# Chapter 5

# Schema Implementation

This chapter focuses on the implementation of the data models discussed in chapter 4. Each database management system has their own procedure for instantiating a new collection, table, node or column family. This chapter discusses the methods and strategies I imposed to create the database solution designs; with a focus on any challenges faced in doing so. Chapter 6 evaluates the process undertaken to physically load the data into the systems.

## 5.1 Creating database systems

Once the process of modelling of the database systems was complete, the next stage was to transform the model plan into actual databases. For each database system, this was relatively simple. However, with this simplicity, brought limitations and restrictions of which I had to construe, to fully achieve my target model.

### 5.1.1 MySQL - schema implementation

The MySQL data model consists of 8 tables; AnatomyStructures, Assays, Genes, Publications, Sources, Specimens, Stages and TextAnnotations. Each of which are discussed in detail in section 4.2.1. The creation of these tables was a relatively straightforward undertaking. The ease in which I found this process, may be due to my previous experience of using MySQL. Another rational explanation would be that the intuitive and logical way in which relational databases are constructed, make implementing a data model, an all round elementary procedure. An example of how a table is created in MySQL can be found in the code snippet 5.1 below. This code illustrates the creation of the Assays table, the indexes and the constraints.

```
--
-- Table structure for table `Assays`
--
CREATE TABLE IF NOT EXISTS `Assays` (
  `emage_id` int(11) NOT NULL,
  `type` varchar(255) DEFAULT NULL,
  `probe_id` varchar(255) DEFAULT NULL,
  `source_id` int(11) NOT NULL,
  `specimen_id` int(11) NOT NULL,
  `stage_id` int(11) NOT NULL,
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
--
-- Indexes for table `Assays`
--
ALTER TABLE `Assays`
  ADD PRIMARY KEY (`emage_id`),
  ADD KEY `source_id` (`source_id`),
  ADD KEY `specimen_id` (`specimen_id`),
  ADD KEY `stage_id` (`stage_id`);
--
-- Constraints for table `Assays`
--
ALTER TABLE `Assays`
  ADD CONSTRAINT `fk_Assays_Sources` FOREIGN KEY (`source_id`)
  REFERENCES `Sources` (`source_id`),
  ADD CONSTRAINT `fk_Assays_Specimens` FOREIGN KEY (`specimen_id`)
  REFERENCES `Specimens` (`id`),
  ADD CONSTRAINT `fk_Assays_Stages` FOREIGN KEY (`stage_id`)
  REFERENCES `Stages` (`id`);
```

Code snippet 5.1: Creation of Assays table in MySQL.

As you can see in lines 4 - 11, the creation of each column takes the form of; column name, data type with length and the default attributes and values i.e. null or not null. The character set for each table, by default was set to utf8. You will notice here that the key values were not in fact instantiated at time of creation. This was as a result of an experiment to evaluate the affect the exclusion of index keys and constraints has at time of data load on each database. The findings of this experiment are illustrated in figure **??** with reference to the performance of the other database solutions for comparison.

Intuitively, one would expect quicker load times and a slower querying performance with no implemented indexes. Figure **??** clearly illustrates that the time taken to load the data without index keys and constraints had little, to no true affect on load performance. While there is slight variance in load time, this can be expected and attributed to a number of reasons, such as other processes running simultaneously on the CPU for example. For in-depth discussion and comparison of use cases, see section 8.

MySQL tables are linked by joining the (unique) primary key of one column to the (unique or non unique) foreign key of another. Lines 15-19 in code snippet 5.1, is where the key columns are created and lines 23 - 29 is where the foreign key constraints are expressed. The notion of keys joining tables can often be a slightly confusing concept to understand on first encounter. Primary and foreign keys are, not always, but in most cases confined to integer values. This is as a result of, data often containing inconsistent, ambiguous and non universal values. For example, a primary key may have the value “Mouse” and a foreign key may have the value “mouse”. Both valid strings however as they do not match exactly the join would fail. The rigidity of these constructs have as many advantages as they do disadvantages. While the concept of joining two tables on matching integers seems logical, many situations occur where there is no unique ID present in the dataset and therefore the ID has to be manually created based on the data available.

The process described was repeated for each of the tables in the devised data model. Creating multiple tables and joining them together in MySQL is relatively straightforward. While the formality of definitively expressing each term and its data type then stipulating the index keys and constrains, can be a tedious process, it is done so in a logical and objective manner, which makes it coherent and understandable for the programmer.

### 5.1.2 MongoDB - schema implementation

Creating a document inside a MongoDB collection (the equivalent to a MySQL table) is done by inserting field and value pairs. Each time a new field and value pair is inserted into a collection, a new document is created. By default, each document in a collection is provided with a unique ID which has an object data type. ObjectIds are small, fast to generate, and ordered. These values consists of 12-bytes, where the first four bytes are a timestamp that reflect the ObjectId's creation [**?**]. A unique ID can also be manually created for each document, if available. By this I mean, if a dataset has a unique identifier, which will be used as a reference, this can be implemented by definitively expressing the "_id" field, to a value. For example "_id : 1234" would by the ID of a single document. This concept is illustrated in line 2 of code snippet 5.2.

MongoDB is an extremely flexible data store. It accepts multiple different data types, from the standard; string, double and boolean values to the more complex regular expressions and even Javascript code. By default, any value without a type cast will be presumed to be either a sting or integer value. This attribute is common of NoSQL systems. It adds to the simplicity and smooth process of implementing a data model.

Documents can be created by either manually inserting on the command line, or by conducting a data dump. The latter is discussed in more detail in section 6.1.2. Code snippet 5.2 below, is an example of how a document can be created and data inserted from the command line.

Creating documents in MongoDB is an extremely simple process. One command and you can insert an entire database of information. While it is unlikely that one would manually insert a large volume of data using the *db.collection.insert({})* method, it is an available option. It is more likely that one would use this insertion process for adding additional data to an already implemented database, as opposed to creating from scratch. For example the EMAGE dataset contains around 200,000 entries. Inserting the full dataset using this methodology, while valid, would certainly not be the optimal solution. An explanation defining the full procedure into how I created the MongoDB documents can be found in section 6.1.2.

An important design decision which has to be taken prior to model implementation is, whether to create multiple collections or embed data within a document. A comprehensive

evaluation of the advantages and disadvantages is discussed in section 4.2.2. The key difference between the two approaches is, that within a multiple collection database, querying is required to be undertaken at application level and is done by referencing data entities. Whereas in an embedded document, all of the data is available in a single schema and can be accessed by a single query. Thus improving query performance and application response time.

As discussed in section 4.2.2 and illustrated in code snippet 5.2, the MongoDB data model I have designed uses the embedded document approach. What do I mean when I say embedded? A good way of thinking of the embedded data concept is, a document in a document. For example, while values in MongoDB can be absolute, such as a string with value "mouse" they can also be an embedded document which in turn has values such as "mouse". Lines 6-34 of code snippet 5.2 represents an example of data, embedded in a document. Looking specifically at lines 14-20, we have a value namely "publication" which has embedded values of "publicationID", "author" and "title". Thus allowing direct querying of additional data, as opposed to messy collection joins.

```
db.emage.insert({
    "_id" : 5354,
    "probeID" : "Flt1 probeA",
    "source" : "emage",
    "type" : "in situ",
    "specimen" : {
        "strain" : "unspecified",
        "type" : "wholemount"
    },
    "stage" : {
        "dpc" : "9.5 dpc",
        "theilerstage" : 15
    },
    "publication" : [
        {
            "publicationID" : 9113979,
            "author" : "Ema M, Taya S, Yokotani N, Sogawa K, Matsuda Y, Fujii-
                Kuriyama Y",
            "title" : "A novel bHLH-PAS factor with close sequence similarity to
                hypoxia-inducible factor 1alpha regulates the VEGF expression and is
                 potentially involved in lung and vascular development."
        }
    ],
    "textannotation" : [
        {
            "anatomystructure" : {
                "term" : "cardiovascular system",
                "structureID" : 16104
            },
            "strength" : "detected",
            "gene" : {
                "name" : "Flt1",
                "geneID" : "MGI:95558"
            }
        }
    ]
})
```

Code snippet 5.2: Example insertion of data into a MongoDB document.

### 5.1.3 Neo4j - schema implementation

To implement a Neo4j structure we use a query language namely Cypher Query Language (CQL). Cypher, is a declarative graph query language that allows for expressive and efficient querying and updating of a graph store [**?**]. CQL was designed to be as user friendly as possible for both programmers and operations professionals alike [**?**]. The structure of CQL is based upon SQL and shares many of its attributes. CQL queries are built using various clauses. For a detailed insight into CQL, see section 2.7.3.

As Neo4j is based upon the property graph model, a database is implemented by constructing nodes and relationships. A node is made up of either a single or a number of properties. A property is a value which is named by a string. The accepted property values in Neo4j are: Numeric, String, Boolean and Collections of any other value type, for example, an array of Strings. A relationship organises the nodes by joining two nodes together on a shared common value. As with nodes, relationships can have definitive value properties.

A node created in Neo4j is automatically instantiated with an ID value. Each and every node in the database has a unique numerical ID, which is incremented from the first node to the last inserted. This value can not be changed. Creating relationships between the nodes and querying the graph, is generally done by imposing various clauses upon the node values as opposed to this unique ID. However this ID value can indeed be used as or within a query clause by using the “ID” clause function.

Creating a Neo4j structure can either be done by implementing a Cypher file; containing the indexes, nodes, constraints and relationships of the data model. The Cypher file can then be bulk loaded into the database. Alternatively a data model can be manually implemented using the Neo4j shell command prompt. Code snippet 5.3 is an example of how to create a structure from the command line. However, for the full EMAGE dataset instantiation, I created a Cypher file and imported the dataset simultaneously. The implementation of this is discussed in section **??**, which also describes how to load data into Neo4j fom a CSV file. Code snippet 5.3 illustrates the implementation of an assay, publication, indexes and constraints via the command prompt.

```
1  //
2  // Create node indexes and constraints
3  //
4  CREATE CONSTRAINT ON (e:Assay) ASSERT e.id IS UNIQUE;
5  CREATE INDEX ON :Assay(emageID);
6  //
7  // Create Assays
8  //
9  MATCH (source:Source {source : 'emage'})
10 MATCH (specimen:Specimen {strain : 'unspecified', type : 'wholemount'})
11 MATCH (stage:Stage {theilerstage : TOINT('15'), dpc : '9.5 dpc'})
12
13 CREATE (assay:Assay {emageID: TOINT('10001'), probeID : 'Epas1 probeA',
14     type : 'in situ'})
15
16 CREATE (assay)-[:COMES_FROM]->(source)
17 CREATE (assay)-[:CLASSIFIED_AS]->(specimen)
18 CREATE (assay)-[:GROUPED_BY]->(stage);
19 //
20 // Create Publications
21 //
22 MATCH (assay:Assay {emageID : TOINT('10001')})
23
24 CREATE (publication:Publication { accession: TOINT('9113979'), title : 'A novel
    bHLH-PAS factor with close sequence similarity to hypoxia-inducible factor 1
    alpha regulates the VEGF expression and is potentially involved in lung and
    vascular development.', author : 'Ema M, Taya S, Yokotani N, Sogawa K, Matsuda Y
    , Fujii-Kuriyama Y'})
25
26 CREATE (publication)-[:DESCRIBES]->(assay);
```

Code snippet 5.3: Example creation of an assay publication indexes and constraints in Neo4j.

Creating an indexes and constraints in Neo4j can be done at any time during implementation. The process is a simple, one line command for each. Lines 4 and 5 in code snippet 5.3 illustrate this. Imposing unique constraints on a node is more relevant when importing multiple nodes at one time by using the "merge" command. This is discussed in detail, in section **??**.

As illustrated in code snippet 5.3, there are 3 main stages when creating a node in my Neo4j data model. Let's take a look at the creation of the Publications node on lines 22-26. Firstly we match another node, "Assay", with a value which we corresponds with our publication node. In SQL terms this is essentially a join. Line 22 is saying "Join this node I am creating, with the assay node which has an ID of 10001". We then move on to create the publication node. To do this, we state the name of the node, then add in the field and value pairs we are looking to associate with this node, in a JSON format. Finally we create the relationship of the publication node. Line 26 represents the creation of this relationship. State the name of the parent node, define the relationship (with a string value of your choosing) then state the name of the child node. As a publication describes an assay, I have named the relationship in this case "DESCRIBES". The two nodes are then joined together as a result of the match we created in step 1. The ordering of this process can be rearranged. The creation of the node can come before the matching stage, however the match must come before the relationship creation. It is evident when creating a Neo4j data model, that the attributes of CQL were certainly implemented with simplicity in mind. The ease in which nodes and relationships are created is effortless.

### 5.1.4 Apache Cassandra - schema implementation

# Chapter 6

# Importing Data

As discussed in section 2.2, the load stage of an ETL procedure can often become the most time consuming phase of the pipeline. This chapter describes the implemented procedures and examines the functionality each solution provides to complete the data load process.

## 6.1 Put the data in databases

The final stage in the data modelling process was to import the EMAGE dataset into the created data models. For each of the database systems, a different approach was required. To ensure a balanced and impartial evaluation, each of the datasets were converted into a CSV file format and manipulated by the functional tools the respective systems provide.

### 6.1.1 MySQL - data load

There are various ways in which data can be loaded into a MySQL database. You can manually insert the data, row by row in the MySQL shell command prompt, using an **INSERT INTO** statement. However, to implement a full database using this method would be extremely time consuming and laborious. While time may not be of the essence in certain circumstances, manually writing 200,000 rows of insert statements is in no way the optimal solution to complete this task. An alternative option available is the **mysqlimport** command. The mysqlimport client is simply a command-line interface to the **LOAD DATA INFILE** statement. For readers unfamiliar with this statement, code snippet 6.1 represents an example LOAD DATA INFILE implementation.

```
mysql > LOAD DATA INFILE '/home/callum/emageData/assay.csv'
    -> INTO TABLE assays
    -> FIELDS TERMINATED BY ','
    -> LINES TERMINATED BY '\n'
    -> (emageID, probeID, type);
```

Code snippet 6.1: Example LOAD DATA INFILE statement.

The mysqlimport command can take a number of parameters some of which include, delete (empty the table before import), lock (lock all tables for writing before processing any text files) and force (continue even if an SQL error occurs). While these are useful functions, they are not required in this instance. The mysqlimport statement used to load the EMAGE dataset into MySQL can be found below in code snippet 7.1.

```
---
--Import data into all tables in one command
---
mysqlimport -u root -p --ignore-lines=1 --fields-terminated-by=, emage assays.csv
    publications.csv sources.csv specimens.csv stages.csv textannotations.csv genes.
    csv anatomystructures.csv
```

Code snippet 6.2: Command used to load data into the MySQL database.

The command works by firstly connecting to the MySQL database as the root user and accepting a password. All of the data files I imported included headings, the “ignore-lines=1” parameter simply imports the data starting on line 2 thus skipping the headings row. The ”fields-terminated-by=,” parameter allows one to stipulate the delimiter of the file, whether it be a comma, semicolon or tab for example. The name of the database is then required to be stated in the command, hence the inclusion of “emage”. Finally the name of the files being imported are required. When using the mysqlimport statement, multiple files can be loaded into multiple tables in one command. The name of the table is matched with the name of the file and the data is imported for each. It is therefore crucial that the ordering of the data in the file matches that of the table. If the two do not match, it is likely that **1.** The load will fail due to an incompatible data type with the values found in the file **2.** The wrong data will be mapped into the wrong columns. As each row in a CSV file is a record, there is a clear commonality between the file format and a MySQL database. Thus resulting in a relatively straightforward dataset load.

### 6.1.2 MongoDB - data load

As discussed in section 5.1.2, MongoDB documents can be created by using the *db.collection.insert({})* command. One simply writes a piece of valid JSON within the curly braces of this command and the document is created. This is a sleek and straightforward method however not the most efficient process available for inserting datasets of large volume.

MongoDB provides an alternative to this procedure in the form of a mongoimport tool. The mongoimport tool imports content from a JSON, CSV, or TSV file into the database. When importing a dataset which maps from your flat file into the format of your data model exactly, this method is extremely resourceful. However, should your dataset be in any other format or require structure manipulation, the mongoimport tool would not be of any use as it is a literal import. The data model I created for MongoDB includes embedded data and value arrays. As the EMAGE dataset is not in a JSON file format, using the mongoimport tool was not a feasible approach.

To load the dataset into the data model, I used the Python MongoDB API, namely PyMongo. PyMongo is a Python distribution containing tools for working with MongoDB, and is the recommended way to work with MongoDB from Python [**?**]. The PyMongo script I created can be simply broken down into three stages; connect, insert and update. Code snippet 6.3 represents the full PyMongo script I wrote to load the data into the database.

1. Connect
    - The first step is to connect to the running MongoDB instance. This is an easy step a consists of calling “MongoClient()” which by default connects to the host and port which is running locally.
    - We can then use the running instance to select the relevant database we want to load data into.
2. Insert
    - To map the data into the database we create a **for** loop which iterates through the CSV file.
    - The loop uses the heading of each column as the field and the row as the value.

3. Update

    - To embed the Publications and Text Annotations data within the MongoDB documents I used the “update_ many” command. This looks for a given value, which in this case was the EMAGE ID and updates the document with the stipulated values.
    - The “upsert” parameter is set to false in this instance. Upsert is the equivalent of saying “If the value I am inserting is not currently in the database, what do I do with it?”. As I have set this to false the data will only be added if there is a match on the EMAGE ID.

```
import csv

from pymongo import MongoClient

connection = MongoClient()
db = connection["mongomodel3"]
emage = db["emage"]

with open("Data/Assays.csv") as file1:
    reader1 = csv.DictReader(file1, delimiter=",")
    for row in reader1:
      emage.insert({
        '_id': int(row['emage_id']), 'probeID': row['probe_id'], 'type': row['
            assay_type'], 'source': row['name'], 'specimen' : {'type':row['type'],'
            strain' : row['strain']}, 'stage' : {'theilerstage' : int(row['
            theilerstage']),'dpc' : row['dpc']}})

with open("Data/Publications.csv") as file2:
   reader2 = csv.DictReader(file2, delimiter=",")
   for row in reader2:
      emage.update_many({'_id': int(row['emage_id'])},
        {'$push' : {'publication' : {'publicationID' : int(row['accession']), '
            title' : row['title'], 'author' : row['author']}}}, upsert=False)

with open("Data/TextAnnotations.csv") as file3:
   reader3 = csv.DictReader(file3, delimiter=",")
   for row in reader3:
      emage.update_many({'_id': int(row['emage_id'])},
        {'$push' : { 'textannotation' : {'strength' : row['strength'], '
            anatomystructure' : {'structureID' : int(row['EMAPA']),'term' : row['
            term']}, 'gene' : {'geneID' : row['accession'],'name' : row['name'
            ]}}}}, upsert=False)
```

Code snippet 6.3: PyMongo script implemented to load data into MongoDB.

### 6.1.3 Neo4j - data load

As discussed in section 4.2.3, inserting a handful of nodes directly into a Neo4j database is relatively straightforward. All that is required are a few commands and you can have a fully functioning data model. For a detailed description on how to do this see section 4.2.3. However, this is on a small scale only. The process for implementing a full data model with a large dataset requires more resource.

The query language which Neo4j is based on, Cypher Query Language, allows for multiple ways of implementing a data model. The main way to do this is to use Cypher's **LOAD CSV** command to transform the contents of a CSV file into a graph structure. Code snippet 6.4 represents the Cypher file created to load the Assay nodes into the Neo4j database. The full Cypher file can be found in appendix **??**.

The main structure of the query and the commands used are similar for the two approaches. However, to load data in via a CSV one requires two additional lines, these are represented in lines 5 and 6 of code snippet 6.4. Line 5, "**USING PERIODIC COMMIT**" is used when loading large amounts of data in a single cypher query. This is because loading large volumes of data within a single query runs the risk of failing due to running out of memory. Thus including this function prevents the query failing for this reason. However, it will also break transactional isolation and should only be used where needed 2.7.3. Line 6 of code snippet 6.4 simply loads the file found at the stipulated location and assigns it to a variable name, "row" in this case.

One additional noteworthy aspect of code snippet 6.4 is the use of the "MERGE" keyword. MERGE either matches existing nodes and binds them, or it creates new data and binds that. MERGE is like a combination of MATCH and CREATE that additionally allows you to specify what happens if the data was matched or created [**?**]. Using this keyword aids the normalisation process.

```
1  CREATE CONSTRAINT ON (e:Assay) ASSERT e.id IS UNIQUE;
2  CREATE INDEX ON :Assay(emageID);
3
4  // Create Assays
5  USING PERIODIC COMMIT
6  LOAD CSV WITH HEADERS FROM "file:/home/callum/Documents/Uni/F20PA/Project/Neo4j/
       Data/Assays.csv" AS row
7
8  // Query the already created nodes and match them based on the following clauses.
9  MATCH (source:Source {sourceID : TOINT(row.source_id)})
10 MATCH (specimen:Specimen {specimenID : TOINT(row.specimen_id)})
11 MATCH (stage:Stage {stageID : TOINT(row.stage_id)})
12
13 // Create Assay nodes.
14 MERGE (assay:Assay {emageID: TOINT(row.emage_id)})
15 SET assay.probeID = row.probe_id, assay.type = row.type
16
17 // Create Assay relationships.
18 CREATE (assay)-[:COMES_FROM]->(source)
19 CREATE (assay)-[:CLASSIFIED_AS]->(specimen)
20 CREATE (assay)-[:GROUPED_BY]->(stage);
```

Code snippet 6.4: Cypher file created to load assay data into the Neo4j data model.

### 6.1.4 Apache Cassandra - data load

There are a number of ways in which one can load a preformed dataset into a Cassandra cluster. In the early days of Cassandra, a low level interface, **BinaryMemtable** was used to ingest data. However, this tool was deemed rather difficult to use and is now defunct functionality [**?**].

Other tools which are available are **json2sstable** and **sstableloader**. While these alternatives are thought to be extremely efficient and powerful for importing millions of rows of data, they require careful network and configuration considerations [**?**]. Thus, making the process unnecessarily complicated for the programmer. For situations such as this, where either the volume of data does not merit the use of sstableloader, nor learning the use of json2sstable is deemed a valuable use of resource, another alternative is available; **COPY FROM**.

The COPY FROM command is used on the Cassandra cqlsh interface. Cqlsh is a python-based command prompt for executing Cassandra Query Language (CQL) queries [**?**]. The usefulness of this command is that it accepts CSV data. COPY FROM accepts a number of parameters which allows the programmer to specify exactly what format the CSV file is in. While a CSV file type is recognised as a common format, there is no one way of structuring the file. By this I mean, a file can have headers, varying escape characters, different delimiters and multiple character encodings. All of which can be specified using the COPY FROM command parameters.

```
1 ---
2 --- Copy the EMAGE submissions data into the assays table.
3 ---
4 COPY assays (emageID,assayType,dpc,probeid,source,specimenstrain,specimentype,
      theilerstage)
5 FROM '~/Desktop/emage_submissions.csv'
6 WITH DELIMITER = ','
7 AND HEADER = TRUE;
```

Code snippet 6.5: Loading data into Apache Cassandra using the cqlsh interface.

Code snippet 6.5 represents the loading of the Assay data into the assays table using the COPY FROM command. The command requires you to state the name of the table you are loading data into; “assays” in this example. Then specify the column names you will be

importing data into in brackets. The important thing to remember when using the COPY FROM command on a CSV file is that the ordering of the file headings must match the ordering of the columns in the table. Cassandra does not offer any way of mapping file headings to table column names. Therefore if there is a misalignment of data columns the values will be loaded incorrectly, and more often than not the load will fail due to incompatible data types. The FROM keyword denotes the location of the CSV file you will be loading. The WITH clause allows you to impose any rules on the COPY FROM command. In this instance, I have stipulated the delimiter of the file as a comma, and that there are headings in the top row of the file. After you run this command, the data will have been imported into your Cassandra tables.

# Chapter 7

# Evaluation Results

The strategies which were devised to evaluate and assess the functional competencies of each database solution can be found in section 3.1. The main method for evaluating the database systems was, by using the relevant querying language, to assess the strengths and weaknesses of the functionality each solution provides. This chapter focuses on the outcome of executing the described methods and provides a high level overview of the results. While this chapter focuses purely on the results of running the queries, chapters 8 and 9 provide discussion and comparative analytical insight into the solutions as a whole.

## 7.1 Results

The tables, graphs and charts illustrated in this chapter are based on the results of running several competency based queries on each database system. Table 7.1 details each of the devised queries in English, provides a short description of the data expected to be returned and a rating characterisation. Each query has a rating with a range of 1 to 5, and is to be used as a guide to aid the reader into understanding **1.** General complexity of the query. **2.** Fruitfulness of the data returned. **3.** Level of expectancy that the system will be able to accomplish the query (on a scale of 1 expected, to 5 unexpected).

| Query Number | English | Expected return | Rating |
|---|---|---|---|
| 1 | All structures at Theiler Stage X. | Theiler Stage. Structure ID. | 1 |
| 2 | All structures between Theiler Stage X and Y. | Structure ID. Theiler Stage X. Theiler Stage Y. | 1 |
| 3 | Where is Gene X expressed? | Name of the gene. The structure where the gene was found. The EMAGE ID where the gene was found. | 2 |
| 4 | What is expressed in structure X? | Name of the gene(s) found in the structure. The structure ID. The name of the structure. The Theiler Stage(s) of the structure. The EMAGE ID of the structure. | 3 |
| 5 | Which genes are stored in structures X and Y? | Name of the gene(s). ID of the gene(s). ID of structure X. ID of structure Y. | 4 |
| 6 | Which Genes are most commonly co-expressed? | Name of the gene(s). Count of unique structures the gene is expressed in. | 5 |
| 7 | Calculate transitive closure. | The name of each structure and its parent. | 5 |

Table 7.1: Competency questions to query each database system.

## Query 1 - All structures at Theiler Stage X

The following code snippets represent the queries written for competency question 1.

- **MySQL**

```
1 SELECT t1.accession AS StructureID, t1.term AS StructureName, t2.theilerstage
    AS TheilerStage, t2.dpc AS DPC
2 FROM AnatomyStructures AS t1
3 INNER JOIN Stages AS t2
4 ON t1.stage_id = t2.id
5 WHERE t2.theilerstage = 4
6 ORDER BY 1
```

| StructureID | StructureName | TheilerStage | DPC |
|---|---|---|---|
| 16041 | inner cell mass | 4 | 3.5 dpc |
| 16048 | polar trophectoderm | 4 | 3.5 dpc |

Table 7.2: Output of running query 1