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. 31

FK foreign key. vii, 3–6, 8

JDBC Java database connectivity. 27

JSON JavaScript object notation. 31

JVM Java virtual machine. vi, 27

RDBMS relational database management system. 8, 10

SQL structured query language. 4, 5, 18

STM software transactional memory. 25, 26

TF-IDF term frequency and inverse document frequency. 12

List of Symbols

schema graph G graph representation of schema. 4, 6, 21

entity group T forest of named tuples. 6, 21, 22

document collection C set of documents. 11–16, 22

terms T set of unique terms in a document collection. 11, 13, 16

document d set of fields. x, 11–16, 18–20

search query q special case of document. 14–16, 18, 19, 23

field f named sub-document in . 20, 23

term τ unique term in document collection. 11–14, 18

N number of documents in collection. 11

database D set of relations. 4, 6, 21

relation r set of named tuples. 2–4, 6, 21

named tuple t ordered set of values. vi, 2–4, 6, 7, 20–22

attribute α named column. 3, 4, 6, 7, 20

key K uniquely identifies a named tuple in a relation. 3

Chapter 1

Background (2 days)

Literature search on:

- DBExplore
- XRank
- BANKS
- ...

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.

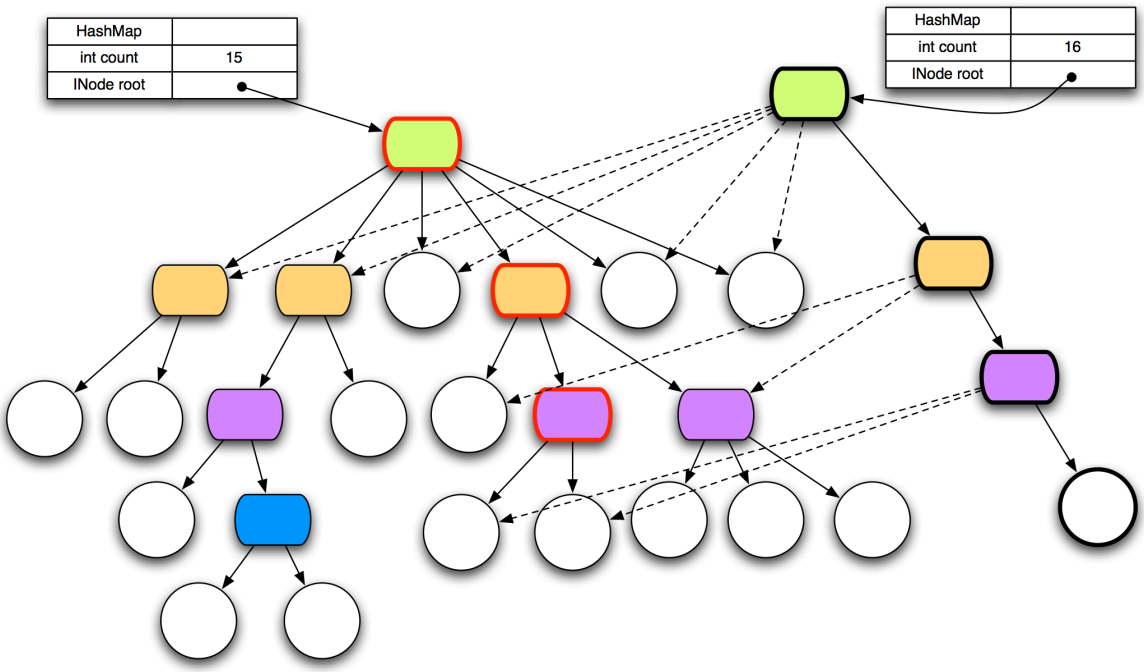

Figure 4.1: Representation of how data structures are "changed" in Clojure (Source: [Hic09])

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 (??)

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

```
15 (defn ^Directory idx-path
16   [path]
17   (-> 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).

```
16 (defprotocol IConfig
17   (connection [this])
18   (schema [this])
19   (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 on page 28. 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`	Structured query language (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 open the database for reading and the index for writing. This paths to the database and index are given by the `.properties` file.

The second step is to read in the configuration file which describes the schema graph. The schema graph is given as `EntitySchema` records. These records contain the SQL required to retrieve the named tuples from the database, as well as other information required to transform each named tuple into a document.

The indexing process iterates over every `EntitySchema` record, calling the `crawl` function on each one. The `crawl` function uses the configuration map attached to the `EntitySchema` record to construct itself.uf, which in turn executes a SQL query using `execute-query` (defined by the `Database` protocol). For every named tuple, a function (provided by the `EntitySchema` record) is executed on it.

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

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

4.2.4 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 6

Conclusion (0 days)

Survived Clojure.

Bibliography

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

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

[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.

[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. 21): More concurrency