Old Dogs Are Great at New Tricks: Column Stores for IR Prototyping

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 implement ranking models using SQL. For rapid prototyping, this is particularly advantageous since researchers can explore new scoring 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 two modern column stores. Experiments on a web collection show that a retrieval engine built in this manner achieves effectiveness and efficiency on par with custom-built retrieval engines, but provides many additional advantages, including cleaner query semantics, a simpler architecture, built-in support for error analysis, and the ability to exploit advances in database technology "for free".

Categories and Subject Descriptors: H.3.4 [Information Storage and Retrieval]: Systems and Software—Performance Evaluation

Keywords: Relational Databases; BM25

1. INTRODUCTION

Information retrieval researchers and practitioners have long implemented specialized, custom-built data structures and query evaluation algorithms for document ranking [15]. Today, these techniques can be quite complex, especially with "structured queries" that span multiple nested clauses with a panoply of query operators [11]. We revisit the idea that the information retrieval community can (and should!) make use of general-purpose data management solutions, instead of building specialized backend technology. We demonstrate that storing posting lists as relations and expressing ranking models as SQL queries is now a viable alternative to custom inverted indexes if we leverage modern column-oriented relational databases (or 'column stores').

SIGIR'14, July 6–11, 2014, Gold Coast, Queensland, Australia.

ACM 978-1-4503-2257-7/14/07 ...$15.00.
http://dx.doi.org/10.1145/2600428.2609460.

The contribution of this work is to demonstrate empirically that we can now safely delegate the data management needs of IR engines to modern column stores without concerns regarding efficiency. We show experimentally that an IR engine built in this manner achieves not only effectiveness and efficiency on par with custom-built retrieval engines, but provides many additional advantages. This suggests a somewhat radical message: IR researchers should stop writing their own retrieval engines and just use column stores. We advocate this approach especially for rapid prototyping—when developing new scoring models, features functions, etc.—but such an architecture has additional advantages. Although we are not the first to make this claim, there have been a number of developments since previous work that make our design more attractive than before.

2. BACKGROUND AND RELATED WORK

What advantages does using a database for IR have? We see many, beginning with a precise formal framework. As more complex query operators are introduced for document ranking, it sometimes becomes unclear how to properly score documents, particularly in corner cases. Developers often resort to heuristics and other shortcuts. 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, 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. Performance is a dominant preoccupation of database researchers, who make regular breakthroughs that propagate to the IR community (for example, PForDelta [17] compression originated from database researchers). By using relational databases, IR systems can benefit from future advances more rapidly, and "for free".

Fourth, a database provides integrated analytical tools useful for error analysis. Consider a simple example, where

we wish to examine whether our scoring model has a length bias. This might be accomplished by examining scatterplots of length vs. retrieval scores. With a database, generating these data can be accomplished with a straightforward join between qrels (easy to store in the database), document representations, and the results set. In contrast, with a custom-built retrieval engine, one would need to write additional code to dump out document lengths from internal data structures and perform the join in an ad hoc fashion. Furthermore, there is a rich ecosystem of external analytical toolkits that are able to interface directly with relational databases, which could also be helpful for IR researchers.

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 model or feature. 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.

Since retrieval operates over collections of documents, analytics-optimized (OLAP) relational databases are more appropriate than transaction-optimized (OLTP) databases. 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 [4]. This storage model allows us to mask random access memory latencies and take full advantage of modern hardware. We use two different column stores, MonetDB [2, 9] and VectorWise [16], to illustrate document ranking on a portion of the ClueWeb12 collection.

We are not the first to claim that databases may have something to offer for IR. Examples of early work include [13, 10, 6]; key references on runtime efficiency are discussed in a 2005 survey [3]. Perhaps the first 'real' SQL results for IR queries were presented in [7], but at excessively high costs in terms of the hardware required. More viable results were presented in the TREC 2006 terabyte track, but the approach required hand-written query plans [8]. Follow-up research by the same group allowed the retrieval model to be expressed at the conceptual level [5], however, using a rather 'exotic' query language based on array comprehensions. After all these years, we have finally reached the state where database engines can take on IR workloads expressed in *standard* SQL, without forcing the database admin to resort to low-level tuning. As far as we know, this paper is the first report of experimental work where competitive IR results (in terms of both efficiency and effectiveness) have been obtained using standard relational database technology.

3. SYSTEM ARCHITECTURE

In a custom-built IR engine, a document collection must first be processed (e.g., tokenized) and indexed before retrieval can be performed. In an architecture based on column stores, there are equivalent steps: the collection must be processed and loaded into the database prior to query evaluation. This section provides details on our system architecture.

table: dict

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

Figure 1: IR data structures in a relational database.

3.1 Document Processing and Loading

We take advantage of the massive scale-out capabilities of Hadoop MapReduce to convert a document collection into a collection of relational tables. Document processing includes tokenization, stemming using the Krovetz stemmer, and stopword removal. The stemmed and filtered terms are 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". After stopword removal, the relational tables generated from this document are as shown in Figure 1. In more detail, these tables are first generated as flat text files using a two-pass approach on Hadoop; the first pass builds the term to term id mapping, and the second pass builds the terms and the docs tables. These flat text files are then bulk loaded into the database.

3.2 Query Evaluation

We implemented Okapi BM25 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 query terms; previous work [1] has shown that this approach yields comparable end-to-end effectiveness to disjunctive query evaluation, but is faster. When scoring documents based on BM25, the only score component that depends on the query is the term frequency $f(q_i, D)$. An obvious opportunity for optimization here would be to precalculate the term frequencies for each term/document combination, since the positions of the terms do not influence the ranking score for conjunctive queries. By also sorting the terms table by term id (in effect performing document inversion), the database is able to avoid scanning the entire table and instead use binary search, with greatly improved efficiency. The complete ranking function can be expressed in SQL, shown in Figure 2.

We map conjunctive BM25 ranking to SQL in three parts: First, we find the entries in the terms table for the query terms (Lines 1 and 2). In this case, the query terms have ids 10575, 1285, and 191.[1] 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 include only combinations with exactly three different term ids. (Lines 9-11). We collect information about document ids and lengths (Line 12) as well as the document frequencies of the terms (Line 13). We calculate the individual BM25 scores for each term/document combination (Lines 5 and 6), sum the results and sort (Lines 14 to 16). With the term frequency precalculation optimization,

[1]It would be simple to "join in" the dictionary by term, but the purpose of the shown query is to explain the basic concept.

```
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;
```

Figure 2: Conjunctive BM25 in SQL. The numbers printed in bold are the only parts of the SQL query that depend on the document collection and query terms.

the grouping of the `qterms` CTE (Line 8) would be replaced by a straight selection from the term frequencies table. Note that this approach can be extended to any scoring function that is a sum of matching query terms. Other modifications are straightforward: disjunctive query evaluation can be implemented by replacing the number of matching terms in Line 10. Phrase queries can be performed by arithmetic over term positions and enforcing distance constraints.

4. EXPERIMENTS

The main point of this paper is that relational database technology – in particular, columnar storage – is suitable for rapid prototyping for IR. So far, we have shown how a retrieval model can be implemented without writing any imperative code. In further support of this claim, we present experimental results demonstrating that our approach achieves effectiveness and efficiency on par with custom-built retrieval engines. We compare two different relational backends to three open-source IR engines: the open-source columnar database MonetDB [9] (v11.17.13), the commercial database VectorWise [16] (v3.0.1); and Lucene (v4.3), Indri [14] (v5.5), and Terrier [12] (v3.5). Comparing MonetDB and VectorWise, the latter combines columnar storage with lightweight compression and a pipelining execution model, which makes it the current top performer on the well-known TPC-H benchmark for OLAP databases, and especially suited for very large workloads.

Our experiments used the first segment on the first disk of ClueWeb12 (∼45 million documents) with queries 201–250 from the TREC 2013 web track. This setup is realistic, since production search engines usually adopt a partitioned architecture (and we focus on a single partition). The qrels from the TREC topics were filtered to only include documents that are contained in the segment we used. To ensure that all the IR engines work on the same text, we used Hadoop to pre-process the documents in the same manner as in the relational setup: this was accomplished by dumping the processed collection as plain text, turning off stemming/stopword removal in the IR engine, and tokenizing by whitespace. In all cases we retrieved the top 1000 results. All experiments were run on a Linux desktop computer (Fedora 20, Kernel 3.12.10, 64 Bit) with 8 cores (Intel Core i7-2600K, 3.4 GHz) and 16 GB of main memory.

System	MAP	P5
Indri	0.246	0.304
MonetDB/VectorWise	0.225	0.276
Lucene	0.216	0.265
Terrier	0.215	0.272

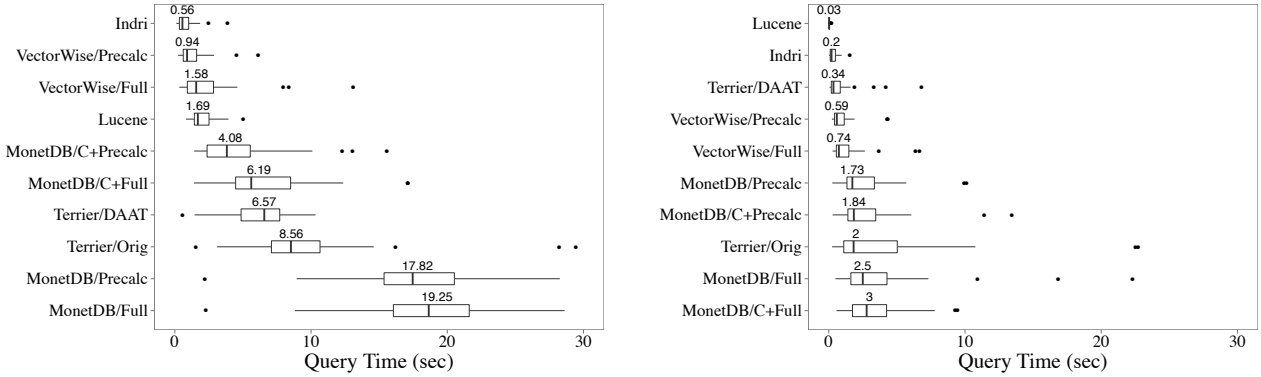
Table 1: MAP and P5 effectiveness scores.

Effectiveness results are shown in Table 1. MonetDB and VectorWise produce exactly the same rankings and scores, which is of course to be expected, given that both execute the same SQL queries. However, the effectiveness differences between Indri and Terrier *do* come as a surprise, since both systems implement BM25—these differences cannot be attributed to tokenization and stopword differences, given the unified document processing step described above.

From these results we draw two conclusions: First, our architecture yields effectiveness that is at least on par with existing custom-built IR engines. Second, these results highlight the advantage of SQL in providing concise yet precise semantics. While we do not claim to have the "correct" BM25 implementation, at the very least our model is concisely specified in a few lines (the SQL query) and thus easy to inspect, not buried in code. Furthermore, different backends produce *exactly* the same results.

In terms of efficiency, we measured query latency. For each system, queries were run sequentially in isolation. As mentioned previously, pre-calculating the term frequency for conjunctive queries is an obvious optimization in the relational representation, and halves the size of the terms table. In our experiments we examined this optimization independently ("Precalc" vs. "Full"). As an additional optimization, we have placed the MonetDB database on a compressed file system (BTRFS with zlib compression, marked as "C"), since MonetDB does not natively compress data. This reduced the disk footprint of the MonetDB database to 1/3 of the original size. In addition, we consulted with the developers of the Terrier system after observing pathologically slow response times. They made two suggestions: 1) changing the index storage model from on-disk to in-memory, which effectively leads to parts of the index being copied into memory on startup, and 2) changing the retrieval model from term-at-a-time (TAAT) to document-at-a-time (DAAT). We did not include the first optimization here, as it makes the cold cache runs pointless (see below). However, the second optimization led to greatly improved performance. For fairness, we report on both the original configuration for Terrier (Orig) as well as on the optimized configuration (DAAT).

Figure 3 shows the query latency distribution across topics as box plots; the boxes contain the observations between the 25^{th} and 75^{th} percentiles. In addition, the plots are annotated with the median query latency. Since the indexes are stored on a hard disk, we expect disk I/O to have a large impact on performance. To test this, we ran the query set two times in succession. Before the first run, we asked the operating system to empty all caches and restarted the



(a) Cold Runs

(b) Hot Runs

Figure 3: Query latencies for TREC 2013 web queries on the first segment of ClueWeb12.

system under test. The two runs are presented in separate plots, the first ("cold" caches) in (a) and the second in (b).

Considering cold runs, we see that Indri was the fastest system, taking a median of 0.56 seconds to complete a query. Second and third were VectorWise, with the precaclulated term frequencies clearly faster, followed by Lucene with a comparable time. MonetDB on a compressed file system and Terrier are comparable. The uncompressed MonetDB experiments clearly show the necessity of compressing an IR index. The vastly greater index size increased the cold response times substantially.

For the hot runs, Lucene leads the field by a large margin, which we suspect to be due to result caching. For Terrier, we see vastly improved timings for the DAAT setting when parts of the index are available in memory and do not have to be loaded from disk. We see that MonetDB (uncompressed) is able to take advantage of caching by the operating system, improving performance by about an order of magnitude compared to the cold runs.

Summarizing these results, it is clear that our database architecture achieves query latencies that are at least on par with existing custom-built IR engines.

5. CONCLUDING REMARKS

This paper argues for the use of column stores for prototyping IR systems. The well-defined relational model allows for the precise expression of retrieval models independent of system implementations. Furthermore, clear abstraction via the declarative SQL interface allows the backend database engine to be swapped, allowing IR systems to quickly benefit from advances in the database field. We have described how to express a standard ranking function using an SQL query and experiments show that using a column store yields effectiveness and efficiency that are at least on par with custom-built IR engines. Hopefully, this will inspire additional IR researchers to think about using columnar databases for building future systems, as doing so would let us focus on the questions on how to best satisfy information needs.

6. ACKNOWLEDGMENTS

This research was supported by the Netherlands Organization for Scientific Research (NWO project 640.005.001), the Dutch national program `COMMIT/`, and the U.S. National Science Foundation (IIS-1144034 and IIS-1218043). Any opinions, findings, or recommendations expressed are the authors' and do not necessarily reflect those of the sponsors.

7. REFERENCES

[1] N. Asadi and J. Lin. Effectiveness/efficiency tradeoffs for candidate generation in multi-stage retrieval architectures. *SIGIR*, 2013.

[2] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, May 2002.

[3] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR technologies: What is the sound of one hand clapping? *CIDR*, 2005.

[4] G. Copeland and S. Khoshafian. A decomposition storage model. *SIGMOD*, 1985.

[5] R. Cornacchia, S. Héman, M. Zukowski, A. de Vries, and P. Boncz. Flexible and efficient IR using array databases. *VLDB*, 2008.

[6] N. Fuhr. Models for integrated information retrieval and database systems. *IEEE Data Eng. Bull.*, 19(1):3–13, 1996.

[7] T. Grabs, K. Bhoem, and H.-J. Schek. PowerDB-IR: scalable information retrieval and storage with a cluster of databases. *Knowledge and Information Systems*, 6(4):465–505, 2004.

[8] S. Héman, M. Zukowski, A. de Vries, and P. A. Boncz. MonetDB/X100 at the 2006 TREC terabyte track. *TREC*, 2006.

[9] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[10] I. Macleod. Text retrieval and the relational model. *JASIS*, 42(3):155–165, 1991.

[11] D. Metzler and W. B. Croft. Combining the language model and inference network approaches to retrieval. *IP&M*, 40(5):735–750, 2004.

[12] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and C. Lioma. Terrier: A high performance and scalable information retrieval platform. *OSIR*, 2006.

[13] H.-J. Schek and P. Pistor. Data structures for an integrated data base management and information retrieval system. *VLDB*, 1982.

[14] T. Strohman, D. Metzler, H. Turtle, and W. B. Croft. Indri: a language-model based search engine for complex queries. *International Conference on Intelligent Analysis*, 2005.

[15] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(6):1–56, 2006.

[16] M. Zukowski and P. A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012.

[17] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. *ICDE*, 2006.