MongoDB Advanced Developer Training

Release 3.2

MongoDB, Inc.

May 20, 2016

Contents

1 Advanced Schema Design

1.1 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

- Silver Spring Networks, the leading provider of smart grid infrastructure, analyzes utility meter data in MongoDB.
- EnerNOC analyzes billions of energy data points per month to help utilities and private companies optimize their systems, ensure availability and reduce costs.
- Server Density uses MongoDB to collect server monitoring statistics.
- Appboy, the leading platform for mobile relationship management, uses MongoDB to track and analyze billions of data points on user behavior.

Note:

- Mention there are a lot of MongoDB customers using MongoDB for time series data

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: {})
  ```

1.2 Case Study: CMS

Learning Objectives

Upon completing this module, students should understand:

- Various methods for effectively designing a CMS schema 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, cnn.com, and huffingtonpost.com are good examples to explore.
- For the purposes of this case study, let's use any article page from huffingtonpost.com.

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:

- 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 MongoDB Design

- Much faster reads
- One query to load a page
- The relational model would require multiple queries.

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 1 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

- How many comments are shown on the first page of an article?
- 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

- Adding a new comment requires writing to two collections
- Write may fail in-between first and second operation

```
> 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

- Use a separate collection for comments
- Now every page load requires at least 2 queries
- Adding new comments are less expensive

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

Include More Information With Each Tag

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

Include More Information With Each Tag

- $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

Fields specific to an article may be 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 in the query 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 above) 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. In case there is a problem, the editor can quickly revert to the 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.
- Assuming replication across multiple datacenters, how might we shard our articles collection to reduce latency?

Note:

- Can pin documents to shard by region

1.3 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

User Relationships

What are the problems with the following approach?

```
{
  "_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:
  - E.g., celebrity users may have millions of followers.
  - 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
- May want to maintain an active count in the user profile
- An active count of followers and folowees will be more expensive for creating relationships

Counting User Relationships

```
{
  "_id" : "bigbird",
  "fullname" : "Big Bird",
  "followers" : 2,
  "following" : 2
}
```

Note:

- Maintain counts in the user profile to 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)

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

- 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

- Design a schema for the user graph
- Design a schema for a Twitter user's newsfeed (including retweets, favorites, and replies)

Note:

- First consideration is what to optimize for: reads of the user's profile vs writes
- Storing a count of the user's followers removes the need to count followers for each pageload
- 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
- The @MongoDB profile may have the following form

```
{
  "username" : "mongodb",
  "username_display" : "MongoDB",
  "description" : "The world's fastest-growing database.",
  "url" : "http://mongodb.com"
  "tweets" : 7901,
  "following" : 1975,
  "followers" : 186000,
  "favorites" : 1046,
  "lists" : 2,
  ...
}
```

- Tweets from @MongoDB may have the following form (used to construct the @MongoDB profile page)

```
{
  _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 may 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
  ...
}
```

1.4 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 Objectives

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

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

1.5 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

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

Break into groups of two or three and work together to create these deliverables.

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" ] },
    ...
  ]
}
```

2 Application Engineering

2.1 Introduction

What is MongoMart

MongoMart is an on-line store for buying MongoDB merchandise. We'll use this application to learn more about interacting with MongoDB through the driver.

MongoMart Demo of Fully Implemented Version

- View Items
- View Items by Category
- Text Search
- View Item Details
- Shopping Cart

Note:

- Mongomart is in the training repo in the /mongomart directory. Distribute the materials in the /mongomart/student subdirectory to students.
- Ensure you have a fully running version of the instructor MongoMart implementation: /instructor/java/final
- The instructor version requires a three node replica set (uses MAJORITY writes)
- Ensure you've loaded the dataset from /mongomart/student/data and create a text index (as described in the /mongomart/student/data/README.md file)
- DO NOT WALK THROUGH THIS VERSION OF THE SOURCE CODE WITH THE STUDENTS, ONLY THE STUDENT VERSION

View Items

- http://localhost:8080
- Pagination and page numbers
- Click on a category

Note:

- Click around the site to show them how it works, click through various pages and categories (into next slide)

View Items by Category

- http://localhost:8080/?category=Apparel
- Pagination and page numbers
- "All" is listed as a category, to return to all items listing

Text Search

- http://localhost:8080/search?query=shirt
- Search for any word or phrase in item title, description or slogan
- Pagination

Note:

- Search for a few strings, such as "shirt", "mug", "mongodb", etc.

View Item Details

- http://localhost:8080/item?id=1
- Star rating based on reviews
- Add a review
- Related items
- Add item to cart

Note:

- Add reviews, show how stars change, etc.

Shopping Cart

- http://localhost:8080/cart
- Adding an item multiple times increments quantity by 1
- Change quantity of any item
- Changing quantity to 0 removes item

Note:

- Add an item to your cart
- Update the quantity in the cart
- Checkout doesn't work, we are only interested in the cart functionality

2.2 Java Driver Labs (MongoMart)

Introduction

- In this lab, we'll set up and optimize an application called MongoMart. MongoMart is an on-line store for buying MongoDB merchandise.

Note:

- Student skeleton code is located in /student/java
- Ensure data is correctly loaded from /student/data

Lab: Setup and Connect to the Database

- Import the "item" collection to a standalone MongoDB server (without replication) as noted in the README.md file of the /data directory of MongoMart
- Become familiar with the structure of the Java application in /java/src/main/java/mongomart/
- Modify the MongoMart.java class to properly connect to your local database instance

Note:

- Solution in /instructor/java/lab_1
- Please type in solution with the class instead of distributing source code

Lab: Populate All Necessary Database Queries

- After running the MongoMart.java class, navigate to “localhost:8080” to view the application
- Initially, all data is static and the application does not query the database
- Modify the ItemDao.java and CartDao.java classes to ensure all information comes from the database (do not modify the method return types or parameters)

Note:

- Solution in /instructor/java/lab_2
- Please type in solution with the class instead of distributing source code

Lab: Use a Local Replica Set with a Write Concern

- It is important to use replication for production MongoDB instances, however, Lab 1 advised us to use a standalone server.
- Convert your local standalone mongod instance to a three node replica set named “shard1”
- Modify MongoMart’s MongoDB connection string to include at least two nodes from the replica set
- Modify your application’s write concern to MAJORITY for all writes to the “cart” collection, any writes to the “item” collection should continue using the default write concern of W:1

Note:

- Solution in /instructor/java/lab_3
- Please type in solution with the class instead of distributing source code

Lab: Improve How Reviews are Stored and Queried

- Currently, all reviews are stored in an “item” document, within a “reviews” array. This is problematic for the cases when the number of reviews for a product becomes extremely large.
- Create a new collection called “review” and modify the “reviews” array within the “item” collection to only contain the last 10 reviews.
- Modify the application to update the last 10 reviews for an item, the average number of stars (based on reviews) for an item, and insert the review into the new “review” collection

Note:

- Solution in /instructor/java/lab_4
- Please type in solution with the class instead of distributing source code

Lab: Use Range Based Pagination

- Pagination throughout MongoMart uses the inefficient sort() and limit() method
- Optimize MongoMart to use range based pagination
- You may modify method names and return values for this lab

Note:

- Solution in /instructor/java/lab_5
- Please type in solution with the class instead of distributing source code

Lab: Related Items

- Modify each item document to include an array of "related_items" (maximum is 4).
- Related items should include other items within the same category
- Randomized items can be used if there are less than 4.

2.3 Python Driver Labs (MongoMart)

Introduction

- In this lab, we'll set up and optimize an application called MongoMart.
- MongoMart is an on-line store for buying MongoDB merchandise.

Note:

- Student skeleton code is located in /student/python
- Ensure data is correctly loaded from /student/python

Lab: Setup and Connect to the Database

- Import the "item" collection to a standalone MongoDB server (without replication) as noted in the README.md file of the /data directory of MongoMart
- Become familiar with the structure of the Python application in /
- Start the application by running "python mongomart.py", stop it by using ctrl-c
- Modify the mongomart.py file to properly connect to your local database instance

Note:

- Solution in /instructor/python/lab_1
- Please type in solution with the class instead of distributing source code

Lab: Populate All Necessary Database Queries

- After running “python mongomart.py”, navigate to “localhost:8080” to view the application
- Initially, all data is static and the application does not query the database
- Modify the itemDao.py and cartDao.py files to ensure all information comes from the database (you will not need to modify parameters for each DAO method)

Note:

- Solution in /instructor/python/lab_2
- Please type in solution with the class instead of distributing source code

Lab: Use a Local Replica Set with a Write Concern

- It is important to use replication for production MongoDB instances, however, Lab 1 advised us to use a standalone server.
- Convert your local standalone mongod instance to a three node replica set named “rs0”
- Modify MongoMart’s MongoDB connection string to include at least two nodes from the replica set
- Modify your application’s write concern to MAJORITY for all writes to the database

Note:

- Solution in /instructor/python/lab_3
- Please type in solution with the class instead of distributing source code

Lab: Improve How Reviews are Stored and Queried

- Currently, all reviews are stored in an “item” document, within a “reviews” array. This is problematic for the cases when the number of reviews for a product becomes extremely large.
- Create a new collection called “review” and modify the “reviews” array within the “item” collection to only contain the last 10 reviews (sorted by date).
- Modify the application to update the last 10 reviews for an item, the average number of stars (based on reviews) for an item, and insert the review into the new “review” collection

Note:

- Solution in /instructor/python/lab_4
- Please type in solution with the class instead of distributing source code

Lab: Use Range Based Pagination

- Pagination throughout MongoMart uses the inefficient sort() and limit() method
- Optimize MongoMart to use range based pagination
- You may modify method names and return values for this lab

Note:

- Solution in /instructor/python/lab_5
- Please type in solution with the class instead of distributing source code

---

Lab: Related Items

- Modify each item document to include an array of “related_items” (maximum is 4).
- Related items should include other items within the same category
- Randomized items can be used if there are less than 4.