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 (MSc)

in

Faculty of Science

Computer Science

University of Ontario Institute of Technology

Supervisor: Dr. Ken Q. Pu

December 2013

Abstract

Traditional relational database systems offer powerful data modelling and querying capabilities. Unfortunately a relational database does not permit users to perform natural (keyword) queries. We present a system for automatic mapping of a relational schema into a document schema in order to facilitate fast and powerful full-text (keyword) search. By providing a facility for users to conduct keyword searches, we improve accessibility and decrease search time complexity. In addition, we explore further improvements to time complexity by utilizing concurrent implementations of our search algorithms.

Keywords relational database; full-text search

Preface

Background and Motivation

The introduction of keyword search has revolutionized how we find information. Over the years, numerous techniques have been developed which make searching through large amounts of information for one or more keywords extremely fast. With the advent of faster computer hardware, we are able to not only search through small attributes of a document (eg. Title, synopsis, etc.) but rather the entirety of the document itself.

An example of an early system which utilized keyword search would be a library catalogue. Such a system would allow a user to search, by keyword, for the title of a book, manuscript, etc. The results would show item titles matching the keyword(s), as well as other information such as whether or not the item is in circulation, as well as where it is located within the library. This information would come from a relational database.

With the rise of the World Wide Web, much information was placed online. This information would be easy to access if one knew how to locate it. Unfortunately, over time, so much information existed on the World Wide Web that it became difficult to keep track of it all. There was a need to index all of this information and make it accessible. This need was filled by a Web search engine.

Initial Web search engines comprised of simple scripts that gathered listings of files on FTP servers; they were essentially link farms. A few short years later, the first full-text (keyword) search engine, WebCrawler, was released.

While full-text search engines provided an excellent means for locating information, as the

Web grew larger, the volume of noise also grew larger. In addition, every Web page could be structured in a different way; the Web was largely a collection of unstructured documents. That is, there were few obvious links between them.

Search engines such as Google attempted to solve this problem by introducing new algorithms, such as PageRank, to rank Web pages on both the relevance of their content as well as their reputation. The idea was if a page is linked to often, it is considered to be more authoritative on a subject than a page with fewer links. This allowed the relevance of a page to be computed based on not only its contents, but its artificial importance.

Molly attempts to avoid some of the issues plaguing search engines. It deals primarily with structured, filtered data. This allows us to provide results with less noise. In addition, the fact that it deals with structured data means links between documents are explicitly stated. Rather than inferring a link between documents based on hyperlinks, we know when two documents are linked together.

This thesis provides an overview of the Molly system.

Outline

Changed.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Background (2 days)

Literature search on:

- DBExplore
- XRank
- BANKS
- ...

Chapter 2

A Tale of Two Data Models

2.1 Relational model with star schema

2.1.1 Relational Model of Data

$\langle DB \rangle ::= \langle name \rangle$

Definition 1. Database

Let d be a database instance. A database is comprised of three main components:

- NAME[d] : string
- REL[d] : list(REL)
- FK[d] : list(FK)

Definition 2. Relation

Let $r \in$ REL[d], where d is defined in Definition 1. A relation is comprised of three main components:

- NAME[r] : string
- ATTR[r] : list(ATTR)
- KEY[r] : list(ATTR)

The first is a `string` representation of the relation. The second is a list of attributes that make up entries, or tuples, in the relation. The third is a list of the relation's attributes that uniquely identify the tuple within the relation. That is, $\text{KEY}[r] \subseteq \text{ATTR}[r]$.

Definition 3. Attribute

Let $a \in \text{ATTR}[r]$, where r is defined in Definition 2. An attribute is comprised of two main components:

- $\text{NAME}[a]$
- $\text{TYPE}[a]$

Note: Lower case letters (e.g. $a, b, c, \ldots$) are attributes.

Definition 4. Foreign Key

Let $\theta \in \text{FK}[d]$ be a FK instance,where d is defined in Definition 1. A foreign key is comprised of two main components:

- $\text{FROM}[fk] = (\text{REL}_S[\theta], \text{ATTR}_S[\theta])$
- $\text{TO}[fk] = (\text{REL}_t[\theta], \text{ATTR}_t[\theta])$

Note: θ, ϕ are the FK constraints

2.1.2 Star Join Schema to Form Entity Groups

A network (forest) of tuples, jointed via some existing $\theta \in \text{FK}[d]$.

The schema of entity group G is defined as vertices of G:

$$V(G) \subseteq \text{REL}(DB)$$

Relation r in the space of vertices of G, $V(G)$, may be a table or a computed view.

$$r \in V(G)$$

It is also defined as the edges of G, or $E(G)$, in the form of

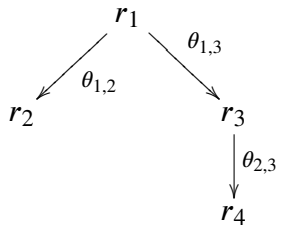
$$r(a_1, a_2, \ldots, a_k) \rightarrow s(b_1, b_2, \ldots, b_k)$$

where $a_i \in \text{ATTR}[r]$, $b_i \in \text{ATTR}[s]$, with the additional constraint of $r, s \in V(G)$.

Example 1.

$$\begin{aligned} Instructor(name) &\rightarrow Schedule(instructor) \\ Schedule(code) &\rightarrow Course(id) \end{aligned}$$

2.1.3 Instances of an Entity Group



Instances are obtained by the following process.

For $r_i(a_{i,1}, ai, 2, \ldots, a_{i,k}) \rightarrow r_j(b_{j,1}, b_{j,2}, \ldots, bj, k)$

$$c_{ii,j} = \bigwedge_{n=1}^{k} (a_{i,n} = b_{j,n})$$

$$\begin{aligned} \text{VIEW}[G] &= \bowtie_{\theta_{ij}} (r_i, r_j) \\ &= r_1 \bowtie_{\theta_{1,2}} r_2 \bowtie_{\theta_{2,3}} r_3 \ldots \bowtie_{\theta_{n,n+1}} r_n \end{aligned}$$

where $r_1, r_2, r_3, \ldots, r_n$ are relations discovered by a depth-first search traversal of G.

Each tuple in $\text{VIEW}[G]$ is an instance of entity group G.

Motivation:

To do (1)

Database schema

Vertices: REL[d]

Edges: FK[d]

The entity graphs are overlapping subgraphs at the schema level.

Question:

How to determine connectivity at the instance level?

- ER style relational schema
- Star join schema to form entity groups
- Expressing relational objects using universal design pattern (describing data using scalar, lists and dictionaries).
- Relational object graph

2.2 Pros and Cons of the Relational Model

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

2.2.1 Pros

- Well supported by relational algebra and relational databases (RDBMS)
- Clean and consistent database instances (ACID?)
- Can use queries to resolve instance-level connectivity
  - How is "Ken" connected to "CSCI 3030U"?

To do (2)

2.2.2 Cons

- Must know the relational schema
    1. Know table/attribute names
    2. Know join paths (schema)
- Inflexible string matching options (basically just have `LIKE`), substring matching
- Must know SQL
- All queries must be re-written upon schema changes (rename, change in join path, etc.)
- Not adaptive to new join path (e.g. newly created entity group, deleted E.G. etc.)
- Good for analytics (aggregation, selection) if user has domain knowledge of the schema.
- Bad for exploratory queries.
- Bad if user doesn't know SQL
- Bad for flexibility

2.3 Document model (4 days, week 2)

- Definition: documents, terms, and the bag of terms model for documents and queries
- Definition of keyword search queries: vectorization of documents (tf-idf) and queries. Models of distance between documents and queries (cosine-distance, jaccard distance, BM25).
- Extended document model with attributes and fields
- Expressing documents in the universal design pattern (aka list+dict)
- Document graph

2.4 Pro and con of document model (1 day)

- Good: exploratory queries using keywords (google)
- Good: easy (or no) syntax
- Good: fuzzy matching (using n-gram)
- Bad: No analytics

2.5 Best of both worlds (4 days, week 3)

- Hybrid database defined by both the relational model and the document model
- Translation between relational objects (entities and entity) groups to documents.
- Translation of documents back to relational objects.
- Proof of lossless translation between relational space and document space

Chapter 3

Along came Clojure

3.1 Basic principles of functional programming (2 days)

- immutable data structures
- persistent data structures using multi-versioning
- functions (and higher order functions) as values

3.2 Features of Clojure (2 days)

- Data structures supporting the universal design pattern
- Concurrency + STM
- Interoperability with JVM (including Lucene)

Chapter 4

Search w/ Clojure

4.1 Thirdparty libraries (1 day, week 4)

- Lucene

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

- Schema definition
- Crawling using SQL
- Indexing using relational objects
- Fuzzy indexing of values (typed by classes)

4.3 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

4.4 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 days, week 8)

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

- Index speed
- Keyword search speed
- Graph search speed:
  - Ford Fulkerson

  - BFS
  - Concurrent BFS using refs
  - Concurrent BFS using atoms

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.

Chapter 7

Definitions

7.1 Data Representation & Notation

The data from the database is represented in various data structures. There are separate representations for each type of data: values, entities, and entity groups.

Value

Definition 5. A **Value** represents a single piece of information. To avoid repetition, each value is unique. That is, $\exists!\ v \in V$, where v is a value in the set V of all values.

Entity

Definition 6. An **Entity** is a collection of attributes, a_n, each mapped to a single value, v_n. An entity also includes additional information such as a unique identifier.

id	$T_n \vert v_{id}$
a_1	v_1
a_2	v_2
$\vdots$	$\vdots$
a_n	v_n

Figure 7.1: The structure of an entity

Entities are analogous to rows in a database table. Thus, the unique identifier is generated based on the table name, T_n, as well as unique key in the table, v_{id}. The unique key identifies the row, and the table name identifies the table. Together they uniquely identify the entity within the entire database. Attributes are analogous to columns in a database table.

$\exists!\, e_{id} \in E$, where E is the set of all entities.

Entity Group

Definition 7. An **Entity Group** joins together two or more entities. These entity groups can also have attributes, a_n, and values, v_n, associated with them much like entities.

$$\begin{array}{ll} e_L & [e_1, e_2, \ldots, e_n] \\ a_1 & v_1 \\ a_2 & v_2 \\ \vdots & \vdots \\ a_n & v_n \end{array}$$

Figure 7.2: The structure of an entity group

Chapter 8

RDBMS to Document Store

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

Figure 8.1: Determining whether or not a field is classified as "special."

```
(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)))))
```

Figure 8.2: Unique identifier generation.

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

Figure 8.3: Creation of a field.

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

Figure 8.4: Creation of a document.

```
(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"))))))
```

Figure 8.5: Transformation of a row into the internal representation.

```
(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))))
```

Figure 8.6: Transformation of a document into the internal representation.

```
(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   []))]
```

Figure 8.7: Transformation of the internal representation into a document.

Chapter 9

Concurrency

```
(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))))
```

Figure 9.1: Retrieves the entity with the given ID.

```
(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}"))))
```

Figure 9.2: Finds all groups containing an entity with the given ID.

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

Figure 9.3: Finds adjacent nodes in the graph (entities linked in groups).

```clojure
(defn update-adj
  [G marked dist prev u max-hops]
  (loop [adj      (find-adj G u)
         marked   marked
         dist     dist
         prev     prev
         frontier []]
    (if (or (empty? adj) (>= (dist u) max-hops))
      [(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)]
```

Figure 9.4: Discovers new nodes along the frontier and updates the state accordingly.

```clojure

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

Figure 9.5: BFS using recursion.

```
(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
```

Figure 9.6: Discovers new nodes along the frontier and updates the state accordingly.

```
                                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 (not (empty? (@state-ref :Q)))
                (not (@state-ref :done)))
```

Figure 9.7: Implements a concurrent version of BFS using atoms.

```
(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
                                            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'  (pop (@state-ref :Q))]
        (dosync (alter state-ref assoc :Q Q'))
        (if (some (fn [node] (= node t)) (@state-ref :marked))
```

Figure 9.8: Discovers new nodes along the frontier and updates the state accordingly.

Figure 9.9: Implements a concurrent version of BFS using references.

Chapter 10

Theoretical

The problem of efficiently searching through a graph is the subject of countless articles and journal papers. Numerous algorithms have been proposed to perform graph search in an efficient manner. Many of these algorithms build upon their predecessors, making assumptions and changing aspects to better suit a particular problem area.

The applications of efficient graph search algorithms are endless. Algorithms such as Dijkstra's are used heavily in path finding, for example in a portable GPS unit. A* Search is used in the area of computer vision to approximate the best path to take. Companies such as Amazon make use of graph search algorithms in order to find related products for consumers to purchase.

This thesis concentrates on building a system which can be utilized in order to discover related information. The system that was built is capable of finding not only related information from neighbouring nodes, but also the best path between two arbitrary nodes.

Section 10.1 provides an overview of how the data is represented in the system. It also defines notation to represent this data. Section 10.2 outlines how the relational database is related to the data in the system. Section 10.3 provides an example data corpus which is utilized throughout this thesis. It details the "mycampus" dataset. Finally, Section 10.4 provides justification for the algorithm chosen to perform the graph search. It discusses multiple different graph search algorithms, as well as why the one used in this thesis was chosen.

10.1 Data Representation & Notation

The data from the database is represented in various data structures. There are separate representations for each type of data: values, entities, and entity groups.

Value

Definition 8. A **Value** represents a single piece of information. To avoid repetition, each value is unique. That is, $\exists!\ v \in V$, where v is a value in the set V of all values.

Entity

Definition 9. An **Entity** is a collection of attributes, a_n, each mapped to a single value, v_n. An entity also includes additional information such as a unique identifier.

id	$T_n \vert v_{id}$
a_1	v_1
a_2	v_2
$\vdots$	$\vdots$
a_n	v_n

Figure 10.1: The structure of an entity

Entities are analogous to rows in a database table. Thus, the unique identifier is generated based on the table name, T_n, as well as unique key in the table, v_{id}. The unique key identifies the row, and the table name identifies the table. Together they uniquely identify the entity within the entire database. Attributes are analogous to columns in a database table.

$\exists!\ e_{id} \in E$, where E is the set of all entities.

Entity Group

Definition 10. An **Entity Group** joins together two or more entities. These entity groups can also have attributes, a_n, and values, v_n, associated with them much like entities.

$$
\begin{array}{ll}
e_L & [e_1, e_2, \ldots, e_n] \\
a_1 & v_1 \\
a_2 & v_2 \\
\vdots & \vdots \\
a_n & v_n
\end{array}
$$

Figure 10.2: The structure of an entity group

10.2 Relational Database Abstraction

10.3 Data Corpus

For illustrative purposes, the mycampus dataset will be used. This data comes from UOIT's course registration system.

There are several different entities which comprise the mycampus dataset:

- Courses
- Instructors
- Schedules
- Sections
- Teaches

The **Courses** entity represents a course. A course has an individual code, along with a title and a description (See Table 10.1). The code uniquely identifies the course.

Column	Type	Description
code	VARCHAR	Unique course code
title	VARCHAR	Title of the course
description	TEXT	A brief description

Table 10.1: Courses entity schema

The **Instructors** entity represents an individual instructor. Each instructor has a unique identifier, as well as a name (See Table 10.2). An instructor can be a Professor, Lecturer, Sessional Instructor, or Teaching Assistant.

Column	Type	Description
id	INT	Unique identifier
name	VARCHAR	The instructor's name

Table 10.2: Instructors entity schema

The **Sections** entity represents a section of a course. Courses may have many sections. For example, one section could be a lecture, while another is a lab. Some courses may have over a dozen sections, depending on the associated term.

Each section contains a unique identifier, capacity information (students enrolled, spots open, etc.), when registration is open for the section, how many credits it is worth, what level (eg. undergraduate or graduate), and what year the section is offered in (See Table 10.3).

Column	Type	Description
id	INT	Unique identifier
actual	INT	Number of people enrolled in the course
campus	VARCHAR	String uniquely identifying the campus
capacity	INT	Maximum number of people that may be enrolled in the section
credits	FLOAT	Number of credits awarded upon successful completion
levels	VARCHAR	The level of the course (eg. undergraduate, graduate, etc.)
registration_start	DATE	Date registration for the section opens
registration_end	DATE	Date registration for the section ends
semester	VARCHAR	String that uniquely identifies the semester
sec_code	INT	Unique section code (called a CRN)
sec_number	INT	Sequential number identifying the number of the section
year	INT	The year the section is offered in

Table 10.3: Sections entity schema

The **Schedules** entity represents a scheduled meeting of a section. A section may have many schedules. For example, a lecture section may meet twice a week.

Each schedule has a unique identifier, a date range in which the schedule is active, the time of the schedule, the type, location, and the day which the class takes place (See Table 10.4).

Column	Type	Description
id	INT	Unique identifier
date_start	DATE	First day of class
date_end	DATE	Last day of class
day	VARCHAR	Single character representing the day of the week
schedtype	VARCHAR	Lecture, tutorial, lab, etc.
hour_start	INT	Hour the class starts at
hour_end	INT	Hour the class ends at
min_start	INT	Minute the class starts at
min_end	INT	Minute the class ends at
classtype	VARCHAR	The type of class
location	VARCHAR	Unique location name where class is held

Table 10.4: Schedules entity schema

A **Teaches** entity is used to link together an instructor and a schedule. Each link has a unique identifier along with the instructor's position (eg. Teaching Assistant, Lecturer, etc.) (See Table 10.5).

Column	Type	Description
id	INT	Unique identifier
position	VARCHAR	Position of the Instructor with regard to a Schedule

Table 10.5: Teaches entity schema

10.4 Graph Search Algorithms

Careful consideration was given to which graph search algorithm was to be used. Among the choices were:

- Breadth-First
- Bellman-Ford
- Dijkstra
- A*

The first choice, Breadth-First Search, is among the simplest of the algorithms. It has the advantage of being simple to implement. It can only handle fixed costs for travelling between nodes, which can be a disadvantage.

Bellman-Ford is similar to BFS. It has the ability to deal with variable cost. This comes at the cost of increased difficulty in implementation. When the cost between nodes is fixed, Bellman-Ford essentially becomes BFS.

A greedy version of BFS is Dijkstra's Algorithm. It utilizes a priority queue rather than a regular queue, allowing it to be faster. As it is a greedy algorithm, Dijkstra's Algorithm may not return the optimal result.

An extension to Dijkstra's is A* search. Graph search spaces can be rather large. A* attempts to prune the search space based on a heuristic. A* is a natural choice to perform graph search in certain areas such as computer vision where an obvious heuristic exists. It has a disadvantage of consuming large amounts of memory (though IDA* attempts to limit memory consumption).

There are many more graph search algorithms. The above were primarily considered as many of the other algorithms are simple extensions with different data structures.

For this application, there is no obvious heuristic. This eliminates A*. There is also no obvious cost function. This eliminates Dijkstra's Algorithm. Bellman-Ford is very similar to BFS with the added ability to deal with variable cost. As the cost function is not obvious and thus constant, Bellman-Ford essentially reverts back to BFS.

For these reasons, BFS was chosen as the graph search algorithm. While the other candidates and others provide numerous advantages over BFS in many situations, this is not one of them.

Chapter 11

Implementation

11.1 Functional vs. Procedural Programming of Algorithms

Functional and procedural programming languages are entirely different paradigms. In functional programming, data is immutable. Functions must be pure and predictable (free of "side effects"). Loops and control flow are accomplished with recursion and pattern matching.

Procedural languages allow for more flexibility at a cost of unpredictability.

Key differences between FP and procedural (for algorithms)

11.1.1 Functional Data Structures

Clojure immutable data structures, versioning (persistent data structures), STM vs. locking

11.2 Tuneable Parameters

Tuneable parameters

Chapter 12

System Implementation

12.1 Technology Selection

12.1.1 Programming Language

Numerous programming languages were considered for the implementation. Among them were: Ruby, Python, and Clojure. Each of the languages presented both pros and cons. Ruby and Python are rather similar object oriented languages. Clojure is a functional language.

Ruby features a clean syntax and is very object oriented. Its object model was inspired by that of Smalltalk. It has a very active web development community and numerous web frameworks (eg. Rails, Sinatra, Padrino, etc.). In contrast to Python, it features a lean core, instead choosing to depend on third party libraries.

Python also features clean syntax and is object oriented. It embraces a "batteries included" philosophy to its standard library, featuring libraries for everything from serial communications to importing/exporting CSV files. It too has a strong web development community and numerous web frameworks (eg. Django, Flask, web2py, etc.).

Python is used extensively by scientists. As such, it has numerous plotting, computation, and simulation libraries. Examples include Matplotlib, SciPy, NumPy, and NetworkX.

Both of the above languages embrace functional programming elements. Python has filter, reduce, and map. Ruby's collections are also capable of performing filter (select), reduce (inject),

and map. They both lack an important aspect of functional programming: immutable objects.

Clojure is a purely functional language built on top of the JVM. Like Python and Ruby, it is dynamically typed. Unlike Python and Ruby, it is a Lisp dialect and as such features macros and code-as-data.

As it runs on the JVM, Clojure allows us to utilize existing Java languages. Using Java libraries through Python or Ruby requires using their respective native code interfaces to Java's JNI. This process is typically automated through a tool such as SWIG.

Clojure was chosen as the implementation language for a number of reasons. First of all, as it is built on top of the JVM, it can utilize JDBC for database access. It can also make use of several other useful libraries. Secondly, its dynamic Lisp nature is a natural way of processing data. Clojure allows us to model our data structures directly after the data. In a language such as Python, one would need to make use of classes.

12.1.2 Relational Database

Several relational database management engines (RDBMS) were evaluated for this project. Key requirements were SQL support, and JDBC driver availability.

SQL, or Structured Query Language, is a powerful way to make complex queries. It is loosely based on Relational Algebra (RA)[3, p. 243]. In addition to being a data-manipulation language (similar to Relational Algebra), it is also a data-definition language.

JDBC, or Java Database Connectivity, was required in order for our application to access the database. JDBC is an API which allows us to access a database using a standard interface. It is the responsibility of the JDBC Driver to translate the API calls into the correct syntax for the database engine.

Any RDBMS that supports SQL-92 or higher and has a JDBC driver for it should be usable in this system. For testing purposes, we chose to use SQLite. SQLite provided a SQL interface, had a stable JDBC driver, and is ubiquitous. SQLite is an embedded database, allowing for local development without running a separate daemon process.

12.1.3 Full-Text Search Database

Two full-featured full-text search databases (FTSDB) were evaluated. The first, Xapian, is written as a C++ library. It features a rich query language, high performance, and a scalable design[5]. Bindings for Java are made available via a simple JNI library.

The second, Lucene, is a pure-Java library. It too features a rich query language and reasonable performance. Lucene also forms the basis of the Apache Solr search platform. As it is written in Java, Lucene runs natively on the JVM.

Both choices provided plenty of features, Java interfaces, and excellent performance. Ultimately Lucene was chosen. As it ran on the JVM, interfacing with the Lucene library from our Clojure code was simple. The fact that it was written in pure-Java and required no native libraries meant the entire project could be bundled as a Java Archive (JAR).

12.1.4 Web Stack

Clojure is a young language. As such, few web frameworks exist specifically for it (it can, however, interface with any Java web framework).

Clojure web frameworks are built on top of Ring, itself an HTTP abstraction layer with adapters for Java servlets as well as the Jetty web server. Ring is similar to WSGI in Python, or Rack in Ruby.

The web framework chosen for this project was Noir. Noir provided a simple Clojure API for building the JSON API to allow for client-side querying of the project's API.

12.2 System Design

The system is split into numerous components.

A configuration file is used to describe the entities and entity groups. It defines the relationship between entities, as well as their attributes and values. This configuration file is used by the crawler and indexer in order to retrieve data from the relational database and index it in a full-text search database.

Once the index is built, another component provides an internal API for querying the full-text search database. It is a thin Clojure wrapper over the Lucene API which provides the basis for search and retrieval of results. This wrapper is utilized by the system to discover relationships among entities.

Finally, the system can be interfaced with via a JSON API. The JSON API provides a simple HTTP wrapper around the search and discovery features of the system. The use of JSON allows client-side JavaScript to send and receive queries with XMLHttpRequest.

To do (3)

12.2.1 Configuration

Configuration files are simply Clojure code that follow a well-defined protocol. The protocol defines three main components of the system:

`connection` The relational database connection. It must be an instance of the `Database` protocol.

`index` The path to the full-text search database index file.

`schema` A definition of the database schema. The definition includes both the definition of entities, as well as the definition of entity groups. The schema is a list of instances of the `Schema` protocol.

`Database` Protocol

The `Database` protocol is defined as follows.

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

Each database instance must provide a function which accepts a query and function as arguments. It must execute the query provided and call function `f` on each row returned.

`Schema` Protocol

The `Schema` protocol is defined as follows.

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

A schema consists of the following information:

- A symbol (typically the name of the table).
- A hash map containing the structure of the schema.

Schemas are also responsible for populating the full-text search index with data from the relational database.

A concrete implementation for this protocol is provided by the `EntitySchema` type.

12.3 Implementation Issues

Chapter 13

Performance & Evaluation

13.1 Environment

All benchmarks were run on the same machine using the same software versions. The machine has the following specs.

Item	Description
Model	Dell PowerEdge 2950
Processor	2 x Quad-Core Intel® Xeon™ 5300 3.0GHz
Memory	8 GB DDR2 ECC RAM 667 MHz

Along with the following software versions.

Item	Description
Operating System	GNU/Linux Ubuntu 10.04.4 LTS
Kernel	2.6.32-28-generic
Java™ SE Runtime Environment	1.6.0_26-b03
Java HotSpot™ 64-Bit Server VM	20.1b02

13.2 Methodology

Evaluating the performance of code is a difficult problem at best. It is difficult to determine the impact on performance of various uncontrollable factors. Virtual Machines add another layer of abstraction which introduces even more factors.

As this project runs a top of the Java Virtual Machine (JVM), there are many factors to consider with regard to performance. In addition to uncontrollable events such as garbage collection, the JVM's behaviour can differ based on the operating system (OS) it is running on.

13.2.1 Pitfalls of the Java Virtual Machine

Reliable benchmarking on the JVM is a difficult problem. The standard HotSpot VM utilizes a non-deterministic garbage collector (GC). In addition, the JIT is also non-deterministic. These two problems combine into a larger issue.

Timing Execution

There are two methods provided by the JVM for getting a precise time from the OS; `System.currentTimeMillis()` and `System.nanoTime()`. The former returns the "wall" time. It is possible for time to leap forward or backward using this method. If daylight savings time occurs between calls to `System.currentTimeMillis()`, it can result in a negative time.

The latter uses a variety of methods in order to obtain the precise time. The method it uses depends on both the OS and the hardware itself. Under Windows, `System.nanoTime()` makes use of the `QueryPerformanceCounter(QPC)` call. This call may make use of the "programmable-interval-timer (PIT), or the ACPI power management timer (PMT), or the CPU-level timestamp-counter (TSC)." [4] Accessing the PIT and PMT is a slow operation. Accessing the TSC is a very fast operation, but may result in varying numbers [2].

Linux attempts to use the TSC when possible. If it finds the values to be unreliable (eg. different cores vary too much), it makes use of the High Precision Event Timer (HPET) [2].

Platform	Resolution (ms)
Windows 95/98	55
Windows NT, 2000, XP (single processor)	10
Windows XP (multi processor)	15.625
Linux 2.4 Kernel	10
Linux 2.6 Kernel	1

Table 13.1: Reported resolutions of `currentTimeMillis()` by platform [1].

The JVM's timing functions are one area where the JVM's behaviour differs based on the OS. Different operating systems provide varying degrees of time resolution. The speed at which the OS-level calls to these timing functions return can even differ.

Garbage Collection

Just-in-Time Compilation

A typical JVM only loads classes when they're first used [1]. This task typically involves disk I/O, along with extra processing. As a result, the initial run of a task may only...

13.2.2 Mitigating JVM Pitfalls

JIT Compilation

```
33
34 (defn -main
35   [& args]
36   (let [[opts arguments banner] (parse-args (flatten args))]
37     (when (or (opts :help) (not (opts :config)))
38       (println banner)
39       (System/exit 0))
40
41     (let [properties  (load-props (opts :config))
```

Figure 13.1: core.clj

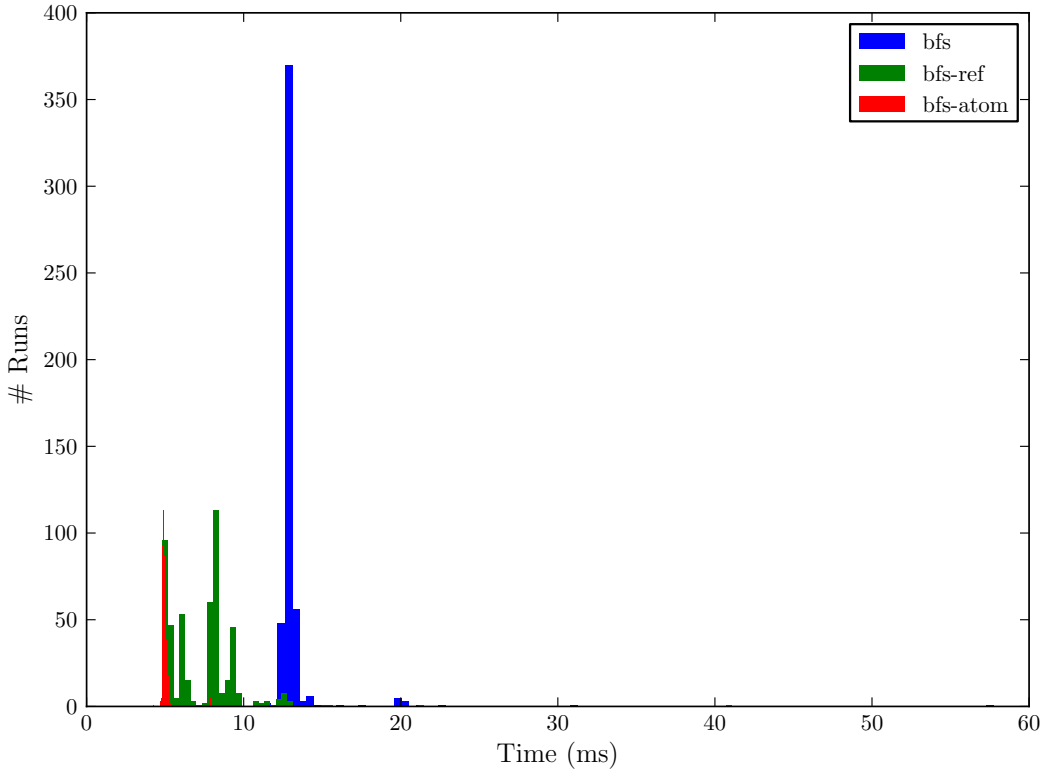


Figure 13.2: 1 Hop

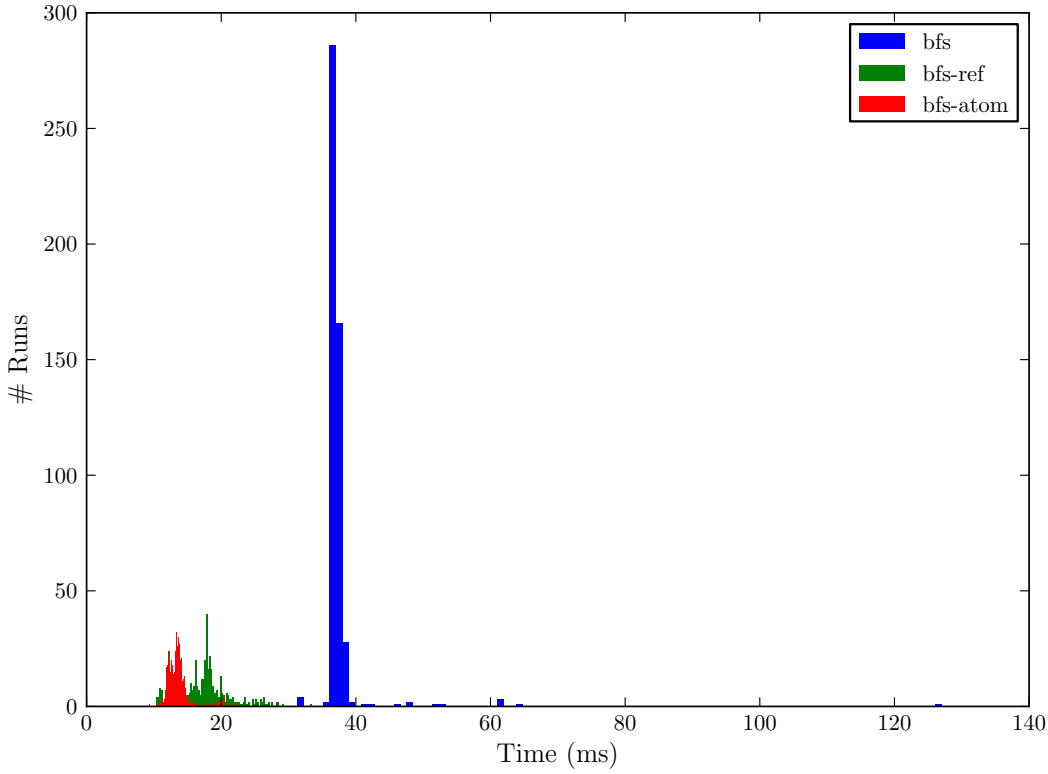


Figure 13.3: 2 Hops

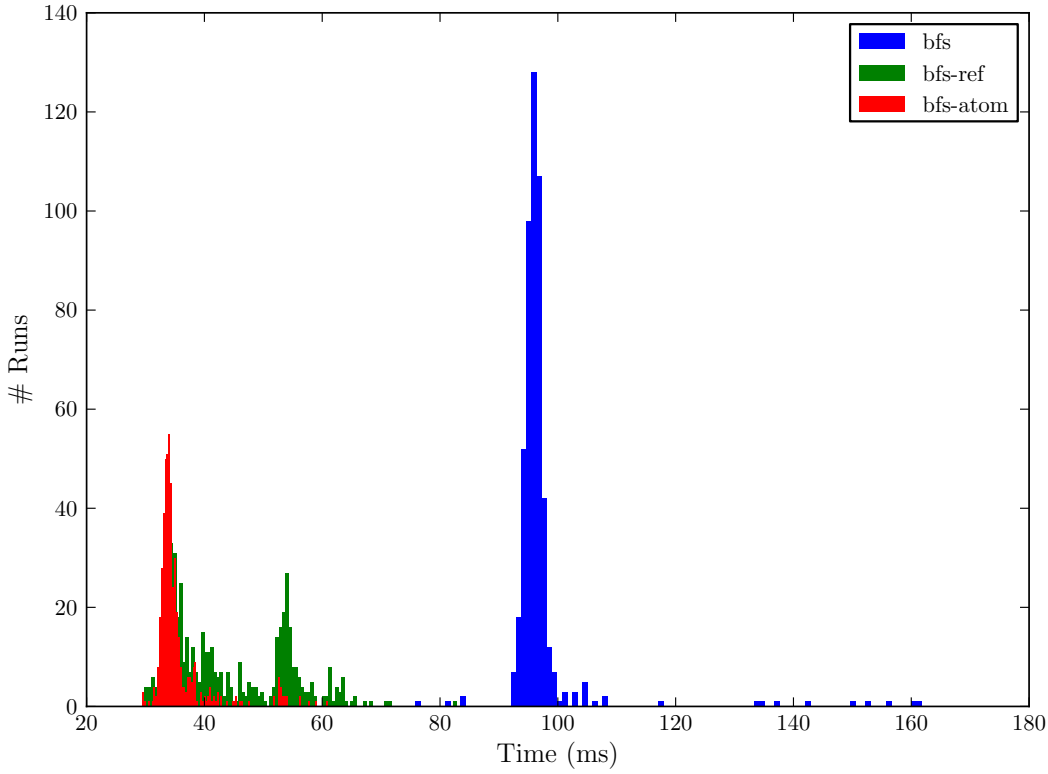


Figure 13.4: 3 Hops

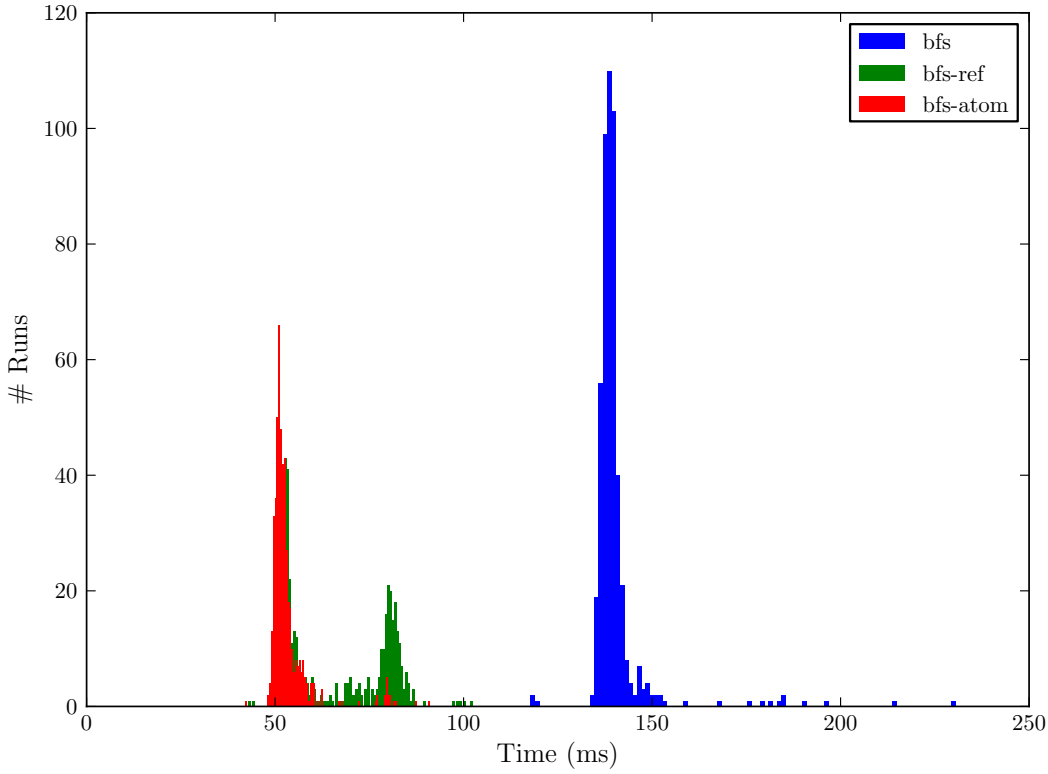


Figure 13.5: 4 Hops

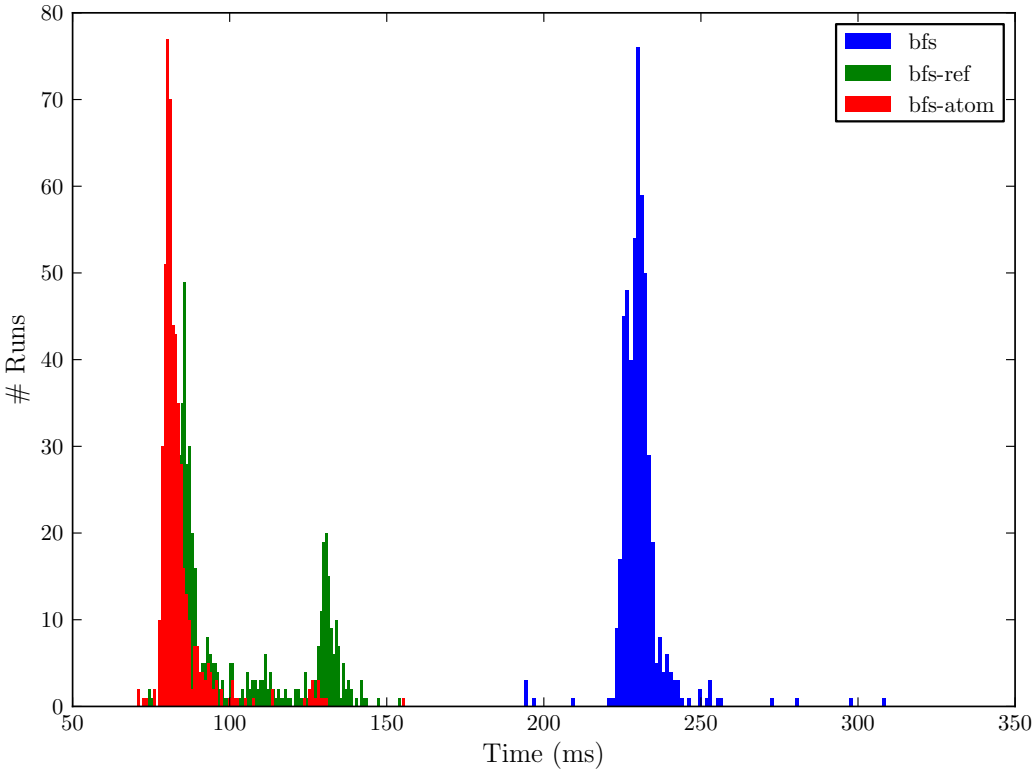


Figure 13.6: 5 Hops

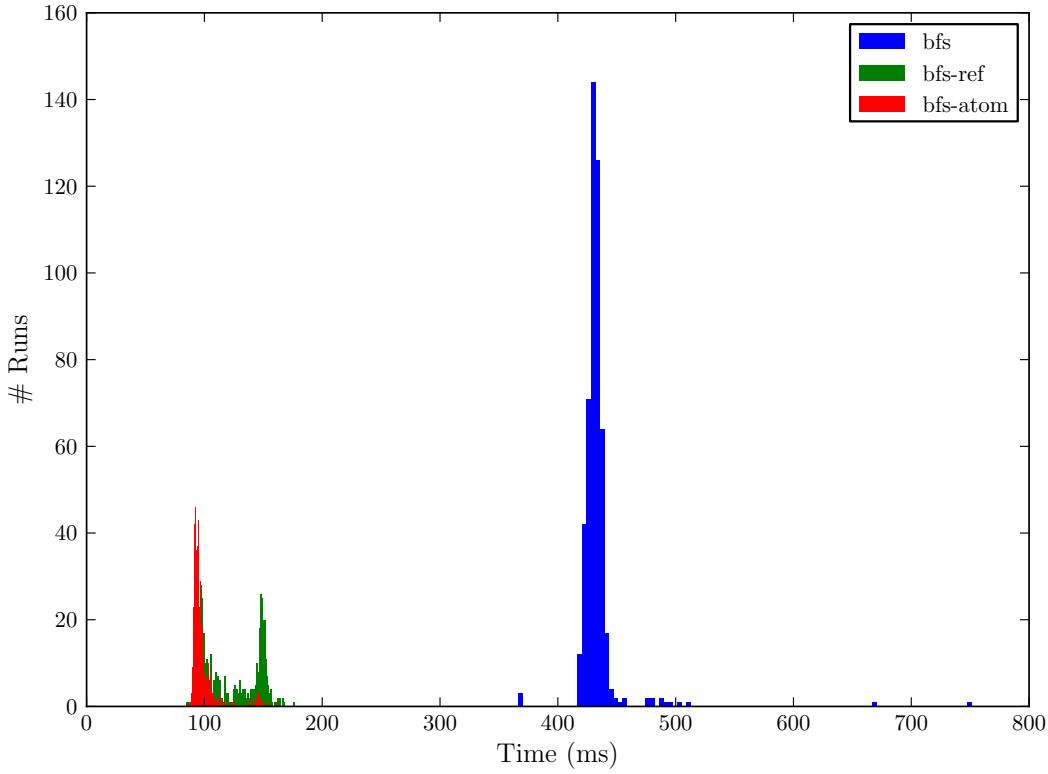


Figure 13.7: 6 Hops

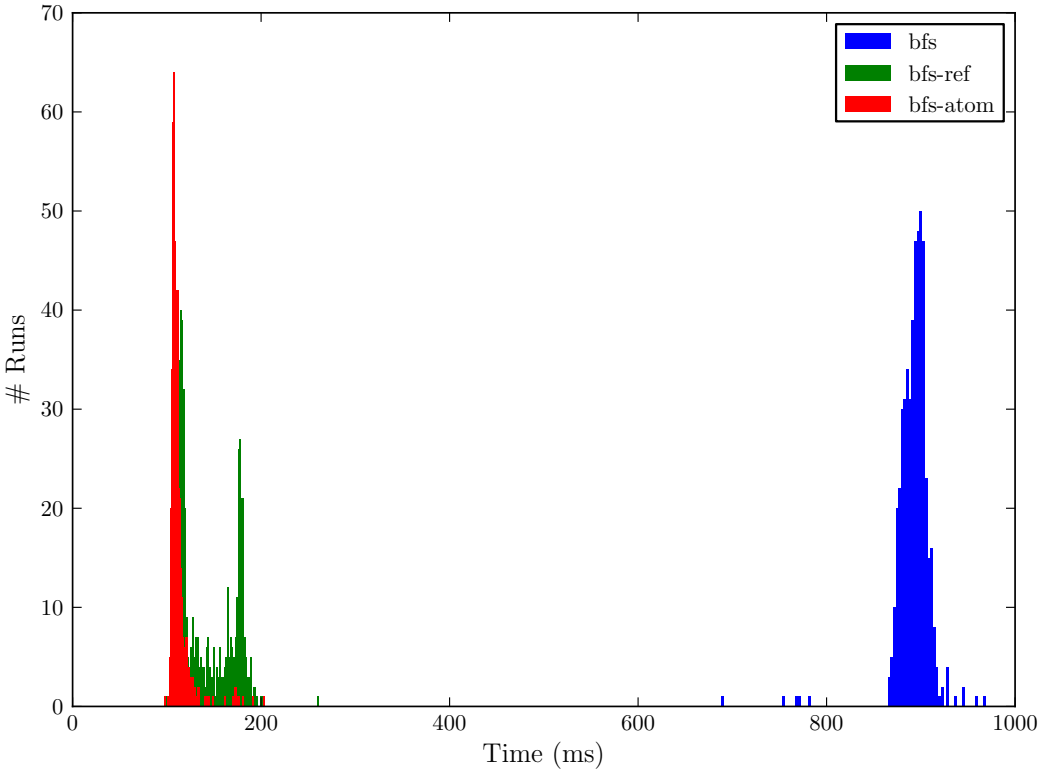


Figure 13.8: 7 Hops

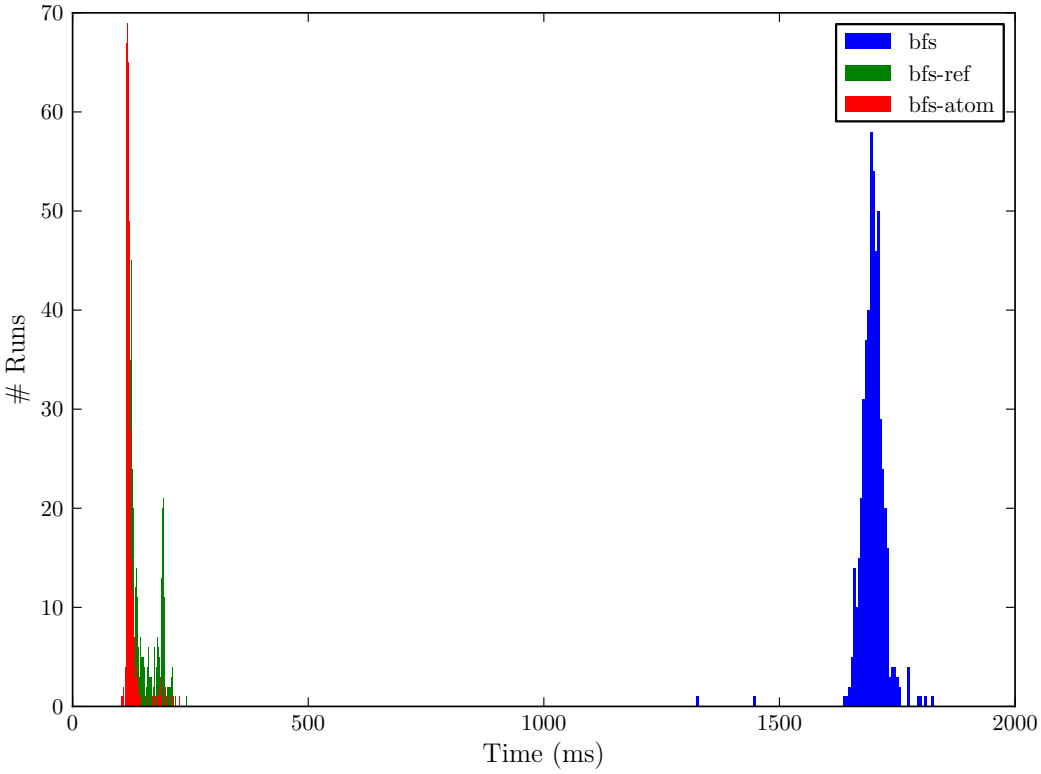


Figure 13.9: 8 Hops

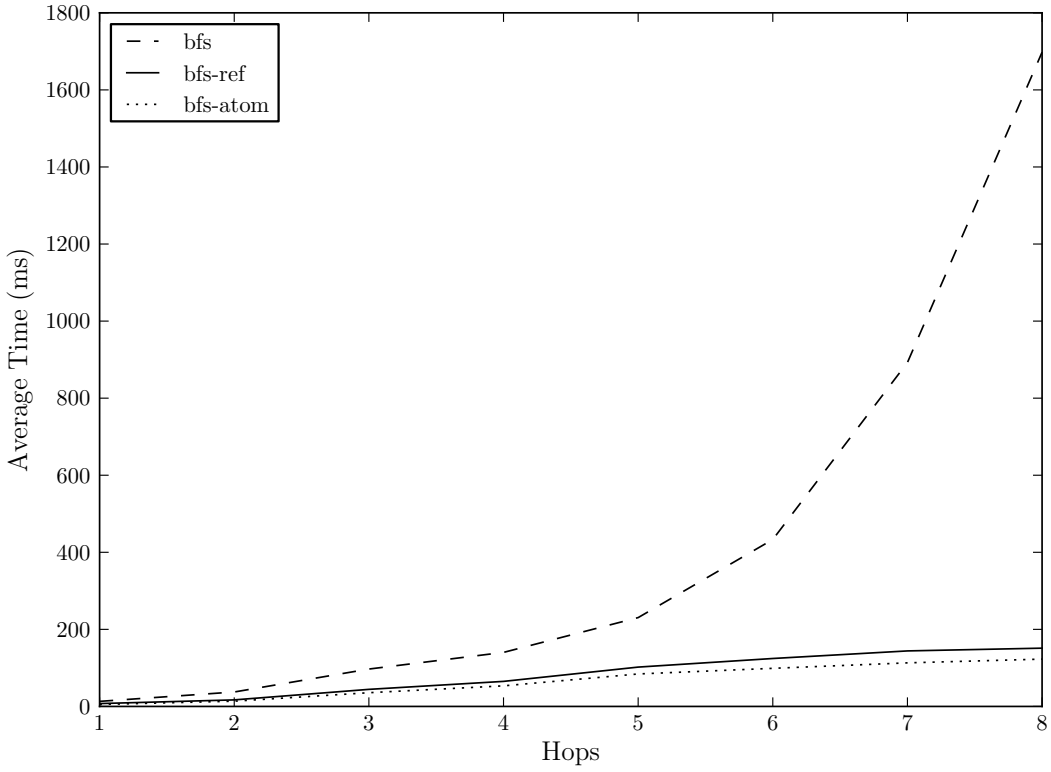


Figure 13.10: Comparison between single threaded and concurrent graph search

Chapter 14

Conclusion

[1]

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

```clojure
(ns molly.core
  (:gen-class)
  (:use molly.conf.config
        molly.index.build
        molly.search.lucene
        [clojure.tools.cli :only (cli)]
        [molly.algo.bfs-atom :only (bfs-atom)]
        [molly.algo.bfs-ref :only (bfs-ref)]
        [molly.algo.bfs :only (bfs)]
        [molly.algo.ford-fulkerson :only (ford-fulkerson)]
        [molly.bench.benchmark :only (benchmark-search)]))

(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"]
       ["--index"       "Build an index of the database"
        :default false
        :flag true]
       ["--benchmark"   "Run benchmarks"
        :default false
        :flag true]
```

```
27          ["-d" "--debug"  "Displays additional information."
28           :default false
29           :flag true]
30          ["-h" "--help"   "Show help"
31           :default false
32           :flag true]))
33
34  (defn -main
35    [& args]
36    (let [[opts arguments banner] (parse-args (flatten args))]
37      (when (or (opts :help) (not (opts :config)))
38        (println banner)
39        (System/exit 0))
40
41      (let [properties  (load-props (opts :config))
42            max-hops    (if (opts :max-hops)
43                          (Integer. (opts :max-hops))
44                          (properties :max-hops))]
45        (if (opts :index)
46          (let [database  (properties :database)
47                index     (properties :index)]
48            (build database index))
49          nil)
50        (if (opts :algorithm)
51          (let [searcher  (idx-searcher
52                           (idx-path
53                             (properties :index)))
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
67                                          target
68                                          max-hops)]
69                (println marked)
70                (println dist)
71                (println prev))
```

```
72              (benchmark-search f searcher source target max-hops))
73            (shutdown-agents))
74          nil))))
```

A.2 molly.conf

A.2.1 molly.conf.config

```
(ns molly.conf.config
  (:require clojure.java.io))

(defn load-props
  ([]
   (load-props ".properties"))
  ([file-name]
   (let [res (clojure.java.io/resource file-name)]
     (if (nil? res)
       (throw
         (IllegalArgumentException. "Unable to load properties."))
       (with-open [^java.io.Reader reader
                   (clojure.java.io/reader res)]
         (let [props (java.util.Properties.)]
           (.load props reader)
           (into {}
                 (for [[k v] props] [(keyword k)
                                     (read-string v)])))))))))

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

A.2.2 molly.conf.mycampus

```
(ns molly.conf.mycampus
  (:use molly.conf.config
        molly.datatypes.database
        molly.datatypes.schema
        korma.core
        korma.db)
  (:import (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))
```

```
43
44 (def mycampus-schema
45   [(EntitySchema.
46      {:T      :entity
47       :C      :course
48       :sql    Course
49       :ID     :code
50       :attrs  [:code :title]
51       :values [:code :title]})
52    (EntitySchema.
53      {:T      :entity
54       :C      :instructor
55       :sql    Instructor
56       :ID     :id
57       :attrs  [:name]
58       :values [:name]})
59    (EntitySchema.
60      {:T      :entity
61       :C      :location
62       :sql    Location
63       :ID     :id
64       :attrs  [:name]
65       :values [:name]})
66    (EntitySchema.
67      {:T      :entity
68       :C      :subject
69       :sql    Subject
70       :ID     :id
71       :attrs  [:id :name]
72       :values [:id :name]})
73    (EntitySchema.
74      {:T      :entity
75       :C      :campus
76       :sql    Campus
77       :ID     :id
78       :attrs  [:name]
79       :values [:name]})
80    (EntitySchema.
81      {:T      :entity
82       :C      :term
83       :sql    Term
84       :ID     :id
85       :attrs  [:id :name]
86       :values [:id :name]})
87    (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"]]})
    ])

(deftype 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
  (:use korma.core
        korma.db))

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

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

A.3.2 molly.datatypes.entity

```
(ns molly.datatypes.entity
  (:use molly.util.nlp)
  (:import
    [clojure.lang IPersistentMap IPersistentList]
    [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.)]
    (do
      (doseq [[field-name field-value] fields]
        (.add doc (field (name field-name) (str field-value))))
```

```
43        doc)))
44
45  (defn row->data
46    ^{:doc "Transforms a row into the internal representation."}
47    [this schema]
48    (let [T         (schema :T)
49          C         (schema :C)
50          attr-cols (schema :attrs)
51          attrs     (if (nil? attr-cols)
52                      this
53                      (select-keys this attr-cols))
54          meta-data {:type T :class C}
55          id-col    (schema :ID)]
56      (with-meta (if (= T :group)
57                   (conj attrs {:entities (uid this id-col)})
58                   attrs)
59                 (condp = T
60                   :value   (assoc meta-data
61                                   :class
62                                   (clojure.string/join "|"
63                                      (map name
64                                           [C (first attr-cols)])))
65                   :entity  (assoc meta-data :id
66                                   (if (coll? id-col)
67                                     (uid this id-col)
68                                     (uid this C id-col)))
69                   :group   (assoc meta-data
70                                   :entities
71                                   (uid this id-col))
72                   (throw
73                     (IllegalArgumentException.
74                       "I only know how to deal with types :value,
75                       :entity, and :group")))))))
76
77  (defn doc->data
78    ^{:doc "Transforms a Document into the internal representation."}
79    [this]
80    (let [fields        (.getFields this)
81          extract       (fn [x] [(keyword (clojure.string/replace
82                                            (.name x) "_" ""))
83                                 (.stringValue x)])
84          check-special (fn [x] (special? (.name x)))
85          filter-fn     (fn [f] (apply hash-map
86                                       (flatten
87                                         (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
  (:use molly.datatypes.database
        molly.datatypes.entity
        molly.search.lucene
        molly.util.nlp
        korma.core))

(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
  (:use molly.conf.config
        molly.conf.mycampus
        molly.datatypes.database
        molly.datatypes.entity
        molly.datatypes.schema
        molly.search.lucene)
  (: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
  ([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
                                   (+ 1 (- (count padded-S) n)))]
                            (. padded-S substring 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]
  (-> (IndexReader/open idx-path) IndexSearcher.))

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

```

```
41 (defn add-doc
42   [idx doc]
43   (. idx addDocument 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]
      (. phrase-query add (Term. field-name (name arg))))

    phrase-query))

(defn boolean-query
  [args]
  (let [query (BooleanQuery.)]
    (doseq [[q op] args]
      (. query add 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.serve

```
(ns molly.server.serve
  (:use compojure.core
        molly.conf.config
        molly.datatypes.entity
        molly.search.lucene
        molly.search.query-builder
        molly.util.nlp
        [molly.algo.bfs :only (bfs)]
        [molly.algo.bfs-atom :only (bfs-atom)]
        [molly.algo.bfs-ref :only (bfs-ref)])
  (:require [noir.response :as response]
            [compojure.handler :as handler]
            [compojure.route :as route]))

(def runtime (Runtime/getRuntime))
(def props (load-props ".properties"))
(def searcher (idx-searcher (idx-path (props :index))))

(defn dox
  [q1 field S op topk]
  (let [bq      (boolean-query
                  (concat [[q1 :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 home-page []
  (response/redirect "/index.html"))

(defn get-value [q]
```

```
  (response/json
    {:result
     (dox (query :type :value)
          :text
          (clojure.string/split (q-gram q) #"\s{1}")
          :or
          (props :topk_value))}))

(defn get-entities [q]
  (response/json
    {:result
     (entities
       :text (clojure.string/lower-case q) (props :topk_entities))}))

(defn get-entity [q]
  (response/json
    {:result
     (entities :id q (props :topk_entity))}))

(defn get-span [e0 eL method]
  (let [start        (System/nanoTime)
        [visited dist prev]
        (condp = method
          "atom"  (bfs-atom searcher e0 eL)
          "ref"   (bfs-ref searcher e0 eL)
          (bfs searcher e0 eL))
        t            (- (System/nanoTime) start)
        eids         (for [[k v] prev] k)
        get-entities (fn [eid]
                       {(keyword eid)
                        (entities :id eid
                                  (props :topk_entity))})
        entities     (into {} (map get-entities eids))]
    (response/json
      {:from     e0
       :to       eL
       :prev     prev
       :entities entities
       :debug    {:time t
                  :mem_total  (.totalMemory runtime)
                  :mem_free   (.freeMemory runtime)
                  :mem_used   (- (.totalMemory runtime)
                                 (.freeMemory runtime))
                  :properties props}})))
```

```
(defroutes app-routes
           (GET "/" [] (home-page))
           (GET "/value" [q] (get-value q))
           (GET "/entities" [q] (get-entities q))
           (GET "/entity" [q] (get-entity q))
           (GET "/span" [e0 eL method] (get-span e0 eL method))
           (route/files "/" {:root "resources/public"}))

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

A.8 molly.algo

A.8.1 molly.algo.common

```
(ns molly.algo.common
  (:use molly.datatypes.entity
        molly.search.lucene
        molly.search.query-builder))

(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 v]
  (remove #{v} (find-group-for-id G v)))

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

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

A.8.3 molly.algo.bfs_atom

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

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

A.8.4 molly.algo.bfs_ref

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

(defn update-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'  (pop (@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
  (use criterium.core)
  (require [clojure.data.json :as json]))

(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

[1] Brent Boyer. *Robust Java benchmarking, Part 1: Issues*. June 2008. URL: http://www.ibm.com/developerworks/java/library/j-benchmark1/index.html.

[2] Jonathan Corbet. *Counting on the time stamp counter*. Nov. 2006. URL: http://lwn.net/Articles/209101/.

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

[4] David Holmes. *Inside the Hotspot VM: Clocks, Timers and Scheduling Events - Part I - Windows*. Oct. 2006. URL: https://blogs.oracle.com/dholmes/entry/inside_the_hotspot_vm_clocks.

[5] *The Xapian Project : Features*. URL: http://xapian.org/features.

To do...

- ☐ 1 (p. 4): Diagram of schema-level graph (FKs, relations)
- ☐ 2 (p. 4): More examples of queries
- ☐ 3 (p. 5): Add overview diagram of system design.