Information Retrieval from Digital Libraries in SQL

Carlos Garcia-Alvarado
University of Houston
Department of Computer Science
Houston, TX 77204, USA

Carlos Ordonez
University of Houston
Department of Computer Science
Houston, TX 77204, USA

ABSTRACT

Information retrieval techniques have been traditionally exploited outside of relational database systems, due to storage overhead, the complexity of programming them inside the database system, and their slow performance in SQL implementations. This project supports the idea that searching and querying digital libraries with information retrieval models in relational database systems can be performed with optimized SQL queries and User-Defined Functions. In our research, we propose several techniques divided into two phases: storing and retrieving. The storing phase includes executing document pre-processing, stop-word removal and term extraction, and the retrieval phase is implemented with three fundamental IR models: the popular Vector Space Model, the Okapi Probabilistic Model, and the Dirichlet Prior Language Model. We conduct experiments using article abstracts from the DBLP bibliography and the ACM Digital Library. We evaluate several query optimizations, compare the on-demand and the static weighting approaches, and we study the performance with conjunctive and disjunctive queries with the three ranking models. Our prototype proved to have linear scalability and a satisfactory performance with medium-sized document collections. Our implementation of the Vector Space Model is competitive with the two other models.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Languages, Algorithms

Keywords

SQL, Documents, Ranking, Query Optimization

1. INTRODUCTION

There exists extensive research in multiple Information Retrieval (IR) applications that work outside the database. Most of these solutions are developed frameworks and implementations in non-restrictive languages, such as the Lemur Project, Mallet or Libbow [19, 13, 12]. Still, the need for integrating database and information retrieval technologies, while taking into consideration the strengths of both research communities [20] continues to grow. According to this perspective, some module extensions were added to perform queries in text by commercial vendors, allowing full-text searches in modern major DBMS. However, these extension modules offer constrained functionalities for ranking documents, such as the incapability of changing certain parameters that depend on the properties of the collection or even relaxing or extending the scoring functions. Due to these limitations and several advantages of in-database work, such as data security, recovery management, memory management and data transfers avoidance, among others, an in-database implementation appears as an alternative for an uncompromised solution to the database's performance, security, and extensibility for collections already existing in a relational database system.

Despite the advantages above, SQL and the Relational Model have not been a "traditional" approach to information retrieval models. This is due to the complexity of implementing ranking models in SQL and non-existing vector operations, constrained element manipulation, lack of integration with existing libraries, storing overhead and low performance. Therefore, the objective of the project is to offer similar capabilities of a search engine in a SQL and User-Defined Function (UDF) based implementation in a Relational Database Management System (DBMS), an idea which has been poorly explored, even though this can be exploited with index-organized tables, clustered registers, recursive queries, UDFs and SQL queries optimizations [3]. In this project we focus on ranking, a capability that is not part of DBMS for querying text databases, where the results can be retrieved based on "matching a pattern," but not on the relevance of the finding. Also, the search engine is intended to be used as an auxiliary tool for ranking, and not as a specialized system for information retrieval. Hence, we favor portability over reliability. This framework includes the pre-processing, storing, indexing and retrieval of documents, mostly done in SQL. We retrieve the documents using popular scoring methods: the Vector Space Model (VSM), the Okapi Probabilistic Model (OPM), and the Dirichlet Prior Language Model (DPLM) in a DBMS.

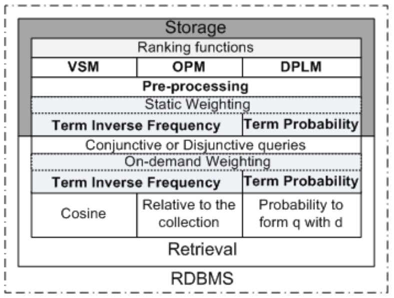


Figure 1: IR in a DBMS.

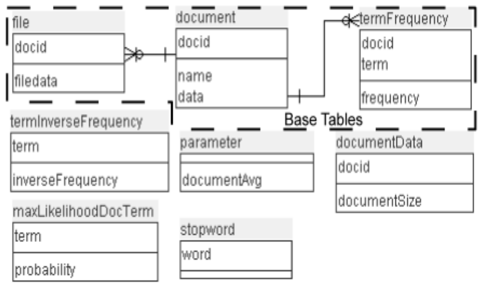


Figure 2: Entity-Relation model.

The article is organized as follows: Section 2 introduces definitions used during the document pre-processing, storage and retrieval models. Section 3 describes the framework for the storage and retrieval phases, and their implementation. Section 4 presents the results of the SQL performance and a discussion of the results. Section 5 shows previous and related work in the area. Section 6 presents conclusions and directions for future research.

2. DEFINITIONS

In our explored scoring techniques, we focused on storing and ranking "a set of terms" in the database. Each collection, or corpus, contains $d_1, d_2, ..., d_N$ documents, and every document contains a set of words w, which are later pre-processed to obtain the relevant words of a document, called keywords. Later, these keywords are stemmed to obtain a term t (root of the word). During this weighting phase, the frequency of a term in a document $tf(t, d_i)$ will be needed, such as the number of documents containing a term in the collection $df(t)$, the number of terms in a document $|d|$, the average number of terms of a document in the collection $avglen$, the norm of a document (vector term weights) $|d|$, and some specific parameters depending on the ranking functions. Also, the same notation applies for a given query by the user, but instead of using d in the notation, we introduce q to refer to the query. The score or rank of a document is represented by a similarity $Sim(q, d_i)$ or a probability $P(q, d_i)$ function.

3. OVERVIEW OF IR IN THE DBMS

We separate the general IR framework into two phases: the storage phase and the retrieval phase (see Figure 1). The flow of the operations for the three ranking models is related to the desired optimizations or selected parameters. The three models have a uniform pre-processing phase, then the term weighting step varies depending on the retrieval model. Later, the retrieval section may contain a weighting task, too, and then it applies the ranking formulas.

The document pre-processing step takes every document and creates a descriptor vector, in a "bag of words" way, which is the result of removing stop-words and applying the Porter Algorithm [17] to every non-stop word. This algorithm has been adopted regularly as the default choice for stemming, although there exist other options such as Lovins, Dawson, and Paice, among others [17]. The main purpose of using a stemming approach is to lower the number of dimensions (words) in a document and still be capable of "describing" the document using a reduced descriptor vector while losing as little information as possible.

We use the *Baseline Inverted File* technique for weighting and indexing the terms [21, 14]. This technique consists of two principal structures. The first of these is a search structure (vocabulary or dictionary) that stores every distinct term t in the collection, and a pointer, or a pair of primary and foreign keys in the relational model, in a corresponding inverted list. The second structure is the inverted list structure in which each entry stores t terms and the associated set of frequencies $tf(t, d_i)$ of the terms in a document [10]. This implies the storage of a descriptor vector of every document d_i in a huge matrix of M different number of terms and N number of documents in the collection. The main advantage of using an Inverted Files technique relies on the fact that the frequency of every term has already been stored. It can then be easily manipulated into several ranking strategies. Storing vectors in the relational model is difficult, and we will assume a vector as a single column relation for the relational model.

The schema in the database (see Figure 2) included a table "stopword" which contains the collection of indexed undesired words. A table "document" stores the metadata of the document, indexed by the document's unique identifier (docid) selected as the primary key. The table "parameter" includes the corpus statistics as a requirement for some ranking strategies. The table "termFrequency"' contains the terms of each document and its frequency. The "termFrequency" table is indexed by term and docid, and the terms are clustered by docid, because ordinarily they are retrieved together to be compared in any of the chosen ranking models. For OPM and VSM models, we compute the weight of each term in the collection with an inverse frequency table "termInverseFrequency," indexed by term. In the Dirichlet Prior Language Model, the probability of each term appears in the "maxLikelihoodDocTerm" table. For OPM and DPLM ranking strategies, we had to collect data from the documents and those values were stored in "documentData".

3.1 Document Storage

The pre-processing phase takes place for every document from the collection (already stored in a DBMS), and extracts a descriptor vector, which is the result of applying a parsing recursive function that we created to guarantee portability in several DBMS. The resulting words are then purged of special characters and punctuation in a SQL/Replace implementation. Then, for each word in the remaining d_i vector, all non relevant stop-words are removed from the original document, and finally, the Porter's algorithm is applied to the whole table to obtain the terms.

The "term weighting phase" needs to generate, initially, a summarized table with the frequency of a term in a document. Thus, the remaining terms after the pre-processing phase are grouped and counted, and inserted into the term frequency matrix. The main advantage of using a Baseline Inverted Files technique relies on the fact that the frequency of every term has already been stored, and can then be easily manipulated for weighting the terms for different ranking strategies (inverse frequency computation or probability of a term).

3.2 Term Weighting

The VSM and the OPM require the computation of the inverted frequency of a term, which corresponds to a variation of the form $\ln(N/df(t))$, depending on the ranking strategy. We studied two techniques for computing this inverted frequency table or probability of a term: static and on-demand. The static technique is well extended for non-relational information retrieval implementations, and the on-demand technique is a newly studied technique for minimizing the cost of computing all of the weights or probabilities in a collection for RBDMS. The selection of either technique requires some slight changes in the IR phases described in previous sections. The first three steps for pre-processing, and obtaining the term frequencies remain unchanged, but in the static weight computation, the computation must occur at the end of the storage phase, and can be done in parallel with the corpus statistics computation.

The on-demand technique works as a lazy policy, dependent upon the idea that just a few terms in the query and the related documents will be included in most of the searches, and performing the computation only when a term is required in a query. Thus, this weighting must be executed during the ranking phase. In the static frequency computation technique, the table must be computed entirely after every document is added to the collection (or a group of documents), but in an on-demand technique, the table is updated by insertions.

When the inverse of one or several terms is required by a ranking operation involving those terms, a search is executed to solve for which terms are required to compute its weights. If in the "partial" inverted frequency or probability table, the weight for the terms have not already been computed, then it inserts all the missing terms in a single operation. Thus, an incomplete weight table is maintained every time, and when the collection changes, such as in the static approach, the inverse frequency table must be deleted. We decided not to apply updating techniques in the relational database model, because this proved to be more time consuming than dropping the table and recomputing all the weights (or just a few in the on-demand technique). This implementation is specific for the relational model and in a "traditional" information retrieval implementation, may be observed as a "non-conventional" technique.

The weighting term phase is the slowest operation in the overall process, especially when the number of terms is higher than 10000, because it must be executed taking into consideration all of the elements and their uniqueness relative to the collection. Normally, the choice between these two techniques is based on the nature of the collection and the requirements of the frequency of insertion of new documents and the "diversity" of searched terms. The inverse frequency for the VSM corresponds to the basic $\ln(N/df(t))$ equation, but in the Okapi implementation, the weight is slightly adjusted (see Equation 1). Okapi's representation can address a problem with the scoring of negative values, as discussed in [2], and normally is solved by modifying the weighting function. The Dirichlet Prior Language Model approach bases the term weight in a probability instead of an inverse frequency, such as VSM or OPM.

$$idf(t) = \ln \frac{N - df(t) + 0.5}{df(t) + 0.5} \tag{1}$$

3.3 Document Retrieval

The document retrieval phase shares the term frequencies with the scoring strategies independently of the desired ranking functions, and just certain additional computations such as the average length of the documents, and the size of a document, among others, are computed to complement certain models. Here, we study three well-known approaches, much more complex than the Boolean models, focusing our research on the Vector Space Model (VSM), Okapi Probabilistic Model (OPM) and the Dirichlet Prior Language Model (DPLM). We implement the VSM, OPM and the DPLM because these are the primary models for vector space, probabilistic or language model ranking approaches. Also, they all share properties such as relevance feedback, and the ranking and pre-processing phase require the same presented techniques. The VSM tries to order the retrieval of the documents based on their relevance. The OPM was designed to work on short records, preferably abstracts, with a consistent length, and it works primarily in the estimation of a document's relevancy. The Dirichlet Prior Language Model is close to the OPM, but it finds the probability of a document to form a given query.

Vector Space Model

Our system implements the Vector Space Model [18], which takes all documents related to the given query and then obtains the rank of all of these related documents. This ranking is generated by taking the stem vector of the given query and every document containing any searched stem and then applying the Similarity Formula (see Equation 2) in SQL. In this formula, the stemmed term vectors of the given query (q) and the i^{th} *document* (d_i) are computed with a dot product between them and then divided by the norm of the $||q||$ vector and the d_i vector. The q vector is a table with the terms given by the user in the query.

$$Sim(q, d_i) = \frac{\vec{q} \cdot \vec{d_i}}{||\vec{q}|| \left\| \vec{d_i} \right\|} \tag{2}$$

Okapi Probabilistic Model

This scoring function has been recognized as a very effective retrieval method (see Equation 3), although it has to be adjusted with some constants: k_1 and k_3 as presented in [2]. This formula may produce negative values when the frequency of the term in the collection is greater than half of the number of total documents in the collection.

$$\begin{aligned} \sum_{t \in q \cap d_i} idf(t) \quad &\times \quad \frac{(k_1 + 1) \times tf(t, d_i)}{k_1((1 - b) + b\frac{|d|}{avglen}) + tf(t, d_i)} \\ &\times \quad \frac{k_3 + 1 \times tf(t, q)}{k_3 + tf(t, q)} \end{aligned} \tag{3}$$

```
WITH TERM( i, word, g, k , j ) AS
( SELECT i, word, g, k, j FROM T
  UNION ALL
  SELECT TERM.i+1
   ,CAST(SUBSTRING(T.word,TERM.i-TERM.k
                  ,TERM.k+1) AS text)
   ,CASE WHEN SUBSTRING(T.word, TERM.i,1 ) = ' '
         THEN TERM.g+1
         ELSE CASE
           WHEN TERM.g=1 THEN TERM.g+1
             ELSE TERM.g END
         END
   ,CASE WHEN SUBSTRING(T.word, TERM.i,1 ) = ' '
         THEN 0 ELSE CASE WHEN TERM.g=1
         THEN 1 ELSE TERM.k+1 END END
   , 1
   FROM T
   INNER JOIN TERM
   ON ( TERM.j = T.j )
   WHERE TERM.i < DATALENGTH(T.word)+2
)
INSERT INTO RESULT
SELECT i, word, g, k, j
FROM TERM;
```

Figure 3: ParseArray recursive view.

Dirichlet Prior Language Model

The Dirichlet Prior Language Model (DPLM) is a language model scoring method that is based on the probability that a document will "form" a given query (see Equation 4). Therefore, it takes into consideration the probability of occurrence of a term in the collection, even though, as in Okapi, DPLM has a μ parameter to adjust the method based on the characteristics collection [2, 7].

$$P(q|d_i) = \prod_{t \in q \cap d_i} \frac{tf(t,d_i) + \mu P(t|C)}{\sum_t tf(t,d_i) + \mu} \tag{4}$$

3.4 Implementation

We designed a web interface to interact with the digital collection, which sends standard SQL queries and retrieves the results for the user. The SQL queries are designed on-the-fly in our web interface to match the preferences of the users, such as using on-demand weighting, conjunctive or disjunctive queries, and parameter selection.

Storage Queries

In the storage process, we proposed a recursive function [15] for portability and for performing the stripping task (see Figure 3), which normally has been solved using external implementations, language DBMS specific implementations or n-gram parsing [8]. We started with the initial text to be parsed in a temporary table, we use "term" as the table that makes the recursion possible, and then we selected the cumulative word before the "word splitter." In this case, we decided to use a blank space as the separator, but this could be replaced with any other character. Then, we stored the final result in a term temporary table for future use.

When the parsing step was concluded, the resulting words in the table were stemmed using the implementation presented in [9], but before this, the stop-words were removed, as described clearly in [8]. These tasks have been executed normally with a NOT-IN operator. Based on previous research, we decided to modify the stop-word removal query by selecting all the non-matching values in a LEFT-JOIN query. This optimization was explored in [16], and has not been presented as an alternative technique for comparing and eliminating stop-words from the original document.

```
SELECT keyword AS term
FROM ParseArray('{Query String}')
LEFT JOIN stopword
ON word = keyword
WHERE word is NULL;
```

Retrieval Queries

The pre-selection of the documents to rank can be achieved for conjunctive or disjunctive queries as well. The idea is to reduce the number of terms and documents to work with in the on-demand weighting and ranking models. Querying a collection for disjunctive or conjunctive operations traditionally requires building a query dynamically for searching for terms in the frequency table. Then these selections are constructed by adding "AND" or "OR" operations at the end of a selection. With this optimization, we re-use the previous framework for parsing and pre-processing a query, by purging the user query and then parsing it in the ParseArray temporary table. In a disjunctive query, the selected documents are the result of a LEFT-JOIN optimization.

```
SELECT DISTINCT docid
FROM termFrequency tf
LEFT JOIN ParseArray('{Query String}') pa
ON PorterAlgorithm(pa.keyword) = tf.term;
```

The conjunctive query corresponds to a division operation in the database, including the "termFrequency" table and the query table, for finding all the documents that contain all the query terms. Thus, we execute an optimization discussed previously in [16], where we execute this division operation by adding a GROUP-BY, to remove duplicates and to count how many unique terms each document contains from the query. Finally, we store in a temporary table the number of words in the query. As part of the implementation, we assume a single occurrence of a term in a user query.

```
SELECT docid
FROM termFrequency tf
LEFT JOIN ParseArray('{Query String}') pa
ON PorterAlgorithm(pa.keyword) = tf.term
WHERE pa.keyword IS NOT NULL
GROUP BY docid
HAVING COUNT(*) = {# Query Words};
```

The retrieval functions for three models imply a sequence of natural joins with the termFrequency and the "weight" table. The objective when designing both implementations was to reduce the size of the term tables as soon as possible. Thus, we created several temporary tables, such as the table "doc", which contains the pre-selected documents from the disjunctive or conjunctive query, but represents the set of all the documents that contain the desired terms. This temporary table is used as part of the on-demand technique to verify if the termInverseFrequency table contains all the required terms. Finally, we assume that the norm of the query has already been pre-computed.

In order to space in the queries, we decided to present them in relational algebra (instead of larger SQL). For the Vector Space Model (see Equation 5) we compute a series of aggregations in the "tmpd" and the "tmpqd" temporary table. The first one represents the norm of the documents that contain the desired terms and "tmpqd" stores the dot product between the query table and each document. At the end, we join the three tables to obtain the score of each document without the need to perform an aggregate function in this operation.

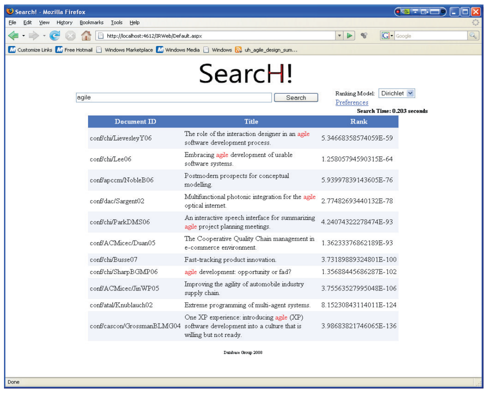
The ranking function involves several joins, including the term frequency table and the inverse frequency table. The first part of the query computes a join between the "query" table and the inverse frequency table, and this query is computed once. Then, the rest of the query computes the norm of the term vector of every selected document, which includes a join operation between the frequency table of the terms involved in the operation and the inverse frequency table. Then, using the selected documents, the norm of each document is computed with an inner join between the term frequency table and the inverse frequency table. The last part of the query is a join with the selected document norm and the "tmpqd" table and the already computed "query" table norm. The ranking function includes five joins for obtaining those results, but some tables are just obtained once and reused (i.e. the doc table). Also, as explained previously, in the static frequency table technique, the inverted frequency table is already pre-computed. But, in the on-demand technique, every term that is needed for its frequency must be searched in the term frequency table and if it is found in the term frequency table but not in the inverse frequency table, then its term inverse frequency must be inserted into the table. The "termFrequency" table is stored in a cluster, based on the docid keeping the terms in the descriptor vector together. This is in order to obtain better performance upon implementation in the DBMS.

$$\begin{aligned}
&\rho_{term}(doc \bowtie termFrequency)\\
&\rho_{tmpd}({}_{docid}\mathcal{F}_{sum(qd)}((term \bowtie Query)\\
&\quad \bowtie termInverseFrequency)\\
&\rho_{tmpqd}({}_{docid}\mathcal{F}_{sum(d)}(term \bowtie termInverseFrequency))\\
&\Pi_{docid,qd/(d*q)}(tmpd \bowtie tmpqd)
\end{aligned} \tag{5}$$

The Okapi Probabilistic Model (see Equation 6) requires additional information compared to the Vector Space Model, by requiring the length of the document (considered without stop-words) and the average size of a document in the collection.

$$\begin{aligned}
&\rho_{term}(doc \bowtie termFrequency)\\
&{}_{docid}\mathcal{F}_{sum(rank)}(termInverseFrequency\\
&\bowtie (documentData \bowtie (Query \bowtie term)))
\end{aligned} \tag{6}$$

The implementation of the Okapi formula in SQL is performed with three inner joins. The deepest query involves the matching documents of the desired queries, as in the Vector Space Model. Then we perform another inner join with this temporary document table and the termFrequency



Search Time: 0.203 seconds

Document ID	Title	Rank
conf/chi/LievesleyY06	The role of the interaction designer in an agile software development process.	5.34668358574059E-59
conf/chi/Lee06	Embracing agile development of usable software systems.	1.25805794590315E-64
conf/apccm/NobleB06	Postmodern prospects for conceptual modelling.	5.93997839143605E-76
conf/dac/Sargent02	Multifunctional photonic integration for the agile optical internet.	2.77482693440132E-78
conf/chi/ParkDMS06	An interactive speech interface for summarizing agile project planning meetings.	4.24074322278474E-93
conf/ACMicec/Duan05	The Cooperative Quality Chain management in e-commerce environment.	1.36233376862189E-93
conf/chi/Busse07	Fast-tracking product innovation.	3.73189889324801E-100
conf/chi/SharpBGMP06	agile development: opportunity or fad?	1.35688445686287E-102
conf/ACMicec/JinWP05	Improving the agility of automobile industry supply chain.	3.75563527995048E-106
conf/atal/Knublauch02	Extreme programming of multi-agent systems.	8.15230843114011E-124
conf/cascon/GrossmanBLMG04	One XP experience: introducing agile (XP) software development into a culture that is willing but not ready.	3.98683821746065E-136

Database Group 2008

Figure 4: Retrieval Web Interface.

table to obtain the second and the third part of the formula, leaving a last inner join operation to multiply each document term by its inverse frequency and then apply an aggregation operation on docid with a SUM on the final score of each term and a GROUP-BY.

The Dirichlet Prior Language Method (see Equation 7), is very close to the implementation in SQL of the OPM, performing joins in tables with similar size. Despite this, the ranking elements and the results of the aggregation are very different, due to the weight term product, the term weighting and the ranking equation.

$$\begin{aligned}
&\rho_{term}(doc \bowtie termFrequency)\\
&{}_{docid}\mathcal{F}_{product(rank)}(maxLikelihoodTermDoc\\
&\bowtie (documentData \bowtie (Query \bowtie term)))
\end{aligned} \tag{7}$$

The DPLM implementation is performed very similarly to the Okapi Probabilistic Function, but instead of performing a SUM in the aggregation, we obtain a product and we apply the given formula. The overall execution time is very similar to the OPM.

4. EXPERIMENTAL EVALUATION

We developed a digital library search engine, with a web interface in ASP .NET, that sends SQL queries to a DBMS, with the desired optimizations or query options, when executing the program for storing or querying. The test engine is divided into a web interface (see Figure 4) and an IR engine. The web interface interacts with the user and displays the results and the total elapsed time, and the IR-engine receives the commands, generates the queries and sends the workload. Also, we collect time measurements for several steps of the IR process.

The IR-engine manages the queries, via OBDC, between the client and the DBMS. The engine has been designed to send all the tasks to the server via SQL, and just serves as an SQL builder for the desired optimizations. All of the operations take place in the server. Also, the IR-engine manages the user configurations (that can be modified by the user), the test bed specifications, the file workload (if desired), the database source table, and server connection strings. The system that we used for the experimental evaluation of the IR engine was an Intel Core DUO E8400 CPU of 3.0 GHz, 4 GB in RAM and 300 GB in disk space, running Microsoft Windows XP SP2 and SQL Server 2005.

4.1 Corpus Preparation

The collection of documents was prepared using the publicly available DBLP bibliography and ACM Digital Library abstracts by an automated program (crawler) for preparing the corpus (see Table 1). In some cases, the abstracts were larger than expected and were reduced to less than one thousand characters. In order to proceed with the experimental section, we decided to work with two collections: a smaller collection from a corpus larger than 18000 documents, creating test runs of 500, 1000, 2000, 4000, 8000 and 18442 sets of documents, and a larger collection, where we repeated the original corpus 10 times for testing the search models with more than 180000 documents. We assumed for the rest of the tests that the documents, abstracts from our test bed, were already loaded into the database. Then we preprocessed these strings and stored them with the Baseline Inverted File technique, as presented in the entity relationship model in section 3.

The selected test workload was designed with five terms based on the maximum, average and minimum frequencies, the selected keywords were: design, simulation, cumulativ, disjunctiv and circumferenc to perform conjunctive and disjunctive queries (see Table 2). The workload was designed for querying all the possible combinations with any of these terms, in order to measure the performance of querying a term in the collection, although the combination with all the terms was ignored for measurement purposes. We considered in these experiments all the retrieved documents as relevant, and we ignored, for the time being, the precision and recall measurements, while focusing our experiments on performance. All the measurements in the experimental part are given in seconds.

Table 1: Documents in the collection.

Docs	Dif. Terms	Word Avg	Term Avg	Std dev
18442	782421	98.43	55.7	24.94

Table 2: Search terms.

Keywords	Frequency
design	8914
simulation	1442
cumulativ	17
disjunctiv	16
circumferenc	1

4.2 Storage

We studied the performance of our implementation for document pre-processing with a recursive query, and the results (see Table 3) showed that the performance for parsing documents is approximately 3690 words per second on average, and the implementation can be extended to work in parallel for a much better ratio, as showed in similar research by [6, 4]. The optimization analyzed in the storage phase corresponds to the NOT-IN and LEFT-JOIN operands for eliminating the stop-words from the document temporary table. The results showed a reduction between 1-6% in the total time of removing the words and storing them in the frequency table (see Table 3). The optimization always proved to be faster than using a NOT-IN operation, but the improvement percentage depends on the number of elements in both tables. Although these optimizations, the parsing and the stop-words removal, represent more than 80% of the total storage time (see Table 4).

Table 3: Stop-word removal & parsing in seconds.

Docs	NOT-IN	LEFT-JOIN	Parsing total time
500	9	8	12
1000	18	17	26
2000	37	35	53
4000	77	73	107
8000	160	150	227
18442	345	328	492

Table 4: Storage time for on-demand in seconds.

Docs	LEFT-JOIN %	Parsing %	Total time
500	34	50	24
1000	35	53	49
2000	35	53	100
4000	35	52	206
8000	34	51	441
18442	35	52	949

4.3 Term Weighting

For the term weighting step, we analyzed the on-demand and static techniques for computing the inverse frequency, or the probability, of the desired terms. Surprising results (see Table 5), that were computed for the Okapi inverse frequency, included a high overhead when verifying the existence of the required terms in the partial inverse frequency table. Thus, the time for executing this background check and then inserting the required terms is higher than weighing the whole collection in one pass.

Although at some point (see Figure 5), if no new terms are searched, the time consumption of the on-demand technique will remain with just the overhead cost of the LEFT-JOIN operation to search for missing terms in the partial term weight table. The performance remains steady when the total number of different terms has already been reached. On the other hand, time consumption of the static inverse frequency or probability computation will surpass that of the on-demand technique as the number of terms keeps increasing. Thus, deciding which strategy to apply depends on the workload and the number of different terms that are part of the collection.

Table 5: Weighting techniques for Okapi in seconds.

Documents	On-demand	Static
500	<1	<1
1000	1	<1
2000	1	<1
4000	2	1
8000	3	1
18442	3	2

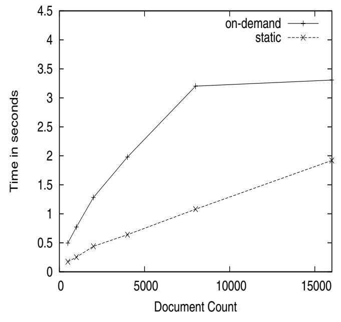


Figure 5: Inverse frequency.

4.4 Retrieval

We tested our document pre-selection in our three ranking models. The performance with conjunctive queries is faster than the performance for disjunctive queries for the premature reduction of the number of terms. The search for documents that contain all the searched terms is a time consuming operation. However, the limited number of findings speeds up the overall execution of a conjunctive query, except in the first case, when obtaining all the documents that contain all the queries proves to be a more time consuming operation than the disjunctive query. After more keywords are added into the query, the disjunctive operation involves a larger number of matches and therefore, a slower performance. According to the results that we obtained (see Table 6), the VSM implementation is slower than OPM and the DPLM implementations, except in the first case when just one keyword is involved in the search. In general, due to the number of operations that the VSM must perform, Okapi and DPLM appear as much better options. Also, it is well-known that Okapi is a good model for fixed-length documents, and due to the similarity amongst the join operations between OPM and DPLM, we observe a similar behavior. Also, the conjunctive query by itself is the most time consuming operation (GROUP-BY) in the scoring, but this early implementation in the document selection retrieves much fewer documents than the disjunctive operation. Thus, in the overall time for the ranking operations, the conjunctive operation appears as a much faster option.

Table 6: Conjunctive (C) & disjunctive (D) queries.

180000 documents						
	VSM		OPM		DPLM	
Terms	C	D	C	D	C	D
1	5	5	8	4	10	4
2	4	7	2	4	2	4
3	4	8	2	4	2	4
4	4	10	2	4	2	4

4.5 Discussion

We consider these results promising for future optimizations, and they proved to reduce the SQL overhead that has been assumed as one of the principal issues when working with a declarative language such as SQL. Also, these experiments helped us to explore and build an application with this SQL alternative taking advantage of indexes, clustered storage of related terms, User-Defined Functions, and faster SQL operations. As a result, we obtained results that showed over which circumstances any of these optimizations can be applied, taking into consideration the nature of the collections and the workloads.

In particular, the recursive query, described as part of the stripping tasks of the documents, has proven to be an effective option for breaking up the documents and storing them, pivoted and purged, in a temporary table in a standard SQL implementation. Contrary to previous research, our solution is not attached to the DBMS proprietary functions, or suited for extracting n-grams. The LEFT-JOIN operation gives an improvement of 1-6% instead of using a "traditionally proposed" NOT-IN operation. This saves a lot of time in the overall process. This task has to be executed for every document, parsed and stored into the collection. The same idea was used also as part of the on-demand term weighting step, reducing the total overhead of verifying the existence of terms in the inverse frequency or probability table.

For the static and on-demand techniques, we observed that normally a static ranking will be a much faster approach, but when we know that the collection includes millions of different terms, and the queries are over a few of these terms, we do not need to rank all of the terms in the collection. We also observed, that for values greater than 10000 different terms, the term weighting becomes the most time consuming operation in the overall process of preprocessing, storing and ranking. Lastly, the fact that we can use previously studied query optimizations and take them into the context of information retrieval to minimize the limitations of a database as a search engine is an idea that has not been previously explored, as we discuss later in the related work section. Also, we consider the improvement that we found significant, especially taking into consideration that these operations run for thousands of documents and terms in the database system.

5. RELATED WORK

The research on integrating IR systems with DBMSs is a proposal that has been around for several years, but just a few implementations have explored a SQL perspective. This is especially true for metadata integration, federated databases, multithreading and RAID systems [20]. Existing C search engines and libraries for information retrieval, such as Lemur Project, Mallet or Libbow [19, 13, 12] need to create their own indexes and own data structures for managing the text files, and they work completely independently from any relational database system. As stated in [8], the concept of using SQL to perform an IR search started with [1] parsing text into words using a C implementation and then storing them in a DBMS and performing operations in the collection with indexed tables. The advantage of using an SQL implementation, as described by [6], over existing solutions, even vendor implementations such as Microsoft SQL Server or Oracle, or "application specific" programs, is the portability of the standard SQL and its independence from the security model of the system, and fact that it does not compromise the stability and performance of the database. Even still, we accept that the overall performance continues to be limited compared to non-restrictive language implementations, and we propose our framework as an alternative option for processing documents already stored in DBMSs.

Subsequently, some authors [6, 4, 5] tried to extend their systems by applying the Vector Space Model without considering pre-processing or query optimizations of clustered databases, or by assuming multi-node processing or parallel computing. The OPM [11] and word stemming and parsing were developed [8], but differ from our research, because they did not address completely the problem of document pre-processing in the database, and different query optimizations are ignored precisely because they used distributed or parallel computation, which disguised their performance results. We additionally explored and tailored the the Dirichlet Prior Language Model to a relational database management system. The costly NOT-IN operation in the stop-word removal was ignored, as well as different inverse frequency computation strategies. Because of these limited numbers of approaches in the previous literature review, especially in the document pre-processing section, we decided to explore those problems without using a computer cluster, which has helped to improve the performance in many of these applications, and support our findings with some time experiments that were discussed in the previous sections.

6. CONCLUSIONS

Our system is an implementation of an IR framework mostly in standard SQL. In particular, we take advantage of recursive queries, SQL optimizations and UDFs for document pre-processing, storing, ranking, and retrieving. Our framework has not been compared in performance to existing IR-engines, but we are working on some benchmarks with non-restrictive language implementations and RBDMS commercial modules. The parsing recursive query presented an alternative SQL approach for breaking up the documents without using pre-loaded tables or DBMS specific implementations. Optimizations in the stop-word removal step proved to be highly effective, the computation of a temporary table when we retrieved the relevant documents for conjunctive or disjunctive queries showed a fast retrieval mechanism for the tested collection, and using static and on-demand techniques gives alternatives for implementation depending on the collection. The scalability of the solution proved to be linear, and we consider that these performance results will be similar between different DBMS, and even better for multithread systems. We also observed a satisfactory performance for a medium-sized digital library, and that our implementation of the VSM is competitive with the other two models.

Part of future work is to improve the time needed to load documents into the DBMS, which has become a bottleneck for an implementation of an IR system. We want to add data mining functionalities to explore the existing parsed documents, such as terms data cubes, and also implement other storing techniques and experiment with different indexes. The retrieval part proved to be fast enough to consider comparing the results with some benchmarks, but loading and pre-processing large amounts of text was reaching the limits of the DBMS capabilities. Also, we observed that our optimizations performed better than some other commonly used techniques for building IR-engines in SQL, and we are considering comparing our SQL application with a similar engine using just User-Defined Functions instead. Finally, we would like to explore how the research in DBMS for implementing text searching models can give acceptable performance in document processing and retrieving by exploiting the database's strengths in structured data management.

7. REFERENCES

[1] J. Driscoll D.A. Grossman. Structuring text within a relational system. *DEXA '92*, September 1992.

[2] H. Fang, T. Tao, and C. Zhai. A formal study of information retrieval heuristics. *SIGIR*, pages 49–56, 2004.

[3] K. Goda, T. Tamura, M. Kitsuregawa, A. Chowdhury, and O. Frieder. Query optimization for vector space problems. *SIGIR*, pages 416–417, 2001.

[4] T. Grabs, K. Böhm, and H.J. Schek. PowerDB-IR: information retrieval on top of a database cluster. *CIKM '01*, pages 411–418, 2001.

[5] T. Grabs, K. Böhm, 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.

[6] D.A. Grossman, D.O. Holmes, and O. Frieder. A parallel DBMS approach to ir in TREC-3. In *Text REtrieval Conference*, 1994.

[7] D. A. Grossman and O. Frieder. *Information Retrieval: Algorithms and Heuristics (second edition).* Springer, 2004.

[8] D. Holmes. SQL text parsing for information retrieval. In *CIKM '03*, pages 496–499, New York, NY, USA, 2003. ACM.

[9] M. Porter K. Lubell. Porter algorithm T-SQL. Url: http://tartarus.org/ martin/PorterStemmer/tsql.txt, May 2006.

[10] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD '06*, pages 563–574, New York, NY, USA, 2006. ACM.

[11] D.A. Grossman O. Frieder M.C. McCabe, D. Holmes. Parallel platform-independent implementation of information retrieval algorithms. *PDPTA'00*, 2000.

[12] A.K. McCallum. Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering. http://www.cs.cmu.edu/ mccallum/bow, 1996.

[13] A.K. McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

[14] M. Marin, C. Gomez. Load balancing distributed inverted files. In *WIDM '07*, pages 57–64, Lisbon, Portugal, 2007.

[15] C. Ordonez. Optimizing recursive queries in SQL. In *ACM SIGMOD Conference*, pages 834–839, 2005.

[16] C. Ordonez and J. García-García. Referential integrity quality metrics. *Decision Support Systems Journal*, 44(2):495–508, 2008.

[17] C.D. Paice. Another stemmer. *SIGIR Forum*, 24(3):56–61, 1990.

[18] G. Salton, A. Wong, and C.S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11), 1975.

[19] L. Si T.T. Avrahami, L. Yau and J. Callan. The fedlemur project: Federated search in the real world. *Journal of the American Society for Information Science and Technology*, 57(3):pp.347–358, 2006.

[20] G. Weikum. DB&IR: Both sides now. pages 25–30, 2007.

[21] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.