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

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

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

2.1 Relational Model

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

Definition 1 (Named Tuple). A named tuple t is an instance of a relation r, consisting of values corresponding to the attributes of r. For example,

$$t = \{\text{name} : \text{“Jack Bauer”}, \text{age} : 39\}$$

We denote the attributes of t as $\text{ATTR}[t] = \{\text{name}, \text{age}\}$ The values are $t[\text{name}] =$ "Jack Bauer", and $t[\text{age}] = 39$.

Definition 2 (Relation)**.** A relation r is a set of named tuples, $r = \left\{t_1, t_2, \ldots, t_n\right\}$, such that all the named tuples share the same attributes.

$$\forall t, t' \in r, \text{ATTR}[t] = \text{ATTR}[t']$$

For example,

$$r = \left\{ \begin{array}{l} \{\text{name} : \text{"Jack Bauer"}, \text{age} : 39\}, \\ \left\{\text{name} : \text{"Bruce Wayne"}, \text{age} : 39\right\}, \\ \{\text{name} : \text{"Clark Kent"}, \text{age} : 45\} \end{array} \right\}$$

Relations are typically represented as tables.

name	age
"Jack Bauer"	39
"Bruce Wayne"	39
"Clark Kent"	45

Table 2.1: Person table

Definition 3 (Keys)**.** Keys are constraints imposed on relations. A key constraint $\boldsymbol{K}$ on a relation r is a subset of $\text{ATTR}[r]$ which may uniquely identify a tuple. Formally, we say r satisfies the key constraint $\boldsymbol{K}$, denoted as $r \vDash \boldsymbol{K}$, subject to

$$\forall t, t' \in r, t \neq t' \implies t[\boldsymbol{K}] \neq t'[\boldsymbol{K}]$$

For example, in Table 2.1, the relation satisfies the key constraint $\{\text{name}\}$, but not $\{\text{age}\}$.

Definition 4 (Foreign Keys). A foreign key constraint applies to two relations, r_1, r_2. It asserts that values of certain attributes of r_1 must appear as values of some corresponding attributes of r_2. A foreign key constraint is written as

$$\theta = r_1(a_1, a_2, \ldots, a_k) \rightarrow r_2(b_1, b_2, \ldots, b_k)$$

where $a_i \subseteq \text{ATTR}[r_1]$ and $b_i \subseteq \text{ATTR}[r_2]$. We say (r_1, r_2) satisfies θ, denoted as $(r_1, r_2) \models \theta$, if

$$\forall t \in r_1, \exists t' \in r_2 \mid t[a_1, a_2, \ldots, a_k] = t'[b_1, b_2, \ldots, b_k]$$

Example 1. Suppose we have a relation Superhero(name, superpower). We can impose a FK constraint of

$$\text{Superhero(name)} \rightarrow \text{Person(name)}$$

Definition 5 (Relational Database). A relational database, d, is a named collection of relations (as defined by Definition 2, keys (as defined by Definition 3), and foreign key constraints (as defined by Definition 4.

We use $\text{NAME}[d]$ to denote the name of d, $\text{REL}[d]$ the list of relations in d, $\text{KEY}[d]$ the list of key constraints of d, and $\text{FK}[d]$ the list of foreign key constraints of d.

2.1.1 Schema Group

Definition 6 (Schema Graph). If we view relations as vertices, and foreign key constraints as edges, a database d can be viewed as a *schema graph* G, formally defined as

$$\begin{aligned} \text{vertices} : V(G) &= \text{REL}[d] \\ \text{edges} : E(G) &= \text{FK}[d] \end{aligned}$$

Example 2. Given the following schema

Superhero(name, power)

Person(name, age, birthplace)

Planet(name, size, age, destroyed, galaxy)

Link(name, peer, relation type)

and the following foreign key constraints

$$\begin{aligned} \text{Superhero(name)} &\rightarrow \text{Person(name)} \\ \text{Person(birthplace)} &\rightarrow \text{Planet(name)} \\ \text{Link(name)} &\rightarrow \text{Person(name)} \\ \text{Link(peer)} &\rightarrow \text{Person(name)} \end{aligned}$$

we produce the following schema graph.

To do (1)

The relational data model is particularly powerful for analytic queries. Given the schema below above, one can formulate the following analytic queries in a query language known as SQL.

List all superheroes whose home planet has not been destroyed.

SELECT Person.name FROM Person JOIN Planet on Person.birthplace = Planet.name WHERE NOT Planet.destroyed;

2.1.2 Entity Group

Definition 7 (Entity Group)**.** An entity group is a forest, T, of tuples interconnected by join conditions defined by the foreign key constraints in the schema graph. Given two vertices $t_1, t_2' \in V(\boldsymbol{T})$, it must be that:

$\exists r_1, r_2 \in \text{REL}[d]$ such that $t_1 \in r_1, t_2 \in r_2$, and $\left(r_1, r_2\right) \in \boldsymbol{G}$. This is to say that t_1 and t_2 belong to two relations that are connected by the schema graph.

Let $r_1\left(a_1, \ldots, a_k\right) \rightarrow r_2\left(b_1, \ldots, b_k\right)$ be the FK that connects r_1, r_2. We further assert that $t_1[a_1, \ldots, a_k] = t_2[b_1, \ldots, b_k]$.

The motivation of entity groups is to define complex structured objects that can include more information than individual tuples in the relations.

Example:

name: Clark Kent age: ? birthplace: Krypton superpower: ? friends: ?. enemies: ?

This object (the profile of Clark Kent) can only be represented as an entity group as no single tuple in any of the relations has such detailed information. This is a result of normalization of the schema.

2.1.3 Pros and Cons of the Relational Model

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

For each item of pro and con, use the running example to illustrate them.

For example:

1. both queries are executable by Postgres 2. if we try to insert "Clark Kent" again, it will be rejected. If we insert "Frog Boy" as a superhero, it will be rejected. If we change the age of "Clark Kent" this will only need to occur in a single place. 3. Change the "Ken" and "CSCI" to some superhero example.

Do the same for CONS.

E.g. Tell me something about "Batman". We need to know the relation and attributes involved in order to author the SQL query. "Supermen" will return nothing.

2.1.4 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)

2.1.5 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

2.2 Document Model

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 semi-structured data. Examples of information suitable to the document model includes emails, memos, book chapters, etc.

Definition 8 (Document). A document is a unit of information. The definition of unit can vary; it may represent an email, a book chapter, a memo, etc. The k-th document is denoted by d_k, which uniquely identifies the document within the document space.

Documents may be comprised of one or more fields, Fields$[d]$. They can be thought of as a dictionary.

$$d_k = \left\{ \begin{array}{c} f_1 \to v_1 \\ f_2 \to v_2 \\ \vdots \\ f_n \to v_n \end{array} \right\}$$

Definition 9 (Field). A field, f, is a dictionary item within a document d.

$$f \in \text{Fields}[d]$$

Definition 10 (Document Collection). Given a set of documents $\boldsymbol{D} = \left\{d_1, d_2, \ldots, d_n\right\}$, we say that $d_1, d_2, \ldots, d_n$ are the documents within the collection $\boldsymbol{D}$. The size of $\boldsymbol{D}$ is denoted by $\boldsymbol{N}$.

Definition 11 (Term).

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 $\mathcal{O}(\log n)$.

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

2.2.1 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 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.

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

To do (3)

2.2.2 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 $q = [t_1, t_2, \ldots, t_n]$, 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.

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

2.2.3 Cosine Similarity

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

2.2.4 Jaccard Similarity

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

Field	Contents
code	CSCI 3030U
title	Database Systems and Concepts

2.2.5 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 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

2.3 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

2.4 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. 5): ER diagram or something of schema.
- ☐ 2 (p. 7): More examples of queries
- ☐ 3 (p. 10): Add consistency for this equation. Maybe find another variant not from Wikipedia.