Towards a Concurrent Implementation of Keyword Search Over Relational Databases

by

Richard J.I. Drake

A thesis submitted in conformity with the requirements
for the degree of Master of Science (M.Sc.)
Faculty of Science (Computer Science)
University of Ontario Institute of Technology

Supervisor(s): Dr. Ken Q. Pu

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

In this chapter we provide a formal definition of the relational data model, discuss its merits, its shortcomings, and contrast it to the document data model. Contrary to the relational model, the document model permits fast and flexible keyword search without requiring explicit domain knowledge of the data. In addition, we demonstrate the feasibility of encoding a relational model into a document model in a lossless manner.

The term "data model" refers to a notation for describing data and/or information. It consists of the data structure, operations that may be performed on the data, as well as constraints placed on the data [GUW09].

2.1 Relational model with star schema

In this section we formally define the relational model.

2.1.1 Relational Model of Data

In its most basic form, the relational data model is built upon sets and tuples. Each of these sets consist of a set of finite possible values. Tuples are constructed from these sets to form relations.

Formally, a Relation is defined as follows [Cod90]:

Definition 1. Relation Given a list of sets $[S_1, S_2, \ldots, S_n]$, let R be a relation on these n sets if it is a set of n-tuples, with the first component from S_1, the second component from S_2, and so on.

More concisely,

$$R \subset S_1 \times S_2 \times \ldots \times S_n$$

R is said to be of degree n, denoted as deg_R. Each of the sets on which one or more relation is based is called the domain, denoted as dom_S.

Example 1. Consider the following sets

$$\begin{aligned} S_1 &= \{\text{“Winter 2014”}, \text{“Fall 2013”}\} \\ S_2 &= \{\text{“CSCI 3030U”}, \text{“CSCI 4020U”}\} \\ S_3 &= \{\text{“Ken Pu”}\} \end{aligned}$$

which have the properties

$$\begin{aligned} \mathrm{dom}_{S_1} &= \text{terms} \\ \mathrm{dom}_{S_2} &= \text{courses} \\ \mathrm{dom}_{S_3} &= \text{instructors} \end{aligned}$$

By taking the Cartesian product, we arrive at the following set of tuples

$$S_1 \times S_2 \times S_3 = \left\{ \begin{array}{l} (\text{“Winter 2014”}, \text{“CSCI 3030U”}, \text{“Ken Pu”}) \\ (\text{“Fall 2013”}, \text{“CSCI 3030U”}, \text{“Ken Pu”}) \\ (\text{“Winter 2014”}, \text{“CSCI 4020U”}, \text{“Ken Pu”}) \\ (\text{“Fall 2013”}, \text{“CSCI 4020U”}, \text{“Ken Pu”}) \end{array} \right\}$$

Furthermore, given the relation

$$R = \left\{ \begin{array}{l} (\text{“Winter 2014”}, \text{“CSCI 4020U”}, \text{“Ken Pu”}) \\ (\text{“Fall 2013”}, \text{“CSCI 3030U”}, \text{“Ken Pu”}) \end{array} \right\}$$

We see that $R \subset S_1 \times S_2 \times S_3$, with $\mathrm{deg}_R = 3$.

Definition 2. Let d be a database instance. A database is comprised of three main components:

- $\textsc{Name}[d] \rightarrow$ string
- $\textsc{Rel}[d] \rightarrow$ list(REL)
- $\text{FK}[d] \rightarrow$ list(FK)

Definition 3. Relation

Let $r \in \textsc{Rel}[d]$, where d is defined in Definition 2. A relation is comprised of three main components:

- $\textsc{Name}[r]$: string
- $\textsc{Attr}[r]$: list(ATTR)
- $\textsc{Key}[r]$: 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, $\textsc{Key}[r] \subseteq \textsc{Attr}[r]$.

Definition 4. Attribute

Let $a \in \textsc{Attr}[r]$, where r is defined in Definition 3. An attribute is comprised of two main components:

- $\textsc{Name}[a]$
- $\textsc{Type}[a]$

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

Definition 5. Foreign Key

Let $\theta \in \text{FK}[d]$ be a FK instance,where d is defined in Definition 2. A foreign key is comprised of two main components:

- $\textsc{From}[fk] = (\textsc{Rel}_s[\theta], \textsc{Attr}_s[\theta])$

- $\textsc{To}[fk] = (\textsc{Rel}_t[\theta], \textsc{Attr}_t[\theta])$

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:

$$V(G) \subseteq \textsc{Rel}(DB)$$

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

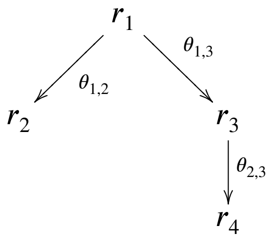
$$r(a_1, a_2, \dots, a_k) \to s(b_1, b_2, \dots, b_k)$$

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

Example 2.

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

2.1.3 Instances of an Entity Group



Instances are obtained by the following process.

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

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

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

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

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

Motivation:

To do (1)

Database schema

Vertices: $\textsc{Rel}[d]$

Edges: $\text{FK}[d]$

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 (A$\underline{\text{C}}$ID?)
- 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)

In this section we formally define the document model.

Documents are a unit of information. The definition of unit can vary. It may represent an email, a book chapter, a memo, etc. Contained within each document is a set of terms.

In contrast to the relational model, the document model represents unstructured data. Examples of information suitable to the document model includes emails, memos, book chapters, etc.

Example 3. Let $\{d_1, d_2, d_3\}$ be the set of documents, each representing a course title.

$$\begin{aligned} d_1 &= \text{Software Design and Analysis} \\ d_2 &= \text{Software Quality Assurance} \\ d_3 &= \text{Analysis and Design of Algorithms} \end{aligned}$$

The simplest method would be to perform a linear scan through every document, returning each document that contains a search query.

If we were to issue a query $q =$ "Design", we would receive a result of $\{d_1, d_3\}$. Unfortunately this search must be performed every time a user issues a search query. Thus the search time would grow linearly with the number of documents, as well as the length of each document.

Another method would be to construct an incidence matrix of each term in each document, then consult this matrix when a search query is issued. Using this method would incur an initial penalty, but this would only occur once.

	d_1	d_2	d_3
Algorithms	0	0	1
Analysis	1	0	1
Assurance	0	1	0
Design	1	0	1
Quality	0	1	0
Software	1	1	0

A search then becomes a simple lookup in the matrix. A search for "Algorithms" would yield the binary string `001`. It also allows for simple boolean operations. A query of "Analysis" AND "Software" would become

$$101 AND 110$$

which would yield 100, or the set $\{d_1\}$.

While an incidence matrix solves the problem of having to scan every document for every search, it introduces additional problems. There may be a large number of documents, each with its own unique terms. This causes the matrix to become very sparse.

In order to deal with the problem of a sparse matrix, we use an inverted list index. It consists of a sorted dictionary of terms, each pointing to one or more documents that contain that term.

This permits more space efficient storage.

Note: The terms in the inverted list index are sorted. This permits binary search, resulting in lookup performance of $\mathcal{O}(\log n)$.

2.3.1 Keyword Query Vectorization

We now have a method for determining which documents contain a particular term. While this allows a user to manually sort through all of the results, it does not provide an indication of the importance, or score, of each result.

Term	Doc ID
software	1
design	1
analysis	1
software	2
quality	2
assurance	2
analysis	3
design	3
algorithm	3

(a) Initial inverted list index

Term	Doc ID
algorithm	3
analysis	1
analysis	3
assurance	2
design	1
design	3
quality	2
software	1
software	2

(b) Sorted inverted list index

Term	df	Doc ID
algorithm	1	[3]
analysis	2	[1, 3]
assurance	1	[2]
design	2	[1, 3]
quality	1	[2]
software	2	[1, 2]

(c) Completed inverted list index

Figure 2.1: Construction of the inverted list index

Term Frequency

In order to measure importance of a term within a document, we look at the number of occurrences of said term within the document versus how many individual terms are in the document. This is the term frequency.

$$\mathrm{tf}_{t,d} = 0.5 + \frac{0.5 \times \mathrm{f}_{t,d}}{\max\{\mathrm{f}_{w,d} \mid w \in d\}}$$

2.3.2 Inverse Document Frequency

The inverse document frequency is a measurement of the rarity of a term in the space of all documents. A term that occurs multiple times within a single document is considered less rare if said term also occurs within many documents.

$$\text{idf}_t = \log \frac{N}{\text{df}_t}$$

These two functions are combined to measure the importance of a term within a document.

$$\text{tf-idf}_{t,d} = \text{tf}_{t,d} \times \text{idf}_t$$

Scoring a Document

With the ability to score a document for a particular term, we can now think of a document d score for a query q, where $q = [t_1, t_2, \dots, t_n]$, as a vector.

$$\text{score}_{q,d} = \begin{bmatrix} \text{tf-idf}_{t_1,d} \\ \text{tf-idf}_{t_2,d} \\ \vdots \\ \text{tf-idf}_{t_n,d} \end{bmatrix}$$

Therefore the overall score is as follows.

$$\text{score}_{q,d} = \sum_{t \in q} \text{tf-idf}_{t,d}$$

2.3.3 Cosine Similarity

$$\text{similarity}_{d_1,d_2} = \cos\theta = \frac{d_1 \cdot d_2}{\|d_1\| \|d_2\|}$$

2.3.4 Jaccard Similarity

$$\text{similarity}_{d_1,d_2} = \frac{|\, d_1 \cap d_2 \,|}{|\, d_1 \cup d_2 \,|}$$

2.3.5 Extending the Document Model

In reality not all documents contain completely unstructured information. For example, an email contains additional information beyond the message such as subject, recipient, etc. This additional

Field	Contents
code	CSCI 3030U
title	Database Systems and Concepts

information is stored in fields.

Example 4. Every course has two main attributes, a code and a title. These attributes could be stored in a single document. However, we may wish to apply different analysis techniques to each attribute type.

In the case of a code, we likely want it copied verbatim. An example tokenization of "CSCI 3030U" would be {`csci`, `3030u`}. A standard analyzer would have eliminated the second token.

Whereas a standard analyzer would work well for course titles. For example, an ideal tokenization of "Database Systems and Concepts" would be {`database`, `system`, `concept`}.

Plural forms of words were reduced to their singular form, and the stop word "and" was eliminated.

- Definition of keyword search queries: vectorization of documents (tf-idf) and queries. Models of distance between documents and queries (cosine-distance, jaccard distance, BM25).
- 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.

Bibliography

[Cod90] E. F. Codd. *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[GUW09] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

To do...

□ 1 (p. 6): Diagram of schema-level graph (FKs, relations)

□ 2 (p. 7): More examples of queries