Dissertation Project

A Comparison of NoSQL and Indexing Solutions for Big Data

Author: Callum George William Guthrie

Heriot-Watt University
Edinburgh, Scotland

A dissertation submitted in partial fulfilment of the requirements for the degree of Bachelor of Science.

25 April, 2016

Project Supervisor: Dr. Albert Burger *Second Reader:* Talal Shaikh

Declaration

I, *Callum George William Guthrie* confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed:

Date:

Acknowledgments

I would like to take this opportunity to express my appreciation to the people who have given their time and support throughout this final years dissertation project. A special thanks to:

- **Dr Albert Burger** - For giving me an opportunity to work on this project, providing guidance and support throughout the year.
- **Kenneth McLeod** - Kenneths's feedback, discussions and support were invaluable to the outcome of this dissertation.
- **Talal Shaikh** - For his constructive feedback on the first deliverable.

Abstract

The era of Big Data is upon us bringing with it a range of new challenges, and encouraging the formulation of new approaches for cleaning, processing and using these enormous amounts of data. These new methods have led to the development of a range of technologies designed to meet the needs of Big Data.

Over the the course of this final year project, I have conducted an investigation into a subset of new technologies which deliver high performance querying of large datasets. I have developed prototype data models using leading NoSQL solutions - MongoDB, Neo4j and Apache Cassandra - and an industry standard relational database management system - MySQL. The project compares and contrasts the functionally, performance and analytical capabilities of the different solutions.

Contents

Introduction **1**

1.1 Objectives . . . 2

1.2 Project Motivation . . . 2

1.3 Definitions . . . 2

2 Background and Literature Review **4**

2.1 Big Data Defined . . . 4

2.1.1 5vs Model . . . 5

2.2 Extract Transform Load . . . 7

2.2.1 Extract . . . 7

2.2.2 Transform . . . 8

2.2.3 Load . . . 8

2.3 NoSQL . . . 8

2.3.1 Distributed Systems and Data Transactions . . . 9

2.3.2 Database Classification . . . 11

2.4 Relational Database . . . 13

2.5 Technology Evaluation . . . 14

2.5.1 MySQL . . . 14

2.5.2 MongoDB . . . 15

2.5.3 Neo4j . . . 15

2.5.4 Apache Cassandra . . . 16

2.6 Data Source and Representation . . . 17

2.6.1 EMAP . . . 17

2.6.2 EMAP anatomy . . . 18

2.6.3 EMAPA dataset . . . 19

2.6.4 EMAGE dataset 20

3 Evaluation Strategy **21**

3.1 Requirements 21

4 Database Modelling **24**

4.1 Cleansing the data 24

4.2 Designing the data models 26

4.2.1 MySQL 27

4.2.2 MongoDB 30

4.2.3 Neo4j 33

4.2.4 Apache Cassandra 35

4.3 Discussion 37

5 Schema Implementation **41**

5.1 Creating database systems 41

5.1.1 MySQL 41

5.1.2 MongoDB 43

5.1.3 Neo4j 47

5.1.4 Apache Cassandra 49

5.2 Discussion 51

6 Importing Data **53**

6.1 Put the data in databases 53

6.1.1 MySQL 53

6.1.2 MongoDB 55

6.1.3 Neo4j 58

6.1.4 Apache Cassandra 60

6.2 Discussion 62

6.2.1 Experiment 1 63

6.2.2 Experiment 2 66

6.2.3 Experiment evaluation 68

7 Query Overview and Results **70**

7.1 Query Overview 70

7.2 Query Results 72

8 Summary and Conclusion **94**

8.1 Evaluation 94

8.2 Future Work 97

8.3 Critique 97

8.4 Thesis Summary 97

Introduction

The era of Big Data is upon us, bringing with it a range of new challenges “Without big data, you are blind and deaf and in the middle of a freeway”[25]. The significance of these challenges encouraged the formulation of new approaches for cleansing, processing and utilising enormous amounts of data.

Data is being collated and stored every second of every day and the value of doing so has never been greater. Billion dollar companies such as Google and Amazon dominate the market in data collection and pride themselves in knowing everything about everything. Former CEO of Google, Eric Schmidt famously said in 2010 “We know where you are. We know where you’ve been. We can more or less know what you’re thinking” [26]. Thus the power of data collection has led to the development of a range of technologies designed to meet the needs of big data.

The purpose of the following research, by way of investigation, is to deliver an insightful examination of a subset of new technologies which deliver high performance querying of large datasets. The ultimate aim of the project is to gain an understanding of these technologies and achieve a level of mastery which permits a thorough scrutiny of their application to big data.

There are a number of different indexing solutions available. In order to encapsulate a comprehensive examination a focus will be on leading NoSQL solutions, modern search and analytics engines and for comparative reasons a conventional relational database management system. The following technologies which will be used for the project:

- Relational database (Section 2.5.1)
- Document-orientated database (Section 2.5.2)
- Graph database (Section 2.5.3)
- Wide column database (Section 2.5.4)

1.1 Objectives

The three key objectives and main intended outcomes for the project are:

1. Investigate the strengths and weaknesses of the functionality each technology provides.
2. Compare and contrast the analytical capabilities of each technology by way of querying prototype models.
3. Conduct a comparative analysis to investigate the scalability of each technology.

1.2 Project Motivation

My interest in the field of data science stems from university modules I have undertaken as part of my BSc Computer Science degree. Modules such as 'Database Management Systems', 'Data Mining and Machine Learning' and a course I am currently studying 'Big Data'. The material involved in these courses have given me an insight into the field of data science and provided me with the opportunity to get a hands on feel for the manipulation, cleansing and processing of a variety of real life data sets and database systems.

Whilst studying for my degree, I have successfully completed modules which have required a working knowledge of MySQL as a prerequisite therefore my comprehension of MySQL is proficient. One of the main attractions to undertaking this project was to be given the opportunity to learn about a number of next generation database management systems. It was important for me to undertake a project in which I will be able to apply my learning and findings to progress in a career path within the data science field.

1.3 Definitions

The data source being used in this document comes from the biological field and therefore relies on an understanding of basic concepts and terms. The below table of definitions provides an overview into the main terms used throughout the report with the aim of providing the reader of the document with a sufficient level of understanding. The table also includes the definitions of generally obscure terms and phrases which will be discussed throughout this project.

Term	**Definition**
Edinburgh Mouse Atlas Project (EMAP)	The combined research projects of Dr Duncan Davidson and Prof Richard Baldock.
EMAP anatomy	A freely available, structured, stage specific list of 13,000+ terms that describe visible anatomical structures in the developing mouse embryo.
Edinburgh Mouse Atlas Project Abstract (EMAPA)	A refined and algorithmically developed non-stage specific anatomical ontology. representation of the EMAP anatomy.
Edinburgh Mouse Atlas of Gene Expression (EMAGE)	A database of in situ gene expression data in the developing mouse embryo.
Theiler Stage (TS)	Each stages defines the development of a mouse embryo by a set of organism structure criteria.
DNA	Deoxyribonucleic acid. Molecule which carries genetic instructions.
RNA	Ribonucleic acid. An acid which is present in all living cells.
Assay	One or more assay comprises an experiment.
Gene	A hereditary unit consisting of a sequence of DNA [1].
Gene expression	The specific activity of a gene when a segment of DNA is copied into RNA.
Specimen	A sample of something. For example, an animal or a plant or a piece of human tissue.
In situ	Latin for 'in place'. This term refers to the original position of the anatomy when being experimented on.
Probe	Used to detect DNA and RNA on membranes and preparing for in situ experiments.
Not only SQL (NoSQL)	A non-relational database environment which is useful for very large sets of distributed data. Allows rapid, ad-hoc organisation and analysis of extremely high-volume, disparate data types.
Ontology	Refers to the science of describing the kinds of entities in the world and how they are related.

Chapter 2

Background and Literature Review

The purpose of this project is to identify and analyse the functional capabilities and running performance of a number of leading NoSQL solutions. This report will assist the reader in understanding the limitations of each of the solutions and provide a level of comprehension, proficient enough to support reasoning in choosing to use one solution versus another. This literature review enhances the theoretical understanding of the project and influenced many of the decisions made throughout the project as a result.

2.1 Big Data Defined

Big Data is a broad, evolving term bound to a complex and powerful application of analytical insight which over recent years has had a variety of definitions. In simplistic terms Big Data can be described as extremely large datasets that may be studied computationally to reveal patterns, trends, and associations for ongoing discovery and analysis.

The 2011 McKinsey Global Institute (MGI), a multinational management consultancy firm, compiled a report namely "Big data: The next frontier for innovation, competition, and productivity" outlines the potential effects big data will have on a number of industries. The report suggests that with the increasing "exponential" growth of data volume, simply recruiting a "few data-orientated managers" will be a temporary fix rather than a lasting solution. MGI suggest that if companies in a variety of sectors, such as the healthcare and retail industry, were to take advantage of the value which big data brings could see potentially huge returns. "...a retailer using big data to the full could increase its operating margin by more than 60 percent" [21]. The report also states that if "healthcare were to use big data

creatively and effectively to drive efficiency and quality, the sector could create more than $300 billion in value every year" [21]. Thus further cementing the view which accepts that big data plays a pivotal role in everyday modern life.

2.1.1 5vs Model

In 2001, Gartner analyst Doug Laney delivered the *3vs model* which defines the challenges and opportunities which have arisen from the increase in data volume. Laney categorises big data into three dimensions; Volume, Velocity and Variety, with the increase of each encapsulating the challenges currently faced today of big data management. The

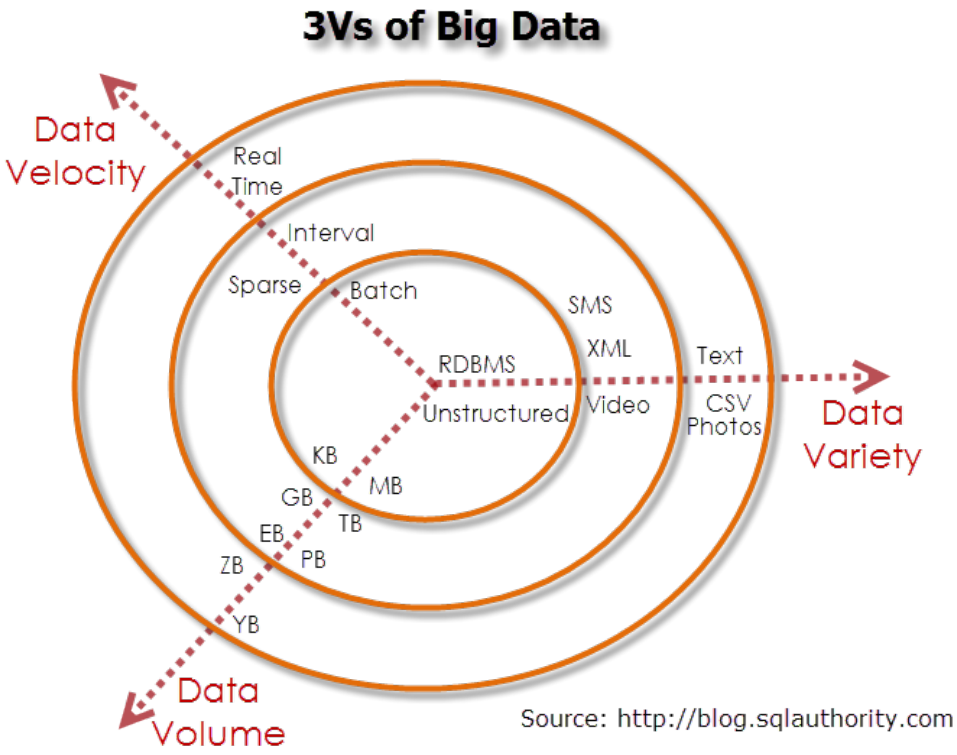

Figure 2.1: 3vs data model

characteristics of each property illustrated in figure 2.1 are defined as: **Volume** - The vast amounts of data generated every second. With the creation and storage of large quantities of data, the scale of this data becomes progressively vast. **Velocity** - The speed at which new data is generated and the speed at which data moves around emphasising the "timeliness" of the big data. In order to fully profit from the commercial value of big data, data collection and data examination must be conducted promptly . **Variety** - This characteristic alludes to the various types of data we can now use; semi-structured and unstructured. Examples being

"audio, video, webpage and text as well as traditional structured data" [22].

Big Data is a term becoming increasingly common in business and society. Overcoming obstacles and implementing effective, actionable Big Data strategies is key for successful big data management. In recent years a fourth category was introduced; **Veracity** - Data inconsistencies and incompleteness result in data uncertainty and unreliability; which creates a new challenge, keeping data organised [22].

The final, and considered by many to be the most important *V* of big data, is **Value**. "All the volumes of fast-moving data of different variety and veracity have to be turned into value" [7]. One of the biggest challenges faced by organisations is having the ability to turn data into something useful. It can be an easy trap to fall into for a business aiming to embark on big data initiatives without a clear understanding of costs and benefits [22]. Thus the importance of establishing clear and achievable business objectives.

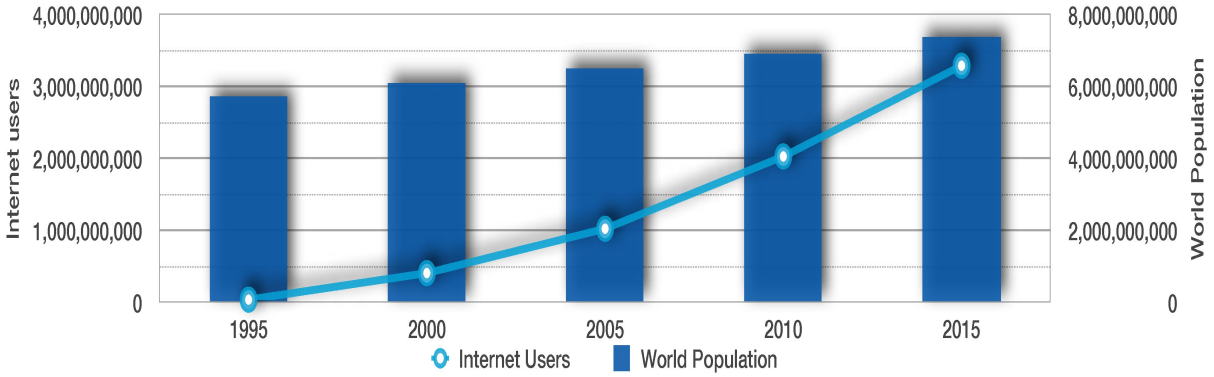

Figure 2.2: World population vs Internet users

The amount of data produced has dramatically increased from when Laney first introduced the 3vs model in 2001. This is in no small part due to the availability and accessibility of the internet. In 1995 the internet had on average 45 million users, 1% of the worlds population. This figure increased to over 1 billion people with internet access worldwide in 2005, and by 2010 nearly 2 billion which was 30% of the worlds population. The latest figures show that in 2015 the penetration of the internet reached 3 billion people, 40% of the entire population. Social media sites such as Facebook, Twitter, Snapchat, Instagram and Pinterest, are some of the main contributors in generating large volumes of user data. Facebook boast a staggering, 1 million links shared, 2 million friend requests and 3 million messages sent on average every twenty minutes [3]. The graph and data table in figure 2.2 illustrate the continual growth of internet accessibility as a whole.

2.2 Extract Transform Load

This project will require the extraction, manipulation and processing of a data source from one data model to another. This process is commonly known as Extract Transform Load (ETL). Section 2.2 discusses each stage involved in the ETL procedure.

A basic definition of the Extract Transform Load (ETL) process is pulling data from one database, refactoring the composition of the data and putting the data into another database. While the name ETL implies there are 3 main categorisation stages - extract, transform, load - the procedure in its entirety is a much broader and expansive process which encompass these stages. Despite this the procedure is split in to these three stages. Figure 2.3 illustrates the ETL process with data coming from a source; a file or database management system for example then being transformed in to the required format for a successful load.

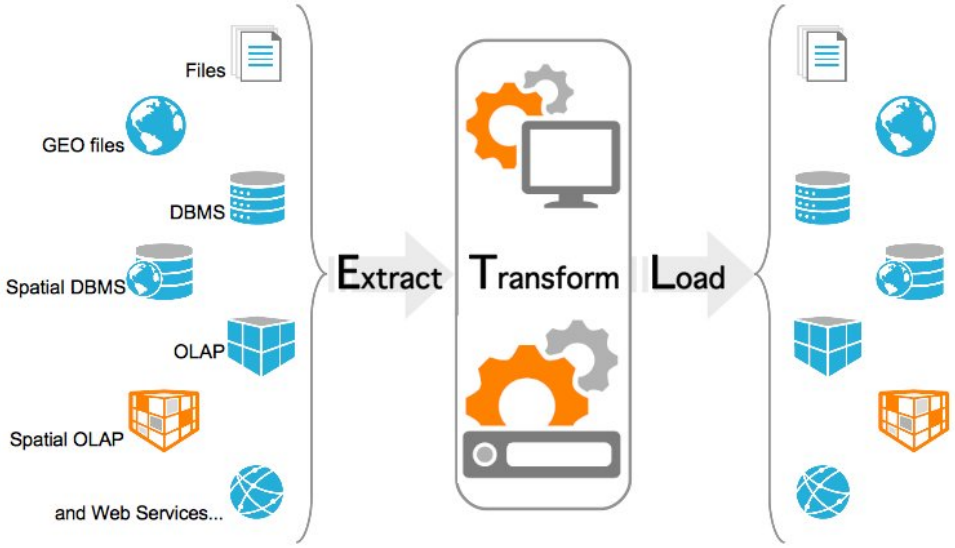

Figure 2.3: ETL process

2.2.1 Extract

Extract is the first step in the ETL procedure in which data is read from a source system, usually but not restricted to a database, and makes it available for processing. The main objective of the extract stage is to retrieve all the required data from a source system using as little resources as possible [9]. It is common for data to be extracted from source systems with different organisations and formats to that of the target system. The extract stage provides an opportunity to cleanse the data from the source system as often there will be redundant or

irrelevant data which is not required.

2.2.2 Transform

Transform is where the extracted data is manipulated from its previous state and converted into a target system format. The step involves the application of a set of rules or functions to transform the data from the source to the target. As well as the applied rules and functions the transformation step is responsible for the validation of records ensuring unacceptable records are removed accordingly. "The most common processes used for transformation are conversion, clearing the duplicates, standardizing, filtering, sorting, translating and looking up or verifying if the data sources are inconsistent" [13].

2.2.3 Load

Load completes the three step procedure and is where data is written into the target system. There are multiple ways in which data is loaded into a system using the ETL methodology. One of which and the most obvious is to physically insert the data. For example if the target repository is a SQL database insert the data as a new row using the relevant *Insert* statement. An alternative to loading the process manually is that some ETL tool implementations have the capability to "...link the extraction, transformation, and loading processes for each record from the source" [13]. Depending on the technique applied the load step of the process can become the most time consuming.

2.3 NoSQL

NoSQL is labeled as a next generation database known to most as "Not only SQL" [24]. This definition however insinuates its defiance against the industry standard SQL. It was originally developed in 1998 by Carlo Strozzi; a member of the Italian Linux society, with the intention of being a non-relational, widely distributable and highly scalable database. Strozzi named the database management system NoSQL to merely state it does not express queries in the traditional SQL format. Sadalage and Fowler believe the definition we commonly refer NoSQL as comes from a 2009 conference in San Fransisco held by Johan Oskarsson, a software developer. Sadalage and Fowler recall Oskarssons desire to generate publicity surrounding the event and in an attempt to do so devised the twitter hashtag "NoSQL

Meetup". The main attendees at the conference debrief session were Cassandra, CouchDB, HBase and MongoDB and so the association stuck [24].

NoSQL solutions are not bound by a definitive schema structure. This permits the ability to freely adapt database records or add custom fields for example without considering structural changes. This is extremely effective when dealing with varying data types and data sets, in comparison to the traditional relational database model which when tackling this issue often resulted in ambiguous field names [24].

2.3.1 Distributed Systems and Data Transactions

Brewer's CAP Theorem

In the late 1990s, computer scientist Eric Brewer formulated what is known-as CAP theorem [27]. CAP theorem suggests that it is impossible for distributed computer systems to provide all of the following three guarantees at the same time:

- **Consistency**: All nodes have a consistent view of the system [27].
- **Availability**: A guarantee that every read/write is acted upon [27].
- **Partition-tolerance**: The system will always operate despite arbitrary partitioning as a result of network failures [27].

The proof of Brewer's theorem states that one can only achieve two out of the three guarantees, and this decision comes at a trade-off. The proof of Brewer's theorem goes as follows:

- For the sake of argument, there are two nodes, one named N1 and the second named N2. They both contain the same dataset (data replication) and are on either side of a partition:
  - To ensure both nodes have the same up-to-date view neither node can allow any changes or updates. Thus lose Availability.
  - If there is a change on node 1 then in this scenario node 2 can not view this change, therefore Consistency is lost.
  - Thus only when the two nodes can communicate can Availability and Consistency be guaranteed. However, this will come at a cost of Partition-tolerance.

Figure 2.4 below is a Venn diagram illustrating the relationship between the three principles.

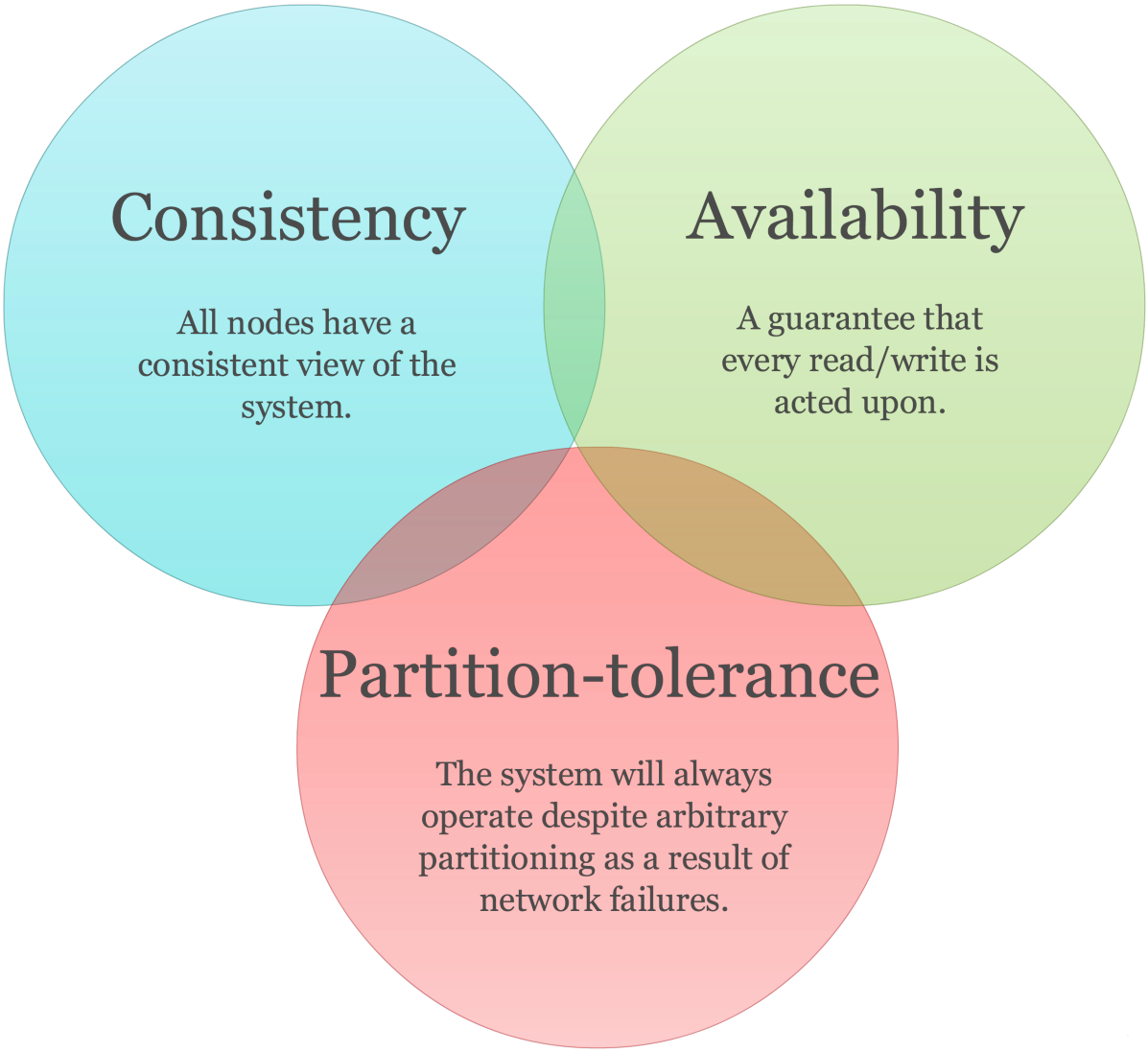

Figure 2.4: Venn diagram of Brewer's CAP Theorem

ACID Transactions

When discussing database management systems, a transaction is a term which refers to a single logical operation performed on data. For example, transferring money from one bank account to another. The term ACID, coined in the 1983 by Andreas Reuter and Theo Härder, describes the properties of a reliable transaction system [19]. The acronym ACID can be defined as follows:

- **Atomicity**: Transactions are "all or nothing". If one part of the transaction fails, then the entire transaction fails cleanly.
- **Consistency**: Data written to the database complies with rules and ensures constraints

are not broken. Any transaction will bring the database from one valid state to another [2].

- **Isolation**: Ensures parallel transactions act as if sequential (one after the other).
- **Durability**: The system will remember a change once a transaction has been committed.

Transactional operations are used for guaranteeing these properties by grouping related actions together. A group of operations can be atomic, deliver consistent data, be isolated from other operations, and be durably recorded [2].

Distributed Database

A distributed database comprises of two or more data files located at different sites and servers on a computer network [29]. The advantage of using a distributed database is that as the database is distributed, multiple users can access a portion of the database at different locations locally and remotely without obstructing one another's work. It is pivotal for the distributed database database management system to periodically synchronise the scattered databases to make sure that they all have consistent data [29]. For example if a user updates or deletes data in one location is is essential this change is mirrored on all databases. This ability to remotely access a database from all across the world lends itself to not only multinational companies for example but also start-up businesses which recruit the expertise of others from various locations.

2.3.2 Database Classification

One of the first decisions to be made when selecting a database is the characteristics of the data one is intending on storing [4]. There are a multitude of options available with many different classifications. The following sections discuss a subset of these which are relevant to the project.

Document-Oriented Database

Document-orientated database (DODB) are designed for storing, retrieving and managing document files such as XML, JSON and BSON. The documents stored in a document-orientated database model are data objects which describe the data in the

document, as well as the data itself. Figure 2.5 illustrates an example document stored in a

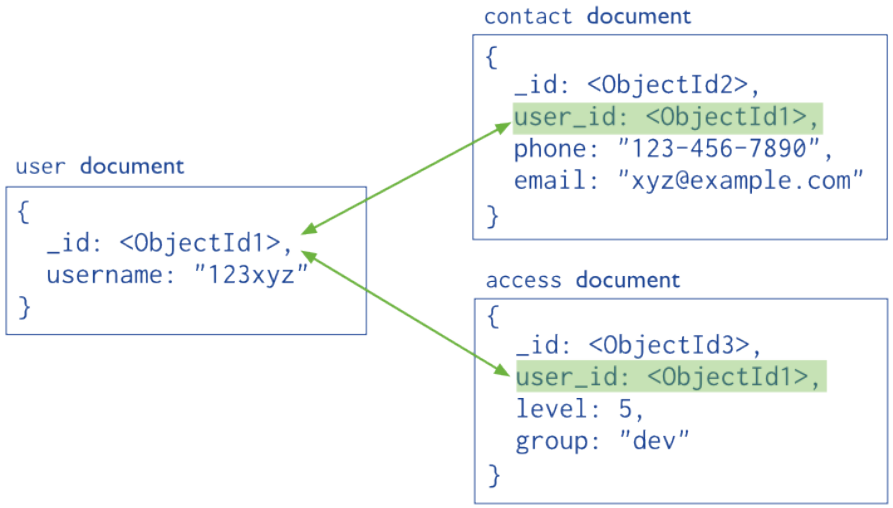

Figure 2.5: MongoDB document

DODB specifically in MongoDB. The data is a recognisable JSON format and the joins of the document are between common variable values within each document.

Graph-Orientated Database

A graph-oriented database (GODB), is a form of NoSQL database solution that uses graph theory to store, map and query relationships. A graph is a collection of nodes connected by relationships. “Graphs represent entities as nodes and the ways in which those entities relate to the world as relationships” [23]. The formation of the graph database structure is extremely useful and eloquent as it permits clear modelling of a vast and often ecliptic array of data types [23]. An example of data represented in a graph structure is the Twitter relationship model. Figure 2.6 illustrates the nodes involved in a standard tweet and the relationship link between them. The labeled nodes indicate the various operations which are involved in one the tweet. One interpretation of the figure 2.6 example is that a user posts a tweet, using the Twitter App which mentions another user and includes a hash-tag and link.

Column-Orientated Database

A column-orientated database (CODB) is a database management system that stores data tables as columns of data rather than as rows of data. The main objective of a CODB is to write and read data from the hard disk efficiently in an attempt to speed up querying time. A CODB

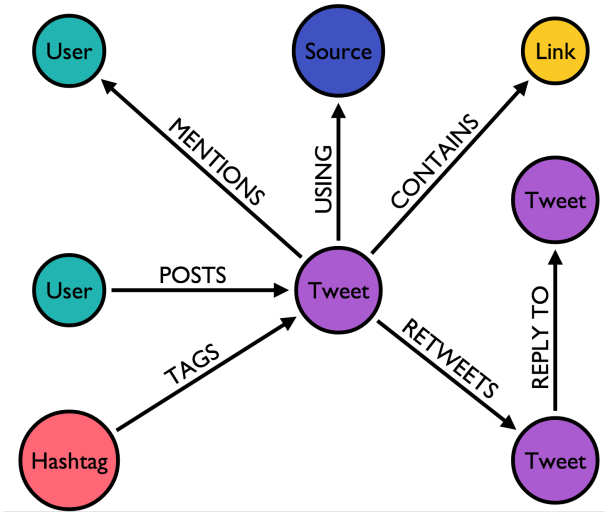

Figure 2.6: Example tweet data relationship

has the ability to self index which uses less disk space than RDBMS which holds the same data. A CODB can also be highly compressed, resulting in aggregate functions such as MIN, MAX and SUM to be performed at a extremely high rate [20].

Figure 2.7 illustrates the comparison of a RDB model against a CODB model. Within the row based model the data contains both multiple values per record and null values. However in the CODB model null values are not required as each record contains a minimum and maximum of one value.

2.4 Relational Database

A relational database (RDB) is a collection of data items organised as a set of tables, records and columns from which data can be accessed or reassembled in many different ways [15]. The connected tables are known as relations and contain one or more columns which comprise of data records called rows. Relations can also be instantiated between the data rows to form functional dependencies.

- One to One: One table record relates to another record in another table.
- One to Many: One table record relates to many records in another table.
- Many to One: More than one table record relates to another table record.
- Many to Many: More than one table record relates to more than one record in another table.

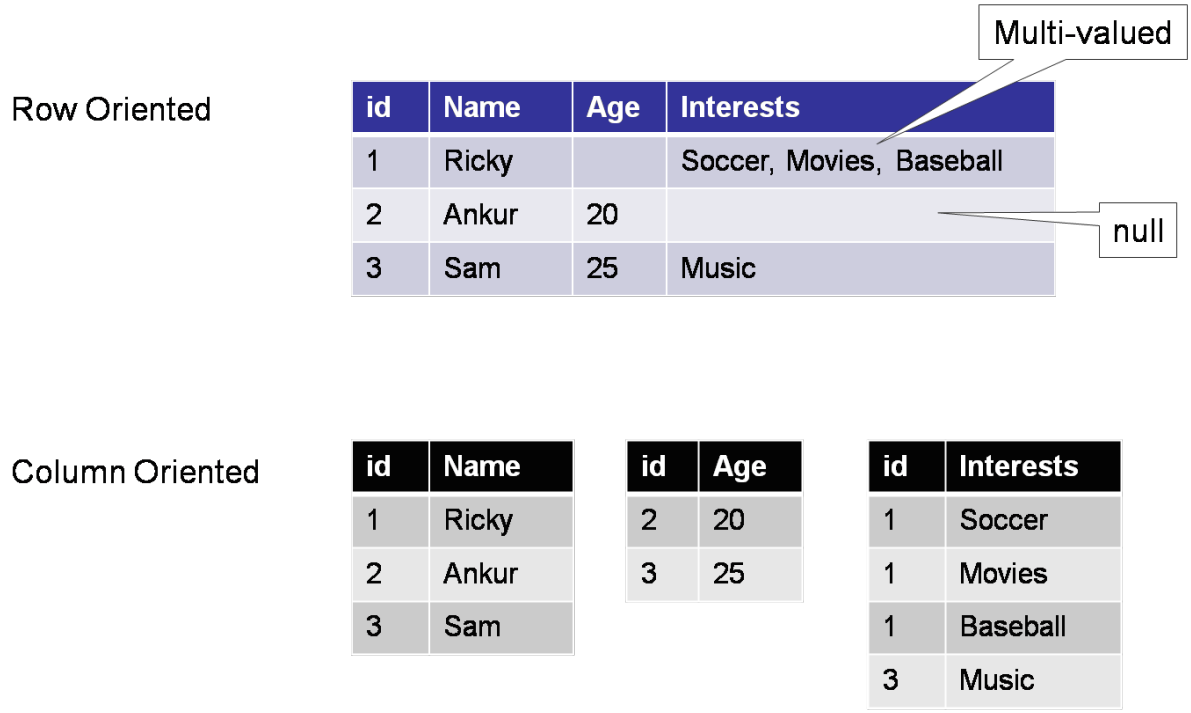

Figure 2.7: Column orientated database example

2.5 Technology Evaluation

2.5.1 MySQL

MySQL is a freely available open source RDB that uses Structured Query Language (SQL). MySQL is commonly used for Web applications with its speed and reliability being a key feature. The MySQL database stores data in tables - a collection of logically related, structured data - which consists of columns and rows. MySQL allows multiple users to manage and create numerous databases.

MySQL adheres closely to the ACID model to ensure results are not distorted by uncontrollable events such as software crashes and hardware malfunctions. A MySQL component used to interact with the ACID model is the InnoDB storage engine. To provide **Atomicity**, InnoDB uses transactions, **Consistency** InnoDB uses internal processing to protect data crashes, **Isolation** again involves transactions and finally **Durability** involves MySQL software interacting with ones hardware configuration [10].

MySQL uses a programming language called SQL which is used to communicate with databases through queries. SQL queries are used to perform tasks such as update or retrieve data in a database. The queries are in the form of command line language which include

keyword statements such as select, insert and update.

2.5.2 MongoDB

MongoDB is an open source cross-platform document-orientated database (DODB). The premise for using MongoDB is simplicity, speed and scalability [18]. Its ever growing popularity, specifically amongst programmers, stems from the unrestrictive and flexible DODB data model which gives you the ability to query on all fields and boasts instinctive mapping of objects in modern programming languages [18]. MongoDB stores documents in a file format named BSON; an extension of JSON. MongoDB uses a query language called CRUD (Create, Read, Update and Delete). CRUD uses the field and value parameters of the JSON document to match clauses in a given query. The output of a query is given in a JSON format.

A record in MongoDB is known as a document; a data structure composed of field and value pairs. Each document is stored in a Collection; a grouping of documents. The values of fields can include other documents, arrays and arrays of other documents. This is known as an embedded document. In order to avoid denormalisation and data duplication, one can embed a document within a document. This allows one to capture relationships between values by storing it in logically similar documents. The alternative to this, is to have multiple collections of documents and join them using a common "_id" value. However, this method has an effect on the implementation and querying of the data model; discussed in detail in section 4.2.2.

The key features of using MongoDB are its high performance data persistence, high availability and automatic scaling [18]. As a result in terms of the CAP data model, MongoDB sits firmly in the Consistency and Partition-tolerance (CP) section of the Venn diagram. However, this is tune-able to be Consistency and Availability (CA) by configuring the system to read from the secondary.

2.5.3 Neo4j

Neo4j is an open-source NoSQL GODB which imposes the Property Graph Model throughout its implementation. The The team behind the development of Neo4j describe it as an "An intuitive approach to data problems" [8]. One of the reasons in which Neo4j is favoured predominantly amongst database administrators and developers is its efficiency and high scalability. This is in part due to its compact storage and memory caching for the graphs.

"Neo4j scales up and out, supporting tens of billions of nodes and relationships, and hundreds of thousands of ACID transactions per second" [8]. This is an interesting aspect of Neo4j, as most NoSQL solutions are tailored towards the Consistency and Partition-tolerance systems. However, while Neo4j has been designed with ACID transactions in mind, in terms of the CAP theorem, it would guarantee Consistency and Availability (CA). This is the union which most relational database management systems fall into.

The Neo4j query language Cypher, is based on SQL. It shares much of the same syntax which has similar semantic value. The key features of Neo4j which lends itself to users, developers and database administrators are its ability to establish relationships on creating, the equality of relationships permits the addition of new relationships being created after initial implementation at no performance cost and its use of memory caching for graphs which allows efficient scaling.

2.5.4 Apache Cassandra

Apache Cassandra is an open source column-orientated DDB that is designed for storing and managing vast amounts of data across multiple servers. "Apache Cassandra is a highly scalable, high-performance distributed database designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure" [6].

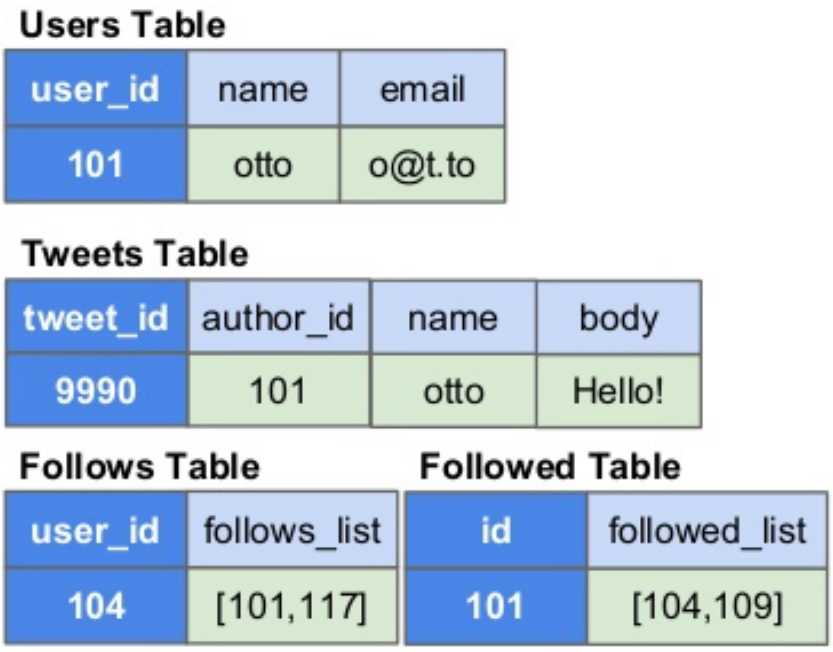

Figure 2.8: Example Cassandra record

Cassandra uses a query langauge called CQL (Cassandra Query Language). As with Neo4j, it is based on SQL semantics with some additional features. Apache Cassandra define the key features of their database management system as "continuous availability, linear scale performance, operational simplicity and easy data distribution across multiple data centres and cloud availability zones" [6]. In terms of the CAP theorem, it is as a result of its strong consistency which guarantees Cassandra Consistency and Partition-tolerance by default. Figure 2.8 illustrates an example record stored in a Cassandra database.

2.6 Data Source and Representation

The datasets used to populate the databases are called EMAP and EMAGE; they are freely available anatomical ontologies of the developmental stages of mouse embryos. The datasets were chosen for this project as my supervisor has much experience in the field and would be able to assist me with any queries I had regarding the data. The size and granularity of the EMAP and EMAGE datasets meet the criteria which will be required to test the database solutions. They will also facilitate the need to explore the limitations of each database comparatively and pose insight into the overall performance of each database.

2.6.1 EMAP

The *Edinburgh Mouse Atlas Project* (EMAP) is an ongoing research project to develop a digital atlas of mouse development. The objective of the EMAP is to implement a digital model of mouse embryos for each time stage in development [1]. The collated model embryo data is then used to form a database from which further research can be conducted and experiments can be mapped.

Each time step in the digital model are named *Theiler Stages* inspired by the research conducted by Karl Theiler. A Theiler Stage defines the development of a mouse embryo by the form and structure of organisms and their specific anatomical structural features. There are 26 individual Theiler Stages which define the growth and evolution of the mouse embryo. The Theiler Stage scheme comprises of both the anatomical developmental stage definition and the estimated length of time since conception. Each Theiler Stage also provides a brief description of the anatomy and any significant changes between the current and previous stage.

Theiler proposed using this scheme as embryos at the same developmental age can have

evolved at different rates and therefore exhibit different structural characteristics EMAP has developed a collection of three dimensional computer models which illustrate and summarise each Theiler Stage [1].

The anatomy at each Theiler Stage has an associated ontological representation. Each, provides an alternative aspect of the evolution of a mouse embryo which corresponds with a respective Theiler Stage. A Theiler Stage is just one method used to define the age of the developing embryo. The term DPC or Days Post Conception, uses the literal number of days since the embryo was created to define its developmental stage. For example, "12.5 dpc" means there has been 12 and a half days since conception. The abbreviated term EMAP carries a certain amount of ambiguity as it is refers to the name of the project, and the name of the original anatomy ontology. For the purpose of this project the main anatomy to be used is the aggregated, non-stage specific, Edinburgh Mouse Atlas Project Abstract (EMAPA) anatomy.

2.6.2 EMAP anatomy

The EMAP ontology was originally developed to deliver a stage-specific anatomical structure for the developing laboratory mouse. As the EMAP research has progressed, the ontology has followed suit, and is continually under development.

The original EMAP ontology consists of a series of relational components organised in a hierarchical tree structure which utilise "part-of" relations and subdivisions which encompass each Theiler Stage [1]. The intention behind the implementation of the original ontology structure was to "...describe the whole embryo as a tree of anatomical structures successively divided into non-overlapping named parts" [1].

Each of the Theiler Stage components has an appropriately named term label, known as the *short name* which describes each respective component. Each Theiler Stage also has a *full name* which comes in the form of the components entire hierarchical path [1]. Neither the *full* nor *short* anatomical name of each component are required to be distinct and can appear in several Theiler Stages. Therefore to avoid ambiguity each component can be addressed by a unique identifier. The unique identifier is in the form of the relevant anatomy followed by a number (EMAP:number). For example "choroid plexus" is the short name of TS20/mouse/body region/head/brain/choroid plexus and has a has unique identifier of EMAP:4218.

The EMAP hierarchical structure facilitates the need for basic data annotation and

integration however, a combination of the lack of hierarchical views, missing or poorly represented Theiler Stages, and label name ambiguity exposed the limitations of the EMAP structure. As a result, the need for a hybrid "abstract" version of the anatomy was identified and subsequently developed; EMAPA [1]. Thus the EMAPA anatomy will be the used as a data source for this work.

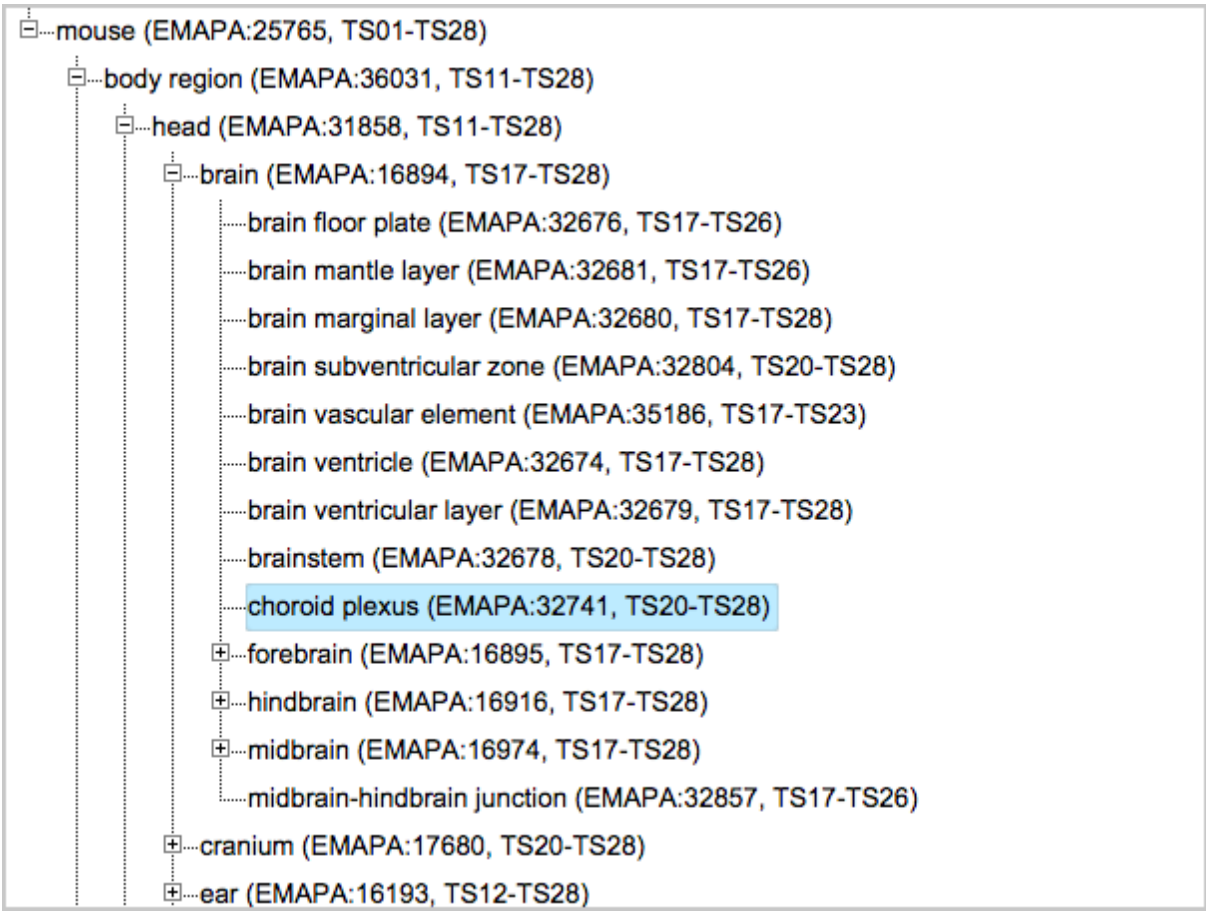

Figure 2.9: EMAPA data structure

The research surrounding the EMAP resource is continually being developed, thus the growth of the project as a whole is progressively increasing with the richness of data at the heart [1].

2.6.3 EMAPA dataset

EMAPA is a refined and algorithmically developed non-stage specific anatomical ontology *abstract* representation of the EMAP anatomy. The EMAPA implementation replaces the EMAP hierarchical tree structure for a *directed acyclic graph* structure; a graph in which it is

impossible to start at some vertex v and follow a sequence of edges that eventually loops back to v again. Thus enabling the ability to represent multiple parental relationships and other forms of "is-a" relations where appropriate [1].

Each anatomical component in the EMAPA anatomy is identified as a single term, coupled with the appropriate start and end Theiler Stage at which the component is considered to be present in the developing embryo [1]. With the aim of enhancing user experience, the EMAPA anatomy implements an alternative naming convention from the EMAP anatomy replacing full path names for components to *"print names"*. Using the above example for comparison, "EMAP:4218" in the EMAP anatomy becomes "TS20 brain choroid plexus" in EMAPA. This naming convention supplements the requirement of uniqueness and is easy to understand.

2.6.4 EMAGE dataset

EMAGE is a database consisting primarily of image data of *in situ* gene expression data of the developing mouse embryo. The data is sourced from in the community and which is then taken by curators who monitor the EMAGE project and implement it in a standardised way that allows data query and exchange. The description includes a text-based component but the unique aspect of EMAGE is its spatial annotation focus [1].

There are a number of terms used in the EMAGE dataset which may be unfamiliar to those who do not have an understanding of the biological field. One basic term is a "gene"; a hereditary unit consisting of a sequence of DNA [1]. "Gene expression" is the specific activity of a gene when a segment of DNA is copied into RNA. Another term is an "assay". A single experiment can comprise of one or assays. For example, if a biologist is researching a specific gene in a finger, this can be split into multiple sections; each being an assay. A "probe" is used to detect DNA and RNA on membranes and is also used when preparing for "in situ" experiments. "In situ" is a Latin term for 'in place'; referring to the original position of the anatomy when being experimented on [1]. For example, if researching a finger, instead of taking it off the hand and crushing it down to find the genes inside, it remains in-tact with the hand.

"Sites of gene/enhancer expression in EMAGE are described by denoting appropriate regions in the EMAP virtual embryos where expression is detected (and not detected) and also describing this information with an accompanying text-based description, which is achieved by referring to appropriate terms in the anatomy ontology" [1].

Chapter 3

Evaluation Strategy

The purpose of this chapter is to discuss the methods used to carry out the primary research for this project. In order to achieve the objectives outlined in section 1.1 a formal set of requirements have been established to meet each intended target. Section 3.1 details how each objective will be evaluated and examples are provided of how each technology will be measured by way of competency questions.

The requirement section below is a high level view of what and how the data solutions will be analysed throughout the project. In order to get a detailed and finalised set of requirements an initial prototype model will be developed which will give an insight in to what will be expected to be achieved from the project. It is likely that the requirements below while they may form the basis of the project objectives will change drastically once the prototype model has been developed.

3.1 Requirements

Objective 1

Evaluate the strengths, weaknesses and limitations of each query language.

The first objective it to investigate the strengths and weaknesses of querying functionality each technology provides. In order to evaluate the functional limitations of each technology, prototype data models will be developed for each solution. The solutions will then be loaded with the EMAPA and EMAGE datasets.

I have created a number of competency based questions. These questions will be translated in to the relevant query language for each of the datasets. The difficulty of the queries range

from a basic level, where it is expected that all of the solutions will return the intended output, to an advanced complexity level. The difficulty of writing the queries will be evaluated to assess the overall performance of the database solution.

Objective 2

Compare and contrast the analytical capabilities of each technology.
This will follow on from the tasks undertaken in the first objective. This objective will also identify and analyse the difficulties which may or may not have arisen in the first objective on the ease of writing the query.

- If the query is possible in a given database how rich a return of the dataset can it provide.
- Does any database offer any querying capabilities/functionality which compared with another which aids the query in any way.
- Extra features which the system offers which are useful for analysis and visualisation.

Objective 3

Assess the scalability of each of the database solutions.
The final key objective to be evaluated in this project is to conduct a comparative analysis of the scalability of each technology. This will be achieved by loading the data sources in to each prototype data model. The specific requirements which will measure the performance of each database are:

- Ease of ETL implementation - The focus of this requirement will be to evaluate any challenges which were faced during the ETL process.
- Did the data model handle the volume of data and were there any issues as a result.

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

Table 3.1: Competency queries for each database system. Rating scale: 1 (simple - complex) 5

Chapter 4

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 4.1. Once this was complete I followed the same database design process for each indexing solution which is discussed in section 4.2. A discussion on the data modelling process as whole, for each database system, can be found in section 4.3

4.1 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.

The EMAPA dataset was consisted of one flat comma-separated file. The file withheld data which represented the hierarchical path of an Anatomy Structure. This consisted of the structure ID of the parent value, the structure term name and also the structure ID of the child value.

- **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, specimen type and specimen strain.

- **Results** - Results from each gene expression. Much of the data in this file was a replicated in the submissions file. Data included Theiler Stage, EMAGE ID, data source, assay type and gene name.

On initial review each of the EMAGE files contained similar, repetitive, meta-data values. Thus creating a level of noise which would not 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. Conversely the EMAPA dataset did not require any data cleansing.

The 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 OR 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 meaning I would be able to load it successfully into the databases. Despite cleansing the datasets successfully, I identified some potential problematic circumstances. While the identified circumstances may not be as significant using the EMAGE dataset, they 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" [14]. 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 2.1.1 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.

4.2 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 reader's 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 already have a 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.

4.2.1 MySQL

As discussed in section 2.5.1 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 4.1 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. Resulting in better, faster, stronger searches. It uses fewer entities to scan in comparison with the earlier searches. Data integrity is improved through database normalisation as it splits all the data into individual entities yet builds strong linkages with the related data. The below description provides an overview into each table and its entities.

- **AnatomyStructures** : This table contains the EMAPA ID, and the term which refers to the part of the anatomy. It has a:
  - 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. It has a:
  - 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. It has a:
  - The Genes table does not reference any other table.
- **Publications** : This table contains the accession, title and every author of each assay publication. It has a:

  - 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. It has a:
  - One to Many relationship with the Assays table as one assay has one specimen however one specimen can feature in many assays.
  - 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. It has a:
  - 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. It has a:
  - Many to One relationship with the Assays table as one assay can have multiple text annotations.
  - One to Many relationship with the Genes table as multiple text annotations can feature the same gene.
  - Many to one relationship with the AnatomyStructures table as many text annotations have the same structure.

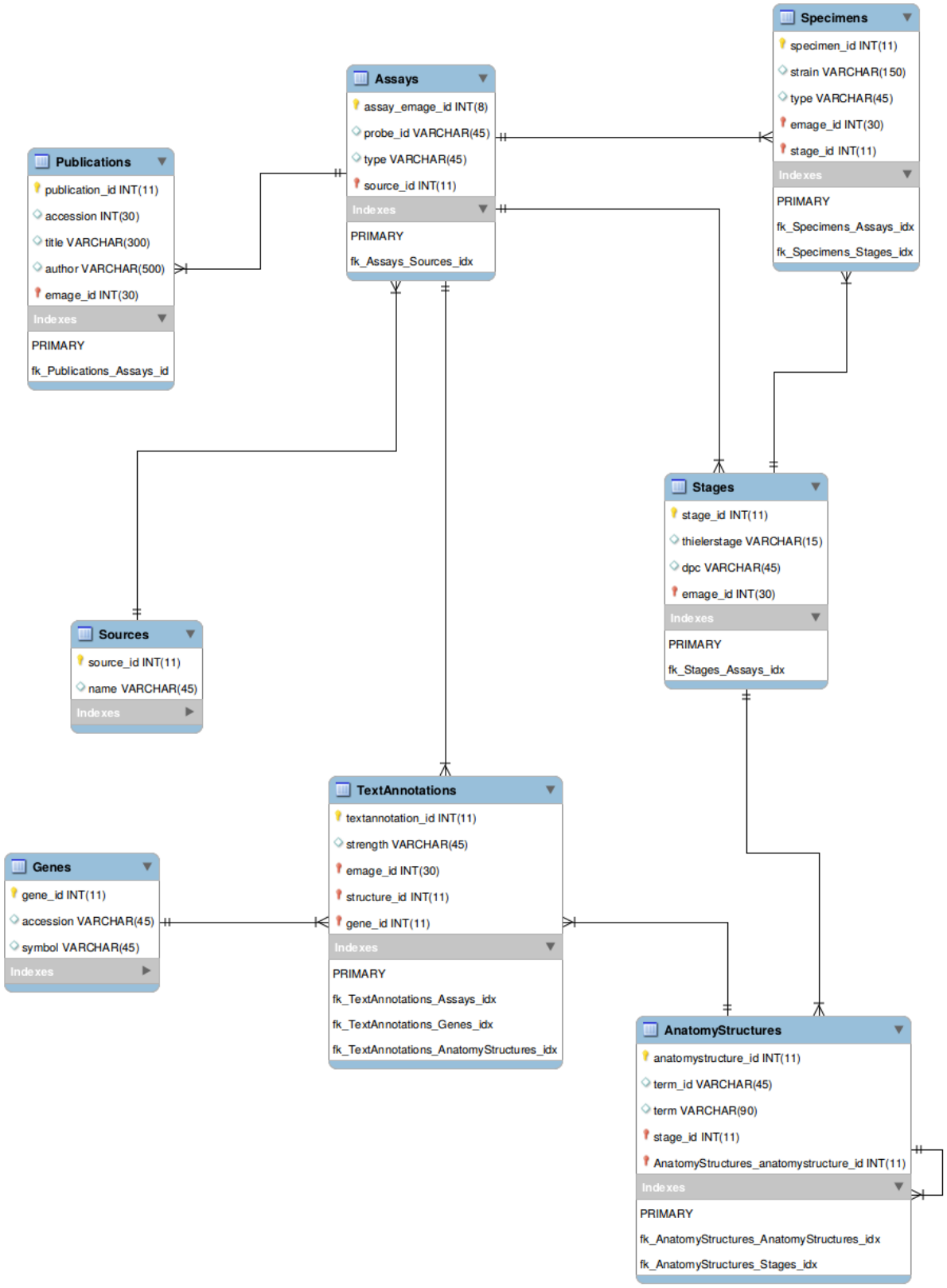

Figure 4.1: MySQL ER table diagram

4.2.2 MongoDB

As discussed in section 2.5.2, MongoDB is a homogeneous, schema-less, NoSQL document store database. There are no formal relations between the data, this makes modelling the database more challenging; especially with data which is as 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 that 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 the data should be connected. As there were a few options I decided to explore all of them to fully comprehend the positive and negative aspects 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 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. One would firstly have to connect the data (similar to a RDBMS join) and then include a further query. This can only be done at the application level as opposed to querying directly with the database. Depending on the query and number of collection joins, this may not be overly time consuming. Nevertheless, additional resource is still required.

After some deliberation, I concluded that the best way to implement the data model was to have all of the data in the one embedded document collection. Code snippet 4.1 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,
    "probeID" : "MGI:1334951",
    "source" : "emage",
    "type" : "in situ",
    "specimen" : {
        "strain" : "Swiss Webster",
        "type" : "wholemount"
    },
    "stage" : {
        "dpc" : "7.5 dpc",
        "theilerstage" : 11
    },
    "publication" : [
        {
            "publicationID" : 1409588,
            "author" : "Tanaka A, Miyamoto K, Minamino N, Takeda M, Sato B, Matsuo H, Matsumoto K",
            "title" : "Cloning and characterization of an androgen-induced growth factor essential for the androgen-dependent growth of mouse mammary carcinoma cells."
        }
    ],
    "textannotation" : [
        {
            "anatomystructure" : {
                "term" : "allantois",
                "structureID" : 16107
            },
            "strength" : "strong",
            "gene" : {
                "name" : "Fgf8",
                "geneID" : "MGI:99604"
            }
        }
    ]
}
```

Code snippet 4.1: Example MongoDB document.

4.2.3 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 row in a MySQL table and edges are the equivalent of a relation. A full description of Neo4j can be found in section 2.5.3.

The data model I have constructed for Neo4j, is semantically similar to that of the MySQL implementation. There are 8 classes of 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 which connects the nodes; Neo4j has an edge.

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 an entity relationship (ER) diagram - figure 4.2 - aiming to convey the look and feel of the graph model. The ER diagram should be translated as: an entity is a node and a join is a relationship. None of the relationships in the data model have any properties. The data within each node is:

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

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

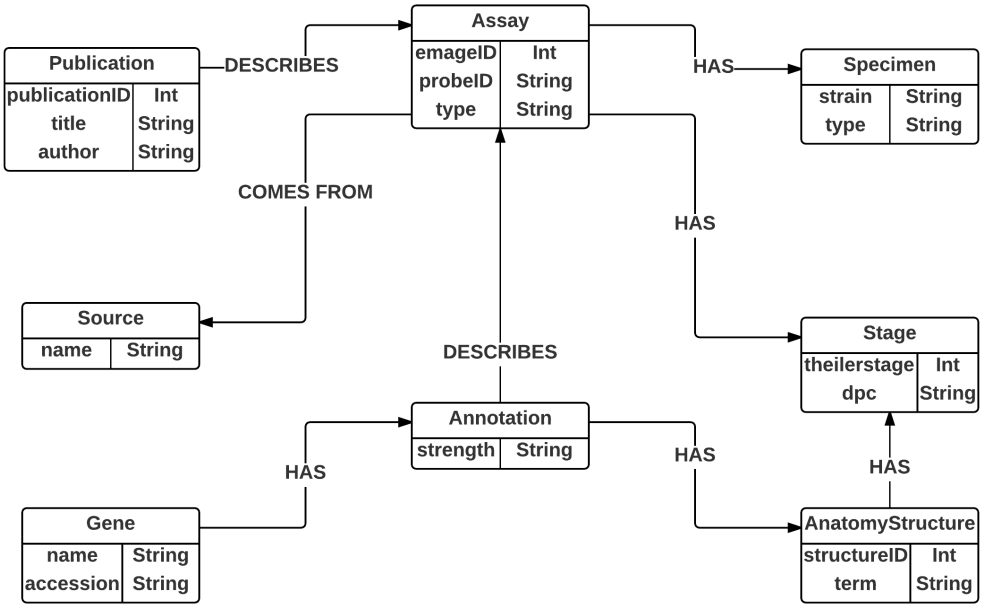

Figure 4.2: Neo4j graph model ER diagram

4.2.4 Apache Cassandra

Apache Cassandra is a column based, data management system. A key characteristic of a column based data store is that it is extremely powerful and can be reliably used to keep important data of very large sizes. It is widely recognised as being highly functional and performant [6]. 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 return the best performing output. This is as a result of Cassandra being optimised to perform a high frequency of writes; this reduces the more costly reads. This is a trade-off which must be taken into consideration when developing the data model [6].

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. Firstly, I had no previous experience of using a Cassandra database, and had to learn a completely new style of working. Secondly, as discussed in section 2.5.4, one of the main aspects of Cassandra family columns and tables is that they do not accept joins. This results in tables being created and this aims to satisfy any potential queries which may be imposed on the database. The repercussions of this concept are discussed in section 4.3.

As with many database systems the optimal way to model a structure is to identify the queries which may be imposed; Apache Cassandra is no different. This can result in multiple tables, containing duplicated data, with either additional or less columns of information. I have modelled the Cassandra data model around the queries outlined in the evaluation strategy of this document (chapter 3).

There are 7 tables: assays, assayByStage, publications, structureByGene3, structureByStage and textAnnotations1. The naming convention I adopted for the Cassandra data model is based on describing each table by its partition key grouping (for a full description of partition keys and Cassandra concepts, see section 2.5.4). For example, AssayByStage is a table consisting of each Assay partitioned on the Stage value. Figure 4.3 is an ER diagram which illustrates the Cassandra tables. The diagram should be read as:

- **assays**: All of the assay data which includes EMAGE ID, probe ID, specimen strain, specimen type, Theiler Stage, DPC, source and assay type. The partitioning key of this table is the EMAGE ID.
- **assayByStage**: Table contains much of the same data as the assays table - EMAGE ID,

Theiler Stage and DPC. The partioning key of this table is the EMAGE ID and the clustering key is the Theiler Stage.

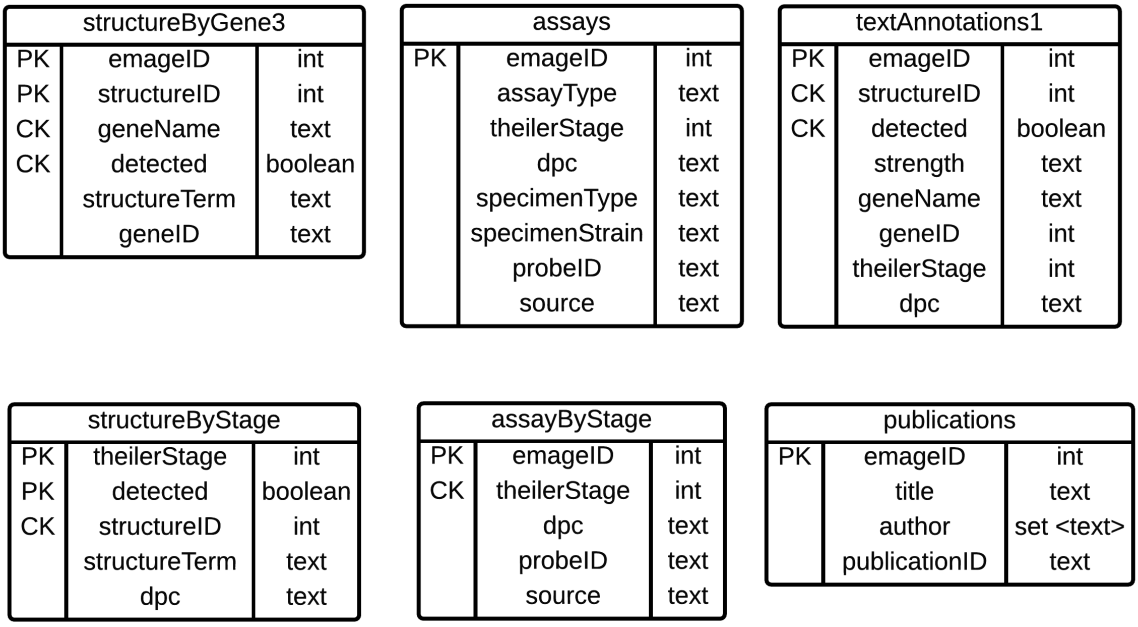

Figure 4.3: Cassandra graph model ER diagram

- **publications**: Table contains all of the data regarding a publication. Such as EMAGE ID, title, author and publication ID. The partioning key is the EMAGE ID.
- **structureByStage**: Table contains data regarding an anatomy structure, grouped by the stage information. The columns include Theiler Satge, detected (boolean), structure ID, structure term and DPC. The partitioning key of the table is composite between Theiler Stage and detected. The clustering key is the structure ID.
- **structureByGene3**: Table contains similar data ast the structureByStage table with the inclusion of gene information as opposed to the stage. The cpartioning key is the EMAGE ID and structure ID. The cluseting keys are the geneName and detected value.
- **textAnnotations1**: Table contains all of the text annotations information. Data such as EMAGE ID, structure ID, detected (boolean), strength, gene name, gene ID, Theiler Stage and DPC.

4.3 Discussion

This section provides a general summation of the design process as a whole for each solution. It will provide the reader with an in-depth understanding of the challenges faced; advantages and the disadvantages of the data modelling procedures.

The first hurdle one must overcome is fully comprehending the data to be modelled. Getting to know the dataset, will allow one to create the optimum system architecture. Having a clear and coherent understanding of how the data is joined, and how relationships are formed in a dataset, is imperative to a successful outcome. Once an application developer knows what they are developing they can accomplish how they can do it. The design process of a data model can sometimes be overlooked. The two main reasons for this are 1. it is often a challenging process and can be difficult to complete correctly; and 2. it can take a long time. Designing the best data model solution for an application can often make or break a project. It is important that the appropriate amount of time is devoted to this process to ensure that the application sits upon a high performing, durable database system.

The first obstacle I faced was that I had no prior experience in the biological field and many of the concepts which accompany the EMAPA and EMAGE datasets took me some time to understand. It was difficult to logically model a database around an unknown entity. After discussing the datasets in detail with my supervisors, and conducting background research, I was able to surpass this barrier. This increased my progress in the data modelling phase of the project and I was able to start physically modelling the systems.

In terms of the technical prowess required, depending on field experience, one would be expected to create a data model for each of these systems with relative ease. With the exception of Cassandra, normalisation and data de-duplication is encouraged for each of the solutions. The objective of database normalisation is to isolate data so that additions, deletions, and modifications of an entity can be made in just one table. In the case of a relational database such as MySQL, this would take the form of splitting one larger table into several smaller tables. An example of this is the Assays table. The Assays table consists of an EMAGE ID, a probe ID and an assay type. The physical data however is made up using the Submissions file (description in section 4.1). To recap, the submissions file consists of EMAGE ID, Theiler Stage, Probe ID, the type of assay, the source of the assay, the specimen type and the specimen strain. For the Assays table in the MySQL data model the Theiler

Stage, specimen data and the source of the assay have all been split into 3 separate tables. I also split the assay publications, text annotation genes and text annotation structures into separate tables. The tables are joined with either a one-to-one or many-to-one relationship. The MySQL data structure is relatively basic. This was a conscious decision made at the time of design. It can be very easy to complicate and escalate the difficulty of a data model. While normalising a dataset often means creating more tables, and thus more relationships, a balance of comprehension vs system performance needs to be found. Implementing the structure in this way makes the information clearer and easier to find for users. It also reduces the need for restructuring the schema as new types of data are introduced.

A key attribute of MySQL, and most relational database systems, is its organised, structured rigidity. This requires developers to strictly structure the data which is stored in their applications. Schema alterations are often one of the driving arguments to use a schema-less, NoSQL system over a relational database. This was not an issue which I encountered using the EMAGE dataset as it is not dynamic and did not change at all. However, as discussed in section 2.6 the EMAP anatomy ontology is ever evolving. There have been many iterations and enhancements made from when the project first began until now. In 2013 a paper published in the Journal of Biomedical Semantics "EMAP/EMAPA ontology of mouse developmental anatomy: 2013 update" stated a number of potential future directions of the project.

Particularly, the "develops-from" relationships will be included to support the analysis of differentiation pathways in databases that deal with expression, phenotypic, and disease-related information. Another goal is the inclusion of a set of textual definitions, computable logical definitions that can be used by automated reasoners, and other forms of metadata" [16].

Looking at the proposed example "other forms of metadata", in a MySQL data model, this may require an additional table with a join, an index and an ID. The ID would then need to be included in the corresponding table, with which the data has a relationship. Even for just two or three extra pieces of metadata the amount of work involved is overly demanding. Comparatively, in a MongoDB or Neo4j data model, should further information become available which is to be included in the database, only requires a simple update or insert statement. No restructuring, joining or manipulation of the data model is necessary.

Creating a MongoDB data model is vastly different to any of the other solutions in this

project. As with Neo4j, MongoDB imposes a flexible schema, meaning the collections do not enforce a specific structure. This allows one to insert data at will without having to conform to a strict schema. It also permits the matching of documents to objects and entities, this makes designing the data model easy. The document structure mirrors that of the EMAGE flat files. Therefore mapping the CSV headings to the MongoDB fields was straightforward. As discussed in section 4.2.2 one consideration, specific to the document structure, was whether to use embedded documents or multiple collections. The decision to use embedded documents relied heavily on what I was looking to achieve from the project. If I created multiple collections I would have had to write cross collection queries, which requires code to be written at the application level. Each of the systems discussed in the project have an API available, mainly Python and Java. While using an API is perfectly achievable, and may be the basis for further research, I focused on what is possible using the command-line interface.

The most interesting data model for me to create was Neo4j. In terms of the physical structure I based this mainly on the already created MySQL system. Each table in the MySQL model was mapped as a Neo4j node. The primary key and foreign key relationships in the MySQL system were converted to a relationship in Neo4j. Like relational data models, Neo4j is at its most performant when the nodes are normalised. The clear link between the two systems allowed me to quickly develop a schema for Neo4j. The main aspect of Neo4j which I found to be intuitive and the most useful was the graph model itself. Being able to visualise the nodes as data, the joins and relationships, makes this clear and extremely easy to follow. There are no complicated joins or key relations to comprehend.

The way in which the Cassandra data model was constructed seemed to me to be rather backward. Data duplication and de-normalisation is the opposite of what I had been used to using relational databases. Basing the design of the structure on how one is going to query the database was a concept, which at first, was difficult for me to comprehend. When creating a MySQL system for example, querying the database is often one of the the last considered aspects when data modelling. An exception to this would be data modelling in high performance production systems. Flipping the entire process on its head, while refreshing and enjoyable to learn, meant the design stage of the modelling processes were prolonged. This resulted in the overall design of the database being a trial and error exercise, increasing the time taken to model the system. The design of the tables and the way in which the data is stored could easily be mistaken for a relational MySQL model. The tables are structured

similarly and contain many of the same properties. Overall the modelling stage of the Cassandra data model was a steep learning curve. However, I often find the best way to learn is to start using the software and get used to the concepts. Often failure can be the best way to learn. This was certainly the case with the first stage of the Cassandra data model.

Chapter 5

Schema Implementation

This chapter focuses on the implementation of the data models discussed in chapter 4. Each database management system has its own procedure for initiating 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.

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, this brought limitations and restrictions which I had to understand to fully achieve my target model.

5.1.1 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 4.2.1. The creation of these tables was a relatively straightforward undertaking. The intuitive and logical way in which relational databases are constructed also makes 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.

```
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 5.1: Creation of Assays table in MySQL.

As shown in lines 4 - 11 of code snippet 5.1, the creation of each column takes the form of column name then data type with length and the value i.e. null or not null. The character set for each table by default, was set to utf8. It is noteworthy that the key values were not instantiated at time of creation. This was as a result of an experiment aiming evaluate the effect index keys and constraints had at time of data load for each database. The full discussion and outcome of this experiment can be found in section 6.2.

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 are where the key columns are create. Lines 23 - 29 are where the foreign key constraints are expressed. The notion of keys joining tables can often be confusing to understand on first encounter. Primary and foreign keys are, in most cases, confined to integer values. This is as a result of data 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, in some situations there is no unique ID present in the dataset. 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 constraints, 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

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 consist of 12-bytes, where the first four bytes are a timestamp that reflect the ObjectId’s creation [18]. A unique ID can also be manually created for each document, if desired. For example, if a dataset has a unique identifier which will be used as a reference, this can be implemented by 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 a schema-free, 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. The structure of a MongoDB collection is generated when data documents are loaded into the system. Each document in the database can be different fields with different values. By default, any value without a type specified will be presumed to be either a string 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. It is unlikely that one would manually insert a large volume of data using the previously discussed *db.collection.insert({})* method. However, 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 for creatung 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 of this is discussed in section 4.2.2. The key difference; between the two approaches, are that within a multiple collection database, one is required to write multiple queries and also undertake client side application level processing. This is for the same outcome as writing a single line query using an embedded structure. All embedded data is available in a single document and can be accessed by a single query. The decision to choose between the two modelling designs should be based purely on a case by case basis. Separate collections are recommended if you need to select individual documents and would like more control over querying [18]. Conversley embedded documents are recommended when you want the entire document all at once [18].

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. A good way of thinking of the embedded data concept is as document in a document. For example, in hierarchical terms, a document can have many fields, some with child fields as opposed to absolute values. Figure 5.1 illustrates an example of a document which has 3 fields and 3 values, one of which is an embedded document. The field named “publication” has 2 embedded (child) values which provide more information in the same collection about the same document.

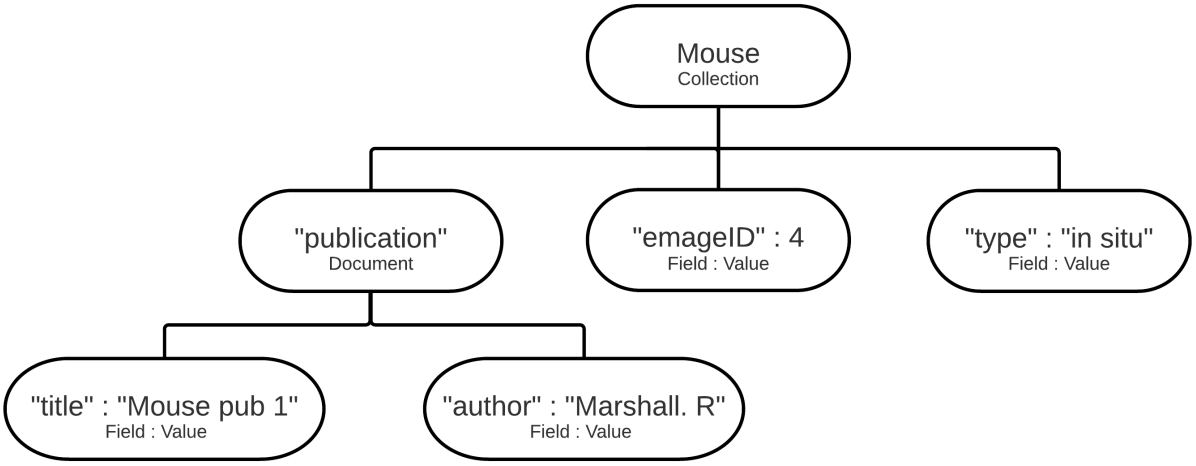

Figure 5.1: Example of a MongoDB embedded document in a hierarchical structure.

Lines 6-34 of code snippet 5.2 represent an example of data embedded in a document. Looking specifically at lines 14-20 there is a value named “publication” which has embedded values of “publicationID”, “author” and “title”. This allows 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

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 [28]. CQL was designed to be as user friendly as possible for both programmers and operations professionals alike [28]. 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 2.5.3.

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 matched value. As with nodes, relationships can have definitive value properties.

A node created in Neo4j is automatically provided with an ID value. Each and every node in the database has a unique numerical ID. This is incremented from the first node to the last inserted. This value can not be changed. The ID value can be used as a standalone statement or within a query clause by using the “ID” clause function.

Like MongoDB, Neo4j is schema-free. Creating a Neo4j structure can 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, I created a Cypher file and imported the dataset simultaneously. The implementation of this is discussed in section 6.1.3, which also describes how to load data into Neo4j from 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 CREATE (assay:Assay {emageID: TOINT('10001'), probeID : 'Epas1 probeA',
13     type : 'in situ'})
14 CREATE (assay)-[:COMES_FROM]->(source)
15 CREATE (assay)-[:CLASSIFIED_AS]->(specimen)
16 CREATE (assay)-[:GROUPED_BY]->(stage);
17 //
18 // Create Publications
19 //
20 MATCH (assay:Assay {emageID : TOINT('10001')})
21 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'})
22 CREATE (publication)-[:DESCRIBES]->(assay);
```

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

Creating indexes and constraints in Neo4j is a simple, one line command process 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 6.1.3.

As illustrated in code snippet 5.3, there are 3 main stages when creating a node in my Neo4j data model. Looking at the Publications node on lines 22-26. Firstly matches another node, "Assay", with a value which we corresponds with the 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 it creates 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 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. Nodes and releationshios can be created effortlessly.

5.1.4 Apache Cassandra

Physically creating a data model in Cassandra is similar to that of a MySQL system. They share many of the same keywords such as CREATE, WITH, SET and FROM. They also use similar key functionalities such as PRIMARY KEY. While semantically these concepts vary, the two systems are comparable syntactically.

The Cassandra data model consists of 7 tables: Assays, AssayByStage, AssayBySpecimen, Publications, StructureByGene, and TextAnnotations. Creating these tables was a simple and quick process. Despite my prior knowledge of Apache Cassandra being at a beginner level the query language CQL is instinctive to learn. This may be as a result of its similarities with SQL. Code snippet 5.4 illustrates the code to create each table in Cassandra. As you can see for each of the tables, the process works by giving the CREATE TABLE command, stating the table name and then stating the table columns. There are a number of different data types in

Cassandra, many of which are standard such as int, double, float and boolean, with the text data type equivalent to a UTF-8 encoded string.

```
1  ---
2  CREATE TABLE assays( emageID int, assayType text, theilerStage int, dpc text,
       specimenType text, specimenStrain text, probeID text, source text, PRIMARY KEY (
       emageID));
3  ---
4  CREATE TABLE assayByStage( emageID int, theilerStage int, dpc text, probeID text,
       source text, PRIMARY KEY (theilerStage, emageID));
5  ---
6  CREATE TABLE assayBySpecimen( emageID int, specimenType text, specimenStrain text,
       probeID text, source text, PRIMARY KEY (specimenType, emageID));
7  ---
8  CREATE TABLE publications( publicationID text, title text, author set <text>,
       emageID int, PRIMARY KEY(emageID));
9  ---
10 CREATE TABLE structureByGene( structureID int, structureTerm text, detected boolean
       , geneName text, geneID text, emageID int, PRIMARY KEY ((geneName, detected,
       structureID), emageID));
11 ---
12 CREATE TABLE structureByStage( structureID int, structureTerm text, detected
       boolean, theilerStage int, dpc text, PRIMARY KEY ((theilerStage, detected),
       structureID));
13 ---
14 CREATE TABLE textAnnotations( structureID int, structureTerm text, strength text,
       detected boolean, geneName text, geneID text, dpc text, theilerStage int,
       emageID int, PRIMARY KEY (emageid,structureID,detected);
```

Code snippet 5.4: Example of how to create tables in Cassandra.

Many of the keywords and concepts of Cassandra found in code snippet 5.4 may be familiar to the reader with a background in SQL. With the exception of the **set** data type in line 8 and the semantics of the PRIMARY KEY, everything else is for the most part the same. The set data type is a collection of one or more elements. This was imposed on the Author column of the publications table. As many of the publications contain multiple authors, using the set data type to contain each was a logical choice. The other main difference is the PRIMARY KEY operator. The primary key is split into two, a partition key and a clustering key. These values are supplied by stating PRIMARY KEY (partition key, clustering key) - lines 4 and 6 of code

snippet 5.4. If only one value is stipulated, then it is regarded as both the partition key and clustering key - line 2 of code snippet 5.4. There can be multiple partition keys, and/or multiple clustering keys - lines 8, 10 and 12 of code snippet 5.4. Each of these statements can be run on the Cassandra command-prompt and the full data model is created within seconds.

5.2 Discussion

One of the major positives of MySQL is the plethora of add ons and third party tools available for modelling and general database management. Open source software such as MySQL workbench and phpMyAdmin, provide the ability to create and maintain a MySQL database. The former provides data modelling, SQL development, and comprehensive administration tools for server configuration and user administration [11]. MySQL workbench also allows one to reverse engineer an ER diagram. This tool eliminates the process of manually writing the code to create tables, instantiate entities and establish table joins. All one has to do is design an ER diagram of their data model and the code is automatically generated. However, if it would be preferable to manually write the script to create the data model, while potentially tedious it is not an overly time consuming process. This is the method I adopted for creating the MySQL data model.

The decision to choose this procedure was based on two reasons. Firstly, I wanted to fully understand and appreciate the data modelling complexities of each database system. Secondly, using a third party tool for implementation would not result in a comprehensive evaluation of each database system. To provide an insightful perspective of the data modelling process, I implemented each system using their respective command-line interfaces. Implementing the MySQL schema was a quick procedure. I wrote the commands for each of the tables out in a text editor, verified for errors, then copied and pasted into the MySQL command-line interface. My previous experience certainly helped with this stage as I was able to complete the full MySQL implementation in around twenty minutes to half an hour.

Similar to the MySQL implementation, the creation of the Cassandra data structure was straightforward. As discussed in section 4.3 the most challenging part of the Cassandra modelling process was understanding which tables required creating. The procedure to develop a Cassandra data model depends on what one is looking to query and pull out of the database. Therefore the design stage effectively had already provided me with the statements

to create the Cassandra model. The relevant commands are run on Cassandra cqlsh tool interface and the schema is implemented.

The implementation stage for the MongoDB and Neo4j data models were completed dynamically at the time of load. Therefore for comparison between the models there is not much detail to convey. These two models do not consist of a formal structure and as a result require no schema implementation. This is certainly one of the most attractive qualities of the two systems. Their flexible nature allows one to manipulate and develop a full data model with little restriction. This provides a sense of control and system awareness to the developer.

Chapter 6

Importing Data

As discussed in section 2.2, the load stage of an ETL procedure can 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 and EMAPA 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

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 6.2.

```
---
--Import data into all tables in one command
---
mysqlimport -u root -p --ignore-lines=1 --fields-optionally-enclosed-by='"' --
    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,be it a comma, semicolon or tab for example. To signify the delimiter which encloses the values we use the parameter "–fields-optionally-enclosed-by="""'. 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, and **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

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 though 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 [18]. 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.

```
1  import csv
2
3  from pymongo import MongoClient
4
5  connection = MongoClient()
6  db = connection["mongomodel3"]
7  emage = db["emage"]
8
9  with open("Data/Assays.csv") as file1:
10     reader1 = csv.DictReader(file1, delimiter=",")
11     for row in reader1:
12       emage.insert({
13           '_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']}})
14
15 with open("Data/Publications.csv") as file2:
16    reader2 = csv.DictReader(file2, delimiter=",")
17    for row in reader2:
18       emage.update_many({'_id': int(row['emage_id'])},
19          {'$push' : {'publication' : {'publicationID' : int(row['accession']), '
                 title' : row['title'], 'author' : row['author']}}}, upsert=False)
20
21 with open("Data/TextAnnotations.csv") as file3:
22    reader3 = csv.DictReader(file3, delimiter=",")
23    for row in reader3:
24       emage.update_many({'_id': int(row['emage_id'])},
25          {'$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

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 a more advanced method.

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 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 [8]. 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 similar to combination of MATCH and CREATE that additionally allows you to specify what happens if the data was matched or created [28]. Semantically it is the same command as the MongoDB "upsert", as discussed in section 6.1.2. 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

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 [17].

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 [17]. This makes 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.

An alternative is to use COPY FROM; a Cassandra command used in the cqlsh interface. Cqlsh is a python-based command prompt for executing Cassandra Query Language (CQL) queries [5]. The usefulness of this command is that it accepts CSV data. COPY FROM accepts a number of parameters which allow 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. A file can have headers, varying escape characters, different delimiters and multiple character encodings. All of these 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 one to state the name of the table one is loading data into; “assays” in this example. Next 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 are headings in the top row. After you run this command, the data will have been imported into the Cassandra tables.

6.2 Discussion

Each system provides the relevant functionality to insert data into the databases on both a large and small scale. Using the query language of each system, inserting data manually is done by using a variation of the SQL INSERT INTO statement. This functionality is seen as one of the minimum requirements for a database management system. While each of the solutions have their own twist on how the physical load is done, they are all relatively similar.

One of the key aspects of a NoSQL system is the flexibility it offers. The main reason for this is because NoSQL systems are schema-less. The benefit of this attribute is nevermore apparent than when loading data into a data model. Creating the optimum data model for a system can often be a trial and error exercise. Thus being able to modify and manipulate your system architecture easily is a big advantage. Where many NoSQL systems differ from relational databases systems such as MySQL is that the data model is created at the time of load. Such is the case for MongoDB and Neo4j. While I had created diagrams to visualise the structure of these systems, the physical schema had not been implemented. Consequently, some changes to the final schema were made after the data had been loaded into the databases. Because the systems schema is completely dynamic and flexible I was able to add, remove and update fields with ease. For example, if I were to add an additional piece of information for say a specific document (in MongoDB) or node (in Neo4j) I could just use the relevant insert statement and the data is loaded into the database. Comparatively if I were to do this in a MySQL structure, I would have to add an entire column and update appropriately. For just additional field I would have an entire row of null values. This is computationally more expensive and is also more time consuming. If we look at Cassandra, we again have to add an entire column, however we do not have null values. Therefore we can add additional data freely, at very little resource cost. This is certainly a big advantage of using a NoSQL system.

One of the main objectives of this project was to analyse the performance of the database solutions. The term performance covers a wide range of components and can be measured in many ways. For this stage of the project the *load* performance of the systems was something which I was interested in analysing. It is rarely the case where an entire database of information will be inserted at any one time. However, if one is looking to migrate from one database management system to another or a database is being restored from a backup for

example, the time it takes to physically load the data is something which should be taken into consideration. Therefore I created an experiment to evaluate the total time it takes to load the data into the respective solutions.

The purpose of the experiment was simple, to identify the system which loads the EMAGE dataset into the data model in the quickest time. More precisely, I was looking to identify the effect imposing indexes and unique/distinct constraints had on the time taken to load data. For each of the solutions, the test was considered complete once each of the insert statements were finished and the databases were fully populated. Once a query was run on each of the systems, the execution time was printed on the command-line. This time was used to measure the outcome of the test. To enhance the fairness of the experiment, each query was run 5 times; with the average time across the tests recorded. After each run, the contents of each of the databases was removed, resulting in empty systems.

Hardware/Software	**Specification**
Computer Model	Dell Inspiron 1545
Operating System	Ubuntu 15.10
Processor	2.3 GHz Pentium(R) Dual-Core
Physical Memory (RAM)	6Gb
Storage	512 GB SSD
Graphics	Intel Integrated GMA 4500 MHD

Table 6.1: Machine configuration

Table 6.1 above, outlines the system specification which the data models were created in. Each of the database solutions were run locally on the same machine. The idea of this table is to aid the reader when discussing the performance of the solutions and allow one to understand the underlying hardware and software behind the systems.

6.2.1 Experiment 1

The first step in creating the experiment was to implement the data models so I could load in data. As the MongoDB and Neo4j schemas are dynamically created, this step was only possible for the MySQL and Cassandra systems. A full discussion on the schema implementation stage can be found in chapter 5. The time taken to create the schemas was not taken into consideration when measuring the execution time. The timings were purely based

on the time taken to run the import statements.

As one of the main aims of this experiment was to identify the effect of implementing indexes pre-load had on import time I only created the tables. Indexes and unique constraints were not applied until after the data was loaded. Primary and foreign keys were applied for the MySQL model. For the Cassandra system, partition and clustering keys were implemented where necessary. To create the test for the MongoDB and Neo4j systems, I initially loaded the data into the databases and did not apply any indexing or unique constraints.

One can use indexing in a database to find data rows with specific column values quickly; much like the indexing of a book. The intention of this is to improve the performance of commonly run queries. Intuitively, one would expect quicker load times and a slower querying performance with no indexes implemented. A number of behaviours should be taken into consideration when deliberating system performance. Factors such as the impact on write operations, and the amount of space each index requires can have a profound affect on the execution of queries. Another consideration is the size of files being loaded. The volume being loaded into the MySQL and MongoDB solutions was around 12mB, and the Neo4j data model was loaded with around 20mB of data. For the Cassandra system, the size of the files being loaded cumulatively came to around 35mB. This was as a result of each of the systems using a normalised model compared to the Cassandra database. If one was to compare the database solutions directly, this is a component which would need to be taken into consideration. However, this experiment was to analyse the affect of implementing indexes and unique constraint as opposed to solely focusing on which system loads the data in quickest time.

Once the tables were created, the next step was to load the data into the systems, using the various import commands. The results of the initial experiment can be found in table 6.2. These values represent the load times for non-indexed database systems. The load times

Database System	Version	Load 1 time (s)	Load 2 time (s)	Load 3 time (s)	Load 4 time (s)	Load 5 time (s)	Average load time (s)
MySQL	5.7.11	5.20	5.14	5.16	5.25	5.33	**5.21**
MongoDB	3.2.4	127.5	186.9	152.4	159.7	133.8	**152.06**
Neo4j	2.3.3	20594.09	20234.41	20911.35	20198.70	20121.86	**20412.08**
Apache Cassandra	3.0.4	189.47	165.11	168.69	182.33	161.85	**173.49**

Table 6.2: Load times for non-indexed database systems

remained consistent throughout each of the tests. This eliminates any concerns regarding

simultaneous processes running on the computer, effecting the performance of the experiment. If we look at the systems individually, the MySQL load times were extremely fast at just 5 seconds. While not quite as impressive but still a reasonable time, the MongoDB model managed to load the data in around 2.5 minutes. Comparatively the Cassandra database loaded all of the data in just under 3 minutes. This time is all the more impressive when you consider the fact that the Cassandra model was loading almost 3 times the data as that of the MySQL and MongoDB systems. Figure 6.1 illustrates the time taken to load the data against the volume of data being uploaded.

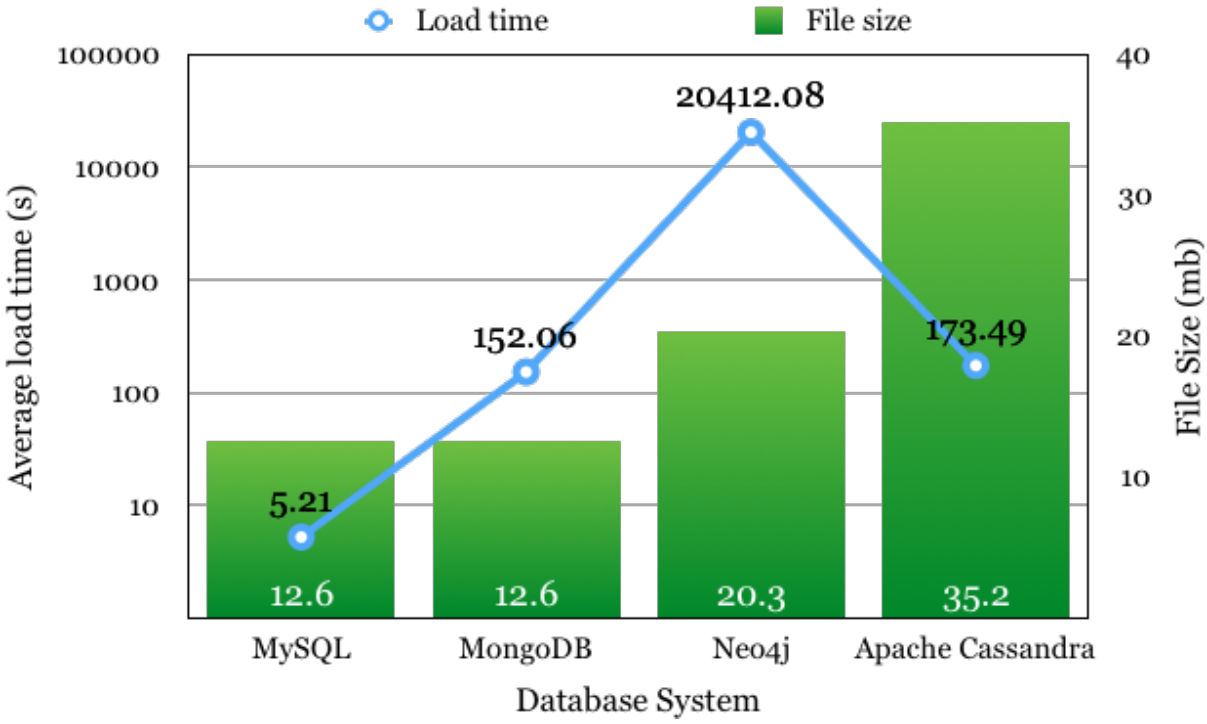

Figure 6.1: Chart illustrating the variance in load times of non-indexed systems vs the file size

As you can see from the graph, the MySQL system is running the import statements at a rate of around 2mB per second. The MongoDB system is importing the data at a rate of around 800kB per second. The Cassandra system is running the import commands at a rate of around 200kB per second. As the Neo4j system is evidently uncomprehendingly slow, the time taken to import the data is disregarded for this comparison.

The most striking result of the experiment was the Neo4j system taking on average around 5.5 hours to complete. That is 117 times slower than the next longest running load time of the Cassandra system and a staggering 3917 times slower than the fastest MySQL database. For

an in-depth discussion on the findings of this experiment, see section 6.2.3. Before moving on to the next stage of the experiment it is worthwhile making some assumptions about the results of test 1.

- As the MySQL tables have been normalised and structured adequately, the system can write a high number of rows at very high speed.
- By default the MongoDB model uses the _id field as an indexed primary key. Therefore after every insert or update operation, MongoDB must update every index associated with the collection in addition to the data itself. This increases the overhead for the performance of write operations.
- Neo4j model is struggling to process the high number of reads compounded with the writes when matching the relationships simultaneously.
- The Cassandra system is processing a reasonable level of operations for a larger volume of data.

6.2.2 Experiment 2

The next stage of the experiment was to implement the indexes and unique constraints on the data models. The processes for imposing these operations are discussed in chapter 5, section 5.1. The structure of the solutions was the same as that of the first experiment. All of the conditions were repeated in the same manner as before. It was important that I recreated the setup for the second test as close as possible to that of the first test. As a result, I was able to draw interesting and accurate conclusions. The overall aim of the second experiment was to identify any changes in load performance when imposing indexes on the solutions. The result of running the second experiment can be found in table 6.3.

Database System	Version	Load 1 time (s)	Load 2 time (s)	Load 3 time (s)	Load 4 time (s)	Load 5 time (s)	Average load time (s)
MySQL	5.7.11	8.10	9.45	8.32	8.15	8.37	**8.47**
MongoDB	3.2.4	370.80	342.11	347.50	393.82	366.25	**364.09**
Neo4j	2.3.3	175.03	121.54	128.0	145.62	120.41	**138.12**
Apache Cassandra	3.0.4	203.16	241.01	285.97	279.40	281.36	**258.18**

Table 6.3: Load times for indexed database systems

Table 6.3 shows the variance in load time after implementing indexes on the database solutions. Again there is a consistency in each of the execution times. With the exception of Neo4j, each of the solutions saw an increase in load time. The MySQL system took around 3 seconds longer. The time taken for the MongoDB model to load the data more than doubled, and the Cassandra system took around a minute longer. Figure 6.2 below illustrates the increase in load time of non-indexed vs indexed database solution. These findings posed

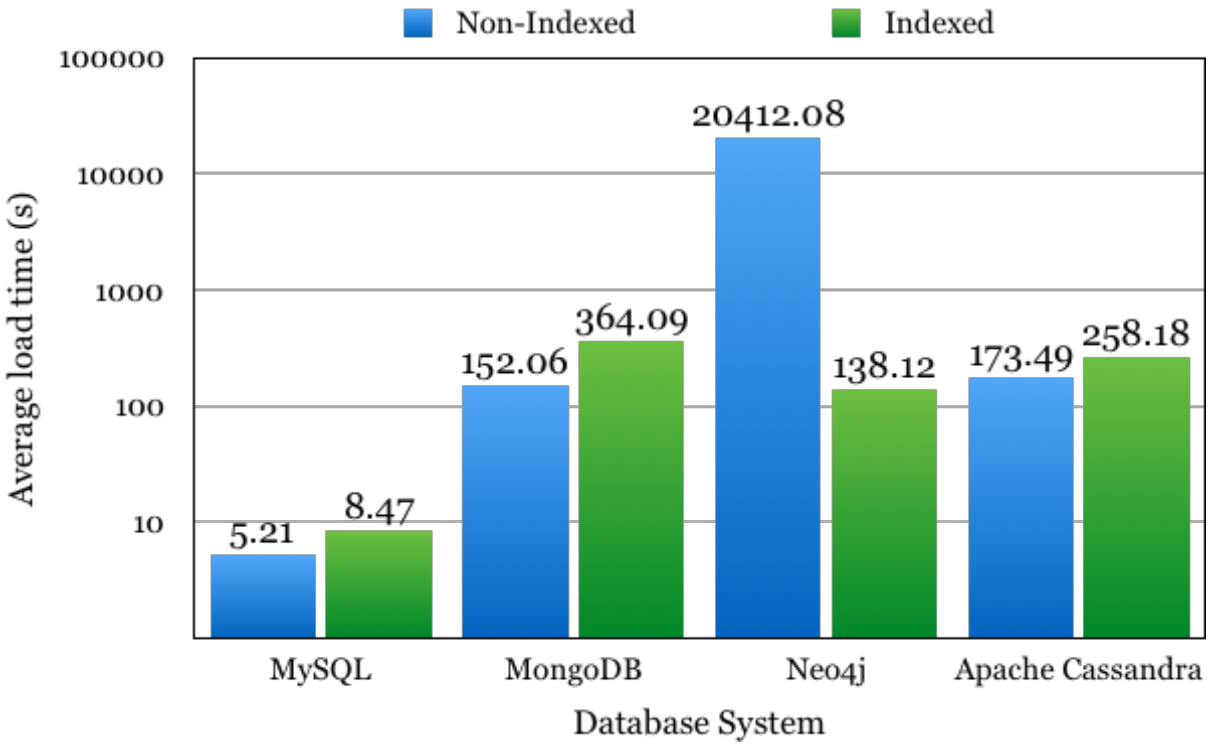

Figure 6.2: Chart illustrating the variance in load times of non-indexed vs indexed systems

the question: why did this alteration cause such a drastic affect? One detail which must be taken into consideration when evaluating these results is the fact that the experiment was not to only load raw CSV data into the solutions. The data was being loaded and modelled concurrently; in a way in which I believed would result in the most performant system. Therefore, in the cases of MongoDB and Neo4j, I used update_many and MATCH statements to create the relationships in the data. This added significant computational resource and time to the overall load process. For the MySQL and Cassandra solutions, as I had already created the table structures, I was able to directly load the data, without having to query the rest of the database.

6.2.3 Experiment evaluation

The official MongoDB documentation suggests that there are a number of operational considerations when implementing indexes. For example, they state that "Each index requires at least 8kB of data space" and "Adding an index has some negative performance impact for write operations. For collections with high write-to-read ratio, indexes are expensive since each insert must also update any indexes" [12]. These are all factors which may have contributed to the performance of the MongoDB system. The MongoDB documentation also suggests that to ensure the fastest possible processing, all indexes should be able to fit entirely in the RAM. Thus avoiding reading any indexes from disk [12]. To find the total index size in a collection, one can use the "db.collection.totalIndexSize()" command. When run on the EMAGE collection this statement returned 8159348 bytes (around 8.1 mB). As shown in table 6.1 the machine I was running the experiment on has 6gB of RAM, thus eliminating any concerns with machine performance contributing to the experiment. However, is something which should be reviewed when collecting a much greater volume of data in a MongoDB data model.

While I was unable to obtain any official Neo4j statistics to suggest indexing has a profound impact on performance, they do provide content to help improve the overall performance of a system. The main discovery from this experiment was the extent in which imposing indexes had on the Neo4j data model. The time taken to load the data decreased from 5.5 hours down to just over 2 minutes. The same procedure and query was run for both experiments; the only difference was the inclusion of indexes. This is not expected behaviour, as implementing indexes generally comes at a cost of additional storage space and slower writes.

To identify which part of the query was taking the longest, I split the process into individual data loads. Instead of one large data import I loaded each of the CSV files as a separate entities on the command-line. After doing this I saw that loading each of the files took a matter of seconds which was in-line with the other NoSQL imports. However, the final import of the Text Annotations data was the process which was taking over 5 hours to complete. The query which was run for experiment 1 (no indexes) can be found in code snippet 6.6 below. This is all one query which was run on the cqlsh command-line. It is useful to think of this statement as having 4 separate parts. Lines 2 and 3 are where the data is pointed to on the machine. The MATCH statements in lines 5, 6 and 7 allow one to specify the

patterns in which Neo4j will search for in the database. This is where the majority of the load time was being spent. Without the indexes, for each of the queries, the system was having to look through the entire database to find or match on one specific value.

```
1  -- Create TextAnnotation
2  USING PERIODIC COMMIT
3  LOAD CSV WITH HEADERS FROM "file:/home/callum/Documents/Uni/F20PA/Project/Neo4j/
       Data/TextAnnotations.csv" AS row
4
5  MATCH (assay:Assay {emageID : TOINT(row.emage_id)})
6  MATCH (structureID:AnatomyStructure {structureID: TOINT(row.structure_id)})
7  MATCH (gene:Gene {geneID: TOINT(row.gene_id)})
8
9  CREATE (annotation:TextAnnotation {strength: row.strength})
10
11 CREATE (annotation)-[:HAS]->(structureID)
12 CREATE (annotation)-[:RECORDS]->(gene)
13 CREATE (annotation)-[r:REPORTS]->(assay);
```

Code snippet 6.6: Cypher file to load text annotations data into the Neo4j data model.

This was run for Assays, Anatomy Structure and Genes. There are over 30,000 Assay nodes, over 5,000 Anatomy Structure nodes, and around 10,000 Gene nodes. Coupled with the fact there were over 140,000 Text Annotation nodes, these expensive read-write operations were the root cause of the length load time. The MySQL and Cassandra systems faired well in these experiments. While they did not show any notable signs of indexing having an effect on the load performance, there was a slight increase in overall execution time. This increase could be attributable to a number of factors, however as they are not significant enough for further examination, I did not pursue the analysis any further.

Chapter 7

Query Overview and Results

To analyse the functionality of the database systems, I devised a number of queries with ranging difficulties. The aim of the queries was to establish the limitations of the systems, and to discover the efficacy of the systems. A further objective was to identify any useful and unique functionality the solutions provide. For example any visualisations of graphical analysis which may enhance the exploration of the data. Chapter 3 provides a detailed explanation of the reasoning behind the queries. This chapter discusses the output and results of running the queries in the various query languages. This chapter also gives insight into the difficulty of writing the query, the usefulness of the output, the time taken for the query to run and any challenges I faced when throughout the process. Additionally included in this chapter is a general discussion which summarises the capabilities of the systems and aims to provide one with an understanding of the limitations of each of the systems.

7.1 Query Overview

The screen captures, tables, graphs and charts illustrated in this chapter are on the results of running the 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 includes a rating characterisation. The rating characterisation ranges from 1 to 5, and while there is no science behind the rating it is to be used as a guide to aid the reader into understanding the general complexity of the query. and fruitfulness of the data returned.

The order in which I wrote, run and evaluated the queries reflects the complexity of the query itself. I started by implementing simple queries which one would expect each of the

systems to successfully achieve. The difficulty of each query increased every time. The end point of this evaluation was when I had written a number of complex queries which would be used to assess the EMAGE dataset in real life scenarios. For example, query number 7 in table 7.1 "Calculate transitive closure". Finding the hierarchical path of a structure is something which researchers and scientists are often trying to achieve when analysing an anatomy. For example in terms of a human anatomy, the finger is part of the hand which is joined to the wrist which is part of the arm which is joined to the shoulder and so forth. Therefore its inclusion in this examination was necessary. A full description of this query, its origins and output is provided below.

Query Number	English	Expected return	Rating
1	All structures at Theiler Stage X.	Theiler Stage and Structure ID.	1
2	All structures between Theiler Stage X and Y.	Structure ID, Theiler Stage X and Theiler Stage Y.	1
3	Where is Gene X expressed?	Name of the gene, the structure where the gene was found and 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 and 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 and ID of the gene(s). The ID of structure X and ID of structure Y.	4
6	Which Genes are most commonly co-expressed?	Name of the gene(s) and the 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 queries for each database system. Rating scale: 1 (simple - complex) 5

There are a number of queries one could write which would output potentially interesting results and provide an alternative view of the data. However, the purpose of this examination was to evaluate the functionality of the solutions, assess the limitations of each of the systems and identify any interesting additional features. Thus there was no requirement to include these additional queries which would fail to affect the outcome of the examination.

7.2 Query Results

The following subsections provide the results of running the queries outlined in table 7.1. Included in the subsections is the statement written for every system for each of the queries, where I was able to achieve the intended outcome. I have also included the physical output for the first query for each database. This is to provide the reader with an insight into what is returned from each database solution when running a query. I have only included the output for the first query as I felt that doing the same process for each query did not add any value to the examination.

If a system was unsuccessful in returning the expected output it has been omitted from the results sections below. A full discussion and evaluation into the reasoning behind the failure of a given system is provided in section 8.1.

Query 1 - All structures at Theiler Stage X

The first query which I wrote for each of the databases aimed to retrieve the anatomy structure information at a given Theiler Stage. For the basis of these tests I hard coded the Theiler stage; Theiler Stage 4 in this instance.

The minimum amount of data expected to be returned from this query is the structure ID, structure name and Theiler Stage. This query is not complex and one would expect that each database will possess the required functionality to handle this query and return the expected output.

MySQL - Query 1 statement

The MySQL query shown in code snippet 7.1 successfully returned the expected output. The query was straightforward and only required a single join of the Stages and Anatomy Structures table.

I used the “AS” keyword to manually define an alias for the column headings of the returned data. The use of this keyword enhances the clarity of the output and allows one to easily identify the data being represented, as illustrated in figure 7.1. Without the inclusion of the “AS” keyword, the headings of the returned data would be the same as the column names in the select statement. For example, “StructureID” would be “AnatomyStructures.accession” and “StructureName” would be “AnatomyStructures.term”. The official MySQL

documentation suggests It is good practice to explicitly use the “AS” keyword for all column names [11].

```
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
```

Code snippet 7.1: MySQL query 1 statement. All structures at Theiler Stage X.

MySQL - Query 1 output

Figure 7.1 is a screen capture of the resulting output from running the MySQL query in the command-line interface. As you can see it returns the data in a table format separated by dashed lines. The number of rows returned and time taken for the query to process is also included below the table.

The output for this query is clear and coherent. However, for this query there is only 4 columns of data being returned. The MySQL command-line tool is run in a machine terminal. The size of the terminal window is at maximum the size of the machines monitor. Thus it is often the case that when returning a large number of columns, the data can become scrambled and illegible.

```
+-------------+---------------------+--------------+---------+
| StructureID | StructureName       | TheilerStage | DPC     |
+-------------+---------------------+--------------+---------+
|       16041 | inner cell mass     |            4 | 3.5 dpc |
|       16048 | polar trophectoderm |            4 | 3.5 dpc |
+-------------+---------------------+--------------+---------+
2 rows in set (0.00 sec)
```

Figure 7.1: Screen capture MySQL command-line tool output - Query 1

One way to avoid this is to change the zoom level of the terminal. Depending on the volume of data returned, this can have little to no affect as the further out one zooms, the smaller the font becomes. This can often be a hindrance of using the MySQL command-line tool for advanced queering of large, multicolumn datasets.

MongoDB - Query 1 statement

The MongoDB query shown in code snippet 7.2 successfully returned the expected output. To retrieve the data I used the *db.collection.find()* command. The query is made up of two main parts. Firstly the query matches the field and value in the curly brackets. This is shown in line 2 of code snippet 7.2. This line alone would have successfully returned the required data. However, it would have also returned the rest of data in the document, such as the publication and specimen information. As I was only looking to pull the Theiler Stage, structure term and structure ID, I had to include an additional parameter to the query.

The second part of this query is known as the projection. Within the projection, one can specify fields to exclude from the result. Lines 5-9 in code snippet 7.2 represent the projection of this query. A field can be included or excluded from a result by specifying the relevant field and then stating either 1 (included) or 0 (excluded) value. Doing so, allows one to filter the result and only return the fields required.

```
1 db.emage.find(
2   { "stage.theilerstage":4  },
3     { "_id" : 0,
4     "stage.theilerstage" : 1,
5     "stage.dpc" : 1,
6     "textannotation.anatomystructure.structureID" : 1,
7     "textannotation.anatomystructure.term" : 1  } ).pretty();
```

Code snippet 7.2: MongoDB query 1 statement. All structures at Theiler Stage X.

One additional command which concludes this query is the “pretty()” method. This command displays the results in an easy-to-read format. A standard *db.collection.find()* query outputs data in a dense and incomprehensible format. Using the “pretty()” method returns the data in a format which is easier for humans to understand.

MongoDB - Query 1 output

Figure 7.2 illustrates the results returned from running query 1 in MongoDB. The query was run on the MongoDB command-line tool. As with MySQL, when running a query which may return a large volume of data, the command-line interface can often be difficult to view the results. However, with the inclusion of the “pretty()” method, analysing the field and value pairs of a MongoDB document is greatly enhanced.

```
{
        "stage" : {
                "dpc" : "3.5 dpc",
                "theilerstage" : 4
        },
        "textannotation" : [
                {
                        "anatomystructure" : {
                                "term" : "inner cell mass",
                                "structureID" : 16041
                        }
                },
                {
                        "anatomystructure" : {
                                "term" : "polar trophectoderm",
                                "structureID" : 16048
                        }
                }
        ]
}
```


Figure 7.2: MongoDB command-line tool output - Query 1

Neo4j - Query 1 statement

The Neo4j query shown in code snippet 7.3 successfully returned the expected output. Cypher, the Neo4j query language, while uniquely different, is similar to SQL. They both share many of the same functions and methods. One distinctive difference is that Cypher queries specify the data to be returned at the end of the query, as opposed to the start. Despite this dissimilarity, the structure and format of SQL and Cypher queries are very similar.

Line 1 in code snippet 7.3 is indicative of this. The “MATCH” command, firstly begins the query, and also specifies which nodes the data will be pulled from. This is the equivalent of a SQL “FROM” keyword. For example, “(struct:AnatomyStructure)” specifies the data will come from an “AnatomyStructre” node and for the basis of the query the alias will be “struct”. This is the same for the “Stage” node. Also included in line 1 is the relationship which will join the “AnatomyStructure” and “Stage” nodes. One can either explicitly state the relationship between the nodes, in the square brackets. Alternatively, using an empty set of brackets or stating just an alias, will tell the system to find the relationship between the nodes regardless of its value.

```
1 MATCH (struct:AnatomyStructure)-[Relationship]->(stage:Stage)
2 WHERE stage.theilerStage = 4
3 RETURN struct.structureID AS StructureID, Relationship, stage.theilerStage AS
    TheilerStage, stage.dpc AS DPC;
```

Code snippet 7.3: Neo4j query 1 statement. All structures at Theiler Stage X.

For example, in this query, the relationship between the nodes is “GROUPED_BY”. However, as I have just provided an alias, the system will match whichever relationship it finds between the nodes.

Line 2 of code snippet 7.3 is a standard “WHERE” clause. It has the same semantic value as an SQL “WHERE” clause. Finally the query is finished with a “RETURN statement”, found on line 3. This is equivalent to a SQL “SELECT” command. One specifies the field and value pairings as required. As with MySQL it is good practice to use alias names, and can be done so by expressing the “AS” keyword.

Neo4j - Query 1 output

By default Neo4j provides two tools for querying the database and outputting the result. The first is on the Neo4j command-line tool; a basic command-line interface for querying a database. The way it works is similar to that of MySQL, MongoDB and Cassandra; start the tool, connect to a database, then run a query.

```
+-------------------------------------------------------------------------------+
| StructureID | Term                | Relationship        | TheilerStage | DPC      |
+-------------------------------------------------------------------------------+
| 3210        | "polar trophectoderm" | :GROUPED_BY[6503]{} | 4            | "3.5 dpc" |
| 2085        | "inner cell mass"     | :GROUPED_BY[5378]{} | 4            | "3.5 dpc" |
+-------------------------------------------------------------------------------+
2 rows
156 ms
```

Figure 7.3: Neo4j query 1 - All structures at Theiler Stage X

The output from running the query in code snippet 7.3 using the command-line tool is illustrated in figure 7.3. The output is almost identical with that of MySQL. This does not come as a surprise as Neo4j is based on SQL constructs. It is a basic format however it meets the needs for most.

A reason why the creators of Neo4j did not differ from the norm regarding the command-line interface, may be down to the fact they have developed a web interface for querying Neo4j models. The web browser application comes as standard with the community edition (the most basic version) of Neo4j. It is the primary user interface for Neo4j. Using this application one can run queries in the same manner as running queries on the command-line tool. The application also displays the nodes, relationships and constraints of the data mode. It is designed to provide one with a intuitive, user-friendly experience of Neo4j. It is also a useful tool for those trying to learn how to use and understand graph database management systems. Having the ability to visualise how data is represented and structured is an extremely helpful learning aid.

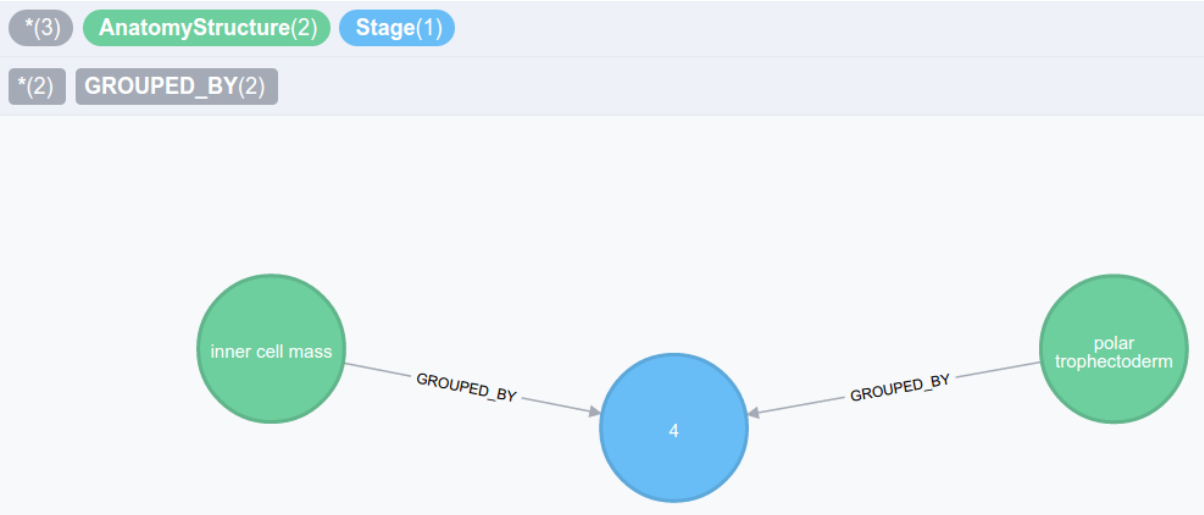

Figure 7.4: Neo4j query 1 - All structures at Theiler Stage X

One feature of the application is its graph visualisation tool. Illustrated in figure 7.4, the web interface allows one to view the result of their query as a graph. The nodes and relationships displayed, represents the of the query in code snippet 7.3. The colours, sizes and labels of the nodes are fully customisable and one can move (by dragging) the nodes around the screen at ones convenience.

The visualisation tool is basic, but easy to use. Its inclusion in the application enhances the usability of Neo4j, for those who are looking for an alternative way of viewing their data. However, it is not very robust. While running a query which returns a large volume of data may not take a long time to run, representing the data in the interface can take some time to load. Furthermore, the web interface is also prone to crashing when returning a lot of data.

It is often the case when querying a database using an API or otherwise, that one intends on loading the data into a visualisation tool or implementing a solution to model the data. Therefore rendering the addition of this application obsolete. That said, it is a useful tool for basic data visualisation and database querying.

Apache Cassandra - Query 1 statement

Code snippet 7.4 represents the query written to find all of the structures found at a given Theiler Stage in the Cassandra data model. The Cassandra query language uses similar syntax to that of MySQL. They both share many of the same keywords and the structure of a query is the same. Start by selecting the columns one needs for the query, state which table the data is stored in and then include any clauses to match the data on.

The query is much shorter and simpler compared with the MySQL statement for the same query. This is as a result of the inability for Cassandra to join tables. Therefore all of the data required to output the result of this query is stored in a single table.

```
1 SELECT *
2 FROM structurebystage
3 WHERE theilerstage = 4
4 AND detected = true;
```

Code snippet 7.4: Cassandra query 1 statement. All structures at Theiler Stage X.

Apache Cassandra - Query 1 output

The output of running the query in code snippet 7.4 can be found in figure 7.5. As with Neo4j, to the naked eye, the output of running queries on the command-line looks very similar to that of MySQL. Despite how Cassandra is semantically structured, visually, there is not much dissimilarity.

One difference with Cassandra is the colouring of the column headings. The colours themselves have no significance on how the table is structured, and are dependent on the configuration of one's terminal window. However, the positioning and distinctness of the colours, do have a bearing on the table structure. In the instance of figure 7.5, the initial two columns coloured red represent the partitioning keys, defined in section 5.1.4. The cyan coloured column represents the clustering key, and the pink coloured headings define the

remaining columns. In terms of the data, the green coloured values depict either an Integer or a Boolean value and the purple coloured data represents a String.

```
 theilerstage | detected | structureid | dpc     | structureterm
--------------+----------+-------------+---------+--------------------
            4 |     True |       16041 | 3.5 dpc |     inner cell mass
            4 |     True |       16048 | 3.5 dpc | polar trophectoderm

(2 rows)
```

Figure 7.5: Cassandra query 1 - All structures at Theiler Stage X

In terms of the effectiveness of this feature, it certainly makes the output easier on the eye and more attractive from a aesthetic standpoint. However, I feel it adds little to the overall impact of the data returned. Depending on the role of the user querying the database, having a colour coordinated keys and data types may have little impact. For example, if one is designing the data model in a Database Administrator type role, colours representing the partitioning and clustering keys can be useful. It allows one to see exactly how the table is structured without having to run a further query to describe the table. Having coloured data values also shows whether a number is indeed an Integer or a String for example. Comparatively, if one is in a business type role and does not understand the underlying architecture of the database, then the various colours may confuse or complicate their analysis. While having many colours does give the output an interesting look and feel, it does not improve the querying experience. It does seem that this feature is to primarily enhance the practicality of the database when querying, which it only achieves to an extent.

Query 2 - All structures between Theiler Stage X and Y

The second query aimed to identify all of the structures found between two given Theiler Stages. For each of the queries I used the same minimum and maximum Theiler Stages values (4 and 7). This is a query which one would anticipate each of the systems would complete with relative ease. The data expected to be returned from the query is, as a minimum, the structure ID and the corresponding Theiler Stage.

MySQL - Query 2 statement

The MySQL query shown in code snippet 7.5 successfully returned the expected output. To construct the query I had to write two inner joins. The first linking the AnatomyStructures table to the Stages table, and the second joining the AnatomyStructures table to the TextAnnotation table.

```
1 SELECT t1.accession, t2.theilerstage
2 FROM AnatomyStructures AS t1
3 INNER JOIN Stages AS t2
4 ON t1.stage_id = t2.id
5 INNER JOIN TextAnnotations AS t3
6 ON t1.id = t3.structure_id
7 WHERE t2.theilerstage BETWEEN 4 AND 7
8 AND t3.detected = 1
9 GROUP BY 1;
```

Code snippet 7.5: MySQL query 2 statement. All structures between Theiler Stage X and Y.

I have imposed two clauses on the query, found in lines 7 and 8 of code snippet 7.5. The first, on line 7 stipulates the Theiler Stage must have a value between 4 and 7. On line 8 I specify that the gene must have a boolean value of 1; detected. The GROUP BY clause on line 9, is used to group the result-set by column 1 (structure ID). This will return a distinct set of structure ID values.

The query was straightforward to create. Compared to Neo4j and Cassandra however, it was a long query in terms of syntax. This can often be the case when writing relational database queries. The more tables in the data model, the longer and more complex the query becomes.

MongoDB - Query 2 statement

The MongoDB query shown in code snippet 7.6 successfully returned the expected output. MongoDB supports a number of logical and comparison query operators. These include $and (logical and), $gte (>=) and $lte (<=), each of which are used in the query below.

The $and logical operator denotes field and value pairs must all satisfy the arguments provided, for a query to return true, within an array of two or more expressions. It is semantically the same as a logical AND. If one of the arguments returns false, MongoDB will

not evaluate the remainder of the expression. The $gte and $lte operators select the documents where the value of the field is either greater/less than or equal to a specified value.

```
db.emage.find(
  { $and:
    [ { "textannotation.detected" : "True" },
      { "stage.theilerstage": {$gte : 4 } },
      {  "stage.theilerstage": {$lte : 7 }  },
      { "textannotation.strength" : "detected" }
    ] },
  { "textannotation.anatomystructure.structureID" : 1, "stage.theilerstage" : 1, "
      _id" : 0  } ).pretty();
```

Code snippet 7.6: MongoDB query 2 statement. All structures between Theiler Stage X and Y.

Neo4j - Query 2 statement

The Neo4j query shown in code snippet 7.7 successfully returned the expected output. This was a basic query and I found it easy to construct as it was extremely similar to that of the MySQL query created previously.

```
MATCH (textannotation:TextAnnotation)-[]->(struct:AnatomyStructure)-[]->(stage:
    Stage)
WHERE stage.theilerStage >= 4
AND stage.theilerStage <=7
AND textannotation.detected = 1
RETURN struct.accession AS Accession, stage.theilerStage AS TheilerStage;
```

Code snippet 7.7: Neo4j query 2 statement. All structures between Theiler Stage X and Y.

Apache Cassandra - Query 2 statement

The Cassandra query shown in code snippet 7.8 successfully returned the expected output. This was a simple query and quick to write.

```
SELECT *
FROM structurebystage
WHERE theilerstage IN (4,5,6,7)
AND detected = true;
```

Code snippet 7.8: Cassandra query 2 statement. All structures between Theiler Stage X and Y.

This was mainly due to the fact I had modelled the table around such an query. One noteworthy point from this query, is the use of the IN operator. Cassandra Query Language only supports the use of = and IN operators when restricting a partition key within a query. Therefore as opposed to using either >= or the BETWEEN operators in MySQL, MongoDB and Neo4j, for Cassandra I had to hard-code the physical values I was restricting the column on. This is illustrated in line 3 of code snippet 7.8.

Query 3 - Where is gene X expressed?

The third query I created was to test if the systems could successfully return information regarding a given gene. One would again expect that each of the solutions should be able to return the expected data with ease. I chose the gene “Hoxb13” as the gene to test the queries. For the solution to be deemed successful in running this query I would expect the gene name, structure name and structure ID to be returned, as a minimum.

MySQL - Query 3 statement

The MySQL query shown in code snippet 7.9 successfully returned the expected output. It returned each of the intended columns. There are two joins in this query, linking the Genes and TextAnnotations, and also the AnatomyStrucutres and TextAnnotations tables together. As with query 2, I used the GROUP BY clause to only return a distinct set of structure ID values.

```
1 SELECT t1.name AS GeneName, t3.term AS StructureTerm, t3.accession AS StructureID
2 FROM Genes AS t1
3 INNER JOIN TextAnnotations AS t2
4 ON t1.id = t2.gene_id
5 INNER JOIN AnatomyStructures AS t3
6 ON t2.structure_id = t3.id
7 WHERE t2.detected = 1
8 AND t1.name = 'Hoxb13'
9 GROUP BY 3
```

Code snippet 7.9: MySQL query 3 statement. Where is gene X expressed?.

MongoDB - Query 3 statement

The MongoDB query shown in code snippet 7.10 successfully returned the expected output. Compared with the MySQL query, there was significantly less overhead to complete this query.

```
db.emage.find(
  { $and:
    [
      { "textannotation.gene.name": "Hoxb13"  },
      { "textannotation.detected": "True" }
    ]
  } ).pretty();
```

Code snippet 7.10: MongoDB query 3 statement. Where is gene X expressed?.

Neo4j - Query 3 statement

The Neo4j query shown in code snippet 7.11 successfully returned the expected output. The clauses and relationship joins in the Neo4j query, mirror that of the MySQL query.

```
MATCH (g:Gene)<-[]-(textannotation:TextAnnotation)-[]->(a:AnatomyStructure)
WHERE g.name = 'Hoxb13'
AND textannotation.detected = 1
RETURN g.name AS Name, a.accession AS  StructureID, a.term AS Term
```

Code snippet 7.11: Neo4j query 3 statement. Where is gene X expressed?.

Apache Cassandra - Query 3 statement

The Cassandra query shown in code snippet 7.12 successfully returned the expected output. One noteworthy takeaway from this query is the use of "ALLOW FILTERING". Cassandra has the ability to determine if a query will execute efficiently. When a query is run and there is a possibility that the efficiency of the query will be compromised, the system interrupts the query with the following statement: *"Bad Request: Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING"*. Thus its inclusion in the query. The only way Cassandra can execute this query is, to return all of the rows in the "structurebygene3" table, and then filter out the rows which do not meet the "genename = 'Hoxb13'" clause.

```
1 SELECT *
2 FROM structurebygene3
3 WHERE genename = 'Hoxb13'
4 AND detected = true
5 ALLOW FILTERING;
```

Code snippet 7.12: Cassandra query 3 statement. Where is gene X expressed?.

There is an alternative to using ALLOW FILTERING, which is to apply a secondary index on the "genename" column. This allows Cassandra to filter on this index thus not require ALLOW FILTERING.

The decision to choose either ALLOW FILTERING or add an index should be considered when implementing the query and also when creating the data model. The correct decision will depend on the specific use case. I decided to use ALLOW FILTERING as this is the only query which requires ALLOW FILTERING and thus felt implementing an index was extra overhead for little gain.

Query 4 - What is expressed in structure X?

This query was designed to identify the data which is expressed in a given structure. With the inclusion of this query, the complexity has again increased however, one would expect each of the solutions to successfully achieve this query. The data returned from this query is what makes it interesting. The ability to view what information is contained within a given structure, provides an essential analytical insight into the data. The minimum data expected to be returned from this query is the gene name, structure ID, structure term, Theiler Stage and EMAGE ID.

MySQL - Query 4 statement

The MySQL query shown in code snippet 7.9 successfully returned the expected output. The one aspect of this query which increased its complexity was the number of joins I had to impose. The table joins are:

- Genes (t1) -> TextAnnotations (t3)
- TextAnnotations (t3) -> AnatomyStructures (t2)

- AnatomyStructures (t2) -> Stages (t4)
- TextAnnotations (t3) -> Assays (t5)

```
1  SELECT t1.name AS GeneName, t2.accession AS StructureID, t2.term AS TermName, t4.
       theilerstage AS TheilerStage, t3.emage_id AS EMAGE_ID
2  FROM Genes AS t1
3  INNER JOIN emage.TextAnnotations AS t3
4  ON t1.id = t3.gene_id
5  INNER JOIN emage.AnatomyStructures AS t2
6  ON t3.structure_id = t2.id
7  INNER JOIN Stages AS t4
8  ON t2.stage_id = t4.id
9  INNER JOIN Assays AS t5
10 ON t3.emage_id = t5.emage_id
11 WHERE t3.detected = 1
12 AND t2.accession = 17451
```

Code snippet 7.13: MySQL query 4 statement. What is expressed in structure X?.

MongoDB - Query 4 statement

The MongoDB query shown in code snippet 7.14 successfully returned the expected output. This is a very short query compared with the other systems. Line 2 of code snippet 7.14 matches the specified structure ID value for the field. Line 2 restricts the fields which will be returned.

```
1  db.emage.find(
2    { "textannotation.anatomystructure.structureID": 17451  },
3    { "_id":1}  ).pretty();
```

Code snippet 7.14: MongoDB query 4 statement. What is expressed in structure X?.

Neo4j - Query 4 statement

The Neo4j query shown in code snippet 7.15 successfully returned the expected output. Apart from the increased relationship joins, this query did not require any additional functionality which has not been discussed previously.

```
1 MATCH (stage:Stage)<-[]-(struct:AnatomyStructure)<-[]-(text1:TextAnnotation)-[]->(
      assay:Assay)
2 WHERE struct.accession = 17451
3 RETURN struct.accession AS StructureID, struct.term AS Term,stage.theilerStage AS
      TheilerStage, assay.emageID AS EMAGEID
```

Code snippet 7.15: Neo4j query 4 statement. What is expressed in structure X?.

Apache Cassandra - Query 4 statement

The Cassandra query shown in code snippet 7.16 successfully returned the expected output. As with Neo4j, there was no additional functionality required to complete this query which has not been discussed previously. As illustrated in code snippet 7.16 the query was relatively short and straightforward.

```
1 SELECT *
2 FROM textannotations1
3 WHERE structureid = 17451
4 AND detected = true
5 ALLOW FILTERING;
```

Code snippet 7.16: Cassandra query 4 statement. What is expressed in structure X?.

Query 5 - Which Genes are stored in structures X and Y?

The first four queries were created with the intention of scoping the minimum level of expectancy for the solutions. While they were returning potentially useful and interesting data, they did not require any specialist functional capabilities. The next query, query five, was the first which was designed to query the systems in terms of advanced real world application.

This query aims to find the genes which are stored in a given two structures, X and Y. For the basis of this query I chose structures 16062 and 16069 at random. For this query to be deemed a success, a solution must return at least the gene name.

The result of this query identifies so-called gene co-expression. Gene co-expression can be represented as an undirected graph, with each node in the graph corresponding to a gene. The edges of the graph connect a pair of nodes (genes) if there is significant co-expression between them. This type of analysis, is of particular interest to biologists and researchers, as co-expressed genes are controlled by functionally related members of the same pathway.

MySQL and Neo4j were the only systems which I was able to successfully fulfil the requirements of queries five, six and seven. An evaluation into the reasoning behind the failure of the other database systems is provided in section 8.1.

MySQL - Query 5 statement

The MySQL query shown in code snippet 7.17 successfully returned the expected output. This query was more complex to write compared with the previous four queries. In terms of functionality, the query used mostly the same operators which I have used and discussed already. There are two inner joins, the Genes table links to the TextAnnotations table, and the TextAnnotations table joins with the AnatomyStructures table. These are common joins for this data model, which we have come to expect as a result of the previously run queries.

```
1  SELECT t1.name AS GeneName
2  FROM Genes AS t1
3  INNER JOIN TextAnnotations AS t3
4  ON t1.id = t3.gene_id
5  INNER JOIN AnatomyStructures AS t2
6  ON t2.id = t3.structure_id
7  WHERE t3.detected = 1
8  AND t2.accession IN (16062, 16069)
9  GROUP BY 1
10 HAVING COUNT(DISTINCT t2.accession) > 1;
```

Code snippet 7.17: MySQL query 5 statement. Which Genes are stored in structures X and Y?.

As I was only interested in positive gene expression, I included a clause on line 7 of code snippet 7.17, which filters out “possible” or “not detected” strength values. The use of the “IN” operator on line 8, was used to determine if the structure ID value matched any one of the values in the given list. The “IN” clause can be used to replace many “OR” conditions.

The final notable operator used in this query was “HAVING”. This operator is used to apply clauses on aggregate functions as the “WHERE” keyword, can not be used. What this query

essentially says is “Find all of the detected genes in structures 16062 and 16069 where the gene appears at least once in each of the structures.”

Neo4j - Query 5 statement

The Neo4j query shown in code snippet 7.18 successfully returned the expected output. As with the previous queries, once I had successfully achieved the intended outcome with MySQL, I was able to use this to structure the Neo4j query. This is as a result of the querying languages sharing many constructs, operators and functional capabilities.

```
1 MATCH (g:Gene)<-[]-(t:TextAnnotation)-[]->(a:AnatomyStructure)
2 WITH COUNT (DISTINCT (a.accession)) AS structureIDCount, t AS textAnnotation, a AS
      anatomyStructure, g AS gene
3 WHERE textAnnotation.detected = 1
4 AND anatomyStructure.accession IN [16062, 16069]
5 AND structureIDCount > 1
6 RETURN DISTINCT gene.name
```

Code snippet 7.18: Neo4j query 5 statement. Which Genes are stored in structures X and Y?.

Query 6 - Which Genes are most commonly co-expressed?

Query six followed on from the co-expression evaluation in query five. This query aimed to identify the genes which were most commonly co-expressed. The main structure of query six was based on query five. However, it also included the added complexity of returning a subset of the data. For this query to be deemed a success, the solution must return the gene name and the number of times it is co-expressed.

Determining the genes which are most commonly co-expressed may be seen as a requirement for business intelligence. Adding this extra difficulty to the query, provides more of an understanding from an analytical standpoint, as opposed to evaluating the functional competence of the systems. However, within the context of this experiment, the query is valid. Having the ability to return data which would be seen as a requirement in a real world situation, is a component in the overall evaluation of the database systems.

MySQL - Query 6 statement

The MySQL query shown in code snippet 7.19 successfully returned the expected output. As previously discussed, the main structure of this query is based on query five (code snippet 7.17). The joins are the same, with the Genes table linking to the TextAnnotations table, and the AnatomyStructures table joining on to the TextAnnotations table. The "WHERE" clauses are also the same with the restriction on the detection strength.

```
1  SELECT t1.name AS GeneName, COUNT(t2.accession) AS CoExpressedCount
2  FROM Genes AS t1
3  INNER JOIN TextAnnotations AS t3
4  ON t1.id = t3.gene_id
5  INNER JOIN AnatomyStructures AS t2
6  ON t2.id = t3.structure_id
7  WHERE t3.detected = 1
8  GROUP BY 1
9  HAVING COUNT(DISTINCT t2.accession) > 1
10 ORDER BY 2 DESC
11 LIMIT 5;
```

Code snippet 7.19: MySQL query 6 statement. Which Genes are most commonly co-expressed?.

To identify the most commonly co-expressed genes, I ordered the data returned by the count of the structure IDs in descending order. This is illustrated in line 10 of code snippet 7.19. I then imposed a limitation on the number of results returned to 5.

While there was an added clause within this query, I did not feel the complexity of writing it was increased. The querying functionality MySQL provides can handle analytical insights such as this with ease.

Neo4j - Query 6 statement

The Neo4j query shown in code snippet 7.20 successfully returned the expected output. This query required the same modifications as MySQL. This query was devised as the output is used for real-world analysis, as opposed to its ability to expose the limitations of the systems.

```
1 MATCH (g:Gene)<-[]-(t:TextAnnotation)-[]->(a:AnatomyStructure)
2 WITH COUNT (DISTINCT (a.accession)) AS structureIDCount, t AS textAnnotation, a AS
      anatomyStructure, g AS gene
3 WHERE textAnnotation.detected = 1
4 AND anatomyStructure.accession = 16062
5 AND structureIDCount > 1
6 RETURN DISTINCT gene.name AS GeneName
7 ORDER BY structureIDCount DESC
8 LIMIT 5
```

Code snippet 7.20: Neo4j query 6 statement. Which Genes are most commonly co-expressed?.

Query 7 - Calculate transitive closure

The criteria for query seven was to calculate transitive closure. The concept of transitive closure is to provide one with ability to answer so-called reachability questions. For example, if there is a direct flight from city A to city B, and a direct flight from city B to city C, does this mean that there is a direct flight from city A to city C? Intuitively, one would assert this as being an indeterminable possibility. Thus, transitive closure aims to find out if it is indeed a possibility, that there is a direct flight from city A to C.

Transitive closure gives one the ability to do queries such as “find me all of the children of X” and “how many parents does child X have?”. This is quite a common and powerful query, once it has been achieved.

In the case of the EMAGE dataset, one is looking to determine the hierarchical path of a given structure. For this query to be deemed a success, the system must return the full hierarchical path from a starting term to its top most relation. I chose structure ID 16201 as the starting node. The expected path of this structure is: arterial system -> vascular system -> cardiovascular system -> organ system -> embryo -> mouse.

MySQL - Query 7 statement

The criteria of this query was to return the hierarchical path of a given structure. I was able to write a MySQL query which did return the expected output. However, to complete this implementation I had to manipulate the functionality and slightly manufacture the result.

Code snippet 7.21 represents the MySQL query I created to achieve in the intended outcome of query 7. In order to run this query, I had to create an extra table Closure. The Closure table was populated with data from the EMAPA dataset. Figure 7.6 is an ER diagram which represents the data contained within the Closure table.

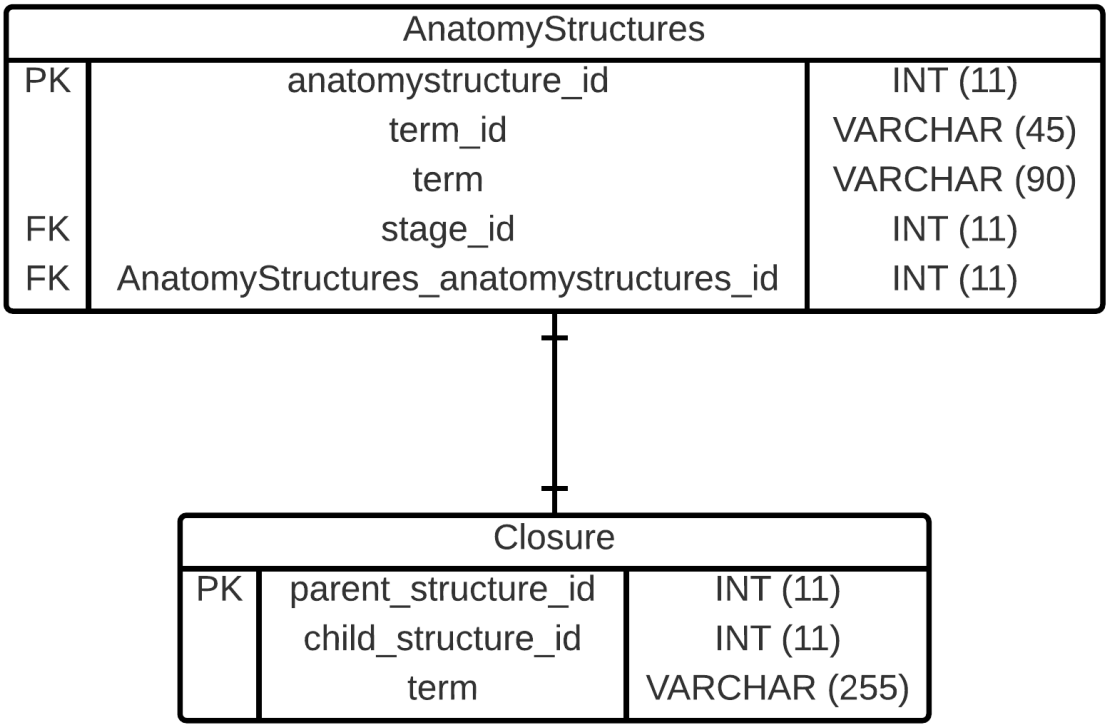

Figure 7.6: ER diagram of the temporary Closure table

The Closure table contains 3 columns:

- **parent_structure_id**: The structure ID of the direct parent of the child.
- **child_structure_id**: The structure ID of the child, which is a direct descendent of the parent.
- **term**: The term name of the child structure ID.

The primary key of the Closure table, “parent_structure_id”, is joined to the AnatomyStructures table using the “anatomystructure_id” value. The EMAPA data was loaded into the database using the same procedure as described in section 6.1.1.

```
1 SELECT DISTINCT tmp.term, t1.parent_term AS lev1, t2.parent_term AS lev2, t3.
      parent_term AS lev3, t4.parent_term AS lev4
2 FROM Closure AS t1
3 INNER JOIN AnatomyStructures AS tmp
4 ON t1.child_id = tmp.accession
5 LEFT JOIN Closure AS t2 ON t2.child_id = t1.parent_id
6 LEFT JOIN Closure AS t3 ON t3.child_id = t2.parent_id
7 LEFT JOIN Closure AS t4 ON t4.child_id = t3.parent_id
8 WHERE tmp.accession = 16201
```

Code snippet 7.21: MySQL query 7 statement. Calculate transitive closure.

The implementation was quite straightforward. The query works by imposing repeated self-joins on the Closure table using the child and parent ID values. As a result, one must know the depth of the tree to return the full path. For example, in code snippet 7.21 there are 4 self-joins; lines 5, 6 and 7. Including the structure ID itself, the depth of structure ID 16201 is therefore 5. Thus, returning the full hierarchical path of the structure. For the basis of this query, one would say that MySQL was successful in returning the required data. However, this is far from an ideal implementation.

Neo4j - Query 7 statement

The Neo4j query shown in code snippet 7.22 successfully returned the expected output. Unlike the workaround I created for the MySQL query, I was able to use the functionality Neo4j provides to achieve the required outcome. Neo4j is a directed acyclic graph, and therefore it has the functional ability to traverse a hierarchical structure. This allowed me to return the full path of a given node using one function; “allShortestPath”.

```
1 MATCH ( child:Child { structureID: 16201 } ),( parent:Parent { term: 'mouse' } ),
2 path = allShortestPath( (child)-[*]-(parent) )
3 RETURN path
```

Code snippet 7.22: Neo4j query 7 statement. Calculate transitive closure.

The “allShortestPath” node works by finding the single shortest path between two nodes (child and parent). Inside the parenthesis on line 2 of code snippet 7.22, I defined the path.

This was the staring node; the child, the connecting relationship and the end node; the parent. The function uses a predicate which returns all of the available paths between the two nodes.

Neo4j was the only system to achieve all 7 of the queries. It provides the functional ability to not only achieve the basic queries, which have become a minimum requirement for all database management systems, but also complete more complex analytical queries. An early task one must undertake when selecting a database system is to get to know ones data to fully understand what data one is modelling. Not only what the data is, but how the relationships are formed and how the data is structured. Only then can one confidently select the correct system for one's application. The EMAPA and EMAGE datasets are both ideal for a Neo4j system as they contain hierarchical data. One should not assume that because MySQL, MongoDB and Cassandra failed to achieve the co-expression and transitive closure queries, they are inferior systems. The Neo4j system was simply more suited towards the dataset and queries used in this examination. Further discussion and analysis on this can be found in section 8.1.

Chapter 8

Summary and Conclusion

8.1 Evaluation

Each of the objectives identified in section 3 have been achieved. These are outlined below:

Database Scalability

The scalability of the databases was evaluated by loading the EMAPA and EMAGE datasets into each data model. This provided one with an insight into how adequate the solutions are in handling a big dataset. Each database indicated that they are more than capable of scaling to meet the necessary requirements; to store and import a large volume of data.

Schema Implementation Complexity

A key attribute of NoSQL technologies is their schema-less characteristics. Removing the strict, rigid, structural boundaries which a relational databases imposes allows one to freely adapt and manipulate ones data model with ease. Specifically so within the document-orientated MongoDB environment. Implementing the MongoDB data model, provides one with a sense of control and flexibility, which I found was lacking using a relational database management system.

Analytical Insight

Many of today's largest, most influential and powerful companies in the world are reaping the rewards of collecting and storing big data successfully. This being said, data is useless if it is unclear or impossible to understand. Having the ability to illustrate and visualise data is a key factor in turning data into value. While each of the solutions

evaluated all offer a standard flat file export which everyone is familiar with (CSV or equivalent); only Neo4j, the graph-orientated database delivers a refreshing approach to data visualisation. There is a plethora of third-party applications and programming language APIs which can transform one's data into a visual masterpiece. However, Neo4j uses its property graph model structure, to provide one with an out-of-the-box modelling alternative, to previously accepted industry standard traditions.

Query Language Capabilities

Each solution has its own dedicated query language. They all possess functionality which has become standard for any database management system. Neo4j's query language Cypher, is based on SQL constructs, and therefore shares much of the same functional capabilities. Cassandra is the most like a relational database in terms of structure; facilitates similar querying abilities to SQL. The MongoDB querying syntax is vastly different, however functionally, it is not dissimilar to the other solutions.

The query testing which was undertaken as part of the project, exposed the limitations of the systems. Basic functional querying was achieved for all of the solutions. I was able to retrieve subsets of data, capable of providing satisfactory analytical insight into the underlying dataset. However, these insights were limited. Using the respective systems querying languages, I was only able to implement more complex querying such as gene co-expression and calculating transitive closure for Neo4j and MySQL (using a workaround). Admittedly gene co-expression is specific to a biological dataset, however it does have application in other fields.

Neo4j was the only solution which was able to successfully return the expected output for all of the queries. The first four simplistic queries were completed with ease, and the more complex co-expression and transitive closure queries were also achieved. This was as a result of Neo4j's acyclic graph model which allows one to query the path of a relationship.

It is common knowledge that the expressiveness of MySQL systems is limited. Especially so, when defining recursive queries such as transitive closure. Unlike other relational database systems, such as PostgreSQL and DB2 for example, MySQL does not provide the built-in functionality to complete queries such as these. For example, using PostgreSQL, one has the ability to use the "RECURSIVE" modifier in a "WITH" query. This feature allows one to accomplish results otherwise not possible in standard legacy

SQL systems. The "RECURSIVE" modifier can use the output of the "WITH" query which allows it to query hierarchical data. This is just one alternative relational database system one can use, however there are others which produce similar results. It is often the case with open source software such as the community-edition of MySQL, that there are limitations to what one can achieve. This is a consideration one must decide upon when selecting a database management system. Ensure that the product can facilitate the needs of the requirements or be forced to pay subscription and maintenance fees for a more advanced product. The rights to MySQL was sold to Oracle in 2008, and since then developmental progress seems to have ground to a halt. There has been just one major software release in the past several years. This is a major problem, as there is no official route for developers to discuss the system with Oracle.

The MongoDB and Cassandra systems were unable to successfully complete all of the devised queries. The systems failed to achieve the required output for calculating transitive closure and determining gene co-expression. One reason for this was due to the fact they lack the querying functionality to bind closely related data. Both have the functional capabilities to join data, but fall short when connecting data recursively. The MongoDB and Cassandra querying languages also can not calculate relational algebra. This restricts their ability to calculate complex queries such as transitive closure. Thus, returning the hierarchical path of a value for example, was not possible using the query language functionality alone.

It is important to note: despite the MongoDB and Cassandra systems failing to complete all of the queries, one should not presume achieving the expected result is an impossibility. There are alternative methods for calculating complex queries in these systems. MongoDB and Cassandra both provide API driver documentation. The APIs are available in popular programming languages such as C, C++, Java, Python and Ruby. Providing an API for software engineers to develop their own application is standard for many software companies; including MongoDB and Cassandra. Thus, with suitable pre-processing or sophisticated application code, both systems would be able to successfully achieve all of the queries. However, as data pre-processing and writing application code was out of scope for this project, the MongoDB and Cassandra systems were unable to complete all of the queries.

8.2 Future Work

Although this project has been successful in achieving each of its intended research objectives, it could be further developed in a number of ways:

- Using a larger and more complex dataset will further scrutinise the integrity of the database solutions. While the databases were able to cope with the volume of data, the tests did show signs of weakness.
- Evaluate a variation of the technologies in this project. The database landscape is vast and there are many more database systems which could be used as a basis for further research. For example, NewSQL which seeks to provide one with the same scalability performance of NoSQL solutions with the benefit from ACID transaction guarantees.
- Assess the usefulness of the data returned with deeper complexity. Each of the solutions evaluated often rely on third-party applications and add ons to make up for their lack of out-of-the-box functionality. Analysing these tools would enhance the justification of using a selected database system.

8.3 Critique

The project mainly focused on the capabilities of the data systems in terms of functional ability for a supplied big dataset. Initially it was my intention to scrutinise the data and analyse its availability in more depth. One of the major challenges faced in storing big data is the usefulness of the data itself. The EMAPA and EMAGE datasets are available in a number of file formats, and a closer look into how the extraction and manipulation process is undertaken would have enhanced the understanding of the dataset as a whole.

8.4 Thesis Summary

I developed four prototype data models using both a relational structured database system and leading NoSQL solutions. The data models were examined to identify their positive and negative aspects, using a real-world dataset from the biological field.

Each of the systems were able to meet the required performance specification for storing a large volume of data. While each solution was more than comfortable for storing the EMAPA

and EMAGE datasets, the load times for each varied. Neo4j was exposed to show weakness when importing the data without indexes implemented pre-load. MySQL performed the best during the load tests, which aids the argument that it can handle virtually limitless volumes of data. However, in terms of scalability, MySQL seems to struggle when dealing with multiple operations at any one time. MongoDB's ability to take nearly any data and successfully store it with little fuss, gives rise to the fact that its flexibility is one of the key selling points of the system. There was nothing surprising about the results of the load tests for the Cassandra data model. It had to load a larger volume of data and while it did not produce ground-breaking results, faired averagely.

Much of the project evaluation was focused on the querying functionality each system provides. It seems that database management systems are focused on providing one with the functional requirements to complete basic transactions. Anything more complex is expected to be done using an the APIs available and create an application to model the data in whichever way necessary.

As outlined in section 8.2, there is scope for taking this project further and in various directions. New technologies are being released constantly, providing one with the ability to collect and store their data in ways which were before not possible. It is important these technologies remain open source as progress within paid services can often stagnate and fail to keep up with the pace of development. Big data is an exciting new challenge. However, to be able to profit from this challenge, one needs to ensure they have the infrastructure and management system in place to deal with such complexities. This project has tested the boundaries of some of the most popular database solutions available. It has provided an insight into the analytical capabilities of both industry standard and new technologies. Further research will enhance the justification of using a NoSQL solution as opposed to a relational database for big data.

Bibliography

[1] KAUFMAN M. H. DUBREUIL C. BRUNE R. M. BURGER A. BALDOCK R. A. BARD, J. B. and D. R. DAVIDSON. An internet-accessible database of mouse developmental anatomy based on a systematic nomenclature. *Mechanisms of Development*, 74(1-2):111–120, 1998.

[2] P. A. BERNSTEIN and E. NEWCOMER. *Principles of transaction processing*. Morgan Kaufmann Publishers, Burlington, MA, first edition, 2009.

[3] Statistic Brain. Statistic brain | market research, rankings, financials, percentages. `http://www.statisticbrain.com/`, 2015. [Online; accessed 01-November-2015].

[4] A. BRUST. Rdbms vs. nosql: How do you pick? | zdnet. `http://www.zdnet.com/article/rdbms-vs-nosql-how-do-you-pick/`, 2013. [Online; accessed 01-November-2015].

[5] Planet Cassandra. Mysql to apache cassandra migrations. `http://www.planetcassandra.org/mysql-to-cassandra-migration/#How_is_Data_Handled`, 2013. [Online; accessed 20-March-2016].

[6] Planet Cassandra. What is apache cassandra? `http://www.planetcassandra.org/what-is-apache-cassandra/`, 2015. [Online; accessed 01-November-2015].

[7] IBM Big Data and Analytics Hub. Why only one of the 5 vs of big data really matters | the big data hub. `http://www.ibmbigdatahub.com/blog/why-only-one-5-vs-big-data-really-matters`, 2015. [Online; accessed 01-November-2015].

[8] Neo4j Graph Database. Neo4j, the world's leading graph database. http://neo4j.com/, 2015. [Online; accessed 01-November-2015].

[9] Dataintegration.info. Etl (extract-transform-load) | data integration info. http://www.dataintegration.info/etl, 2015. [Online; accessed 01-November-2015].

[10] Dev.mysql.com. Mysql :: Mysql 5.7 reference manual. https://dev.mysql.com/doc/refman/5.6/en/mysql-acid.html, year = "2016", note = "[Online; accessed 20-April-2016]".

[11] Dev.mysql.com. Mysql :: Mysql 5.7 reference manual :: 13.2.9 select syntax. http://dev.mysql.com/doc/refman/5.7/en/select.html, 2016. [Online; accessed 20-March-2016].

[12] Docs.mongodb.org. Operational factors and data models ? mongodb manual 3.2. https://docs.mongodb.org/manual/core/data-model-operations/#data-model-indexes, 2016. [Online; accessed 20-March-2016].

[13] Etltools.org. Etl - extract transform load. http://www.etltools.org/, 2015. [Online; accessed 01-November-2015].

[14] Google Refine FAQ. Google-refine.googlecode.com. https://google-refine.googlecode.com/svn-history/r1407/wiki/FAQ.wiki, 2016. [Online; accessed 22-March-2016].

[15] J. L. HARRINGTON. *Relational database design clearly explained*. Morgan Kaufmann Publishers, first edition, 2002.

[16] WICKS M. N. DAVIDSON D. R. BURGER A. RINGWALD M. HAYAMIZU, T. F. and R. A. BALDOCK. Emap/emapa ontology of mouse developmental anatomy: 2013 update. *J Biomed Sem*, 4(1):15, mar 2013.

[17] E. HEWITT. *Cassandra*. O'Reilly, first edition, 2011.

[18] MEMBREY P. HOWS, D. and E. PLUGGE. *MongoDB basics*. Apress, 2014, first edition, 2014.

[19] T. H□RDER and A. REUTER. *Principles of transaction oriented database recovery*. Fachbereich Informatik, Univ., Kaiserslautern, first edition, 1982.

[20] S. LAMBA C. KUMARDWIVEDI, A. and S. SHUKLA. Performance analysis of column oriented database vs row oriented database. *International Journal of Computer Applications*, 50:31–34, 2012.

[21] CHUI M. BROWN B. BUGHIN J. DOBBS R. ROXBURGH C. MANYIKA, J. and A. HUNG BYERS. *Big Data*. McKinsey Global Institute, first edition, 2011.

[22] Yin Zhang Victor CM Leung Min Chen, Shiwen Mao. *Big Data: Related Technologies, Challenges and Future Prospects*. Springer, 2014, 2014.

[23] WEBBER J. ROBINSON, I. and E. EIFREM. *Graph databases*. O'Reilly, 2013.

[24] P. J. SADALAGE and M. FOWLER. *NoSQL distilled*. Addison-Wesley, first edition, 2013.

[25] Smartdatacollective.com. Best 10 big data quotes of all time | smartdata collective. `http://www.smartdatacollective.com/bernardmarr/232941/top-10-big-data-quotes-all-time`, 2015. [Online; accessed 01-November-2015].

[26] Theregister.co.uk. Google's schmidt: We know what you're thinking. `http://www.theregister.co.uk/2010/10/04/google_ericisms/`, 2015. [Online; accessed 01-November-2015].

[27] Toadworld.com. Consistency models in nonrelational dbs - toad for cloud - toad for cloud databases - toad world. `http://www.toadworld.com/products/toad-for-cloud-databases/w/wiki/320.consistency-models-in-nonrelational-dbs`, 2016. [Online; accessed 18-April-2016].

[28] WATT N. ABEDRABBO T. FOX D. VUKOTIC, A. and J. PARTNER. *Neo4j in action*. Manning, 2014, 2014.

[29] M. T. □ZSU and P. VALDURIEZ. *Principles of distributed database systems*. Springer, first edition, 2011.