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

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.”[3] . 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” [6]. 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:

- MongoDB (Section 2.6.2)
- Neo4j (Section 2.6.3)
- Apache Cassandra (Section 2.6.4)
- MySQL (Section 2.6.1)

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 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.

Since the summer of 2015 I have been working as a Data Analyst for a software company in Edinburgh. In this role I have had the opportunity to use some of the indexing solutions and analytics engines discussed in this project such as MongoDB and Elastic Search. This background knowledge has contributed to my understanding of this project, as it has provided me with an already functioning knowledge of some of the concepts used.

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.
Assay	One or more *assay* comprises an experiment.
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.
Web Ontology Language (OWL)	A language representation standard for designing and authoring Web ontologies produced from the World Wide Web Consortium W3C.
OBO	A flat file format ontology representation language

Chapter 2

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" [18]. 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" [18]. 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

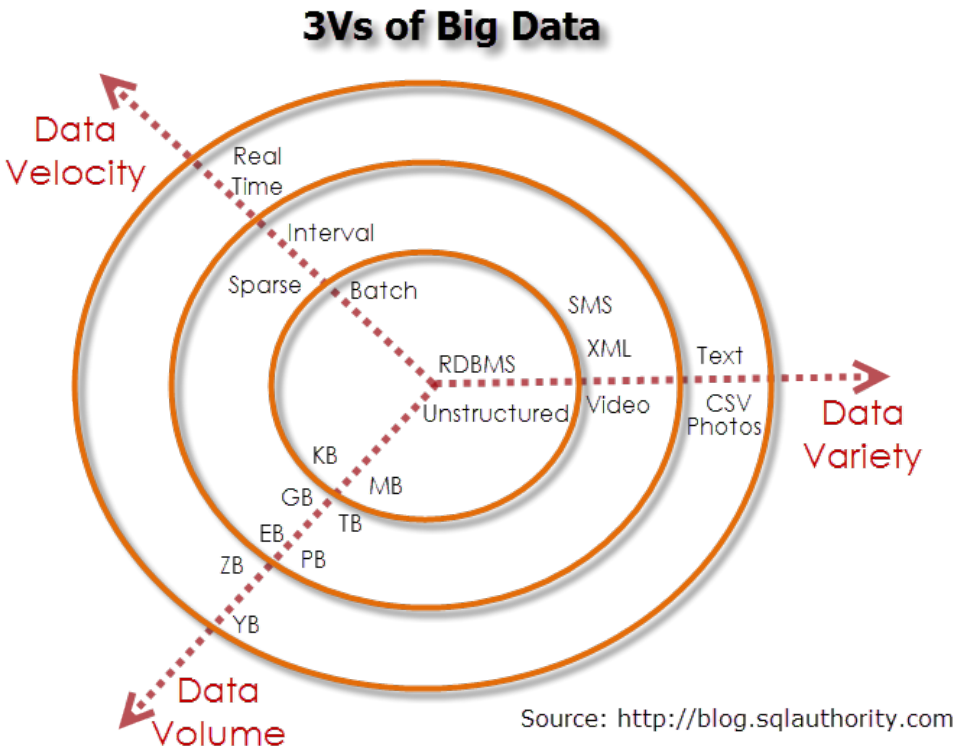

Figure 2.1: 3vs data model

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" [19].

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 [19].

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" [12]. 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 [19]. Thus the importance of establishing clear and achievable business objectives.

Year	Internet Users	World Population
1995	44,838,900	5,741,822,412
2000	413,425,190	6,082,966,429
2005	1,029,717,906	6,514,094,605
2010	2,034,259,368	6,916,183,482
2015	3,294,336,400	7,397,525,900

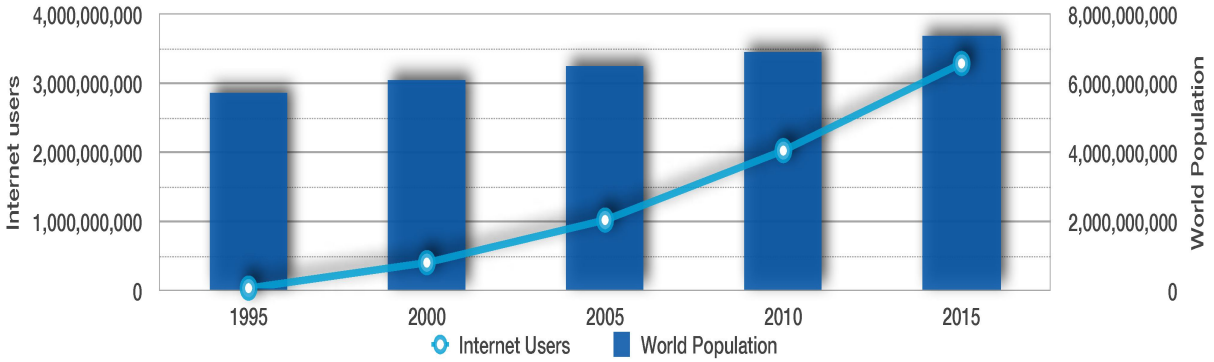

Figure 2.2: World population vs Internet users

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 [10]. 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 followed by section **??** which examines the ETL implementation methodology and its relevance to this project.

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.

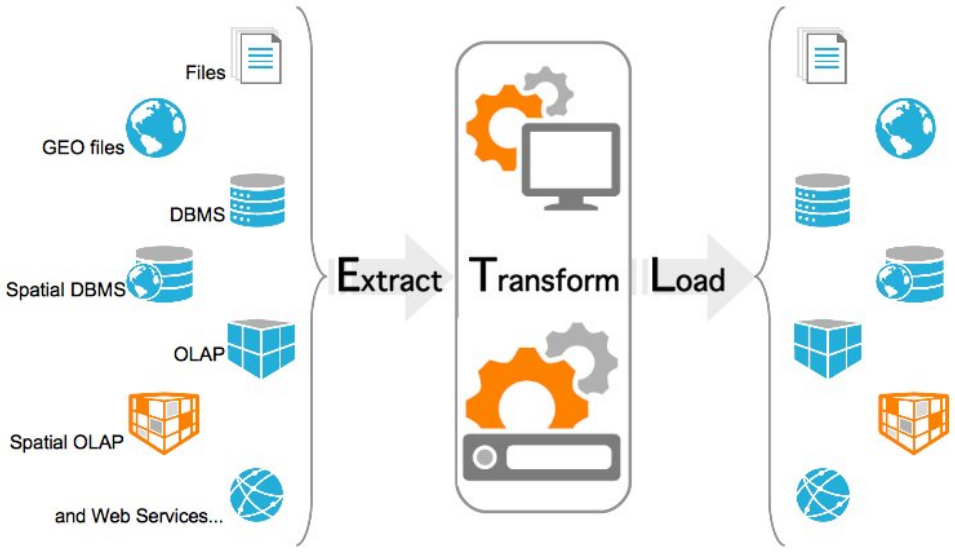

Figure 2.3: ETL process

2.2.1 Extract

Extract is the first step in the ETL procedure in which data is read from a source system, usually a database but not restricted to, 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 [5]. 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." [4].

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 ELT 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." [4]. Depending on the technique applied the load step of the process can become the most time consuming.

2.3 Knowledge Representation Languages

A Knowledge Representation (KR) language can be defined as: for any given interpretation of a sentence or string of text the KR must have the ability to effectively and unambiguously

express knowledge in a both a human and computer manageable form.

There are a number of options and possibilities for communicating data and information which range from binary representation to meta markup languages such as Extensible Markup Language (XML) for example. Markup languages which are easily read by humans such as XML which as a result of its rigid set of rules lends its self to both humans and machines. Comparatively binary notation which uses 1's and 0's to represent data, while relatively cheap in terms of computing power the ability to comprehend this notation requires a unique and specific skill set.

2.3.1 Semantic Web

The Semantic Web has two main intended outcomes. The first is about the standardised formats of data pulled from variety of sources, whereas the original Web concentrated on the interchange of documents [1]. The second outcome of the Semantic Web is the language for recording the relation between data and objects in the real world. "That allows a person, or a machine, to start off in one database, and then move through an unending set of databases which are connected not by wires but by being about the same thing." [1]. The Semantic Web is about taking information which is already on the Web and making it meaningful to computers. It is the next major step in connecting information on a global scale. Web pages are documents designed to be humanly readable. The aim of the Semantic Web is to take these documents and by extending the principles of the Web,transform them into data, readable by machines.The term was originally coined by Tim Berners-Lee, the inventor of the World Wide Web.Berners-Lee became frustrated with current state of the Web. He believed that the Web as a whole should be more intelligent about the way it serves user's needs. W3C, the organisation who define web standards and founded by Berners-Lee, describe the Semantic Web as "...a common framework that allows data to be shared and reused across application, enterprise,and community boundaries" [**?**] While search engines index the majority of the Web's content to find what a user thinks they are looking for, their ability to select pages which a user truly wants or needs is lacking. This is because we do not have a web of data [**?**]. Figure 2.4 represents the so called Semantic Web layer cake,created by Berners-Lee. It illustrates the architectural hierarchy of languages used for the SemanticWeb.

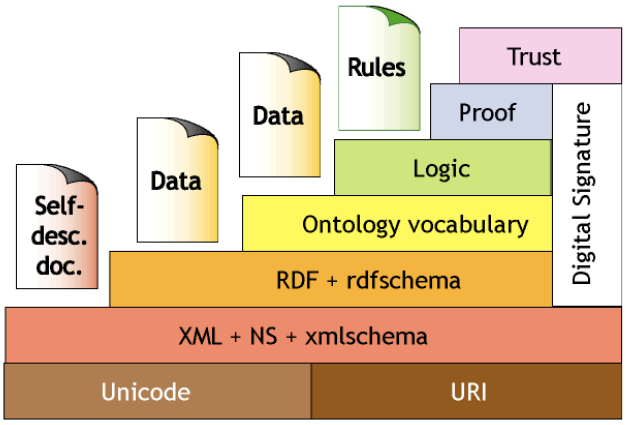

Figure 2.4: Semantic Web Layer Cake [**?**]

Each layer, from top to bottom,uses the capabilities of the layers below. The top 3 layers Trust,Proof and Logic are technologies which are not yet standardised and are just ideas that should be implemented in order to realise SemanticWeb. The middle layers Ontology Vocabulary and RDF + rdfschema are standardised technologies by W3C which enable the building of Semantic Web applications.The bottom layers contain technologies that are well known from hypertext web and that without change provide basis for the Semantic Web.There are a number of technologies for the Semantic Web. The main three are RDF, SPARQL and OWL. The Ontology Vocabulary layer of the Semantic Web stack is one of the most important. Renowned Computer Scientist, Tom Gruber's, simple definition of an ontology is often cited by authors "An ontology is an explicit specification of a conceptualization". The Web Ontology Language OWL is a language representation standard for designing and authoring Web ontologies produced by W3C. [8]. The OWL file format is designed to be used by applications required to process the content of the information and to be humanly readable. "[OWL] is intended to provide a language that can be used to describe the classes and relations between them that are inherent in Web documents and applications."[8]. The OWL languages are characterised by formal semantics and are built upon the W3C standard RDF format. While there is no universally accepted definition of an ontology, they are considered as one of the main pillars of the Semantic Web. A Vocabulary is often described as a special form of ontology or simply as a collection of URIs [**?**]. A UniformResource Identifier (URI) is is the way one identifies a point of content. For example a page of text, a video or sound clip, a still or animated image, or a program. The OWL language is characterised by formal semantics and is built upon a W3C XML standard for objects known as Resource Description Framework (RDF) [**?**]. There are many fields which have shown significant interest into OWL and RDF specifically academic, medical and commercial circles. RDF is part of the W3C family originally designed as a metadata data model. It is a general framework allowing one to describe Internet resources, for example a

website, and it's content. One of the main concepts of RDF is a RDF triple. A triple consists of:

- **Subject** - A RDF URI reference or a blank node.
- **Predicate** - A RDF URI reference.
- **Object** - A RDF URI reference, a literal or a blank node.

The aim of RDF is to enable people to share website and other descriptions with ease. Consequently giving software engineers the opportunity to develop more intelligent search engines and directories. Thus giving Web users more control of what they are browsing on the Web. A typical RDF description can include data such as resource author, date of creation/updated, subject categories and information summarising the content for audience. The way in which an RDF document is queried is predominately by using a query language called SPARQL. SPARQL is a query language designed to retrieve and manipulate data stored in a RDF structure. It became a standardised technology in 2008 by W3C and remains to be one of the key technologies of the Semantic Web. A standard SPARQL query consists of a set of triples where the subject, predicate and/or object can consist of variables [**?**]. The way in which a SPARQL query works is by matching the triples in the SPARQL query with the existing RDF triples and find solutions to the variables.

2.4 NoSQL

NoSQL is labeled as a next generation database known to most as "Not only SQL" [21]. 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. [21]

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. [21]

2.4.1 Distributed Systems and Data Transactions

Brewer's CAP Theroem

ACID Transactions

Distributed Database

A distributed database comprises of two or more data files located at different sites and servers on a computer network [23]. 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. [23] 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 startup businesses which recruit the expertise of others from various locations.

2.4.2 Database Classification

One of the first decisions to be made when selecting a database is the characteristics of the data you are looking to . [14] There are a multitude of options available with many different classifications. The following sections discuss a subset of these which are relevant to this project.

Document-Oriented Database

Document-orientated database 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 DODB specifically in MongoDB.

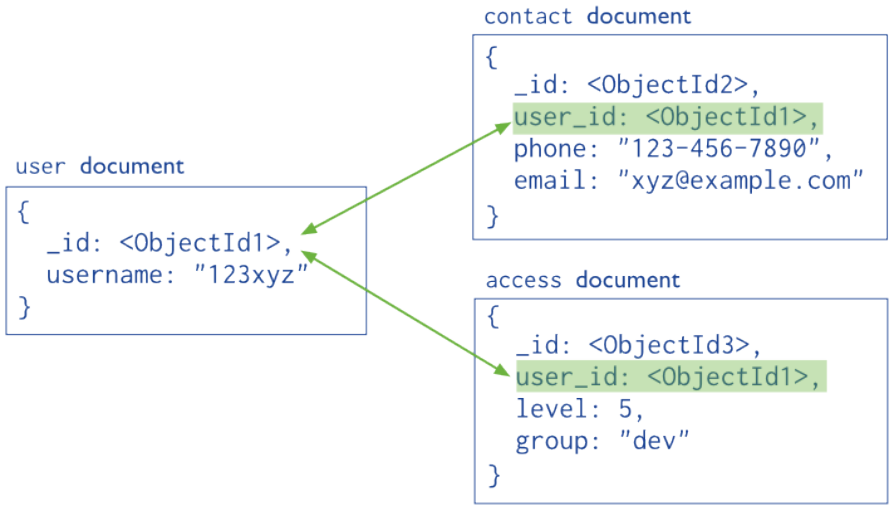

Figure 2.5: MongoDB document

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.” [20] 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. [20] 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 hashtag 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

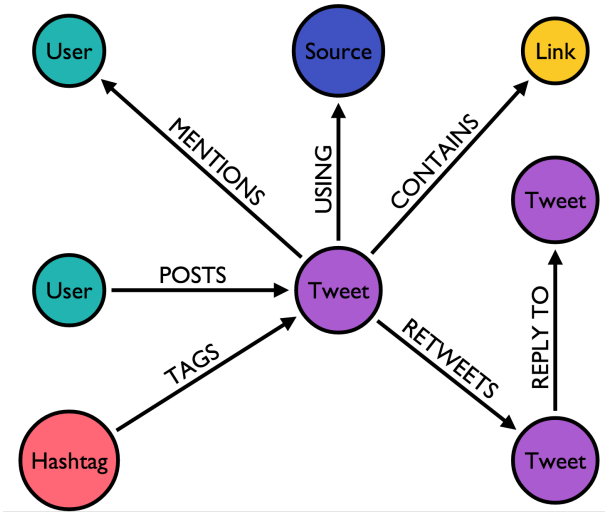

Figure 2.6: Example tweet data relationship

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. [17].

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.

2.5 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.

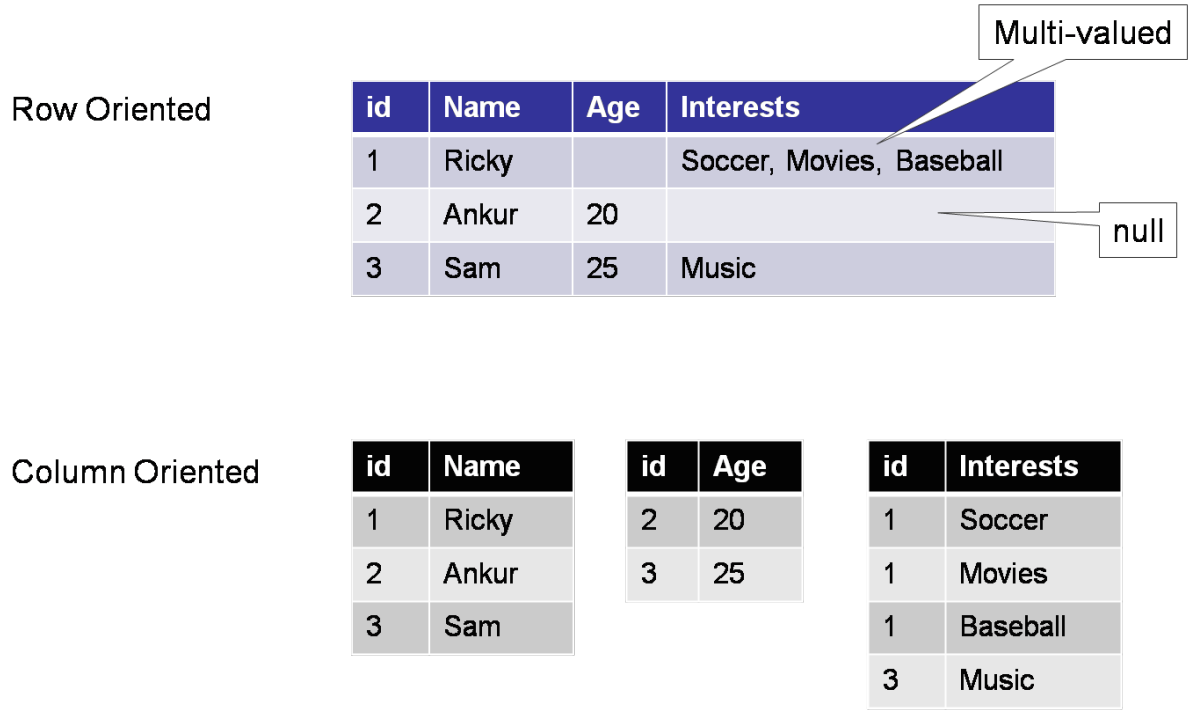

Figure 2.7: Column orientated database example

2.6 Technology Evaluation

The technologies being evaluated in this project are outlined below. **DISCUSS QUERY LANGUAGE!**

2.6.1 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 related data - which consists of columns and rows. MySQL runs as a server and allows multiple users to manage and create numerous databases.

SQL is a programming language 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.

2.6.2 MongoDB

MongoDB is an open source cross-platform DODB. The premise for using MongoDB is simplicity, speed and scalability [16]. 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. [16] The database design of MongoDB is based on the JSON file format named BSON.

A record in MongoDB is known as a document; a data structure composed of field and value pairs. The values of fields can include other documents, arrays and arrays of other documents. The key features of using MongoDB are its high performance data persistence, provide high availability and automatic scaling [16].

2.6.3 Neo4j

Neo4j is an open-source NoSQL GODB which imposes the Property Graph Model throughout its implementation. The team behind the development of Neo4j describe it as an "An intuitive approach to data problems" [7]. 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." [7]

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.

2.6.4 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."

[11]. 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." [11]. Figure 2.8 illustrates an example record stored in a Cassandra database.

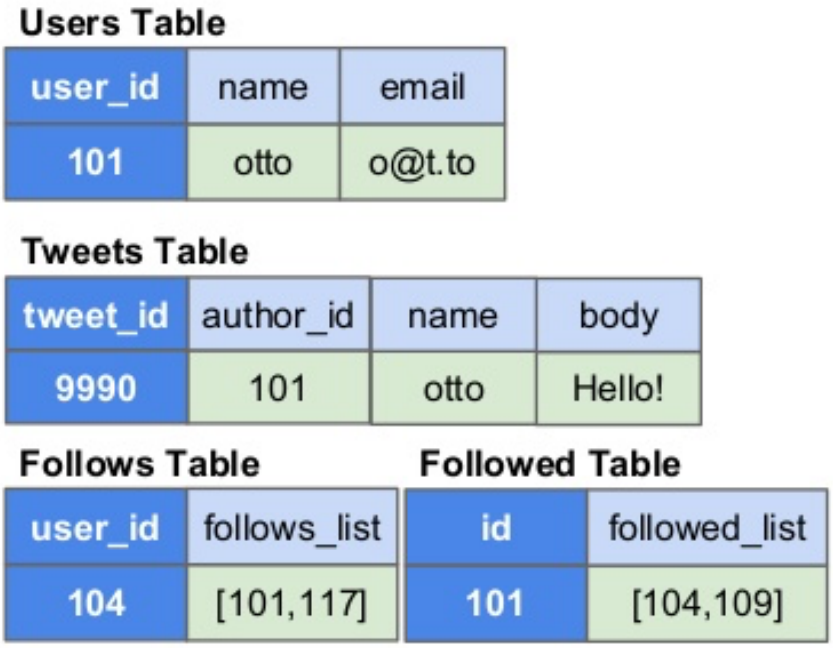

Figure 2.8: Example Cassandra record

CQL Type	Constants	Description
ascii	strings	US-ASCII character string
bigint	integers	64-bit signed long
blob	blobs	Arbitrary bytes (no validation), expressed as hexadecimal
boolean	booleans	true or false
counter	integers	Distributed counter value (64-bit long)
decimal	integers, floats	Variable-precision decimal
double	integers	64-bit IEEE-754 floating point
float	integers, floats	32-bit IEEE-754 floating point
inet	strings	IP address string in IPv4 or IPv6 format*
int	integers	32-bit signed integer
list	n/a	A collection of one or more ordered elements
map	n/a	A JSON-style array of literals: { literal : literal, literal : literal ... }
set	n/a	A collection of one or more elements
text	strings	UTF-8 encoded string
timestamp	integers, strings	Date plus time, encoded as 8 bytes since epoch
uuid	uuids	A UUID in standard UUID format
timeuuid	uuids	Type 1 UUID only (CQL 3)
varchar	strings	UTF-8 encoded string
varint	integers	Arbitrary-precision integer

Table 2.1: Cassandra data types

2.7 Data Source and Representation

The dataset used as a resource to populate the database solutions is called EMAP; a freely available anatomical ontology of the developmental stages of mouse embryos. The EMAP dataset was 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 dataset also meets the criteria which will be required to test the database solutions, explore the limitations of each database comparatively and pose insight into the overall performance of each database.

2.7.1 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 [13]. 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.[13]

The anatomy generated 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. The abbreviated term EMAP carries a certain amount of ambiguity as it is refers to the name of the project, and one of the stages in the implemented anatomy. For the purpose of this project the main anatomy to be utilised is the aggregated non stage specific Edinburgh Mouse Atlas Project Abstract (EMAPA) anatomy.

2.7.2 EMAP anatomy

The EMAP anatomy 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 anatomy 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 [13]. 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" [13].

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 [13]. 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.

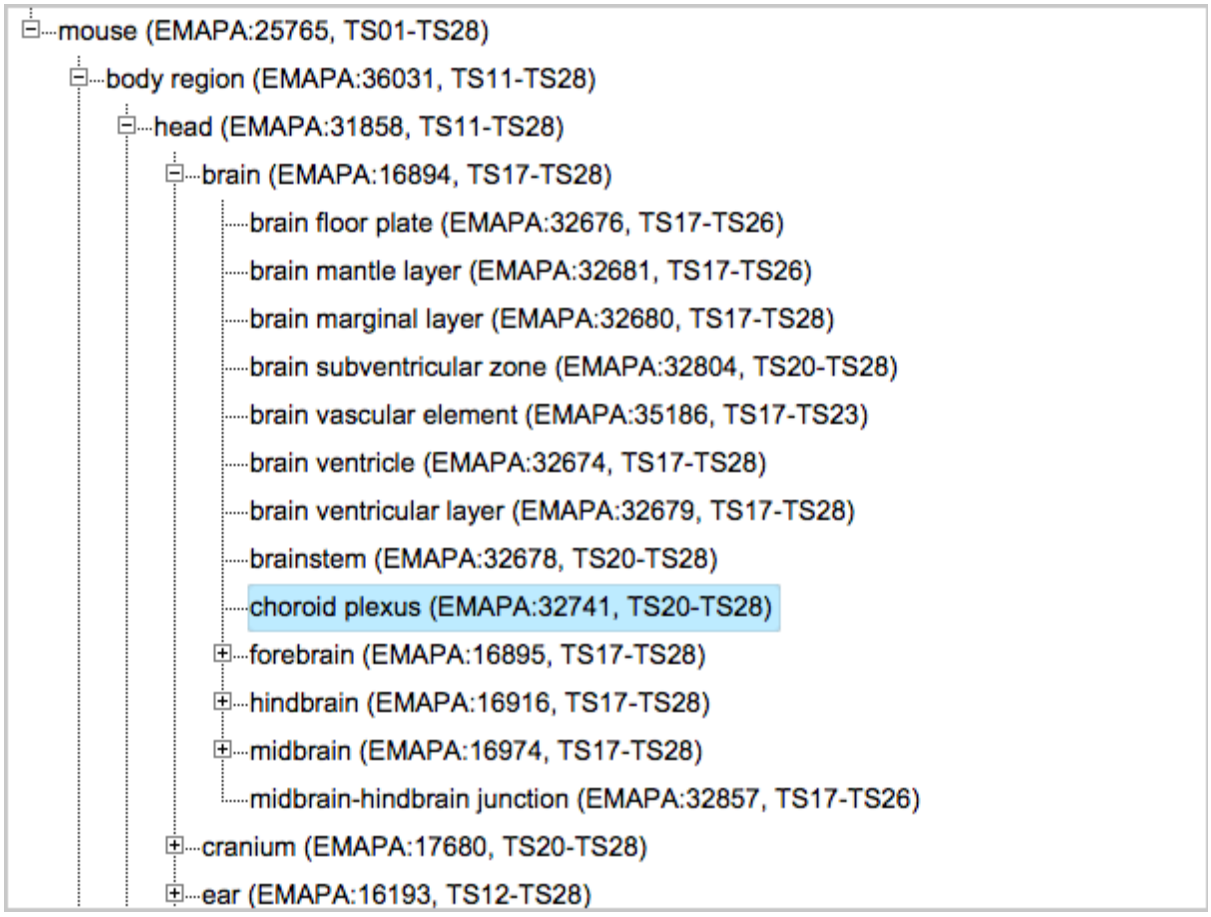

Figure 2.9: EMAPA data structure

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. [13] Thus the EMAPA anatomy will be the main data source for this work.

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 [13]. The EMAPA ontology discussed below in section 2.7.3 is now considered the primary data source thus the EMAP dataset is available in a combined EMAP and EMAPA standard ontological format developed by the Open Biological Ontologies (OBO) consortium (Section **??**).

2.7.3 EMAPA anatomy

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 [13].

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. [13] 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 easily comprehensible.

The EMAPA ontology is available in a standard ontological format developed by the Open Biological Ontologies (OBO) consortium (Section **??**) and is also available in a Web Ontology Language (OWL); a standard produced from W3C discussed in section **??**.This enhanced version of the ontology EMAPA, is now considered to be the primary EMAP anatomy ontology thus will be the main source for the work on this project.

2.7.4 EMAGE anatomy

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 [13].

"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" [13]

The EMAGE anatomy provides an alternative view of the EMAP anatomical ontology. The main data source for this project will be EMAPA however should the need for a data set which holds larger data then the EMAGE data set will be used.

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

The first objective of the project is to investigate the strengths and weaknesses of the 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 dataset. Competency questions will be devised which will return a simple yes or no answer. Example competency questions can be found at section 3.1.1. These questions will be translated in to the relevant query language for each of the datasets. The competency questions will range from a beginner level where it will be expected that all database solutions will be able to return the intended value to an advanced level. The

difficulty of writing these queries will also be recorded and used to analyse the performance of the database solution. The time taken for the query to run will also be recorded and used for analysis

1. Is this query possible in database X?

   For example the first query in the EMAPA competency question table asks : "For structure X, find all the ancestors." If this query is possible in a given database, move on to the next query in the list and continue until failure. Once this step has been complete for all database solutions, by way of comparison evaluate the strength of each database.

Objective two of the project is to 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.

1. If the query is possible in a given database how rich a return of the dataset can it provide.
2. Does any database offer any querying capabilities/functionality which compared with another which aids the query in any way.

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 :

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

3.1.1 Data Source Competency Questions

EMAPA	**EMAGE**
For structure X, find all the ancestors.	Where is gene X expressed?
For structure X, find all the decedents.	What is expressed in structure X?
For structure X, find all structures in the same group (this only makes sense in limited situations, e.g., find all "hair").	Which genes are expressed in both structures X and Y? (This is so-called co-expression).
Find all the structures in Theiler Stage X.	Which genes are most commonly co-expressed? (This is getting towards BI and is possibly out with the scope of your thesis, but it does no harm to discuss it.
Find all the structures in Theiler Stages X to Y (e.g., 17-19).	
What stages does structure X appear in?	
Given search term X return the "best match" structure (e.g., if Isearch on "heart" I would expect the output to be heart, heart atrium, heart septum heart mesentery)	
Given an EMAPA ID return the name of the structure.	
Given a name return the EMAPA ID.	

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 **??**

4.1 Cleansing the data

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

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

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.” [**?**]. 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 - data model design

As discussed in section 2.6.1 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. It aids in better, faster, stronger searches as it entails fewer entities to scan in comparison with the earlier searches based on mixed entities. Data integrity is improved through database normalisation as it splits all the data into individual entities yet building strong linkages with the related data. The below description provides and 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.
  - 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.
  - Many to One relationship with the Sources table. While one assay can only have one source, many assays can have the same source.
- **Genes** : This table contains the accession number and symbol of each gene.
  - The Genes table does not reference any other table.
- **Publications** : This table contains the accession, title and every author of each assay publication.
  - Many to One relationship with the Assays table as there can be many publications for one assay.

- **Sources** : This table contains the source of the assay.
  - The Sources table does not reference any other table.
- **Specimens** : This table contains the ID, strain and type of each specimen. The type field refers to whether the assay is a section, wholemount, sectioned wholemount or unknown.
  - One to One relationship with the Assays table as one assay has one specimen.
  - Many to One relationship with the Stages table as many specimens can have the same Stage.
- **Stages** : This table contains the theiler stage and number of days post conception (dpc) of each assay, specimen and anatomy.
  - Many to One relationship with the Assays table as one assay has can have multiple stages.
- **TextAnnotations** : This table contains the structure and strength of each assay.
  - Many to One relationship with the Assays table as one assay can have multiple text annotations.
  - One to Many relationship with the Genes table as multiple text annotations can the same gene.
  - Many to one relationship with the AnatomyStructures table as many text annotations has one structure.

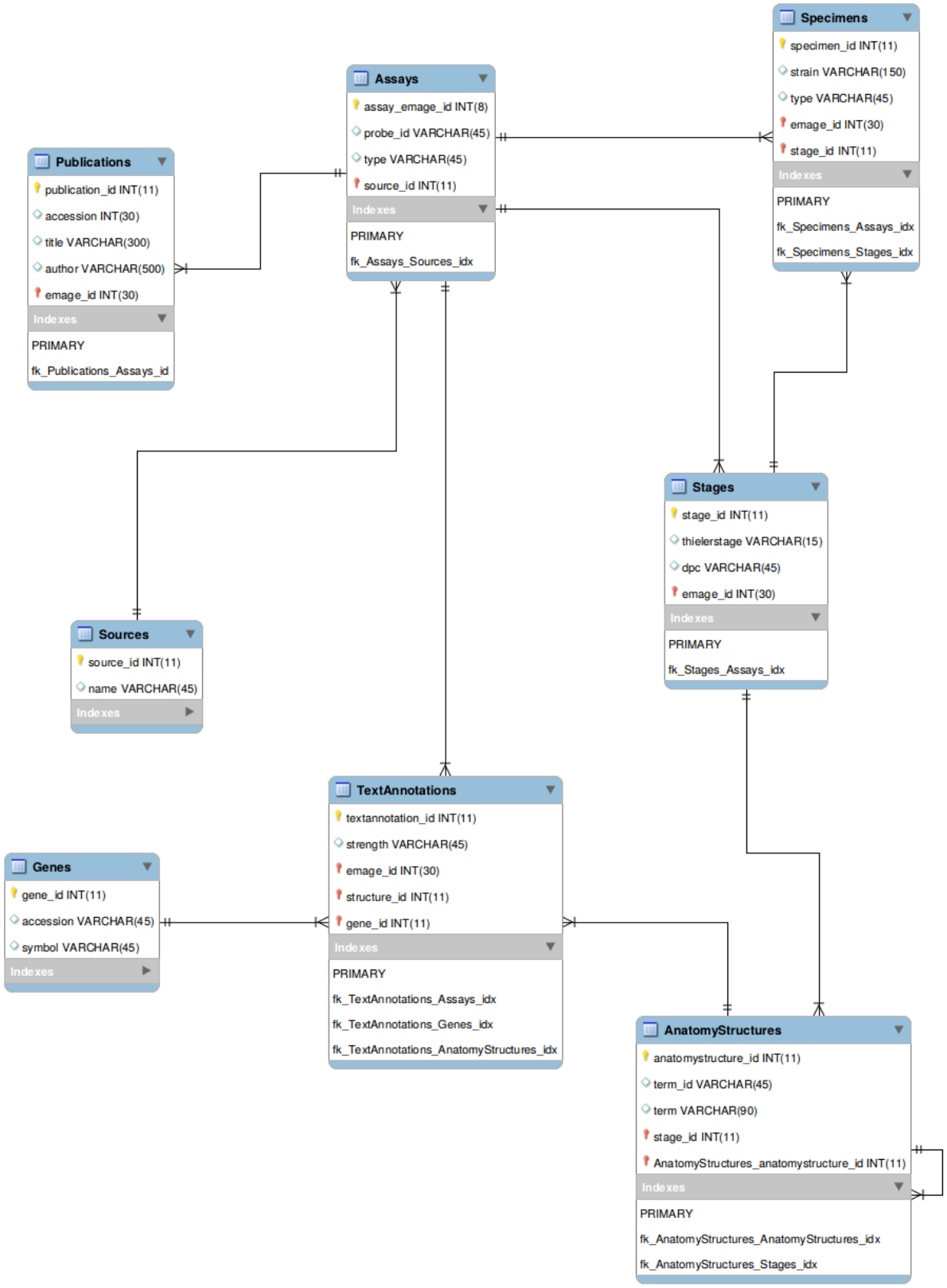

Figure 4.1: MySQL ER table diagram

4.2.2 MongoDB - data model design

As discussed in section 2.6.2, MongoDB is a homogeneous, schema-less, NoSQL document store database. There are no formal relations between the data which makes modelling the database a little more challenging; especially with data which is so closely bound as the EMAGE dataset.

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

The MongoDB implementation was the second prototype data model I created for this project. I followed the same structural process 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 should the data be connected. As there were a few options I decided to explore all of them to fully comprehend the pros/cons of each.

My initial design was based around using multiple collections to store the various aspects of the data. The design followed the MySQL model, with 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 would be to have all of the data in the one 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 - data model design

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 2.6.3.

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, Neo4j has an edge, which connects the nodes.

As with MongoDB and indeed many NoSQL system, there is no formal way to diagrammatically convey the database "schema". To illustrate the Neo4j data model, I have created 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. The data within each node is:

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

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

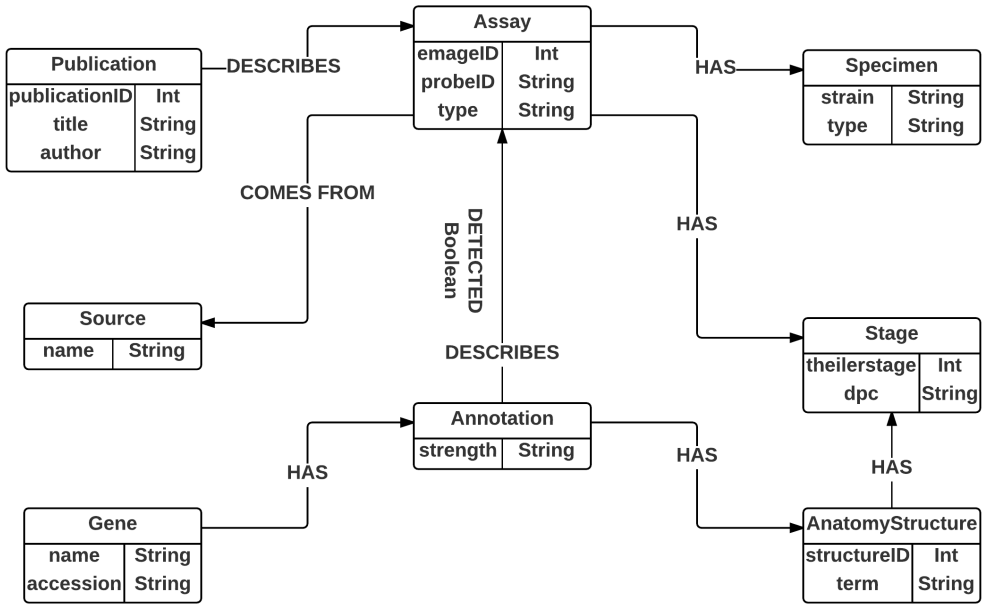

Figure 4.2: Neo4j graph model ER diagram

4.2.4 Apache Cassandra - data model design

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

The Apache Cassandra data model was perhaps the most difficult of all the database system designs to create. This was due to a number of reasons. One of which was, that I had no previous experience of using a Cassandra database, and therefore had to learn a completely new style of working. Secondly, as discussed in section 2.6.4, one of the main aspects of Cassandra family columns and tables is that they do not accept joins. Resulting in tables being created, aiming to satisfy any potential queries which may be imposed on the database. The repercussions of this concept is 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. The queries outlined in the evaluation strategy of this document, chapter **??**, provides the queries in which I have modelled the Cassandra data model around.

There are 7 tables; Assays, AssayByStage, AssayBySpecimen, Publications, StructureByGene, StructureByStage and TextAnnotations. The naming convention I adopted for the Cassandra data model is based on describing each table by their partition key grouping (for a full description of partition keys and Cassandra concepts, see section 2.6.4). For example, AssayByStage is a table consisting of each Assay partitioned on the Stage value. Figure **??** illustrates each Cassandra table.

4.3 Discussion - data model design

This section provides a general summation of the design process as a whole for each solution. The overall intention of this section, is to provide the reader with an in-depth understanding of the challenges faced and the overall advantages and disadvantages of the data modelling procedures, for each database management system.

The first hurdle one must overcome is to fully comprehend what data they are modelling. Getting to know your dataset, will allow you to develop the optimum system architecture. Having a clear and coherent understanding of how the data is joined and the relationships are formed in your dataset is imperative to a successful outcome. Once an application developer knows **what** they are developing they can accomplish the **how**. 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. 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 your application sits upon a high performing, durable database system.

The first obstacle I faced was that my knowledge of the EMAP dataset was lacking. I had no prior experience in the biological field and many of the concepts which accompany this dataset took me some time to understand. Therefore being able to logically model a database around an unknown entity was difficult. After discussing the dataset in detail with my supervisors and doing some background research, I was able to surpass this barrier which was slowing my progress in the data modelling phase of the project.

Once the initial hurdle of gaining experience and understanding the theory behind the dataset was overcome, 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 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 2.7 the EMAP is ever evolving. There has been many iterations and enhancements made from when the project first began until now. Therefore it is highly likely that, while the current EMAGE structure may be considered the leading source, this will change in the future. The result of this would be that the EMAGE data model would require to be restructured. As this is only a hypothesis we can only speculate as to how the data model would need to be changed. 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." [**?**]. If we just take the last proposed example of "other forms of metadata", in a MySQL data model, depending on the data, this may require an additional table with a join, an index and an ID. The ID would then need to be included into the corresponding table 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 data model, should further information become available and require to be included into the database, all that is required is 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 that of any of the other solutions in this project. MongoDB imposes a flexible schema, meaning the collections do not enforce a specific structure. This allows one to insert data at their own will without having to conform to a strict schema. It also permits the matching of documents to objects and entities. Because of this, designing the data model was easy. The document structure mirrors that of the EMAGE flat files. Therefore mapping the CSV headings to the Mongo 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 been forced 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 the API's 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 and and the joins and relationships is clear and extremely easy to follow. There is no complicated joins or key relations to comprehend.

The way in which the Cassandra data model was constructed seemed to me to be rather backwards. Data duplication and de-normalisation was something which was the opposite to what I had been used to for many years 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. Flipping the entire process

on its head, while refreshing and enjoyable to learn, meant the design stage of the modelling processes was prolonged. This resulted in the overall design of the database being a trial and error exercise. Thus again increasing the time taken to model the system. To the naked eye 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, as with many processes I often find the best way to learn is to start playing with 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 their own procedure for instantiating a new collection, table, node or column family. This chapter discusses the methods and strategies I imposed to create the database solution designs; with a focus on any challenges faced in doing so. Chapter 6 evaluates the process undertaken to physically load the data into the systems.

5.1 Creating database systems

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

5.1.1 MySQL - schema implementation

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

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

Code snippet 5.1: Creation of Assays table in MySQL.

As you can see in lines 4 - 11, the creation of each column takes the form of; column name, data type with length and the default attributes and values i.e. null or not null. The character set for each table, by default was set to utf8. You will notice here that the key values were not in fact instantiated at time of creation. This was as a result of an experiment to evaluate the affect the exclusion of index keys and constraints has at time of data load on each database. The full discussion and outcome of this experiment can be found in chapter 7.2.

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

The process described was repeated for each of the tables in the devised data model. Creating multiple tables and joining them together in MySQL is relatively straightforward. While the formality of definitively expressing each term and its data type then stipulating the index keys and 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 - schema implementation

Creating a document inside a MongoDB collection (the equivalent to a MySQL table) is done by inserting field and value pairs. Each time a new field and value pair is inserted into a collection a new document is created. By default, each document in a collection is provided with a unique ID which has an object data type. ObjectIds are small, fast to generate, and ordered. These values consists of 12-bytes, where the first four bytes are a timestamp that reflect the ObjectId’s creation [**?**]. A unique ID can also be manually created for each document, if 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 an extremely flexible data store. It accepts multiple different data types, from the standard string, double and boolean values to the more complex regular expressions and even Javascript code. By default, any value without a type 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 into how I created the MongoDB documents can be found in section 6.1.2.

An important design decision which has to be taken prior to model implementation is, whether to create multiple collections or embed data within a document. A comprehensive evaluation of the advantages and disadvantages is discussed in section 4.2.2. The key difference between the two approaches is, that within a multiple collection database, one is required to write multiple queries and also undertake client side application level processing 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 [**?**]. Whereas embedded documents are recommended when you want the entire document all at once [**?**].

As discussed in section 4.2.2 and illustrated in code snippet 5.2, the MongoDB data model

I have designed uses the embedded document approach. What do I mean when I say embedded? A good way of thinking of the embedded data concept is, a document in a document. For example, if we look at this concept 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 with an embedded document. The field named “publication” has 2 embedded (child) values which provide more information in the same collection about the same document.

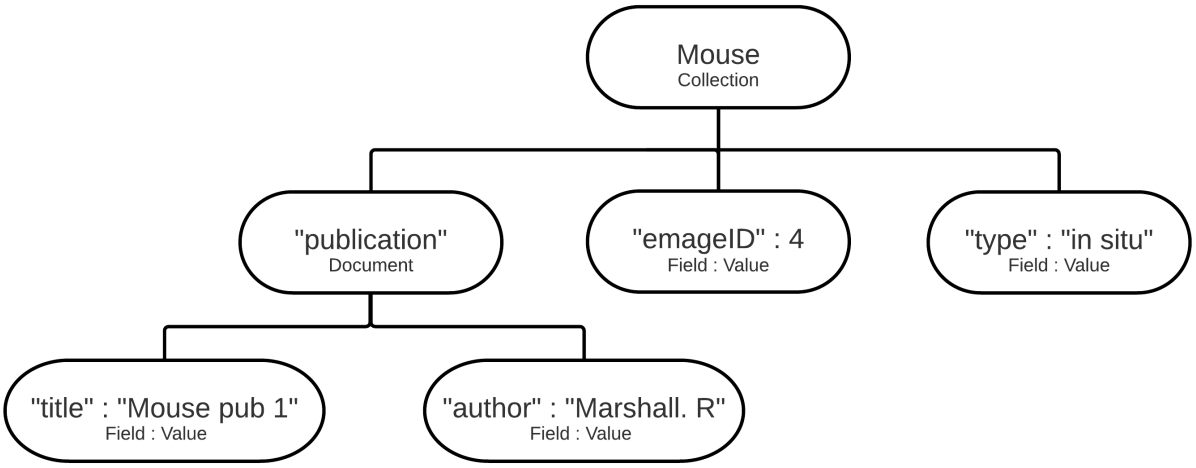

Figure 5.1: Example of a MongoDB embedded document in a hierarchical structure.

Lines 6-34 of code snippet 5.2 represents an example of data, embedded in a document. Looking specifically at lines 14-20, we have a value namely “publication” which has embedded values of “publicationID”, “author” and “title”. Thus allowing direct querying of additional data, as opposed to messy collection joins.

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

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

5.1.3 Neo4j - schema implementation

To implement a Neo4j structure we use a query language namely Cypher Query Language (CQL). Cypher, is a declarative graph query language that allows for expressive and efficient querying and updating of a graph store [**?**]. CQL was designed to be as user friendly as possible for both programmers and operations professionals alike [**?**]. The structure of CQL is based upon SQL and shares many of its attributes. CQL queries are built using various clauses. For a detailed insight into CQL, see section 2.6.3.

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 instantiated with an ID value. Each and every node in the database has a unique numerical ID, which is incremented from the first node to the last inserted. This value can not be changed. The ID value can be used as a standalone statement or within a query clause by using the "ID" clause function.

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 instantiation, I created a Cypher file and imported the dataset simultaneously. The implementation of this is discussed in section **??**, which also describes how to load data into Neo4j 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 **??**.

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

5.1.4 Apache Cassandra - schema implementation

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. A full description of all data types in Cassandra can be found in table 2.1.

```
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 Implementation Discussion

One of the major *pros* 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, with the former providing data modelling, SQL development, and comprehensive administration tools for server configuration and user administration [**?**]. 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 one would prefer to manually write the script to create the data model, while potentially tedious, 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 mainly on two reasons. Firstly, I wanted to fully understand and appreciate the data modelling complexities of each database system. Secondly, I felt that using a third party tool for implementation would not result in a comprehensive evaluation of each database system. Therefore 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 copy 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 what 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. All one has to do is run the relevant commands on the 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. I feel 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 often become the most time consuming phase of the pipeline. This chapter describes the implemented procedures and examines the functionality each solution provides to complete the data load process.

6.1 Put the data in databases

The final stage in the data modelling process was to import the EMAGE 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, whether it be 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 **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 however not the most efficient process available for inserting datasets of large volume.

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

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

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

3. Update

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

```
import csv

from pymongo import MongoClient

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

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

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

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

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

6.1.3 Neo4j

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

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

The main structure of the query and the commands used are similar for the two approaches. However, to load data in via a CSV one requires two additional lines, these are represented in lines 5 and 6 of code snippet 6.4. Line 5, "USING PERIODIC COMMIT" is used when loading large amounts of data in a single cypher query. This is because loading large volumes of data within a single query runs the risk of failing due to running out of memory. Thus including this function prevents the query failing for this reason. However, it will also break transactional isolation and should only be used where needed 2.6.3. Line 6 of code snippet 6.4 simply loads the file found at the stipulated location and assigns it to a variable name, "row" in this case.

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

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

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

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

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

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

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

regarding simultaneous processes running on the computer, affecting 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.

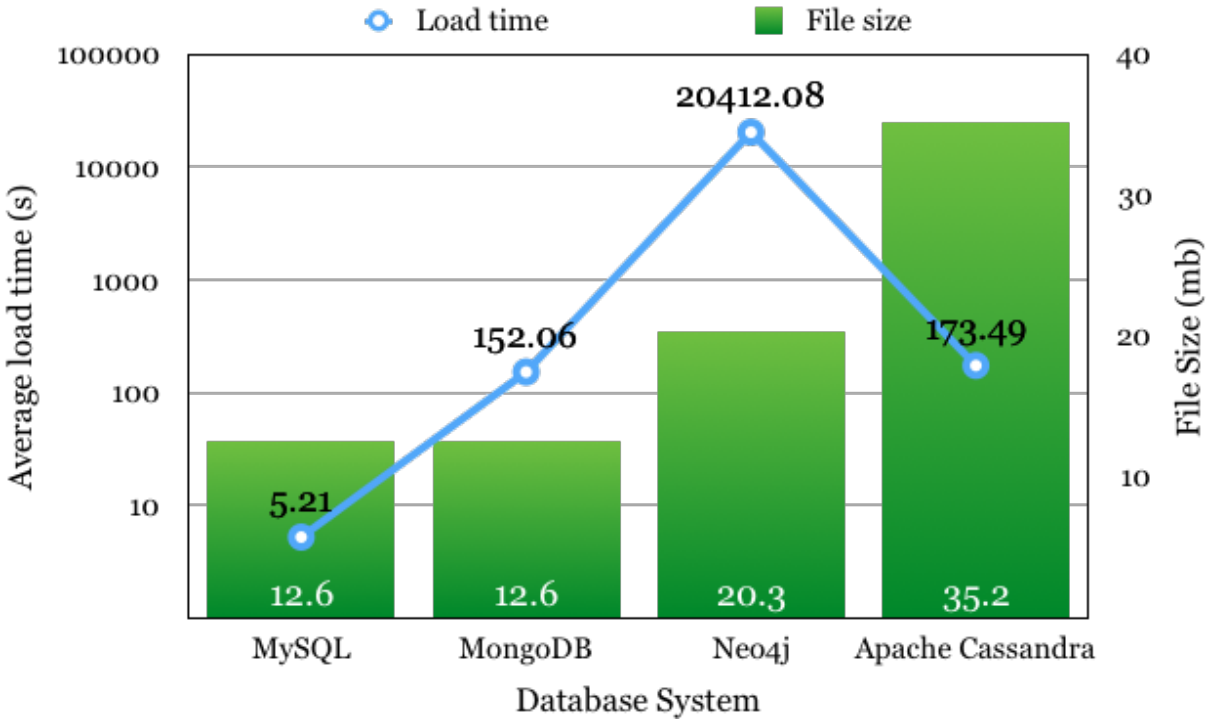

Figure 6.1: Chart illustrating the variance in load times of non-indexed systems vs the file size

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 a sort of 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. Thus increasing the amount of overhead for the performance of write operations.
- Neo4j model is struggling to process the high number of reads compound 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 analyse and examine the results closely allowing me 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. This bar chart shows the

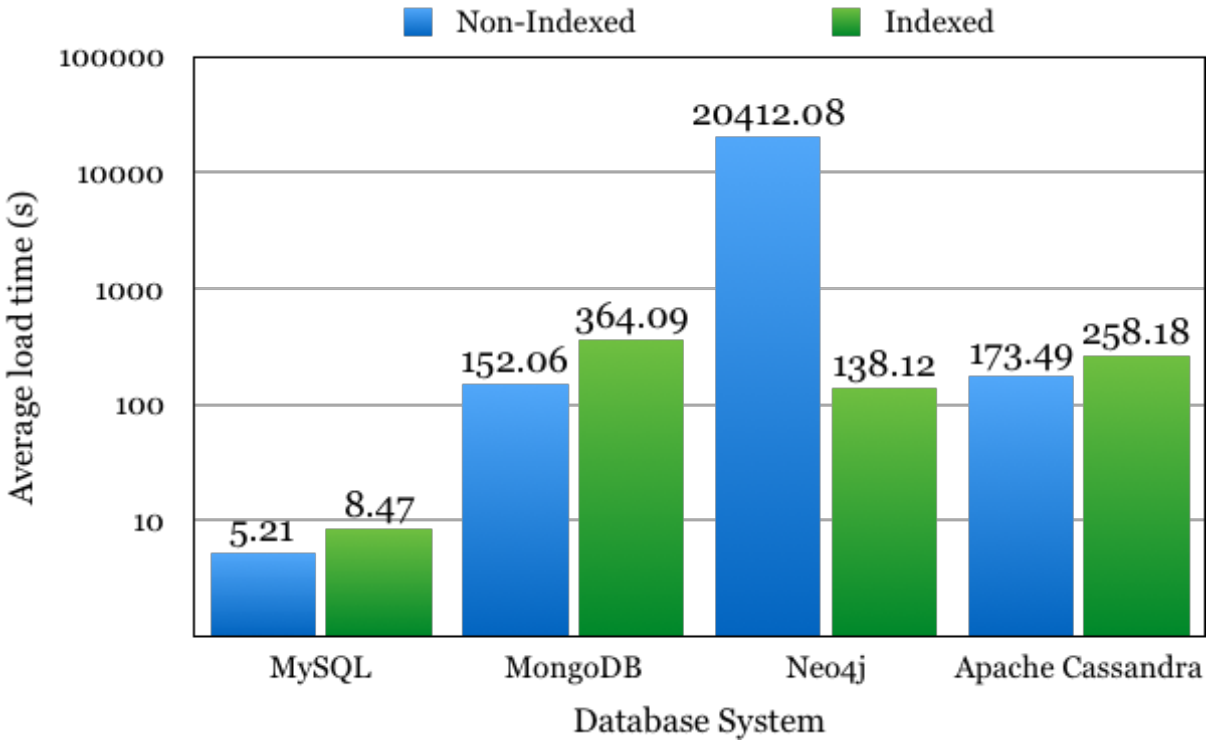

Figure 6.2: Chart illustrating the variance in load times of non-indexed vs indexed systems

affect implementing indexes has on the load time of data models. These findings posed 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 just to purely 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. In contrast, 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." [**?**]. 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 [**?**]. 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. This 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 exact same procedure and query was ran 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 to have 4 separate parts. Lines 2 and 3 is where the data is pointed to on the machine. The MATCH statements in lines 5, 6 and 7 allows 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/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 affect 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 Evaluation, Results and Discussion

To analyse the functionality of the database systems, I devised a number of queries with ranging difficulties. The aim of the queries was to identify any limitations of the systems, and to discover the efficacy of the systems. Chapter 3 provides a detailed explanation of the reasoning behind the queries and this chapter discusses the output and results of running the queries in the various query languages. This chapter 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 actioning the evaluation strategy. Also 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, section 7.2.

7.1 Query Output

The screenshots, tables, graphs and charts illustrated in this chapter are based 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 a rating characterisation. Each query has a rating with a range of 1 to 5, and is to be used as a guide to aid the reader into understanding **1.** General complexity of the query. **2.** Fruitfulness of the data returned. **3.** Level of expectancy that the system will be able to accomplish the query (on a scale of 1 expected, to 5 unexpected).

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

Table 7.1: Competency queries for each database system.

Query 1 - All structures at Theiler Stage X

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

- **MySQL**

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

- **MongoDB**

```
db.emage.find({ "stage.theiler_stage": "4" }).pretty()
```

- **Neo4j**

```
MATCH (struct:AnatomyStructure)-[r]->(stage:Stage)
WHERE stage.theilerStage = 4
RETURN struct, r, stage;
```

- **Cassandra**

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

Query 2 - All structures between Theiler Stage X and Y

The following code snippets represent the queries written for competency question 2.

- **MySQL**

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

- **MongoDB**

```
db.emage.find({$and: [ { "stage.theilerstage": {$gte : 4 }},{ "stage.
    theilerstage": {$lte : 7 }}, {"textannotation.strength" : "detected"}]}).
    pretty()
```

- **Neo4j**

```
MATCH (struct:AnatomyStructure)-[r]->(stage:Stage)
WHERE stage.theilerStage >= 4 AND stage.theilerStage <=7
RETURN struct, r, stage;
```

- **Cassandra**

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

Query 3 - Where is Gene X experessed?

The following code snippets represent the queries written for competency question 3.

- **MySQL**

```
SELECT t1.name AS GeneID, t2.emage_id AS EMAGE_ID, t3.accession AS StructureID
FROM Genes AS t1
INNER JOIN TextAnnotations AS t2
ON t1.id = t2.gene_id
INNER JOIN AnatomyStructures AS t3
ON t2.structure_id = t3.id
WHERE t2.detected = 1
AND t1.name = 'Hoxb13'
ORDER BY 2,3;
```

- **MongoDB**

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

- **Neo4j**

```
MATCH (g:Gene)<-[]-(TextAnnotation)-[]->(a:AnatomyStructure)
WHERE g.name = 'Hoxb13'
RETURN distinct g.name AS Name,g.accession AS ID, a.accession AS  StructureID,
     a.term AS Term
ORDER BY a.accession ASC
```

- **Cassandra**

```
SELECT *
FROM structurebygene1
WHERE genename = 'Hoxb13'
AND detected = true
ALLOW FILTERING;
```

Query 4 - What is expressed in structure X?

The following code snippets represent the queries written for competency question 4.

- **MySQL**

```
SELECT t1.name AS GeneName, t2.accession AS StructureID, t2.term AS TermName,
    t4.theilerstage AS TheilerStage, t3.emage_id AS EMAGE_ID
FROM Genes AS t1
INNER JOIN emage.TextAnnotations AS t3
ON t1.id = t3.gene_id
INNER JOIN emage.AnatomyStructures AS t2
ON t3.structure_id = t2.id
INNER JOIN Stages AS t4
ON t2.stage_id = t4.id
INNER JOIN Assays AS t5
ON t3.emage_id = t5.emage_id
WHERE t3.detected = 1
AND t2.accession = 17451
ORDER BY 5;
```

- **MongoDB**

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

- **Neo4j**

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

- **Cassandra**

```
SELECT *
FROM textannotations1
WHERE structureid = 17451
AND detected = true
ALLOW FILTERING;
```

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

The following code snippets represent the queries written for competency question 5.

- **MySQL**

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

- **Neo4j**

```
MATCH (n:Gene)<-[]-(t:TextAnnotation)-[]->(a:AnatomyStructure)
with count (distinct (a.accession)) as c, t as ta, a as an, n as ge
WHERE ta.detected = 1 and an.accession = 16062 or an.accession = 16069 and c >
    1
return distinct ge.name
order by ge.name
```

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

The following code snippets represent the queries written for competency question 6.

- **MySQL**

```
SELECT t1.name AS GeneName, COUNT(DISTINCT t2.accession) AS Co_Expressed_Count
FROM Genes AS t1
INNER JOIN TextAnnotations AS t3
ON t1.id = t3.gene_id
INNER JOIN AnatomyStructures AS t2
ON t2.id = t3.structure_id
WHERE t3.detected = 1
GROUP BY 1
HAVING COUNT(DISTINCT t2.accession) > 1
ORDER BY 2 DESC
LIMIT 5;
```

Query 7 - Calculate transitive closure

The following code snippets represent the queries written for competency question 7.

- **MySQL**

```
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
```

7.2 Discussion

Chapter 8

Conclusion

Bibliography

[1] Introduction to the semantic web, nov 2014.

[2] Apache jena -, nov 2015.

[3] Best 10 big data quotes of all time | smartdata collective, nov 2015.

[4] Etl - extract transform load, nov 2015.

[5] Etl (extract-transform-load) | data integration info, nov 2015.

[6] Google's schmidt: We know what you're thinking, nov 2015.

[7] Neo4j, the world's leading graph database, nov 2015.

[8] Owl 2 web ontology language document overview (second edition), nov 2015.

[9] Rdf - semantic web standards, nov 2015.

[10] Statistic brain | market research, rankings, financials, percentages, nov 2015.

[11] What is apache cassandra?, nov 2015.

[12] Why only one of the 5 vs of big data really matters | the big data hub, nov 2015.

[13] 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.

[14] A. BRUST. Rdbms vs. nosql: How do you pick? | zdnet, nov 2013.

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

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

[17] 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.

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

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

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

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

[22] AITKEN S. MOREIRA D. A. MUNGALL C. SEQUEDA J. SHAH N. H. TIRMIZI, S. and D. P. MIRANKER. Mapping between the obo and owl ontology languages. *J Biomed Sem*, 2:S3, 2011.

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