Chapter 8

Database Modelling

In order for me to design a complete database model for each of the technologies, an initial investigation into the specific dataset values was required discussed in section 8.1. Once this was complete I followed the same database design process for each indexing solution which is discussed in section 8.2.

8.1 Cleansing the data

The EMAGE data which was supplied was made up of 4 tab separated files. Each file contained information pertaining a certain aspect of the dataset; Annotations, Publications, Submissions and Results.

- **Annotations** - Data such as the EMAPA structure ID, the EMAPA structure term, the Theiler Stage, the EMAGE structure ID and the strength in which the each gene was detected.
- **Publications** - Data regarding all authored publications for the EMAGE dataset. Data included the title, author(s), Theiler Stage and EMAGE ID for the genes.
- **Submissions** - Information on each EMAGE assay. Data such as EMAGE ID, Theiler Stage, probe ID, type of assay (in situ or part of), specimen type and specimen strain.

On initial review each of the files contained similar, repetitive, meta-data values. Thus creating a level of noise which would not be add anything to the project in terms of analysis and evaluation. Therefore I undertook an initial cleansing of the data before implementing the database design.

This cleansing process consisted of loading the data into Open Refine (OR) and manually manipulating the data using the filtering and editing tools the package provides. Using this tool allowed me to identify any erroneous rows of data which would affect the integrity of the dataset. In each file there was at most 5 rows of data which were either blank or inconsistent with the convention of the rest of the file. I decided to remove these rows as there inclusion in the file was unnecessary.

Another irregularity found in the *Publications* data file was the characters used in the title and author fields. There were over 600 rows of data which contained a non-ascii character. For example - ten Berge D, Brouwer A, el Bahi S, GuÃ©net JL, Robert B, Meijlink F. These rows would be rejected when importing into the databases as by default I decided to apply a UTF-8 character encoding to each indexing solution. Despite these values only contributing to around 5% of the file, the issue needed to be addressed. To do this I devised a regular expression which would identify and subsequently remove any of these characters.

Using a software tool such as OR enabled me to manipulate and cleanse the data in such a way that I would be able to load it successfully into the databases. Despite accomplishing the cleansing successfully, from the challenges I faced, I identified some problematic circumstances which, despite not being as prevalent using the EMAGE data, may affect other big data sets.

The maximum number of rows uploaded into OR was around 150,000. The EMAGE dataset is relatively small in comparison to large scale data collation, however the volume of data was a factor in my decision to use a software tool in an ETL workflow as opposed to rolling my own scripting solution. It is important when choosing a methodology or tool to enhance the veracity of a dataset that the volume of data is taken into consideration. OR is a Java application that utilises the Java Virtual Machine (JVM) and therefore it is integral to allocate enough memory to handle processing large files and thus avoid Java heap space errors. The OR developers suggest that a typical best practice is “start with no more than 50% of your available physical memory, since certain OS’s will utilize up to 1 Gig or more of physical RAM.” [**?**]. While using this software solution was sufficient for the data in this project, should the dataset be of a greater scale, a more robust and resilient system would need to be considered.

As discussed in section 1.1.1 a major challenge in data collection and manipulation is ensuring the veracity of your data. A leading contributor to this challenge is human error. It is

a fact of life that humans are error prone and can often make mistakes, therefore where possible the minimal amount of manual handling of a dataset is key. An example of an issue which can arise from this may be as simple as date formatting changing over time. The data may initially be input in a UK standard date format of DD-MM-YYYY by one person and then stored in a US standard data format of MM-DD-YYYY by another person. A simple example, however one which can have serious repercussions on the validity of a dataset. The cleansing of the EMAGE dataset relied on my knowledge of the data and any obvious flaws such as blank values where a value was required. While the data provided was reliable and generally healthy; a richer more granular dataset may require a more rigorous method of validation. One way to do this would be to implement a software script which takes a subset of the data, defines a format, and restructures the remaining data in the dataset accordingly.

8.2 Designing the data models

In order for me to develop multiple and reliable data models which accurately represent the dataset, I created a database diagram for each indexing solution. Each diagram effectively illustrates the relationship between the data entities. The order in which I decided to create the database model designs, was based upon two main reasons; which hinge upon aiding the readers comprehension of this thesis. The first reason was based upon my previous experience of using each of the database systems. My previous experience ranged from a competent level to the complete unknown. Secondly, as MySQL is a well known database management system, and is widely used for a number of applications, it is expected that the reader will have an already functioning knowledge of the system. As a result I decided the first data model I would develop would be for the relational database management system, MySQL. The next system I developed was MongoDB, followed by Neo4j and finally Apache Cassandra. Each implementation presented a different challenge all of which will be discussed below.

MySQL

As discussed in section 4.3.4 MySQL is a relational database management system which stores and represents structured data through entity tables and relationships. There are a number of variations in which the design of the MySQL database could be modelled for the EMAGE data. Figure 8.1 is an entity-relationship (ER) diagram which illustrates the implementation of my

MySQL normalised database design. Normalisation in database design is a process by which an existing schema is modified to bring its component tables into compliance through a series of progressive normal forms. It aids in better, faster, stronger searches as it entails fewer entities to scan in comparison with the earlier searches based on mixed entities. Data integrity is improved through database normalisation as it splits all the data into individual entities yet building strong linkages with the related data. A description of the tables is below the diagram.

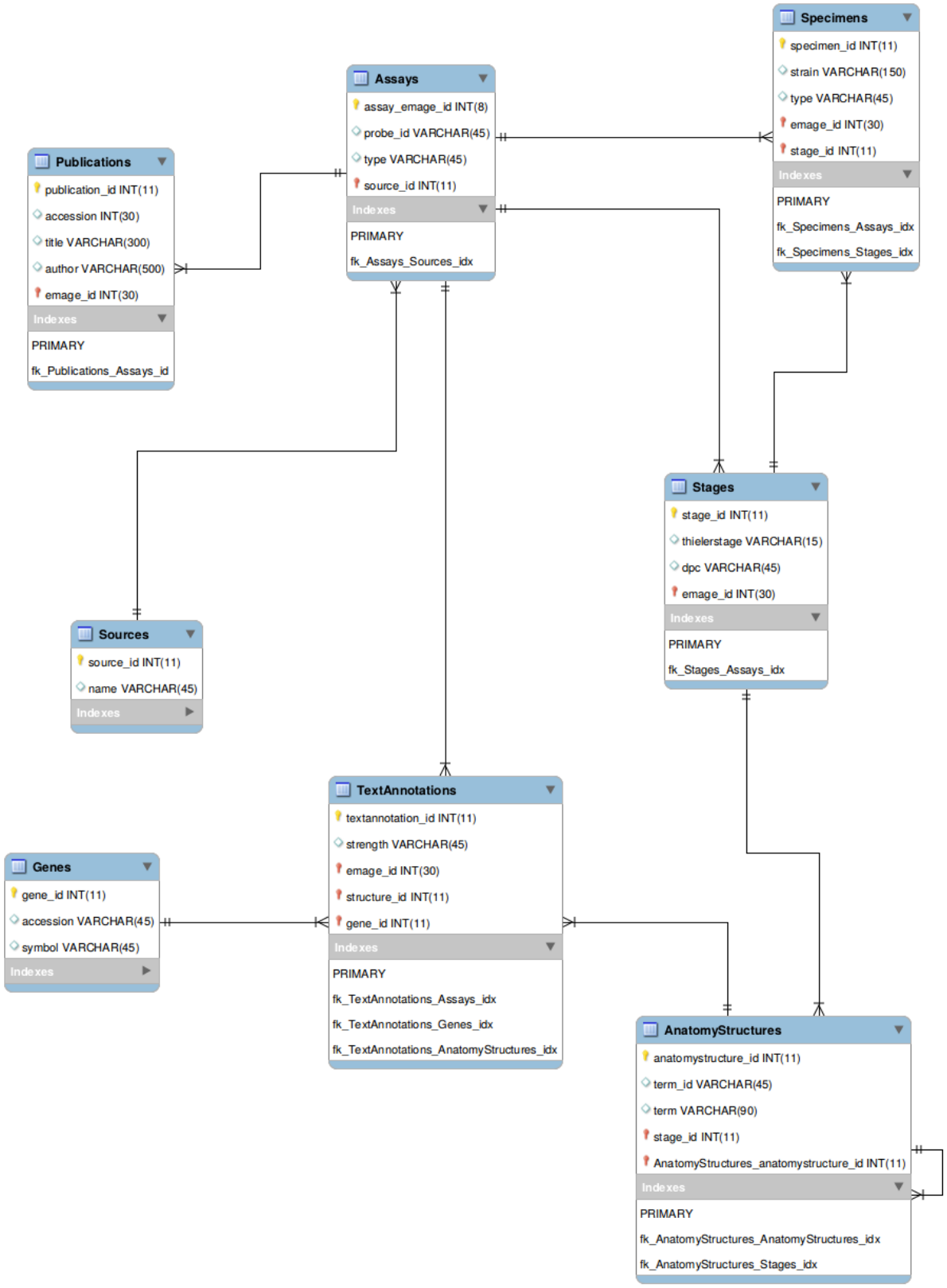

Figure 8.1: MySQL ER table diagram

- **AnatomyStructures** : This table contains the EMAPA ID, and the term which refers to the part of the anatomy.
  - Many to One relationship with the Stages table as one anatomy structure can have the same stage.
  - Many to One relationship with itself on the ID as one structure can have many parts.
- **Assays** : This table contains the EMAGE ID number, the ID of the probe and the type of assay. The *type* field refers to whether the assay is *in situ* or *part of*.
  - Many to One relationship with the Sources table. While one assay can only have one source, many assays can have the same source.
- **Genes** : This table contains the accession number and symbol of each gene.
  - The Genes table does not reference any other table.
- **Publications** : This table contains the accession, title and every author of each assay publication.
  - Many to One relationship with the Assays table as there can be many publications for one assay.
- **Sources** : This table contains the source of the assay.
  - The Sources table does not reference any other table.
- **Specimens** : This table contains the ID, strain and type of each specimen. The *type* field refers to whether the assay is a section, wholemount, sectioned wholemount or unknown.
  - One to One relationship with the Assays table as one assay has one specimen.
  - Many to One relationship with the Stages table as many specimens can have the same Stage.
- **Stages** : This table contains the theiler stage and number of days post conception (dpc) of each assay, specimen and anatomy.

  - Many to One relationship with the Assays table as one assay has can have multiple stages.

- **TextAnnotations** : This table contains the structure and strength of each assay.
  - Many to One relationship with the Assays table as one assay can have multiple text annotations.
  - Many to One relationship with the Genes table as one text annotation can have multiple genes.
  - One to One relationship with the AnatomyStructures table as one text annotation can have one structure.

MongoDB

As discussed in section 4.3.1, MongoDB is a homogeneous, schema-less, NoSQL document store database. There are no formal relations between the data which makes modelling the database a little more challenging; especially with data which is so closely bound as the EMAGE dataset.

Data in MongoDB sits in *collections*, a grouping of documents which are stored on a database. A collection exists within a single database and is the equivalent of an RDBMS table. Documents within a collection can have different fields and typically, all documents in a collection have a similar or related purpose.

The MongoDB implementation was the second prototype data model I created for this project. I followed the same structural process which I had undertaken for the previous MySQL data model. When creating a MongoDB data model there are a number of factors and considerations which need to be identified before starting the formal implementation. Firstly the biggest decision I deliberated over was how should the data be connected. As there were a few options I decided to explore all of them to fully comprehend the pros/cons of each.

My initial design was based around using multiple collections to store the various aspects of the data. The design followed the MySQL model, with were 4 collections; Assays, Text Annotations, Anatomy Structures and Genes. To connect the data and bind the values required an additional manually developed ID field for every document. While this was not a complex task, it was one which I felt was unnecessary and added extra unwanted noise to the data. Using this option would have also incurred more overhead when writing the queries for

the database as one would firstly have to connect the data (similar to a RDBMS join) and then include a further query.

After some deliberation I concluded that the best way to implement the data model would be to have all of the data in the one collection. Figure 8.2 illustrates an example document in the developed MongoDB data model. The diagram should be read as follows:

- **id** : This is the EMAGE id of each assay and is the value which binds all of the data together. The id value has been manually configured to correspond with the EMAGE value.
- **specimen** : The specimen value is an array of size 2 which holds data regarding the strain and type of the assay.
- **probe id** : This value is the id of the probe accession.
- **assay source** : The source in which the assay has been retrieved from.
- **assay type** : The type of assay which is being analysed. By type I am referring to whether the assay is *in situ* or otherwise.
- **stage** : An array containing the information regarding the stage of the assay; Theiler stage and DPC.
- **publication** : An array containing all publication information regarding that specific assay; id, title, author.
- **text annotation** : The text annotation array is the grouping of strength, anatomy structure and gene of an assay. An anatomy structure has a term id and the name of the term and a gene has the symbol and id.

```
{
  "_id":"6",
  "specimen":
   {
     "specimen_strain":"Swiss Webster",
     "specimen_type":"wholemount"
  },
  "probe_id":"MGI:1334951",
  "assay_source":"emage",
  "assay_type":"in situ",
  "stage":
   {
     "theiler_stage":"11",
     "User_Stage":"7.5 dpc"
  },
  "publication":
   {
     "publication_id" : "PMID:1409588",
     "publication_author" : "Tanaka A, Miyamoto K, Minamino N, Takeda M, Sato
B, Matsuo H, Matsumoto K",
     "publication_title" : "Cloning and characterization of an androgen-induced
growth factor essential for the androgen-dependent growth of mouse mammary
carcinoma cells."
  },
  "textannotation":
   [
     {
       "anatomystructure":
       {
         "term_id":"16107",
         "term":"allantois"
       },
       "strength":"strong",
       "gene":
       {
         "symbol":"Fgf8",
         "gene_id":"MGI:99604"
       }
     }
  ]
}
```


Figure 8.2: Example MongoDB document diagram

Neo4j

Neo4j is a graph orientated, NoSQL database solution. It uses the Property Graph Model methodology of connecting data by nodes and weighted edges. Nodes are the equivalent of a MySQL table and edges are the equivalent of a relation. A full description of Neo4j can be found in section 4.3.2.

The data model I have constructed for Neo4j, is semantically similar to that of the MySQL implementation. There are 8 nodes, which contain relatively the same data as that of each MySQL table. The main difference between the two systems is how the data is joined. Where MySQL has a foreign key join, Neo4j has an edge, which connects the nodes.

As with MongoDB and indeed many NoSQL system, there is no formal way to diagrammatically convey the database "schema". To illustrate the Neo4j data model, I have created two diagrams. The first is an entity relationship (ER) diagram - figure 8.3 - and the second - figure 8.4 - is an example visualisation, aiming to convey the look and feel of the graph model. In terms of the ER diagram, it should be translated as, an entity is a node and a join is a relationship. The data within the nodes is:

- **Assay**: This node contains data such as EMAGE ID, probe ID and type e.g. in situ. The Assay node has 4 self-defining relationships.
  - Assay COMES FROM Source
  - Assay HAS Specimen
  - Assay HAS Stage
- **Publication**: This node contains all publication data, which includes; title, author and id of each assay publication. The Publication node has one relationship.
  - Publication DESCRIBES Assay
- **Source**: The source node has just one field, source name. It defines the source of each assay.
- **Specimen**: Each assay has a specimen node. This node stores the specimen strain and specimen type. It has one relationship, which is the link between the stage node.
  - Specimen HAS Stage

- **Stage**: The stage node contains each Theiler Stage and DPC value.
- **Annotation**: This node contains information regarding the strength of each annotation. It is the join between the Gene and Anatomy Structure nodes. The annotation node has two relationships.
  - Annotation REPORTS Assay
  - Annotation HAS AnatomyStructure
- **Gene**: This node contains all data regarding each gene found. Data such as; gene name and gene accession. The Gene node has one relationship.
  - Gene HAS Annotation
- **AnatomyStructure**: This node contains all of the data regarding the respective anatomy structures, such as; structure ID and structure term name

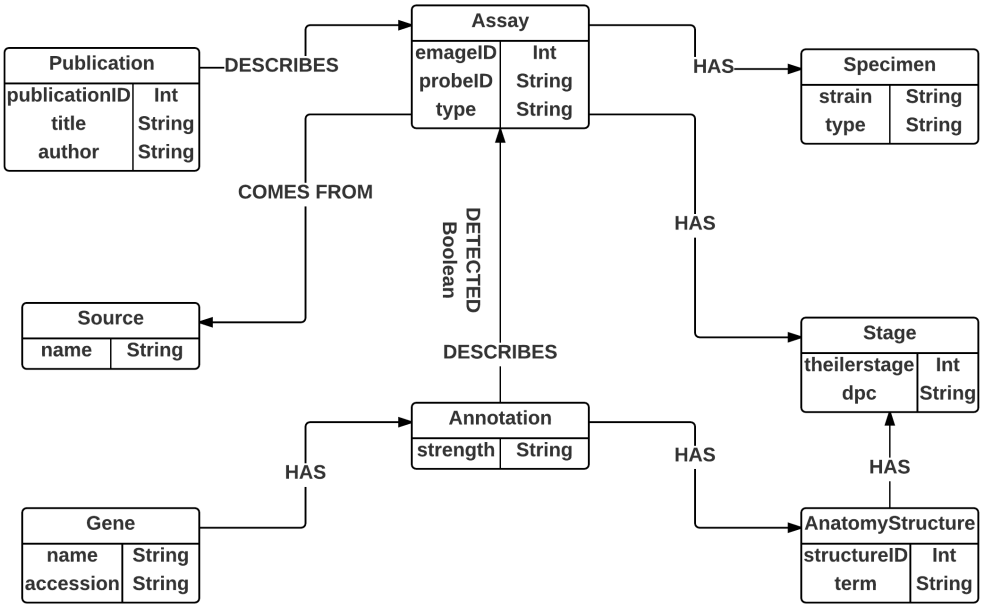

Figure 8.3: Neo4j graph model ER diagram

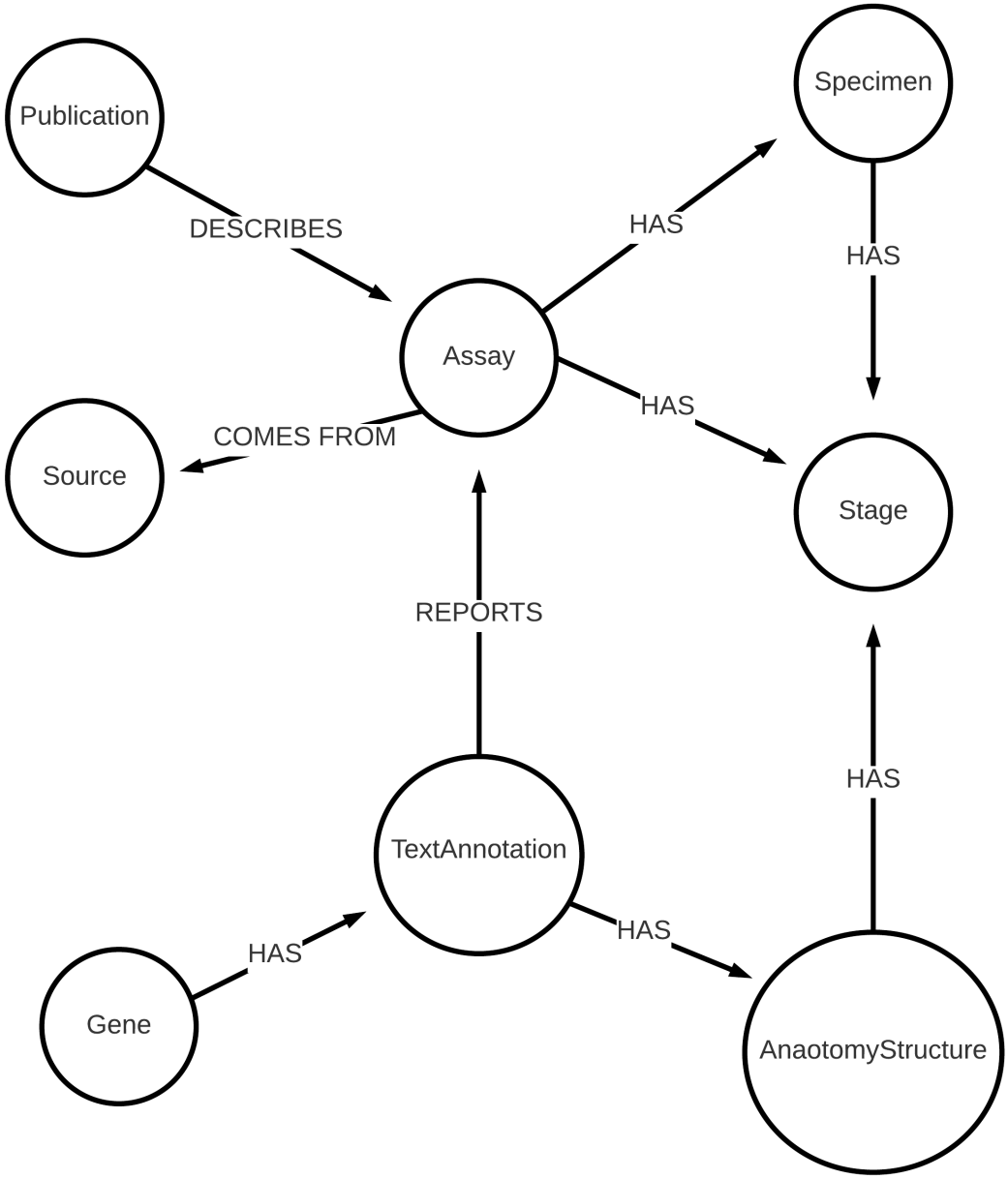

Figure 8.4: Example Neo4j graph model diagram

Apache Cassandra

The Apache Cassandra data model was perhaps the most difficult of all the database system designs to create. This was due to a number of reasons. One of which was, that I had no previous experience of using a Cassandra database, and therefore had to learn a completely new style of working. Secondly, as discussed in section 4.3.3, one of the main aspects of Cassandra family columns and tables is that they do not accept joins. Resulting in tables being created, aiming to satisfy the any potential queries which may be imposed on the database. The repercussions of this concept is discussed in section **??**.

Apache Cassandra is a column based data management system. A key characteristic of a column based data store is that they are extremely powerful and can be reliably used to keep important data of very large sizes. Despite not being *flexible* in terms of what constitutes as data, they are recognised as being highly functional and performant [11]. With this is mind, normalisation of a dataset is in fact not the optimal way of developing a Cassandra data model. It is proposed that *denormalisation* and data duplication within the data model returns the best performing output. This is as a result of Cassandra being optimised to perform a high frequency of writes which in turn reduces the more costly reads. This is a trade-off which must be taken into consideration when developing the data model [11].

Chapter 9

Database Implementation

This chapter focuses on the implementation of the data models discussed in chapter 8. 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. Section **??** evaluates the process undertaken to physically load the data into the systems.

9.1 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.

MySQL

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 8.2. The creation of these tables was a relatively straightforward undertaking. While the ease in which I found this process, may be due to my previous experience of using MySQL, the intuitive and logical way in which relational databases are constructed, make the implementation of 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 9.1 below. This code illustrates the creation of the Assays table, the indexes and the constraints.

```
1  --
2  -- Table structure for table `Assays`
3  --
4  CREATE TABLE IF NOT EXISTS `Assays` (
5    `emage_id` int(11) NOT NULL,
6    `type` varchar(255) DEFAULT NULL,
7    `probe_id` varchar(255) DEFAULT NULL,
8    `source_id` int(11) NOT NULL,
9    `specimen_id` int(11) NOT NULL,
10   `stage_id` int(11) NOT NULL,
11 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
12 --
13 -- Indexes for table `Assays`
14 --
15 ALTER TABLE `Assays`
16   ADD PRIMARY KEY (`emage_id`),
17   ADD KEY `source_id` (`source_id`),
18   ADD KEY `specimen_id` (`specimen_id`),
19   ADD KEY `stage_id` (`stage_id`);
20 --
21 -- Constraints for table `Assays`
22 --
23 ALTER TABLE `Assays`
24   ADD CONSTRAINT `fk_Assays_Sources` FOREIGN KEY (`source_id`)
25   REFERENCES `Sources` (`source_id`),
26   ADD CONSTRAINT `fk_Assays_Specimens` FOREIGN KEY (`specimen_id`)
27   REFERENCES `Specimens` (`id`),
28   ADD CONSTRAINT `fk_Assays_Stages` FOREIGN KEY (`stage_id`)
29   REFERENCES `Stages` (`id`);
```

Code snippet 9.1: Creation of Assays table

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.

This 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.

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 9.1, 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 to create multiple tables and join 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.

MongoDB

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 inserted for each document, if available.

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.

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 **??**. Code snippet 9.2 below, is an example of how a document can be created and data inserted from the command line.

```
db.emage.insert(
    [
      {
      _id: 15423,
      probeID: "T31182",
      source: "eurexpress",
      type: "in situ",
     [
         {
              strain : "C57BL/6",
              type : "section"
         }
     ],
     [
         {
              dpc : "14.5 dpc",
              theilerstage : 23
         }
      ]
      }
)
```

Code snippet 9.2: Inserting data into a mongo document

As illustrated in code snippet 9.2, 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. 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.

In comparison to a MySQL implementation; where tables are constructed and data inserted after, MongoDB allows the programmer to see exactly what and how the data will be populated in the database.

Neo4j

TO BE COMPLETE

Cassandra

TO BE COMPLETE