DF300

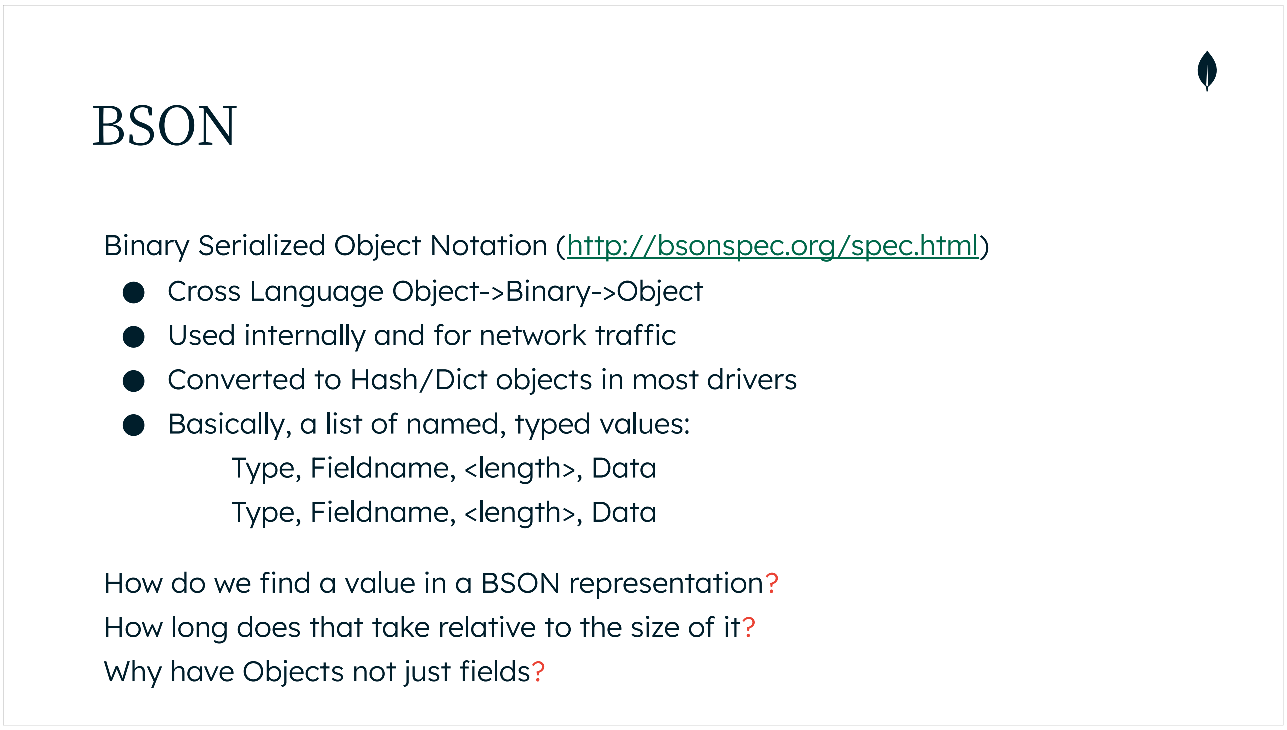
Schema Design

Design Skills and Advanced Features

Release: 20230414

Topics we cover

- Container Types - Arrays vs Documents
- Document model vs Relational model
- Document design fundamentals
- Linking models
- Payload and Process fields
- Dynamic schema
- Schema Design patterns



BSON

Binary Serialized Object Notation (http://bsonspec.org/spec.html)

- Cross Language Object->Binary->Object
- Used internally and for network traffic
- Converted to Hash/Dict objects in most drivers
- Basically, a list of named, typed values:

  Type, Fieldname, <length>, Data

  Type, Fieldname, <length>, Data

How do we find a value in a BSON representation?

How long does that take relative to the size of it?

Why have Objects not just fields?

- BSON is a format in which zero or more ordered key/value pairs are stored as a single entity
- We call this entity a document
- As field names are stored inline - large field names take more storage and cache and are therefore an anti-pattern, prefer compact field names.
- Answers at the end

Container Types

From the BSON Spec

```
document    ::= int32 e_list "\x00"
e_list  ::= element e_list
element ::= "\x01" e_name double    64-bit binary floating
point
        |   "\x02" e_name string    UTF-8 string
        |   "\x03" e_name document  Embedded document
        |   "\x04" e_name document  Array
```

Are Arrays and Embedded Documents both just types of document then?

BSON Document. int32 is the total number of bytes comprising the document

Arrays vs. Documents

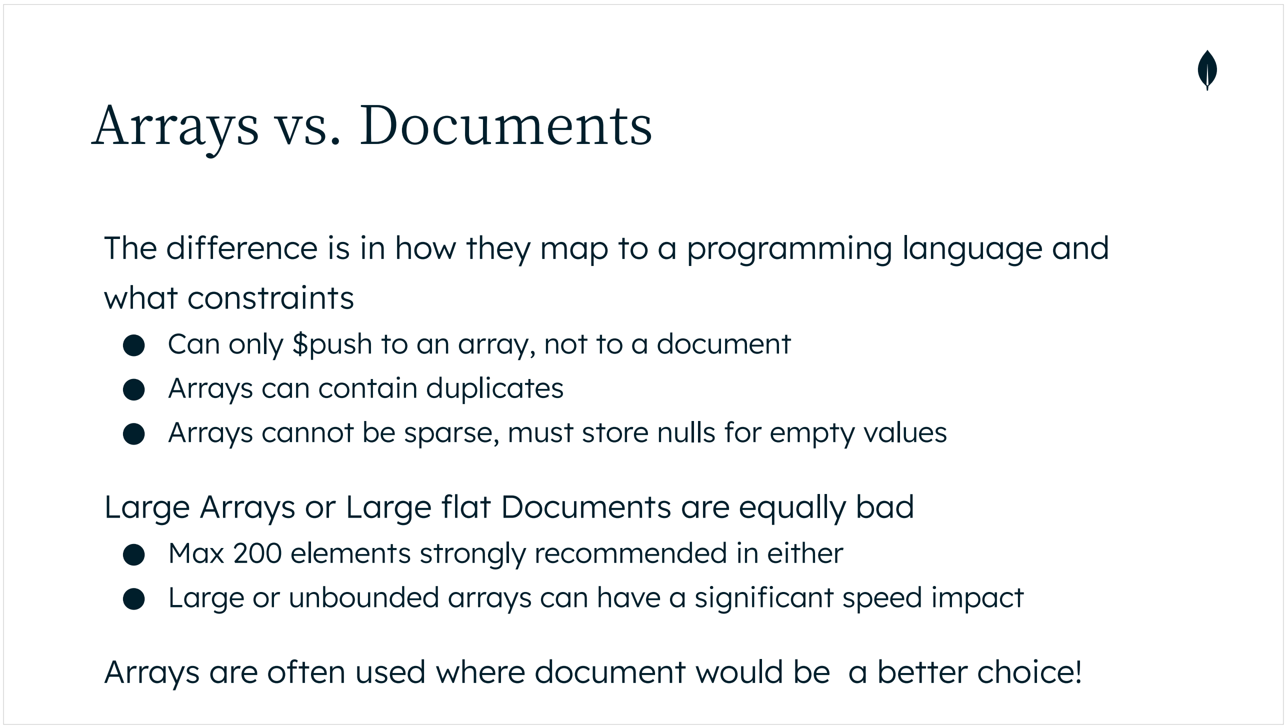
Arrays and documents differ by one byte – storage is the same

There are no Arrays, just Objects

There are no Objects, Just lists of fields, types, and values

```
{ 0 : "Red",

  1 : "Orange",

  2 : "Yellow" }

[ "Red", "Orange", "Yellow" ]
```

Arrays are stored as BSON documents with integer values for the keys, starting with 0 and continuing sequentially.



Arrays vs. Documents

The difference is in how they map to a programming language and what constraints

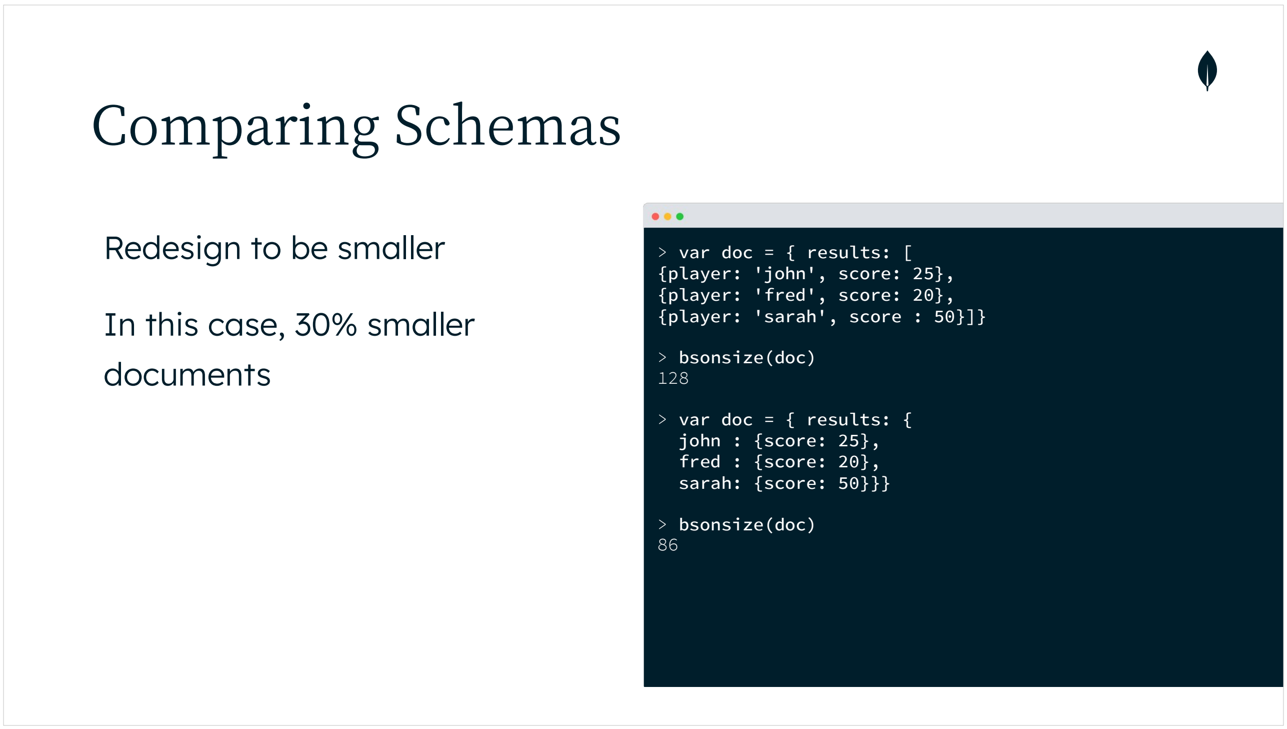
- Can only $push to an array, not to a document
- Arrays can contain duplicates
- Arrays cannot be sparse, must store nulls for empty values

Large Arrays or Large flat Documents are equally bad

- Max 200 elements strongly recommended in either
- Large or unbounded arrays can have a significant speed impact

Arrays are often used where document would be a better choice!

- Arrays and documents should not exceed 200 elements - try hard to stay under at least don't go a long way over.
  - As an array continues to grow, reading and building indexes on that array gradually decrease in performance. A large, growing array can strain application resources and put your documents at risk of exceeding the BSON Document Size limit.
- Sometimes it is better to use a document instead of an array.



Comparing Schemas

Redesign to be smaller

In this case, 30% smaller documents

```
> var doc = { results: [
{player: 'john', score: 25},
{player: 'fred', score: 20},
{player: 'sarah', score : 50}]}

> bsonsize(doc)
128

> var doc = { results: {
  john : {score: 25},
  fred : {score: 20},
  sarah: {score: 50}}}

> bsonsize(doc)
86
```

- Could we make that smaller still?
- Notably, the first is a fixed schema; the second is dynamic/schemaless
- The second would require a wildcard index if we wanted to be able to search it - and that would be larger as it contains the keys as well as values.
- One consideration we note here is that the second manner of document creation, would assume that we know the user's name whom we are trying to get the score of.

Note: If we use the legacy mongo shell, we must use `Object.bsonsize()` method instead of just `bsonsize()`

When to use Arrays

Use Arrays when

- Just need to $push to the end
- Don’t have unique identifiers
- Sparse data is not an issue
- Need to index the values to search them

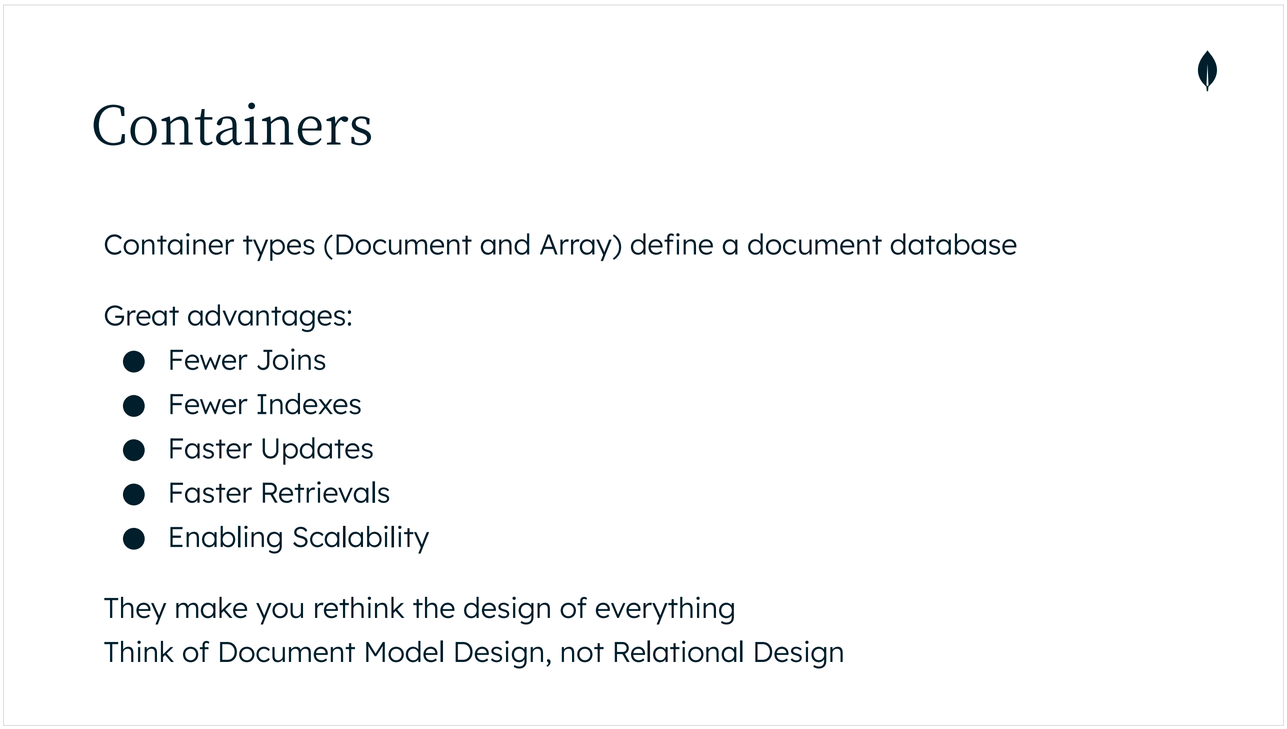
Otherwise, consider objects

- Projection is faster than $filter
- Querying a single member is faster

One big advantage of Arrays over Joined tables is not needing an indexed (foreign) key for each element

One advantage of using an Array to store a number of sub elements - versus putting them in their own collection and joining them is at the very least you don't need the _id value and index on _id you would have in the other collection.

Storing small things like sensor readings or bank transactions (one per document in MongoDB) would likely be very inefficient, doubly so as _id is technically a secondary index and we not only have to store a value for _id (technically we could be clever about that and use a value we were storing anyway) but we need to add an index too. Arrays let us store multiple values without needing an _id value for each element or having to store keys to link records.



Containers

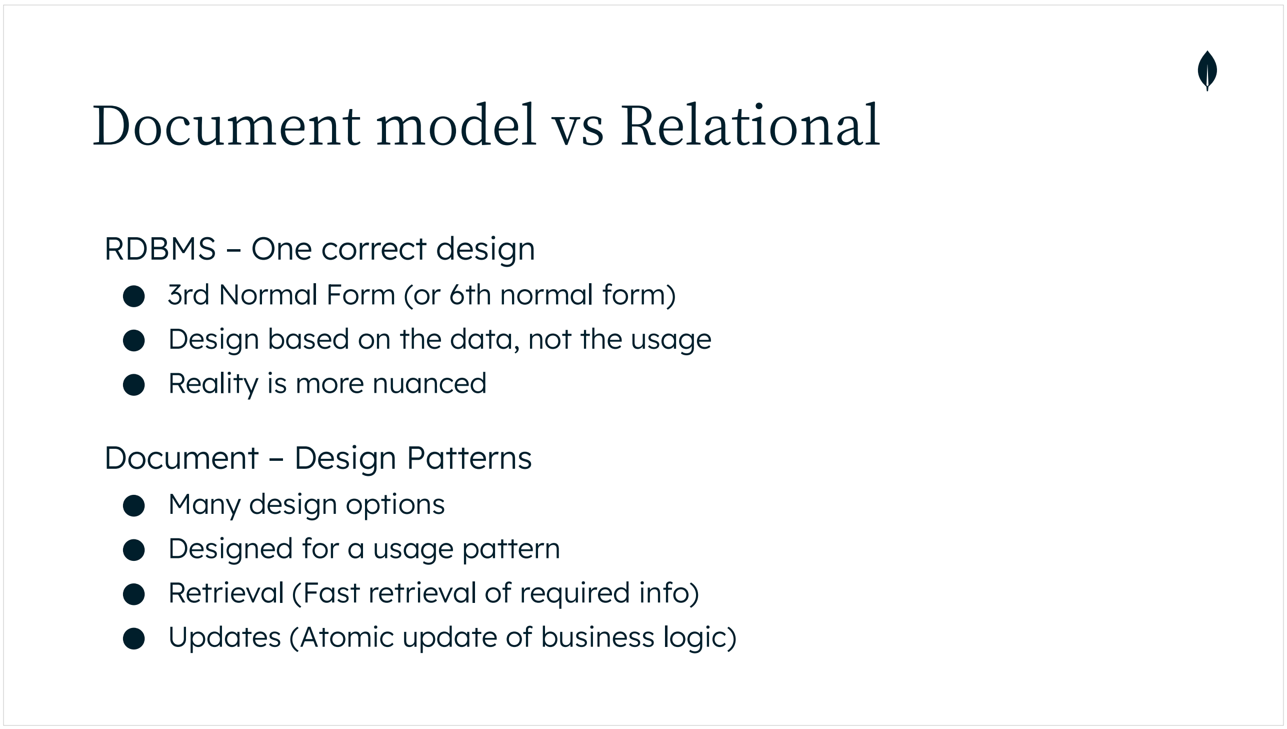
Container types (Document and Array) define a document database

Great advantages:

- Fewer Joins
- Fewer Indexes
- Faster Updates
- Faster Retrievals
- Enabling Scalability

They make you rethink the design of everything
Think of Document Model Design, not Relational Design

Documents and Arrays are known as container types.



Document model vs Relational

RDBMS – One correct design

- 3rd Normal Form (or 6th normal form)
- Design based on the data, not the usage
- Reality is more nuanced

Document – Design Patterns

- Many design options
- Designed for a usage pattern
- Retrieval (Fast retrieval of required info)
- Updates (Atomic update of business logic)

Document design targets fast data retrieval and updates rather than normalization

Exercise - Histogram

Keep track of what time of day things happen, getting events that include a local time in minutes 0-1440 and create a histogram of events occurring for each minute in the day. Increment the value of a specific minute by one when an events occurs in that minute.

```
a:[5,10,15,3,50 … ]
```

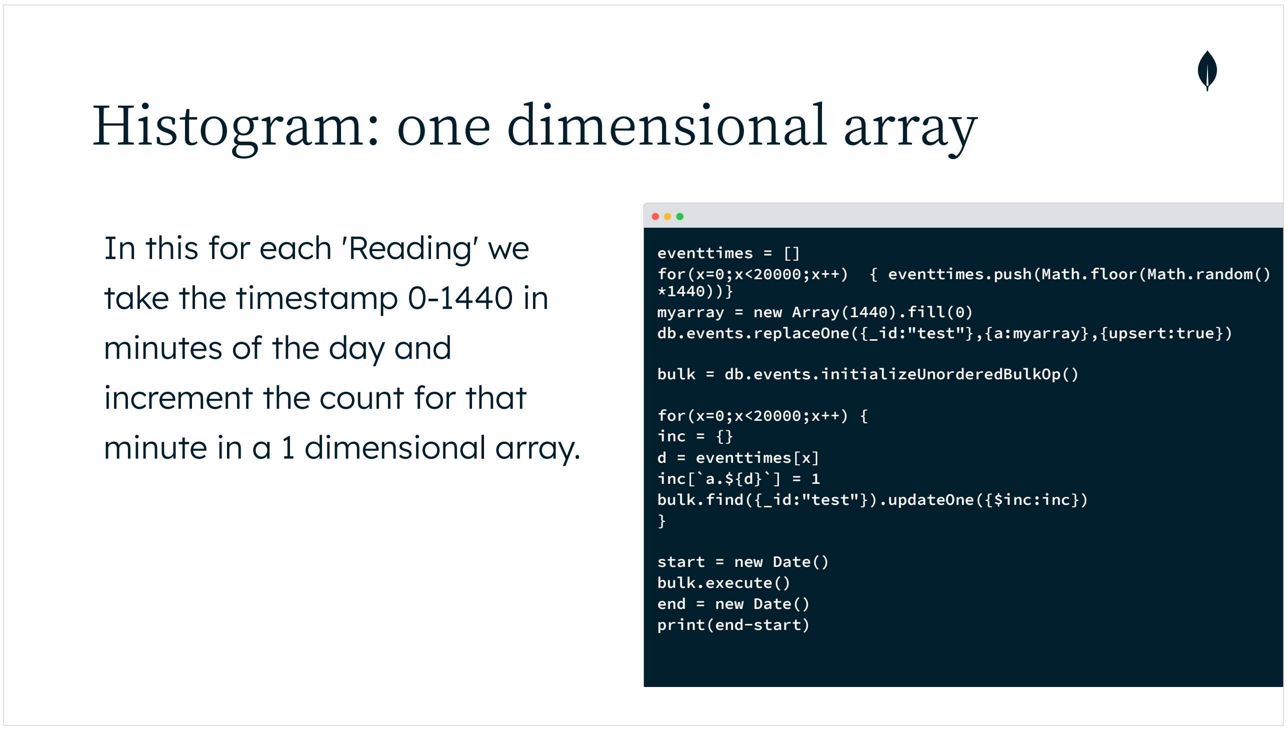
See how long it takes

Now, what if we record in a 2d array hours and minutes [0-23][0-59] - how fast is that?

Read and run the code on the next two slides and compare the performance of one large array versus a 2d array when updating values.

Note how we need to use bulk operations, especially as this is one thread.

Exercise
Initialize the array with zeros for the number of elements we expect... which is? :-)



Histogram: one dimensional array

In this for each 'Reading' we take the timestamp 0-1440 in minutes of the day and increment the count for that minute in a 1 dimensional array.

```
eventtimes = []
for(x=0;x<20000;x++)  { eventtimes.push(Math.floor(Math.random()
*1440))}
myarray = new Array(1440).fill(0)
db.events.replaceOne({_id:"test"},{a:myarray},{upsert:true})

bulk = db.events.initializeUnorderedBulkOp()

for(x=0;x<20000;x++) {
inc = {}
d = eventtimes[x]
inc[`a.${d}`] = 1
bulk.find({_id:"test"}).updateOne({$inc:inc})
}

start = new Date()
bulk.execute()
end = new Date()
print(end-start)
```

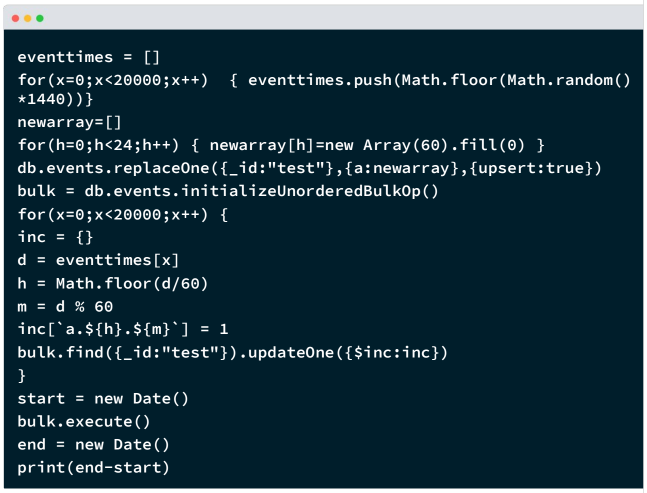
In this for each 'Reading' we take the timestamp 0-1440 in minutes of the day and increment the count for that minute in a 1 dimensional array.

```
eventtimes = []
for(x=0;x<20000;x++)  { eventtimes.push(Math.floor(Math.random()*1440))}
myarray = new Array(1440).fill(0)
db.events.replaceOne({_id:"test"},{a:myarray},{upsert:true})

bulk = db.events.initializeUnorderedBulkOp()

for(x=0;x<20000;x++) {
inc = {}
d = eventtimes[x]
inc[`a.${d}`] = 1
bulk.find({_id:"test"}).updateOne({$inc:inc})
}

start = new Date()
bulk.execute()
end = new Date()
print(end-start)
```

Histogram: two dimensional array

In this for each 'Reading' we take the timestamp 0-1440 in minutes of the day , we then compute the hour and minutes within the hour.

Now we increment the count for that minute in a 2 dimensional array of hours and minutes - reducing the access time to get to the correct part of the BSON.



```
eventtimes = []
for(x=0;x<20000;x++)  { eventtimes.push(Math.floor(Math.random()
*1440))}
newarray=[]
for(h=0;h<24;h++) { newarray[h]=new Array(60).fill(0) }
db.events.replaceOne({_id:"test"},{a:newarray},{upsert:true})
bulk = db.events.initializeUnorderedBulkOp()
for(x=0;x<20000;x++) {
inc = {}
d = eventtimes[x]
h = Math.floor(d/60)
m = d % 60
inc[`a.${h}.${m}`] = 1
bulk.find({_id:"test"}).updateOne({$inc:inc})
}
start = new Date()
bulk.execute()
end = new Date()
print(end-start)
```

In this for each 'Reading' we take the timestamp 0-1440 in minutes of the day , we then compute the hour and minutes within the hour.
Now we increment the count for that minute in a 2 dimensional array of hours and minutes - reducing the access time to get to the correct part of the BSON.

We have a 24 x 60 array with a mean access time of 24/2 + 60/2 (42) compared to the previous example with an array of 1440 elements and a mean access time of 1440/2 (720)
This should be much faster to update.

```
eventtimes = []
for(x=0;x<20000;x++)  { eventtimes.push(Math.floor(Math.random()*1440))}
newarray=[]
for(h=0;h<24;h++) { newarray[h]=new Array(60).fill(0) }
db.events.replaceOne({_id:"test"},{a:newarray},{upsert:true})
bulk = db.events.initializeUnorderedBulkOp()
for(x=0;x<20000;x++) {
        inc = {}
        d = eventtimes[x]
        h = Math.floor(d/60)
        m = d % 60
        inc[`a.${h}.${m}`] = 1
        bulk.find({_id:"test"}).updateOne({$inc:inc})
}
start = new Date()
bulk.execute()
end = new Date()
print(end-start)
```

Document Design Fundamentals

Schema Design in MongoDB

Schema is defined at the application-level

- Focus on the needs of the application
- Not the abstract nature of the data

Design is the part of each phase in its lifetime

Schema design should focus on the application and not the data

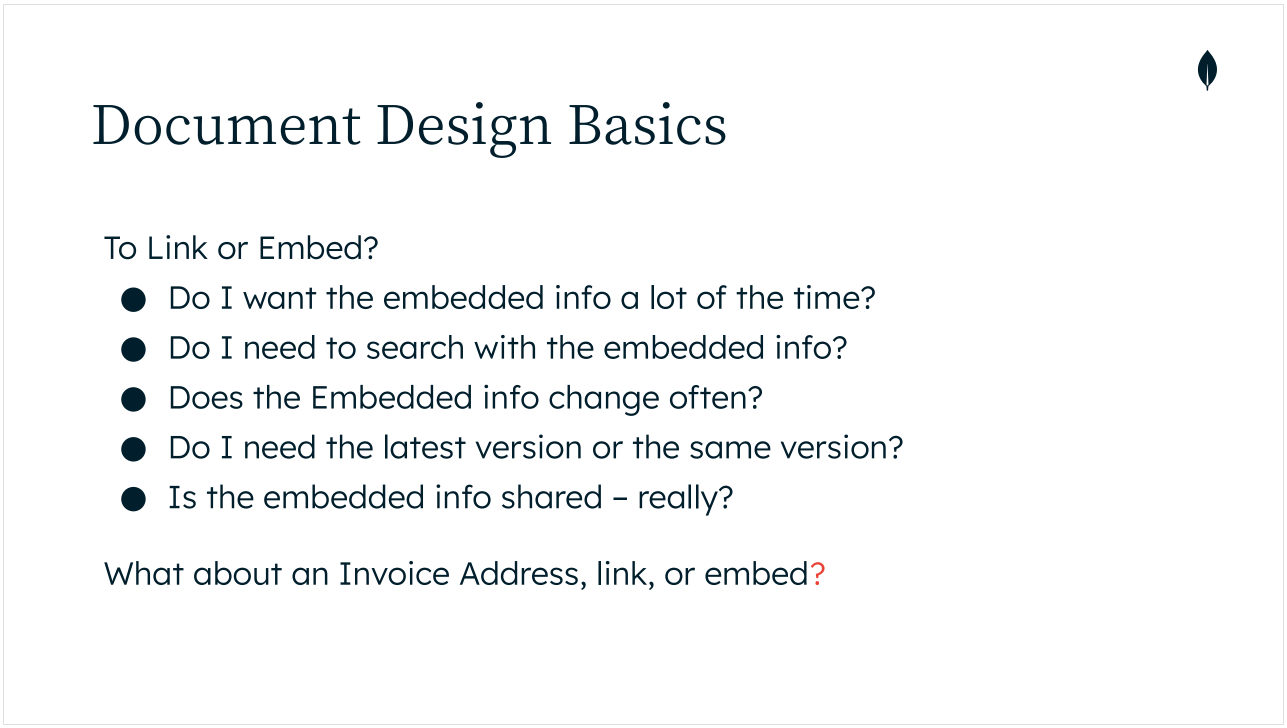
Schema design should focus on the application and not the data.

Three Considerations

The data the application needs

Application’s read usage of the data

Application’s write usage of the data



Document Design Basics

To Link or Embed?

- Do I want the embedded info a lot of the time?
- Do I need to search with the embedded info?
- Does the Embedded info change often?
- Do I need the latest version or the same version?
- Is the embedded info shared – really?

What about an Invoice Address, link, or embed?

Linking or Embedding data are two approaches to consider

Linking models

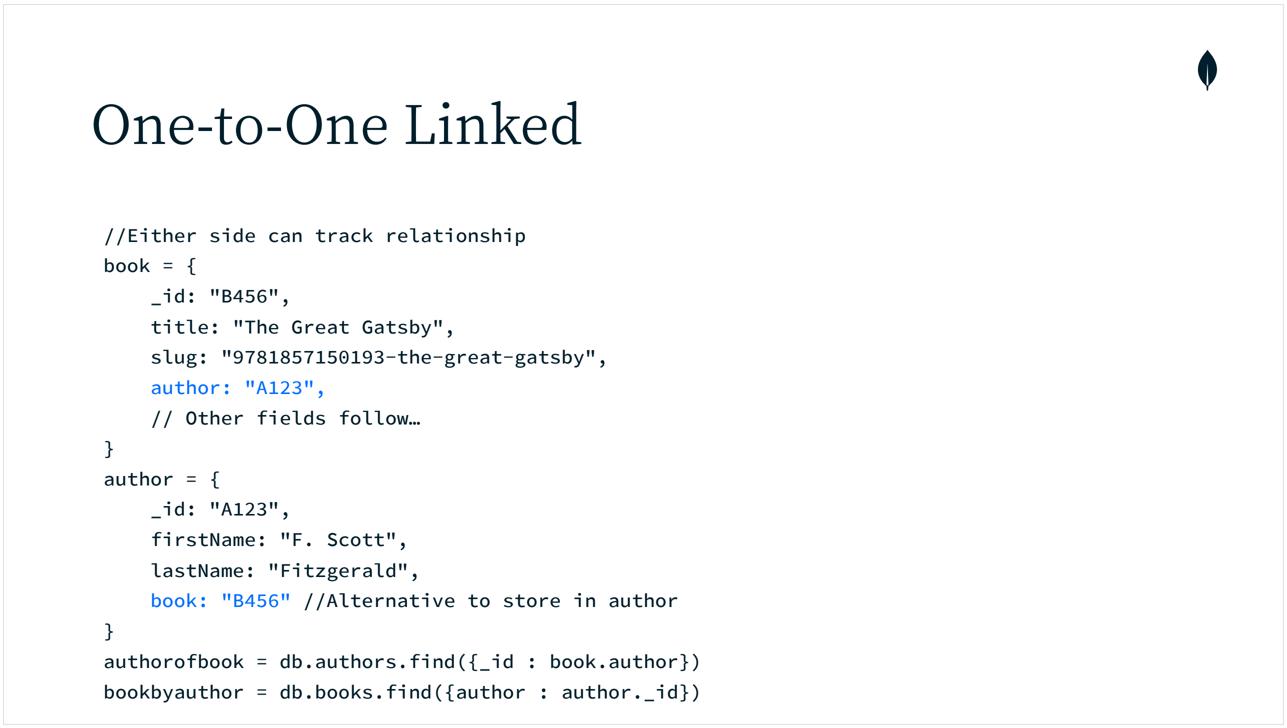
Link types

One-to-One Relationships

One-to-Many Relationships

Many-to-Many Relationships

Link types will be familiar from the relational world

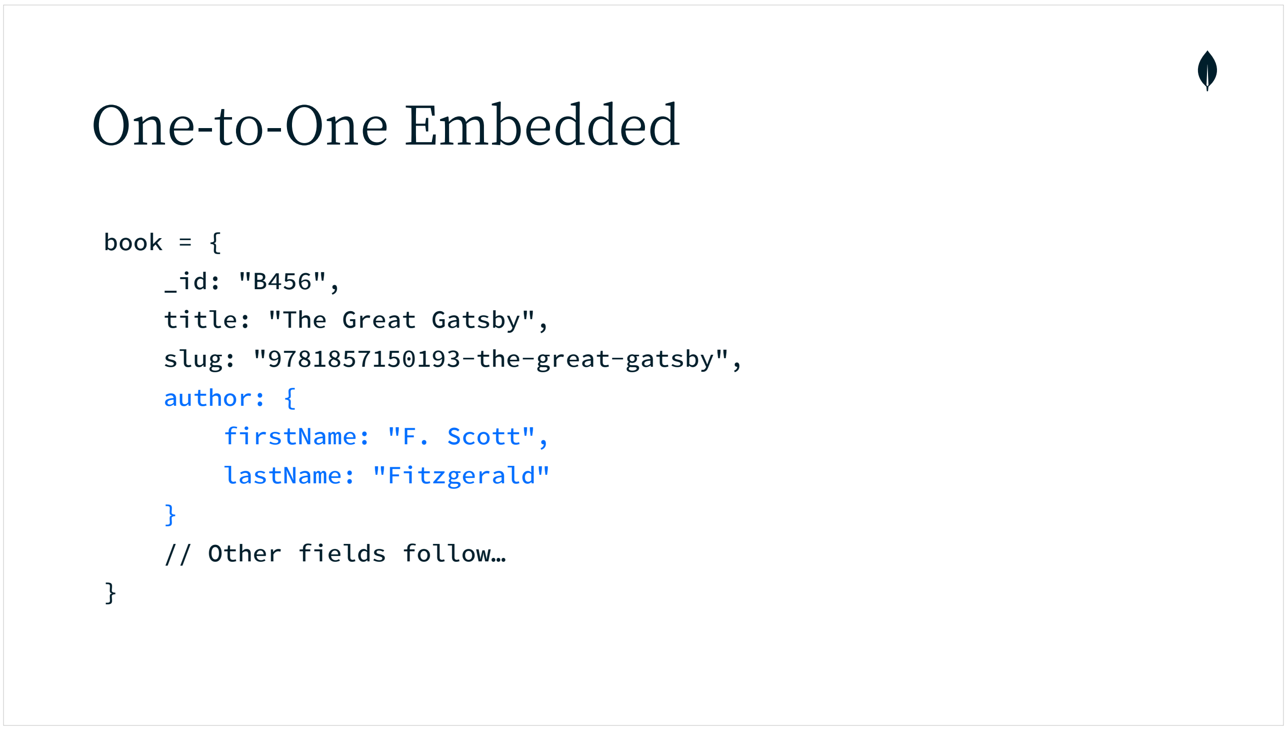


One-to-One Linked

```
//Either side can track relationship
book = {
    _id: "B456",
    title: "The Great Gatsby",
    slug: "9781857150193-the-great-gatsby",
    author: "A123",
    // Other fields follow…
}
author = {
    _id: "A123",
    firstName: "F. Scott",
    lastName: "Fitzgerald",
    book: "B456" //Alternative to store in author
}
authorofbook = db.authors.find({_id : book.author})
bookbyauthor = db.books.find({author : author._id})
```

If you store at both ends you don't need an additional index.

Note: We are assuming in this particular slide that there is just one book stored per author.



One-to-One Embedded

```
book = {
    _id: "B456",
    title: "The Great Gatsby",
    slug: "9781857150193-the-great-gatsby",
    author: {
        firstName: "F. Scott",
        lastName: "Fitzgerald"
    }
    // Other fields follow…
}
```

- Fast to fetch, fast and efficient for query or aggregation
- Less efficient even 1:1 if, say, author info is large and seldom used as the whole document pulled into the cache - not just some fields.
- See later design patterns.

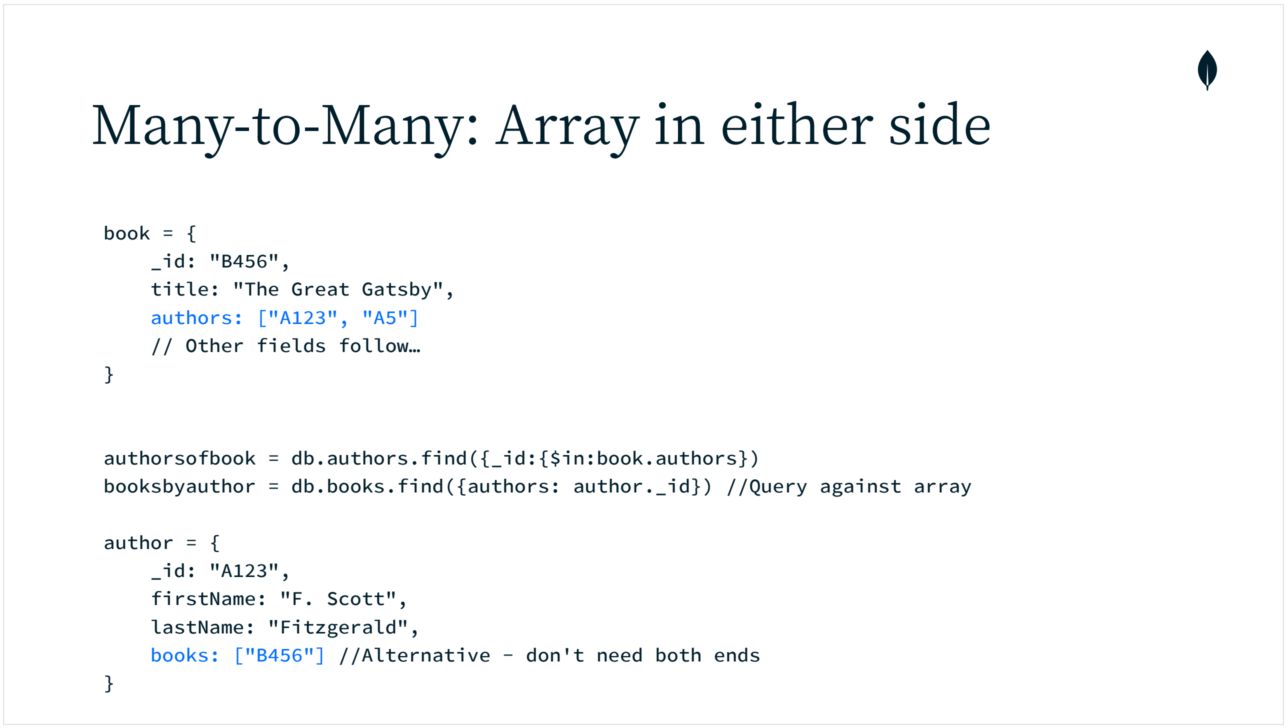
One-to-Many: Array in Parent

```
author = {
    _id: "A123",
    firstName: "F. Scott",
    lastName: "Fitzgerald",
    books: ["B456", "B789", "B20"]
}

booksbyauthor = db.books.find({_id: { $in : author.books}})
authorofbook = db.author.find({books : book._id})
```

This is where arrays become a really powerful tool for data modeling.

One-to-Many: Scalar in Child

```
book1 ={
    _id: "B456",
    title: "The Great Gatsby",
    slug: "9781857150193-the-great-gatsby",
    author: "A123",
    // Other fields follow…
}
book2 = {
    _id: "B789",
    title: "This Side of Paradise",
    slug: "9780679447238-this-side-of-paradise",
    author: "A123",
// Other fields follow…
}
authorofbook = db.authors.find({_id:book.author})
booksbyauthor = db.books.find({author:author._id})
```

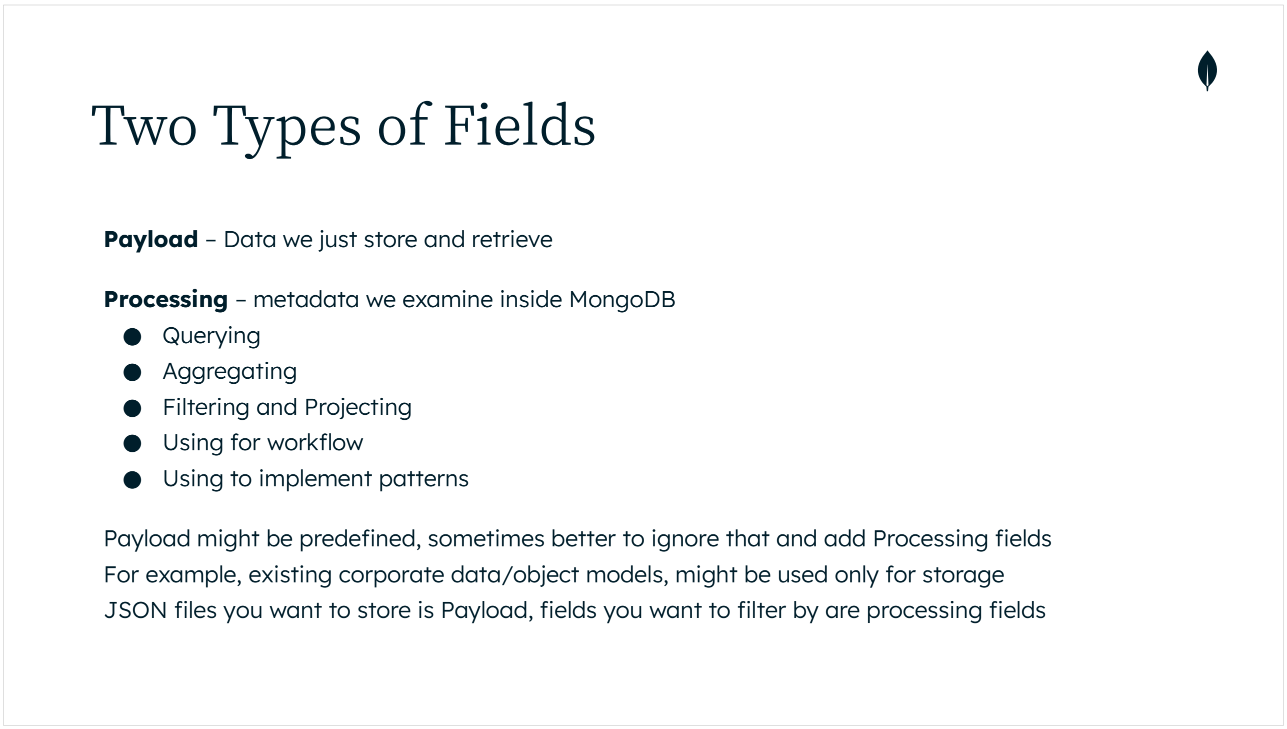


Many-to-Many: Array in either side

```
book = {
    _id: "B456",
    title: "The Great Gatsby",
    authors: ["A123", "A5"]
    // Other fields follow…
}


authorsofbook = db.authors.find({_id:{$in:book.authors}})
booksbyauthor = db.books.find({authors: author._id}) //Query against array

author = {
    _id: "A123",
    firstName: "F. Scott",
    lastName: "Fitzgerald",
    books: ["B456"] //Alternative - don't need both ends
}
```

Arrays can also model many-to-many relationships very easily

Payload fields
vs
Process fields



Two Types of Fields

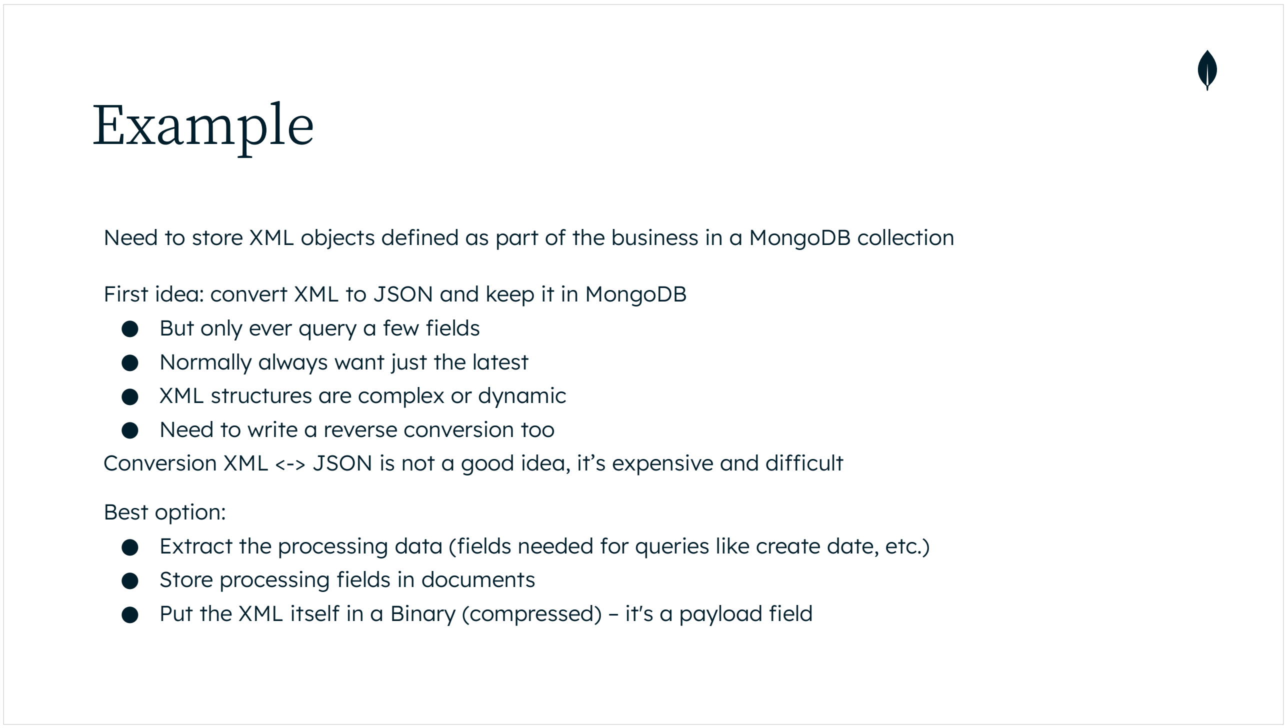
Payload – Data we just store and retrieve

Processing – metadata we examine inside MongoDB

- Querying
- Aggregating
- Filtering and Projecting
- Using for workflow
- Using to implement patterns

Payload might be predefined, sometimes better to ignore that and add Processing fields
For example, existing corporate data/object models, might be used only for storage
JSON files you want to store is Payload, fields you want to filter by are processing fields

- There are two types of terminology we classify fields into:
    - Payload - data we just store and retrieve from the app but the database server never looks at these fields or do anything with them
    - Processing - metadata we examine in MDB- fields the server uses them to query/look at the values
- Corporate data models can be stored in MDB but sometimes traleting to JSON is not necessary if the fields are not processing fields - see example (next slide)



Example

Need to store XML objects defined as part of the business in a MongoDB collection

First idea: convert XML to JSON and keep it in MongoDB

- But only ever query a few fields
- Normally always want just the latest
- XML structures are complex or dynamic
- Need to write a reverse conversion too

Conversion XML <-> JSON is not a good idea, it's expensive and difficult

Best option:

- Extract the processing data (fields needed for queries like create date, etc.)
- Store processing fields in documents
- Put the XML itself in a Binary (compressed) – it's a payload field

Example - Storing XML compressed as binary in MongoDB is usually better than converting it to JSON.

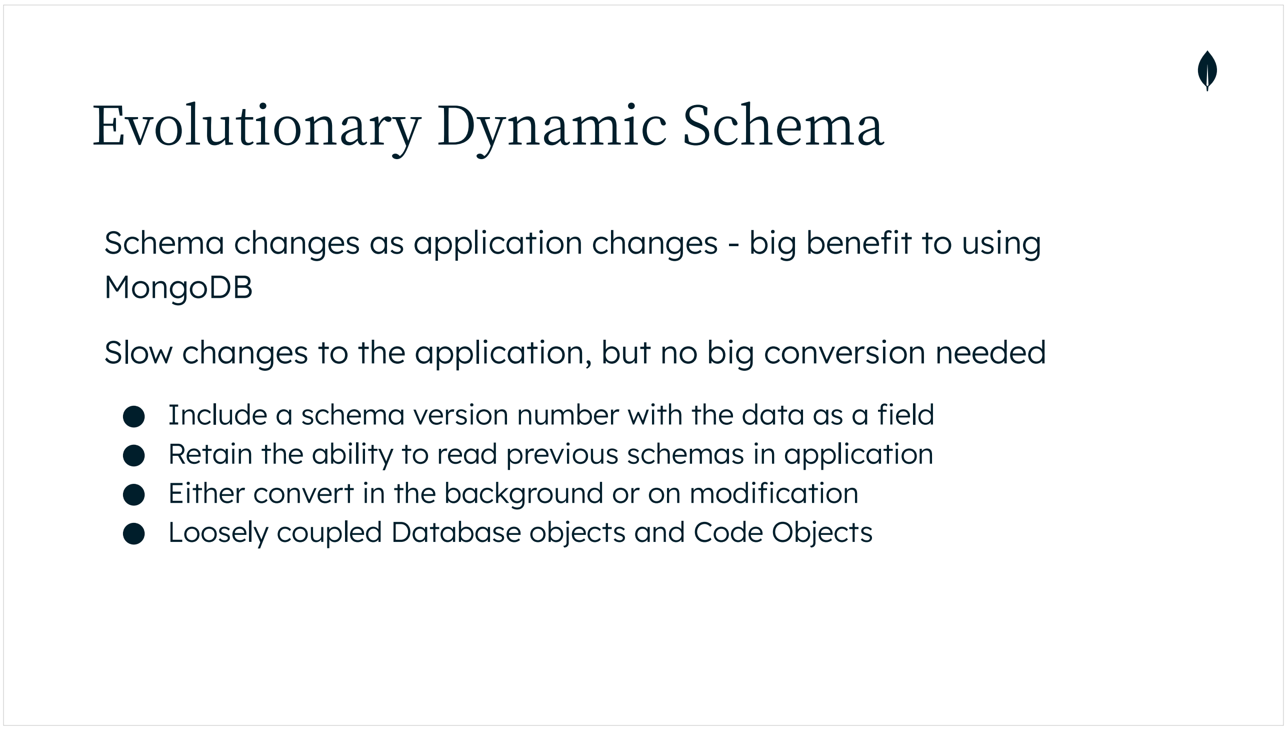
Dynamic Schema

Three Dynamic Models

Evolutionary Dynamic Schema

Payload Driven Dynamic Schema

Data Driven Dynamic Schema



Evolutionary Dynamic Schema

Schema changes as application changes - big benefit to using MongoDB

Slow changes to the application, but no big conversion needed

- Include a schema version number with the data as a field
- Retain the ability to read previous schemas in application
- Either convert in the background or on modification
- Loosely coupled Database objects and Code Objects

Schema versioning can be used to achieve a dynamic schema that changes with your application.

Payload Driven Dynamic Schema

Developer has **no control** over what data goes in – the application's purpose is to store some arbitrary data

Schema is unpredictable

- Cannot optimize much for arbitrary data
- Performance will be poor
- Wildcard indexes can be used but are not a great solution

Think Payload vs Processing:

Can you optimize by using metadata?

Data Driven Dynamic Schema

Field Names are the data values

- Uncomfortable new concept for many designers
- Requires a truly dynamic coding approach
- Clean and performant for many things
- Can be used to do some amazing optimization of retrieval using Tries

Very simple example:

```
results: {   john: {score: 25},
             fred: {score: 20},
             sarah: {score: 50} }
```

- Imagine a document where you need many thousands of keys - you can have each character be a hierarchy level like {t : {h: { i : { s: 5}}}}
- This can make for optimal internal traversal and retrieval.
- Tries are a data structure based on prefix trees. https://en.wikipedia.org/wiki/Trie

Design Patterns

What Are Design Patterns?

Schema Concepts that meet a specific need

- Analogous to *Design Patterns: Elements of Reusable Object-Oriented Software* book
- Provide generalised solutions to common design problems in the form of templates
- Powerful tool for software developers - Important to apply patterns appropriately

Common Design Patterns

Attribute pattern

Bucket pattern

Computed pattern

Versioning pattern

Subset pattern

Outlier or Overspill pattern

Design patterns provide generalised solutions to common design problems in the form of templates.
They are a powerful tool for software developers. However, it is important to apply patterns appropriately.
Using the incorrect pattern for a situation or applying a design pattern to a trivial solution can overcomplicate the code and lead to maintainability issues.

Attribute Pattern

Used where documents have a number of searchable named attributes

- Not all values may be present in a given record
- Any new record might have a new field name not seen before

What systems might have that?
If each of the documents has different field names, what indexes are defined?
Are we confusing Payload and Processing?

Attribute pattern is common in e-commerce applications.

Attribute Pattern Example

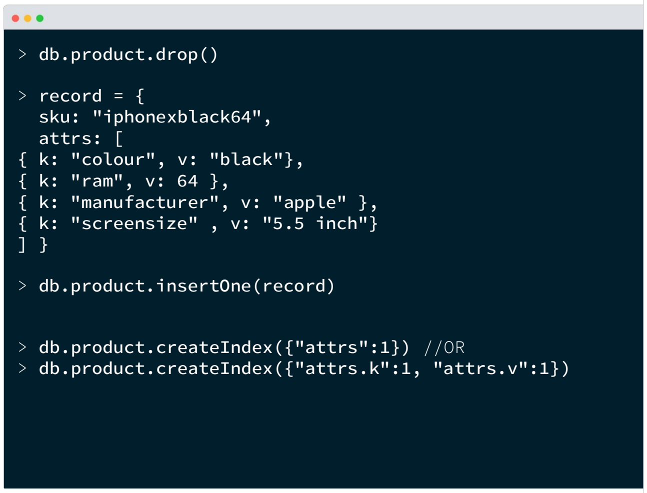
Fixed schema

Array of Objects

Index objects OR Members

Variations exist to optimize some cases

Common in e-commerce



```
> db.product.drop()

> record = {
  sku: "iphonexblack64",
  attrs: [
{ k: "colour", v: "black"},
{ k: "ram", v: 64 },
{ k: "manufacturer", v: "apple" },
{ k: "screensize" , v: "5.5 inch"}
] }

> db.product.insertOne(record)


> db.product.createIndex({"attrs":1}) //OR
> db.product.createIndex({"attrs.k":1, "attrs.v":1})
```

Bucket Pattern

Most common design pattern - Strongly matches Document Model

Used to store many small, related data items

- Bank Transactions – related by account and date
- IoT Readings – related by sensor and date

Reduces index sizes up to 200X!
Speeds up retrieval of related data
Enables computed pattern
MongoDB 5.0 timeseries collections abstract this concept

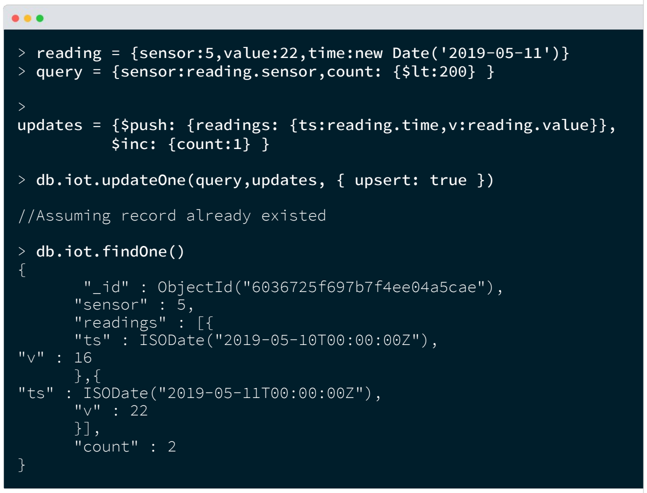
- The Bucket Pattern is a bit like index organized storage in an RDBMS
- Store multiple closely related data points in one document - updating it to add them.

Bucket Pattern Example

Use update with a size limit

Upsert creates new buckets automatically

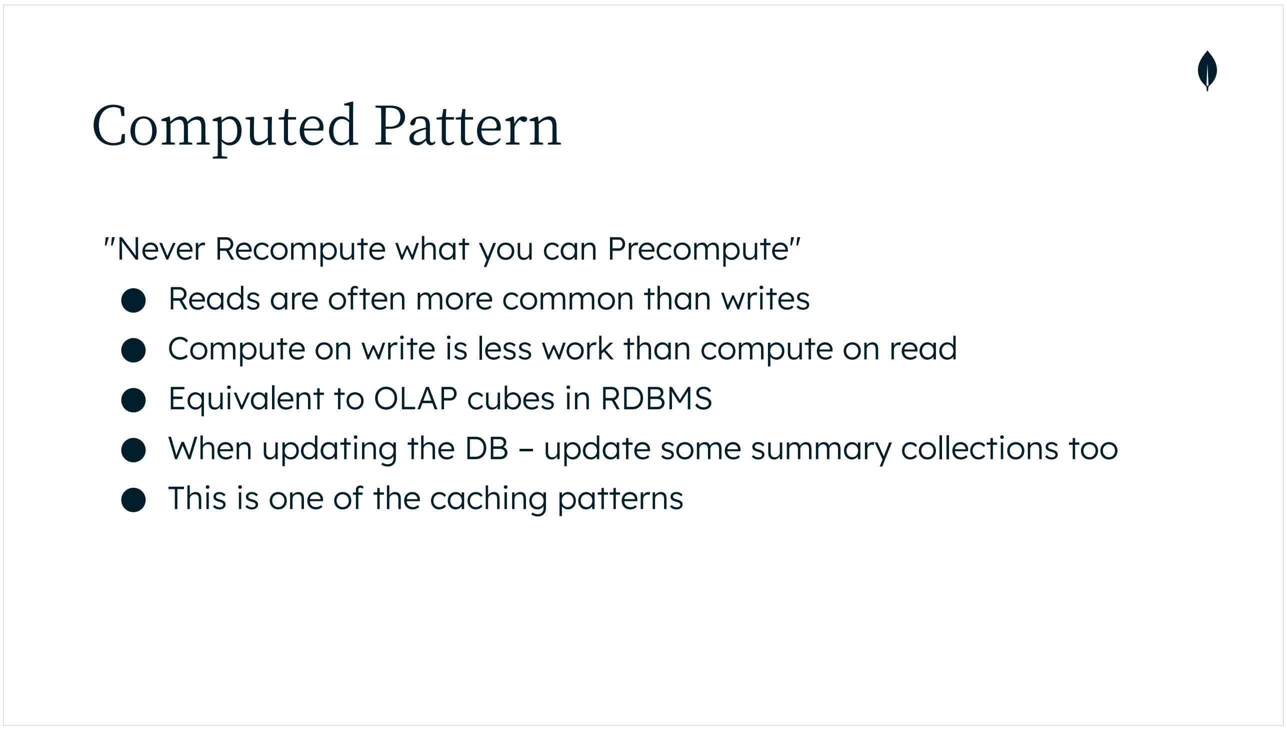
Can bucket on size, timestamp, or both



```
> reading = {sensor:5,value:22,time:new Date('2019-05-11')}
> query = {sensor:reading.sensor,count: {$lt:200} }

>
updates = {$push: {readings: {ts:reading.time,v:reading.value}},
          $inc: {count:1} }

> db.iot.updateOne(query,updates, { upsert: true })

//Assuming record already existed

> db.iot.findOne()
{
       "_id" : ObjectId("6036725f697b7f4ee04a5cae"),
      "sensor" : 5,
      "readings" : [{
      "ts" : ISODate("2019-05-10T00:00:00Z"),
"v" : 16
      },{
"ts" : ISODate("2019-05-11T00:00:00Z"),
      "v" : 22
      }],
      "count" : 2
}
```

Here we start with a reading take from a sensor. Our reading has a value and time it was talken.

We look for a document for that sensor where it currently contains fewer than 200 readings.
If we find one we add the reading to the document and increment the count.
If we don't find one we use upsert to create one.
We assume that we will be retrieving readings over a time period rather than individual readings most of the time.
So we need to index individual readings. In the next pattern we will see how to augment this with metadata.

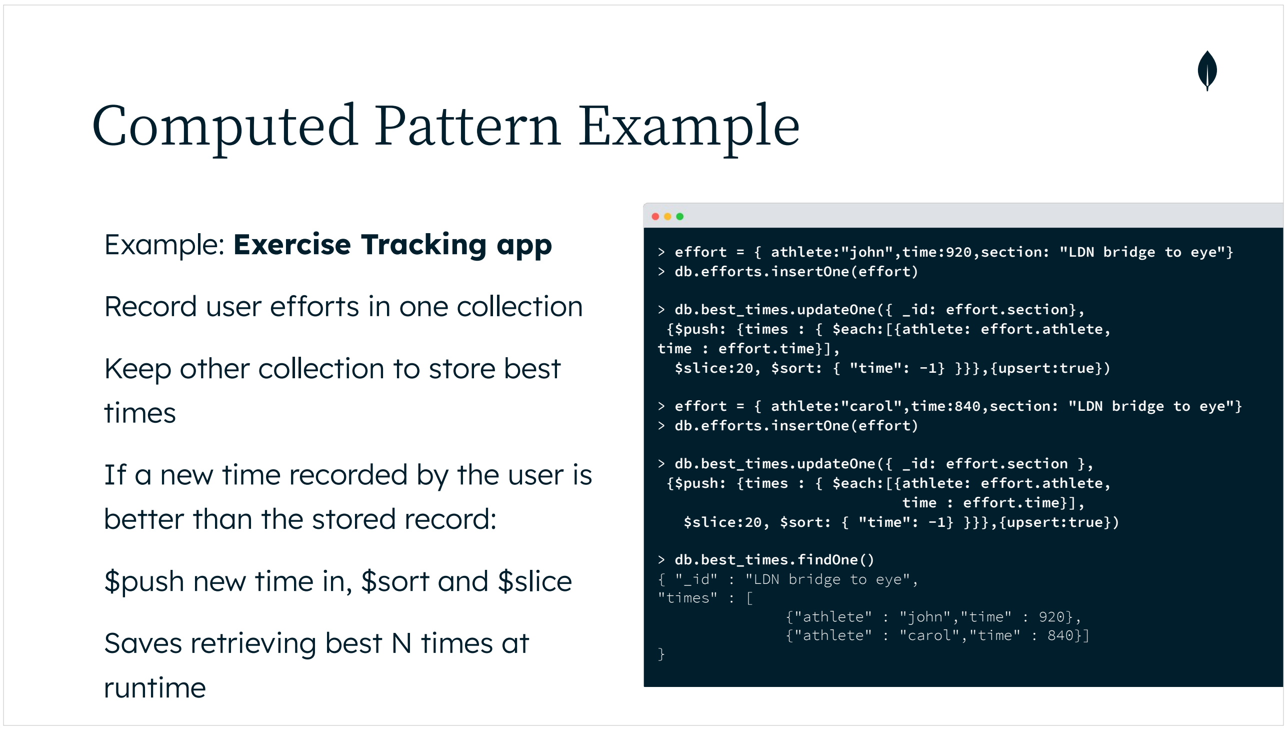


Computed Pattern

"Never Recompute what you can Precompute"

- Reads are often more common than writes
- Compute on write is less work than compute on read
- Equivalent to OLAP cubes in RDBMS
- When updating the DB – update some summary collections too
- This is one of the caching patterns

- Computed Pattern - prevents having to recompute on query
- Instead, compute on update or on a schedule



Computed Pattern Example

Example: **Exercise Tracking app**

Record user efforts in one collection

Keep other collection to store best times

If a new time recorded by the user is better than the stored record:

$push new time in, $sort and $slice

Saves retrieving best N times at runtime

```
> effort = { athlete:"john",time:920,section: "LDN bridge to eye"}
> db.efforts.insertOne(effort)

> db.best_times.updateOne({ _id: effort.section},
 {$push: {times : { $each:[{athlete: effort.athlete,
time : effort.time}],
  $slice:20, $sort: { "time": -1} }}},{upsert:true})

> effort = { athlete:"carol",time:840,section: "LDN bridge to eye"}
> db.efforts.insertOne(effort)

> db.best_times.updateOne({ _id: effort.section },
 {$push: {times : { $each:[{athlete: effort.athlete,
                            time : effort.time}],
   $slice:20, $sort: { "time": -1} }}},{upsert:true})

> db.best_times.findOne()
{ "_id" : "LDN bridge to eye",
"times" : [
               {"athlete" : "john","time" : 920},
               {"athlete" : "carol","time" : 840}]
}
```

With a Computed pattern you keep a separate bit of info computed at write time in the database to avoid computing at read time.

Here we add a runner/riders recorded time to the **efforts** collection in our exercise tracking app.
We update the record in **best_times** for the section they ran/rode if their time is better than any recorded in there.
We add it to the list, sort the list and trim the list to 20 entries with $each and $slice
If the update fails because there is no best_times record for that section yet - it will add one due to upsert

It is possible to optimise this further by checking to see if either their time is faster than one of the times in the record or the record has less than 20 entries.
If you do this then mongodb does not need to perform the editing part as the find fails, if it fails with the upsert clause then it will try to add a record
But that will then fail with a duplicate value of _id - this is an example of having to think about whether errors really are errors.

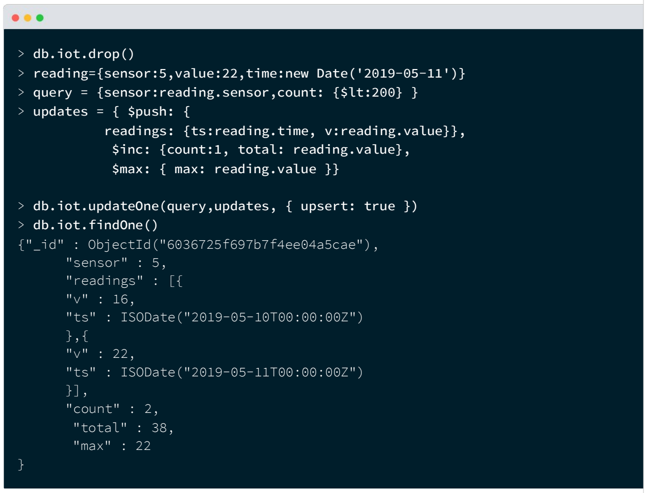
Computed fields - bucket

Add data summarising the bucket

No additional write costs

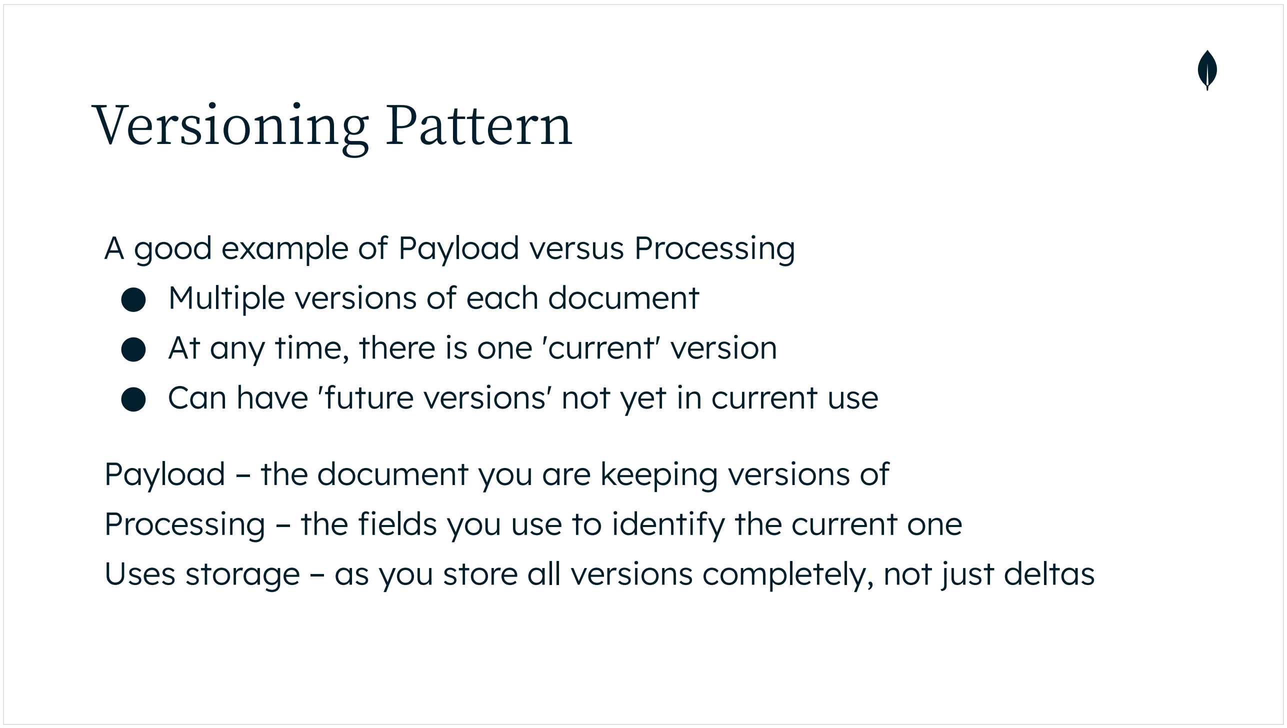
Huge savings in read time computation

Add totals (and mean), Max, Min, and even distribution of value histograms for free



```
> db.iot.drop()
> reading={sensor:5,value:22,time:new Date('2019-05-11')}
> query = {sensor:reading.sensor,count: {$lt:200} }
> updates = { $push: {
           readings: {ts:reading.time, v:reading.value}},
            $inc: {count:1, total: reading.value},
            $max: { max: reading.value }}

> db.iot.updateOne(query,updates, { upsert: true })
> db.iot.findOne()
{"_id" : ObjectId("6036725f697b7f4ee04a5cae"),
      "sensor" : 5,
      "readings" : [{
      "v" : 16,
      "ts" : ISODate("2019-05-10T00:00:00Z")
      },{
      "v" : 22,
      "ts" : ISODate("2019-05-11T00:00:00Z")
      }],
      "count" : 2,
       "total" : 38,
       "max" : 22
}
```

This time when we add our reading, we also update a total and max field to allow us to quickly compute a mean for this or for several buckets.
Or the highest value in this one or several buckets.
We could keep the min and max of the timestamp too and index that to allow us to quickly find all values in a time range.
You could also have an array where you increment array[reading.value] by 1 so the array is a histogram of the frequency of values.



Versioning Pattern

A good example of Payload versus Processing

- Multiple versions of each document
- At any time, there is one 'current' version
- Can have 'future versions' not yet in current use

Payload – the document you are keeping versions of

Processing – the fields you use to identify the current one

Uses storage – as you store all versions completely, not just deltas

The versioning pattern is a good example of payload vs. processing

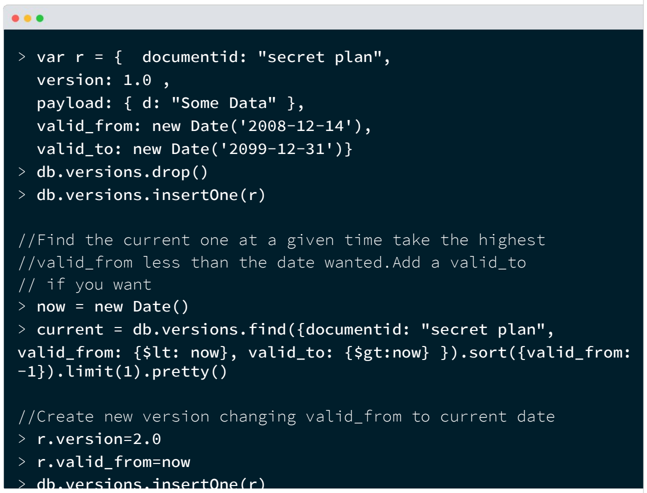
Document Version Pattern 1

A simple `latest_version` field is simple but very limited

Handles publishing future versions

Finding 'all' current needs an aggregation with $sort and $first

Indexing is tricky if using valid_to as well



```
> var r = {  documentid: "secret plan",
  version: 1.0 ,
  payload: { d: "Some Data" },
  valid_from: new Date('2008-12-14'),
  valid_to: new Date('2099-12-31')}
> db.versions.drop()
> db.versions.insertOne(r)

//Find the current one at a given time take the highest
//valid_from less than the date wanted.Add a valid_to
// if you want
> now = new Date()
> current = db.versions.find({documentid: "secret plan",
valid_from: {$lt: now}, valid_to: {$gt:now} }).sort({valid_from:
-1}).limit(1).pretty()

//Create new version changing valid_from to current date
> r.version=2.0
> r.valid_from=now
> db.versions.insertOne(r)
```

Document version pattern 1 - This is good but if you want to find the latest version of more than a single document at at time you need to use $sort, $group and $first in an aggregation.

It would seem like a good idea to add a field latest: true to each document but this has a couple of limitations. First you need to remove the entry from the old record as well as add it to the new one when adding a new "latest" version - this is not a big issue.

More importantly - you want to be able to add a document now that will automatically become the latest version at some future date, you could do that with a scheduled process or trigger that "moves" the latest flag but better to simply make use of a valid from/valid to date combo.

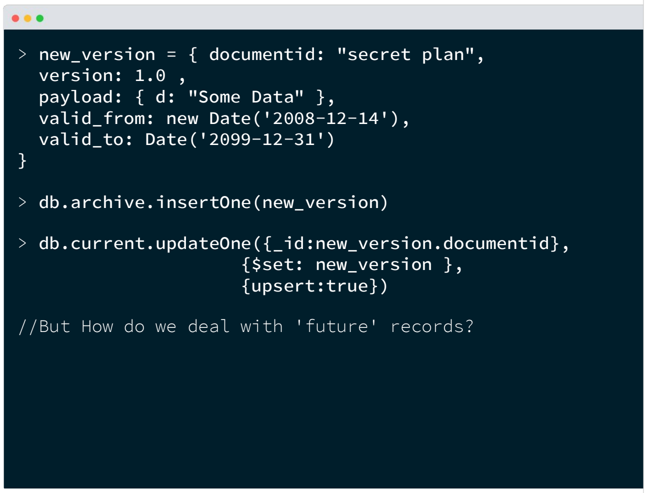
A Query that looks at two ranges to find if a date is inside them {valid_from: { $lt:x}, valid_to: {glt:x}} is difficult to index well as the query must walk the whole index for one of those clauses, generally put the valid_to first as there will be fewer index entries in the future than the past. it will simply need to check all versions that are not expired already to find the first that's valid already.

Document Version Pattern 2

Simpler option - two collections, easier to query current

Dealing with **future records is difficult** - need to be copied to current when ready

Avoid backend batch/scheduled ops where possible in designs



```
> new_version = { documentid: "secret plan",
  version: 1.0 ,
  payload: { d: "Some Data" },
  valid_from: new Date('2008-12-14'),
  valid_to: Date('2099-12-31')
}

> db.archive.insertOne(new_version)

> db.current.updateOne({_id:new_version.documentid},
                       {$set: new_version },
                       {upsert:true})

//But How do we deal with 'future' records?
```

Document version pattern 2

Subset Pattern

Use Linking to another document

Maintain a subset of the linked data embedded for speed

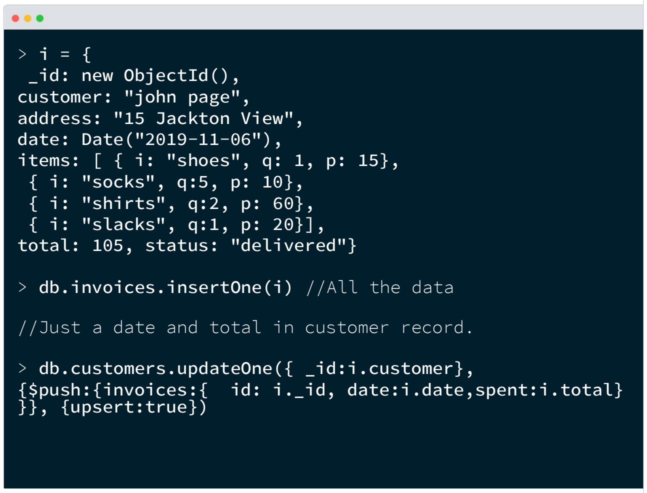
Hybrid of linking and Embedded - common caching pattern:

- Keep an embedded set of the linked data in the main document
- Read the most frequently/recently accessed docs directly from the parent doc
- Can contain only a **subset of all** linked documents or all of them

Subset of all Pattern

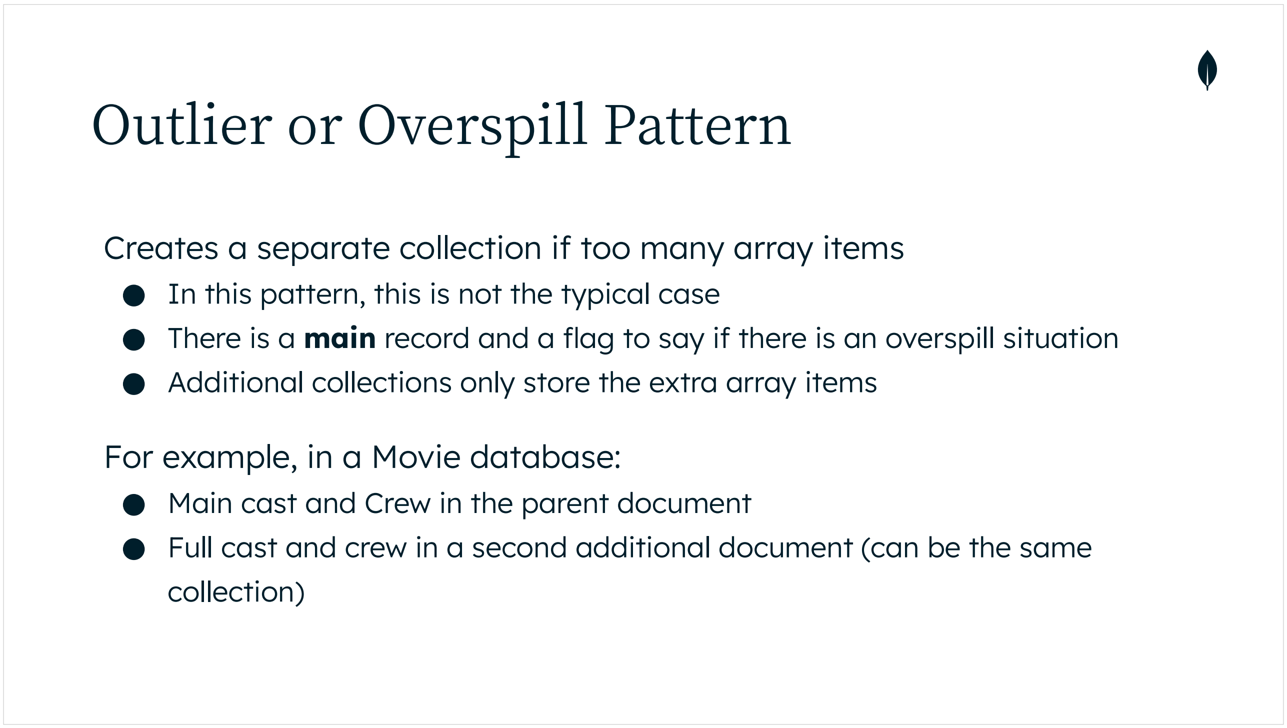
Some data from every linked record

Customer record has just a list of order dates and values



```
> i = {
 _id: new ObjectId(),
customer: "john page",
address: "15 Jackton View",
date: Date("2019-11-06"),
items: [ { i: "shoes", q: 1, p: 15},
 { i: "socks", q:5, p: 10},
 { i: "shirts", q:2, p: 60},
 { i: "slacks", q:1, p: 20}],
total: 105, status: "delivered"}

> db.invoices.insertOne(i) //All the data

//Just a date and total in customer record.

> db.customers.updateOne({ _id:i.customer},
{$push:{invoices:{  id: i._id, date:i.date,spent:i.total}
}}, {upsert:true})
```

Example of subset pattern



Outlier or Overspill Pattern

Creates a separate collection if too many array items

- In this pattern, this is not the typical case
- There is a **main** record and a flag to say if there is an overspill situation
- Additional collections only store the extra array items

For example, in a Movie database:

- Main cast and Crew in the parent document
- Full cast and crew in a second additional document (can be the same collection)

As a standard practice we usually go with a separate collection over the same collection to store outlier data.

Common Design Anti-Patterns

Things that can make the database slower:

- Very large 1 dimensional arrays
- Long field names
- Lots of fields at the same level / flat documents
- Unnecessary indexes - especially on arrays
- Storing seldom used data fields with frequently used data fields
- Using $lookup instead of using embed objects
- Case insensitive queries without case insensitive indexes

Using large arrays (typically over 200 items) means access times get long for items in the array, prefer shorter arrays with limits to their growth.
Field names take storage so avoid using very long field names
Many fields at the same level take longer to find and retrieve than a document with more structure
Each index we have that we don't need increases time to insert and delete data. An unnecessary index on an Array may have many entries which exacerbates the problem.
If we have some data in a business object that is seldom needed and other parts that are frequently needed, consider splitting the object.
Conversely data which is needed together would be in the same object not fetched with $lookup - do not make a relational schema.
If we need a case insensitive query - make sure we have a case insensitive index.

Exercise - Social network

Build a Twitter-style social network where:

- Users have followers
- Users make posts
- Users view posts from those they follow

Questions to answer:

- What collections would you have, when and how would you update them?
- What edge cases do you need to deal with, and how?
- How will you track followers and following counts?
- How can you keep it fast and efficient?

Come up with a schema for a basic Twitter clone and the queries needed to update it. Think about what is run on a reading, a post, a follow, and an unfollow

Exercise

Quiz Time!

#1. Which of the following is true about MongoDB Schema Design

A	Schema design revision is important in bad performing applications
B	Schemas should be designed to support how the data is used
C	Schema design should take into account the datatype of values and how frequently the data is accessed
D	Schema design practice recommends one specific schema for any given patterns
E	Schema design best practice recommends to always embed data as nested objects

Answer in the next slide.

#1. Which of the following is true about MongoDB Schema Design

[✓ A] Schema design revision is important in bad performing applications

[✓ B] Schemas should be designed to support how the data is used

[✓ C] Schema design should take into account the datatype of values and how frequently the data is accessed

D Schema design practice recommends one specific schema for any given patterns

E Schema design best practice recommends to always embed data as nested objects

#2. Benefits that Bucket pattern paired with Computed pattern provide compared to a schema design of one document per measurement

A Reduced RAM requirements

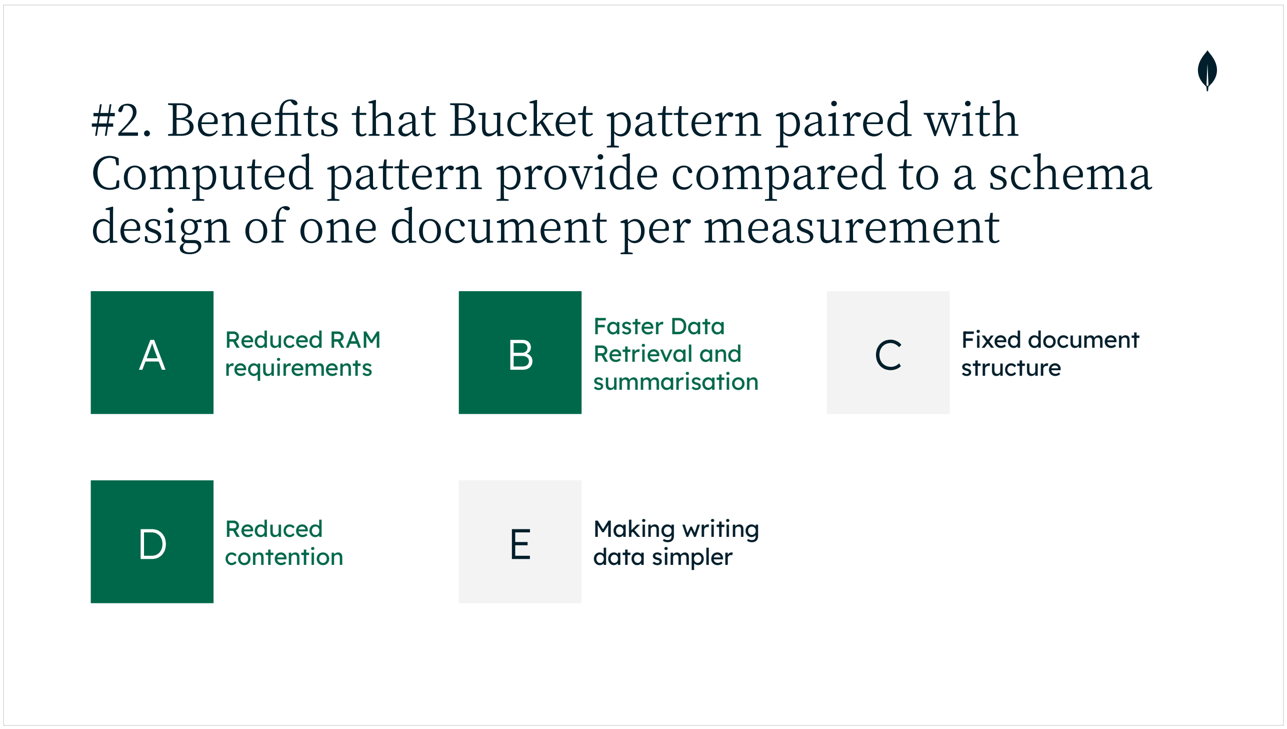
B Faster Data Retrieval and summarisation

C Fixed document structure

D Reduced contention

E Making writing data simpler

Answer in the next slide.



#2. Benefits that Bucket pattern paired with Computed pattern provide compared to a schema design of one document per measurement

[✓ A] Reduced RAM requirements

[✓ B] Faster Data Retrieval and summarisation

C Fixed document structure

[✓ D] Reduced contention

E Making writing data simpler

Answer:A, B, D

Recap

Container Types in a collection define embedded (documents) or linked (arrays) models

Schema design focus on the application needs (Payload vs Process fields)

Design patterns can help on the schema design based on use cases

Different Schema Design patterns are:

Attribute pattern, Bucket pattern, Computed pattern, Versioning pattern, Subset pattern, Outlier or Overspill pattern

Answers

Answer - Questions: BSON

How to find a value in a BSON representation?
Start at the beginning, check the field name and either it's right, and that is the field, or jump over that field (using the length if needs be) and read the next field name. Checking a list of fields until the field is found.

How long does that take relative to the size of it ?
It's proportional to the number of fields you are looking through multiplied by the number you are looking for (some optimizations apply)

Why have Objects not just fields?
An object is a single field with a length - so you can jump the whole object - for example if not looking for address (or a subfield of it) you can skip all address fields.

Answer - Exercise: Social Network

db.users

```
{
     _id: String (userid),
     schemaversion: (integer),
     dateCreated: Date(),
     follows: ArrayOfString
     (people followed) [Indexed]
}
```

social.posts

```
{
      _id: ObjectId(),
      schemaVersion: Int,
      postedBy: String,
      date: Date,
      text: String
}
```

We are modeling followers as who someone follows - not who follows them. The very high cardinality model, and one we need to support is millions follow one - and where this happens following and unfollowing should not make changes to one hot record. Also we don't want arrays of millions. You can't limit how many followers someone has.

On the other hand limiting someone to following no more than a couple of thousand people is a reasonable thing to do and if someone is responsible for data about who they follow and it's deleted with them that seems more in line with GDPR etc. If someone deletes themselves and you follow them we can ignore removing them from your followers list, at least asynchronously. Deletion of a user is less common than changing follow/unfollow.