Object-Relational Database Representations for Text Indexing

Panagiotis Papadakos  Yannis Theoharis  Yannis Marketakis
Nikos Armenatzoglou  Yannis Tzitzikas

Email: {papadako, theohari, marketak, armenan, tzitzik}@ics.forth.gr

Institute of Computer Science, FORTH-ICS, GREECE, and
Computer Science Department, University of Crete, GREECE

Abstract. One of the distinctive features of Information Retrieval systems comparing to Database Management systems, is that they offer better compression for posting lists, resulting in better I/O performance and thus faster query evaluation. In this paper, we introduce database representations of the index that reduce the size (and thus the disk I/Os) of the posting lists. This is not achieved by redesigning the DBMS, but by exploiting the non 1NF features that existing Object-Relational DBM systems (ORDBMS) already offer. Specifically, four different database representations are described and detailed experimental results for one million pages are reported. Three of these representations are one order of magnitude more space efficient and faster (in query evaluation) than the plain relational representation.

1 Introduction

Most information retrieval systems and Web search engines use inverted files, which have been proven to be very efficient for answering queries [31]. However, the last years the scope of services that such systems offer (or should offer) is getting wider. For instance, they should be able to handle structured data (e.g. Google Base), structured documents or semi-structured data (e.g. XML), annotations/tags and multimedia data types. Furthermore a plethora of new tasks, quite different from the classical query evaluation task, are being performed including data mining algorithms, machine learning, facet-based exploration (e.g. [29,6]), collaborative recommendation and filtering.

For these reasons, the index of an engine should be easily extensible and able to accommodate various types of data and metadata. The typical amenities that a DBMS offers (e.g. declarative query languages, query optimizers), are very useful when coping with multiple types of data (and metadata). Moreover, several other techniques and algorithms (e.g. for OLAP) could be exploited for enabling services beyond simple search. In brief, it is widely accepted, that almost all advanced applications (including search engines) need to manage both structured data and text documents [9]. Fortunately, recent work on DB brings it closer to IR. For instance there have been proposed

1 http://www.google.com/base
methods for ranking query results [1], keyword searching in databases [17,26], computing efficiently top-k queries [8,18,22], optimizing text-centric tasks [19], offering exploration services [7], and systems that somehow blend such capabilities have emerged (e.g. [5]). All these works focus on providing efficient best-match retrieval services for structured data. However the management of texts is of prominent importance. For this reason, in this paper we elaborate on building and managing the index of documents using a DBMS. One of the distinctive features of an IR index (e.g. inverted file) is that it offers better compression for sparse arrays resulting in better I/O performance. In order to alleviate this inefficiency of DBMSs, in this paper we introduce database representations of the index that reduce the size (and thus the disk I/Os) of the representation of the posting lists. This is not achieved by redesigning the DBMS, nor by implementing an additional data type, but by exploiting the non 1NF features that existing ORDBMS systems offer. In brief, Object-Relational DBMSs extend the relational model to include useful features from object-orientation, e.g. complex types, and extends relational query languages, e.g. SQL, to deal with these extensions.

In this paper, we introduce four different representations (database schemas) for indexing texts and we report comparative experimental results. All the experiments have been performed over Mitos\(^2\). The index of Mitos is based on PostgreSQL (from now on PSQL). Four different database representations of its index were tested for various tasks. The crux of our findings is that the support of set-valued attributes by ORDBMSs can offer significant storage space savings and query evaluation speedup. To the best of our knowledge this is the first work that exploits the Object-Relational features of existing DBMS for the benefit of the index. There are only few slightly related works that are discussed in Section 2. We do not compare these database representations with inverted files because our focus is to identify the more scalable DB representations and not to replace inverted files. Our findings can significantly speedup text-centric tasks in settings where a DBMS is already in place. For instance, YouTube uses MySQL\(^3\), while there are Semantic Web repositories, like SWKM\(^4\), that are based on PSQL.

The rest of this paper is organized as follows: Section 2 summarizes previous work on DBMS-based IR systems. Section 3 discusses DBMS indices and presents four possible database index representations. Section 4 reports experimental results. Finally, Section 5 concludes the paper and identifies issues for further work and research.

2 Related work

One of the first attempts to provide information retrieval functionality such as keyword and proximity searches by using user defined operators, is described in [15]. Some years later, the first IR system over a DBMS was presented [16]. Relevance ranking queries were implemented using unchanged SQL on an AT&T DBC-1012 parallel machine for TREC-3. They found that the DBMS overhead was somewhat high, but tolerable for a large scale machine, emphasizing that using a DBMS can spread the workload

\(^2\) [http://groogle.csd.uoc.gr:8080/mitos/]
\(^3\) See [http://highscalability.com/youtube-architecture]
\(^4\) [http://athena.ics.forth.gr:9090/SWKM/]
across large numbers of processors. Recently, several approaches to merge DB’s structured data management and IR unstructured text search facilities have been proposed. According to [21], they can be classified in four different categories:

- **Middleware approach** This approach integrates DB and IR engines at the application level [9]. Query evaluation and indexing is provided by the IR engine, while the DBMS manages the documents and other metadata. According to [21] the basic drawback of this approach is the difficulty to synchronize the DBMS document contents and the IR’s index.

- **DBMS extension by loose coupling** Most DBMS offer extensible architectures using a high level interface, which can be used to integrate IR functionalities. Although such extensions can be easily implemented, it is not recommended according to [30] when high performance is desired. Systems based on this approach include PowerDB-IR [14] (a scalable IR system for frequently changing data sets), QUIQ [20] (a collaborative customer support application, where a DBMS holds all the data and an external server maintains the index), TopX [28] (an Oracle based engine for XML and plain text data with top-k retrieval) and HySpirit [13] (a hypermedia retrieval engine using probabilistic Datalog).

- **DBMS extension by tight coupling** In this approach, new data types and functionality for IR features are integrated into the core of the DBMS engine or the reverse (IRMS Information Retrieval & Management System) [21]. Tight coupled systems include Oddyseus [30], an engine build over an ORDBMS engine, and MonetDB/X100 [10], a column oriented storage management based system.

- **DB-IR system from scratch** This approach suggests developing new DB-IR architectures from scratch [2,9] aiming at providing structural data independence, generalized scoring, and flexible and powerful query languages.

The approach that we investigate in this paper falls more into the loose coupling approach. No special data types are introduced and the retrieval models are implemented on top (at a separate API that connects through jdbc to the DBMS). However we do exploit the SQL:1999 ARRAY type, allowing the storage of a collection of values directly in a column of a table, and the PSQL (8.2 and above) hstore data type that is useful for storing semi-structural data and variable in number fields. To the best of our knowledge, the only related work is that of Oddyseus [30] and MonetDB [10]. The difference with our work is that Oddyseus adopts a tight-coupling approach where the DBMS is extended with new data types, while MonetDB implements an inverted file-like data structure at the physical layer. Specifically Oddyseus adds a B-tree at the posting list of each term in order to speedup the lookup of document identifiers and the evaluation of multi-word queries. However detailed experimental results, regarding the space overhead and the speedup of this approach, are not reported.

In comparison to [24], this paper (a) contains a detailed discussion of all related works, (b) introduces and investigates an additional database representation (that yields smaller in size tables), (c) reports experimental results over a one order of magnitude bigger corpus, and (d) reports experimental results for document-based access tasks.
3 On DBMS-based Indices

3.1 DBMS Limitations

Roughly, an inverted file comprises entries of the form \((t, occ)\) where \(t\) is a term while \(occ\) stands for the occurrences of \(t\) in the corpus. Occurrences may comprise only document identifiers, or also the weight and/or the positions (exact or block-based) of \(t\) in each document. Term occurrences occupy most of the space of the index and for this reason special number encodings \(^3\) are usually employed to reduce the space required.

A straightforward implementation over a relational DBMS would occupy much more space than an inverted file. Consider for example the entry \((t, \{d_1, d_3, d_5\})\). That would be represented by three tuples \([t, d_1], [t, d_3], [t, d_5]\) resulting in wasted space. Furthermore, special number encoding schemes are not currently supported by DBMSs. Apart from the higher storage space requirements, we expect the query response time to be higher for a DBMS based index, since more I/O’s are expected to be needed. This has been experimentally verified in \(^{[25]}\), where Mitos was found less efficient than Terrier \(^{[23]}\). However, the adoption of set-valued attributes that are offered by ORDBMSs can alleviate these problems as we will describe in detail later on.

3.2 DBMS Features

Since the scope of services that IR systems and Web search engines should offer is constantly widening, it is important that they are based on an easily extensible index. Using a DBMS index, the extension of the index schema with additional columns and relations is rather straightforward. For instance, the index can be extended with various information, like users, dates, tags, metadata, in order to support more sophisticated queries and retrieval models. Furthermore, as the physical layer is handled by the DBMS, the processes of index construction and maintenance can be simplified (i.e. there is no need for creating and merging partial indices in order to construct the index of a big corpus).

Finally, the advances in DBMS for multicore and clustered systems can transparently benefit IR systems that are built on top, simplifying the creation of parallel and distributed systems. For instance, PSQL can take advantage of more than one available system CPUs/cores (e.g. for dispatching queries), while the ongoing project pgpool-II\(^5\) works on supporting more advanced distributed query processing features, such as the dispatching of parts of a query plan to the available CPUs.

3.3 The Indexer of Mitos

Mitos is a recently developed Web search engine in Java, that offers a wide spectrum of functionalities (for a detailed description see \(^{[25]}\)). Synoptically, Mitos is equipped with an advanced stemmer for the Greek language, offers real time result clustering, advanced link analysis techniques and facet-based exploration services \(^{[29]}\). Mitos adopts the \(tf-idf\) weighting scheme and uses PSQL for managing its index. For each term it keeps a) its document frequency (\(df\)) in the collection and b) its term frequency (\(tf\))

\(^5\) http://pgpool.projects.postgresql.org/
for each document. One of the main differences of Mitos compared to other search engines, is that it does not store to the index the positions of term occurrences in documents. Instead, Mitos stores the lexically analyzed extracted text of the crawled pages, to the filesystem. When Mitos returns the query results to the user, it parses the stored copies of the texts of the relevant documents, to find the snippets with respect to the query terms. This is needed only for the documents that lie in the result pages the user will visit.

To compute the answer of a query the index should provide efficient term-based access (i.e. inverted files). However there are other tasks that require document-based access. Such tasks include document deletion, query expansion (retrieve the most highly ranked terms of the top-ranked documents) and relevance feedback (retrieve the terms of the documents for which the user provided feedback).

### 3.4 DB Representations for Occurrences

Here we introduce four different database representations for the index (shown in Table 1). All comprise a relation `document`, that stores for each document its id, url, norm, and PageRank score. They only differ on how they store words and occurrences.

#### (PR) Plain-Relational

This is the representation currently in use by Mitos and is like the one used in [16][14][28]. The relation `word` stores the words, their identifiers and their `df`, while triples of the form `[word_id, doc_id, tf]` are stored in the relation `occurrence`. The main drawback of this representation is that each word_id is stored for each document in which it appears in. This redundancy results in high storage space.

#### (OR) Object-Relational

This representation exploits the set-valued attributes supported by PSQL in order to reduce the space occupied by occurrences. It exploits the `point` datatype offered by PSQL for representing the pairs `(doc_id, tf)`. For each word_id an array of points is stored. In this way each word_id is stored exactly once in the table `occurrence`.

#### (COR) Compact Object-Relational

This representation drops the relation `word`, since word_id is a primary key in both `word` and `occurrence` tables, and moves `word_name` and `df` to `occurrence` table.

#### (HOR) HStore Object-Relational
This representation is like COR, except that it uses the PSQL hstore data type instead of a point array. hstore is a data type for storing sets of (key,value) text pairs in a single PSQL data field. For HOR the key is the doc_id and the value is the tf.

### 3.5 PSQL Indices

In order to provide more efficient access paths to the relations, we need to build appropriate PSQL indices. Regarding document table, the access is done given the doc_id, i.e., an attribute of integer type. We have two choices for the index type we can build on doc_id, namely either a $B^+\text{Tree}$ or Hash index. Regarding word table, the access is done given the name, and we can use a $B^+\text{Tree}$ or Hash index. Furthermore, we could also exploit the Trie index, which has been implemented on top of PSQL, as a part of the SP-GiST index family [4,11]. According to [12], the Trie index offers more than 150% performance increase for exact search matches over to PSQL $B^+\text{Trees}$, and scales better regarding size. Finally, for the occurrence table, possible choices are either a $B^+\text{Tree}$ or Hash index, on word_id. For the COR and HOR though, the word and occurrence tables have been merged. Since the access is done given the name, we can create either a $B^+\text{Tree}$, Hash or Trie index on it. Moreover in order to accelerate document based access for HOR, GeneralizedInvertedIndex(GIN) indices can be build on top of the hstore occur attribute. Unfortunately we could not accelerate document based access for OR and COR, since PSQL does not offer functionality to build indices on top of arrays. Table 2 summarizes the possible combinations.

### 3.6 Bulk Index Creation/Updates

It is more than evident, that the benefits from using a DBMS are at the expense of the data storage and retrieval efficiency. Specifically, the guarantee of the ACID properties, the concurrency control, the update of DBMS indices and their possible reorganization on disc, may harm the efficiency of the index. In order to reduce such overheads, we use the copy function of PSQL during the index creation. In this manner, we skip the concurrency control, as well as several integrity constraints checks, while at the same time we minimize the I/O’s needed to insert a specific amount of new tuples. Moreover, in case we want to add a new document collection to an existing index, we first drop the DBMS indices, then we insert the new tuples, and finally re-create the indices at the end. After all documents have been indexed, for each document $d$ we compute the norm ($\|d\|$) of its vector ($d$) as defined by the tf-idf weighting scheme, and store it in the norm field, in order to speed-up query evaluation.

---

<table>
<thead>
<tr>
<th>Repr.</th>
<th>Document Table</th>
<th>Word Table</th>
<th>Occurrence Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attr.</td>
<td>Type</td>
<td>Attr.</td>
<td>Type</td>
</tr>
<tr>
<td>Doc.</td>
<td>doc_id</td>
<td>word_id</td>
<td>word_name</td>
</tr>
<tr>
<td>ORM</td>
<td>$B^+\text{Tree}$</td>
<td>word_id</td>
<td>word_name</td>
</tr>
<tr>
<td>COR</td>
<td>$B^+\text{Tree}$</td>
<td>word_id</td>
<td>word_name</td>
</tr>
<tr>
<td>HOR</td>
<td>$B^+\text{Tree}$</td>
<td>word_id</td>
<td>word_name</td>
</tr>
</tbody>
</table>

Table 2. Combinations Between Representations and Indices
3.7 Query Evaluation

Table 3 shows the queries needed according to the vector space model for each representation, assuming the query "information retrieval" (transformed to "informat retriev" because of stemming). The query $q_{word}$ is issued to get the $df$ values of the query terms, $q_{occ}$ to get the $tf$ values of the query terms in the documents they appear in, and $q_{doc}$ to get the norms and ranks of the corresponding documents. In COR and HOR, the number of issued queries is decreased by one, since the $df$ values are now stored in the occurrence table instead of the word table. These elementary queries can be used for implementing various ranking methods. Essentially, they provide the interface that a classical inverted file exposes to the query evaluation component.

4 Experimental Results

We conducted experiments on a desktop PC with a Pentium IV 3.4 GHz processor, 2 GB main memory and a single 7200 rpm SATA hard disk, on top of Linux distribution Ubuntu v8.04, using a 2.6.24 kernel and the ext3 filesystem (mounted with the default options). We used PSQL v8.3.3, configured with 1600 MB as shared_buffers. Our collection contained documents of various formats (.html, .pdf, .doc, etc) including pages crawled from our university and FORTH domains. Specifically, it comprises 1,004,721 documents, 216,449 distinct terms and its total size is approximately 198 GB. The average size of each document is around 200 KB (due to the large number of .doc and .pdf files), and the average number of words in each document is 239.

4.1 Database Size and Copy Times

In this section we focus on the storage requirements of the occurrences, as this is the crucial point and the main difference between the four representations. We will use Object Relational Inverted File (ORIF) to refer to the OR, COR and HOR representations, since they represent occurrences roughly the same (the only difference is the

---

Table 3. Queries for each Representation

<table>
<thead>
<tr>
<th>Repr.</th>
<th>Word</th>
<th>Occ</th>
<th>Doc</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR</td>
<td>SELECT id, df FROM word WHERE name IN ('informat', 'retriev')</td>
<td>FROM occurrence WHERE word_id IN (informat_id, retriev_id)</td>
<td>SELECT id, norm, rank FROM document WHERE id IN (doc1, doc2, ..., docN)</td>
</tr>
<tr>
<td>OR</td>
<td>SELECT id, df FROM word WHERE name IN ('informat', 'retriev')</td>
<td>FROM occurrence WHERE word_id IN (informat_id, retriev_id)</td>
<td>SELECT id, norm, rank FROM document WHERE id IN (doc1, doc2, ..., docN)</td>
</tr>
<tr>
<td>COR</td>
<td>-</td>
<td>SELECT word, name, occur, df FROM occurrence WHERE word_name IN ('informat', 'retriev')</td>
<td>SELECT id, norm, rank FROM document WHERE id IN (doc1, doc2, ..., docN)</td>
</tr>
<tr>
<td>COR</td>
<td>-</td>
<td>SELECT word, name, occur, df FROM occurrence WHERE word_name IN ('informat', 'retriev')</td>
<td>SELECT id, norm, rank WHERE FROM document id IN (doc1, doc2, ..., docN)</td>
</tr>
</tbody>
</table>

---

6 http://www.uoc.gr
7 http://www.forth.gr
### Table 4. Size Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>number of word occurrences in the entire collection</td>
</tr>
<tr>
<td>D</td>
<td>number of documents</td>
</tr>
<tr>
<td>N_d</td>
<td>∑_{i=1..D} w(d_i) where w(d_i) is the number of distinct words in document d_i.</td>
</tr>
<tr>
<td>W</td>
<td>number of distinct words of the entire collection</td>
</tr>
<tr>
<td>t</td>
<td>tuple size (the space overhead for a tuple in a DBMS)</td>
</tr>
<tr>
<td>f</td>
<td>field size of a tuple</td>
</tr>
</tbody>
</table>

particular PSQL data type employed). For each case, i.e. PR and ORIF, we consider two different settings, depending on whether the positions of the occurrences are stored or not in the index. By adopting the notations described in Table 4, the size of each representation can be estimated as follows:

- **PR (without positions):** \( N_d (3f + t) \)
  
  \( N_d \) is multiplied by \( (3f + t) \), because for each occurrence we have to keep a tuple containing the corresponding \text{word_id}, \text{doc_id} and \text{tf}. Recall that \( t \) is the tuple overhead of the DBMS and is independent of the attributes of the tuple.

- **PR (with positions):** \( N_d (3f + t) + N (3f + t) \)
  
  \( N_d \) is multiplied by \( (3f + t) \) for the same reason as in the PR (without positions) case. In addition, \( N \) is multiplied by \( (3f + t) \), since for each occurrence we have to keep a tuple containing the corresponding \text{word_id}, \text{doc_id} and \text{position}.

- **ORIF (without positions):** \( W(f + t) + N_d 2f \)
  
  We have to store \( W \) tuples holding for each word the \text{word_id} and \( N_d \) pairs of \text{doc_id} and \text{tf}. No extra \( t \) has to be payed as they are stored in the same tuple.

- **ORIF (with positions):** \( W(f + t) + N_d 2f + Nf \)
  
  Again we have to store \( W \) tuples holding for each word the \text{word_id}, and \( N_d \) pairs of \text{doc_id} and \text{tf} in the same tuple. Moreover, we have to store in the same tuple \( N \) fields holding the positions where terms appear in.

Hereafter we focus on the case where we do not store positions. It is clear that ORIF occupies less space than PR. The inequality \( ORIF < PR \) yields: \( W(f + t) + N_d 2f < N_d (3f + t) \) \( \Leftrightarrow \) \( W(f + t) < N_d f + N_d t \) \( \Leftrightarrow \) \( W < N_d \) \( \Leftrightarrow \) \( W < N_d \) where \( W \) denotes the vocabulary, not its cardinality) will appear in at least one document (\( W = N_d \) if each distinct word appears in exactly one document and each document contains exactly one distinct word). Regarding the lower and upper bounds of \( N_d \) (recall that \( N_d = \sum_{i=1..D} w(d_i) \)), as \( 1 \leq w(d_i) \leq W \), it follows that \( D \leq N_d \leq DW \). Moreover if we assume that each document has \( w_{avg} \) distinct words, then \( N_d = w_{avg} D \). In our collection \( w_{avg} \) is 239 words, so \( PR \) is expected to occupy much more space than \( ORIF \). In the case that we also store term positions, it is clear that \( ORIF \) will again require less space, as in \( PR \) we use \( N(3f + t) \) to store the positions instead of just \( N \ast f \) in \( ORIF \).

Regarding the physical database size for each representation, we consider that the PSQL storage requirement for string types is 4 bytes plus the actual string size, while the storage requirement for integers and floats (considering the \text{int4} and \text{float4}
The sizes of the tables for each representation that correspond to our collection are given in Table 5. Notice that the sizes of ORIF are significantly smaller (more than one order of magnitude). Specifically, PR occupies 10.7 GB, while ORIF occupy around 0.5 GB. This means that the storage space of ORIF is roughly 0.25% of the total collection size. As a consequence, the times to copy the tables are significantly smaller for ORIF, in comparison to PR, offering a much more scalable solution, as far as indexing time and size are concerned. Specifically, PR needs almost 3 days to copy the tables, while the other representations need only 30 minutes or less.

### 4.2 Indices Size and Creation Times

The sizes of the PSQL indices for each representation are shown in Table 6. Again the space difference between the representations is more than one order of magnitude. This is also reflected to the PSQL index creation times. Specifically the process takes some seconds in ORIF, and roughly a day for PR. We could not evaluate the Trie index, as it only accepts words with latin characters and our test collection mainly contained Greek documents. In addition we could not evaluate GIN indices, in order to accelerate

---

<table>
<thead>
<tr>
<th>Repr.</th>
<th>Number of Rows (8KB each) per Table</th>
<th>Number of Tuples per Table</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Document</td>
<td>Word</td>
<td>Occurrence</td>
</tr>
<tr>
<td>PR</td>
<td>35,654</td>
<td>1,278</td>
<td>1,301,657</td>
</tr>
<tr>
<td>OR</td>
<td>35,654</td>
<td>28,860</td>
<td>64,514</td>
</tr>
<tr>
<td>COR</td>
<td>35,654</td>
<td>24,106</td>
<td>59,760</td>
</tr>
<tr>
<td>HOR</td>
<td>35,654</td>
<td>24,106</td>
<td>59,760</td>
</tr>
</tbody>
</table>

| Table 5. DB Tables Size in Pages (8 Kb) and Indexing Times |

<table>
<thead>
<tr>
<th>Repr.</th>
<th>Database Table Indices</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Document</td>
<td>Word</td>
</tr>
<tr>
<td>PR using Hash</td>
<td>4,666</td>
<td>2,050</td>
</tr>
<tr>
<td>PR using B^+ Tree</td>
<td>2,208</td>
<td>720</td>
</tr>
<tr>
<td>OR using Hash</td>
<td>4,666</td>
<td>2,050</td>
</tr>
<tr>
<td>OR using B^+ Tree</td>
<td>2,208</td>
<td>720</td>
</tr>
<tr>
<td>COR using Hash</td>
<td>4,666</td>
<td>−</td>
</tr>
<tr>
<td>COR using B^+ Tree</td>
<td>2,208</td>
<td>−</td>
</tr>
<tr>
<td>HOR using Hash</td>
<td>4,666</td>
<td>−</td>
</tr>
<tr>
<td>HOR using B^+ Tree</td>
<td>2,208</td>
<td>−</td>
</tr>
</tbody>
</table>

| Table 6. Indices Size in Pages (8 KB) and Creation Times |

---

8 PSQL version 8.3 supports arrays of composite types. Thus we could create a composite type (holding an int4 and a float4 (8 bytes) instead of the point type), reducing the memory size of the array to half.
In general, the results show that $B^+\text{Tree}$ indices occupy half of the size of Hash indices.

### 4.3 Query Evaluation Times

To measure query evaluation times, we adopted the following scenario: for each of the four representations and for each PSQL index combination, we: a) execute all the queries of the corresponding representation with 1, 2, 3 and 4 terms, b) repeat the above queries 10 times and c) calculate average times. We do not include the time to receive/scan the results. The terms contained in the above queries were different (for each query and for each iteration of the experiment) and they were selected based on their $df$. Specifically, we selected frequently occurring terms with a $df$ value about 300,000. The big $df$ number implies big overhead to the DBMS. The crash we had encountered in our previous experiments using PSQL 8.0, due to the large number of $doc ids$ passed in the IN list of the $q_{doc}$ queries, was solved after upgrading to PSQL 8.3. The aforementioned times were gathered through the Aggregator toolkit which is written in Java. This means that the measured times include the overhead of the JDBC driver (version 8.3-603 JDBC 4), an overhead that also exists in the Mitos engine.

As one can observe from the times reported in Table 7, ORIF representations are one order of magnitude more efficient than $PR$, due to the efficiency in occurrence table. More precisely, ORIF are approximately 20 times faster than $PR$ for all queries, although the ORIF index is only an order of magnitude (see Table 5) smaller than $PR$ index. This is due to the fact that ORIF indices, fit in main memory, so every page that is fetched in memory, is constantly kept there. Comparing ORIF representations, we observe that $OR$ and $COR$ have an identical performance, while $HOR$ is slightly

<table>
<thead>
<tr>
<th>Repr.</th>
<th>1 term</th>
<th>2 terms</th>
<th>3 terms</th>
<th>4 terms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>q</td>
<td>w</td>
<td>q</td>
<td>w</td>
</tr>
<tr>
<td>$OR\ (H)$</td>
<td>10</td>
<td>846</td>
<td>1,952</td>
<td>2,314</td>
</tr>
<tr>
<td>$PR\ (B^+)$</td>
<td>6</td>
<td>766</td>
<td>2,143</td>
<td>2,915</td>
</tr>
<tr>
<td>$COR\ (H)$</td>
<td>586</td>
<td>3,391</td>
<td>4,247</td>
<td>1,618</td>
</tr>
<tr>
<td>$COR\ (B^+)$</td>
<td>598</td>
<td>4,447</td>
<td>5,245</td>
<td>1,569</td>
</tr>
<tr>
<td>$HOR\ (H)$</td>
<td>1,023</td>
<td>3,944</td>
<td>4,967</td>
<td>1,280</td>
</tr>
<tr>
<td>$HOR\ (B^+)$</td>
<td>127</td>
<td>3,208</td>
<td>3,335</td>
<td>255</td>
</tr>
</tbody>
</table>

Table 7. Query Evaluation Times (msec)
faster, especially when using a $B^+\text{Tree}$ index. A common behavior for all representations is the slow $q_{doc}$ query times, which is actually the bottleneck for ORIF. This is due to the long IN list of doc_ids. We found that passing more than 250,000 doc_ids makes the $q_{doc}$ query too slow. We tackled this problem by dividing the IN list in blocks of 250,000 doc_ids and submitting one query for each block of the list. Subsequently we summed the gathered times. In the future we plan to investigate whether we can reduce the overhead of such queries by using temporary tables. Regarding the DBMS indices, we can conclude that $B^+\text{Tree}$ indices are the best choice since they provide equivalent or slightly better performance to Hash indices, while occupying half the storage space.

4.4 Query Expansion Times

To measure query expansion times we used the following scenario: for all terms in the top-5 results of a given query, we compute the sum of their $tfs$ in these documents and suggest to the user the 5 terms with the highest sum. Unfortunately, the gathered times were unacceptable for PR (almost 16 hours), since no index over doc_ids existed and PSQL performed a slow sequential scan over a table of 240,000,000 tuples. This task is much faster for ORIF (19.8 minutes), but again expensive, since an index over the array and hstore values, was not build. An approach to speed-up such tasks, is to store also a direct index (keeping for each doc_id the set of word_ids it contains). This index can be represented in an ORIF-like representation. We expect the total size of the two ORIF representations (inverted and direct) to be less than that of PR, and at the same time should provide faster term-based and document-based access services.

![Table Size vs Log(MB)](image)

**Fig. 1.** Size of Tables and Indices (in log scale)

5 Conclusion

In this paper we proposed and evaluated four different ORDBMS representations for text indexing. ORIF representations were found to be the most efficient, being one order of magnitude less space consuming and more than 20 times faster in query evaluation compared to PR. Specifically, for a collection of 1 million documents occupying
200 GB, \(PR\) needs almost 3 days to copy the tables, while \(ORIF\) representations need 30 minutes or less. The \(PR\)-index occupies 10.5 GB, while the rest representations need only 500 MB. This means that the \(ORIF\) index is the 0.25% of the collection size. Figures 1.2(a), 2(b), 2(c) and 2(d) summarize the results of the experimental evaluation. It is worth mentioning that almost all previous related works (e.g. \([16,14,28]\)) adopt a \(PR\)-like representation. As there are numerous applications based on ORDBMS, our findings can be exploited for enriching these applications with more scalable IR capabilities. To avoid misunderstandings, we do not suggest the adoption of databases instead of inverted indices, we just identified ways to speedup DBMS-based text indices.

![Graphs](image1.png)

(a) Times to Copy DB Tables (in log scale)  
(b) Creation Times of PSQL Indices (in log scale)  
(c) Query Evaluation using \(B^+\)Trees  
(d) Query Evaluation using Hash

**Fig. 2.** Index Creation and Query Evaluation Times

Although some DBMSs currently provide tuple ranking based on text-valued attributes (e.g. Oracle 9i Text extension, postgreSQL Full Text Search, etc), an implementation over these services does not allow supporting different retrieval models. Instead the ranking would be tightly coupled with the peculiarities of the particular DBMS. For this reason we based query evaluation on a small set of elementary queries that enable implementing several retrieval models in a flexible manner. However, an alternative approach would be to use fewer and more complex SQL queries that could even compute the ranked set of objects in one shot (depending on the retrieval model). This is an additional issue for further research. Furthermore, we plan to compare the DBMS approach with the classical inverted file approach on the same collection, and to compare the ef-
iciency of $B^+Tree$ indices with the $tree-Trie$ index that has been proposed to
index relationships with set-value attributes. Finally, and in order to optimize document
based access on the HOR representation, we plan to evaluate $GIN$ indices on top of the $hstore$ values. The evaluation of the efficiency in case of concurrent queries, as well
as the investigation of the applicability of parallelization techniques (e.g. map-reduce)
over a DBMS-index, are subject for future research.

References

1. S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated Ranking of Database Query
Results. In Proc of the First Biennial Conference of Innovative Data Systems Research
(CIDR 2003), Asilomar, California, USA, January 2003.
& DB Integration. In Proc of the 3rd Biennial Conference on Innovative Data Systems
6. O. Ben-Yitzhak, N. Golbandi, N. Har’El, R. Lempel, A. Neumann, S. Ofek-Koifman,
of the International Conference on Web Search and Web Data Mining (WSDM’08), pages
Proc. of the 2004 ACM SIGMOD International Conference on Management of Data (SIG-
8. S. Chaudhuri and L. Gravano. Evaluating Top-k Selection Queries. Proc. of the 25th In-
ternational Conference on Very Large Data Bases (VLDB’99), pages 397–410, September
1999.
9. S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR Technologies:
What is the Sound of One Hand Clapping? Proc. of the Conference of Innovative Database
IR using array databases. The VLDB Journal The International Journal on Very Large Data
11. M. Y. Eltabakh, W. G. Aref, and R. Eltarras. To Trie or Not to Trie? Realizing Space-
partitioning Trees inside PostgreSQL: Challenges, Experiences and Performance. Technical
Report TR-05-008, Department of Computer Science, Purdue University, USA, April 2005.
12. M. Y. Eltabakh, R. Eltarras, and W. G. Aref. Space-Partitioning Trees in PostgreSQL: Real-
ization and Performance. In Proc. of the 22nd International Conference on Data Engineering
(ICDE’06), page 100, Washington, DC, USA, April 2006. IEEE Computer Society.
13. N. Fuhr and T. Rölleke. HySpirit — A Probabilistic Inference Engine for Hypermedia Re-
Cluster. In Proc. of the 10th International Conference on Information and Knowledge Man-
agement (CIKM’01), pages 411–418, New York, NY, USA, November 2001. ACM.


