Mitos: Design and Evaluation of a DBMS-based Web Search Engine

Panagiotis Papadakos   Yannis Theoharis   Yannis Marketakis
Nikos Armenatzoglou   Yannis Tzitzikas
Institute of Computer Science, FORTH-ICS, GREECE, and
Computer Science Department, University of Crete, GREECE
Email: {papadako, theohari, marketak, armenan, tzitzik}@ics.forth.gr

Abstract

Engineering a Web search engine offering effective and efficient information retrieval is a challenging task. Mitos is a recently developed search engine that offers a wide spectrum of functionalities. A rather unusual design choice is that its index is based on an object-relational database system. This paper discusses the benefits and the drawbacks of this choice (compared to the classical inverted files), proposes three different database representations, and reports comparative experimental results. Two of these representations are one order of magnitude more space efficient and two orders of magnitude faster in query evaluation, than the plain relational representation.

1 Introduction

Mitos[1] (formerly known as grOOGLE) is a recently developed Web search engine that offers a wide spectrum of functionalities. Synoptically, Mitos is equipped with an advanced stemmer for the Greek language, offers real time result clustering, advanced link analysis techniques (also for spam page detection) and facet-based exploration services [10]. For a detailed description of Mitos see [8].

Although the most widely used data structure for the index of an information retrieval (IR) system is the *inverted file* (else called inverted index), the index of Mitos is based on a DBMS (PostgreSQL). In this paper we discuss the benefits and drawbacks of this choice. Specifically, we introduce three different representations (database schemas) for the index and we report some interesting comparative experimental results. In brief, the support of set-valued attributes by object-relational DBMSs can offer significant storage space savings and query evaluation speedup.

The rest of this paper is organized as follows: Section 2 discusses the benefits and drawbacks of using a DBMS,

---

[1] http://groogle.csd.uoc.gr:8080/mitos/

while Section 3 presents three possible database representations of Mitos index. Section 4 reports experimental results. Finally, Section 5 concludes the paper and identifies issues for further work and research.

2 DBMS versus Inverted Files

Most information retrieval systems and web search engines use inverted files, which have been proven to be very efficient for answering queries [3]. However, the last years the scope of such systems is getting wider. For instance, they should be able to handle structured documents (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: from data mining algorithms and machine learning to collaborative recommendation and filtering.

For these reasons, the index of an engine should be easily extensible and able to accommodate various types of data. In this paper we elaborate on building and managing the index using a DBMS. Below we list some of the advantages and drawbacks of this choice.

2.1 DBMS Advantages

Extensibility of Index Scope
The extension of the index schema with additional columns and relations for widening the spectrum of the functionalities offered is rather straightforward if a DBMS is adopted. For instance, we can extend the index with various information, like users, dates, tags, metadata, and support more sophisticated queries and retrieval models.

Index Construction Process
As a DBMS handles the physical layer we do not have to create and merge *partial indices* for constructing the index of a big corpus (as we have to do if we adopt inverted files).

Index Maintenance
Many pages change or disappear rapidly [6], probably more than 20% in a daily basis. Deleting the entries that

Repr.	document	word	occurrence
PR	[id:int, url:varchar, norm:float, rank:float]	[id:int, name:varchar, df:int]	[word_id:int, doc_id:int, tf:float]
OR	[id:int, url:varchar, norm:float, rank:float]	[id:int, name:varchar, df:int]	[word_id:int, occur:Array⟨Point⟩]
COR	[id:int, url:varchar, norm:float, rank:float]	-	[word_name:varchar, occur:Array⟨Point⟩, df:int]

Table 1. Three different representations of the index

concern a particular document is a very expensive operation in an inverted file. Specifically, its cost is in $\mathcal{O}(n)$ where n the size of the collection in words. With a DBMS such operation can be performed more efficiently (depending on the adopted representation as it will be described later on).

Single Index

Classical IR systems use separate indices: one for answering queries and one for updates. The second is the index of the recently crawled pages and when it is fully constructed (by merging all partial indices) it replaces the first index. With a DBMS this distinction and duplication is not necessary, i.e. we can have a single index (for both updating and querying) as we do not have to create partial indices.

Distributed Query Processing

The advances in DBMS for multicore and clustered systems can transparently benefit IR systems that are built on top. For instance, PostgreSQL can take advantage of more than one available system CPUs/cores (e.g. for dispatching queries), while the ongoing project *pgpool-II*[2] works on supporting more advanced distributed query processing features, such as the dispatching of parts of a query plan to the available CPUs. Although Mitos currently runs on a single machine, we plan to exploit the above functionalities.

2.2   DBMS Drawbacks

Higher Storage Space

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. Occurences 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 [1] 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 $(t, \{d_1, d_3, d_5\})$. In a relational DBMS 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.

More I/O operations

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.

---

[2]http://pgpool.projects.postgresql.org/

This has been experimentally verified in [8], where Mitos was found less efficient than Terrier [7].

However, the adoption of *set-valued* attributes that are offered by object-relational DBMSs (like PostgreSQL) can alleviate these problems as we will describe in detail later on. Specifically we will study the trade-off, between the index size (and query evaluation times) and the ability of the index to support multiple access paths, e.g. by term versus by document. In the future we plan to compare such DBMS indices with inverted files.

2.3   Term versus Document based access

To compute the answer of a query the index should provide efficient *term-based* access (this is what inverted files offer). However there are some other tasks that require *document-based* access and could be faster in a DBMS system. Such queries implemented in Mitos include document deletion (locate all those entries that concern a particular document), query expansion (retrieve the most highly ranked terms that appear in the top-ranked documents) and relevance feedback (retrieve the terms of the documents for which the user provided feedback).

3   On DBMS-based Indices

3.1   The Indexer of Mitos

Mitos adopts the *tf-idf* weighting scheme. Therefore, for each term we have to keep a) its document frequency (df) in the collection and b) its term frequency (tf) for each document. One of the main differences of Mitos compared to other search engines is that it does not store the positions terms appear in documents. Thus, when Mitos returns the query results to the user, it parses the cached file of that document, to find the "best text" with respect to the query terms. However, this is needed only for the documents that lie in the result pages the user will visit. In this manner, Mitos pays only for the relevant documents that the user will visit. On the other hand, without storing term positions, Mitos can't support phrase queries and proximity operators.

3.2   DB Representations for Occurrences

Here we introduce three different database representations for the index (shown in Table 1). All three comprise

a relation *document*, that stores information about documents: for each document it keeps its identifier, its url, the norm of its vector, and its PageRank score. The three representations differ on how they store words and occurrences. Below we describe each one of them.

(PR) Plain-Relational

This is the representation currently in use by Mitos. The relation *word* stores the words, their identifiers and their *df* (document frequencies). The relation *occurrence* stores triples of the form $[word\_id, doc\_id, tf]$. The main drawback of this representation is that each $word\_id$ is stored for each document in which it appears in. This duplication results in high storage space.

(OR) Object-Relational

This representation exploits the set-valued attributes supported by PostgreSQL in order to reduce the space occupied by occurrences. Specifically, it exploits the *point* datatype offered by PostgreSQL for representing the pairs $\langle doc\_id, tf \rangle$. 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 the $word\_id$ serves as a primary key in both *word* and *occurrence* tables, and moves the attributes $word\_name$ and *df* to *occurrence* table.

3.3 Bulk Index Creation/Updates

Initially, it seems that the benefits from using a DBMS are at the expense of the data storage and retrieval efficiency. The guarantee of the ACID properties, the concurrency control, the update of DBMS indices (e.g., B^+Tree etc.) and their possible reorganization on disc due to the insertion of new tuples may harm the efficiency of the index.

In order to reduce the effect of these problems, we use the copy function of PostgreSQL during the indexing 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 and then we insert the new tuples, re-creating the indices at the end. In this manner, we pay time only to compute the final indices, instead of computing temporal ones, that will need to be changed after the next tuple(s) insertion. After all documents have been indexed, for each document d we compute the norm ($\|\vec{d}\|$) of its vector ($\vec{d}$) as defined by the tf-idf weighting scheme, and store it in the norm field, in order to speed-up the evaluation of a query at the searching phase.

Repr.		Queries
PR	q_{word}	SELECT id, df FROM word WHERE name="informat" OR name = "retriev"
	q_{occ}	SELECT word_id, doc_id, tf FROM occurrence WHERE word_id IN (informat_id, retriev_id)
	q_{doc}	SELECT id, norm, rank FROM document WHERE id IN(doc1, doc2, …, docN)
OR	q_{word}	SELECT id, df FROM word WHERE name="informat" OR name = "retriev"
	q_{occ}	SELECT word_id, doc_id, tf FROM occurrence WHERE word_id IN (informat_id, retriev_id)
	q_{doc}	SELECT id, norm, rank FROM document WHERE id IN(doc1, doc2, ..., docN)
COR	q_{occ}	SELECT word_name, occur, df FROM occurrence WHERE word_name="informat" OR name = "retriev"
	q_{doc}	SELECT id, norm, rank FROM document WHERE id IN(doc1, doc2, ..., docN)

Table 2. Queries for each representation

3.4 Query Evaluation

Table 2 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, issued queries are decreased by one, since the *df* values are now stored in the occurrence table instead of the word table used by the other two representations.

3.5 PostgreSQL Indices

In order to provide more efficient access paths to the relations, we need to build appropriate PostgreSQL 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 attribute, namely either B^+Tree or $Hash$ index. Regarding *word* table, the access is done given the $name$. In that case, we can again use a B^+Tree or $Hash$ index. Furthermore, we could exploit the *Trie* index, which has been implemented on top of PostgreSQL, as a part of the SP-GiST index family [2, 4]. According to [5], the *Trie* index offers more than 150% performance increase for exact search matches over to PostgreSQL B+trees, and scales better regarding size. Finally, about the *occurrence* table, once again, possible choices are either a B^+Tree or $Hash$ index, on the $word\_id$ attribute. For the COR though, the *word* and *occurrence* tables have

Repr.	doc.	word	occur.	total	copy time
PR	48.1 MB	12.2 MB	1.85 GB	1.916 GB	35351s
OR	48.1 MB	12.2 MB	93.7 MB	154 MB	681s
COR	48.1 MB	-	96.1 MB	145.9 MB	569s

Table 3. DB size and copy times

Repr.	doc.	word	occur.	time
PR using $Hash$	5.34 MB	16.02 MB	1.38 GB	280s
PR using B^+Tree	2.67 MB	6.26 MB	638.21 MB	562s
OR using $Hash$	5.34 MB	16.02 MB	9.35 MB	5.1s
OR using B^+Tree	2.67 MB	6.26 MB	3.72 MB	5.9s
COR using $Hash$	5.34 MB	−	16.03 MB	1.7s
COR using B^+Tree	2.67 MB	−	6.26 MB	2.3s

Table 4. Indices size and Creation times

been merged. So, since the access is done given the $name$ attribute, we can create either a B^+Tree, $Hash$ or $Trie$ index on it.

4 Experimental Results

The experiments were performed on a desktop PC with a Pentium IV 3.4 GHz processor and 2 GB main memory, on top of Linux distribution Ubuntu v8.04. We used PostgreSQL v8.0.15 and gave to it 100,000 buffers (i.e. 860 MB). Our collection contained documents of various formats (.html, .pdf, .doc, etc) crawled from our university[3] and FORTH[4] domains. Specifically, it comprises $155,661$ documents (mainly in Greek), $216,449$ distinct terms and its total size is 28.5 GB.

4.1 Database Size and Copy Times

To compute the physical database size for each representation we consider that the PostgreSQL storage requirement for string types is 4 Bytes plus the actual string size, while the storage requirement for integers and floats (considering the `int4` and `float4` types respectively) is 4 Bytes. In addition, we should take into account the extra storage cost per tuple due to an internal id of 40 bytes generated by PostgreSQL to identify the physical location of a tuple within its table (block number, tuple index within block).

Regarding *document* table (employed by all representations), the tuple size is $S_d = 4 + (4 + uri.length * 1) + 4 + 4 + 40 = 56 + uri.length$ Bytes. The total size of the table equals $S_d * D$, where D is the number of collection documents. Similarly, each tuple of *term* table (employed by PR and OR), takes $S_t = 4 + (4 + name.length * 1) + 4 + 40 = 52 + name.length$. The total table size is $S_t * T$, where T is the number of collection terms. Regarding *occurrence* table, for PR each tuple takes $S_{o_1} = 4 + 4 + 4 + 40 = 52$ Bytes. Considering that the size of type *point* is 16 bytes (2 * sizeOf(float8))[5], for OR each tuple takes $S_{o_2} = 4 + (df * 16) + 40 = 44 + 16 * df$ Bytes, where df is the document frequency of the term to which a tuple corresponds. Finally, for COR each *occurrence* tuple takes $S_{o_3} = (4 + word\_name.length * 1) + (df * 16) + 4 + 40 = 48 + word\_name.length + 16 * df$ Bytes.

---

[3]http://www.uoc.gr

[4]http://www.forth.gr

[5]PostgreSQL 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

The sizes of the tables for each representation that correspond to our collection can be seen in Table 3. The sizes of the OR and COR are significantly smaller (more than one order of magnitude), since the number of tuples (hence the cost of the 40 Bytes for each tuple) for the *occurrence* table is the same as the number of terms in the vocabulary. Thus the times to copy the tables are significantly smaller for the OR and COR, in comparison to PR, offering a much more scalable solution, as far as indexing time and size are concerned. In addition, to reduce the I/O overhead during query evaluation for PR we clustered the occurrence table on $word\_id$ (clustering time is not included in Table 3).

4.2 Indices Size and Creation Times

The sizes of the PostgreSQL indices for each representation are shown in Table 4. Unfortunately, we could not evaluate the *Trie* index, as it only accepts words of latin characters and our UTF-8 encoded test collection mainly contained greek documents. B^+Tree indices are much more space efficient than $Hash$ indices (half size for occurrence index in PR). Moreover, the creation times for OR and COR indices are significantly lower than that of PR, leading also to smaller index sizes due to the small size of the occurrence table.

4.3 Query Evaluation Times

To measure query evaluation times, we adopted the following scenario: for each of the three representations and for each PostgreSQL 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. The terms contained in the above queries were different (for each of the 1, 2, 3 or 4-sized queries) and they were selected based on their df. Specifically, we selected frequently occuring terms with a df value about 45,000. The big number of documents that these terms appear in, implies big overhead to the DBMS. Due to the large number of doc_ids passed in the IN list of the q_{doc} queries, we encountered a PostgreSQL crash. We tackled this problem by dividing the IN list in blocks of 45,000 doc_ids and submitting one query for each block of the list. Subsequently we summed the times required by these queries. We gathered the aforementioned times through the

Repr.	1 term			2 terms			3 terms			4 terms		
	q_{word}	q_{occ}	q_{doc}	q_{word}	q_{occ}	q_{doc}	q_{word}	q_{occ}	q_{doc}	q_{word}	q_{occ}	q_{doc}
PR using $Hash$	0.212	30.477	2.250	0.2127	60.034	4.450	0.233	81.206	6.705	0.174	112.107	9.898
PR using B^+Tree	0.196	35.338	2.118	0.219	63.460	4.556	0.228	88.889	6.802	0.164	123.049	9.894
OR using $Hash$	0.209	0.154	1.617	0.232	0.297	3.259	0.013	0.483	4.752	0.223	0.567	6.194
OR using B^+Tree	0.194	0.152	1.655	0.246	0.295	3.214	0.013	0.488	4.680	0.202	0.586	6.142
COR using $Hash$	–	0.183	1.698	–	0.341	3.277	–	0.536	4.769	–	0.613	6.175
COR using B^+Tree	–	0.168	1.641	–	0.310	3.190	–	0.513	4.706	–	0.597	6.112

Table 5. Query evaluation times (sec)

Aggregator[6] toolkit which is written in Java. This means that the measured times include the overhead of the JDBC driver (version 8.0-322 JDBC 3), an overhead that also exists in the Mitos engine, since it is written in Java.

As one can observe from the times reported in Table 5, OR and COR representations are orders of magnitude more efficient than PR, mainly due to the efficiency in occurrence table. More precisely, OR and COR are approximately 200 times faster than PR for all queries, although OR and COR index is only an order of magnitude (see Table 3) smaller than PR index. This is due to the fact that OR and COR indices, fit in main memory, so every page that is fetched in memory, is constantly kept there. Comparing OR and COR, we observe that COR is slightly faster than OR. A common behaviour for all three representations is the slow q_{doc} query times. This query is actually the bottleneck for both OR and COR representations. This is due to the long IN list as described earlier. In the future, we plan to upgrade to PostgreSQL 8.3 which offers an optimized IN operator, and to investigate whether we can reduce the overhead of such queries by using temporary tables.

Regarding the DBMS indices, we can conclude that $Hash$ indices are the best choice for PR while B^+Tree indices for COR. For OR both of them have almost equivalent performance. For a bigger collection though, B^+Tree indices could be a better choice also for PR and OR, due to their smaller index size.

5 Conclusions

In this paper we elaborated on using object-relational DBMSs for managing a Web search engine index. Specifically we proposed and evaluated three different representations. COR was found to be the most efficient, being one order of magnitude less space costly and two orders of magnitude faster in query evaluation compared to PR. In future work, we plan to compare the efficiency of B^+Tree index with a $tree - Trie$ index [9] that has been proposed to index relationships with set-value attributes. Moreover, we need to evaluate the cost of document-based access in OR and COR againtst PR. Finally we plan to compare the DBMS-approach with the classical inverted file on the same collection.

---

[6]http://www.csd.uoc.gr/∼andreou

References

[1] V. N. Anh and A. Moffat. "Inverted Index Compression Using Word-Aligned Binary Codes". *Information Retrieval*, 8(1):151–166, 2005.

[2] W. G. Aref and I. F. Ilyas. "SP-GiST: An Extensible Database Index for Supporting Space Partitioning Trees". *J. Intell. Inf. Syst.*, 17(2-3):215–240, 2001.

[3] R. Baeza-Yates and B. Ribeiro-Neto. *"Modern Information Retrieval"*. Addison Wesley, May 1999.

[4] 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.

[5] M. Y. Eltabakh, R. Eltarras, and W. G. Aref. "Space-Partitioning Trees in PostgreSQL: Realization and Performance". In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 100, Washington, DC, USA, 2006. IEEE Computer Society.

[6] T. L. Harrison and M. L. Nelson. "Just-In-Time Recovery of Missing Web Pages". In *HYPERTEXT '06: Proceedings of the seventeenth conference on Hypertext and hypermedia*, pages 145–156, New York, NY, USA, 2006. ACM.

[7] I. Ounis, C. Lioma, C. Macdonald, and V. Plachouras. "Research Directions in Terrier". *Novatica/UPGRADE Special Issue on Web Information Access, Ricardo Baeza-Yates et al. (Eds), Invited Paper*, 2007.

[8] P. Papadakos, G. Vasiliadis, Y. Theoharis, N. Armenatzoglou, S. Kopidaki, Y. Marketakis, M. Daskalakis, K. Karamaroudis, G. Linardakis, G. Makrydakis, V. Papathanasiou, L. Sardis, P. Tsialiamanis, G. Troullinou, K. Vandikas, D. Velegrakis, and Y. Tzitzikas. "The Anatomy of Mitos Web Search Engine". *CoRR, Information Retrieval*, abs/0803.2220, 2008. Available at http://arxiv.org/abs/0803.2220.

[9] M. Terrovitis, S. Passas, P. Vassiliadis, and T. Sellis. "A Combination of Trie-trees and Inverted Files for the Indexing of Set-valued Attributes". In *CIKM '06: Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 728–737, New York, NY, USA, 2006. ACM Press.

[10] Y. Tzitzikas, N. Armenatzoglou, and P. Papadakos. "FleXplorer: A Framework for Providing Faceted and Dynamic Taxonomy-based Information Exploration". In *Procs of FIND'2008 (at DEXA '08)*, Turin, Italy, Sept. 2008 (to appear).