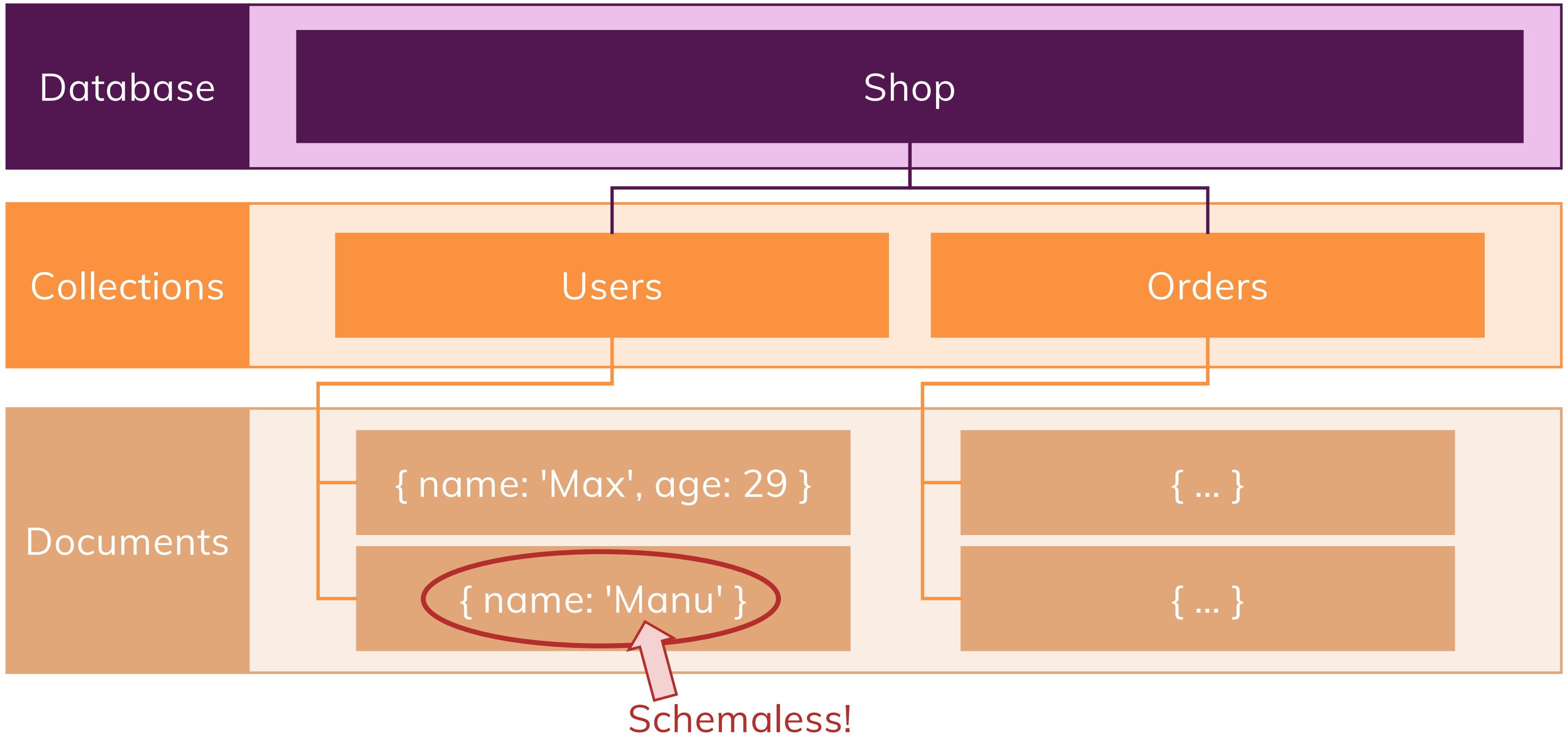
What?

Humongous

Because it can store lots and lots of data

How it works

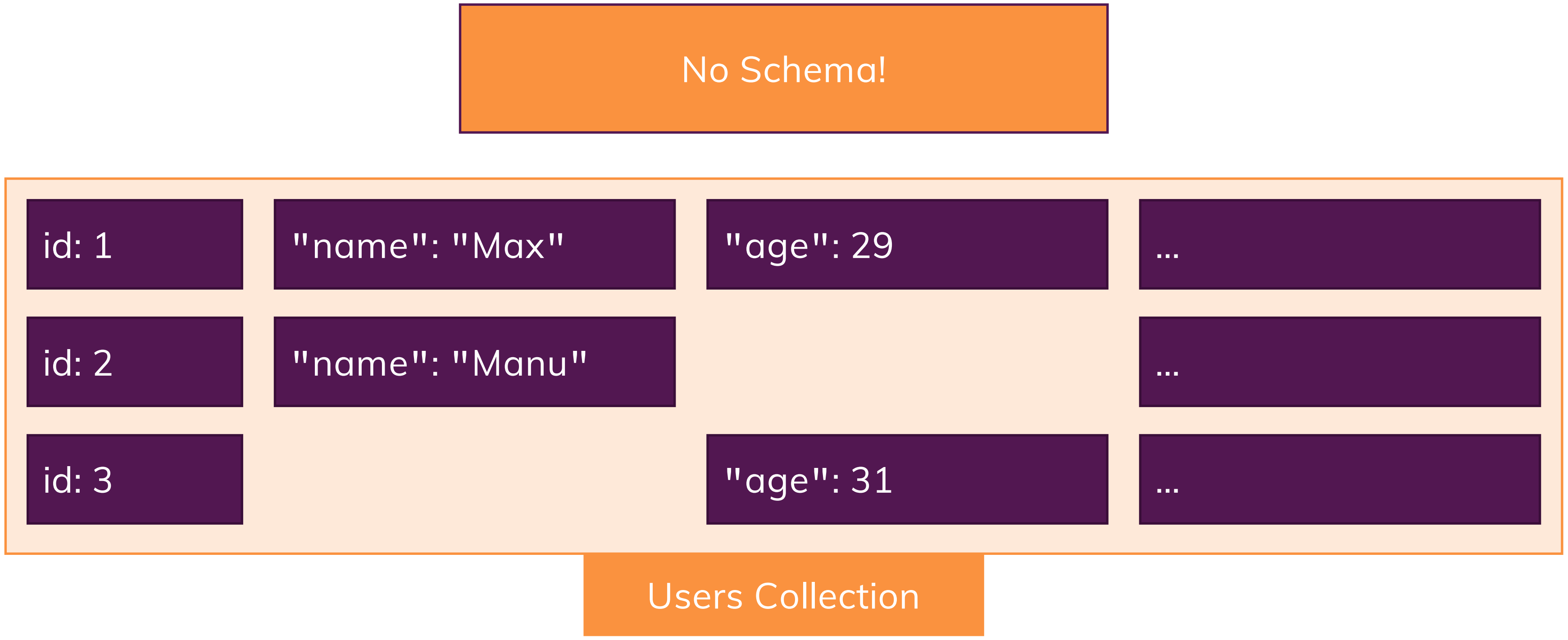


Database
Shop
Collections
Users
Orders
Documents
{ name: 'Max', age: 29 }
{ name: 'Manu' }
{ ... }
{ ... }
Schemaless!

JSON (BSON) Data Format

```
{
     "name": "Max",
     "age": 29,
     "address":
          {
               "city": "Munich"
          },
     "hobbies": [
          { "name": "Cooking" },
          { "name": "Sports" }
     ]
}
```

BSON Data Structure



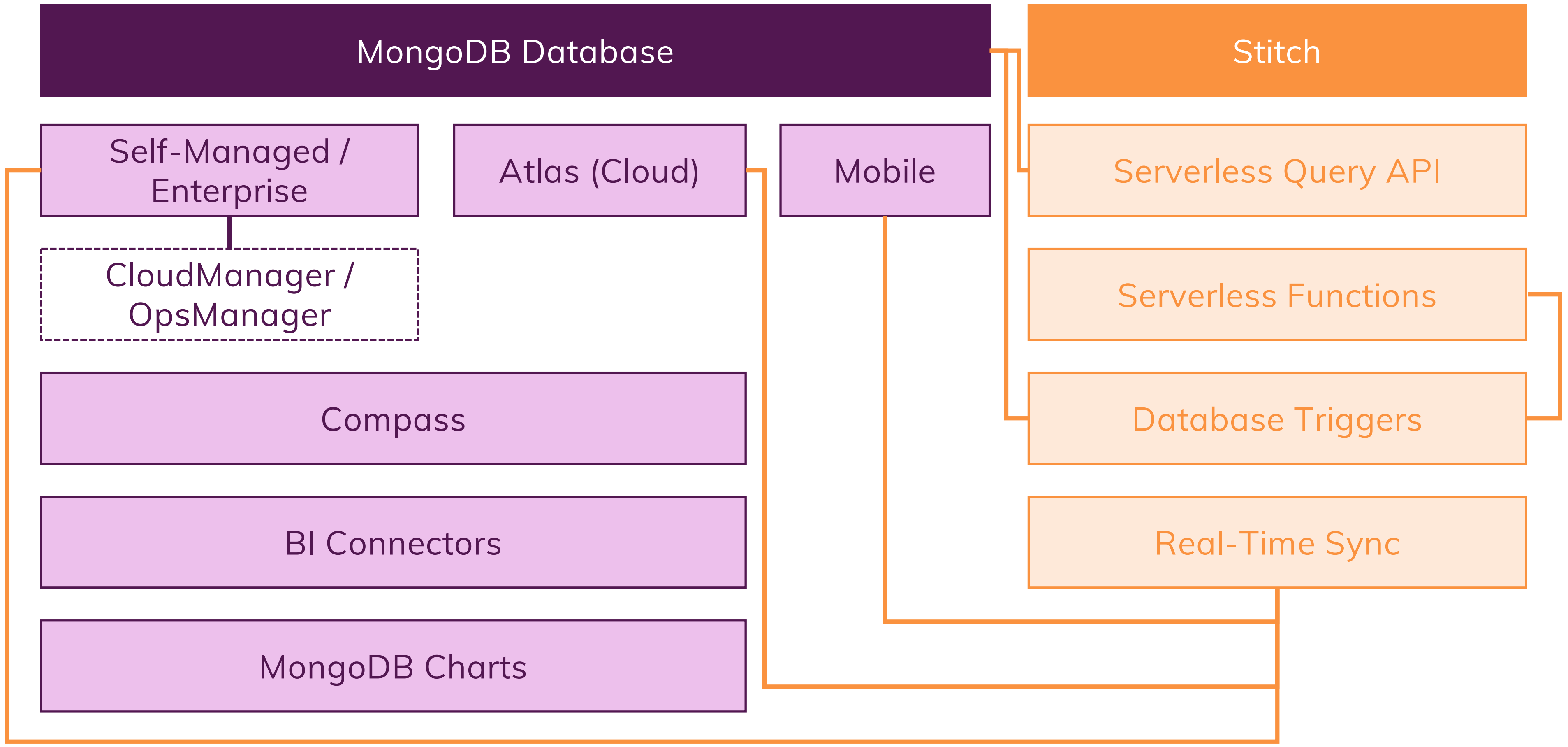
Relations

No / Few Relations!

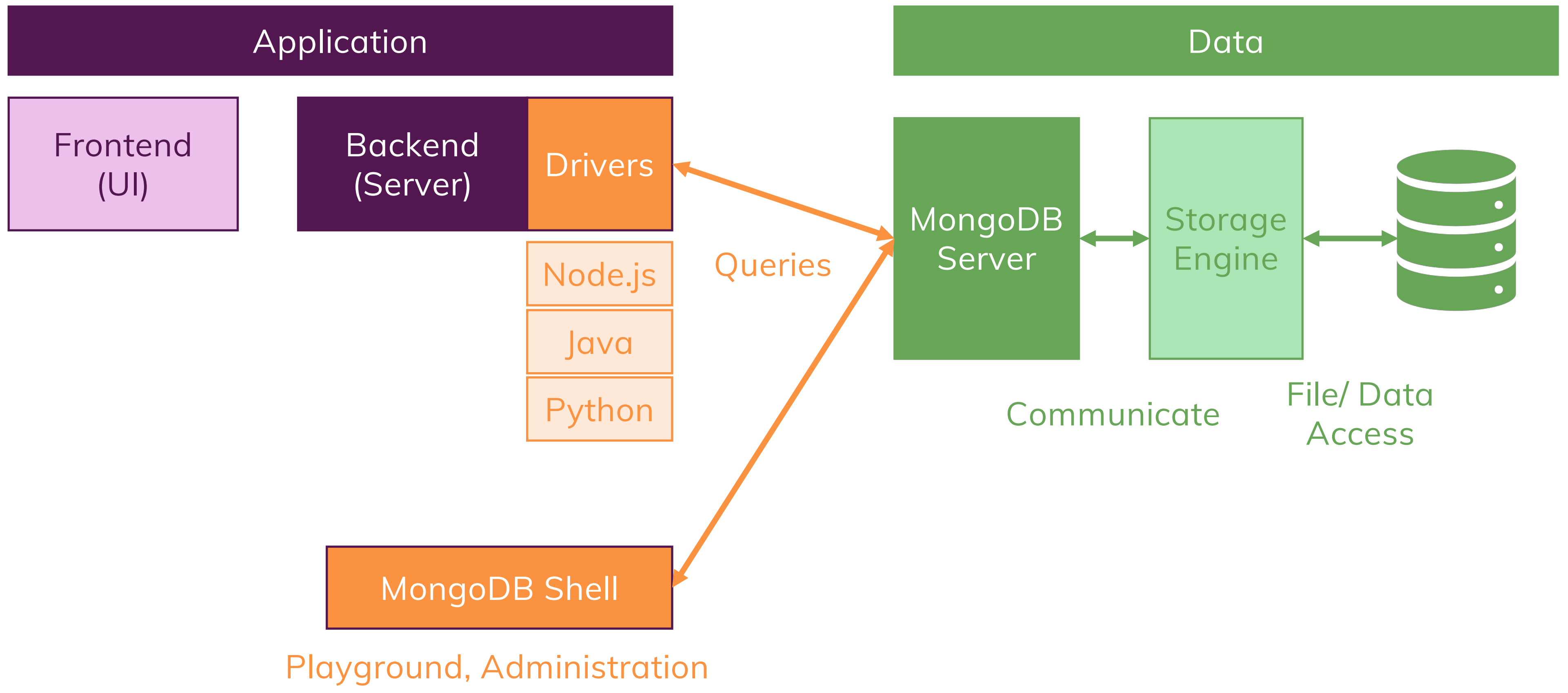
Relational Data needs to be merged manually!

Kind of...

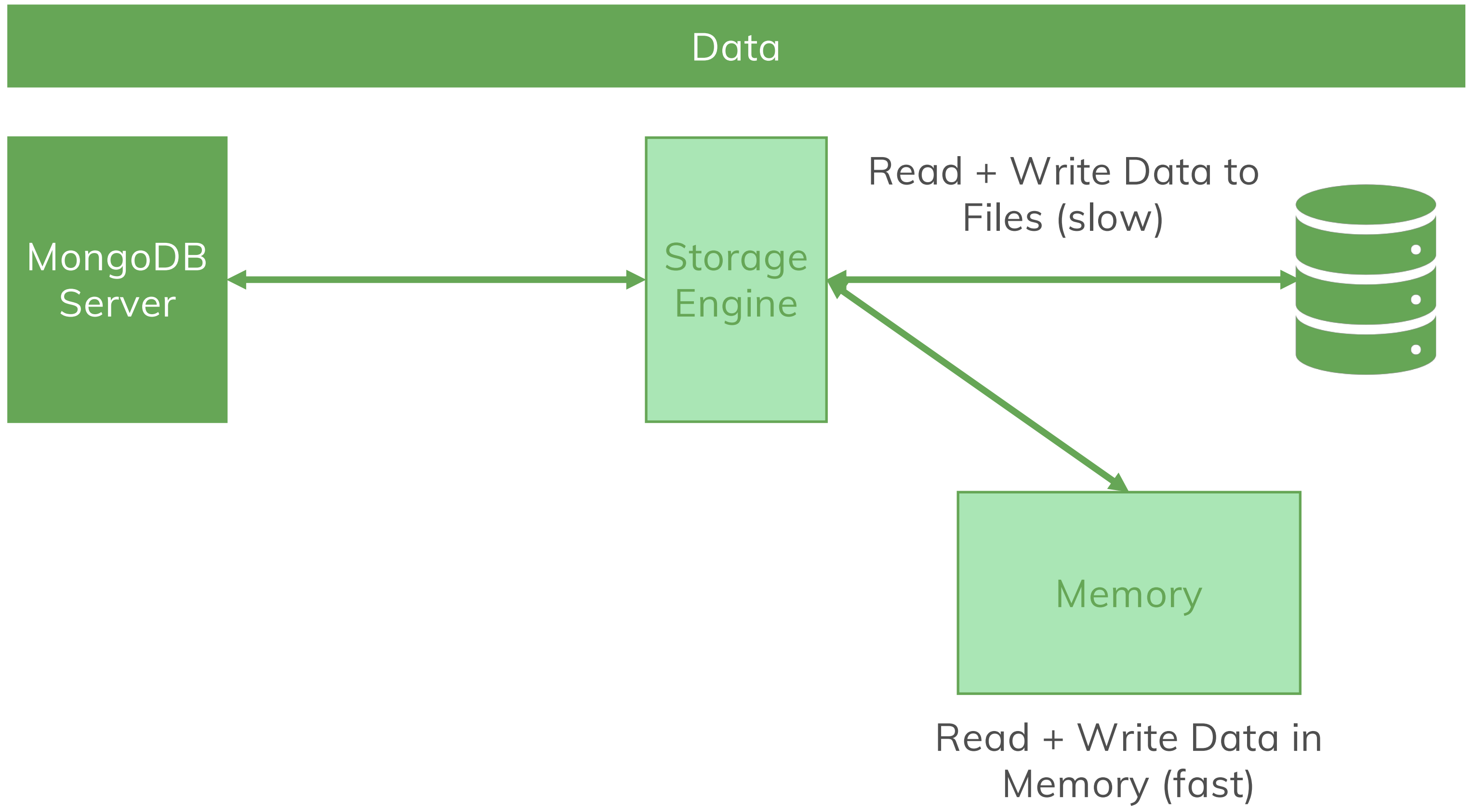
MongoDB Ecosystem



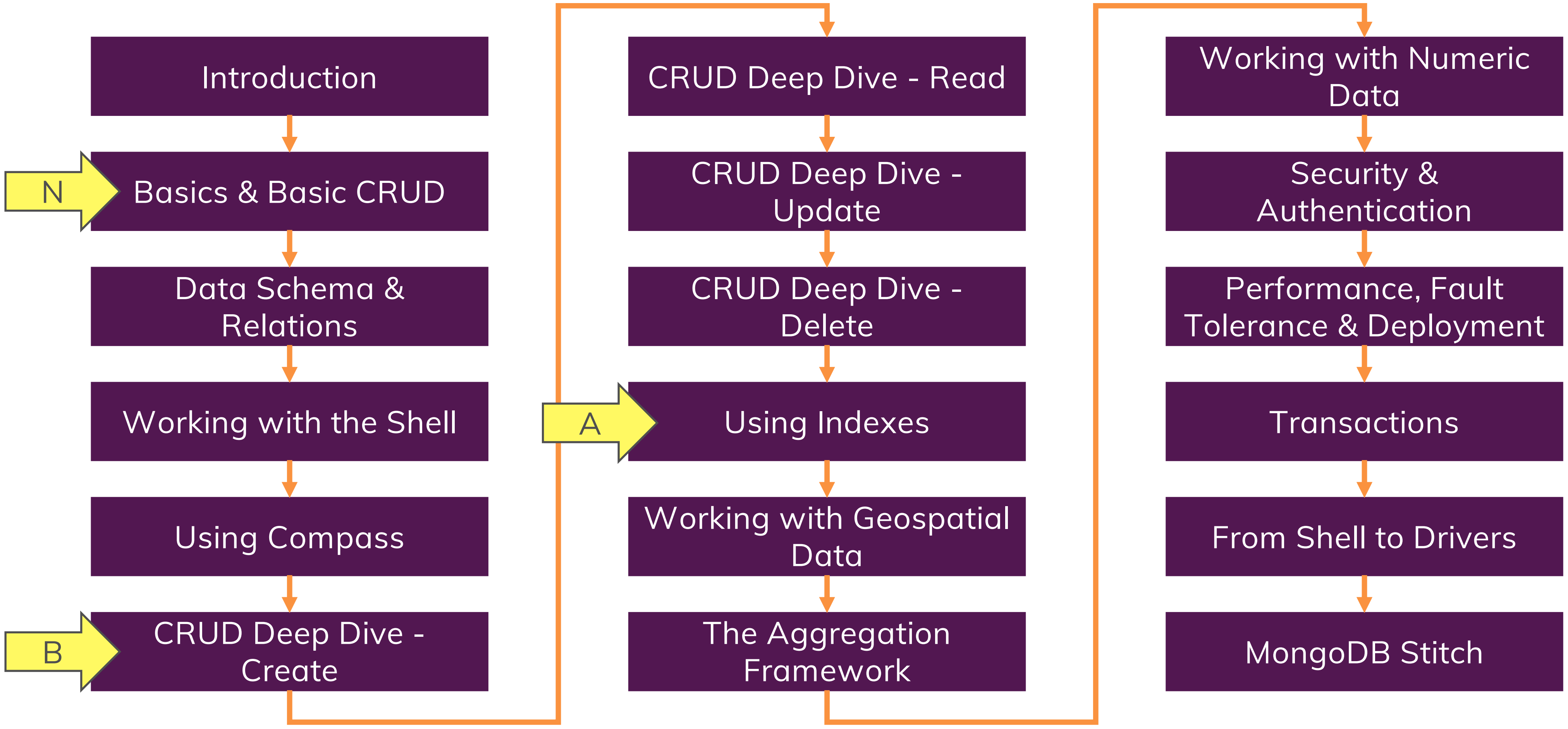
Working with MongoDB



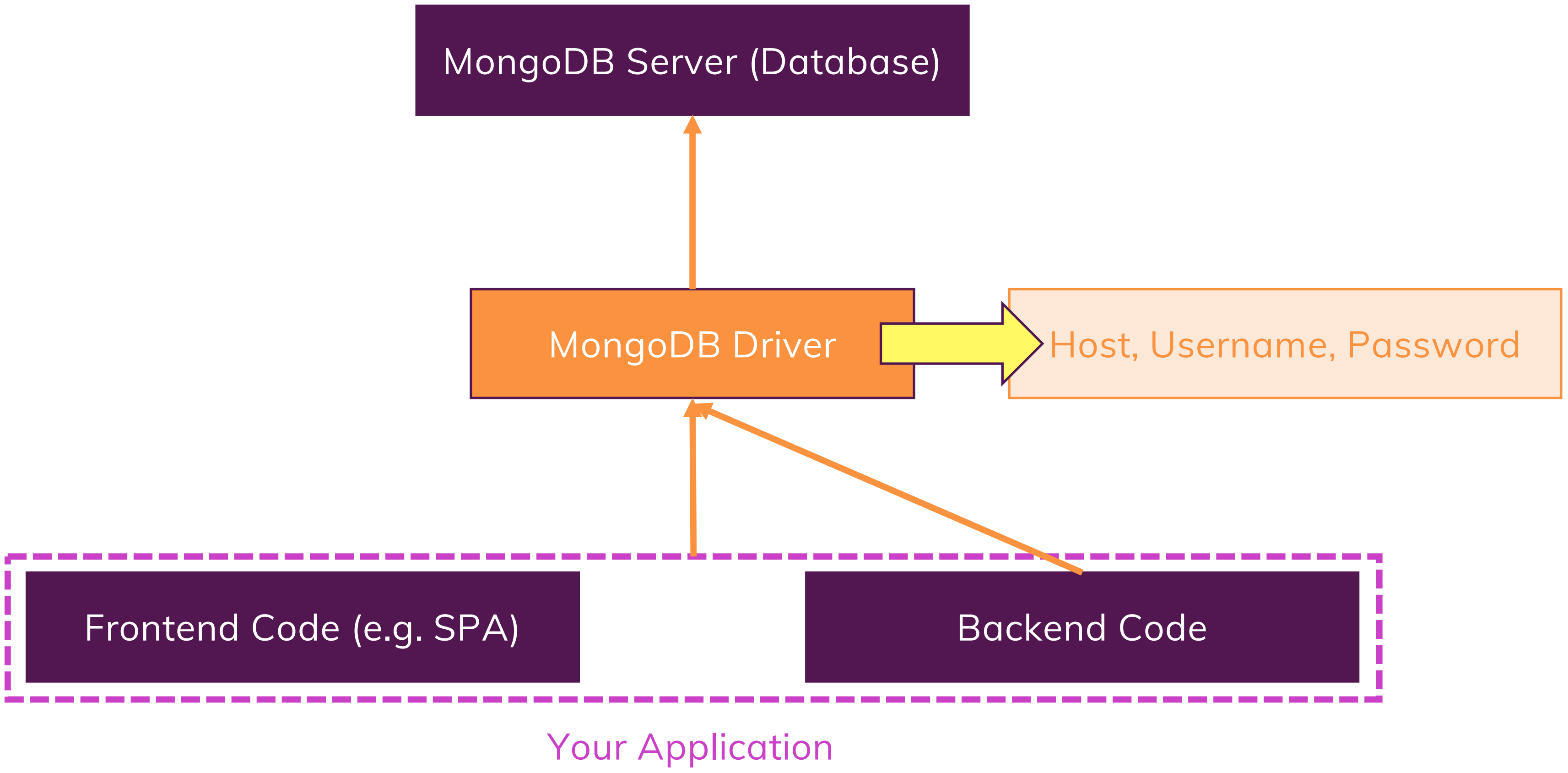
A Closer Look


Data
MongoDB Server
Storage Engine
Read + Write Data to Files (slow)
Memory
Read + Write Data in Memory (fast)

Outline


Introduction
N
Basics & Basic CRUD
Data Schema & Relations
Working with the Shell
Using Compass
B
CRUD Deep Dive - Create
CRUD Deep Dive - Read
CRUD Deep Dive - Update
CRUD Deep Dive - Delete
A
Using Indexes
Working with Geospatial Data
The Aggregation Framework
Working with Numeric Data
Security & Authentication
Performance, Fault Tolerance & Deployment
Transactions
From Shell to Drivers
MongoDB Stitch

Using MongoDB Drivers



How To Get The Most Out Of The Course

Watch the Videos	At your Pace!
Pause & Code Along	Active > Passive
Do the Assignments	Did you really understand it?
Dive into the Official Docs + Google	Learn to Solve Problems
Ask + Answer in the Q&A Section	Answering makes you a Pro!

Document & CRUD Basics

Working with the Database

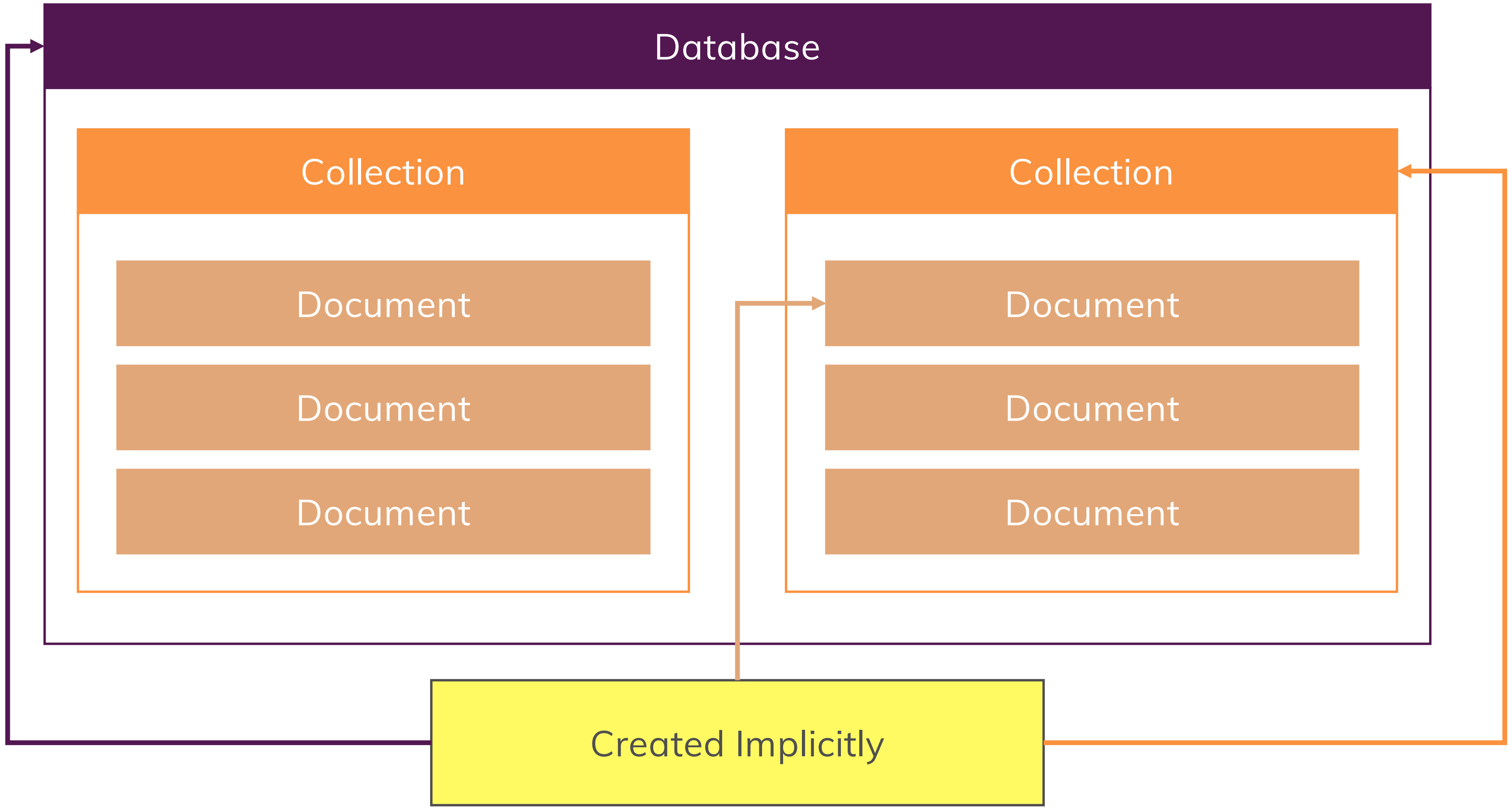
What's Inside This Module?

Basics about Collections & Documents

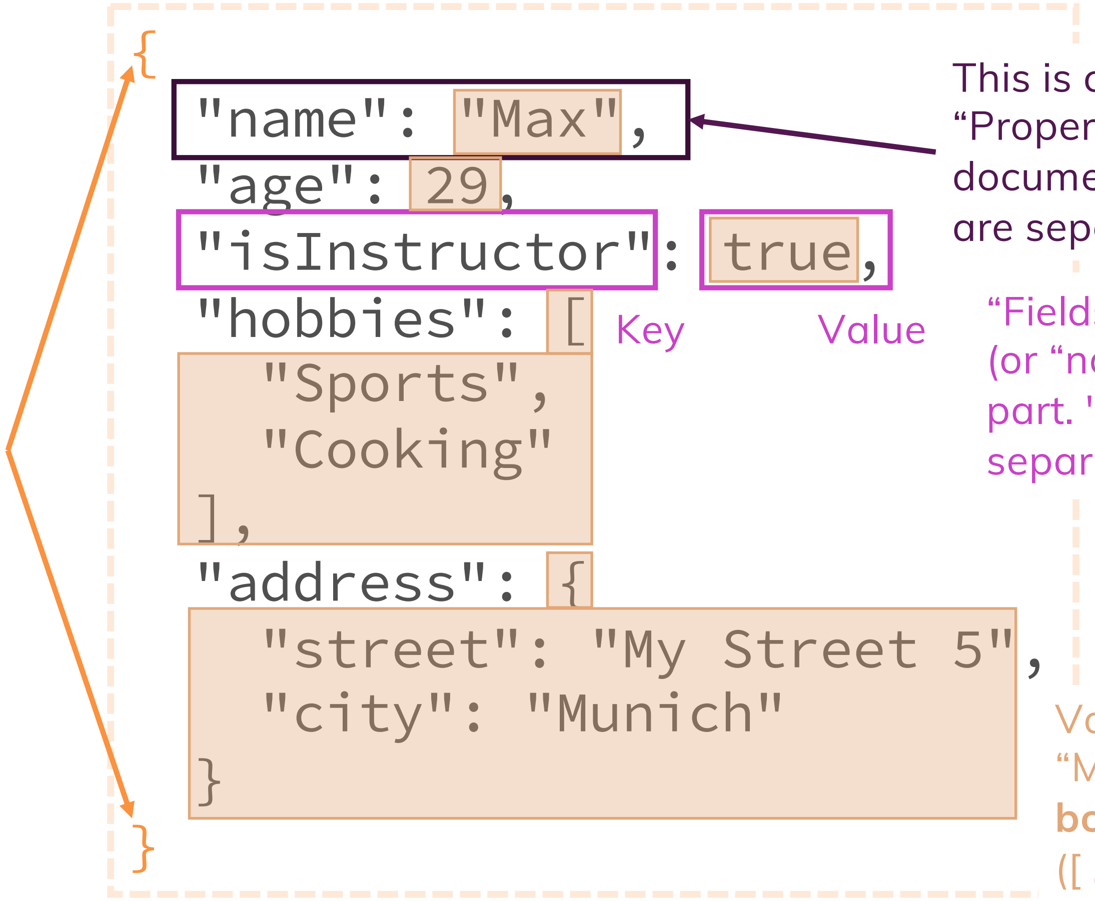
Basic Data Types

Performing CRUD Operations

Databases, Collections, Documents



JSON



```
{
  "name": "Max",
  "age": 29,
  "isInstructor": true,
  "hobbies": [
    "Sports",
    "Cooking"
  ],
  "address": {
    "street": "My Street 5",
    "city": "Munich"
  }
}
```

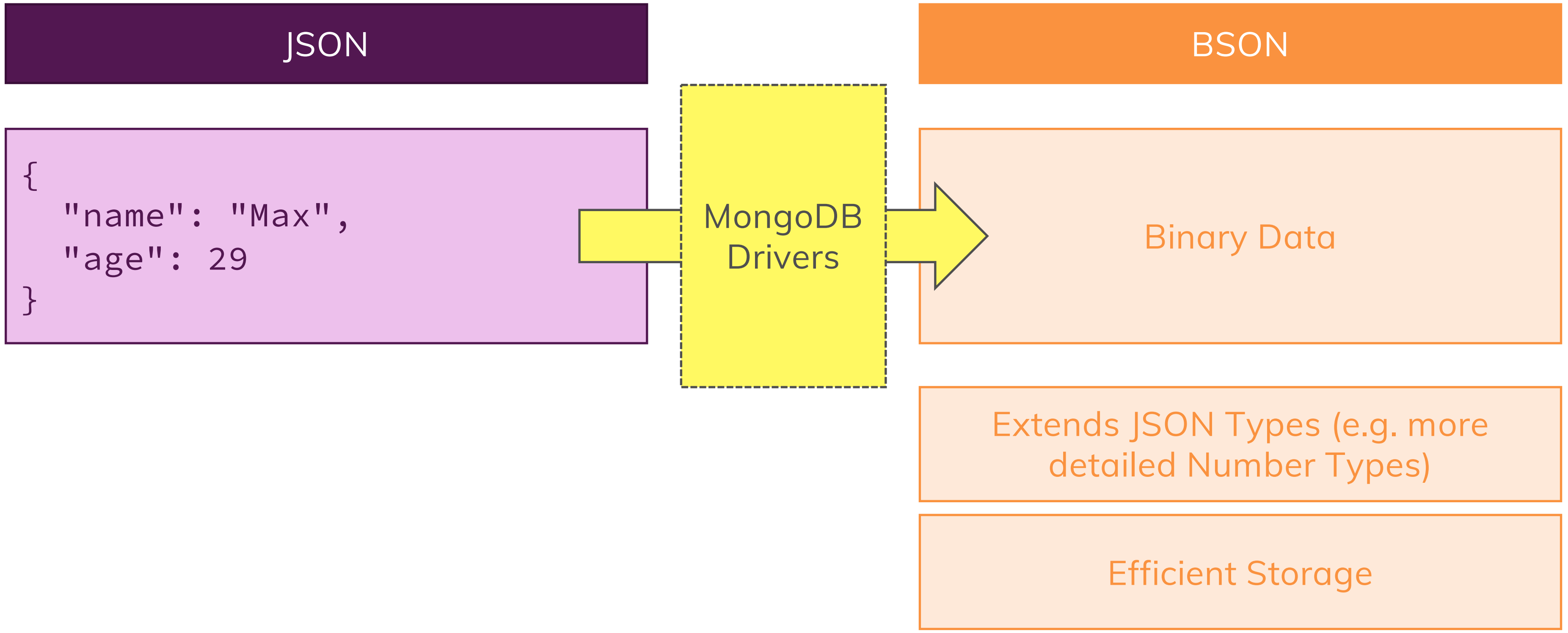
Surrounding curly braces delimit the JSON document

This is called a “**Field**” or “Property” of the JSON document. Multiple Fields are separated by commas

“Fields” consist of a “**Key**” (or “name”) and “**Value**” part. "Key and Value are separated by a colon.

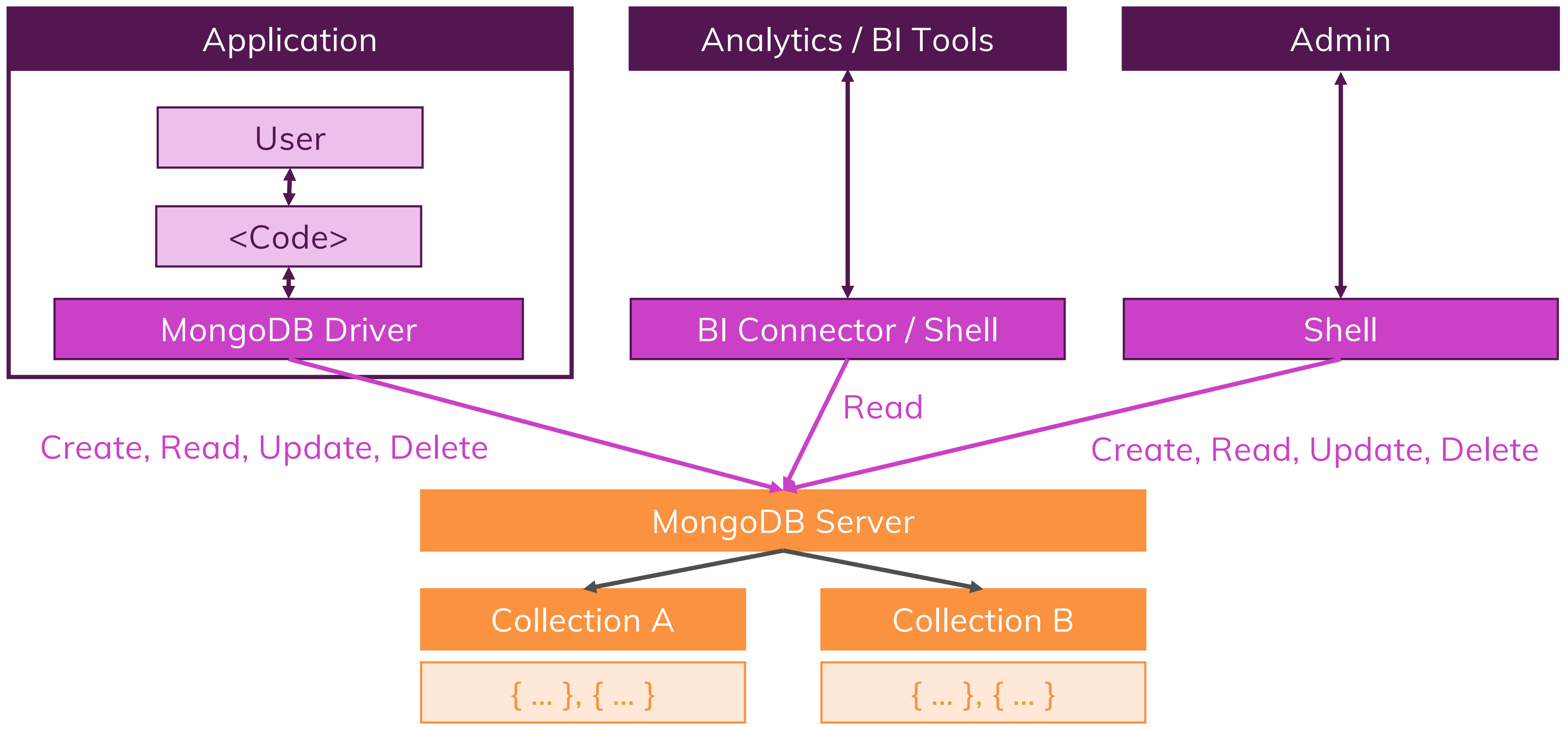
Values can be **strings** (e.g. “Max”), **numbers** (e.g. 29), **booleans** (e.g. true), **arrays** ([…]) and other **documents** (also called objects; { … })

JSON vs BSON



```
{
  "name": "Max",
  "age": 29
}
```

CRUD Operations & MongoDB



CRUD Operations

Create

```
insertOne(data, options)
insertMany(data, options)
```

Update

```
updateOne(filter, data, options)
updateMany(filter, data, options)
replaceOne(filter, data, options)
```

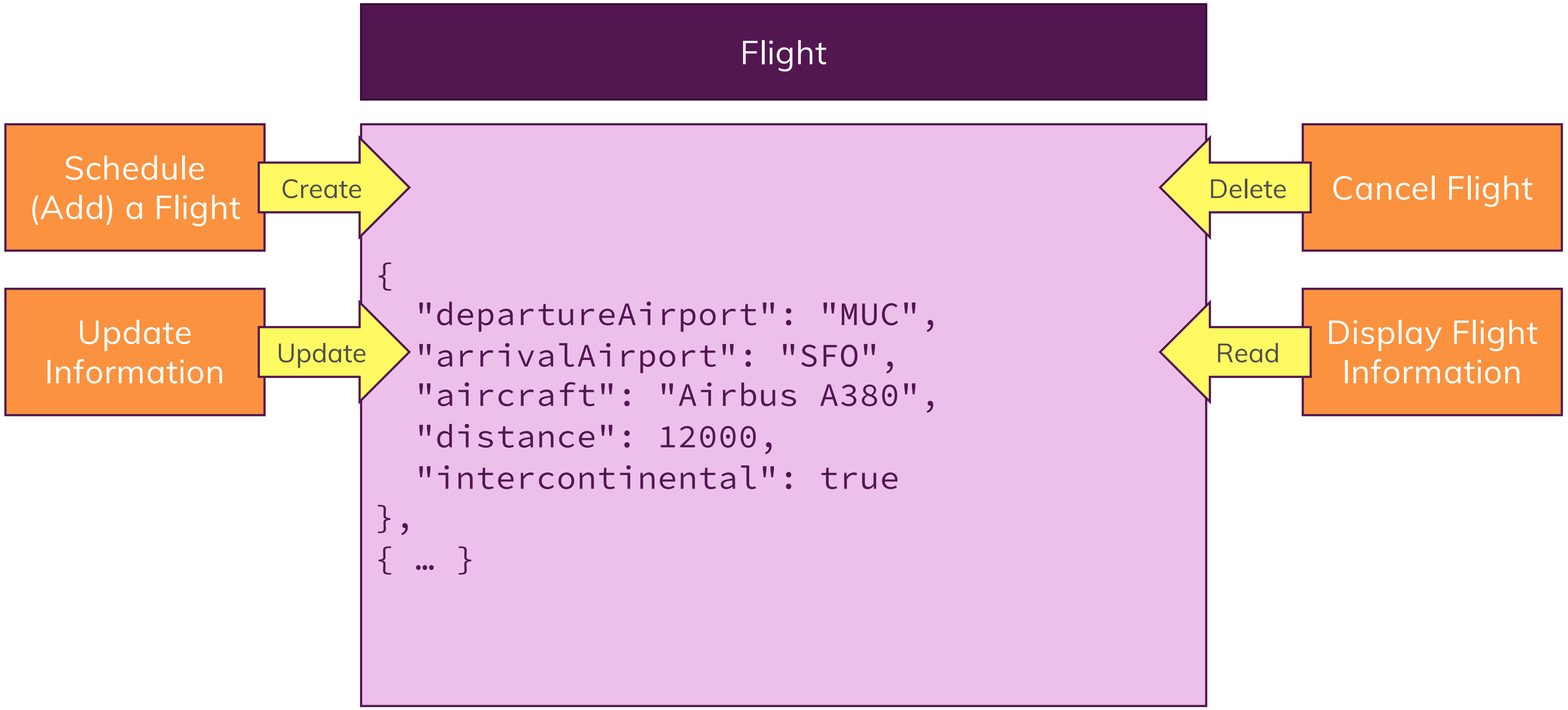
Read

```
find(filter, options)
findOne(filter, options)
```

Delete

```
deleteOne(filter, options)
deleteMany(filter, options)
```

Example #1: Flight Data



```
{
  "departureAirport": "MUC",
  "arrivalAirport": "SFO",
  "aircraft": "Airbus A380",
  "distance": 12000,
  "intercontinental": true
},
{ … }
```

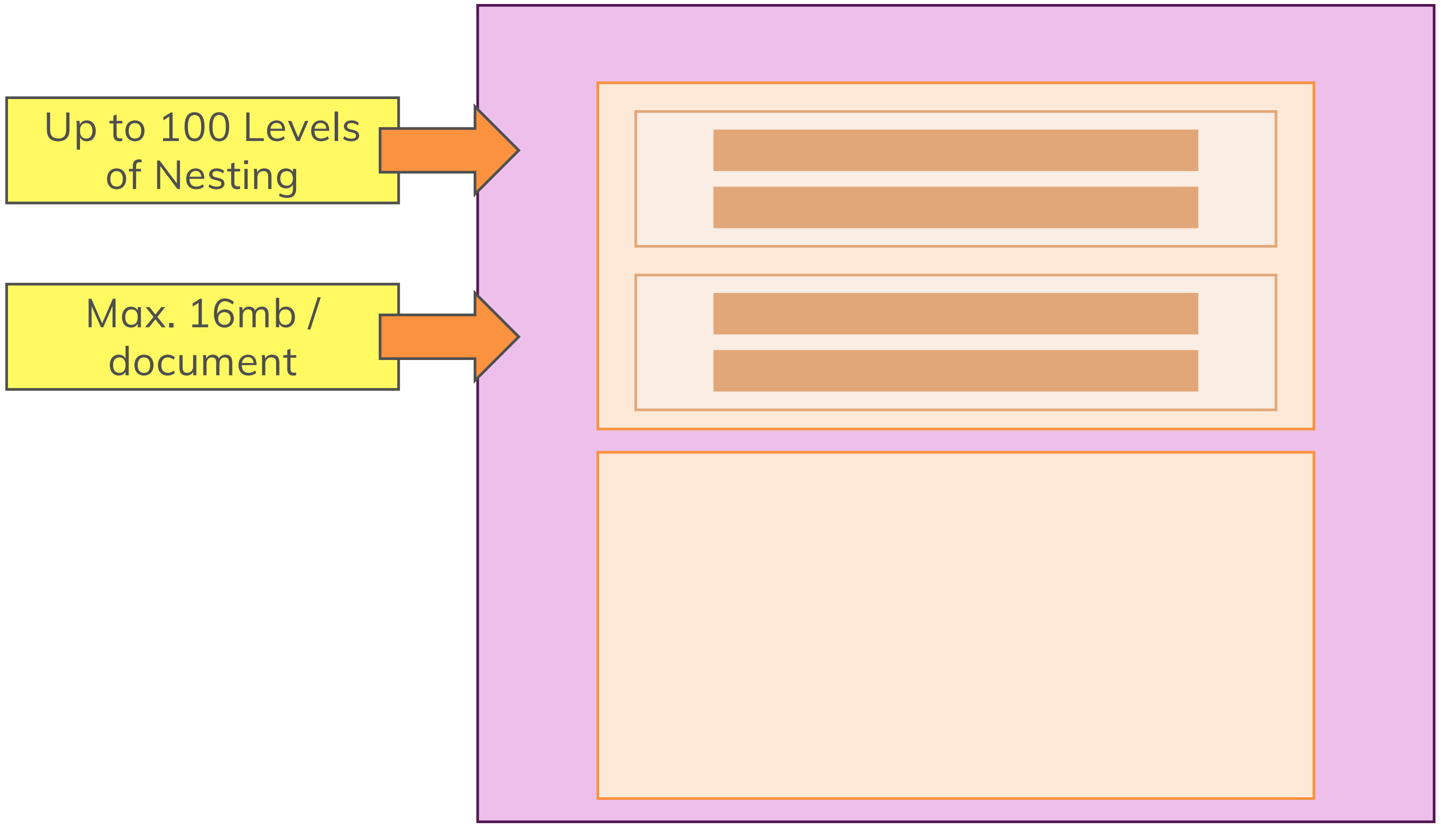
Unique IDs

You MUST have an _id

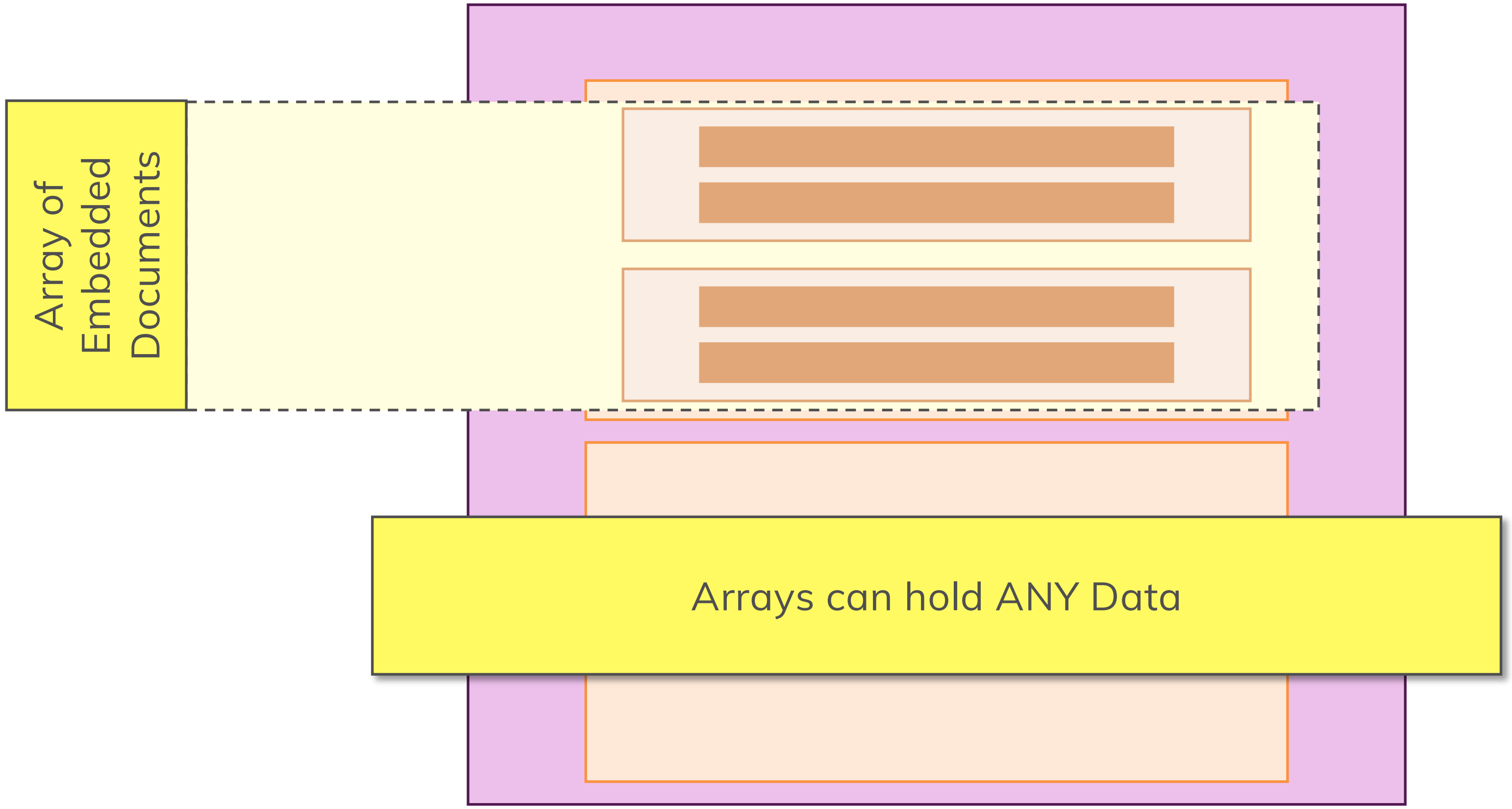
MongoDB creates an ObjectId() for you

You can set any other Value

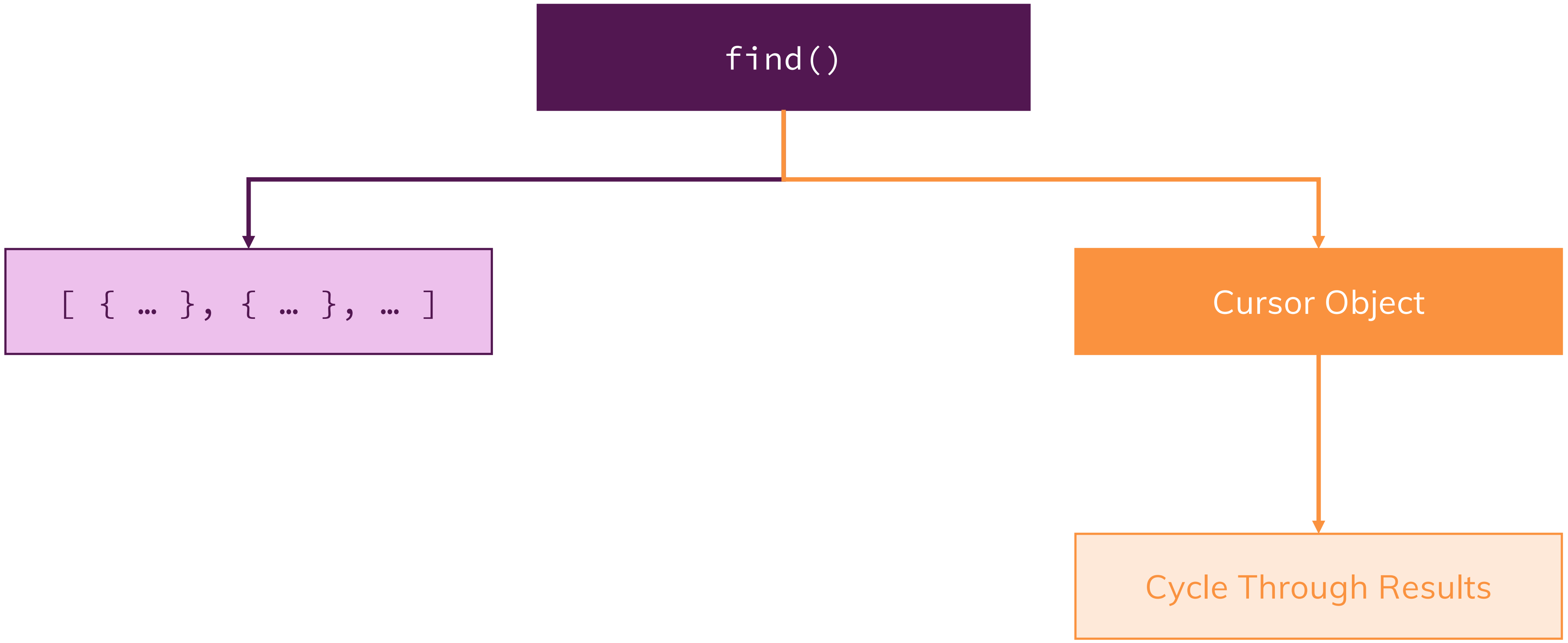
Embedded Documents



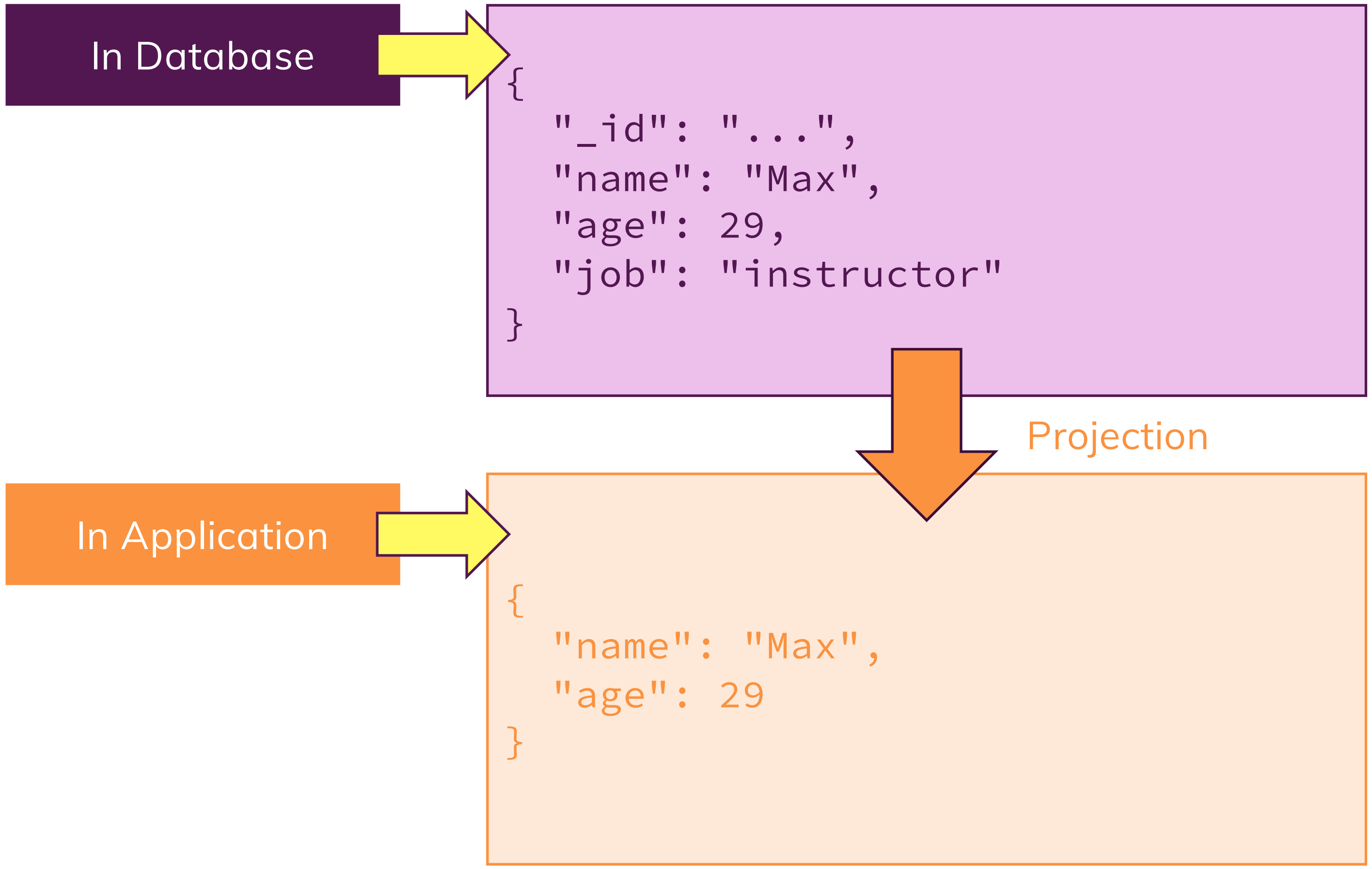
Arrays



Cursors



Projection



```
{
  "_id": "...",
  "name": "Max",
  "age": 29,
  "job": "instructor"
}
```

```
{
  "name": "Max",
  "age": 29
}
```

update() vs updateOne() vs updateMany()

update()	updateOne()	updateMany()
Overwrite by default	Error without $set (or other update operators)	Error without $set (or other update operators)
Use $set to patch values	Use $set to patch values	Use $set to patch values
Update all identified elements	Update first identified element	Update all identified elements

Use these!

Example #2: Patient Data

Patient

```
{
  "firstName": "Max",
  "lastName": "Schwarzmueller",
  "age": 29,
  "history": [
    { "disease": "cold", "treatment": … },
    { … }
  ]
}
```

Tasks

1	Insert 3 patient records with at least 1 history entry per patient
2	Update patient data of 1 patient with new age, name and history entry
3	Find all patients who are older than 30 (or a value of your choice)
4	Delete all patients who got a cold as a disease

Module Summary

Databases, Collections, Documents

- A Database holds multiple Collections where each Collection can then hold multiple Documents
- Databases and Collections are created "lazily" (i.e. when a Document is inserted)
- A Document can't directly be inserted into a Database, you need to use a Collection!

CRUD Operations

- CRUD = Create, Read, Update, Delete
- MongoDB offers multiple CRUD operations for single-document and bulk actions (e.g. insertOne(), insertMany(), …)
- Some methods require an argument (e.g. insertOne()), others don't (e.g. find())
- find() returns a cursor, NOT a list of documents!
- Use filters to find specific documents

Document Structure

- Each document needs a unique ID (and gets one by default)
- You may have embedded documents and array fields

Retrieving Data

- Use filters and operators (e.g. $gt) to limit the number of documents you retrieve
- Use projection to limit the set of fields you retrieve

Data Schemas & Data Modelling

Storing your Data Correctly

What’s Inside This Module?

Understanding Document Schemas & Data Types

Modelling Relations

Schema Validation

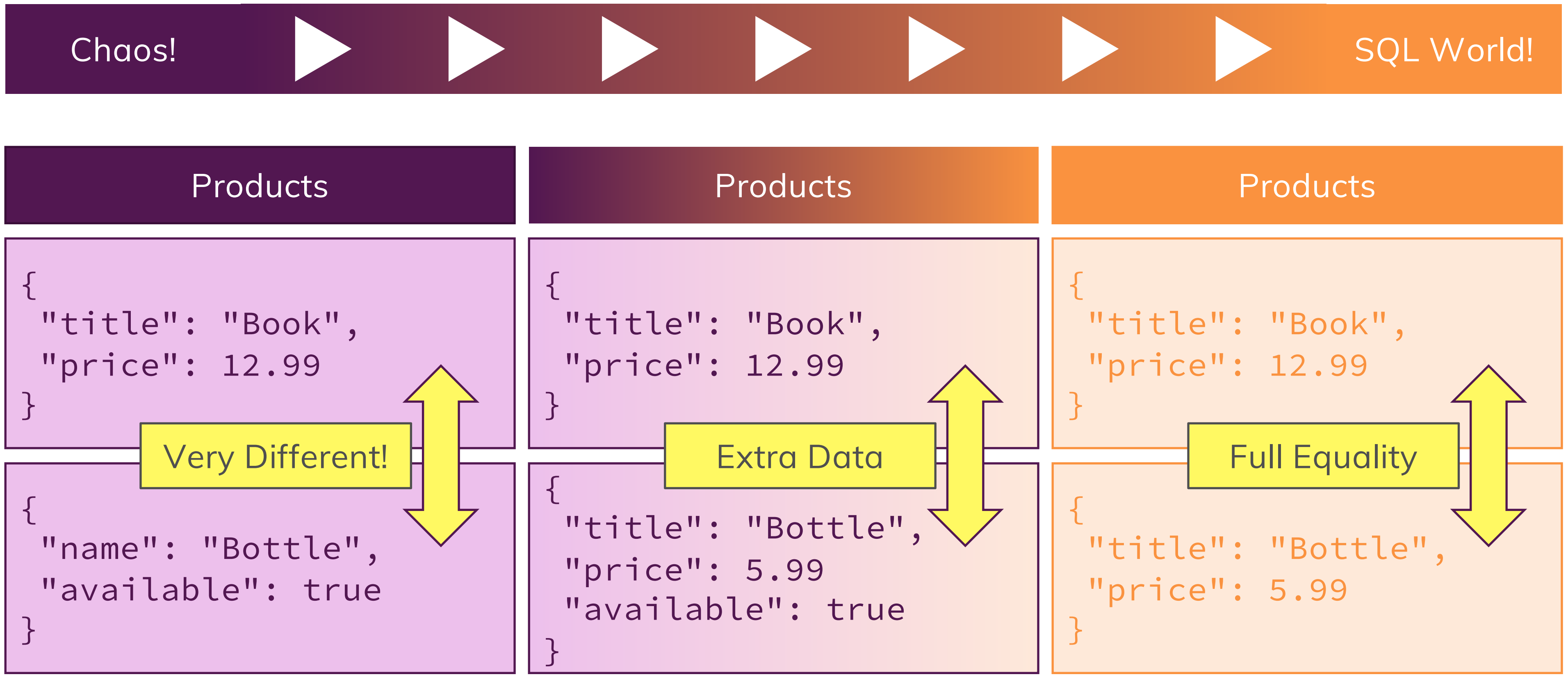
Schema-less Or Not?

Isn’t MongoDB all about having **NO** data Schemas?

MongoDB enforces no schemas! Documents don’t have to use the same schema inside of one collection

But that does not mean that you can’t use some kind of schema!

To Schema Or Not To Schema



```
{
 "title": "Book",
 "price": 12.99
}
```

```
{
 "name": "Bottle",
 "available": true
}
```

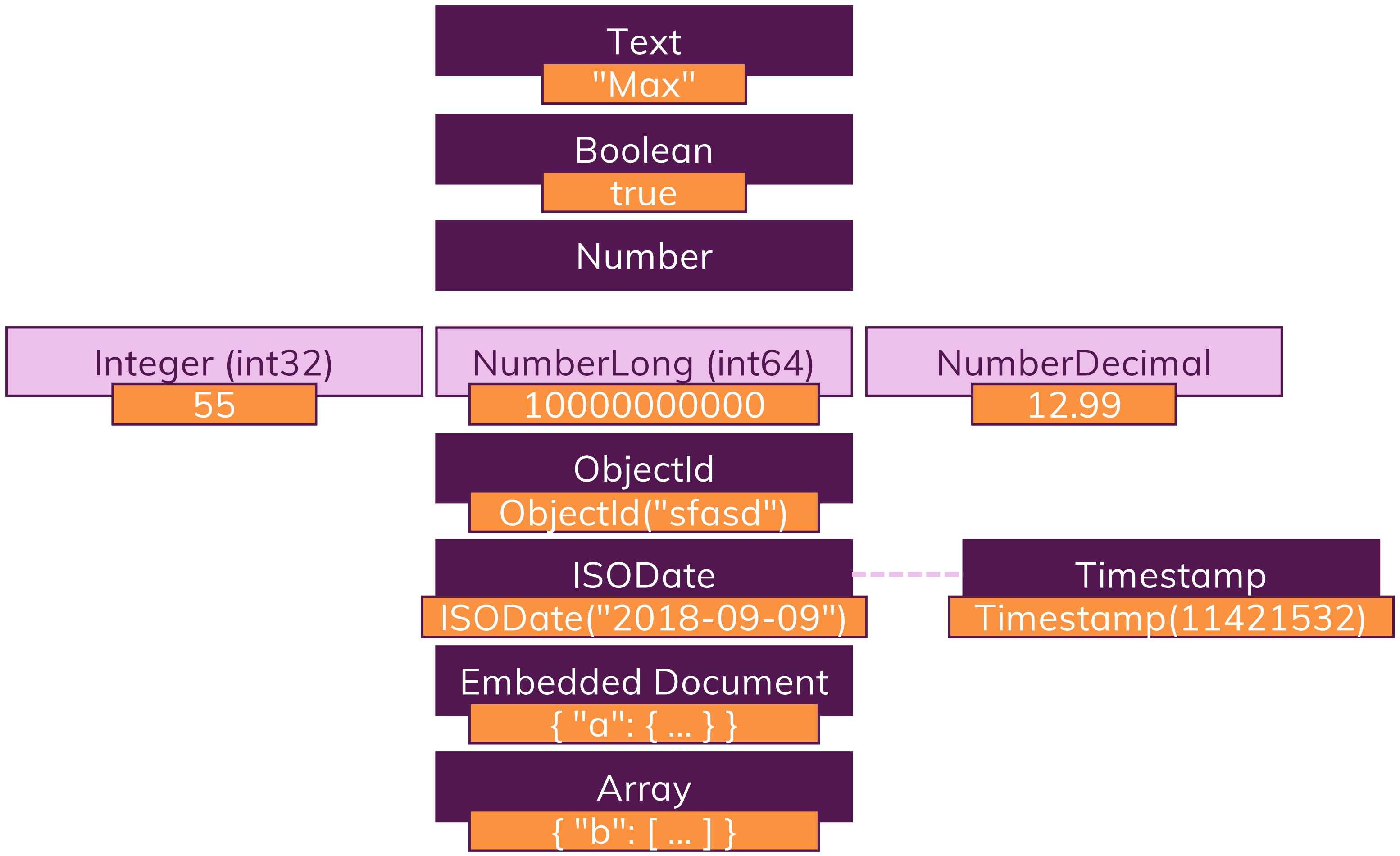
```
{
 "title": "Book",
 "price": 12.99
}
```

```
{
 "title": "Bottle",
 "price": 5.99
 "available": true
}
```

```
{
 "title": "Book",
 "price": 12.99
}
```

```
{
 "title": "Bottle",
 "price": 5.99
}
```

Data Types


Text
"Max"
Boolean
true
Number
Integer (int32)
55
NumberLong (int64)
10000000000
NumberDecimal
12.99
ObjectId
ObjectId("sfasd")
ISODate
ISODate("2018-09-09")
Timestamp
Timestamp(11421532)
Embedded Document
{ "a": { ... } }
Array
{ "b": [...] }

Data Schemas & Data Modelling

Which Data does my App need or generate?	User Information, Product Information, Orders, …	Defines the Fields you'll need (and how they relate)
Where do I need my Data?	Welcome Page, Products List Page, Orders Page	Defines your required collections + field groupings
Which kind of Data or Information do I want to display?	Welcome Page: Product Names; Products Page: …	Defines which queries you'll need
How often do I fetch my data?	For every page reload	Defines whether you should optimize for easy fetching
How often do I write or change my data?	Orders => Often Product Data => Rarely	Defines whether you should optimize for easy writing

Data Schemas & Data Modelling

Fetching a lot? → Goal: Store Data in the Format you need in your Frontend! Don’t run complex transformation queries!

Writing a lot? → Goal: Store Data such that you avoid duplicates and redundancy!

ACADEMIND

Relations - Options

Nested / Embedded Documents

Customers

```
{
  userName: 'max',
  age: 29,
  address: {
    street: 'Second Street',
    city: 'New York'
  }
}
```

References

omers

```
{
  user
  favBooks: [{…}, {…}]
}
```

Lots of data duplication!

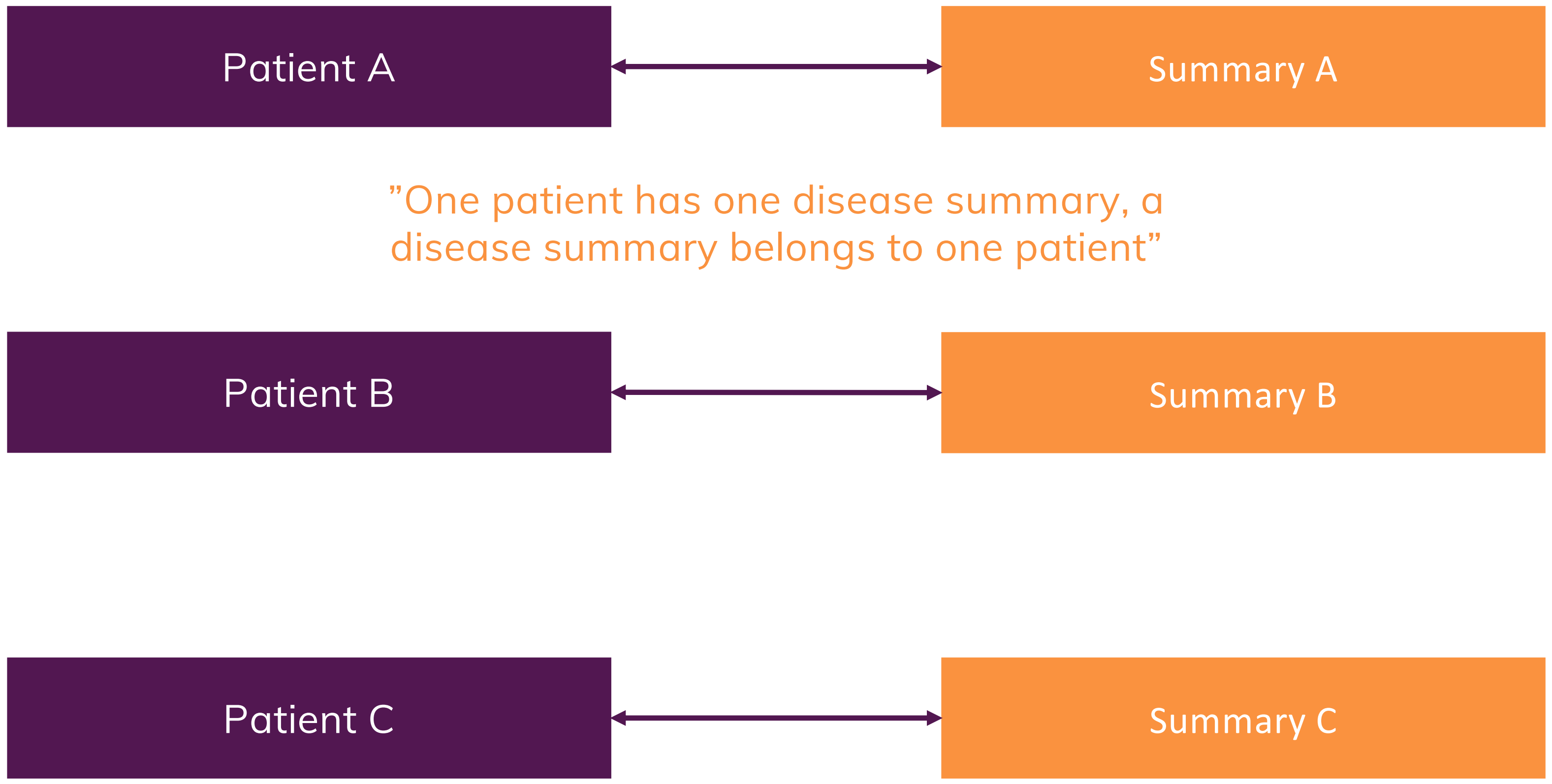
Customers

```
{
  userName: 'max',
  favBooks: ['id1', 'id2']
}
```

Books

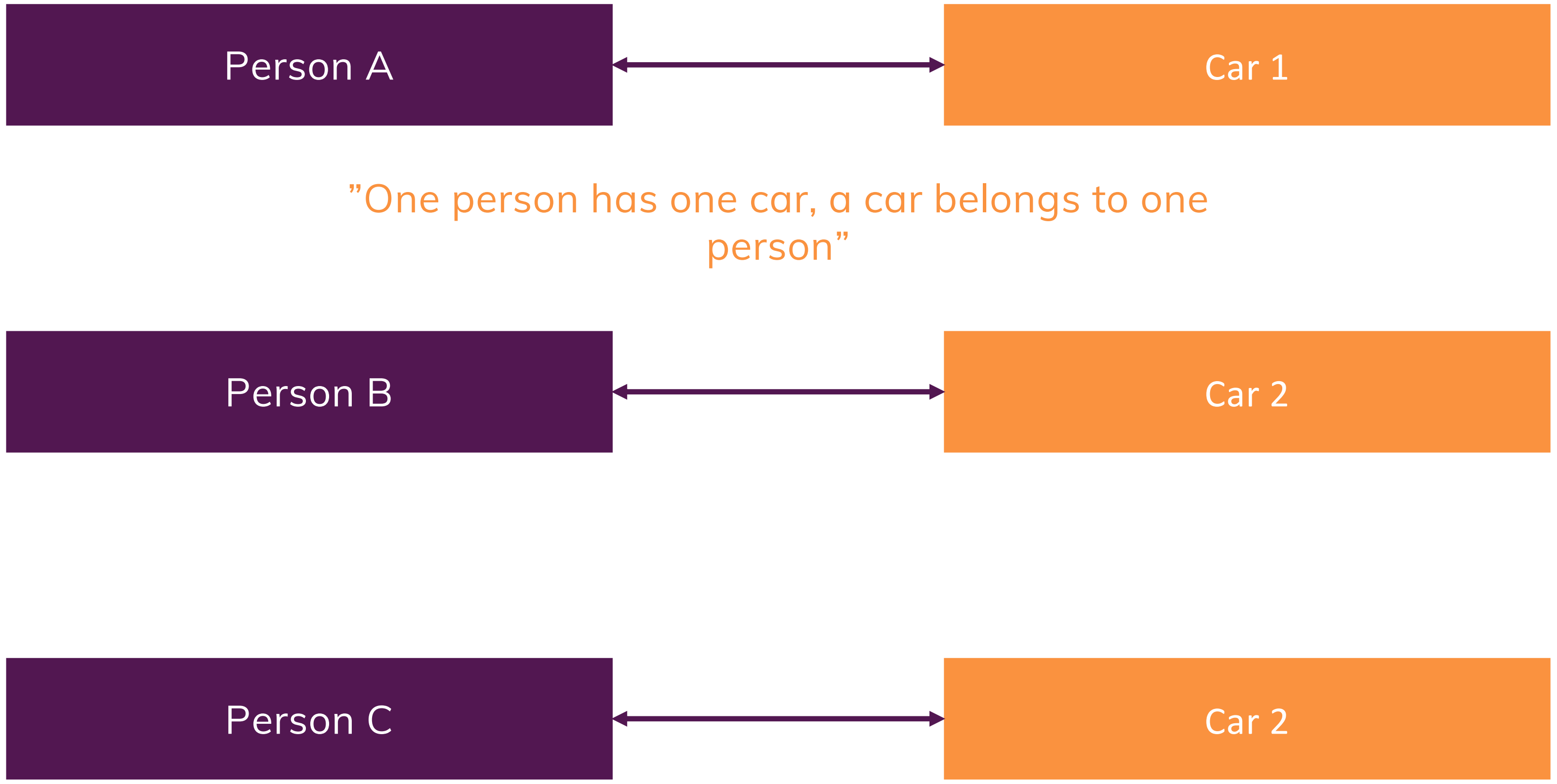
```
{
  _id: 'id1',
  name: 'Lord of the Rings 1'
}
```

Example #1 – Patient <-> Disease Summary



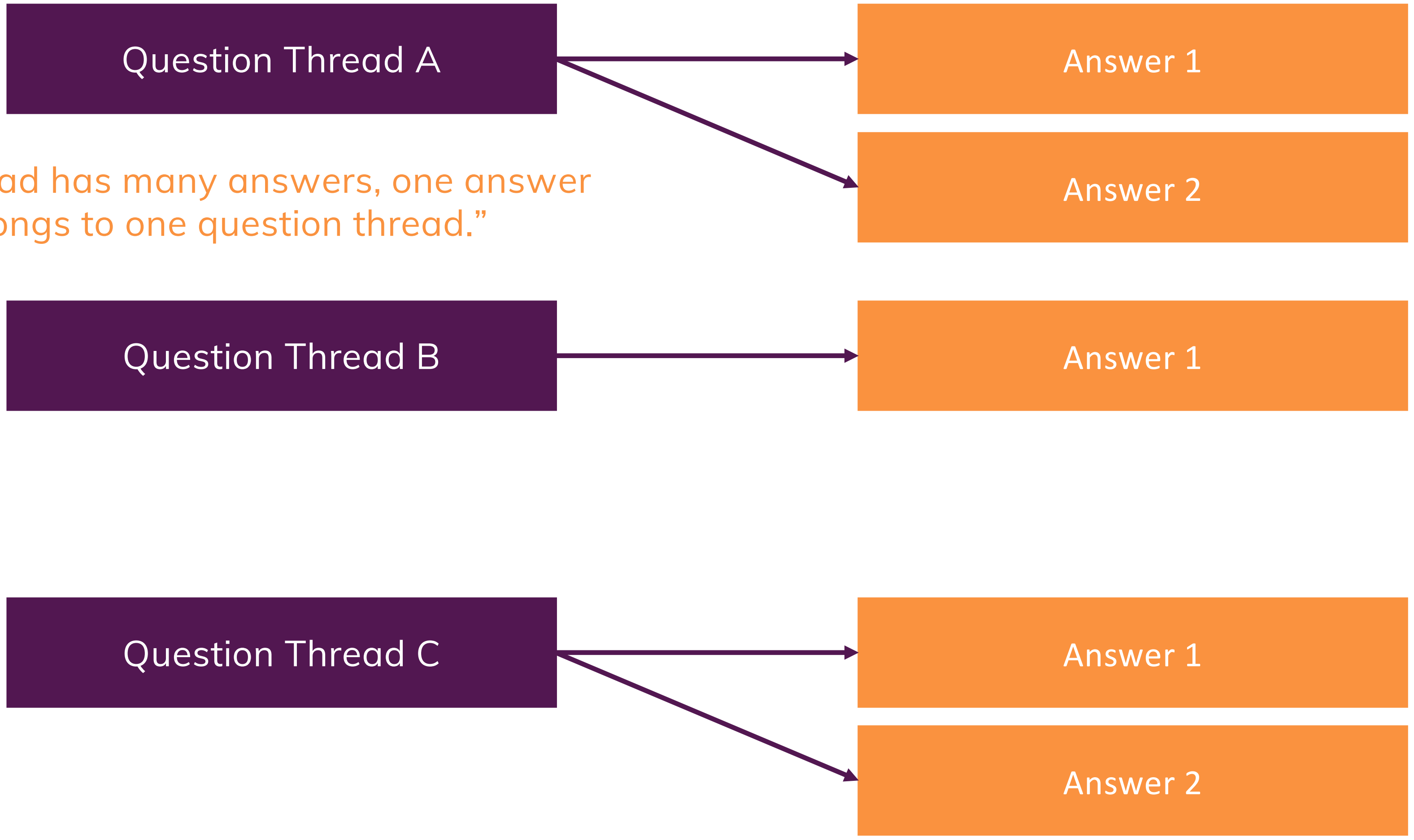
”One patient has one disease summary, a disease summary belongs to one patient”

Example #2 – Person <-> Car



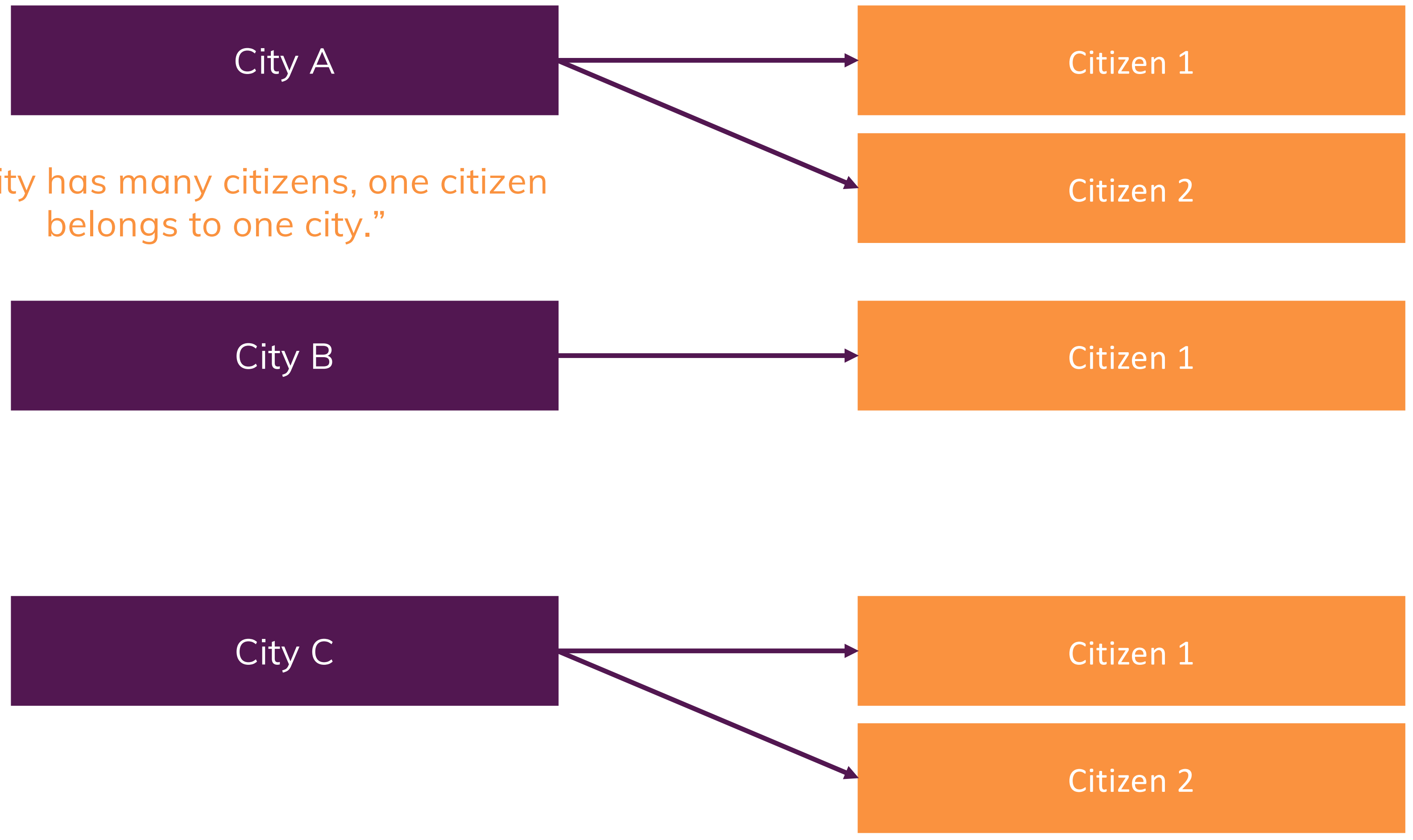
"One person has one car, a car belongs to one person"

Example #3 – Thread <-> Answers



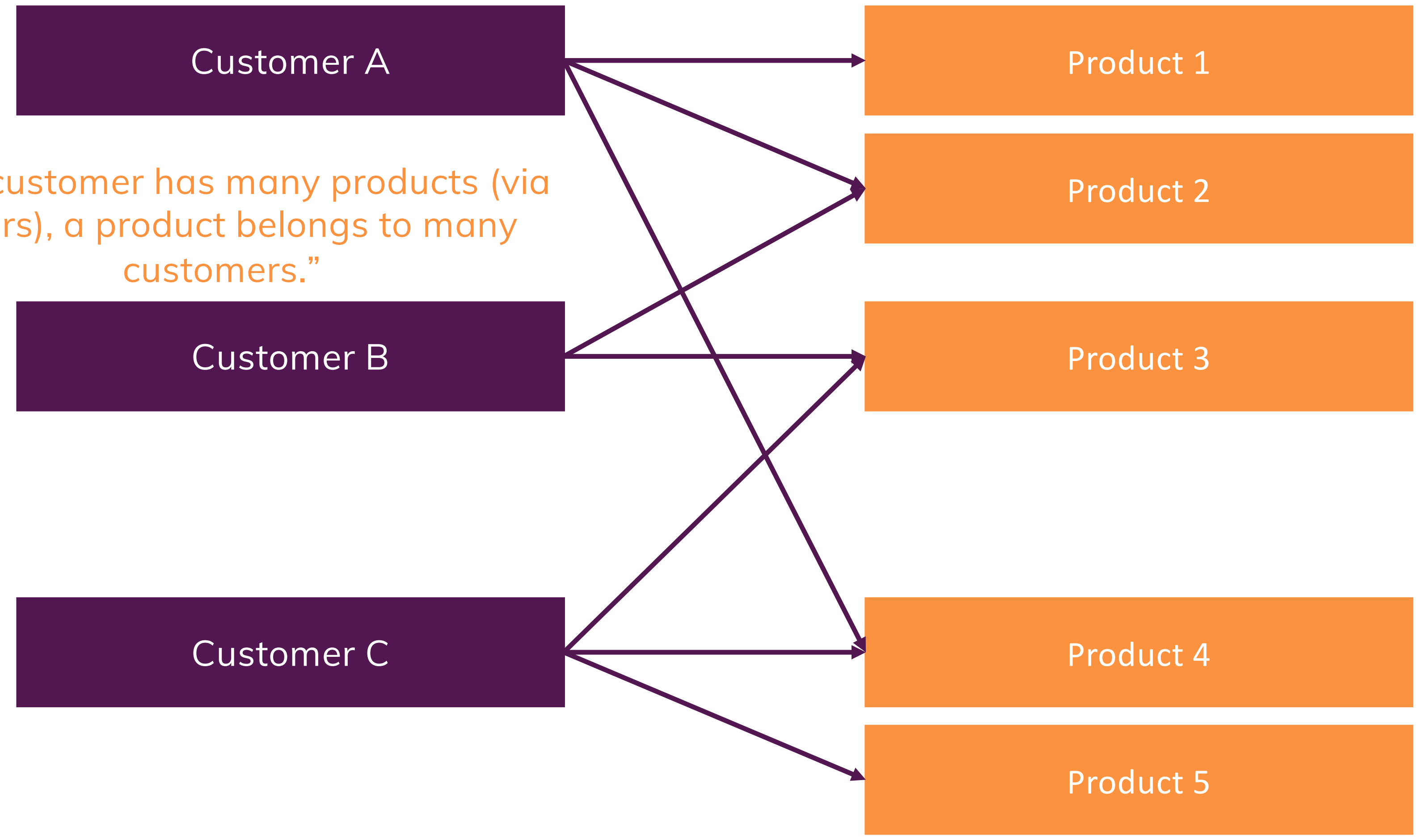
“One thread has many answers, one answer belongs to one question thread.”

Example #4 – City <-> Citizens



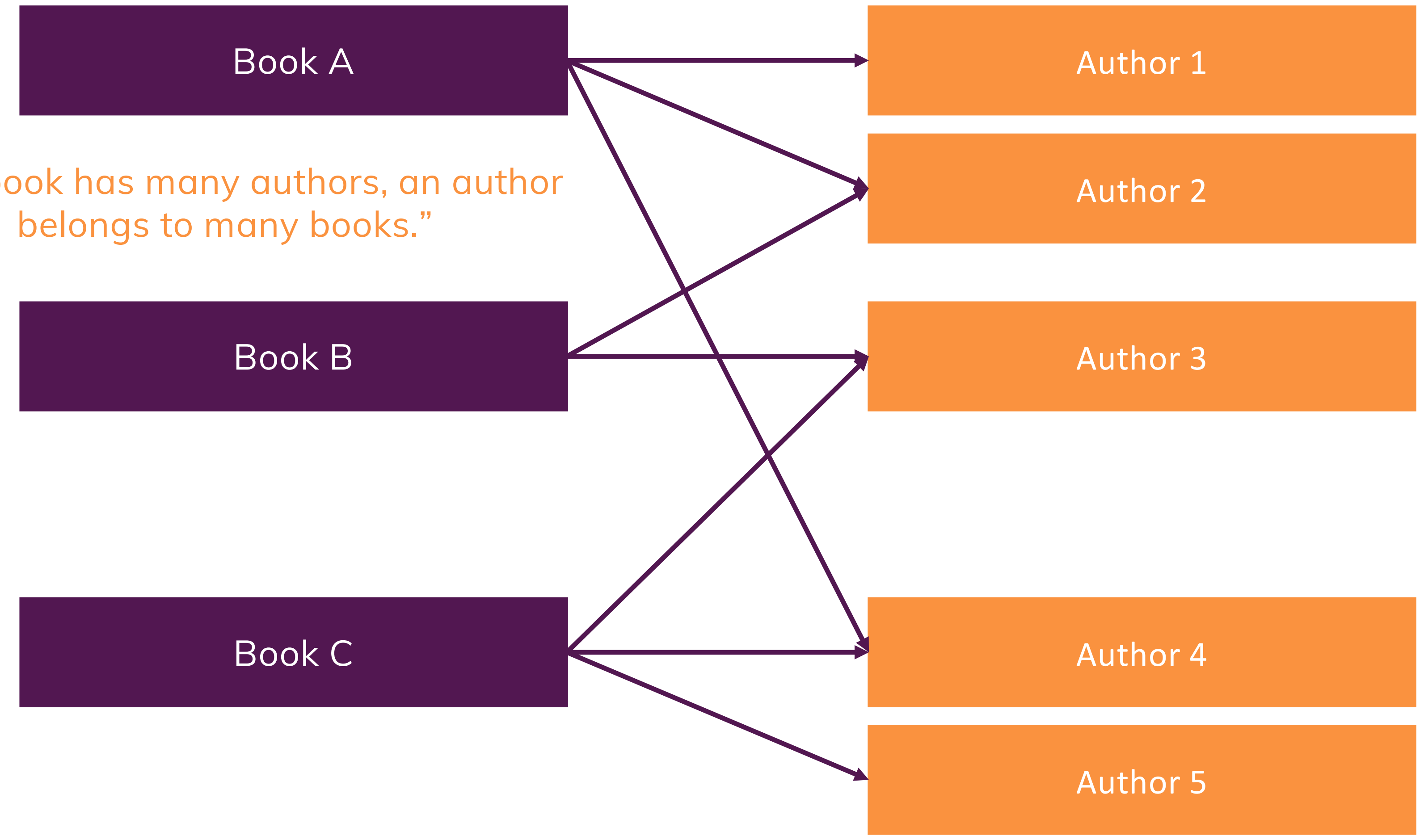
“One city has many citizens, one citizen belongs to one city.”

Example #5 – Customers <-> Products (Orders)



"One customer has many products (via orders), a product belongs to many customers."

Example #6 – Books <-> Authors

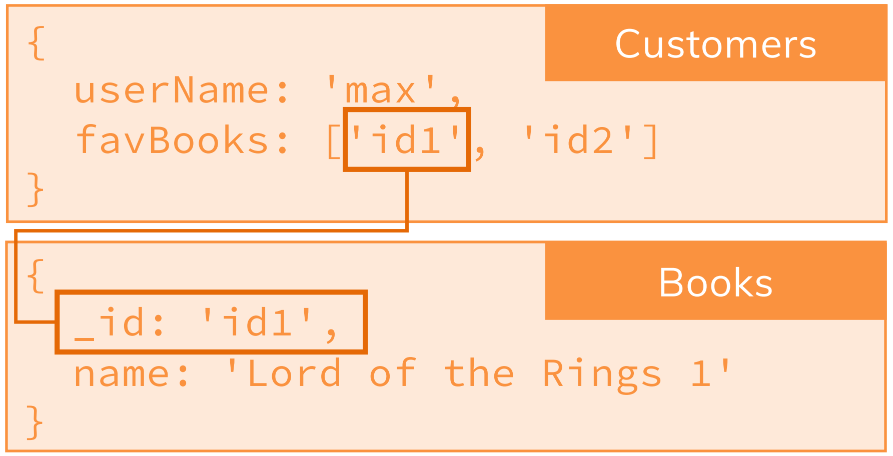


"One book has many authors, an author belongs to many books."

Relations - Options

Nested / Embedded Documents	References
Group data together logically	Split data across collections
Great for data that belongs together and is not really overlapping with other data	Great for related but shared data as well as for data which is used in relations and standalone
Avoid super-deep nesting (100+ levels) or extremely long arrays (16mb size limit per document)	Allows you to overcome nesting and size limits (by creating new documents)

Joining with $lookup

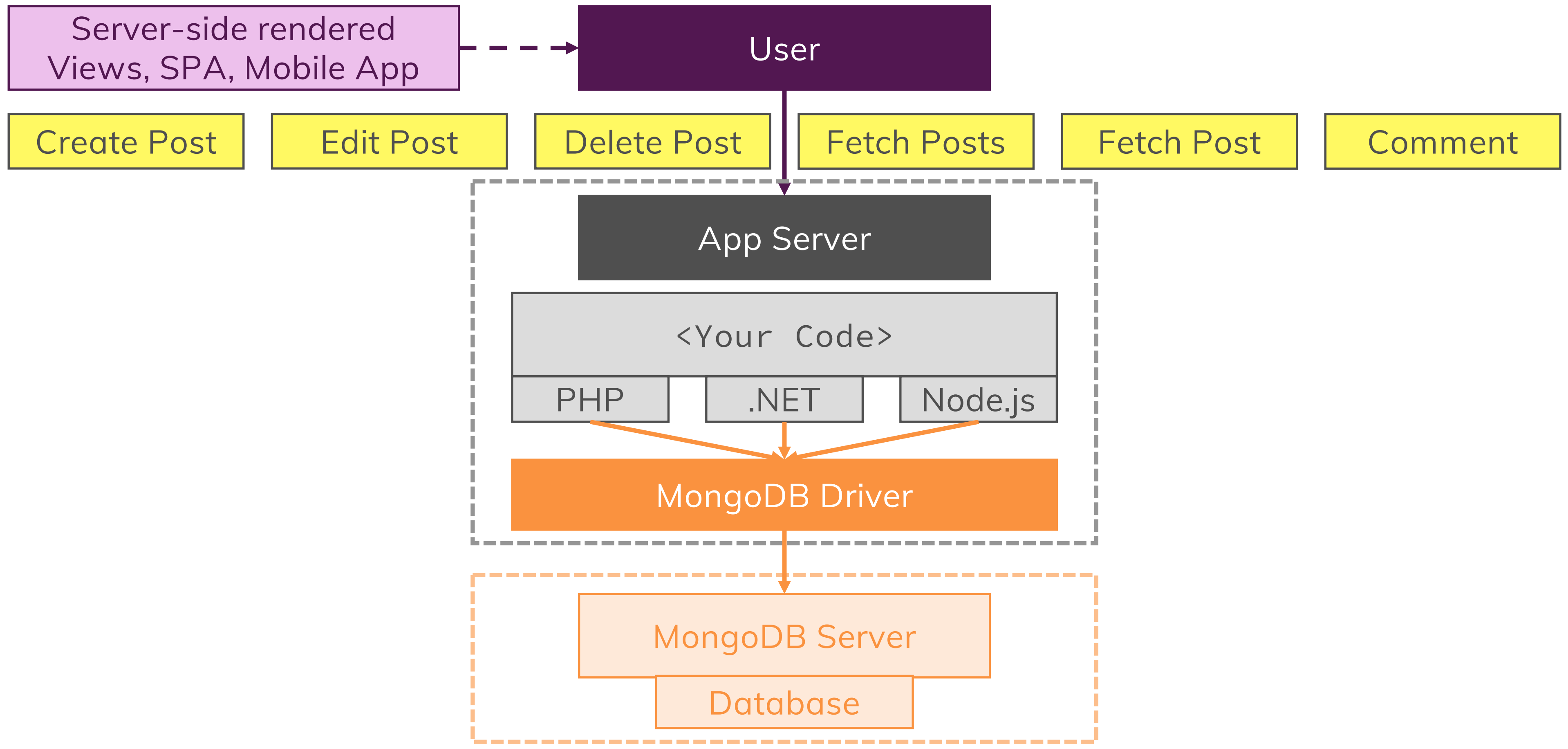


```
{
  userName: 'max',
  favBooks: ['id1', 'id2']
}
```

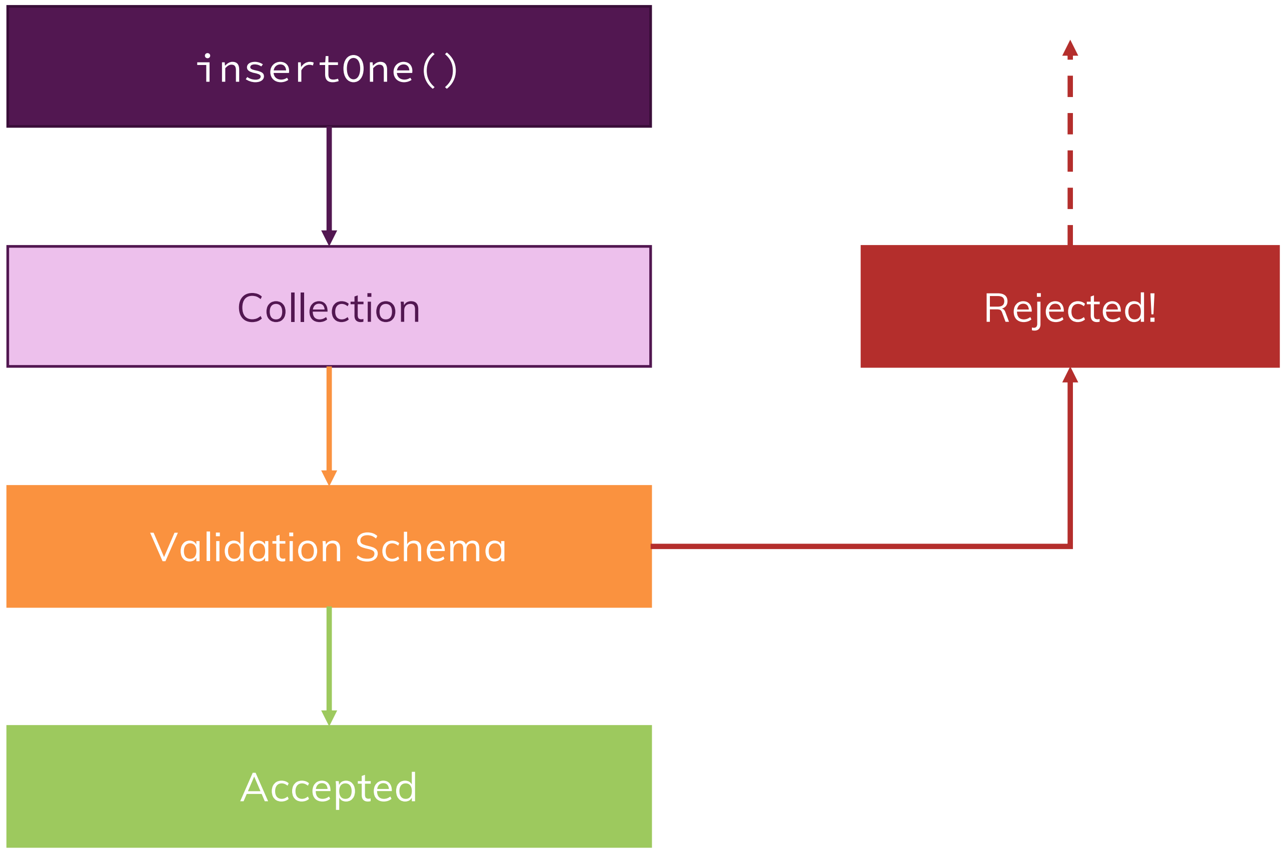
```
{
  _id: 'id1',
  name: 'Lord of the Rings 1'
}
```

```
customers.aggregate([
  { $lookup: {
      from: "books",
      localField: "favBooks",
      foreignField: "_id"
      as: "favBookData"
    }
  }
])
```

Example Project: A Blog



Schema Validation



Schema Validation

`validationLevel`	
Which documents get validated?	
strict	All inserts & updates
moderate	All inserts & updates to correct documents

`validationAction`	
What happens if validation fails?	
error	Throw error and deny insert/ update
warn	Log warning but proceed

`bypassDocumentValidation()`

Data Modelling & Structuring – Things to Consider

In which Format will you fetch your Data?

How often will you fetch and change your Data?

How much data will you save (and how big is it)?

How is your Data related?

Will Duplicates hurt you (=> many Updates)?

Will you hit Data/ Storage Limits?

Module Summary

Modelling Schemas

- Schemas should be modelled based on your application needs
- Important factors are: Read and write frequency, relations, amount (and size) of data

Schema Validation

- You can define rules to validate inserts and update before writing to the database
- Choose your validation level and action based on your application requirements

Modelling Relations

- Two options: Embedded documents or references
- Use embedded documents if you got one-to-one or one-to-many relationships and no app or data size reason to split
- Use references if data amount/ size or application needs require it or for many-to-many relations
- Exceptions are always possible => Keep your app requirements in mind!

Working with Shell & Server

Beyond Start & Stop

What’s Inside This Module?

Start MongoDB Server as Process & Service

Configuring Database & Log Path (and Mode)

Fixing Issues

ACADE
MIND

Diving Deeper Into CREATE

A Closer Look at Creating & Importing Documents

What’s Inside This Module?

Document Creation Methods (CREATE)

Importing Documents

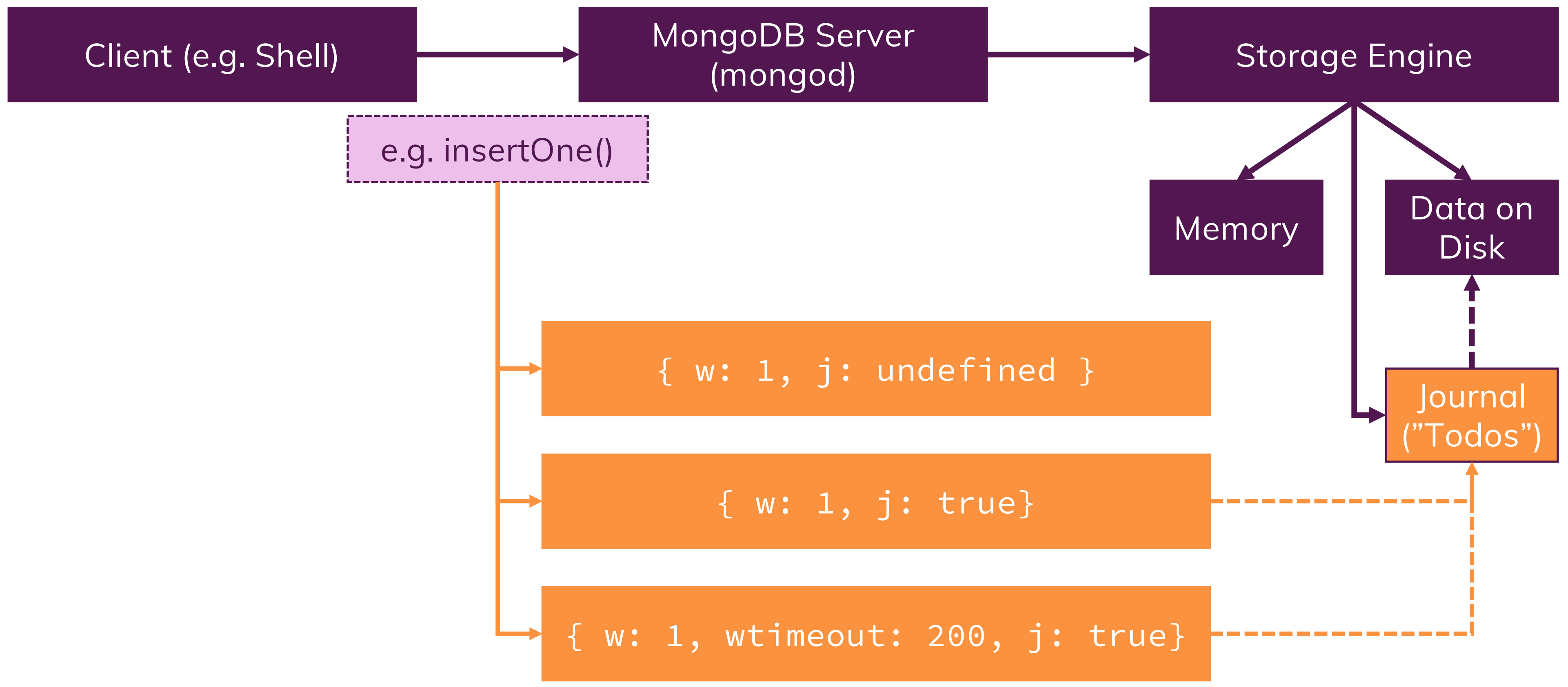
ACADE MIND

CREATE Documents

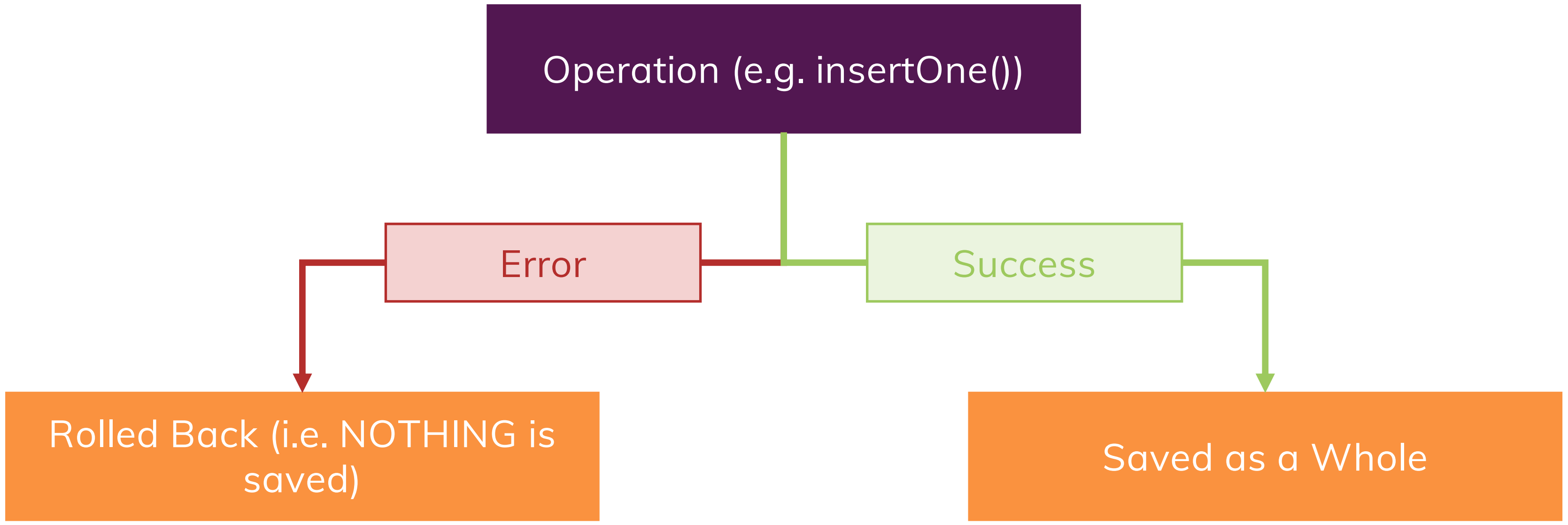
insertOne()	`db.collectionName.insertOne({field: "value"})`
insertMany()	`db.collectionName.insertMany([ {field: "value"}, {field: "value"}])`
insert()	`db.collectionName.insert()`
mongoimport	`mongoimport -d cars -c carsList --drop --jsonArray`

```
db.collectionName.insertMany([
                    {field: "value"},
                    {field: "value"}])
```

WriteConcern



What is “Atomicity”?


Operation (e.g. insertOne())
Error
Success
Rolled Back (i.e. NOTHING is saved)
Saved as a Whole

MongoDB CRUD Operations are Atomic on the Document Level (including Embedded Documents)

Tasks

1. Insert multiple companies (company data of your choice) into a collection – both with insertOne() and insertMany()
2. Deliberately insert duplicate ID data and “fix” failing additions with unordered inserts
3. Write data for a new company with both journaling being guaranteed and not being guaranteed

Module Summary

insertOne(), insertMany()

- You can insert documents with insertOne() (one document at a time) or insertMany() (multiple documents)
- insert() also exists but it's not recommended to use it anymore – it also doesn't return the inserted ids

WriteConcern

- Data should be stored and you can control the "level of guarantee" of that to happen with the writeConcern option
- Choose the option value based on your app requirements

Ordered Insertes

- By default, when using insertMany(), inserts are ordered – that means, that the inserting process stops if an error occurs
- You can change this by switching to "unordered inserts" – your inserting process will then continue, even if errors occurred
- In both cases, no successful inserts (before the error) will be rolled back

ACADE MIND

READing Documents with Operators

Accessing the Required Data Efficiently

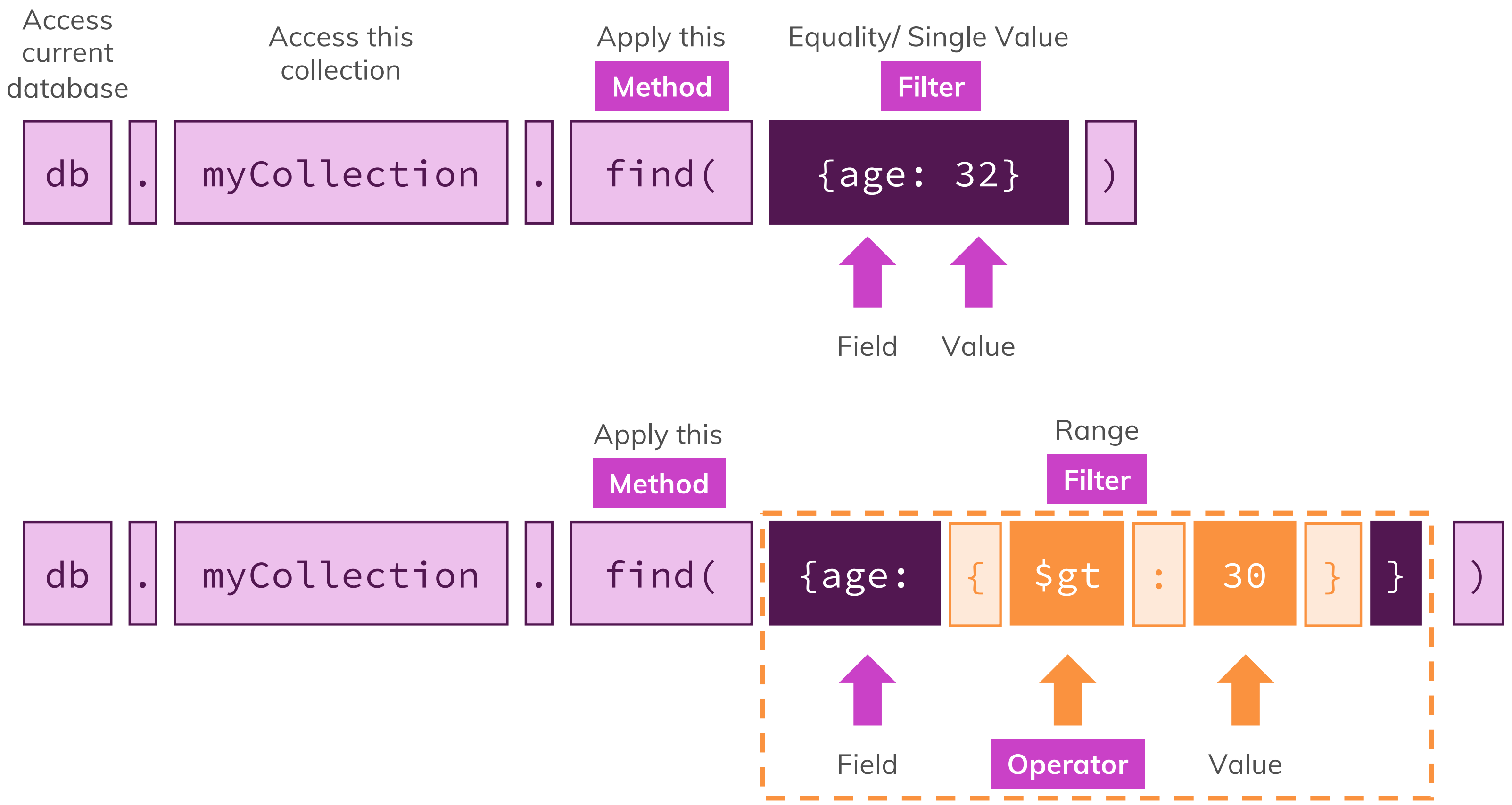
What’s Inside This Module?

Methods, Filters & Operators

Query Selectors (READ)

Projection Operators (READ)

Methods, Filters & Operators



Operators

Read	Update		
Query & Projection	Update	Query Modifiers	Aggregation
Query Selectors	Fields	Cha our Deprecated	Pipeline St Aggregation Module
Projection Operators	Arrays		perators

How Operators Impact our Data

Type	Purpose	Changes Data?	Example
Query Operator	Locate Data	🚫	$eq
Projection Operator	Modify data presentation	🚫	$
Update Operator	Modify + add additional data	✓	$inc

Query Selectors & Projection Operators

Query Selectors		Projection Operators
Comparison	Evaluation	`$`
Logical	Array	`$elemMatch`
Element	Comments	`$meta`
Geospatial (Geospatial Module)		`$slice`

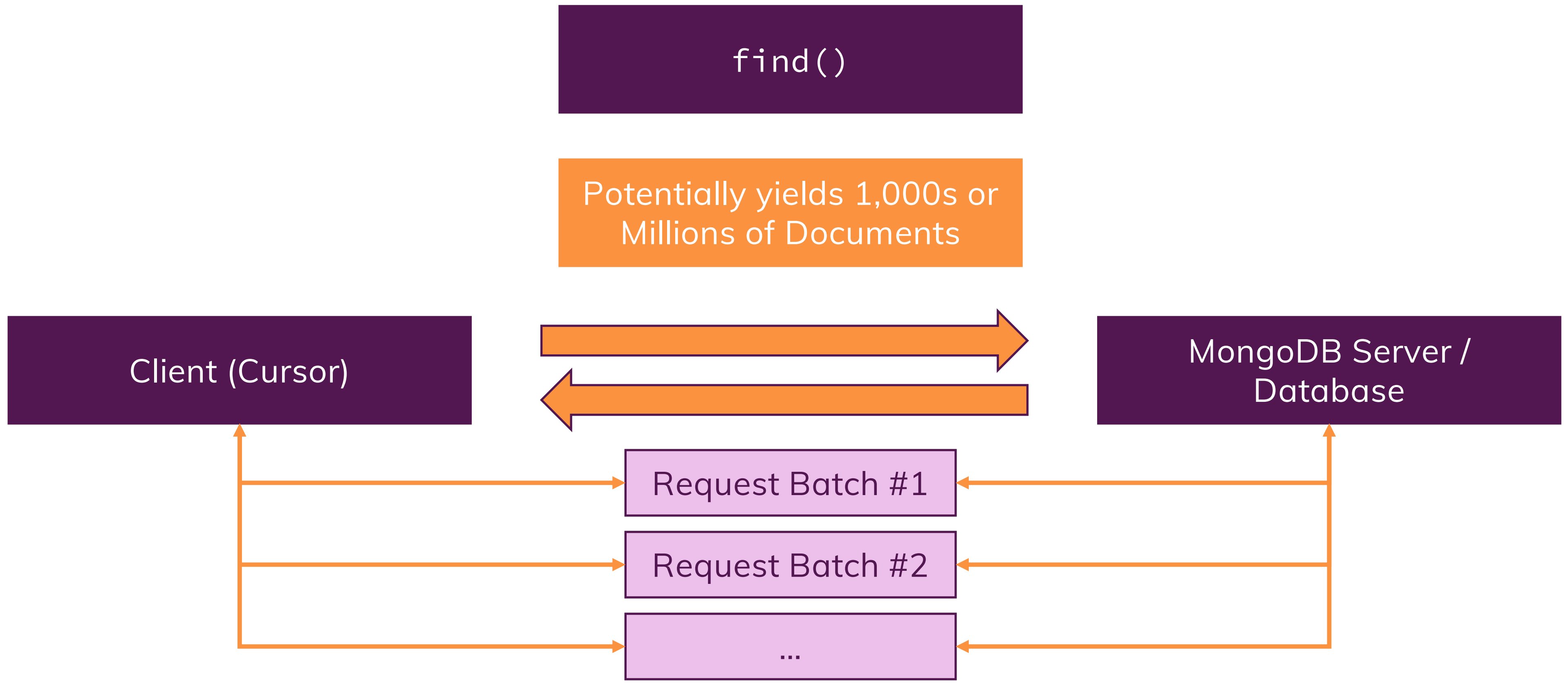
Tasks

1	Import the attached data into a new database (e.g. boxOffice) and collection (e.g. movieStarts)
2	Search all movies that have a rating higher than 9.2 and a runtime lower than 100 minutes
3	Search all movies that have a genre of “drama” or “action”
4	Search all movies where visitors exceeded expectedVisitors

Tasks

1	Import the attached data file into a new collection (e.g. exmoviestarts) in the boxOffice database
2	Find all movies with exactly two genres
3	Find all movies which aired in 2018
4	Find all movies which have ratings greater than 8 but lower than 10

Understanding Cursors



find()
Potentially yields 1,000s or Millions of Documents
Client (Cursor)
MongoDB Server / Database
Request Batch #1
Request Batch #2
...

Tasks

1	For this assignment, we'll work on the "extended boxoffice" dataset (which was imported in the previous assignment)
2	Filter for any data of your choice (e.g. all data) and make sure to only include title + visitors in your result data.
3	Search for all movies that have an entry of 10 in their ratings array and return just that array entry (inside of the array) in the result data
4	Repeat step 3) but return all "action" genre entries instead

Module Summary

Query Selectors & Operators

- You can read documents with find() and findOne()
- find() returns a cursor which allows you to fetch data step-by-step
- Both find() and findOne() take a filter (optional) to narrow down the set of documents they return
- Filters can use a variety of query selectors/ operators to control which documents are retrieved

Cursors

- find() returns a cursor to allow you to efficiently retrieve data step by step (instead of fetching all the documents in one step)
- You can use a cursor to move through the documents
- sort(), skip() and limit() can be used to control the order, portion and quantity of the retrieved results

Projection

- Projection allows you to control which fields are returned in your result set
- You can include fields (field: 1) and exclude them (field: 0)
- For arrays, special projection operators help you return the right field data

Understanding Document UPDATEs

Because we Always need the Latest Information

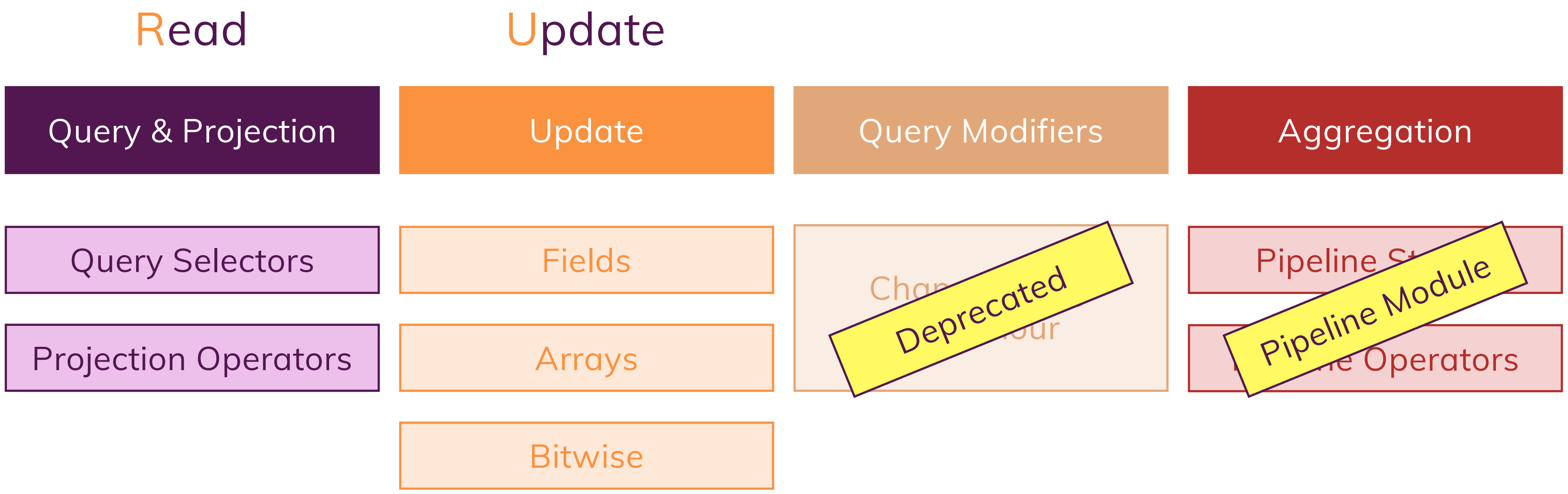
What’s Inside This Module?

Document Updating Operator (UPDATE)

Updating Fields

Updating Arrays

Operators


Read
Update
Query & Projection
Update
Query Modifiers
Aggregation
Query Selectors
Projection Operators
Fields
Arrays
Bitwise
Chan
our
Deprecated
Pipeline St
ne Operators
Pipeline Module

How Operators Impact our Data

Type	Purpose	Changes Data?	Example
Query Operator	Locate Data	⊘	$eq
Projection Operator	Modify data presentation	⊘	$elemMatch
Update Operator	Modify + add additional data	✓	$rename

Update Operators

Operators		Operator Examples	
Fields		`$currentDate`	`$mul`
Arrays	Operators	`$push`	`$pop`
	Modifiers	`$position`	`$slice`

Tasks

1	Create a new collection (“sports”) and upsert two new documents into it (with these fields: “title”, “requiresTeam”)
2	Update all documents which do require a team by adding a new field with the minimum amount of players required
3	Update all documents that require a team by increasing the number of required players by 10

Module Summary

updateOne() & updateMany()

- You can use updateOne() and updateMany() to update one or more documents in a collection
- You specify a filter (query selector) with the same operators you know from find()
- The second argument then describes the update (e.g. via $set or other update operators)

Update Operators

- You can update fields with a broad variety of field update operators like $set, $inc, $min etc
- If you need to work on arrays, take advantage of the shortcuts ($, $[] and $[<identifier>] + arrayFilters)
- Also use array update operators like $push or $pop to efficiently add or remove elements to or from arrays

Replacing Documents

- Even though it was not covered again, you also learned about replaceOne() earlier in the course – you can use that if you need to entirely replace a doc

DELETE Documents

Sometimes we have to Get Rid of Data

What’s Inside This Module?

Document Deletion Methods (DELETE)

Indexes

Retrieving Data Efficiently

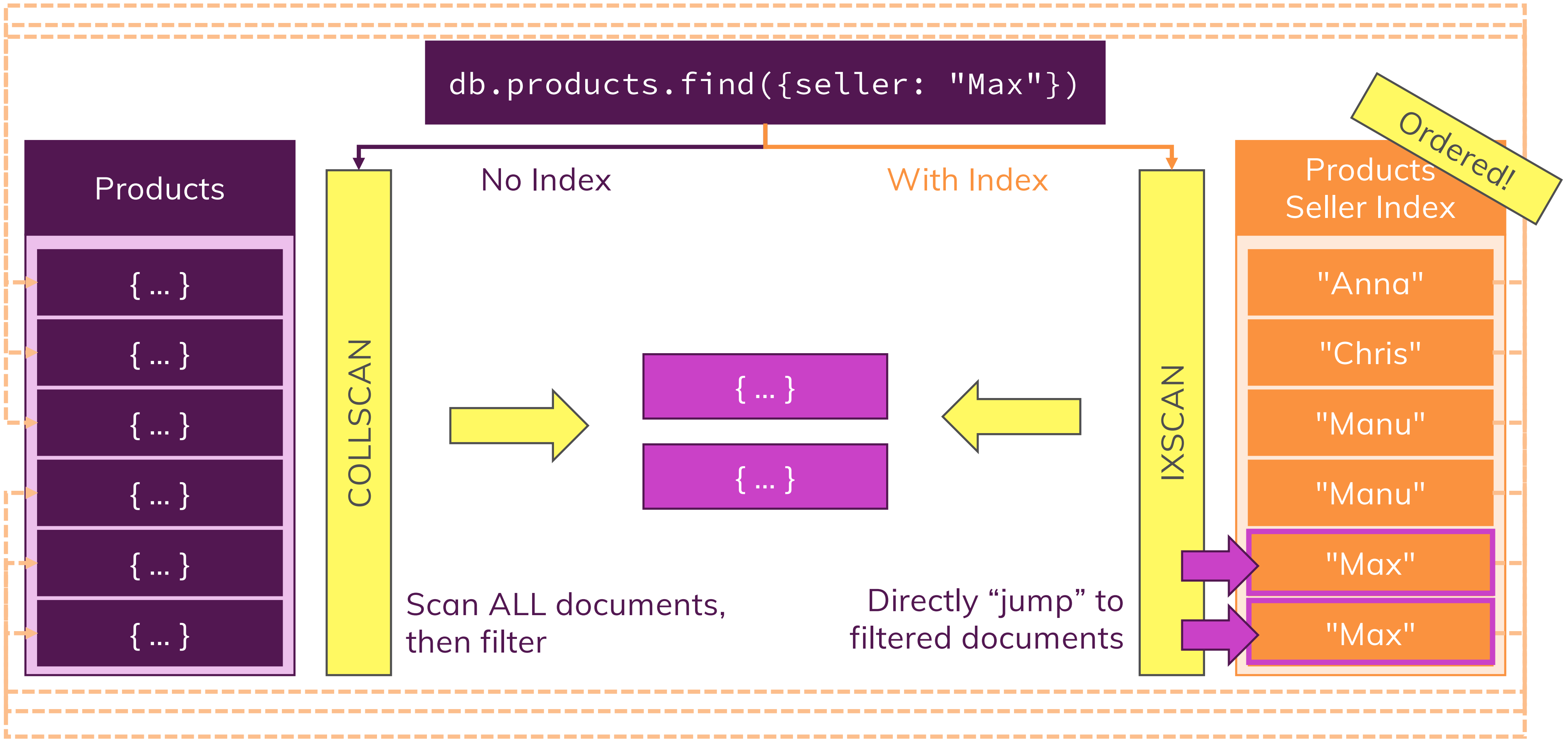
What's Inside This Module?

What are Indexes?

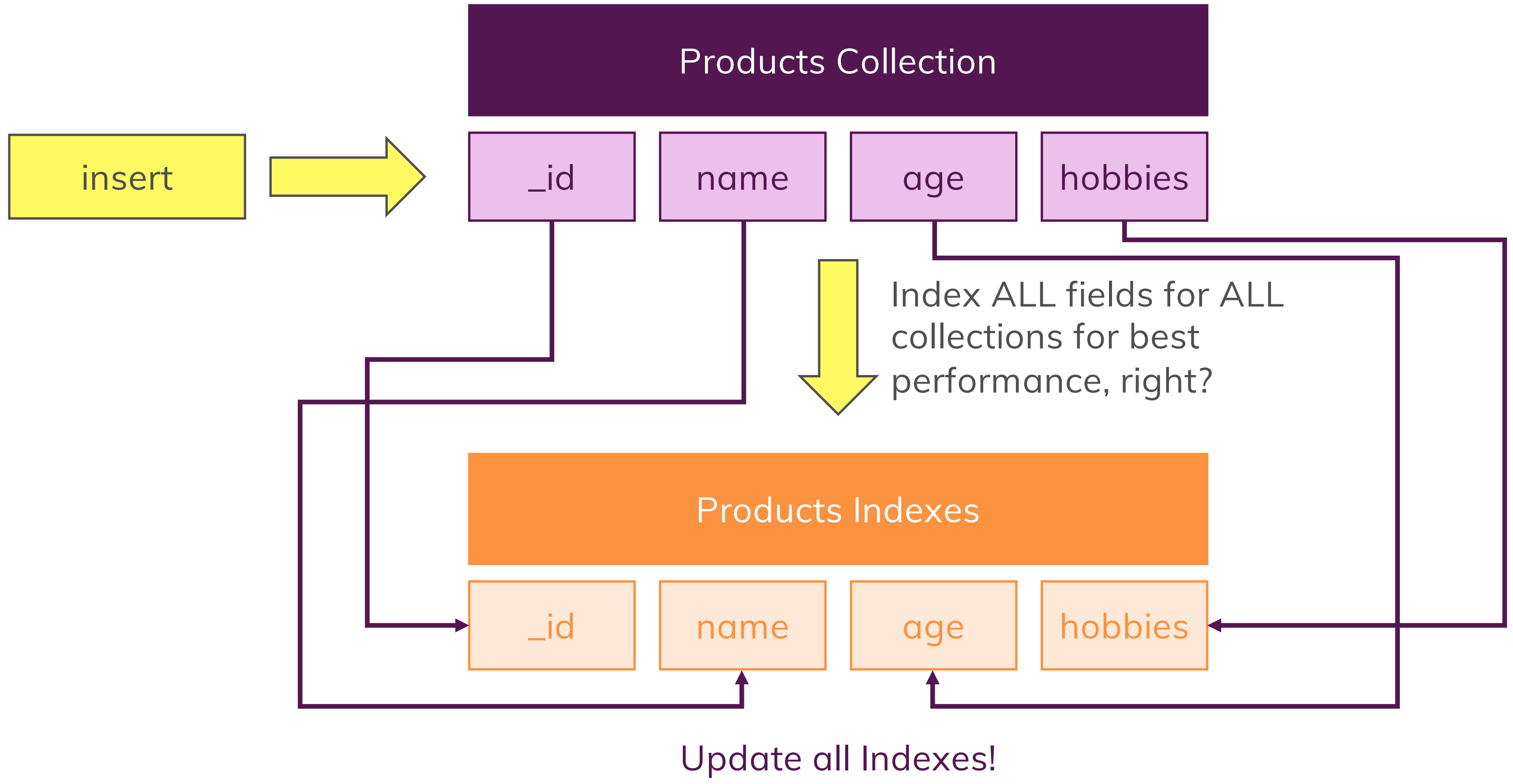
Different Types of Indexes

Using & Optimizing Indexes

Why Indexes?


db.products.find({seller: "Max"})
No Index
With Index
Products
{ ... }
{ ... }
{ ... }
{ ... }
{ ... }
{ ... }
COLLSCAN
{ ... }
{ ... }
IXSCAN
Products Seller Index
Ordered!
"Anna"
"Chris"
"Manu"
"Manu"
"Max"
"Max"
Scan ALL documents, then filter
Directly "jump" to filtered documents

Don't Use Too Many Indexes!



Index Types

"Normal"	Ordered field	{ name: 1 }
Compound	Multiple, combined ordered fields	{ name: 1, age: -1 }
Multikey	Ordered array values	{ hobbies: 1 }
Text	Ordered text fragments	{ description: "text" }
Geospatial	Ordered geodata	{ location: "2d" }

Index Config

Custom Name

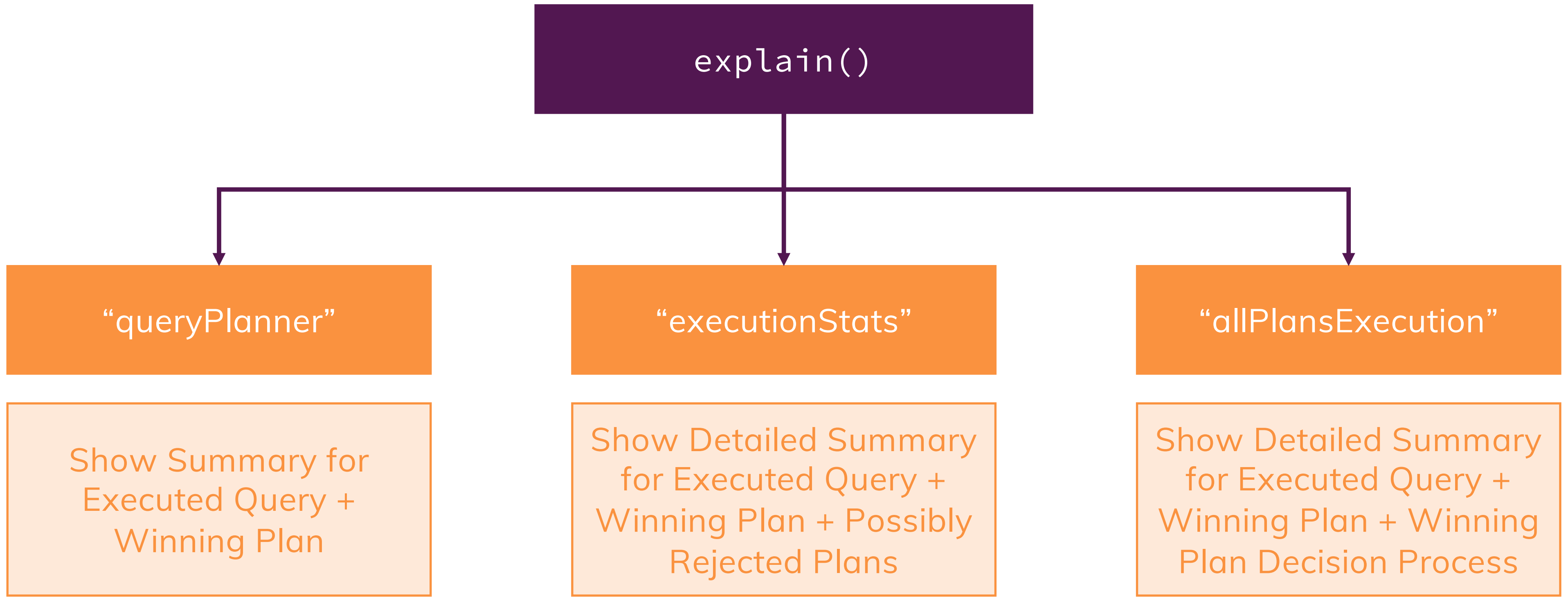
Unique

Partial

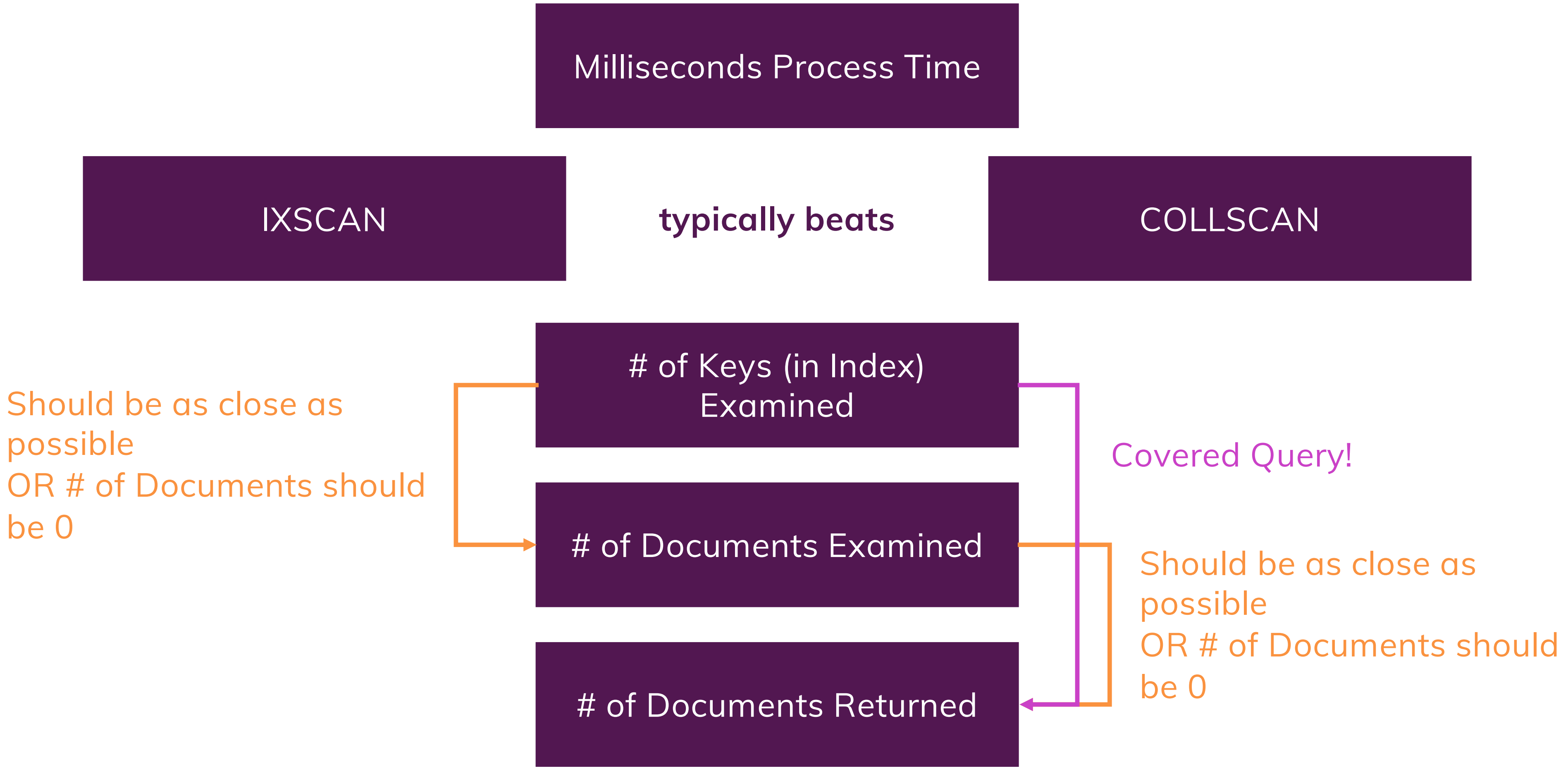
Sparse

TTL

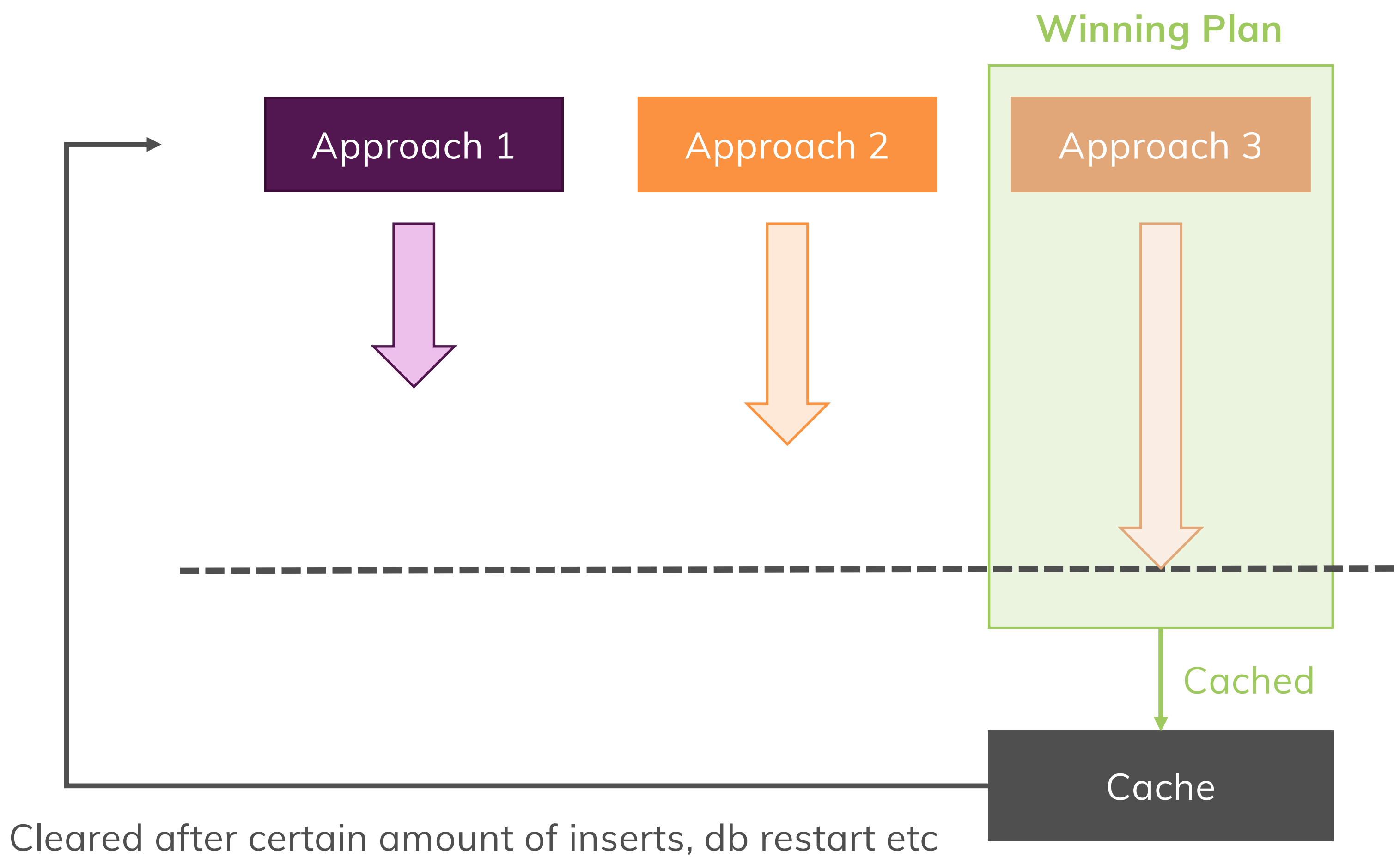
Query Diagnosis & Query Planning


explain()
“queryPlanner”
“executionStats”
“allPlansExecution”
Show Summary for Executed Query + Winning Plan
Show Detailed Summary for Executed Query + Winning Plan + Possibly Rejected Plans
Show Detailed Summary for Executed Query + Winning Plan + Winning Plan Decision Process

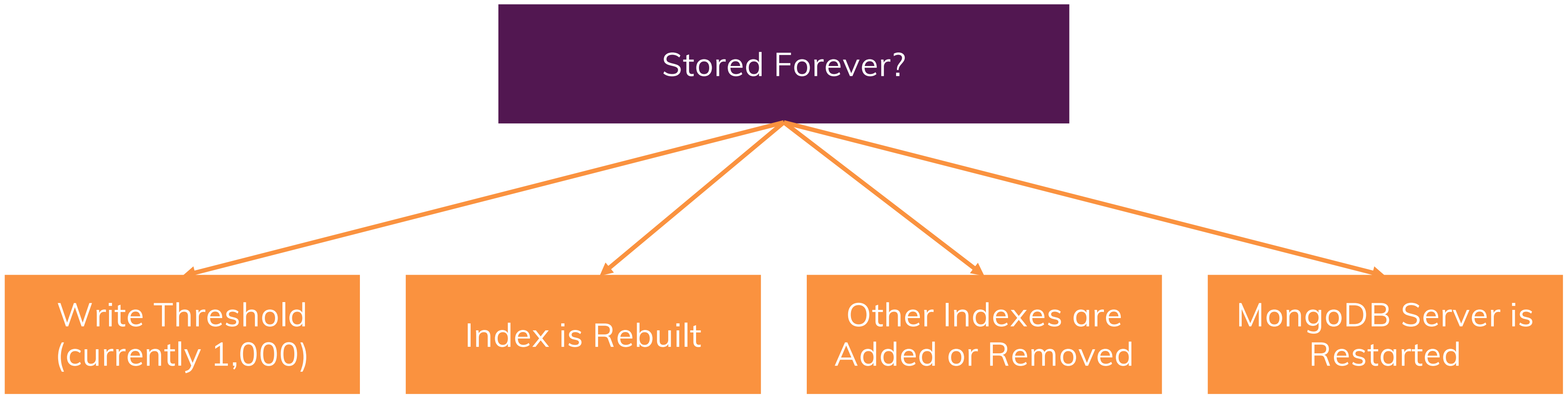
Efficient Queries & Covered Queries


Milliseconds Process Time
IXSCAN
typically beats
COLLSCAN
# of Keys (in Index) Examined
# of Documents Examined
# of Documents Returned
Should be as close as possible OR # of Documents should be 0
Covered Query!
Should be as close as possible OR # of Documents should be 0

“Winning Plans”



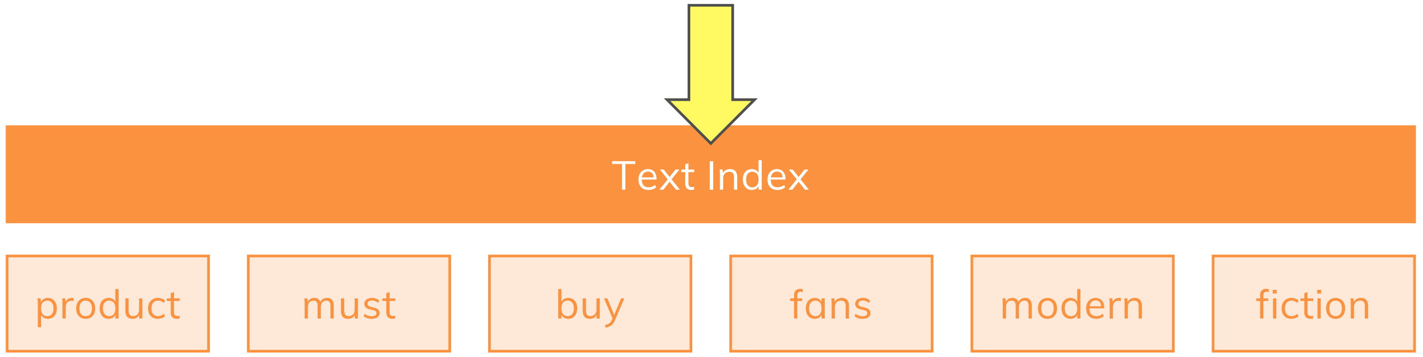
Clearing the Winning Plan from Cache



Stored Forever?
Write Threshold (currently 1,000)
Index is Rebuilt
Other Indexes are Added or Removed
MongoDB Server is Restarted

Understanding ”text” Indexes

This product is a must-buy for all fans of modern fiction!



Stopwords (e.g. “a”) are eliminated!

ACADEMIND

Building Indexes

Foreground	Background
Collection is locked during index creation	Collection is accessible during index creation
Faster	Slower

Module Summary

What and Why?

- Indexes allow you to retrieve data more efficiently (if used correctly) because your queries only have to look at a subset of all documents
- You can use single-field, compound, multi-key (array) and text indexes
- Indexes don't come for free, they will slow down your writes

Queries & Sorting

- Indexes can be used for both queries and efficient sorting
- Compound indexes can be used as a whole or in a "left-to-right" (prefix) manner (e.g. only consider the "name" of the "name-age" compound index)

Query Diagnosis & Planning

- Use explain() to understand how MongoDB will execute your queries
- This allows you to optimize both your queries and indexes

Index Options

- You can also create TTL, unique or partial indexes
- For text indexes, weights and a default_language can be assigned

Geospatial Queries

Finding Places

What's Inside This Module?

Storing Geospatial Data in GeoJSON Format

Querying Geospatial Data

Tasks

1	Pick 3 Points on Google Maps and store them in a collection
2	Pick a point and find the nearest points within a min and max distance
3	Pick an area and see which points (that are stored in your collection) it contains
4	Store at least one area in a different collection
5	Pick a point and find out which areas in your collection contain that point

Module Summary

Storing Geospatial Data

- You store geospatial data next to your other data in your documents
- Geospatial data has to follow the special GeoJSON format – and respect the types supported by MongoDB
- Don’t forget that the coordinates are [longitude, latitude], not the other way around!

Geospatial Queries

- $near, $geoWithin and $geoIntersects get you very far
- Geospatial queries work with GeoJSON data

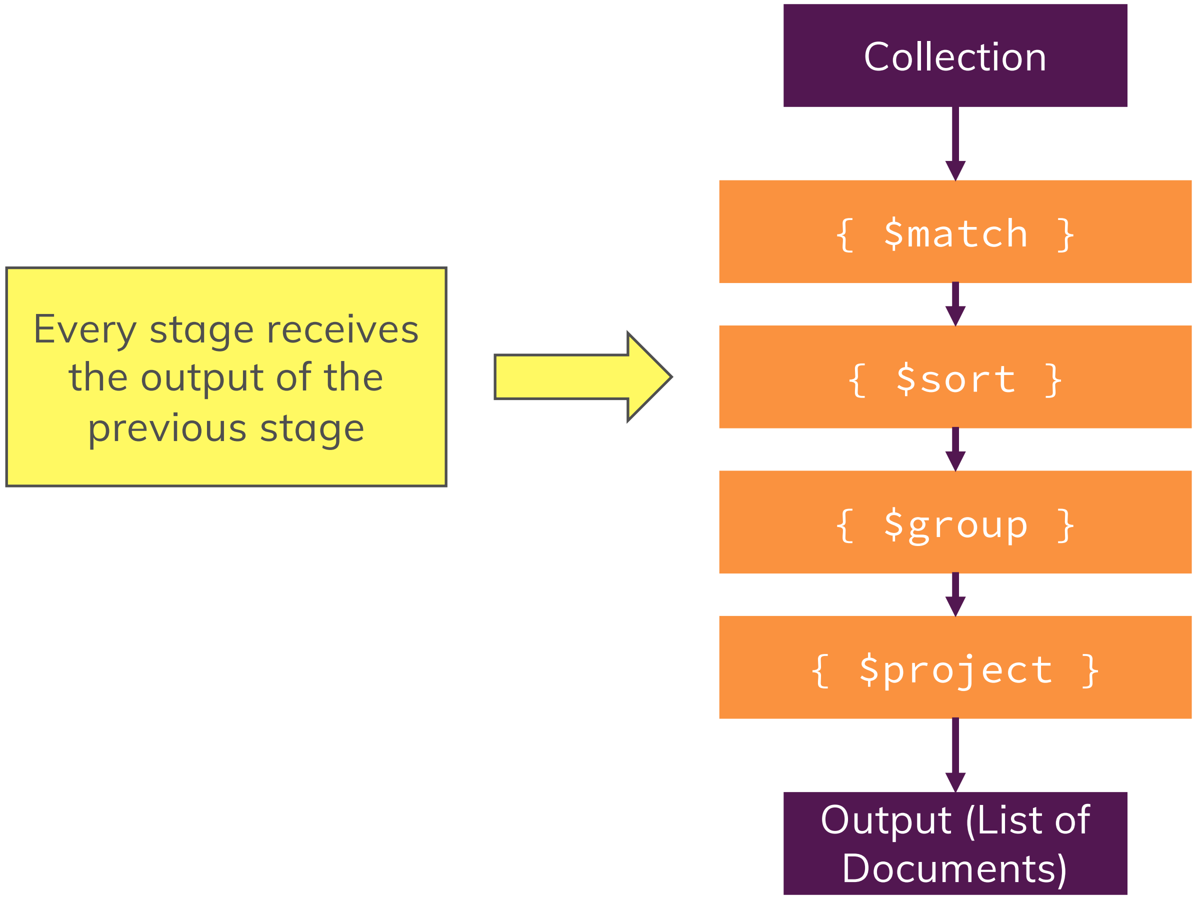
Geospatial Indexes

- You can add an index to geospatial data: “2dsphere”
- Some operations ($near) require such an index

Using the Aggregation Framework

Retrieving Data Efficiently & In a Structured Way

What is the “Aggregation Framework”?

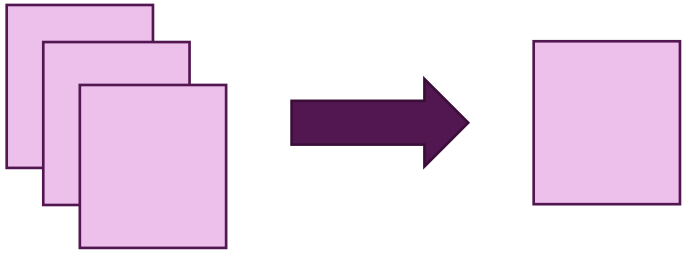

Every stage receives the output of the previous stage
Collection
{ $match }
{ $sort }
{ $group }
{ $project }
Output (List of Documents)

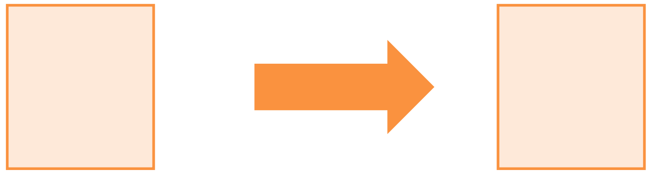
Pipeline Stages

Check official docs

$group vs $project

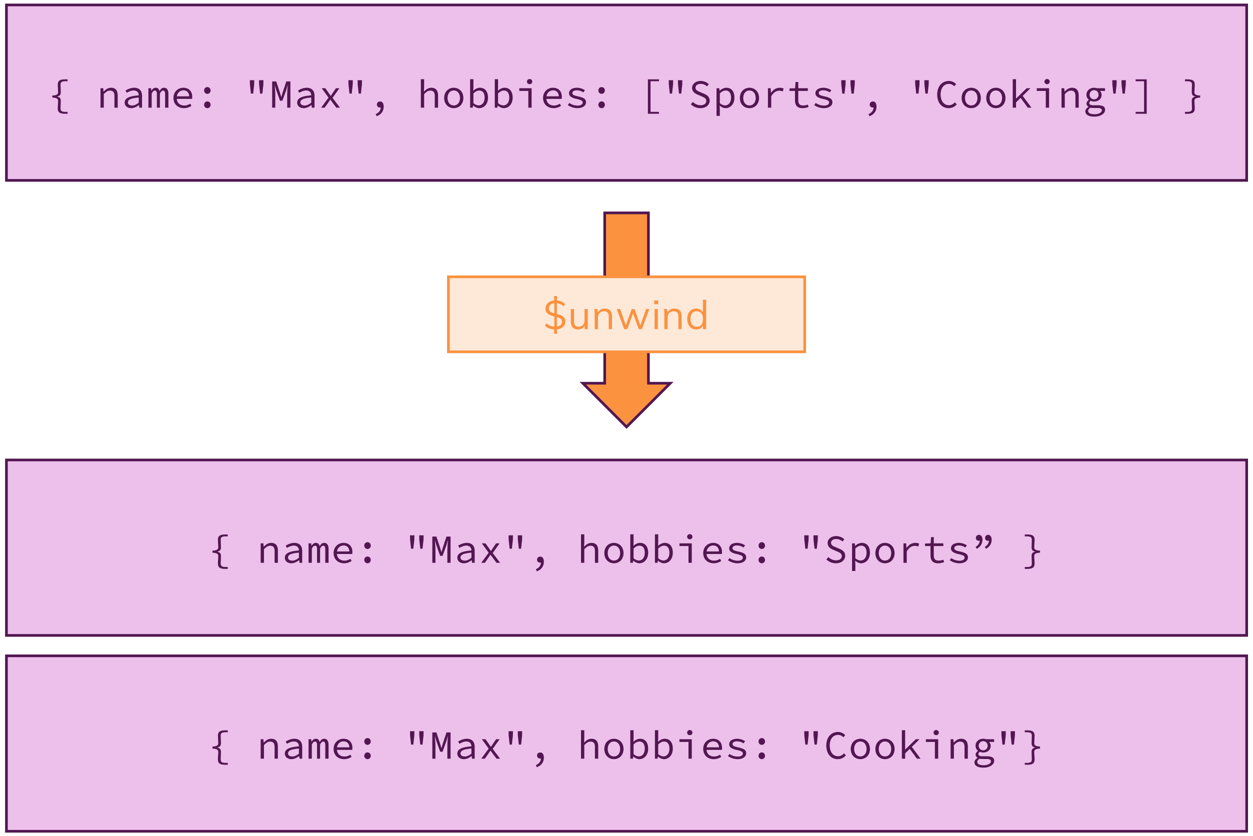
$group	$project
n:1	1:1





Sum, Count, Average, Build Array	Include/ Exclude Fields, Transform Fields (within a Single Document)

$unwind



```
{ name: "Max", hobbies: ["Sports", "Cooking"] }
```

```
{ name: "Max", hobbies: "Sports” }
```

```
{ name: "Max", hobbies: "Cooking"}
```

$skip + $limit + $sort

The Order Matters!

$$\begin{tikzcd}\texttt{\$sort} \arrow[r] & \texttt{\$skip} \arrow[r] & \texttt{\$limit}\end{tikzcd}$$

$text

Do a Text Index Search

Has to be the First Pipeline Stage!

Aggregation Pipeline Optimization

MongoDB automatically optimizes for you!

Module Summary

Stages & Operators

- There are plenty of available stages and operators you can choose from
- Stages define the different steps your data is funneled through
- Each stage receives the output of the last stage as input
- Operators can be used inside of stages to transform, limit or re-calculate data

Important Stages

- The most important stages are $match, $group, $project, $sort and $unwind – you'll work with these a lot
- Whilst there are some common behaviors between find() filters + projection and $match + $project, the aggregation stages generally are more flexible

Working with Numeric Data

More Complex Than You Might Think

Integers, Longs, Doubles

Integers (int32)	Longs (int64)	Doubles (64bit)	"High Precision Doubles" (128bit)
Only full Numbers	Only full Numbers	Numbers with Decimal Places	Numbers with Decimal Places
-2,147,483,648 to 2,147,483,647	-9,223,372,036,854, 775,808 to 9,223,372,036,854, 775,807	Decimal values are approximated	Decimal values are stored with high precision (34 decimal digits)
Use for "normal" integers	Use for large integers	Use for floats where high precision is not required	Use for floats where high precision is required

High Precision Floating Point Numbers

Doubles (64bit Floats)	Decimal (128bit Floats)
MongoDB Default for ALL Numbers	Has to be Created Explicitly
Higher Range of Numbers but lower Decimal Precision	Lower Range of Numbers but higher Decimal Precision

ACADE
MIND

Security & User Authentication

Lock Down Your Data

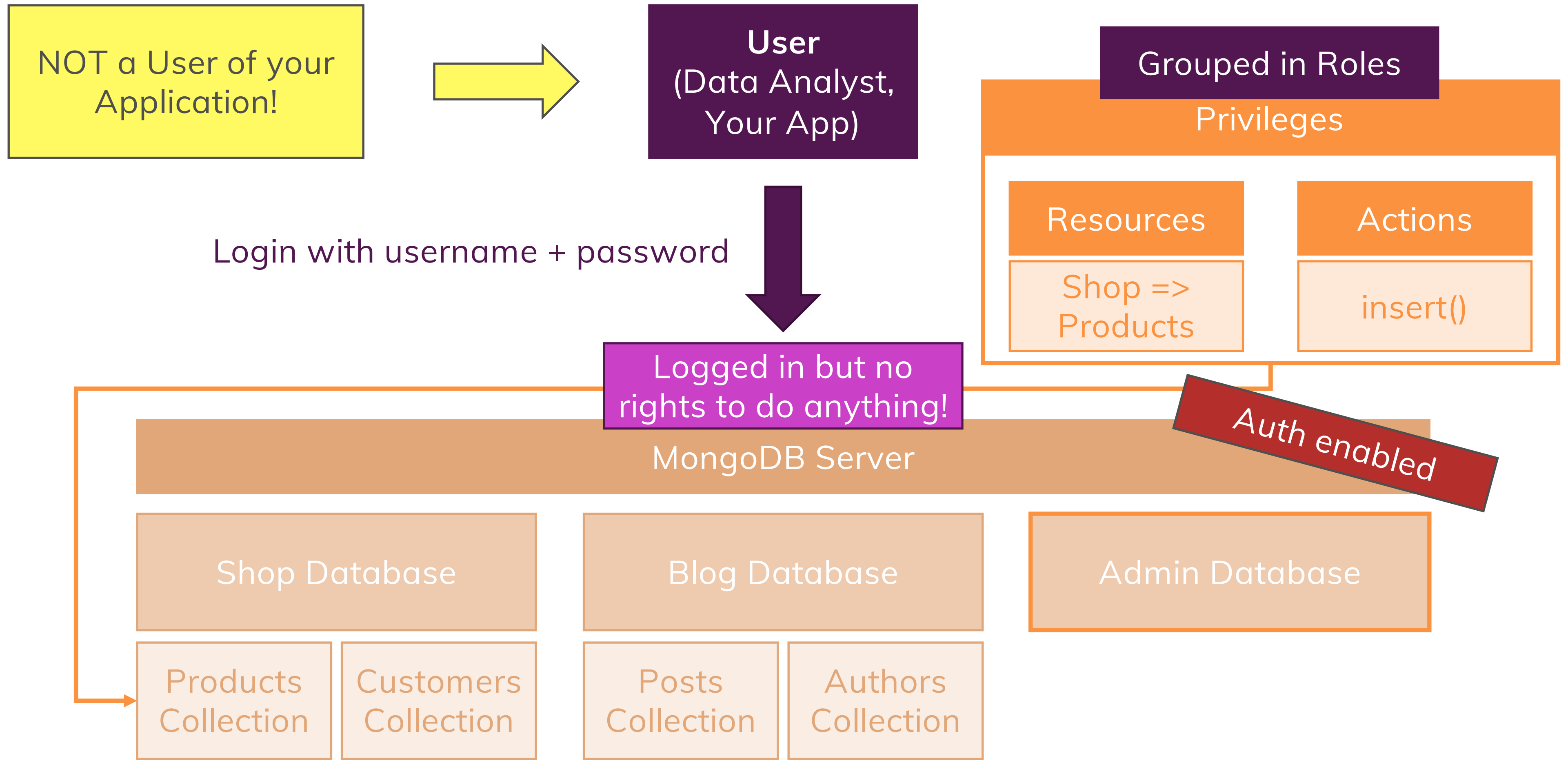
Security Checklist

- Authentication & Authorization
- Transport Encryption
- Encryption at Rest
- Auditing
- Server & Network Config and Setup
- Backups & Software Updates

Authentication & Authorization

Authentication	Authorization
Identifies valid users of the database	Identifies what these users may actually do in the database
Analogy: You are employed and therefore may access the office	Analogy: You are employed as an account and therefore may access the office and process orders

Role Based Access Control

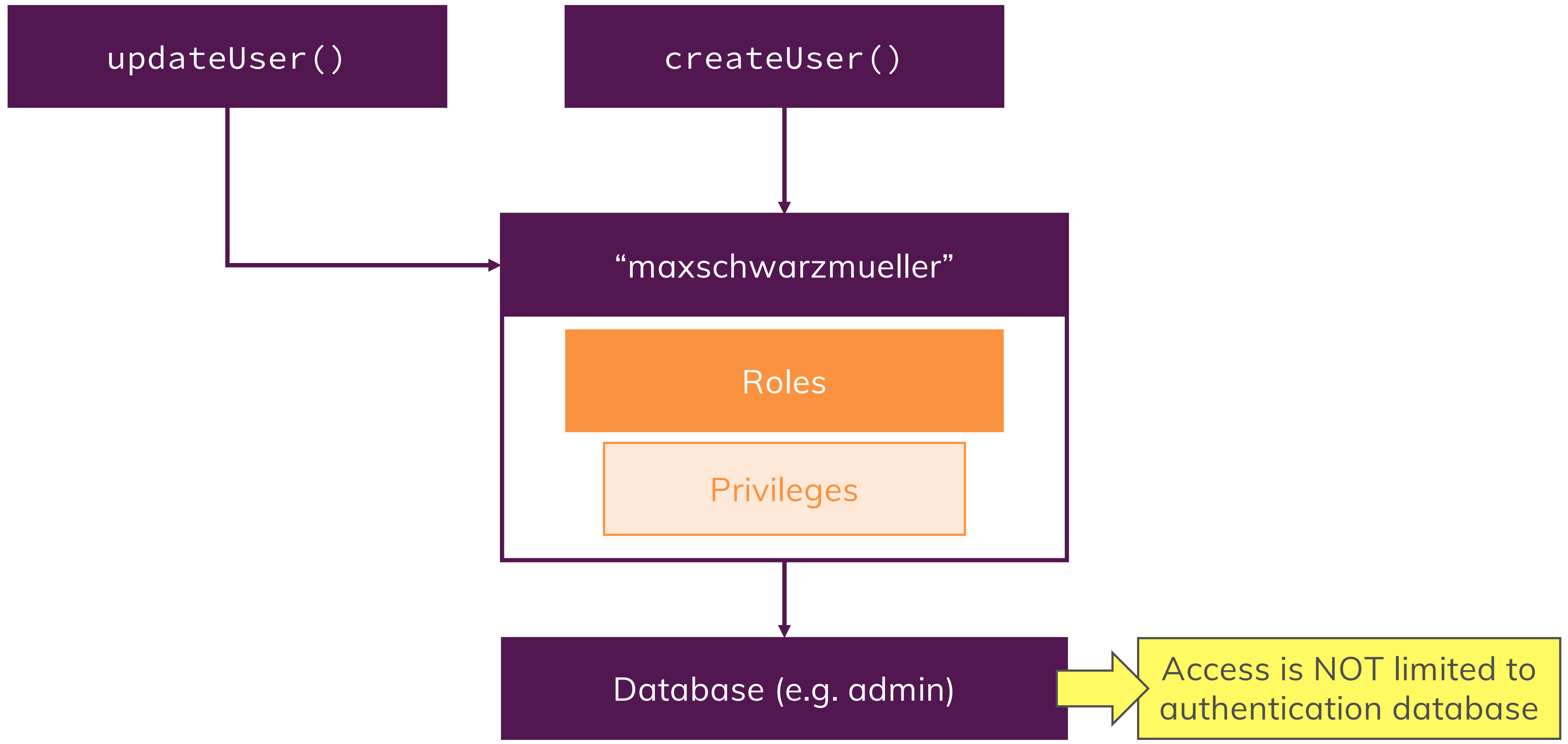


Why Roles?

Different Types of Database Users

Administrator	Developer / Your App	Data Scientist
Needs to be able to manage the database config, create users etc	Needs to be able to insert, update, delete or fetch data (CRUD)	Needs to be able to fetch data
Does NOT need to be able to insert or fetch data	Does NOT need to be able to create users or manage the database config	Does NOT need to be able to create users, manage the database config or insert, edit or delete data

Creating & Editing Users

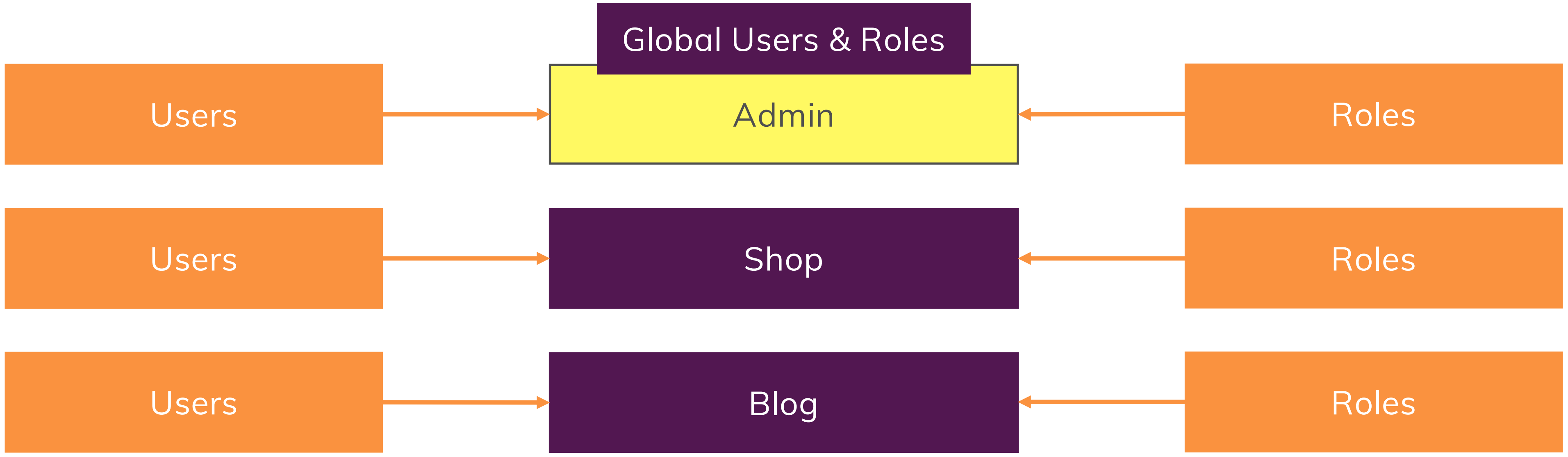


Built-in Roles

Database User	Database Admin	All Database Roles
read readWrite	dbAdmin userAdmin dbOwner	readAnyDatabase readWriteAnyDatabase userAdminAnyDatabase dbAdminAnyDatabase

Cluster Admin	Backup/ Restore	Superuser
clusterManager clusterMonitor hostManager clusterAdmin	backup restore	dbOwner (admin) userAdmin (admin) userAdminAnyDatabase root

What’s Up With The Databases?



User authenticate against their Database

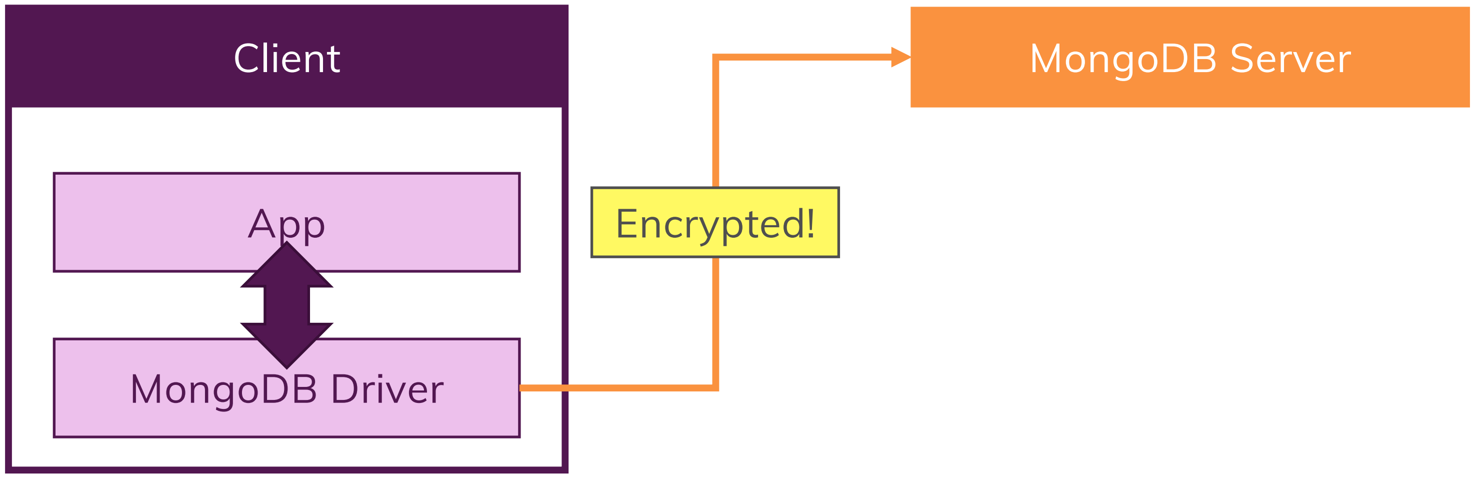
Access is **NOT limited** to that Database though because **Roles define Access Rights**

Roles are attached to Databases and can only be assigned to Users who use this Database as an Authentication Database

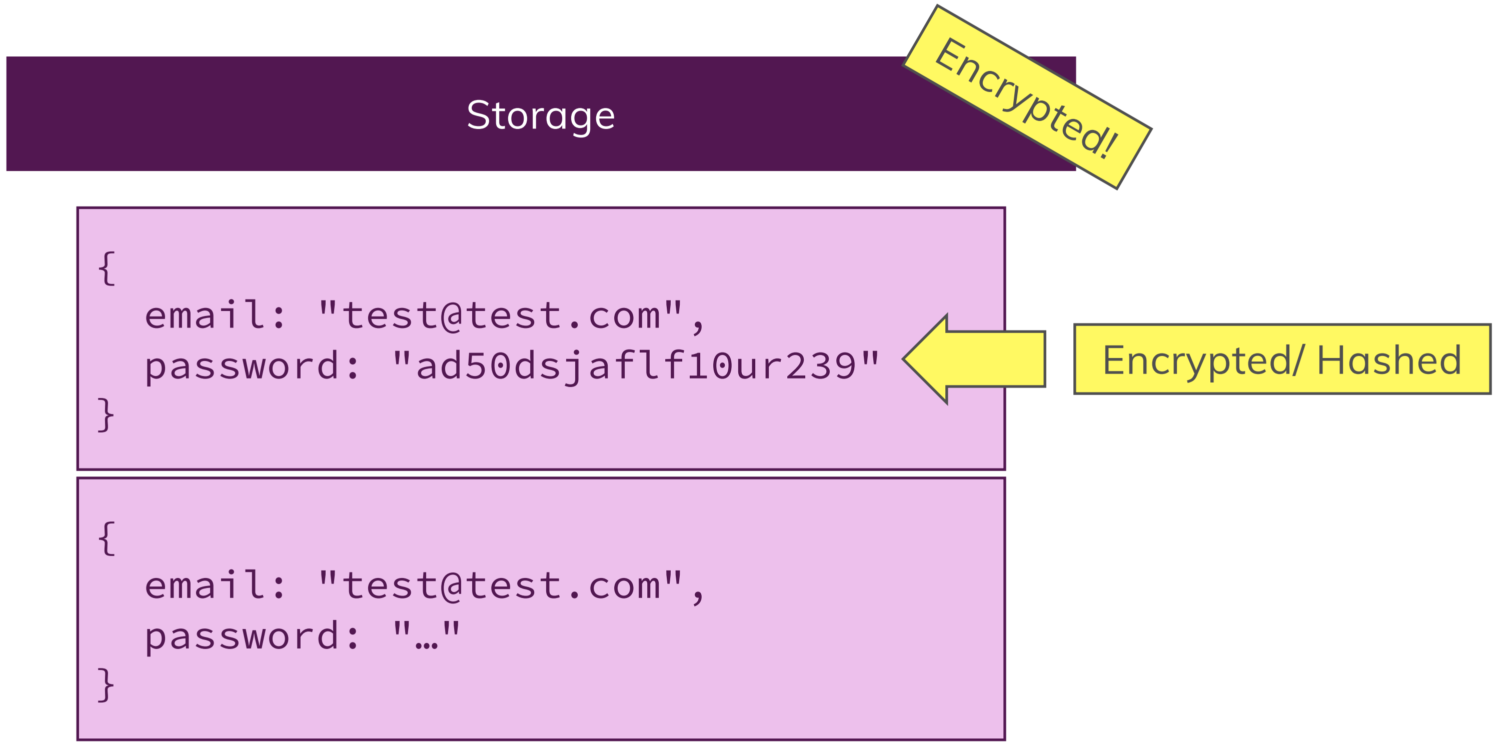
Practice!

Database Admin	User Admin	Developer
Work on Database, Create Collections, Create Indexes	Manage Users	Read & Write Data in ”Customers” and “Sales” Databases

Transport Encryption



Encryption at Rest



```
{
  email: "test@test.com",
  password: "ad50dsjaflf10ur239"
}
```

```
{
  email: "test@test.com",
  password: "…"
}
```

Module Summary

Users & Roles

- MongoDB uses a Role Based Access Control approach
- You create users on databases and you then log in with your credentials (against those databases)
- Users have no rights by default, you need to add roles to allow certain operations
- Permissions that are granted by roles ("Privileges") are only granted for the database the user was added to unless you explicitly grant access to other databases
- You can use "AnyDatabase" roles for cross-database access

Encryption

- You can encrypt data during transportation and at rest
- During transportation, you use TLS/ SSL to encrypt data
- For production, you should use SSL certificates issues by a certificate authority (NOT self-signed certificates)
- For encryption at rest, you can encrypt both the files that hold your data (made simple with "MongoDB Enterprise") and the values inside your documents

Performance, Fault Tolerance & Deployment

Entering the Enterprise World

What's Inside This Module?

What influences Performance?

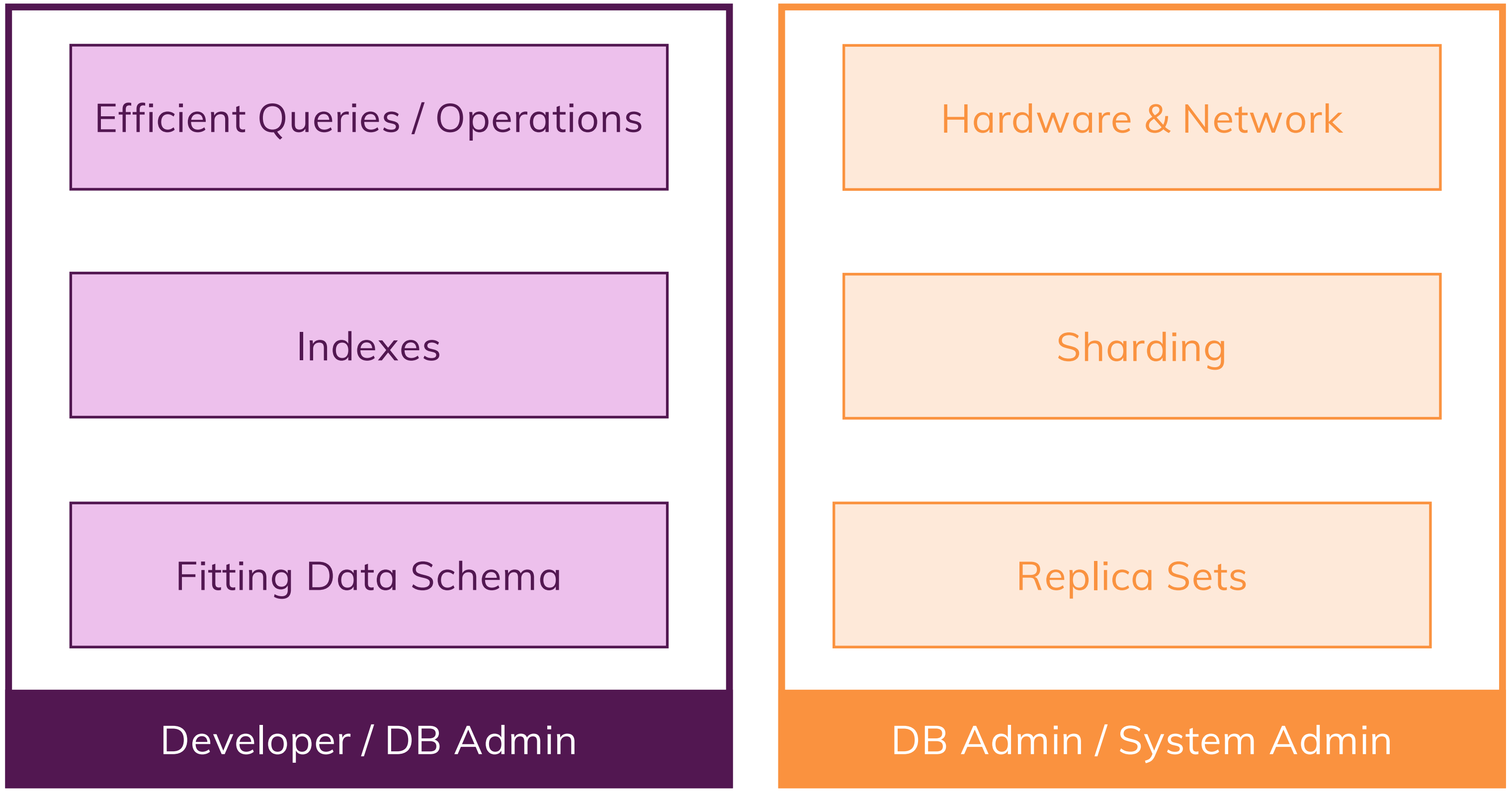
Capped Collections

Replica Sets

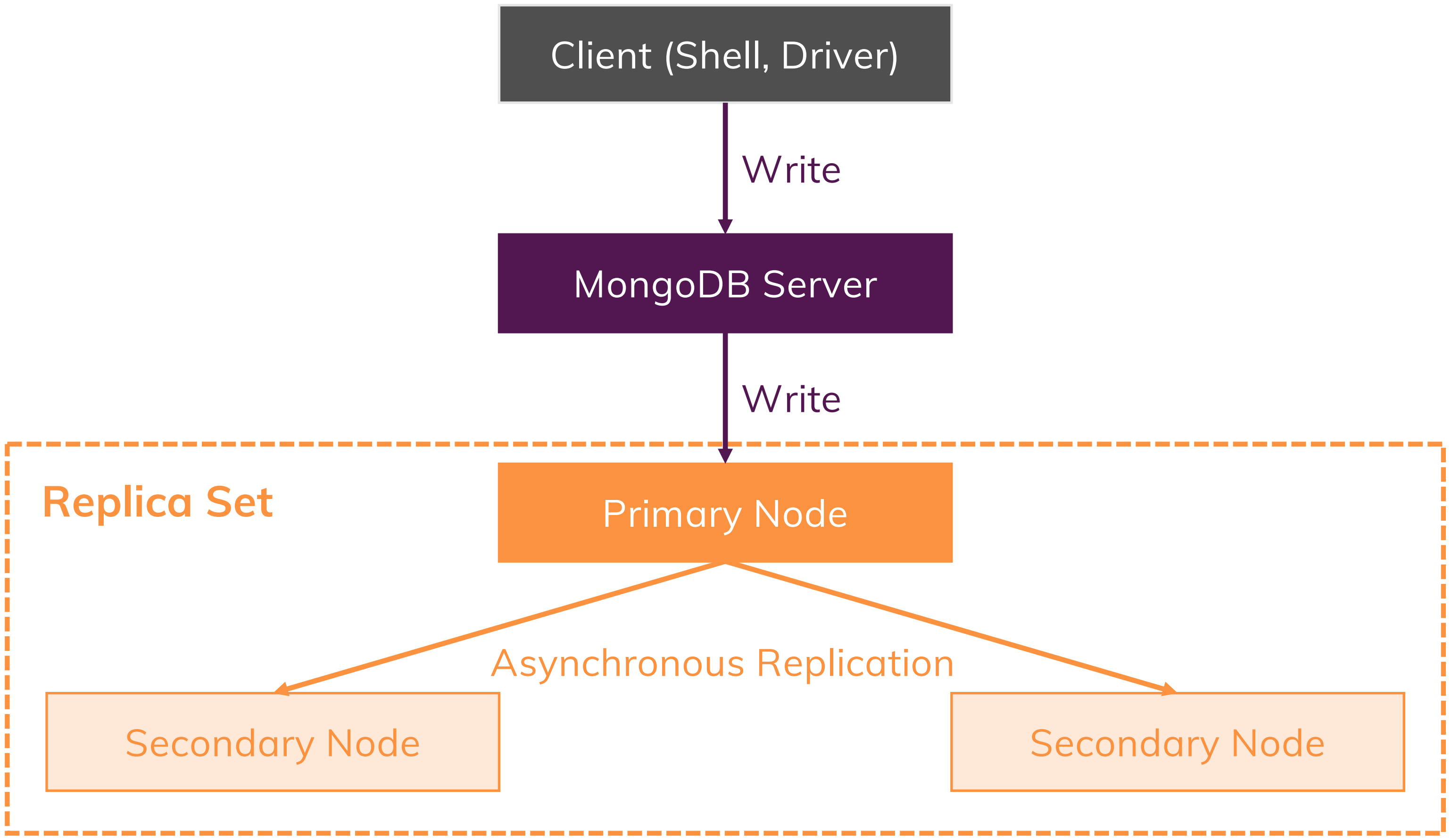
Sharding

MongoDB Server Deployment

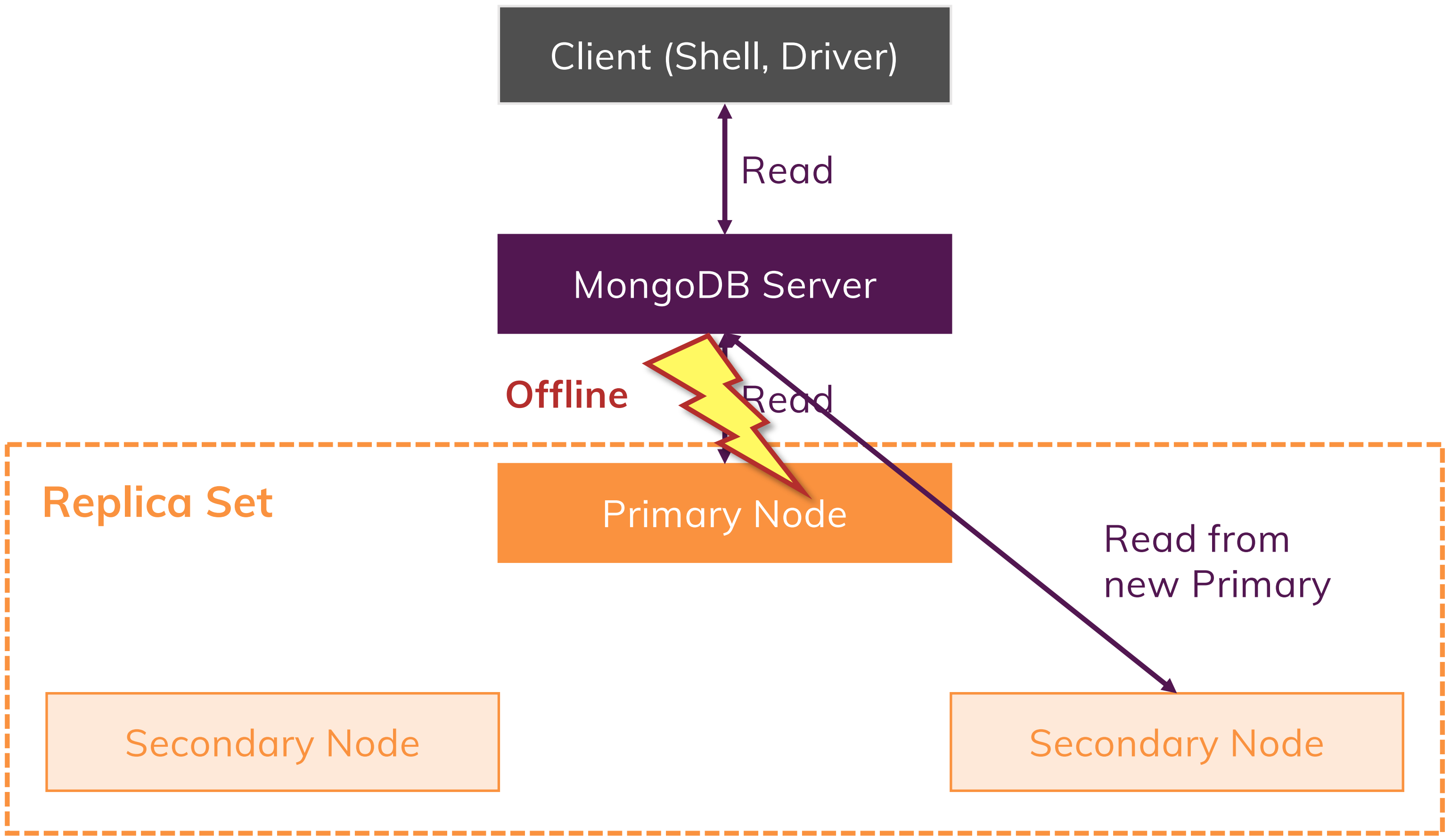
What Influences Performance?



Replica Sets



Replica Sets Reads

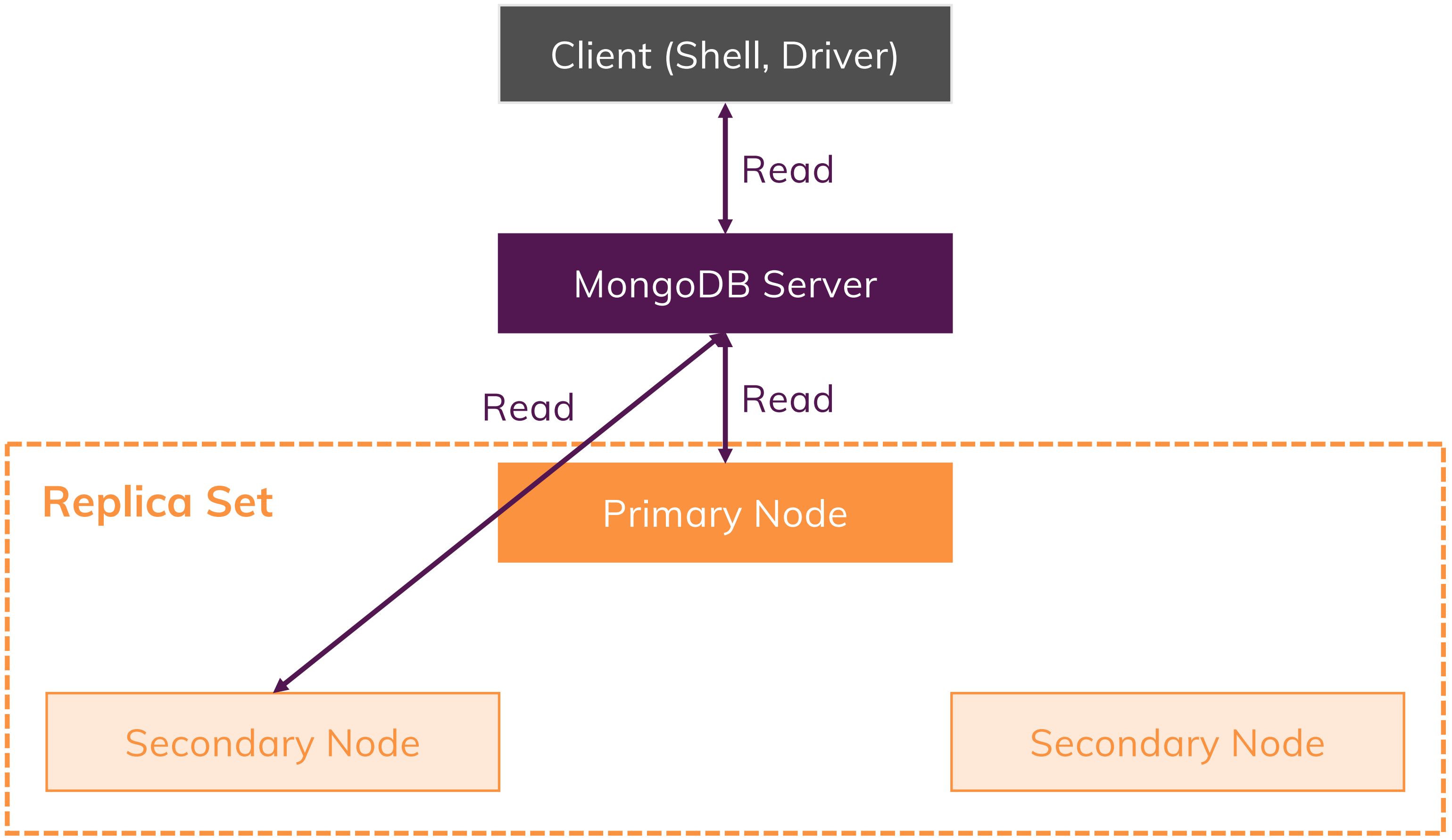


Why Replica Sets?

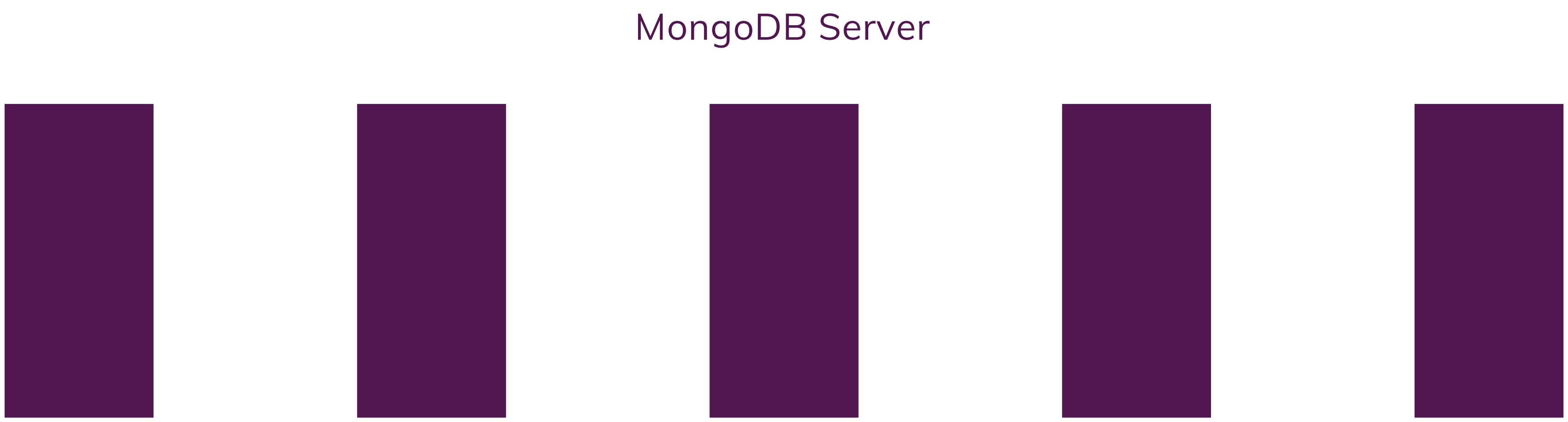
Backup / Fault Tolerancy

Improve Read Performance

Replica Sets Secondary Reads



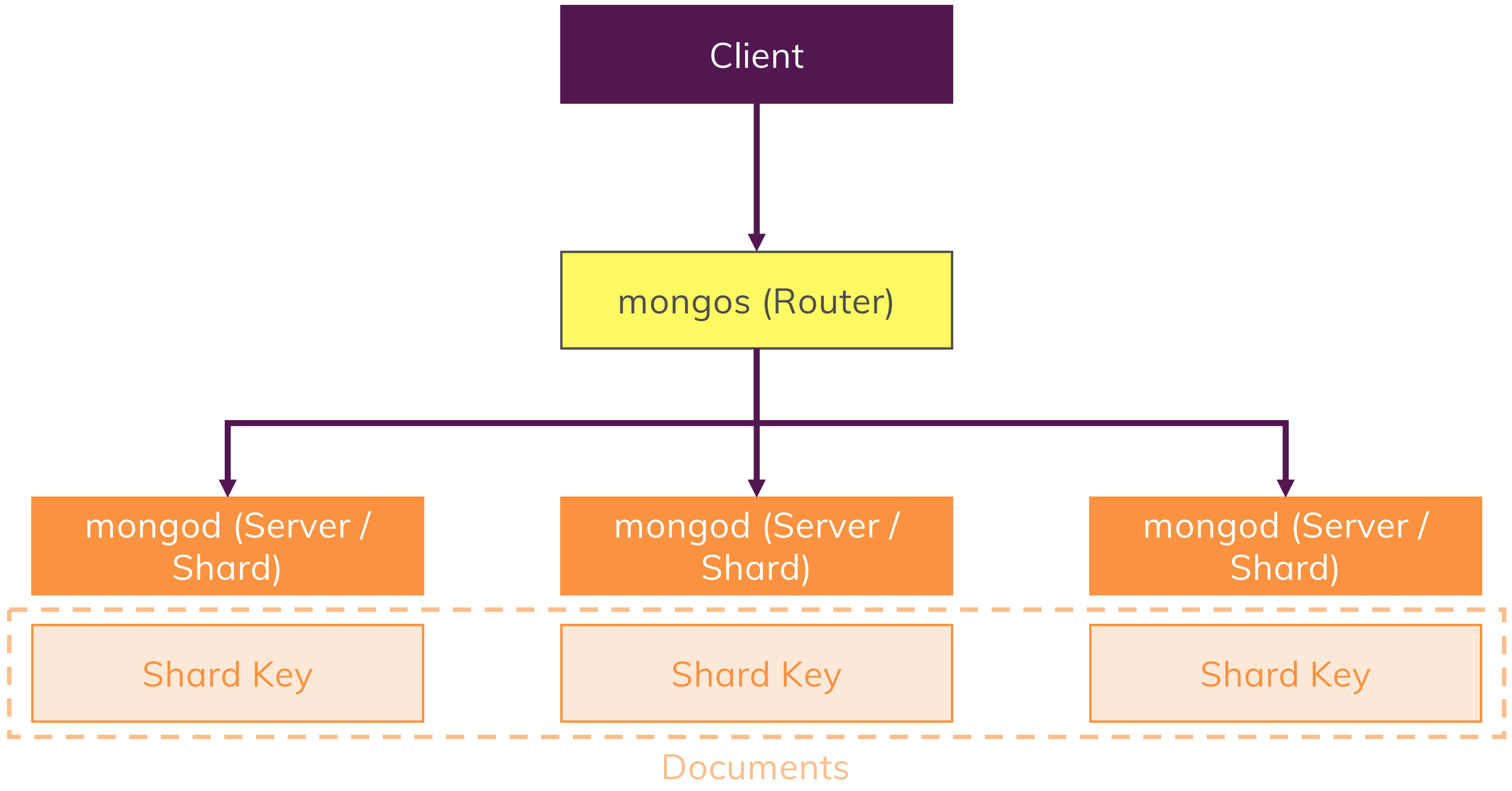
Sharding (Horizontal Scaling)



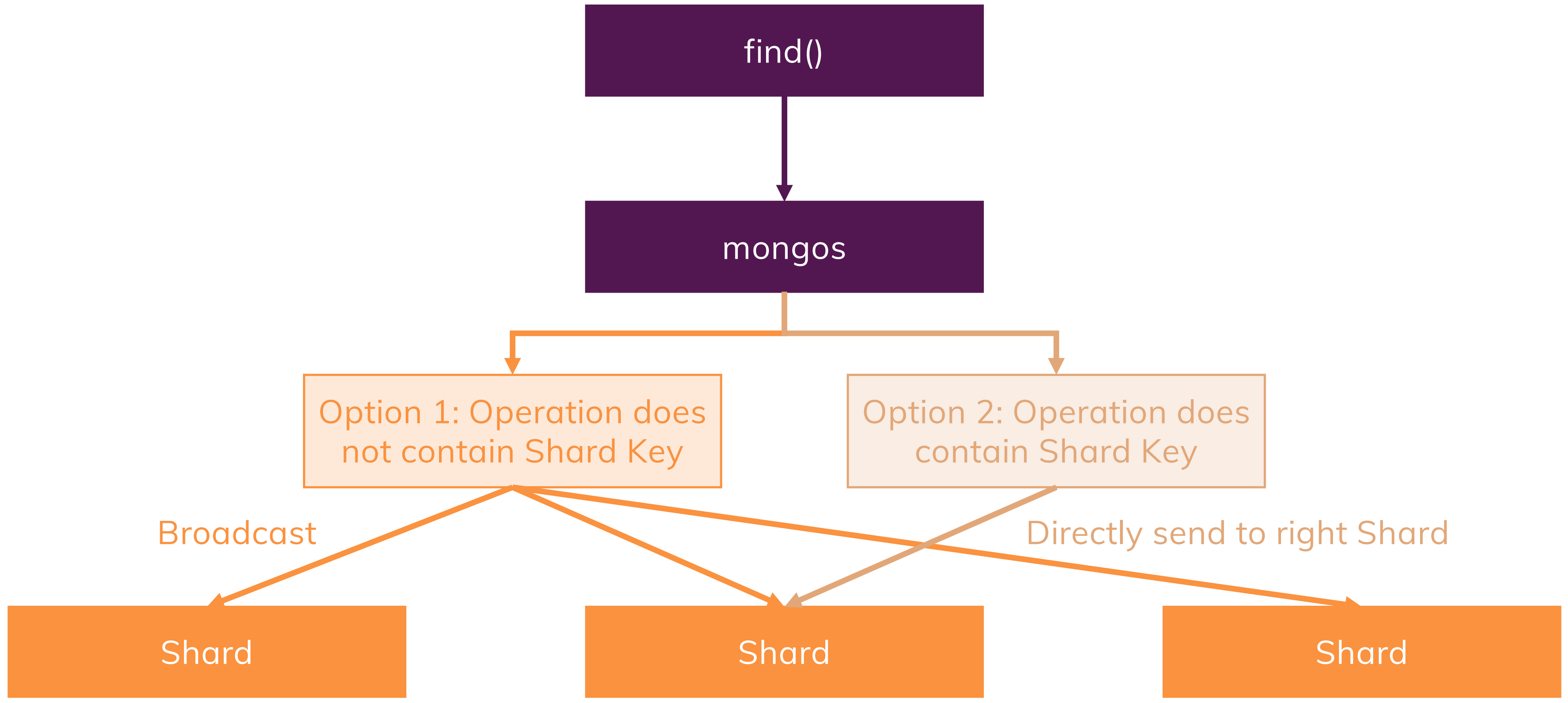
Data is distributed (not replicated!) across Shards

Queries are run across all Shards

How Sharding Works

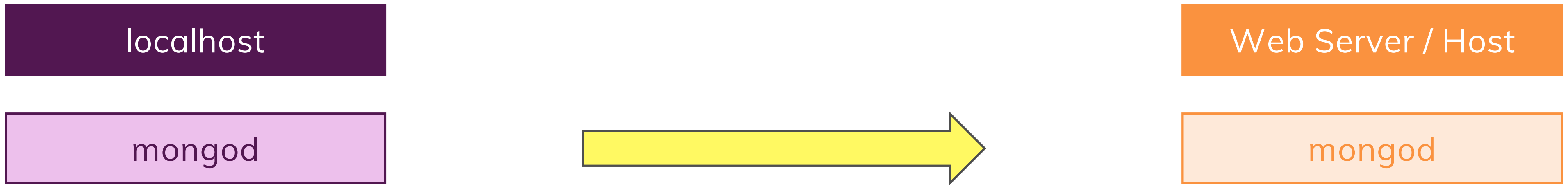


Queries & Sharding



find()
mongos
Option 1: Operation does not contain Shard Key
Option 2: Operation does contain Shard Key
Broadcast
Directly send to right Shard
Shard
Shard
Shard

Deploying a MongoDB Server



Manage Shards

Secure User / Auth Setup

Protect Web Server / Network

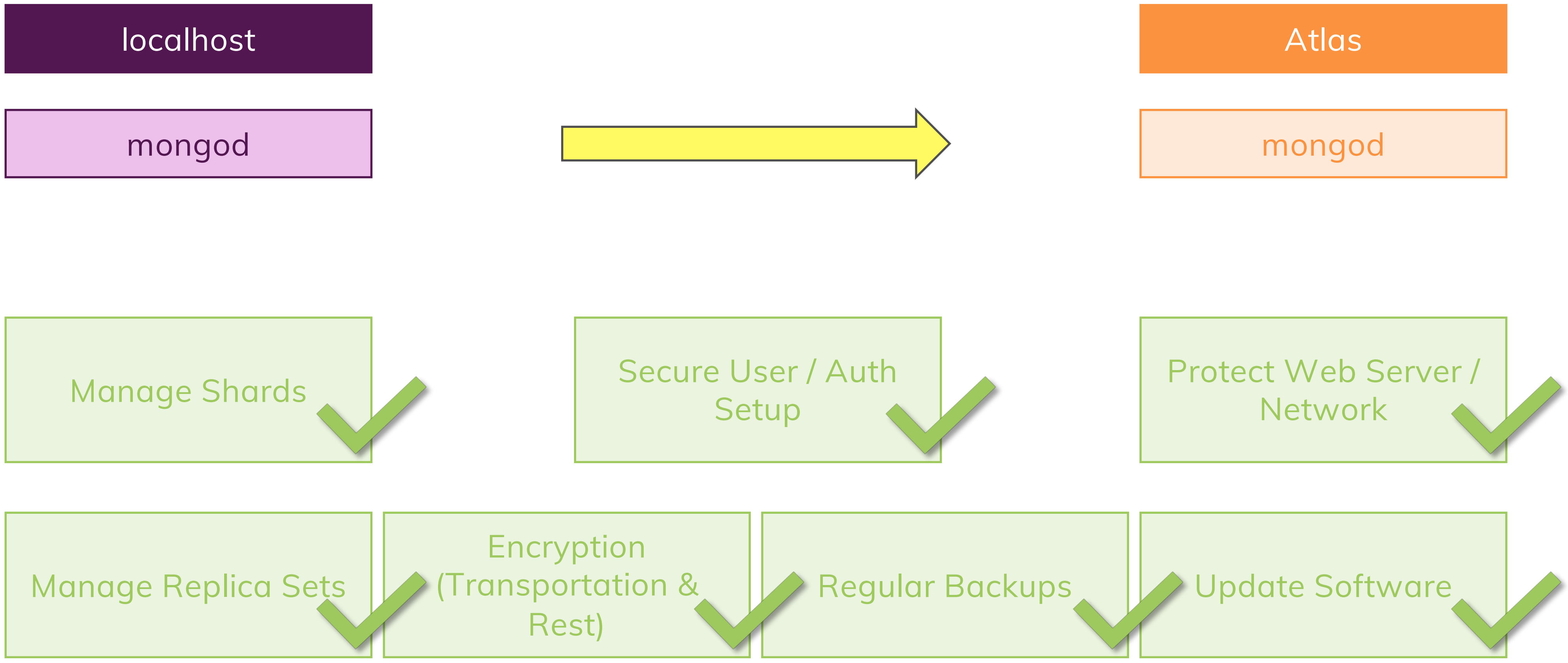
Manage Replica Sets

Encryption (Transportation & Rest)

Regular Backups

Update Software

MongoDB Atlas is a Managed Solution



ACADE MIND

Module Summary

Performance & Fault Tolerancy

- Consider Capped Collections for cases where you want to clear old data automatically
- Performance is all about having efficient queries/ operations, fitting data formats and a best-practice MongoDB server config
- Replica sets provide fault tolerancy (with automatic recovery) and improved read performance
- Sharding allows you to scale your MongoDB server horizontally

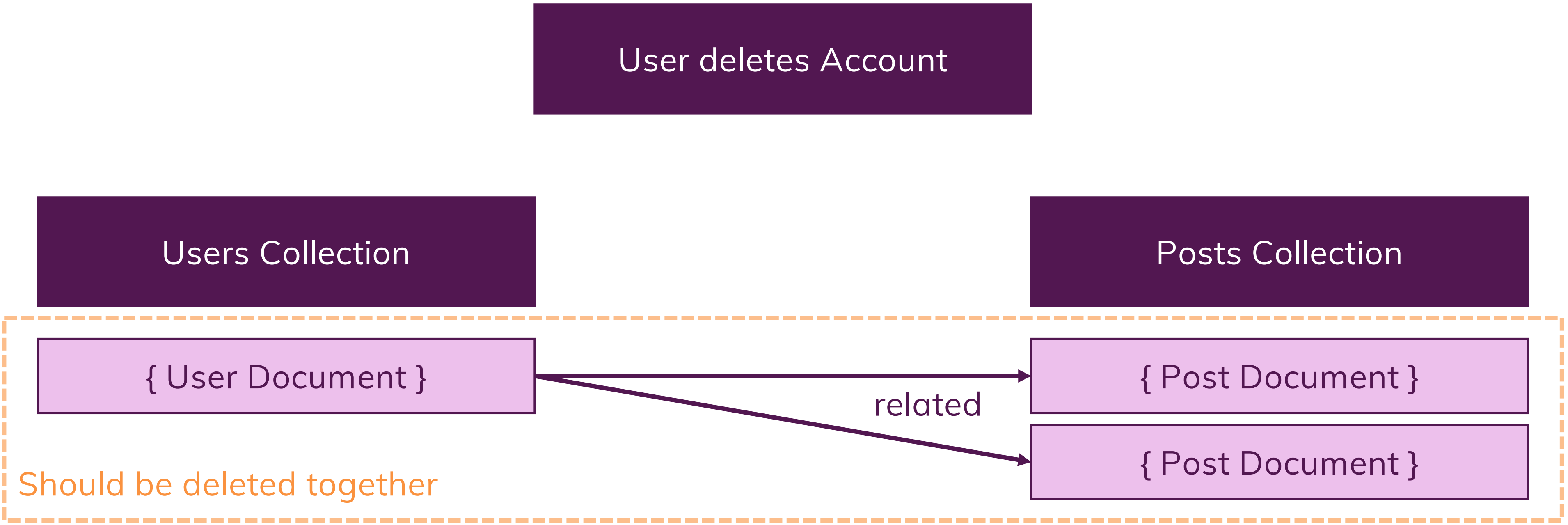
Deployment & MongoDB Atlas

- Deployment is a complex matter since it involves many tasks – some of them are not even directly related to MongoDB
- Unless you are an experienced admin (or you got one), you should consider a managed solution like MongoDB Atlas
- Atlas is a managed service where you can configure a MongoDB environment and pay at a by-usage basis

Transactions

Fail Together

Transactions



From Mongo Shell to Drivers

Writing Application Code

What’s Inside This Module?

How translate “Shell Commands” to “Driver Commands”

Connecting to MongoDB Servers

CRUD Operations

Splitting Work between Drivers & Shell

Shell

- Configure Database
- Create Collections
- Create Indexes

Driver

- CRUD Operations
- Aggregation Pipelines

MongoDB Stitch

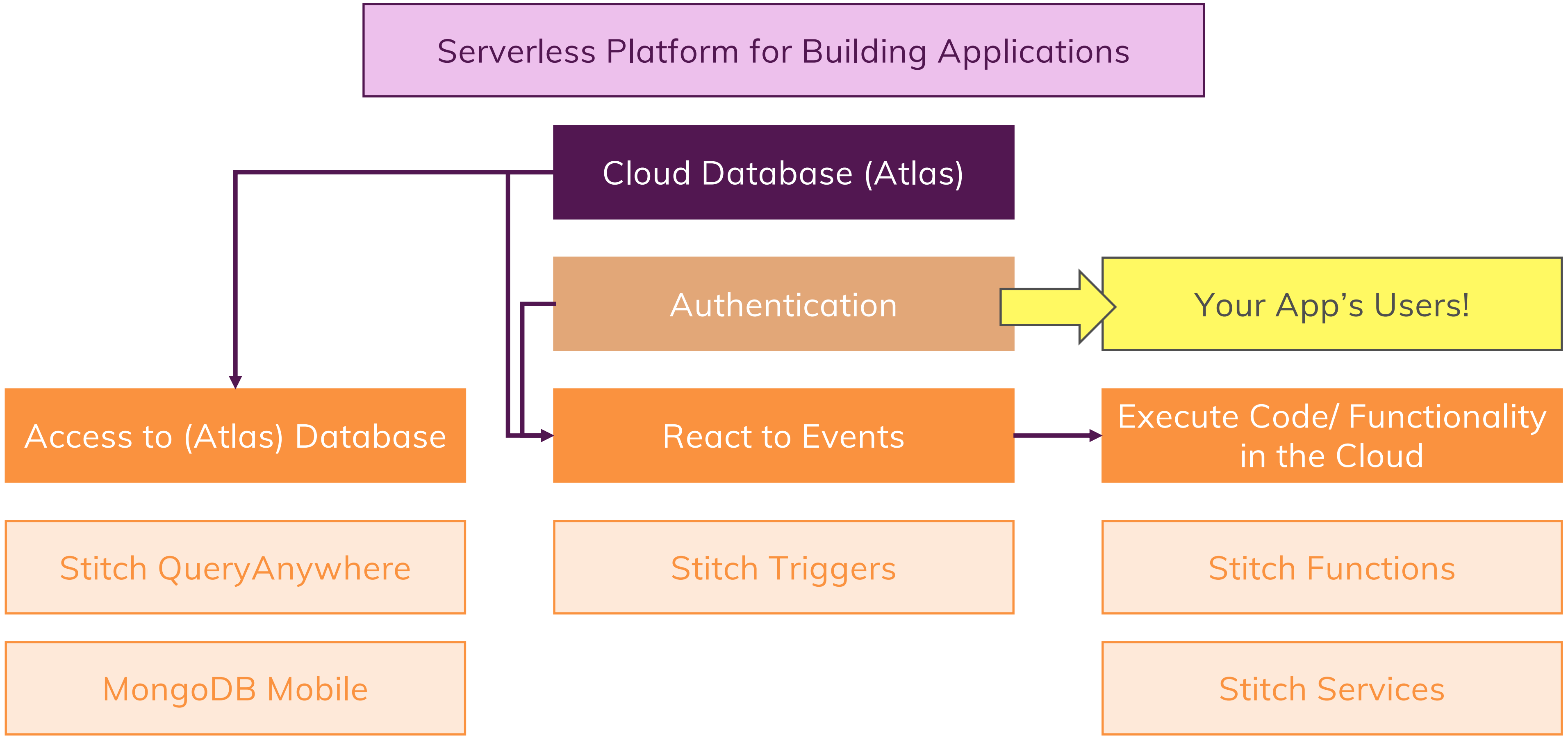
Beyond Data Storage

What’s Inside This Module?

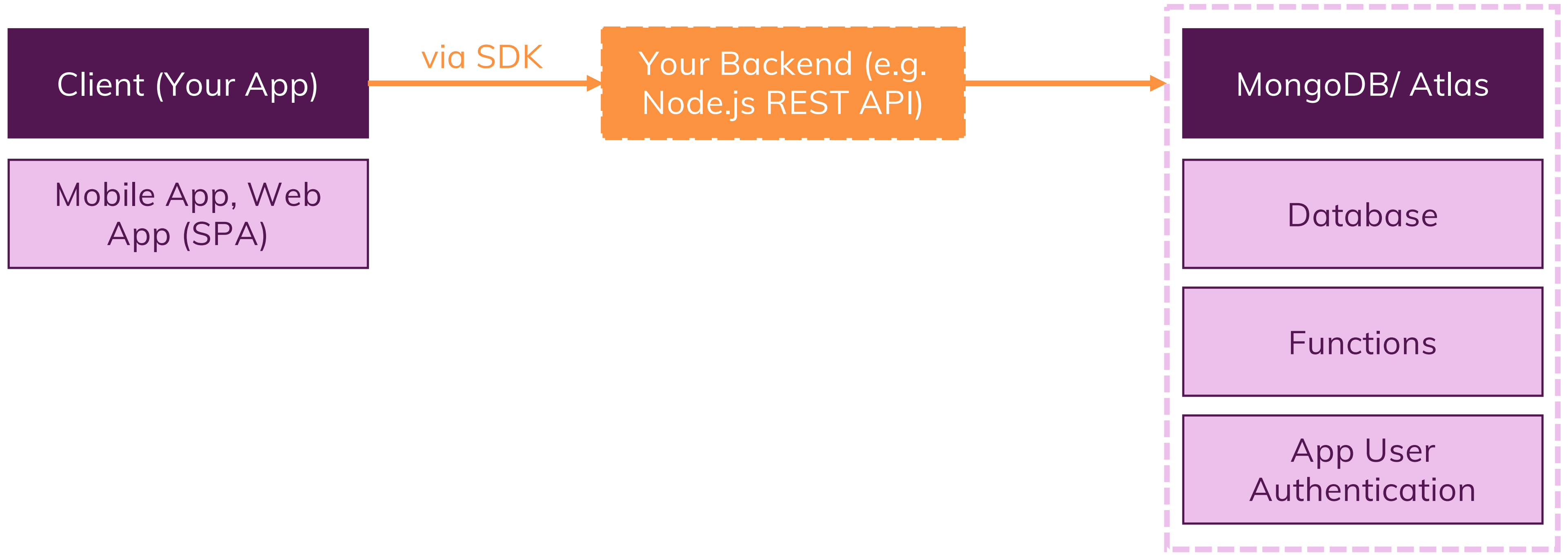
What is Stitch?

Using Stitch

What is Stitch?



Serverless?


Client (Your App)
Mobile App, Web App (SPA)
via SDK
Your Backend (e.g. Node.js REST API)
MongoDB/ Atlas
Database
Functions
App User Authentication

Stitch Authentication vs MongoDB Authentication

Stitch Authentication	MongoDB Authentication
MongoDB stores + manages your Application Users	Your create + manage Database Users
Signup + Login via Stitch SDK	Login during Connection
No Credentials have to be exposed in Clients	MongoDB Credentials have to be exposed => Not usable in Clients
Highly Granular Permissions	Role-based Permissions

ACADE
MIND

Roundup & Next Steps

What Next?

Play Around!

Practice, Practice, Practice

Use the Shell as a Playground

Build Dummy/ Demo Apps that use MongoDB

Build Dummy/ Demo Apps that use MongoDB + Stitch

Dive into the Official Docs

Dive into Stackoverflow + Blog Posts (Google!) to learn Best Practices

Use YouTube + Other Courses to Learn more about Specific Topics

Resources