

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

© Richard J.I. Drake, 2013

Contents

1	Background (2 days)	1
2	A Tale of Two Data Models	2
2.1	Relational Model	2
2.1.1	Schema Group	4
2.1.2	Entity Group	6
2.1.3	Pros and Cons of the Relational Model	7
2.2	Document Model	11
2.2.1	Vectorization of Documents	12
2.2.2	Extending the Document Model	16
2.2.3	Approximate String matching	17
2.2.4	Pros and Cons of the Document Model	18
3	Best of Both Worlds	21
3.1	Encoding Named Tuples into Documents	21
4	Along Came Clojure	22
4.1	Basic principles of functional programming (2 days)	22
4.1.1	Features of Clojure (2 days)	22
4.2	Search w/ Clojure	23
4.2.1	Thirdparty libraries (1 day, week 4)	23

4.2.2	Indexing of relational objects (5 days, week 5)	23
4.2.3	Keyword Search in document space (5 days, week 6)	23
4.2.4	Graph Search in document space (5 days, week 7)	23
5	Experimental evaluation (5 days, week 8)	24
5.1	Implementation	24
5.2	The data set	24
5.3	Runtime Evaluation	24
5.4	Lessons learned	25
6	Conclusion (0 days)	26

List of Tables

2.1	Person table	3
2.2	Superman’s properties	7
2.3	Person document for Batman	16

List of Figures

2.1	Query to find superheroes with a home to go back to	6
2.2	Superhero entity group	7
2.3	Comparison between n -grams of G and G'	18

List of Algorithms

1	N-GRAM(S, n, s)	17
---	-------------------------------	----

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 $t = \{\text{name} : \text{“Superman”}, \text{age} : 33, \text{birthplace} : \text{“Krypton”}\}$, we denote the attributes of t as $\text{ATTR}[t] = \{\text{name}, \text{age}, \text{birthplace}\}$. The values are $t[\text{name}] = \text{“Superman”}$,

$t[\text{age}] = 33$, and $t[\text{birthplace}] = \text{"Krypton"}.$

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

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

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

$$r = \left\{ \begin{array}{l} \{\text{name} : \text{"Superman"}, \text{age} : 33, \text{birthplace} : \text{"Krypton"}\}, \\ \{\text{name} : \text{"Batman"}, \text{age} : 30, \text{birthplace} : \text{"Earth"}\}, \\ \{\text{name} : \text{"Flash"}, \text{age} : 53, \text{birthplace} : \text{"Earth"}\}, \\ \{\text{name} : \text{"Wonder Woman"}, \text{age} : 28, \text{birthplace} : \text{"Earth"}\}, \\ \{\text{name} : \text{"General Zod"}, \text{age} : 42, \text{birthplace} : \text{"Krypton"}\} \end{array} \right\}$$

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 $\text{ATTR}[r]$ which may uniquely identify a tuple. Formally, we say r satisfies the key constraint K , denoted as $r \models K$, subject to

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

For example, in Table 2.1, the relation satisfies the key constraint {name}, but not {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

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

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 $(r_1, r_2) \models \theta$, if

$$\forall t \in r_1, \exists t' \in r_2 \mid t[\alpha_{11}, \alpha_{12}, \dots, \alpha_{1k}] = t'[\alpha_{21}, \alpha_{22}, \dots, \alpha_{2k}]$$

Example 3. Suppose we have a relation Superhero (name, superpower). We can impose a **FK** constraint of

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

Definition 5 (Relational Database). A relational database, \mathbb{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 $\text{NAME}[\mathbb{D}]$ 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* G , formally defined as

$$\text{vertices} : V(G) = \text{REL}[\mathbb{D}]$$

$$\text{edges} : E(G) = \text{FK}[\mathbb{D}]$$

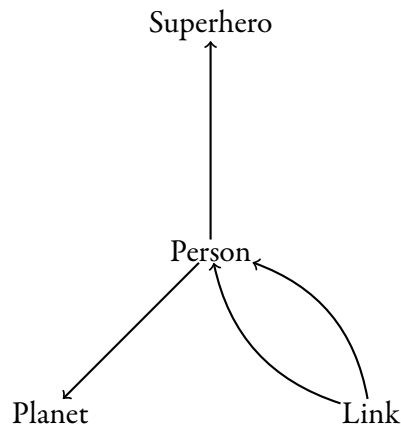
Example 4. Given the following schema

Superhero (name, power)
 Person (name, age, birthplace)
 Planet (name, size, age, destroyed, galaxy)
 Link (name, peer, relationtype)

and the following foreign key constraints

Superhero (name) → Person (name)
 Person (birthplace) → Planet (name)
 Link (name) → Person (name)
 Link (peer) → Person (name)

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.

Results in the following output, given the data from Table 2.1,

```

1 SELECT Person.name
2 FROM   Person
3         JOIN Planet
4         ON Person.birthplace = Planet.name
5 WHERE  NOT Planet.destroyed;

```

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

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

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 all relates to Superman, however no single tuple in the database has all of this information as a result of database normalization.

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

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

```
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.”

```
1 INSERT INTO Person
2 VALUES      ('Superman',
```

```
3         35,  
4         '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.”

```
1 INSERT INTO Superhero  
2 VALUES      ('Aquaman',  
3              '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.

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

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

```
1 SELECT name,  
2         Count(name)  
3 FROM   Link  
4 WHERE   relationtype = 'foe'  
5 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.

```
1 UPDATE planet  
2 SET    galaxy = 'Xeno'  
3 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.

```
1 SELECT peer
2 FROM   Link
3       JOIN Person
4         ON Person.name = Link.name
5       JOIN Planet
6         ON Planet.name = Person.birthplace
7 WHERE  relationtype = 'foe'
8       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.”

```
1 SELECT *
2 FROM   Person
3 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 $\text{freq}(\tau, d)$ be the frequency of terms τ in document d .

Let T denote all possible terms, and $\text{BAG}[T]$ 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 $DC = \{d_1, d_2, \dots, d_k\}$. 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.

$$d_1 = \{ \text{"and"} : 1, \text{"on"} : 2, \text{"is"} : 1, \text{"lives"} : 1, \text{"earth"} : 2, \text{"strong"} : 1, \text{"superman"} : 1 \}$$

$$d_2 = \{ \text{"batman"} : 1, \text{"on"} : 1, \text{"was"} : 1, \text{"earth"} : 1, \text{"born"} : 1 \}$$

$$d_3 = \{ \text{"on"} : 1, \text{"is"} : 1, \text{"superwoman"} : 1, \text{"fast"} : 1, \text{"earth"} : 1 \}$$

$$d_4 = \{ \text{"krypton"} : 1, \text{"born"} : 1, \text{"on"} : 1, \text{"was"} : 1, \text{"superman"} : 1 \}$$

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 $\text{freq}(\tau, d)$. The document frequency of a term τ , denoted by $\text{df}(\tau)$, is the number of documents in \mathbb{C} that contains τ . It is defined as

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

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

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

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, 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 $T = [\tau_1, \tau_2, \dots, \tau_n]$, each document d can be represented by an M -dimensional vector.

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

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

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

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, $\{d_1, d_2, \dots, d_k\}$, 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 $q = \{\text{superwoman, was, born, on, krypton}\}$, compute the vector \vec{q} within the document collection \mathbb{C} (as defined in Example 14).

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

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 $\text{similarity}(\vec{d}_1, \vec{d}_2)$.

$$\text{similarity}(\vec{d}_1, \vec{d}_2) = \frac{\vec{d}_1 \cdot \vec{d}_2}{\|\vec{d}_1\| \cdot \|\vec{d}_2\|} \quad (2.1)$$

$$= \frac{\sum_{i=1}^N \vec{d}_{1,i} \times \vec{d}_{2,i}}{\sqrt{\sum_{i=1}^N (\vec{d}_{1,i})^2} \times \sqrt{\sum_{i=1}^N (\vec{d}_{2,i})^2}} \quad (2.2)$$

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

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

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

$$\text{similarity}(\vec{d}_1, \vec{q}) = 0.000000 \quad (2.3)$$

$$\text{similarity}(\vec{d}_2, \vec{q}) = 0.191533 \quad (2.4)$$

$$\text{similarity}(\vec{d}_3, \vec{q}) = 0.265877 \quad (2.5)$$

$$\text{similarity}(\vec{d}_4, \vec{q}) = 0.618553 \quad (2.6)$$

2.2.2 Extending the Document Model

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

$$d : \text{FIELD}[d] \rightarrow \text{BAG}[T]$$

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:  $S \leftarrow \text{REPLACE}(S, ' ', p)$ 
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 $S = \text{"superman"}$, compute the trigram of S using Algorithm 1. Lines 1-4 would yield $S = \text{"$$superman$$"}$, resulting in $l = 12$.

$$G = \{ \text{"$$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 $S' = \text{"superwoman"}$. 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$$"} \}$$

Comparing G to G' results in the following matrix

	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

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

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) 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 Google¹.

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), users who substitute, delete, or insert characters from the desired term may still receive results for their intended term (see Example 21 for a demonstration of how n -grams overcome character substitutions).

Unfortunately the document model does not support the concept of foreign keys (Definition 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), a document d consists of fields f_1, f_2, \dots, 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 $a \in \text{ATTR}[t]$ maps to field f in d . Every attribute value must be analyzed into an indexable form in order to store it in a field.

$$\begin{array}{ccc} \text{ATTR}[t] & \xrightarrow{\text{analyzed}} & \text{FIELDS}[d] \\ a_1, a_2, \dots, a_n & \xrightarrow{\text{analyzed}} & f_1, f_2, \dots, f_n \end{array}$$

We denote the document encoding of t as $\text{DOC}[t]$.

- 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 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 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.

To do...

- ☐ 1 (p. 7): Revise this figure