The OldDog Docker Image for OSIRRC at SIGIR 2019

Chris Kamphuis
ckamphuis@cs.ru.nl
Radboud University
Nijmegen, The Netherlands

Arjen P. de Vries
arjen@acm.org
Radboud University
Nijmegen, The Netherlands

ABSTRACT

Modern column-store databases are perfectly suited for carrying out IR experiments, but they are not widely used for IR research. A plausible explanation would be that setting up a database system and populating it with the documents to be ranked provides enough of a hurdle to never get started on this route.

We took up the OSIRRC challenge to produce an easily replicable experimental setup for running IR experiments on modern database architecture. OldDog, named after a short paper on SIGIR proposing the use of column-stores for IR experiments, implements standard IR ranking using BM25 as SQL queries issued to MonetDB SQL. This provides a baseline system on par with custom IR implementations and a perfect starting point for the exploration of more advanced integrations of IR and databases.

Reflecting on our experience in OSIRRC 2019, we found a much larger effectiveness penalty than anticipated in the prior work for using the conjunctive variant of BM25 (requiring all query terms to occur). Simplifying the SQL query to rank the documents using the disjunctive variant (the normal IR ranking approach) results in longer runtimes but higher effectiveness. The interaction between query optimizations for efficiency and the resulting differences in effectiveness remains a research topic with many open questions.

CCS CONCEPTS

- **Information systems** → *Search engine indexing*; *Evaluation of retrieval results.*

KEYWORDS

information retrieval, replicability, column store

Image Source: https://github.com/osirrc/olddog-docker
Docker Hub: https://hub.docker.com/r/osirrc2019/olddog
DOI: https://doi.org/10.5281/zenodo.3255060

1 OVERVIEW

OldDog is a software project to replicate and extend the database approach to information retrieval presented in Mühleisen et al. [2]. The authors proposed that IR researchers would use column store relational databases for their retrieval experiments. Specifically, researchers should store their document representation in such a database. The ranking function can then be expressed as SQL queries. This allows for easy comparison of different ranking functions. IR researchers will only need to focus on the retrieval methodology while the database takes care of efficiently retrieving the documents.

OldDog represents the data using the schema proposed by Mühleisen et al. [2]. An extra 'collection identifier' column has been added to include the original collection identifiers. The original paper [2] produced the database tables to represent 'postings' using a custom program running on Hadoop. Instead, we rely on the Anserini toolsuite [4] to create a Lucene[1] index. Anserini takes care of standard document pre-processing.

Like [2], OldDog uses column store database MonetDB [1] for query processing. Term and document information is extracted from the Lucene index, stored as CSV files representing the columns in the database, and loaded into MonetDB using a standard `COPY INTO` command.[2]

After initialisation, document ranking is performed by issuing SQL queries that specify the retrieval model. Interactive querying can support the researcher with additional examples.

2 TECHNICAL DETAILS

Supported Collections:

`robust4, core18`

Supported Hooks:

`init, index, search, interact`

The OldDog docker image itself consists of `bash`/`python` 'hooks' that wrap the underlying OldDog, Anserini and MonetDB commands.

Anserini builds a Lucene index, where we are happy end-users of the utilities provided to index common test collections. The code has been tested for Robust04 and Core18 in our first release; extending to other collections readily supported by Anserini should be trivial.

OldDog provides the Java code to convert the Lucene index created by Anserini into CSV files that are subsequently loaded into the MonetDB database. OldDog further contains the Python code necessary to call Java code to pre-process the topics (calling the corresponding Anserini code, to guarantee that topics are processed exactly the same way as the documents) and issue SQL queries to the MonetDB database to rank the collection.

OSIRRC 2019 co-located with SIGIR 2019, 25 July 2019, Paris, France.

[1] https://lucene.apache.org, last accessed June 26th, 2019.

[2] The intermediate step exporting and importing CSV files is not strictly necessary, but simplifies the pipeline and is robust to failure.

Apart from the required *init*, *index* and *search* hooks, OldDog supports the *interact* hook to spawn an[3] SQL shell that allows the user to query the database interactively.

2.1 Schema

Consider an example document *doc1* with contents *I put on my shoes after I put on my socks.* to illustrate the database schema. Indexing the document results in tables 1, 2 and 3:

Table 1: dict

termid	term	df
1	put	2
2	shoes	1
3	after	1
4	socks	1

Table 2: terms

termid	docid	count
1	1	2
2	1	1
3	1	1
4	1	1

Table 3: docs

collection_id	id	len
doc1	1	5

2.2 Retrieval Model

OldDog implements the BM25 [3] ranking formula. The values for k_1 and b are fixed to 1.2 and 0.75, respectively. The original paper [2] uses a conjunctive variant of this formula, that only produces results for documents where all query terms appear the document; this yields lower effectiveness scores, but speeds up query processing leading to a better runtime performance.

OldDog implements disjunctive query processing as well, included after noticing a surprisingly large difference in effectiveness when compared to other systems applied to Robust04. In the disjunctive variant, documents are considered when they contain at least one of the query terms. As expected, runtimes for an evaluation increase when using this strategy.

Table 4 summarises effectiveness scores for both methods on two test collections, Robust04 and Core18.

Listing 1 shows the conjunctive BM25 SQL query for robust topic 301: *International Organized Crime*. The disjunctive variant simply omits the having clause.

Table 4: Effectiveness scores

	Robust04		Core18	
	MAP	P@30	MAP	P@30
Conjunctive BM25	0.1736	0.2526	0.1802	0.3167
Disjunctive BM25	0.2434	0.2985	0.2381	0.3313

```
/* For all topic terms */
WITH qterms AS (SELECT termid, docid, count FROM terms
  WHERE termid IN (591020, 720333, 462570)),
/* Calculate the BM25 subscores */
subscores AS (SELECT docs.collection_id, docs.id, len,
  term_tf.termid, term_tf.tf, df,
  (log((528030−df+0.5)/(df+0.5))*((term_tf.tf*(1.2+1)/
  (term_tf.tf+1.2*(1−0.75+0.75*(len/188.33)))))) AS
     subscore
/* Calculate BM25 components */
FROM (SELECT termid, docid, count as tf FROM qterms) AS
     term_tf
  JOIN (SELECT docid FROM qterms
    GROUP BY docid HAVING COUNT(distinct termid) = 3)
    AS cdocs ON term_tf.docid = cdocs.docid
  JOIN docs ON term_tf.docid = docs.id
  JOIN dict ON term_tf.termid = dict.termid)
/* Aggregate over the topic terms */
SELECT scores.collection_id, score
  FROM (SELECT collection_id, SUM(subscore) AS score
    FROM subscores
    GROUP BY collection_id) AS scores
    JOIN docs ON scores.collection_id=docs.collection_id
  ORDER BY score DESC;
```

Listing 1: Conjunctive BM25

3 OSIRRC EXPERIENCE

Overall, we look back at an excellent learning experience taking part in the OSIRRC challenge. The setup using Docker containers worked out very well during coding, by multiple people on different machines. Automated builds on Docker Hub and the integration with Zenodo complete a fully replicable experimental setup.

The standardised API for running evaluations provided by the 'jig' made it easy to learn from other groups; and mixing version management using git (multiple branches) with building (locally) Docker containers with different tags allowed progress in parallel when implementing different extensions of the initial code (in our case, including disjunctive query processing and adding Core18).

Recording the evaluation outcomes at release time let us catch a bug that would have been easily overlooked without such a setup - after including Core18, a minor bug introduced in the code to parse topic files lead to slightly different scores on Robust04, that we could easily detect and fix (one topic

[3] Or, '*a* SQL shell', pronouncing SQL as *sequel* like database folk do.

was missing) thanks to the structured approach of recording progress.[4]

4 INTERACT EXAMPLES

Let us conclude the paper by discussing a few advantages of database-backed IR experiments. Using the *interact* hook, it is possible to issue SQL queries directly to the database. This is useful if one wants to try different kinds of ranking functions, or just to investigate the content of the database. We show some examples of queries on the Robust04 test collection.

The *three most occurring terms* are easily extracted from the dict table:

```
SELECT * FROM dict ORDER BY df DESC LIMIT 3;
```

```
+--------+-------+--------+
| termid | term  | df     |
+========+=======+========+
| 541834 | from  | 355901 |
| 563475 | ha    | 320097 |
| 894136 | which | 302365 |
+--------+-------+--------+
```

As expected, the term distribution is skewed with a very long tail; consider for example the number of distinct terms that occur only once:

```
SELECT COUNT(*) AS terms FROM dict WHERE df = 1;
```

```
+--------+
| terms  |
+========+
| 516956 |
+--------+
```

Apart from applying a brief static stopword list to all preprocessing (defined in `StandardAnalyzer.STOP_WORDS_SET`), Anserini 'stops' the query expansions in its RM3 module by filtering on document frequency, thresholded at 10% of the number of documents in the collection.

Having such a collection-dependent stoplist would be an interesting option in the initial run as well, so let us use the interactive mode to investigate the effect on query effectiveness of applying this *df* filter to the initial run.

We can easily evaluate the effect of removing the terms with high document frequency, e.g. by modifying the dictionary table as follows:

```
ALTER TABLE dict RENAME TO odict;
CREATE table dict AS
SELECT * FROM odict WHERE df <=
  (SELECT 0.1 * COUNT(*) FROM docs);
```

We find the effectiveness scores shown in table 5 for disjunctive BM25. Performance drops for both MAP and early precision, suggesting that filtering query term presence based on document count is not a good idea, and should be limited to pseudo relevance feedback (not yet implemented in OldDog).

Table 5: Effectiveness scores after high *df* term removal

	Robust04		Core18	
	MAP	P@30	MAP	P@30
Disjunctive BM25	0.2285	0.2727	0.1907	0.2693

A natural next step is to include the 'qrel' files in the database, to explore more easily the relevant documents that are (not) retrieved by specific test queries.

5 CONCLUSION

We conclude that we could successfully apply the methods from [2], and have learned that conjunctive query processing for BM25 degrades retrieval effectiveness more than we expected a priori. The Docker image produced for the workshop is a perfect starting point for exploration of IR on relational databases, where we build on standard pre-processing and test collection code in the Anserini project. Of course, we should extend the retrieval model beyond plain BM25 to obtain more interesting results from an IR perspective. Interactive querying the database representation of the collection, especially after including relevance assessments, seems like a promising avenue to pursue. Finally we found that the 'jig' setup not only allows for easy replication of the software, it serves as a tool for supporting continuous integration.

ACKNOWLEDGMENTS

This work is part of the research program Commit2Data with project number 628.011.001 (SQIREL-GRAPHS), which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).

We also want to thank Ryan Clancy and Jimmy Lin for the excellent support with the 'jig' framework.

REFERENCES

[1] Peter Boncz. 2002. *Monet: A next-generation DBMS kernel for query-intensive applications.* Universiteit van Amsterdam.

[2] Hannes Mühleisen, Thaer Samar, Jimmy Lin, and Arjen De Vries. 2014. Old dogs are great at new tricks: Column stores for IR prototyping. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval.* ACM, 863–866.

[3] Stephen E Robertson and Steve Walker. 1994. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *Proceedings of the 17th international ACM SIGIR conference on Research & development in information retrieval.* Springer, 232–241.

[4] Peilin Yang, Hui Fang, and Jimmy Lin. 2017. Anserini: Enabling the use of Lucene for information retrieval research. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval.* ACM, 1253–1256.

[4] We may also conclude that a unified topic format for all TREC collections would be a useful improvement to avoid errors in experiments carried out on these test collections.