

Implement a MongoDB Database for InterMine

Name: *Evaldas Latoškinas*

Overview

Core Problem

A relational database for the InterMine data model is slow and restrictive. Therefore, a modern approach using a NoSQL system is required.

Moreover, Neo4J has been tried and tested in a previous project, however, the conclusion was that the InterMine data model is hard and impractical to fit into a graph.

These issues can be remedied by choosing a different NoSQL database management system.

Proposed Solution

Implement MongoDB as the database for InterMine data.

MongoDB would fit the IM data model very well compared to the relational or the graph database approach, as the MongoDB data model is less restrictive, more adaptable and extendable for InterMine than the previous options.

Brief Overview of the Process

- MongoDB database model design
- Setting up the database in the server
- Building the actual database & loading data
- Parse PathQuery to MongoDB queries
- Optimize Querying
- RESTful API for MongoDB queries



Statement of the Problem

Why MongoDB?

Relational database core issues

- High number of table joins - most of the data stored in the current data model revolves around foreign key referencing for inheritance as well as complex attributes. As such, multiple chained table joins are needed, which drastically impacts performance due to the amount of intermediate tables required for the large hierarchical data structure.
- Data redundancy - Due to the nature of the relational model, inheritance and hierarchical data is rather complex to represent. For instance, a single entity of Protein is stored both in the BioEntity table and in the Protein table. The database therefore grows rapidly in size due to data redundancy and this affects both query performance and the database size.
- Scalability - due to ACID properties, horizontal scalability would be very hard to achieve using a SQL database. Implementing a NoSQL database system would future-proof against this issue due to database sharding support.

Neo4J core issues

- Unnecessary complexity - everything in the graph model is represented as nodes and relationships. Therefore, modelling the IM data in a graph becomes relatively complex, especially when extending the data model with new types of data.
- Graph size - as more data is added, the graph grows rapidly due to the relatively high depth of relationships. This has implications on performance and the database size.

MongoDB Implementation Problems

Main Problem

Replace PostgreSQL with MongoDB as the backend database.

Sub-Problems

- Designing a MongoDB database model that is optimal for the IM Model
 - How to model the data into documents? (attributes, references, collections)
 - How to represent relationships? (1-to-1, 1-to-N, N-to-N)
 - How to split the data into collections?
 - How to make the model optimized for IM data?
 - What is the optimal approach to modelling inheritance (for hierarchical data) in MongoDB for this case?
- Setting up the database in the server
 - How to setup & access the database from the server?
 - How to integrate the database into the server?
- Building the actual database

- How to store Java objects in MongoDB?
 - How to map MongoDB documents to the respective collection?
 - How to load the actual data?
 - How to handle additional inserting, deleting and updating?
- Querying the data
 - How to parse PathQuery queries to MongoDB?
- Optimization
 - What are the most optimal MongoDB query operations to perform in various scenarios? How should queries be optimized in general?
 - Where to apply indices to reach optimal performance?
- RESTful API integration
 - How to build a RESTful API that works independently, on its own, and has access to the database?
 - How to link the server functionality with the REST API?

Implementation Plan

Designing the database model

The general idea

Let us first define three important definitions in a MongoDB database:

Database - a database instance in the actual MongoDB database (equivalent to a PostgreSQL database instance, which contains all the tables)

Collection - a list of records (equivalent to a SQL table/relation)

Document - a single record (equivalent to a tuple record in SQL)

The core concept of MongoDB data storage is that the data is stored in a BSON (JSON-like) format. Each document is essentially a single JSON string containing all the relevant fields. Each field is a key value pair, where both key and value are strings. The value is not necessarily atomic - it may even be an array, or an object, for instance.

The JSON-like structure is a huge advantage for Java, since mapping objects to the database is comparably easy compared to mapping objects to a SQL or a graph database.

Moreover, the MongoDB data model is schemaless and does not have constraints whatsoever. If we want to add a completely new field into a document in a collection, and that field did not exist anywhere in the collection before, we can easily do so. There are no limitations against this, unlike in SQL systems, where tables have a fixed number of columns.

MongoDB documents

A document should generally represent a single Java Object, such as Comment, Gene, Ontology or Protein from the InterMine database. A document in our case of the InterMine server running on Java is basically a Java object parsed to JSON format with a few minor modifications (for instance, we might not store entire objects at some points, but references instead)

Important definition:

_id field - the primary identifier of an object. This is generated by default. However, this may be initialized with own values (e.g. integers, Strings, etc.)

The simple case (attributes only)

Consider the Java class Comment, found in the core.xml definition. It has two fields - description and type. Suppose we create a new Comment instance with description "This

is a description" and type "Type1". Now suppose we want to convert the object and store it as a document.

For this case, this simply involves parsing the Object to JSON format and storing it to MongoDB as a document. The resulting structure of the stored document would look as follows:

```
{
  "_id": "507f191e810c19729de860ea",
  "description": "This is a description",
  "type": "Type1"
}
```

The simple case II (replacing references with nested objects)

Let's now consider the `OntologyEvidence` Java class now. We see that currently, there is a code attribute which is a reference to `OntologyAnnotationEvidenceCode`. In SQL, this is a foreign key constraint. However, since this is a simple object with only three attributes, it is in fact better to store this object as a nested object in our document so we have easy access and do not need to look into other collections.

Using nested objects instead of references is in most cases more optimized for MongoDB and should generally be preferred if the data redundancy is not overly high as a result of this.

So if we ignore the publications field (we will move to that concept later), we would have the following structure for an `OntologyEvidence` document:

```
{
  "code": {
    "code": "evidenceCode",
    "name": "evidenceName",
    "url": "evidenceURL"
  },
  "publications": "..."
}
```

Referenced Objects

While nesting objects is practical in most cases due to the nature of MongoDB, for the IM data model we probably want to use references instead due to the high depth of the hierarchical data. By using references, we highly reduce data redundancy and make certain operations (namely updating and deleting) far more easier.

Consider for instance the Class `OntologyRelation`. We see that there are 3 simple attributes which we can easily express, and there are two references `parentTerm` and

childTerm. While we can use nested objects in this case, we most likely do not want to, because imagine there are millions of `OntologyRelations`, and around 100 thousand (as an example) would share the same `parentTerm`. This results in 100 thousand instances of exactly the same object, therefore, we should prefer references. Moreover, `parentTerm` and `childTerm` form recursive relations. If we would nest objects, we would end up with a huge nested substructure, which would be highly inefficient (and perhaps not even possible to represent)

The most common way to reference objects is simply have a field where the value is the “_id” field of the referenced object. This is referred to as manual referencing, and in this case this is used to model 1-to-N (one-to-many) relationships

So assume that we have an `OntologyRelation` object instance, and the “_id” field of `parentTerm` is “parentID”, and for `childTerm` it is “childID”. Our `OntologyRelation` would then look something like this:

```
{
  "parentTerm": "parentID",
  "childTerm": "childID",
  "relationship": "relationship-attribute",
  "direct": true,
  "redundant": true
}
```

To retrieve the actual referenced objects, we will need to write queries to check against this referenced id.

There are also `DBRefs` in MongoDB which directly reference a collection and (optionally) the database instance, however, [this is not recommended officially by the MongoDB documentation](#). The reason for this is the associated performance issues, as well as MongoDB’s nature (database sharding is allowed, but `DBRefs` would not work if data is sharded)

Reverse References

If we have a reverse-reference tag, this is actually as simple as putting a reference on the referencing object, where the reference ID would be the ID of the object being referenced. We follow the same steps as in the last section.

Alternatively, we may use nested objects for increased performance. However, if we decide to do so, the nested object should not contain any references whatsoever, because otherwise we might end up in a data overflow (we nest the reference in the nested object, that reference has another nested object as reference, that reference has another nested object as reference, ...)

Collections

Another way to model 1-to-N relationships is by the use of nested collection objects.

For instance, let's take the DataSource object, which has a collection "publications". We see that this references the Publication Object, so we will apply the previously mentioned concept of referencing.

If we omit the underlying structure of the dataSets field for now, our DataSource instance would look something like this:

```
{
  "name": "sourceName",
  "url": "someUrl",
  "description": "The description of the data source",
  "publications":
  [
    "id1",
    "id2",
    "id3",
    "id4",
    "id5"
  ],
  "dataSets": "...
}
```

Here we assume that the dataSource publications references five publications, whose ids are "id1", "id2", "id3", "id4" and "id5". We see that publications is an array, and can be stored like this in MongoDB.

Also, there is an ability to have array entries be nested objects, although this might not be very practical for the IM data model for the majority of the classes, especially if we have a lot of entries in one collection. Generally, it is recommended to have an array of nested objects in 1-to-few relationships, and as our array grows, we typically want to have references instead.

Many-to-Many Referencing

Now let's consider the final relationship type that we have not covered: N-to-M (many-to-many) relationships.

To keep data redundancy minimal, the optimal way to do this would be two-way referencing.

For instance, consider the Class Author. We see that there is a collection named publications, which references publications, and is also a reverse reference to authors.

We may model Author objects as follows:

```
{
  "_id": "jdoe",
  "firstName": "John",
  "lastName": "Doe",
  "name": "John Doe",
  "initials": "JD",
  "publications":
  [
    0,
    1,
    2
  ]
}
```

```
{
  "_id": "jdoe2",
  "firstName": "Jane",
  "lastName": "Doe",
  "name": "Jane Doe",
  "initials": "JD",
  "publications":
  [
    0
  ]
}
```

Where publications array entries are ids of publications.

Then, we may model each publication as follows (note that the majority of attributes is omitted here for brevity):

```
{
  "_id": 0,
  "title": "title",
  "authors":
  [
    "jdoe",
    "jdoe2"
  ]
}
```

```
{
  "_id": 1,
  "title": "title2",
  "authors":
  [
    "jdoe"
  ]
}
```

```
{
  "_id": 2,
  "title": "title3",
  "authors":
  [
    "jdoe"
  ]
}
```

Splitting data into collections

Splitting data into collections is actually rather intuitive. For the majority of the classes, one class should correspond to one collection. However, it is a bit more complicated when it comes to inheritance.

Modelling Inheritance

Certain classes do not have their own instances and only act as an abstraction (namely BioEntity). As such, we will not ever need to store BioEntity instances in the database, but rather store the inherited class objects instead. This works surprisingly well in MongoDB due to the flexibility of the data model.

So for an inherited object, the object will then have both the inherited fields and it's own fields in the document itself (much like in Java). For instance, if we take a Protein object stored in a document, it would look like this:


```

{
  "primaryIdentifier": "...",
  "ontologyAnnotations": [],
  "publications": [],
  "secondaryIdentifier": "...",
  "symbol": "...",
  "name": "...",
  "organism": "...",
  "locatedFeatures": [],
  "locations": [],
  "synonyms": [],
  "dataSets": [],
  "crossReferences": [],
  "md5checksum": "...",
  "primaryAccession": "...",
  "length": 0,
  "molecularWeight": 0.0,
  "sequence": "...",
  "genes": []
}

```

Since we have the inheritance chain Protein -> BioEntity -> Annotable, note how the document has all the fields from all three of these classes.

There are generally two reasonable approaches for modelling inheritance.

Approach 1: Single Collection for all inherited classes

For this approach, the main idea is that we have a single collection and that collection stores -ALL- the objects that inherit from the very base class. For instance, if we have the base class Annotable, all the instances of Annotable would be stored in one collection (even those that go deep in the inheritance chain, such as Protein)

To distinguish between the classes, we will have an additional “type” field which indicates the type. For example, we would distinguish “Protein” objects by setting the “type” to “Protein”.

For instance, we want to store all Strains and Proteins in the database, since these are actual instances that are to be stored. Both Strain and Protein extend BioEntity, and BioEntity extends Annotable. Therefore, we have an Annotable collection, which would store both Strains and Proteins in this case (here I’m simplifying the case a bit). The resulting structure looks as follows:

```

{
  "type": "Protein",
  "primaryIdentifier": "...",
  "ontologyAnnotations": [],
  "publications": [],
  "secondaryIdentifier": "...",
  "symbol": "...",
  "name": "...",
  "organism": "...",
  "locatedFeatures": [],
  "locations": [],
  "synonyms": [],
  "dataSets": [],
  "crossReferences": [],
  "md5checksum": "...",
  "primaryAccession": "...",
  "length": 0,
  "molecularWeight": 0.0,
  "sequence": "...",
  "genes": []
},
{
  "type": "Strain",
  "primaryIdentifier": "...",
  "ontologyAnnotations": [],
  "publications": [],
  "secondaryIdentifier": "...",
  "symbol": "...",
  "name": "...",
  "organism": "...",
  "locatedFeatures": [],
  "locations": [],
  "synonyms": [],
  "dataSets": [],
  "crossReferences": [],
  "annotationVersion": "...",
  "assemblyVersion": "...",
  "features": []
}

```

The core issue with this approach is that the collections would be relatively large, and this might impact performance. However, if database sharding is ever considered, it would really assist in making the performance better and this approach might be better.

Approach 2: Collection per class

Another possible approach is to strictly have one collection per class. For example, Protein would have its own collection.

This approach is advantageous over the last due to the reduced size of collections.

However, there are scenarios where this approach would complicate things - for instance, consider the Publication class, which references Annotables. However, we do not know the exact type of the Annotable, therefore we do not know which Collection to look in. Moreover, we will most likely not even have an Annotable collection, because Annotable is simply an abstraction (although, we could create a collection that contains references, but that would not be optimal for the MongoDB structure). Also we have to keep in mind that MongoDB is not exactly optimized for aggregation operations, therefore this approach might drastically impact performance.

A document would then look exactly like the one introduced at the beginning of the "Modelling Inheritance" section. (No extra type field whatsoever)

I believe the second approach might work better here, although both might have to be tried in practice to see which one works best.

Either way, we have one main issue with inheritance. Some classes reference, for instance, the base class Annotable, others reference Protein, others reference Strain and so on. There will need to be some additional Java logic in place to actually know which collection to look for the data. The first approach allows ease of finding any object that is referenced as "Annotable", while the second approach makes it easy to find any class that is not referenced by the abstract base class (Protein, Strain, etc.)

MongoDB database instances

Much like the current setup with PostgreSQL, in my eyes, one mine should correspond to one MongoDB database instance inside the actual MongoDB database. This way, we have all the needed data centralized in one place.

For sharding, a different approach should most likely be taken (by simply splitting up collections into separate database instances), but that is outside the scope of this project.

Normalized vs Denormalized

An important thing to note is that MongoDB is mostly optimized for denormalized, embedded data, meaning that nested objects rather than separate collections usually give more performance, since MongoDB is optimized for document querying rather than joins.

In general, for the IM data model, we will most likely want it mostly normalized, however, we might keep some parts of the data model denormalized for performance

Right now, I believe that the IM data model should be mainly normalized for the following reasons:

- Updating entries will be significantly easier - rather than searching many records for nested data, simply update the data for the document and the references now point to the updated document. This is significantly faster than having embedded data. This is mainly a huge benefit for GFF3/FASTA importing, which imports data to entries that already exist.
- Avoids huge amount of data redundancy - The same object might be referenced thousands of times. If we instead reuse the object over and over as an embedded

document, we have huge data redundancy.

- A document has a data limit of 16MB. If we happen to exceed that with embedded documents, we would run into a lot of issues trying to remedy the issue.
- Moreover, with these gained benefits, we may simply reduce the amount of aggregation operations to optimize our querying.
- [Official MongoDB documentation](#) recommends normalizing for large hierarchical data sets, complex many-to-many relationships and severe reduction of data redundancy. This all applies to the IM data model.

However, at the cost of (possibly highly) increased data redundancy, denormalizing might highly increase querying performance. Therefore, I will be testing this in practice with the IM data model with large data sets.

By following all these guidelines, I will be able to design a MongoDB database model that is optimized for IM data and is easy to extend with additions. One of the most important crucial decisions that I will make is one for denormalization vs normalization.

Database integration with the server

Setting up & accessing the database

Since the InterMine Server currently runs on PostgreSQL as it's backing database, code needs to be written to actually integrate MongoDB with a server by establishing a connection with the local or remote server database.

First of all, the prerequisite is that the server has MongoDB installed and the service is running. While developing this, I will be running Mongo on my local machine, and all connections will go through localhost.

The Java server application needs to interact with the MongoDB database. This can easily be accomplished by making use of the official [MongoDB Java Driver](#), which enables easy access and connection to the database, as well as providing methods for building and querying the data.

However, instead of using the core MongoDB Java Driver, I will be installing [Morphia](#), which has all the functionality of MongoDB Java Driver plus additional functionality that will be really useful for object mapping (e.g. additional annotations) and querying. This dependency can be included in the Gradle build.

A brief overview of my experimenting & understanding the library that I suggest using (Morphia) as well as MongoDB and Java interaction in general can be found in [one of my repositories](#).

We may then connect to MongoDB from the Server as follows:

```
MongoClient mongoClient = new MongoClient("localhost");
```

This initializes the connection with the MongoDB database with the host name as “localhost”. We may instead pass in another host name as well as a port number (both of which can be converted to command line arguments to allow flexibility, or environment variables).

```
Morphia morphia = new Morphia();
```

This creates a Morphia object which will assist in handling various otherwise tedious functionality (for instance, Java object mapping to collections)

```
Datastore datastore = morphia.createDatastore(mongoClient,
"testdb");
```

This initializes a database instance called “testdb” in our MongoDB database, which will contain all our collections.

Now, we may simply use methods from the DataStore to perform basic CRUD operations. The changes will then be reflected to the database.

Moreover, after running these three lines of code, we may open the command line, type `mongo testdb` and this will open a connection with the database testdb locally, where we may query data and run various functions (more on this in later sections)

Additionally, we may access a single collection in Morphia as follows:

```
DBCollection collection =
datastore.getCollection(OntologyTerm.class);
```

This will return the collection which stores OntologyTerm objects. We may then perform operations on this collection.

Integrating the database into the Server

In order to have a modular, working MongoDB environment that is easily extendable, adaptable and modifiable, I believe it is crucial that database integration into the server would be one of the major priorities when working on this project early on. I plan to build something like the SQL environment found in `intermine/objectstore/src/main/java/org/intermine/sql`, although I would build a more barebones version initially which will be extended as the project goes on.

The classes written for this part of the project should include:

- A general utility class that links all the other classes (somewhat like DatabaseUtil, although a bit different in the way that it would be more modular/extendable). This would be the core center of the database operations, which also holds the Morphia DataStore.
- A class that allows update, delete & insert operations (this will be really useful for

testing and for providing overall functionality)

- A general class that takes in search queries and returns the results. This would also include some standard search queries (which will again assist in the testing process)
- A specialized class that builds MongoDB queries from PathQuery objects - to be handled properly in the later weeks.

Later on when writing the REST API logic, I imagine that the Utility class would serve as the Service for the database, which could then be linked to a REST Controller to interact with the database.

Building the Database

Storing Java Objects in MongoDB

With the Morphia library, storing data in MongoDB becomes easier, since Morphia handles all the conversions to BSON for us. Moreover, we may specify annotations which are crucial for our data model.

POJOs

In order to successfully store our Java objects in the database, there is one requirement - the objects must be POJOs (Plain Old Java Objects). For a Java object to be a POJO, it must satisfy three conditions:

- The object should not extend pre-specified classes (in other words, classes that are part of the Java library)
- The object should not implemented pre-specified interfaces (interfaces that are part of the Java library)
- The object should not contain prespecified annotations (however, we may use library (e.g. Morphia's) annotations)

This, however, is not an issue for this project, because all built Java classes are POJOs.

Annotations

```
@Id
private String identifier;
```

An example of an annotation (@Id)

Id - giving this annotation to a field in a Java class specifies that the field is an identifier (corresponding to `_id`) in the collection.

Reference - specifies that the field should be a reference rather than an embedded document. To avoid the use of DBRefs (which are bad practice), we will use the option `onlyId` to have references only store the `_id` of the referenced object.

Entity("name") - specifies the name of the collection to store the object in. We may exclude the parentheses with the name, in which case the collection name will become the case-sensitive name of the class. However, if we want to store multiple classes in the same collection, we need the value there.

Index - specifies indexes for the entire class. This is added at the top of the class (below or above the Entity annotation). More on this in the section of Query Optimization.

Storing Objects - Simple Case

Consider the following simplified class of the object `OntologyTerm`:

```
@Entity("OntologyTerm")
public class OntologyTerm {
    @Id
    private String identifier;

    private String name;
    private String description;
    private String namespace;
    private boolean obsolete;
```

We will be storing all `OntologyTerms` in the collection “`OntologyTerm`” (see Entity annotation). Our id will be the String identifier.

Storing objects such as this is actually really simple with Morphia. Provided we have these annotations and we have our `dataStore`, we may store an `OntologyTerm` instance as follows:

```
OntologyTerm term1 = new OntologyTerm("term1",
    "Term 1", "Term 1 Description", "namespace", false);

datastore.save(term1);
```

Let’s check if this worked. If we open the command line and enter “`mongo testdb`”, and type “`show collections;`”, we will see the following result:

```
> show collections;
OntologyTerm
```

Let us now see if our data has been added by issuing a `find()` command on `OntologyTerm`:

```
> db.OntologyTerm.find().pretty();
{
  "_id" : "term1",
  "className" : "mapping.OntologyTerm",
  "name" : "Term 1",
  "description" : "Term 1 Description",
  "namespace" : "namespace",
  "obsolete" : false
}
```

We see that the data has been indeed added and the conversion to BSON has been done for us.

Storing Objects - References

Consider now the case where we want to add OntologyTerm instances to the database. As a Java Object, we would have the following structure:

```
@Entity("OntologyRelation")
public class OntologyRelation {
    @Reference(idOnly = true)
    private OntologyTerm parentTerm;

    @Reference(idOnly = true)
    private OntologyTerm childTerm;

    private String relationship;
    private boolean direct;
    private boolean redundant;
```

Note the Reference annotations with idOnly parameters, specifying that childTerm and parentTerm should not be stored as the actual object in the database, but rather only as an “_id” manual reference (if we did not have this annotation, these objects would be stored as nested objects. If we did not have the idOnly parameter, the Reference would be a DBRef)

Suppose we have term1 in the database from our previous step, and we add another OntologyTerm as follows:

```
OntologyTerm term2 = new OntologyTerm("term2", "Term 2",
    "Term 2 Description", "namespace", false);
```

```
datastore.save(term2);
```


Afterwards, we add `OntologyRelation`:

```
OntologyRelation relation = new OntologyRelation(term1,
term2, "relationship", true, false);
```

```
datastore.save(relation);
```

The result:

```
> show collections;
OntologyRelation
OntologyTerm

> db.OntologyRelation.find().pretty();
{
  "_id" : ObjectId("5c8850caa9556a30c4c33ec4"),
  "className" : "mapping.OntologyRelation",
  "parentTerm" : "term1",
  "childTerm" : "term2",
  "relationship" : "relationship",
  "direct" : true,
  "redundant" : false
}
```

Note here that `parentTerm` and `childTerm` are indeed “_id” manual references rather than the actual objects.

Mass Storing Objects

Consider we want to store an entire list of objects in one go. We may easily do so and Morphia handles this for us.

Assume the simplified `Publication` class:

```
@Entity("Publication")
public class Publication {
    @Id
    private int publicationId;
    private String title;
    private String issue;
    private String pages;
    private int year;
```

And then consider the following code:

```
ArrayList<Publication> publications = new
ArrayList<Publication>();
```

```

Publication publication1 = new Publication(52, "Publication",
"3", "100", 2019);
Publication publication2 = new Publication(25, "Publication
2", "5", "125", 2019);
Publication publication3 = new Publication(50, "Publication
3", "7", "50", 2019);

publications.add(publication1);
publications.add(publication2);
publications.add(publication3);

datastore.save(publications);

```

Result:

```

> db.Publication.find().pretty();
{
  "_id" : 52,
  "className" : "mapping.Publication",
  "title" : "Publication",
  "issue" : "3",
  "pages" : "100",
  "year" : 2019
}
{
  "_id" : 25,
  "className" : "mapping.Publication",
  "title" : "Publication 2",
  "issue" : "5",
  "pages" : "125",
  "year" : 2019
}
{
  "_id" : 50,
  "className" : "mapping.Publication",
  "title" : "Publication 3",
  "issue" : "7",
  "pages" : "50",
  "year" : 2019
}

```

Storing Objects - Collections & Reference List

Suppose the following Author class (note the ArrayList of Publications and reference annotation):

```

@Entity("Author")
public class Author {
    private String firstName;
    private String lastName;
    private String name;
    private String initials;

    @Reference(idOnly = true)
    private ArrayList<Publication> publications;

```

Luckily, Morphia also handles storing reference lists for us. Moreover, Morphia handles storing collections for us as well. So if we take the publications list from last step and run the following code:

```

Author author = new Author("John", "Doe", "John Doe", "jdie",
publications);

```

```

datastore.save(author);

```

We get:

```

> db.Author.find().pretty();
{
  "_id" : ObjectId("5c885545a9556a0ef80575e6"),
  "className" : "mapping.Author",
  "firstName" : "John",
  "lastName" : "Doe",
  "name" : "John Doe",
  "initials" : "jdie",
  "publications" : [
    52,
    25,
    50
  ]
}

```

Storing Objects - Embedded Documents

Supposing the following classes:

```

@Entity("OntologyEvidence")
public class OntologyEvidence {
    OntologyAnnotationEvidenceCode code;

```

```
public class OntologyAnnotationEvidenceCode {
    private String code;
    private String name;
    private String url;
}
```

By running the following code:

```
OntologyAnnotationEvidenceCode code = new
OntologyAnnotationEvidenceCode("code", "name", "url");
```

```
OntologyEvidence ontologyEvidence = new
OntologyEvidence(code);
```

```
datastore.save(ontologyEvidence);
```

We get the result:

```
> db.OntologyEvidence.find().pretty();
{
  "_id" : ObjectId("5c8856fea9556a3328ffcd86"),
  "className" : "mapping.OntologyEvidence",
  "code" : {
    "code" : "code",
    "name" : "name",
    "url" : "url"
  }
}
```

So we can see that by default, Morphia stores embedded objects instead of references.

Constructing Java Classes With Annotations

The way that Java classes are currently built is mostly fine to store in MongoDB. This is a huge advantage over SQL, since we do not require any object relational mapping, as our Java objects are nearly ready for use with MongoDB.

However, we need to take care of adding annotations for optimal storage of the IM data.

It is important to note that the addition of these annotations does not actually change the Java object structure in any way, as this is merely just metadata for the MongoDB database.

ID Annotations (Optional, but possible)

Some classes have their own primary identifiers rather than using a generated ID. If we want to reduce redundancy a bit, we may consider using those values as primary identifiers, rather than generating a numeric ID (or using generated ObjectIDs).

To accomplish this task, we will need to store some additional data (possibly in the XML, or a separate metadata file) to indicate that a value is an identifier, and then according to that, while building the Java class, we would then add the `@Id` annotation. If no such field is found, the default ObjectID will be used.

Reference Annotations

The most important part of Java Class construction to handle is reference annotations. Assuming the near full normalization approach is followed, I would take the following approach:

1. For each xml "reference" field, add a `@Reference(onlyId = true)` annotation while building the Java classes
2. For each xml "collection" field, add a `@Reference(onlyId = true)` annotation while building the Java classes

In fact, these are the only changes required to prepare our Java classes for storage in the MongoDB powered database for the normalized version.

Additionally, if it is decided that some objects should be embedded, there should be some sort of indication (in some metadata file possibly) that it should be embedded, and then no annotation would be added on `@Reference` attributes. This will all depend on the decisions while modelling the data model. An alternate approach that I have in mind, for instance, is to make all collections references, but make single references embedded documents, which might be highly beneficial for performance, but also comes with its own downsides (large nested chains of objects).

Building the actual database

The key difference between PostgreSQL and MongoDB here is that after building the Java classes with annotations as specified in the previous section, we are ready to load our data into the database, since Morphia will handle the mapping for us.

The steps to building the database would be as follows:

1. Follow the ordinary steps of building the Java classes (also take care of constructing all the required annotations)
2. Create a collection for each class (unless another structure has been decided on, such as putting all inherited classes into a single collection) -> This is also optional, since Morphia automatically creates the collections when inserting data.
3. When importing data, construct instances of the matching Java classes and put them into the corresponding collections of the database
 - a. Morphia allows us to have automated class collection identifying. This way, we do not even need extra steps to decide on which collection to add the instance to.

After all of this, we are done and have a database with all the data loaded.

Data Updating & Deleting

We have fully covered 1 out of 4 major operations of CRUD, which is Create. However, we will also need functionality to Update & Delete records from a collection. We specifically

want the Update functionality to be able to efficiently import FASTA or GFF3 data.

I will assume that the following record is added before running delete & update operations:

```
OntologyTerm term3 = new OntologyTerm("term3", "Term 3",
"Term 3 Description", "namespace", false);
datastore.save(term3);
```

Introduction to Query & Deleting Data [OPTIONAL]

Suppose we want to delete term3 from the database.

Morphia has Query objects, which allow to specify conditions for the object to look for. Consider:

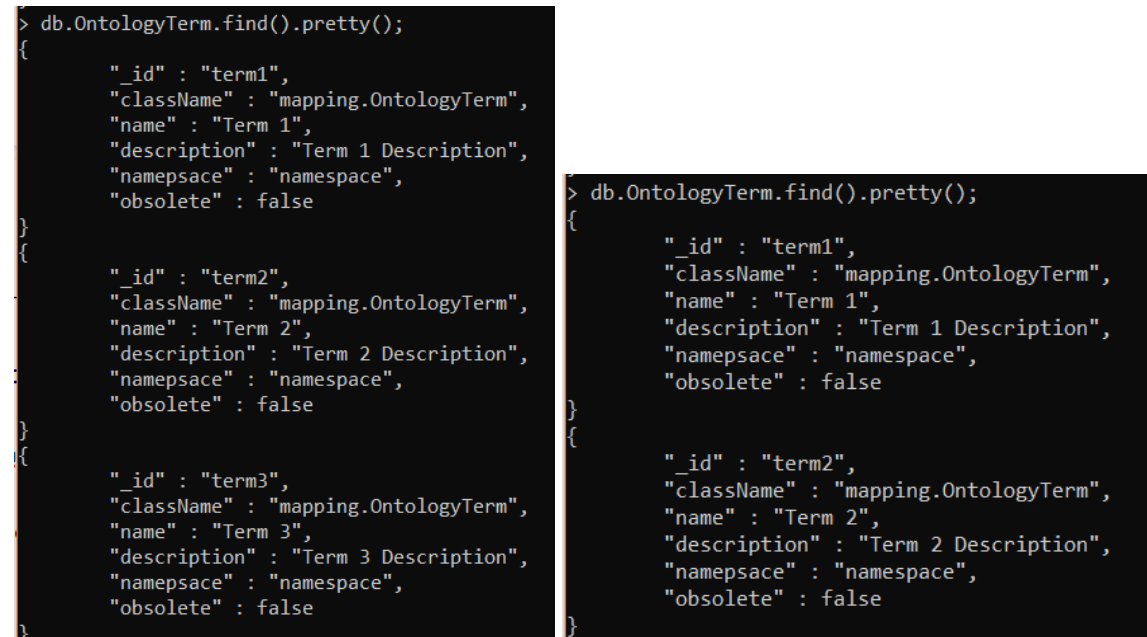
```
Query query =
datastore.createQuery(OntologyTerm.class).field("_id").equal(
"term3");
```

The following query indicates to find an OntologyTerm object (in OntologyTerm collection) which has the "_id" of "term3".

We may then plug the query into the findAndDelete method:

```
datastore.findAndDelete(query);
```

And here is the result:



```
> db.OntologyTerm.find().pretty();
{
  "_id" : "term1",
  "className" : "mapping.OntologyTerm",
  "name" : "Term 1",
  "description" : "Term 1 Description",
  "namespace" : "namespace",
  "obsolete" : false
}
{
  "_id" : "term2",
  "className" : "mapping.OntologyTerm",
  "name" : "Term 2",
  "description" : "Term 2 Description",
  "namespace" : "namespace",
  "obsolete" : false
}
{
  "_id" : "term3",
  "className" : "mapping.OntologyTerm",
  "name" : "Term 3",
  "description" : "Term 3 Description",
  "namespace" : "namespace",
  "obsolete" : false
}
```

```
> db.OntologyTerm.find().pretty();
{
  "_id" : "term1",
  "className" : "mapping.OntologyTerm",
  "name" : "Term 1",
  "description" : "Term 1 Description",
  "namespace" : "namespace",
  "obsolete" : false
}
{
  "_id" : "term2",
  "className" : "mapping.OntologyTerm",
  "name" : "Term 2",
  "description" : "Term 2 Description",
  "namespace" : "namespace",
  "obsolete" : false
}
```

Before & After

Updating Data

Update operations can be done as follows:

```
UpdateOperations<OntologyTerm> update =
datastore.createUpdateOperations(OntologyTerm.class).set("des
cription", "Term 3 Description Updated!");
```

Specifies the update operation to perform. This updates the description to be "Term 3 Description Updated!". The following query also works if the field currently does not exist in the document. In that case, the field will be added with the value at the end of the document.

```
datastore.update(query, update);
```

The following executes the query.

```
> db.OntologyTerm.find().pretty();
{
  "_id" : "term1",
  "className" : "mapping.OntologyTerm",
  "name" : "Term 1",
  "description" : "Term 1 Description",
  "namespace" : "namespace",
  "obsolete" : false
}
{
  "_id" : "term2",
  "className" : "mapping.OntologyTerm",
  "name" : "Term 2",
  "description" : "Term 2 Description",
  "namespace" : "namespace",
  "obsolete" : false
}
{
  "_id" : "term3",
  "className" : "mapping.OntologyTerm",
  "name" : "Term 3",
  "description" : "Term 3 Description Updated!",
  "namespace" : "namespace",
  "obsolete" : false
}
```

Result

We may decide that we want to remove a field. This can be done as follows:

```
UpdateOperations<OntologyTerm> update2 =
datastore.createUpdateOperations(OntologyTerm.class).unset("description");
```

The unset method removes the field.

```
datastore.update(query, update2);
```

```
{
  "_id" : "term3",
  "className" : "mapping.OntologyTerm",
  "name" : "Term 3",
  "namespace" : "namespace",
  "obsolete" : false
}
```

Result

Querying the Data

Since our Java objects are pretty much directly mapped to the database, we have the huge advantage that querying becomes simpler than in the Neo4J or PostgreSQL version of the InterMine data storage.

To execute a query, we will first need to parse a given PathQuery and build our query accordingly.

Brief overview of querying in Morphia

Recall the Query object that Morphia provides, which we generally initialize as follows:

```
Query<QueriedClass> query =
datastore.createQuery(QueriedClass.class);
```

Where QueriedClass is the object that we want to query.

We can use various methods which append query conditions to build a specific query. After we are done with building our query, we can retrieve the result as a List:

```
query.asList();
```

Or as an iterator:

```
query.iterator();
```

Field method

```
query.field("field");
```

Specifies to only check against the indicate field. As a result, we may chain a query like so to only return objects where "field" is equal to "value":

```
query.field("field").equal("value");
```

Instead of using equal(), we have a lot of other choices here, such as contains() [pattern], hasAllOf(), hasAnyOf(), etc. which will come very handy when building the queries.

Chaining Appends

As a side note, we may also chain appends and append as many times as we want, for example like this:

```
query.field("field1").equal("value1").field("field2").equal("value2");
```

And then this query would return the objects where “field1” value is “value1” and “field2” value is “value2”. Whenever we call these methods on a query, the conditions are automatically added to the query in-place.

Parsing PathQuery queries to MongoDB

Note that the majority of my plan here is just a concept. It does not include the most optimal solutions, but just a direction in which I will be going in.

Defining Output

Supposing we have the following query:

```
<query model="genomic" view="Publication.pages
Publication.year"/>
```

The rough general approach, in my eyes, would be:

1. Retrieve the view of the PathQuery object
2. For each String in the view list, split it at the ‘.’ character.
3. From the split strings, we iterate from the 1st to the (i-1)-th element. We follow the path of access.
4. The very last string indicates the field that we want to access. This is the field that we are going to project. Everything before that is the object we should access.

After parsing, our result query would look something like this:

```
Query<Publication> publicationExample1 =
    datastore.createQuery(Publication.class).project("pages",
true).project("year", true);
```

Note the project here. In MongoDB, when the project method is called, automatically all values are hidden except the attributes that are specified in the projection (much like in SQL)

Null values

```
<query model="genomic" view="Publication.title
Publication.issue">
  <join path="Publication.issue" style="OUTER"/>
</query>
```

One of the key differences between SQL querying and Morphia MongoDB querying is that even if by using `project()` we're looking for a certain type of value and it's null, the document is still returned. Therefore, for OUTER join, we will typically not need any extra case and simply return the documents as-is (so we actually need to handle the INNER join case, which is opposite to SQL handling for the OUTER join case)

However, provided the join is not OUTER, we will want to filter only the results that do not have the value as null. To do so, our query would look, for example, like this (provided the JOIN specified there is INNER):

```
Query<Publication> publicationNullQuery =
    datastore.createQuery(Publication.class)
        .project("title", true)
        .project("issue", true)
        .field("issue").notEqual(null);
```

Constraints

The general approach to begin parsing constraints:

1. From the PathQuery object, call the `getConstraints()` method that returns a Map of the query constraints.
2. Iterate over each constraint, and parse it accordingly to the guidelines below.
3. For getting the path, I will follow the same methods as I would in getting the view path. (Luckily, we have a `getPath()` method to get the path, but we will still need to parse it, which would be done in the same manner as the general approach that I will follow for the previous section)

Attribute Constraints (PathConstraintAttribute)

```
<constraint path="Publication.title" op="="
value="Publication 1"/>
```

Consider the following constraint in a query. The rough general approach I would take for parsing such a query is as follows (assuming 3 general approach steps are done):

1. `getValue()` of the PathConstraintAttribute. This will be used for the field method.
2. `getOp()` of the PathConstraint. This will be used for the field operation.
3. Map `getOp()` result to Morphia method.
 - a. `ConstraintOp.EQUALS => equals()`
 - b. `ConstraintOp.GREATER_THAN => greaterThan()`
 - c. `ConstraintOp.CONTAINS => contains()`
 - d. Etc.

The resulting query of the above would look as follows:

```
Query<Publication> publicationExample2 =
    datastore.createQuery(Publication.class).field("title").equal
("Publication 1");
```

MultiValue Constraints (PathConstraintMultiValue)

```
<constraint path="Publication.title" op="ONE OF">
  <value>Publication 5</value>
  <value>Publication 7</value>
  <value>Publication 12</value>
</constraint>
```

Parsing multi-value constraints are really straightforward. Supposing we parse the path and `getValue()`, Morphia has a method to directly check if a value is in a given list. The resulting query therefore looks like:

```
Query<Publication> publicationMultiValue =
datastore.createQuery(Publication.class).field("title").hasAnyOf(
pathQueryObject.getValues());
```

Similarly, for List constraints, we may specify “hasAllOf” instead to check if the array is a subset of the given list.

For OVERLAPS queries (PathConstraintRange), however, we will need to devise some more complicated parsing logic.

Lookup Constraints (PathConstraintLookup)

```
<constaint path="Publication" op="LOOKUP" value="9"/>
```

There is one prerequisite if we want lookup constraints to work. We must add a `@Indexes` annotation for wildcard searching on the class we want to do the search on (so this case, on `Publication`, but ideally we'll want this index on every class), which will allow us to use Morphia's `search()` method, which will then search all the fields for the specified value:

```
@Indexes(@Index(fields = @Field(value = "$**", type = TEXT)))
```

This annotation is added on top of the class definition, in the same place where the `@Entity` annotation is added.

Having retrieved the value from the `PathConstraintLookup` (and potentially the `extraValue`, which we can then use for a second search), and doing the mentioned prerequisite, a final Lookup Constraint query would look as follows:

```
Query<Publication> publicationLookupQuery =
datastore.createQuery(Publication.class).search("9");
```

Loop Constraints & Joins (PathConstraintLoop)

```
<constraint path="Publication.title" op="="
value="Author.initials"/>
```

The hardest parts to handle in querying are joins & loop constraints due to the increased complexity of the query. With the relatively [recent changes in MongoDB](#), aggregation & joins are now possible in MongoDB and are fit for this task.

Deciding upon how to do MongoDB aggregations in these cases is quite a complex topic and requires the previous steps of this implementation plan to be finished to decide on the optimal solution. However, here is an example of writing a query for the following constraint:

```
AggregationPipeline aggregationPipeline =
datastore.createAggregation(Publication.class)
    .project(Projection.projection("title"))
    .lookup("Author", "title", "initials",
"titles-initials")
    .unwind("titles-initials");

AggregationOptions aggregationOptions =
AggregationOptions.builder().outputMode(AggregationOptions.Ou
tputMode.CURSOR).build();
Iterator<Publication> iterator =
aggregationPipeline.aggregate(Publication.class,
aggregationOptions);
```

The steps of the code go as follows:

1. We first create an AggregationPipeline, since we require a complex aggregation query.
2. We project only the "title" of the Publication, since this is our field of interest.
3. Then we do a lookup - this is essentially a left outer join in MongoDB. We compare Author's initials field with Publication's title field, and store the matches in "titles-initials".
4. We then unwind "titles-initials", giving us documents where "titles-initials" is a field of a single nested object, rather than array.
5. One more step that is not done here: We do not keep distinct values only. It is possible that we would, for instance, have 5 instances of the same Publication. Additional logic is required here to eliminate duplicates.
6. Afterwards, we need an AggregationOptions object to retrieve our aggregation result.
7. We retrieve our aggregation result as an iterator of Publication, on which we can work on further.

The code is essentially equivalent to the following MongoDB query:

```
db.Publication.aggregate([
  {
    $lookup:
    {
      from: "Author",
      localField: "title",
      foreignField: "initials",
      as: "titles-initials"
    }
  },
  { $unwind: "$titles-initials" }
```

```
] )
```

This query will return all Publication objects in which the “title-initials” array is of size > 0, meaning that a match has been found. This satisfies our loop constraint conditions.

Joins, in general, will be the most time consuming and complex task. They are hard to optimize and will need careful designing to reach optimal performance.

Sort Order

```
<query model="genomic" view="Publication.title"
sortOrder="Publication.title DESC Publication.year ASC"/>
```

For sort order, the general approach would be:

1. Get the sort order by getOrderBy() method from the PathQuery object
2. For each OrderElement element in the list, parse the OrderPath to get the attribute to sort on.
3. Together with the path and OrderDirection, create a Sort object for each entry.
4. Map the ordering to Morphia methods

The resulting query would then look as follows:

```
Sort[] sorts = new Sort[] { Sort.descending("title"),
Sort.ascending("year") };
```

```
Query<Publication> publicationSort =
datastore.createQuery(Publication.class).project("title",
true).order(sorts);
```

Query Optimization

It is crucial that our search queries will be optimized. Although it's hard to say which optimization techniques will work best without having the full design of the data model, there are a few general things to take care of. Denormalization is not included in this section, because this is a decision to make while designing the data model.

Indexing

One of the major optimization steps to take is indexing, since that will make search queries extremely fast compared to no indexes. Moreover, special indexes are required to satisfy lookup constraints.

However, one important thing to note is that if we have too many indexes, then we are, in fact, losing performance. Therefore, I will need to find the right balance such that the performance is most optimal. The general idea is that the indexes should fit in RAM for them to be fast, since disk access is significantly slower than RAM access.

Default indexes & Viewing Indexes

It is important to note that MongoDB automatically creates indexes for the “_id” field, which is highly advantageous for us, since if we choose to use references, they should reference “_id” fields, which are pre-indexed by default.

Consider the example of the Publication class. We have added an additional text index, and we may view this index in MongoDB by specifying `db.Publication.getIndexes()`. Here is the result, where we can verify that “_id” is indeed indexed:

```
> db.Publication.getIndexes();
[
  {
    "v" : 2,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_",
    "ns" : "testdb.Publication"
  },
  {
    "v" : 2,
    "key" : {
      "_fts" : "text",
      "_ftsx" : 1
    },
    "name" : "$**_text",
    "ns" : "testdb.Publication",
    "weights" : {
      "$**" : 1
    },
    "default_language" : "english",
    "language_override" : "language",
    "textIndexVersion" : 3
  }
]
```

Specifying Indexes

Specifying indexes in Morphia is done via [annotations](#). For instance, considering the example of the Publication class and a single text index, the top of our class definition looks as follows:

```
@Entity
@Indexes(@Index(fields = @Field(value = "$**", type = TEXT)))
public class Publication {
```

Alternatively, we can create indexes via the base [Mongo Java Driver](#) functionality.

One more important thing to note is that after adding our indices, we need to call: `datastore.ensureIndexes()` ;

This ensures the indices (in other words, builds them) on all the classes.

However, let's look into a more general example. Suppose we have our example `OntologyRelation` class and we want to index `childTerm` and `parentTerm`. Then we may specify two indexes as follows:

```
@Entity
@Indexes({
    @Index(fields = @Field("parentTerm")),
    @Index(fields = @Field("childTerm"))
})
public class OntologyRelation {
```

And then we can indeed see that the indices have been properly applied:

```
> db.OntologyRelation.getIndexes();
[
  {
    "v" : 2,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_",
    "ns" : "testdb.OntologyRelation"
  },
  {
    "v" : 2,
    "key" : {
      "parentTerm" : 1
    },
    "name" : "parentTerm_1",
    "ns" : "testdb.OntologyRelation"
  },
  {
    "v" : 2,
    "key" : {
      "childTerm" : 1
    },
    "name" : "childTerm_1",
    "ns" : "testdb.OntologyRelation"
  }
]
```

Indexing Options

There are a variety of indexing options available when creating indexes, all fit for different situations. When working on query optimization, I will explore them in greater detail and apply them properly to ensure the most optimal indexing.

In general, this will be a trial and error task. My general approach will be:

- Use MongoDB's `explain()` method to analyze the query execution time
- Try to find an optimal way to apply indexes, and then do so.
- Analyze the query runtime again

- Query became faster? -> Test with more queries to see if they also benefit from it such that it's worth keeping the indices
- Query became slower? -> Discard created indices, try something else

Search Query Optimization

As per the [official MongoDB documentation](#), we must ensure a few strategies to make the search queries as optimal as possible. Besides indexing, there are a few extra steps we can take.

Query Selectivity

If our queries are as selective as possible, meaning there are specific match & filter predicates to reduce the size of the result set, the queries will be significantly faster. Therefore in every situation where we can reduce the document count of the query, we should. This means that most of our matching logic should be through MongoDB, rather than filtering the data manually in the Java application.

Query Covering

If we cover our query properly with only the required fields, we will see an increase in performance. That is, we should use project() as much as possible, because this can significantly reduce the size of the retrieved documents. Moreover, we may specify a limit for the size of document set to retrieve (such as for the result set of one page). Combined with selectivity, our result set becomes smaller and therefore the data is obtained faster.

Lookup (Join) Optimization

Lookups also have techniques for optimal performance, as per [MongoDB official documentation](#). If we follow a few crucial steps, we would gain significant performance while joining collections.

Sequence optimization

We should specify match conditions as soon as possible to decrease final aggregated collection size. This is highly important to do, and we might even consider splitting up matches and placing them in between for full optimization.

Choosing the right collection to left outer join on

It might be the case that joining from one collection is more expensive than from another, because it might be that we have more conditions on one collection than the other, therefore we might be more selective in our query.

To reach the best possible optimal performance, I will have to decide on certain ordering & sequencing in our Java application logic.

Benchmarking

Luckily, MongoDB provides the explain() method, which provides information about the query performance, such as how many documents were accessed, how many indexes were scanned, how the collections were scanned (column scan, index scan, ...) and so on. Morphia also has the method in-built, we can simply call query.explain() to retrieve information about the query.

Taking advantage of Morphia's and Java's capabilities, I plan to build a benchmarking tool that will display the performance of the query. I plan to make the benchmark tool as follows:

- Run query.explain() on a specified Query object (while also keeping track of elapsed time, possibly rerun this more than once for more accurate time)
- Parse the necessary data (query specifics, recorded time)
- Build an .html file, which will compactly visualize the data. Name the .html accordingly (query name, date)
 - Potentially use some sort of frontend framework (such as Bootstrap) for charts
 - JavaScript might be the better way to go to generate the .html
- Bonus: If time allows, I will also include PostgreSQL and perhaps even Neo4J query performance results, so that they may be compared with MongoDB's.

RESTful API integration

After everything is done above, I will be taking the final step of integrating the database as a RESTful API. Thanks to the PathQuery API, this should not be a very delicate task, and the database setup stage in the previous section will help us a lot.

The way I imagine doing this right now is as follows:

- Use the utility class mentioned in the database integration section as the main service for the REST controller
- Make a main service class which actually runs the REST service
- Program a REST controller, which interacts with the main database utility class to access data
- Let all requests map from URL to the service methods
 - This will be based on the PathQuery API & the current SQL implementation
 - GET requests when querying with PathQuery API
 - Arguments for Read operations (search queries) should be JSONized PathQuery objects
 - Possibly POST requests if update/delete functionality is of relevance
- Return the appropriate results as JSON in the REST controller
 - I will make sure this works without any need to change the actual code which handles receiving responses, meaning that I will ensure that the responses are the same, no matter if PostgreSQL or MongoDB is used.

Timeline

Milestones

Commitments

<Omitted>

Community Bonding (May 6 - May 27)

- Meet the community and mentors
 - Introduce myself to everyone, get acquainted with the community in general, get well acquainted with the mentors.
 - Understand the workflow in the InterMine organization
 - Establish primary communication options
 - Discuss evaluation process
- Understanding the codebase
 - Take time to understand the relevant parts of the codebase. There are still parts of the codebase that I have not yet analyzed and they might be crucial to understand while working on this project.
- Further learning
 - Research MongoDB optimization deeper - for example, are there any scenarios in the project where a certain query is more faster than the other?
 - Java REST - since this is my weakest part of the required skills (although I am capable), I will require to understand all the functionality of the codebase API and, if needed, study REST APIs additionally.
 - Swagger - I have seen from the previous Neo4J project that this was used throughout the project and have not had any experience with Swagger. I will therefore spend time to learning & understanding it.
 - While analyzing the relevant parts of the codebase, I might run into unfamiliar concepts. In that case, I will spend time to learn and understand them.

Phase 1 Coding (May 27 - June 28)

Week 1 (May 27 - June 3)

On the first week, I will be setting up MongoDB to work with the InterMine environment or a completely new environment (however it is specified by the mentors)

This will include all the mentioned classes in the implementation plan, and a fully working, easily extendable MongoDB environment with barebones classes and some standard functionality (such as connecting to the database, retrieving an entire collection, adding data, etc.)

By the end of this milestone, the following should be done:

- There is a working MongoDB environment in our Java Project
- All required dependencies are installed (namely Morphia)
- A high level of test coverage is reached using automated tests (possibly unit tests for now, although integration tests are also possible). This also includes picking the right choice to test the database (possibly by using an embedded MongoDB database)
- Proper documentation of all the classes and how they should be used.

Week 2-3 (June 3 - June 17)

Having setup the database environment, I will be working on designing the data model. I will mostly be making tough decisions for optimal data storage (when to use references, when to use embedded documents, how to separate collections) and finish the concept of the MongoDB data model for the IM data model (the one specified in this implementation plan).

Moreover, after having carefully designed the data model, I will be working towards properly constructing the Java classes for the needs of the MongoDB IM data model, mapping Java classes properly to the database and then populating the database with data via build commands (much like the SQL data is built right now)

- The full MongoDB data model & mapping is documented fully for the IM data model
- Full building of Java classes to adapt to the MongoDB data model
- Mapping of Java classes to MongoDB
- Ability to populate data automatically from files
- Appropriate testing for the implemented functionality
- Full documentation of the mentioned functionality
- If things go wrong: I will then delay the work to the next week. I believe issues might arise when importing data into the database, therefore it might require a bit more time. For instance, FASTA and GFF3 file importing might take longer than expected.

Week 4 (June 17 - June 24)

Having (at least mostly) the logic for mapping the data into the database and the ability to construct the database itself, I will then be working on the Java logic to handle Create, Update, Read and Delete queries (direct follow up to Week 1 setup). This week I plan to entirely finish the actual generalized query code for the database, and setup code for PathQuery (search query) parsing. Moreover, due to my absence in Phase 2 and the evaluation period, I will start doing work corresponding to Phase 2 and be sure to put in extra work hours. **Therefore, Week 6-7's work also corresponds to Week 4's work in my plan due to absence!**

Also, this is also the week where I will be handling any issues that occurred in Weeks 1-3 and could not be done due to potential lack of time.

In general, I will be doing the following:

- Finished general logic for querying the database
- General method code for querying the database
- The above two will ensure a seamless start for PathQuery parsing, and as such, work on parsing will start this week.
- Documentation and initial implementation (setup) of PathQuery query parsing
- Query method testing (possibly integration testing)
- + Week 6-7 work

Phase 1 Evaluation (June 24 - June 28)

During Phase 1 Evaluation, I will use this time for reflection of any issues that I encountered and any flaws that I had during the first month of working. Moreover, I will be putting some time into researching more into the work that needs to be done for the upcoming weeks.

Phase 2 Coding (June 28 - July 26)

Week 6-7 (June 28 - July 15)

On these two weeks, I plan to spend the majority of time on parsing the queries from PathQuery to MongoDB. This will include writing the necessary Java logic to efficiently convert the query to a Morphia query object, which will then be used to return the right result.

- Query parsing logic & strategy implementation
- ~60% of the constraints implemented & successfully parsed
- Proper documentation for query parsing
- Unit tests for query parsing

Week 8 (July 15 - July 22)

This is a direct follow-up to the previous milestone. This week, I plan to fully finish PathQuery parsing, which should mainly include implementing the remainder of the constraints and query conditions.

The work of this week also heavily depends on the progress of the last 2 weeks. If by the beginning of this week I am nearly done, I will simply move on to the work of Week 10.

- Query parsing logic & strategy continuation - mainly constraints and conditions, such as sorting
- 100% of the constraints implemented & successfully parsed
- Proper documentation for query parsing & unit tests

Phase 2 Evaluation (July 22 - July 26)

During the Phase 2 Evaluation, again I plan on reflecting on my mistakes and issues that I had during Phase 2. This week, I also plan to put in a lot more work to make up for the lost time when I was absent. I will absolutely make sure that during Phase 2 evaluation, all of the work is done, and I have an excellent head start and some work done for Phase 3.

Phase 3 Coding (July 26 - August 26)

Week 10 (July 26 - August 5)

Assuming everything from the previous weeks is in working order, I will then take the steps to implement the database as a RESTful API to work with InterMine services. Should any major problems arise and I find myself needing more time on this part of the project, I will push some of the work to Week 11.

- REST controllers that link to the database
- Assured integration with InterMine services (making sure that migration from SQL to MongoDB is seamless)
- Full documentation of how to use the controllers
- Testing the API (possibly using integration tests, or Mock testing)

Week 11-12 (August 5 - August 19)

Now that the base, core functionality has been done, I will then be spending time to fully optimize the database. This will include indexing, analyzing which indexes work best, documenting via benchmarks, optimizing current queries (especially joins) as well as looking for alternative solutions for the most optimal querying.

I would also like to create some sort of benchmarking tool, since that is possible with Java's & Morphia's capabilities. I would consider this an optional feature, but ideally, if time allows, I would like to make this visualized and highly informative for the IM data model. Moreover, I am considering (provided there is time) to make comparisons with PostgreSQL and Neo4J in their current working state in terms of querying time to get a good impression of MongoDB as the backend database.

- Indexing strategy implementation
- Query performance optimization
- Lookup optimization
- Benchmarks & documentation of query performance
- [OPTIONAL] Benchmarking Tool

Phase 3 Evaluation (August 19 - 26)

During the final evaluation, I will make sure that the completed product is 100% working, there are no bugs, the documentation is properly completed and clear and then I will finalize my final report.

Deliverables

Deliverables:

- Java project (new repo) with necessary dependencies for MongoDB installed & set-up
- Working MongoDB Java setup that connects to the database and allows accessing database instances, collections and documents
- Full design of the IM data model adapted for MongoDB - in the form of

- documentation, schemas/diagrams
- Java Class construction fit for MongoDB
- Automated data population to the database (much like is done now with PostgreSQL)
- Data importing into the database (GFF3/FASTA)
- Create, Read, Update, Delete (CRUD) operations implemented as Java code (Delete Optional!)
- PathQuery parsing to MongoDB queries in the form of Java code
- PathQuery constraint realization in MongoDB
- 100% working search querying from PathQuery to MongoDB
- RESTful API for the MongoDB implementation that can seamlessly be integrated into InterMine
- Integration Tests
 - Integration testing for MongoDB, possibly by using an embedded database.
- Unit Tests
 - Mock tests to test API functionality
 - Relevant tests for methods, such as query parsing
- Proper documentation of the entire work
 - Will be delivered on a milestone basis (if the milestone spans two weeks, then I will make sure to have the documentation ready by the end of the milestone)
- Query Optimizations & Progress Reports
- Search Query Performance Benchmarks
- [OPTIONAL] Search Query Benchmark Tool
- [OPTIONAL] Benchmark comparison with PostgreSQL
- [OPTIONAL] Benchmark comparison with Neo4J
- Final Report
- Final Presentation (15 August, 2019 5pm BST)

Personal background

(limit to one page)

General

Education: First Year Computer Science & Engineering student in TU Delft

Personal Website: <https://elatoskinas.github.io/>

GitHub: <https://github.com/elatoskinas>

Operating Systems: Windows 10, Ubuntu (Native Windows user but well capable with Linux)

Skills

- Java (extensive)
 - [MongoDB Java Driver](#)
 - Spring MongoDB Integration. This included coding & using a REST Controller for client-server interaction, utilizing the DAO pattern, writing extensive queries, adding & importing data into the database, writing unit & integration tests for the database.
 - Object-Oriented Programming, Algorithms & Data Structures, OOP Project courses in University
- Databases
 - SQL (mainly PostgreSQL)
 - Query exercises in University
 - Relational Database extensive knowledge gained from Information & Data Management course
 - Android Application storage (SQLite was used)
 - MongoDB
 - Query exercises in University
 - Neo4J
 - Query exercises in University
 - A lot of knowledge that I hold for the 3 above mentioned databases comes from the Web & Database Technology course in university
- JavaScript & Node.js
 - The gained understanding of how servers, route (path) mapping, requests and responses work heavily assisted me in understanding how APIs and REST works.
- Git
 - Extensively being used in my learning process, both for open source development & collaboration projects (mainly university)
- Other programming skills that I coded in: C++ (extensive), C# (extensive), Python (beginner), Assembly (intermediate)

Projects & Work

- University OOP Project (Java, Spring, MongoDB)
 - Ongoing Java Desktop Application project with 6 other people. The application is an online CO2 tracking app with built-in social media-like functionality. I am responsible for the data storage and queries, which I do in MongoDB and Spring.
- [Chess Web Application](#) (JavaScript, Node.js, Express)
- [Open Source Contributions](#) (MIT App Inventor)
- [FinBuddy](#) (Android Studio, Java)
- [Sphere Archer](#) (Unity3D, C# language)
- [Virtual Poker Chips](#) (Unity3D, C# language)
- [University Project](#) (Assembly Game)

- Programming Competitions & Challenges (Mainly C++)
 - [Lithuanian IT Olympiad Final participation](#)
 - [HackerRank](#)