Novel Database Design for Extreme Scale Corpus Analysis

A thesis submitted to Lancaster University

for the degree of Ph.D. in Computer Science

Matthew Parry Coole

January 2021

Abstract

This thesis presents the patterns and methods uncovered in the development of a new scalable corpus database management system, LexiDB, which can handle the ever-growing size of modern corpus datasets. Initially, an exploration of existing corpus data systems is conducted which examines their usage in corpus linguistics as well as their underlying architectures. From this survey, it is identified that existing systems are designed primarily to be vertically scalable (i.e. scalable through the usage of bigger, better and faster hardware). This motivates a wider examination of modern distributable database management systems and information retrieval techniques used for indexing and retrieval. These techniques are modified and adapted into an architecture that can be horizontally scaled to handle ever bigger corpora. Based on this architecture several new methods for querying and retrieval that improve upon existing techniques are proposed as modern approaches to query extremely large annotated text collections for corpus analysis. The effectiveness of these techniques and the scalability of the architecture is evaluated where it is demonstrated that the architecture is comparably scalable to two modern No-SQL database management systems and outperforms existing corpus data systems in token level pattern querying whilst still supporting character level pattern matching.

Declaration

I declare that the work presented in this thesis is my own work. The material has not been submitted, in whole or in part, for a degree at any other university.

[illegible]

Matthew Parry Coole

List of Papers

The following papers have been published during the period of study. Extracts from these papers make up elements (in part) of the thesis chapters noted below.

- Coole, Matthew, Paul Rayson, and John Mariani. "Scaling Out for Extreme Scale Corpus Data." 2015 IEEE International Conference on Big Data (Big Data). IEEE, 2015. *Chapter 5.*
- Coole, Matthew, Paul Rayson, and John Mariani. "LexiDB: A Scalable Corpus Database Management System." 2016 IEEE International Conference on Big Data (Big Data). IEEE, 2016. *Chapters 3 and 5.*
- Coole, Matthew, Paul Rayson, and John Mariani. "LexiDB: Patterns Methods for Corpus Linguistic Database Management." Proceedings of The 12th Language Resources and Evaluation Conference. 2020. *Chapters 4 and 5.*
- Coole, Matthew, Paul Rayson, and John Mariani. "Unfinished Business: Construction and Maintenance of a Semantically Tagged Historical Parliamentary Corpus, UK Hansard From 1803 to the Present Day." Proceedings of the Second ParlaCLARIN Workshop. 2020. *Chapter 5.*

Contents

List of Figures **xi**

1 Introduction **1**

1.1 Overview 1

1.2 Area of Study 2

1.3 Motivation 3

1.4 Research Questions 4

1.5 Thesis Structure 5

2 Literature Review **7**

2.1 Background in Corpus Linguistics 7

2.1.1 Corpus Queries 9

2.2 Existing Corpus Data Systems and their Query Languages . . . 13

2.2.1 Overview of Corpus Data Systems 13

2.2.2 Corpus Data Systems Architecture 18

2.3 Proposed Systems & Related Work 20

2.4 Database Management Systems and IR systems 23

2.4.1 Indexing 23

2.4.2 Querying Corpora with Database Management Systems and their Query Languages 27

2.4.3 Distribution Methodologies & No-SQL Architectures . . 37

2.5 Summary 39

3 Architecture **41**

3.1 Architecture Overview 42
3.2 Column Stores 43
3.2.1 Numeric Data Representation 43
3.2.2 Zipfian Columns 44
3.2.3 Continuous Columns 51
3.3 Indexing 53
3.4 Distribution 54
3.4.1 Distributed Querying 55
3.4.2 Redundancy 56
3.5 Summary 57

4 Querying **59**
4.1 Introduction 59
4.2 Linguistic Query Types 61
4.3 Overview of Query Syntax and Capabilities 64
4.4 Resolving QBE objects 65
4.4.1 Preamble 66
4.4.2 Algorithm 66
4.4.3 Example 66
4.5 Token stream regex 69
4.5.1 Preamble 70
4.5.2 Algorithm 70
4.5.3 Worked example 70
4.5.4 Limitations 71
4.6 Resolving Query Types 72
4.7 Asynchronous Querying 74
4.8 Sorting 77
4.9 Summary 79

5 Evaluation **81**
5.1 Overview 81
5.2 Quantitative Evaluation 82

5.2.1 Experiment 1: Scalability of existing DBMSs 82
5.2.2 Experiment 2: Scalability of LexiDB 91
5.2.3 Experiment 3: Comparative Evaluation 95
5.3 Hansard Case Study 99
5.3.1 Building the Tool 99
5.4 Summary 117

6 Conclusions 119

6.1 Summary of Thesis 119
6.2 Proposed Design 121
6.3 Limitations and Future Work 122
6.4 Research Questions and Contributions 123

References 124

Appendices 135

A Experimental Results 1 137

B Experimental Results 2 145

C Experimental Results 3 149

D Sample Focus Group Questions 157

D.1 Using the web interface 157
D.2 Comparisons to other systems 157
D.3 Future developments 157

List of Figures

2.1 Growth of corpora over time 8

3.1 Architecture diagram 42

3.2 Inserting Documents into Data Blocks 47

4.1 DFA representing `^[ab]a.*$` 68

4.2 Radix tree representing D 68

5.1 MongoDB Query Times (High frequency words) 87

5.2 MongoDB Avg. Query Times (medium & low frequency words) 89

5.3 Cassandra Query Times (High frequency words) 89

5.4 Cassandra Avg. Query Times (medium & low frequency words) 90

5.5 Insertion and Indexing 92

5.6 Concordance Lines 92

5.7 Collocations 93

5.8 N-grams 94

5.9 Frequency List 94

5.10 Simple querying for Part-of-Speech (c5 tagset) 96

5.11 Common POS bigram search 97

5.12 Common POS bigram search 98

5.13 Annotation Processing pipeline 101

5.14 Hansard UI search bar 101

5.15 Hansard UI Concordance Results 102

5.16 Hansard UI Histogram Visualisation 103

5.17 Hansard UI Query Options 104

5.18 Hansard UI Collocation Word Cloud 106

Chapter 1

Introduction

1.1 Overview

Corpora are samples of real-world text designed to act as a representation of a wider body of language and they are growing rapidly. From samples of millions of words (Brown) to 100s millions (BNC) by the 90s. The magnitude of modern corpora is now measured in the billions of words (Hansard, EEBO). This trend is only accelerating with recent linguistic studies making more use of online data streams from Twitter, Google Books and other data analytic services. Compounding this is the increased use of multiple layers of annotation through part-of-speech tagging, semantic analysis, sentiment marking, dependency parsing, all increasing the dimensionality of ever-growing corpus datasets. Traditional corpus tools (AntConc, CWB) were not built to handle this scale and as such are often unable to fulfil specific use cases, leading to many bespoke solutions being developed for individual linguistic research projects. Many larger corpora are now hosted online in systems such as BYU, CQPWeb and SketchEngine. However, if the system does not have the functionality to perform the querying or analysis a linguist needs they are often left with nowhere to go.

The areas of Database Systems and Information Retrieval Systems have also faced the problems encountered in corpus linguistics in recent years - an explosion in data scale. In those areas, this has lead to the development of scalable approaches and solutions for other "big data" problems. These techniques apply to the problems in modern corpus linguistics. These database techniques can handle but may not be well suited to Zipfian style language data so adjustments to the techniques are investigated.

This thesis describes a set of approaches, methods and practises that apply to database management systems (DBMSs) tailored to fulfil corpus linguistic requirements. These approaches are realised in a corpus DBMS, LexiDB, that is capable of fulfilling not only the traditional needs of corpus linguists but also the modern needs for scalability and data management that have become ever more prevalent in the field in recent years.

The remaining sections of this chapter outline the area of corpus linguistics, expand on the motivations for this project, formalise the research question and objectives and finally outline the thesis structure.

1.2 Area of Study

Corpora are built as representative samples to allow language analysis to be performed reliably without the need to examine every text available in the analysis domain. Corpora can sometimes be general-purpose, meant to represent a language as a whole (BNC) or can be built to examine a specific area such as political discourse (Hansard).

Typically corpus linguists apply a standard set of approaches to analysing corpora (concordances, collocation, frequency lists etc.). As such various software tools have been developed over the years to allow linguists to do just that. These tools are often used as a basis for what may build into a wider analysis based linguistic research questions.

1.3 Motivation

For some corpus analyses, the standard, widely available tools may not be able to meet the functional or non-functional requirements of the project. There can be many reasons for this; perhaps the corpus data cannot be easily converted to a format for the tool, the type of analysis is different from those the tools typically provide or the tool is not capable of supporting the size of the corpora being studied. This can often lead (particularly in large well-funded projects) to specialist tools to manage, query and analyse corpora being built solely for use with a single corpus or a single research project.

Beyond this, often multiple tools are required for the creation and use of a corpus. Typically query tools such as CQP or SketchEngine rely on the corpus being static and once indexed unlikely to change or be extended. This can lead to projects needing to utilise multiple tools for both the corpus creation process and the querying and analysis routine. Dynamic corpora that change as the project grows are harder to manage using the classical tools within a corpus linguistic workflow. A secondary consequence of this is linguists have less chance to explore the data as corpora are built, eliminating or curtailing the ability to perform a more data-driven analysis of the corpora during compilation. Being able to use and explore dynamic corpora, whilst they are being built or compiled, would be a great advantage to both corpus builders and regular linguists.

Many existing corpus tools are designed to only be used locally and have no mechanisms by which they can scale to handle larger corpora, specifically the ability to scale out across multiple machines is not explored. Whilst for small specialist corpora this may be sufficient, linguists now regularly wish to analyse corpora consisting of hundreds of millions or even billions of words.

The "big data" trend in recent years has led to many developments in data systems. Distributed systems and techniques have become ubiquitous when dealing with large datasets and have become accepted approaches for dealing with this issue of scale. Many big data analytics systems are built as Software as a Service (SaaS) which has often led to individual researchers relying on classical DBMSs (which have become distributed themselves) for corpus research databases. This can lead to researchers developing many bespoke corpus systems for individual projects based on existing classical DBMSs. Many of these databases are not well suited to corpus requirements, both in the form they handle and store data and the mechanisms they allow for querying of the data. Clearly, a gap is present between highly specialised/bespoke tools and the less functionally capable general-purpose databases that they are often built on.

1.4 Research Questions

The main research question investigated in this thesis is whether information retrieval techniques and modern database approaches can be married to corpus linguistics to handle the ever-changing array of corpus data requirements? This research question can be broken down into three sub-questions;

1. How can modern database and IR techniques be used to build, store and query corpus data?
   (a) How can corpus techniques and requirements be fulfilled by existing IR and database retrieval mechanisms?
   (b) How effective are modern distribution methods at scaling to meet the needs of extreme-scale corpora?
   (c) Are there any un-tapped methods that can be developed into novel solutions in this problem domain?
2. How can any system developed be evaluated quantitatively to existing systems, both database management systems and corpus data management systems?
3. What qualitatively evaluation can demonstrate the usage of the system

and its usefulness in real-world applications?

1.5 Thesis Structure

The chapters in this thesis are as follows;

Chapter two provides a detailed look at the corpus linguistic data requirements. It examines existing corpus tools and the techniques that they employ as well as analysing their shortcomings. The chapter reviews the modern literature around storage and indexing within information retrieval and DBMSs, with particular focus on text retrieval methods and how they can be applied to the problems faced in traditional corpus software.

Chapter three describes the architecture designed and implemented by LexiDB. The approaches drawn from the literature review are highlighted as well as where these approaches have been extended or modified to be applied to a corpus database.

Chapter four outlines the mechanisms developed to fulfil corpus queries and describes a novel corpus query language/technique that allows for corpus linguists to express queries powerfully and intuitively.

Chapter five is an evaluation of the LexiDB software. It includes a quantitative comparison to existing DBMSs for large scale corpora as well as evaluating the scalability of the database. The qualitative evaluation takes the form of a case study and a focus group where corpus linguists have used a custom-built user interface for LexiDB on a modern corpus dataset.

Chapter six is a summary of the thesis and a conclusion. It provides an examination of the weaknesses of the work as well as looking at potential future work to address this.

Chapter 2

Literature Review

This chapter discusses the background of corpus linguistics with particular attention to how corpus data is typically queried, with definitions of the five most common query types. This is followed by a survey of existing corpus data systems, how they are used, what types of corpus queries they support with an emphasis on how their query languages work. This is then contrasted to modern DBMSs and IR systems, examining how these systems work, the methods employed by typical indexing schemes and how such systems' query languages are not suited to corpus linguistic analysis approaches. Finally, we analyse how modern DBMS, specifically No-SQL systems, utilise distribution methodologies in order to be scalable for larger and larger datasets and what these approaches can teach us with regards to making corpus data systems scalable.

2.1 Background in Corpus Linguistics

Corpus linguistics concerns itself with the study of natural language via methods and techniques centred around corpora, described by McEnery[59] as "a large body of linguistic evidence composed of attested language use". The study of these corpora is intended to provide a means of analysis that can be

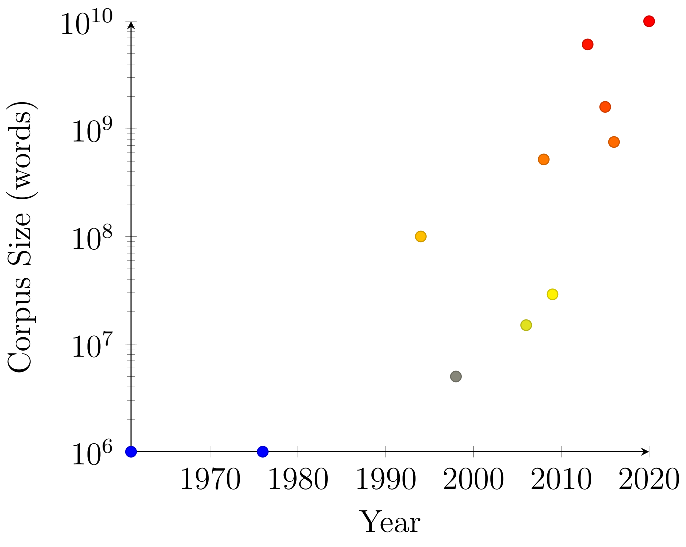


Figure 2.1: Growth of corpora over time

generalised to all language in the domain and should hold true for texts outside the corpus being studied. Historically, construction of corpora was a slow, manual task; searching, gathering and curating hundreds if not thousands of texts into a collection that would be representative of a wider body of language as observed by Bauer[12]. Many modern corpora are often constructed using the web[79]. Corpora can be constructed both to be general representations of language, an example of which would be the BNC (British National Corpus)[55] which are intended to represent a language as a whole but corpora can also be highly specialised such as the NOW (News on the Web)[23] corpus which is built from online news articles. What can be inferred from the analysis of the corpus is always bound to the domain from which the corpus was derived.

Corpora are often (but not always) thought of as large collections of texts. However, the exact meaning of “large” in this context has changed over the years. A typical corpus in the 1960s such as the Brown corpus[38] may have been only around one million words in total. Through the 80s and 90s corpora grew by orders of magnitude, quickly approaching the 100 million word mark with the aforementioned BNC. This trend continued into the new millennium (Figure 2.1[1]) with corpora quickly accelerating through the hundreds of millions of words (COHA[22], EEBO[4]) until now modern, popular corpora regularly consisting of over one billion words (UK Hansard[88], Wikipedia Corpus[30]). This trend cannot be ignored as live corpora creating an ever-expanding plethora of what can be termed extreme-scale corpora, the need for corpus data systems to be updated for the big data age has never been more pertinent.

Beyond simple discovery, gathering and curation of texts into corpora, many techniques to process and enhance the insight of corpora are applied. Almost universally these processing techniques will begin with a form of tokenisation[89], breaking down the text into its small constituent chunks, typically whole words. As Palmer[70] observes, any form of tokenisation will likely involve sentence segmentation as well. Traditionally achieved through classical Markov models[52] and rule-based methods[64], modern tokenisation techniques have moved towards machine learning[85] in recent years. After tokenisation, various levels of linguistic annotation are commonly applied. The most prevalent of these is tagging each word in a corpus with a part-of-speech (POS) tag to explicitly mark its linguistic significance. POS tags are typically verbs, nouns etc. but various tools such as CLAWS[39] that perform this task utilise ever more finely grained tagsets to more and more precisely define the linguistic category of words. While POS tagging is the most ubiquitous of techniques, many modern systems for processing corpora go far beyond this, utilising methods for tagging named entities[49], semantic categories[74], lemmatization[72] and dependency marking[68] to name but a few.

2.1.1 Corpus Queries

Once a corpus is constructed, several methods of analysis are available to corpus linguists. Although not limited to those presented here, what follows

[1]Corpora: NOW[23], EEBO[4], Hansard[3], USENET[83], YCCQA[28], COCA[21], CLEMETEV[27], CEEC[69], BNC[55], LOB[56], Brown[38]

is an overview of the five most pertinent based on what is usually available in corpus analysis tools, discussed in section 2.2. Here we discuss them in general terms although their usage, application and manifestations may vary and is highly intrinsic to the software tool being utilised when performing each analysis. This initial discussion is agnostic of such considerations.

2.1.1.1 Concordance Lines

Concordance analysis is a common technique used by corpus linguists. This involves the analysis of words surrounding a particular search word or phrase to derive the potential meaning of the word (in lexicography) or to explore patterns of usage. Displaying the search word or phrase with the words immediately adjacent and aligning them into a view with other occurrences of the search word vertically is a typical means of displaying concordance lines and allows linguists to see typical patterns of usage in context. In information retrieval, this is sometimes known as keyword in context (KWIC). Conclusions can be drawn from this analysis such as in a study conducted on the portrayal of Muslims in the British Press [9]. How easily conclusions can be drawn is tied to having sufficiently large corpora to find enough evidence in the form of concordance lines. Common words can be examined in relatively small corpora. When using a corpus that is representative of a larger body of language, less frequent words or phrases require larger and larger corpora to gather sufficient data for a meaningful concordance analysis. Concordance analysis, when done at scale, can be thought of as combining aspects of both qualitative and quantitative analysis[8]. This has motivated the need for larger and larger corpora and entails a need for software tools capable of efficiently handling them.

2.1.1.2 Frequency Lists

Another type of query supported by various corpus tools is the generation of frequency lists. Frequency lists can be generated for entire corpora, this can often be done to compare how frequent particular words occur between two different corpora. The ability to filter and search these frequency lists is powerful, it can allow for regular expressions to be utilised to search for a particular pattern of word. Beyond this in the case of annotated corpora which include lemmatisation, the lemma can be used and a frequency of the different words that share the same lemma analysed. From such frequency lists, dispersion across a whole corpus can also be analysed[41]. This looks not only at the frequency of a word but also where in a corpus the word occurs, words may occur highly frequently in one specific text in a corpus but scarcely appear in the remainder of the corpus.

2.1.1.3 Collocations

Collocations of a particular lexical feature for example a word, are a statistical measure of how commonly that particular feature co-occurs with another in a corpus. Typical examples of collocations might be the word "white" and "house". Various statistical measures exist for calculating collocations. Usually such metrics require a contingency table be constructed by finding the frequency which a feature occurs in the corpus with and without a particular collocate. The resulting contingency table would be as such[54];

$$\begin{array}{cc} f(xy) & f(x\overline{y}) \\ f(\overline{x}y) & f(\overline{xy}) \end{array}$$

In a simple case this table represents when a word x occurs and does not occur with a collocate y i.e. $f(xy)$ means the frequency of x with y, $f(x\overline{y})$ means the frequency of x without y etc. Metrics such as mutual information(MI)[19] can be calculated from this basis;

$$MI = log\frac{P(xy)}{P(x)P(y)}$$

where the probability is $P(xy) = \frac{f(xy)}{N}$ when N is the total size of the corpus. Further metrics can be calculated from the simple calculation of a contingency table between two corpora, such as log likelihood;

$$-2ln\lambda = 2\sum_i O_i ln\left(\frac{O_i}{E_i}\right)$$

Here E represents the expected values and O the observed values[75][31].

Other calculations are of course possible such as chi-squared(χ^2)[48] etc.[60] From the perspective of a corpus data system, the choice of metric is trivial, what is essential is that the data is structured in a way that collation of the initial contingency table is fast and efficient.

2.1.1.4 N-Grams

N-Grams are sequences of n items in a sequence of data. Within a corpus n-grams typically are handled at the token or word level but are often applied in computational linguistics at the character level. Searching for n-grams (sometimes referred to as clusters) in corpus data systems will provide a means of analysing common patterns that occur within a corpus. N-Grams are typically computed initially when a corpus dataset is first loaded into a system and the most frequent unigrams, bigrams etc. can be viewed (it is common to only generate up to 5-grams). As well as this it is often useful to be able to search within n-grams for specific words so that commonly occurring words with a word of interest can be viewed. N-Grams can be used for many statistical analysis techniques, including but not limited to authorship analysis in forensic linguistics[90] and for corpus-based machine translation[37].

2.1.1.5 Keywords

Some corpus tools allow for keyword analysis. In this context, a keyword is a word that occurs more commonly than expected. Usually, such an analysis will involve two corpora and calculating some metric (possibly log-likelihood discussed above or similar) between a target corpus and a reference corpus. Usually the target corpus will be a specialist corpus (e.g. a health care corpus[2]) and the reference corpus will often be a general-purpose language corpus. The metric is commonly known as keyness and the results of the analysis can be sorted by keyness. Much like with the calculations for collocations, efficient generation of frequency lists between corpora is essential for calculating keyword lists. This approach can be extended out further within annotated corpora to rather than simply look at keywords but keyness between annotations[73].

2.2 Existing Corpus Data Systems and their Query Languages

2.2.1 Overview of Corpus Data Systems

Corpus workbench (CWB) [18] is a set of corpus management tools for indexing and querying static corpora. It has undergone many changes and developments[34] since its inception and is often used as the backend for various corpus interfaces including CQPWeb[45]. Corpus Query Processor (CQP) is CWB's means of providing meaningful linguistic query functionality through means of a query language that is often referred to as CQL (Corpus Query Language). CQP's flavour of CQL can be used to express a pattern in running text by specifying not only words but also utilising word-level annotation such as part-of-speech tags e.g. `[word="in"][word="the"][pos="NN"]` would search for nouns preceded by the words "in the". CQL can also support regular expressions at both the character level and the word level, with the word level support for regular expression operators being implemented within CQP[33].

Regular expressions can be handled at the character level in CQL; `[word="s.*"]` this query would find all words that begin with the letter s. On a word-level, regular expression operators can be handled such as; `[pos="JJ"]+`. This query would find one or more general adjectives (CLAWS7 POS tagset[2]). This form of querying is powerful when searching a tokenised corpus. A tokenised corpus, including an arbitrary number of annotation layers, can be described as a token stream. Token streams allow for linguistically meaningful interrogation. However more traditional methods for searching i.e. character-level regular expressions across multiple words are less feasible in this approach as this may require the original untokenised text, alternatively a mechanism by which the text can be reconstructed from multiple tokens. Despite this limitation, this style of querying over token streams has remained the de facto standard for querying style in corpus data systems since the introduction of CWB in the mid-1990s.

As a result of regular expression operators being used at both the character (within tokens) and at the word level, CQL can allow for multiple methods of expressing what is fundamentally the same query e.g. `[word="(tall|long)"]` would be considered synonymous with `([word="tall"]|[word="long"])`. Despite this, due to the way CQP attempts to resolve these queries, expressing synonymous queries may result in more or less efficient query execution. This illustrates that even with older systems, users are required to possess a level of understanding of how the underlying architecture resolves queries. In some respects, this could be considered a barrier for entry to using such systems but it is also reasonable to postulate that such limitations would only occur on the edges of a system's capabilities, particularly in regard to its scalability i.e. posing a query to CQP, expressed in CQL in a less than optimally efficient manner, might have no noticeable effect if searching for something relatively

[2]http://ucrel.lancs.ac.uk/claws7tags.html

simple or infrequent in a small corpus.

The interface and functionality of CWB and CQP have been extended into a web-based medium in CQPWeb. CQPWeb provides a more modern interface to using many of CWB's features (as opposed to a command-line interface). CQPWeb allows for query types such as concordances to be visualised more easily as well as allowing users to examine dispersion in a corpus. The primary advantage to CQPWeb is the simplified admin methods that it allows for, meaning subcorpora and parallel corpora can be easily created allowing for more refined querying as well as enabling other query types such as keywords. A final noteworthy feature of CQPWeb is a more simplified query language, CEQL (Common Elementary Query Language), which allows for a simpler, but more tightly coupled query syntax e.g. `word_POSTAG`. This allows for simple queries to be performed more easily but is less expressive than the full CQL syntax.

Sketch Engine[3] is another corpus management tool which utilises it's own very similar flavour of CQL as its query language of choice. Sketch Engine is a hosted service operating with a SaSS (software as a service) model. This allows for various existing corpora to be pre-loaded and hosted. Sketch Engine is capable of performing many of the typical linguistic searches; concordances, frequency lists, collocation etc. and is built on top of Manatee and Bonito[77]. Manatee is a database management system to allow corpora to be indexed using an inverted text index (described in section 2.4.1) and Bonito provides a web interface in a client-server model paradigm. A free alternative to Sketch Engine exists; NoSketch Engine allows users to access many of the features of Sketch Engine without the need to pay for a hosted service.

Poliqarp[51] utilises a custom query language whose syntax appears very similar to CQL but includes subtle variations and a certain amount of syntactic sugar to allow for other forms of specialist queries regarding sentence ambiguity etc. Poliqarp is used as the search interface to the national corpus of

[3]https://www.sketchengine.eu/

Polish[4]. This provides a web-based search interface that is becoming more and more common means to perform corpus searches due to the ubiquity of web technologies and HTML. The implementation of Poliqarp as described[50] states that sequences of words from a corpus are stored as bytes corresponding to offset positions within a dictionary of orthographic words. Details are also given on how Poliqarp's custom query language could not be mapped easily onto an SQL or set of SQL statement, motivating the wider need for custom data layers for corpus query systems.

Another platform to consider is COSMAS 2[5] (Corpus Search, Management and Analysis System). This hosted service provides access to over 500 corpora that can be queried using a custom query language. This query language differs from CQL but provides similar functionality. This platform was developed at the Institute for the German Language Mannheim and succeeded by KorAp[10]. As well as providing a hosted deployment[6] containing several sample corpora, KorAp is also open source and individual modules are freely available[7] (under BSD license).

The management tool Corpuscle[63] developed at the University of Bergen provides a fully bespoke system for storing and querying annotated corpora. Similar to other systems its query language, CorpusQueL, draws inspiration from and shares similarities with CQL although as stated in the literature does not share precisely the same functionality. Whilst a hosted version of this tool exists[8] which allows access to many different corpora, the code itself, is not published anywhere and therefore cannot be used by linguists to store and query their own corpora locally. Much like CWB, Corpuscle allows for regular expressions to be resolved at both the character and word level, utilising finite state automata and minimising the number of corpus positions needed to be checked within the corpus to resolve these[62], Corpuscle has been demon-

[4]http://nkjp.pl/poliqarp/
[5]https://www.ids-mannheim.de/cosmas2
[6]https://korap.ids-mannheim.de/
[7]https://github.com/KorAP
[8]http://clarino.uib.no/korpuskel/

strated to be faster than CWB for many specific query examples, particularly phrases beginning or ending in relatively low-frequency terms.

More user-friendly corpus management tools exist that provide more flexibility (allowing users to query their own corpora) and require less technical experience to setup. Tools such as AntConc[5] and WordSmith[78] are widely utilised by corpus linguists for several reasons. AntConc provides means of handling various corpus queries including KWIC, frequency lists, collocations & keywords. Its interface takes the form of a desktop application as opposed to many other tools that have moved to a web-based interface. AntConc allows users to not only query but explore their corpora by looking at dispersion and making it easy to compare multiple corpora. Whilst both widely adopted and highly usable, AntConc struggles to provide the kind of support from extreme-scale corpora that many other systems seek to provide (although many other systems operate as hosted SaSS solutions).

BYU[9] contains various English language corpora, including several in the order of billions of words such as EEBO[4] (Early English Books Online) and Hansard[3]. Although BYU has some disadvantages to other systems in terms of functionality and expressiveness of linguistic queries, one key advantage to its architecture (discussed in section 2.2.2) is the potential for corpora to be dynamic. Most other corpus data systems require the corpora they use to be static, pre-processing is required upon insertion to prepare the corpus for querying. Davies 2019[24] demonstrates how utilising relational tables and the architecture of iWeb, corpora can be added to dynamically. Work outlines how this can be done automatically, web scraping articles and passing them through a pre-defined toolchain for annotation before insertion into the database. It cannot be overstated the importance and potential advantages in terms of research avenues this presents for corpus methods in the 21st century and the big data age.

[9]https://corpus.byu.edu/

2.2.2 Corpus Data Systems Architecture

Many of the underlying algorithms and techniques utilised by existing corpus data systems are not richly conveyed in published materials, rather systems that are described are done based on a high-level architecture that they employ. What follows is an assessment of the architecture of some of the systems described in the above overview, where such information on their architecture exists.

Corpus workbench's architecture is split between the data model storing the indexed corpora and a registry which describes them. On top of this sits a set of library functions which access these low-level data structures. These functions can be accessed through a set of utilities or using the Corpus Query Processor which can handle queries in CQL. This modular like architecture splits the various components of CWB down such that other functionality or accessor patterns might be applied, for example, it allows the possibility to replace CQP with a different query processor or for other tools to access the lower level corpus library functions. This modular type approach can be seen in other architectures for corpus data systems.

KorAp presents a very modular approach to a corpus data management platform. KorAp utilises Neo4j (see section 2.4) as its database and uses Lucene (see section 2.4.1) for indexing[81]. The access layer to these other tools is a module called Koral. Similar to other tools that use custom query languages, KorAp too has a custom query language, KoralQuery, but KorAp provides the means of translating queries from other widely used corpus query languages such as CQL, AnnisQL as well as COSMAS. This removes a barrier for entry for users who may be familiar with other systems and thus do not need to learn another query language to use KorAp. KorAp provides a web-based interface called Kalamar but also provides an application programming interface (API) which allows for different services to easily be built on top of it. Whilst the modular nature of KorAp makes it flexible, it also means setup and configuration is difficult without proficient technical knowledge and understanding.

Again the modular design of the components allows potential for replacing things like the query processor or the underlying data access layer with other alternatives.

The architecture of BYU's system, sometimes referred to as iWeb, is based on a relational database running on Microsoft SQL Server[10]. Limited information on the design of this database shows that it primarily split into three relations[11]; a lexicon table representing a dictionary for a particular corpus, a corpus table containing the token stream for a given corpus and a sources table containing information on individual texts in the corpora. The lexicon relation contains columns for the word and associated tags (lemma, POS). Within the corpus table, each row represents an occurrence of a word and includes a foreign key referencing the word in the lexicon table, as well as a foreign key referencing a text to which the word instance pertains in the sources table.

Beyond the lexicon and corpus table, an additional table is required in order to retrieve contextual information around hits. On insert a table containing columns for each corpus position as well as the 5-10 words either side of a word is computed (as described in Davies 2019[25]). This can effectively be thought of as an ngram table, with at a maximum 10-gram. As described the iWeb querying method is to find the lowest frequency term in a search string and perform a lookup within this table, the middle column being the indexed column which is the target of the lookup. Although this approach is demonstrated to be efficient in certain circumstances, it remains prohibitively expensive for searches containing only high-frequency terms due to the sheer size of index entries that must be retrieved for each term. Beyond this, the query syntax does not allow for more complex linguistic query expressions that other corpus query languages such as CQL do.

BYU demonstrates a typical way in which corpora can be stored in a classical relational database. The speed of the architecture for performing simple

[10]http://www.microsoft.com/sqlserver/

[11]As described https://www.corpusdata.org/database.asp

sequence queries[12] has been compared favourably to Sketch Engine. Whilst this data is compelling in demonstrating the potential speed advantages of the approach over Sketch Engine, there is no discussion of the fact that both are running (or assumed to be) as hosted services, therefore direct comparisons without knowing anything about the underlying hardware, or potential load on the system when the tests were performed, or even the fact that the tests were not run on the same corpora (only two available corpora of comparative size) all illustrate the need for such systems to have more uniform methodologies for evaluation and comparison which in part is what this work hopes to address.

2.3 Proposed Systems & Related Work

An extension to the underlying data model of CWB has been proposed in a new data model Ziggurat[35]. This proposes moving from a token level data model towards a multi-layered approach that could handle the hierarchical structure of corpora in XML format but also concurrent annotation layers. Whilst the proposed model suggests great functional capability, methods and algorithms to allow for greater scalability are not given, beyond simply switching from CWB's usage of 32-bit integers, which limits corpus size to 2.3 billion words (signed integers must be assumed here) to 64-bit integers. The proposed solution to handle hierarchical structures, such as XML, is fundamentally the same approach utilised by existing XML databases such as BaseX[13].

Typically corpora are stored in the form of XML files. Various XML databases exist that can query XML documents using XQuery, a subset of the functionality of XQuery is XPath. LPath[15] presents an alternative to XPath that is based around resolving linguistic queries, specifically it adds new axes to the XPath syntax. On top of the typical ancestor, descendant, sibling axes already available in XPath, LPath proposes adding two new axes; "imme-

[12]https://www.english-corpora.org/speed.asp

[13]http://basex.org/

diately-following" and "preceding". These axes allow for queries in this new syntax to more precisely express contextual queries that are commonly utilised in linguistic data systems. Algorithms are presented to translate such queries from this custom XPath like syntax into SQL statements so linguistic data can be stored in a traditional database but still queried in a more powerful way. Storing XML in traditional database tables is discussed in section 2.4.2.2.

Banski[11] presents a strategy for classifying the capabilities of query languages; CQLF (Corpus Query Lingua Franca). The purpose of this work is to label the specific capabilities of the various querying languages prevalent in corpus data systems. The work briefly examines various querying languages discussed previously such as the COSMAS query language and the Sketch Engine flavour of CQL. The three main strata for which the specification classifies query languages are; simple, complex and concurrent. "Simple" applying the query over plain text or token streams, "complex" over potentially hierarchical annotations and "concurrent" requiring support for dependency annotations. Whilst this work does not provide a basis for approaches to the scalability aspect, it does provide a convenient means for classifying any query language developed to support extreme-scale corpora.

The COW14[80] architecture describes a set of tools used to create, process, store and query corpora up to 20 billion words. This toolchain's search system, known as Colibri, is based on CWB. It provides a web-based interface that uses the data storage and lookup capabilities of CWB in a similar way to CQPWeb. In a related vein, Colibri also offers a simplified query language as an alternative to CQL, although full use of the CQL syntax is also supported. Despite the fairly common methods employed by the search or querying component of this architecture, the processing component presents a compelling case for the benefits of distributed computing for extremely large corpora. This demonstrated that corpora in the order of 10s billions of words can be linguistically annotated across a large cluster consisting of dozens of nodes in under 5 hours (as opposed to 3 days). Whilst corpus annotation is not a goal of this thesis, scalability is and this kind of distribution is not classically seen in corpus systems but is starting to become prevalent.

This movement towards distributed approaches to handling corpora is also demonstrated by Porta 2014[71]. The demonstration of Map-reduce algorithms both for compiling n-grams from large text collection consisting of 800 million words as well as the demonstration of collating inverted files (a key method for indexing) across multiple workers. It is noted that time for building such file inversions does not scale linearly as corpus size increases but is limited by the bottleneck of disk access. Much like with other modern DBMSs dividing data up into blocks (see section 2.4) the approach here also dumps data from the file inversion process to disk every million documents that are indexed, thus creating the disk bottleneck, without which the processing ability of Map Reduce would likely scale more uniformly.

Vandeghinste[87] proposes the reuse of existing tools and techniques from information retrieval, specifically XQuery and XPath, making use of BaseX, to create a query tool that can scale to support a treebank corpus of around 500 million words, SoNaR. The approach taken requires the data to be separated into smaller databases so that complex XPath queries can be computed more efficiently over what have effectively become sub-trees of the entire corpus. The search engine design put forward, referred to as GrETEL, uses an amalgamation of a classical database query by example approach and combines this with TIGER[47], a semantic web orientated corpus navigator. Within the architecture, queries are translated into a base XPath form and evaluated in BaseX against the many split parts of the whole corpus.

2.4 Database Management Systems and IR systems

2.4.1 Indexing

2.4.1.1 Postings Lists

Postings lists[53][6] are not only used in classical databases as a way to index and search for specific values but they are also widely utilised by modern information retrieval systems and search engines. Postings lists applied to databases contain a list of records where a particular value occurs in a table. In information retrieval systems a postings list may contain a list of documents where a particular word occurs[58]. This is sometimes known as a record-level inverted index. A full inverted index differs from this as it not only describes the documents (or records) where the value or word occurs but also its position within the document. Postings lists are used for text indexes frequently in all manner of systems including DBMS and search engines[46].

Given a set of documents $D = \{d_1, d_2 ... d_n\}$ a dictionary of all words in D can be compiled with each word's postings list w_n being expressed as a subset of all documents $w_n \subseteq D$. The set of all postings lists which expresses the index is thought of as $P = \{w_1, w_2 ... w_n\}$. In many cases (particularly for common words or values) it may be the case that $w_n = D$. If the documents are written in a natural language Zipf's Law[92] tells us that around 50% of the set P will contain only one entry[91]. These factors would potentially allow for common words to share a postings list D without the need to store each separately but this is only the case when storing a record-level inverted index, a full inverted index would require positional information within each document to be stored as well.

A full inverted index can be conceptualised by extending the above interpretation further to w_n no longer being a subset of documents $w_n D$ but rather a set of tuples where each tuple contains a document and an integer that is a word's offset from the beginning of the document $w_n = \{(d_i, n), ...\} : d_i \in D, 0 <= n < length(d_i)$. In practice, for the sake of saving memory and or disk space, the postings list may be stored as a tuple containing the document (or document ID) and a set of positions where the word occurs in the document $w_n = \{(d_i, \{n_1, n_2...n_j\}...)\}$. In many cases, where documents or records are stored consecutively (perhaps in a single file on disk) the postings lists may simply be stored as a set of offsets from the beginning of the database. The offset may represent a byte length or indeed a multiple of a byte-length if the document's contents is converted to some fixed length set of values e.g. if all words in the document were converted to 32-bit numbers the offset may be m but the offset in bytes would be $4m$ (32 bits equals 4 bytes).

Whilst postings lists provide the basis for performing an efficient search within any database or data store containing textual data, specifically in our case corpus data, it is only the beginning of the process. Once a postings list is established and its dictionary compiled its dictionary must be, itself, searchable. Searching a small dictionary may seem trivial, a simple brute-force scan may be sufficient given a dictionary of only a few thousand words, but dictionaries of thousand of documents will be many orders of magnitude larger than this. Beyond finding a specific postings list based on a lookup in the dictionary, an individual postings list itself may become inordinately large at the type of scale of data that modern data scientist and corpus linguists are now commonly working. This means storing the entirety of the index in memory, including all postings lists, for the sake of efficiency is no longer practical as would have been done classically in older far smaller relational databases. Compression is therefore essential, not only to reduce the size of the index in memory but to minimise the amount of data that must be read from the disk in the ever more likely circumstance that an index cannot fit into memory even with compression. Discussed below are the structures and approaches that exist in the literature for solving these problems.

2.4.1.2 Searchable Dictionary Structures

Once such dictionaries are constructed they require a means by which to be searched. Traditional data structures such as b-trees, b+ trees and hash tables are suitable candidates to provide such functionality, but they fall short when dictionaries become vast and particularly when, as is often the case with the scale of corpora we are discussing here, they cannot fit in memory. Radix trees[26] or tries[57] have been demonstrated to be an efficient means of searching dictionary data[13] particularly in the big data age. Tries can be constructed from a dictionary where each node of the trie has a branching factor equal to the number of different letters that occur in that position within the dictionary. Radix trees differ from conventional tries in that each edge need not only be indicative of a single character but a string of characters for improved compression. Variations on radix tries are also possible to eliminate suffix redundancy in the form of deterministic acyclic finite state automaton[20], but since these structures do not use a vertex for each word, pointers to dictionary entries must be stored at each node as a list or array leading to potential further searching within this set.

2.4.1.3 Index Compression

2.4.1.3.1 FOR PFOR-delta Frame of reference (FOR)[40] can be used to minimise the number of bits needed to encode index positions in a postings list across a large number of records in a database. Given an unsigned 32-bit integer the maximum potential length or number of records in a database starting with the first record 0 is approximately 4.3 billion. Every position in the index would take up 4 bytes. By arbitrarily splitting a large group of records down into smaller chunks and only storing the position information at these splitting points as an integer, subsequent entries in the index between these points can be stored simply as the offset from this point or frame of reference. This means not all index entries need be stored as 32-bit integers but can be stored as the minimum number of bits needed to represent the gap between the reference points e.g. if a reference point was created every million records $n_{for} = 1 \times 10^6$ then the intermediary index entries between these could be guaranteed to be stored using 20 bits since $2^{20} > n_{for}$. This reduces the size of any postings list in the index whilst still allowing the list to be read sequentially.

Although FOR can reduce the number of bits required for almost all index entries (besides the reference points for each frame), the total space required can be reduced further by encoding the delta (based on Elias Fano encoding[32][36]) between each index entry as opposed to its offset from the starting frame of reference. The FOR approach can still be utilised to traverse large lengths of the postings list but between each frame of reference, the number of bits needed is now the minimum number required to express the large gap between entries. As such, extremely common values in the index will have huge postings lists with potentially millions of entries but on average their deltas will be smaller and thus require fewer bits to express than lower frequency, sparsely spaced values. If within a particular frame from a starting reference point the list of deltas $D = \{\delta_0, \delta_1 ... \delta_n\}$ contains a largest delta $\delta_{max} = max(D)$ then the list itself can be expressed by $\lceil log_2(\delta_{max}) \rceil$. Consider a postings list $P = \{12, 25, 33, 46, 57, 70\}$, the minimum number of bits to express these would be $\lceil log_2(70) \rceil = 7$, so storing this relatively small list would require $7bits \times length(P) = 7bits \times 6 = 42bits$. Encoding the deltas $D = \{12, 13, 8, 13, 11, 13\}$ would be possibly using $\lceil log_2(13) \rceil bits \times 6 = 24bits$ or 43% less bits in total.

Patched Frame of Reference (PFOR)[94] presents a method for increasing the achievable compression from the previous methods even further. Beyond applying the approach in FOR, PFOR-delta indexing uses a frame of reference for each block and when determining a minimum number of bits needed to store the deltas, exceptions can be made where values are larger than typical, these can then be stored separately (typically after the list of deltas) and a reference pointer left in the original list. This method has been shown to achieve higher levels of compression and throughput for storing inverted indexes. One disadvantage it has over previously described methods is the entire reference block must be read to compute various offsets. FOR/FOR-delta can be read and values from it computed on the fly with no need for the entire index block to be loaded into memory at one time, although this would be typical, PFOR-delta does not make this possible.

2.4.2 Querying Corpora with Database Management Systems and their Query Languages

2.4.2.1 Lucene

Lucene[14] is an open-source search engine that has become ubiquitous in modern internet applications as well as various No-SQL databases. Lucene constructs a full-text index and provides a query interface in the flavour of the Lucene Query Syntax. This does not allow for the same kind of expressiveness in linguistic queries such as CQL but allows for more flexibility than traditional query languages such as SQL. The syntax can support fuzzy queries as well as multi-word queries, searching for a series of words within some arbitrary distance of each other within a document. As Lucene was designed as a search engine, like web search engines it computes metrics for ranking documents returned from its search results, generally based on information retrieval principles specifically term frequency, inverse document frequency. Lucene has many derivatives and extensions as well as several internal algorithms and data structures that could be utilised by a modern corpus data system or database.

ElasticSearch ElasticSearch[15] is a distributed search engine that is built on top of Lucene. Its design allows for sharding (splitting a single dataset) of Lucene indexes across multiple nodes, this allows the system to scale out for increased availability and throughput. ElasticSearch has a domain-specific

[14]https://lucene.apache.org/
[15]https://www.elastic.co/elasticsearch/

language based on JSON which is used for querying but also includes extensions that can allow it to handle SQL like queries, although unlike most classic SQL based systems it does not support distributed transactions. Similar to Lucene, ElasticSearch allows for multi-word queries and ranges between these words to find documents that contain a particular phrase. Unfortunately, this approach does not extend far enough to support the kind of token level regular expression functionality that is available in corpus query languages such as CQL.

2.4.2.2 XML

Due to the commonality of XML as a storage and interchange medium or file format for modern corpora, it is important to consider what XML database and storage techniques can teach us about how these documents are searched, both when used for corpora and in more general cases. Many of these techniques may be adaptable or relevant to provide novel means by which extreme-scale corpora can be handled. XQuery[16] is the general query language used for querying XML documents. XQuery is based upon XPath but extends this out with various functional forms and features. XSLT (XML Stylesheets) are a means by which XML can be transformed into some meaningful presentable form. Many systems utilise a combination of XQuery and XSLT; XQuery firstly to produce a query result, typically one or more XML fragments, and then an XSLT stylesheet transformation to return to the user and present the results of the query in the desired manner.

eXist DB[16] is an example of a native XML database. Similar to other XML databases, eXist can be classified in the NoSQL flavour of databases as a document store. The database makes use of Lucene for text indexing and searching (section 2.4.2.1). Much like other NoSQL databases, eXist can operate in a clustered configuration. The distributed architecture is a master/slave configuration with a single database instance acting as a front end and by use of a message-passing system (publish-subscribe pattern) storing data on various

[16]http://exist-db.org/

slave nodes. This type of distribution is essential to scalability and is something that is clearly missing from typical bespoke approaches to corpus data systems.

Using XQuery for searching corpus-based XML documents is something of a two-edged sword. Whilst the expressive power of XQuery can make it possible to interrogate structural aspects of XML corpora in a more meaningful way, the syntax itself is perhaps more difficult to learn than CQL and other specialised corpus query languages. Take for example the following snippet (In Parla CLARIN XML format[17]) as the way a set of corpus documents are stored;

```
        <?xml version="1.0" encoding="UTF-8"?>
        <teiCorpus xmlns="http://www.tei-c.org/ns/1.0">
        ...
        <text xml:lang="en">
            <body>
        ...
            <s>
                <w lemma="the" pos="AT">The </w>
                <w lemma="only" pos="JJ">only </w>
```

Listing 2.1: ParlaClarin XML format schema

To query for all verbs in the corpus an XQuery expression may look something like this;

```
        doc("*)//w[@pos="V.*"]
```

Listing 2.2: XQuery for all verbs

This syntax uses XPath to express a tree hierarchy of the desired query target, i.e. any `w` element with a `pos` attribute that begins with a `v`. This syntax is currently straightforward but only to those with a good grasp of XML or familiarity with XPath. Taking this a stage further pushes the syntax into territory only experts (or at least experienced users) on the syntax would likely follow. We can extend the previous example to return the sentence to

[17]https://github.com/clarin-eric/parla-clarin

which these verbs belong, rather than returning the verbs themselves. These sentences can then be built into concordance lines;

```
        for $x in doc("*)//s
        where $x//w[@pos="V.*"]
```

Listing 2.3: XQuery for sentences containing verbs

Very quickly this syntax builds to a point that is not feasible for users of a corpus data system to learn easily. In the above example, there is no easy way in XQuery (at least within this XML structure) to retrieve the words at the end of the previous sentence. This limitation is in no small part the motivation for LPath, discussed in section 2.3.

Within eXist, additional functional capabilities make keyword in context searches possible using Lucene. Through a combination with XSLT transformation a method by which concordance lines can be retrieved is arrived at as follows;

```
    xquery version "3.0";
    import module namespace kwic="http://exist-db.org/xquery/
kwic";
    declare namespace ft = "http://exist-db.org/xquery/lucene";
    let $keyword := "time"
    let $number_of_results := 50
    let $context_size := 30
    return
    <html>
        <head>
            <style>
                p {{
                    font-family: "Courier New", Courier,
monospace;
                }}
                .hi {{
                    background-color: yellow;
                }}
            </style>
        </head>
        <body>
            <h3>Results for search "{$keyword}"</h3>
            {
                for $results in subsequence(collection("/db/bnc
")//s[ft:query(.//w, $keyword)], 1, $number_of_results)
                return
                    kwic:summarize($results, <config width="{$
context_size}"/>)
            }
        </body>
    </html>
```

Whilst the above listing allows for much of what linguists might expect as a concordance line set of results, its structure has grown ever more complex, including mechanisms for transforming the results as HTML. The listing searches for the keyword `time` and uses an eXist module to generate the context window around the hits, found using a Lucene index. Whilst this does demonstrate the utility of XQuery, it also illustrates the necessity of any corpus system built on top of eXist, or any XML database which uses XQuery, to abstract away the underlying query language from the user for the sake of ease of use. This can lead to the corpus system becoming more bespoke than one might assume, but the same could be said for any data query system.

BaseX[18] implements a document store specifically for XML, similar to eXist. BaseX is implemented as a tabular method of storing XML, with nodes represented like records within its internal structure. This allows BaseX to gain some of the mechanisms developed for traditional relational databases, such as accelerated joins, but allows these to be utilised in its XQuery implementation. Unlike eXist, BaseX has its own implementation[42] of the XQuery Full Text recommendation[19]. This implementation is realised with a bespoke indexing system, as opposed to relying on Lucene as many modern DBMSs do. This method is shown to be effective in no small part due to the customization

[18]http://basex.org/
[19]https://www.w3.org/TR/xpath-full-text-10/

that can be applied to the index format that would not be possible using a pre-packaged generalised indexing tool.

2.4.2.3 SQL

Structured Query Language (SQL) has been the basis for many relational database management systems for nearly half a century. Since its development at IBM in the 1970s, it has become the defacto query language for tabular data and data stores. Beyond simply being a query language, SQL can also be used to express the structure of relations with a database (schema) as being used to insert data and provide meta information. In the context of linguistic data, SQL based systems such as MySQL, SQLServer and SQLite have been used by various linguistic tools such as iWeb. One of the primary advantages of SQL systems is the ER (entity relational) model, which can be used to model almost any structure of data imaginable. Modelling a simple annotated token stream with tabular relations is fairly trivial.

A naive model for a corpus database in SQL might be;

```sql
        CREATE TABLE Doc(DocID int,
            Title varchar(255),
            PRIMARY KEY (DocID)
        );
        CREATE TABLE Tokens(Word varchar(255),
            POS varchar(5),
            SEM varchar(6),
            DocID int,
            Position int,
            FOREIGN KEY (DocID) REFERENCES Doc(DocID)
        );
```

Listing 2.4: Creating corpus tables in SQL

This would create a pair of tables capable of storing sequences of words and the associated documents where the words occur along with two annotation layers; part-of-speech and semantic tags. Whilst this structure seems usable it quickly becomes problematic in most modern relational database systems as building and maintaining a full index on the `Word` column as the number of words in the corpus grows and grows is untenable, particularly when considering database systems' reliance on keeping indexes in memory to provide faster lookups.

Assuming the structure described above, the table itself can be easily queried to find a specific search term or keyword $\$x$;

```sql
        SELECT *
        FROM Tokens, Doc
        WHERE Tokens.Word = "$x" AND Tokens.DocID = Doc.DocID;
```

Listing 2.5: Querying for keyword in SQL

This select statement would return all occurrences of the word x in the database along with the document where the word occurs. Considering the most basic of corpus linguistic queries concordance lines, we quickly see such a query in SQL does not provide nearly enough information to create concordance lines from. An additional query would need to be made for each record returned from the first query to build a concordance line around the "hit", taking the value of Position from each record returned as n and the DocID as d;

```sql
        SELECT *
        FROM Tokens, Doc
        WHERE Tokens.DocID = "d" AND
        Tokens.Position < (x-10) AND
        Tokens.Position > (x+10);
```

Listing 2.6: Naive concordance query in SQL

Having to perform such queries becomes inordinately expensive, particularly with larger query sets. Although this approach is naive it demonstrates that SQL cannot provide context-sensitive data returns en mass when querying, although the iWeb architecture discussed in section 2.2.1 demonstrates a way around this problem.

2.4.2.4 NoSQL

NoSQL databases have emerged in the 21st century as a meaningful alternative to classical relational models and the ACID[44] model. With the proliferation of big data, modern NoSQL databases primarily concern themselves with distribution and scalability and are intrinsically tied into CAP theorem[17]. CAP theorem suggests that there are three primary concerns of a distributed system; consistency, availability and fault tolerance and theorises that no system can have high levels of all three, to achieve one or two of these properties the third must be sacrificed. Consistency ensures that all data between nodes in a distributed system is the same and high consistency would ensure all nodes can see and access the same data at the same time. Availability in this context is how readily the data is accessible to users. Fault tolerance is how the system deals with node failure; is there potential data loss or some redundancy in play. Different NoSQL databases demonstrate this principle in practice, often knowingly sacrificing one of these properties to achieve different functionality. How these systems operate and the properties in terms of CAP that they exhibit must be considered when contemplating the properties and requirements a scalable and potentially distributed corpus database would posses. What follows is an overview of the most popular NoSQL databases, an illustration of how some might be used to store and query corpus data, and a summary of the advantages and disadvantages of their storage paradigms[67] as well as their distribution strategies.

2.4.2.5 MongoDB

MongoDB[20] is a document orientated database. Documents are stored within the database as JSON (javascript object notation) and are queried using a set of javascript functions. One of the biggest advantages of MongoDB is scalability. The database is designed around the ability to scale out, adding additional nodes to increase both data capacity, and query throughput. Within

[20]https://www.mongodb.com/

MongoDB a corpus might be stored as a set of documents including a text field that contains all text within the document;

```
        {
            docID: "ndh1m2h",
            title: "Financial News report",
            text: "Lorem epsom et il..."
        }
```

Listing 2.7: MongoDB document

For storing corpus annotations however it may be more appropriate to treat each JSON document as representing a single word in a token stream;

```
        {
            word: "bank",
            docID: "$ndh1m2h",
            position: 5
            pos: "NN",
            sem: "I1"
        }
```

Listing 2.8: MongoDB token representation

Such a structure again lends itself to being queried as a corpus database but suffers similar pitfalls as with SQL, requiring both the position information of the word to be interrogated as well as requiring multiple queries to fulfil even the most basic of linguistic queries.

A query can be performed in MongoDB by using a Query by Example type form, searching for some token x in a collection "corpus" could be achieved by;

```
        db.corpus.find({word:"x"})
```

Listing 2.9: MongoDB simple query

Using the approach above for storing each word with its accompanying annotations as a separate document would produce similar problems as would be encountered storing all words of a corpus as a relation in a table as in the SQL example above. This would result in further queries being required to retrieve contextual information from around the word to build various query results such as concordances, ngrams, collocations etc. Therefore to retrieve a context around some word at position n in document d would require;

```
        db.corpus.find({
            docID: "d",
            $and[{position: {$gt: n-10}}, {position: {$lt: n
    +10}}]
        })
```

Listing 2.10: MongoDB concordance retrieval

Evidently, the syntax of MongoDB is less intuitive than SQL and would likely be even more challenging for linguists to learn, even those familiar with languages such as CQL.

2.4.2.6 CouchDB

An alternative to MongoDB is CouchDB[21]. Similar to Mongo, CouchDB is a document store that saves data in the form of JSON documents (meaning issues of word level querying remain). Unlike MongoDB, it provides a certain level of ACID semantics on a document level, relying on eventual consistency for collection across replicated nodes. To avoid issues with updates whilst simultaneously accessing data on other nodes, CouchDB implements a version control mechanism rather than locking the database during updates. Since CouchDB's distribution model is a replication-based system it can make use of a map-reduce approach to resolving queries as all nodes will have access locally to all data. This means complex or expensive queries can adopt the map-reduce paradigm for resolution. Using map-reduce to help process and parse corpus data has already been described in section 2.3, but has not been adopted in corpus data systems as a means of resolving corpus queries.

[21]https://couchdb.apache.org/

2.4.2.7 Neo4j

Neo4j[22] is a graph-based database system. Unlike other graph data systems that typically use SPARQL as a query language, Neo4j uses a bespoke query language, Cypher. The query language is written with the intent of being familiar to SQL users but having the power and expressiveness to extract relationships and meaningful links and information from tuples that make up graph data stores. Within graph databases, everything is stored as either an edge, node or attribute and internally these are all stored as tuples (triplets). Language (and indeed any series data) can be expressed in a graph database as a series of tokens expressed as nodes, each token having annotation layers or tags associated with it as attributes and edges connecting the tokens to provide sequences. More complex structures such as the sentence structure and potentially syntax trees can also be expressed in this type of graph structure. On a practical level, the XML structure of many corpora available could be converted and stored in a graph structure that could be queried.

2.4.3 Distribution Methodologies & No-SQL Architectures

Many No-SQL databases are designed for scalability and the modern big data age. They follow various architectures to achieve this, but these primarily involve scaling out across multiple nodes. The three primary categories utilised by existing No-SQL databases are ring, master-slave and replication-based architectures. What follows is a discussion of the benefits and drawbacks fundamental to each of these architectures, where they are used in No-SQL databases, how they are implemented and an analysis of how they might be useful in allowing corpus data systems to be more scalable by adapting and or utilising such approaches.

[22]https://neo4j.com/

2.4.3.1 Master-slave architecture

Master-slave architecture is a common distribution approach and wide-spread amongst No-SQL database technologies[86]. Database such MongoDB[1][23] and HBase[24] use the architecture successfully. In the design, each slave node in a distribution is typically responsible for a single entry or record in the database whilst a master maintains where all records are stored. The advantage of this approach is nodes will only be accessed if the data they hold is contained within the node. The disadvantage being that the master node becomes a single point of failure, as well as the needing to serve all requests to the database, even if only acting as a proxy. Approaches to minimise this downside have been taken; MongoDB, for example, allows for several master nodes that can sit behind a web server that acts as a load balancer. As well as this modern systems that use this architecture often implement replication strategies to ensure the failure of slave nodes does not result in lost data.

2.4.3.2 Replication systems

A replication model where nodes replicate from each is used in some databases like CouchDB. Bi-directional replication strategies mean that systems using this architecture work under the paradigm of eventual consistency. These types of systems are ideally suited to high availability where many writes are expected and consistency across all nodes is less important than the throughput of reads. Replication based systems can be built into tree-like structures using unidirectional replication. This flavour of design does result in a master node at the head of the tree leading to a single point of failure which is typically undesirable. Another disadvantage to replication architecture (particularly bidirectional) is the redundancy of data, whilst a certain amount of redundancy is desirable in distributed systems to allow elegant failover, having all data replicated across all nodes may not be considered efficient, particularly for

[23]https://www.mongodb.com/

[24]https://hbase.apache.org/

huge datasets.

2.4.3.3 Ring systems

Many No-SQL databases, as well as other distributed systems such as DHTs (distributed hash tables), utilise a ring-based peer-to-peer style distribution architecture[66][82]. In a ring distribution, the system is completely decentralized, meaning there is no single point of failure. The nodes in such a system can be thought of as linked to one another forming a ring. Databases such as Apache Cassandra build and maintain their ring using a consistent hashing algorithm. The Chord[84] algorithm describes how a ring system can implement a distributed hash table. Chord describes methods via which efficient insertion, lookup and redundancy can be achieved as well as how to handle node failures. A widely used implementation of this approach is Amazon's Dynamo[29] which uses the ring based approach to achieve a highly scalable key-value store that is used for many large scale web applications such as Netflix[25] and Expedia[26].

2.5 Summary

This chapter has reviewed the various approaches taken by existing corpus data systems to allow corpus linguists to store and query corpora. It has identified the lack of any consistent architecture and the absence of a clear means for off-the-shelf, locally deployable corpus systems to scale out.

This observation motivated the review of alternative methods taken from other database systems, particularly with NoSQL architectures and the indexing compression techniques developed in recent years, both of which could help support a more modern scalable corpus database design. Many of these systems support text data, but do not provide the support for corpus based queries

[25]https://www.netflix.com
[26]https://www.expedia.co.uk/

linguists expect. This paired with the adaptability of paradigms such as document and column stores in NoSQL databases and treating multi-layered annotated documents as a tabular format as demonstrated in iWeb serves as a basis moving forward.

Chapter 3 will explore how these disparate techniques can be consolidated into a coherent architecture for extreme-scale corpus data.

Chapter 3

Architecture

LexiDB is built internally as a distributed column-store but externally exhibits properties of a file store. Although the underlying architecture consists of column stores that are indexed and queried, the externally visible API behaves as if the database were simply a collection of files. This is to allow the database to function as would be expected by corpus linguists as described in the overview of requirements in Chapter 2. As previously discussed, approaches such as those used in iWeb (see section 2.2.1), making use of column stores has been shown to be an effective and scalable approach to storing corpus data. To maintain a familiarity for linguists to other existing corpus data systems already discussed, the interface or API must allow users to treat the database as a file store. LexiDB's design is also distributed allowing for it to be scaled out across multiple nodes and meet the extreme-scale data needs described in Chapter 1. This chapter begins with an overview of the design of LexiDB and then describes in detail the fundamental design aspects of the column stores it uses, the indexing scheme and the distribution methods. The following sections describe each aspect in detail and the key design consideration of each area. How effectively the approach described in this chapter works and how it compares to other systems and approaches is evaluated in chapter 5.

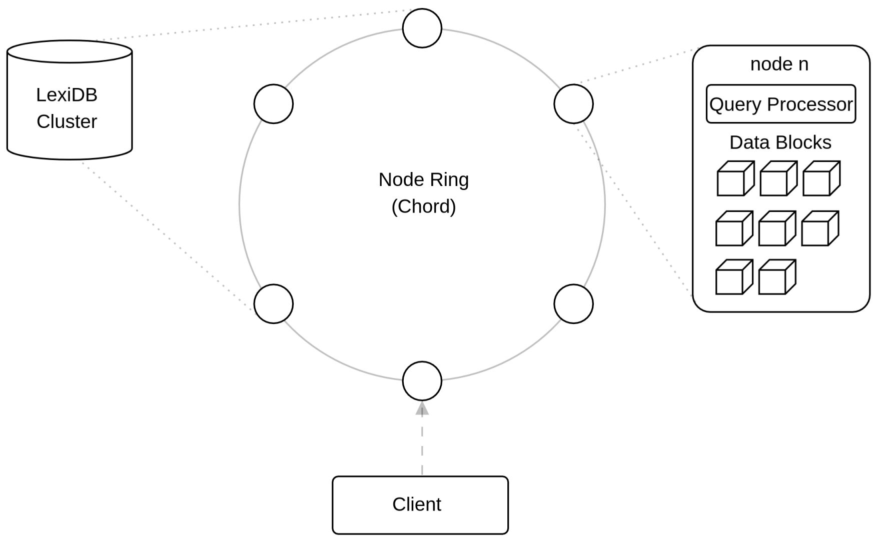


Figure 3.1: Architecture diagram

3.1 Architecture Overview

LexiDB uses a common database idea of splitting various regions of data up and storing them separately. This concept is touched on in Chapter 2 and is used by many modern DBMS such as MongoDB. For the remainder of this chapter, these separate data regions will be referred to as data blocks. Each data block within LexiDB can be treated as its own individual database and so can be queried independently. As a result, these data blocks can be moved between nodes within a distributed configuration easily. Furthermore, each block maintains its own set of columns and indexes. Each block must be queried by the query processor and the results of each collated to create a final result set for any query (this is described in Chapter 4).

Section 3.4.1 describes how the distribution mechanism utilises the Chord[84] algorithm to enable replication of data blocks across nodes. At a higher level, each node can be seen as a query processor that interprets queries and performs them by retrieving data from each block within that node and distributing the query to other nodes in the cluster. A query processor for each node handles querying of the blocks stored on that node, but does not access blocks on other nodes in the cluster. Instead, the queries are relayed to the query processors of each node and the original node that received the query replies with a response to the query. While this adds a level of in-direction to how queries are handled it does allow for individual nodes to be queried and managed separately if so desired and allows for more intricate cluster setups to be managed by a database administrator - this allows for more flexibility to move from single node-setups to ever-larger distributions as needs arise.

3.2 Column Stores

3.2.1 Numeric Data Representation

For storing the values of words and tags within a corpus, a numeric data representation is used. This can be thought of as equivalent to iWeb's relational model discussed in section 2.2.2. Expressed in classical relational terms any sequence of words (or tokens) W can be expressed as a pair of relations. R_1 containing all token types within the corpus and can be thought of as a dictionary and R_2 containing an ordered appearance of those token types within the corpus D. Using this model R_1 need only contain two columns, one to represent the value of the token type and the second a key. R_2 can contain a single column which contains a foreign key from R_1. R_2 can also contain additional foreign keys to other tables/relations potentially containing metadata.

R_1s keys can be compressed to the smallest number of bits possible e.g. in the British National Corpus (BNC) $W = 100,000,000$ and $D = 760,000$ ($132 : 1$ ratio). This means that rather than using a 32 or 64-bit integer for the key an unsigned 20-bit integer is sufficient; $2^{20} = 1,048,576$. This allows for roughly 37.5% saving in space from a 32-bit integer key and 70% saving compared to 64-bit integers. Whilst at a conceptual level this may seem trivial, at larger scales such space saving is significant both for saving on disk usage and for improving query performance by effectively reducing the amount of data that needs to be read, reducing the dependency of performance on disk speed.

This basic structure allows for corpora to be expressed in terms of these two simple relations however it does not take into account that the rate of growth of R_2 is far faster than R_1 as the size of the corpus grows. Typically the number of records in R_1 will be several orders of magnitude smaller than R_2. In a classical SQL DBMS both relations might be treated like tables, but the high usage of R_1 means it is better suited to being held using some kind of in-memory data structure rather than retrieved consistently from disk. The rapid growth in size of R_2 as corpora grow means it is not feasible to hold in memory in the same way as R_1.

This conceptual idea is realised in the design of two specific column store types. Zipfian column stores and continuous column stores each of which is specialised for storing a particular form of data, textual and corpus metadata respectively. These column store types are novel in both name and function to LexiDB.

3.2.2 Zipfian Columns

As the name suggests, Zipfian column stores take advantage of the Zipfian nature of corpus data. The design of these column stores encapsulates the conceptual relations R_1 and R_2 described above.

When creating a Zipfian column store, it is necessary to perform two parses of input data. The first parse is intended to compute R_1 or the dictionary for the corpus. Once all token types have been found they are then sorted lexically (alphabetically) and their keys assigned in order using the minimum numbers of bits required. The data is then re-parsed and each token is stored on disk (in R_2) using the foreign key from R_1. In practise this multi-parse approach can be improved upon by creating an initial lookup trie for the dictionary D and assigning an initial set of values for token types and storing the token stream using these values on the first parse. Then subsequently sorting D and assigning new key values and creating a mapping from the original values to the final values. This eliminates the need to perform any trie lookups on the second parse and simply use random access lookups in from the key map to associate the tokens to the final token-type key. A sequence of tokens {*"one"*, *"and"*, *"only"*, *"one"*} would initially be mapped to the numeric sequence $\{0, 1, 2, 0\}$ with a dictionary $\{0 \rightarrow$ *"one"*$, 1 \rightarrow$ *"and"*$, 2 \rightarrow$ *"only*$\}$, this dictionary sorted lexically would yield $\{0 \rightarrow$ *"and"*$, 1 \rightarrow$ *"one"*$, 2 \rightarrow$ *"only"*$\}$ with a mapping from the original dictionary to the final being $\{0 \rightarrow 1, 1 \rightarrow 0, 2 \rightarrow 2\}$ thus the final numeric values are computed as $\{1, 0, 2, 1\}$.

This multiple parse approach has several advantages and disadvantages. The primary advantage is that the minimum number of bits can be precisely determined and as a result, maximum compression can be achieved. The main reason for using a single bit length and not variable bit length (which could achieve greater compression) is to allow for the token stream to be treated as fixed-length records, thus allowing for easier lookups by allowing easy random access and the ability to skip bytes in the stored file without the need to read what occurs previous to the desired location. For other languages with longer words, this fixed-length approach will still work as the only thing that will result in longer records will be bigger and bigger vocabularies.

The obvious disadvantage is the time taken to parse the data twice. This is alleviated somewhat by the temporary conversion of the data to intermediary numeric values as described above. However, this does not entirely compensate for the cost of multiple parses. An alternative to multiple parses could be to split the textual range into buckets and for each subsequent character in each token assign a range of characters based on how likely the character is to occur, similar to arithmetic encoding[76]. Using this approach would mean the benefits of compression and random access, gained through fixed-length encoding, would be lost.

3.2.2.1 Annotation layers

In practice when looking at multi-layer annotated corpora, multiple Zipfian column stores can be used. Each column store can contain a single annotation layer e.g. part-of-speech tag (POS), semantic tags etc. However, in some circumstances, this may lead to a level of redundancy. Across multiple annotation layers, a certain level of "Zipfianess" is often maintained, for example, if multiple POS tagsets are used one would expect many of these to align. A perfect example of this would be between token-types and lemmas. Typically the same token-type would always share a lemma. The design of the column store means that R_1 can be extended to include multiple columns. In this way rather than storing an entirely separate column, two annotation layers can be combined. Using the previous example of the BNC, which contains 605,000 lemmas, rather than storing these separately to the tokens, they can be combined and a second column store can be eliminated. In a traditional relational table, these would need to be stored in a separate column or incorporated using a second table and a key pairing creating additional redundancy. In many cases, various annotation layers can be combined but it is necessary to understand the data being used and how the annotation layers relate to each other. When combining columns in this way, individual values for lemmas and tokens etc. can still be matched in the same way as before because they will each have their own lookup tree.

When multiple annotation layers are stored in a single column store, the numeric values that are assigned are based on sorting across the summation of these layers. When storing tokens and part-of-speech tags in a single column store, the numeric value w_1 representing $\{token : bank, pos : noun\}$ would be less than w_2 representing $\{token : bank, pos : verb\}$, $w_1 < w_2$. This means it is important to consider how combining multiple annotations or columns into a single column store will affect the database's ability to sort lexically based on numeric values. In the mentioned example, sorting on the first column *token* will still work roughly as expected but sorting on the second column *pos* would

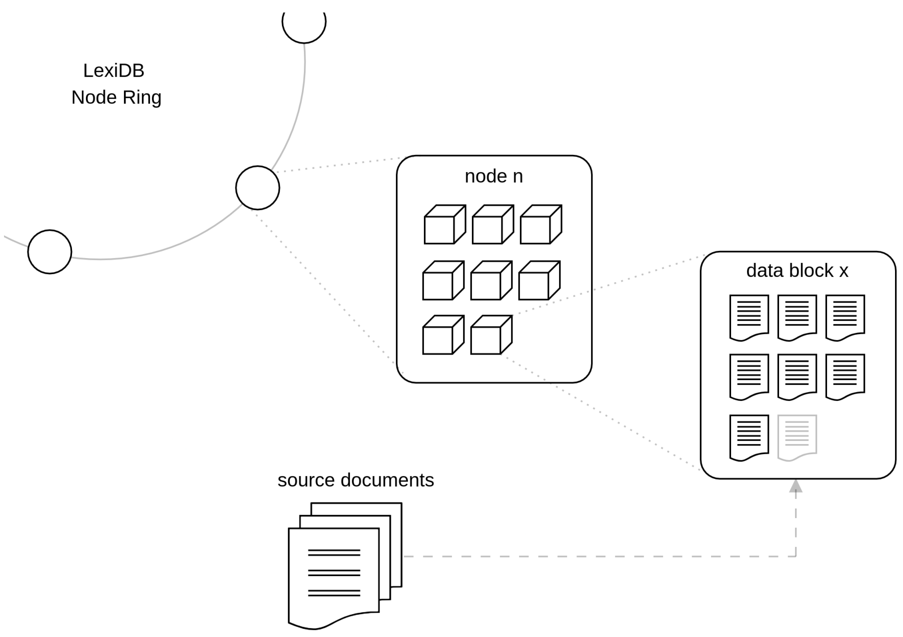


Figure 3.2: Inserting Documents into Data Blocks

not be possible using the numeric values - to sort on this column would require the query processor to resolve the numeric values first and then manually sort on the string values which would be far less efficient. This is a choice any user of LexiDB would need to make when deciding whether to combine columns into a single column store; will those columns need to be sorted separately?

An additional meta-annotation layer that is added to the dictionary for each column store is frequency information. This frequency information is simply a count of how often that value (or set of values) occurs within the particular data block. This information is essential to compute ahead of time so that it can be retrieved quickly when performing queries requiring word count or other frequency information such as generating word lists or calculating collocation metrics such log-likelihood or mutual information. The frequency information is held in memory as an array so the numeric value can be used to look it up quickly at query time when needed.

3.2.2.2 Insertion and Deletion

Figure 3.2 illustrates how source documents are stored in data blocks when inserted into LexiDB. Each data block within LexiDB will contain a set of source documents in their entirety, no source document is split between two blocks. When adding data to a column store it is necessary to finalize the block (i.e. sort the dictionary and calculate final numeric values for the data) before the block can then be queried. Typically if a data block has a size limit of 10,000,000 tokens (this limit can be set through configuration) and a single large corpus is being added in a single insertion, as each block reaches its limit it can be finalized and saved to disk. However, if the corpus is not static and is slowly built up over time it may be necessary to finalize and serialize the block to disk multiple times before this limit is reached. To allay this problem the dictionary and data for the current block is cached in memory, this allows for them to still be queried after each insertion without needing to wait for the disk serialization to occur. This separates LexiDB from other existing corpus management systems as it allows for live data to be inserted on the fly and does not rely on the entire corpus being static or available. Corpora can be built within LexiDB and added to over time, a key point for storing social media streams, for example.

In support of this, it is also necessary for data blocks within LexiDB to be combined. Particularly when the desired token limit for each block may be changed at any given point during the database's lifecycle. In addition to this, the greater number of blocks introduces an overhead in storing separate dictionaries. For example, when storing the BNC as a single block the dictionary length for the tokens column was around 760,000 but storing the same corpus as 12 separate blocks resulted in each dictionary being on average 180,000 (180,000 x 12 = 2,160,000) resulting in almost (2,160,000 - 760,000) 1,400,000 redundant dictionary entries when blocking the database in this way. Merging blocks can eliminate this problem. Merging blocks can be done via a simple scanning process, reading each record in turn and adding it to a new temporary block before removal of the original blocks.

To support deletions and allow LexiDB to function as a fully-fledged DBMS supporting both static and dynamic corpora, LexiDB maintains a delete list in the form of a single bit length column store, effectively a bitmap, keeping track of deletions. Although it might seem logical to keep track of each record or token for deletions, this is not practical as for performance sake it is important to maintain the deleted list in memory and serialize to disk when possible. This would mean for a large corpus of say 1,000,000,000 tokens a bitmap of 125 MB (1 bit per token) would need to be maintained. When considering that this bitmap will need to be persisted to disk after each deletion operation it is not practical to keep it in memory. To avoid this problem, and to keep in line with the facade of LexiDB acting as a file store, a bitmap of deletions is kept based upon files inserted. Even if the files added to a corpus are relatively small, say 100 words on average, this would still reduce the size of the bitmap by two orders of magnitude making it relatively inexpensive to persist to disk regularly.

In practise this delete list is held alongside a continuous column store (described in section 3.2.3) containing the filenames. Filenames are the primary piece of metadata that are stored regardless of whether specified in the schema being used for the corpus. Whilst maintaining this list allows for deletions to be quick (similar to file systems keeping a free list of disk blocks) it does mean data can become fragmented within column stores and result in wasted disk space. This is resolved when blocks are merged or split as described above. During merging and splitting of blocks the deletion list is consulted and the records of deleted files are not copied over to the new block. This can be seen as a defragmentation[1] process that occurs during merging and splitting of blocks. In practise a block can be de-fragged by performing a merge task but only including a single block.

[1]Typically this is a process whereby file systems reorganize the contents of a storage device to contiguous regions.

3.2.2.3 Implementation considerations

From an implementation perspective, the above design of a Zipfian column store is stored as two separate files; a binary file (*.dat) which will contain the token stream as a set of fixed bit length integer values and a tab-separated values (TSV) file to store all of the entries in the dictionary (*.dict). The TSV file can be used to store multiple columns if the column store contains more than one column e.g. token and lemma (as described above). The bit length used in the binary file is not necessary to store as this can be computed by the number of entries in the dictionary file with the intention being that the dictionary file will be stored in memory for ease of lookup and retrieval. When loaded into memory the dictionary file is stored in two forms, firstly as a radix trie to facilitate index lookups (described later in section 4.4) and in the form of an array to allow for reverse lookup up of column values based on their numeric representation within the binary *.dat file. It is essential to keep the dictionary in memory as it is impossible to predict ahead of time which values may be required or the ordering of such lookups. Whilst lookups of the binary data may be retrieved and fulfilled in the order they occur in the corpus, dictionary lookups, particularly when considering a contextual window around a search term may require any values to be retrieved at any time. Whilst this in-memory storage may be an expensive memory requirement, as described above the dictionary file is typically several orders of magnitude smaller than the whole corpus and it is therefore usually practical to store the dictionary in memory.

When multiple annotation layers are contained within a single column store, a radix trie for each annotation layer or column is built. This means each column can be searched independently even if they are part of a column store containing other columns. In this way, column stores or sets within the design can be thought of as synonymous with column families in other DBMSs. The radix trie will return the numeric value (or values) for the lookup key (typically a word, linguistic tag or regular expression). Conversely, the values stored within the array lookup for random access contain all column values so only a single array is needed.

3.2.3 Continuous Columns

To store certain types of metadata continuous column stores are utilised. This is specifically for metadata that remains consistent for many records e.g. the speaker within a transcription document will remain the same for many tokens at a time. In a conventional relational model, this type of data could be stored using a separate relation R_m and then including a key from R_m as a foreign key in records within R_2 (the token stream). This creates the problem of having many duplicate keys repeated record after record in R_2 and an unnecessary join to be performed between R_2 and R_m during any data retrieval. An alternative, to eliminate this join, would be to include the metadata directly in R_2 but this still leaves many duplicate data values in this column in R_2 and in fact makes the problem worse as the data may take up more space than the foreign key of R_m and increase redundancy even further.

To resolve this, continuous column stores are used to store metadata that does not frequently change in their own column alongside the token stream. To achieve this, the design of a Zipfian column store is combined with run-length encoding. Run-length encoding allows for data values to be stored in a dictionary R_1, similar to Zipfian stores, however now the token stream is stored as the numeric value from the dictionary followed by a run-length for the data value. One primary example where this will always be used is to store the file name within a corpus, as well as things such as the author, title, date, origin etc.

Whilst this approach saves significantly on space for such metadata, there are some caveats to be considered. Firstly in this context, it is no longer feasible to provide a means of random access lookup within a continuous column store. Much like with variable length encoding it is now essential to read all values prior to the desired lookup point. For example, looking up the value at position 100 in a continuous column store may be the first data entry or the 10th or the 100th, there is no way to know without reading all previous values and summing the run-length of each until that value at position 100 is found. Whilst this presents a limitation and would be harmful if this type of column store was used extensively on inappropriate data, typically for long-running metadata the total number of values is so small it would not present a large performance impact to scan and sum previous entries. For example, in the BNC although the token stream is 100,000,000 the total number of files (filenames stored in a continuous column store) is only 4050 (25,000:1) so scanning such a comparatively small list of values is reasonable, even for larger corpora this is still reasonable (this is evaluated in section 5.3.1 using a version of the Hansard corpus with nearly 9 million files). Obviously, metadata may not always be at such a ratio compared to the length of the token stream but typically is several orders of magnitude shorter in length such that the space saving from storing in this fashion out-weighs the cost of scanning and summing the run-length encoded values.

Similarly to Zipfian columns, continuous column stores can use variable-length bits. After the first parse of the data is completed, the maximum bit length for both the number of values in the dictionary and the maximum run-length for any occurrence of those values can be computed. A minimum bit length needed to store the largest value can then be used. This allows two integers of this known bit length at a time to be read when scanning and summing to find a value at a desired position. The bit length can then be unary encoded at the beginning of the run-length encoded file or explicitly stored in a schema file for the data block.

3.3 Indexing

To facilitate fast lookups, a postings list for all the values in each column store is built upon insert. The principles of postings listings is described previously in Section 2.4.1.1. Within LexiDB the postings lists for column stores are stored using PFOR delta indexing. Traditionally DBMSs would prefer to store their indexes in memory to allow them to be accessed quickly when performing queries and lookups. The scale of data that this design is built to cater for means that the index fitting in memory cannot be assumed. The indexes are as such kept on disk and only the posting lists required whilst executing a query are loaded. PFOR delta is used as it offers one of the highest levels of compression of the current state of the art indexing schemes. This compression is important not necessarily for saving disk space but to ensure that the overhead of reading the index data from the disk is minimised.

Each posting list stores the locations of the value (or values) within the column store for that data block. Each data block contains their own postings list. All postings lists are stored in a single file. This file itself is then indexed and stored in an index lookup file. When a postings list is required the value being searched for is retrieved from the index lookup file and this gives the offset within the main lookup file from which to load the postings list. All postings lists are maintained within the same index file as a query may require several postings lists e.g. when resolving a regex there could be hundreds of postings lists required. Storing the postings lists in this manner means only a single file need be open (two counting the index lookup file) to retrieve all the index information required.

Much like with the bit packing applied to the main data storage within columns, the index lookup file is compressed in a similar way. Similarly to the data files, the index lookup file cannot be compressed using variable-length bit encoding as when performing a lookup for a particular numeric value, the ability to perform random access on the file must be maintained to ensure the entire file need not be read in order to find the location of a postings list in the main index file. This limitation need not apply to the index file as it is assumed that the entire postings list for any particular value being looked up would be desired.

Simple lookups at the level required by individual data blocks, i.e. looking up a value in a particular column, is handled by simply returning the position information from the index. To perform a lookup using a data block's index, firstly the column store being queried for must be identified by its key. The value being queried for is then looked up in the column store's index which will return a set of numeric values that represent the value being searched for. The position of these values postings lists are then retrieved from the index lookup file, subsequently the postings lists are returned to the query processor (described in Chapter 4). It may seem logical to combine these postings lists and sort them at this point. Instead the postings lists are returned including the numeric values for which they represent for the query processor to allow the query processor to handle cases such as regular expressions or multiple column values from a single column store.

3.4 Distribution

To facilitate scalability, LexiDB is designed so it can be deployed in a cluster configuration. This means the database can be scaled out, potentially allowing for larger and larger corpus datasets. The architecture of this distribution operates on two principles. The first is to ensure that queries can be sped up as a result of the distribution, effectively turning the query processing into something akin to map-reduce. The second is to ensure redundancy of the data between nodes so if one node fails another will be able to take its place without any loss of access to data stored on the failed node. This redundancy is achieved through the use of a modified Chord algorithm applied to the data blocks LexiDB uses for storage. The effectiveness and scalability of the distribution methodology described here is evaluated in section 5.2.2.

3.4.1 Distributed Querying

When a query is received by a particular node, the API requires that it be flagged as a local or distributed query. This allows the query process to handle it in one of two ways. For local queries, the data blocks stored locally are queried and the results returned to the caller. For distributed queries, the same occurs but the query is also forwarded to all nodes in the cluster (but flagged as local), this can conceptually be thought of as mapping in the map-reduce paradigm. The original node that received the distributed query then becomes responsible for aggregating the results from all other nodes in the cluster and returning these results to the original caller, or reducing if keeping to the previous equivalency. No one node within the cluster is treated as a front-end or outward-facing node. All nodes are equal and any can be queried.

The ability to query nodes locally is enabled within the API to allow for database users to better understand how and where their data is stored. In practise, it may be desirable to place a clustered configuration of the database behind a web server that can act as a load balancer. Although distributed queries are forwarded to all nodes, the aggregation step always means the node originally receiving the query has more work to do, particularly if the query involves some kind of post-processing such as sorting or grouping. A front-end load balancer could then be utilised to ensure that all nodes have an equal share in this overhead.

To make better use of the design's distributed nature, results are sorted using an approach that has been termed a latent distributed merge sort. The general principle of this is for a given query the results available on each node are compiled and sorted locally and the final stage of the sort essentially becomes the final step of a merge sort with all sub lists having already been sorted. The advantage of this is that the final stage, drawing together the results from all nodes, can be completed lazily as results are paged. In practice, this means that the top x results from each node can be skimmed and sorted in order to create the first final results page of x items without the need to complete a full final stage of the merge sort. The merge sort can be progressed by requesting the next page of results through the API.

Query results across nodes are cached locally. The final aggregated results, however, are always dynamically computed, this ensures that a single machine (serving an initial client query) is not memory starved by a query which would return a huge result set, such as querying for a highly frequent term within the corpus or potentially a regular expression matching numerous less frequent terms. This does mean the latent sorting approach described above overrides this behaviour, as otherwise it would be necessary to sort the result set from the beginning whilst paging through results. Unsorted results can maintain this behaviour, the server responding to the client query can simply page through the results from each node in turn without having to concern itself with ordering.

3.4.2 Redundancy

To enable a clustered configuration of LexiDB to be fault-tolerant to node failures, a simplified version of the Chord algorithm approach is taken. While Chord can handle storage, distribution, queries and redundancy only the redundancy aspects and parts of the distribution approach (as it relates to redundancy) apply to LexiDB. Upon insertion to a clustered configuration, as each data block is inserted at a node, a hash is computed based on the file names contained within the block using SHA-1. During insertion time the node handling the insert operation will distribute the files that are being inserted between all nodes in the cluster in order to maximise insertion speed and minimise the time taken to index and finalise all the blocks. When each block is completed it is shared to the adjacent node in the outer ring (as specified by the Chord algorithm). As such when the adjacent node detects that the node for which it holds the backup blocks has failed it can make them available itself (this will also trigger a further backup to the next adjacent node). The default number of hops to store a backup around is one (i.e. a single backup), but this can be changed if the probability of node failure is particularly high, of course this can lead to potential duplication of data if only a small cluster is being used where the number of nodes n is proportional to the number of hops to backup along h, $n \alpha h$.

3.5 Summary

In this chapter, we have discussed the architectural paradigms that have been applied to the approach utilised by LexiDB. This has been the first stage towards answering research question 1 from section 1.4 - how can database and IR techniques be used to build a corpus data store?

An architectural pattern based upon numeric representation similar to that of existing relational models utilised in other systems was proposed but modernised to bring it in line with more modern No-SQL data management approaches such that it is not constrained as much by ACID but adheres more towards CAP theorem as discussed in section 2.4.3. This allows for the architecture to be distributed whilst utilising an existing approach (Chord) to ensure redundancy. Unlike existing approaches of other systems such a design can scale to handle corpus data by scaling out (across multiple nodes), not merely scaling up (using more powerful hardware) which is the only feasible solution to handle larger corpora in other, off-the-shelf, corpus data systems.

In chapter 4 how this architecture can be used to handle typical corpus queries is examined and the effectiveness of the distribution methodology is evaluated and compared to other existing distributed DBMSs in chapter 5.

Chapter 4

Querying

4.1 Introduction

Within existing systems, and considering corpus analysis techniques as a whole, there is a particular distinction to be made between querying and results types. The same linguistic query may be used e.g. searching for a word matching a regular expression i.e. `[token = run.*]` (CQL syntax) but various result types may be returned e.g. concordances, collocations etc. This creates a clear separation in the tasks required to be performed by a database management system capable of fulfilling these kinds of corpus queries between searching the data to find the results and processing and formatting these results in a way that is meaningful to corpus linguists, this separation also means future techniques developed for corpus analysis can be easily adapted for. For the remainder of this chapter, the distinction will be made between querying i.e. looking up data values in the database and processing these results into a linguistic result type. This will result in a minimisation of the number of distinct query types there are compared to the number of result types (although a database user may not be aware of this distinction).

Query by example is a typical form of querying classically in databases [93]. Traditionally such a querying technique would involve a user interface and (typically) a graphical form to fill in. This form would include all the columns or fields available in the database table being queried and allow an end-user a simple way of finding what they are looking for. This method also allows for complexities such as performing joins and having an understanding of the underlying database schema to be abstracted away from the user and instead relies on the interface possessing an understanding of the user's workflow and typical tasks. This is also the case when considering corpus data systems, the structure of the underlying data is hidden from an end-user and the system is tailored to a particular analysis. This can lead to disparate, bespoke systems that can only be used for a single purpose, for a single data format or that are only compatible with certain other tools in a corpus processing pipeline (see Chapter 2). Most existing corpus data systems can be thought of or related closely to query by example in the way that they allow users to perform searches, whether this is in a graphical interface or employing a specialised corpus query language.

Result types returned by corpus systems are often varied but can be reduced to several primary types which will be discussed and considered in this chapter. Whilst each of these linguistic result types may have many specialised variants, one of the most crucial things for a DBMS designed to support them is the flexibility to return as much data as is necessary. The intent being that any highly specialised corpus result type (or indeed unknown or future types) can still be supported with the need of nothing more than a thin application layer on top of the data management layer to further process the data returned. For example, calculating various collocation metrics can be done from a contingency table and word frequency lists, therefore it is more essential that a corpus DBMS supports these as a return type than supporting every known collocation metric. These data consideration are discussed further in section 4.6.

While querying and result types will be the primary considerations within this chapter and how these processes can be optimized for language data, it is essential for corpus query systems to also return metadata regarding results. This metadata typically will be in the form of data regarding a source file e.g. filename, origin etc. but may also include data regarding the author or speaker of the text that is a hit in the query. Whilst author or speaker may simply be considered part of the file metadata it could be metadata within the file itself as there may be several authors e.g. in transcription documents. Other metadata may also include section information from within the file or potentially cross-reference information. As corpus datasets get ever more complex some of this data may even include URLs and semantic web links to external sources. Whatever the form of metadata, it is necessary for a modern DBMS for corpus linguistics to be able to handle these facets of corpora gracefully.

4.2 Linguistic Query Types

When considering the capabilities required of any query language designed to meet the requirements of corpus linguists, the form in which results of such queries will take is important. As opposed to typical lists of records or documents as would traditionally be returned by classic and modern DBMSs, linguists expect results to be returned in 1 of 4 (or 5) primary forms; concordance lines, n-grams, collocations and frequency lists. These are the four query types supported by LexiDB. Keywords could be considered in this list as well but can be considered an application-level concern (discussed below). Each of these four return types has different consideration & implications for how the linguistic query could and should be expressed as well as how the query processor should execute and return the results of such a query.

Concordance lines are the most common and straightforward of return types for linguistic queries. They are analogous to keyword-in-context (KWIC) searches and in most cases can be treated as such. Typically such searches will have a hit region beyond a single word and as described above the search criteria will go beyond simply the text content of the word to include multiple levels of tagging and annotation, these tags and annotations must, of course, be returned in the results.

It can be helpful to view a set of concordance lines as a list of lists of records, where each record represents a single token, each list of tokens represents a single concordance line and the list of concordance lines is the complete result set. This allows for the consideration of sorts and joins within the result sets. Whereby, for example, a join might be executed with some metadata table against a concordance line as opposed to each individual record. Likewise, for sorting, it would be improper to sort all records within the results sets but simply to sort concordance lines within the result sets i.e. taking every word from different concordance lines and combining them into a large set of sorted words would be meaningless since the results would be jumbled. Other systems such as AntConc[5] allow for concordance lines to be sorted alphabetically using tokens to the left or right of the hit region (see section 4.8). LexiDB also supports this.

Frequency lists usually consist of frequency tables of tokens that can be queried, potentially using regular expressions and return a list of several items from this frequency table. Frequency tables are often computed at insertion time based on a corpus' dictionary, where each atomic value (word, tag annotation etc.) has its frequency maintained. Beyond simply comparisons of term frequencies, frequency lists can also be used to calculated keyword metrics, when comparing two sets of frequencies, one from a target corpus and one from a reference corpus. Although this is a fundamental feature of many corpus tools, it is simply a metric that can be performed by comparing two sets of frequencies and will be considered as such an application layer consideration, not a database management system concern i.e. if the frequencies from two corpora can be easily retrieved through the data layer; it is trivial for an application layer to calculate a keyword metric.

N-grams within corpus linguistics are usually used up to 5-grams. Typically n-grams are pre-built within systems and data sets. An example of which would be Google Books n-gram corpus[1] where pre-processed n-grams and their frequencies can be viewed and searched. However, a limiting factor of pre-building n-grams is the inability to view n-grams around a particular search phrase or sequence as well as a limitation to the maximum size of n that may have been arbitrarily decided beforehand. The ability to dynamically and quickly compute n-grams based on a typical corpus query search i.e. for a phrase or POS sequence would represent a significant capability beyond that offered by existing systems and is supported by LexiDB.

Collocations are a more subtle query type to resolve. From the perspective of a DBMS, calculating metrics such as log-likelihood and mutual information scores appears to be more of an application-driven consideration and should not be perceived as a data layer requirement much the same way as keyword metrics. Unlike keyword metrics, though it is often a more involved process in calculating metrics, another single return type from the database cannot provide all the necessary information necessary to calculate many of the collocation metric calculations discussed in Chapter 2. It can be assumed that such calculations would be commonplace in any application built upon a corpus database it is essential that such a database can provide a means to easily retrieve the data necessary to make these calculations i.e. a contingency table. Providing a query result type such as this the final stage of calculating a metric is trivial and would always be expected thus two common metrics are built into the databases query processor. An alternative to this would be to provide a programmer like environment or API to allow for lambdas to be injected into queries so that the required metrics can be calculated by the database. This may or may not lead to improved performance when making calculations on extremely large data sets, but it most definitely would lead to increased technical understanding required by the database user.

[1]https://books.google.com/ngrams

4.3 Overview of Query Syntax and Capabilities

The query syntax utilized by LexiDB is based on JSON (consistent with many modern systems discussed in 2.4.2). The use of JSON allows modern applications - particularly those based around web technologies to interface easily with the API. An example of a typical query in LexiDB is illustrated in Listing 4.1. This shows how a query is formulated specifying both a pattern to be queried for *query* and stipulations for the return type *result*. Full specifications for the JSON query format can be found online[2]. This section will discuss primarily the query syntax and how it relates to the resolution and retrieval of query results.

```
{
  "query": {"tokens": "{'pos':'J.*'}.{'pos':'NN.*'}"},
  "result": {"type": "kwic"}
}

```

Listing 4.1: LexiDB query example

Within the *query* portion of the JSON, a particular syntax is used. This syntax can be treated as a regular expression over a token stream or record stream. The key specifies the table within the database to query and the value is a pattern to be matched against. This pattern replaces how characters are typically used in regular expressions with QBE (query by example) JSON objects. These objects allow for a token or record to be searched for based on values of given columns or fields, these values themselves can be specified by regular expressions. This syntax allows for complex linguistic expressions to be searched for within the database and expressed succinctly.

The example from Listing 2.4.2 can be interpreted as searching for;

{'pos':'J.*'} Any token whose POS (part-of-speech) field begins with *J* (for

[2]https://github.com/matthewcoole/lexidb/wiki/Query-Syntax

the C7 tagset [3] an adjective).

. Any token.

{‘pos’:‘NN.*’} A token with a POS starting with NN (a noun).

This illustrates how typical regular expression special characters can be used like `.` where instead of matching any character this matches any token or record. Other special regex characters can be used as well such as token quantifiers `*+?{1}`, unions `|` and character classes `[]`. The effectiveness of this query syntax in resolving linguistic queries is discussed in Chapter 5.

In this query syntax, values of Zipfian columns and continuous columns can be queried for. So, as well as being able to search for particular tokens or part of speech tags, you can also search on metadata such as date, author name etc.

4.4 Resolving QBE objects

The simplest of corpus queries may be the lookup of a particular token type (i.e. a word or part-of-speech tag) which can be expressed in the syntax described above as a QBE object. Any QBE object that can be expressed as a set of key-value pairs $Q = k_1, k_2...k_n$ and can be resolved by taking each key-value pair k_n in turn and compiling a DFA based on the value (using an existing regular expression library) and resolve against a radix trie T built from the dictionary of the column indicated of the key of k_n. This approach is described in algorithm 1 and closely mirrors the approach taken by Baeza-Yates[7] utilising a more general radix trie as opposed to a patricia trie and applying it to dictionary entries rather than sistrings derived from full documents.

[3] http://ucrel.lancs.ac.uk/claws7tags.html

4.4.1 Preamble

Take any regular expression that can be converted to a non-deterministic finite automata (NFA)[61] and subsequently an equivalent DFA[14], represented as $M = (Q, \Sigma, \delta, q_0, F)$. Take a dictionary of token types D and compile them into a depth reduced radix tree $T = (R, E, L)$ where R is the set of all nodes in the tree numbered in a breadth-first traversal $r_0, r_1 ... r_n$, E is a set of all edges in the tree expressed as a tuple (r_i, I, r_j) where I is the input sequence from node r_i to r_j and L represents a set of leaf nodes $L \subset R$ and corresponds to the set of token types in the dictionary D.

4.4.2 Algorithm

```
input  : A DFA $M = (Q, \Sigma, \delta, q_0, F)$
         Radix trie $T = (r_0, R, E, L)$
output: A set of integer pairs $h$
1  $h \leftarrow \emptyset$
2  $k \leftarrow \{(q_0, r_0)\}$
3  while $k \neq \emptyset$ do
4      $(q_i, ri) \leftarrow k.pop()$
5      for $e$ in $r_i$ do
6          if $\delta(q_i,e) \in F$ then
7              $\delta(q_i,\text{e}) \cup h$
8          else if $\delta(q_i,e) \in Q$ then
9              $(\delta(q_i,\text{e}), R(r_i, e)) \cup k$
10     end
11 end
```

Algorithm 1: Algorithm for resolving a DFA over a radix trie

4.4.3 Example

Take the regular expressing `^[ab]a.*$` represented as a DFA in Figure 4.1 and the dictionary of token types $D = \{abb, aab, abba, aabb, baa\}$ represented as a radix tree in Figure 4.2.

1. Initialize k and h as empty. $k = \{\}, h = \{\}$

2. Add (q_0, r_0) to k. $k = \{(q_0, r_0)\}$
3. Pop (q_0, r_0) from k. $k = \{\}$
4. Take each edge leading from r_0; (r_0, a, r_1) & $(r_0, baa\$, r_2)$.
    (a) From (r_0, a, r_1) pass a into the DFA from state q_0.
    (b) State is valid - add (q_1, r_1) to k. $k = \{(q_1, r_1)\}$
    (c) From $(r_0, baa\$, r_1)$ pass $baa\$$ into the DFA from state q_0.
    (d) DFA in accept state - add r_2 to h. $h = \{r_2\}$
5. Pop (q_1, r_1) from k. $k = \{\}$
6. Take each edge leading from r_1; (r_1, ab, r_3) & (r_1, bb, r_4).
    (a) From (r_1, ab, r_3) pass ab into the DFA from state q_1.
    (b) State is valid - add (q_2, r_3) to k. $k = \{(q_2, r_3)\}$
    (c) From (r_1, bb, r_4) pass bb into the DFA from state q_1.
    (d) DFA cannot progress - stop.
7. Pop (q_2, r_3) from k. $k = \{\}$
8. Take each edge leading from r_3; $(r_3, b\$, r_5)$ & $(r_3, \$, r_6)$.
    (a) From $(r_3, b\$, r_5)$ pass $b\$$ into the DFA from state q_2.
    (b) DFA in accept state - add r_5 to h. $h = \{r_2, r_5\}$
    (c) From $(r_3, \$, r_6)$ pass bb into the DFA from state q_2.
    (d) DFA in accept state - add r_6 to h. $h = \{r_2, r_5, r_6\}$
9. k is empty. STOP. $h = \{r_2, r_5, r_6\}$

Given the use of algorithm 1 as $f()$ to resolve each key-value pair of a QBE object $Q = \{k_1, k_2 ... k_n\}$ within a query to a set of sets of integers representing index positions $S = \{s_1, s_2 ... s_n\}$ the intersection of these can produce the final

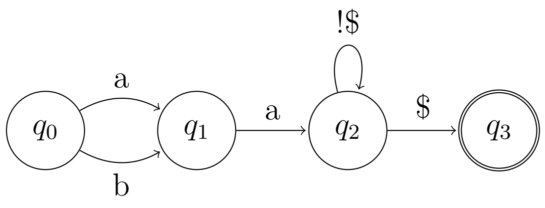


Figure 4.1: DFA representing `^[ab]a.*$`

-

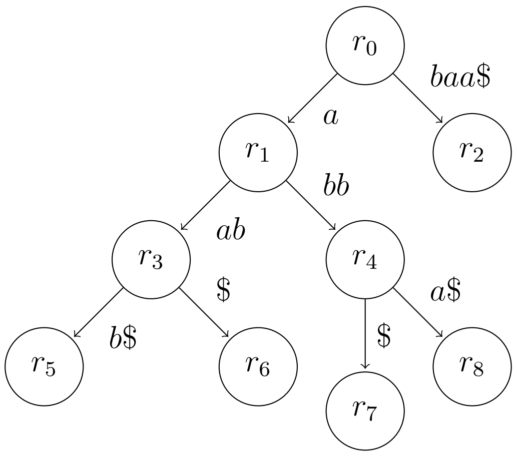


Figure 4.2: Radix tree representing D

set of integer positions that represents the index positions of that QBE object;

$$\bigcap_{i=0}^{n} s_i : s_i = f(k_i) \tag{4.1}$$

This will produce the set of integer positions that can be used to resolve the QBE object. The procedure for handling queries with multiple QBE object and resolving regular expression operators at the token level is discussed in section 4.5.

Whilst the approach described in Algorithm 1 efficiently resolves QBE objects to index positions given a dictionary trie of index positions there are certain limitations inherent to it. Primary among these is its inability to support certain common regular expression functions such as look-aheads and look-behinds. Although these functions are common in many styles of regular expression syntax and in various libraries, they are not supported by any known libraries which compile regular expressions to DFAs and so are also not supported in LexiDB. However this approach does allow for potential future developments of new regular expression libraries which may support these operations in the form of a DFA to be used as alternatives, thus this limitation is more tied in nature to the capabilities of DFA based regular expressions rather than a flaw in the approach itself.

4.5 Token stream regex

To resolve regular expression type queries over token streams it is necessary to provide a method of resolving the operators i.e. Kleene star etc. Other systems such as CWB have used CQL as a query language supporting similar regular expression operators but have needed to implement their own query processor specifically to resolve the queries. The approach presented here contrasts to this as it can use any existing regular expression library to resolve regular expression like queries over an indexed token stream. Beyond this it is also capable of resolving such queries with no need to inspect the actual data, only the indexes are used to resolve the expressions. This gives the approach an advantage over that taken in existing systems as less data (and therefore less disk access/memory caching) is needed to compute query results.

At the heart of this approach is converting a query made up of QBE objects (as described above) and regular expression operators into a form that a regular expression engine can resolve. This leads to a query language of types which is not designed but formed through the merging of JSON syntax for describing QBE objects and standard regular expression syntax. A simple example of a query in this form will look familiar to those used to CQL `{"pos":"JJ"}+` would search for a series of one or more adjectives (JJ = general adjective C7 POS tagset). To do this, it is necessary that the regular expression engine creates a DFA that can be interrogated by the below algorithm and its state transition table accessed. To allow a DFA to be built by any such engine, each QBE object in the query is converted to a character representation, as described below. In the implementation of LexiDB, the regular expression library used for resolving token stream queries is dk.brics.Automaton[65] engine because it is one of the faster[4] libraries that compiles regular expressions into DFAs. As with resolving QBE objects, support for other operations can be added by swapping out the library for another library that includes the desired operation (provided the library compiles regular expressions to DFAs).

4.5.1 Preamble

Take any query Q made up of a set of QBE objects $a_0...a_n$ and regular expression operators. Convert these QBE objects to pairs of character representations ($a_0 \rightarrow$ 'a', $a_1 \rightarrow$ 'b' etc.) and resolved index lookup positions within a corpus ('a' = 0,12,15...). Map these characters onto their respective index positions in a sparse 2D matrix C representing the entire corpus. Replace the QBE object in the original expression with their character mapped representations and construct a DFA; $M = (Q, \Sigma, \delta, q_0, F)$.

4.5.2 Algorithm

4.5.3 Worked example

Consider the token stream regular expression $q_0{*}q_1$ where the index lookup results of resolving the QBE objects are $q_0 = 0, 1, 5, 6$ and $q_1 = 1, 4, 5$ in some corpus of length 7. The token stream regex can be converted to a typical character regex $a{*}b$ and the sparse 2D matrix representation of this corpus can be built as such `[[a],[a,b],[],[],[b],[a,b],[a]]`. Passing this through algorithm 2 would produce $h =$ `[(0,1),(1,1),(4,4),(5,5)]`.

[4]https://tusker.org/regex/regex_benchmark.html

```
input  : A DFA $M = (Q, \Sigma, \delta, q_0, F)$
         Sparse 2D matrix $C$ of length $l$
output: A set of integer pairs $h$
1  $h \leftarrow \emptyset$
2  for $i \leftarrow 0$ to $l$ do
3      $S \leftarrow [(q_0, i)]$
4      while $S \neq \emptyset$ do
5          $(q_n, k) \leftarrow S.pop()$
6          for $j \leftarrow 0$ to $C[k].length$ do
7              if $\delta(q_n, C[k][j])$ then
8                  $S$.push($\delta(q_n, C[k][j])$, $k+1$)
9              end
10             if $\delta(q_n, C[k][j])$ in $F$ then
11                 $h$.add($(i, k+1)$)
12             end
13         end
14     end
15 end
```

Algorithm 2: Algorithm for resolving QBE based regular expressions over a token stream

4.5.4 Limitations

The primary limitations of this approach are that the query language is limited by the capabilities of the regular expression engine employed, and more specifically by the typical drawbacks of DFA based regular expression resolution. Using DFAs to resolve regular expressions is very efficient but certain capabilities, as mentioned look-aheads and look-behinds are not supported by engines that use DFAs. This means that LexiDB does not support these operations either. However the versatility of this approach means that if an engine capable of supporting this wider range of operation using finite state machines was developed in future, it could easily be used as the engine for LexiDB and the approach described in Algorithm 2 would still work as intended, leading to a wider range of operations being supported.

Another less obvious drawback to this approach is the expressiveness of QBE syntax in this form. Each QBE expression must be an atomic object in its definition, i.e. it cannot refer to other QBE objects e.g. looking for repeated words. To achieve this form of processing, an additional resolution layer would need to be added to the query processor, most likely this would take the form of a post-processor to resolve such operators after results sets have already been returned which would make the query processor far closer to CQP in its operation and as a result would potentially create a large performance dip as all hits (or potential hits) in results sets would need to be retrieved before complete query resolution could occur. This would go against the primary aim of this approach which is to be efficient and scalable for extremely large corpus datasets. To maintain this, some limitations to querying capabilities (at least in comparison to CQL) are acceptable.

4.6 Resolving Query Types

Once query resolution has been performed by the above algorithms and a set of hits h has been compiled, the results must then be processed into a meaningful form and returned by LexiDB's API. The result types described above (concordances, n-gram) can be thought of purely as contextual. To resolve them and compute a set of results, only proximal information around the hits is needed i.e. a context window. For collocation and frequency lists additional information regarding the dictionary of values and their frequency is required or can be used to improve the efficiency at which result sets for these query types can be computed.

Concordance lines can be easily computed by retrieving all numeric values surrounding each hit and displaying the results to the user. Due to LexiDB's column storage architecture and random access ability, retrieving spans of tokens around hits is straightforward; however large results sets will still require substantial disk reads. This can and is mitigated somewhat in the LexiDB approach by automatic supporting of paging, meaning only a specific number of concordances are returned to the user as it is unlikely users will want to view thousands or indeed millions of concordance lines in one go. For example, the previous sample expression `{"pos":"JJ"}+` returns 100+ million concordance lines in the Historical UK Hansard corpus of 1.6 billion words. More likely such queries would be refined and/or sorted (see section 4.8).

Likewise, NGrams can be computed by first retrieving a set of all concordance lines within a context window of $n-1$ such that all NGrams that are computed will include at least a single token from the hit (whether the hit is a single token or several). Unlike with other systems, NGrams can be computed on any column or annotation layer e.g. a common phrase can be queried for and a set of NGrams based on POS tags can be returned rather than NGrams based on tokens. When retrieving trigrams for the word “colourful” a context window of $2(n-1)$ will be used and once all concordances are retrieved and the trigrams computed such that “colourful” will always be in the trigrams, whether as the first, second or third word. Other systems may compute NGrams ahead of times which can allow them to be retrieved more rapidly but this allows searches on NGrams in LexiDB to be far more complex as typically other systems do not store beyond 5-grams. Using this approach any value for n can be used and the hit being searched for can be a mix of different annotation layers, as well as the hit being as long as desired. A limitation to dynamically building NGrams is that they cannot be paged to reduce disk reads; all spans around the hit term must be retrieved for the list of NGrams to be computed.

Frequency lists can be retrieved by the query processor by two means. When a single token is searched for the frequency lists can be accessed directly by retrieving the frequency information stored in each column's dictionary e.g. searching for `run.*` can be resolved to a series of numeric values on the token column and the frequency of each (which is held in memory) can be immediately returned. For frequency lists generate from a query with multiple tokens, the frequency is computed in a similar way to NGrams, the entire set of concordances is retrieved (with a context of 0) and then the frequency of each hit based on a particular column is returned e.g. `{"pos":"JJ"}{"token":"house"}` might return; `[big house: 15, tall house: 7....]`.

Calculating collocation metrics is perhaps the most involved of the return types supported by LexiDB. The collocation metrics available in LexiDB are Log-Likelihood and Mutual Information, how each of these metrics is calculated is described in Chapter 2. To perform these calculations much like with NGrams the proximal tokens to the search term must be retrieved and summed (similar to calculating unigrams). To calculate Log-Likelihood for example, a contingency table for each term must be created, which can be done easily once a set of unigrams around the hit term has been found be accessing the frequency information for each term. Like with NGrams, retrieval of the spans around the hit terms cannot be paged; all results must be retrieved first to construct the contingency table. Once this table is built for each term the collocation metric desired can easily be calculated. Much like with other query return types, LexiDB goes above and beyond the capabilities of other systems, as any column can be used to calculate collocations i.e. rather than looking at collocations of a particular search term based on words, LexiDB can compute POS collocations. This can lead to further information extraction from a corpus, for example, using metadata one might query for `{"speaker":"Mary"}+` in a corpus with multiple speakers per file, collocations based on the `speaker` column could then reveal who speaks before and after Mary more typically based on a collocation metric.

4.7 Asynchronous Querying

In the previous sections, querying was discussed from a synchronous perspective, the database is queried and the results are retrieved and computed before being returned to the user in whatever form they requested. However, this poses an issue for distributed systems. As discussed in Chapter 2, CAP theorem tells us that the availability of every node in a cluster cannot always be guaranteed, beyond this there is also a huge advantage to be gained in potential responsiveness in leverage the potential higher availability of certain nodes and more specifically the ease of compatibility of results on certain nodes as a result of general data dispersion. Asynchronous querying allows for a set or subset of intermediary results to be returned to the user to reduce perceived latency of the database whilst allowing for any additional latency that might be incurred from the compilation of full results sets across multiple nodes in a distributed environment to be transparent (or at least close to transparent).

When querying for a simple set of concordance lines LexiDB will return the results in a page of x results. If the first data block queried, n, contains at least x results $x_n >= x$ then the first page of results can be computed directly from this set prior to querying additional data blocks, assuming no sorting is required (this is discussed in section 4.8). Synchronously this would not be possible with potentially dozens or even hundreds of data blocks left to be queried. Asynchronously querying allows for this first set of results to be returned as soon as it becomes available which can massively reduce query response time compared to synchronously querying particularly with large result sets. The reduction in initial query response time is evaluated in section 5.2.1. For other result types the initial results from a single block are returned as estimates. For example, a collocation query will return the results as if the contingency table was constructed from only the first block queried; these results will subsequently change as the contingency table is updated with results from additional blocks.

Along with the first page of results, the total number of results is returned with an indication of the number of pages available. When querying asynchronously the number of results and pages will not be known at this point, but an estimate can be calculated based on the number of results found within the data block. As each data block is queried the result set displayed to the user can be asynchronously updated, if unsorted the initial page of results will not change but the estimation of the number of results and pages will be updated. This estimation assumes that the dispersion of the query term between data blocks is fairly uniform. In practise, cases where dispersion is far from uniform between data blocks results sets are typically smaller and as a result the full set of results will be retrieved fast enough as to not make a poor estimate of the result set size on the initial asynchronous return of the query to make a significant difference on corpora in the order of several billion tokens. It should be stressed this estimate is not expected to be utilised; it is to give users an indication of the result size whilst waiting for slower queries to return their final results

Whilst this form of asynchronous querying is simple to process for unsorted concordance lines, the processing overhead to perform searches in this way increases when considering result types that go beyond the retrieval of data to the compilation of information e.g. frequencies, computed metrics etc. For simple frequency lists similar to unsorted concordances a simple estimation can be used to extrapolate frequencies across the whole corpus. This method again has the potential to give poor estimates, but is intended as nothing more than an indicator to the user when awaiting final results.

Frequencies when calculating ngrams can also be retrieved asynchronously in much the same way as frequency lists for individual terms or phrases. The process of computing ngrams is more involved than for simple frequencies but no more overhead is incurred than when performing a query for the frequency of phrases rather than individual values i.e searching the phrase “far from *” cannot simply be looked up in the frequency tables for the words themselves. In the case of ngrams the results must be sorted by frequency after each data blocks results are retrieved and added to the aggregate results as usually users will want to view the most frequent ngrams first.

Collocation metrics can be returned asynchronously as well assuming that only estimations are required initially and that the size of the data blocks is large enough to be somewhat representative of the corpus as a whole such that collocation metrics themselves may change between blocks but perhaps the ordering will remain fairly consistent. As such collocation metrics are built a block at a time, unlike with simply collating results (as with concordances and frequencies) metrics must be recalculated as each block is queried and the contingency table from the most recently queried block is collated with the aggregate table. This process is straightforward for the collocation metrics supported by LexiDB (log-likelihood and mutual information) and the calculations are trivial to handle. Greater cost is incurred in the constant re-sorting that will be required as each block is queried (this is discussed in section 4.8).

4.8 Sorting

Part of resolving linguistic queries will inevitably involve some form of sorting of results. Whilst sorting of small sets of results is trivial, it is more likely the case that larger result sets will be those most desired to be sorted by linguistic database users to make better sense of the results and to put them to better use in further linguistic analysis. What follows is a discussion of the techniques employed in LexiDB to handle sorting of various types of linguistic queries, how they can be applied more effectively in a distributed environment and a discussion of their strengths limitations.

Perhaps the most typical form of sorting used by many concordancers is a lexical sort of concordance lines. This involves sorting concordance lines alphabetically based on a position relative to the hit term. This may simply be sorting on the hit itself (if the search were a regular expression say, rather than a single word) but other tools allow you to sort on several positions (given an order of precedence) left or right of the search term. To facilitate this in a database environment it is necessary to retrieve the linguistic data required to perform this sort, this occurs on two levels; firstly within each data block the numeric values at the positions specified by the sort can be retrieved and sorted, second, the values in the sort positions across data blocks must be combined and sorted - this cannot be done by a simple comparison of their numeric value (since the values will differ between blocks) but by resolving the values to text and sorting the result. Whilst other tools can sort on the word level LexiDB can sort not only on this but on any level of annotation layer e.g.

POS tags, semantic tags etc. as well as on any level of metadata e.g. author name, text origin etc.

Whilst sorting on the final values of the text (which may be words, linguistic annotation or metadata) will never be as fast as simply sorting numeric values, this final step can be sped up by latent sorting of the results as and when they wish to be viewed by the user. Since the results within each block will already be sorted (by sorting the numeric values) the first page of x results can be computed simply by skimming off the first page of x results from each blocks result sets and performing a final sort on the textual values from this set. In this way, the first page of results is guaranteed to be present and be computed quickly and displayed to the user without the need to combine and sort the entire result set. This can be thought of as a lazy merge sort. Whilst this may be a less efficient method to sort the results, in a distributed environment and particularly one that can make use of asynchronous querying (as described above) this can mean a far a higher level of perceived responsiveness to the end-user which would be expected in more modern IR systems, but less common from classical databases.

Beyond a lexical sort of concordance lines, frequency sorting is also possible in LexiDB. A frequency sort allows concordances to be sorted by how frequently a term occurs e.g. a search for “habit” could be sorted by the most frequently occurring term on position to the left i.e. the word before. Unlike lexical sorts which can utilise a lazy merge sort mechanism, there are no short cuts that can be taken to improve query response times for a frequency sort. Within block results can be computed by summing the frequencies of numeric values however between block results (including across nodes in a distributed setup) must be computed in full as there is no guarantee that the top x results from within each separate block will contain the top x results summed between all blocks. Sorting concordances based on the frequency of nearby terms is not something supported by any other known concordancer.

As described previously, frequency lists for particular words can be retrieved by accessing the in-memory dictionary of each data block and computing the total sum between the blocks for each word. Generally, frequency lists may be generated based on a regular expression looking for a particular word, but LexiDB can also generate frequency lists based on a query pattern over several tokens. To sort frequency lists generated in this way the same approach as with frequency sort of concordances is utilised but the sort term is simply the hit item i.e. the database is effectively performing a concordance search with a context window of 0 and sorting the results. Once again generating a frequency list across blocks, all results must be summed before being sorted. The exception to this is when querying asynchronously where the frequency lists are sorted as each blocks frequencies are added. Much like frequency lists, ngrams, when dynamical computed in LexiDB, are sorted using a merge sort based on their frequency.

Collocation results when being computed asynchronously have several methods available for sorting. The default approach is for the results to be sorted using a merge sort according to the specified collocation metric as each data blocks results are computed. Whilst this results in multiple sorts (sometimes redundant), the overhead is generally acceptable as this pales to insignificance when compared to the overhead required in retrieving the necessary data to create a contingency table and compute the collocation metrics as described above. When not querying asynchronously, collocation results are simply sorted when the collation of all contingency tables is complete and the final collocation metrics have been calculated.

4.9 Summary

This chapter has explored various techniques and approaches that can be used to query the data structures described in chapter 3 to satisfy corpus queries and has presented two novel algorithms. The contributions of this chapter are in fulfilment of sub-objective of research question one in chapter 1 - are there any un-tapped methods that can be developed into novel solutions in this problem domain?

The first algorithm presented here is strongly based on existing work and allows a character level regular expression that is built as a finite state machine to be resolved over a dictionary of strings of a corpus that are expressed as a radix tree. This is needed for resolving linguistic queries to identify patterns within tokens, whether it be words themselves or some other annotation layer within the data. The second algorithm presents a method for resolving regular expressions at a token level, which has been demonstrated as a requirement of corpus data systems in chapter 2, using purely the indexing data (described in chapter 3) of the corpus. This allows the positions that match the expression to be found without the needed to scan the underlying token stream itself, whether in full or in part.

Chapter 5 will evaluate the effectiveness of the approaches described in the previous chapters. The methods presented in this chapter and how they compare to existing systems is discussed in section 5.2.3.

Chapter 5

Evaluation

5.1 Overview

In developing an evaluation methodology for the techniques and approaches used in the design of LexiDB it was hoped inspiration could be drawn from other work on the evaluation of similar corpus data systems. However, there are scarce attempts at quantitatively evaluating many of the existing corpus data systems, and particularly little using a comparative methodology. One of the few such examples of this is the work by Meurer[63] quantitatively comparing Corpuscle to CWB for a small sample of custom queries picked to demonstrate the strengths and weaknesses of each system. In light of this, the evaluation methodology presented here is split into two distinct approaches. The first to explore research question 2 (1.4) of how corpus data systems and databases can be evaluated quantitatively involves three experimental setups that test the developed prototypes for LexiDB, examine the scalability of the system, examine the scalability of other DBMSs and finally compare the query performance to an existing corpus data system and a common indexing tool used by other corpus data systems. The second phase of evaluation was to perform a case study that involved constructing a new large scale corpus based on parliamentary data, building a web interface around the LexiDB API, and conducting a focus group to explore qualitatively how useful LexiDB is to corpus linguists and how easily it can be utilised as a data platform on which corpus tools can be built. This case study serves as a proof of concept, that LexiDB can be used as a platform upon which to build corpus data systems, as opposed to serving as a direct evaluation of the database itself.

5.2 Quantitative Evaluation

The quantitative portion of the evaluation is divided into three experimental setups. The first experimental setup tests the scalability of two existing DBMSs on a cloud computing platform when utilising corpus data described in section 5.2.1. The second experimental setup examines the scalability of a prototype of the LexiDB system on the same cloud platform. Finally, the final prototype of LexiDBs query processor is compared in experiment 3 to an existing corpus data system (CWB) and a commonly used indexing system, Lucene. Each experiment is described in turn beginning with an overview of the experimental setup, followed by a brief discussion of the results along with the implications of them concerning other systems and approaches. This quantitative evaluation can be thought of one side of a coin of a methodology evaluating corpus data systems that could be used as a basis to evaluate other systems in the future (although the experimental setups may be refined or changed based on requirements or based on deeper insights that may hereafter be gained into the nature of such systems).

5.2.1 Experiment 1: Scalability of existing DBMSs

5.2.1.1 Setup

To test the capabilities of modern DBMSs, two such systems were deployed into various clustered configurations and loaded with an extremely large corpus dataset. The DBMSs tested were MongoDB and Cassandra. Each database

Low Frequency	Medium Frequency	High Frequency
gauntly	weeny	it
croquet	kilometers	I
patronym	plebs	is
ratpayers	appraiser	a
thugutt	earldoms	in
ogies	candlemas	and
fecias	laudations	that
gacious	coachmakers	to
unspared	heinkel	of
moyland	conegate	the

Table 5.1: Keyword frequencies

was deployed onto the Amazon Web Services (AWS) EC2 platform using m4.xlarge instances (4 vCPUs, 16GB Memory, 100GB EBS Volume -500 provisioned IOPS). Each database was deployed in 4, 8 and 16 node configuration so that the scalability of each DBMS when storing corpus data could be examined. The corpus dataset loaded into the databases was the Hansard corpus (1.6 billion words). Each database was then queried for particular keywords and all hits of the word in the database were returned in the form they are stored, i.e. as a document or record representing an instance of the queried word. The keywords that made up the queries were derived from pre-generated word lists (each keyword query was run 5 times and an average taken). To gather an accurate reflection of each DBMSs capabilities queries were split into three groups; High frequency — the top 10 most commonly occurring words in the corpus. Medium frequency -– 10 randomly selected words from the 60% range of words. Low frequency -– 10 randomly selected words from the bottom 50% of words in the corpus.

5.2.1.1.1 MongoDB A minimal cluster configuration for MongoDB requires 3 components; a central query processor front end server, 3 configuration servers and a data node (shard). In practise many of these components can run on the same server. For testing the central query processor (mongos instance) plus the 3 configuration servers were deployed to a single AWS instance. Each data node was then deployed to its own separate AWS instance. This was deemed acceptable as keeping the configuration servers separate would be the best practice to ensure some redundancy in the cluster but this was only a test environment. Having the mongos server on the same AWS instance was also deemed acceptable as the tests are not designed to stress the database in terms of a high number of queries per second but rather how the database handles a large amount of corpus data. Three configurations were tested with 4, 8 and 16 data nodes.

The schema followed for MongoDB was for each word in the corpus to be inserted as a BSON (Binary Javascript Object Notation) document as described in Listing 5.1. The JSON document below describes the BSON documents inserted into MongoDB (note that the form of the word is not lemmatised in these tests);

The “docid” field is utilised as the shard key to ensure not only that the data is distributed evenly between the shards in the cluster but also to ensure when querying to build a concordance line for an occurrence of a word, one data node should be able to return all the BSON documents necessary to build the concordance line. A text index is defined in MongoDB on the “searchableform” field to search for word instances. A hashed index is automatically built on the “docid” field to distribute the data but this index is also used when performing a ranged query to build concordance lines.

In MongoDB, data distribution is handled by selecting a shard key from within the data field. This key will be used to determine which data node the BSON document should reside on. Initially the source document ID was selected as the field for the shard key - this would ensure that all words from the same original source document would be present on the same data node - thus making it easier to build a concordance line as for each individual line all the word BSON documents could be gathered from the same node. The incremental nature of the source document IDs meant using this field as a shard key lead to slower parse times because the MongoDB cluster was consistently shuffling data around, between the nodes, to ensure that the data was distributed evenly.

To remedy this an artificial shard key was created. This shard key was a randomly generated number between a group of pre-defined ranges. Key range chunks were set up prior to parsing on the MongoDB cluster to define which range of keys to handle. Each alternate BSON document was then assigned a random value in this range when parsed to ensure during the initial insertion of the data the writes were distributed evenly across the cluster.

```
{
  //unique automatically generated id
    _id: ObjectId("5553324ca7986c0c3d6b3a97"),
    //word as it originally appeared
  originalform:  The     ,
    //word trimmed of white space and forced to lower case
  searchableform    the   ,
    //id of the source document the word appeared in
  docid:   S6CV0196P0  -00481   ,
    //position within the source document
  pos: 103
}
```

Listing 5.1: JSON word entry in MongoDB

5.2.1.1.2 Cassandra Cassandra can be configured into a cluster almost as easily as a standalone server can. Since it is designed with distribution in mind and makes use of several p2p architecture principles such as seeding and the equality of nodes each node need only be configured to point to a single commonly known set of nodes (seed nodes) from which configuration about the cluster is retrieved. As before Cassandra was deployed in 4, 8 and 16 node configuration onto AWS instances. During testing an additional seed was added per 4 nodes. Meaning the 4 node configuration contained 1 seed, the 8 node configuration 2 seeds etc. The seeds are of course nodes themselves and behave in just the same way as the rest of the configuration except they also provide management information to the rest of the cluster.

Although it is considered a No-SQL database Cassandra's query language CQL

(Cassandra Query Language) bears many similarities to SQL and simple commands and queries will often look identical between the two query languages. The schema used again followed a simple one word per record form and is described in listing 5.2.

```
CREATE TABLE hansard.words (
  doc text,
  pos int,
  s_f text,
  o_f text,
  PRIMARY KEY (doc, pos)
);
```

Listing 5.2: Cassandra CQL Schema definition

Similar to MongoDB the schema stores 4 values per record. "doc" the original source document, "pos" the words position in the source document, "s_f" the searchable form of the word (lower-case, removed punctuation), "o_f" the word as it originally appeared in the text. The primary key used is a composition of the source document and the position of the word in the source document - guaranteeing its uniqueness. A secondary index is then built on the "s_f" column to allow for keyword searches.

Cassandra allows for insertion to be performed and targeted at each node. This allows multiple large batch insertions to take place on different nodes. Cassandra uses its SSTable loader tool to import large quantities of data quickly. Prior to insertion the Hansard corpus was converted from its original XML to this SSTable format to allow for quicker insertion into the database.

5.2.1.2 Results and Discussion

5.2.1.2.1 MongoDB Fig. 5.1 shows the average query times retrieved using MongoDB for the sample tokens listed in table 5.1 for 4, 8 & 16 node cluster configurations. Here the results show the time to return all hits of a word across all nodes in the cluster. As would be expected based on other work around big data and the concept of scaling out the average query time improves significantly as the number of nodes in the cluster doubles - further parallelizing the index lookup and retrieval tasks. The graph also illustrates how, despite the queries limiting the number of results returned to 20, the query time actually increases with the frequency of the word in the corpus. This implies that the text index used by MongoDB requires a significant amount of time to read the inverted file entry when it has performed the lookup - if the entry in the index is larger because the word being looked up contains significantly more occurrences then the query response time will be significantly impacted.

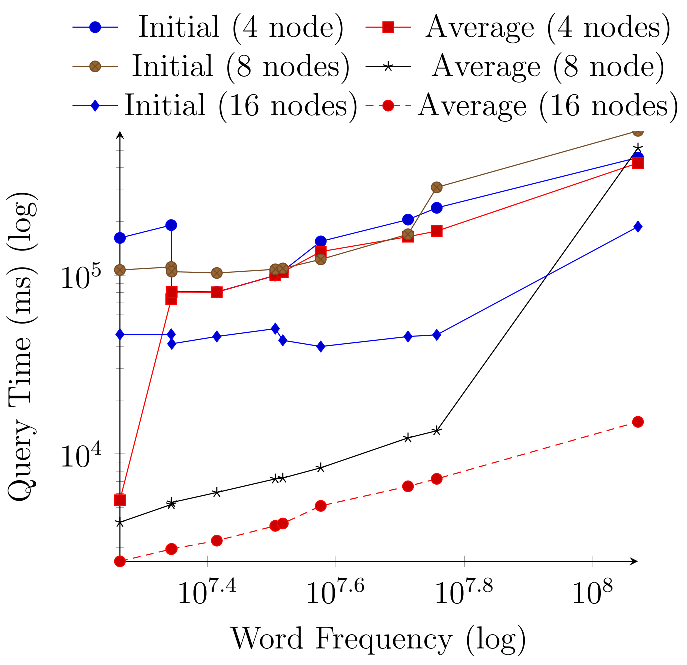


Figure 5.1: MongoDB Query Times (High frequency words)

Interestingly the raw numbers (see appendices) illustrated a pattern whereby the first test run of each query for high frequency words would be significantly slower (by an order of magnitude) than subsequent queries for the same word. This served to push up the average query times reported in Fig. 5.1. This pattern is likely the result of the text index being too large to contain wholly in memory meaning that when the text index is first searched it is read from disk and subsequently the entry for the word is then cached in memory. It is still however important to consider the figures whilst including this initial slow query as when working with big data sets such as Hansard it is likely indexes will not fit entirely into memory and will be read from disk.

A further anomaly observed in these results is the longer query time for the most frequent word "the" when querying on an 8 node cluster. Whilst typically this could be heralded simply as an outlier the method of gathering these numbers through the average of several tests runs suggests this is not the case. Furthermore the point raised above of the initial high query time for the first query in the run could have contributed to push the average query time higher than that of a 4 node cluster for the same word however the raw results consistently showed longer query times across all test runs for the word "the".

The results of the low and medium frequency words are difficult to see in the form shown in Fig. 5.1 and are presented expanded in Fig. 5.2. For the case of low frequency words that occur only once in the corpus it seems that increasing the number of nodes in the distributed setup has no clear effect on the retrieval time. This can also be seen looking at the retrieval times for medium frequency words with no obvious performance improvements gained by increasing the number of nodes in the cluster. Whilst for low frequency words occurring only once this might not be too interesting as a corpus linguist would likely need an even larger corpus to perform a meaningful concordance analysis with these words - the medium frequency words do have enough results for some kind of concordance analysis to be performed which shows that at this range of word (i.e. a minimum level for a concordance analysis) on this scale of corpus there is little to no benefit of large cluster setups with dozens of machines.

These results also further indicate the idea of a strong correlation between query times and index entry size - clearly when the index entry size is small enough very little variation can be seen in query time even if this index is distributed and the lookup is taking place in parallel. This further supports the need for better indexing techniques to be developed for large text indexes which cannot fit into memory to allow for simple queries to be fulfilled more readily.

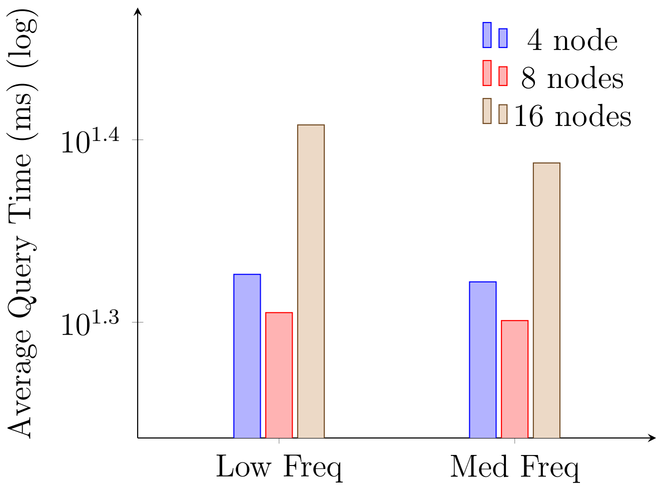


Figure 5.2: MongoDB Avg. Query Times (medium & low frequency words)

For simple queries such as those utilized in these experiments a strategy of index entry paging and lazy loading could be utilized.

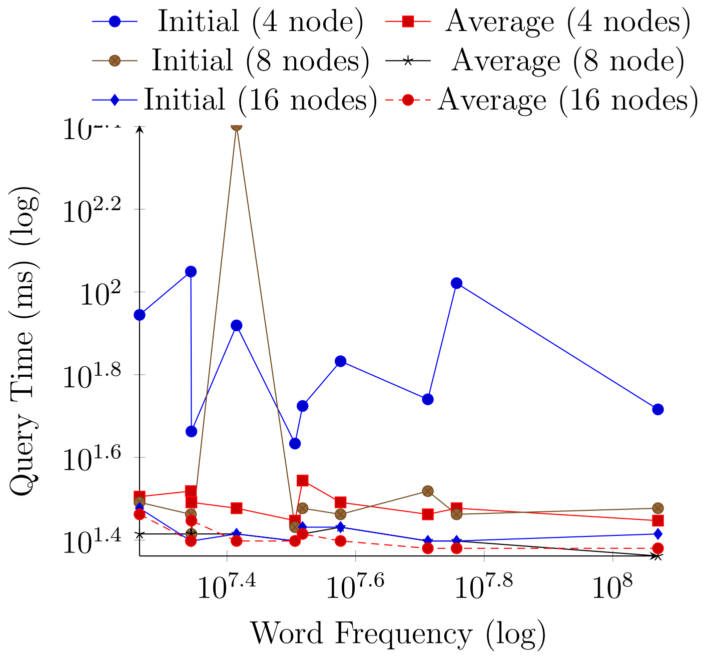


Figure 5.3: Cassandra Query Times (High frequency words)

5.2.1.2.2 Cassandra The results for Cassandra (Fig. 5.3) are immediately interesting as the raw figures seem to illustrate little to no correlation between the word frequency and the query time for the simple query being performed by our tests. Unlike with MongoDB where as the frequency of the word increased so to did the typical query time (suggesting increased time required to read the index as discussed above) Cassandra appears to demonstrate minimal overhead to reading large index entries for highly frequent words. In exploratory testing it appeared that Cassandra is far more influenced by the total number of results which are returned, indicating that it is capable of reading its index in a far more efficient way than MongoDB (possibly by only reading the required initial part of the index). This means that for simple queries such as those performed during testing which limits the results set to just 20, Cassandra was able to return results with far more consistent times across all frequency ranges than MongoDB.

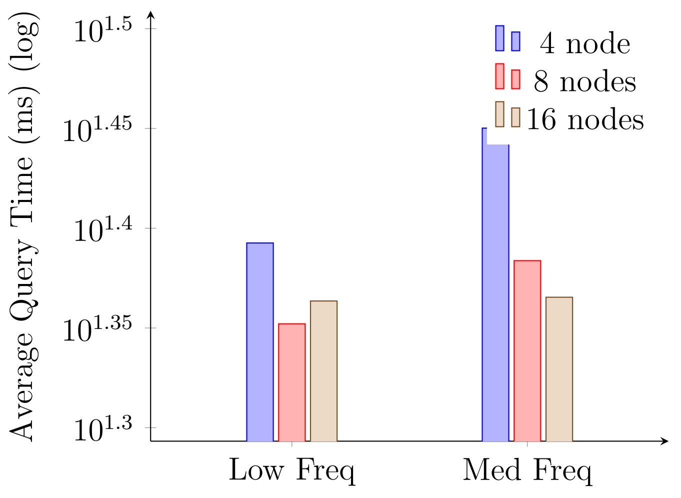


Figure 5.4: Cassandra Avg. Query Times (medium & low frequency words)

Fig. 5.4 shows the average query times for each word group listed above (low, medium & high frequency). From these results it is clear to see that for the scale of the Hansard corpus on the AWS infrastructure used there was a noticeable increase in performance between a 4 node and 8 node cluster configuration. However there was a negligible difference seen between results between the 8 and 16 node configurations suggesting that at this data scale there would be little to no benefit of scaling out such a configuration any further - albeit for these relatively simple corpus queries. It should be noted that some queries were marginally faster on the 8 node configuration than the 16 node configuration, although the difference in many cases was less than 1 ms so may be explained by any cluster management information shared on the network between the nodes at the time of the tests.

Examining the raw results for Cassandra it also becomes clear that there is no initial index read period as noted in the discussion of MongoDB's results above. Perhaps due to Cassandra's seemingly better handling of indexes for these simple queries it is not burdened by the need to load the index entry into memory from disk and therefore the significantly longer query response time seen on the first query for each word in MongoDB is not seen at all when querying Cassandra. This would mean that in any system put in place in the real world for say a corpus search tool users would not experience significantly slower response times for a search on a keyword that has not been searched for before - or recently enough for its index entry to still remain in memory as in the case of MongoDB.

5.2.2 Experiment 2: Scalability of LexiDB

To demonstrate the scalability of LexiDB, we conducted a series of test queries on one, two and four node distributed configurations using two, billion token scale corpora - Historical Hansard (1.68 billion tokens) and EEBO TCP phase 1 texts (0.91 billion tokens). Each of these corpora were tagged with lemma, POS, HT (Historical Thesaurus) and USAS semantic tags and stored in the form of `*.tsv` files. KWIC(concordances), NGram(bigram) and collocations(log-likelihood) queries were all performed using the ten most common words in the corpus. Each query was performed ten times and a mean query time was found. The most common words were chosen as this represents a worst case scenario for each query i.e. a query that has the most results to retrieve and it will likely be the most computationally and hard disk intensive and memory consumptive. All LexiDB instances were run on AWS m3.2xlarge VMs (8 vCPUs, 30Gb RAM, 2 x 80Gb SSDs).

Figure 5.5 shows the time taken to insert and index each corpus on the three

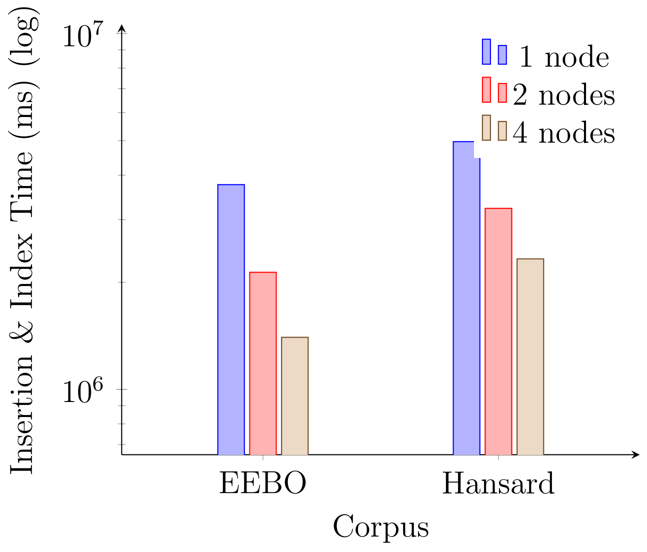


Figure 5.5: Insertion and Indexing

database configurations. The increase in speed is clearly evident as LexiDB is scaled out to multiple nodes, allowing for ever faster insertion times. This insertion operation is a one time process that a user of LexiDB would need to perform just once to execute queries against a static corpus.

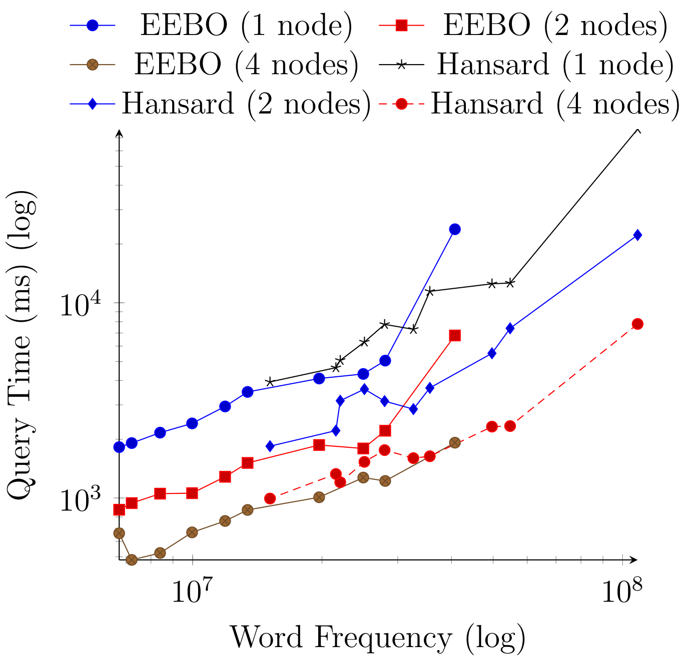


Figure 5.6: Concordance Lines

The ten most common word types in each corpus used as search terms were {the, of, and, to, in, that, a, is, it, his} for EEBO and {the, of, to, that, and, in, a, I, is, not} for Hansard. Figures 5.6, 5.7 and 5.8 shows the result for the respective context sensitive queries (Concordances, Collocations and N-Grams) using these lists of words. Because each corpus varies in size and the frequency of the words differs, the results are presented as the average query time against the frequency of the word type within the corpus.

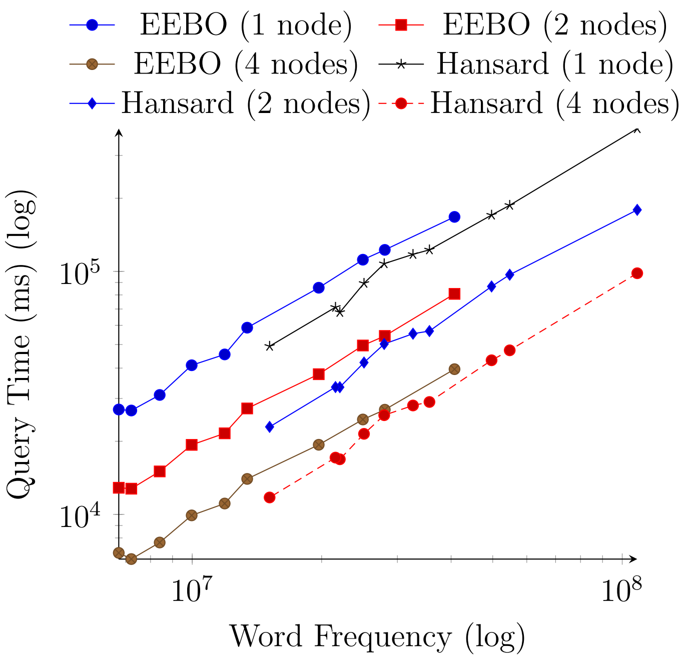


Figure 5.7: Collocations

Figure 5.6 shows the increased performance and reduction in query time for generating concordance lines, unsorted, with a span of five words. A concordance query for the word type "the" in the Hansard corpus which took 77,554ms on a single node was reduced to 22,242ms on a two node configuration and just 7,792ms on a four node cluster (this is the time to find and retrieve all concordance lines). This significant increase in speed is likely down to a reduction in the processing time that will be utilized by the Java garbage collector while executing the query because retrieving the 108 million concordance lines returned from this query carries a significant memory overhead in LexiDB.

Collocation searches were run with a default context window of two words from

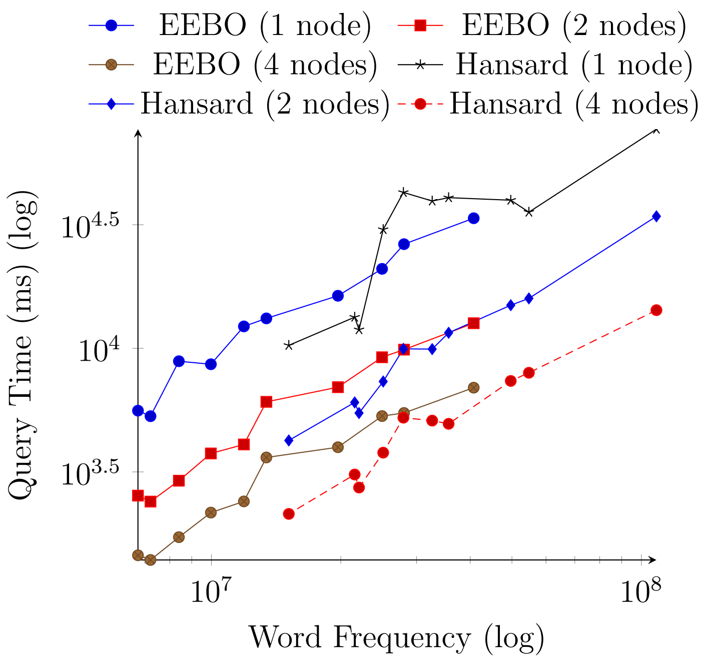


Figure 5.8: N-grams

the search term. Figure 5.7 shows the results of these. Again performance increases substantially as the configuration is scaled out from one to two to four nodes. The collocation queries themselves take far longer than the simple concordance queries due to the additional task of computing a contingency table and collocation metric for the search term.

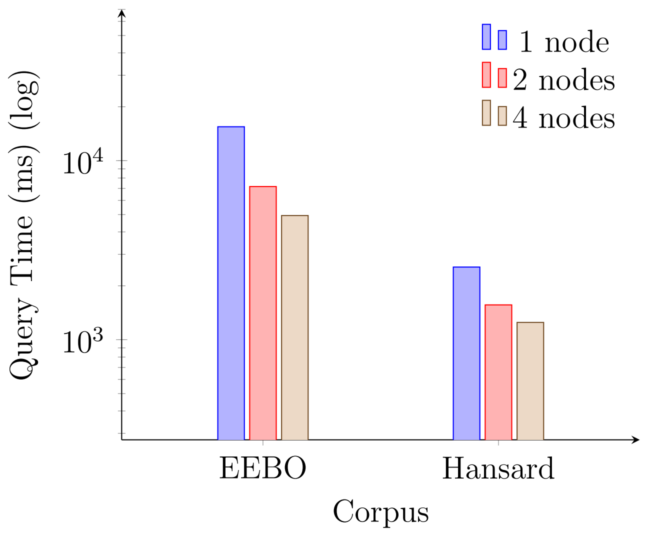


Figure 5.9: Frequency List

Figure 5.8 shows the results for ngram searches. The query used was for a bigram of the specified search term. Similar to the results of the collocation queries, we see far longer query times than with a concordance search, again as a result of the computational expense of constructing n-grams from the retrieved tokens. This method of building n-grams on the fly, while more time consuming than pre-computing them at insertion time, saves significantly on disk space.

The time taken to generate the frequency lists is shown in figure 5.9. This was simply a list query for the regular expression `.*` which will return the frequency of all word types in the database. Obviously for this general case improvements could be made as the frequency lists for all words are pre-computed in LexiDB's dictionary and could be returned without the need to match a regular expression against the entire dictionary.

5.2.3 Experiment 3: Comparative Evaluation

5.2.3.1 Setup

To evaluate the performance of LexiDB we compared the query response time of searching for the most common uni, bi & trigrams on part-of-speech (C5) tags in the British National Corpus. The BNC was chosen as it is a typical corpus linguists may use locally on their own personal devices. The performance was compared to the query response time of CWB - CQP. Further to this, the indexing lookup times were compared to the index lookup times of Lucene (since many other search systems and corpus tools use this as an indexer e.g. Korap, ElasticSearch).

All benchmarks were conducted on the same workbench, specs: i7, 16GB RAM, 256GB SSD (271 MB/s read speeds measured by bonni++[1]). The top 20 unigrams, bigrams and trigrams were queried on each of the supported systems 10 times and the average response times are plotted below. In the case of LexiDB, the query cache was disabled so each result would not be

[1]https://sourceforge.net/projects/bonnie/

retrieved from the cache after each run, rather than performing the query again. The corpus used in all tests was the BNC (British National Corpus) with annotation layers; POS, lemma & C5 tags.

For each of the test systems, the original XML format of the BNC needed to be converted to a format that was supported. LexiDB supports the use of standard TSV files and can support any kind of text data not just typical corpus data annotation layers. CWB requires files to be presented in a single VRT (vertical) file format - this format looks similar to TSV but with no headers and structural data about texts placed all within the same file. Lucene can index various file formats but the simplest way to index is to use text files. This creates a problem for linguistic data marked up with several levels of annotation. To overcome this each of the annotation layers from the original XML was used to generate a text file for that annotation (i.e. one for token, POS, c5 etc.) and then each of these files were indexed as a field within Lucene. This meant although all annotations could be queried on, only one could be accessed in a single query.

5.2.3.2 Results

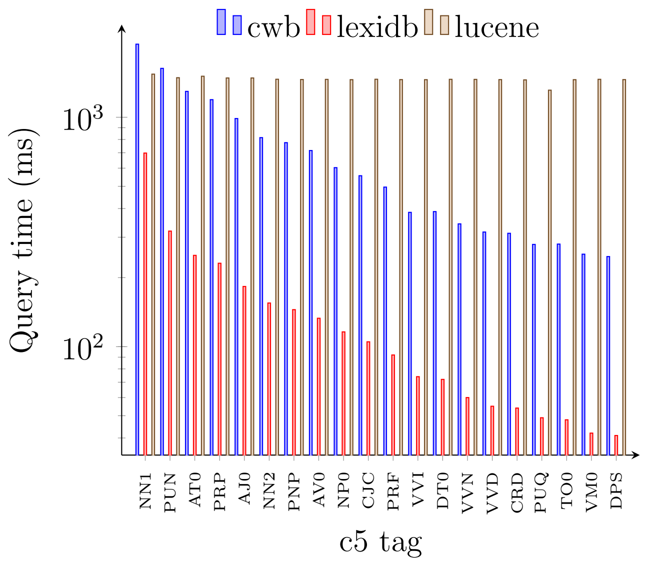


Figure 5.10: Simple querying for Part-of-Speech (c5 tagset)

As can be seen from figure 5.10, the LexiDB and CWB query times for searching for single tags (c5) was proportional to the number of occurrences of the tag within the corpus. This, for all systems, is likely due to the time taken to read the index for that tag. In this case, 'NN1' being the most popular tag searched for and 'DPS' the least. One would expect this trend to continue with lower frequency items such as low-frequency words or more fine-grained tags (obviously most POS tags will be fairly high frequency within any corpus). LexiDBs query response times in this instance were on average 535% faster than CWB/CQP.

Lucene demonstrated very constant query times for single-term queries (as well as bi and trigrams), but this is due to how Lucene returns a set of hit documents as opposed to retrieving sets of concordances lines. Although Lucene could be used for such linguistic type queries it is necessary to build an additional application layer on top of Lucene to achieve this. This would inevitably end up resulting in greater overhead and slower query response times.

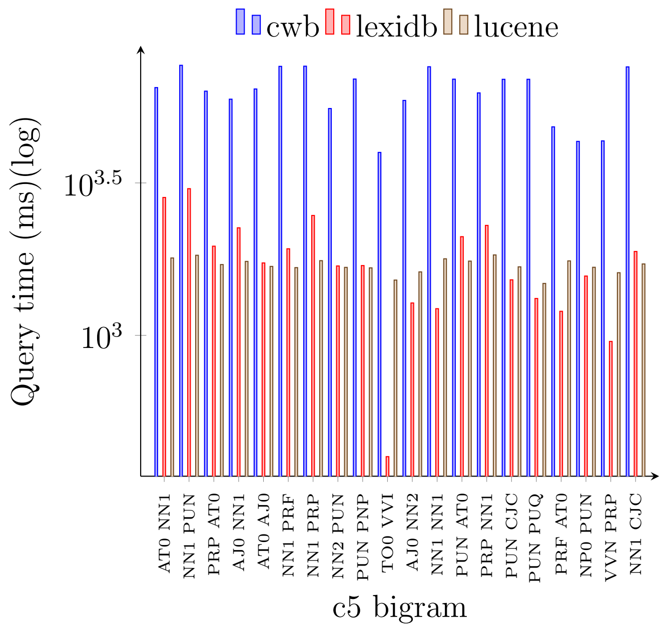


Figure 5.11: Common POS bigram search

Figure 5.11 shows the effect of searching for phrases. Here LexiDB and CWB return concordance lines again, where as Lucene returns a set of documents. Using the most popular POS bigrams within the BNC we can see that the results are no longer affected by the raw frequency of the results but rather by the frequencies of the items involved in the lookup. This is likely due to the link between the query time and the size of the index or postings list that must be loaded for each query. An interesting observation here is that CWB/CQP is slowed more when querying for a phrase where the first item is higher frequency. By comparison, LexiDB is slowed more so when the sum of the frequencies of the items in the phrase is high. We can see that as with unigrams Lucene's response times are fairly constant. This can also be observed when looking at trigrams in figure 5.12.

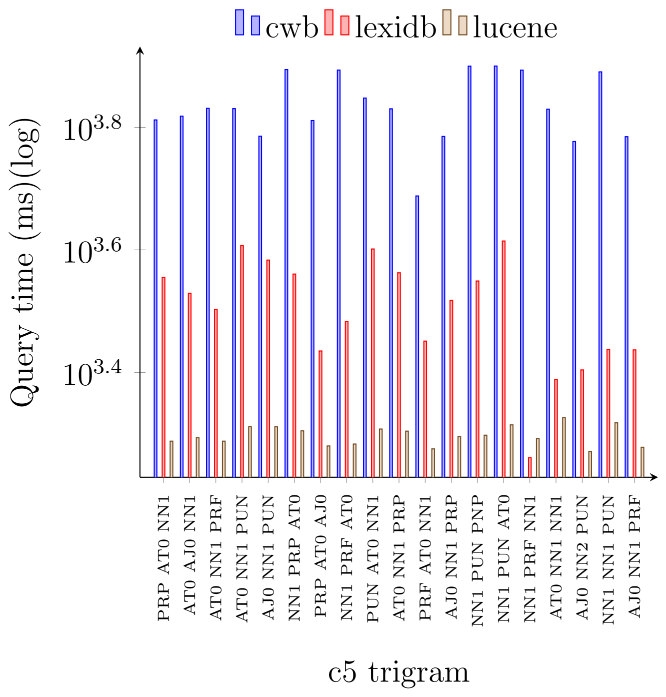


Figure 5.12: Common POS bigram search

Whilst it can be shown from these common POS bigram search results that CWB/CQP is comfortably outperformed by LexiDB which is in turn comfortably outperformed by Lucene, it should be noted that as Lucene is only presenting a set of documents and not a set of hits themselves. CQP and LexiDB are both performing a full query evaluation and retrieval of concordance lines and presenting them to the user. Lucene, by contrast, is only performing a lookup of a phrase query and presenting the set of documents the phrase occurs. Additional steps beyond this would need to be taken to present the results to any end-user in a meaningful way. It should also be noted that in this experimental setup Lucene is only capable of searching a single annotation layer at a time. LexiDB would see similar query response times querying across multiple annotation layers, as would likely CQP.

5.3 Hansard Case Study

To examine the usefulness of LexiDB to corpus linguists, a proof-of-concept case study was devised that included the construction of a complete live corpus based on the historic Hansard parliamentary corpus. This corpus was placed into a LexiDB instance and a toolchain setup (described below) that would update the database daily with the latest data published to the parliamentary website. From this starting point, a web interface was constructed that made use of LexiDB's API and presented users with a graphical interface that allowed usage of LexiDB's query language to perform all the linguistic queries that are supported (see Chapter 4) as well as providing visualisations that could be used by none linguists such as social scientists and historians. Finally, this case study was concluded by the holding of a focus group where several linguists shared their thoughts on the interface and approach, how useful it is, how it compares to other systems and ideas for its future development.

5.3.1 Building the Tool

5.3.1.1 Corpus construction

The Historical Hansard corpus covers transcriptions from 1803 - 2005 in both the House of Lords and the House of Commons. This data is freely available online[2] to anyone who wishes to use it. Previously the historic portion of the Hansard corpus was processed by Lancaster University's linguistic toolchain

[2]https://hansard.parliament.uk/

```
token pos sem
"<meta file=""/commons/2005/01/10/0.tsv"" date=""2005-01-10""
    member=""Peter Luff""/>"
<s>
If   CS   Z7
he   PPHS1 Z8m
will   VM   T1.1.3
make   VVI A1.1.1
a AT1 Z5
statement NN1 Q2.1
on   II   Z5
...
```

Listing 5.3: Lancaster toolchain TSV output

(described below) as part of the SAMUELS[3] project. This historic section of the corpus consists of just under 1.7 billion words (when tokenised through CLAWS) in around 7.5 million files. Data from post-2005 was retrieved via means of TheyWorkForYou[4] parser and scraping tools. The latest parliamentary data, once the initial retrieval to the present day from 2005 was complete, was then downloaded daily.

Newly downloaded data is parsed through the same toolchain as the historic data, which performs tokenisation and POS tagging by way of CLAWS, and semantic tagging utilising USAS. The output of this toolchain (Figure 5.13) is a set of TSV files consisting of a single contribution text per file. These files can then be added to the existing database in LexiDB on the fly, something which is not possible with many other existing corpus data systems. Sample output of the toolchain is shown in figure 5.3.

5.3.1.2 User Interface

With the data loaded into a LexiDB instance and accessible via an API, a web-based user interface was constructed using widely adopted HTML and CSS libraries (Bootstrap[5], JQuery[6], AmCharts[7]). This user interface provides

[3]https://www.gla.ac.uk/schools/critical/research/fundedresearchprojects/samuels/
[4]https://www.theyworkforyou.com/
[5]https://getbootstrap.com/
[6]https://jquery.com/
[7]https://www.amcharts.com/

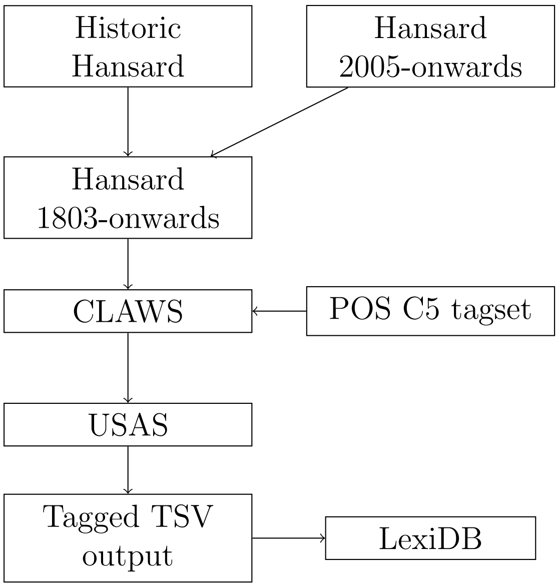


Figure 5.13: Annotation Processing pipeline

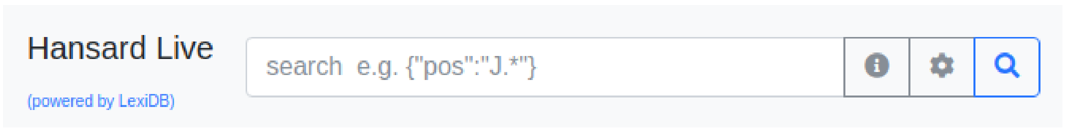
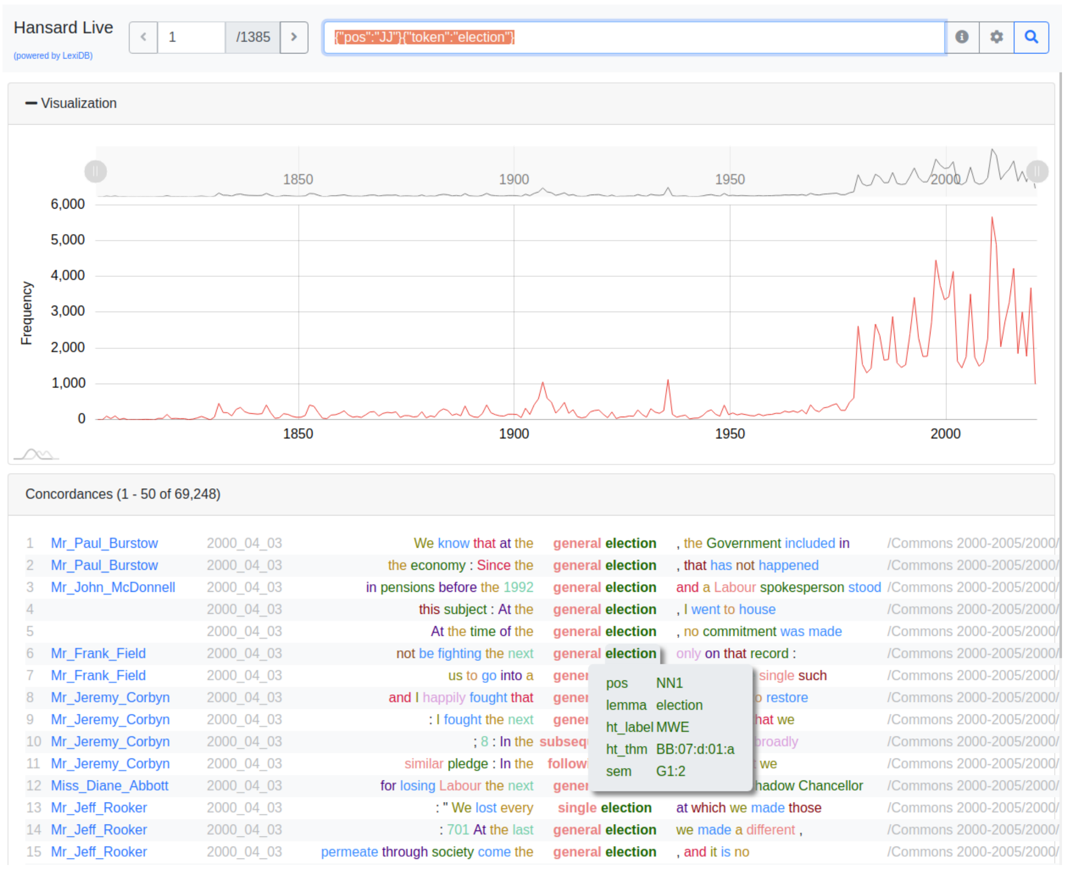


Figure 5.14: Hansard UI search bar

a method for corpus linguists to access all the data in the constructed Hansard corpus using a search bar and provides a graphical way of specifying query parameters. Beyond this, results are generated into HTML output that is displayed on the page along with various tooltips and visualizations that allow for users other than linguists such as social scientists or historians to meaningfully interpret the data.

The main interface page presents the user with a search bar (see Figure 5.14) similar to that found on any modern search engine. Users can fill in this search bar with a word or sequence of words to search for and the results for the default query type (concordance lines) will be retrieved and displayed. For a query the search string entered by the user is converted into a query string used by LexiDB (see section 4.4), defaulting to searching the words or tokens column. Alternatively to this, users once familiarised with the query syntax can enter a LexiDB style query string directly to utilise the ability to perform both character level and token level regular expression matching on the query



Concordances (1 - 50 of 69,248)

1	Mr_Paul_Burstow	2000_04_03	We know that at the	general election	, the Government included in	/Commons 2000-2005/2000/
2	Mr_Paul_Burstow	2000_04_03	the economy : Since the	general election	, that has not happened	/Commons 2000-2005/2000/
3	Mr_John_McDonnell	2000_04_03	in pensions before the 1992	general election	and a Labour spokesperson stood	/Commons 2000-2005/2000/
4		2000_04_03	this subject : At the	general election	, I went to house	/Commons 2000-2005/2000/
5		2000_04_03	At the time of the	general election	, no commitment was made	/Commons 2000-2005/2000/
6	Mr_Frank_Field	2000_04_03	not be fighting the next	general election	only on that record :	/Commons 2000-2005/2000/
7	Mr_Frank_Field	2000_04_03	us to go into a	gener	single such	/Commons 2000-2005/2000/
8	Mr_Jeremy_Corbyn	2000_04_03	and I happily fought that	gener	o restore	/Commons 2000-2005/2000/
9	Mr_Jeremy_Corbyn	2000_04_03	: I fought the next	gener	hat we	/Commons 2000-2005/2000/
10	Mr_Jeremy_Corbyn	2000_04_03	; 8 : In the	subseq	broadly	/Commons 2000-2005/2000/
11	Mr_Jeremy_Corbyn	2000_04_03	similar pledge : In the	follow	we	/Commons 2000-2005/2000/
12	Miss_Diane_Abbott	2000_04_03	for losing Labour the next	gener	hadow Chancellor	/Commons 2000-2005/2000/
13	Mr_Jeff_Rooker	2000_04_03	: " We lost every	single election	at which we made those	/Commons 2000-2005/2000/
14	Mr_Jeff_Rooker	2000_04_03	: 701 At the last	general election	we made a different ,	/Commons 2000-2005/2000/
15	Mr_Jeff_Rooker	2000_04_03	permeate through society come the	general election	, and it is no	/Commons 2000-2005/2000/

pos	NN1
lemma	election
ht_label	MWE
ht_thm	BB:07:d:01:a
sem	G1:2

Figure 5.15: Hansard UI Concordance Results

as well as allowing the ability to query across multiple other columns stored in the database such as POS or semantic tags, or even query based on metadata to further refine the results (see section 3.2.3).

For a simple initial search, the results are displayed to the user in the form of concordance lines. Beyond the typical highlighting of the hit term and vertical alignment of the prior and post text, the interface also provides colour-coded highlighting for each word based on its POS tags (see Figure 5.15 for an example of unsorted results), in this case, all nouns appear in green. This can theoretically be changed to highlight based on any tag available, in this case study the main options available to highlight on are POS tags and semantic tags. Mousing over a particular word will open a tooltip showing all the tags that a particular word contains. Concordance lines also provide metadata alongside the text itself, from the Hansard corpus the metadata displayed is the speaker name, the date of the contribution and the original file name.

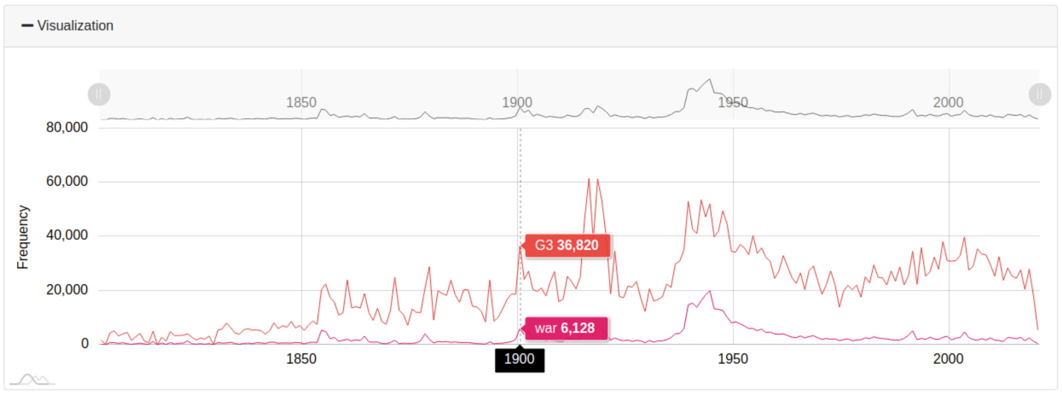
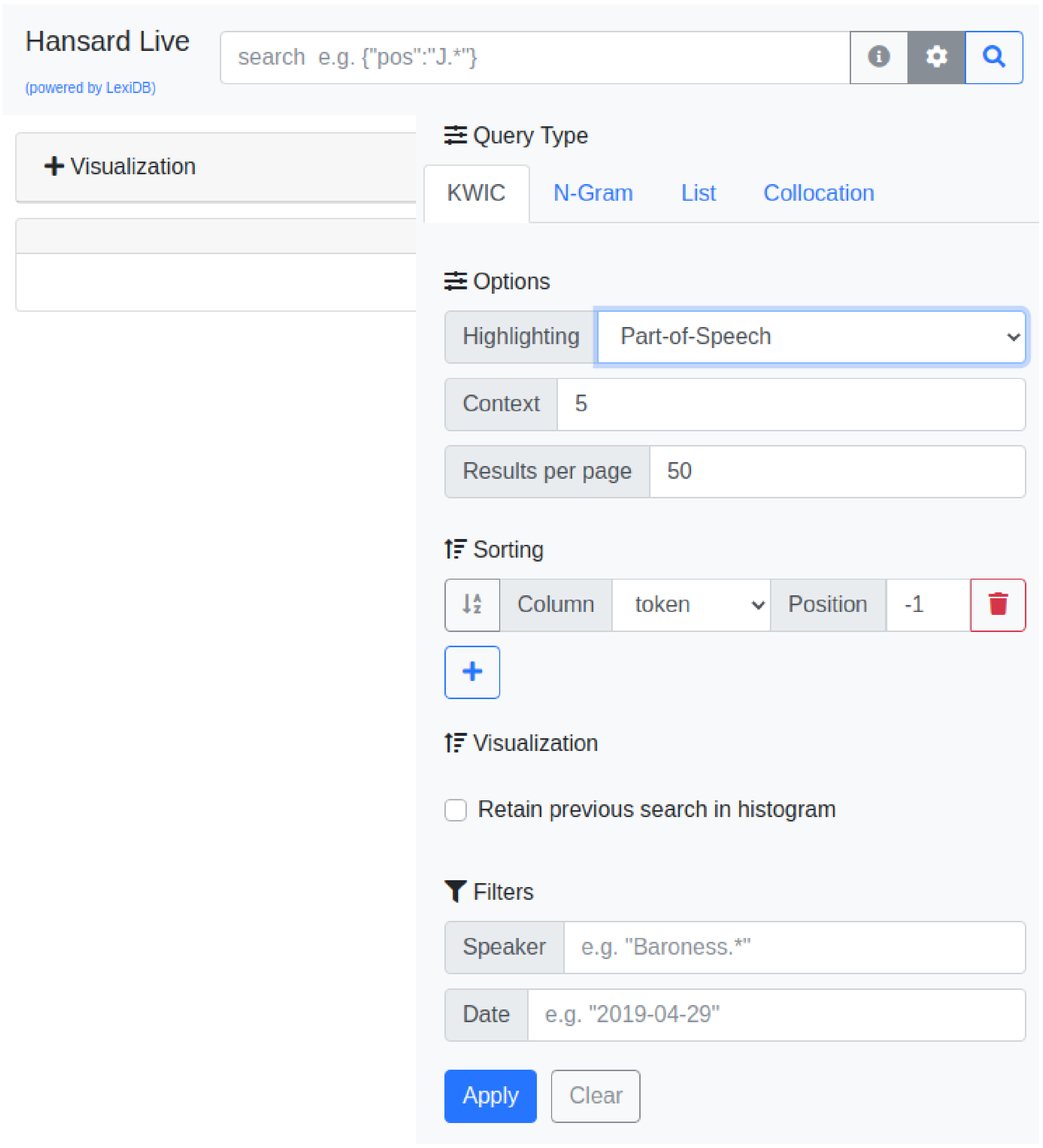


Figure 5.16: Hansard UI Histogram Visualisation

Finally, this view generates a histogram of the occurrences of the query string over time within the corpus, in the case of Hansard from 1803 up until the present day. This visualisation can also retain previous searches so that two searches can be compared over time (see Figure 5.16).

5.3.1.2.1 Query Options Options for the different query types available in the user interface can be accessed from the side panel (see Figure 5.17). When a search has already been performed the different query types can be directly applied and the previous query will be automatically performed again and the new query type displayed to the user. The query types available are a concordance line view (discussed above) referred to as a KWIC type, an N-Gram view, a frequency list and a collocation search. All of these query types can be performed with any type of search string, both individual words or phrases utilising a token level regular expression.

Concordance searches have the options to change the highlighting on words, as previously mentioned but also the ability to specify the context range of the concordance lines as well as the number of concordances displayed per page. As LexiDB allows for asynchronous searching, this mechanism is utilised by the Hansard user interface to immediately display the first available page of results to the user without the need to wait for the entire query to complete. This allows the system to remain highly responsive which is essential for the end user experience. Concordance lines can also be sorted at several levels


Hansard Live
(powered by LexiDB)
search e.g. {"pos":"J.*"}

Visualization

Query Type
KWIC | N-Gram | List | Collocation

Options
Highlighting: Part-of-Speech
Context: 5
Results per page: 50

Sorting
Column: token | Position: -1

Visualization
☐ Retain previous search in histogram

Filters
Speaker: e.g. "Baroness.*"
Date: e.g. "2019-04-29"

Apply | Clear

Figure 5.17: Hansard UI Query Options

based on both the word and tag level at various positions relative to the hit, the sort can also be done both lexically and by the frequency of the term or tag at that position within the results, a feature not typically available on other concordancers. Filters can also be applied to metadata so users can refine results by data or by speaker.

NGram searches can have limitless values for ‘n’ as they are computed based on the context of the search not from a pre-constructed table of NGrams which are typically only constructed up to 5-grams. NGrams can also be constructed based on annotation layers as well as words themselves meaning patterns in part-of-speech for particular searches can easily be examined. As the search can be for a phrase or sequence as well as an individual word, the context around which to build the NGrams can also be specified. Searching for a single word and constructing trigrams around this would need a minimum context of an additional token either side of the search hit for three tokens to be constructed. For a longer phrase, that will consist of at least three tokens, no additional context is required to construct a set of trigrams (although additional context may be desirable).

Frequency lists can be created and groupings can be performed on any annotation level i.e. a search for a particular word can be performed and a frequency list of the POS tags that the word is tagged within the corpus generated.

Collocation search can be performed and the results displayed to the user in the form of a word cloud (see Figure 5.18). The collocation metrics can be calculated based on either log-likelihood or mutual information (see Chapter 2). From this the word cloud visualisation can be generated based on the top x results as specified by the user in the query options. As calculating collocations is sometimes a lengthy operation (particularly when searching for a highly frequent term) the results of the collocation search are updated asynchronously so that estimates of the final results can be displayed to the user in a timely fashion even if the final results take several seconds or even several minutes.

5.3.1.3 Focus Group

The focus group[8] was conducted with two participants, both very experienced corpus linguists who have worked as conference chairs and given numerous plenary talks. Neither had any previous connection to this project. The participants will be hereafter referred to as participant A and B. They were given

[8]This focus group was approved by the Lancaster University FST Research Ethics Committee

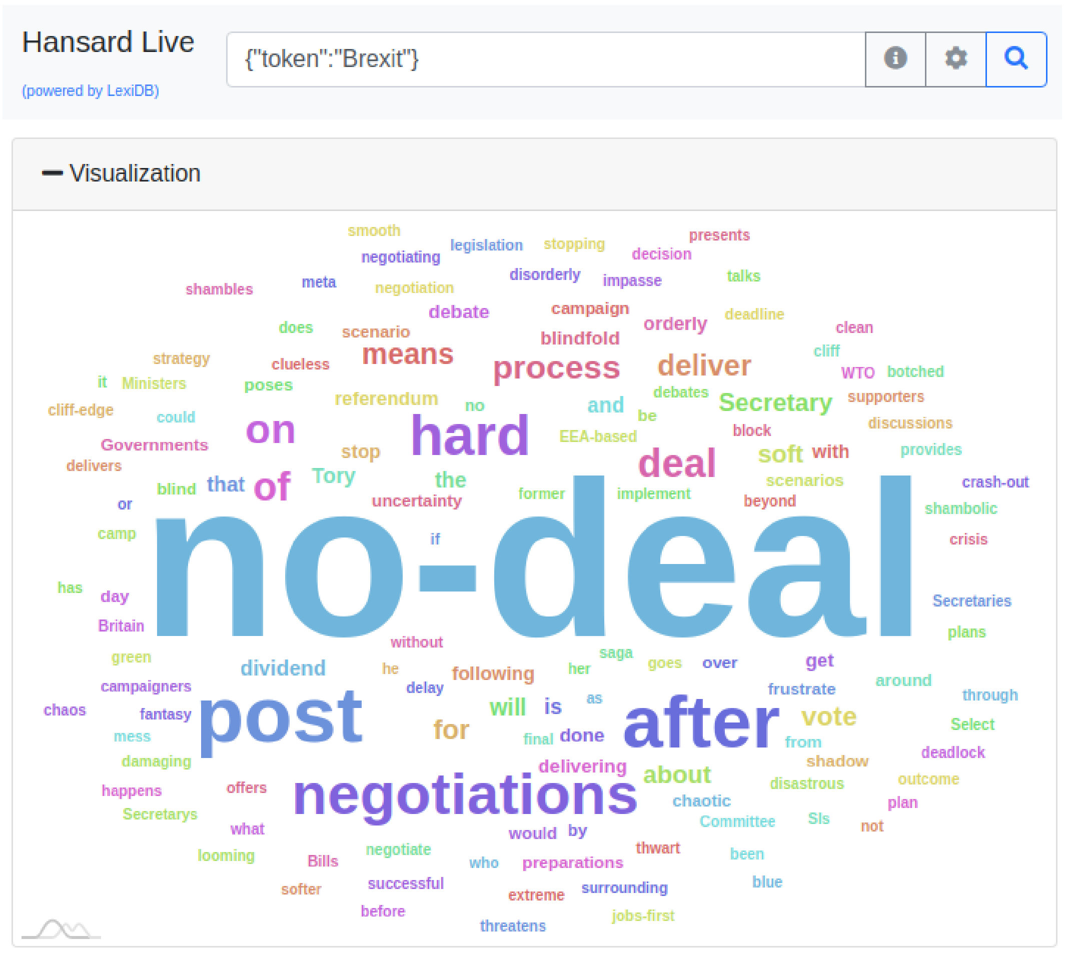


Figure 5.18: Hansard UI Collocation Word Cloud

prior access to the tool before being invited to share their thoughts and opinions on the usability and usefulness of the interface itself, the underlying tool as well as more general views on the current state of corpus linguistics, the tools utilised and how they can work with modern large scale corpora. A set of sample questions asked within the focus group can be found in Appendix D, this is not an exhaustive list as many of the discussion points evolved naturally within the focus group. These questions were asked initially to set up areas of discussion. Both participants were part of the same focus group so ideas could be discussed between each of them and grow naturally into wider discussions.

What follows is a summarization of the interesting or important points raised within the focus group. These points are categorized into three areas, the usage of the Hansard user interface itself, issues of scalability as it relates to modern corpora and the use of other tools and finally thoughts on where the Hansard user interface as well as the underlying database might go next, what strategies might be feasible to increase adoption by corpus linguists. Points raised by the participants are quoted verbatim where possible from the transcription of the focus group and the implications of the point as it relates to the project are briefly discussed to put the issues more into context.

5.3.1.3.1 Using the tool The initial discussion within the focus group centred around the participants first attempts at using the tool and their overall impressions. This discussion was largely positive with the participants having no problems getting started with using the user interface. However certain elements of the functionality of the tool were not immediately apparent and queried by the participants.

"... but I wanted to know how do you search for the company, something like "antisemitism" keeps which is why I was asking you for the best way to look for adjectives for example with antisemitism."

Participant A

This highlighted the lack of clear guidance on how to search for POS tags within the tool which requires knowledge of the query language. Although a query syntax guide is available other simpler ways of guiding users through the usage of the tool are discussed later within the group in section 5.3.1.3.3. Further observations of some more subtle usages were also discussed.

"... search for say an anger semantic field ... I don't want all of the words that co-occur with antisemitism I want anything to do with E3 minus, which is anger."

Participant A

Whilst such a search would be possible within the tool[9], performing a search for the semantic field desired around a specific keyword and subsequently limiting the collocations to within the hit field, such a search requires a deeper understanding of how the tool works and so how to do similar tasks was not immediately apparent to the participants. This is a limitation LexiDB shares with other corpus data systems discussed in Chapter 2.

Another item that was observed was the ability to select different forms of collocation metrics within the UI. As described, LexiDB supports log-likelihood and mutual information metrics for collocations but the participants commented further measures would be desirable.

"... I think it's really nice to have alternative measures available."

Participant B

Features within the UI, specifically options available when querying were then discussed. Although many options exist to customise searching within the tool, it was noted that to change the query type you need to access the options panel displaying all options that are available to the user.

"... some people don't like having too many options because they

[9]`{"sem":"E3-"}.{0,5}{"token":"antisemitism"}|b.{0,5}a=`

don't know which one to choose so I guess it's striking a balance between those."

Participant B

This may necessitate a more simple method of selecting the query type within the UI without the need to be overwhelmed with options.

This difficulty with an awareness of options and methods was observed further when regarding the histogram for searches available in the concordance view.

"Could you overlap them at all? ... search result for one thing and then overlap it with a search result for another thing so that you can see differences over time."

Participant A

While this is an option that is available in the tool it is not readily apparent and could be better utilised by being immediately available. This is also discussed later in section 5.3.1.3.3. When highlighted, this ability was greatly appreciated by the participants.

"I think that's great, if you can do something like that and it visually represents in something like this which is to do with change over time, those possibilities are really useful."

Participant A

The usefulness of this histogram was discussed at length with a great deal of enthusiasm surrounding the potential ability to enhance the functionality of it further to highlight specific time periods and filter your results and then save this for later.

"... if you could then save that so you're continually able to save something and then do something with that specific part of the dataset."

Participant A

5.3.1.3.2 Scalability of corpora and other tools After gathering some feedback on the participants first thoughts on using the tool the discussion then turned to how the participants use other tools with a specific focus on if they found other tools easy to use for larger-scale corpora.

The consensus within the group seemed to be that tools such as CQPWeb were widely used for larger corpora if the desired corpora was hosted and available.

"... it depends what corpus I'm working on, so if it's just a case of where people have put various things, but yeah it's all CQPWeb at the moment."

Participant B

It was also noted that various hosted services are becoming more restrictive, possibly if they are turning to more commercially driven models to provide their services.

"...I've used SketchEngine a lot but liking it less at the moment because they restricted things that you can do with it quite a lot, but otherwise AntConc and Wordsmith."

Participant B

The participants' experience with using software and tools locally with large scale corpora was probed with observations that many systems work well with smaller scale corpora but are hard to use as the dataset grows.

"...I've definitely felt restrictions of not being able to get software to work on very large corpora."

Participant B

As well as the deficiencies noted in using larger corpora, the difficulties in simply setting up software for personal usage was debated with examples found of commonly used hosted tools being frustrating to set up to use yourself.

"... I do know people who've tried to install CQPWeb and just

given up with it, that it required a lot more setting up than they expected."

Participant B

Such difficulties may have hindered many corpus linguists in research projects in recent years, having software available that may very well perform and do the task you want but not being able to make proper use of it due to difficulties in setup or inadequacies for handling large datasets.

Digging further into this discussion surrounding the scale of corpora that were typically used by participants, the question turned to whether corpora seemingly, in the participants' views are growing more and more or if smaller more specialised corpora were still a more pressing concern.

"It depends if it's historic ... Hansard is quite big for a historical corpus... it depends on period for some as to what's an acceptable size of corpus. "

Participant A

"...it varies so much across different projects that its nice to have software that has the flexibility that can use the bigger datasets when you need to."

Participant B

This brought things back around to reliance on software that is hosted elsewhere, limiting linguists to corpora that are already available and not being able to use their own.

"...I've been using historical corpora which have been quite big but they've all been hosted elsewhere so I haven't had to deal with the problems of getting them to work on software. "

Participant B

Looking more towards the process of corpus collection and particularly the growing need to have live as opposed to static corpora specifically for corpora built from mediums such as social media and live news feeds, observations were made that existing tools make this handling and management of live data cumbersome.

"...we also looked at social media so some of it was tweets some of it was manifestos and so you're basically making a corpus up of various types of dataset and then that becomes unwieldy if you've then got to manage it all."

Participant A

This led to a wider debate on the ability of LexiDB to handle live data so that linguists could build up their corpora in stages without the need to keep the entire dataset a static entity and then load the entire thing into a tool again when it is updated.

"I think that would be excellent... but yeah if you can add to it and you can look at collecting data over a period of time, that I think would help a lot of different people do different projects. "

Participant A

Various examples of projects that participants were involved in, again using social media data, were highlighted as good examples of where such a facility would have been helpful.

"... being able to continually update that over a period of time instead of having to choose when to start and when to stop... it would have been easier if we'd had the facility to be able to keep adding."

Participant A

Limitations of other corpus tools as they relate to the scale of data that they can handle were discussed by the participants. Particularly visualization tools and the problematic nature of handling big corpora, despite how useful many other tools are these issues are a source of frustration for the participants regularly.

"I guess the only thing to say on the scale is problems I've had on other interfaces, especially GraphColl, is visualisations, that they are brilliant and then you put them to work on large corpora and it's just extremely difficult to get anything out of it."

Participant B

When questioned further the types of visualizations that tend to struggle in these tools are network graphs that can be expanded and collapsed. These types of networks would be possible to generate using LexiDB as a back end, making use of the ngram querying to gather data on the fly as the network graph visualisations are expanded. It is likely existing tools need all the information gathered and loaded into memory initially in order to load the visualisation - meaning larger corpora are not practical to this style of visualisation. Beyond these, many of these visualisations themselves will need adapting for larger corpora, e.g. network expansion of occurrences may work well as a visualisation with only a few dozen paths, but in extreme-scale corpora, there could be hundreds or thousands of paths.

Beyond this, further functional limitations of other systems were debated. Other tools such as Cone[43] allow for useful visualisations to be used but only raw text corpora can be worked with as opposed to tagged and annotated corpora that are available in the interface being considered here.

"With Cone, you can also delete some, so if you wanted to show a particular pattern you could delete the ones that weren't part of the pattern. Because yours can work on semantic fields as well there would be less need to do that, because this was working on raw corpora that weren't pos tagged or semantic tagged. So if you want to show for instance adjectives around a particular term."

Participant B

It appears that problems consistently exist when using larger modern corpora for corpus linguists. Problems seem to arise far more frequently when using bigger corpora regardless of whether they are annotated or not. This suggests that the ability of LexiDB to more readily handle this scale of corpora and provide an API upon which it is possible to build simple web-based corpus tools like the Hansard UI would be of great benefit to both corpus linguists and the developers of corpus tools.

5.3.1.3.3 Thoughts for adoption and future development The final topic considered within the focus group was potential for future developments of the interface that had been examined by the participants. As a proof-of-concept, these suggestions relate primarily to improvements to the user interface design and how such a tool could be used in the future.

The primary suggestion that opened into a wider discussion is the usage of the query language of LexiDB and the options available within the interface. As was discussed earlier in the focus group a query syntax guide exists but this may not be the best way to encourage adoption or get people started with using the tool.

"...I think people would want to be able to have a file with lots of examples probably to get them started."

Participant B

"Yes same, same definitely."

Participant A

Clearly, a strong idea for easing people into the usage of the interface and the underlying database is a more example-based suggestion on how to use the interface, perhaps through pop-ups and walk-throughs available directly in the UI. This could direct users to different aspects of the interface they might not otherwise be aware of and ensure that the query language is better understood and utilised.

"Yeah I think it's fine I just think people like to have lots of examples and then they can work out how to do their own if they've got questions."

Participant B

Nested within this discussion was an observation on the type of users who might utilise the system, whether it be experienced corpus linguists who are familiar with similar tools and other corpus query languages or students who might not be as familiar with such systems. Providing clear worked examples or a related set of examples that could be seen and followed through within the interface it was suggested might provide a strong platform for people getting to grips with corpus methods and techniques.

"Yeah, I found with SketchEngine when I was using it with students what I had to do was do quite a lot of "if I wanted to do x, how would I do it?" type things with them, and then once they worked their way through those they just sort of go off and be creative with it but people sort of need that security at the beginning to work out what they need to do."

Participant B

"...I think it depends on who you're aiming it at. If you're aiming it at people who use CL tools already, (you) probably don't have to provide as many examples... but if it's for students and thinking about making something that they can then use to put their own datasets in, (then) I can imagine a lot of students who would really want to do that, but again they would need a bit of support to get them going."

Participant A

Participants indicated a desire to be able to save many of the aspects of their work using the tool. This included the wish to save both the visualisations produced from their searches so that they could potentially use them in their research but also the ability to save complex searches as they do them to provide a means of picking up where they left off when they return to the tool later. It was communicated that many other tools used by the participants could not save certain aspects of their work, particularly when performing exploratory steps digging into the data within corpora that meant doing more complex things became more difficult than necessary.

"And the other thing was you couldn't save them at all..."

Participant B

"Save so you can search again within that particular space you've identified."

Participant A

'... if you can save complex searches as well. It's quite handy to be able to just pick something up again."

Participant B

This could be achieved in many ways, both within the interface and within the architecture of LexiDB itself, perhaps by providing a search history that is saved within the database. In many other search systems, query histories can also be leveraged into creating search suggestions.

"I think that the more you can bring it to life for people by maybe building into it a mini case study and saying how this is this search this is this visualisation this is how we build, I think you'll get a lot of people interested in it."

Participant A

"It could just be a series of searches around a theme I suppose that'd help show people. "

Participant B

This final thought of building a wider case study (on top of the previously suggested list of worked examples) is very much an area that would get potential users interested in using the Hansard corpus and user interface here for specific purposes by providing a clear path to using the tool to conduct research, and demonstrating the types of visualisations and the types of insights into the data that the tool can help uncover. This is less to do with the underlying database; however, by demonstrating the types of capability that the database can support, the participants believe that this could help to fuel interest and adoption of not only the Hansard user interface but also of LexiDB itself.

5.4 Summary

In this chapter, we have discussed an evaluation methodology that attempts to resolve the lack of any standardised approach for evaluating corpus data systems or corpus databases within the literature. This methodology hopes to answer research questions 2 and 3 in chapter 1. The methodology can also serve as the foundations upon which a standardised approach can be built. The method presented is two-pronged including both a quantitative comparison of the proposed system LexiDB to other corpus data systems, modern database management systems and indexing systems as well as a qualitative analysis of how LexiDB can be used as a data layer to develop a simple corpus tool based on a large scale diachronic corpus, Hansard.

The quantitative analysis demonstrated that LexiDB's design can outperform existing corpus data tools. This is demonstrated through its ability to consistently outperform CQP and Lucene for corpus queries and patterns with typically faster query response times than both systems (see section 5.2.3). The ability of LexiDB to scale out to handle an ever greater size of corpora is also demonstrated in section 5.2.2 and how the specialised architecture of the system is comparably scalable to modern database systems (MongoDB and Cassandra) when scaling out in a clustered configuration.

Section 5.3.1 illustrated how easily the API of the system can be used to rapidly develop a web-based corpus tool in a single stand-alone HTML page. The effectiveness of this prototype is qualitatively examined in the focus group presented in section 5.3.1.3. The corpus linguists who participated, were excited by the possibilities of such a flexible tool that can also scale to accommodate large modern corpora.

Chapter 6 will discuss some of the shortcomings of the evaluation methodology and suggest potential future enhancements that can take this approach further.

Chapter 6

Conclusions

6.1 Summary of Thesis

Chapter 2 surveyed the plethora of related techniques and approaches. This established the foundational background needed to understand and reconcile established methods into a coherent design for performing corpus analysis. This began with a discussion of the types of querying that are necessary to perform generic corpus analysis and moves into how existing corpus tools allow linguists to store and interrogate corpora. From the examination of existing tools it was concluded that a wider range of approaches must be taken to modernise the methods that corpus software use and this motivated further consideration of patterns utilised by modern database management and information retrieval systems which are tailored to handling big data. This examination of processes for indexing and distributing databases acted as the basis for the novel design and approach of this project.

The fundamental architecture that the design for LexiDB is built upon is outlined in chapter 3. Here the proposal for a distributed architecture for a corpus database is put forth. This allows for the design to be both scalable and differentiates it from existing corpus tools that do not have this capability and are only able to handle greater scales of corpora by scaling up i.e. running on bigger, faster more powerful machines. By contrast, LexiDB's design enables for distribution across multiple machines or nodes which would allow the design to be deployable on various cloud infrastructures (an example of which would be AWS which it was demonstrated on in evaluation). Whilst the architectural design shifts away somewhat from traditional ACID style SQL databases it is demonstrated that the modern style of No-SQL type databases which act as document stores is far more in keeping with the paradigm of data management used in corpus analysis and can also be observed in the methods and manner of usage of other corpus tools discussed in chapter 2. Methods by which existing indexing techniques for both looking up and compressing postings lists can be applied in this context are described. Finally, ways such a design can be distributed through the adaption of an existing distributed hash table algorithm are put forth. Disparate approaches are synergised into a single coherent design and the implications for how it can be used to resolve corpus queries are considered.

In chapter 4, how the architecture proposed in chapter 3 can be used to perform searching and querying in the context of corpus linguistics is discussed. This goes beyond the simple lookups of information retrieval and indexing systems to the ability to resolve regular expressions at not only the character level within words but at the token level within documents. To do this, two algorithms are proposed; the first an adaption of an existing approach[7] to resolve finite-state automata over tries, the second an entirely novel algorithm to resolve finite state machines based purely on postings lists with no need to access the underlying data. These algorithms are then applied to the architecture of LexiDB and further methods are proposed to utilise these techniques to resolve the commonly used corpus queries.

The evaluation performed and discussed in chapter 5 examines the scalability of the architecture proposed in chapter 3. It contrasts this with the scalaibility of existing distributed database management systems. It also explores how the approaches in chapter 4 allow LexiDB to outperform an existing corpus data system and a commonly used indexing library that is utilised by other corpus data systems. Beyond this, the chapter explored qualitatively the usefulness of the proposed database system to experienced corpus linguists through the use of a custom-built web interface designed as part of a case study where linguists could explore a corpus consisting of around two billion words (Hansard).

6.2 Proposed Design

Chapters 3 and 4 describe the proposed design implemented in LexiDB to allow for corpus analysis to be performed on extreme-scale corpora in the order of billions or even tens of billions of words. The design allows for the system to be scaled out to handle the ever-greater size of corpora by scaling out not up, this as identified in chapter 2, is a key limitation of existing, off the shelf, corpus data systems that only allow for scaling up to support bigger sizes of corpora. LexiDB achieves this by making use of an adapted use of the Chord algorithm which is demonstrated to be an effective approach for distributing data sets and maintaining redundancy.

The design as implemented has been demonstrated to scale similarly to existing NoSQL database management systems MongoDB and Cassandra (see sections 5.2.1 and 5.2.2). This means that the approach can be considered an appropriate alternative to less specialised modern database systems due to the tailoring of the API towards corpus data. The design acts as a column store internally whilst exhibiting the outward appearance in the API of a document store (see chapter 3). This makes it far better suited to storing corpus data than NoSQL systems that both inwardly and outwardly have either one of these properties i.e. document stores have well-suited APIs to corpus data but do not have the internal storage advantages of column stores and column stores vice versa.

In what way this approach compares to existing corpus tools can be seen in section 5.2.3 where it is found that the prototype implementation of LexiDB outperforms CWB consistently across a multitude of uni, bi and trigram part-of-speech pattern searches where it is up 535% faster on basic searches and consistently faster across all types. It is also demonstrated that the querying approach can outperform Lucene on many occasions, even despite the added overhead of actual retrieval of data in context after index lookups which Lucene does not perform in the associated tests. This demonstrates that the approach taken in the design has advantages over many other hosted corpus data systems discussed in chapter 2, many of which utilise Lucene for indexing.

6.3 Limitations and Future Work

While the design of LexiDB allows for efficient storage and retrieval of multi-layered annotated corpora in the form of token streams, it does not fully support the ability to handle corpora at multiple layers. Although the column store approach can be customised to handle structures such as XML (XPath can be used to extract metadata), the primary file format that must be used when inserting data is TSV or CSV. While sufficient for the aims of this project this does create a problem of conversion being required for many corpora whose annotations are not stored as columns in a TSV or CSV file but as attributes or elements in an XML document. Alternative approaches and methods to handle XML based corpus formats such as LPath are discussed in chapter 2, as well as methods in which column stores can be modified to store and efficiently execute XPath queries, are illustrated by the design of XBase. Future work to combine these approaches and incorporate them into the architecture of LexiDB would provide a good foundation to build on going forward. This would make the design capable of supporting any potential format of source data without any need to convert to some tabular type format.

Beyond this, how the proposed approach can be compared meaningfully to existing systems is somewhat limited. This is in no small part due to the unfortunate lack of availability of many existing corpus data systems. As discussed in section 2.2 many of these systems are only available as hosted online resources and users are tied into using both the corpora that are readily made available on these systems as well as the hardware upon which they are hosted. This makes meaningful comparisons between systems extremely difficult to attain, even in situations where the same, or comparably similar corpora are available on two hosted systems, information on the hosted setup, specifically the hardware being used, is consistently lacking. Greater dissemination of source code or at least deployable versions of existing systems so that linguists can utilise the systems in their own environment could help to remedy this problem but this is a community-wide concern and does not look likely to change soon due to the high proportion of systems built for a specific research project and/or with specific corpora in mind, the motivation to allow the systems to be released for more general purposes may simply not be there or in many cases to do so may simply not be practical.

Finally, the focus within the evaluation was on English language corpora. This could conceivably be considered a problem in other languages, perhaps where the text reads right to left, unlike English. However, in practice, this is less likely to be a problem as this would need to be handled by a toolchain responsible for tokenisation, once in the form of a token stream any language could be handled by LexiDB. In terms of displaying query results in a user interface, this would again be an application-level concern. The user interface described chapter 5 was designed specifically for the Hansard corpus. Interfaces for corpora in other languages may need a custom application level to display results appropriately.

6.4 Research Questions and Contributions

The main research question of this thesis was to investigate how modern database and information retrieval techniques could be tailored to meet the requirements of the ever-increasing scale of corpora utilised for corpus analysis.

Through the development of a new architecture combining several existing approaches and the formulation of unique algorithms to handle corpus querying, it has been demonstrated that this new design can be more effective than existing systems for fast querying of extreme-scale corpora. The sub-questions outlined in section 1.4 have been answered and the following novel contributions are claimed;

1. A novel database architecture tailored specifically for corpus data that has the ability to scale out to handle ever greater sizes of corpora.
2. An adapted method for applying regular expressions over a dictionary of strings based on resolving finite state machines against pre-constructed radix trees.
3. A novel method of performing token regular expression lookups utilising only index information without any need to access the underlying token stream.
4. A proposal for an evaluation methodology that allows for corpus query systems to be meaningfully compared quantitatively based on their response times to the most heavyweight or typically time-consuming queries.
5. Demonstration that the proposed design can be meaningfully used by corpus linguists via the Hansard case study.

References

[1] V. Abramova and J. Bernardino. NoSQL Databases: MongoDB vs Cassandra. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, pages 14–22, 2013.

[2] S. Adolphs, B. Brown, R. Carter, P. Crawford, and O. Sahota. Applying Corpus Linguistics in a Health Care Context. *Journal of Applied Linguistics and Professional Practice*, 1(1):9–28, 2007.

[3] M. Alexander and M. Davies. The Hansard Corpus 1803-2005. 2015.

[4] M. Alexander, M. Davies, and F. Dallachy. Semantic EEBO. 2017.

[5] L. Anthony. AntConc (Version 3.4. 3)[Computer Software]. Tokyo, Japan: Waseda University, 2014.

[6] R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern Information Retrieval*, volume 463. ACM press New York, 1999.

[7] R. A. Baeza-Yates and G. H. Gonnet. Fast Text Searching for Regular Expressions or Automaton Searching on Tries. *Journal of the ACM (JACM)*, 43(6):915–936, 1996.

[8] P. Baker. Corpus Methods in Linguistics. *Research Methods in Linguistics*, pages 93–113, 2010.

[9] P. Baker, C. Gabrielatos, and T. McEnery. Sketching Muslims: A Corpus Driven Analysis of Representations Around the Word 'Muslim'in the British Press 1998–2009. *Applied linguistics*, 34(3):255–278, 2013.

[10] P. Bański, J. Bingel, N. Diewald, E. Frick, M. Hanl, M. Kupietz, P. Pzik, C. Schnober, and A. Witt. KorAP: The New Corpus Analysis Platform at IDS Mannheim. *Proceedings of the 6th Language and Technology Conference*, 2013.

[11] P. Bański, E. Frick, and A. Witt. Corpus Query Lingua Franca (CQLF). In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 2804–2809, 2016.

[12] M. W. Bauer and B. Aarts. Corpus Construction: A Principle for Qualitative Data Collection. *Qualitative Researching With Text, Image and Sound: A Practical Handbook*, pages 19–37, 2000.

[13] H. R. Bazoobandi, S. de Rooij, J. Urbani, A. ten Teije, F. van Harmelen, and H. Bal. A Compact In-Memory Dictionary for RDF Data. In *European Semantic Web Conference*, pages 205–220. Springer, 2015.

[14] G. Berry and R. Sethi. From Regular Expressions to Deterministic Automata. *Theoretical computer science*, 48:117–126, 1986.

[15] S. Bird, Y. Chen, S. B. Davidson, H. Lee, and Y. Zheng. LPath, a Symmetric XPath Dialect for Linguistic Queries.

[16] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML Query Language. 2002.

[17] E. A. Brewer. Towards Robust Distributed Systems. In *PODC*, volume 7. Portland, OR, 2000.

[18] O. Christ. A Modular and Flexible Architecture for an Integrated Corpus Query System. *arXiv preprint cmp-lg/9408005*, 1994.

[19] K. W. Church and P. Hanks. Word Association Norms, Mutual Information, and Lexicography. *Computational linguistics*, 16(1):22–29, 1990.

[20] J. Daciuk, S. Mihov, B. W. Watson, and R. E. Watson. Incremental Construction of Minimal Acyclic Finite-State Automata. *Computational Linguistics*, 26(1):3–16, 2000.

[21] M. Davies. The Corpus of Contemporary American English (COCA): 560 Million Words, 1990-Present, 2008.

[22] M. Davies. Expanding Horizons in Historical Linguistics With the 400-Million Word Corpus of Historical American English. *Corpora*, 7(2):121–157, 2012.

[23] M. Davies. Corpus of News on the Web (NOW): 3+ billion words from 20 countries, updated every day. *Retrieved January*, 25:2019, 2013.

[24] M. Davies. The Best of Both Worlds: Multi-Billion Word "Dynamic" Corpora. *Challenges in the Management of Large Corpora (CMLC-7) 2019*, page 23, 2019.

[25] M. Davies and J.-B. Kim. The Advantages and Challenges of "Big Data": Insights From the 14 Billion Word iWeb Corpus. *Linguistic Research*, 36:1–34, 2019.

[26] R. De La Briandais. File Searching Using Variable Length Keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 295–298, 1959.

[27] H. De Smet. Corpus of Late Modern English texts (extended version). 2008.

[28] H. De Smet. Yahoo-based Contrastive Corpus of Questions and Answers. 2009.

[29] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[30] L. Denoyer and P. Gallinari. The wikipedia xml corpus. In *International Workshop of the Initiative for the Evaluation of XML Retrieval*, pages 12–19. Springer, 2006.

[31] T. Dunning. Accurate Methods for the Statistics of Surprise and Coincidence. *Computational linguistics*, 19(1):61–74, 1993.

[32] P. Elias. Efficient Storage and Retrieval by Content and Address of Static Files. *Journal of the ACM (JACM)*, 21(2):246–260, 1974.

[33] S. Evert. The CQP Query Language Tutorial. *IMS Stuttgart. CWB version*, 2:b90, 2005.

[34] S. Evert and A. Hardie. Twenty-first Century Corpus Workbench: Updating a Query Architecture for the New Millennium. *Proceedings of Corpus Linguistics*, 2011.

[35] S. Evert and A. Hardie. Ziggurat: A New Data Model and Indexing Format for Large Annotated Text Corpora. *Challenges in the Management of Large Corpora (CMLC-3)*, page 21, 2015.

[36] R. M. Fano. *On the Number of Bits Required to Implement an Associative Memory.* Massachusetts Institute of Technology, Project MAC, 1971.

[37] P. Forchini and A. Murphy. N-Grams in Comparable Specialized Corpora: Perspectives on Phraseology, Translation, and Pedagogy. *International journal of corpus linguistics*, 13(3):351–367, 2008.

[38] W. N. Francis and H. Kucera. Brown Corpus Manual. *Letters to the Editor*, 5(2):7, 1979.

[39] R. Garside. The CLAWS Word-Tagging System. *The Computational Analysis of English: A Corpus-Based Approach. London: Longman*, pages 30–41, 1987.

[40] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In *Proceedings 14th International Conference on Data Engineering*, pages 370–379. IEEE, 1998.

[41] S. T. Gries. Dispersions and Adjusted Frequencies in Corpora. *International Journal of Corpus Linguistics*, 13(4):403–437, 2008.

[42] C. Grün, S. Gath, A. Holupirek, and M. H. Scholl. XQuery Full Text Implementation in BaseX. In *International XML Database Symposium*, pages 114–128. Springer, 2009.

[43] D. Gullick, P. Rayson, J. Mariani, S. Piao, and F. Taiani. CONE: COllocational Network Explorer [Computer Software], 2010.

[44] T. Haerder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM computing surveys (CSUR)*, 15(4):287–317, 1983.

[45] A. Hardie. CQPweb—combining Power, Flexibility and Usability in a Corpus Analysis Tool. *International journal of corpus linguistics*, 17(3):380–409, 2012.

[46] D. Hawking. Web Search Engines: Part 2. *Computer*, 39(8):88–90, 2006.

[47] S. Hellmann, C. Chiarcos, and A.-c. N. Ngomo. The Tiger Corpus Navigator. In *In Proc. 9th International Workshop on Treebanks and Linguistic Theories (TLT-9*. Citeseer, 2010.

[48] K. Hofland and S. Johansson. *Word Frequencies in British and American English*. Norwegian Computing Centre for the Humanities, 1982.

[49] S. Z. W. Huilin. Overview on the Advance of the Research on Named Entity Recognition. *Data Analysis and Knowledge Discovery*, 26(6):42–47, 2010.

[50] D. Janus and A. Przepiórkowski. Poliqarp 1.0: Some Technical Aspects of a Linguistic Search Engine for Large Corpora. In *The proceedings of Practical Applications of Linguistic Corpora*, 2005.

[51] D. Janus and A. Przepiórkowski. Poliqarp: An Open Source Corpus Indexer and Search Engine with Syntactic Extensions. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics*

Companion Volume Proceedings of the Demo and Poster Sessions, pages 85–88, 2007.

[52] B. Jurish and K.-M. Würzner. Word and Sentence Tokenization with Hidden Markov Models. *JLCL*, 28(2):61–83, 2013.

[53] D. E. Knuth. *The Art of Computer Programming*, volume 3. Pearson Education, 1997.

[54] O. Kolesnikova. Survey of Word Co-Occurrence Measures for Collocation Detection. *Computación y Sistemas*, 20(3):327–344, 2016.

[55] G. Leech. 100 Million Words of English: The British National Corpus. *Language Research*, 28(1):1–13, 1992.

[56] G. Leech, R. Garside, and E. S. Atwell. The Automatic Grammatical Tagging of the LOB Corpus. *ICAME Journal: International Computer Archive of Modern and Medieval English Journal*, 7:13–33, 1983.

[57] F. M. Liang. Word Hy-phen-a-tion by Com-put-er. Technical report, Calif. Univ. Stanford. Comput. Sci. Dept., 1983.

[58] C. D. Manning, H. Prabhakar Raghavan, R. Baeza-Yates, and B. Ribeiro-Neto. Information Retrieval Systems, 2009.

[59] T. McEnery. *Corpus linguistics*, volume 978019. Oxford University Press Inc, 2012.

[60] T. McEnery and A. Wilson. Corpus Linguistics. *The Oxford Handbook of Computational Linguistics*, pages 448–463, 2003.

[61] R. McNaughton and H. Yamada. Regular Expressions and State Graphs for Automata. *IRE transactions on Electronic Computers*, (1):39–47, 1960.

[62] P. Meurer. Corpuscle–A New Search Engine for Large Annotated Corpora. Technical report, Technical report, Uni Digital, Uni Research AS, Bergen, Norway, 2010.

[63] P. Meurer. Corpuscle–A New Corpus Management Platform for Annotated Corpora. *G. Andersen (ed)*, pages 31–50, 2012.

[64] J. Michelfeit, J. Pomikálek, and V. Suchomel. Text Tokenisation Using unitok. In *RASLAN*, pages 71–75, 2014.

[65] A. Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2017. `http://www.brics.dk/automaton/`.

[66] Y. Murata, T. Inaba, H. Takizawa, and H. Kobayashi. A Distributed and Cooperative Load Balancing Mechanism for Large-Scale P2P Systems. In *International Symposium on Applications and the Internet Workshops (SAINTW'06)*, pages 4–pp. IEEE, 2006.

[67] A. Nayak, A. Poriya, and D. Poojary. Type of NOSQL Databases and Its Comparison With Relational Databases. *International Journal of Applied Information Systems*, 5(4):16–19, 2013.

[68] J. Nivre and M. Scholz. Deterministic Dependency Parsing of English Text. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 64–70, 2004.

[69] A. Nurmi. The Corpus of Early English Correspondence Sampler (CEECS). *ICAME journal*, 23:53–64, 1999.

[70] D. D. Palmer. Tokenisation and Sentence Segmentation. *Handbook of natural language processing*, pages 11–35, 2000.

[71] J. Porta, M. Kupietz, H. Biber, H. Lüngen, P. Bański, and E. Breiteneder. From Several Hundred Million to Some Billion Words: Scaling Up a Corpus Indexer and a Search Engine With MapReduce. In *Proceedings of the workshop on Challenges in the Management of Large Corpora (CMLC-2)*, pages 25–29, 2014.

[72] M. F. Porter et al. An Algorithm for Suffix Stripping. *Program*, 14(3):130–137, 1980.

[73] P. Rayson. From Key Words to Key Semantic Domains. *International Journal of Corpus Linguistics*, 13(4):519–549, 2008.

[74] P. Rayson, D. Archer, S. Piao, and A. M. McEnery. The UCREL Semantic Analysis System. In *Proceedings 4th International Conference on Language Resources and Evaluation*, 2004.

[75] P. Rayson and R. Garside. Comparing Corpora Using Frequency Profiling. In *Proceedings of the workshop on Comparing corpora-Volume 9*, pages 1–6. Association for Computational Linguistics, 2000.

[76] J. Rissanen and G. G. Langdon. Arithmetic Coding. *IBM Journal of Research and Development*, 23(2):149–162, 1979.

[77] P. Rychlỳ. Manatee/Bonito-A Modular Corpus Manager. In *RASLAN*, pages 65–70, 2007.

[78] A. B. Sardinha. WordSmith tools. *Computers & Texts 12 (1996)*, 1996.

[79] R. Schäfer and F. Bildhauer. Building Large Corpora From the Web Using a New Efficient Tool Chain. In *LREC*, pages 486–493, 2012.

[80] R. Schäfer and L. W. C. DFG. Processing and Querying Large Web Corpora With the COW14 Architecture. *Challenges in the Management of Large Corpora (CMLC-3)*, page 28, 2015.

[81] C. Schnober. Using Information Retrieval Technology for a Corpus Analysis Platform. In *KONVENS*, pages 199–207, 2012.

[82] A. Shaker and D. S. Reeves. Self-Stabilizing Structured Ring Topology p2p Systems. In *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*, pages 39–46. IEEE, 2005.

[83] C. Shaoul and C. Westbury. A USENET Corpus (2005–2009). *University of Alberta, Canada*, 2009.

[84] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applica-

tions. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[85] M. Straka, J. Hajic, and J. Straková. UDPipe: Trainable Pipeline for Processing CoNLL-U Files Performing Tokenization, Morphological Analysis, Pos Tagging and Parsing. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 4290–4297, 2016.

[86] C. J. Tauro, S. Aravindh, and A. Shreeharsha. Comparative Study of the New Generation, Agile, Scalable, High Performance NOSQL Databases. *International Journal of Computer Applications*, 48(20):1–4, 2012.

[87] V. Vandeghinste and L. Augustinus. Making a Large Treebank Searchable Online. The SoNaR Case. In *Proceedings of the LREC2014 2nd Workshop on Challenges in the Management of Large Corpora (CMLC-2)*, pages 15–20. ELRA; Paris, 2014.

[88] S. Wattam, P. Rayson, M. Alexander, and J. Anderson. Experiences with Parallelisation of an Existing NLP Pipeline: Tagging Hansard. In *LREC*, pages 4093–4096, 2014.

[89] J. J. Webster and C. Kit. Tokenization As the Initial Phase in NLP. In *COLING 1992 Volume 4: The 15th International Conference on Computational Linguistics*, 1992.

[90] D. Wright. Using Word N-Grams to Identify Authors and Idiolects: A Corpus Approach to a Forensic Linguistic Problem. *International journal of corpus linguistics*, 22(2):212–241, 2017.

[91] R. E. Wylls. Empirical and Theoretical Bases of Zipf's Law. *Illinois*, 53, 1981.

[92] G. K. Zipf. Human Behavior and the Principle of Least Effort. 1949.

[93] M. M. Zloof. Query by Example. In *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition*, pages 431–438, 1975.

[94] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 59–59. IEEE, 2006.

Appendices

Appendix A

Experimental Results 1

t1 = time of first test, t2 = time of second test etc. ta = average (mean) time.

All times are in milliseconds (ms).

word	freq	t1	t2	t3	t4	t5	ta	ta-1
gauntly	1	58	21	21	26	21	29	22
croquet	1	56	21	21	20	22	28	21
patronym	1	56	21	24	21	20	28	22
ratpayers	1	59	21	21	20	21	28	21
thugutt	1	47	21	20	21	21	26	21
ogies	1	31	21	21	21	21	23	21
fecias	1	63	20	21	21	21	29	21
gacious	1	48	20	20	21	20	26	20
unspared	1	60	21	21	21	21	29	21
moyland	1	21	20	21	21	21	21	21
weeny	28	139	21	22	25	21	46	22
kilometers	28	103	21	22	21	22	38	22
plebs	28	81	21	21	22	21	33	21
appraiser	28	54	21	21	20	21	27	21
earldoms	28	105	21	21	21	21	38	21
candlemas	28	73	22	21	21	21	32	21
laudations	28	50	21	21	20	21	27	21
coachmakers	28	60	21	21	20	21	29	21
heinkel	28	111	21	21	21	21	39	21
conegate	28	142	21	21	21	21	45	21
it	18358013	161617	5357	5429	5597	5627	36725	5503
i	22062590	190707	70376	77344	72931	72892	96850	73386
is	22102628	81032	80796	80591	80365	80125	80582	80469
a	25980641	80596	80428	80398	80390	80021	80367	80309
in	32033919	99537	99773	99761	100039	99803	99783	99844
and	32911704	105452	104283	104154	104228	104088	104441	104188
that	37707967	154634	135725	135410	135455	135441	139333	135508
to	51541721	204702	164010	164241	164610	164101	172333	164241
of	57134689	238192	176470	176444	176507	176705	188864	176532
the	117510509	455524	424382	424640	424597	424560	430741	424545

Table A.1: Query times (ms) MongoDB 4 node cluster

word	freq	t1	t2	t3	t4	t5	ta	ta-1
gauntly	1	69	20	20	20	23	30	21
croquet	1	21	20	20	20	20	20	20
patronym	1	22	20	20	20	20	20	20
ratpayers	1	21	20	20	19	20	20	20
thugutt	1	20	20	19	20	20	20	20
ogies	1	22	19	20	20	20	20	20
fecias	1	21	20	20	20	19	20	20
gacious	1	21	20	19	20	20	20	20
unspared	1	22	20	20	20	20	20	20
moyland	1	21	20	20	20	20	20	20
weeny	28	25	20	20	21	21	21	21
kilometers	28	30	20	20	20	20	22	20
plebs	28	28	20	20	21	21	22	21
appraiser	28	26	21	20	21	21	22	21
earldoms	28	29	20	20	20	20	22	20
candlemas	28	29	20	20	20	20	22	20
laudations	28	30	19	20	20	20	22	20
coachmakers	28	35	20	21	20	20	23	20
heinkel	28	31	20	20	20	20	22	20
conegate	28	41	20	20	20	20	24	20
it	18358013	106998	4361	4094	4034	4074	24712	4141
i	22062590	111147	5201	5189	5216	5227	26396	5208
is	22102628	104715	5385	5314	5357	5323	25219	5345
a	25980641	102845	6030	6053	6087	6161	25435	6083
in	32033919	107884	7260	7238	7242	7233	27371	7243
and	32911704	109151	7394	7261	7377	7277	27692	7327
that	37707967	122596	8423	8437	8347	8234	31207	8360
to	51541721	169546	12369	12299	12237	12237	43738	12286
of	57134689	310229	13308	13566	13822	13195	72824	13473
the	117510509	642588	521226	516511	512301	510788	540683	515207

Table A.2: Query times (ms) MongoDB 8 node cluster

word	freq	t1	t2	t3	t4	t5	ta	ta-1
gauntly	1	359	22	22	27	24	91	24
croquet	1	24	27	22	21	22	23	23
patronym	1	21	22	22	21	21	21	22
ratpayers	1	23	21	21	21	23	22	22
thugutt	1	23	22	21	21	21	22	21
ogies	1	23	25	20	22	22	22	22
fecias	1	23	21	26	32	21	25	25
gacious	1	23	24	28	25	28	26	26
unspared	1	30	22	25	27	36	28	28
moyland	1	25	26	55	24	22	30	32
weeny	28	32	24	26	21	26	26	24
kilometers	28	28	24	24	22	22	24	23
plebs	28	28	25	22	22	22	24	23
appraiser	28	34	23	90	22	22	38	39
earldoms	28	25	22	22	21	25	23	23
candlemas	28	44	25	24	22	22	27	23
laudations	28	28	24	25	22	25	25	24
coachmakers	28	29	31	26	27	32	29	29
heinkel	28	36	25	25	22	22	26	24
conegate	28	36	22	21	32	24	27	25
it	18358013	46650	2516	2490	2551	2457	11333	2504
i	22062590	46670	2994	2894	2913	2917	11678	2930
is	22102628	41307	2972	2930	2915	2931	10611	2937
a	25980641	45397	3207	3283	3248	3333	11694	3268
in	32033919	50214	4093	3992	3859	3879	13207	3956
and	32911704	43207	4125	4152	4122	3908	11903	4077
that	37707967	39895	5154	5055	5026	5223	12071	5115
to	51541721	45373	6860	6505	6463	6507	14342	6584
of	57134689	46361	7122	7125	7471	7269	15070	7247
the	117510509	186992	14821	15002	15420	15188	49485	15108

Table A.3: Query times (ms) MongoDB 16 node cluster

word	freq	t1	t2	t3	t4	t5	ta	ta-1
gauntly	1	60	24	24	24	25	31	24
croquet	1	24	23	22	23	22	23	23
patronym	1	33	25	24	24	26	26	25
ratpayers	1	24	27	23	24	25	25	25
thugutt	1	25	25	24	23	27	25	25
ogies	1	31	33	29	22	24	28	27
fecias	1	26	24	23	23	22	24	23
gacious	1	24	24	27	27	28	26	27
unspared	1	25	23	24	23	23	24	23
moyland	1	39	23	24	27	30	29	26
weeny	28	67	33	33	30	31	39	32
kilometers	28	38	31	28	28	27	30	29
plebs	28	46	29	32	26	26	32	28
appraiser	28	44	31	32	35	33	35	33
earldoms	28	36	29	27	25	26	29	27
candlemas	28	30	27	31	28	25	28	28
laudations	28	35	29	26	26	26	28	27
coachmakers	28	34	27	27	27	25	28	27
heinkel	28	35	27	25	27	26	28	26
conegate	28	31	29	26	26	27	28	27
it	18358013	88	32	33	35	29	43	32
i	22062590	112	38	26	36	30	48	33
is	22102628	46	28	28	26	41	34	31
a	25980641	83	31	29	29	29	40	30
in	32033919	43	28	28	27	27	31	28
and	32911704	53	26	59	27	27	38	35
that	37707967	68	26	41	27	31	39	31
to	51541721	55	26	36	26	27	34	29
of	57134689	105	31	29	28	31	45	30
the	117510509	52	28	28	28	26	32	28

Table A.4: Query times (ms) Cassandra 4 node cluster

word	freq	t1	t2	t3	t4	t5	ta	ta-1
gauntly	1	38	22	23	23	22	26	23
croquet	1	24	22	22	22	22	22	22
patronym	1	32	23	23	23	24	25	23
ratpayers	1	25	22	23	24	24	24	23
thugutt	1	23	23	25	22	23	23	23
ogies	1	22	23	21	21	23	22	22
fecias	1	22	22	22	22	22	22	22
gacious	1	22	23	22	21	22	22	22
unspared	1	21	22	22	22	22	22	22
moyland	1	23	22	22	22	24	23	23
weeny	28	33	27	24	23	24	26	25
kilometers	28	24	24	25	24	25	24	25
plebs	28	26	23	24	24	25	24	24
appraiser	28	33	27	26	27	26	28	27
earldoms	28	26	23	24	24	23	24	24
candlemas	28	25	25	25	25	25	25	25
laudations	28	22	23	22	22	22	22	22
coachmakers	28	25	24	24	24	23	24	24
heinkel	28	25	25	23	27	23	25	25
conegate	28	21	23	22	23	24	23	23
it	18358013	31	27	25	25	25	27	26
i	22062590	29	25	27	28	25	27	26
is	22102628	26	26	25	25	27	26	26
a	25980641	253	25	26	25	26	71	26
in	32033919	27	25	25	24	25	25	25
and	32911704	30	26	27	26	26	27	26
that	37707967	29	26	26	28	26	27	27
to	51541721	33	25	26	24	23	26	25
of	57134689	29	24	25	25	24	25	25
the	117510509	30	24	22	24	22	24	23

Table A.5: Query times (ms) Cassandra 8 node cluster

word	freq	t1	t2	t3	t4	t5	ta	ta-1
gauntly	1	40	23	23	23	22	26	23
croquet	1	24	33	25	28	23	27	27
patronym	1	23	23	23	25	23	23	24
ratpayers	1	22	23	22	24	22	23	23
thugutt	1	23	24	24	23	22	23	23
ogies	1	23	24	24	24	23	24	24
fecias	1	23	22	22	21	22	22	22
gacious	1	25	21	21	22	22	22	22
unspared	1	22	22	23	22	21	22	22
moyland	1	22	22	23	21	22	22	22
weeny	28	39	26	27	26	26	29	26
kilometers	28	22	22	24	22	22	22	23
plebs	28	22	22	22	23	23	22	23
appraiser	28	29	26	26	25	25	26	26
earldoms	28	32	23	23	23	23	25	23
candlemas	28	23	22	22	21	28	23	23
laudations	28	23	22	23	22	24	23	23
coachmakers	28	23	22	22	22	22	22	22
heinkel	28	22	23	23	23	23	23	23
conegate	28	21	22	22	21	21	21	22
it	18358013	30	27	32	31	27	29	29
i	22062590	25	26	24	25	25	25	25
is	22102628	25	25	28	30	27	27	28
a	25980641	26	25	25	25	24	25	25
in	32033919	25	24	26	24	25	25	25
and	32911704	27	25	26	27	25	26	26
that	37707967	27	25	25	24	24	25	25
to	51541721	25	24	24	24	24	24	24
of	57134689	25	22	23	26	23	24	24
the	117510509	26	24	25	23	23	24	24

Table A.6: Query times (ms) Cassandra 16 node cluster

Appendix B

Experimental Results 2

term	freq	corpus	1 node	2 nodes	4 nodes
the	40743718	eebo	23804	6801	1921
of	28049678	eebo	5056	2212	1223
and	24941692	eebo	4320	1794	1273
to	19692295	eebo	4094	1869	1009
in	13423220	eebo	3495	1513	869
that	11899383	eebo	2945	1285	763
a	9971399	eebo	2411	1059	667
is	8401733	eebo	2162	1053	523
it	7217334	eebo	1912	942	482
his	6748581	eebo	1823	871	660
the	108400120	hansard	77554	22242	7792
of	54755688	hansard	12626	7400	2337
to	49684555	hansard	12508	5512	2321
that	35621789	hansard	11447	3664	1637
and	32621665	hansard	7313	2854	1601
in	27959294	hansard	7752	3136	1760
a	25092849	hansard	6299	3613	1532
I	22046663	hansard	5071	3151	1205
is	21539984	hansard	4652	2215	1328
not	15135930	hansard	3934	1843	994

Table B.1: Query times (ms) LexiDB KWIC search

term	freq	corpus	1 node	2 nodes	4 nodes
the	40743718	eebo	167576	80798	39590
of	28049678	eebo	122585	54237	26965
and	24941692	eebo	111693	49544	24577
to	19692295	eebo	85640	37750	19342
in	13423220	eebo	58655	27287	13995
that	11899383	eebo	45529	21558	11085
a	9971399	eebo	41088	19325	9907
is	8401733	eebo	31007	15026	7661
it	7217334	eebo	26761	12758	6549
his	6748581	eebo	27015	12869	6944
the	108400120	hansard	386920	179127	98252
of	54755688	hansard	187106	96880	47333
to	49684555	hansard	170487	86686	43072
that	35621789	hansard	122651	56805	29010
and	32621665	hansard	117492	55401	28066
in	27959294	hansard	107583	50259	25595
a	25092849	hansard	89335	42165	21452
I	22046663	hansard	67831	33378	16857
is	21539984	hansard	71195	33419	17126
not	15135930	hansard	49202	22916	11738

Table B.2: Query times (ms) LexiDB Collocation search

term	freq	corpus	1 node	2 nodes	4 nodes
the	40743718	eebo	33605	12636	6931
of	28049678	eebo	26389	9879	5475
and	24941692	eebo	20969	9201	5319
to	19692295	eebo	16317	6963	3980
in	13423220	eebo	13215	6073	3617
that	11899383	eebo	12265	4083	2402
a	9971399	eebo	8620	3755	2164
is	8401733	eebo	8868	2913	1721
it	7217334	eebo	5313	2398	1392
his	6748581	eebo	5595	2535	1454
the	108400120	hansard	77074	34222	14268
of	54755688	hansard	35568	15923	7964
to	49684555	hansard	39755	14948	7379
that	35621789	hansard	40699	11558	4949
and	32621665	hansard	39457	9927	5099
in	27959294	hansard	42739	9950	5240
a	25092849	hansard	30250	7341	3783
I	22046663	hansard	11897	5471	2733
is	21539984	hansard	13368	6041	3086
not	15135930	hansard	10280	4238	2138

Table B.3: Query times (ms) LexiDB NGram search

Appendix C

Experimental Results 3

query										
”””c5””:””NN1”””	1033	837	670	641	656	636	628	630	624	626
”””c5””:””PUN”””	329	317	316	318	316	316	317	323	316	319
”””c5””:””AT0”””	249	277	248	247	255	245	246	245	247	244
”””c5””:””PRP”””	225	231	227	230	227	227	228	232	233	249
”””c5””:””AJ0”””	184	182	183	182	183	183	183	185	181	182
”””c5””:””NN2”””	152	156	155	155	154	165	154	153	153	154
”””c5””:””PNP”””	148	148	156	146	144	143	142	141	141	142
”””c5””:””AV0”””	133	131	134	134	131	138	135	132	131	134
”””c5””:””NP0”””	114	114	113	113	115	122	117	114	114	121
”””c5””:””CJC”””	112	103	104	105	110	109	103	102	103	102
”””c5””:””PRF”””	91	91	94	89	91	89	95	96	92	92
”””c5””:””VVI”””	81	73	76	71	73	73	72	73	72	71
”””c5””:””DT0”””	70	70	73	80	71	72	71	70	71	70
”””c5””:””VVN”””	60	60	60	60	62	62	57	59	61	61
”””c5””:””VVD”””	55	58	53	53	53	57	58	54	55	54
”””c5””:””CRD”””	56	54	54	53	52	55	54	54	54	55
”””c5””:””PUQ”””	49	51	48	49	49	48	50	48	49	49
”””c5””:””TO0”””	49	48	49	45	46	51	48	46	49	45
”””c5””:””VM0”””	44	42	41	41	41	40	40	43	42	41
”””c5””:””DPS”””	40	41	41	40	40	40	43	42	43	40
”””c5””:””AT0””””c5””:””NN1”””	3134	2978	2913	2792	2746	2669	2850	2834	2767	2648
”””c5””:””NN1””””c5””:””PUN”””	3086	3077	2992	3045	3047	3053	3022	3009	2963	2988
”””c5””:””PRP””””c5””:””AT0”””	1986	2066	1851	2066	1915	1995	1845	2013	1960	1916
”””c5””:””AJ0””””c5””:””NN1”””	2250	2240	2215	2258	2233	2252	2213	2323	2249	2297
”””c5””:””AT0””””c5””:””AJ0”””	1774	1730	1703	1718	1720	1772	1684	1748	1658	1774
”””c5””:””NN1””””c5””:””PRF”””	2050	2001	1793	2042	1781	1922	1781	1804	2060	1991
”””c5””:””NN1””””c5””:””PRP”””	2473	2386	2455	2475	2519	2405	2570	2478	2498	2471
”””c5””:””NN2””””c5””:””PUN”””	1701	1672	1687	1777	1621	1650	1762	1622	1624	1776

Table C.1: Query times (ms) LexiDB common POS patterns

query										
"""c5"":""PUN""""c5"":""PNP"""	1641	1646	1815	1652	1674	1676	1669	1700	1814	1650
"""c5"":""TO0""""c5"":""VVI"""	460	410	428	419	416	418	438	429	293	286
"""c5"":""AJ0""""c5"":""NN2"""	1120	1205	1234	1328	1301	1309	1314	1313	1325	1317
"""c5"":""NN1""""c5"":""NN1"""	1216	1225	1246	1218	1248	1216	1217	1209	1222	1213
"""c5"":""PUN""""c5"":""AT0"""	2087	2099	2168	2111	2077	2119	2094	2103	2094	2121
"""c5"":""PRP""""c5"":""NN1"""	2425	2278	2258	2304	2308	2255	2255	2295	2322	2239
"""c5"":""PUN""""c5"":""CJC"""	1616	1565	1378	1549	1579	1449	1425	1382	1683	1587
"""c5"":""PUN""""c5"":""PUQ"""	1368	1364	1399	1341	1339	1257	1166	1262	1355	1358
"""c5"":""PRF""""c5"":""AT0"""	1237	1221	1240	1224	1221	1220	1224	1082	1118	1204
"""c5"":""NP0""""c5"":""PUN"""	1385	1666	1656	1528	1539	1579	1535	1629	1537	1600
"""c5"":""VVN""""c5"":""PRP"""	916	937	1006	902	1029	1022	970	897	900	972
"""c5"":""NN1""""c5"":""CJC"""	1967	1918	1774	1836	1790	1964	1862	1951	1984	1791
"""c5"":""PRP""""c5"":""AT0""""c5"":""NN1"""	3630	3624	3446	3653	3684	3489	3599	3652	3458	3649
"""c5"":""AT0""""c5"":""AJ0""""c5"":""NN1"""	3438	3313	3424	3323	3387	3369	3421	3336	3441	3363
"""c5"":""AT0""""c5"":""NN1""""c5"":""PRF"""	3001	3194	3205	3159	3229	3099	3286	3209	3287	3175
"""c5"":""AT0""""c5"":""NN1""""c5"":""PUN"""	4126	4085	3987	4085	3909	4147	4054	4036	3922	4083
"""c5"":""AJ0""""c5"":""NN1""""c5"":""PUN"""	3829	3811	3805	3849	3825	3844	3768	3937	3797	3844
"""c5"":""NN1""""c5"":""PRP""""c5"":""AT0"""	3695	3695	3516	3687	3499	3674	3527	3715	3565	3778
"""c5"":""PRP""""c5"":""AT0""""c5"":""AJ0"""	2866	2674	2735	2712	2693	2654	2743	2767	2735	2643
"""c5"":""NN1""""c5"":""PRF""""c5"":""AT0"""	2958	3062	3080	3160	3014	3057	2875	3013	3086	3115
"""c5"":""PUN""""c5"":""AT0""""c5"":""NN1"""	3952	3950	4056	3976	3833	4052	3981	4072	3985	4087
"""c5"":""AT0""""c5"":""NN1""""c5"":""PRP"""	3653	3670	3710	3741	3731	3466	3722	3713	3654	3456
"""c5"":""PRF""""c5"":""AT0""""c5"":""NN1"""	2760	2914	2771	2968	2754	2759	2946	2950	2735	2690
"""c5"":""AJ0""""c5"":""NN1""""c5"":""PRP"""	3303	3276	3356	3273	3315	3248	3400	3203	3323	3244
"""c5"":""NN1""""c5"":""PUN""""c5"":""PNP"""	3632	3419	3623	3439	3677	3476	3626	3457	3595	3462
"""c5"":""NN1""""c5"":""PUN""""c5"":""AT0"""	4248	4104	3958	4219	4173	4147	4008	4203	4160	3927
"""c5"":""NN1""""c5"":""PRF""""c5"":""NN1"""	1839	1924	1754	1731	1837	1929	1782	1837	1774	1821
"""c5"":""AT0""""c5"":""NN1""""c5"":""NN1"""	2399	2402	2384	2563	2441	2488	2432	2423	2521	2415
"""c5"":""AJ0""""c5"":""NN2""""c5"":""PUN"""	2653	2560	2596	2472	2524	2528	2441	2565	2464	2546
"""c5"":""NN1""""c5"":""NN1""""c5"":""PUN"""	2811	2626	2613	2860	2692	2641	2801	2825	2689	2832
"""c5"":""AJ0""""c5"":""NN1""""c5"":""PRF"""	2738	2705	2732	2782	2647	2816	2733	2697	2751	2733

Table C.2: Query times (ms) LexiDB common POS patterns (continued)

query										
"[c5=""NN1""]"	2084	2074	2090	2074	2085	2071	2086	2083	2090	2080
"[c5=""PUN""]"	1634	1640	1631	1628	1632	1641	1635	1627	1625	1641
"[c5=""AT0""]"	1303	1301	1293	1292	1301	1292	1299	1290	1291	1294
"[c5=""PRP""]"	1188	1198	1195	1197	1193	1196	1189	1192	1189	1189
"[c5=""AJ0""]"	986	985	985	984	996	987	989	980	989	987
"[c5=""NN2""]"	820	814	814	814	815	809	813	814	825	816
"[c5=""PNP""]"	780	777	784	772	773	771	776	772	771	774
"[c5=""AV0""]"	719	716	716	716	714	715	716	716	715	719
"[c5=""NP0""]"	603	605	603	603	603	615	604	597	603	597
"[c5=""CJC""]"	556	558	555	556	554	555	556	558	557	552
"[c5=""PRF""]"	495	501	496	497	500	497	494	494	494	492
"[c5=""VVI""]"	387	385	388	385	383	384	389	382	387	382
"[c5=""DT0""]"	389	388	391	383	390	388	386	388	385	387
"[c5=""VVN""]"	345	344	343	344	342	345	345	344	340	342
"[c5=""VVD""]"	319	316	317	319	315	315	313	316	318	313
"[c5=""CRD""]"	316	318	311	311	311	312	309	313	310	311
"[c5=""PUQ""]"	284	278	279	280	279	276	278	277	279	282
"[c5=""TO0""]"	286	282	280	280	283	277	278	277	278	281
"[c5=""VM0""]"	255	251	252	253	253	254	253	250	253	252
"[c5=""DPS""]"	250	248	246	245	248	245	245	247	246	245
"[c5=""AT0""][c5=""NN1""]"	6495	6485	6490	6525	6502	6504	6487	6483	6498	6498
"[c5=""NN1""][c5=""PUN""]"	7703	7704	7674	7673	7673	7678	7708	7712	7681	7688
"[c5=""PRP""][c5=""AT0""]"	6308	6347	6333	6417	6351	6290	6302	6305	6292	6304
"[c5=""AJ0""][c5=""NN1""]"	6034	5943	5951	5950	5943	5943	5942	5939	5938	5939
"[c5=""AT0""][c5=""AJ0""]"	6428	6459	6435	6425	6417	6429	6423	6427	6423	6421
"[c5=""NN1""][c5=""PRF""]"	7653	7626	7620	7638	7621	7618	7628	7622	7621	7622
"[c5=""NN1""][c5=""PRP""]"	7632	7636	7623	7626	7641	7642	7629	7634	7692	7623
"[c5=""NN2""][c5=""PUN""]"	5534	5541	5566	5532	5537	5545	5569	5534	5536	5575
"[c5=""PUN""][c5=""PNP""]"	6931	6920	6919	6934	6961	6929	6924	6921	6936	6920

Table C.3: Query times (ms) CWB common POS patterns

query										
”[c5=””TO0””][c5=””VVI””]”	3980	3975	4008	3979	3986	3982	3976	3982	3980	3975
”[c5=””AJ0””][c5=””NN2””]”	5892	5897	5892	5900	5888	5886	5923	5887	5901	5897
”[c5=””NN1””][c5=””NN1””]”	7614	7619	7605	7600	7605	7605	7601	7603	7605	7600
”[c5=””PUN””][c5=””AT0””]”	6918	6916	6921	6948	6908	6915	6919	6907	6928	6955
”[c5=””PRP””][c5=””NN1””]”	6235	6237	6246	6240	6234	6275	6241	6236	6233	6245
”[c5=””PUN””][c5=””CJC””]”	6917	6915	6911	6920	6914	6911	6904	6948	6908	6910
”[c5=””PUN””][c5=””PUQ””]”	6913	6914	6958	6910	6912	6913	6909	6917	6917	6911
”[c5=””PRF””][c5=””AT0””]”	4829	4860	4824	4824	4827	4826	4828	4836	4826	4824
”[c5=””NP0””][c5=””PUN””]”	4339	4324	4334	4324	4327	4333	4328	4323	4318	4320
”[c5=””VVN””][c5=””PRP””]”	4339	4338	4348	4370	4337	4334	4332	4352	4344	4343
”[c5=””NN1””][c5=””CJC””]”	7628	7590	7593	7595	7598	7596	7608	7586	7583	7591
”[c5=””PRP””][c5=””AT0””][c5=””NN1””]”	6484	6480	6480	6486	6485	6486	6497	6491	6481	6487
”[c5=””AT0””][c5=””AJ0””][c5=””NN1””]”	6571	6568	6581	6574	6573	6570	6574	6616	6573	6576
”[c5=””AT0””][c5=””NN1””][c5=””PRF””]”	6766	6764	6766	6768	6764	6771	6764	6784	6841	6765
”[c5=””AT0””][c5=””NN1””][c5=””PUN””]”	6756	6770	6771	6766	6769	6764	6759	6758	6801	6758
”[c5=””AJ0””][c5=””NN1””][c5=””PUN””]”	6095	6100	6094	6092	6090	6132	6101	6091	6125	6091
”[c5=””NN1””][c5=””PRP””][c5=””AT0””]”	7806	7847	7817	7809	7806	7836	7950	7821	7850	7824
”[c5=””PRP””][c5=””AT0””][c5=””AJ0””]”	6470	6462	6459	6535	6468	6466	6464	6462	6462	6452
”[c5=””NN1””][c5=””PRF””][c5=””AT0””]”	7825	7855	7805	7808	7827	7827	7805	7815	7817	7846
”[c5=””PUN””][c5=””AT0””][c5=””NN1””]”	7058	7046	7035	7047	7032	7030	7052	7055	7030	7038
”[c5=””AT0””][c5=””NN1””][c5=””PRP””]”	6797	6760	6761	6751	6755	6761	6748	6755	6790	6750
”[c5=””PRF””][c5=””AT0””][c5=””NN1””]”	4898	4858	4856	4854	4851	4849	4890	4854	4848	4979
”[c5=””AJ0””][c5=””NN1””][c5=””PRP””]”	6188	6088	6084	6080	6083	6088	6087	6091	6085	6084
”[c5=””NN1””][c5=””PUN””][c5=””PNP””]”	7977	7927	7943	7924	7936	7934	7958	7931	7946	7927
”[c5=””NN1””][c5=””PUN””][c5=””AT0””]”	7951	7935	7960	7948	7937	7935	7925	7940	7932	7971
”[c5=””NN1””][c5=””PRF””][c5=””NN1””]”	7848	7807	7808	7805	7800	7849	7808	7839	7817	7841
”[c5=””AT0””][c5=””NN1””][c5=””NN1””]”	6754	6749	6742	6784	6745	6782	6738	6745	6748	6746
”[c5=””AJ0””][c5=””NN2””][c5=””PUN””]”	5993	5967	5969	5972	5977	5968	6004	5971	5967	6008
”[c5=””NN1””][c5=””NN1””][c5=””PUN””]”	7765	7840	7761	7754	7759	7763	7779	7765	7756	7765
”[c5=””AJ0””][c5=””NN1””][c5=””PRF””]”	6078	6116	6086	6080	6088	6080	6081	6093	6120	6081

Table C.4: Query times (ms) CWB common POS patterns (continued)

query											
NN1	1970	1500	1491	1487	1487	1526	1509	1486	1473	1478	1541
PUN	1479	1501	1517	1494	1505	1493	1465	1458	1477	1481	1487
AT0	1470	1489	1540	1522	1573	1588	1480	1490	1466	1474	1509
PRP	1486	1465	1456	1474	1495	1465	1475	1496	1531	1482	1483
AJ0	1471	1472	1474	1565	1541	1455	1456	1462	1462	1469	1483
NN2	1463	1471	1459	1457	1466	1465	1466	1465	1463	1463	1464
PNP	1461	1457	1454	1472	1456	1456	1469	1460	1460	1455	1460
AV0	1457	1460	1463	1457	1451	1461	1458	1466	1473	1471	1462
NP0	1469	1454	1456	1452	1457	1461	1458	1458	1461	1457	1458
CJC	1465	1467	1460	1464	1478	1461	1458	1461	1465	1458	1464
PRF	1457	1456	1458	1454	1462	1455	1452	1464	1467	1460	1459
VVI	1460	1456	1458	1460	1457	1462	1456	1455	1465	1455	1458
DT0	1476	1465	1459	1465	1460	1465	1455	1464	1457	1462	1463
VVN	1460	1469	1455	1470	1461	1455	1453	1462	1470	1462	1462
VVD	1461	1454	1460	1464	1462	1453	1452	1458	1459	1466	1459
CRD	1465	1457	1455	1455	1452	1452	1455	1453	1453	1453	1455
PUQ	1307	1307	1307	1319	1311	1323	1308	1309	1317	1312	1312
TO0	1462	1457	1465	1458	1454	1453	1455	1455	1456	1466	1458
VM0	1466	1473	1464	1459	1461	1472	1466	1453	1456	1457	1463
DPS	1454	1452	1460	1457	1459	1468	1467	1454	1456	1464	1459
AT0 NN1	1819	1812	1792	1786	1784	1785	1791	1786	1784	1804	1794
NN1 PUN	1824	1823	1821	1842	1842	1825	1836	1836	1831	1830	1831
PRP AT0	1702	1705	1702	1710	1699	1722	1710	1705	1700	1711	1707
AJ0 NN1	1756	1754	1733	1750	1749	1747	1743	1741	1750	1754	1748
AT0 AJ0	1670	1678	1702	1689	1703	1692	1679	1680	1669	1667	1683
NN1 PRF	1668	1671	1680	1674	1659	1668	1667	1663	1664	1667	1668
NN1 PRP	1752	1757	1767	1749	1756	1768	1763	1754	1759	1757	1758
NN2 PUN	1679	1677	1669	1672	1668	1673	1669	1671	1665	1667	1671
PUN PNP	1666	1667	1674	1668	1662	1662	1660	1660	1667	1660	1665
TO0 VVI	1528	1524	1541	1507	1510	1514	1514	1505	1515	1520	1518

Table C.5: Query times (ms) Lucene common POS patterns

query											
AJ0 NN2	1617	1621	1611	1611	1609	1619	1612	1611	1616	1618	1615
NN1 NN1	1779	1779	1772	1779	1783	1786	1792	1790	1786	1785	1783
PUN AT0	1715	1714	1720	1716	1723	1720	1723	1781	1761	1942	1752
PRP NN1	1810	1760	1767	1790	1956	1947	1879	1886	1776	1784	1836
PUN CJC	1697	1643	1636	1655	1637	1631	1644	1702	1781	1757	1678
PUN PUQ	1461	1421	1471	1464	1444	1521	1465	1608	1449	1492	1480
PRF AT0	1788	1746	1828	1735	1722	1741	1713	1725	1734	1822	1755
NP0 PUN	1676	1677	1666	1651	1795	1658	1632	1639	1675	1654	1672
VVN PRP	1606	1612	1576	1573	1615	1591	1583	1626	1624	1639	1605
NN1 CJC	1685	1661	1707	1697	1711	1722	1755	1724	1737	1747	1715
PRP AT0 NN1	1906	1934	1912	1959	1930	1941	1901	1893	1957	2062	1940
AT0 AJ0 NN1	1988	1980	1969	1984	1978	1944	1964	1953	1940	1948	1965
AT0 NN1 PRF	1915	1922	1929	1966	1954	1932	1949	1945	1941	1949	1940
AT0 NN1 PUN	2155	2016	2018	1992	1961	1933	1925	2060	2268	2149	2048
AJ0 NN1 PUN	2086	2015	2037	2005	2151	2063	2033	2017	2007	2053	2047
NN1 PRP AT0	2037	2019	2022	2047	2000	2015	2005	2009	2005	1999	2016
PRP AT0 AJ0	1918	1895	1903	1920	1920	1900	1906	1895	1891	1899	1905
NN1 PRF AT0	1921	1922	1922	1902	1905	1913	1930	1931	1924	1917	1919
PUN AT0 NN1	2021	2072	2120	2046	2008	2007	1997	1983	2056	1986	2030
AT0 NN1 PRP	1999	2018	1983	2006	2003	2065	2039	2015	1998	2018	2014
PRF AT0 NN1	1948	1892	1853	1860	1862	1870	1936	1865	1865	1891	1884
AJ0 NN1 PRP	1959	1961	1961	1967	1999	1990	1964	1957	1970	2014	1974
NN1 PUN PNP	1975	1974	1969	1994	2007	2013	2058	1971	1988	1894	1984
NN1 PUN AT0	2095	2071	2062	2062	2017	2053	2059	2079	2053	2065	2062
NN1 PRF NN1	1981	1935	1956	1956	1965	1985	1981	1947	1945	1937	1959
AT0 NN1 NN1	2090	2133	2128	2092	2112	2114	2138	2176	2113	2091	2119
AJ0 NN2 PUN	1872	1867	1867	1860	1875	1872	1862	1869	1865	1854	1866
NN1 NN1 PUN	2045	2079	2056	2125	2122	2057	2088	2047	2069	2089	2078
AJ0 NN1 PRF	1891	1891	1883	1878	1934	1928	1899	1898	1878	1875	1896
VVN PRP AT0	1791	1792	1797	1851	1775	1777	1783	1780	1817	1798	1796

Table C.6: Query times (ms) Lucene common POS patterns (continued)

Appendix D

Sample Focus Group Questions

D.1 Using the web interface

Which elements of the graphical interface were most useful?

How did the query language compare to similar languages such as CQL?

Are there any obvious deficiencies that could be easily addressed?

Do you foresee that you would use the system in the future?

D.2 Comparisons to other systems

What other corpus data systems have you used in the past?

Does LexiDB compare favourably to them?

Are there any features that other systems have that you feel LexiDB is lacking?

Can you envisage research questions that LexiDB can help you answer, that previous systems you have used could not?

D.3 Future developments

Is the query language presented sufficient or would the ability to use existing

query language in LexiDB be preferable?
Are there any other visualizations within the web interface that would be useful for certain corpus query types?
With regards to corpora that you use regularly, what is their typical scale?
Is a system capable of supporting billions of words sufficient?
Are you interested in studies using live data sources, such as Twitter etc?
If a similar interface was available that you could upload your own corpora to, would you use it?