooked", "potion_count": 1 }
```

Top 3 Vendors With Potions Under \$15

Best value

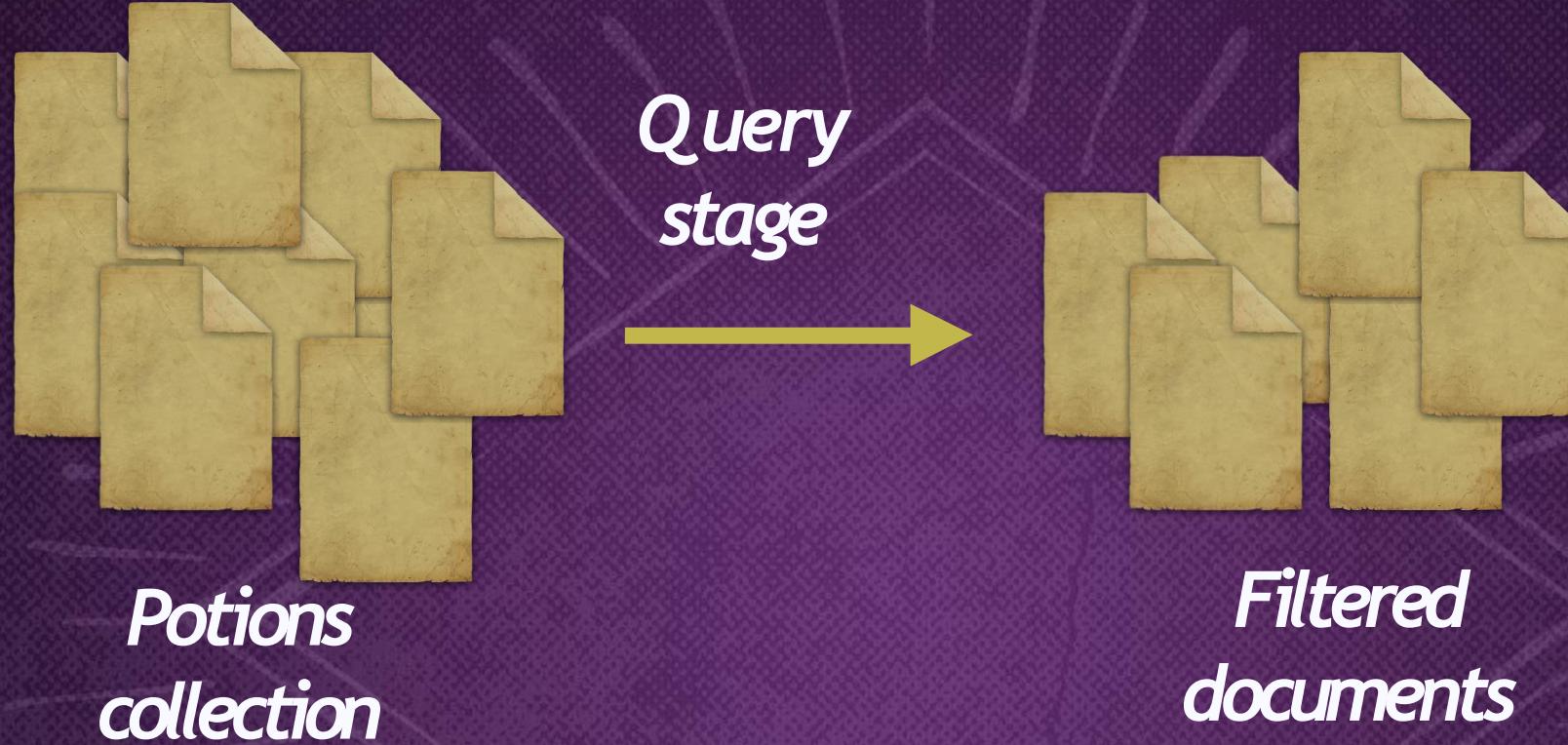
1) Query for potions with a price less than 15

2) Group potions by vendor and average their grades

3) Sort the results by grade average

4) Limit results to only 3 vendors

Matching Potions Under \$15



```
db.potions.aggregate([  
  { "$match": { "price": { "$lt": 15 } } }  
])
```

SHELL

It's good practice to limit the number of results early on

Matching Potions Under \$15

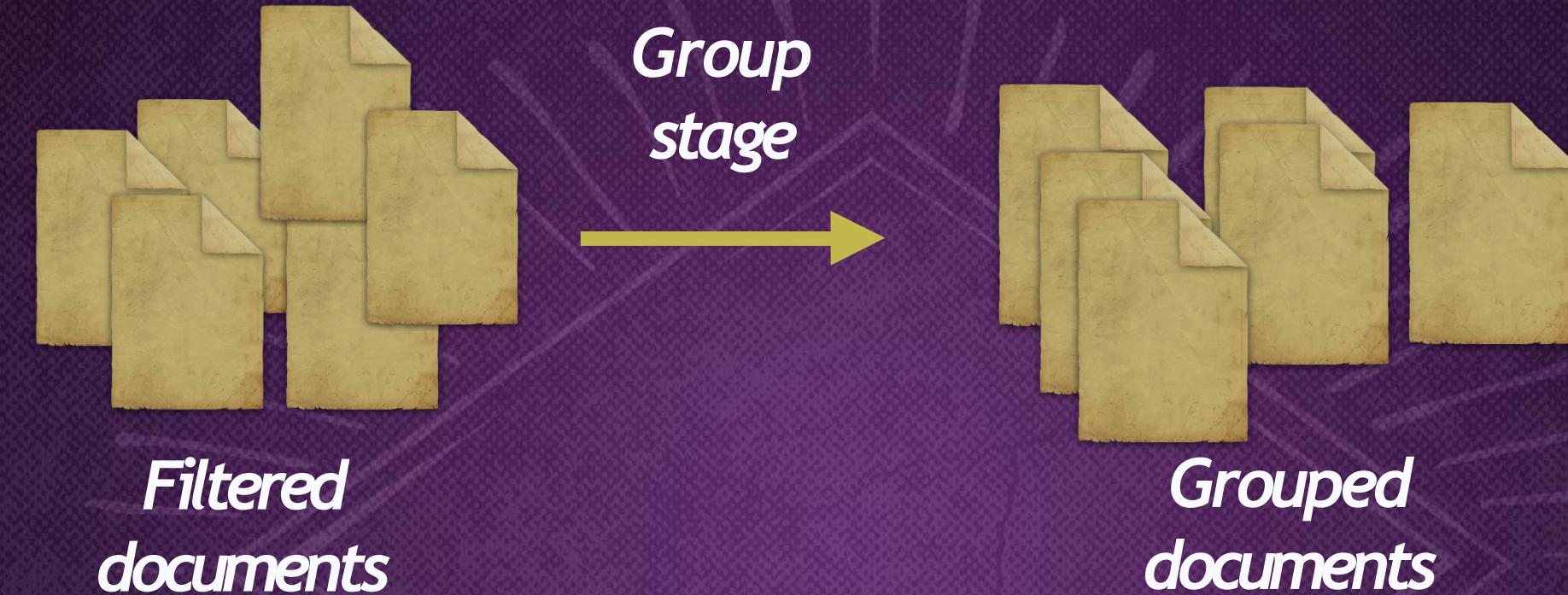


*Filtered
documents*

```
db.potions.aggregate([  
  { "$match": { "price": { "$lt": 15 } } }  
])
```

SHELL

Grouping Potions by Vendor



```
db.potions.aggregate([  
  { "$match": { "price": { "$lt": 15 } } },  
  { "$group": { "_id": "$vendor_id", "avg_grade": { "$avg": "$grade" } } }  
])
```

SHELL

Vendor name as group key

Average of potion grades

Sorting Vendors by Average Grade

We can sort potions by using the `$sort` stage operator.



```
db.potions.aggregate([  
  {"$match": {"price": {"$lt": 15}}},  
  {"$group": {"_id": "$vendor_id", "avg_grade": {"$avg": "$grade"}},  
  {"$sort": {"avg_grade": -1}}])
```

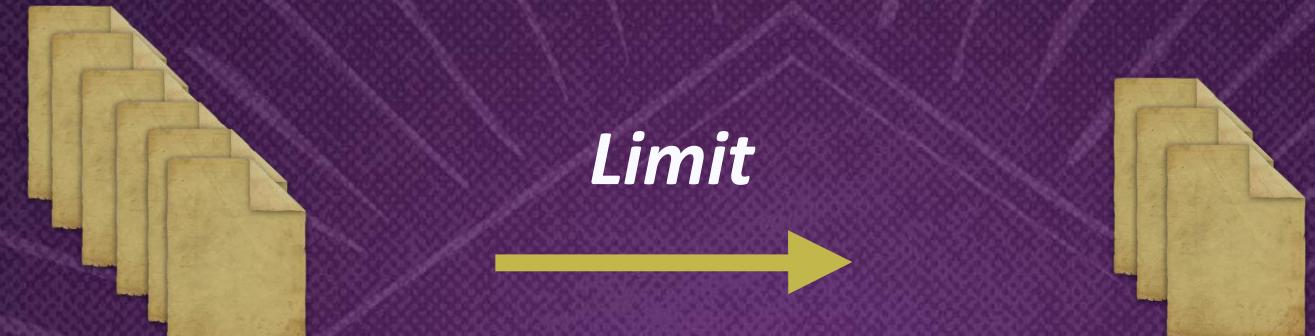
SHELL

We can sort the field we created during the group stage

Sort by the field in descending order

Limiting the Number of Documents

We can use the **\$limit** stage operator to limit the number of documents that get passed on.



Sorted documents

Limited documents

SHELL

```
db.potions.aggregate([
  { "$match": { "price": { "$lt": 15 } } },
  { "$group": { "_id": "$vendor_id", "avg_grade": { "$avg": "$grade" } } },
  { "$sort": { "avg_grade": -1 } },
  { "$limit": 3 } ] )
```

Specify the number of documents to limit

Optimizing the Pipeline

It's best practice to only send the needed data from stage to stage. Let's see what data we really need.

```
db.potions.aggregate( [  
  { "$match": { "price": { "$lt": 15 } } },  
  { "$group": { "_id": "$vendor_id", "avg_grade": { "$avg": "$grade" } } },  
  { "$sort": { "avg_grade": -1 } },  
  { "$limit": 3 }  
] )
```

SHELL



Only need vendor and grade for each potion after the match stage

Projections While Aggregating

We can limit the fields we send over by using `$project`, which functions the same way as projections when we're querying with `find()`.

```
db.potions.aggregate( [  
  { "$match": { "price": { "$lt": 15 } } },  
  { "$project": { "_id": false, "vendor_id": true, "grade": true } },  
  { "$group": { "_id": "$vendor_id", "avg_grade": { "$avg": "$grade" } } },  
  { "$sort": { "avg_grade": -1 } },  
  { "$limit": 3 }  
]
```

SHELL

Vendor and grade for each potion after the match stage

We want to use \$project as soon as possible



It's common to see \$match and \$project used together early on and throughout the pipeline.

Aggregation Results

SHELL

```
db.potions.aggregate([
  {"$match": {"price": {"$lt": 15}}},
  {"$project": {"_id": false, "vendor_id": true, "grade": true}},
  {"$group": {"_id": "$vendor_id", "avg_grade": {"$avg": "$grade}}},
  {"$sort": {"avg_grade": -1}},
  {"$limit": 3}
])
```

```
{ "_id": "Kettlecooked", "avg_grade": 99 },
{ "_id": "Leprechaun Inc", "avg_grade": 95 },
{ "_id": "Brewers", "avg_grade": 90 }
```



Group key



Average potion grade

