Challenges for industrial-strength Information Retrieval on Databases

Roberto Cornacchia
Spinque

Michiel Hildebrand
Spinque

Arjen P. de Vries
Radboud University

Frank Dorssers
Spinque

ABSTRACT

Implementing keyword search and other IR tasks on top of relational engines has become viable in practice, especially thanks to high-performance column-store technology. Supporting complex combinations of structured and unstructured search in real-world heterogeneous data spaces however requires more than “just” IR-on-DB. In this work, we walk the reader through our industrial-strength solution to this challenge and its application to a real-world scenario. By treating structured and unstructured search as first-class citizens of the same computational platform, much of the integration effort is pushed from the application level down to the data-management level. Combined with a visual design environment, this allows to model complex search engines without a need for programming.

1. INTRODUCTION

There is a growing demand for solving complex search tasks in heterogeneous data spaces, such as enterprise search [9], expert finding [7, 2], recommendation [3].

These types of tasks require unstructured as well as structured search. We argue that by implementing information retrieval on a database it becomes easier to support complex search tasks. Already in 1981, Crawford suggested in [6] that using standard query languages and proven relational calculus: eases engineering; ensures repeatability of results across systems; enables data-independence in text search algorithms; allows search applications to benefit “for free” from any advances in the database engine. In more recent years, [5] and [10] emphasized these benefits and showed that relational technology can compete, performance-wise, with specialized data structures, especially when implemented in modern column-store engines optimized for online analytical processing (OLAP) work-loads.

In this paper we describe the challenges identified by Spinque, a spin-off company from CWI Amsterdam, as it converted exciting results from research into industrial-strength complex search solutions. We illustrate our approach on a simplified as well as a real-world use case.

2. REAL-WORLD IR-on-DB

This section describes the challenges for a unified approach to structured and unstructured search and Spinque’s solution to these challenges:

1. efficient database implementation of IR tasks to search unstructured data;
2. a flexible data model to accommodate queries over any type of structured data;
3. a mechanism to compute and propagate partial scores consistently over unstructured *and* structured data;
4. an abstraction layer to model complex tasks easily.

The “toy” scenario. In the reminder of this section, let us use the following running example: to perform keyword search on a product database, but only consider the description section of products in the category “toy”.

2.1 Keyword search in MonetDB/SQL

The core of keyword search implementations is fast lookup of query term occurrences within text documents. *Inverted index* structures [14, 4] are used to map each term to its “posting list” – the positions at which it appears in a collection of text “documents”. Terms occurrences are then used to build the statistical data employed by the ranking algorithm of choice.

An inverted index can be easily implemented with any relational DBMS. As shown in Figure 1, term lookup requires an inner join on terms between a table containing query terms and a table containing term occurrences.

The ability to create such index structures on-demand is crucial to support scenarios where keyword search is part of more complex tasks, because their parameters (e.g. stemming language) are often hard to decide upfront. Data fed to our system undergoes almost no pre-processing, so that the original text can be ranked at any time by e.g. custom distance functions, tokenization strategies, stemming choices. The only additions needed to MonetDB to support on-demand indexing were two user-defined functions to implement a text tokenizer and Snowball stemmers for several languages.

term		posting-list
book	$\longrightarrow$	(3,23),(10,55)
cake	$\longrightarrow$	(10,51)
history	$\longrightarrow$	(3,19)

(a) Inverted index

term	doc	pos
book	3	23
book	10	55
cake	10	51
history	3	19

term-doc

$\bowtie_{term}$

term
book
about
history

query

(b) Inverted index as a relational join on term

Figure 1: Term look-up

Let us assume that the toy scenario introduced in Section 2 has already been partially solved, so that a table `(productID int, description string)` provides us with pairs of "toy" products and their description, and products have to be ranked according to the relevance of their description to the query keywords. We show how Okapi BM25 ranking function can be implemented in MonetDB/SQL. The following query turns a generic `(docID int, data string)` table into the equivalent of a term-doc matrix:

```
CREATE VIEW term_doc AS
SELECT stem(lcase(token),'sb-english') as term, docID
FROM   tokenize( (SELECT docID, data FROM docs) );
```

From this doc-term matrix we can produce some simple counts and a term dictionary:

```
CREATE VIEW doc_len AS
SELECT docID, count(*) as len
FROM   term_doc   GROUP BY docID;

CREATE VIEW termdict AS
SELECT row_number() over() as termID, terms.term
FROM (SELECT DISTINCT term FROM term_doc) AS terms;
```

From a string-based doc-term matrix of boolean values, we generate an integer-based doc-term matrix of frequencies:

```
CREATE VIEW tf AS
SELECT termdict.termID, term_doc.docID, count(*) as tf
FROM   term_doc, termdict
WHERE  term_doc.term = termdict.term
GROUP BY termdict.termID, term_doc.docID;
```

Inverse document-frequency (IDF) of terms can be formulated as follows:

```
CREATE VIEW idf AS
SELECT termID, log(
        ((SELECT count(*) FROM doc_len) - count(*) + 0.5)
        / ( count(*) + 0.5 ) ) as idf
FROM   tf  GROUP BY termID;
```

BM25's term frequency is controlled by two free parameters, `k1` (saturation) and `b` (doc-length normalization):

```
CREATE VIEW tf_bm25 AS
SELECT tf.docID, tf.termID, tf.tf / (
        tf.tf + (k1 * (1 - b + b * doc_len.len /
         ( SELECT avg(len) FROM doc_len ) ))) as tf
FROM   tf, doc_len
WHERE  tf.docID = doc_len.docID;
```

We apply the normalization steps and dictionary mapping seen above to the "query document" (a string singleton):

```
CREATE VIEW qterms AS
SELECT termdict.termID
FROM   tokenize((SELECT data from query)) AS qt, termdict
WHERE  stem(lcase(qt.token),'sb-english') = termdict.term;
```

Finally, the tf-idf contributions of all query terms are summed up to define the relevance score of each document:

```
SELECT tf_bm25.docID, sum(tf_bm25.tf) as score
FROM   tf_bm25, idf, qterms
WHERE  tf_bm25.termID = qterms.termID
AND    idf.termID = qterms.termID
GROUP BY tf_bm25.docID;
```

Most alternative ranking functions would easily adapt or reuse large parts of this implementation. Also, most of the SQL queries above are independent of query-terms, which allows to materialize intermediate results for reuse in different search scenarios on the same data. While beating specialized text retrieval systems on raw speed is not the focus of this study, reaching reasonable performance is a requirement for the development of real search solutions. In accordance with [5, 10], we can report runtime performance in the range of 20ms (hot data) for 3-term queries against a 2.3GB collection of raw text (1.1M documents), on a standard Linux desktop machine (8-core, Intel i7-3770S, 3.10GHz, 16GB RAM, 256GB SSD), using MonetDB v11.23.14.

2.2 Flexible data model

Relational tables can store and query structured data efficiently, but they are not particularly application-friendly: the schema of each table must be known by applications using them, but the optimal schema depends on the data at hand. Triple-stores can offer a valid alternative.

Semantic triples are best known in the Semantic Web community as the atomic data unit in the RDF data model. Triples encode statements about resources, in the form of `(subject,property,object)`, and a collection of such statements can be interpreted as a graph. Triple-stores are (mostly relational) database systems specifically designed to store and manipulate this special kind of 3-column tables (or 4, when managing quadruples for named graphs), although any relational DBMS would also serve the purpose. In fact, we use the standard SQL interface of MonetDB to implement and query a triple-store, with an important custom addition that we describe in Section 2.3. One direct advantage of using a standard SQL interface is that it allows to exploit the strengths of both triples and standard tables.

For the toy scenario of Section 2, the `docs` table to be provided as input for keyword search (see Section 2.1) can be generated at query time by the following SQL view:

```
CREATE VIEW docs as
SELECT t2.subject as docID, t2.object as data
FROM triples t1, triples t2
WHERE t1.property = 'category' AND t1.object = 'toy'
AND   t2.property = 'description'
AND   t1.subject = t2.subject;
```

The triple model allows to write this and other queries using simple and application-independent patterns. However, this flexibility comes at the price of having to reconstruct the typical relational row `(product,category,description, ...)` at query time, which requires self-joins of a possibly large `triples` table. Vertical partitioning of the set of triples can address performance and scalability issues, as long as the right partitioning approach is chosen for the application context at hand. The only data-driven partitioning that we apply is by the physical data type of objects (rather than serializing every literal into strings). It is always applicable and can improve efficiency, but does not solve scalability issues. With the assumption that product categories and descriptions are accessed often, storing `(?,category,?)` and `(?,description,?)` triples into separate tables would dramatically improve our scenario. This is the main reasoning of [1], where the authors propose a vertical partitioning of all properties into separate tables. However, [13] shows that this solution is less scalable when the number of properties is high. We use an on-demand approach to vertical partitioning, which is applied not only to selections on the `property` column of triples, but to any intermediate result generated in our database. This creates an adaptive, query-driven set of "cache" tables each corresponding to a specific sub-query on the original data. When the same computation is requested several times, its full result is already materialized. An interesting alternative to consider would be the detection of emergent schemas [11], a data-driven technique to find a relational schema that is considered optimal for a given graph, thus eliminating many join operations.

2.3 Score propagation

Sections 2.1 and 2.2 show how to combine filtering (structured search) and ranking (unstructured search) of the text collection defined on-the-fly by such filtering. What makes these operations still disconnected is their inherently different computational model: one produces certain answers from facts, the other applies statistical methods to produce likely relevant answers.

We reduce this gap by implementing a probabilistic relational database with tuple-level uncertainty: a probability column `p` is appended to all tables, including triples, in our RDBMS. Semantic triples no longer encode facts, but rather uncertain events: `(subject,property,object,p)`. Probabilities smaller than 1 can originate from the data (e.g. due to confidence-based data extraction techniques), or from any intermediate computation that produces ranked results. With probabilistic tuples, structured search need not be restricted to boolean facts and can play alongside unstructured search with the very same tools.

Encoding probabilistic information is one part of the solution. We still need to combine and propagate such probabilities when tuples are processed by relational operators. For this, we use a proprietary domain specific language called SpinQL, which implements the Probabilistic Relational Algebra (PRA) developed in [8, 12], with particular focus on efficient translation to SQL. SpinQL is used everywhere in the system, including the implementation of BM25 and other retrieval models. Extracting toy descriptions is expressed in SpinQL as:

```
docs = PROJECT [$1,$6] (
  JOIN INDEPENDENT [$1=$1] (
   SELECT [$2="category" and $3="toy"] (triples),
   SELECT [$2="description"] (triples) ) );
```

and translates to:

```
CREATE VIEW docs as
SELECT t2.subject as docID, t2.object as data,
       t1.p * t2.p as p
FROM triples t1, triples t2
WHERE t1.property = 'category' AND t1.object = 'toy'
AND   t2.property = 'description'
AND   t1.subject = t2.subject;
```

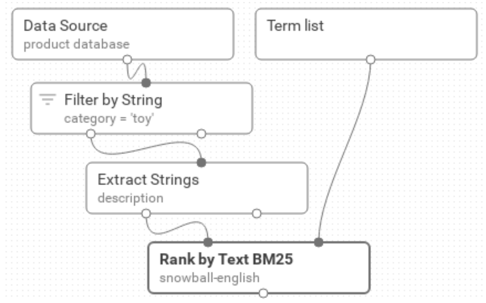

Figure 2: rank toy products by their description

Each relational operator defines how to compute probability columns. For example, the query above joins independent events from the two tables, which makes the resulting probability of each join match be computed as the product of two input tuples' probabilities. If applied correctly, this algebra allows to keep the probabilistic computation sound. Using SpinQL leads to more concise query plans and is less error-prone than hand-specifying probability computations, as these are only made explicit upon translation into SQL.

2.4 Modeling complexity

While SQL / SpinQL interfaces allow to express mixed structured and unstructured search and can be evaluated efficiently, they are not well suited for search engine designers. A basic search engine would easily require tens of queries with hundreds of lines of code. Therefore, we created a graphical environment where a so-called *search strategy* is modeled out of building blocks.

Figure 2 shows the search strategy that defines the toy scenario used throughout Section 2. Block `Rank by Text BM25` contains the BM25 implementation shown in Section 2.1, though expressed in SpinQL rather than SQL. It takes a probabilistic `(docID, data)` table on the left and a list of query terms on the right. The sub-strategy on the left corresponds to the sub-collection filtering of sections 2.2 and 2.3.

Connecting blocks is a convenient way to express complex search scenarios declaratively without programming efforts. The SpinQL queries contained in each block are combined automatically under the hood.

3. A REAL-WORLD SCENARIO

Figure 3 depicts a simplified version (due to space and confidentiality constraints) of a real strategy used by one of our customers in the business of online auctions. Via the website's search-bar, users activate this strategy to find the items they are interested in. The primary retrieval unit in the database is a lot, which is an item or a set of items for sale in an auction. Lots are connected to auctions via triples like `(lot23,hasAuction,auction12)`. Both lots and auctions have their own identifier and a textual description, as part of a rich semantic graph.

Let us summarize the strategy in Figure 2 in a few steps:

1. The strategy first selects nodes of type `lot` from the graph, then it splits in two branches.

2. The branch on the left extracts the lot descriptions, on which it ranks the lots with the given query keywords, similarly to the toy scenario (Figure 2).

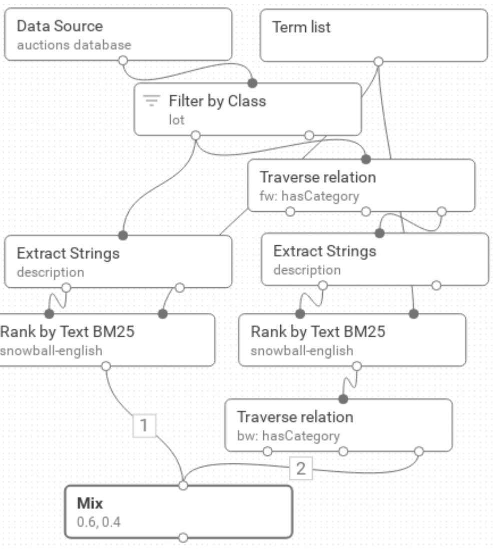

Figure 3: rank auction lots

3. The branch on the right uses the same keywords to rank lots by the description of the auctions in which they are contained: it first traverses property `hasAuction` to find the containing auction; then ranks auctions by their description; finally, it traverses `hasAuction` backward, to obtain lots again.

4. The two different ranked lists of lots are mixed via linear combination, with the given weights.

This strategy exemplifies the concepts expressed in Section 2:

Keyword search. No specific indexing configuration was required. Two distinct inverted indices were created on-demand, given the selected sub-collection.

Flexible data model. Structured search, such as filtering and graph traversal steps, rely on a data-agnostic database schema. This allowed the design of strategy blocks that work consistently on any collection.

Score propagation. All the operations in this strategy propagate probabilities through the graph, including the first ones, which carry unaltered probabilities (1.0) from initial data. On the right branch however, the last traverse operation finds lots with probabilities that depend on those of their ranked auctions. This happens transparently, thanks to the underlying probabilistic relational algebra layer.

Strategy abstraction. Despite the mix of structured and unstructured search that this strategy involves, it remains understandable at a glance, with technical details hidden, and can be engineered and modified easily.

Industrial-strength implementation. The production version of this strategy (which includes 5 parallel keyword search branches and query expansion with synonyms and compound terms), runs, together with several others, on a single VM server (8-core, Intel Xeon E5-2620, 2.40GHz, 16GB RAM, 256GB SSD). It searches about 8 million lots in 25 thousand auctions, 150,000 times per day (with peaks of 450 per minute) with response times of about 150ms per request (hot database). We consider this performance adequate to the complexity of this task, but more importantly it was achieved with no programming or optimization effort.

4. WRAP UP

This work explored the long-standing IR and DB integration issue with particular emphasis on the implementation of industrial-strength search solutions. While [10] already claimed that "*databases form a flexible rapid prototyping tool*", we can add that "*databases are also a solid and viable solution for search in production environments*".

We showed that by pulling information retrieval into a database it becomes possible to realize a transparent combination of structured and unstructured queries. This opens up new ways to support complex search scenarios. With the right abstractions on top of this infrastructure, realizing effective and efficient search solutions becomes a task for domain and information specialists instead of programmers.

5. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. VLDB '07, pages 411–422. VLDB Endowment, 2007.

[2] K. Balog, Y. Fang, M. de Rijke, P. Serdyukov, and L. Si. Expertise retrieval. *Foundations and Trends in Information Retrieval*, 6(2-3):127–256, 2012.

[3] A. Bellogín, J. Wang, and P. Castells. Bridging memory-based collaborative filtering and text retrieval. *Inf. Retr.*, 16(6):697–724, Dec. 2013.

[4] S. Büttcher, C. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. The MIT Press, 2010.

[5] R. Cornacchia, S. Héman, M. Zukowski, A. P. Vries, and P. Boncz. Flexible and efficient ir using array databases. *The VLDB Journal*, 17(1):151–168, 2008.

[6] R. G. Crawford. The relational model in information retrieval. *Journal of the American Society for Information Science*, 32(1):51–64, 1981.

[7] M. Fazel-Zarandi, H. J. Devlin, Y. Huang, and N. Contractor. Expert recommendation based on social drivers, social network analysis, and semantic data representation. HetRec '11, pages 41–48, 2011.

[8] N. Fuhr and T. Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inf. Syst.*, 15(1), 1997.

[9] D. Hawking. Challenges in enterprise search. ADC '04, pages 15–24, Darlinghurst, Australia, 2004. ACS, Inc.

[10] H. Mühleisen, T. Samar, J. Lin, and A. de Vries. Old dogs are great at new tricks: Column stores for ir prototyping. SIGIR '14, pages 863–866, New York, NY, USA, 2014. ACM.

[11] M.-D. Pham and P. Boncz. *Exploiting Emergent Schemas to Make RDF Systems More Efficient*, pages 463–479. Springer, Cham, 2016.

[12] T. Roelleke, H. Wu, J. Wang, and H. Azzam. Modelling retrieval models in a probabilistic relational algebra with a new operator: The relational bayes. *The VLDB Journal*, 17(1):5–37, Jan. 2008.

[13] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for rdf data management: Not all swans are white. *Proc. VLDB Endow.*, 1(2):1553–1563, Aug. 2008.

[14] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), July 2006.