Lecture Notes for Lecture 11 of CS 5200 (Database Management System) for the Summer 1, 2019 session at the Northeastern University Silicon Valley Campus.

MongoDB Data Modeling and Queries

Philip Gust,
Clinical Instructor
Department of Computer Science

Material contained in this presentation is based in part on and uses content from recommended readings for the course.

Lecture 10 Review

- This lecture introduced a different database model that is not based on SQL, often referred to as non-relational or NoSQL.
- We learned that there is no single NoSQL database, but a number of database models that are referred to in this way, including Key-Value Stores, Document Stores, and Object Databases.
- The remained of the semester will focus on MongoDB, a Document Store that is available on many platforms and for a number of the major programming languages.
- The DDL and DML is based on JavaScript, and the document model is based on JSON objects that include key-value pairs of basic types, arrays, and embedded documents.
- Documents are identified by keys that are either manually assigned or automatically assigned when the document is created

Today's Topics

- In this lecture we will present techniques for data modeling in MongoDB, including document structure, atomicity of write operations, document growth, data use and performance.
- Next, we will look at how to perform queries on documents in both the Mongo shell and the Java APIs, and how to control the order and which fields are returned.
- Finally, we will discuss several options for graphical development tools to develop MongoDB databases and applications.

MongoDB Data Modeling and Queries

- Data in MongoDB has a flexible schema. Unlike SQL databases, where you determine and declare a table's schema before inserting data, MongoDB's collections do not enforce document structure.
- This flexibility facilitates the mapping of documents to an entity or an object. Each document matches data fields of the represented entity, even if the data has substantial variation. In practice, however, the documents in a collection share a similar structure.
- The key challenge in data modeling is balancing the needs of the application, the performance characteristics of the database engine, and the data retrieval patterns.
- When designing data models, always consider the application usage of the data (i.e. queries, updates, and processing of the data) as well as the inherent structure of the data itself.

MongoDB Data Modeling and Queries

Document Structure

- The key decision in designing data models for MongoDB applications revolves around the structure of documents and how the application represents relationships between data.
- There are two tools that allow applications to represent these relationships: embedded and normalized data models.
  - ***Embedded data model*** captures relationships between data by storing related data in a single document structure. MongoDB documents make it possible to embed document structures in a field or array within a document.
  - ***Normalized data model*** stores relationships between data by including links or references from one document to another. Applications resolve references to related data. Broadly, these are normalized data models.

MongoDB Data Modeling and Queries

Embedded Data Model

- Embedded data models allow applications to store related pieces of information in the same database record.

MongoDB Data Modeling and Queries

Embedded Data Model

- With MongoDB, you embed related data in a single structure or document. These schema are generally known as “denormalized” models.
- As a result, applications may need to issue fewer queries and updates to complete common operations.
- In general, use embedded data models when:
  - you have “contains” relationships between entities
  - you have one-to-many relationships between entities. In these relationships the “many” or child documents always appear with or are viewed in the context of the “one” or parent documents.

MongoDB Data Modeling and Queries

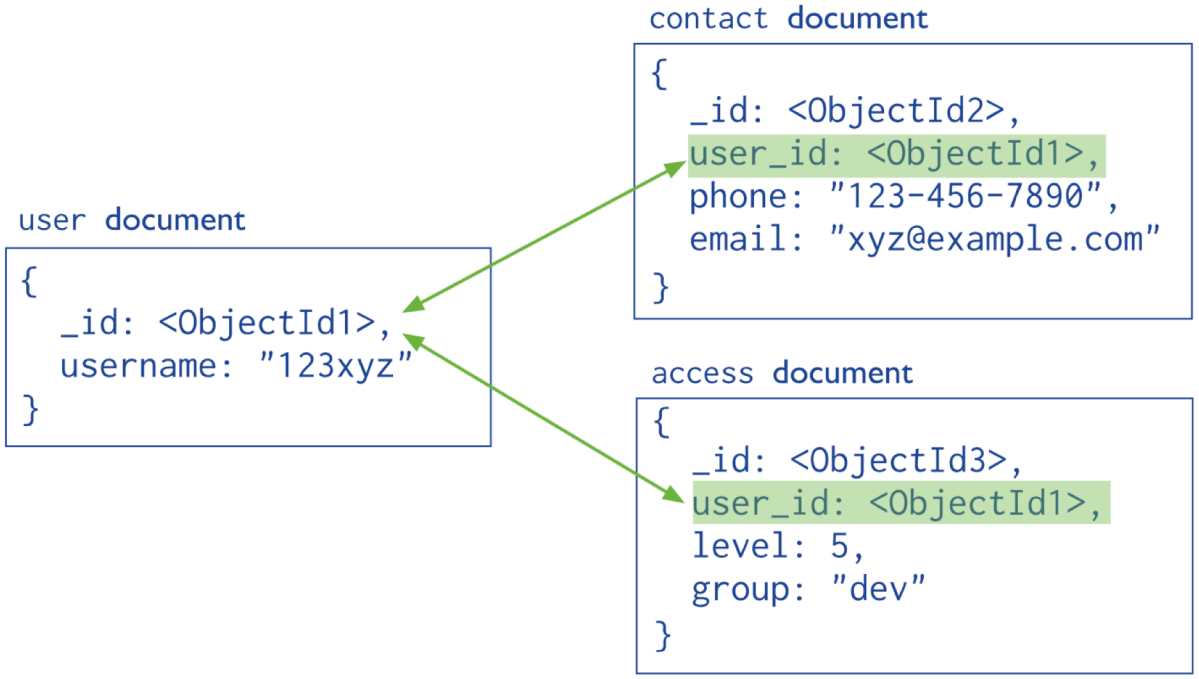
Embedded Data Model

- Embedding can provide better performance for read operations, as well as the ability to request and retrieve related data in a single database operation.
- Embedded data models make it possible to update related data in a single atomic write operation.
- However, embedding related data in documents may lead to situations where documents grow after creation.
- With the MMAPv1 storage engine, document growth can impact write performance and lead to data fragmentation.
- To interact with embedded documents, use dot notation to “reach into” embedded documents.

MongoDB Data Modeling and Queries

Normalized Data Model

- Normalized data models describe relationships using references between documents.



MongoDB Data Modeling and Queries

Normalized Data Models

- In general, use normalized data models:
  - when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.
  - to represent more complex many-to-many relationships
  - to model large hierarchical data sets.
- References provides more flexibility than embedding. However, client-side applications must issue follow-up queries to resolve the references.
- In other words, normalized data models can require more round trips to the server.

MongoDB Data Modeling and Queries

Atomicity of Write Operations

- In MongoDB, write operations are atomic at the document level, and no single write operation can atomically affect more than one document or more than one collection.
- A *denormalized* data model with embedded data combines related data for a represented entity in a single document. This facilitates atomic write operations since a single write operation can insert or update the data for an entity.
- Normalizing the data would split the data across multiple collections and would require multiple write operations that are not atomic collectively.
- However, schemas that facilitate atomic writes may limit ways that applications can use the data or modify it.

MongoDB Data Modeling and Queries

Document Growth

- Some updates, such as pushing elements to an array or adding new fields, increase a document's size.
- For the MMAPv1 storage engine, if the document size exceeds the allocated space for that document, MongoDB relocates the document on disk.
- When using the MMAPv1 storage engine, growth consideration can affect the decision to normalize or denormalize data.
- See "Document Growth Considerations" in the MongoDB Manual for more about planning for and managing document growth for MMAPv1.

MongoDB Data Modeling and Queries

Data Use and Performance

- When designing a data model, consider how applications will use your database.
- For instance, if your application only uses recently inserted documents, consider using Capped Collections.
- If your application needs are mainly read operations to a collection, adding indexes to support common queries can improve performance.

MongoDB Data Modeling and Queries

- MongoDB does not support traditional SQL table queries. Instead, it has a query language that uses documents to describe the desired results, and corresponding documents are returned from the database.
- Queries can include traditional boolean and value equality operators, as well as composition operators including conjunction, disjunction and negation.
- How queries are built depends on the language being used, and the style has differences depending on whether working in the Mongo Shell or a programming language like Java.
- We will examine how to perform queries in both the Mongo Shell and in Java.

MongoDB Data Modeling and Queries

Using the Mongo Shell

- The MongoDB shell is loosely based on JavaScript but provides simplified syntax in some cases.
  - To start the mongo shell and connect to a local MongoDB instance with default port, run the ***mongo*** command.
  - To exit, use the ***exit*** helper command.
  - To view a summary of commands, type *help*.
  - To display the available databases, type *show dbs*.
  - To switch to a database, type *use* ***db***.
  - To display the database you are using, type ***db***.
  - To switch databases, use the *use <database>* helper command.
  - To list containers in the current database, type *show collections*.

MongoDB Data Modeling and Queries

Using the Mongo Shell

- If you end a line with an open parenthesis ('('), an open brace ('{'), or an open bracket ('['), then the subsequent lines start with ellipsis ("...") until you enter the corresponding closing parenthesis (')'), the closing brace ('}') or the closing bracket (']').
- Use the up/down arrow keys to scroll through command history. Use <Tab> to autocomplete or to list the completion possibilities

MongoDB Data Modeling and Queries

Query Data from Mongo Shell

- To query data from a MongoDB collection, use the find() method.
- Syntax:

  myDb.*collection_name*.find(*<query filter>*, *<projection>*)

  You can specify the following optional fields:

  - a query filter to specify which documents to return.
  - a query projection to specifies which fields from the matching documents to return. The projection limits the amount of data that MongoDB returns to the client over the network.
  - You can optionally add a cursor modifier to impose limit(), skip(), and sort() orders. The order of documents returned by a query is not defined unless you specify a sort().

MongoDB Data Modeling and Queries

Query Data from Mongo Shell

- To display results in a formatted way, use the pretty() method

  myDb.*collection_name*.find().pretty()

- To return just the first document, use the findOne() method.

  myDb.*collection_name*.findOne().pretty()

MongoDB Data Modeling and Queries

Query Data from Mongo Shell

- The following example retrieves from the ”mycont” container all documents where the ”by” field is “tutorials point”.

  db.sampleCollection.find({ by: “tutorials point” });

- A query filter document can use the *query operator* to specify conditions in the following form:

  { <field>: { <operator>: <value> }, ... }

- Here is an example that retrieves all documents from “sampleCollection” where the description is “database”, “sql”, or “nosql”:

  db.sampleCollection.find(

  { description: { $in: [”database", ”sql”, “nosql”] } })

MongoDB Data Modeling and Queries

Query Data from Mongo Shell

- To query a document based on some condition, use the following:

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:<value>}	db.mycol.find({ by: "tutorials point"})	where by = 'tutorials point'
Less Than	{<key>:{$lt:<value>}}	db.mycol.find({ likes: {$lt:50}})	where likes < 50
Less Than Equals	{<key>:{$lte:<value>}}	db.mycol.find({ likes: {$lte:50}})	where likes <= 50
Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({ likes:{$gt:50}})	where likes > 50
Greater Than Equals	{<key>:{$gte:<value>}}	db.mycol.find({ likes: {$gte:50}})	where likes >= 50
Not Equals	{<key>:{$ne:<value>}}	db.mycol.find({ likes: {$ne:50}})	where likes != 50
Matches	{<key>: {$regex:<pat>}	db.mycol.find({ by: {$regex: /^tut/}})	where name like tut%

MongoDB Data Modeling and Queries

Query Data from Mongo Shell

- To query documents based on the AND condition, use $and:

```
> db.mycol.find(
   {
      $and: [
         { key1: value1 },
         { key2: value2 }
      ]
   }
)
```

MongoDB Data Modeling and Queries

Query Data from Mongo Shell

- Example:

  Show all tutorials by 'tutorials point' and whose title is 'MongoDB'.

  ```
  db.sampleCollection.find(
      {$and: [ { by: ”MongoDB"}, { title: "MongoDB Architecture"} ] } )
  ```

  This is equivalent to the SQL where clause:

  where by = ‘MongoDB’ and ‘title = ‘MongoDB Architecture”

MongoDB Data Modeling and Queries

Query Data from Mongo Shell

- To query documents based on the OR condition, use $or:

```
> db.mycol.find(
   {
      $or: [
         { key1: value1 },
         { key2: value2 }
      ]
   }
)
```

MongoDB Data Modeling and Queries

Query Data from Mongo Shell

- Example:

  Show all tutorials by 'tutorials point' or whose title is 'MongoDB'.

  db.sampleCollection.find(

  {$or: [{ by: ”MongoDB"}, { title: "MongoDB Architecture"}] })

  This is equivalent to the SQL where clause:

  where by = ‘MongoDB’ or ‘title = ‘MongoDB Architecture”

.

MongoDB Data Modeling and Queries

Query Data from Mongo Shell

- To query documents based on the AND and OR together:

```
> db.mycol.find(
   { $and: [
      { key1: value1 },
      { $or: [
         { key2: value2 },
         { key3: value3 }
       ] }
      }
   ] }
)
```

MongoDB Data Modeling and Queries

Query Data from Mongo Shell

- Example:

  Show the documents that have likes greater than 10 and either whose title is 'MongoDB Overview' or by is 'tutorials point'.

  db.sampleCollection.find({$and: [{ likes : {$gt:10}}
  { $or: [{ by : ”MongoDB"}, { title: "MongoDB Architecture"}] }] })

  This is equivalent to the SQL where clause:

  where (likes>10)

  AND (by = MongoDB' OR title = 'MongoDB Architecture')

.

MongoDB Data Modeling and Queries

Query Data from Mongo Shell

- To specify an exact equality match on the whole embedded document, use the query document

  { *<field>*: *<value>* }

  where <*value*> is the document to match.

  Equality matches on an embedded document require an *exact* match of the specified <*value*>, including the field order.

MongoDB Data Modeling and Queries

Query Data from Mongo Shell

- In the following example, the query matches all documents where the *favorites* field is an embedded document that contains only the fields *artist* equal to "Picasso" and *food* equal to "pizza", in order:

  db.mycont.find({ favorites: { artist: "Picasso", food: "pizza" } })

- Use the dot notation to match by specific fields in an embedded document. This query uses the dot notation to match all documents where the fields in an embedded document match

  db.mycont.find({ favorites.artist: "Picasso" })

MongoDB Data Modeling and Queries

Query Data from Mongo Shell

- When the field holds an array, you can query for an exact array match or for specific values in the array.
- If the array holds embedded documents, you can query for specific fields in the embedded documents using dot notation.

MongoDB Data Modeling and Queries

Query Data from Mongo Shell

- The following example queries for all documents where the field badges is an array that holds exactly two elements, "blue", and "black", in this order:

  db.mycont.find({ badges: ["blue", "black"] })

- The following example queries for all documents where the field badges is an array that has an element that starts with “bl”:

  db.mycont.find({ badges: { $elemMatch: {$regex: /^bl/ } } })

- See the “Query Document” section of the MongoDB v. 4.0.3 manual for more on querying arrays and more advanced queries.

MongoDB Data Modeling and Queries

Query from Java to Retrieve All Documents

```
import com.mongodb.MongoClient;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;

import org.bson.Document;
import org.bson.json.JsonWriterSettings;

public class RetrieveAllDocuments {

   public static void main( String args[] ) {
      // Creating a Mongo client
      MongoClient mongoClient = new MongoClient( "localhost" , 27017 );
      System.out.println("Connected to the server successfully");

      // Accessing the database
      MongoDatabase database = mongoClient.getDatabase("myDb");
      System.out.printf("Database %s opened\n", database.getName());
```

MongoDB Data Modeling and Queries

```
        // Retrieving a collection
        MongoCollection<Document> collection = database.getCollection("sampleCollection");
        System.out.println("Collection sampleCollection selected");

         // sort results by descending number of likes
        FindIterable<Document> iterDoc =
                            collection.find().sort(new Document("likes", -1));

        // JasonWriter renders Bson document as formatted Json document
        JsonWriterSettings.Builder settingsBuilder = JsonWriterSettings.builder().indent(true);
        JsonWriterSettings settings = settingsBuilder.build();

        // iterating documents
        System.out.println("Documents:");
        for (Document doc : iterDoc) {
             System.out.println(doc.toJson(settings));
        }

        mongoClient.close();
    }
}
```

MongoDB Data Modeling and Queries

Query from Java Retrieves Selected Documents Using Filters

```
import com.mongodb.MongoClient;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.FindIterable;

import org.bson.Document;
import org.bson.conversions.Bson;
import org.bson.json.JsonWriterSettings;

import static com.mongodb.client.model.Filters.and;
import static com.mongodb.client.model.Filters.gte;
import static com.mongodb.client.model.Filters.eq;

public class RetrievingAllDocumentsFiltered {

  public static void main( String args[] ) {

    // Creating a Mongo client
    MongoClient mongoClient = new MongoClient("localhost", 27017);
    System.out.println("Connected to the server");
```

MongoDB Data Modeling and Queries

```
// Accessing the database
 MongoDatabase database = mongoClient.getDatabase("myDb");
 System.out.printf("Database %s opened\n", database.getName());

 // Retrieving a collection
 MongoCollection<Document> collection = database.getCollection("sampleCollection");
 System.out.println("Collection sampleCollection selected");

 // Use filters to select: by = "MongoDB" and likes >= 100
 Bson query = and(eq("by", "MongoDB"), gte("likes", 100));

 // sort results by descending number of likes
 FindIterable<Document> iterDoc =
    collection.find(query).sort(new Document("likes", -1));
```

MongoDB Data Modeling and Queries

```
        // JasonWriter renders Bson document as formatted Json document
        JsonWriterSettings.Builder settingsBuilder = JsonWriterSettings.builder().indent(true);
        JsonWriterSettings settings = settingsBuilder.build();

        System.out.println("Documents:");
        for (Document doc : iterDoc) {
            System.out.println(doc.toJson(settings));
        }

        mongoClient.close();
    }
}
```

MongoDB Data Modeling and Queries

Query from Java Retrieves Selected Documents Using Basic Objects

```
import com.mongodb.MongoClient;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.BasicDBObject;

import org.bson.Document;
import org.bson.json.JsonWriterSettings;

public class RetrieveAllDocumentsFilteredBasic {

  public static void main( String args[] ) {

        // Creating a Mongo client
        MongoClient mongoClient = new MongoClient("localhost", 27017);
        System.out.println("Connected to the server successfully");

        // Accessing the database
        MongoDatabase database = mongoClient.getDatabase("myDb");
        System.out.printf("Database %s opened\n", database.getName());
```

MongoDB Data Modeling and Queries

```
// Retrieving a collection
 MongoCollection<Document> collection = database.getCollection("sampleCollection");
 System.out.println("Collection sampleCollection selected successfully");

 // Use basic objects to select: by = "MongoDB" and likes >= 100
 BasicDBObject query = new BasicDBObject(
       "$and", new BasicDBObject[] {
                                 new BasicDBObject("by", "MongoDB"),
                                 new BasicDBObject("likes", new BasicDBObject("$gte", 100))
 });

 // sort results by descending number of likes
 FindIterable<Document> iterDoc =
                     collection.find(query).sort(new Document("likes", -1));
```

MongoDB Data Modeling and Queries

```
        // JasonWriter renders Bson document as formatted Json document
        JsonWriterSettings.Builder settingsBuilder = JsonWriterSettings.builder().indent(true);
        JsonWriterSettings settings = settingsBuilder.build();

        for (Document doc : iterDoc) {
            System.out.println(doc.toJson(settings));
        }

        mongoClient.close();
    }
}
```

MongoDB Data Modeling and Queries

Query Data from Java With Limit

```
import com.mongodb.MongoClient;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;

import org.bson.Document;
import org.bson.json.JsonWriterSettings;

public class RetrievingAllDocumentsLimit {

  public static void main( String args[] ) {

    // Creating a Mongo client
    MongoClient mongoClient = new MongoClient("localhost", 27017);
    System.out.println("Connected to the server successfully");

    // Accessing the database
    MongoDatabase database = mongoClient.getDatabase("myDb");
    System.out.printf("Database %s opened\n", database.getName());
```

MongoDB Data Modeling and Queries

```
        // Retrieving a collection
        MongoCollection<Document> collection = database.getCollection("sampleCollection");
        System.out.println("Collection sampleCollection selected successfully");

        // limit to 1, sort results by descending number of likes,
        FindIterable<Document> iterDoc =
            collection.find().limit(1).sort(new Document("likes", -1));

        // JasonWriter renders Bson document as formatted Json document
        JsonWriterSettings.Builder settingsBuilder = JsonWriterSettings.builder().indent(true);
        JsonWriterSettings settings = settingsBuilder.build();

        System.out.println("Documents:");
        for (Document doc : iterDoc) {
            System.out.println(doc.toJson(settings));
        }

        mongoClient.close();
    }
}
```

MongoDB Data Modeling and Queries

Projections from Mongo Shell

- When using the container.find() method, it is possible to control which fields are included in the document results.
- By default, all fields are projected. We can use an addition list for find() to specify which fields to include and which to exclude.

MongoDB Data Modeling and Queries

Projections from Mongo Shell

- Examples:

  - Select only documents with more than 100 likes, and project only the ”likes” and “by” fields, also excluding the “$_id” field from the returned documents:

    ```
    db.mycol.find( { likes: {$lte: 100} }, { likes: 1, by: 1, _id : 0 } )
    ```

  - Specify to include all fields except “likes”, “by”, and “_id”:

    ```
    db.mycol.find( { likes: {$lte: 100} }, { likes: 0, by: 0, _id : 0 } )
    ```

  - Specify to include ”_id”, “name”, “status”, and the “food” field inside the “favorites” field embedded document:

    ```
    db.mycol.find( { likes: {$lte: 100} }, { name: 1, status: 1, favorites.food: 1  } )
    ```

  Note that ”_id” is always included in a projection unless explicitly excluded.

MongoDB Data Modeling and Queries

Projections from Java

- The Projections class provides static factory methods for all the MongoDB projection operators.
- By default, all fields are projected. We can use the include() and exclude() methods to determine which fields should be projected into our output.

MongoDB Data Modeling and Queries

Projections from Java

- Example: exclude "_id" field from the output

```
import com.mongodb.MongoClient;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;

import static com.mongodb.client.model.Projections.excludeId;
import static com.mongodb.client.model.Filters.and;
import static com.mongodb.client.model.Projections.include;

import org.bson.Document;
import org.bson.json.JsonWriterSettings;

public class RetrieveAllDocumentsProject {

    public static void main( String args[] ) {
```

MongoDB Data Modeling and Queries

Projections from Java

```
// Creating a Mongo client
MongoClient mongoClient = new MongoClient( "localhost" , 27017 );
System.out.println("Connected to the server");

// Accessing the database
MongoDatabase database = mongoClient.getDatabase("myDb");
System.out.printf("Database %s opened\n", database.getName());

// Retrieving a collection
MongoCollection<Document> collection = database.getCollection("sampleCollection");
System.out.println("Collection sampleCollection selected");

// only include fields, explicitly exclude("_id") using excludeId() else
// implicitly included;  sort results by descending number of likes
FindIterable<Document> iterDoc =
        collection.find()
            .projection(and(include("description", "likes", "title"),  excludeId()))
            .sort(new Document("likes", -1));
```

MongoDB Data Modeling and Queries

Projections from Java

```
        // JasonWriter renders Bson document as formatted Json document
        JsonWriterSettings.Builder settingsBuilder = JsonWriterSettings.builder().indent(true);
        JsonWriterSettings settings = settingsBuilder.build();

        // iterating documents
        System.out.println("Documents:");
        for (Document doc : iterDoc) {
          System.out.println(doc.toJson(settings));
        }

        mongoClient.close();
    }
}
```

MongoDB Data Modeling and Queries

- Several other tools are available including:
  - Mongo Compass, is available from the Mongo website at https://www.mongodb.com/products/compass. It is an officially supported GUI for Visually exploring data and run ad hoc queries. It is available on Linux, Mac, or Windows.
  - Studio 3T (https://studio3t.com/) is a professional-level tool that is available as a commercial version, and a non-commercial use version. Studio 3T takes MongoDB Query and gives Tee View/Table View/JSON View of a collection.
  - The same company also has a free basic GUI, Robo 3T that works on MacOS and Windows (https://robomongo.org/) .
- You should evaluate these tools and choose one that works well for you. Also look for introductory videos for these three on YouTube.