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

April 2014

Contents

List of Tables

List of Figures

List of Algorithms

Acronyms

API application programming interface. 3, 31, 35, 36

BFS breadth-first search. 36, 42, 43, 46

DSL domain-specific language. 31

FF Ford-Fulkerson. 36

FK foreign key. vii, 6–11

HTML HyperText Markup Language. 2, 39

HTTP HyperText Transfer Protocol. 66–68

JAR Java archive. 47

JDBC Java database connectivity. 30

JSON JavaScript object notation. vii, 3, 40, 41

JVM Java virtual machine. vi, 30, 31, 38, 40, 41

MDX MultiDimensional eXpressions. 2

OLAP online analytical processing. 2

RDBMS relational database management system. 1, 2, 12, 14

SQL structured query language. 1–3, 7, 10, 21, 32, 59

STM software transactional memory. 29, 46

TF-IDF term frequency and inverse document frequency. 16

UOIT University of Ontario Institute of Technology. i, 39

XML Extensible Markup Language. 2, 3

List of Symbols

schema graph (G) graph representation of schema. 7, 10, 24

entity group (T) forest of named tuples. 10, 24, 25

document collection (C) set of documents. xi, 15–19, 25

terms (T) set of unique terms in a document collection. 15, 16, 19

document (d) set of fields. xi, 14–19, 21–23

search query (q) special case of document. 17–19, 21, 22, 26

field (f) named sub-document in document. 23, 26, 35

term (τ) unique term in document collection. xi, 14–17, 21

N number of documents in collection. 15

database (D) set of relations. 7, 10, 24

relation (r) set of named tuples. xi, 5–7, 10, 24

named tuple (t) ordered set of values. vi, xi, 5–7, 10, 23–25

attribute (α) named column. 6, 7, 10, 23

key (K) uniquely identifies a named tuple in a relation. 6

Chapter 1

Background & Introduction

In this chapter, we provide a background to the motivation of this thesis. We will discuss the evolution of data sets, their logical models, and the corresponding query languages. We feel that modern day data sets call for a new data model with a new query language. This thesis is an attempt to make an incremental advance toward a new data model and new way of querying data.

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.

1.2 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.

1.3 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 [10], 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 []) for JSON data sets.

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.

1.5 Outline of Thesis

Chapter 2 provides a formal definition of the relational data model and the document data model. Rather than introducing a new hybrid model, our thesis is to provide efficient mappings between the two models, and thus allow the data to freely be exchanged between query operators of the domains. We demonstrate that complex relational data can be encoded into a linked document space (and back).**To do** (1)

Chapter 4 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 5 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

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 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.

2.1 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 $\textsc{Attr}[t] = \{\text{code}, \text{title}, \text{subject}\}$. The values are $t[\text{code}] =$

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

Table 2.1: Course relation

"CDPS 101", t[title] = "Human-Mutant Relations", and t[subject] = "CDPS".

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.

$$\forall t, t' \in r, \textsc{Attr}[t] = \textsc{Attr}[t'] \tag{2.1}$$

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 2.1, 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 $\alpha_{1i} \subseteq \text{Attr}[r_1]$ and $\alpha_{2i} \subseteq \text{Attr}[r_2]$. 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 $t[\alpha_{11}, \alpha_{12}, \ldots, \alpha_{1k}] = t'[\alpha_{21}, \alpha_{22}, \ldots, \alpha_{2k}]$.

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

$$\theta = \text{Course(subject)} \rightarrow \text{Subject(id)} \tag{2.2}$$

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 6), keys (as defined by Definition 3 on page 6), and foreign key constraints (as defined by Definition 4 on page 6).

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

2.1.1 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

$$\text{vertices} : \text{V}(G) = \text{Rel}[D] \tag{2.3}$$

$$\text{edges} : \text{E}(G) = \text{FK}[D] \tag{2.4}$$

Example 4. Given the schema in Fig. 2.1 on the next page

and the FK constraints in Fig. 2.2 on the following page

we produce the schema graph in Fig. 2.3 on page 9

The relational data model is particularly powerful for analytic queries. Given the schema graph in Fig. 2.3 on page 9, one can formulate the following analytic queries in a query language known as SQL.

Subject(id, name)

Course(code, title, subject)

Term(id, name)

Section(crn, term, course)

Schedule(id, days, sch_type, time_start, time_end, location, section, instructor)

Instructor(id, name)

Figure 2.1: Subset of mycampus dataset schema

Course(subject) → Subject(id)

Section(term) → Term(id)

Section(course) → Course(code)

Schedule(section) → Section(crn)

Schedule(instructor) → Instructor(id)

Figure 2.2: FK constraints on schema in Fig. 2.1

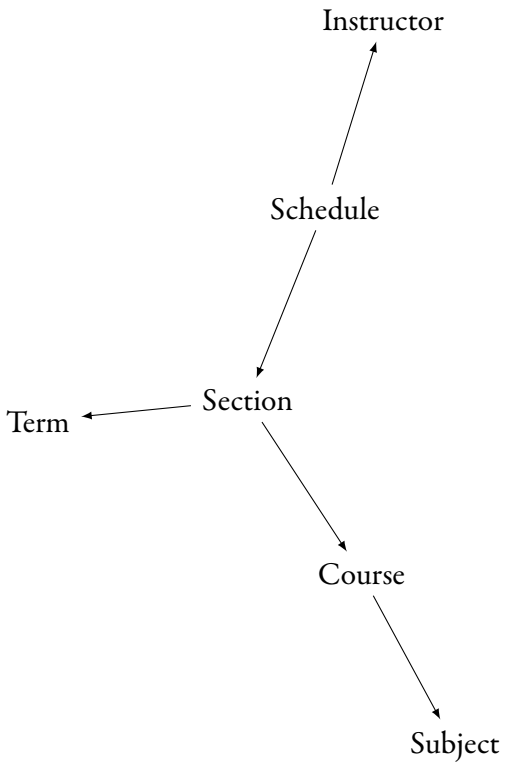

Figure 2.3: Graph representation of relations (Fig. 2.1) and FK (Fig. 2.2)

```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 2.4: Query to find section CRNs for a subject name.

crn
10000
10001
10002

Table 2.2: Results of the query in Fig. 2.4.

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

The SQL query in Fig. 2.4 results in Table 2.2.

2.1.2 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 $r_1(\alpha_{11}, \alpha_{12}, \ldots, \alpha_{1k}) \to r_2(\alpha_{21}, \alpha_{22}, \ldots, \alpha_{2k})$ 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 2.3: Properties of the Course titled Human-Mutant Relations.

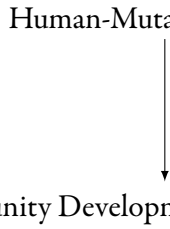

Figure 2.5: Human-Mutant Relations entity group

Example 6. The information in Table 2.4 on page 19 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 (Fig. 4.6 on page 36) to join together all pieces of information related to this course.

2.1.3 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 2.1 on page 6) 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
```

```
4           ON section.course = course.code
5         JOIN subject
6           ON subject.id = course.subject
7   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.

```
1   UPDATE subject
2   SET    name = 'Community Destruction & Policy Studies'
3   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. 2.4 on page 10.

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."

```
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.

2.2 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 $\mathsf{freq}(\tau, d)$ be the frequency of term τ in d, $\mathbb{T}$ denote all possible terms, and $\textsc{Bag}[\mathbb{T}]$ 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

$$d_1 = \{\text{“math”} : 2, \text{“a”} : 1, \text{“is”} : 1, \text{“360”} : 1, \text{“class”} : 1\} \tag{2.5}$$

$$d_2 = \{\text{“a”} : 1, \text{“boring”} : 1, \text{“is”} : 1, \text{“cdps”} : 1, \text{“lecture”} : 1, \text{“101”} : 1\} \tag{2.6}$$

$$d_3 = \{\text{“mathematics”} : 1, \text{“great”} : 1, \text{“was”} : 1, \text{“lecture”} : 1\} \tag{2.7}$$

2.2.1 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 $\text{freq}(\tau, d)$. The document frequency of a term τ, denoted by $\text{df}(\tau)$, is the number of documents in C that contains τ. It is defined as

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

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

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

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

Example 13. Using the documents from Example 12 on page 15, the TF-IDF scores are as follows.

	d_1	d_2	d_3
τ_1 : "101"	0.0000	0.2642	0.0000
τ_2 : "360"	0.3170	0.0000	0.0000
τ_3 : "a"	0.1170	0.0975	0.0000
τ_4 : "boring"	0.0000	0.2642	0.0000
τ_5 : "cdps"	0.0000	0.2642	0.0000
τ_6 : "class"	0.3170	0.0000	0.0000
τ_7 : "great"	0.0000	0.0000	0.3962
τ_8 : "is"	0.1170	0.0975	0.0000
τ_9 : "lecture"	0.0000	0.0975	0.1462
τ_{10} : "math"	0.6340	0.0000	0.0000
τ_{11} : "mathematics"	0.0000	0.0000	0.3962
τ_{12} : "was"	0.0000	0.0000	0.3962

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 15 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 14). 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 15).

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

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 $\text{similarity}(\vec{d}_1, \vec{d}_2)$.

$$\text{similarity}(\vec{d}_1, \vec{d}_2) = \frac{\vec{d}_1 \cdot \vec{d}_2}{\|\vec{d}_1\| \cdot \|\vec{d}_2\|} \tag{2.8}$$

$$= \frac{\sum_{i=1}^{N} \vec{d}_{1,i} \times \vec{d}_{2,i}}{\sqrt{\sum_{i=1}^{N} (\vec{d}_{1,i})^2} \times \sqrt{\sum_{i=1}^{N} (\vec{d}_{2,i})^2}} \tag{2.9}$$

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

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

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

Table 2.4: Properties of the Course titled Human-Mutant Relations.

Example 16. Given the document collection C (from Example 12 on page 15) and search query q, compute the similarity between q and every document $d \in C$.

$$\text{similarity}(\vec{d}_1, \vec{q}) = 0.390890 \tag{2.10}$$

$$\text{similarity}(\vec{d}_2, \vec{q}) = 0.061592 \tag{2.11}$$

$$\text{similarity}(\vec{d}_3, \vec{q}) = 0.252789 \tag{2.12}$$

2.2.2 Extending the Document Model

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

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

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 2.5 on the next page.

which is similar in structure to Table 2.4.

2.2.3 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 on the next page.

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

Table 2.5: Course document for MATH 360.

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 ← []
p ← Repeat(s, n − 1)
S ← Pad(S, p)
S ← 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, Repeat(S, n) repeats s character n times, Pad(S, p) prefixes and postfixes S with p, and 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. 2.6 on the following page shows, using n-grams yield a similarity of $\frac{5}{10}$.

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

Figure 2.6: Comparison between n-grams of G and G'.

2.2.4 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.

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

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

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

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

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 on page 21.

$$\text{query}(\text{“mathematics”}) \neg \text{query}(\text{“lecture”}) \rightarrow \varnothing$$

While none of the above queries required domain knowledge, it is possible to use the extended document model (Section 2.2.2 on page 19) 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 2.2.3 on page 19), users who substitute, delete, or insert characters from the desired term may still receive results for their intended term (see Example 19 on page 20 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 6). 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.

Chapter 3

Best of Both Worlds

3.1 Encoding Named Tuples into Documents

Recall in the extended document model (Section 2.2.2 on page 19), 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.

$$\textsc{Attr}[t] \xrightarrow{analyzed} \textsc{Field}[d] \tag{3.1}$$

$$\alpha_1, \alpha_2, \ldots, \alpha_n \xrightarrow{analyzed} f_1, f_2, \ldots, f_n \tag{3.2}$$

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

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 $\textsc{Doc}[t]$.

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

Table 3.1: Doc[t]

3.2 Mapping of Entity Groups to Documents

Recall that an entity group (Definition 7 on page 10) is a forest T of tuples t such that for every $(t, t') \in T$, where $t \neq t'$, implies Rel[t] $\neq$ 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 V(T), and the edges of T as E(T).

Example 25. Using the schema graph...

Claim 1. Given V(T), we are always able to reconstruct T.

Proof. Given V(T), we must reconstruct E(T) in order to complete T.

Choose any $(t, t') \in$ V(T). If (Rel[t], 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. □

To do (2)

3.3 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.

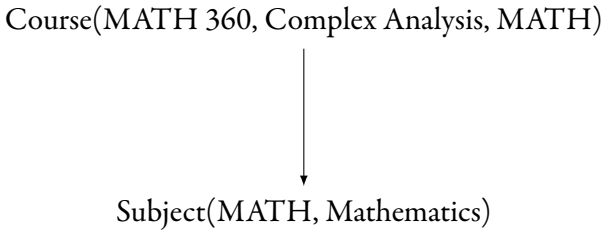

Figure 3.1: Example entity group

For every $t \in \mathrm{V}(T)$, we construct a document $\textsc{Doc}[t]$ (Section 3.1 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.

$$x[\text{“entities”}] = \bigcup_{t \in \mathrm{V}(T)} \mathrm{UID}[t] \tag{3.3}$$

Thus, the encoding of T is defined as

$$T \xrightarrow{\text{encode}} \{\textsc{Doc}[t] : t \in \mathrm{V}(T)\} \cup \{x\} \tag{3.4}$$

Example 26. An entity group produced from the schema in Fig. 2.3 on page 9 is shown in Fig. 3.1.

Transforming the example entity group in Fig. 3.1 would produce documents shown in Table 3.2 on the following page

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

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

3.4 Encoding Attribute Values into Searchable Documents

Each value for user selected attributes are converted into *n*-grams, and stored in special documents.

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 3.2: Document encoding of Fig. 3.1 on page 25

3.5 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 2.2.3) for values, searching for entities containing keywords (see Example 27), and the discovery of intermediate entities given two known entities.

Example 27 (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 28 (Entity Graph Search)**.** Find the shortest path between the two entities with the unique identities of "subject|math" and "instructor|5".

To do (3)

Chapter 4

Along Came Clojure

To do (4)

4.1 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].

4.1.1 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.

In Fig. 4.1 on the next page, 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.

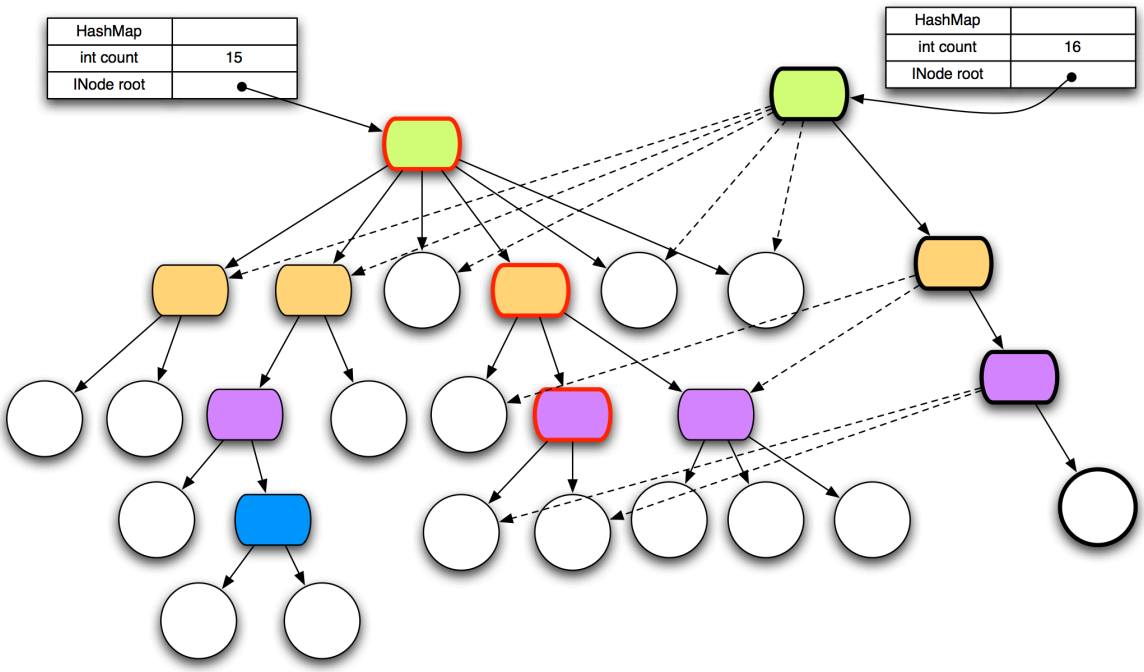

Figure 4.1: Representation of how data structures are "changed" in Clojure (Source: [Hic09])

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 4.1.

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.

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 4.1: Comparison between Clojure's four systems for concurrency

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 4.2: JVM interoperability

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.

To do (5)

Interoperability With the JVM

To do (6) 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 syntactic sugar provided by 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 4.2.

We utilize Clojure's JVM interoperability to make use of Apache Lucene and Java database connectivity (JDBC).

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

Figure 4.2: Clojure code that, given a path, returns a `Directory` object

4.2 Search With Clojure

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

4.2.1 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 2.2), 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 2.2.1), utilize the extended document model to provide semi-structured documents (Section 2.2.2), and issue search queries against the index (Definition 12).

4.2.2 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 3.3) in order to satisfy Section 3.2.

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

Schema Graph Definition

The schema graph is defined using Korma, which "is a domain-specific language (DSL) for Clojure" [Gra]. 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

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 6)	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 4.3: Keys expected by `EntitySchema` records

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

Figure 4.3: Building the index

and identified. The keys of this map are given in Table 4.3.

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. 4.3).

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

Key	Value
:type	:value
:class	<rel>|<attr>

(a) Value

Key	Value
:type	:entity
:class	<rel>
:ID	<rel>|<pk>

(b) Entity

Key	Value
:type	:group
:entities	<rel>|<pk>
	[<rel>|<pk> ...]

(c) Entity Group

Table 4.4: Metadata associated with each type

attribute) values.

Transformation

For every named tuple, a document is constructed (see Section 3.1). 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 4.4.

The internal representation of the named tuple given in Table 2.4 on page 19 is given in Fig. 4.4 on the next page.

Once the internal representation is constructed, it may be transformed into a document. The

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

Figure 4.4: Internal representation of named tuple from Table 2.4 on page 19

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

Table 4.5: Document of internal representation from Fig. 4.4

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 key name. This allows the system to differentiate between metadata and attributes. The transformation from internal representation to document is given in Table 4.5.

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 3.3) of every entity group is added.

4.2.3 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 2.2.3). 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

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

The search API provides a function that accepts a symbol defining the field to search in, as well

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

Figure 4.5: Boolean query in Clojure

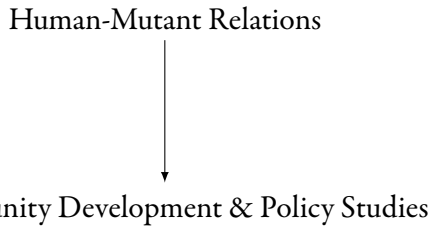

Figure 4.6: Human-Mutant Relations entity group

one or more words to search for. A phrase query comprised of every word is constructed.

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

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 $\cap$, $\cup$, and $\neg$, respectively.

Example 29 (`boolean-query`). For example, the query in Example 23 is given in Fig. 4.5.

Translation of Search Results to Relational Space

4.2.4 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 4.2.3), 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 breadth-first search (BFS).

- Why we need graph search

- Search in document graph using graph search algorithms with functional implementations: (Ford Fulkerson, BFS)
- Speed up using concurrency
- Clojure specific optimization: ref + atom

Chapter 5

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 2. We cover the implementation details in Section 5.1, and the methodology and evaluation in Section 5.3.

5.1 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 4.1.1.

5.1.1 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 [Dra]. **To do** (7)

5.2 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 campuses (Table B.1a), courses (Table B.1c), instructors (Table B.1b), locations (Table B.1d), schedules (Table B.1g), sections (Table B.1h), subjects (Table B.1e), and terms (Table B.1f). A graph representation of how these classes of objects are related can be found in Fig. 2.3. The data corpus is defined in Appendix B

The number of objects, as of the publication of this thesis, can be found in Table 5.1.

Class	Count
Campus	8
Course	1797
Instructor	560
Location	220
Schedule	20985
Section	6211
Subject	68
Term	32

Table 5.1: Number of objects in data corpus, grouped by class

5.3 Runtime Evaluation

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

5.3.1 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. 5.1.

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

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

Figure 5.1: Partial JSON output from Criterium.

5.3.2 Performance

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

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.

Number of Groups	Elapsed Time (s)
1	16.895
2	21.624
3	24.436
4	24.528
5	25.334

Table 5.2: Indexing time growth by number of entity groups

We see in Table 5.2 the indexing time increases considerably between 1 and 2 groups. The number of entity groups also grew considerably, explaining the time increase. The number of entity groups grew at a slower pace from 2 to 3, and 3 to 5, causing the reduced increase in elapsed time.

Keyword Search

Graph Search

The worst-case performance of BFS is $\mathcal{O}(n^2)$. This is reflected in Fig. 5.3 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. 5.3 the decrease in rate of growth is not as high as expected. BFS is the slowest, closely followed by BFS with references and BFS with atoms.

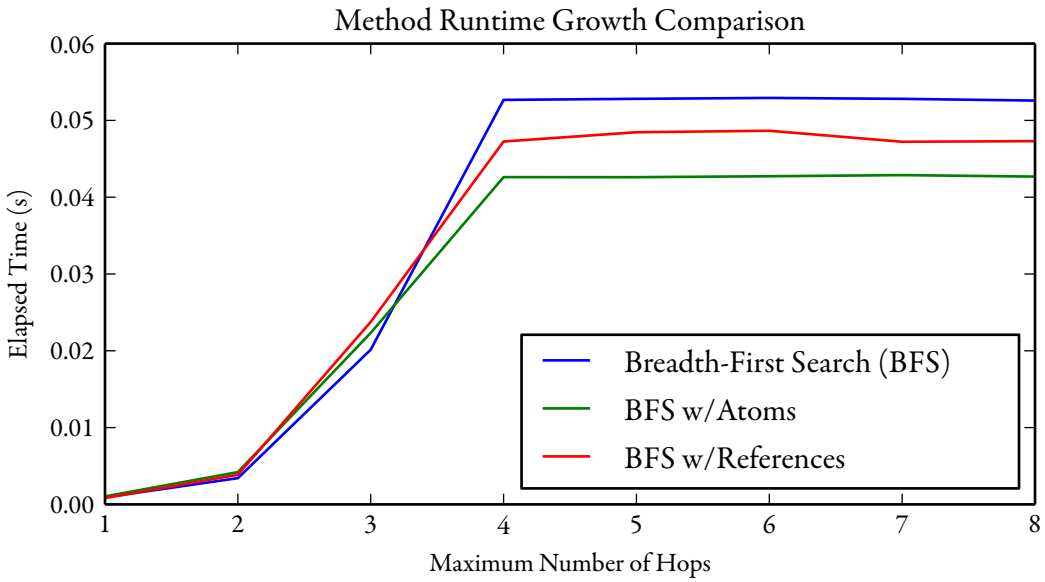

Figure 5.2: Comparison of time taken per hop between all methods (top-k entities: 25), target found

The most likely cause for the similarity is how the concurrency is implemented. In our implementation, we concurrently search for neighbours. These neighbours must be added to the frontier before proceeding. Therefore the algorithm must synchronize upon completion of the neighbour search in order to populate the new frontier.

Atoms are less expensive to use than references as they are handled in an uncoordinated manner in Clojure, reducing the amount of overhead [Nar12]. This explains the difference between the two concurrent implementations of BFS.

Also of note is the growth of Fig. 5.2 compared to that of Fig. 5.3. In the former figure, the target is located by the fourth hop. In the latter, the target is not found. This leads to hops 4 through 8 having approximately the same runtime as one another.

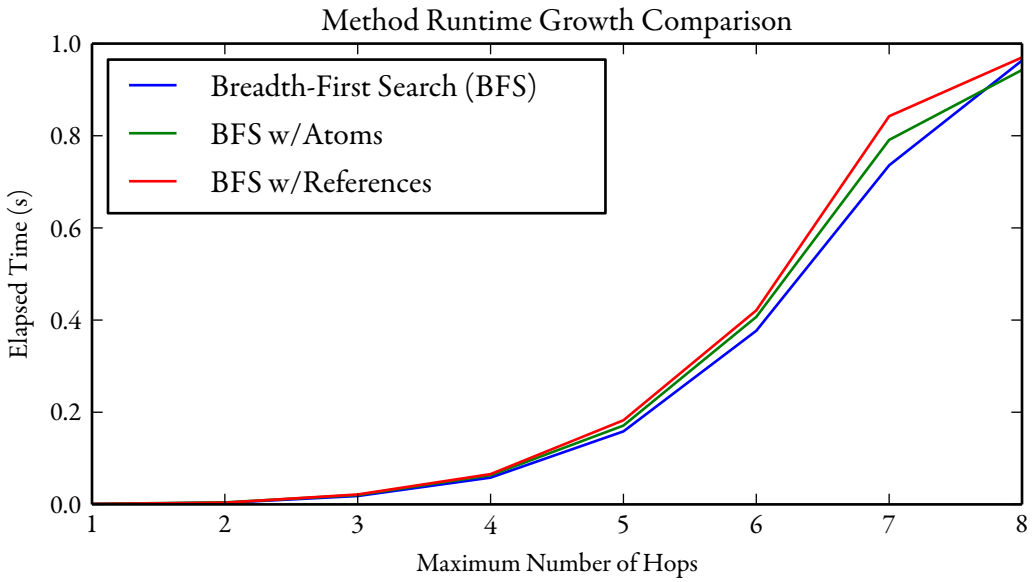

Figure 5.3: Comparison of time taken per hop between all methods (top-k entities: 25), target not found

Chapter 6

Conclusion

6.1 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 2 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 3, providing a formal definition of a framework for the translation between the relational and document data models. Our implementation was introduced in Chapter 4, which also covered Clojure. In Chapter 5, we evaluated our implementation, identifying performance characteristics.

6.2 Lessons Learned

The system evaluation in Chapter 5 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.

```clojure
(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.algo.ford-fulkerson :refer [ford-fulkerson]]
            [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. %)]
```

```
22          ["--index"       "Build an index of the database"
23           :default false
24           :flag true]
25          ["--benchmark"   "Run benchmarks"
26           :default false
27           :flag true]
28          ["-d" "--debug"  "Displays additional information."
29           :default false
30           :flag true]
31          ["-h" "--help"   "Show help"
32           :default false
33           :flag true]))
34
35  (defn -main
36    [& args]
37    (let [[opts arguments banner] (parse-args (flatten args))]
38      (when (or (opts :help) (not (opts :config)))
39        (println banner)
40        (System/exit 0))
41
42      (let [properties  (load-props (opts :config))
43            max-hops    (if (opts :max-hops)
44                          (opts :max-hops)
45                          (properties :idx.search.max-hops))]
46        (when (opts :index)
47          (let [database  (properties :db.path)
48                index     (properties :idx.path)]
49            (build database index)))
50        (when (opts :algorithm)
51          (let [searcher  (idx-searcher
52                           (idx-path
53                             (properties :idx.path)))
54                source    (opts :source)
55                target    (opts :target)
56                f         (condp = (opts :algorithm)
57                            "bfs"             bfs
58                            "bfs-atom"        bfs-atom
59                            "bfs-ref"         bfs-ref
60                            "ford-fulkerson"  ford-fulkerson
61                            (throw
62                              (Exception.
63                                "Not a valid algorithm choice.")))]
64            (if (opts :debug)
65              (let [[marked dist prev] (f searcher
66                                          source
```

```
                                        target
                                        max-hops)]
            (println marked)
            (println dist)
            (println prev))
          (benchmark-search f searcher source target max-hops))
        (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])))

(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
  (:require [korma.core :refer [belongs-to
                                defentity
                                has-many
                                pk
                                select*
                                with]]
            [korma.db :refer [defdb sqlite3]])
  (:import (molly.conf.config IConfig)
           (molly.datatypes.database Sqlite)
           (molly.datatypes.schema EntitySchema)))

(declare Campus Course Subject Term Section Schedule
         Location Instructor db-conn)

(defentity Campus
  (has-many Location))

(defentity Location
  (belongs-to Campus))

(defentity Subject
  (has-many Course))

(defentity Course
  (pk :code)
  (belongs-to Subject)
  (has-many Section))

(defentity Instructor
  (has-many Schedule))

(defentity Term
  (has-many Section))

(defentity Section
  (pk :crn)
  (has-many Schedule)
  (belongs-to Term)
  (belongs-to Course {:fk :course_code}))

```

```
(defentity Schedule
  (belongs-to Section)
  (belongs-to Instructor)
  (belongs-to Location))

(def mycampus-schema
  [(EntitySchema.
     {:T      :entity
      :C      :course
      :sql    Course
      :ID     :code
      :attrs  [:code :title]
      :values [:code :title]})
   (EntitySchema.
     {:T      :entity
      :C      :instructor
      :sql    Instructor
      :ID     :id
      :attrs  [:name]
      :values [:name]})
   (EntitySchema.
     {:T      :entity
      :C      :location
      :sql    Location
      :ID     :id
      :attrs  [:name]
      :values [:name]})
   (EntitySchema.
     {:T      :entity
      :C      :subject
      :sql    Subject
      :ID     :id
      :attrs  [:id :name]
      :values [:id :name]})
   (EntitySchema.
     {:T      :entity
      :C      :campus
      :sql    Campus
      :ID     :id
      :attrs  [:name]
      :values [:name]})
   (EntitySchema.
     {:T      :entity
      :C      :term
      :sql    Term
```

```
87        :ID     :id
88        :attrs  [:id :name]
89        :values [:id :name]})
90    (EntitySchema.
91      {:T      :entity
92       :C      :section
93       :sql    Section
94       :ID     :crn
95       :attrs  [:crn :reg_start :reg_end :credits
96                :section_num :levels]
97       :values [:crn]})
98    (EntitySchema.
99      {:T      :entity
100      :C      :schedule
101      :sql    Schedule
102      :ID     :id
103      :attrs  [:days :sch_type :date_start :date_end
104               :time_start :time_end :week]
105      :values []})
106   (EntitySchema.
107     {:T      :group
108      :C      "Instructor schedule"
109      :sql    (->
110                (select* Schedule)
111                (with Instructor))
112      :ID     [[:instructor :instructor_id  "Instructor ID"]
113               [:schedule   :id             "Schedule ID"]]
114      :attrs  []
115      :values []})
116   (EntitySchema.
117     {:T      :group
118      :C      "Course schedule"
119      :sql    (->
120                (select* Schedule)
121                (with Section
122                      (with Course)))
123      :ID     [[:section  :crn  "CRN"]
124               [:course   :code "Code"]
125               [:schedule :id   "Schedule ID"]]
126      :attrs  []
127      :values []})
128   (EntitySchema.
129     {:T      :group
130      :C      "Schedule location"
131      :sql    (->
```

```
                (select* Schedule)
                (with Location
                      (with Campus)))
      :ID     [[:campus   :campus_id    "Campus ID"]
               [:location :location_id  "Location ID"]
               [:schedule :id           "Schedule ID"]]
      :attrs  []
      :values []})
   (EntitySchema.
     {:T      :group
      :C      "Course subject"
      :sql    (->
               (select* Course)
               (with Subject))
      :ID     [[:course   :id         "Course"]
               [:subject  :subject_id "Subject"]]
      :attrs  []
      :values []})
   (EntitySchema.
     {:T     :group
      :C     "Section term"
      :sql   (->
              (select* Section)
              (with Term))
      :ID    [[:section  :id       "Section"]
              [:term     :term_id  "Term"]]})
   ])

(defrecord Mycampus [db-path idx-path]
  IConfig
  (connection
    [this]
    (defdb db-conn (sqlite3 {:db db-path}))
    (Sqlite. db-conn))
  (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 [korma.core :refer [select]] [korma.db :refer [with-db]]))

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

(deftype Sqlite [conn]
  Database
  (execute-query
    [this query f]
    (with-db conn
             (doseq [result (select query)]
               (f result)))))
```

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.

```
(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))
```

```
40
41  (defn row->data
42    ^{:doc "Transforms a row into the internal representation."}
43    [this schema]
44    (let [T         (schema :T)
45          C         (schema :C)
46          attr-cols (schema :attrs)
47          attrs     (if (nil? attr-cols)
48                      this
49                      (select-keys this attr-cols))
50          meta-data {:type T :class C}
51          id-col    (schema :ID)]
52      (with-meta (if (= T :group)
53                   (conj attrs {:entities (uid this id-col)})
54                   attrs)
55                 (condp = T
56                   :value   (assoc meta-data
57                                   :class
58                                   (clojure.string/join "|"
59                                      (map name
60                                           [C (first attr-cols)])))
61                   :entity  (assoc meta-data :id
62                                   (if (coll? id-col)
63                                     (uid this id-col)
64                                     (uid this C id-col)))
65                   :group   (assoc meta-data
66                                   :entities
67                                   (uid this id-col))
68                   (throw
69                     (IllegalArgumentException.
70                       "I only know how to deal with types :value,
71                       :entity, and :group"))))))
72
73  (defn doc->data
74    ^{:doc "Transforms a Document into the internal representation."}
75    [this]
76    (let [fields        (.getFields this)
77          extract       (fn [x] [(keyword (clojure.string/replace
78                                            (.name x) "_" ""))
79                                 (.stringValue x)])
80          check-special (fn [x] (special? (.name x)))
81          filter-fn     (fn [f] (apply hash-map
82                                       (flatten
83                                         (map extract
84                                              (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 [korma.core :refer [fields group modifier]]
            [molly.datatypes.database :refer [execute-query]]
            [molly.datatypes.entity :refer [data->doc row->data]]
            [molly.search.lucene :refer [add-doc]]))

(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
                       (modifier "DISTINCT")
                       (fields value)
                       (group 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))
```

```
38    (schema-map
39      [this]
40      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.algo.ford-fulkerson :refer [ford-fulkerson]]
            [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)
```

```clojure
    (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 max-hops)
          "atom"  (bfs-atom searcher s t max-hops)
          "ref"   (bfs-ref searcher s t max-hops)
          "ff"    (ford-fulkerson searcher s t max-hops))
        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

```clojure
(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 t]
  {:Q       (conj (clojure.lang.PersistentQueue/EMPTY) s)
   :marked  #{s}
   :dist    {s 0}
   :prev    {s nil}
   :done    false
   :target  t})

(defn update-state
  [state u v max-hops]
  (let [Q       (state :Q)
        marked  (state :marked)
        dist    (state :dist)
        prev    (state :prev)
```

```
        target  (state :target)
        done    (or (>= (dist u) max-hops)
                    (= v target))]
    (if done
      (assoc state
             :marked  (conj marked v)
             :dist    (assoc dist v (inc (dist u)))
             :prev    (assoc prev v u)
             :done    done)
      (assoc state
             :Q       (conj Q v)
             :marked  (conj marked v)
             :dist    (assoc dist v (inc (dist u)))
             :prev    (assoc prev v u)))))

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

A.8.2 molly.algo.bfs

```clojure
(ns molly.algo.bfs
  (:require [molly.algo.common :refer [find-adj initial-state
                                       update-state]]))

(defn update-adj
  [state-ref G u max-hops]
  (let [marked?   (@state-ref :marked)
        deferred  (doall
                    (for [v (find-adj G u)]
                      (when-not (marked? v)
                        (swap!
                          state-ref
                          update-state
                          u
                          v
                          max-hops))))]))

(defn bfs
  [G s t max-hops]
  (let [state-ref (atom (initial-state s t))]
    (while (and (seq (@state-ref :Q))
                (not (@state-ref :done)))
      (let [u     (first (@state-ref :Q))
            Q'    (pop (@state-ref :Q))]
        (swap! state-ref assoc :Q Q')
        (update-adj state-ref G u max-hops)))
    [(@state-ref :marked) (@state-ref :dist) (@state-ref :prev)]))
```

A.8.3 molly.algo.bfs_atom

```clojure
(ns molly.algo.bfs-atom
  (:require [molly.algo.common :refer [deref-future
                                       find-adj
                                       initial-state update-state]]))

(defn update-adj
  [state-ref G u max-hops]
  (let [marked?   (@state-ref :marked)
        done?     (@state-ref :done)
        deferred  (if done?
                    []
                    (doall
                      (for [v (find-adj G u)]
                        (when-not (marked? v)
                          (future
                            (swap!
                              state-ref
                              update-state
                              u
                              v
                              max-hops))))))]
    (doall (map deref-future deferred))))

(defn bfs-atom
  [G s t max-hops]
  (let [state-ref (atom (initial-state s t))]
    (while (and (seq (@state-ref :Q))
                (not (@state-ref :done)))
      (let [u     (first (@state-ref :Q))
            Q'    (pop (@state-ref :Q))]
        (swap! state-ref assoc :Q Q')
          (update-adj state-ref G u max-hops)))
    [(@state-ref :marked) (@state-ref :dist) (@state-ref :prev)]))
```

A.8.4 molly.algo.bfs_ref

```clojure
(ns molly.algo.bfs-ref
  (:require [molly.algo.common :refer [deref-future
                                       find-adj
                                       initial-state update-state]]))

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

(defn bfs-ref
  [G s t max-hops]
  (let [state-ref (ref (initial-state s t))]
    (while (and (seq (@state-ref :Q))
                (not (@state-ref :done)))
      (let [u   (first  (@state-ref :Q))
            Q'  (pop    (@state-ref :Q))]
        (dosync (alter state-ref assoc :Q Q'))
        (update-adj state-ref G u max-hops)))
    [(@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 max-hops]
  (let [method  (last (clojure.string/split (str (class f)) #"\$"))
        result
        (dissoc
          (benchmark (f G s t max-hops) {:verbose false})
          :results)]
    (println
      (json/write-str
        {:method    method
         :max-hops  max-hops
         :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) Campus Data Structure

Property	Data Type
id	INT
name	VARCHAR

(b) Instructor Data Structure

Property	Data Type
code	VARCHAR
title	VARCHAR
subject_id	VARCHAR

(c) Course Data Structure

Property	Data Type
id	INT
name	VARCHAR
campus_id	VARCHAR

(d) Location Data Structure

Property	Data Type
id	VARCHAR
name	VARCHAR

(e) Subject Data Structure

Property	Data Type
id	VARCHAR
name	VARCHAR

(f) Term Data Structure

Property	Data Type
id	INT
days	VARCHAR
sch_type	VARCHAR
date_start	DATE
date_end	DATE
week	VARCHAR
time_start	TIME
time_end	TIME
location_id	INT
section_id	INT
instructor_id	INT

(g) Schedule Data Structure

Property	Data Type
crn	VARCHAR
reg_start	DATE
reg_end	DATE
credits	FLOAT
section_num	VARCHAR
term_id	VARCHAR
course_code	VARCHAR
levels	VARCHAR

(h) Section Data Structure

Table B.1: Structure of data corpus

Bibliography

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

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

[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.

[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.

[Dra] Richard Drake. Rdrake/molly. URL: https://github.com/rdrake/Molly.

[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/.

[Gra] Chris Granger. Korma: tasty sql for clojure. URL: http://sqlkorma.com/.

[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. London, United Kingdom: 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.

[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.

[Nar12] Ronen Narkis. Clojure concurrency. Slide 17. 2012. URL: http://narkisr.github.io/clojure-concurrency/.

[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.

[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.

To do (8)

To do...

- ☐ 1 (p. 4): last sentence doesn't flow
- ☐ 2 (p. 24): Above proof could use some love
- ☐ 3 (p. 26): Finish example
- ☐ 4 (p. 27): Talk about how great Clojure is.
- ☐ 5 (p. 30): More concurrency
- ☐ 6 (p. 30): Compare to other functional languages with decent library support
- ☐ 7 (p. 38): Expand on this. Why is GitHub significant?
- ☐ 8 (p. 81): Fix bibliography entries with short or no keys