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

December 2013

Contents

List of Tables

List of Figures

List of Algorithms

Acronyms

CSV comma-separated values. 35

DSL domain-specific language. 30

FK foreign key. vii, 3–8

JDBC Java database connectivity. 28, 29

JSON JavaScript object notation. 35

JVM Java virtual machine. vi, 28, 29

RDBMS relational database management system. 9, 11

SQL structured query language. 4, 7, 19, 30

STM software transactional memory. 27

TF-IDF term frequency and inverse document frequency. 13

List of Symbols

Chapter 1

Background (2 days)

Literature search on:

- DBExplore
- XRank
- BANKS
- ...

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

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

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 $r \models \mathrm{K}$, 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

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

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

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

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 $t'[\text{id}] = \text{“CDPS”}$.

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

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 6

The relational data model is particularly powerful for analytic queries. Given the schema graph in Fig. 2.3 on page 6, one can formulate the following analytic queries in a query language known as structured query language (SQL).

Subject($\underline{\text{id}}$, name)

Course($\underline{\text{code}}$, title, subject)

Term($\underline{\text{id}}$, name)

Section($\underline{\text{crn}}$, term, course)

Schedule($\underline{\text{id}}$, days, sch_type, time_start, time_end, location, section, instructor)

Instructor($\underline{\text{id}}$, 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

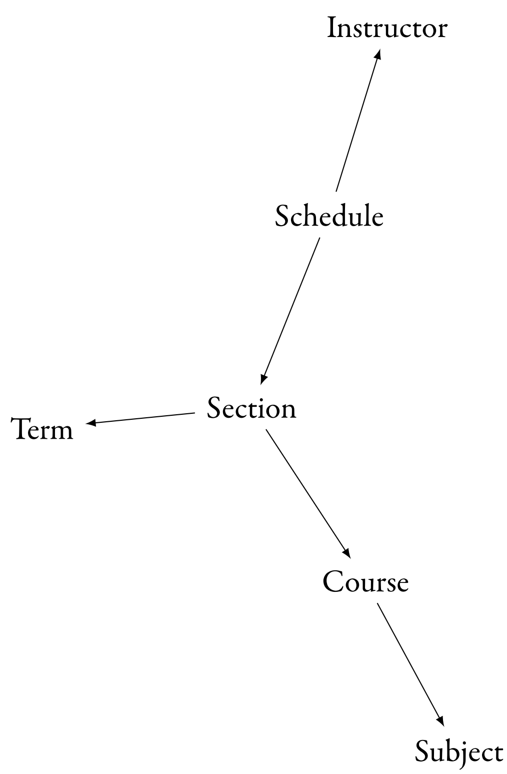

Figure 2.3: Graph representation of relations (Fig. 2.1) and FK (Fig. 2.2)

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

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

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

Table 2.3: Properties of the Course titled Human-Mutant Relations.

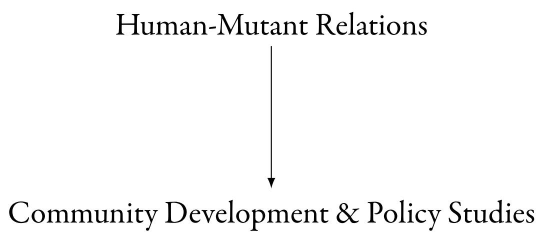

Figure 2.5: Human-Mutant Relations entity group

Example 6. The information in Table 2.4 on page 17 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. 2.5) 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 3) 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 relational database management system (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."

```sql
SELECT Count(*)
FROM   section
       JOIN course
         ON section.course = course.code
       JOIN subject
         ON subject.id = course.subject
WHERE  subject.name = 'Community Development & Policy Studies';
```

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

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

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

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

Cons

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

An example of a complicated query involving two joins is give in Fig. 2.4 on page 7.

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

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

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

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

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

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

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

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

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

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

Example 12. Consider the following short phrases

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

Each sentence phrase produces a document, giving us the following

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

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

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

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

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

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

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

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

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

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

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

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

Example 14. The documents in Example 12 on page 12 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 12). The top-k answers to q with respect to a collection C is defined as the k documents, $\{d_1, d_2, \dots, d_k\}$ in C, such that $\{\vec{d}_1, \vec{d}_2, \dots, \vec{d}_k\}$ 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 12).

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

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

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

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

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

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 12) 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 $\textsc{Field}[d]$, and each field has a value. Thus

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

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

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

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

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

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

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

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

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

Comparing G to G' results in the following matrix

As Fig. 2.6 on the following page shows, using n-grams yield a similarity of $\frac{5}{10}$.

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

Figure 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

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

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

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

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 the previous page.

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

While none of the above queries required domain knowledge, it is possible to use the extended document model (Section 2.2.2 on page 17) 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 17), users who substitute, delete, or insert characters from the desired term may still receive results for their intended term (see Example 19 on page 18 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 3). 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 17), 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: $\textsc{Doc}[t]$

3.2 Mapping of Entity Groups to Documents

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

Given the restriction

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

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

Example 25. Using the schema graph...

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

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

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

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

To do (1)

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.

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

Figure 3.1: Example entity group

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

(a) Course

Field	Terms
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

For every $t \in \mathrm{V}(T)$, we construct a document $\textsc{Doc}[t]$ (Section 3.1 on page 21). 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 6 would be as follows

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

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

By Claim 1 on the preceding page, 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.

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.

$\mathsf{query}(q)$ 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

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

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".

Chapter 4

Along Came Clojure

Talk about how great Clojure is.

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 following 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.

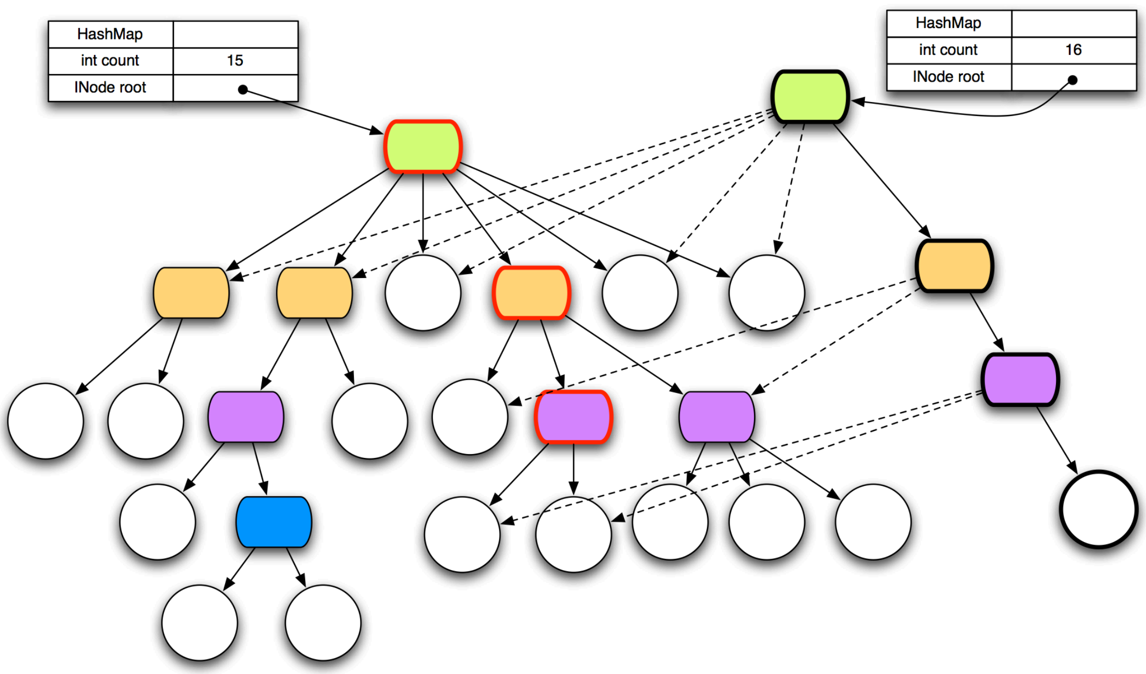

Figure 4.1: Representation of how data structures are "changed" in Clojure (Source: [Hic09])

Concurrency

Clojure supports four systems for concurrency: software transactional memory (STM), agents, atoms, and dynamic vars. The differences between these systems are summarized in Table 4.1.

To do (2)

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: Syntactic sugar for JVM interoperability

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

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

Interoperability With the JVM

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

The 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).

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

Figure 4.3: `IConfig` protocol all configurations must adhere to

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]

4.2.2 Indexing Relational Database

The process of indexing a relational database is a multi-step one. It begins with the declaration of the database connection information, the path to the index, and the schema definition.

In our implementation, this information is specified by a record that adheres to the protocol in Fig. 4.3. The record which defines the Mycampus dataset uses SQLite for its database engine, so it accepts two strings; one specifies the path to the database file, while the other specifies the path to the index.

The first component, `connection`, returns a JDBC-compatible object. The second component, `schema`, returns a list of `EntitySchema` records. The `EntitySchema` record is defined in Section 4.2.2 on the following page. The final component, `index`, specifies the path to the index.

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 3)	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

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

The `EntitySchema` records contain not only the information required to construct them, but also the required behaviour. Every record, given the database and index connections, is capable of retrieving the set of all named tuples it represents in the database. It iterates through every tuple and constructs a document for each tuple, as well as any value documents, if applicable.

Indexing Process

With the database, index, and schema graph defined, the system is able to transform the data from the relational model into the document model.

The first step is to retrieve the list of named tuples from the database. Each `EntitySchema` record is iterated over; the SQL defined for each record is executed against the database. A function is exe-

Attribute	Value
code	CDPS 101
title	Human-Mutant Relations
subject	CDPS

(a) Tuple

Key	Value
`:type`	`:entity`
`:class`	`:course`
`:id`	`“course|cdps_101”`
`:code`	`“CDPS 101”`
`:title`	`“Human-Mutant Relations”`
`:subject`	`“CDPS”`

(b) Internal Representation

Field	Terms
__type__	:entity
__class__	:course
__id__	course|cdps_101
__all__	“cdps 101 human mutant relations cdps”
code	{cdps, 101}
title	{human, mutant, relations}
subject	{cdps}

(c) Document

Table 4.4: Transformation from tuple to internal representation to document

cuted on every tuple that is retrieved.

The second step is to transform these tuples into the internal representation. The function executed on every tuple is responsible for this conversion.

The third step is to convert the internal representation into a document. Again this is performed by the function operating on every tuple.

Finally, every document is indexed.

4.2.3 Indexing of relational objects (5 days, week 5)

- Fuzzy indexing of values (typed by classes)

4.2.4 Keyword Search in document space (5 days, week 6)

- Disambiguate keywords using fuzzy search (suggestion, overloaded terms)

- Flexibility keyword search for documents
- Translate search result back to relational space

4.2.5 Graph Search in document space (5 days, week 7)

- 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

5.1 Implementation

- Choice of language
- Statistics about the code base: LOC, classes, ?
- Github hosted

5.2 The data set

- Description of the data set
- Statistics of the data set

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 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 GC after testing to estimate impact of cleanup on the timing results

Unfortunately this requires a much longer runtime as each function must be invoked numerous times. In extreme cases (Ford-Fulkerson, 8 hops) this can take upwards of 4 hours in our test environment.

Data Collection

Criterium provides us with a Clojure map with performance data. It performs analysis, presenting us with outliers, samples, etc. As this data collection process can take several hours or more, this data is collected and stored for offline analysis.

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 Python captures the output, JSONifies it, and stores it in an array. This array is written to disk in JSON as well so it can be loaded into the data transformation script.

For example:

```
[{”results”: {...}, ”method”: ”bfs”, ”max-hops”: 1}, ...]
```

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

Data Processing

Several scientific computing libraries are used in the processing and visualization.

There are two forms the data takes:

- comma-separated values (CSV)
- JavaScript object notation (JSON)

The CSV data is generated from the JSON data which is generated as described in Section 5.3.1 on the previous page.

With the data loaded, we're interested in a handful of pieces of date per each entry.

- Max hops
- Method
- Mean execution time

We can easily load and parse the JSON data.

- Index speed
- Keyword search speed
- Graph search speed:
  - Ford Fulkerson
  - BFS
  - Concurrent BFS using refs
  - Concurrent BFS using atoms

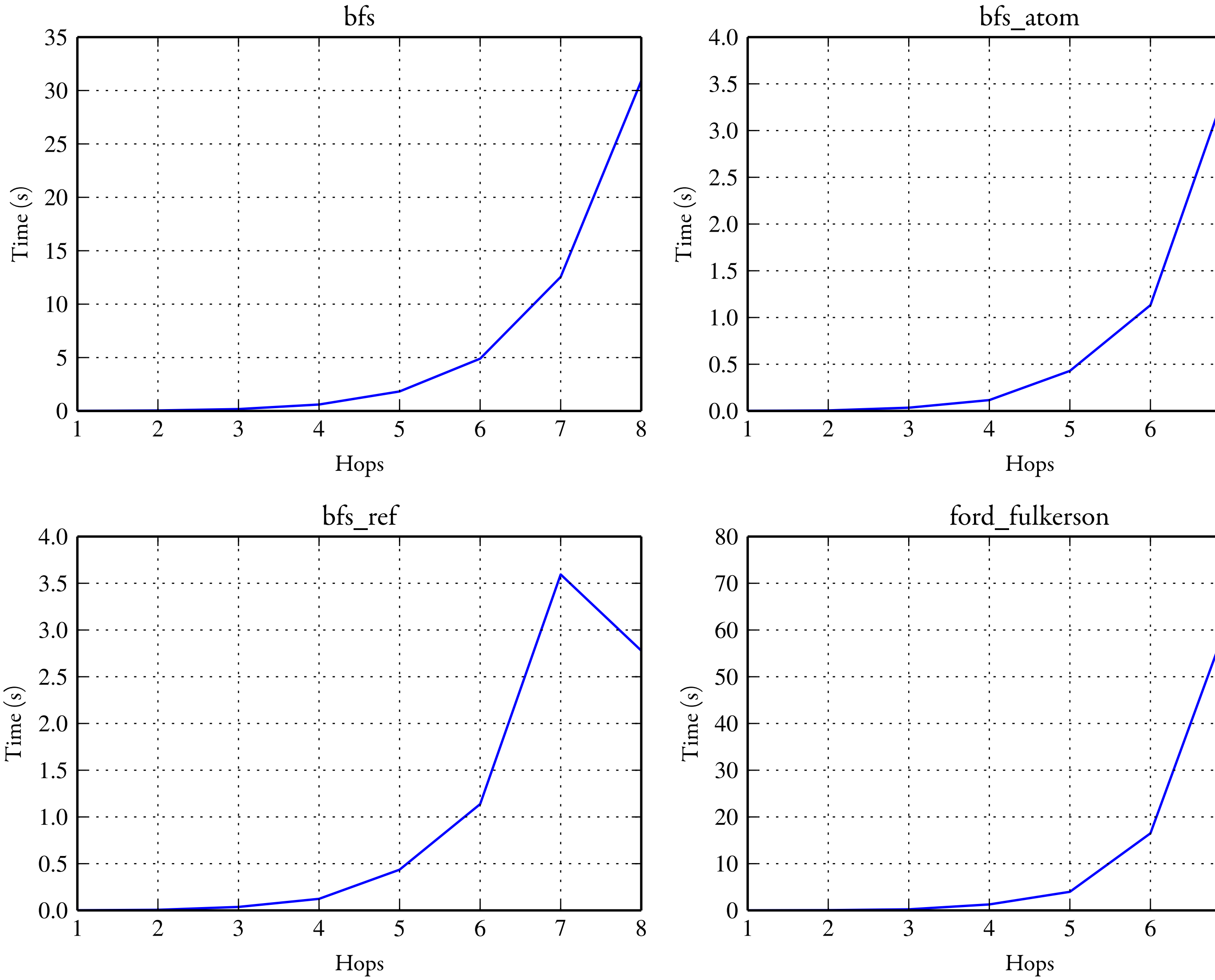

Figure 5.1: Growth of graph search times based on number of hops, plotted separately

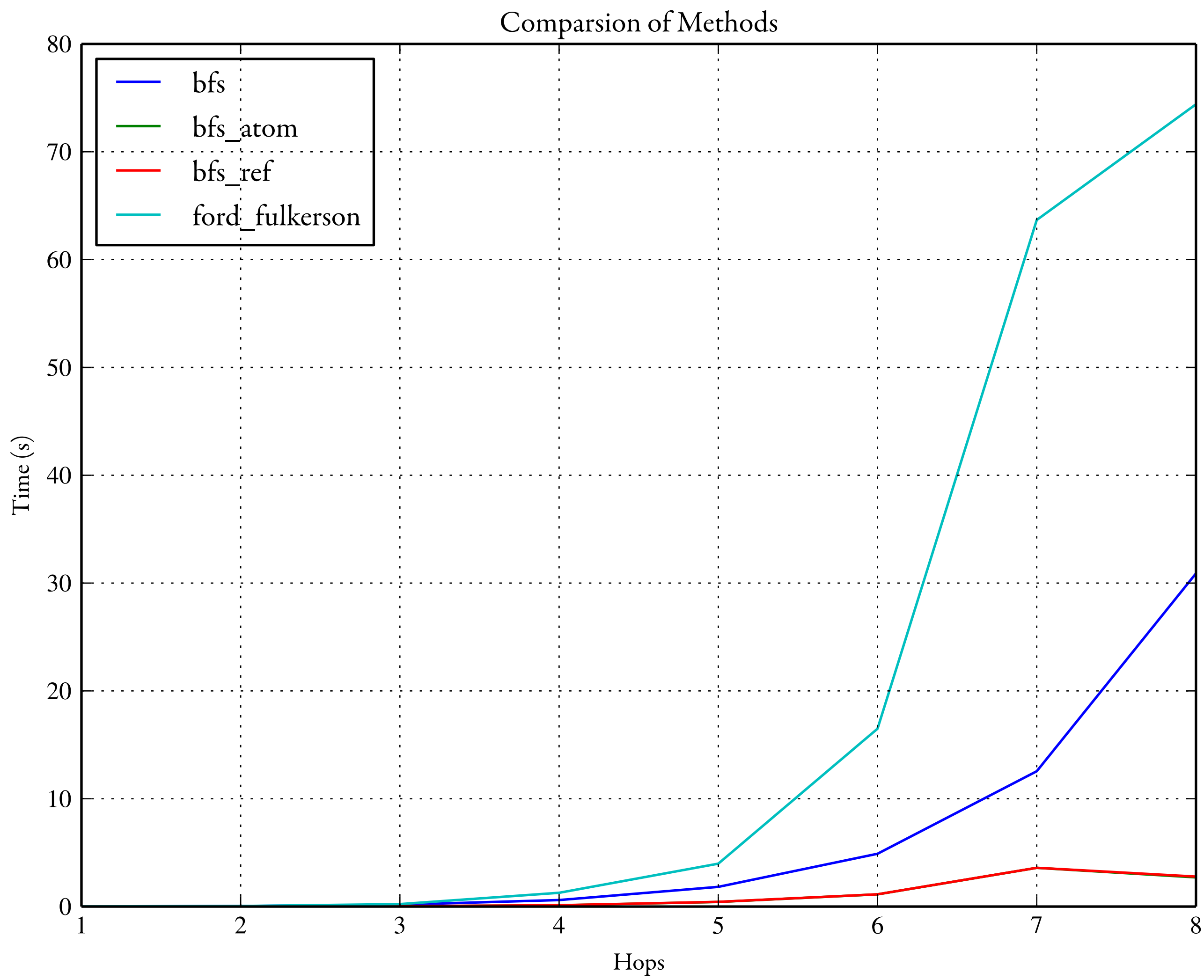

Figure 5.2: Growth of graph search times based on number of hops, combined plot

5.4 Lessons learned

- Simple algorithms are easier to parallelize
- STM is effective: transactions do not rollback (that much), so we observe impressive speed-up in concurrent versions.
- Fine tuning is beneficial: atom is better than ref.
- The clojure way: correctness first, runtime optimization latter (ref to atom is natural).

Chapter 6

Conclusion (0 days)

Survived Clojure.

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

```
(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. %)]
       [”--index”       ”Build an index of the database”
        :default false
        :flag true]
       [”--benchmark”   ”Run benchmarks”
```

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

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

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

```clojure
(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

```
(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
      :ID     :id
```

```
      :attrs  [:id :name]
      :values [:id :name]})
   (EntitySchema.
     {:T      :entity
      :C      :section
      :sql    Section
      :ID     :crn
      :attrs  [:crn :reg_start :reg_end :credits
               :section_num :levels]
      :values [:crn]})
   (EntitySchema.
     {:T      :entity
      :C      :schedule
      :sql    Schedule
      :ID     :id
      :attrs  [:days :sch_type :date_start :date_end
               :time_start :time_end :week]
      :values []})
   (EntitySchema.
     {:T      :group
      :C      "Instructor schedule"
      :sql    (->
                (select* Schedule)
                (with Instructor))
      :ID     [[:instructor :instructor_id  "Instructor ID"]
               [:schedule   :id             "Schedule ID"]]
      :attrs  []
      :values []})
   (EntitySchema.
     {:T      :group
      :C      "Course schedule"
      :sql    (->
                (select* Schedule)
                (with Section
                      (with Course)))
      :ID     [[:section  :crn  "CRN"]
               [:course   :code "Code"]
               [:schedule :id   "Schedule ID"]]
      :attrs  []
      :values []})
   (EntitySchema.
     {:T      :group
      :C      "Schedule location"
      :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

A.3.1 molly.datatypes.database

```
(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

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

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

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

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

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

(defn row->data
  ^{:doc "Transforms a row into the internal representation."}
```

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

(defn doc->data
  ^{:doc ”Transforms a Document into the internal representation.”}
  [this]
  (let [fields        (.getFields this)
        extract       (fn [x] [(keyword (clojure.string/replace
                                          (.name x) ”_” ””))
                               (.stringValue x)])
        check-special (fn [x] (special? (.name x)))
        filter-fn     (fn [f] (apply hash-map
                                     (flatten
                                       (map extract
                                            (filter f fields)))))]
    (with-meta (filter-fn (fn [x] (not (check-special x))))
               (filter-fn check-special))))
```

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

A.3.3 molly.datatypes.schema

```
(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))
  (schema-map
    [this]
    S))
```

A.4 molly.index

A.4.1 molly.index.build

```
(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

```
(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

```
(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)))

(defn add-doc
```

```
  [idx doc]
  (.addDocument idx doc))
```

A.6.2 molly.search.query_builder

```
(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

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

```
(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)))
```

A.7.3 molly.server.search

```
(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)
    (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.7.4 molly.server.util

```
(ns molly.server.util)

(println "loading properties")
```

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

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

```
    (assoc state
           :Q       (if done
                      Q
                      (conj Q v))
           :marked  (conj marked u v)
           :dist    (assoc dist v (inc (dist u)))
           :prev    (assoc prev v u)
           :done    done)))

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

A.8.2 molly.algo.bfs

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

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

(defn bfs
  [G s t max-hops]
  (loop [Q      (conj (clojure.lang.PersistentQueue/EMPTY) s)
         marked #{s}
         dist   {s 0}
         prev   {s nil}]
    ; Terminate when nothing is left to explore.
    (if (seq Q)
      (let [u   (first Q)
            Q'  (rest Q)]
        ; Terminate when the target is found, or we reach a limit.
        (if (or (= u t)
                (>= (dist u) max-hops))
          [marked dist prev]
          (let [[marked' dist' prev' frontier]
                (update-adj G marked dist prev u)]
            (recur (concat Q' frontier) marked' dist' prev'))))
      [marked dist prev])))
```

A.8.3 molly.algo.bfs_atom

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

(defn update-adj
  [G state-ref u max-hops]
  ;(if (or (empty? (
  (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))]
    (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')
        ;(if (some (fn [node] (= node t)) (@state-ref :marked))
        ;  (swap! state-ref assoc :done true)
          (update-adj state-ref G u max-hops)));)
    [(@state-ref :marked) (@state-ref :dist) (@state-ref :prev)]))
```

A.8.4 molly.algo.bfs_ref

```
(ns molly.algo.bfs-ref
  (:use clojure.pprint)
  (: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))]
    (while (and (not (empty? (@state-ref :Q)))
                (not (@state-ref :done)))
      (let [u   (first  (@state-ref :Q))
            Q'  (rest   (@state-ref :Q))]
        (dosync (alter state-ref assoc :Q Q'))
        (if (some (fn [node] (= node t)) (@state-ref :marked))
          (dosync (alter state-ref assoc :done true))
          (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}))))
```

Bibliography

[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.

[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.

[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.

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

[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://doi.acm.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...

- ☐ 1 (p. 22): Above proof could use some love
- ☐ 2 (p. 27): More concurrency