Column Stores as an IR Prototyping Tool

Hannes Mühleisen[1], Thaer Samar[1], Jimmy Lin[2], and Arjen de Vries[1]

[1] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[2] University of Maryland, College Park, Maryland, USA

{hannes|samar}@cwi.nl, jimmylin@umd.edu, arjen@acm.org

Abstract. We make the suggestion that instead of implementing custom index structures and query evaluation algorithms, IR researchers should simply store document representations in a column-oriented relational database and write ranking models using SQL. For rapid prototyping, this is particularly advantageous since researchers can explore new ranking functions and features by simply issuing SQL queries, without needing to write imperative code. We demonstrate the feasibility of this approach by an implementation of conjunctive BM25 using MonetDB on a part of the ClueWeb12 collection.

1 Introduction

Information retrieval researchers and practitioners have long implemented specialized data structures and algorithms for document ranking [6]. Today, these techniques can be quite complex, especially with "structured queries" that span multiple nested clauses with a panoply of query operators [4]. We make the suggestion that the information retrieval community should look towards the database community for general purpose data management solutions—namely, column-oriented relational databases (or column stores). We show that one such system, MonetDB, can be used for storing postings lists and that conjunctive query evaluation with BM25 can be translated into an SQL query. We advocate this approach especially for rapid prototyping in an IR research setting.

What advantages does using a database have? We see many, beginning with a precise formal framework. Relational databases provide a formal and theoretically-sound framework in which to express any query evaluation algorithm—namely, relational calculus (or, practically, SQL). This forces IR researchers to be precise about query semantics, which may be especially useful when complex query operators are introduced in document ranking.

Second, taking advantage of relational databases yields a cleaner architecture. Almost all IR systems today are monolithic agglomerations of components for text processing, document inversion, integer compression, memory/disk management, query evaluation, feature extraction, machine learning, etc. By offloading the storage management to a relational database, we introduce a clean abstraction (via SQL) between the "low-level" components of the engine and the IR-specific components (e.g., learning to rank). This (hopefully) reduces overall system complexity and may allow different IR engines to inter-operate.

Third, retrieval systems can benefit from advances in data management. In IR, efficiency is a relatively niche topic. In contrast, performance is a dominant preoccupation of database researchers, who make regular breakthroughs that eventually propagate to IR (for example, PForDelta [7], the best practice for index compression today, originated from database researchers). By using relational databases, IR systems can benefit from future advances "for free".

Finally, databases form a flexible rapid prototyping tool. Many IR researchers do not really care about index structures and query evaluation *per se*—they are merely means to an end, such as assessing the effectiveness of a particular ranking function. In this case, forcing researchers to design data structures and query evaluation algorithms is a burden. Using a relational database, researchers can rapidly experiment by issuing declarative SQL queries without needing to write (error-prone) imperative or object-oriented code.

There are many similarities between query evaluation in document retrieval and online analytical processing (OLAP) tasks in modern data warehouses. Both frequently involve scans, aggregations, and sorting. Thus, we believe that column-oriented databases, which excel at OLAP queries, are amenable to retrieval tasks. An overview of such databases is beyond the scope of this work, but the basic insight is to decompose relations into columns for storage and processing [2]. In this paper, we use exactly such a system, MonetDB, to illustrate document ranking on a portion of the ClueWeb12 collection.

2 Okapi BM25 in a Relational Database

All IR engines pre-process the input collection prior to indexing, and this is no different when using a relational database. For this task, we take advantage of Hadoop MapReduce to convert a document collection into a collection of relational tables. We performed tokenization and stemming using the Krovetz stemmer with a stopwords list from Terrier. The stemmed and filtered terms were mapped to integer ids and stored in a dictionary table. In the main terms table, we store all terms in a document (by term id), along with the position in which they occur. To give a concrete example, consider the document `doc1` with the content "I put on my robe and wizard hat". If we assume "I", "on", "my" and "and" are stopwords, the relational tables generated from this document have the following content:

table: dict

term	termid	df
put	1	1
robe	2	1
wizard	3	1
hat	4	1

table: terms

termid	docid	pos
1	1	2
2	1	5
3	1	7
4	1	8

table: docs

docid	name	len
1	doc1	8

We have chosen Okapi BM25 to implement as an SQL query, but our approach can be easily extended to other ranking functions. Our experiments focused on conjunctive query evaluation, where the document must contain all terms;

previous work [1] has shown that this approach yields comparable end-to-end effectiveness to disjunctive query evaluation, but is substantially faster. When scoring documents based on BM25, the only score component that depends on the query–document combination is the term frequency $f(q_i, D)$. The document length $|D|$ for each document and the document frequency $n(q_i)$ for each term can be precomputed. By sorting the terms table (from above) by term id (in effect performing document inversion), we can avoid scanning the table entirely. The complete ranking function can be expressed as follows:

```
1  WITH qterms AS (SELECT termid, docid FROM terms
2      WHERE termid IN (10575, 1285, 191)),
3  subscores AS (
4  SELECT docs.docid, len, term_tf.termid,
5      tf, df, (log((45174549-df+0.5)/(df+0.5))* ((tf*(1.2+1)/(tf+1.2*(1-
6      0.75+0.75*(len/513.67)))))) AS subscore
7      FROM (SELECT termid, docid, COUNT(*) AS tf FROM qterms
8      GROUP BY docid, termid) AS term_tf
9      JOIN (SELECT docid FROM qterms
10         GROUP BY docid HAVING COUNT(DISTINCT termid) = 3)
11         AS cdocs ON term_tf.docid=cdocs.docid
12     JOIN docs ON term_tf.docid=docs.docid
13     JOIN dict ON term_tf.termid=dict.termid)
14 SELECT name, score FROM (SELECT docid, sum(subscore) AS score
15     FROM subscores GROUP BY docid) AS scores JOIN docs ON
16     scores.docid=docs.docid ORDER BY score DESC LIMIT 1000;
```

To explain, we map conjunctive BM25 ranking to SQL in three parts: First, we find the entries in the terms table for query terms (Lines 1 and 2).[3] In this case, the query terms have ids 10575, 1285, and 191. The second step calculates the individual scores for all term/document combinations (Lines 4-13). To express the conjunctivity in the query, we filter this intermediate result to only include term/document combinations with exactly three different term ids for each document. (Lines 9 to 11). We also collect information about document ids and lengths (Line 12) as well as the document frequency of the terms (Line 13). Then, we calculate the individual BM25 scores for each term/document combination (Lines 5 and 6) and sum the results up and sort (Lines 14 to 16). The numbers printed in bold are the only parts of the SQL query that depend on the document collection and the query terms.

[3] Although in theory it would be possible to "join in" the dictionary by term, in practice, performing the mapping from query terms to identifiers turns out to be preferable outside SQL.

Note that the entire ranking function is expressed as an SQL query and this approach can be extended to any scoring function that is a sum of matching query terms. Disjunctive query evaluation can be implemented by replacing inner joins with outer joins. Phrase queries are performed by simple arithmetic over term positions and relative phrase positions. In other words, IR researchers can rapidly explore different retrieval options without writing imperative code.

We have implemented the above ranking function using the open-source columnar database MonetDB [3]. Our experiments used the first segment on the first disk of ClueWeb12 (∼45 million documents). We ran queries 201–250 from the TREC 2013 web track and verified that both query latency and effectiveness are comparable to existing open-source engines such as Terrier [5] and Indri [4].

3 Demonstration Description

Our demonstration allows a user to interactively compare our MonetDB solution to existing open-source search engines. Users submit queries in a web interface, which are concurrently dispatched to multiple backends. The interface displays results from each backend as soon as it completes, allowing the user to compare the performance of each search engine along with the effectiveness of the results.

Acknowledgments. This research was supported by the Netherlands Organization for Scientific Research (NWO project 640.005.001) and the Dutch national program `COMMIT/`.

References

1. Asadi, N., Lin, J.: Effectiveness/efficiency tradeoffs for candidate generation in multi-stage retrieval architectures. In: SIGIR. pp. 997–1000 (2013)
2. Copeland, G., Khoshafian, S.: A decomposition storage model. In: SIGMOD (1985)
3. Idreos, S., Groffen, F., Nes, N., Manegold, S., Mullender, K.S., Kersten, M.L.: MonetDB: Two decades of research in column-oriented database architectures. IEEE Data Engineering Bulletin 35(1), 40–45 (2012)
4. Metzler, D., Croft, W.B.: Combining the language model and inference network approaches to retrieval. IP&M 40(5), 735–750 (2004)
5. Ounis, I., Amati, G., Plachouras, V., He, B., Macdonald, C., Lioma, C.: Terrier: A high performance and scalable information retrieval platform. In: SIGIR Workshop on Open Source Information Retrieval (2006)
6. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Computing Surveys 38(6), 1–56 (2006)
7. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-scalar RAM-CPU cache compression. In: ICDE. pp. 59–59 (2006)