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. 28

FK foreign key. 4, 6

JSON JavaScript object notation. 28

RDBMS relational database management system. 8, 10

SQL structured query language. 5, 18

List of Symbols

N number of documents in collection. 11

M size of T. 11

T all terms in document collection. ix, 11

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 [**dbsys-06**].

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,

Example 1. Given a tuple $t = \{\text{name} : \text{“General Zod”}, \text{age} : 42, \text{birthplace} : \text{“Krypton”}\}$, we denote the attributes of t as $\textsc{Attr}[t] = \{\text{name}, \text{age}, \text{birthplace}\}$. The values are $t[\text{name}] =$

"General Zod", $t[\text{age}] = 42$, and $t[\text{birthplace}] =$ "Krypton".

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

$$\forall t, t' \in r, \textsc{Attr}[t] = \textsc{Attr}[t']$$

Example 2. An example Person relation, r, would be as follows,

$$r = \left\{ \begin{array}{l} \{\text{name} : \text{"Superman"}, \text{age} : 33, \text{birthplace} : \text{"Krypton"}\}, \\ \{\text{name} : \text{"Batman"}, \text{age} : 30, \text{birthplace} : \text{"Earth"}\}, \\ \{\text{name} : \text{"Flash"}, \text{age} : 53, \text{birthplace} : \text{"Earth"}\}, \\ \{\text{name} : \text{"Wonder Woman"}, \text{age} : 30, \text{birthplace} : \text{"Earth"}\}, \\ \{\text{name} : \text{"General Zod"}, \text{age} : 42, \text{birthplace} : \text{"Krypton"}\}, \end{array} \right\}$$

Relations are typically represented as tables.

name	**age**	**birthplace**
Superman	33	Krypton
Batman	30	Earth
Flash	53	Earth
Wonder Woman	30	Earth
General Zod	42	Krypton

Table 2.1: Person table

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

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

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

Definition 4 (Foreign Keys). A foreign key (FK) 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 FK constraint is written as

$$\theta = r_1(\alpha_{11}, \alpha_{12}, \ldots, \alpha_{1k}) \to r_2(\alpha_{21}, \alpha_{22}, \ldots, \alpha_{2k})$$

where $\alpha_{1i} \subseteq \text{ATTR}[r_1]$ and $\alpha_{2i} \subseteq \text{ATTR}[r_2]$. We say (r_1, r_2) satisfies θ, denoted as $(r_1, r_2) \vDash \theta$, if

$$\forall t \in r_1, \exists t' \in r_2 \text{ such that } t[\alpha_{11}, \alpha_{12}, \ldots, \alpha_{1k}] = t'[\alpha_{21}, \alpha_{22}, \ldots, \alpha_{2k}]$$

Example 3. Suppose we have a relation $\text{Superhero}(\text{name}, \text{superpower})$. We can impose a FK constraint of

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

Definition 5 (Relational Database). A relational database, $\mathbb{D}$, is a named collection of relations (as defined by Definition 2 on the previous page), keys (as defined by Definition 3 on the preceding page), and foreign key constraints (as defined by Definition 4.

We use *General Zod* $\mathbb{D}$ to denote the name of $\mathbb{D}$, $\text{REL}[\mathbb{D}]$ the list of relations in $\mathbb{D}$, $\text{KEY}[\mathbb{D}]$ the list of key constraints of $\mathbb{D}$, and $\text{FK}[\mathbb{D}]$ the list of foreign key constraints of $\mathbb{D}$.

2.1.1 Schema Group

Definition 6 (Schema Graph). If we view relations as vertices, and foreign key constraints as edges, a database $\mathbb{D}$ can be viewed as a *schema graph* $\text{SCHEMAGRAPH}[]$, formally defined as

$$\text{vertices} : V\left(\textsc{SchemaGraph}[]\right) = \textsc{Rel}[\mathbb{D}] \tag{2.1}$$

$$\text{edges} : E\left(\textsc{SchemaGraph}[]\right) = \text{FK}[\mathbb{D}] \tag{2.2}$$

Example 4. Given the following schema

$$\text{Superhero}(\underline{\text{name}}, \text{power})$$

$$\text{Person}(\underline{\text{name}}, \text{age}, \text{birthplace})$$

$$\text{Planet}(\underline{\text{name}}, \text{size}, \text{age}, \text{destroyed}, \text{galaxy})$$

$$\text{Link}(\underline{\text{name}}, \text{peer}, \text{relationtype})$$

and the following foreign key constraints

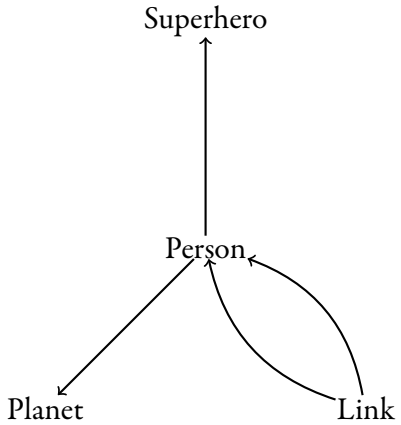
$$\text{Superhero}(\text{name}) \rightarrow \text{Person}(\text{name}) \tag{2.3}$$

$$\text{Person}(\text{birthplace}) \rightarrow \text{Planet}(\text{name}) \tag{2.4}$$

$$\text{Link}(\text{name}) \rightarrow \text{Person}(\text{name}) \tag{2.5}$$

$$\text{Link}(\text{peer}) \rightarrow \text{Person}(\text{name}) \tag{2.6}$$

we produce the following schema graph



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 structured query language (SQL).

Example 5. 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;
```

Figure 2.1: Query to find superheroes with a home to go back to

Results in the following output, given the data from Table 2.1 on page 3,

name
Batman
Flash
Wonder Woman

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 $t_1, t_2 \in V(T)$, it must be that:

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

Let $r_1(\alpha_{11}, \alpha_{12}, \ldots, \alpha_{1k}) \rightarrow r_2(\alpha_{21}, \alpha_{22}, \ldots, \alpha_{2k})$ be the FK that connects r_1, r_2. We further assert that $t_1[\alpha_{11}, \alpha_{12}, \ldots, \alpha_{1k}] = t_2[\alpha_{21}, \alpha_{22}, \ldots, \alpha_{2k}]$.

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

Example 6. The information in Table 2.2 on the next page all relates to Superman, however no single tuple in the database has all of this information as a result of database normalization.

Attribute	Value
name	Superman
age	33
birthplace	Krypton
superpower	flying, strength, speed
friends	Wonder Woman, Batman
enemies	General Zod

Table 2.2: Superman's properties

We require an entity group (Figure 2.2) to join together all pieces of information related to Superman.

To do (1) `SELECT * FROM Link JOIN Superhero ON Link.name = Superhero.name JOIN Person ON Person.name = Superhero.name JOIN Planet ON Person.birthplace = Planet.name WHERE Person.name = 'Superman';`

Figure 2.2: Superhero entity group

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 both the strong and weak points of the relational model.

Pros

The relational model was first proposed by Edgar F. Codd in 1969 while working at IBM [**codd-69**].

The enforcement of constraints is essential to the relational model. There are several types of constraints. The first constraint maintains uniqueness.

The Person relation has the attribute name as its primary key. In order for other relations to reference a specific named tuple, the name attribute must be unique.

Example 7 (Unique Constraint). Attempt to insert another person named "Superman."

```
INSERT INTO Person
VALUES      ('Superman',
             35,
             'Earth');
```

The relational database management system (RDBMS) enforces the primary key constraint on the `name` attribute, rejecting the insertion.

```
ERROR:  duplicate key value violates unique constraint "person_pkey"
DETAIL:  Key (name)=(Superman) already exists.
```

With the uniqueness of named tuples guaranteed (as demonstrated in Example 7), we must ensure that any named tuples that are referenced actually exist. If they do not, the database must not permit the operation to continue. Doing so would lead to dangling references.

Example 8 (Referential Integrity). Attempt to insert the superhero "Aquaman" with the superpower "telepathy."

```
INSERT INTO Superhero
VALUES      ('Aquaman',
             'telepathy');
```

Again we see the RDBMS protecting the integrity of the data.

```
ERROR:  insert or update on table "superhero" violates foreign
key constraint "superhero_name_fkey"
DETAIL:  Key (name)=(Aquaman) is not present in table "person".
```

In addition to enforcing consistency, the relational model is capable of providing higher-level views of the data through aggregation.

Example 9 (Aggregation (Simple)). Find the number of friends Superman has.

```
SELECT COUNT(*)
FROM   Link
WHERE  name = 'Superman'
       AND relationtype = 'friend';
```

Example 10 (Aggregation (Group By)). List the number of enemies of each Person.

```
SELECT name,
       Count(name)
FROM   Link
WHERE  relationtype = 'foe'
GROUP  BY name;
```

The above query produces the following output.

name	count
Superman	1
General Zod	1

Information stored within a properly designed database is normalized. That is, no information is repeated.

Example 11 (Normalization). For example, suppose the planet Krypton is discovered to have been in the "Xeno" Galaxy rather than the "Andromeda" Galaxy. If this information were not normalized, each person whose birthplace was Krypton would need to be updated. Since this information is normalized, the following query will suffice.

```
UPDATE planet
SET    galaxy = 'Xeno'
WHERE  name = 'Krypton';
```

The above examples are some of the most important reasons for choosing the relation model over others. Unfortunately the relational model is not without its downsides.

Cons

While the relational model excels at ensuring data consistency, aggregation, and reporting, it is not suitable for every task. In order to issue queries, a user must be familiar with the schema. This requires specific domain knowledge of the data.

Example 12. Find all enemies of superheroes from Earth.

The above query requires the use of two joins.

```
SELECT peer
FROM   Link
       JOIN Person
         ON Person.name = Link.name
       JOIN Planet
         ON Planet.name = Person.birthplace
WHERE  relationtype = 'foe'
       AND Planet.name = 'Earth';
```

A casual user is unlikely to determine the correct join path, name of the tables, name of the attributes, etc. This is in contrast to the document model, where the data is semi-structure or unstructured, requiring minimal domain knowledge.

The relational model is also rigid in structure. If a relation is modified, every query referencing said relation may require a rewrite. Even a simple attribute being renamed (e.g. $\rho_{\text{name/alias}}(\text{Person})$) is capable of modifying the join paths. This rigidity places additional cognitive burden on users.

In addition to having a rigid structure, most relational database management systems lack flexible string matching options. Assuming basic SQL-92 compliance, a RDBMS only supports the `LIKE` predicate [**sql-2011**].

Example 13 (`LIKE` Predicate). Find all people with a name ending in "man."

```
SELECT *
FROM   Person
WHERE  name LIKE '%man';
```

There are a couple of limitations to the `LIKE` predicate. First, it only supports basic substring matching. If a user accidentally searches for all people with a name ending in "men," nothing would be found.

Second, unless the column used in the predicate is indexed, performance may be poor ($\mathcal{O}(n)$).

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 as well as unstructured data. Examples of information suitable to the document model includes emails, memos, book chapters, etc.

These pieces, or units, of information are broken into documents. Groups of related documents (for example, a library catalogue) are referred to as a document collection.

Definition 8 (Terms and Document)**.** A term, τ, is an indivisible string (e.g. a proper noun, word, or a phrase). A document, d, is a bag of words. Let $\mathrm{freq}(\tau, d)$ be the frequency of terms τ in document d.

Let T denote all possible terms, and $\textsc{Bag}[\mathrm{T}]$ be all possible bag of terms.

Remark 1. We use the bag-of-words model for documents. This means that position information of terms in a document is irrelevant, but the frequency of terms are kept in the document. Documents are non-distinct sets.

Definition 9 (Document Collection)**.** A document collection $\mathbb{C}$ is a set of documents, written $D\mathbb{C} = \{d_1, d_2, \ldots, d_k\}$. The size of $\mathbb{C}$ is denoted N. The number of unique terms, or size of T, in $\mathbb{C}$, is denoted M.

Example 14. Consider the following short sentences.

1. Superman is strong on Earth and lives on Earth.
2. Batman was born on Earth.
3. Superwoman is fast on Earth.
4. Superman was born on Krypton.

Each sentence represents a document, giving us the following documents.

$$d_1 = \{\text{“and”} : 1, \text{“on”} : 2, \text{“is”} : 1, \text{“lives”} : 1, \text{“earth”} : 2, \text{“strong”} : 1, \text{“superman”} : 1\} \quad (2.7)$$

$$d_2 = \{\text{“batman”} : 1, \text{“on”} : 1, \text{“was”} : 1, \text{“earth”} : 1, \text{“born”} : 1\} \quad (2.8)$$

$$d_3 = \{\text{“on”} : 1, \text{“is”} : 1, \text{“superwoman”} : 1, \text{“fast”} : 1, \text{“earth”} : 1\} \quad (2.9)$$

$$d_4 = \{\text{“krypton”} : 1, \text{“born”} : 1, \text{“on”} : 1, \text{“was”} : 1, \text{“superman”} : 1\} \quad (2.10)$$

$$(2.11)$$

2.2.1 Vectorization of Documents

One of the most fundamental approach for search documents is to treat documents as high dimensional vectors, and the document collection as a subset in a vector space. The search query becomes a nearest neighbour query in a vector space equipped with a distance measure.

The first step is to convert bag of terms into vectors. The standard technique [**ir-08**] uses a scoring function that measures the relative importance terms in documents.

Definition 10 (TF-IDF Score)**.** The term frequency is the number of times a term τ appears in a document d, as given by $\mathrm{freq}(\tau, d)$. The document frequency of a term τ, denoted by $\mathrm{df}(\tau)$, is the number of documents in $\mathbb{C}$ that contains τ. It is defined as

$$\mathrm{df}(\tau) = |\{d \in \mathbb{C} : \tau \in d\}|$$

The combined TF-IDF score of τ in a document d is given by

$$\text{tf-idf}\left(\mathbb{C}, \tau, d\right) = \frac{\text{freq}\left(\tau, d\right)}{|d|} \cdot \log \frac{N}{\text{df}\left(\tau\right)}$$

Remark 2. The first component, $\frac{\text{freq}(\tau,d)}{|d|}$, measures the importance of a term within a document. It is normalized to account for document length. The second component, $\log \frac{N}{\text{df}(\tau)}$, is a measure of the rarity of the term within the document collection $\mathbb{C}$.

Example 15. Using the documents from Example 14 on page 11, the TF-IDF scores are as follows.

	d_1	d_2	d_3	d_4
τ_1 : "and"	0.2857	0.0000	0.0000	0.0000
τ_2 : "on"	0.0000	0.0000	0.0000	0.0000
τ_3 : "superwoman"	0.0000	0.0000	0.4000	0.0000
τ_4 : "batman"	0.0000	0.4000	0.0000	0.0000
τ_5 : "is"	0.1429	0.0000	0.2000	0.0000
τ_6 : "fast"	0.0000	0.0000	0.4000	0.0000
τ_7 : "born"	0.0000	0.2000	0.0000	0.2000
τ_8 : "krypton"	0.0000	0.0000	0.0000	0.4000
τ_9 : "earth"	0.1186	0.0830	0.0830	0.0000
τ_{10} : "lives"	0.2857	0.0000	0.0000	0.0000
τ_{11} : "strong"	0.2857	0.0000	0.0000	0.0000
τ_{12} : "was"	0.0000	0.2000	0.0000	0.2000
τ_{13} : "superman"	0.1429	0.0000	0.0000	0.2000

Definition 11 (Document Vector). Given a document collection $\mathbb{C}$ with M unique terms $\mathrm{T} = \left[\tau_1, \tau_2, \ldots, \tau_n\right]$, each document d can be represented by an M-dimensional vector.

$$\vec{d} = \begin{bmatrix} \text{tf-idf}\left(\tau_1, d\right) \\ \text{tf-idf}\left(\tau_2, d\right) \\ \vdots \\ \text{tf-idf}\left(\tau_n, d\right) \end{bmatrix}$$

Example 16. The documents in Example 14 on page 11 would produce the following vectors.

$$\vec{d}_n = \begin{bmatrix} \text{tf-idf}\left(\tau_1, d_n\right) \\ \text{tf-idf}\left(\tau_2, d_n\right) \\ \text{tf-idf}\left(\tau_3, d_n\right) \\ \text{tf-idf}\left(\tau_4, d_n\right) \\ \text{tf-idf}\left(\tau_5, d_n\right) \\ \text{tf-idf}\left(\tau_6, d_n\right) \\ \text{tf-idf}\left(\tau_7, d_n\right) \\ \text{tf-idf}\left(\tau_8, d_n\right) \\ \text{tf-idf}\left(\tau_9, d_n\right) \\ \text{tf-idf}\left(\tau_{10}, d_n\right) \\ \text{tf-idf}\left(\tau_{11}, d_n\right) \\ \text{tf-idf}\left(\tau_{12}, d_n\right) \\ \text{tf-idf}\left(\tau_{13}, d_n\right) \end{bmatrix}, \vec{d}_1 = \begin{bmatrix} 0.2857 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.1429 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.1186 \\ 0.2857 \\ 0.2857 \\ 0.0000 \\ 0.1429 \end{bmatrix}, \vec{d}_2 = \begin{bmatrix} 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.4000 \\ 0.0000 \\ 0.0000 \\ 0.2000 \\ 0.0000 \\ 0.0830 \\ 0.0000 \\ 0.0000 \\ 0.2000 \\ 0.0000 \end{bmatrix}, \vec{d}_3 = \begin{bmatrix} 0.0000 \\ 0.0000 \\ 0.4000 \\ 0.0000 \\ 0.2000 \\ 0.4000 \\ 0.0000 \\ 0.0000 \\ 0.0830 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.0000 \end{bmatrix}, \vec{d}_4 = \begin{bmatrix} 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.2000 \\ 0.4000 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.2000 \\ 0.2000 \end{bmatrix}$$

Definition 12 (Search Query). A search query q is simply a document, namely a bag of terms. The top-k answers to q with respect to a collection dc is defined as the k documents, $\{d_1, d_2, \ldots, d_k\}$, in $\mathbb{C}$, such that $\{\vec{d}_i\}$ are the closest vectors to $\vec{q}$ using Euclidean distance measure in $\mathbb{R}^N$.

Example 17. Given the search query $q = \{\text{superwoman}, \text{was}, \text{born}, \text{on}, \text{krypton}\}$, compute the vector $\vec{q}$ within the document collection $\mathbb{C}$ (as defined in Example 14 on page 11).

$$\vec{q} = \begin{bmatrix} 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.0000 \\ 0.1474 \\ 0.2644 \\ 0.0000 \\ 0.2644 \\ 0.1474 \\ 0.0000 \end{bmatrix}$$

In order to determine the top-k documents for search query q, we need a way of measuring the similarity between documents.

Definition 13 (Cosine Similarity). Given two document vectors, $\vec{d}_1$ and $\vec{d}_2$, the cosine similarity is the dot product $\vec{d}_1 \cdot \vec{d}_2$, normalized by the product of the Euclidean distance of $\vec{d}_1$ and $\vec{d}_2$ in $\mathbb{R}^N$. It is denoted as $\text{similarity}(\vec{d}_1, \vec{d}_2)$.

$$\text{similarity}(\vec{d}_1, \vec{d}_2) = \frac{\vec{d}_1 \cdot \vec{d}_2}{\|\vec{d}_1\| \cdot \|\vec{d}_2\|} \tag{2.12}$$

$$= \frac{\sum_{i=1}^{N} \vec{d}_{1,i} \times \vec{d}_{2,i}}{\sqrt{\sum_{i=1}^{N} (\vec{d}_{1,i})^2} \times \sqrt{\sum_{i=1}^{N} (\vec{d}_{2,i})^2}} \tag{2.13}$$

Recall we may represent search queries as documents and thus document vectors. Therefore we may compute the score of a document d for a search query q as

$$\text{similarity}\left(\vec{d}, \vec{q}\right)$$

Example 18. Given the document collection $\mathbb{C}$ (from Example 14 on page 11) and search query q, compute the similarity between q and every document $d \in \mathbb{C}$.

$$\text{similarity}\left(\vec{d}_1, \vec{q}\right) = 0.000000 \tag{2.14}$$

$$\text{similarity}\left(\vec{d}_2, \vec{q}\right) = 0.191533 \tag{2.15}$$

$$\text{similarity}\left(\vec{d}_3, \vec{q}\right) = 0.265877 \tag{2.16}$$

$$\text{similarity}\left(\vec{d}_4, \vec{q}\right) = 0.618553 \tag{2.17}$$

2.2.2 Extending the Document Model

In the extended document model, documents have fields: $\textsc{Field}\left[d\right]$, and each attribute have values (e.g. date, string, integer), or bag of terms. Thus:

$$d \ : \ \textsc{Field}\left[d\right] \rightarrow \textsc{Bag}\left[\mathrm{T}\right]$$

Example 19 (Semi-Structured Document)**.** We see that d_2 is about Batman. The document contents are semi-structured, containing both a name and the name of a planet. By adding fields to the document, we are left with Table 2.3.

Field	Value
name	Batman
birthplace	Earth
body	Batman was born on Earth.

Table 2.3: Person document for Batman

which is similar in structure to the `Person` table.

2.2.3 Approximate String matching

Definition 14 (N-Gram). An n-Gram is a contiguous sequence of substrings of string S of length n. An algorithm for computing the n-gram of S is given in Algorithm 1.

Algorithm 1 N-Gram(S, n, s)

Require: S is a string, $n \geq 1$, and s is a character

Ensure: the list of n-grams of S

1: $G \leftarrow []$
2: $p \leftarrow \text{Repeat}(s, n-1)$
3: $S \leftarrow \text{Pad}(S, p)$
4: $S \leftarrow \text{Replace}(S, \text{' '}, p)$
5: **for** $i = 0$ **to** $l - n + 1$ **do**
6: append $S[i, i+n]$ to G
7: **end for**
8: **return** G

Where l is the length of S, $\text{Repeat}(S, n)$ repeats s character n times, $\text{Pad}(S, p)$ prefixes and postfixes S with p, and $\text{Replace}(S, s, p)$ replaces character s with p in string S.

Example 20. Given a string $S =$ "superman", compute the trigram of S using Algorithm 1.

$$G = \{\text{"\$\$s"}, \text{"\$su"}, \text{"sup"}, \text{"upe"}, \text{"per"}, \text{"erm"}, \text{"rma"}, \text{"man"}, \text{"an\$"}, \text{"n\$\$"}\}$$

We use n-grams in order to permit approximate string matching.

Example 21. Given a string S (Example 20), let $S' =$ "superwoman". Compute the trigram of S' and compare it to S.

$$G' = \{\text{"\$\$s"}, \text{"\$su"}, \text{"sup"}, \text{"upe"}, \text{"per"}, \text{"erw"}, \text{"rwo"}, \text{"wom"}, \text{"oma"}, \text{"man"}, \text{"an\$"}, \text{"n\$\$"}\}$$

$$
\begin{array}{lcc}
 & G & G' \\
\tau_1 : \text{"an\$"} & 1 & 1 \\
\tau_2 : \text{"oma"} & 0 & 1 \\
\tau_3 : \text{"\$su"} & 1 & 1 \\
\tau_4 : \text{"rwo"} & 0 & 1 \\
\tau_5 : \text{"rma"} & 1 & 0 \\
\tau_6 : \text{"man"} & 1 & 1 \\
\tau_7 : \text{"erw"} & 0 & 1 \\
\tau_8 : \text{"\$\$s"} & 1 & 1 \\
\tau_9 : \text{"upe"} & 1 & 1 \\
\tau_{10} : \text{"n\$\$"} & 1 & 1 \\
\tau_{11} : \text{"per"} & 1 & 1 \\
\tau_{12} : \text{"wom"} & 0 & 1 \\
\tau_{13} : \text{"sup"} & 1 & 1 \\
\tau_{14} : \text{"erm"} & 1 & 0
\end{array}
$$

Figure 2.3: Comparison between n-grams of G and G'.

Comparing G to G' results in the following matrix

As Figure 2.3 shows, using n-grams yield a similarity of $\frac{8}{14}$.

2.2.4 Pros and Cons of the Document Model

There are numerous reasons to use the document model. It allows users without domain knowledge and working knowledge of a complex query language such as SQL to find information.

Example 22 (Simple Queries)**.** Find all documents related to "Superman" or "Earth". This query, if the default operator is `OR`, would simply be `Superman Earth`. The result of the query q would be

$$\text{query}(\text{“superman”}) \cup \text{query}(\text{“earth”}) \rightarrow \{d_1, d_2, d_3, d_4\}$$

Users can also modify queries to require certain terms be present or not present.

Example 23 (`AND` Query). Find all documents containing both “Superman” and “Earth”. This query would return the following set of documents

$$\text{query}(\text{“superman”}) \cap \text{query}(\text{“earth”}) \rightarrow \{d_1\}$$

as only d_1 contains both terms.

Example 24 (`NOT` Query). Find all documents containing “Superman” but not “Earth”. This query would return different results than Example 23.

$$\text{query}(\text{“superman”}) \neg \text{query}(\text{“earth”}) \rightarrow \{d_4\}$$

While none of the above queries required domain knowledge, it is possible to use the extended document model (Section 2.2.2 on page 16) to search specific fields. Doing so allows users to have finer control over what documents are retrieved.

Example 25 (Extended Query). Find all documents with a superhero named “Superman” that contain the term “Earth”.

$$\text{query}(\text{“name”}, \text{“superman”}) \cap \text{query}(\text{“earth”}) \rightarrow \{d_3\}$$

Assuming the first term of every document is also the value of the name attribute.

People utilize keyword query search every day through web search engines such as Google[1].

Not only does the document model provide a familiar interface to search for information with, it also ranks the results. In the relational model a search for “Superman” would return all named tuples

[1] https://www.google.ca/

that contained that term. In the document model, documents are ranked against the query q and the top-k documents are returned.

The advantage is that users have the result of q already ranked so only the most relevant documents may be explored. As the number of documents matching q for a large corpus can be high, showing only the top-k relevant documents may save the user a substantial amount of time.

The relational model does not permit approximate string matching. By utilizing the document model with n-grams (Section 2.2.3 on page 17), users who substitute, delete, or insert characters from the desired term may still receive results for their intended term (see Example 21 on page 17 for a demonstration of how n-grams overcome character substitutions).

Unfortunately the document model does not support the concept of foreign keys (Definition 4 on page 4). While information is easily accessible due to flexible search, each document is a discrete unit of information. Aggregate queries are unsupported, as these units are not linked amongst one another.

Chapter 3

Best of Both Worlds

3.1 Encoding Named Tuples into Documents

Recall in the extended document model (Section 2.2.2 on page 16), a document d consists of fields $f_1, f_2, \ldots, f_n$. Using the extended document model, we are left with a straight forward mapping of a tuple t to document d.

For tuple t, every attribute $\alpha \in \textsc{Attr}[t]$ maps to field f in d. Every attribute value must be analyzed into an indexable form in order to store it in a field.

$$\textsc{Attr}[t] \xrightarrow{analyzed} \textsc{Field}[d] \tag{3.1}$$

$$\alpha_1, \alpha_2, \ldots, \alpha_n \xrightarrow{analyzed} f_1, f_2, \ldots, f_n \tag{3.2}$$

We denote the document encoding of t as $\textsc{Doc}[t]$.

3.2 Mapping of Entity Groups to Documents

Recall that an entity group (Definition 7 on page 6) is a forest G of tuples t such that

$$\forall (t, t') \in G, t \neq t' \Rightarrow \textsc{Rel}[t] \neq \textsc{Rel}[t']$$

That is, all tuples are from distinct relations.

Given the restriction

$$\forall (r, r') \in \textsc{SchemaGraph}[\mathbb{D}], \exists! (r, r') \vDash \theta$$

we assert that if $(t, t') \in G$, then $(t, t') \in \textsc{Rel}[t] \bowtie \textsc{Rel}[t']$.

Let $V(G)$ be the vertices of G, $E(G)$ be the edges of G.

Claim 1. G can always be reconstructed from $V(G)$ without loss of information.

Proof. Given $V(G)$, we must reconstruct $E(G)$ in order to complete G.

Choose any $(t, t') \in V(G)$. If $(\textsc{Rel}[t], \textsc{Rel}[t']) \in \textsc{SchemaGraph}[\mathbb{D}]$, then (t, t') is an edge in G.

Recall our earlier assertion that $\textsc{SchemaGraph}[\mathbb{D}]$ is cycle-free and foreign keys must be unique. □

3.3 Encoding an Entity Group as a Document Group

Given a entity group G, we construct two or more documents in order to represent the entity group in the document model.

For every $t \in V(G)$, we construct a document $\textsc{Doc}[t]$ (Section 3.1 on the preceding page). With each tuple t stored in the document collection $\mathbb{C}$, we construct an additional document which stores the association information.

Let x be the indexing document of G.

$$x[\text{"entities"}] = \bigcup_{t \in V(G)} \text{UID}[t]$$

Thus, the encoding of G is defined as

$$G \xrightarrow{\text{encode}} \left\{ \textsc{Doc}[t] : t \in V(G) \right\} \cup \left\{ x \right\}$$

It's easy to see that from encode(G) we can recover V(G), the tuples in G.

By Claim 1 on the previous page, this is sufficient to recover G entirely.

3.4 Encoding Attribute Values into Searchable Documents

Each value for user selected attributes are converted into *n*-grams, and stored in special documents.

3.5 Iterative Search Using Document Encodings

A document database supports fast and flexible keyword search queries. A search query is characterized by $q = (f, w)$, where f is a field name, and w is a search phrase.

Search[q] is the set of documents returned by the text index. The search function allows us to do many things:

1. Suggest values, correcting spelling errors
2. Given attribute values, search all relevant entities
3. Search for all relevant entity groups of one or more entities, using the indexing document
4. We can connect entities via entity groups using (hyper)graph search algorithms.

Chapter 4

Along Came Clojure

Talk about how great Clojure is.

4.1 Basic principles of functional programming (2 days)

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

4.1.1 Features of Clojure (2 days)

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

4.2 Search w/ Clojure

4.2.1 Thirdparty libraries (1 day, week 4)

- Lucene

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

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

4.2.3 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

4.2.4 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

5.1 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

Scripts were written to coordinate the execution, collection, and transformation of the performance data of our implementation.

5.3.1 Methodology

We used Criterium[1] to handle the execution of the benchmarks as it handles unique concerns stemming from benchmarking on the JVM. These include:

- Statistical processing of multiple evaluations
- Inclusion of a warm-up period, designed to allow the JIT compiler to optimize its code
- Purging of the garbage collector before testing, to isolate timings from GC state prior to testing
- A final forced GC after testing to estimate impact of cleanup on the timing results

Unfortunately this requires a much longer runtime as each function must be invoked numerous times. In extreme cases (Ford-Fulkerson, 8 hops) this can take upwards of 4 hours in our test environment.

Data Collection

Criterium provides us with a Clojure map with performance data. It performs analysis, presenting us with outliers, samples, etc. As this data collection process can take several hours or more, this data is collected and stored for offline analysis.

In order to utilize the Clojure output in Python, a data interchange format (JSON) is used. The benchmark function writes the Criterium performance analysis out as a JSON string to stdout and Python captures the output, JSONifies it, and stores it in an array. This array is written to disk in JSON as well so it can be loaded into the data transformation script.

For example:

```
[{"results": {...}, "method": "bfs", "max-hops": 1}, ...]
```

[1] http://hugoduncan.org/criterium/

Data Processing

Several scientific computing libraries are used in the processing and visualization.

There are two forms the data takes:

- comma-separated values (CSV)
- JavaScript object notation (JSON)

The CSV data is generated from the JSON data which is generated as described in Section 5.3.1 on the preceding page.

With the data loaded, we're interested in a handful of pieces of date per each entry.

- Max hops
- Method
- Mean execution time

We can easily load and parse the JSON data.

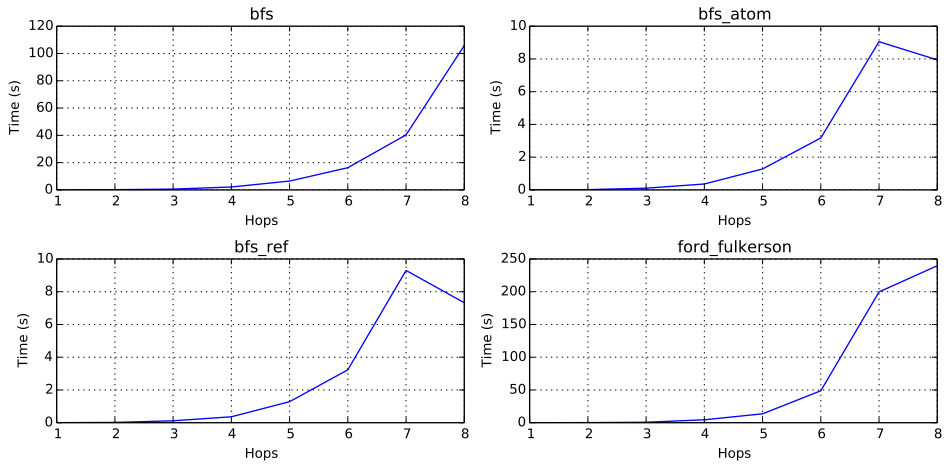


Figure 5.1: Growth of graph search times based on number of hops, plotted separately

- Index speed

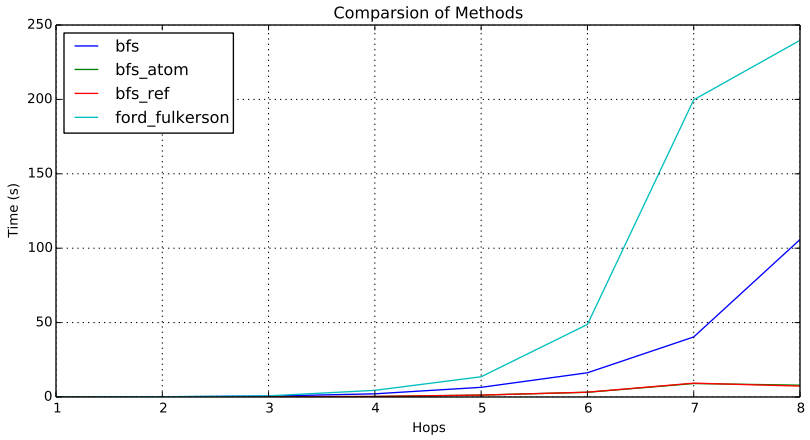


Figure 5.2: Growth of graph search times based on number of hops, combined plot

- 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.

To do...

☐ 1 (p. 7): Revise this figure