MongoDB Data Modeling Workshop

MongoDB Data Modeling Workshop

Release 3.4

MongoDB, Inc.

May 26, 2017

Contents

1 MongoDB Data Modeling

1.1 Underlying Reasons for Schema Design

Learning Objectives

Upon completing this module, students will be able to:

- Explain what good schema design minimizes
- Evaluate how different schemas will affect performance

Why Learn Schema Design?

- The structure of data affects performance.
- A good schema can mean the difference between doing many queries, or few.
- It can make the difference between having data in RAM, or touching the disk.
- It can mean the difference between having a data store that scales out, and one that doesn't.

What Affects the Speed of Reading Data?

- A server is limited in its RAM
  - Going to disk is slower
- Queries take time to make a round trip
  - Serial round trips are even worse
- Writing to disk takes time
- Different schemas will produce different bottlenecks

Goal: Minimize your Number of Queries

- Queries take a finite amount of time
  - Especially if you need to make round-trips
- Fewer queries for all the data is a net win
- Good schema design tries to minimize the number of queries
- Data that gets queried together should be stored in the same document

---

Note:

- Round trip example:
  - the application queries the “books” collection and finds a document that includes an “author_id”
  - the application queries for the “authors” collection based on that “author_id” to find out more
  - Improve this by putting all the author information you need into the book

---

Goal: Avoid Touching Data you Won’t Use

- Documents in RAM will get queried faster than documents not in RAM
- The same is true of indexes
- RAM is measured in bytes
- Bytes of data you’re not using will push out bytes of data you are

Goal: Avoid Touching Data you Won't Use (cont'd)

- Don't put data into documents if you won't use it
- Push rarely-used data into a different collection
  - This allows you to store more documents in RAM
- Make use of covered queries
  - The fastest read is the one that can be answered by the index

What to Prioritize

- Prioritize for your most common queries
  - Optimizing these at the expense of others is a net win
- The patterns you see today are ways of prioritizing different operations
- Your schema depends on your access patterns.
- To figure out what you need, start by writing your queries. *Then* arrange the data so that they're answered efficiently.

Summary

- Minimize your number of queries
- Minimize your document size
- Optimize your most common use cases
- Write your queries first to determine what to prioritize

1.2 Schema Design Core Concepts

Note: There are a lot of exercises to chose from. You are likely to only go through a subset of those in the class. Either, you already know which ones are appropriate for the class, or you can ask them to vote.

Some exercises are precedeed by a lot of context and examples of queries/code. Those are labelled "case studies". Otherwise you have the pure exercises that only have the statement of the problem.

Some of the exercises come with the elements of the solution in the instructor notes on the last page.

Learning Objectives

Upon completing this module, students should understand:

- Basic schema design principles for MongoDB
- Tradeoffs for embedded documents in a schema
- Tradeoffs for linked documents in a schema
- The use of array fields as part of a schema design

What is a schema?

- Maps concepts and relationships to data
- Sets expectations for the data
- Minimizes overhead of iterative modifications
- Ensures compatibility

Example: Normalized Data Model

```
User:           Book:           Author:
- username      - title         - firstName
- firstName     - isbn          - lastName
- lastName      - language
                - createdBy
                - author
```

Example: Denormalized Version

```
User:           Book:
- username      - title
- firstName     - isbn
- lastName      - language
                - createdBy
                - author
                  - firstName
                  - lastName
```

Schema Design in MongoDB

- Schema is defined at the application-level
- Design is part of each phase in its lifetime
- There is no magic formula

Three Considerations

- The data your application needs
- Your application's read usage of the data
- Your application's write usage of the data

Case Study

- A Library Web Application
- Different schemas are possible.

Author Schema

```
{   "_id": int,
    "firstName": string,
    "lastName": string
}
```

User Schema

```
{   "_id": int,
    "username": string,
    "password": string
}
```

Book Schema

```
 {  "_id": int,
    "title": string,
    "slug": string,
    "author": int,
    "available": boolean,
    "isbn": string,
    "pages": int,
    "publisher": {
        "city": string,
        "date": date,
        "name": string
    },
    "subjects": [ string, string ],
    "language": string,
    "reviews": [ { "user": int, "text": string },
```

```
                { "user": int, "text": string } ]
}
```

Example Documents: Author

```
{   _id: 1,
    firstName: "F. Scott",
    lastName: "Fitzgerald"
}
```

Example Documents: User

```
{   _id: 1,
    username: "emily@10gen.com",
    password: "slsjfk4odk84k209dlkdj90009283d"
}
```

Example Documents: Book

```
{   _id: 1,
    title: "The Great Gatsby",
    slug: "9781857150193-the-great-gatsby",
    author: 1,
    available: true,
    isbn: "9781857150193",
    pages: 176,
    publisher: {
        name: "Everyman's Library",
        date: ISODate("1991-09-19T00:00:00Z"),
        city: "London"
    },
    subjects: ["Love stories", "1920s", "Jazz Age"],
    language: "English",
    reviews: [
        { user: 1, text: "One of the best..." },
        { user: 2, text: "It's hard to..." }
    ]
}
```

Embedded Documents

- AKA sub-documents or embedded objects
- What advantages do they have?
- When should they be used?

Example: Embedded Documents

```
{   ...
    publisher: {
        name: "Everyman's Library",
        date: ISODate("1991-09-19T00:00:00Z"),
        city: "London"
    },
    subjects: ["Love stories", "1920s", "Jazz Age"],
    language: "English",
    reviews: [
        { user: 1, text: "One of the best..." },
        { user: 2, text: "It's hard to..." }
    ]
}
```

Embedded Documents: Pros and Cons

- Great for read performance
- One seek to find the document
- At most, one sequential read to retrieve from disk
- Writes can be slow if constantly adding to objects

Linked Documents

- What advantages does this approach have?
- When should they be used?

Example: Linked Documents

```
{   ...
    author: 1,
    reviews: [
        { user: 1, text: "One of the best..." },
        { user: 2, text: "It's hard to..." }
    ]
}
```

Linked Documents: Pros and Cons

- More, smaller documents
- Can make queries by ID very simple
- Accessing linked documents requires extra seeks + reads.
- What effect does this have on the system?

Arrays

- Array of scalars
- Array of documents

Array of Scalars

```
{   ...
    subjects: ["Love stories", "1920s", "Jazz Age"],
}
```

Array of Documents

```
{   ...
    reviews: [
        { user: 1, text: "One of the best..." },
        { user: 2, text: "It's hard to..." }
    ]
}
```

Exercise: Users and Book Reviews

Design a schema for users and their book reviews. Usernames are immutable.

- Users
  - username (string)
  - email (string)
- Reviews
  - text (string)
  - rating (integer)
  - created_at (date)

Solution A: Users and Book Reviews

Reviews may be queried by user or book

```
// db.users (one document per user)
{   _id: ObjectId("..."),
    username: "bob",
    email: "bob@example.com"
}

// db.reviews (one document per review)
{   _id: ObjectId("..."),
    user: ObjectId("..."),
    book: ObjectId("..."),
    rating: 5,
    text: "This book is excellent!",
    created_at: ISODate("2012-10-10T21:14:07.096Z")
}
```

Solution B: Users and Book Reviews

Optimized to retrieve reviews by user

```
// db.users, one document per user with all reviews
{   _id: ObjectId("..."),
    username: "bob",
    email: "bob@example.com",
    reviews: [
      {   book: ObjectId("..."),
          rating: 5,
          text: "This book is excellent!",
          created_at: ISODate("2012-10-10T21:14:07.096Z")
      }
    ]
}
```

Solution C: Users and Book Reviews

Optimized to retrieve reviews by book

```
// db.users (one document per user)
{   _id: ObjectId("..."),
    username: "bob",
    email: "bob@example.com"
}

// db.books, one document per book with all reviews
{   _id: ObjectId("..."),
    // Other book fields...
    reviews: [
      {   user: ObjectId("..."),
          rating: 5,
          text: "This book is excellent!",
          created_at: ISODate("2014-11-10T21:14:07.096Z")
      }
    ]
}
```

Store Binary Files in MongoDB with GridFS

- Application may have a requirement for binary file storage
- GridFS is a specification for storing files larger than 16MB in MongoDB
- Handled automatically by most drivers
- “mongofiles” is the command line tool for working with GridFS

How GridFS Works

- Files are split into chunks
- Default chunk size is 255k
- fs.files collection stores meta data for the file (name, size, etc.)
- fs.chunks collection stores chunks for binary file

Schema Design Use Cases with GridsFS

- Store large video files and stream chunks to a user
- Enterprise assets, replicated across data centers
- Medical record attachments (x-rays, reports, etc.)

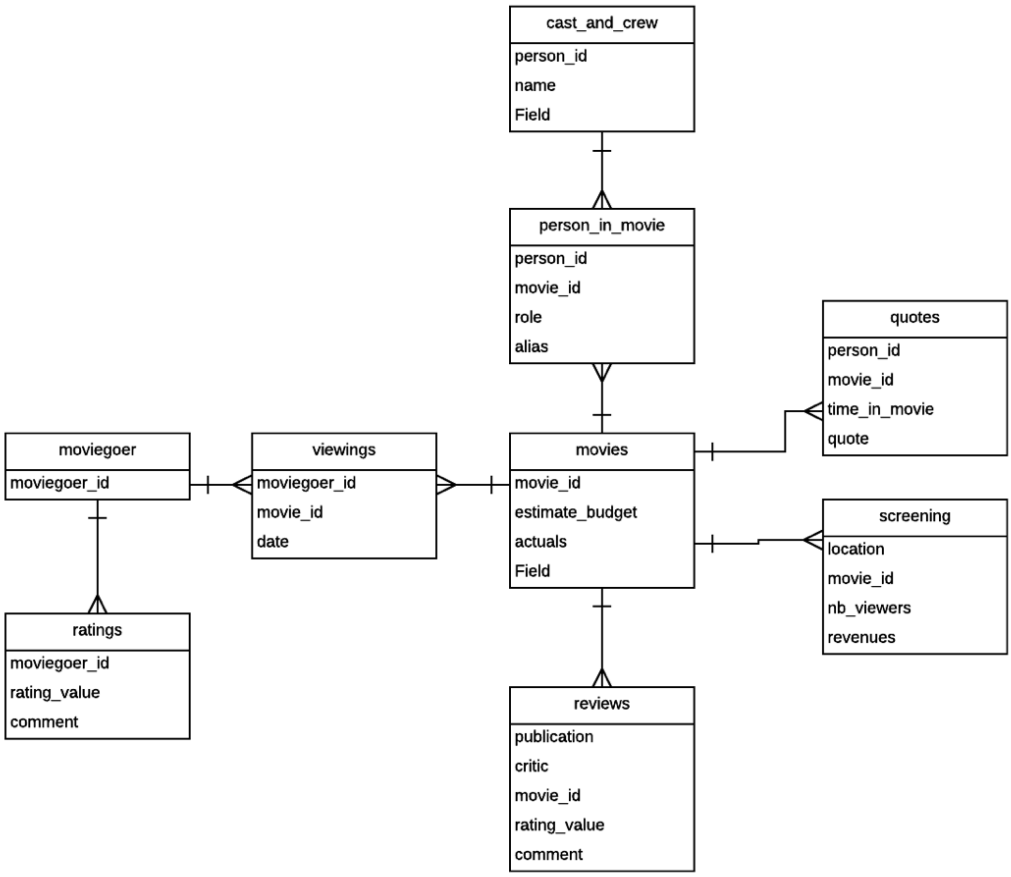
1.3 Schema Design Relationships

Leaning Objectives

Be able to:

- Model 1-1, 1-N and N-N relationships
- Create a model for some common problems

Entity Relationship Diagram Example



Note:

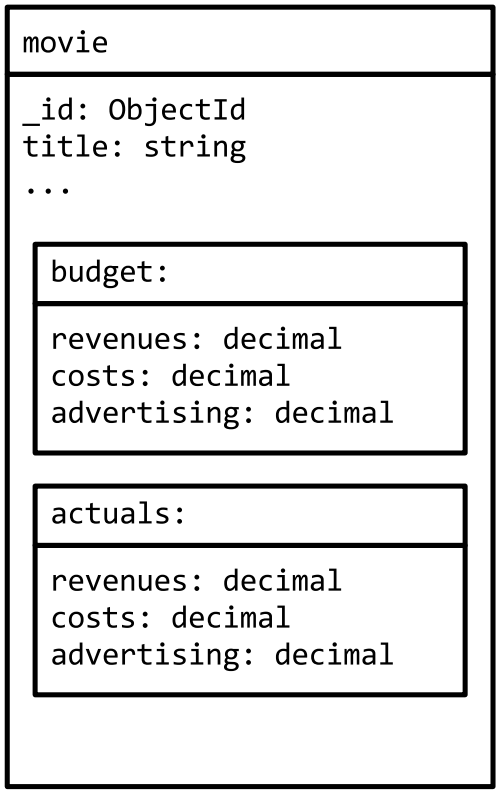
- [ASK] Which ER diagram methodologies they are familiar with.
- This is the ER diagram of movies/actors/viewers/...
- It will serve as our base for all the following examples in this session.
- Explain the different entities/collections, so they are understood for the following examples.

Relationship - 1 to 1, embedding

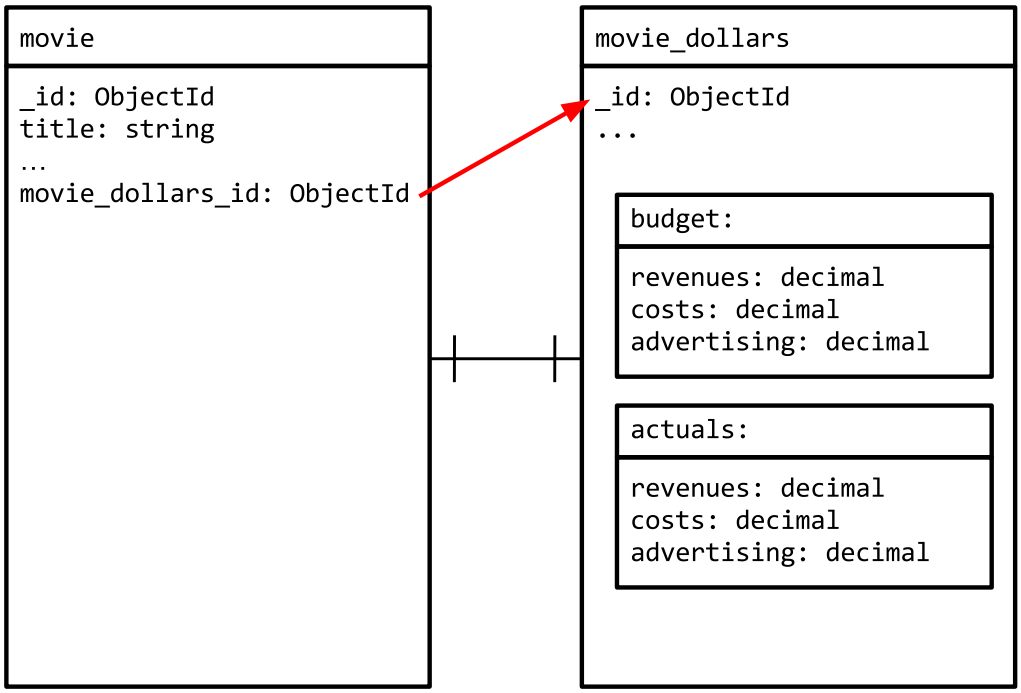
Note:

- We keep all the information together for a given entity This is the simplest model

---



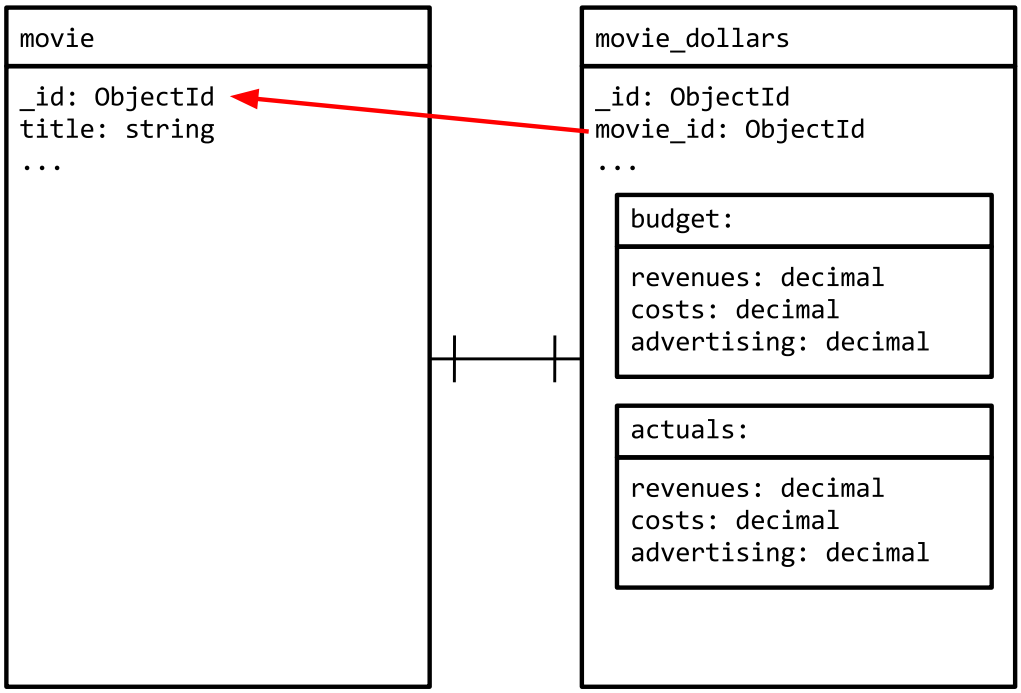
Relationship - 1 to 1, reference to the children



Note:

- Even if it is a 1:1 relationship, we are putting some information of an entity into another collection.
- [ASK] what is the use case?
  - You don't need the information put in the children that often
  - Examples:
    - Detailed description for a given product

Relationship - 1 to 1, reference to the parent



Note:

- Similar to the reference to the children, however we want to query directly the children collection for a given movie
- Again, you can always link both ways.
  - We will not show examples of linking both ways, those are done by simply using the fields from the 2 single side examples.

Embed or Store Separately

Given MongoDB hierarchical schema the question around *embed* or *separate document* is very common.

A few tips on how to approach these discussions:

- **You cannot perform atomic updates on more than one document**
- Combine objects that you will use together
  - Efficiency for reads
  - Atomicity for writes
  - Avoid application level joins
- Store documents in separate collections when
  - Read pattern are different
- Different lifecycle between relationships

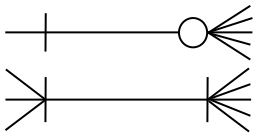
Retrieving the Documents

In a 1-N or N-N relationship, do you need to consider:

- How to identify the referencing documents without doing another query?
- How many documents do you want to update when the relationship changes?
  - Nested arrays, usually bad if you want to query in the embedded array
- Can use bi-directional referencing if it optimizes your schema and you are willing to live without atomic updates
- Growing documents is an important consideration when using MMAPv1

Cardinality of Relationships

- 1 to 1
- 1 to N
- N to N
- 1 to zillions and N to **zillions**
  - Let's add a new notation for **zillions**!



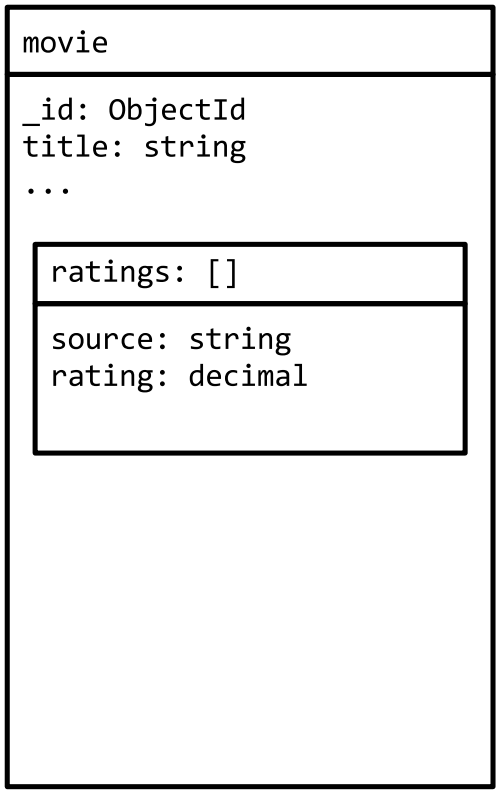
Representing the Cardinalities

- [10]
  - Exactly 10 elements
- [0, 20]
  - Minimum of 0 elements
  - Maximum of 20 elements
- [0, 10, 1000000]
  - Minimum of 0 elements
  - Median of 10 elements
  - Maximum of 1000000 elements

Note:

- You often get a push back that those numbers may be difficult to estimate, however it is worth the effort, even if you are way off.

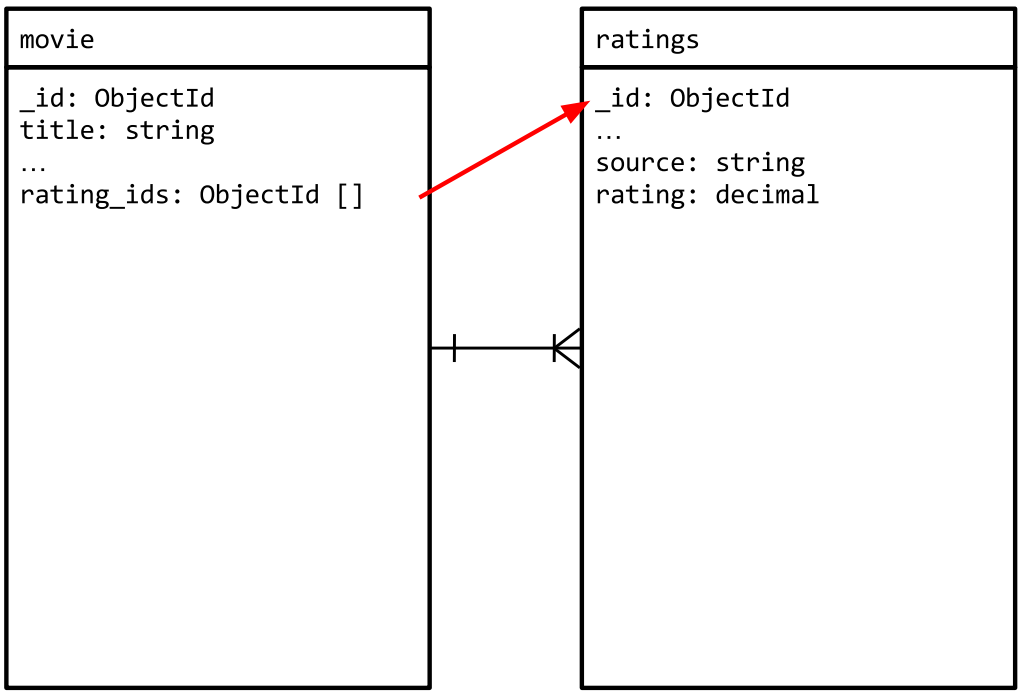
Relationship - 1 to N, embedding



Note:

- TBD

Relationship - 1 to N, reference to the children



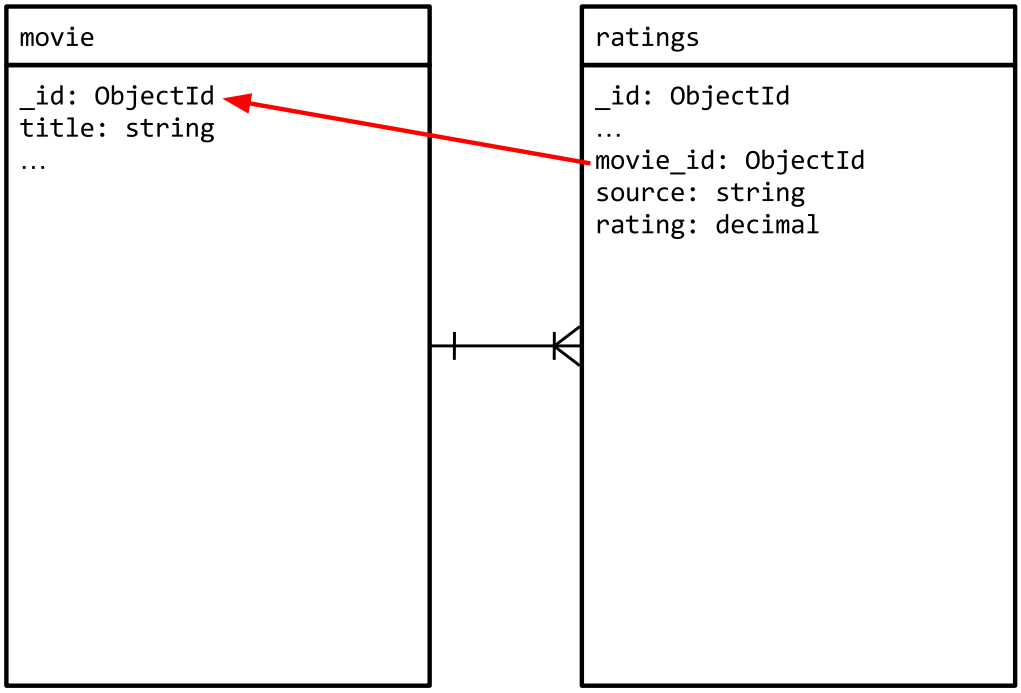
Note:

- TBD

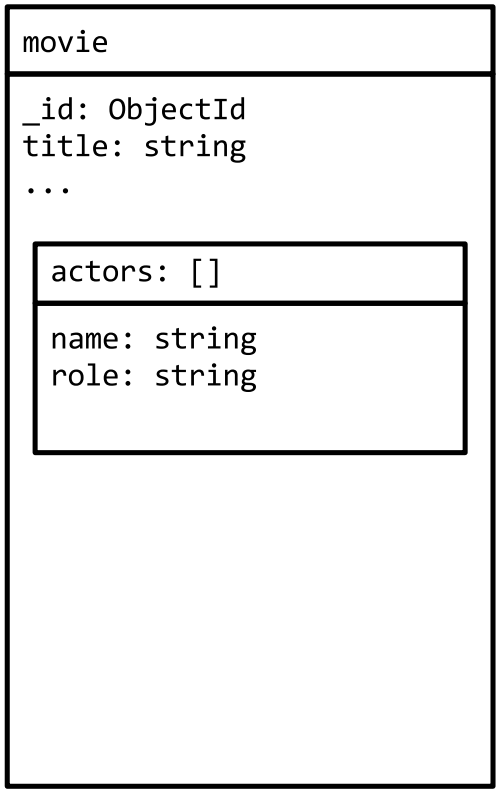
Relationship - 1 to N, reference to the parent

Note:

- TBD



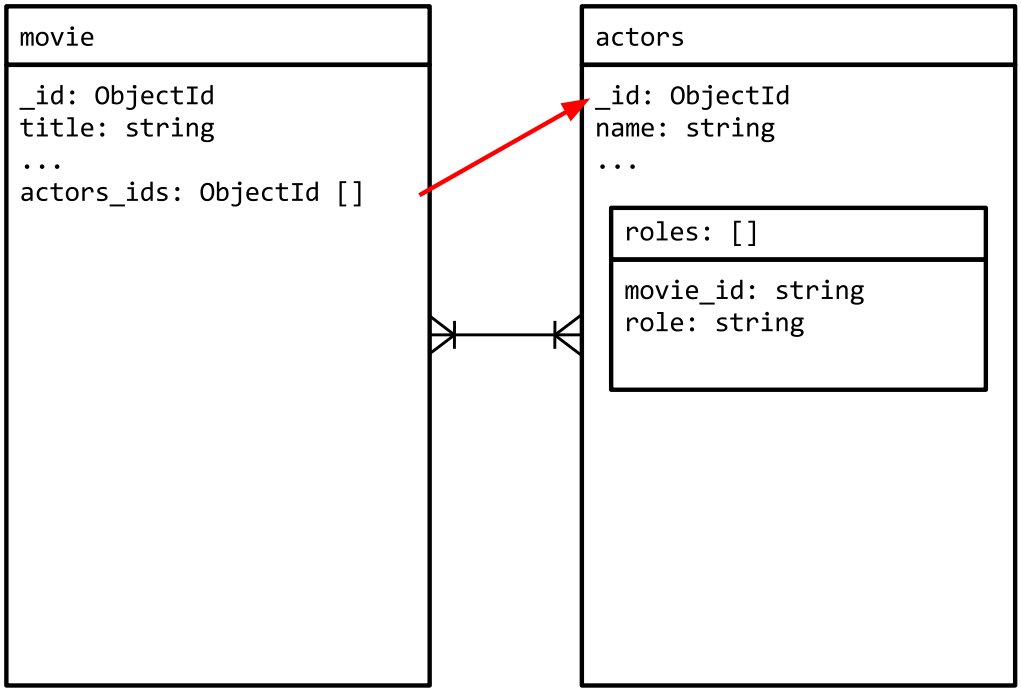
Relationship - N to N, embedding



Note:

- TBD

Relationship - N to N, reference to the children



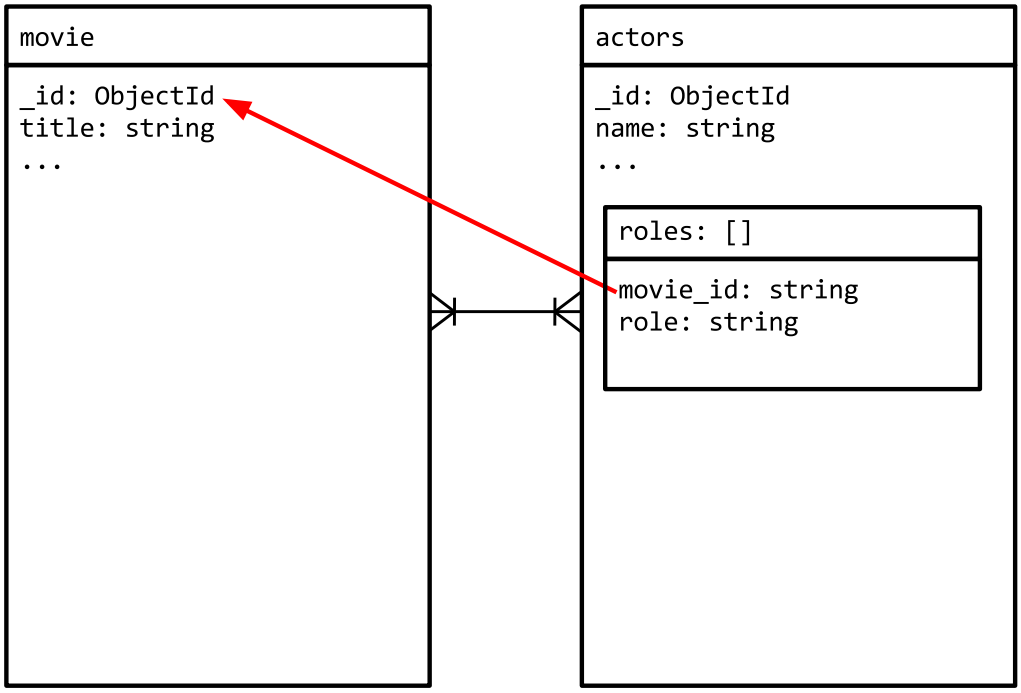
Note:

- TBD

Relationship - N to N, reference to the parent

Note:

- TBD



Relationships - Summary

relation type	1 to 1	1 to N	N to N
Embed	one read	no join	no join, however duplication of data
Reference	smaller reads, more read ops	smaller reads, more read ops	smaller reads, avoid duplication

Note:

- TODO - add the summary for each cell

1.4 Schema Design Patterns

Note: Avoid using the term **pattern** as it is used for representing partial solutions, like the Gang of Four uses it for software design patterns

Leaning Objectives

Be able to:

- Identify schema design patterns

Patterns

- They are not:
  - Modeling of relationships
  - The full "solution" of a problem
- They only address a precise use case in a problem
  - Similar to the GoF (Gang of Four) with their patterns for Object Oriented Design

Pattern Characteristics

- NoSQL patterns ...
  - Often address performance concerns
    - Like reducing the number of reads
  - May create duplication

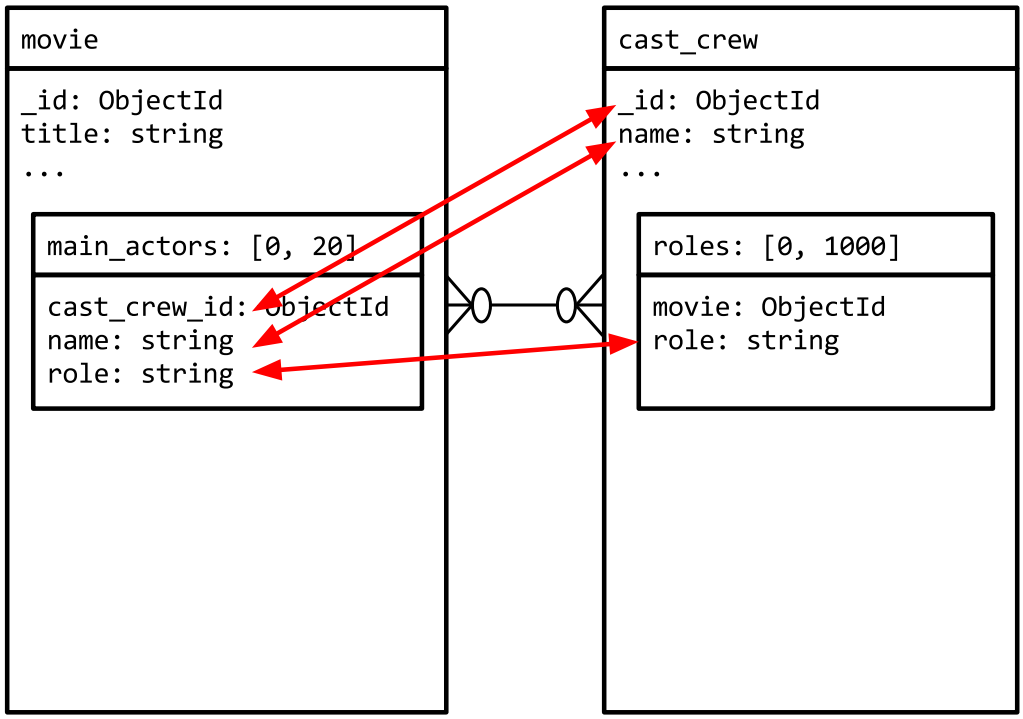
Pattern #1

- You want to display dependent information, however only part of it
- The rest of the data is fetch only if needed
- Examples:
  - Last 10 comments on an article

Note:

- TBD

Example #1



Note:

- TBD

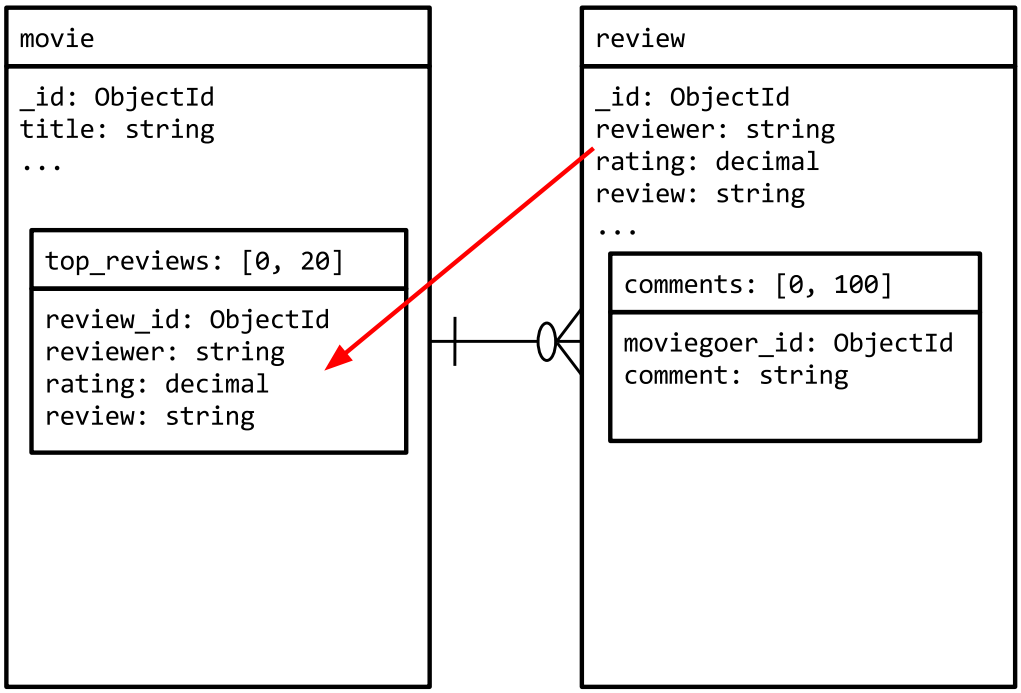
Pattern #2

- You want to display dependent data
- Does not have to be up to date
- Too expensive to update the document all the time
- Updates can be done in batch
- Examples:
  - Top reviews on a movie
  - Top reviews on a product
  - Number of seats available on a flight

Note:

- TBD

Example #2



Note:

- TBD

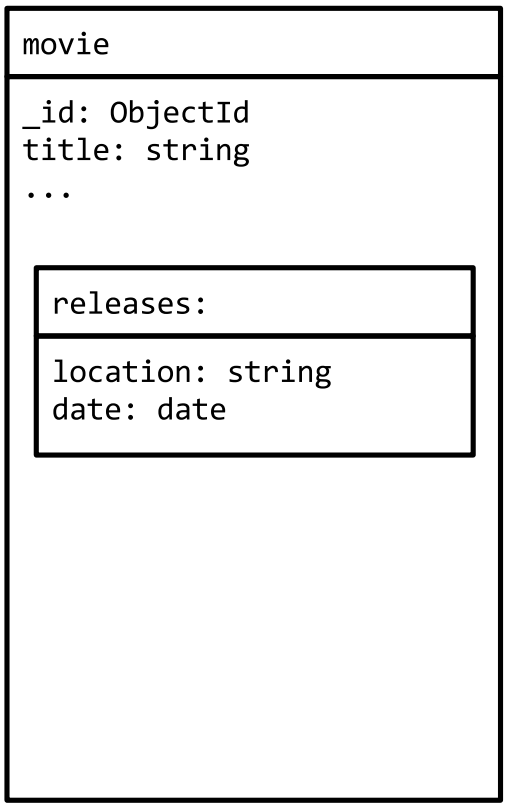
Pattern #3

- A lot of different and predictable values
- Need to index the attributes
- Examples:
  - Catalog items
    - A shirt is XL, blue, iron-free

Note:

- TBD

Example #3




```
{
  title: “Moonlight”,
  releases: [

    { location: “USA”,
      date: “2016/09/02” },

    { location: “Mexico”,
      date: “2017/01/27” },

    { location: “France”,
      date: “2017/02/01” }
  ]
}
```

Note:

- TBD

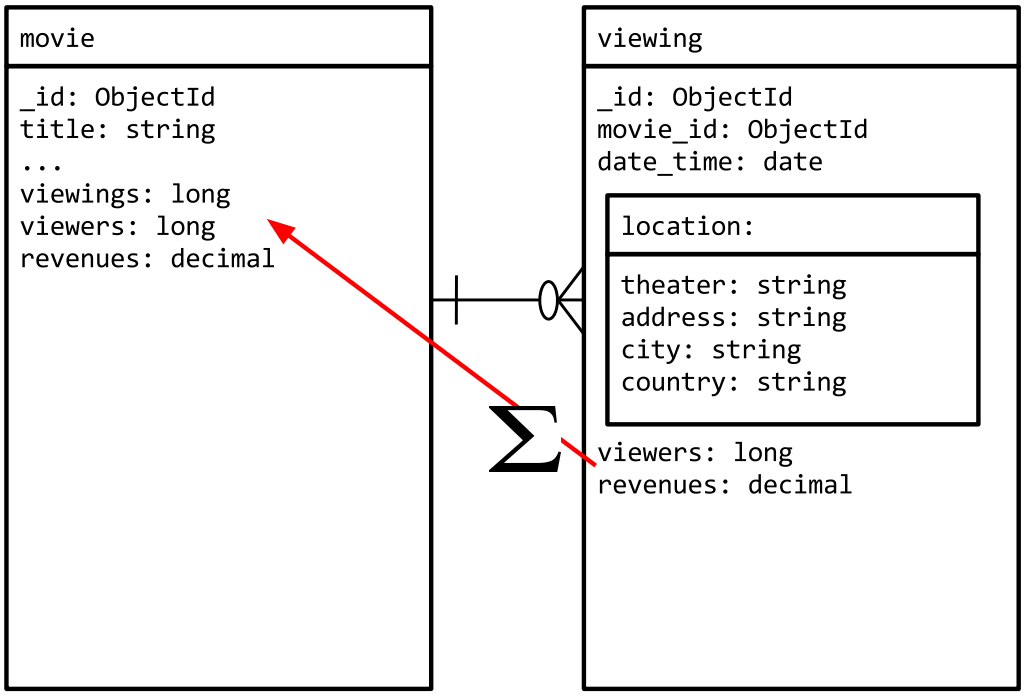
Pattern #4

- One document shows a sum of data from other documents
- Examples:
  - Cumulative sales from many theaters

Note:

- TBD

Example #4



Note:

- TBD

Pattern - Tree as Nodes

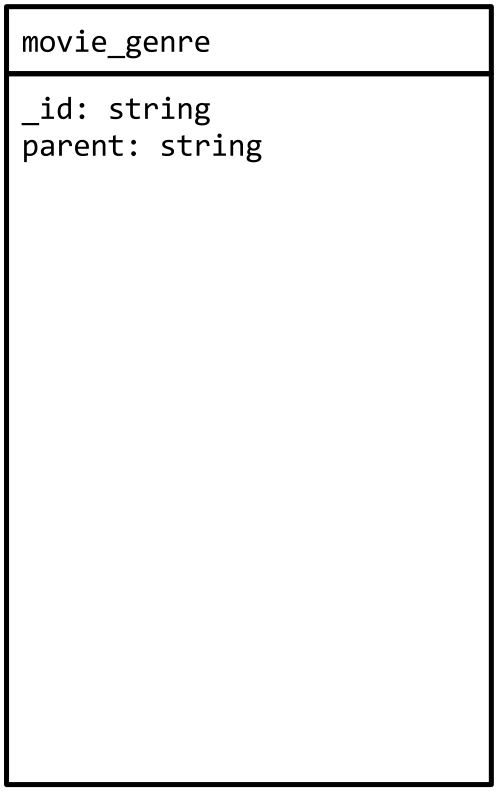
Note:

- Node is defined by itself and its parent
- Likely need recursive lookup to build it

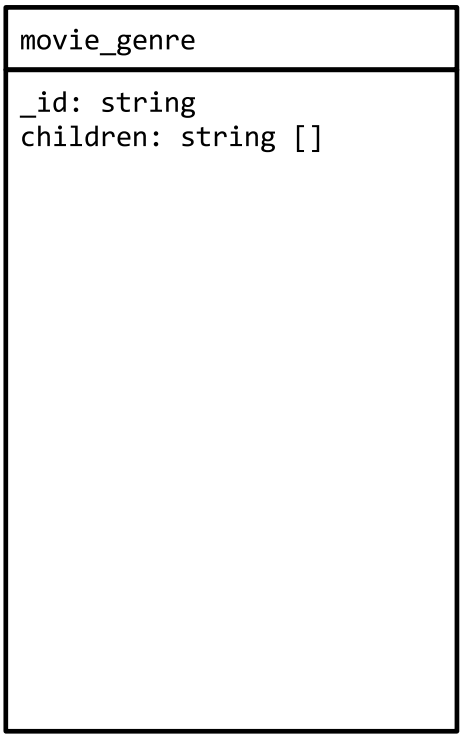
Pattern - Tree as Children

Note:

- Node is defined by itself and its children

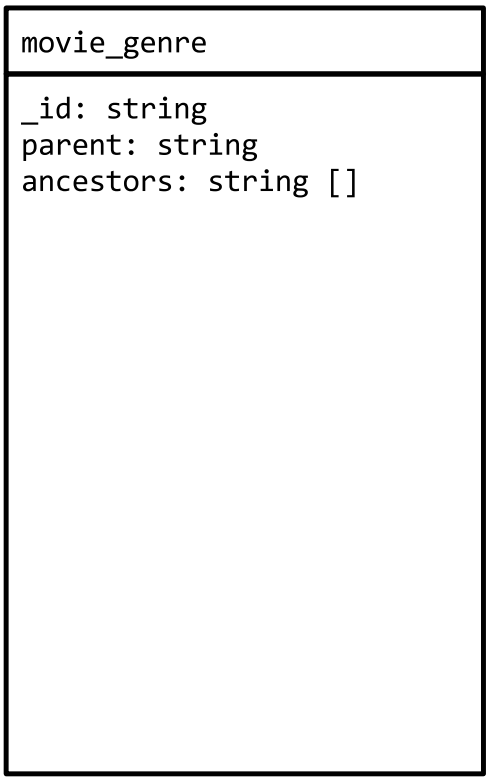



```
{
  _id: “Documentary”,
  parent: “null”
}

{
  _id: “War Documentary”,
  parent: “Documentary”
}

{
  _id: “WWII Documentary”,
  parent: “War Documentary”
}
```




```
{
  _id: “Documentary”,
  children: [
    “Biography”,
    “Business Documentary”,
    “War Documentary”
    ...
  ]
}

{
  _id: “War Documentary”,
  children: [
    “US Civil War Documentary”,
    “WWI Documentary”,
    “WWII Documentary”
    ...
  ]
}
```

Pattern - Tree as Ancestors




```
{
  _id: “WWII Documentary”,
  parent: “War Documentary”,
  ancestors: [
    “Documentary”,
    “War Documentary”
  ]
}
{
  _id: “War Documentary”,
  parent: “Documentary”,
  ancestors: [
    “Documentary”,
  ]
}
{
  _id: “Documentary”,
  parent: “null”,
  ancestors: [ ]
}
```

Note:

- Node is defined by itself and all its parent/ancestors

Comparing Tree Patterns

	Nodes	Children	Ancestors
Restructure tree (num of updates)	One per moved node	One per moved node	Many updates per moved node
Information per document	Very little	Very little	Much more

Future work

- More work on patterns coming out soon
- New class on **Data Modeling** likely before the end of the year

1.5 Replica Sets and Performance

Learning Objectives

Upon completing this module, students should be able to:

- Define write concern, read preference, read concern, and linearizable writes
- Evaluate the performance tradeoffs that occur with increased durability guarantees

Defining Write Concern

- MongoDB acknowledges its writes
- Write concern determines when that acknowledgment occurs
  - How many servers
  - Whether on disk or not
- Clients may define the write concern per write operation, if necessary.
- Standardize on specific levels of write concerns for different classes of writes.
- In the discussion that follows we will look at increasingly strict levels of write concern.

Performance Tips: Write Concern

- Increasing the write concern increases data safety.
- This will have an impact on performance, however.
- This is especially true when there are network issues.
- You will want to balance business needs against speed.

What is Read Preference?

- Read preference allows you to specify the nodes in a replica set to read from.
- Clients only read from the primary by default.
- There are some situations in which a client may want to read from:
  - Any secondary
  - A specific secondary
  - A specific type of secondary
- Only read from a secondary if you can tolerate possibly stale data, as not all writes might have replicated.

Use Cases

- Running systems operations without affecting the front-end application.
- Providing local reads for geographically distributed applications.
- Maintaining availability during a failover.

Note:

- If you have application servers in multiple data centers, you may consider having a geographically distributed replica set[1] and using a read preference of `nearest`.
- This allows the client to read from the lowest-latency members.
- Use `primaryPreferred` if you want an application to read from the primary under normal circumstances, but to allow possibly stale reads from secondaries during failover.

Not for Scaling

- In general, do *not* read from secondaries to provide extra capacity for reads.
- Sharding[2] increases read and write capacity by distributing operations across a group of machines.
- Sharding is a better strategy for adding capacity.

[1] http://docs.mongodb.org/manual/core/replica-set-geographical-distribution
[2] http://docs.mongodb.org/manual/sharding

Read Preference Modes

MongoDB drivers support the following read preferences. Note that `hidden` nodes will never be read from when connected via the replica set.

- **primary**: Default. All operations read from the primary.
- **primaryPreferred**: Read from the primary but if it is unavailable, read from secondary members.
- **secondary**: All operations read from the secondary members of the replica set.
- **secondaryPreferred**: Read from secondary members but if no secondaries are available, read from the primary.
- **nearest**: Read from member of the replica set with the least network latency, regardless of the member's type.

Tag Sets

- There is also the option to used tag sets.
- You may tag nodes such that queries that contain the tag will be routed to one of the servers with that tag.
- This can be useful for running reports, say for a particular data center or nodes with different hardware (e.g. hard disks vs SSDs).

For example, in the mongo shell:

```
conf = rs.conf()
conf.members[0].tags = { dc : "east", use : "production" }
conf.members[1].tags = { dc : "east", use : "reporting" }
conf.members[2].tags = { use : "production" }
rs.reconfig(conf)
```

Read Concerns

- **Local**: *Default*
- **Majority**: Added in MongoDB 3.2, requires WiredTiger and election protocol version 1
- **Linearizable**: Added in MongoDB 3.4, works with MMAP or WiredTiger

Local

- Default read concern
- Will return data from the primary.
- Does not wait for the write to be replicated to other members of the replica set.

Majority

- Available only with WiredTiger.
- Reads majority acknowledged writes from a snapshot.
- Under certain circumstances (high volume, flaky network), can result in stale reads.

Linearizable

- Available with MongoDB versions > 3.4
- Will read latest data acknowledged with `w: majority`, or block until replica set acknowledges a write in progress with `w: majority`
- Can result in **very slow** queries.
  - Always use **maxTimeMS** with **linearizable**
- Only guaranteed to be a linearizable read when the query fetches a single document

---

Note:

- For replica sets that run with **writeConcernMajorityJournalDefault** set to true, linearizable read concern returns data that will never be rolled back.
- With **writeConcernMajorityJournalDefault** set to false, MongoDB will not wait for *w: "majority"* writes to be durable before acknowledging the writes. As such, "majority" write operations could possibly roll back in the event of a loss of a replica set member.
- Only primary servers may be queried with **linearizable**.

Questions to ask:

- Can I avoid dirty reads if I write a document with write concern "majority" and read preference: "primary"?
  - Answer: No. Without using read concern level : "majority", reads can be dirty
- What can happen if I use a write concern of `{ w: 1 }` and read concern level of "majority"?
  - Answer: You will not have dirty reads ... but you may be unable to read your own writes

---

Replica Sets and Performance

- You must balance performance costs against read/write guarantees
- Write concern and read concern allow you to set this
- Use the weakest guarantee your application can tolerate
- Think about technical ways to get by with weaker guarantees

1.6 Schema Evolution

Learning Objectives

Upon completing this module, students should understand the basic philosophy of evolving a MongoDB schema during an application's lifetime:

- Development Phase
- Production Phase
- Iterative Modifications

Development Phase

Support basic CRUD functionality:

- Inserts for authors and books
- Find authors by name
- Find books by basics of title, subject, etc.

Development Phase: Known Query Patterns

```
// Find authors by last name.
db.authors.createIndex({ "lastName": 1 })

// Find books by slug for detail view
db.books.createIndex({ "slug": 1 })

// Find books by subject (multi-key)
db.books.createIndex({ "subjects": 1 })

// Find books by publisher (index on embedded doc)
db.books.createIndex({ "publisher.name": 1 })
```

Production Phase

Evolve the schema to meet the application's read and write patterns.

Production Phase: Read Patterns

List books by author last name

```
authors = db.authors.find({ lastName: /^f.*/i }, { _id: 1 });

authorIds = authors.map(function(x) { return x._id; });

db.books.find({author: { $in: authorIds }});
```

Addressing List Books by Last Name

"Cache" the author name in an embedded document.

```
{
    _id: 1,
    title: "The Great Gatsby",
    author: {
        firstName: "F. Scott",
        lastName: "Fitzgerald"
    }
    // Other fields follow...
}
```

Queries are now one step

```
db.books.find({ "author.firstName": /^f.*/i })
```

Production Phase: Write Patterns

Users can review a book.

```
review = {
    user: 1,
    text: "I thought this book was great!",
    rating: 5
};

db.books.updateOne(
    { _id: 3 },
    { $push: { reviews: review }}
);
```

Caveats:

- Document size limit (16MB)
- Storage fragmentation after many updates/deletes

Exercise: Recent Reviews

- Display the 10 most recent reviews by a user.
- Make efficient use of memory and disk seeks.

Solution: Recent Reviews, Schema

Store users' reviews in monthly buckets.

```
// db.reviews (one document per user per month)
{   _id: "bob-201412",
    reviews: [
        {   _id: ObjectId("..."),
            rating: 5,
            text: "This book is excellent!",
            created_at: ISODate("2014-12-10T21:14:07.096Z")
        },
        {   _id: ObjectId("..."),
            rating: 2,
            text: "I didn't really enjoy this book.",
            created_at: ISODate("2014-12-11T20:12:50.594Z")
        }
    ]
}
```

Solution: Recent Reviews, Update

Adding a new review to the appropriate bucket

```
myReview = {
    _id: ObjectId("..."),
    rating: 3,
    text: "An average read.",
    created_at: ISODate("2012-10-13T12:26:11.502Z")
};

db.reviews.updateOne(
      { _id: "bob-201210" },
      { $push: { reviews: myReview }}
);
```

Solution: Recent Reviews, Read

Display the 10 most recent reviews by a user

```
cursor = db.reviews.find(
    { _id: /^bob-/ },
    { reviews: { $slice: -10 }}
).sort({ _id: -1 }).batchSize(5);

num = 0;

while (cursor.hasNext() && num < 10) {
    doc = cursor.next();

    for (var i = 0; i < doc.reviews.length && num < 10; ++i, ++num) {
        printjson(doc.reviews[i]);
    }
}
```

Solution: Recent Reviews, Delete

Deleting a review

```
db.reviews.updateOne(
    { _id: "bob-201210" },
    { $pull: { reviews: { _id: ObjectId("...") }}}
);
```

1.7 Document Validation

Learning Objectives

Upon completing this module, students should be able to:

- Define the different types of document validation
- Distinguish use cases for document validation
- Create, discover, and bypass document validation in a collection
- List the restrictions on document validation

Introduction

- Prevents or warns when the following occurs:
  - Inserts/updates that result in documents that don't match a schema
- Prevents or warns when inserts/updates do not match schema constraints
- Can be implemented for a new or existing collection
- Can be bypassed, if necessary

Note:

- We'll get to updates later; it's a little ambiguous for now
- Document validation does not affect deletion at all
- Document validation applies to non-CRUD operations that act like inserts
  - e.g. Aggregation's `$out` stage

Example

```
db.createCollection( "products",
   {
      validator: {
          price : { $exists : true }
      },
      validationAction: "error"
   }
)
```

Note: validationAction: "error" means that the server will reject non-matching documents

---

Why Document Validation?

Consider the following use case:

- Several applications write to your data store
- Individual applications may validate their data
- You need to ensure validation across all clients

Why Document Validation? (Continued)

Another use case:

- You have changed your schema in order to improve performance
- You want to ensure that any write will also map the old schema to the new schema
- Document validation is a simple way of enforcing the new schema after migrating
  - You will still want to enforce this with the application
  - Document validation gives you another layer of protection

Anti-Patterns

- Using document validation at the database level without writing it into your application
  - This would result in unexpected behavior in your application
- Allowing uncaught exceptions from the DB to leak into the end user's view
  - Catch it and give them a message they can parse

`validationAction` and `validationLevel`

- Two settings control how document validation functions
- `validationLevel` – determines how strictly MongoDB applies validation rules
- `validationAction` – determines whether MongoDB should error or warn on invalid documents

Details

		validationLevel		
		off	moderate	strict
validationAction	warn	No checks	Warn on validation failure for inserts & updates to existing valid documents. Updates to existing invalid docs OK.	Warn on any validation failure for any insert or update.
	error	No checks	Reject invalid inserts & updates to existing valid documents. Updates to existing invalid docs OK.	Reject any violation of validation rules for any insert or update. **DEFAULT**

`validationLevel`: "strict"

- Useful when:
  - Creating a new collection
  - Validating writes to an existing collection already in compliance
  - Insert only workloads
  - Changing schema and updates should map documents to the new schema
- This will impose validation on update even to invalid documents

---

Note:

- The application will still need to perform validation
- This is more of a backstop for the application

`validationLevel`: "moderate"

- Useful when:
  - Changing a schema and you have not migrated fully
  - Changing schema but the application can't map the old schema to the new in just one update
  - Changing a schema for new documents but leaving old documents with the old schema

`validationAction`: "error"

- Useful when:
  - Your application will no longer support valid documents
  - Not all applications can be trusted to write valid documents
  - Invalid documents create regulatory compliance problems

`validationAction`: "warn"

- Useful when:
  - You need to receive all writes
  - Your application can handle multiple versions of the schema
  - Tracking schema-related issues is important
    - For example, if you think your application is probably inserting compliant documents, but you want to be sure

Creating a Collection with Document Validation

```
db.createCollection( "products",
   {
      validator: {
          price: { $exists: true }
      },
      validationAction: "error"
   }
)
```

Note:

- This is the same example we saw at the beginning of this lesson

---

Seeing the Results of Validation

To see what the validation rules are for all collections in a database:

```
db.getCollectionInfos()
```

And you can see the results when you try to insert:

```
db.products.insertOne( { price: 25, currency: "USD" } )
```

Adding Validation to an Existing Collection

```
db.products.drop()
db.products.insertOne( { name: "watch", price: 10000, currency: "USD" } )
db.products.insertOne( { name: "happiness" } )
db.runCommand( {
   collMod: "products",
   validator: {
       price: { $exists: true }
   },
   validationAction: "error",
   validationLevel: "moderate"
} )
db.products.updateOne( { name : "happiness" }, { $set : { note: "Priceless." } } )
db.products.updateOne( { name : "watch" }, { $unset : { price : 1 } } )
db.products.insertOne( { name : "inner peace" } )
```

Note:

- First two inserts worked b/c there was no validation at first
- First update worked b/c the document didn't match before the update
- Second update failed because it doesn't match the validator and the document matched before the update was attempted
- Final insert failed because it didn't match the validator

Bypassing Document Validation

- You can bypass document validation using the `bypassDocumentValidation` option
  - On a per-operation basis
  - Might be useful when:
    * Restoring a backup
    * Re-inserting an accidentally deleted document
- For deployments with access control enabled, this is subject to user roles restrictions
- See the MongoDB server documentation for details

Note:

- User roles are covered in the security section

Limits of Document Validation

- Document validation is not permitted for the following databases:
  - admin
  - local
  - config
- You cannot specify a validator for `system.*` collections

Note:

- Ask the students why you can't write to these databases.
  - It's because MongoDB holds metadata for security, replication, and sharding, respectively, in these databases.

Document Validation and Performance

- Validation adds an expression-matching evaluation to every insert and update
- Performance load depends on the complexity of the validation document
  - Many workloads will see negligible differences

Quiz

What are the validation levels available and what are the differences?

Note: Answers:

- Strict - every insert or update must pass validation
- Moderate:
  - Updates to invalid documents are permitted even if the update does not pass validation
  - New documents must be validated
  - Updates to valid documents must pass validation
- Off - disables document validation

Quiz

What command do you use to determine what the validation rule is for the *things* collection?

Note:

- Trick question. You can find out for all collections in the database with db.getCollectionInfos(), but there's no way to do it for just one collection.

Quiz

On which three databases is document validation not permitted?

Note:

- admin: holds security
- local: holds the oplog and other replication information
- config: holds sharding metadata
- Your application should not write directly to these databases anyway

---

1.8 Lab: Document Validation

Exercise: Add validator to existing collection

- Import the `posts` collection (from posts.json) and look at a few documents to understand the schema.
- Insert the following document into the `posts` collection

```
{"Hi":"I'm not really a post, am I?"}
```

- Discuss: what are some restrictions on documents that a validator could and should enforce?
- Add a validator to the `posts` collection that enforces those restrictions
- Remove the previously inserted document and try inserting it again and see what happens

---

Note: Discuss the potential restrictions as a class, but have students write and add validators individually.

Some example restrictions:

- `body` is a string
- `permalink` matches a regex
- `date` is a date
- `author` is a string
- `title` is a string

An example of adding a validator to the `posts` collection:

```
db.runCommand( {
    "collMod": "posts",
    "validator": {
        "body": {"$type": "string"},
        "permalink": {"$regex": "^[A-z]{20}$"},
        "date": {"$type":"date"},
        "author": {"$type": "string"},
        "title": {"$type": "string"}
    }
})
```

Exercise: Create collection with validator

Create a collection `employees` with a validator that enforces the following restrictions on documents:

- The `name` field must exist and be a string
- The `salary` field must exist and be between 0 and 10,000 inclusive.
- The `email` field is optional but must be an email address in a valid format if present.
- The `phone` field is optional but must be a phone number in a valid format if present.
- At least one of the `email` and `phone` fields must be present.

Note:

- Examples on next slide.

Exercise: Create collection with validator (expected results)

```
// Valid documents
{"name":"Jane Smith", "salary":45, "email":"js@example.com"}
{"name":"Tim R. Jones", "salary":30,
 "phone":"234-555-6789","email":"trj@example.com"}
{"name":"Cedric E. Oxford", "salary":600, "phone":"918-555-1234"}

// Invalid documents
{"name":"Connor MacLeod", "salary":9001, "phone":"999-555-9999",
 "email":"thrcnbnly1"}
{"name":"Herman Hermit", "salary":9}
{"name":"Betsy Bedford", "salary":50, "phone":"", "email":"bb@example.com"}
```

Note: A possible solution could be the following:

```
db.createCollection("employees", { "validator":
  { "$and": [
    { "name": { "$type": 2 } },
    { "salary": { "$lte": 10000, "$gte": 0 } },
    { "$or": [
      { "email": { "$regex": "[A-z0-9_.+]+@[A-z0-9_.+]+\\.com" } },
      { "email": { "$exists": false } },
      { "email": null }
    ] },
    { "$or": [
      { "phone": { "$regex": "\\(?[0-9]{3}\\)?[- ]?[0-9]{3}[- ]?[0-9]{4}" } },
      { "phone": { "$exists": false } },
      { "phone": null }
    ] }
  ] }
})
```

- Ensure that students can correctly implement the subtleties of the restrictions on the phone and email fields.
- Actual regular expressions for email addresses and phone numbers can be very complex, so less comprehensive approximations such as those above are acceptable.

```
// example of actual email validation regex
/(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\
x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-
\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a
-z0-9])?|\[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0
-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1
f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])/
// example of actual phone validation regex
/^(?:(?:\+?1\s*(?:[.-]\s*)?)?(?:\(\s*([2-9]1[02-9]|[2-9][02-8]1|[2-9][02-8
][02-9])\s*\)|([2-9]1[02-9]|[2-9][02-8]1|[2-9][02-8][02-9]))\s*(?:[.-]\s*)
?)?([2-9]1[02-9]|[2-9][02-9]1|[2-9][02-9]{2})\s*(?:[.-]\s*)?([0-9]{4})(?:\
s*(?:#|x\.?|ext\.?|extension)\s*(\d+))?$/
```

- Ensure that students successfully insert the valid example documents; they are needed in a later exercise.

Exercise: Change validator rules

Modify the validator for the `employees` collection to support the following additional restrictions:

- The `status` field must exist and must only be one of the following strings: "active", "on_vacation", "terminated"
- The `locked` field must exist and be a boolean

Note: Examples on next slide

Exercise: Change validator rules (expected results)

```
// Valid documents
{"name":"Jason Serivas", "salary":65, "email":"js@example.com",
 "status":"terminated", "locked":true}
{"name":"Logan Drizt", "salary":39,
 "phone":"234-555-6789","email":"ld@example.com", "status":"active",
 "locked":false}
{"name":"Mann Edger", "salary":100, "phone":"918-555-1234",
 "status":"on_vacation", "locked":false}

// Invalid documents
{"name":"Steven Cha", "salary":15, "email":"sc@example.com", "status":"alive",
 "locked":false}
{"name":"Julian Barriman", "salary":15, "email":"jb@example.com",
 "status":"on_vacation", "locked":"no"}
```

Note: A possible solution could be the following:

```
db.runCommand( {
  "collMod": "employees",
  "validator": {
    "$and": [
      { "name": { "$type": 2 } },
      { "salary": { "$lte": 10000, "$gte": 0 } },
      { "locked": { "$type": 8 } },
      { "status": { "$in": [ "active", "on_vaction", "terminated" ] } },
      { "$or": [
        { "email": { "$regex": "[A-z0-9_.+]+@[A-z0-9_.+]+\\.com" } },
        { "email": { "$exists": false } },
        { "email": null }
      ] },
      { "$or": [
        { "phone": { "$regex": "\\(?[0-9]{3}\\)?[- ]?[0-9]{3}[- ]?[0-9]{4}"} },
        { "phone": { "$exists": false } },
        { "phone": null }
      ] }
    ]
  }
} )
```

Exercise: Change validation level

Now that the `employees` validator has been updated, some of the already-inserted documents are not valid. This can be a problem when, for example, just updating an employee's salary.

- Try to update the salary of "Cedric E. Oxford". You should get a validation error.
- Now, change the validation level of the `employees` collection to allow updates of existing invalid documents, but still enforce validation of inserted documents and existing valid documents.

Note: Note that this permanently changes the validation level for all operations on the collection. Overriding the validation level on a per-operation basis is covered later.

You could ask the questions:

- When would this be required?
- What happens with updates?
- What happens with new inserts?

Example solution:

```
db.runCommand({"collMod":"employees", "validationLevel":"moderate"})
```

Exercise: Use Compass to Create and Change validation rules

Now that we've explored document validation in the Mongo shell, let's explore how easy it is to do with MongoDB Compass.

- Click below for an overview of MongoDB Compass.

`http://docs.mongodb.org/training/training-instructor/modules/compass`

- Connect to your local database with Compass
- Open the employees collection, and view the validation rules.

Exercise: Compass Validation (continued)

- From a Mongo shell, create a new collection called `employees_v2`
- Implement the initial validation rules for the `employees` collection on `employees_v2` using Compass
  - Ensure you select "strict" as the validation level, and "error" as the validation action
  - Try inserting some documents either through Compass or the shell to confirm your validation is working.

Exercise: Bypass validation

In some circumstances, it may be desirable to bypass validation to insert or update documents.

- Use the `bypassDocumentValidation` option to insert the document `{"hi":"there"}` into the `employees` collection
- Use the `bypassDocumentValidation` option to give all employees a salary of 999999.

Note: Example solutions:

```
db.runCommand({"insert":"employees","documents":[{"hi":"there"}],
               "bypassDocumentValidation":true})
db.runCommand({
    "update":"employees",
    "updates":[{
        "q":{},
        "u":{"$set":{"salary":999999}},
        "multi":true
    }],
    "bypassDocumentValidation":true})
```

- Students may ask, is it possible to bypass only part of the validation?
- No, there is no partial validation.

Exercise: Change validation action

In some cases, it may be desirable to simply log invalid actions, rather than prevent them.

- Change the validation action of the `employees` collection to reflect this behavior

---

Note: Example solution:

```
db.runCommand({"collMod":"employees", "validationAction":"warn"})
```

You could ask the questions:

- When would this be useful?

1.9 Schema Visualization With Compass

Learning Objectives

Upon completing this module, students should understand:

- How to use Compass to explore and visualize schema
- Point and click queries
- GeoJSON queries
- How to use Compass to update a document

Note:

- If you have not covered the basics of compass, click below to give the students a brief overview

Using Compass to Visualize Schema

- Schema tab shows an overview of document schema, to include types
- Based on a $sample[3] of the overall collection, up to 1000 documents
- Fields can be clicked for interactive query and further visualization

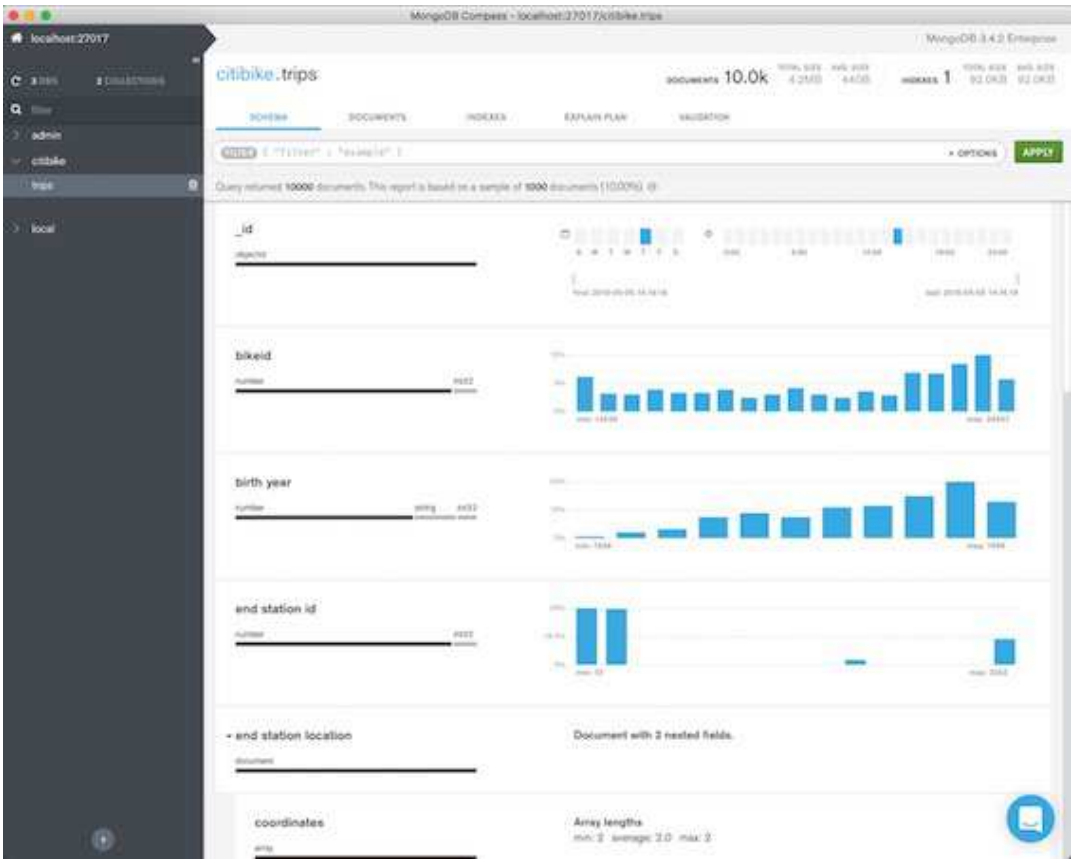
Lesson Setup

- Import the `trips.json` collection to a running `mongod`

```
mongoimport --drop -d citibike -c trips trips.json
```

- Connect to your `mongod` with Compass and select the trips collection

Schema Visualization

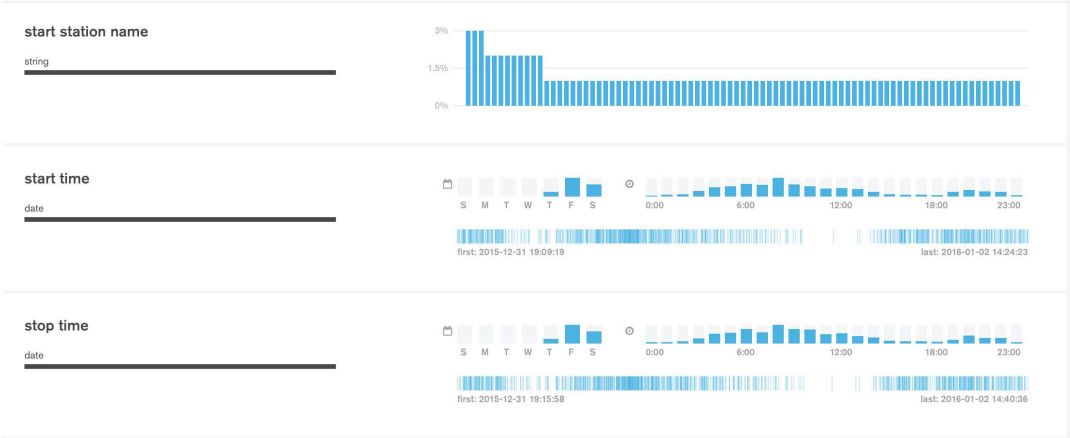


Note:

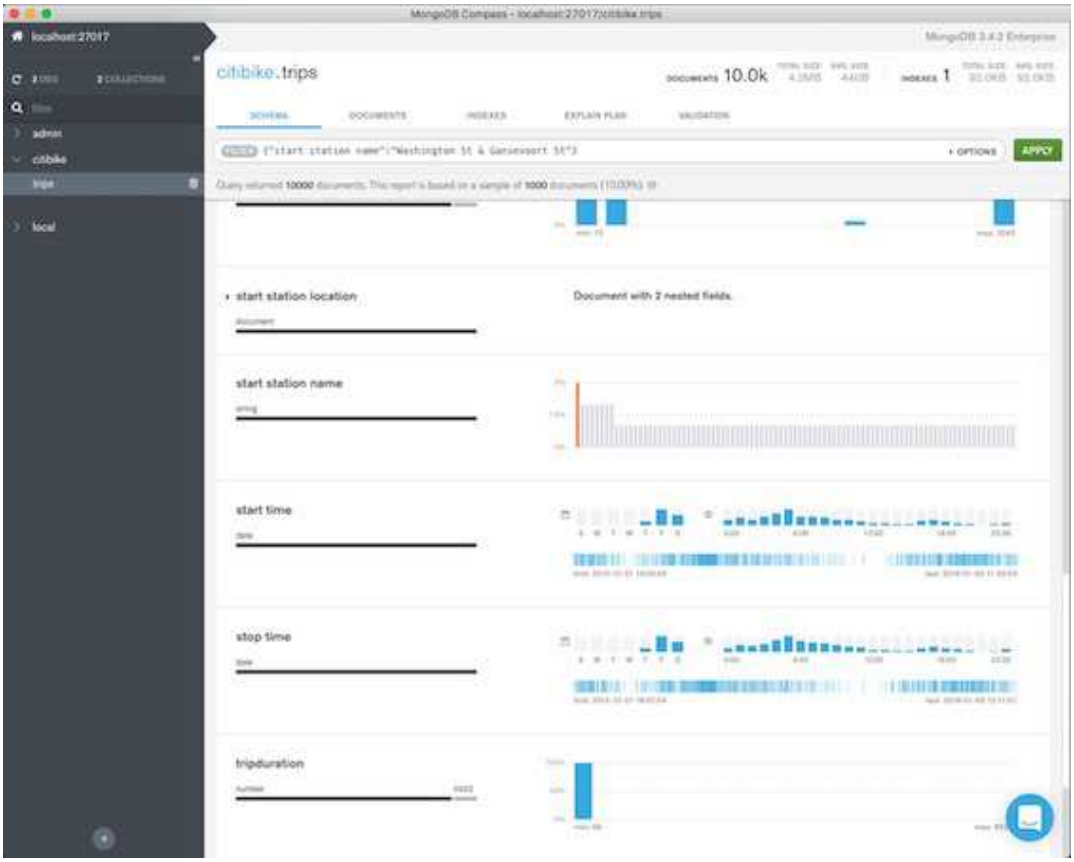
- This is only a sample, scroll down to show the full schema

[3] https://docs.mongodb.com/manual/reference/operator/aggregation/sample/

Schema Visualization Detail



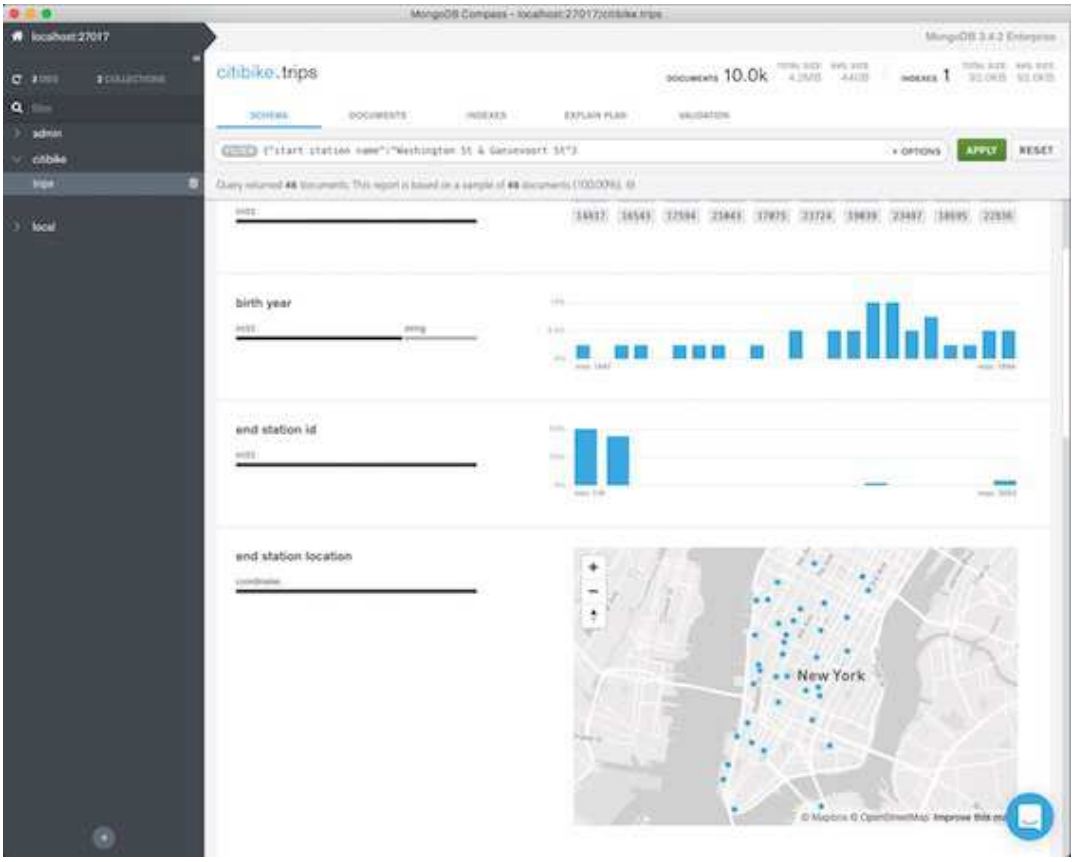
Compass Interactive Queries



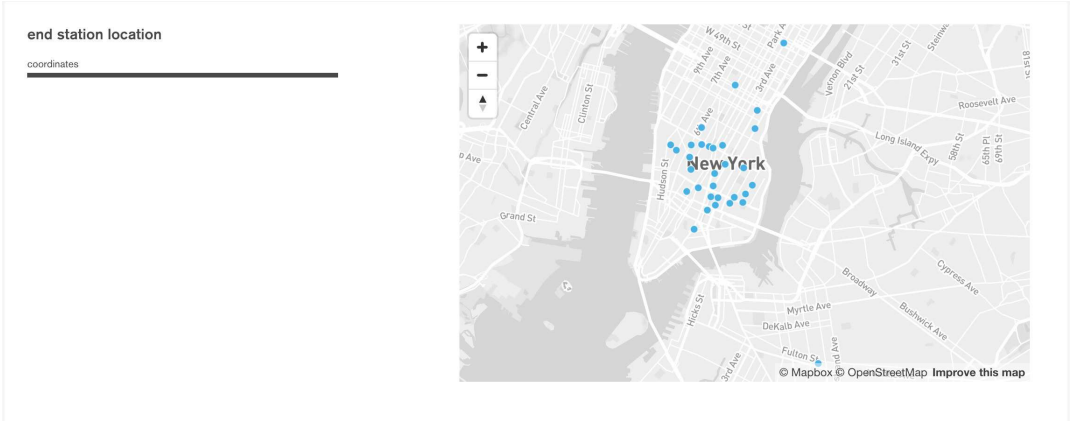
Note:

- In the image, we've selected the largest distribution of "`start station name`" by clicking on the bar
- Students will see the bar highlight orange when it is selected
- After students have made the selection instruct them to click the apply button, illustrating point and click queries

Visualizing geoJSON



Visualizing geoJSON Detail



Note:

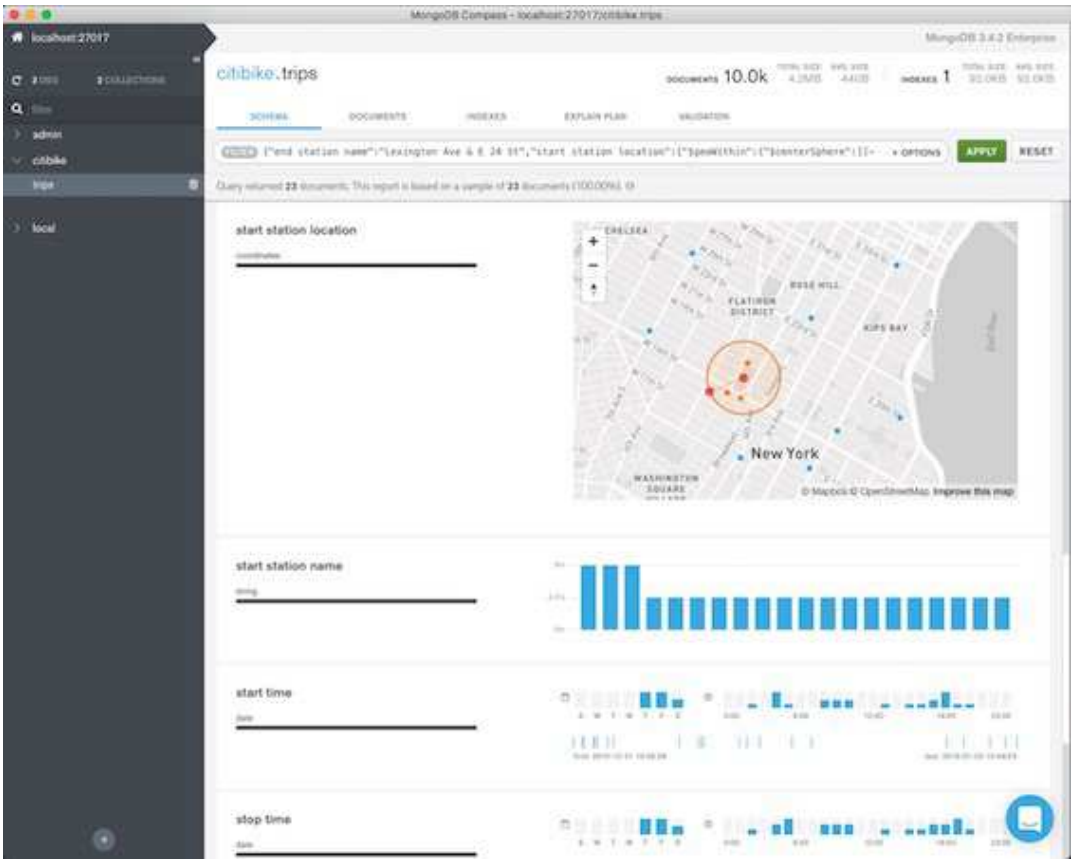
- The image shows the result of the previous point and click query

- It also shows the schema and values of a sample of the returned documents, in this case all of them
- This is because our returned result set is smaller than the default sample size
- The map is fully interactive. Pan and zoom in/out

Interactively Build a geoJSON query

- Select the "`start station location`" visualizer
- Pan the map around and find a location that interests you
- If you are unfamiliar with New York and Manhattan, pan down to Battery Park on the furthest most southwest tip of Manhattan
- Center your mouse in your area of interest, hold shift, click and drag outwards
- You will see an orange circle appear, and the filter/query bar being updated to include a `$geoWithin` query
- When you are satisfied, click apply to see the results

geoJSON Query Results



Note:

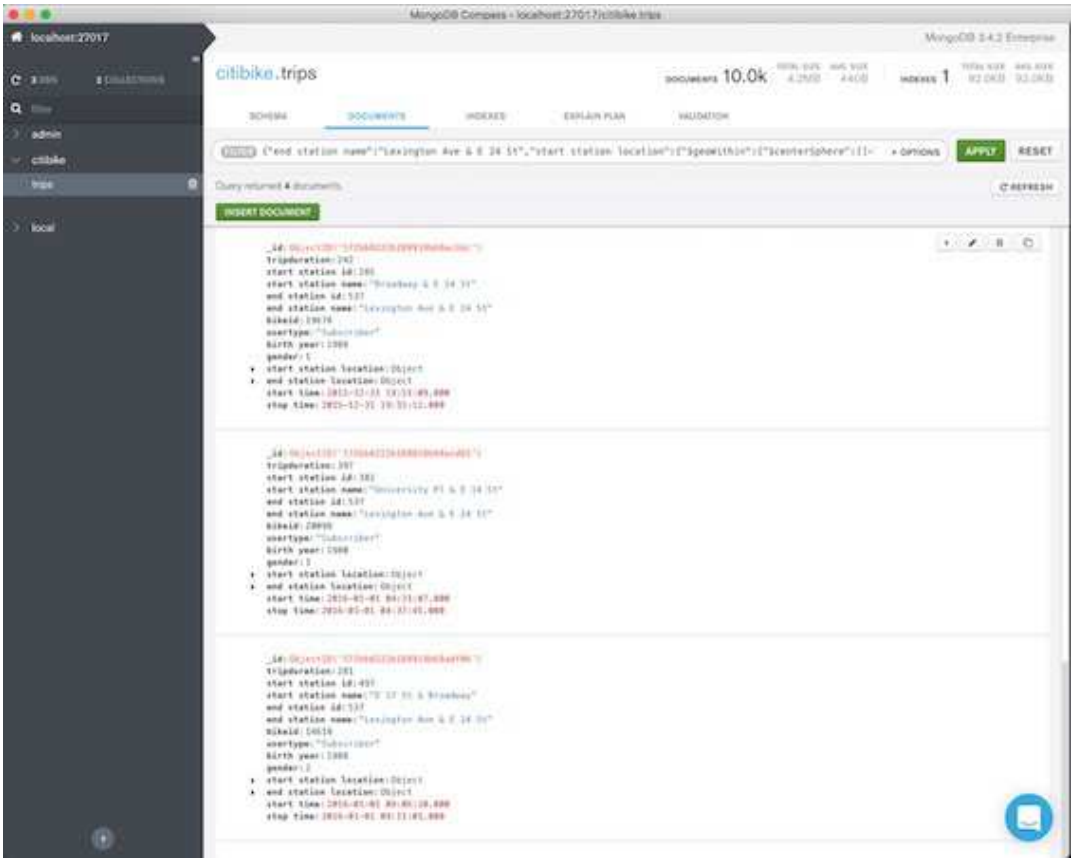
- In this image we've centered on Union Square and expanded our circle to encompass the 3 closest stations

---

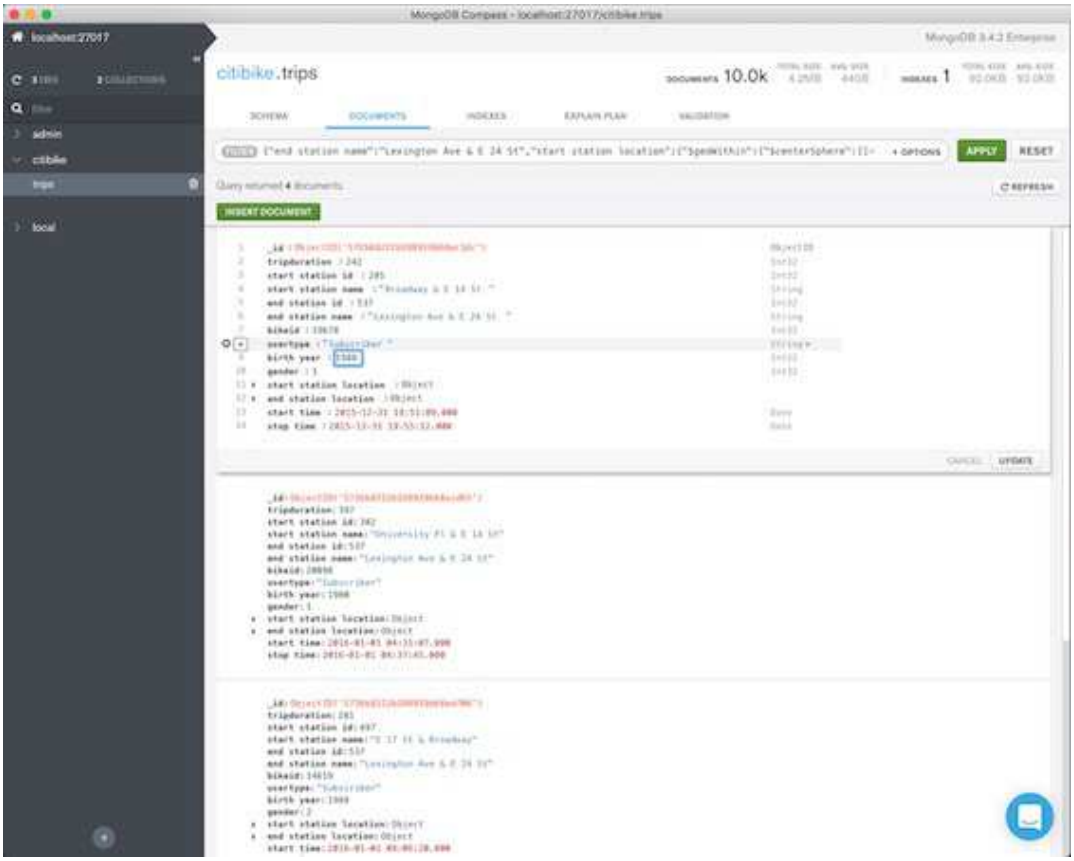
Documents Explorer

- After executing your query, navigate to the `documents` tab
- Mouse over one of the documents
- In the upper right corner of the results window you'll see a toolbar
- This allows us to expand, edit, delete, or clone the document with a single click
- Click the pencil icon

Documents Explorer Example



Updating a Document



Updating Detail

- Document update allows many things, including:
  - Adding or deleting fields
  - Changing the value of fields
  - Changing the type of fields, for example from *Int32* to *Int64* or *Decimal128*

Note:

- If you have covered document validation, have students enable strict validation on one of the fields
- Then, have students try to update that field to a different type

---

1.10 Case Study: Content Management System

Learning Objectives

Upon completing this module, students should understand:

- Various methods for effectively designing the schema for a Content Management System (CMS) in MongoDB
- Optimizations to the schema, and their tradeoffs

Note:

- RDBMS vs. MongoDB
- Pre-joining
- The way comments are handled (another example of pre-aggregation)
- How do we handle the potential for inconsistency between the comments collection and an individual article.
  - Don't pre-aggregate; just do two queries.
    - Do this on read – two queries and, possibly and update or message queues.
    - Handle pre-joining all in one place. Background process identifies top comments.

Building a CMS with MongoDB

- CMS stands for Content Management System.
- nytimes.com[4], cnn.com[5], and huffingtonpost.com[6] are good examples to explore.
- For the purposes of this case study, let's use any article page from huffingtonpost.com[7].

Building a CMS in a Relational Database

There are many tables for this example, with multiple queries required for every page load.

Potential tables

- article
- author
- comment
- tag
- link_article_tag
- link_article_article (related articles)
- etc.

Note: Great time to whiteboard how a CMS would look in an RDBMS, then MongoDB. E.g.:

- Article:

[4] http://nytimes.com
[5] http://cnn.com
[6] http://huffingtonpost.com
[7] http://huffingtonpost.com

    - id
    - headline
    - author_id
    - body
    - summary
    - region
    - published (timestamp)
    - edited (timestamp)
- Author:
    - id
    - name
    - title
- Comment
    - id
    - article_id
    - user_id
    - sentiments
    - responded
    - updated
- User
    - id
    - username
    - user_email
    - user_password
- Tag
    - id
    - label
    - type
- Article_Tag
    - id
    - article_id
    - tag_id
    - tagged (timestamp)
- Article_Article
    - article_newer_id
    - article_older_id

- Front_Page
  - article_id
  - added_timestamp
- Article_Stats
  - views
  - favorited
  - emailed
  - shared
  - commented

Building a CMS in MongoDB

```
{
  "_id" : 334456,
  "slug" : "/apple-reports-second-quarter-revenue",
  "headline" : "Apple Reported Second Quarter Revenue Today",
  "date" : ISODate("2015-03-24T22:35:21.908Z"),
  "author" : {
    "name" : "Bob Walker",
    "title" : "Lead Business Editor"
  },
  "copy" : "Apple beat Wall St expectations by reporting ...",
  "tags" : [
    "AAPL", "Earnings", "Cupertino"
  ],
  "comments" : [
    { "name" : "Frank", "comment" : "Great Story", "date" : ISODate(...) },
    { "name" : "Wendy", "comment" : "+1", "date" : ISODate(...) }
  ]
}
```

Note:

- Spend a lot of time on the pros/cons of these two designs (RDBMS vs Mongo)
- One design optimized for reads, another design optimized for writes

---

Benefits of the Relational Design

With Normalized Data:

- Updates to author information are inexpensive
- Updates to tag names are inexpensive

Benefits of the Design with MongoDB

- Much faster reads
- One query to load a page
- The relational model would require multiple queries and/or many joins.

Every System has Tradeoffs

- Relational design will provide more efficient writes for some data.
- MongoDB design will provide efficient reads for common query patterns.
- A typical CMS may see 1000 reads (or more) for every article created (write).

Optimizations

- Optimizing comments
  - What happens when an article has one million comments?
- Include more information associated with each tag
- Include stock price information with each article
- Fields specific to an article type

Optimizing Comments Option 1

Changes:

- Include only the last N comments in the "main" document.
- Put all other comments into a separate collection
  - One document per comment

Considerations:

- How many comments are shown on the first page of an article?
  - This example assumes 10.
- What percentage of users click to read more comments?

```
{
  "_id" : 334456,
  "slug" : "/apple-reports-second-quarter-revenue",
  "headline" : "Apple Reported Second Quarter Revenue Today",
  ...
  "last_10_comments" : [
```

```
    { "name" : "Frank", "comment" : "Great Story", "date" : ISODate() },
    { "name" : "Wendy",
      "comment" : "When can I buy an Apple Watch?",
      "date" : ISODate() }
  ]
}
```

Optimizing Comments Option 1

Considerations:

- Adding a new comment requires writing to two collections
- If the 2nd write fails, that's a problem.

```
> db.blog.updateOne(
 { "_id" : 334456 },
   { $push: {
       "comments": {
          $each: [ {
            "name" : "Frank",
            "comment" : "Great Story",
            "date" : ISODate()
          } ],
          $sort: { date: -1 },
          $slice: 10 } } } )
> db.comments.insertOne( { "article_id" : 334456, name" : "Frank",
  "comment" : "Great Story", "date" : ISODate() })
```

Optimizing Comments Option 2

Changes:

- Use a separate collection for comments, one document per comment.

Considerations:

- Now every page load will require at least 2 queries
- But adding new comments is less expensive than for Option 1.
  - And adding a new comment is an atomic operation

```
> db.comments.insertOne( { "article_id" : 334456, name" : "Frank",
  "comment" : "Great Story", "date" : ISODate() })
```

Include More Information With Each Tag

Changes:

- Make each tag a document with multiple fields.

```
{
  "_id" : "/apple-reports-second-quarter-revenue",
  ...
  "tags" : [
    { "type" : "ticker", "label" : "AAPL" },
    { "type" : "financials", "label" : "Earnings" },
    { "type" : "location", "label" : "Cupertino" }
  ]
}
```

Include More Information With Each Tag

Considerations:

- $elemMatch is now important for queries

```
> db.article.find( {
  "tags" : {
    "$elemMatch" : {
      "type" : "financials",
      "label" : "Earnings"
    }
  }
} )
```

Include Stock Price Information With Each Article

- Maintain the latest stock price in a separate collection

General Rule:

- Don't de-normalize data that changes frequently!

Fields Specific to an Article Type

Change:

- Fields specific to an article are added to the document.

```
{
  "_id" : 334456,
  ...
  "executive_profile" : {
    "name" : "Tim Cook",
    "age" : 54,
    "hometown" : {
      "city" : "Mobile",
      "state" : "AL"
    },
    "photo_url" : "http://..."
  }
}
```

Note:

- As long as the fields aren't part of the query.
- Ideally, every field that is queried should be indexed.
- Wikipedia also provides many examples of article-specific fields.
- Examples of more fields specific to a particular type of article:
  - http://www.nytimes.com/interactive/2015/07/13/science/space/13after-pluto.html?ref=space
  - http://www.nytimes.com/interactive/2015/06/18/world/europe/encyclical-laudato-si.html?ref=earth

---

Class Exercise 1

Design a CMS similar to the above example, but with the following additional requirements:

- Articles may be in one of three states: "draft", "copy edit", "final"
- History of articles as they move between states must be captured, as well as comments by the person moving the article to a different state
- Within each state, every article must be versioned. If there is a problem, the editor can quickly revert to a previous version.

Note:

- There are a lot of solutions to this problem, one approach is to only insert new documents and remove update functionality:

```
{
  "_id" : ...,
  "headline" : ...,
  ...

  // important fields
  "state" : "draft",
  "version" : 34,
  "modified_date" : ISODate(),
  "modified_author" : "Jason",
  "modified_comments" : "Moving article to draft form, needs rewrite"
}
```

- Now the application must do more work to only load the latest version: db.article.find({ "_id" : 1234 }).sort({ "version" : -1 }).limit(2) // see next comment for the limit(2) explanation
- If two articles are inserted with the same version number (say two people worked on it at the same time), the application should issue a subsequent query without a limit to gather all articles with the same version (maybe there are 10 conflicts). The application should then allow the user to merge those conflicts and commit the next version number. More application logic can be used if conflicts become a larger problem (such as locking the article edits until the conflict is resolved)

Class Exercise 2

- Consult NYTimes, CNN, and huff post for some ideas about other types of views we might want to support.
- How would we support these views?
- Would we require other document types?

Note:

- What indexes are required?
- Do we want to index all versions of our articles?
  - How could we avoid doing so in MongoDB 3.0?
  - How about in MongoDB 3.2?

Class Exercise 3

- Consider a production deployment of our CMS.
- First, what should our shard key be?
- Second, assuming our Primary servers are distributed across multiple data centers in different regions, how might we shard our articles collection to reduce latency?

Note:

- Can pin documents to shard by region with Shard Zones[8]

---

[8] http://docs.mongodb.org/manualtutorial/manage-shard-zone

1.11 Case Study: Social Network

Learning Objectives

Upon completing this module, students should understand:

- Design considerations for building a social network with MongoDB
- Maintaining relationships between users
- Creating a feed service (similar to Facebook's newsfeed)

Note:

- Contrast this with time series and with CMS. In both of those we are aggregating many items into a single document.
- In this case, we are replicating a single event across many documents. We are pre-joining a many-to-many relationship.
- Possibly discuss the Socialite implementation?

Design Considerations

- User relationships (followers, followees)
- Newsfeed requirements

Note: Mention how the Facebook newsfeed works:

- You get stories ranked by importance, determined by an algorithm.
- Examples:
  - Your sister got married
  - Major news story people are linking to
  - Someone RSVP'd for an event you're hosting

User Relationships

What are the problems with the following approach?

```
db.users.find()
{
  "_id" : "bigbird",
  "fullname" : "Big Bird",
  "followers" : [ "oscar", "elmo"],
  "following" : [ "elmo", "bert"],
  ...
}
```

Note: Ask them what the performance impact will be for someone with a lot of followers.

User Relationships

Relationships must be split into separate documents:

- This will provide performance benefits.
- Other motivations:
  - Some users (e.g., celebrities) will have millions of followers.
    - Embedding a "followers" array would literally break the app: documents are limited to 16 MB.
  - Different types of relationships may have different fields and requirements.

Note:

- Relationships from work should have different privacy settings, etc.

User Relationships

```
> db.followers.find()
{ "_id" : ObjectId(), "user" : "bigbird", "following" : "elmo" }
{ "_id" : ObjectId(), "user" : "bigbird", "following" : "bert" }
{ "_id" : ObjectId(), "user" : "oscar", "following" : "bigbird" }
{ "_id" : ObjectId(), "user" : "elmo"", "following" : "bigbird" }
```

Improving User Relationships

Now meta-data about the relationship can be added:

```
> db.followers.find()
{
  "_id" : ObjectId(),
  "user" : "bigbird",
  "following" : "elmo",
  "group" : "work",
  "follow_start_date" : ISODate("2015-05-19T06:01:17.171Z")
}
```

Note:

- Expand further on relationship meta-data, such as how they created the connection, the group the relationship falls into, etc.
- Specific fields can be added per group, e.g. if the group is "school", fields for "graduated" (true/false) and "graduate_school" (true/false) can be easily added

---

Counting User Relationships

- Counts across a large number of documents may be slow
  - Option: maintain an active count in the user profile
- An active count of followers and folowees will be more expensive for creating relationships
  - Requires an update to both user documents (plus a relationship document) each time a relationship is changed
  - For a read-heavy system, this cost may be worth paying

Counting User Relationships

```
> db.users.find()
{
  "_id" : "bigbird",
  "fullname" : "Big Bird",
  "followers" : 2,
  "following" : 2,
  ...
}
```

Note:

- Maintaining counts in the user profile will make it less expensive to display the total number of followers for a user on their profile page (if the profile page is heavily viewed).
- However, it will also mean 3 writes each time one user follows another.
  - One for the user, one for the user it's following, and one for the followers collection entry.

User Relationship Traversal

- Index needed on (followers.user, followers.following)
- For reverse lookups, index needed on (followers.following, followers.user)
- Covered queries should be used in graph lookups (via projection)
- May also want to maintain two separate collections: followers, followees

---

Note:

- If someone asks about $lookup or $graphLookup, tell them not to use it.
  - They're not terribly performant
  - They can't look up documents from sharded collections. Only the initial source collections can be sharded.

User Relationships

- We've created a simple, scalable model for storing user relationships

Building a Feed Service

- Newsfeed similar to Facebook
- Show latest posts by followed users
- Newsfeed queries must be extremely fast

Feed Service Design Considerations

Two options:

- Fanout on Read
- Fanout on Write

Fanout on Read

- Newsfeed is generated in real-time, when page is loaded
- Simple to implement
- Space efficient
- Reads can be very expensive (e.g. if you follow 1 million users)

When to Use Fanout on Read

- Newsfeed is viewed less often than posts are made
- Small scale system, users follow few people
- Historic timeline information is commonly viewed

Fanout on Write

- Modify every users timeline when a new post or activity is created by a person they follow
- Extremely fast page loads
- Optimized for case where there are far less posts than feed views
- Scales better for large systems than fanout on read
- Feed updates can be performed asynchronously

Note:

- This is actually what the large-scale social networks have to do, fanout on read is far too expensive

---

Fanout on Write

```
> db.feed.find({"user : "bigbird"}).sort({"date" : -1})
{
  "_id" : ObjectId(),
  "user" : "bigbird",
  "date" : ISODate("2015-05-19T06:01:17.171Z"),
  "content" : {
    "user" : "cookiemonster",
    "post" : "I love cookies!"
  }
}
```

Fanout on Write

- What happens when Cookie Monster creates a new post for his 1 million followers?
- What happens when posts are edited or updated?

Fanout on Write (Non-embedded content)

```
> db.feed.find({"user : "bigbird"}).sort({"date" : -1})
{
  "_id" : ObjectId(),
  "user" : "bigbird",
  "date" : ISODate("2015-05-19T06:01:17.171Z"),
  "content_id" : ObjectId("...de1")
}

> db.content.find({"_id" : ObjectId("...de1")"})
```

Fanout on Write Considerations

- Content can be embedded or referenced
- Feed items may be organized in buckets per user per day
- Feed items can also be bucketed in batches (such as 100 posts per document)

Fanout on Write

- When the following are true:
  - The number of newsfeed views are greater than content posts
  - The number of users to the system is large
- Fanout on write provides an efficient way to maintain fast performance as the system becomes large.

Class Exercise

Look through a Twitter timeline. E.g. http://twitter.com/mongodb

- Create an example document for a user's tweets
- Design a partial schema just for for a Twitter user's newsfeed (including retweets, favorites, and replies)
- Build the queries be for the user's newsfeed?
  - Initial query
  - Later query if they scroll down to see more
- What indexes would the user need?
- Don't worry about creating the newsfeed documents; assume an application is creating entries, and just worry about displaying it.

Note:

- First consideration is what to optimize for: reads of the user's feed vs writes
- Working through the various pages in the MongoDB Twitter profile should follow a similar theme, design the schema for reads and perform more work in the background every time a new tweet occurs
- Tweets could have the following form

```
{
  _id : 1234
  "username" : "mongodb",
  "tweet" : "The Call for Speakers for #MDBDays closes tomorrow! Submit a talk proposal for a ch
  "date" : ISODate(),
  ...
}
```

- Each user's feed could have the following form (tweets they see when they log in)

```
{
  _id : ObjectId()
  "for_user" : "jason", // feed for the user "jason" following "mongodb"
  "user" : "mongodb",
  "tweet" : "The call for #MDBDays speakers closes tomorrow! Submit a proposal here http://spr.l
  "date" : ISODate(),
  // can duplicate info or refer to the original tweet directly,
  // which would require another query
  "tweet_id" : 1234,
  "retweet" : false,
  "retweet_tweetid" : null // used for displaying retweet status
  ...
}
```

- There would be a number of such documents for each user.
- If this is the schema, you could perform the following query to load the feed:

```
db.user_feeds.find(
  { for_user : "jason" }
).sort( { date : -1 } ).limit(10)
```

- Later loads might look like this:

```
db.user_feeds.find(
  {
    for_user : "jason",
    date : { $lt : <date of last tweet visible in feed> }
  }
).sort( { date : -1 } )
```

- You would want to build the following index:

```
db.user_feeds.createIndex( { for_user : 1, date : 1 } )
```

- For a shard key, this is a good candidate:

```
{
  for_user : 1,
  date : 1
}
```

- Shard key considerations:
  - Putting `for_user` first avoids a monotonically increasing shard key.
  - Putting `date` in the shard key ensures high cardinality
  - Since you're query uses the shard key, it will be targeted
- There would be other collections needed to build the full application:
  - user documents
  - relationships
  - etc.
- Another process would need to go through and delete old documents in the feed.
  - Alternatively, there could be a TTL index on the user_feed's `date` field.

1.12 Case Study: Time Series Data

Learning Objectives

Upon completing this module, students should understand:

- Various methods for effectively storing time series data in MongoDB
- Trade-offs in methods to store time series data

Note:

- Introduce the idea of pre-aggregation
- Discuss doing background processing
- TTL indexes

Time Series Use Cases

- Atlas/Cloud Manager/Ops Manager pre-record a lot of stats in time series fields

---

Note:

- Mention there are a lot of MongoDB customers using MongoDB for time series data
- may want to add Man AHL: https://www.mongodb.com/press/man-ahl-arctic-open-source

---

Building a Database Monitoring Tool

- Monitor hundreds of thousands of database servers
- Ingest metrics every 1-2 seconds
- Scale the system as new database servers are added
- Provide real-time graphs and charts to users

Note:

- Similar to MongoDB Cloud Manager

Potential Relational Design

RDBMS row for client "1234", recording 50k database operations, at 2015-05-29 (23:06:37):

```
"clientid" (integer): 1234
"metric (varchar): "op_counter"
"value" (double): 50000
"timestamp" (datetime): 2015-05-29T23:06:37.000Z
```

Translating the Relational Design to MongoDB Documents

RDBMS Row for client "1234", recording 50k database operations, at 2015-05-29 (23:06:37):

```
{
  "clientid": 1234,
  "metric": "op_counter",
  "value": 50000,
  "timestamp": ISODate("2015-05-29T23:06:37.000Z")
}
```

Problems With This Design

- Aggregations become slower over time, as database becomes larger
- Asynchronous aggregation jobs won't provide real-time data
- We aren't taking advantage of other MongoDB data types

Note:

- If you use Hadoop, or some other process to aggregate data, charts are as real-time as how often the process runs
- "Other MongoDB data types", such as arrays

A Better Design for a Document Database

Storing one document per hour (1 minute granularity):

```
{
  "clientid" : 1234,
  "timestamp": ISODate("2015-05-29T23:06:00.000Z"),
  "metric": "op_counter",
  "values": {
    0: 0,
    ...
    37: 50000,
    ...
    59: 2000000
  }
}
```

Note:

- 0,1,2,3 in the values sub-document represent the minute in the hour

---

Performing Updates

Update the exact minute in the hour where the op_counter was recorded:

```
> db.metrics_by_minute.updateOne( {
  "clientid" : 1234,
  "timestamp": ISODate("2015-05-29T23:06:00.000Z"),
  "metric": "op_counter"},
  { $set : { "values.37" : 50000 } })
```

Performing Updates By Incrementing Counters

Increment the counter for the exact minute in the hour where the op_counter metric was recorded:

```
> db.metrics_by_minute.updateOne( {
  "clientid" : 1234,
  "timestamp": ISODate("2015-05-29T23:06:00.000Z"),
  "metric": "insert"},
  { $inc : { "values.37" : 50000 } })
```

Displaying Real-time Charts

Metrics with 1 minute granularity for the past 24 hours (24 documents):

```
> db.metrics_by_minute.find( {
  "clientid" : 1234,
  "metric": "insert"})
  .sort ({ "timestamp" : -1 })
  .limit(24)
```

Condensing a Day's Worth of Metric Data Into a Single Document

With one minute granularity, we can record a day's worth of data and update it efficiently with the following structure (values.<HOUR_IN_DAY>.<MINUTE_IN_HOUR>):

```
 {
    "clientid" : 1234,
    "timestamp": ISODate("2015-05-29T00:00:00.000Z"),
    "metric": "insert",
    "values": {
      "0": { 0: 123, 1: 345, ..., 59: 123},
      ...
      "23": { 0: 123, 1: 345, ..., 59: 123}
    }
}
```

Considerations

- Document structure depends on the use case
- Arrays can be used in place of embedded documents
- Avoid growing documents (and document moves) by pre-allocating blank values

Class Exercise

Look through some charts in MongoDB's Cloud Manager, how would you represent the schema for those charts, considering:

- 1 minute granularity for 48 hours
- 5 minute granularity for 48 hours
- 1 hour granularity for 2 months
- 1 day granularity forever
- Expiring data
- Rolling up data
- Queries for charts

Note:

- Use the schema below to capture data at one second granularity

```
// document to represent one minute
{
  minute: ISODate("2013-10-10T23:00:00.000Z"),
  type: "memory_used",
  values: {
    0: 999999,
    ...
    37: 1000000,
    38: 1500000,
    ...
    59: 2000000
  }
}
```

- Use background processes to roll up data into minute granularity for each hour.

```
// document to represent one hour, values.MINUTE.SECOND
{
  hour: ISODate("2013-10-10T23:00:00.000Z"),
  type: "memory_used",
  values: {
    0: { 0: 999999, 1: 999999, ..., 59: 1000000 },
    1: { 0: 2000000, 1: 2000000, ..., 59: 1000000 },
    ...,
    58: { 0: 1600000, 1: 1200000, ..., 59: 1100000 },
    59: { 0: 1300000, 1: 1400000, ..., 59: 1500000 }
  }
}
```

- Use background process to roll up data into 1 day granularity forever.

```
// document to represent one hour, values.MINUTE.SECOND
{
  day: ISODate("2013-10-10T00:00:00.000Z"),
  type: "memory_used",
  values: {
    0: { 0: 999999, 1: 999999, ..., 59: 1000000 },
    1: { 0: 2000000, 1: 2000000, ..., 59: 1000000 },
    ...,
```

```
    23: { 0: 1600000, 1: 1200000, ..., 59: 1100000 },
    24: { 0: 1300000, 1: 1400000, ..., 59: 1500000 }
  }
}
```

- Use TTL indexes to expire minutes, and hours.
  - Use a 96 hour window for minutes
  - Use a 4 month window for hours
- Displaying realtime charts now becomes incredibly simple
- Only one document is needed to show activity over an hour (if one hour granularity is used), over 24 hours is 24 documents.
- Comparing different metrics over 24 hours is now very simple, and easier to find correlations
- Queries

```
// 1 minute granularity for 48 hours
db.metrics.aggregate({ match: {})
```

1.13 Case Study: Shopping Cart

Learning Objectives

Upon completing this module, students should understand:

- Creating and working with a shopping cart data model in MongoDB
- Trade offs in shopping cart data models

Shopping Cart Requirements

- Shopping cart size will stay relatively small (less than 100 items in most cases)
- Expire the shopping cart after 20 minutes of inactivity

Advantages of using MongoDB for a Shopping Cart

- One simple document per cart (note: optimization for large carts below)
- Sharding to partition workloads during high traffic periods
- Dynamic schema for specific styles/values of an item in a cart (e.g. “Red Sweater”, “17 Inch MacBook Pro 20GB RAM”)

Modeling the Shopping Cart

```
{
  "_id": ObjectId("55932ef370c32e23e6552ced"),
  "userid": 1234,
  "last_activity": ISODate(...),
  "status" : "active",
  "items" : [
    {
      "itemid": 4567,
      "title": "Milk",
      "price": 5.00,
      "quantity": 1,
      "img_url": "milk.jpg"
    },
    {
      "itemid": 8910,
      "title": "Eggs",
      "price": 3.00,
      "quantity": 1,
      "img_url": "eggs.jpg"
    } ]
}
```

Modeling the Shopping Cart

- Denormalize item information we need for displaying the cart: item name, image, price, etc.
- Denormalizing item information saves an additional query to the item collection
- Use the "last_activity" field for determining when to expire carts
- All operations to the "cart" document are atomic, e.g. adding/removing items, or changing the cart status to "processing"

Add an Item to a User's Cart

```
db.cart.updateOne({
    "_id": ObjectId("55932ef370c32e23e6552ced")
  }, {
    $push : {
      "items" : {
          "itemid": 1357,
          "title": "Bread",
          "price": 2.00,
          "quantity": 1,
          "img_url": "bread.jpg"
      }
    },
    $set : {
      "last_activity" : ISODate()
    }
  }
)
```

Updating an Item in a Cart

- Change the number of eggs in a user's cart to 5
- The positional $ operator identifies an element in an array to update without explicitly specifying the position of the element in the array
- Make sure to update the "last_activity" field

```
db.cart.updateOne({
    "_id": ObjectId("55932ef370c32e23e6552ced"),
    "items.itemid" : 4567
  }, {
    $set : {
      "items.$.quantity" : 5,
      "last_activity" : ISODate()
  }
})
```

Remove an Item from a User's Cart

```
db.cart.updateOne({
    "_id": ObjectId("55932ef370c32e23e6552ced")
  }, {
    $pull : {
      "items" : { "itemid" : 4567 }
    },
    $set : {
      "last_activity" : ISODate()
  }
})
```

Tracking Inventory for an Item

- Use a "item" collection to store more detailed item information
- "item" collection will also maintain a "quantity" field
- "item" collection may also maintain a "quantity_in_carts" field
- When an item is added or removed from a user's cart, the "quantity_in_carts" field should be incremented or decremented

```
{
  "_id": 8910,
  "img_url": "eggs.jpg"
  "quantity" : 2000,
  "quantity_in_carts" : 3
  ...
}
```

Tracking Inventory for a item

Increment “quantity_in_carts”

```
db.item.updateOne(
  { "_id": 8910 },
  { $inc : { "quantity_in_carts" : 1 } } )
```

Decrement “quantity_in_carts”

```
db.item.updateOne(
  { "_id": 8910 },
  { $inc : { "quantity_in_carts" : -1 } } )
```

Using aggregate() to Determine Number of Items Across User Carts

- Aggregate can be used to query for number of items across all user carts

```
// Ensure there is an index on items.itemid
db.cart.createIndex({"items.itemid" : 1})

db.cart.aggregate(
  { $match : { "items.itemid" : 8910 } },
  { $unwind : "$items" },
  { $group : {
      "_id" : "$items.itemid",
      "amount" : { "$sum" : "$items.quantity" }
    } }
)
```

Expiring the Shopping Cart

Three options:

- Use a background process to expire items in the cart collection and update the “quantity_in_carts” field.
- Create a TTL index on “last_activity” field in “cart” collection. Remove the “quantity_in_carts” field from the item document and create a query for determining the number of items currently allocated to user carts
- Create a background process to change the “status” field of expired carts to “inactive”

Shopping Cart Variations

- Efficiently store very large shopping carts (1000+ items per cart)
- Expire items individually

Efficiently Storing Large Shopping Carts

- The array used for the “items” field will lead to performance degradation as the array becomes very large
- Split cart into “cart” and “cart_item” collections

Efficiently Storing Large Shopping Carts: “cart” Collection

- All information for the cart or order (excluding items)

```
{
  "_id": ObjectId("55932ef370c32e23e6552ced"),
  "userid": 1234,
  "last_activity": ISODate(...),
  "status" : "active",
}
```

Efficiently Storing Large Shopping Carts: “cart_item” Collection

- Include “cartid” reference
- Index required on “cartid” for efficient queries

```
{
  "_id" : ObjectId("55932f6670c32e23f119073c"),
  "cartid" : ObjectId("55932ef370c32e23e6552ced"),
  "itemid": 1357,
  "title": "Bread",
  "price": 2.00,
  "quantity": 1,
  "img_url": "bread.jpg",
  "date_added" : ISODate(...)
}
```

Expire Items Individually

- Add a TTL index to the "cart_item" document for the "date_added" field
- Expiration would occur after a certain amount of time from when the item was added to the cart, similar to a ticketing site, or flash sale site

Class Exercise

Design a shopping cart schema for a concert ticket sales site:

- Traffic will dramatically spike at times (many users may rush to the site at once)
- There are seated and lawn sections, only one ticket can be sold per seat/concert, many tickets for the lawn section per concert
- The system will be sharded, and appropriate shard keys will need to be selected

1.14 Lab: Data Model for an E-Commerce Site

Introduction

- In this group exercise, we're going to take what we've learned about MongoDB and develop a basic but reasonable data model for an e-commerce site.
- For users of RDBMSs, the most challenging part of the exercise will be figuring out how to construct a data model when joins aren't allowed.
- We're going to model for several entities and features.

Product Catalog

- **Products.** Products vary quite a bit. In addition to the standard production attributes, we will allow for variations of product type and custom attributes. E.g., users may search for blue jackets, 11-inch macbooks, or size 12 shoes. The product catalog will contain millions of products.
- **Product pricing.** Current prices as well as price histories.
- **Product categories.** Every e-commerce site includes a category hierarchy. We need to allow for both that hierarchy and the many-to-many relationship between products and categories.
- **Product reviews.** Every product has zero or more reviews and each review can receive votes and comments.

Note:

- The most difficult part to this exercise is querying on arbitrary attributes, e.g. macbooks with 11 inch screens. Something like "tags" : [{ "key" : "screen_size", "value" : 11}, { "key" : "brand", "value" : "Apple" }] can be used, but $elemMatch is extremely important here!
- Representing the category hierarchy is also interesting

Product Metrics

- **Product views and purchases.** Keep track of the number of times each product is viewed and when each product is purchased.
- **Top 10 lists.** Create queries for top 10 viewed products, top 10 purchased products.
- **Graph historical trends.** Create a query to graph how a product is viewed/purchased over the past.
- **30 days with 1 hour granularity.** This graph will appear on every product page, the query must be very fast.

Note:

- Use arrays here (for hour/day granularity), and aggregate the arrays for other reports

---

Deliverables

Break into groups of two or three and work together to create the following deliverables:

- Sample document and schema for each collection
- Queries the application will use
- Index definitions

Solution - Collections

- Products
  - Cached reviews
- Reviews
  - Cached comments and ratings
- Comments and ratings
- Prices History
- Views and Purchases

Solution - Considerations

- Caching last reviews, and comments
- Prices: currency, representing decimals,
- Product categories: tree, list of parents
- Keeping track of views, increment by ten every ten views
- Updates to ‘caching pattern’ done in batch

Products

```
{
  _id: ObjectId(),
  name: "MongoDB Logo T-Shirt",
  current_price: 14.99,
  price_history: [ 10.99, 11.99, 12.99, 13.99 ],
  parent_category: "Shirts",
  ancestor_categories: [ "Apparel", "Mens", "Shirts" ],
  tags: [ { key: "color", value: "gray" }, { key: "size", value: "Large" } ],
  top_reviews: [
    { review: "I love this shirt.", upvotes: 30, downvotes: 3, comments: [
      "totally agree",
      "my favorite shirt" ] },
    ...
  ]
}
```

1.15 Lab: Data Model for an "Internet of Things" App

Introduction (1 of 2)

Consier an internet-connected pill bottle.

- It will:
  - Weigh its contents.
  - Log the following when it is opened or closed:
    - the time
    - the weight of its contents
    - how many pills are removed (if it's being closed)
  - Log heartbeats periodically (every 30 minutes)

Introduction (2 of 2)

There will also be a "notification" server.

- It will query periodically to find, for each bottle:
  - Which users are late in taking a pill
  - Which bottles are left open
  - Which bottles are *not* logging heartbeats
- It will then notify the appropriate users
- The time between checks would depend on the frequency of dosage, but typically 1/hour.

Information Outside of the Scope of this Problem

To limit scope of this lab, you should *not* model the following data:

- userdata
- mediation info
- notification records.

You can:

- assume they exist in some other collection
- reference a `bottle_id`, `user_id` or anything else needed

Pill Bottle Operations

Each pill bottle will perform the following queries:

- A heartbeat
  - Frequency: once every 30 minutes
- An operation to log when the bottle is opened or closed.
  - Its contents' weight
  - If closed, how many pills were removed
  - Frequency: Assume an average of 2 times per day

Notifcation Server Operations

The notification server will run queries to determine:

- Whether any given bottle is late in dispensing a dosage
  - Frequency of this depends on the medication
  - Assume an *average* of 1 check per hour
- Whether any given bottle has not sent a heartbeat for over an hour
- Whether any given bottle has been left open

Regardless, the server will also need to know:

- Which user the bottle is associated with
- Which medication the bottle contains

Deliverables

Break into groups of two or three.

Work together to create the following deliverables:

- Sizing estimates, including:
  - Data size for each collection
  - Frequency of requests
- Sample documents for each collection
- Queries (read AND write) that the applications will use
- Index creation commands
- Should you shard a collection? Now? Later?
  - Assume a user base of 10M users with 3 bottles each

Solution

All slides from now on should be shown only after a solution is found by the groups & presented.

Solution - Assumptions

- Assume 10M customers
- Average of 3 bottles per customer
- Average of 2 dosages per day
  - Log bottle open and bottle closed for each
- Log a heartbeat every 30 minutes

Solution - Sizing: Bottle Operations (1 of 2)

Record bottle heartbeats:

- 3 bottles per user
- times 10M users (30 M bottles)
- divide by 1800 sec per heartbeat
- ... so 17K heartbeats per second (inserts)

Solution - Sizing: Bottle Ops (1 of 2)

Record bottles opened/closed:

- 3 bottles per user
- times 10M users
- times 2 dosages per day
- times 2 entries per dosage (open + closed)
- divide by 86400 seconds per day
- ... so ~1400 entries per second (inserts)

Solution - Sizing: Notification Server Ops (1 of 3)

Query for last heartbeat every hour:

- 3 bottles per user
- times 10M users
- divided by 3600 seconds per hour
- ... so 8K checks per second for most recent heartbeat

Solution - Sizing: Notification Server Ops (2 of 3)

Query for Missing Dosages every Hour (check the last 3 dosages)

- 3 bottles per user
- times 10M users
- divided by an average of 3600 seconds between checks
- ... so about 8k checks per second
  - Some more often, some less

Solution - Sizing: Notification Server Ops (3 of 3)

Query for leaving a bottle open

- Assume a peak rate of 1 check every 10 minutes
- 3 bottles per user
- times 10M users
- divided by 600 seconds between checks
- ... so 50K checks per second at peak

Solution - Sizing: Data (1 of 2)

Heartbeats:

- 3 bottles per user
- times 10M users
- times 1000 documents per week
- times 100 bytes per doc
- ... so `3TB per week`
  - plus indexes
  - we will want to shard this
  - and possibly delete old heartbeats

Solution - Sizing: Data (2 of 2)

Opening & Closing Bottles:

- 3 bottles / user
- times 10M users
- times 2 dosages per day
- times 2 entries per dose
- times 100 bytes
- ... so `6GB per week`
- Indexes will take up additional space

Note: Document sizes are approximate, and here to give us an "order of magnitude" sense of things.

Solution - Collections (1 of 2)

Collection: `bottle_actions`

- Bottle it's associated with
- Medication in the bottle
- User who owns the bottle
- Mass of contents
- Estimated number of pills
- Timestamp of measurement
- Action: `"opened"`, or `"closed"`
  - If `"closed"`, also store the number of pills taken while the bottle was open

Solution - Collections (2 of 2)

Collection: `bottle_heartbeats`

- Bottle sending in the heartbeat
- Medication in the bottle
- User who owns the bottle
- timestamp

Solution - Considerations

- Finding open bottles is the *most* common query
  - optimize for that
- The heartbeat inserts are also frequent
  - ... but don't benefit from an index
- Are there health regulations that require us to keep any data?
  - Assume we keep the heartbeats for 2 weeks
  - Keep "actions" (opened/closed) for ever

Solution: Document (1 of 2)

`bottle_action` collection

```
{
  _id : ObjectId(...),
  bottle_id : ObjectId(...),  // Which bottle it refers to
  user_id : "willcross1024",
  medication : "Cordrazine",
  mass_of_contents : 44.25,
  number_of_pills : 25,
  time : ISODate(...),
  action : "closed",  // other possible value: "opened"
  pills_taken : 2  // if action is "closed"
}
```

Solution: Document (2 of 2)

`bottle_heartbeats` collection

```
{
  _id : ObjectId(...),
  bottle_id : ObjectId(...),
  user_id : "willcross1024",
  medication : "Cordrazine",
  time : ISODate(...)
}
```

Note: The students may point out that this is a fairly flat data structure.

- This happens sometimes.
- There is still clever optomizations to be done; see below.

Solution: Pill Bottle Write (1 of 3)

pill bottle opened

```
db.bottle_actions.insertOne(
{
  bottle_id : ObjectId(...),
  user_id : "williamcross1024",
  medication : "Cordrazine",
  mass_of_contents : 44.25,
  number_of_pills : 25,
  time : ISODate(...),
  action : "opened"
}
```

Solution: Pill Bottle Write (2 of 3)

pill bottle closed

```
db.bottle_actions.insertOne(
{
  bottle_id : ObjectId(...),
  user_id : "willcross",
  medication : "Cordrazine",
  mass_of_contents : 40.48,
  number_of_pills : 23,
  time : ISODate(),
  action : "closed",
  pills_taken : 2
} )
```

Solution: Pill Bottle Write (3 of 3)

log the heartbeat

```
db.bottle_heartbeats.insertOne(
{
  bottle_id : ObjectId(...),
  user_id : "williamcross1024",
  medication : "Cordrazine",
  time : ISODate(...)
}
```

Solution: Query (1 of 3)

Check last 3 dosages for a skipped one

```
db.bottle_actions.find(
  {
    bottle_id : ObjectId(...),
    action : "closed",
    pills_taken : { $gt : 0 },
    time : { $gte : <two days ago> }
  },
  { _id : 0, time : 1, pills_taken : 1, user_id : 1, medication : 1 }
)
```

Note:

- This finds the last two days' records
- When the bottle is closed, it records the number of pills taken out while it was open

---

Solution: Query (2 of 3)

Determine if the bottle was left open

```
db.bottle_actions.find(
  {
      bottle_id : ObjectId(...)
  },
  { _id : 0, time : 1, action : 1, user_id : 1, medication : 1 }
).sort( { time : -1 } ).limit(1)
```

If the last action was `"opened"` and the time was >5 minutes ago, it's most likely been left open.

Solution: Query (3 of 3)

Check last heartbeat to see if it has been >60 minutes

```
db.bottle_heartbeats.find(
  {
      bottle_id : ObjectId(...),
      time : { $gte : ISODate( <1 hour ago> ) }
  },
  { _id : 0, time : 1, user_id : 1, medication : 1 }
).sort( { time : -1 } ).limit(1).count()
```

If it's 0, then the bottle is disconnected from wi-fi

Solution: Indexes (1 of 2)

```
db.bottle_actions.createIndex(
  {
      bottle_id : 1, action : 1, time : 1,
      pills_taken : 1, user_id : 1, medication : 1
  },
  { name : "find_missed_dosages" }
)  // optimizes Query 1, AND allows for covered queries

db.bottle_actions.createIndex(
  { bottle_id : 1, time : 1, action : 1, user_id : 1, medication : 1 },
  { name : "find_open_bottles" }
)  // optimizes Query 2 for filtering/sorting, *and* allows for covered queries
```

Note: Each of these indexes is constructed very deliberately.

- `find_missed_dosages`
  - Exact match on `bottle_id` and `action`, so those come first
  - Range match on `time` and `pills_taken`, so those come next
    - `time` is more discriminating, which matters for a range query
  - Projection requires `user_id` and `medication`, so those are added to make it a covered query.
- `find_open_bottles`
  - Exact match on `bottle_id`, so that comes first
  - Sort by `time`, so that comes next
  - Projection also returns `action`, `user_id`, and `medication`, so those fields are included to make it a covered query.

Solution: Indexes (2 of 2)

```
db.bottle_heartbeats.createIndex(
  { bottle_id : 1, time : 1, user_id : 1, medication : 1 },
  { name : "find_last_heartbeat" }
)  // optimizes Query 3 for filtering/sort, *and* allows for covered queries

db.bottle_heartbeats.createIndex(
  { time : 1 }, { expireAfterSeconds : 1209600 },
  { name : "TTL" }
) // TTL Index: Delete the heartbeats after 2 weeks
```

Note:

- `find_last_heartbeat`
  - Exact match on `bottle_id`, so that comes first
  - Sort by `time` (along with a range), so that comes next
  - Add `user_id` and `medication` to make it a covered query.
- `name : "TTL"`
  - Not used in any queries
  - TTL index: the database deletes the heartbeat documents after 1 week

---

Sharding

From a data storage point of view:

- Sharding needed to keep heartbeats over a long period of time (30 TB / week)
- The open & close log entries are just a few terabytes per year; no real need to shard.

From an operation point of view:

- 500K writes per second is a lot for a simple replica set. Either shard, or reduce the number of writes
  - We'll shard the `bottle_heartbeats` collection.

Good shard key for "bottle_heartbeats" collection

Good options:

1. `{ user_id : 1, time : 1 }`
2. `{ bottle_id : 1, time : 1 }`

- We can see that the index named `find_last_heartbeat` has a prefix that matches `B.`, so that will be our shard key.

Note:

- Including the `time` field ensures good cardinality, even if we have a *lot* of documents for each bottle (or each user).
- Putting the time field after a non-monotonically increasing field ensures that there will never be a "hot" server for the inserts.

- The `bottle_id` field *is* an ObjectId, *but* once a bottle is created, its heartbeats all have a fixed `bottle_id`, so this is fine.

Note: "Hot" servers happen in cases where a shard key has values that are monotonically increasing with time. In these cases, one chunk will include the range from `<some point in the past>` to `MaxKey`, and all inserts will go to that server.

Reference: https://docs.mongodb.com/manual/core/sharding-shard-key/#monotonically-changing-shard-keys

---

Find out more
mongodb.com | mongodb.org
university.mongodb.com

Having trouble?
File a JIRA ticket:
jira.mongodb.org

Follow us on twitter
@MongoDBInc
@MongoDB