Towards a Concurrent Implementation of Keyword Search Over Relational Databases

by

Richard J.I. Drake

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

Master of Science (M.Sc.)

in

The Faculty of Science

Computer Science

University of Ontario Institute of Technology

Supervisor: Dr. Ken Q. Pu

May 2014

Abstract

Write me

Keywords: relational database, full-text search, graph search, algorithms

Contents

List of Tables

List of Figures

List of Algorithms

Acronyms

API application programming interface. 3, 35, 40, 41

BFS breadth-first search. 41, 48, 52, 54

FK foreign key. viii, 6–11

HTML HyperText Markup Language. 2, 45

HTTP HyperText Transfer Protocol. 73–75

JAR Java archive. 55

JDBC Java database connectivity. 30, 34

JIT just-in-time. 52

JSON JavaScript object notation. viii, 3, 46, 47

JVM Java virtual machine. vii, 30, 34, 35, 44, 46, 47, 52

MDX MultiDimensional eXpressions. 2

OLAP online analytical processing. 2

RDBMS relational database management system. 1, 2, 11–13

SQL structured query language. 1–3, 9, 20, 36, 37, 66

STM software transactional memory. 32, 33, 54

TF-IDF term frequency and inverse document frequency. 15, 16

UOIT University of Ontario Institute of Technology. i, 45

XML Extensible Markup Language. 2, 3

List of Symbols

Chapter 1

A Tale of Two Data Models

The term “data model” refers to a notation for describing data and/or information. It consists of the data structure, operations that may be performed on the data, as well as constraints placed on the data [GUW09].

In this chapter we provide background and motivation for this thesis. We will discuss the evolution of data sets, their logical models, and their corresponding query languages. We feel that modern day data sets call for a new data model with a new query languages.

We also provide a formal definition of the relational data model, discuss its merits, its shortcomings, and contrast it to the document data model. Contrary to the relational model, the document model permits fast and flexible keyword search without requiring explicit domain knowledge of the data. In addition, we demonstrate the feasibility of encoding a relational model into a document model in a lossless manner.

1.1 The Evolution of Data Models

It is important to understand the evolution of modern data models. By understanding

1.1.1 Structured Data and Structured Language: 1970 – 2000

The proliferation of database system research and development started with the disruptive invention of the relational data model by Edgar F. Codd [Cod79].

The invention of the relational data model was a significant achievement as it decoupled the task of data analytics from any one language, precisely and accurately described data sets across a variety of storage and analytical systems, and lead to the creation of the structured data analytics language known as structured query language (SQL).

SQL itself deserves further discussion; the relational data model provided a foundation upon which languages for data manipulation were designed. One can describe their data set and operations using first-order logic and relational algebra, then realize it using SQL.

The success of the relational model can only be appreciated when one looks at the continuous success of relational database management system (RDBMS) such as Oracle and IBM DB2 which span over 3 decades without any significant decline.

Since the 90s, the emergence of Business Intelligence [CKKS02] furthered the development of RDBMS by specializing the relational data model to multidimensional data model [Col96]. The family of databases known as online analytical processing (OLAP) produced a new query language known as MultiDimensional eXpressions (MDX).

Both SQL and MDX are highly structured query languages: they are completely described by their respective grammar and operational semantics. Users who wish to harness the power of SQL and MDX must be well versed in the languages themselves, and understand precisely the semantics of each syntactic constructs.

1.1.2 Text data and keyword search: 1970 – 2014

Parallel to the development of the relational data base technology, research in the information retrieval has been focusing on text data [SB88; Jon72]. Unlike the relational data, text data does not have much complex structure to its schema. Thus, it's not immediately possible to design a rich set of data operators (as was the case for relational algebra). Consequently, for text data, there is no structured query language like SQL.

The research, thus, has been focusing on pattern matching queries using keyword search. Though the semantics of keyword search is very simple, the statistical methods developed by the information retrieval community [SB88; RZ09; DFLDH88] have been extremely effective. In fact, one can argue that the modern World Wide Web and its related commercial successes founded on the ideas of text databases and keyword search over Internet scale data sets.

1.1.3 Semi-structured Data and Query Languages: 1990 – 2010

The growth of the World Wide Web popularized the usage of markup languages (such as HyperText Markup Language (HTML) and Extensible Markup Language (XML)) as the underlying Web content description. Thus, researchers have designed data models [Suc98] to formalize the semantics of XML and related data formats. Subsequently, the logical characterization of XML led the to design and implementation of XQuery [Sco10], a navigational based query language for analysis of XML data sets.

Interestingly enough, XML has proven to be inefficient as a data description format. Nonetheless, a semi-structured data description language is highly sought after for message passing in Internet scale software systems. Modern Web services are built on the concept of RESTful application programming interface (API) [Sch11], with semi-structured data message passing. In order to minimize network overhead, XML based message passing has been replaced by the more efficient data encoding standard of JSON [ECM13].

The query language community responded to the popularization of JSON encoded data sets with several query languages [BCNS13] (for example Jaql [Li13]) for JSON data sets.

1.1.4 Hybrid data models and query languages: 2010 – Present

With the explosive growth of social networks, we are witnessing the emergence of a new type of data sets. These data sets exhibit the following properties:

- The data has relational characteristics: such as relationships of friends on Facebook, their preferences over different Web sites, and their account information.
- The data also has many text attributes: such as blog articles, or tweets on Twitter.
- The volume of data is often Internet scale.

The mixture of relational structure and rich text components of such data sets make them challenging for the purpose of data management and data analytics. There has been several attempts to integrate keyword search from information retrieval with SQL [BHNCS02; LJLF11; HGP03]. However, these methods, thus far, suffer from scalability and restricted search capabilities.

In this thesis, we are motivated by the following problems:

- Define a formal framework to describe data sets with relational structures and text components.
- Design a collection of expressive query operators for analyzing text relational data sets.
- Investigate implementation techniques to make the query operators performant for modern multi-core computers.

1.2 Relational Model

In its most basic form, the relational data model is built upon sets and tuples. Each of these sets consist of a set of finite possible values. Tuples are constructed from these sets to form relations.

Definition 1 (Named Tuple). A named tuple t is an instance of a relation r, consisting of values corresponding to the attributes of r. For example,

Example 1. Given a tuple $t = \{\text{code} : \text{“CDPS 101”}, \text{title} : \text{“Human-Mutant Relations”}, \text{subject} : \text{“CDPS”}\}$, we denote the attributes of t as $\textsc{Attr}[t] = \{\text{code, title, subject}\}$. The values are $t[\text{code}] =$ “CDPS 101”, $t[\text{title}] =$ “Human-Mutant Relations”, and $t[\text{subject}] =$ “CDPS”.

code	title	subject
CDPS 101	Human-Mutant Relations	CDPS
CDPS 201	Humans and You	CDPS
MATH 360	Complex Analysis	MATH

Table 1.1: Course relation

Definition 2 (Relation). A relation r is a set of named tuples, $r = \{t_1, t_2, \ldots, t_n\}$, such that all the named tuples share the same attributes.

$$\forall t, t' \in r, \textsc{Attr}[t] = \textsc{Attr}[t'] \tag{1.1}$$

Example 2. An example Course relation, r, would be

$$r = \left\{ \begin{array}{llllll} \{\text{code} & : \text{“CDPS 101”}, & \text{title} & : \text{“Human-Mutant Relations”}, & \text{subject} & : \text{“CDPS”}\}, \\ \{\text{code} & : \text{“CDPS 201”}, & \text{title} & : \text{“Humans and You”}, & \text{subject} & : \text{“CDPS”}\}, \\ \{\text{code} & : \text{“MATH 360”}, & \text{title} & : \text{“Complex Analysis”}, & \text{subject} & : \text{“MATH”}\} \end{array} \right\}$$

Relations are typically represented as tables.

Definition 3 (Keys). Keys are constraints imposed on relations. A key constraint K on a relation r is a subset of $\textsc{Attr}[r]$ which may uniquely identify a tuple. Formally, we say r satisfies the key constraint K, denoted as $r \vDash \mathrm{K}$, subject to

$$\forall t, t' \in r, t \neq t' \implies t[\mathrm{K}] \neq t'[\mathrm{K}]$$

For example, in Table 1.1, the relation satisfies the key constraint {code} or {title}, but not {subject}.

Definition 4 (Foreign Keys). A FK constraint applies to two relations, r_1, r_2. It asserts that values of certain attributes of r_1 must appear as values of some corresponding attributes of r_2. A FK constraint is written as

$$\theta = r_1(\alpha_{11}, \alpha_{12}, \ldots, \alpha_{1k}) \to r_2(\alpha_{21}, \alpha_{22}, \ldots, \alpha_{2k})$$

where $\alpha_{1i} \subseteq \textsc{Attr}[r_1]$ and $\alpha_{2i} \subseteq \textsc{Attr}[r_2]$. We say (r_1, r_2) satisfies θ, denoted as $(r_1, r_2) \vDash \theta$, if for every tuple $t \in r_1$, there exists a tuple $t' \in r_2$ such that $t[\alpha_{11}, \alpha_{12}, \dots, \alpha_{1k}] = t'[\alpha_{21}, \alpha_{22}, \dots, \alpha_{2k}]$.

We say r_1 is the source, while r_2 is the target.

Example 3. Suppose we have a relation Course(code, title, subject). We impose a FK constraint of

$$\theta = \text{Course(subject)} \rightarrow \text{Subject(id)} \tag{1.2}$$

which asserts (Course, Subject) $\vDash \theta$. Therefore, if

$$t = \{\text{code} : \text{“CDPS 101”}, \text{title} : \text{“Human-Mutant Relations”}, \text{subject} : \text{“CDPS”}\}$$

then $\exists!\, t' \in$ Subject such that $t'[\text{id}] =$ “CDPS”.

Definition 5 (Relational Database)**.** A relational database, D, is a named collection of relations (as defined by Definition 2 on page 5), keys (as defined by Definition 3 on page 5), and foreign key constraints (as defined by Definition 4 on page 5).

We use $\textsc{Name}[D]$ to denote the name of D, $\textsc{Rel}[D]$ the list of relations in D, $\textsc{Key}[D]$ the list of key constraints of D, and $\text{FK}[D]$ the list of foreign key constraints of D.

1.2.1 Schema Group

Definition 6 (Schema Graph)**.** If we view relations as vertices, and foreign key constraints as edges, a database D can be viewed as a *schema graph* G, formally defined as

$$\text{vertices} : \text{V}(G) = \textsc{Rel}[D] \tag{1.3}$$

$$\text{edges} : \text{E}(G) = \text{FK}[D] \tag{1.4}$$

Example 4. Given the schema in Table 1.2 on the next page

and the FK constraints in Fig. 1.1 on the following page

we produce the schema graph in Fig. 1.2 on page 8

The relational data model is particularly powerful for analytic queries. Given the schema graph in Fig. 1.2 on page 8, one can formulate the following analytic queries in a query language known as SQL.

Relation	Attributes
Course	code, title, subject
Section	id, actual, campus, capacity, credits, levels, registration_start, registration_end, semester, sec_code, sec_number, year, course
Schedule	id, date_start, date_end, day, schedtype, hour_start, hour_end, min_start, min_end, classtype, location, section_id
Instructor	id, name
Teaches	id, schedule_id, instructor_id, position

Table 1.2: Subset of mycampus dataset schema

$$\begin{aligned}
\text{Schedule(section_id)} &\rightarrow \text{Section(id)}\\
\text{Section(course)} &\rightarrow \text{Course(code)}\\
\text{Teaches(schedule_id)} &\rightarrow \text{Schedule(id)}\\
\text{Teaches(instructor_id)} &\rightarrow \text{Instructor(id)}
\end{aligned}$$

Figure 1.1: FK constraints on schema in Table 1.2

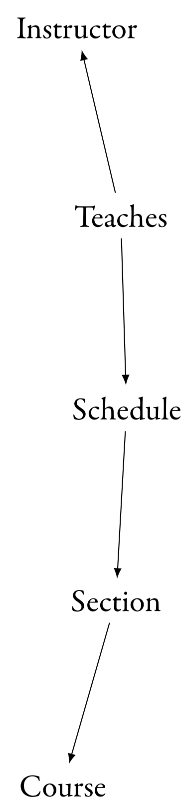


Figure 1.2: Graph representation of relations (Table 1.2) and FK (Fig. 1.1)

```sql
SELECT section.crn
FROM   section
       JOIN course
         ON section.course_code = course.code
       JOIN subject
         ON subject.id = course.subject_id
WHERE  subject.name = 'Community Development & Policy Studies';
```

Figure 1.3: Query to find section CRNs for a subject name.

crn
10000
10001
10002

Table 1.3: Results of the query in Fig. 1.3.

Example 5. Using SQL, find all section CRNs for the subject titled "Community Development & Policy Studies."

The SQL query in Fig. 1.3 results in Table 1.3.

1.2.2 Entity Group

Definition 7 (Entity Group). An entity group is a forest, T, of tuples interconnected by join conditions defined by the FK constraints in the schema graph G.

Given two vertices $t, t' \in \mathrm{V}(T)$, $\exists r_1, r_2 \in \textsc{Rel}[D]$ such that $t \in r_1, t' \in r_2$, and $(r_1, r_2) \in G$. That is, t and t' belong to two relations that are connected by the schema graph.

Let $r_1(\alpha_{11}, \alpha_{12}, \ldots, \alpha_{1k}) \rightarrow r_2(\alpha_{21}, \alpha_{22}, \ldots, \alpha_{2k})$ be the FK that connects r_1, r_2. We further assert that $t[\alpha_{11}, \alpha_{12}, \ldots, \alpha_{1k}] = t'[\alpha_{21}, \alpha_{22}, \ldots, \alpha_{2k}]$.

Entity groups define complex, structured objects that include more information than individual tuples in the relations.

Attribute	Value
code	CDPS 101
title	Human-Mutant Relations
subject	Community Development & Policy Studies

Table 1.4: Properties of the Course titled Human-Mutant Relations.

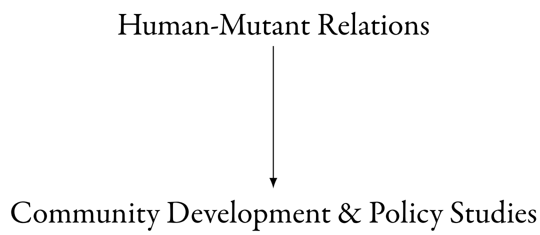


Figure 1.4: Human-Mutant Relations entity group

Example 6. The information in Table 1.4 all relates to the Course titled Human-Mutant Relations, however no single tuple in the database has all of this information as a result of database normalization.

We require an entity group to join together all pieces of information related to this course. An example of this is given in Fig. 1.4, where the an example entity group comprised of a Course and Subject is shown.

1.2.3 Pros and Cons of the Relational Model

In order to better understand the motivation behind this work, it is important to examine both the strong and weak points of the relational model.

Pros

The enforcement of constraints is essential to the relational model. There are several types of constraints, including uniqueness and FKs. The first constraint maintains uniqueness.

The Course relation (Table 1.1 on page 5) has the attribute `code` as its primary key. In order for other relations to reference a specific named tuple, the `code` attribute must be unique.

Example 7 (Unique Constraint)**.** Attempt to insert another course with a `code` of "CDPS 101."

```
INSERT INTO course
VALUES      ('CDPS 101',
             'Mutant-Human Relations',
             'CDPS');
```

The RDBMS enforces the primary key constraint on the `code` attribute, rejecting the insertion.

```
Error: column code is not unique
```

With the uniqueness of named tuples guaranteed (as demonstrated in Example 7), we must ensure that any named tuples that are referenced actually exist. If they do not, the database must not permit the operation to continue. Doing so would lead to dangling references.

Example 8 (Referential Integrity)**.** Attempt to insert the tuple ("CHEM 101", "Introductory Chemistry", "'CHEM") in the Course relation.

```
INSERT INTO course
VALUES      ('CHEM 101',
             'Introductory Chemistry',
             'CHEM');
```

Again we see the RDBMS protecting the integrity of the data.

```
Error: foreign key constraint failed
```

In addition to enforcing consistency, the relational model is capable of providing higher-level views of the data through aggregation.

Example 9 (Aggregation)**.** Find the number of sections offered for the subject named "Community Development & Policy Studies."

```
SELECT Count(*)
FROM   section
       JOIN course
```

```
          ON section.course = course.code
        JOIN subject
          ON subject.id = course.subject
WHERE   subject.name = 'Community Development & Policy Studies';
```

Information stored within a properly designed database is normalized. That is, no information is repeated.

Example 10 (Normalization)**.** For example, suppose Emma Frost became headmistress and the subject named "Community Development & Policy Studies" was renamed to "Community Destruction & Policy Studies." If this information were not normalized, each course in this subject would need to be updated. Since this information is normalized, the following query will suffice.

```
UPDATE subject
SET    name = 'Community Destruction & Policy Studies'
WHERE  id = 'CDPS';
```

The above examples are some of the most important reasons for choosing the relational model over others. Unfortunately, the relational model is not without its downsides.

Cons

While the relational model excels at ensuring data consistency, aggregation, and reporting; it is not suitable for every task. In order to issue queries, a user must be familiar with the schema. This requires specific domain knowledge of the data.

An example of a complicated query involving two joins is give in Fig. 1.3 on page 9.

A casual user is unlikely to determine the correct join path, name of the tables, name of the attributes, etc. This is in contrast to the document model, where the data is semi-structured or unstructured, requiring minimal domain knowledge.

The relational model is also rigid in structure. If a relation is modified, every query referencing said relation may require a rewrite. Even a simple attribute being renamed (e.g. $\rho_{\text{name/alias}}(\text{Person})$) is capable of modifying the join paths. This rigidity places additional cognitive burden on users.

In addition to having a rigid structure, most relational database management systems lack flexible string matching options. Assuming basic SQL-92 compliance, a RDBMS only supports the `LIKE` predicate [ISO11].

Example 11 (`LIKE` Predicate)**.** Find all courses with a title that contains "man."

```sql
SELECT *
FROM   course
WHERE  title LIKE '%man%';
```

There are a couple of limitations to the `LIKE` predicate. First, it only supports basic substring matching. If a user accidentally searches for all courses with a title containing "men," nothing would be found.

Second, unless the predicate is applied to the end of the string and the column is indexed, performance will be poor. The database must scan the entire table in order to answer the query, resulting in performance of $\mathcal{O}(n)$, where n is the number of named tuples in the relation.

1.3 Document Model

In contrast to the relational model, the document model represents semi-structured as well as unstructured data. Examples of information suitable to the document model includes emails, memos, book chapters, etc.

These pieces, or units, of information are broken into documents. Groups of related documents (for example, a library catalogue) are referred to as a document collection.

Definition 8 (Terms and Document)**.** A term, τ, is an indivisible string (e.g. a proper noun, word, or a phrase). A document, d, is a bag of words; order is irrelevant.

Let $\mathtt{freq}(\tau, d)$ be the frequency of term τ in d, T denote all possible terms, and $\textsc{Bag}[\mathrm{T}]$ be all possible bag of terms.

Remark 1. We use the bag-of-words model for documents. This means that position information of terms in a document is irrelevant, but the frequency of terms are kept in the document. Documents are non-distinct sets.

Definition 9 (Document Collection)**.** A document collection C is a set of documents, written $C = \{d_1, d_2, \ldots, d_k\}$. The cardinality of C is denoted by N.

Example 12. Consider the following short phrases

1. math 360 is a math class
2. cdps 101 is a boring lecture
3. mathematics lecture was great

Each sentence phrase produces a document, giving us the following

$$d_1 = \{\text{“math”} : 2, \text{“a”} : 1, \text{“is”} : 1, \text{“360”} : 1, \text{“class”} : 1\} \tag{1.5}$$

$$d_2 = \{\text{“a”} : 1, \text{“boring”} : 1, \text{“is”} : 1, \text{“cdps”} : 1, \text{“lecture”} : 1, \text{“101”} : 1\} \tag{1.6}$$

$$d_3 = \{\text{“mathematics”} : 1, \text{“great”} : 1, \text{“was”} : 1, \text{“lecture”} : 1\} \tag{1.7}$$

1.3.1 Vectorization of Documents

One of the most fundamental approaches for searching documents is to treat documents as high-dimensional vectors, and the document collection as a subset in a vector space. Search queries become a nearest neighbour search in a vector space using a distance metric.

The first step is to convert a bag of terms into vectors. The standard technique [MRS08] uses a scoring function that measures the relative importance of terms in documents.

Definition 10 (Term frequency and inverse document frequency (TF-IDF) Score)**.** The term frequency is the number of times a term τ appears in a document d, as given by $\mathtt{freq}(\tau, d)$. The document frequency of a term τ, denoted by $\mathrm{df}(\tau)$, is the number of documents in C that contains τ. It is defined as

$$\mathrm{df}(\tau) = |\{d \in C : \tau \in d\}|$$

The combined TF-IDF score of τ in a document d is given by

$$\text{tf-idf}(C, \tau, d) = \frac{\mathtt{freq}(\tau, d)}{|d|} \cdot \log \frac{N}{\mathrm{df}(\tau)}$$

The first component, $\frac{\mathtt{freq}(\tau, d)}{|d|}$, measures the importance of a term within a document. It is normalized to account for document length. The second component, $\log \frac{N}{\mathrm{df}(\tau)}$, is a measure of the rarity of the term within the document collection C.

Example 13. Using the documents from Example 12 on page 14, the TF-IDF scores are as follows.

$$\begin{array}{ll} & \begin{array}{ccc} d_1 & d_2 & d_3 \end{array} \\ \begin{array}{l} \tau_1 : \text{“101”} \\ \tau_2 : \text{“360”} \\ \tau_3 : \text{“a”} \\ \tau_4 : \text{“boring”} \\ \tau_5 : \text{“cdps”} \\ \tau_6 : \text{“class”} \\ \tau_7 : \text{“great”} \\ \tau_8 : \text{“is”} \\ \tau_9 : \text{“lecture”} \\ \tau_{10} : \text{“math”} \\ \tau_{11} : \text{“mathematics”} \\ \tau_{12} : \text{“was”} \end{array} & \begin{pmatrix} 0.0000 & 0.2642 & 0.0000 \\ 0.3170 & 0.0000 & 0.0000 \\ 0.1170 & 0.0975 & 0.0000 \\ 0.0000 & 0.2642 & 0.0000 \\ 0.0000 & 0.2642 & 0.0000 \\ 0.3170 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.3962 \\ 0.1170 & 0.0975 & 0.0000 \\ 0.0000 & 0.0975 & 0.1462 \\ 0.6340 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.3962 \\ 0.0000 & 0.0000 & 0.3962 \end{pmatrix} \end{array}$$

Definition 11 (Document Vector)**.** Given a document collection C with M unique terms $\mathrm{T} = [\tau_1, \tau_2, \ldots, \tau_n]$, each document d can be represented by an M-dimensional vector.

$$\vec{d} = \begin{bmatrix} \text{tf-idf}(\tau_1, d) \\ \text{tf-idf}(\tau_2, d) \\ \vdots \\ \text{tf-idf}(\tau_n, d) \end{bmatrix}$$

Example 14. The documents in Example 12 on page 14 would produce the following vectors.

$$\vec{d}_n = \begin{bmatrix} \text{tf-idf}(\tau_1, d_n) \\ \text{tf-idf}(\tau_2, d_n) \\ \text{tf-idf}(\tau_3, d_n) \\ \text{tf-idf}(\tau_4, d_n) \\ \text{tf-idf}(\tau_5, d_n) \\ \text{tf-idf}(\tau_6, d_n) \\ \text{tf-idf}(\tau_7, d_n) \\ \text{tf-idf}(\tau_8, d_n) \\ \text{tf-idf}(\tau_9, d_n) \\ \text{tf-idf}(\tau_{10}, d_n) \\ \text{tf-idf}(\tau_{11}, d_n) \\ \text{tf-idf}(\tau_{12}, d_n) \end{bmatrix}, \vec{d}_1 = \begin{bmatrix} 0.0000 \\ 0.3170 \\ 0.1170 \\ 0.0000 \\ 0.0000 \\ 0.3170 \\ 0.0000 \\ 0.1170 \\ 0.0000 \\ 0.6340 \\ 0.0000 \\ 0.0000 \end{bmatrix}, \vec{d}_2 = \begin{bmatrix} 0.2642 \\ 0.0000 \\ 0.0975 \\ 0.2642 \\ 0.2642 \\ 0.0000 \\ 0.0000 \\ 0.0975 \\ 0.0975 \\ 0.0000 \\ 0.0000 \\ 0.0000 \end{bmatrix}, \vec{d}_3 = \begin{bmatrix} 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.3962 \\ 0.0000 \\ 0.1462 \\ 0.0000 \\ 0.3962 \\ 0.3962 \end{bmatrix}$$

Definition 12 (Search Query). A search query q is simply a document (as defined by Definition 8 on page 13). The top-k answers to q with respect to a collection C is defined as the k documents, $\{d_1, d_2, \ldots, d_k\}$ in C, such that $\{\vec{d}_1, \vec{d}_2, \ldots, \vec{d}_k\}$ are the closest vectors to $\vec{q}$ using a Euclidean distance measure in $\mathbb{R}^N$.

Example 15. Given the search query $q = \{\text{math, lecture, was, great}\}$, compute the vector $\vec{q}$ within the document collection C (as defined in Example 12 on page 14).

$$\vec{q} = \begin{bmatrix} 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.2500 \\ 0.1038 \\ 0.1038 \\ 0.2500 \\ 0.0000 \\ 0.0000 \end{bmatrix}$$

In order to determine the top-k documents for search query q, we need a way of measuring the similarity between documents.

Definition 13 (Cosine Similarity). Given two document vectors, $\vec{d}_1$ and $\vec{d}_2$, the cosine similarity is the dot product $\vec{d}_1 \cdot \vec{d}_2$, normalized by the product of the Euclidean distance of $\vec{d}_1$ and $\vec{d}_2$ in $\mathbb{R}^N$. It is denoted as similarity$(\vec{d}_1, \vec{d}_2)$.

$$\text{similarity}(\vec{d}_1, \vec{d}_2) = \frac{\vec{d}_1 \cdot \vec{d}_2}{\|\vec{d}_1\| \cdot \|\vec{d}_2\|} \tag{1.8}$$

$$= \frac{\sum_{i=1}^{N} \vec{d}_{1,i} \times \vec{d}_{2,i}}{\sqrt{\sum_{i=1}^{N} (\vec{d}_{1,i})^2} \times \sqrt{\sum_{i=1}^{N} (\vec{d}_{2,i})^2}} \tag{1.9}$$

Recall we may represent search queries as documents and thus document vectors. Therefore we may compute the score of a document d for a search query q as

$$\text{similarity}(\vec{d}, \vec{q})$$

Field	Value
code	MATH 360
subject	MATH
body	math 360 is a math class

Table 1.5: Course document for MATH 360.

Example 16. Given the document collection C (from Example 12 on page 14) and search query q, compute the similarity between q and every document $d \in C$.

$$\text{similarity}(\vec{d}_1, \vec{q}) = 0.390890 \tag{1.10}$$

$$\text{similarity}(\vec{d}_2, \vec{q}) = 0.061592 \tag{1.11}$$

$$\text{similarity}(\vec{d}_3, \vec{q}) = 0.252789 \tag{1.12}$$

1.3.2 Extending the Document Model

In the extended document model, documents have fields, denoted as $\textsc{Field}[d]$, and each field has a value. Thus

$$d : \textsc{Field}[d] \to \textsc{Bag}[\mathrm{T}]$$

Example 17 (Semi-Structured Document)**.** We see that d_1 is about MATH 360. The document contents are semi-structured, containing both a course code and the subject ID. By adding fields to the document, we are left with Table 1.5.

which is similar in structure to Table 1.4 on page 10.

1.3.3 Approximate String matching

Definition 14 (N-Gram)**.** An n-gram is a contiguous sequence of substrings of string S of length n. An algorithm for computing the n-gram of S is given in Algorithm 1.

Algorithm 1 N-GRAM(S, n, s)

Require: S is a string, $n \geq 1$, and s is a character
Ensure: the list of n-grams of S

$G \leftarrow []$
$p \leftarrow \textsc{Repeat}(s, n-1)$
$S \leftarrow \textsc{Pad}(S, p)$
$S \leftarrow \textsc{Replace}(S, `\ ', p)$
for $i = 0$ **to** $l - n + 1$ **do**
  append $S[i, i+n]$ to G
end for
return G

Where l is the length of S, $\textsc{Repeat}(S, n)$ repeats s character n times, $\textsc{Pad}(S, p)$ prefixes and postfixes S with p, and $\textsc{Replace}(S, s, p)$ replaces character s with p in string S.

Example 18. Given a string S = "human", compute the trigram of S using Algorithm 1.

$$G = \{\text{“\$\$h”}, \text{“\$hu”}, \text{“hum”}, \text{“uma”}, \text{“man”}, \text{“an\$”}, \text{“n\$\$”}\}$$

We use n-grams in order to permit approximate string matching.

Example 19. Given a string S (Example 18), let S' = "humans". Compute the trigram of S' and compare it to S.

$$G' = \{\text{“\$\$h”}, \text{“\$hu”}, \text{“hum”}, \text{“uma”}, \text{“man”}, \text{“ans”}, \text{“ns\$”}, \text{“s\$\$”}\}$$

Comparing G to G' results in the following matrix

As Fig. 1.5 on the following page shows, using n-grams yield a similarity of $\frac{5}{10}$.

1.3.4 Pros and Cons of the Document Model

There are numerous reasons to use the document model. The most significant reason is that it allows users without domain knowledge and working knowledge of a complex query language such as SQL to find information.

$$
\begin{array}{ll}
 & \begin{array}{cc} \boldsymbol{G} & \boldsymbol{G'} \end{array} \\
\begin{array}{l}
\tau_1 : \text{“ns\$”} \\
\tau_2 : \text{“n\$\$”} \\
\tau_3 : \text{“s\$\$”} \\
\tau_4 : \text{“ans”} \\
\tau_5 : \text{“man”} \\
\tau_6 : \text{“uma”} \\
\tau_7 : \text{“\$\$h”} \\
\tau_8 : \text{“hum”} \\
\tau_9 : \text{“\$hu”} \\
\tau_{10} : \text{“an\$”}
\end{array} &
\begin{pmatrix}
0 & 1 \\
1 & 0 \\
0 & 1 \\
0 & 1 \\
1 & 1 \\
1 & 1 \\
1 & 1 \\
1 & 1 \\
1 & 1 \\
1 & 0
\end{pmatrix}
\end{array}
$$

Figure 1.5: Comparison between n-grams of $\boldsymbol{G}$ and $\boldsymbol{G'}$.

Example 20 (Simple Queries). Find all documents related to “mathematics” or “lecture”. The result of the query q would be

$$\mathrm{query}(\text{“mathematics”}) \cup \mathrm{query}(\text{“lecture”}) \to \{d_2, d_3\}$$

Users can also modify queries to require certain terms be present or not present.

Example 21 (`AND` Query). Find all documents containing both “mathematics” and “lecture”. This query would return the following set of documents

$$\mathrm{query}(\text{“mathematics”}) \cap \mathrm{query}(\text{“lecture”}) \to \{d_3\}$$

as only d_1 contains both terms.

Example 22 (`NOT` Query). Find all documents containing “mathematics” but not “lecture”. This query would return different results than Example 21.

$$\mathrm{query}(\text{“mathematics”}) \neg\, \mathrm{query}(\text{“lecture”}) \to \varnothing$$

While none of the above queries required domain knowledge, it is possible to use the extended document model (Section 1.3.2 on page 18) to search specific fields. Doing so permits users to leverage their existing domain knowledge in order to achieve finer control over what documents are retrieved.

Example 23 (Extended Query)**.** Find all documents with a subject of "MATH" that contain the term "class".

$$\text{query}(\text{“subject”}, \text{“MATH”}) \cap \text{query}(\text{“class”}) \rightarrow \{d_1\}$$

Not only does the document model provide a familiar interface to search for information with, it also ranks the results. In the relational model a search for "mathematics" would return all named tuples that contained that term. In the document model, documents are ranked against the query q and the top-k documents are returned.

The advantage is that users have the result of q already ranked so only the most relevant documents may be explored. As the number of documents matching q for a large corpus can be high, showing only the top-k relevant documents may save the user a substantial amount of time.

The relational model does not permit approximate string matching. By utilizing the document model with n-grams (Section 1.3.3 on page 18), users who substitute, delete, or insert characters from the desired term may still receive results for their intended term (see Example 19 on page 19 for a demonstration of how n-grams overcome character insertion).

Unfortunately the document model does not support the concept of foreign keys (Definition 4 on page 5). While information is easily accessible due to flexible search, each document is a discrete unit of information. Aggregate queries are unsupported, as these units are not linked amongst one another.

1.4 Outline of Thesis

Chapter 3 describes the details of our implementation the data transformation and query operators for graph search in the linked document space. Our choice of utilizing a modern functional programming language for our implementation makes high degree of concurrency possible.

Chapter 4 provides further justification to our choice of data model mapping and the choice of programming language used. Through a series of experiments, we see that our proposal allows a tight integration of the relational database engines and keyword search libraries. Furthermore, the implementation enjoys a linear speed up with respect to the number of processors available.

Chapter 2

Best of Both Worlds

In this chapter we present a framework for the transformation between the relational and document models. The first component of this framework, detailed in Section 2.1, involves the encoding of named tuples into documents. In Section 2.3 we demonstrate the mapping of entity groups to documents. In Section 2.3 we show how to construct documents that encode the relations between tuples in the document model.

We further demonstrate how to perform iterative search using these document encodings.

2.1 Encoding Named Tuples into Documents

Recall in the extended document model (Section 1.3.2 on page 18), a document d consists of fields $f_1, f_2, \ldots, f_n$. Using the extended document model, we are left with a straight forward mapping of a tuple t to document d.

For tuple t, every attribute $\alpha \in \textsc{Attr}[t]$ maps to a field f in document d. Every attribute value must be analyzed into an indexable form in order to store it in a field.

$$\textsc{Attr}[t] \xrightarrow{analyzed} \textsc{Field}[d] \tag{2.1}$$

$$\alpha_1, \alpha_2, \ldots, \alpha_n \xrightarrow{analyzed} f_1, f_2, \ldots, f_n \tag{2.2}$$

We denote the document encoding of t as $\textsc{Doc}[t]$.

Field	Terms
code	{cdps, 101}
title	{human, mutant, relations}
subject	{cdps}

Table 2.1: $\textsc{Doc}[t]$

Example 24. Given the tuple

$$t = \{\text{code} : \text{“CDPS 101”}, \text{title} : \text{“Human-Mutant Relations”}, \text{subject} : \text{“CDPS”}\}$$

produce the document encoding $\textsc{Doc}[t]$.

2.2 Mapping of Entity Groups to Documents

Recall that an entity group (Definition 7 on page 9) is a forest T of tuples t such that for every $(t, t') \in T$, where $t \neq t'$, implies $\textsc{Rel}[t] \neq \textsc{Rel}[t']$. That is, every distinct tuple is from a distinct relation.

Given the restriction

$$\forall (r, r') \in G, \exists! \, (r, r') \vDash \theta$$

we assert that if t and t' are in the entity group T, then there is a foreign key constraint between t and t'. We denote the vertices of T as $\mathrm{V}(T)$, and the edges of T as $\mathrm{E}(T)$.

Claim 1. Given $\mathrm{V}(T)$, we are always able to reconstruct T.

Proof. Given $\mathrm{V}(T)$, we must reconstruct $\mathrm{E}(T)$ in order to complete T.

Choose any $(t, t') \in \mathrm{V}(T)$. If $(\textsc{Rel}[t], \textsc{Rel}[t']) \in GD$, then (t, t') is an edge in T.

Recall our earlier assertion that GD is cycle-free and foreign keys must be unique. □

$$\begin{tikzcd}
\text{Course(MATH 360, Complex Analysis, MATH)} \arrow[d] \\
\text{Subject(MATH, Mathematics)}
\end{tikzcd}$$

Figure 2.1: Example entity group

2.3 Encoding an Entity Group as a Document Group

Given a entity group T, we construct two or more documents in order to represent the entity group in the document model.

For every $t \in \mathrm{V}(T)$, we construct a document $\textsc{Doc}[t]$ (Section 2.1 on page 23). With each tuple t stored in the document collection C, we construct an additional document which stores the association information.

Let x be the indexing document of T.

$$x[\text{“entities”}] = \bigcup_{t \in \mathrm{V}(T)} \mathrm{UID}[t] \tag{2.3}$$

Thus, the encoding of T is defined as

$$T \xrightarrow{\text{encode}} \{\textsc{Doc}[t] : t \in \mathrm{V}(T)\} \cup \{x\} \tag{2.4}$$

Example 25. An entity group produced from the schema in Fig. 1.2 on page 8 is shown in Fig. 2.1.

Transforming the example entity group in Fig. 2.1 would produce documents shown in Table 2.2 on the following page

It's easy to see that from encode(T) we can recover $\mathrm{V}(T)$, the tuples in T.

By Claim 1 on page 24, this is sufficient to recover T entirely.

Field	Terms
__id__	{subject|math_360}
code	{math, 360}
title	{complex, analysis}
subject	{math}

(a) Course

Field	Terms
__id__	{subject|math}
id	{math}
name	{mathematics}

(b) Subject

Field	Terms
entities	{course|math_360, subject|math}

(c) Indexing document

Table 2.2: Document encoding of Fig. 2.1 on page 25

Field	Terms
__class__	courses|code
value	{biol, 1010u}
code	{biol, 1010u}
__all__	{$$b, bi, bio, iol, ol, l$$, $$1, $10, 101, 010, 10u, 0u$, u$$}

Table 2.3: Document representing a value of class "courses|code"

2.4 Encoding Attribute Values into Searchable Documents

Each value for user selected attributes are converted into *n*-grams, and stored in special documents. These value documents permit users to perform fuzzy search for attributes, allowing attributes to be located despite character substitutions. While this is not strictly necessary for the use of the system, it reduces user fatigue.

As shown in Table 2.3, a value document permits *n*-gram search over values while providing the original value. The system uses the value of found value documents to search for entities based on attributes.

These values are unique within the space of all attribute values of a specific class. They are not guaranteed to be unique within the entire space of documents. This allows the system to provide additional information to users. Rather than searching for a value, they are able to search for both a value and entity class.

2.5 Iterative Search Using Document Encodings

A document database supports fast and flexible keyword search queries. A search query is characterized by $q = (f, w)$, where f is an optional field name, and w is a search phrase.

$\text{query}(q)$ is the set of documents returned by the text index. The query function, combined with the extended document model, permits powerful search queries to be issued. Our implementation supports approximate string matching using n-grams (Section 1.3.3) for values, searching for entities containing keywords (see Example 26), and the discovery of intermediate entities given two known entities.

Example 26 (Entity Search)**.** Find all entities that match the keyword “math”.

Let $q =$ “math” be the search query. The results are

$$\begin{aligned} \text{query}(q) &= \text{query}(\text{“math”}) \\ &= \{\text{subject|math, course|math\_360}\} \end{aligned}$$

which are coincidentally related. The results of an entity search query are not necessarily related.

Example 27 (Entity Graph Search)**.** Find the shortest path between the two entities with the unique identities of “subject|math” and “instructor|5”.

The entity graph is shown in Fig. 2.2.

There are two possible paths between “subject|math” and “instructor|5”. The path in Fig. 2.3b is shorter than that in Fig. 2.3a, meaning it is likely the most relevant.

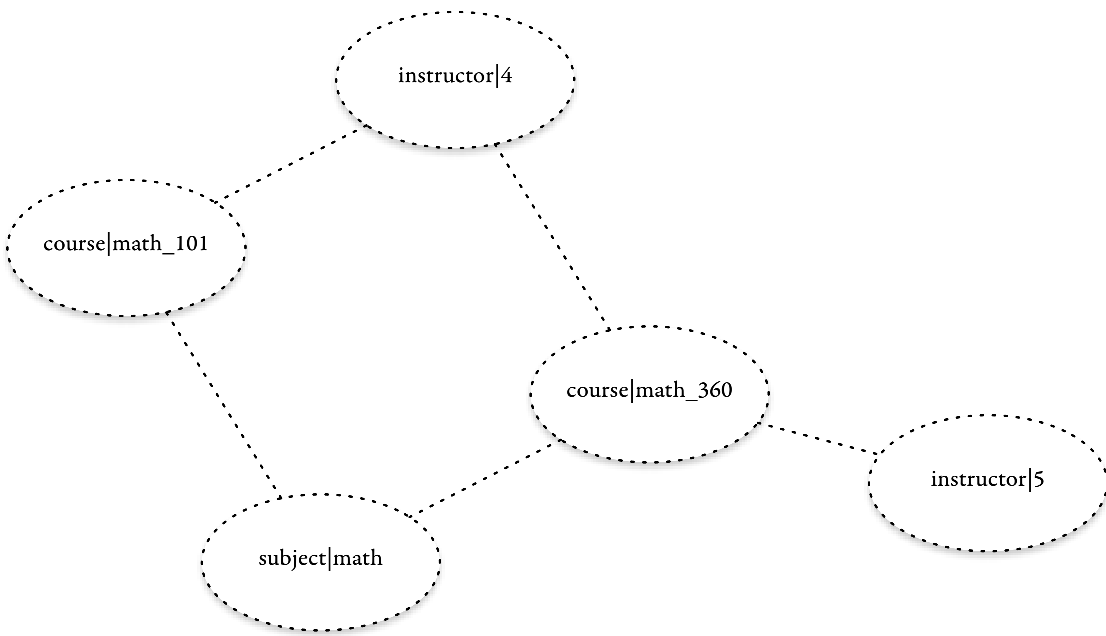
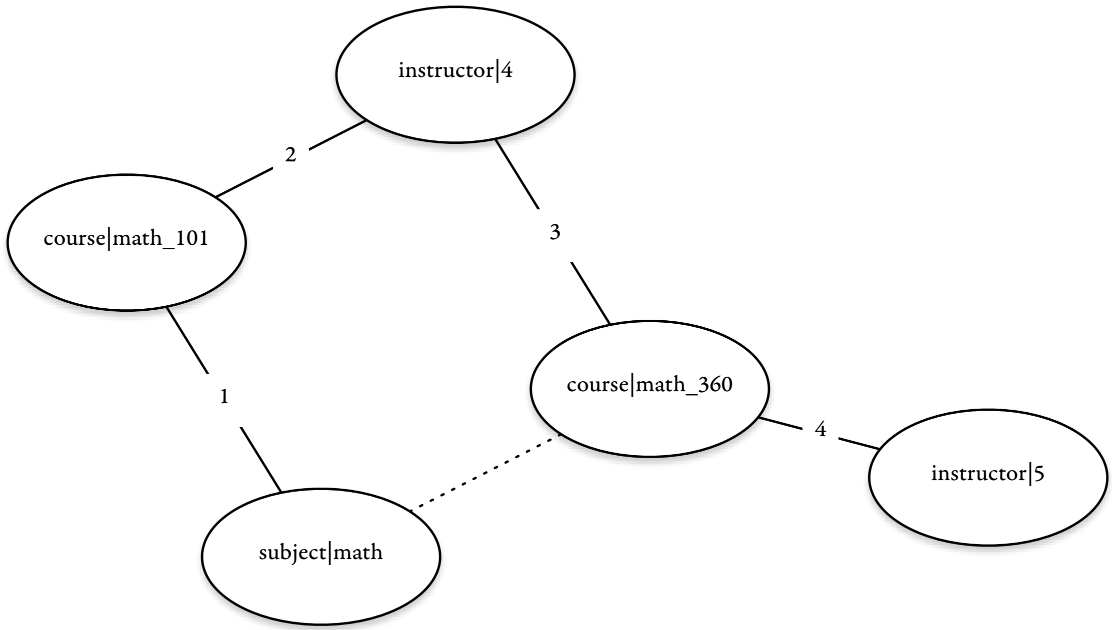
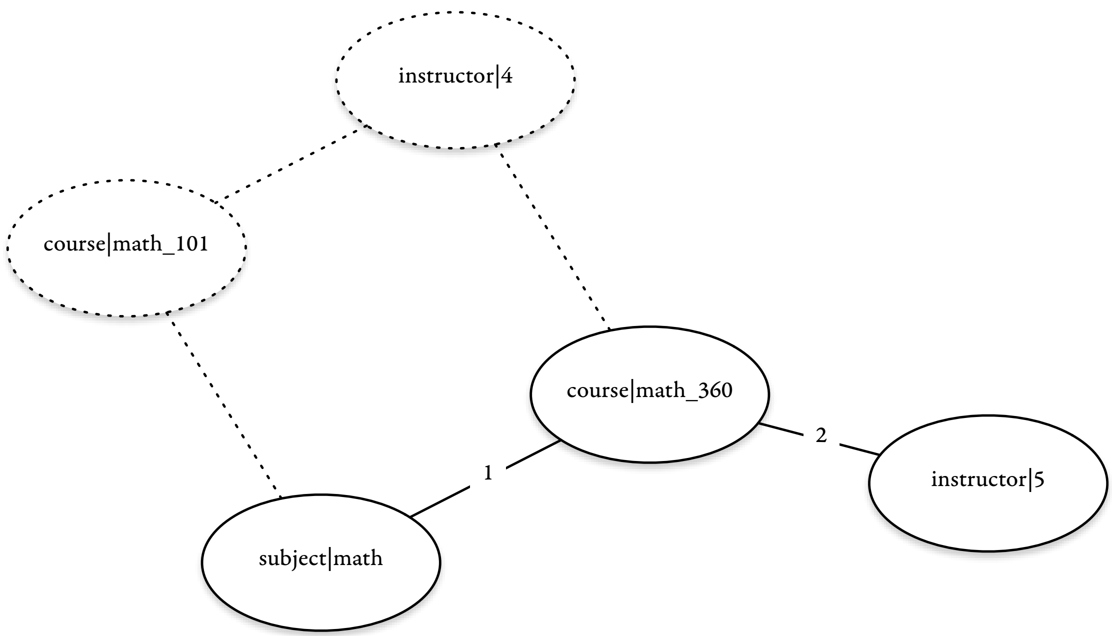


Figure 2.2: Initial graph



(a) Valid path between two entities



(b) Shortest valid path between two entities

Figure 2.3: Two possible paths between "subject|math" and "instructor|5"

Chapter 3

Along Came Clojure

In this chapter we further discuss our implementation. In Section 3.1 we discuss the basic tenets of functional programming with an emphasis on how Clojure implements these tenets. Further attention is paid to how Clojure uses these tenets to implement its concurrency model.

In Section 3.2 we illustrate how Clojure's JVM interoperability is used to interface with Lucene and Java database connectivity (JDBC) drivers in order to perform indexing and search on data. This section also covers keyword as well as graph search in document space.

3.1 Basic Principles of Functional Programming

The functional programming paradigm follows a handful of basic tenets; values are immutable, and functions must be free of side-effects [Hug89].

The first tenet, that values are immutable, refers to the fact that once a value is bound, this value may not change. In procedural programming there is the concept of assignment, whereas in functional programming, a value is bound. Assignment allows a value to change, binding does not.

Immutable values are advantageous as they remove a common source of bugs; state must explicitly be changed. This removes the ability for different areas of a program to modify the state (i.e. global variables).

Unfortunately immutable values can also lead to inefficiency. For example, in order to add a key-value pair to a map, an entirely new map must be created with the existing key-value pairs copied to it. In practice this is avoided through the use of persistent data structures with multi-versioning.

The second tenet, that functions must be free of side-effects, means that the output of a function must be predictable for any given input. This purity reduces a large source of bugs, and allows out-of-order execution. [Hug89].

3.1.1 Features of Clojure

The creator of Clojure, Rich Hickey, describes his language as follows:

> Clojure is a dialect of Lisp, and shares with Lisp the code-as-data philosophy and a powerful macro system. Clojure is predominantly a functional programming language, and features a rich set of immutable, persistent data structures. When mutable state is needed, Clojure offers a software transactional memory system and reactive Agent system that ensure clean, correct, multithreaded designs. ([Hica])

As the above quote describes, Clojure follows the basic tenets of functional programming.

Immutable, Persistent Data Structures

Clojure supports a rich set of data structures. These are immutable, satisfying the first tenet, as well as persistent, in order to overcome the inefficiency described previously.

The provided data structures range from scalars (numbers, strings, characters, keywords, symbols), to collections (lists, vectors, maps, array maps, sets) [Hicb]. These data structures are sufficient enough to allow us to use the universal design pattern [Yeg08].

Clojure also has the concept of persistent data structures. These are used in order to avoid the inefficiency of creating a new data structure and copying over the contents of the old data structure simply to make a change. Clojure creates a skeleton of the existing data structure, inserts the value into the data structure, then retains a pointer to the old data structure. If an old property is accessed on the new data structure, Clojure follows the pointers until the property is found on a previous data structure.

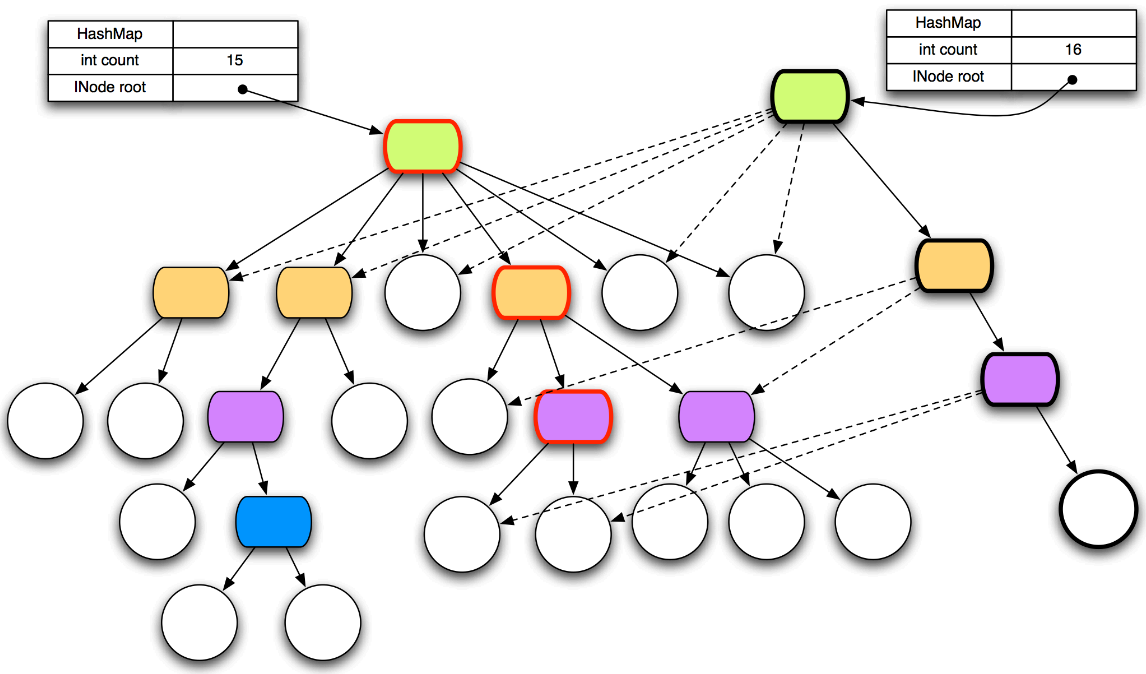


Figure 3.1: Representation of how data structures are "changed" in Clojure (Source: [Hic09])

In Fig. 3.1, we see what happens when a persistent data structure is "changed" in Clojure. The root of the left tree is the data structure before, and the root of the right tree is the data structure after. Note how the changed map retains pointers to all but the updated value; the newly created value is pointed to instead of the previous one.

Concurrency

Clojure supports four systems for concurrency: software transactional memory (STM), agents, atoms, and dynamic vars. The differences between these systems – including whether or not they are synchronous, coordinated, and what scope they encompass – are summarized in Table 3.1 on the following page.

In this system we use references and atoms. Changes made to an atom are, as the name suggests, atomic. While a change is occurring, any threads dereferencing the value of an atom will see its old version. Only upon completion of the change will the new value be visible. In Clojure this isolation is accomplished via persistent data structures (Fig. 3.1). For a demonstration using the canonical bank

System Name	Synchronous	Coordinated	Scope
STM	Yes	Yes	Application
Agents	No	No	Application
Atoms	Yes	No	Application
Dynamic Vars	Not Applicable	Not Applicable	Thread

Table 3.1: Comparison between Clojure's four systems for concurrency

Timestamp	Action	Code	Result
00000001	Create account with value of 1000	`(def account (atom 1000))`	
00000002	Begin deposit of 1000	`(alter account + 1000)`	
00000003	Read value of account	`(deref account)`	1000
00000004	Complete deposit of 1000	N/A	
00000005	Read value of account	`(deref account)`	2000

Table 3.2: Updating an atom

account transfer problem, see Table 3.2 [10].

STM provides more control than atoms. In addition to updates being atomic, they are also guaranteed to be consistent (constraints may be placed on a transaction), as well as isolated (due to persistent data structures).

STM supports two high-level methods of manipulating values within a transaction: `alter` and `commute`. Within a transaction, operations performed by `alter` must occur in order. That is, they are coordinated. One may also mark operations as commutable using `commute`.

In the context of the bank account transfer problem, one would use `alter` when order does matter, such as performing a deposit and then a withdrawal. If the money is withdrawn before the deposit, there may not be enough left in the account and the operation may fail when it should have succeeded. Contrast this to two deposits or two withdrawals where the order would not matter, and thus `commute`

Operation	Form	Example
Member Access	(.<member> <obj> [args])	(.toString 5)
	(. <obj> <member> [args])	(. 5 toString)
	(<class>/<member> [args])	(Integer/parseInt "5")
Object Instantiation	(<class>. [args])	(Integer. 5)
	(new <class> [args])	(new Integer 5)
Multiple Operations	(doto <obj> [forms])	(doto (Vector.) (.add 1))

Table 3.3: JVM interoperability

would be suitable.

In this context, synchronous refers to the fact that each operation waits for the proceeding operation to complete before continuing. When a system is coordinated, operations occur in a transaction. If one operation fails, the entire transaction fails and is rolled back. This results in additional overhead, but is considered safer as the system shall not be left in an inconsistent state as a result of a failed transaction.

Interoperability With the JVM

Traditionally, functional programming languages have been undesirable for numerous reasons: compatibility, libraries, portability, availability, packagability, and tools [Wad98]. Clojure attempts to avoid many of these reasons by running on the JVM. The JVM allows Clojure to both call and be called by Java and other languages. It includes syntactic sugar – features of a language added in order to simplify the language from a human perspective – to transparently call Java code, as well as make itself available to Java. This avoids the above issues.

The syntax of Clojure allows for the accessing of object members, the creation of objects, the calling of methods on an instance or class, etc. Clojure also includes shortcuts to perform multiple operations on the same object. The syntax is given in Table 3.3.

```
(defn ^Directory idx-path
  [path]
  (SimpleFSDirectory. (File. path)))
```

Figure 3.2: Clojure code that, given a path, returns a `Directory` object

We utilize Clojure's JVM interoperability to make use of Apache Lucene and JDBC.

3.2 Search With Clojure

Clojure's excellent JVM interoperability permits the use of countless third-party libraries. The most extensively used was Lucene.

3.2.1 Full-Text Search Using Lucene

"Apache Lucene™ is a high-performance, full-featured text search engine library written entirely in Java." [Fou] Lucene implements the Document Model (Section 1.3), providing a simple yet powerful API to perform full-text search. Among these features is the ability to vectorize documents according to the Vector Space Model (Section 1.3.1), utilize the extended document model to provide semi-structured documents (Section 1.3.2), and issue search queries against the index (Definition 12).

3.2.2 Indexing Relational Database

The indexing of relational objects is a complicated process. The objects must be retrieved from the relational database, transformed from named tuples into documents, then placed in the index. Additionally, all foreign keys (Definition 4) must be encoded as documents (Section 2.3) in order to satisfy Section 2.2.

The schema graph (Definition 6) must be defined before the relational database may be crawled.

Key	Description	Type(s)
`:T`	Entity (`:entity`) or entity group (`:entity`)	Symbol
`:C`	Table name for entities, brief description for entity groups	Symbol or String
`:sql`	SQL query used to construct the entity or entity schema	Expression
`:ID`	Attribute or attributes that comprise the key (Definition 3 on page 5)	Symbol or list of symbols
`:attrs`	List of attributes to analyze to fields	List of symbols
`:values`	List of attributes to index as values, must be subset of `:attrs`	List of symbols

Table 3.4: Keys expected by `EntitySchema` records

```
(doseq [ent-def schemas]
  (crawl ent-def db-conn idx-w))
```

Figure 3.3: Building the index

Schema Graph Definition

The schema graph is defined using a list. Each schema component, whether an entity or entity group, is defined by `EntitySchema` records. Each record accepts a map which specifies how each class of document should be indexed and identified. The keys of this map are given in Table 3.4.

Crawling the Relational Database

With the schema graph defined, the system is able to crawl the relational database, yielding a sequence of named tuples. It iterates through every `EntitySchema` record, instructing every record to crawl itself given a database connection and index writer (Fig. 3.3).

The `Database` protocol provides an `execute-query` method that permits access to the database. In the current implementation, the `Sqlite3` record implements the `Database` protocol. This record issues the query as-is, applying a given function to every result.

Each record issues a SQL query against the database that retrieves all named tuples that it represents. This query is given by the `:sql` key of the record. For every symbol defined by the `:values` key, an additional query is issued. These queries retrieve all distinct (within the context of that relation and attribute) values.

Transformation

For every named tuple, a document is constructed (see Section 2.1). In addition to the attributes, several other fields may be added to the document. These special fields contain additional meta information about the document. For example, the class, type, and unique identifier are added to an entity, while an entity group has a space-delimited list of unique identifiers that comprise the group.

Before becoming a document, named tuples are transformed into an internal representation. The internal representation adds flexibility to the system; so long as functions exist to convert between the internal representation and other forms, the system does not care about the source.

Clojure permits the annotation of data with metadata. Named tuples are returned as maps, with key-value pairs representing attributes and values for each tuple. Metadata may be associated with a named tuple that does not affect its key-value pairs.

The map of attributes and values of a named tuple is annotated with the function `(with-meta obj map)`. The `obj` parameter is the named tuple, while `map` is a map of metadata as defined by the system. The keys of `map` for each type (value, entity, or entity group) are defined in Table 3.5 on the next page.

The internal representation of the named tuple given in Table 1.4 on page 10 is given in Fig. 3.4 on the next page.

Once the internal representation is constructed, it may be transformed into a document. The mapping of a map to document is trivial; a field is created for every key in the map and the value corresponding to the key is the value of the field. Unfortunately Lucene does not facilitate the storage of metadata. Therefore the system must deal with metadata in a different way.

The system modifies each key of the metadata; two underscores are prepended and appended to the

<table>
<tr><th>Key</th><th>Value</th></tr>
<tr><td>:type</td><td>:value</td></tr>
<tr><td>:class</td><td><rel>|<attr></td></tr>
</table>

(a) Value

<table>
<tr><th>Key</th><th>Value</th></tr>
<tr><td>:type</td><td>:entity</td></tr>
<tr><td>:class</td><td><rel></td></tr>
<tr><td>:ID</td><td><rel>|<pk></td></tr>
</table>

(b) Entity

<table>
<tr><th>Key</th><th>Value</th></tr>
<tr><td>:type</td><td>:group</td></tr>
<tr><td>:entities</td><td><rel>|<pk></td></tr>
<tr><td></td><td>[<rel>|<pk> ...]</td></tr>
</table>

(c) Entity Group

Table 3.5: Metadata associated with each type

```
(with-meta
  {:code    ”CDPS 101”
   :title   ”Human-Mutant Relations”
   :subject ”CDPS”}
  {:type  :entity
   :class :course
   :id    ”course|cdps_101”})
```

Figure 3.4: Internal representation of named tuple from Table 1.4 on page 10

Field	Value
code	cdps 101
title	human mutant relations
subject	cdps
__type__	entity
__class__	course
__id__	course|cdps_101

Table 3.6: Document of internal representation from Fig. 3.4 on page 38

key name. This allows the system to differentiate between metadata and attributes. The transformation from internal representation to document is given in Table 3.6.

Indexing

With the named tuples transformed into documents, the index may be constructed. The first step is to open the index for writing. In Lucene, this is accomplished by creating an `IndexWriter` object on a `Directory` object that points to the index location. The `IndexWriter` object also expects an analyzer to be used by default on documents it indexes. The system uses a `WhitespaceAnalyzer` by default.

For every named tuple, the transformed document is written to the index by the index writer object. In addition, the indexing document (Section 2.3) of every entity group is added.

3.2.3 Keyword Search in Document Space

With the entity graph encoded into the document model, users may begin issuing search queries. Every query follows the following pattern: users look up values (optionally using fuzzy search), these values are used to locate entities, and once two entities are selected, the system attempts to locate connections between the two.

Fuzzy Value Search

Recall that entity values are stored as their *n*-gram (Section 1.3.3). This allows users to make character substitutions, deletions, or additions, and still return values they may have intended on finding. Without the use of *n*-grams, a misspelling on the user's part would result in either no or irrelevant values being returned. Values that are approximately matched would be assigned a lower score than those which are fully matched, but they would at least appear in the results.

Rather than guessing the user's intention, the system presents the user with a list of values in order to give them the option of substituting their entry for an approximate match. This autosuggest feature is intended to improve the user experience by eliminating a source of frustration – irrelevant results as a result of a simple spelling error.

Flexible Keyword Search API

The system provides a simple – yet flexible – keyword search API. Recall the extended query (Example 23) is in the form

$$\mathrm{query}(f, w)$$

The search API provides a function that accepts a symbol defining the field to search in, as well one or more words to search for. A phrase query comprised of every word is constructed.

$$\mathrm{query}(f, w_1, w_2, \ldots, w_n) = \bigcap_{w \in \{w_1, w_2, \ldots, w_n\}} \mathrm{query}(f, w)$$

Another function, `boolean-query`, accepts one or more `query` functions as well as a boolean operator for each and returns the result. The symbols `:and`, `:or`, and `:not` provide ∩, ∪, and ¬, respectively.

Example 28 (`boolean-query`)**.** For example, the query in Example 23 is given in Fig. 3.5 on the next page.

```
(boolean-query
 [[:and (query :subject ''MATH'')]
  [:and (query :text ''class'')]])
```

Figure 3.5: Boolean query in Clojure

Field	Terms
code	{math, 360}
title	{complex, analysis}
subject	{math}

(a) Fact representing a Course

Field	Terms
id	{math}
name	{mathematics}

(b) Fact representing a Subject

Table 3.7: Fragments, or facts, of information in a dataset

3.2.4 Graph Search in Document Space

With the ability for users to find relevant entities using fuzzy value search and the flexible keyword search API (Section 3.2.3), they must be able to find connections between two entities. As stated previously (Claim 1), the document encoding of relational data is a graph. This allows the system to search for links between entities by utilizing one or more graph search algorithms, such as BFS.

Why We Need Graph Search

Tuples are fragments, or facts, of larger pieces of knowledge. By utilizing graph search, we amalgamate these facts to provide a user with a broader view.

By utilizing graph search, we can take the facts in Table 3.7 and compose new facts. For example, we could learn who teaches Complex Analysis. It also allows us to ask questions, such as "who taught Complex Analysis with Instructor X?"

This automated discovery of relations between facts is why graph search is important.

Graph Search Algorithms

Recall we defined a graph as $G = (\mathrm{V}, \mathrm{E})$, where V is the set of all facts in the database, and $\mathrm{E}(T)$ is the set of entities in entity group T. We say that two vertices in T are connected if, and only if.

$$\mathrm{E}(T) \cap \mathrm{E}(T') \neq \varnothing$$

We must, given a keyword query q, find a subgraph, H of G, such that

- The vertices, or hops, in H are minimized; and
- The satisfaction of keywords in q by the vertices in H are maximized

In order to accomplish this, we find all vertices by entity group search and call this C. A graph search algorithm is used to minimally connect the vertices in C.

Algorithm 2 Graph-Search(C)

Require: C is a list of vertices
Ensure: minimal path between vertices in C

```
s ∈ C
for t ∈ C − {s} do
    find path connecting s → t such that Length(path) ≤ max-hops
end for
return path
```

Graph Search in Document Space

The question becomes: why do we perform this graph search in document space? Why not on the original relational space? There are two main answers to this question: speed and flexibility.

Rather than relying on scanning every tuple in every relation in a relational database, the document model represents every tuple as a semi-structured document. The contents of every field in these documents is indexed by an inverted list index data structure which permits fast lookup of documents based on keywords. We utilize this property to quickly locate the initial "source" vertices.

During the indexing process, various analyzers are applied against the text. These may, for instance, remove the suffix of words; a process called stemming [Por97].

Example 29 (Porter Stemmer). A course may have "mathematics" or even "mathematical" in the title. By utilizing a stemmer, we match both. The Porter stemming algorithm returns "mathemat" as the root word for both of the examples.

Other analyzers may remove stop words. Stop words may include "and", "or", "not", etc. These are functional words that may be removed from the text corpus without adversely affecting the meaning whose presence may otherwise affect the quality of search results [SR03].

Our implementation uses a document to represent edges in the entity graph. By using the document model, we are able to quickly locate all other vertices connected to a specific vertex.

Example 30 (Search for Connected Entities). Given the indexing documents x_1, x_2, x_3

$$x_1[\text{“entities”}] = \{\text{course}|\text{math\_360}, \text{subject}|\text{math}\}$$

$$x_2[\text{“entities”}] = \{\text{course}|\text{math\_101}, \text{subject}|\text{math}\}$$

$$x_3[\text{“entities”}] = \{\text{course}|\text{cdps\_101}, \text{subject}|\text{cdps}\}$$

A query, q for entities in the subject "math".

$$q = \text{query}(\text{“entities”}, \text{“subject}|\text{math”}) \cap \text{query}(\text{“\_\_type\_\_”}, \text{“entity”})$$

Would yield the results x_1, x_2.

Breadth-First Search

Cormen et al. define BFS as follows

> Given a graph $G = (V, E)$ and a distinguished source vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a "breadth-first tree" with root s that contains all reachable vertices. For

> any vertex v reachable from s, the path in the breadth-first tree from s to v corresponds to a "shortest path" from s to v in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs. ([CLRS09])

Essentially BFS populates the frontier before exploring the next hop. Thus BFS is able to explore a large, sparsely connected graph quickly. However, if the graph is dense, BFS would consume a substantial amount of memory, making an algorithm such as depth-first search (DFS) more suitable.

Functional BFS

The simple nature of BFS makes it ideal to implement in a functional manner. Rather than modifying objects, the search is conducted recursively. Newly discovered vertices are conjoined with the existing queue to form a new queue. Due to Clojure's persistent data structures, this operation is more efficient than intuition would dictate.

The functional implementation of BFS combines state changes into larger units. This leaves large segments of the implementation free of side effects. We exploit this fact to implement BFS concurrently in Clojure.

Concurrent BFS

In our implementation, the exploration of adjacent nodes is performed concurrently. Rather than exploring each node sequentially, as show in Fig. 3.6, Fig. 3.7, and finally Fig. 3.8, nodes "course|math_101" and "course|math_360" are explored simultaneously.

3.3 Web Interface

In order to make the system more accessible to novice users, a web-based interface was created.

The first step involves searching for entities by a value. We use approximate (n-gram) string matching to find relevant values despite potential character substitutions, deletions, or additions.

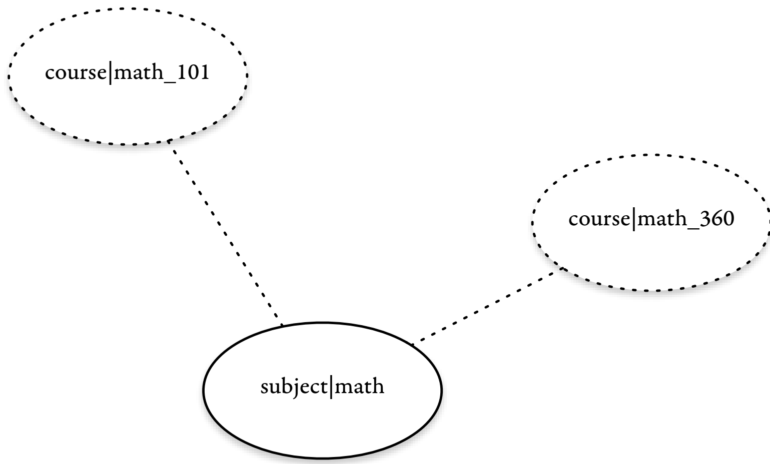


Figure 3.6: Initial graph

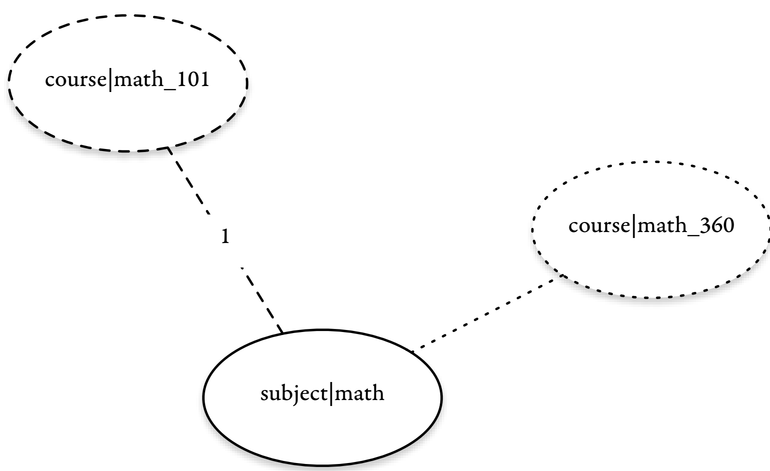


Figure 3.7: Exploring the first adjacent node sequentially

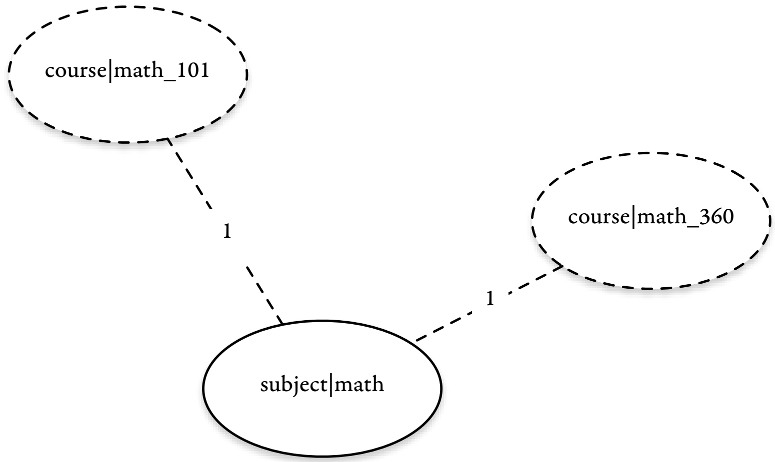


Figure 3.8: Both adjacent nodes explored

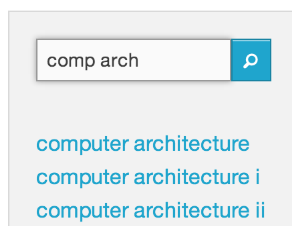


Figure 3.9: Approximate string matching of values

Entities that match the desired values are displayed to the user. They have the option of specifying the entity as either the source, or the target.

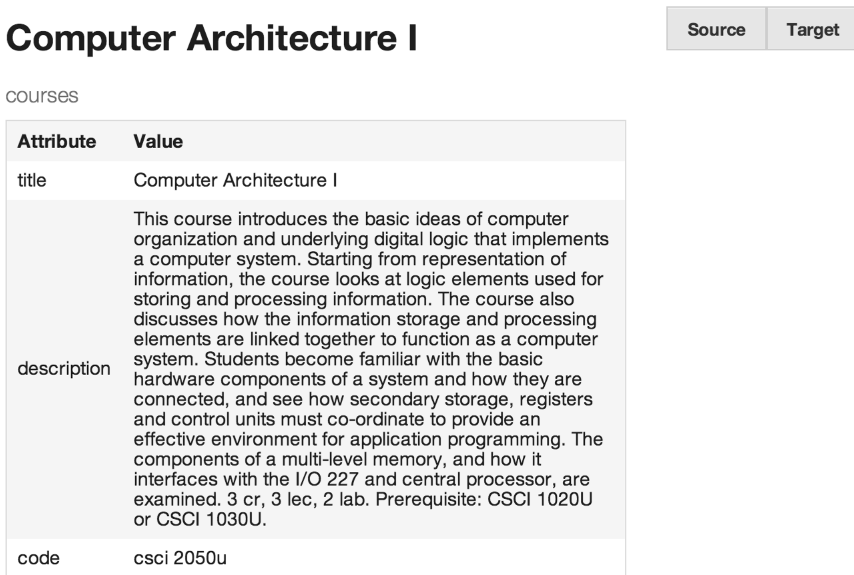


Figure 3.10: Tabular display of entities

When an entity is chosen, the navigation bar at the top of the page is updated to reflect the new selection. This allows the user to hover their cursor over the respective element in order to remind themselves of their selection.

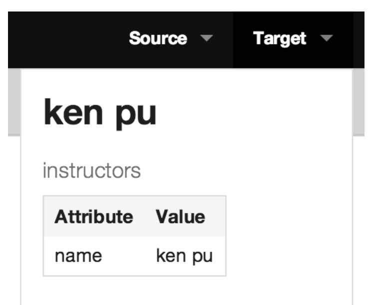


Figure 3.11: Chosen entities are displayed at the top

When both a source and target entity are selected, the user is able to search for the shortest path between them. They are given the option of which graph search algorithm implementation to use.

Figure 3.12: The algorithm implementation may be selected

A short message displaying the search duration as well as memory consumption is followed by a series of tables representing the intermediate entities between the source and target entities.

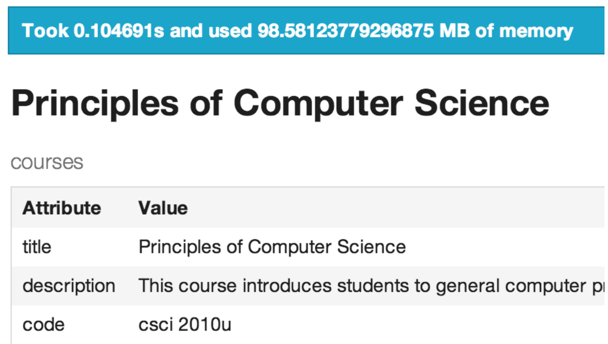


Figure 3.13: Result of a search between entities

This interface allows users to query the database for information in a familiar manner.

Chapter 4

Experimental Evaluation

In this chapter we evaluate our implementation of the system for transforming data in the relational model to the document model and vice versa described in Chapter 1. We cover the implementation details in Section 4.1, and the methodology and evaluation in Section 4.3.

4.1 Implementation

The system was implemented in Clojure, which "is a dynamic programming language that targets the JVM" [Hica]. Clojure was chosen due to its rich, immutable, and persistent data structures, excellent concurrency support, and seamless JVM interoperability. These features were discussed in detail in Section 3.1.1.

4.1.1 Code Base Statistics

The system consists of over 800 lines of Clojure, along with approximately 550 lines of Python. The Python code is used to construct the data set by crawling the course information site, as well as to aggregate the benchmark data produced by the system, producing graphs.

All development has occurred on GitHub [Dra14]. The use of Git and GitHub permits collaboration between researchers. With the code publicly available, future researchers may study and run it.

Class	Count
Course	1340
Instructor	849
Schedule	25755
Section	14463
Teaches	15358

Table 4.1: Number of objects in data corpus, grouped by class

4.2 The Data Corpus

The data corpus was derived from the University of Ontario Institute of Technology (UOIT) mycampus database. An HTML crawler was written in Python that scraped the information from the UOIT class schedule search page. This data was parsed, normalized, then placed in a SQLite database.

The data corpus consists of numerous classes of objects. These are: courses (Table B.1b), instructors (Table B.1a), schedules (Table B.1e), sections (Table B.1d), and teaches (Table B.1c). A graph representation of how these classes of objects are related can be found in Fig. 1.2. The data corpus is defined in Appendix B

The number of objects, as of the publication of this thesis, can be found in Table 4.1.

4.3 Runtime Evaluation

Scripts were written to coordinate the execution, collection, and transformation of the performance data of our implementation.

4.3.1 Methodology

We used Criterium[1] to handle the execution of the benchmarks as it handles unique concerns stemming from benchmarking on the JVM. These issues, identified by Boyer [Boy08], include:

- Statistical processing of multiple evaluations
- Inclusion of a warm-up period, designed to allow the JIT compiler to optimize its code
- Purging of the garbage collector before testing, to isolate timings from GC state prior to testing
- A final forced garbage collection after testing to estimate impact of cleanup on the timing results

This requires a much longer runtime as each function must be invoked numerous times.

During evaluation, Criterium collects performance metrics. Upon completion of the evaluation, it performs statistical analysis of these metrics using the bootstrap procedure developed by Efron [Efr87]. These metrics include mean, samples, variance, quartiles, outliers, and more.

Data Collection

The performance metrics computed by Criterium are returned as a Clojure map data structure. The evaluation process may take several hours to complete, necessitating a separation between data collection and post-processing. These metrics are stored offline for further processing.

In order to utilize the Clojure output in Python, a data interchange format (JSON) is used. The benchmark function writes the Criterium performance analysis out as a JSON string to stdout and the output is captured by the benchmark script. An example of this JSON output is given in Fig. 4.1.

4.3.2 Performance

Performance was measured for the various system components. An analysis of the metrics collected is presented in this section.

[1] `http://hugoduncan.org/criterium/`

```
[{
    "max-hops": ...,
    "method": ...,
    "results": {
        "execution-count": ...,
        "final-gc-time": ...,
        "lower-q": [...],
        "mean": [...],
    ...
}, ...]
```

Figure 4.1: Partial JSON output from Criterium.

Number of Groups	Elapsed Time (s)
0	11.800
1	12.446
2	19.771

Table 4.2: Indexing time growth by number of entity groups, averaged over 5 runs

Indexing

The indexing process is computationally intensive but short lived. After the initial JVM warmup period, the time required to construct the index scales with the number of named tuples and relations between them.

We see in Table 4.2 the indexing time increases minimally between 0 and 1 group. The number of entity groups added by the first entity schema is relatively small. Contrast this to the indexing time between 1 and 2 groups, which increases considerably. The number of entity groups also grew considerably, explaining the time increase.

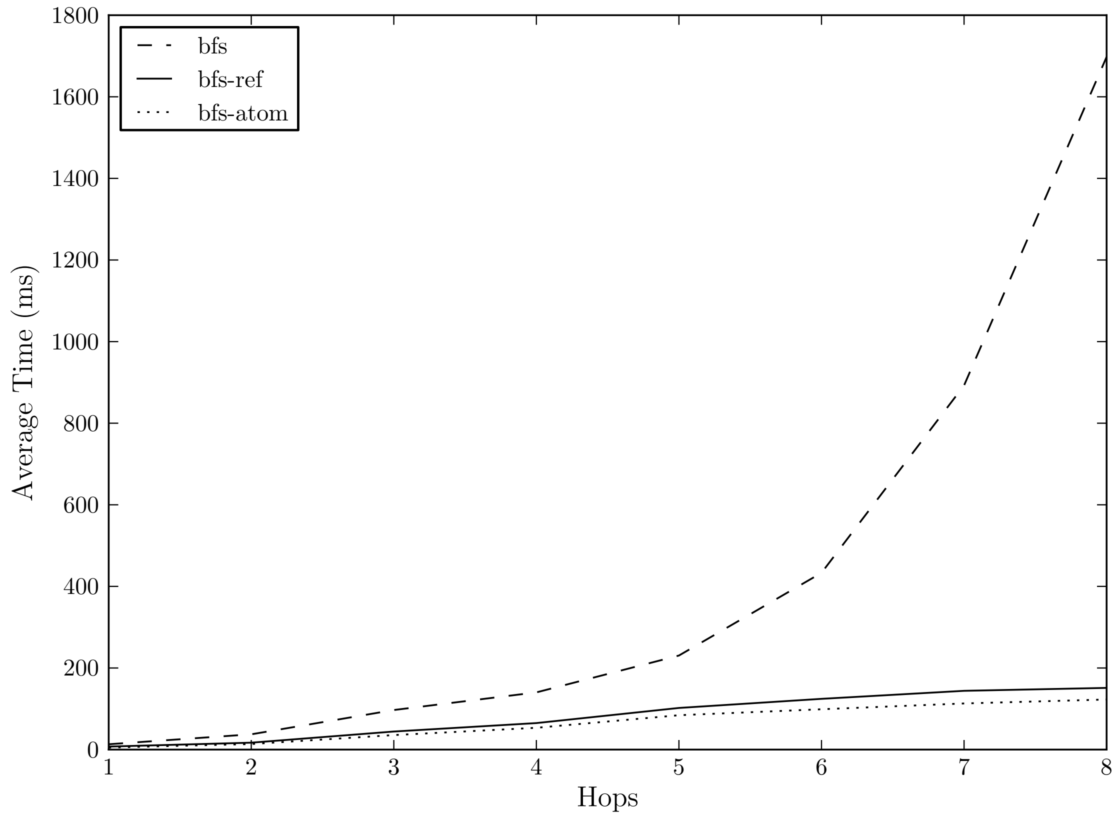


Figure 4.2: Growth of each graph search algorithm implementation by number of hops

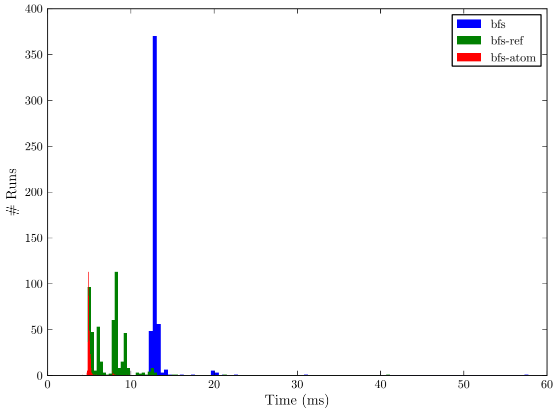
Graph Search

The worst-case performance of BFS is $\mathcal{O}(n^2)$. This is reflected in Fig. 4.2 which follows an exponential growth curve. In an attempt to mitigate the rapid increase in search time, concurrent variants of BFS were also implemented and benchmarked.

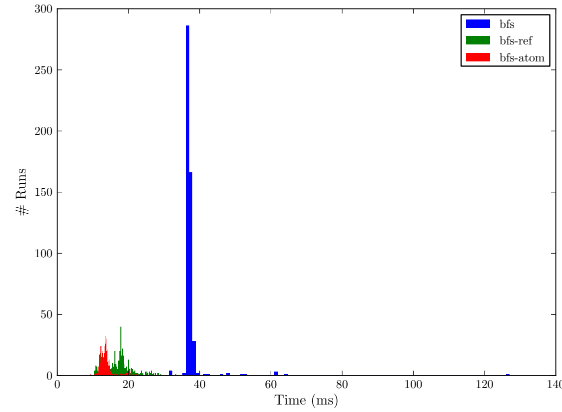
We see in Fig. 4.2 the rate of growth of BFS is as expected. The rate of growth of BFS with references and BFS with atoms is nearly linear. The atom implementation is slightly more performant as it lacks some of the overhead associated with references.

All captions must be at least 10pt

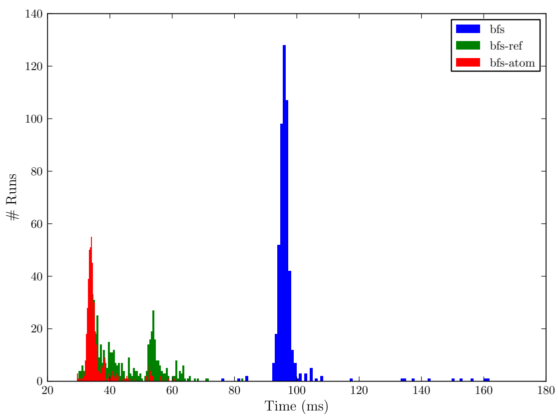
The difference in rate of growth is further illustrated in Fig. 4.3. As seen previously, Fig. 4.3a shows little difference in runtime between the three methods. The difference becomes clearer in Fig. 4.3b, and by Fig. 4.3h, the difference is obvious.



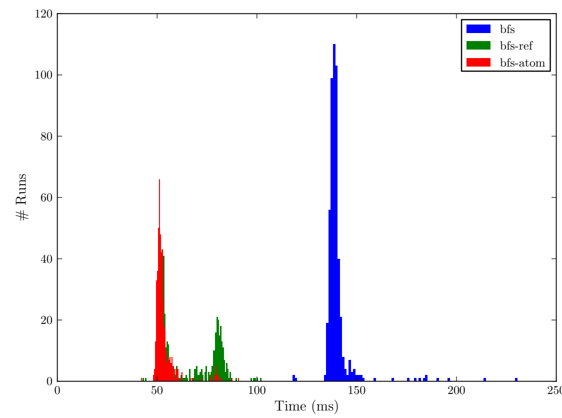
(a) 1 Hop



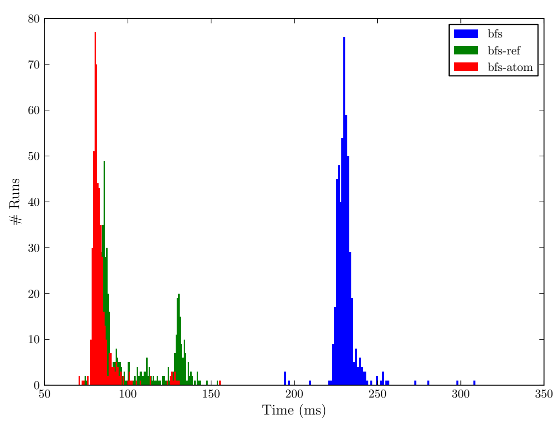
(b) 2 Hops



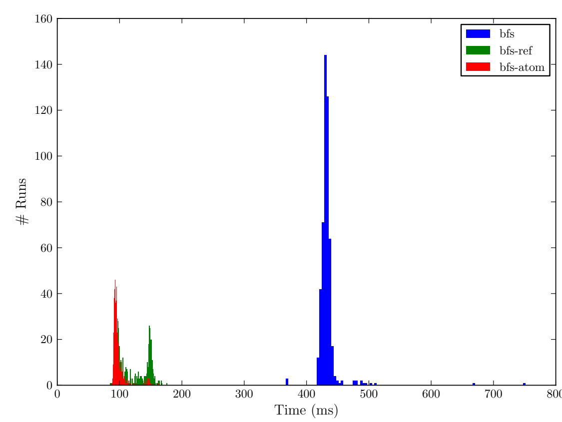
(c) 3 Hops



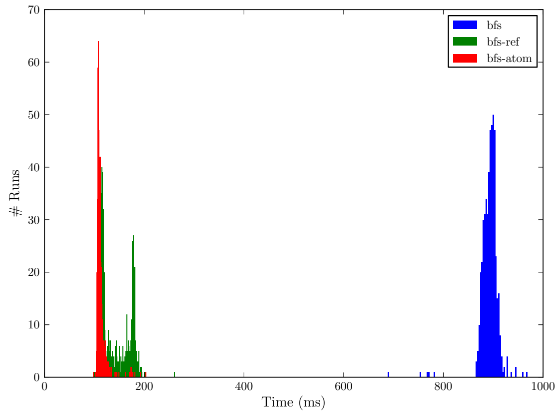
(d) 4 Hops



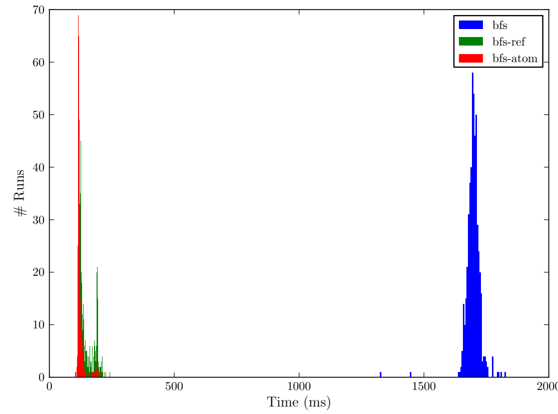
(e) 5 Hops



(f) 6 Hops



(g) 7 Hops



(h) 8 Hops

Figure 4.3: Distribution of samples per method, broken down by hops

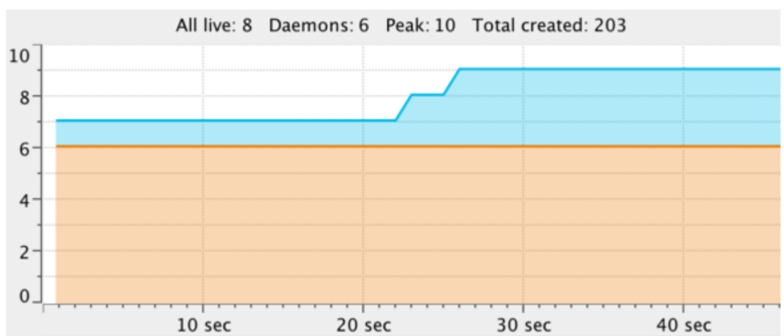


Figure 4.4: Thread count while running BFS atom benchmark

By using a profiling tool[2], we see the behaviour of Clojure's concurrency implementation.

In Fig. 4.4 we see that a number of threads are created and destroyed. Recall a new thread is created every time the frontier is populated.

4.4 Conclusion

In evaluating the system, we came to several conclusions.

Benchmarking any code is difficult. The process may not have exclusive control over the processor, memory is paged in and out, disk I/O is cached, etc. The JVM complicates matters with just-in-time

[2] `http://yourkit.com/`

(JIT) compilation and garbage collection.

The growth of BFS can be mitigated by the use of concurrency. Clojure facilitated a natural transition from a classical implementation of BFS to a highly concurrent one.

Chapter 5

Conclusion

5.1 Summary

While the relational model is a powerful and well understood method of storing data, it is not without its shortcomings. The rigidity of the relational model comes at the cost of usability. A change to the data model may require a rewrite of queries to account for the different join paths, increasing the cognitive burden on users.

In contrast, the document model represents semi-structured and unstructured data. The queries issued against the document model are unstructured and flexible. This allows users with little or no prior domain knowledge to issue queries. Unfortunately this flexibility comes at the cost of foreign keys, data consistency, and aggregate queries.

In Chapter 1 we introduced the relational and document data models. We compared and contrasted the two data models, paving the way for our contribution. We introduced our contribution in Chapter 2, providing a formal definition of a framework for the translation between the relational and document data models. Our implementation was introduced in Chapter 3, which also covered Clojure. In Chapter 4, we evaluated our implementation, identifying performance characteristics.

5.2 Lessons Learned

The system evaluation in Chapter 4 yielded several important insights.

- We have found that the simplest algorithms are the easiest to parallelize. The reduced complexity, and thus state, reduces the amount of shared data that must be synchronized. This allows for higher concurrency.
- Clojure's STM implementation is simple to use and effective. A few simple functions provide a powerful concurrency model.
- Sometimes a simpler approach to concurrency is the most appropriate one. In our evaluation, atoms provided better performance than references. Atoms allow for finer granularity in concurrency, reducing the overhead associated with references. This is desirable in situations that do not require much shared state.
- Clojure is a powerful language that encouraged us to write correct code first, then optimize it later. The transition to a concurrent implementation of BFS was trivial. The switch from atoms to references was also trivial.

Appendix A

Source Code

Each namespace in the code is divided into sections in the thesis document.

A.1 molly

A.1.1 molly.core

The core namespace is responsible for determining, provided a series of command line arguments, which action to take. This namespace is invoked as the main class when the Java archive (JAR) is executed.

```
(ns molly.core
  (:require [clojure.tools.cli :refer [cli]]
            [molly.algo.bfs :refer [bfs]]
            [molly.algo.bfs-atom :refer [bfs-atom]]
            [molly.algo.bfs-ref :refer [bfs-ref]]
            [molly.bench.benchmark :refer [benchmark-search]]
            [molly.conf.config :refer [load-props]]
            [molly.index.build :refer [build]]
            [molly.search.lucene :refer [idx-path idx-searcher]])
  (:gen-class))

(defn parse-args
  [args]
  (cli args
       [”-c” ”--config” ”Path to configuration (properties) file”]
       [”--algorithm”   ”Algorithm to run”]
       [”-s” ”--source” ”Source node”]
       [”-t” ”--target” ”Target node”]
       [”--max-hops”    ”Maximum number of hops before stopping”
        :parse-fn #(Integer. %)]
       [”--index”       ”Build an index of the database”
```

```
        :default false
        :flag true]
       ["--benchmark"   "Run benchmarks"
        :default false
        :flag true]
       ["-d" "--debug"  "Displays additional information."
        :default false
        :flag true]
       ["-h" "--help"   "Show help"
        :default false
        :flag true]))

(defn -main
  [& args]
  (let [[opts arguments banner] (parse-args (flatten args))]
    (when (or (opts :help) (not (opts :config)))
      (println banner)
      (System/exit 0))

    (let [properties  (load-props (opts :config))
          max-hops    (if (opts :max-hops)
                        (opts :max-hops)
                        (properties :idx.search.max-hops))]
      (when (opts :index)
        (let [database  (properties :db.path)
              index     (properties :idx.path)]
          (build database index)))
      (when (opts :algorithm)
        (let [searcher  (idx-searcher
                          (idx-path
                            (properties :idx.path)))
              source    (opts :source)
              target    (opts :target)
              f         (condp = (opts :algorithm)
                          "bfs"             bfs
                          "bfs-atom"        bfs-atom
                          "bfs-ref"         bfs-ref
                          (throw
                            (Exception.
                              "Not a valid algorithm choice.")))]
          (if (opts :debug)
            (let [[visited dist prev] (f searcher source target)]
              (println visited)
              (println dist)
              (println prev)
```

```

              (loop [node target]
                (println node)
                (let [prev-node (prev node)]
                  (when-not (nil? prev-node)
                    (recur prev-node))))
              )
            (benchmark-search f searcher source target))
          (shutdown-agents))))))
```

A.2 molly.conf

A.2.1 molly.conf.config

This namespace contains helper functions for loading part of the system configuration. It also provides a protocol, `IConfig`, that is used to define the rest of the system configuration.

```
(ns molly.conf.config
  (:require [propertea.core :refer [read-properties]]))

(defn load-props
  ([]
   (load-props "config/molly.properties"))
  ([file-name]
   (read-properties file-name
                    :parse-int  [:idx.topk.value
                                 :idx.topk.entities
                                 :idx.topk.entity
                                 :idx.search.max-hops]
                    :required   [:db.path
                                 :idx.path
                                 :idx.topk.value
                                 :idx.topk.entities
                                 :idx.topk.entity
                                 :idx.search.max-hops])))

(defprotocol IConfig
  (connection [this])
  (schema [this])
  (index [this]))
```

A.2.2 molly.conf.mycampus

This is a sample configuration. It defines the entities and relations in the mycampus dataset.

```
(ns molly.conf.mycampus
  (:use [clojureql.core :only (table project join where rename)])
  (:import (molly.conf.config IConfig)
           (molly.datatypes.database Sqlite)
           (molly.datatypes.schema EntitySchema)))

(def schedules-table
  (->
    (join
      (->
        (table :sections)
        (project [[:id :as :sec_id] [:id :as :section_id] :actual
                  :campus :capacity :credits :levels
                  :registration_start :registration_end :semester
                  :sec_code :sec_number :year :course]))
      (->
        (join
          (->
            (table :schedules)
            (project [[:id :as :sch_id] :date_start :date_end :day
                      :schedtype :hour_start :hour_end :min_start
                      :min_end :classtype :location]))
          (->
            (table :teaches)
            (project [:position :teaches.schedule_id
                      :teaches.instructor_id [:id :as :teaches_id]]))
          (where (= :schedules.id :teaches.schedule_id))))
      (where (= :sections.id :section_id)))))

(def schedules-id
  [[:schedules  :schedule_id]
   [:teaches    :teaches_id]
   [:sections   :section_id]])

(def mycampus-schema
  [(EntitySchema.
     {:T      :entity
      :C      :courses
      :sql    (table :courses)
      :ID     :code
      :attrs  [:code :title :description]
```

```
      :values [:code :title]})
   (EntitySchema.
     {:T      :entity
      :C      :instructors
      :sql    (table :instructors)
      :ID     :id
      :attrs  [:name]
      :values [:name]})
   (EntitySchema.
     {:T      :entity
      :C      :schedules
      :sql    (table :v_schedules)
      :ID     :id
      :attrs  [:position :actual :campus :capacity :credits :levels
               :registration_start :registration_end :semester
               :sec_code :sec_number :year :course :date_start
               :date_end :day :schedtype :hour_start :hour_end
               :min_start :min_end :classtype :location]
      :values [:campus :location]})
   (EntitySchema.
     {:T      :group
      :C      "Instructor schedule"
      :sql    (->
                (join
                  (table :v_schedules)
                  (->
                    (table :instructors)
                    (project [:id]))
                  (where (= :instructor_id :instructors.id))))
      :ID     [[:instructors  :instructors.id "Instructor ID"]
               [:schedules    :id "Schedule ID"]]
      :attrs  []
      :values []})
   (EntitySchema.
     {:T      :group
      :C      "Course schedule"
      :sql    (->
                (join
                  (table :v_schedules)
                  (->
                    (table :courses)
                    (project [:code]))
                  (where (= :course :code))))
      :ID     [[:courses    :courses.code   "Course code"]
               [:schedules  :id     "Schedule ID"]]
```

```
      :attrs  []
      :values []})
   ])

(deftype Mycampus [db-path idx-path]
  IConfig
  (connection
    [this]
    (Sqlite. {:classname "org.sqlite.JDBC"
              :subprotocol "sqlite"
              :subname db-path}))
  (schema
    [this]
    mycampus-schema)
  (index
    [this]
    idx-path))
```

A.3 molly.datatypes

There are several datatypes used in the system.

A.3.1 molly.datatypes.database

A protocol and concrete datatype are defined which provide access to a relational database. Users creating an instance of this datatype are able to execute arbitrary queries, and must provide a function to apply to every tuple that is returned.

```
(ns molly.datatypes.database
  (:require
    [clojure.java.jdbc :as sql]
    [clojureql.core :as cql]))

(defprotocol Database
  (execute-query [this query f]))

(deftype Sqlite [conn]
  Database
  (execute-query
    [this query f]
    (sql/with-connection conn (cql/with-results [rs query]
                                                (doseq [res rs]
                                                  (f res))))))
```

A.3.2 molly.datatypes.entity

One of the most important namespaces represents entities. It includes functions to transform a named tuple from a database row into the internal representation as well as into documents. It also includes auxiliary functions to produce a unique identifier.

```clojure
(ns molly.datatypes.entity
  (:require [molly.util.nlp :refer [q-gram]])
  (:import (org.apache.lucene.document Document Field Field$Index
                                       Field$Store)))

(defn special?
  [field-name]
  (and (.startsWith field-name "__") (.endsWith field-name "__")))

(defn uid
  "Possible inputs include:
    row :T :ID
    row [[:T :ID] [:T :ID]]
    row [[:T :ID :desc] [:T :ID :desc]]"
  ([row C id]
     (if (nil? (row id))
       (throw
         (Exception.
           (str "ID column " id " does not exist in row " row ".")))
       (str (name C)
            "|"
            (clojure.string/replace (row id) #"\s+" "_"))))
  ([row Tids]
   (clojure.string/join " " (for [[C id] Tids]
                              (uid row C id)))))

(defn field
  [field-name field-value]
  (Field. field-name
          field-value
          Field$Store/YES
          Field$Index/ANALYZED))

(defn document
  [fields]
  (let [doc (Document.)]
    (doseq [[field-name field-value] fields]
      (.add doc (field (name field-name) (str field-value))))
    doc))
```

```
(defn row->data
  ^{:doc ”Transforms a row into the internal representation.”}
  [this schema]
  (let [T          (schema :T)
        C          (schema :C)
        attr-cols  (schema :attrs)
        attrs      (if (nil? attr-cols)
                     this
                     (select-keys this attr-cols))
        meta-data  {:type T :class C}
        id-col     (schema :ID)]
    (with-meta (if (= T :group)
                 (conj attrs {:entities (uid this id-col)})
                 attrs)
               (condp = T
                 :value   (assoc meta-data
                                 :class
                                 (clojure.string/join ”|”
                                    (map name
                                         [C (first attr-cols)])))
                 :entity  (assoc meta-data :id
                                 (if (coll? id-col)
                                   (uid this id-col)
                                   (uid this C id-col)))
                 :group   (assoc meta-data
                                 :entities
                                 (uid this id-col))
                 (throw
                   (IllegalArgumentException.
                     ”I only know how to deal with types :value,
                     :entity, and :group”))))))

(defn doc->data
  ^{:doc ”Transforms a Document into the internal representation.”}
  [this]
  (let [fields        (.getFields this)
        extract       (fn [x] [(keyword (clojure.string/replace
                                          (.name x) ”_” ””))
                               (.stringValue x)])
        check-special (fn [x] (special? (.name x)))
        filter-fn     (fn [f] (apply hash-map
                                     (flatten
                                       (map extract
                                            (filter f fields)))))]
```

```
    (with-meta (filter-fn (fn [x] (not (check-special x))))
               (filter-fn check-special))))

(defn data->doc
  ^{:doc ”Transforms the internal representation into a Document.”}
  [this]
  (let [int-meta  (meta this)
        T         (int-meta :type)
        all       (clojure.string/lower-case
                    (clojure.string/join ” ”
                                         (if (= T :entity)
                                           (conj (vals this)
                                                 (name
                                                   (int-meta :class)))
                                           (vals this))))
        luc-meta  [[:__type__  (name T)]
                   [:__class__ (name (int-meta :class))]
                   [:__all__   (if (= T :value)
                                   (q-gram all)
                                   all)]]
        raw-doc   (concat luc-meta
                          this
                          (condp = (int-meta :type)
                            :value   [[:value all]]
                            :entity  [[:__id__ (int-meta :id)]]
                            :group   []))]
    (document raw-doc)))
```

A.3.3 molly.datatypes.schema

The final datatype represents a schema. These schemas contain a function that is used to execute the necessary SQL statements to retrieve all data from the relational database and place it in the full-text search index.

Several of these schema datatypes are joined together in a configuration in order to produce a schema graph.

```
(ns molly.datatypes.schema
  (:require [molly.datatypes.database :refer [execute-query]]
            [molly.datatypes.entity :refer [data->doc row->data]]
            [molly.search.lucene :refer [add-doc]]
            [clojureql.core :as cql]))

(defprotocol Schema
  (crawl [this db-conn idx-w])
  (klass [this])
  (schema-map [this]))

(deftype EntitySchema [S]
  Schema
  (crawl
    [this db-conn idx-w]
    (let [sql (S :sql)]
      (execute-query db-conn sql
                     (fn [row]
                       (add-doc idx-w
                                (data->doc (row->data row S)))))

      (if (= (S :T) :entity)
        (doseq [value (S :values)]
          (let [query (->
                        sql
                        (cql/project [value])
                        (cql/grouped [value]))]
            (execute-query db-conn query
                           (fn [row]
                             (add-doc idx-w (data->doc
                               (row->data row
                                          (assoc S
                                                 :T :value))))))))))))
  (klass
    [this]
    ((schema-map this) :C))
  (schema-map
```

```
    [this]
    S))
```

A.4 molly.index

A.4.1 molly.index.build

This namespace contains the function used to build the full-text search database. It takes advantage of the fact that each schema knows how to construct its own documents. The function simply iterates through every schema in the configuration, instructing them to index themselves.

```
(ns molly.index.build
  (:require [molly.datatypes.schema :refer [crawl klass]]
            [molly.search.lucene :refer [close-idx-writer
                                         idx-path
                                         idx-writer]]
            [molly.conf.mycampus])
  (:import [molly.conf.mycampus Mycampus]))

(defn build
  [db-path path]
  (let [conf    (Mycampus. db-path path)
        db-conn (.connection conf)
        ft-path (idx-path (.index conf))
        idx-w   (idx-writer ft-path)
        schemas (.schema conf)]
    (doseq [ent-def schemas]
      (println "Indexing" (name (klass ent-def)) "...")
      (crawl ent-def db-conn idx-w))
    (close-idx-writer idx-w)))
```

A.5 molly.util

A.5.1 molly.util.nlp

The `q-gram` function computes the *q*-gram of a string. Optionally a value for *q* and the padding character can be specified.

```
(ns molly.util.nlp)

(defn q-gram
  ^{:doc ”Given a string S, an integer n (optional), and a character
          s (optional), returns the n-gram of S using s as the
          padding character.”}
  ([S]
   (q-gram S 3 ”$”))
  ([S n]
   (q-gram S n ”$”))
  ([S n s]
   (let [padding  (clojure.string/join ”” (repeat (dec n) s))
         padded-S (str padding
                       (clojure.string/replace S ” ” padding)
                       padding)]
     (clojure.string/join ” ”
                          (for [i (range
                                    (inc (- (count padded-S) n)))]
                            (.substring padded-S i (+ i n)))))))
```

A.6 molly.search

A.6.1 molly.search.lucene

This namespace contains various functions for interfacing with the Lucene library. These functions include opening, adding documents, searching, and closing indices.

```
(ns molly.search.lucene
  (:import (java.io File)
           (org.apache.lucene.analysis.core WhitespaceAnalyzer)
           (org.apache.lucene.index IndexReader IndexWriter
                                    IndexWriterConfig)
           (org.apache.lucene.search IndexSearcher)
           (org.apache.lucene.store Directory SimpleFSDirectory)
           (org.apache.lucene.util Version)))

(def version
     Version/LUCENE_44)
(def default-analyzer
  (WhitespaceAnalyzer. version))

(defn ^Directory idx-path
  [path]
  (-> path File. SimpleFSDirectory.))

(defn idx-searcher
  [^IndexSearcher idx-path]
  (IndexSearcher. (IndexReader/open idx-path)))

(defn ^IndexWriter idx-writer
  ([^Directory idx-path analyzer]
    (IndexWriter. idx-path (IndexWriterConfig. version analyzer)))
  ([^Directory idx-path]
    (idx-writer idx-path default-analyzer)))

(defn close-idx-writer
  [^IndexWriter idx-writer]
  (doto idx-writer
    (.commit)
    (.close)))

(defn idx-search
  [idx-searcher query topk]
  (let [results (.scoreDocs (.search idx-searcher query topk))]
    (map (fn [result] (.doc idx-searcher (.doc result))) results)))
```

```clojure

(defn add-doc
  [idx doc]
  (.addDocument idx doc))
```

A.6.2 molly.search.query_builder

Phrase queries are used as they require each term in the phrase to be in a specific order. This permits more accurate results as course titles and other items are in a specific order.

These queries may be combined, creating a boolean query.

```
(ns molly.search.query-builder
  (:import (org.apache.lucene.index Term)
           (org.apache.lucene.search BooleanClause$Occur BooleanQuery
                                     PhraseQuery)))

(defn query
  [kind & args]
  (let [field-name    (condp      = kind
                        :type     "__type__"
                        :class    "__class__"
                        :id       "__id__"
                        :text     "__all__"
                        ; Assume "kind" is an attribute name.
                        (condp = (type kind)
                          clojure.lang.Keyword  (name kind)
                          java.lang.String      kind))
        phrase-query  (PhraseQuery.)]
    (doseq [arg args]
      (.add phrase-query (Term. field-name (name arg))))

    phrase-query))

(defn boolean-query
  [args]
  (let [query (BooleanQuery.)]
    (doseq [[q op] args]
      (.add query q (condp = op
                      :and BooleanClause$Occur/MUST
                      :or  BooleanClause$Occur/SHOULD
                      :not BooleanClause$Occur/MUST_NOT)))

    query))
```

A.7 molly.server

This namespace contains functionality to expose the system functionality to clients over HyperText Transfer Protocol (HTTP).

A.7.1 molly.server.core

```
(ns molly.server.core
  (:require [compojure.core :refer [GET defroutes]]
            [compojure.handler :refer [site]]
            [compojure.route :refer [not-found resources]]
            [molly.conf.config :refer [load-props]]
            [molly.search.lucene :refer [idx-path idx-searcher]]))

(defroutes app-routes
           (GET "/" [] "root")
           (resources "/")
           (not-found "Can't find that one."))

(def config (load-props))
(def searcher (idx-searcher (idx-path (config :idx.path))))

(def handler
  (site app-routes))
```

A.7.2 molly.server.remotes

Rather than handle serialization over HTTP manually, the system uses the Shoreleave library[1]. It permits ClojureScript clients to transparently call functions exposed on the server. The `defremote` macro is used to expose these functions.

```
(ns molly.server.remotes
  (:require [compojure.handler :refer [site]]
            [molly.server.core :refer [handler]]
            [molly.server.search :refer [compute-span
                                         find-entities
                                         find-entity
                                         find-value]]
            [shoreleave.middleware.rpc :refer [defremote wrap-rpc]]))

(defremote get-value [q]
           (find-value q))

(defremote get-entities [q]
           (find-entities q))

(defremote get-entity [id]
           (find-entity id))

(defremote get-span [s t method]
           (compute-span s t method))

(def app (->
           (var handler)
           (wrap-rpc)
           (site)))
```

[1] https://github.com/shoreleave/shoreleave-remote

A.7.3 molly.server.search

This namespace provides the "glue" between the system and HTTP interface.

```clojure
(ns molly.server.search
  (:require [molly.algo.bfs :refer [bfs]]
            [molly.algo.bfs-atom :refer [bfs-atom]]
            [molly.algo.bfs-ref :refer [bfs-ref]]
            [molly.datatypes.entity :refer [doc->data]]
            [molly.search.lucene :refer [idx-search]]
            [molly.search.query-builder :refer [boolean-query query]]
            [molly.server.core :refer [config searcher]]
            [molly.util.nlp :refer [q-gram]]))

(def runtime (Runtime/getRuntime))

(defn dox
  [q field S op topk]
  (let [bq      (boolean-query
                  (concat [[q :and]]
                          (for [s S]
                            [(query field s) op])))
        result  (map doc->data (idx-search searcher bq topk))
        fmt     (fn [data] {:meta (meta data) :results data})]
    (map fmt result)))

(defn entities
  [field q topk]
  (dox (query :type :entity)
       field
       (clojure.string/split q #"\s{1}")
       :and
       topk))

(defn find-value [q]
  (dox (query :type :value)
       :text
       (clojure.string/split (q-gram q) #"\s{1}")
       :or
       (config :idx.topk.value)))

(defn find-entities [q]
  (entities
    :text (clojure.string/lower-case q)
    (config :idx.topk.entities)))
```

```
(defn find-entity [id]
  (entities :id id (config :idx.topk.entity)))

(defn compute-span [s t method]
  (let [max-hops (config :idx.search.max-hops)
        start         (System/nanoTime)
        [visited dist prev]
        (condp = method
          "bfs"   (bfs searcher s t)
          "atom"  (bfs-atom searcher s t)
          "ref"   (bfs-ref searcher s t))
        time-taken    (- (System/nanoTime) start)
        eids          (conj (for [[k v] prev] k) s)
        get-entities  (fn [eid]
                        {(keyword eid)
                         (entities :id eid
                                   (config :idx.topk.entity))})
        entities      (into {} (map get-entities eids))]
    {:from      s
     :to        t
     :prev      prev
     :entities  entities
     :debug     {:time        time-taken
                 :mem_total   (.totalMemory runtime)
                 :mem_free    (.freeMemory runtime)
                 :mem_used    (- (.totalMemory runtime)
                                 (.freeMemory runtime))
                 :properties  config}}))
```

A.8 molly.algo

A.8.1 molly.algo.common

```
(ns molly.algo.common
  (:use clojure.pprint)
  (:require [molly.datatypes.entity :refer [doc->data]]
            [molly.search.lucene :refer [idx-search]]
            [molly.search.query-builder :refer [boolean-query query]]))

(defn find-entity-by-id
  [G id]
  (let [query (boolean-query [[(query :type :entity) :and]
                              [(query :id id) :and]])]
    (map doc->data (idx-search G query 10))))

(defn find-group-for-id
  [G id]
  (let [query   (boolean-query [[(query :type :group) :and]
                                [(query :entities id) :and]])
        results (map doc->data (idx-search G query 10))
        big-str (clojure.string/join ” ”
                                     (map #(% :entities) results))]
    (distinct (clojure.string/split big-str #”\s{1}”))))

(defn find-adj
  [G u]
  (remove #{u} (find-group-for-id G u)))

(defn initial-state
  [s]
  {:Q       (conj (clojure.lang.PersistentQueue/EMPTY) s)
   :marked  #{s}
   :dist    {s 0}
   :prev    {s nil}})

(defn deref-future
  [dfd]
  (if (future? dfd)
    (deref dfd)
    dfd))
```

A.8.2 molly.algo.bfs

```
(ns molly.algo.bfs
  (use molly.algo.common))

(defn update-adj
  [G marked dist prev u]
  (loop [adj      (find-adj G u)
         marked   marked
         dist     dist
         prev     prev
         frontier []]
    (if (empty? adj)
      [(conj marked u) dist prev frontier]
      (let [v     (first adj)
            adj’  (rest adj)]
        (if (marked v)
          (recur adj’ marked dist prev frontier)
          (let [dist’ (assoc dist v (inc (dist u)))
                prev’ (assoc prev v u)]
            (recur adj’ marked dist’ prev’ (conj frontier v))))))))

(defn bfs
  [G s t]
  (loop [Q      (-> (clojure.lang.PersistentQueue/EMPTY) (conj s))
         marked #{s}
         dist   {s 0}
         prev   {s nil}]
    (if (or (empty? Q)
            (some (fn [node] (= node t)) marked))
      [marked dist prev]
      (let [u   (first Q)
            Q’  (rest Q)
            [marked’ dist’ prev’ frontier]
            (update-adj G marked dist prev u)]
        (recur (concat Q’ frontier) marked’ dist’ prev’)))))
```

A.8.3 molly.algo.bfs_atom

```
(ns molly.algo.bfs-atom
  (use molly.algo.common))

(defn update-state
  [state u v]
  (let [Q        (state :Q)
        marked   (state :marked)
        dist     (state :dist)
        prev     (state :prev)]
    (assoc state
           :Q       (conj Q v)
           :marked  (conj marked v)
           :dist    (assoc dist v (inc (dist u)))
           :prev    (assoc prev v u))))

(defn update-adj
  [state-ref G u]
  (let [marked?   (@state-ref :marked)
        deferred  (doall
                    (for [v (find-adj G u)]
                      (if (marked? v)
                        nil
                        (future (swap! state-ref update-state u v)))))]
    (doall (map deref-future deferred))))

(defn bfs-atom
  [G s t]
  (let [state-ref (atom (initial-state s))]
    (while (and (not (empty? (@state-ref :Q)))
                (not (@state-ref :done)))
      (let [u     (first (@state-ref :Q))
            Q’    (pop (@state-ref :Q))]
        (swap! state-ref assoc :Q Q’)
        (if (some (fn [node] (= node t)) (@state-ref :marked))
          (swap! state-ref assoc :done true)
          (update-adj state-ref G u))))
    [(@state-ref :marked) (@state-ref :dist) (@state-ref :prev)]))
```

A.8.4 molly.algo.bfs_ref

```
(ns molly.algo.bfs-ref
  (use molly.algo.common))

(defn update-state
  [state u v]
  (let [Q       (state :Q)
        marked  (state :marked)
        dist    (state :dist)
        prev    (state :prev)]
    (assoc state
           :Q       (conj Q v)
           :marked  (conj marked v)
           :dist    (assoc dist v (inc (dist u)))
           :prev    (assoc prev v u))))

(defn update-adj
  [state-ref G u]
  (let [marked?   (@state-ref :marked)
        deferred  (doall
                    (for [v (find-adj G u)]
                      (if (marked? v)
                        nil
                        (future (dosync (alter
                                          state-ref
                                          update-state
                                          u
                                          v))))))]
    (doall (map deref-future deferred))))

(defn bfs-ref
  [G s t]
  (let [state-ref (ref (initial-state s))]
    (while (and (not (empty? (@state-ref :Q)))
                (nil? ((@state-ref :marked) t)))
      (let [u   (first (@state-ref :Q))
            Q'  (pop (@state-ref :Q))]
        (dosync (alter state-ref assoc :Q Q'))
        (update-adj state-ref G u)))
     [(@state-ref :marked) (@state-ref :dist) (@state-ref :prev)]))
```

A.9 molly.bench

A.9.1 molly.bench.benchmark

```
(ns molly.bench.benchmark
  (:require [clojure.data.json :as json]
            [criterium.core :refer [benchmark]]))

(defn benchmark-search
  [f G s t]
  (let [method  (last (clojure.string/split (str (class f)) #"\$"))
        result
        (dissoc
          (benchmark (f G s t) {:verbose false})
          :results)]
    (println
      (json/write-str
        {:method    method
         :results   result}))))
```

Appendix B

Data Corpus

Tables representing various object classes of data corpus used in thesis implementation and evaluation.

Property	Data Type
id	INT
name	VARCHAR

(a) Instructor Data Structure

Property	Data Type
code	VARCHAR
title	VARCHAR
description	VARCHAR

(b) Course Data Structure

Property	Data Type
id	INT
schedule_id	INT
instructor_id	INT
position	VARCHAR

(c) Teaches Data Structure

Property	Data Type
id	INT
registration_start	DATE
registration_end	DATE
credits	FLOAT
sec_code	VARCHAR
sec_number	VARCHAR
semester	VARCHAR
course	VARCHAR
levels	VARCHAR
campus	VARCHAR
capacity	INT
actual	INT
year	VARCHAR

(d) Section Data Structure

Property	Data Type
id	INT
day	VARCHAR
schedtype	VARCHAR
date_start	DATE
date_end	DATE
hour_start	INT
hour_end	INT
min_start	INT
min_end	INT
classtype	VARCHAR
location	VARCHAR
section_id	INT

(e) Schedule Data Structure

Table B.1: Structure of data corpus

Bibliography

[10] Bank account transfer problem. Cunningham & Cunningham, Inc. September 2010. URL: http://c2.com/cgi/wiki?BankAccountTransferProblem.

[BCNS13] Véronique Benzaken, Giuseppe Castagna, Kim Nguyen, and Jérôme Siméon. Static and dynamic semantics of nosql languages. *Sigplan not.*, 48(1):101–114, January 2013. ISSN: 0362-1340. DOI: 10.1145/2480359.2429083. URL: http://dx.doi.org/10.1145/2480359.2429083.

[BHNCS02] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Data engineering, 2002. proceedings. 18th international conference on*, 2002, pages 431–440. DOI: 10.1109/ICDE.2002.994756. URL: http://dx.doi.org/10.1109/ICDE.2002.994756.

[Boy08] Brent Boyer. Robust java benchmarking, part 1: issues. June 2008. URL: http://www.ibm.com/developerworks/java/library/j-benchmark1/index.html.

[CKKS02] W. F. Cody, J.T. Kreulen, V. Krishna, and W. S. Spangler. The integration of business intelligence and knowledge management. *Ibm systems journal*, 41(4):697–713, 2002. ISSN: 0018-8670. DOI: 10.1147/sj.414.0697. URL: http://dx.doi.org/10.1147/sj.414.0697.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms, third edition*. The MIT Press, 3rd edition, 2009. ISBN: 0262033844, 9780262033848.

[Cod79] E. F. Codd. Extending the database relational model to capture more meaning. *Acm trans. database syst.*, 4(4):397–434, December 1979. ISSN: 0362-5915. DOI: 10.1145/320107.320109. URL: http://dx.doi.org/10.1145/320107.320109.

[Col96] George Colliat. Olap, relational, and multidimensional database systems. *Sigmod record*, 25(3):64–69, 1996. DOI: 10.1145/234889.234901. URL: http://dx.doi.org/10.1145/234889.234901.

[DFLDH88] S. T. Dumais, G. W. Furnas, T. K. Landauer, S. Deerwester, and R. Harshman. Using latent semantic analysis to improve access to textual information. In *Proceedings of the sigchi conference on human factors in computing systems*. In CHI '88. ACM, Washington, D.C., USA, 1988, pages 281–285. ISBN: 0-201-14237-6. DOI: 10.1145/57167.57214. URL: http://dx.doi.org/10.1145/57167.57214.

[Dra14] Richard Drake. Rdrake/molly. 2014. URL: http://dx.doi.org/10.6084/m9.figshare.963594.

[ECM13] ECMA International. *Standard ecma-404 - the json data interchange format*, 1st edition, October 2013. URL: http://www.ecma-international.org/publications/standards/Ecma-404.htm.

[Efr87] Bradley Efron. Better bootstrap confidence intervals. *Journal of the american statistical association*, 82(397):171–185, 1987. DOI: 10.1080/01621459.1987.10478410. URL: http://dx.doi.org/10.2307/2289144.

[Fou] Apache Software Foundation. URL: http://lucene.apache.org/core/.

[GUW09] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[HGP03] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient ir-style keyword search over relational databases. In *Proceedings of the 29th international conference on very large data bases - volume 29*. In VLDB '03. VLDB Endowment, Berlin, Germany, 2003, pages 850–861. ISBN: 0-12-722442-4. URL: http://dl.acm.org/citation.cfm?id=1315451.1315524.

[Hica] Rich Hickey. Clojure. URL: http://clojure.org/.

[Hicb] Rich Hickey. Data structures. URL: http://clojure.org/data_structures.

[Hic09] Rich Hickey. Persistent data structures and managed references. QCon London. March 2009. URL: http://qconlondon.com/dl/qcon-london-2009/slides/RichHickey_PersistentDataStructuresAndManagedReferences.pdf.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer journal*, 32(2):98–107, 1989. DOI: 10.1093/comjnl/32.2.98. URL: http://dx.doi.org/10.1093/comjnl/32.2.98.

[ISO11] ISO. Information technology – database languages – sql – part 2: foundation (sql/foundation). ISO (ISO/IEC 9075-2:2011). Geneva, Switzerland: International Organization for Standardization, 2011.

[Jon72] Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28:11–21, 1972. DOI: 10.1108/eb026526. URL: http://dx.doi.org/10.1108/eb026526.

[Li13] Fanshu Li. What is jaql? Technical report. 2013. URL: http://www-01.ibm.com/software/data/infosphere/hadoop/jaql/.

[LJLF11] Guoliang Li, Shengyue Ji, Chen Li, and Jianhua Feng. Efficient fuzzy full-text type-ahead search. *The vldb journal*, 20(4):617–640, August 2011. ISSN: 1066-8888. DOI: 10.1007/s00778-011-0218-x. URL: http://dx.doi.org/10.1007/s00778-011-0218-x.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN: 0521865719, 9780521865715.

[Por97] M. F. Porter. Readings in information retrieval. In Karen Sparck Jones and Peter Willett, editors, part An Algorithm for Suffix Stripping, pages 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN: 1-55860-454-5. URL: http://dl.acm.org/citation.cfm?id=275537.275705.

[RZ09] Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: bm25 and beyond. *Found. trends inf. retr.*, 3(4):333–389, April 2009. ISSN: 1554-0669. DOI: 10.1561/1500000019. URL: http://dx.doi.org/10.1561/1500000019.

[SB88] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. In *Information processing and management*, 1988, pages 513–523. DOI: 10.1016/0306-4573(88)90021-0. URL: http://dx.doi.org/10.1016/0306-4573(88)90021-0.

[Sch11] Silvia Schreier. Modeling restful applications. In *Proceedings of the second international workshop on restful design*. In WS-REST '11. ACM, Hyderabad, India, 2011, pages 15–21. ISBN: 978-1-4503-0623-2. DOI: 10.1145/1967428.1967434. URL: http://dx.doi.org/10.1145/1967428.1967434.

[Sco10] Don Chamberlin Scott Boag. Xquery 1.0: an xml query language. XML Query WG. December 2010. URL: http://www.w3.org/TR/xquery/ (visited on 12/14/2010).

[SR03] C. Silva and B. Ribeiro. The importance of stop word removal on recall values in text categorization. In *Neural networks, 2003. proceedings of the international joint conference on*. Volume 3, July 2003, 1661–1666 vol.3. DOI: 10.1109/IJCNN.2003.1223656.

[Suc98] Dan Suciu. An overview of semistructured data. *Sigact news*, 29(4):28–38, December 1998. ISSN: 0163-5700. DOI: 10.1145/306198.306204. URL: http://dx.doi.org/10.1145/306198.306204.

[Wad98] Philip Wadler. Why no one uses functional languages. *Sigplan not.*, 33(8):23–27, August 1998. ISSN: 0362-1340. DOI: 10.1145/286385.286387. URL: http://dx.doi.org/10.1145/286385.286387.

[Yeg08] Steven Yegge. The universal design pattern. October 2008. URL: http://steve-yegge.blogspot.ca/2008/10/universal-design-pattern.html.

Todo list