MongoDB University

MongoDB Data Modeling Workshop

MongoDB Data Modeling Workshop

Release 3.4

MongoDB, Inc.

June 06, 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?

A good schema can mean the difference between:

- Good, or poor performance.
- Doing few queries, or too many.
- Having data in RAM, or touching the disk.
- Having a data store that scales out, and one that doesn't.

What Affects the Speed of Reading Data?

- A server has a limited amount of RAM
  - Going to disk is slower
- Queries take time to make a round trip
  - Many consecutive queries (round trips) are even worse
- 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

Goal: Avoid Touching Data you Won't Use

WiredTiger and MMAPv1 bring complete documents in RAM.

- 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 only looking at the index, without bringing the document in RAM

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.

However ...

Everything we said is true if you try to optimize for Performance

- limited resources on hardware
- sharded clusters

If you have resources to spare and model for simplicity, you will likely try to have few collections, matching the data model in your application

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

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.)

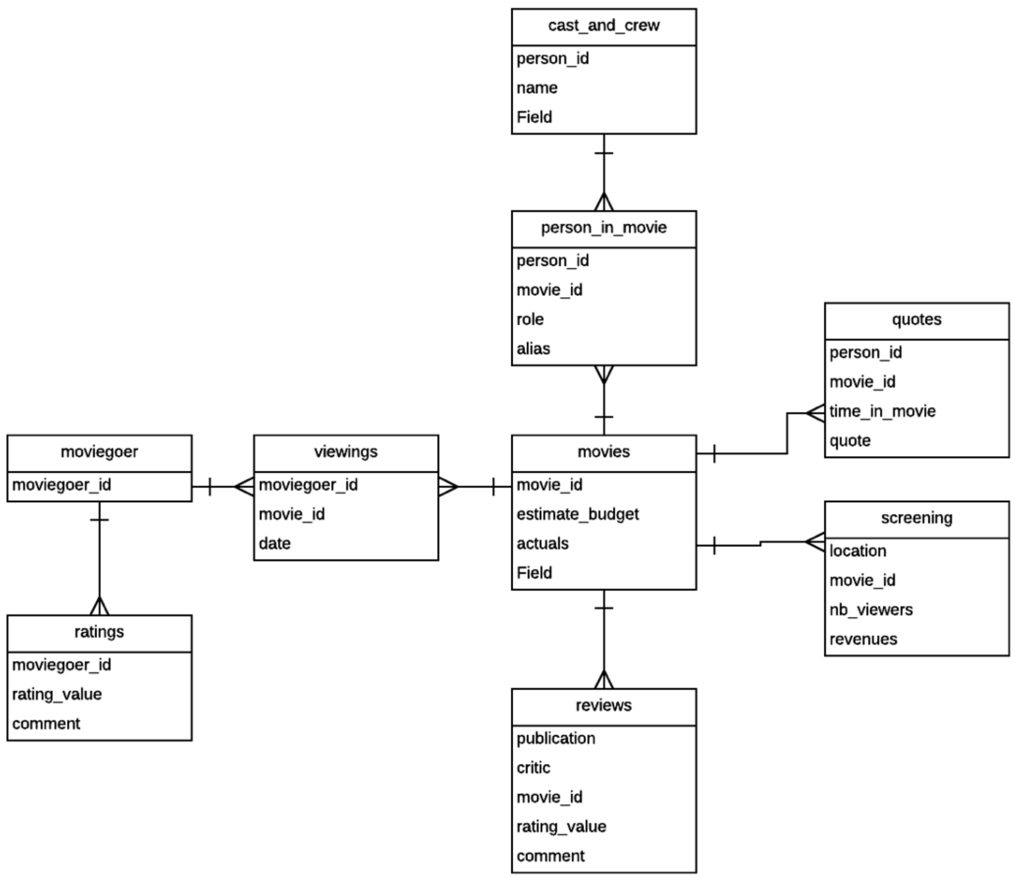
1.3 Schema Design Relationships

Leaning Objectives

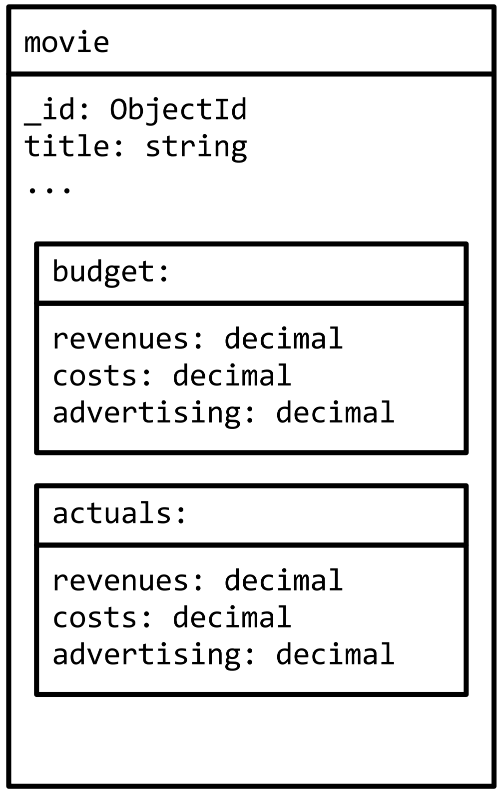
Be able to:

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

Entity Relationship Diagram Example



Relationship - 1 to 1, embedding



Relationship - 1 to 1, reference to the children

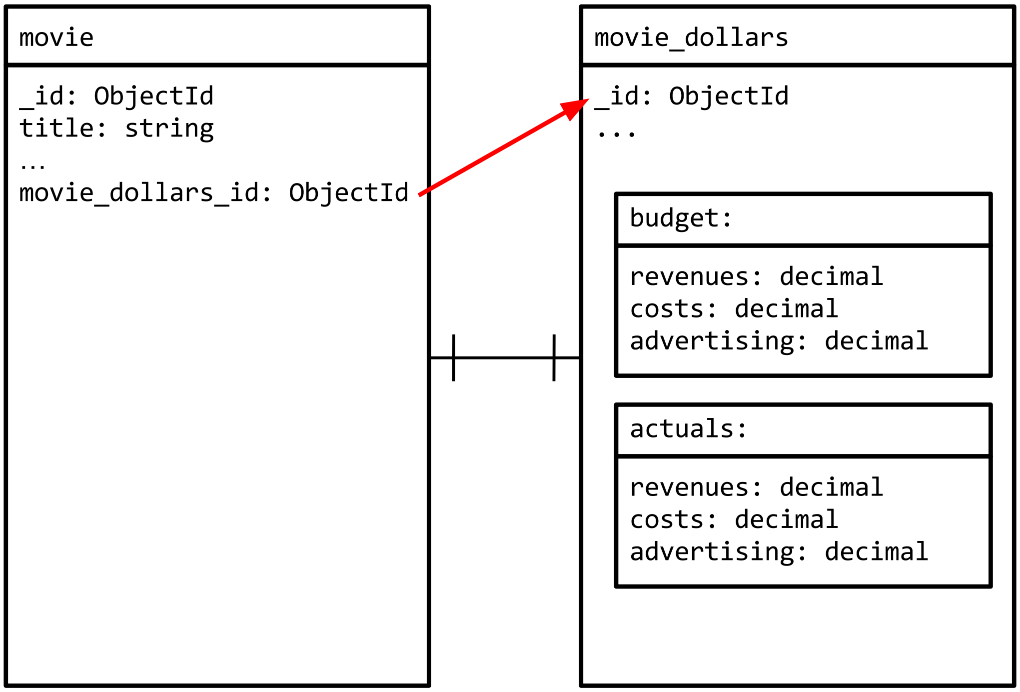
Relationship - 1 to 1, reference to the parent

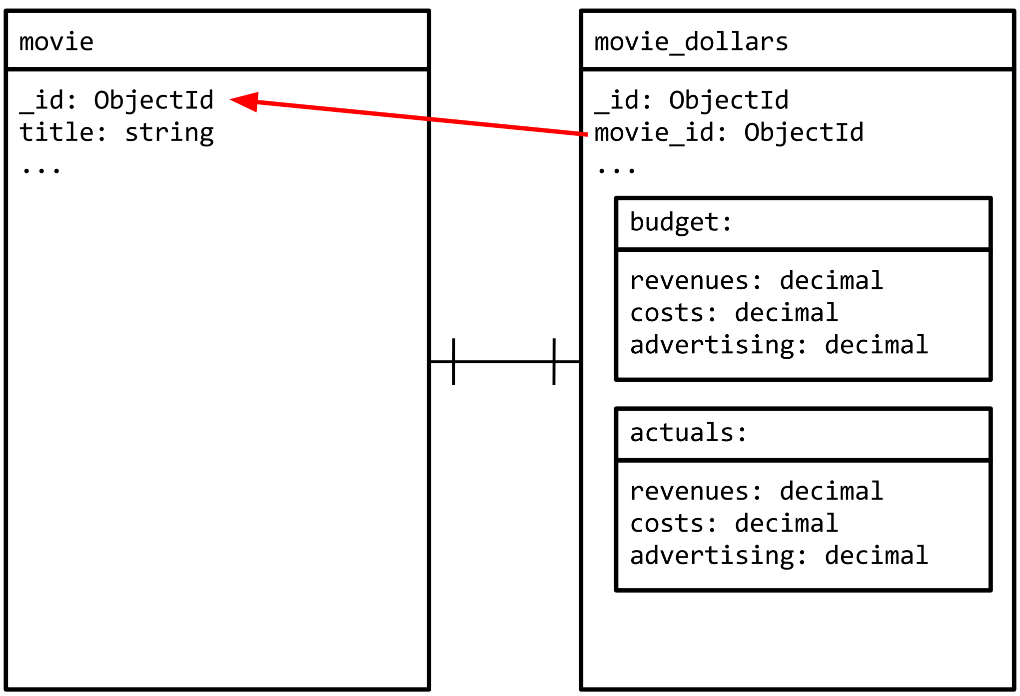
Embed or Store Separately

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

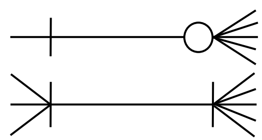
In a 1-N or N-N relationship, do you need to consider:

- 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

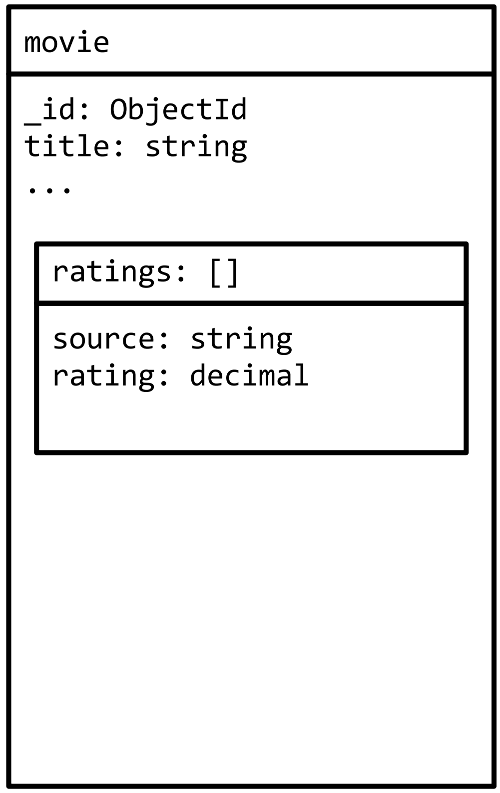
- 1 to 1
- 1 to N
- N to N
- 1 to zillions and N to **zillions**
  - Let's add a new notation for **zillions**!



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

Relationship - 1 to N, embedding



Relationship - 1 to N, reference to the children

Relationship - 1 to N, reference to the parent

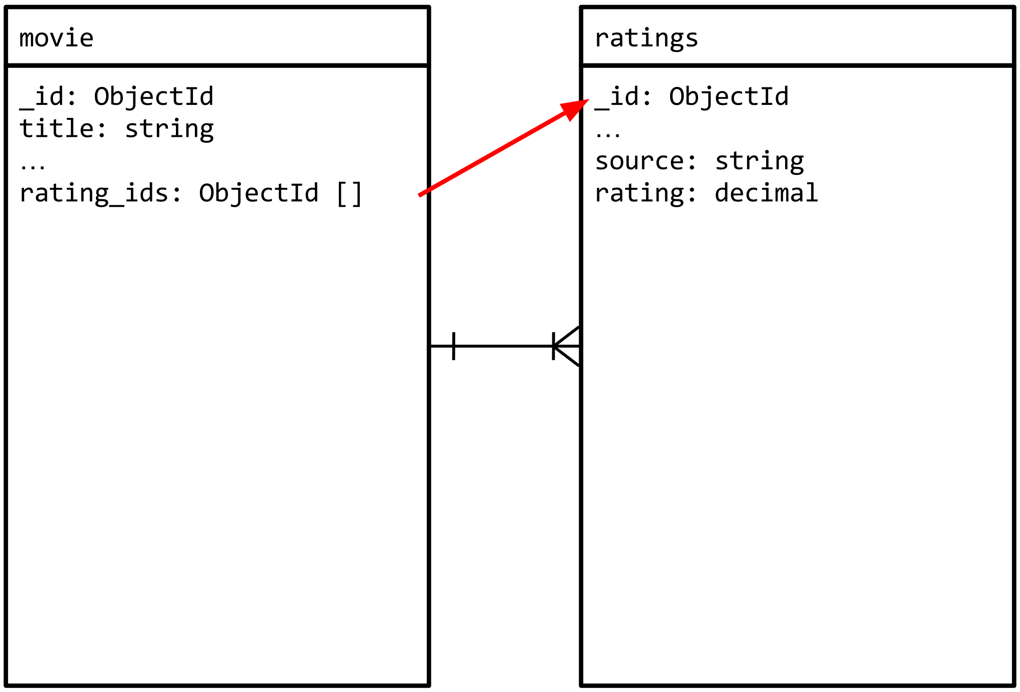
Relationship - N to N, embedding

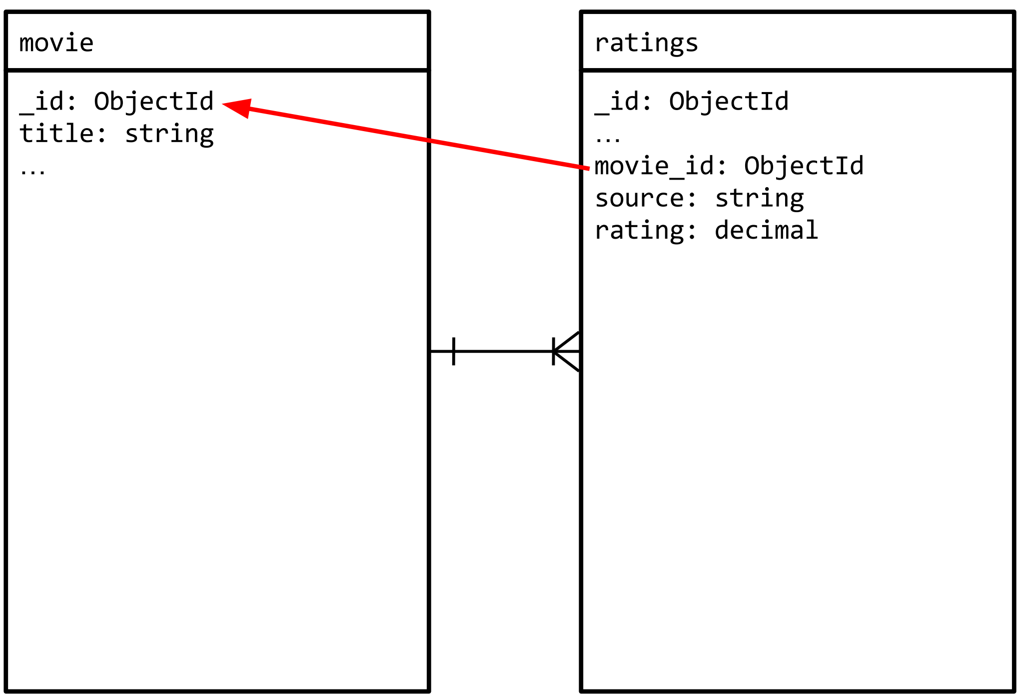
Relationship - N to N, reference to the children

Relationship - N to N, reference to the parent

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

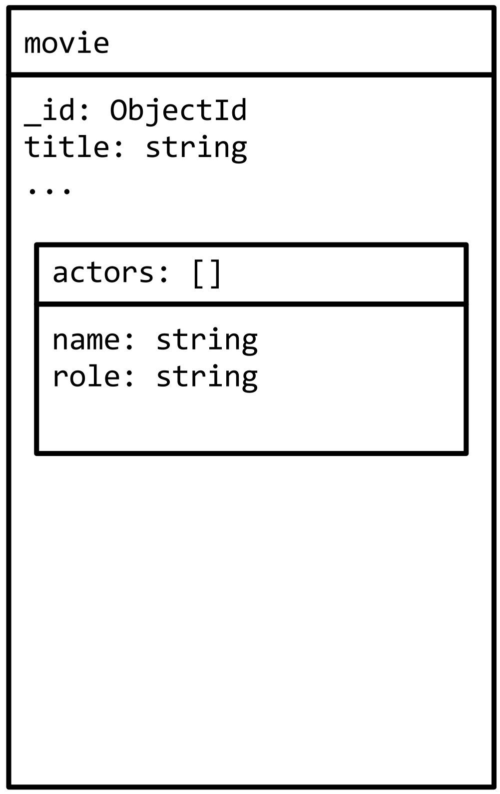


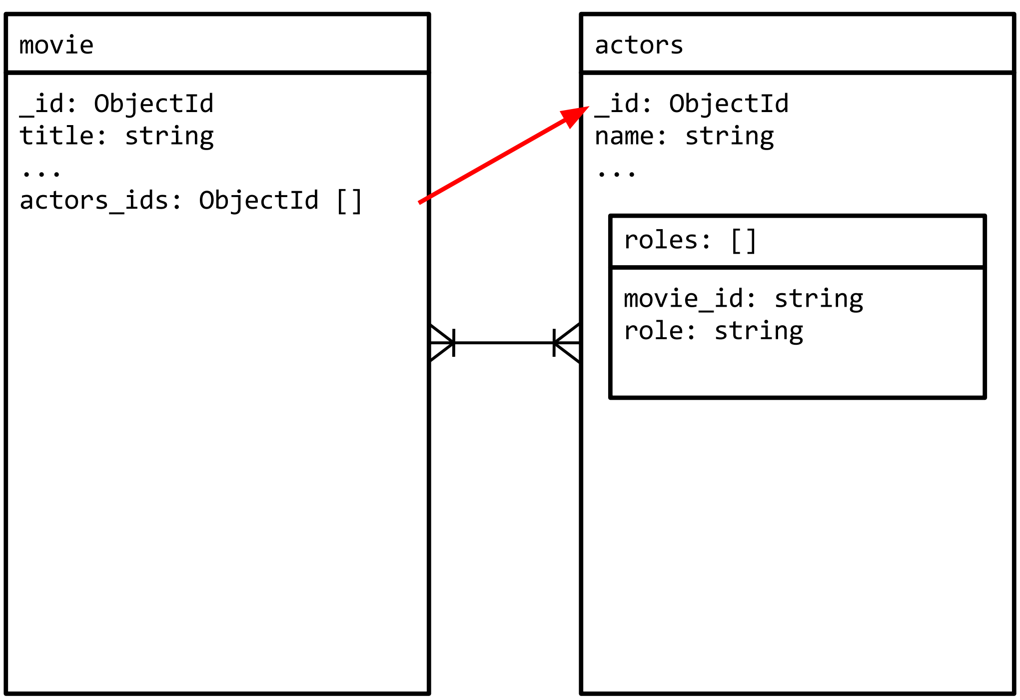


1.4 Schema Design Patterns

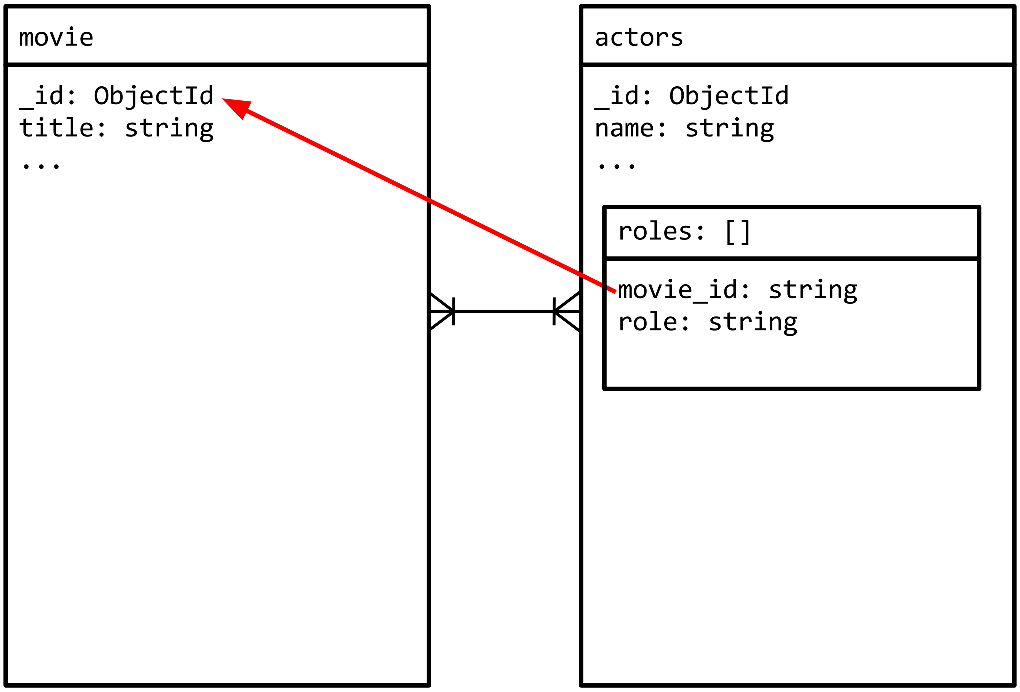
Leaning Objectives

Be able to:





- Identify schema design patterns



- Identify by a common name those patterns
- Use those patterns as building blocks for their solutions

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

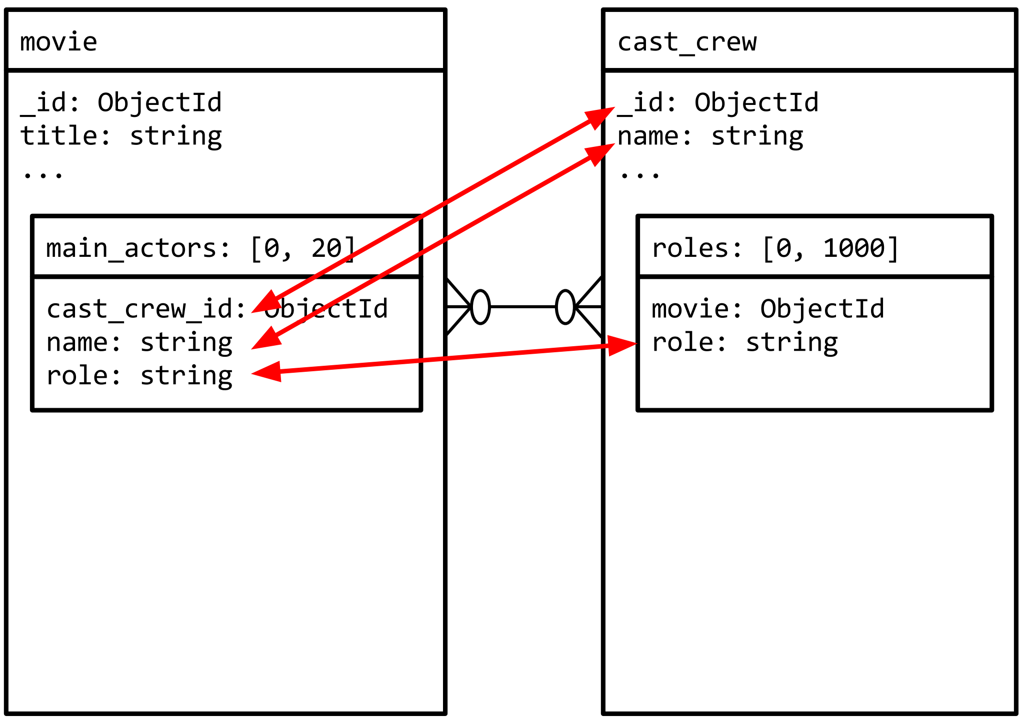
NoSQL patterns ...

- Often address performance concerns
  - Like reducing the number of reads
  - If performance is not an issue, you may want to design for simplicity
- May create duplication in the data
  - important aspect of the duplication of data is how and when you want to handle it
    * is stale data fine for a while?
    * should you batch the updates?

Pattern - Subset

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

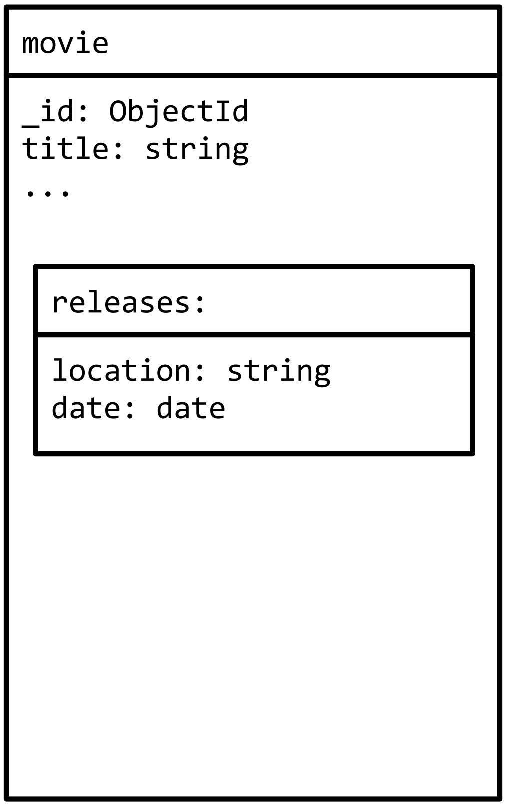
Pattern - Example of Subset



Pattern - Attributes

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

Pattern - Example of Attributes




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

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

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

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

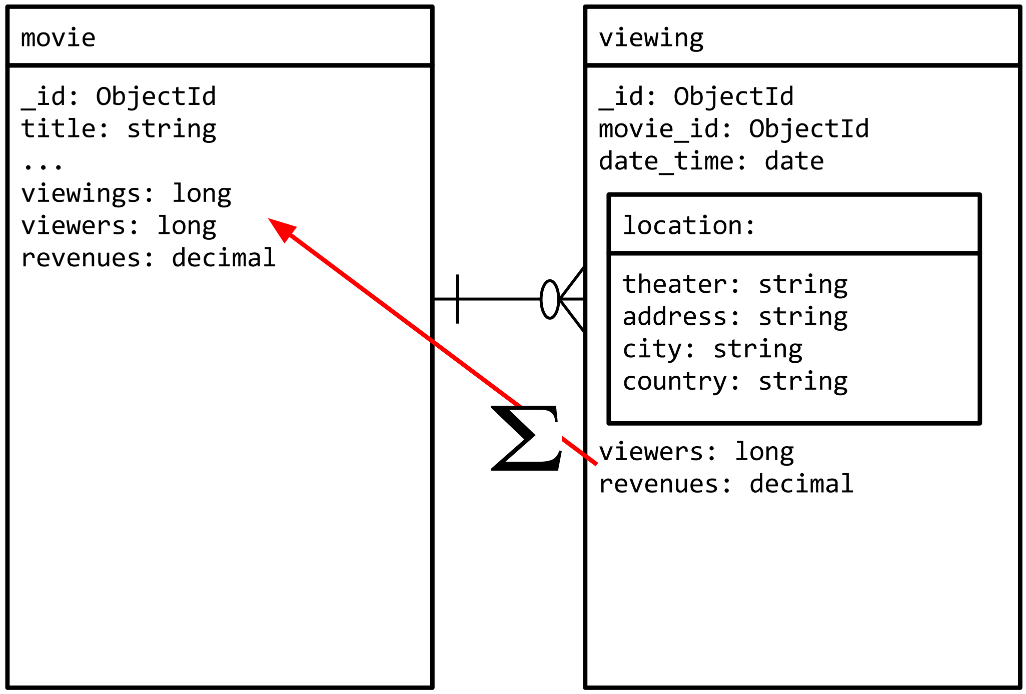
Pattern - Computed

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

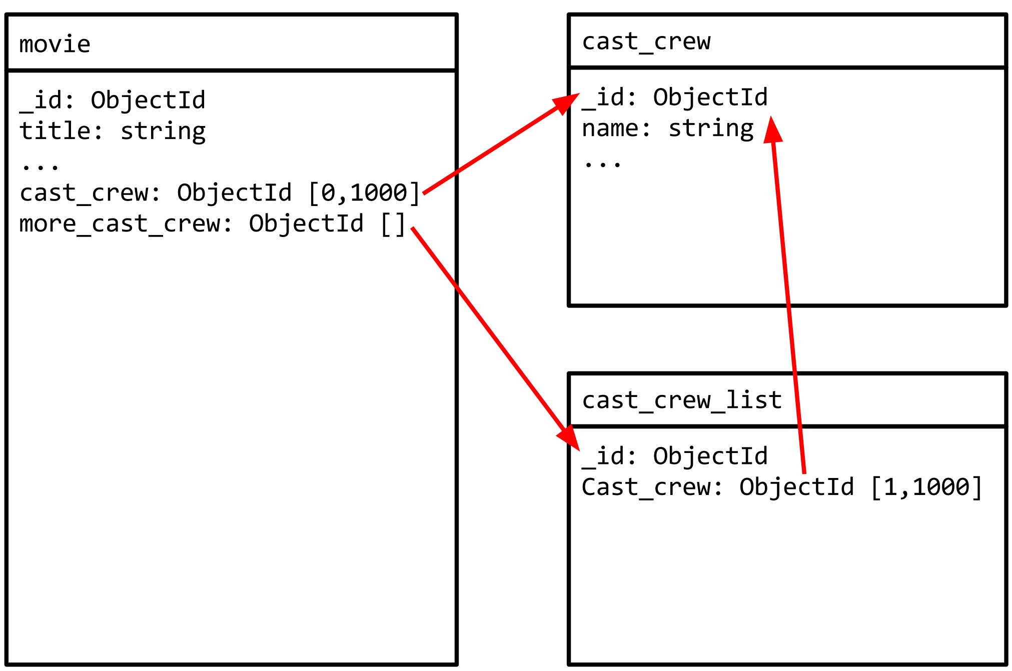
Pattern - Example of Computed

Pattern - Bucket

- Embedding seems preferable to linking
- Too much data to have one document for each piece
- Too much data to fit as one array



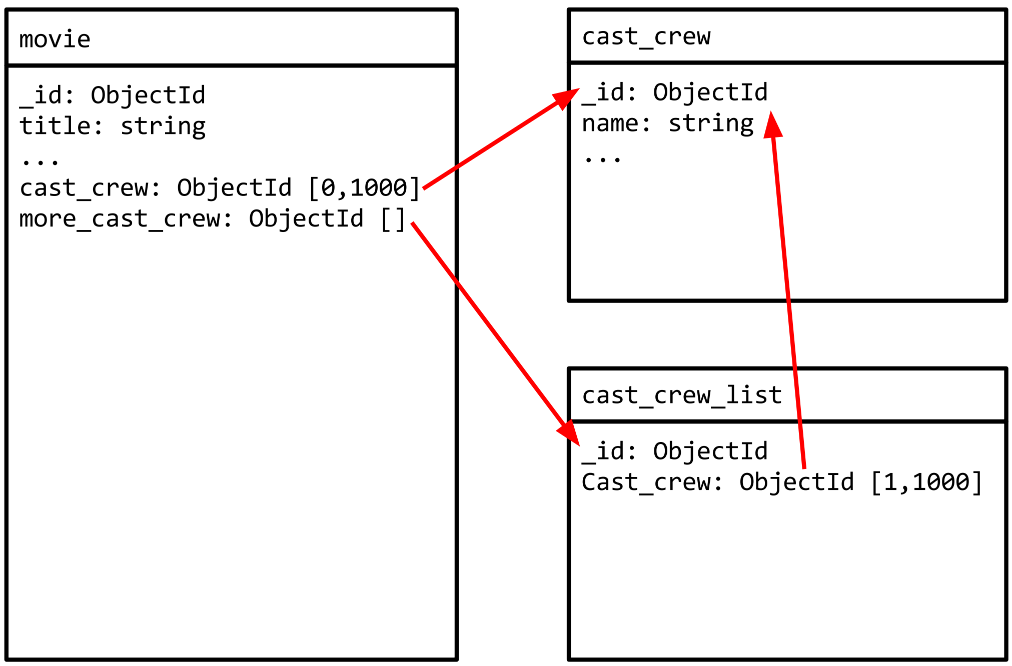
Pattern - Example of Bucket



Pattern - Overflow

- Only a few documents are too big
- Don't want to model the relationship in a different way
- Example:
  - Justin Bieber has too many followers to fit into an array of refs
- Outliers should not drive the design, sacrificing performance for 99.9 percent of the use cases

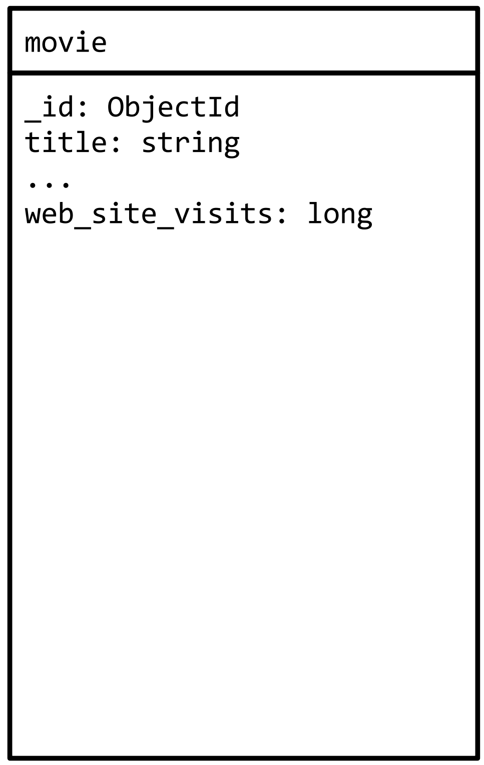
Pattern - Example of Overflow



Pattern - Approximation

- Does not have to be exact
- No source of truth
- Examples:
  - Population of a country
  - Web page hits
    - Only count once in 100, increment by 100

Pattern - Example of Approximation



Pattern - Pre-allocation

- Superset of “Time Series”
- Examples:
  - Metrics collected every minute
  - Seats in a concert

Pattern - Example of Pre-allocation

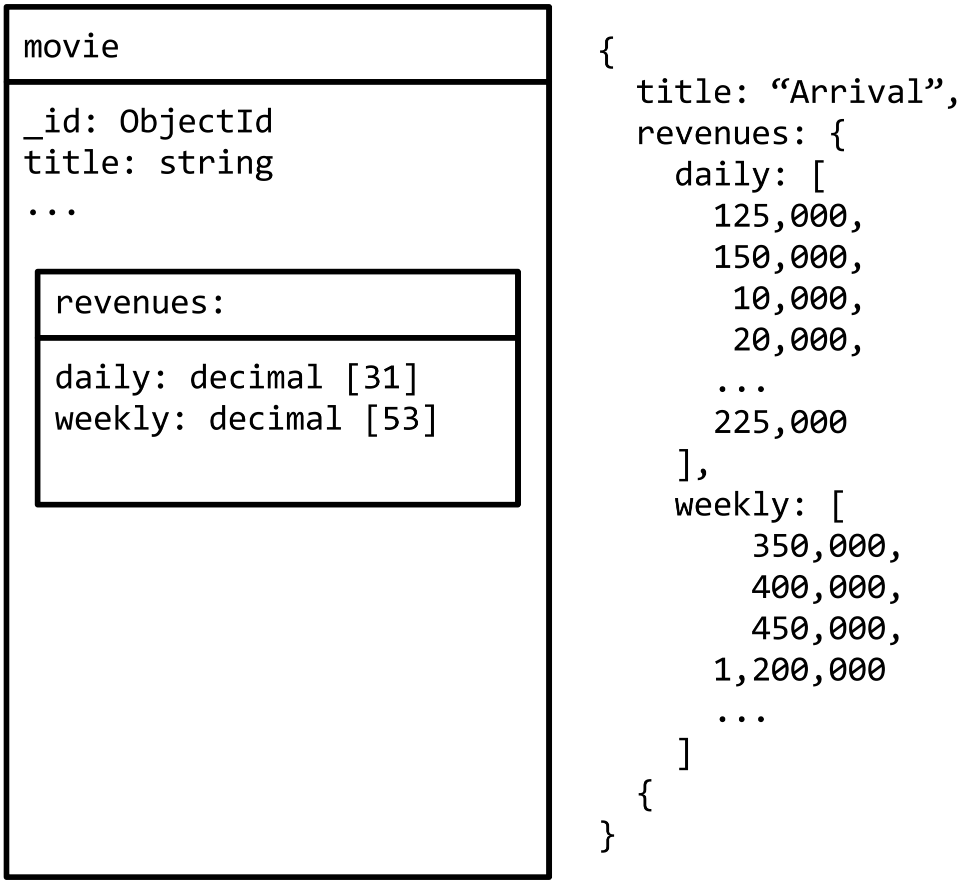
Pattern - Tree as Nodes

Pattern - Tree as Children

Pattern - Tree as 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



Summary of Patterns

- Subset
- Attributes
- Computed
- Bucket
- Overflow
- Approximation
- Pre-allocation
- Tree
  - Node
  - Children
  - Ancestors

movie_genre
_id: string parent: string

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

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

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

movie_genre
_id: string children: string []

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

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

```
movie_genre
-----------------------
_id: string
parent: string
ancestors: string []
```

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

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.

Not for Scaling

- In general, do *not* read from secondaries to provide extra capacity for reads.
- Sharding[1] 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/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

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

Example

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

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

`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"
   }
)
```

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" } )
```

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

Limits of Document Validation

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

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?

Quiz

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

Quiz

On which three databases is document validation not permitted?

1.8 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

Using Compass to Visualize Schema

- Schema tab shows an overview of document schema, to include types
- Based on a $sample[2] of the overall collection, up to 1000 documents
- Fields can be clicked for interactive query and further visualization

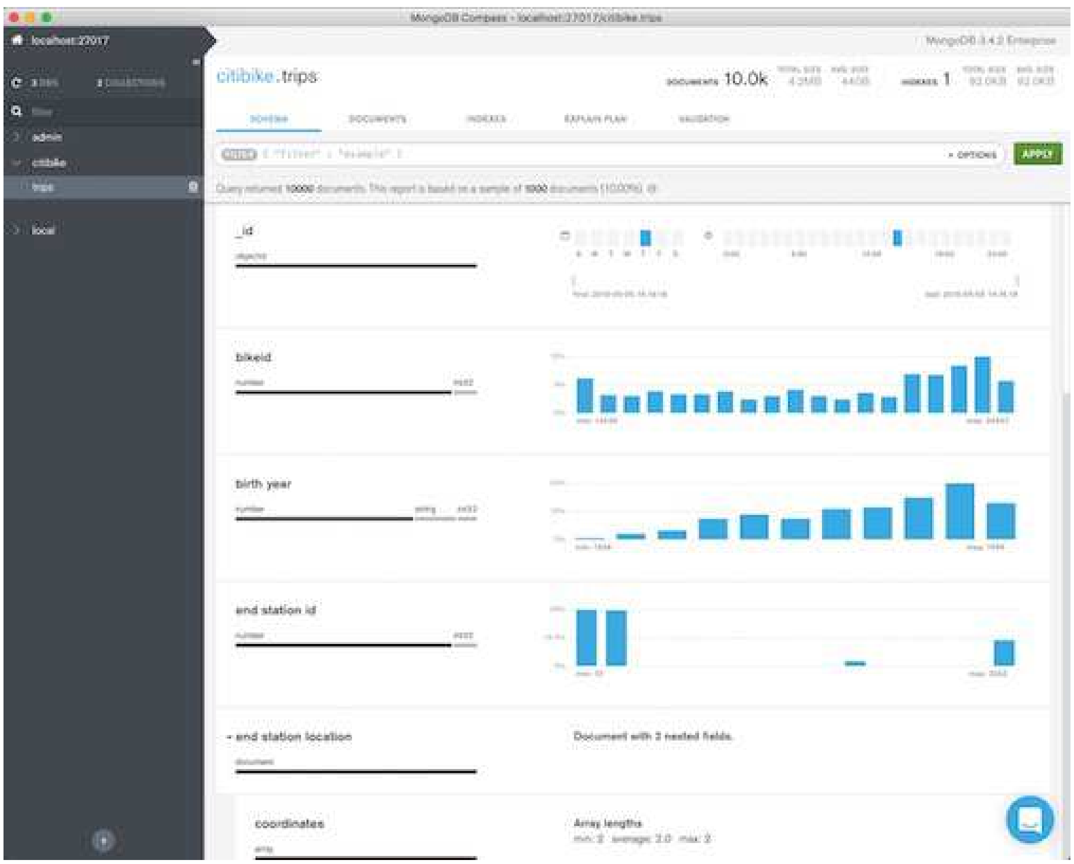
Lesson Setup

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

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

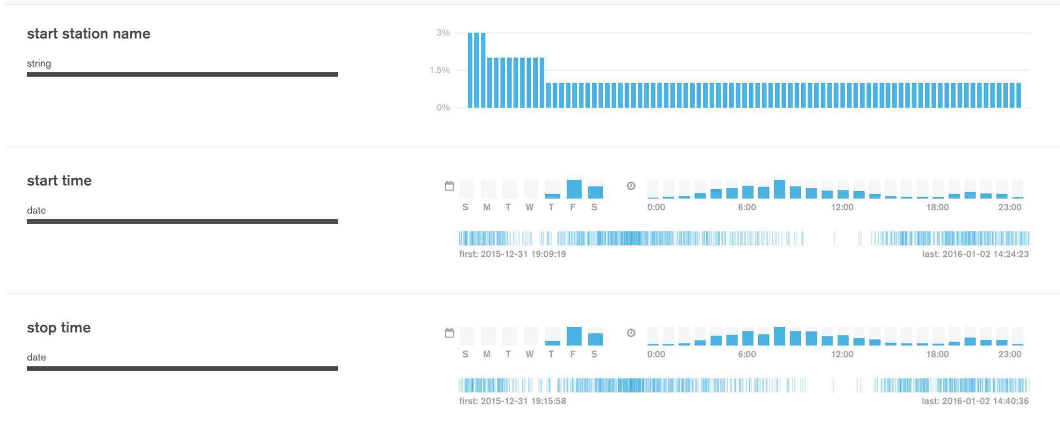
- Connect to your `mongod` with Compass and select the trips collection

Schema Visualization

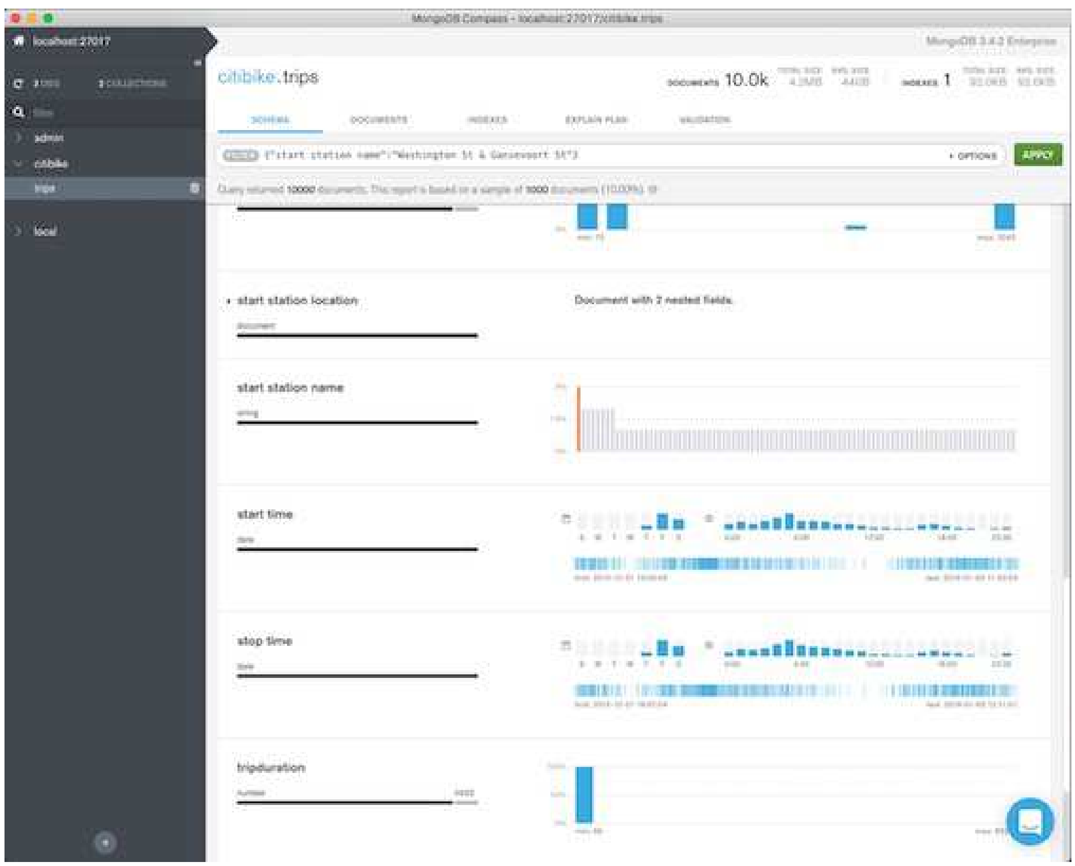


[2] https://docs.mongodb.com/manual/reference/operator/aggregation/sample/

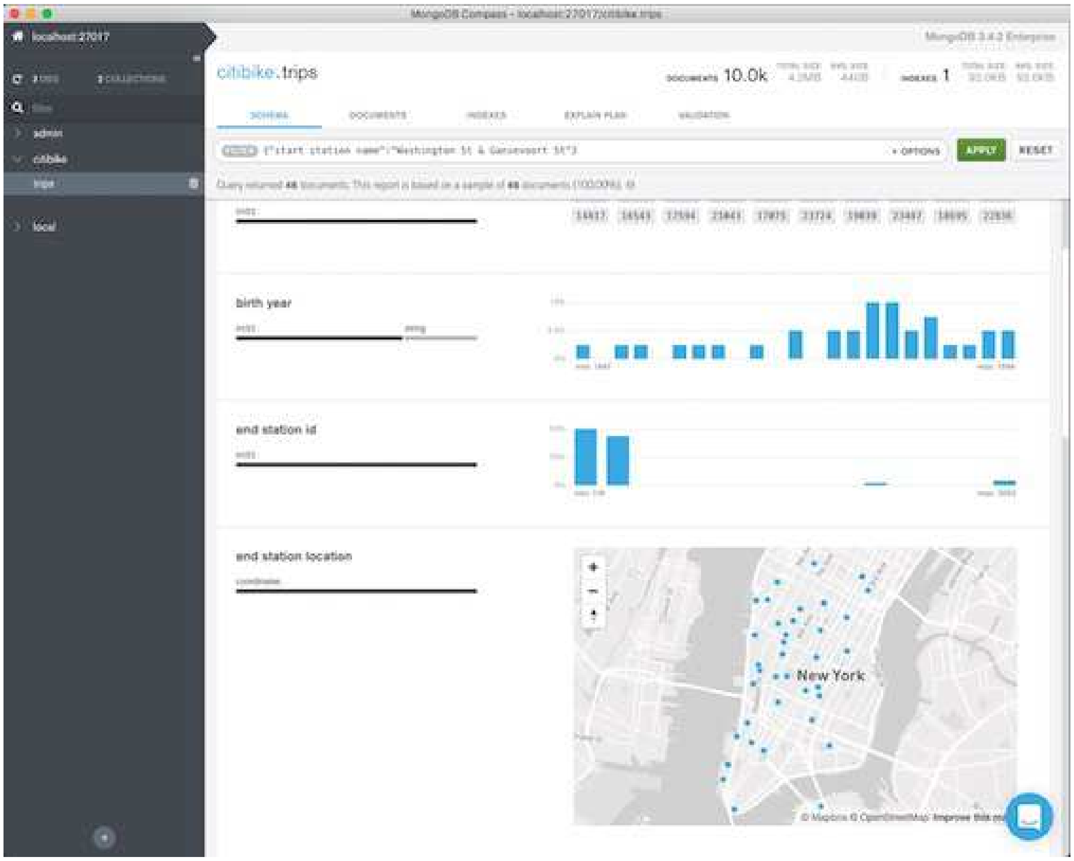
Schema Visualization Detail



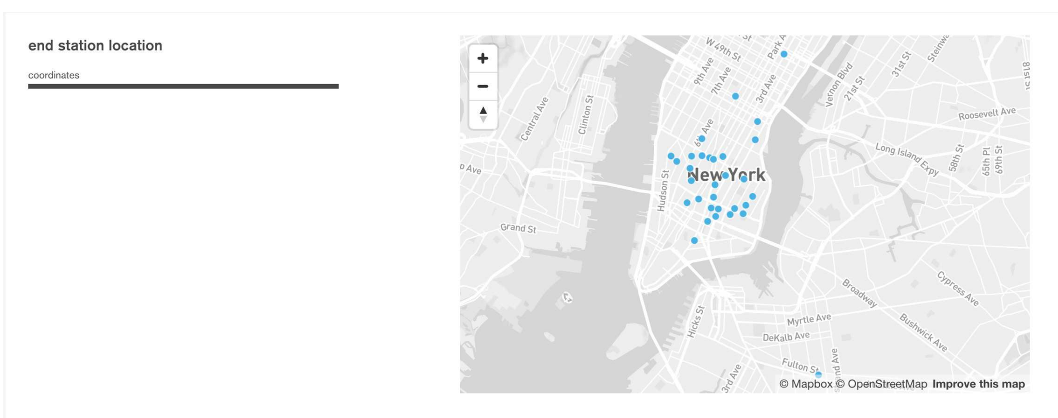
Compass Interactive Queries



Visualizing geoJSON



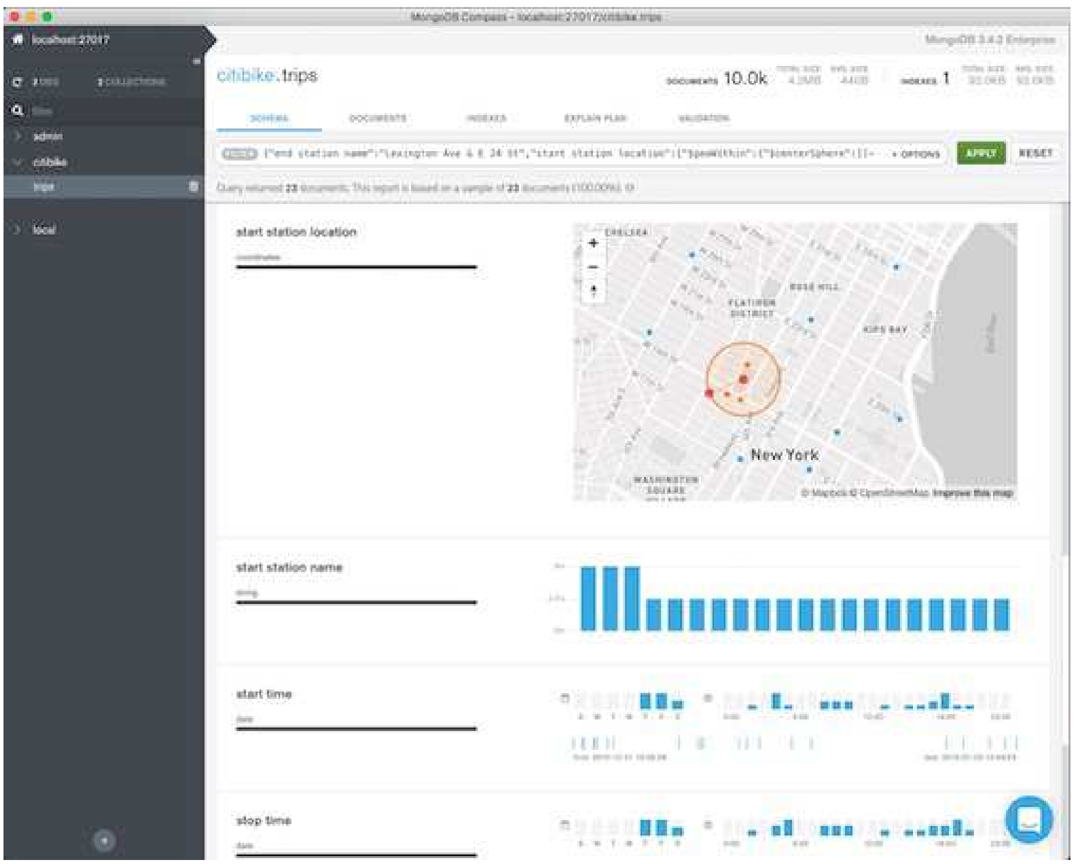
Visualizing geoJSON Detail



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

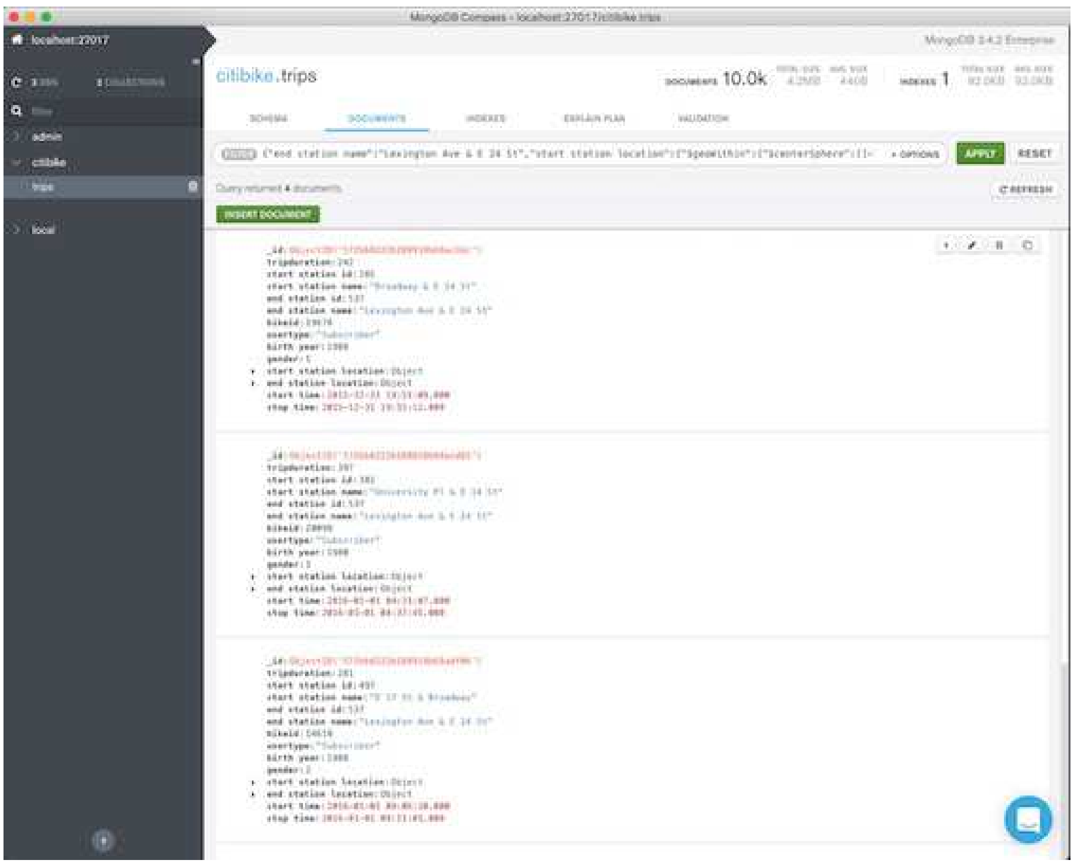
geoJSON Query Results



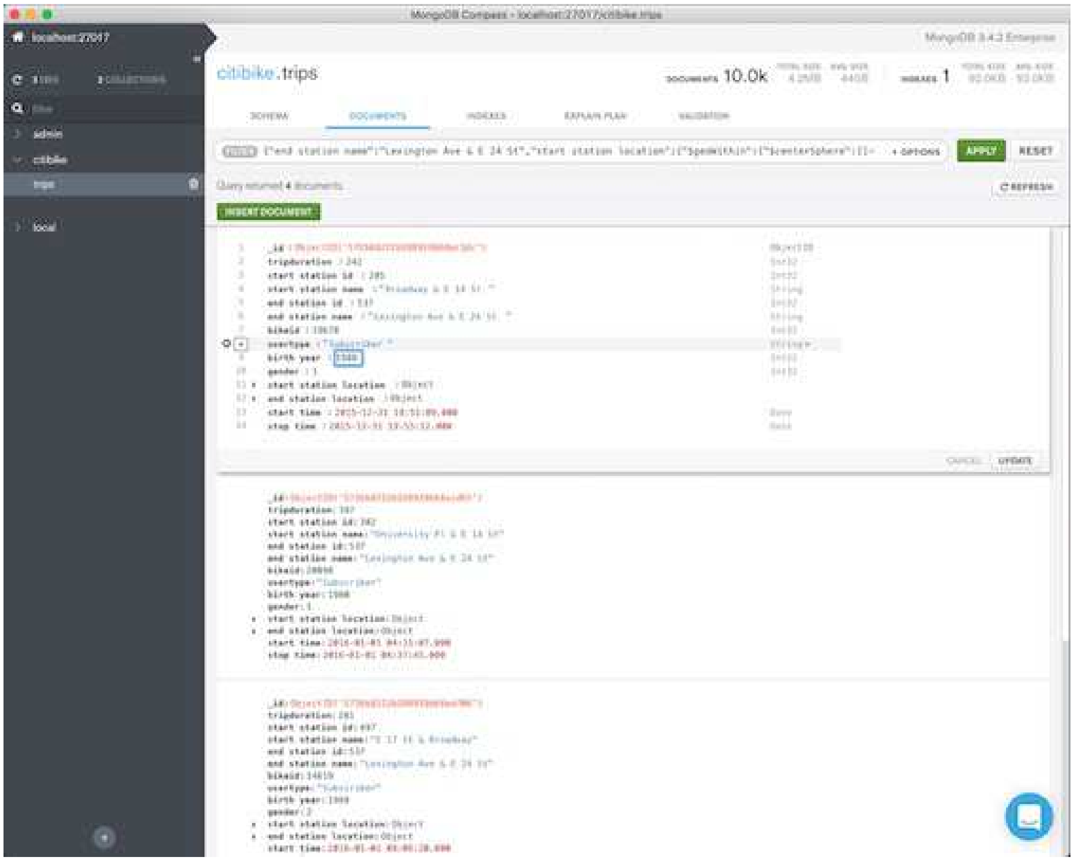
Documents Explorer

- 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*

1.9 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

Building a CMS with MongoDB

- CMS stands for Content Management System.
- nytimes.com[3], cnn.com[4], and huffingtonpost.com[5] are good examples to explore.
- For the purposes of this case study, let's use any article page from huffingtonpost.com[6].

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.

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(...) }
  ]
}
```

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

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://..."
  }
}
```

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.

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?

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?

1.10 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)

Design Considerations

- User relationships (followers, followees)
- Newsfeed requirements

User Relationships

What are the problems with the following approach?

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

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.

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")
}
```

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,
  ...
}
```

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

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

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.

1.11 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

Time Series Use Cases

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

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

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

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
  }
}
```

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

1.12 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

1.13 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.

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.

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

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

1.14 Lab: Data Model for an "Internet of Things" Application

Introduction (1 of 2)

Consider 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:

- user data
- 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

Notification 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

1.15 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

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.

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"}
```

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

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"}
```

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.

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-student/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.

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

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