Molly

by

Richard Drake

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

Masters of Science

in

Faculty of Science

Computer Science

University of Ontario Institute of Technology

Supervisor: Dr. Ken Q. Pu

September 2012

Abstract

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

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

Literature Review

Chapter 2

Theoretical

Problem definition

2.1 Data Representation & Notation

Formal definition data model

2.2 Relational Database Abstraction

Value -> entity -> graph abstract of a relational database Connected subgraph from keyword queries

2.3 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 2.1). 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 2.1: Courses entity schema

The **Instructors** entity represents an individual instructor. Each instructor has a unique identifier, as well as a name (See Table 2.2). 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 2.2: 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 2.3).

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 2.4).

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 2.3: Sections entity schema

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 2.4: 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 2.5).

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

Table 2.5: Teaches entity schema

2.4 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 eliminiates 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 3

Implementation

3.1 Functional vs. Procedurial Programming of Algorithms

Key differences between FP and procedurial (for algorithms)

3.1.1 Functional Data Structures

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

3.2 Tunable Parameters

Tunable parameters

Chapter 4

System Implementation

4.1 Technology Selection

4.1.1 Programming Language

4.1.2 Relational Database

4.1.3 Full-Text Search Database

4.1.4 Web Stack

4.2 Implementation Issues

Chapter 5

Performance & Evaluation

Screenshots? Graphs Experiments

Chapter 6

Conclusion