Building an Inverted Index at the DBMS Layer for Fast Full Text Search

Ciprian-Octavian Truică * **Florin Rădulescu** *
Alexandru Boicea *

* *Computer Science and Engineering Department, Faculty of Automatic Control and Computers, University Politehnica of Bucharest, Bucharest, Romania (e-mail: ciprian.truica@cs.pub.ro, alexandru.boicea@cs.pub.ro, florin.radulescu@cs.pub.ro)*

Abstract: In order to make accurate and fast keywords and full text searches it is recommended to index the words in the corpus. One way to do this is to use an inverted index to maintain in a structured form the words occurrence in a set of documents. A stemming algorithm can be used to minimize the number of indexed words, so only the root word is kept for each term. This paper presents how to build an inverted index for documents stored in MongoDB and Oracle databases. Different approaches are presented in order to compare and determine which one has the best performance. These approaches take advantage of the frameworks and tools provided by the database systems to build the index: the MapReduce framework for MongoDB and Pipelined Table Functions for Oracle.

Keywords: Inverted index, stemming, MapReduce, Oracle, MongoDB, pipelined table functions

1. INTRODUCTION

An *inverted index* is a data structure that stores the location of words in different files, documents or database records. Its main purpose is to allow fast full text search, being a core component of a typical search engine. Adding an *inverted index* to a set of documents lets the user search for keywords in those documents without processing the text each time.

There are different methods that can be used to build an *inverted index*. The simplest is to process each document, adding new words and documents to the index. This method can be optimized by using multithreading algorithms when processing the documents. Another approach is to process the text and build the index directly inside the database that stores the documents. Using this method, the communication latency between the search engine and the database is minimized. It is more efficient to process or pre-process as much as possible of the data at the database management system (DBMS) layer then to send unprocessed data to a superior layer - in this case the search engine - to be processed. Because the data is already inside the database, the communication latency between different engines inside the DBMS is negligible and by definition a DBMS is designed to process large amounts of data very fast. Also, by implementing an *inverted index* at the database layer, all the data is processed inside the DBMS and the result is stored locally instead of sending the document to an upper layer to be processed and, after that, sending back the result to the database. This approach is well suited in distributed systems scaled between different data centers.

The classic *inverted index* adds to the data structure all the words. A better approach for minimizing the data stored in the index is to use a *stemming algorithm* to extract the *stem* (the base root of a word). *Porter Stemmer* is the best candidate because it is a self-contained algorithm that does not need an external corpus, n-grams or a model trained using machine learning. One can argue that a stemmer is less accurate than other algorithms that are used to extract the base root of the word, because it cannot make the distinction between words which have a different meaning depending on the part of speech. However, a stemming algorithm runs faster.

During the construction of the *inverted index* the number of appearances for each word of a document is also kept. The number of occurrences can be used to classify the documents using an unsupervised classification algorithm, for text mining(Ilic et al. 2014) or for an Information Retrieval ranking function.

The research problems addressed in this paper are:

i) Constructing an *inverted index* at the DBMS level for full text search.
ii) Improving performance using the *MapReduce* paradigm in a single instance and in a distributed environment.
iii) Comparing the results of the proposed methods to determine if the construction of the *inverted index* at the DBMS level is better than constructing the index at the search engine layer.
iv) Reducing the storage space of the *inverted index* structure by using a *stemming algorithm*.
v) Improving text processing and index building performance by using *MapReduce*.
vi) Comparing full text search performance with and without using the proposed index or database build-in indexes.

This paper is structured as follows. Section 2 presents previous research related to construction of full text search and space efficient inverted indexes. Section 3 presents the *inverted index* design and how it is built using different approaches. Section 4 contains the experimental validation of the proposed methods and analyzes the results. The last section concludes with a summary and hints at future research.

2. RELATED WORK

The *inverted index* is a data structure used by search engines. Its main purpose is to optimize the speed of query response. By using an *inverted index*, the search is made in this data structure - where the searched words are stored using IDs and not directly in the document.

Since the first appearance in the literature of this concept, different techniques and methods were developed to construct an *inverted index*. Qiuying Bai et al. propose a new methodology for creating and updating such an index. The performance improvements of this method - compared to the classic approaches - are proved by experiments (Bai et al. 2012). Dean et al. propose a method for improving the construction time of an *inverted index* using *MapReduce* (Dean et al. 2010).

*Inverted index*es are also used in Information Retrieval. A great emphasis is put on reducing the index storage size. One method uses the static index pruning approach which is useful to reduce the index size. S.K. Vishwakarma et al. propose that the pruning should be done on the basis of the *term frequency-inverse document frequency (TFIDF)* score of each individual document (Vishwakarma et al. 2014).

Another research direction focuses on improving the query response time and minimizing the storage space. Xiaozhu Liu proposes a random access blocked *inverted index* and an efficient dynamic maintenance scheme that are used to improve the retrieval performance and minimize storage space for large scale full text search systems (Liu 2010). Liu et al. propose a new effective keyword search in relational databases that improves query performance based on a simple *inverted index*(Liu et al. 2006).

A. Babenko et al. proposes a new structure, called an inverted multi-index, which improves the search of similarity in very large databases. In their experimental setup, this new structure achieves a much denser subdivision of the search space while retaining their memory efficiency (Babenko et al. 2012). Hao Wu et al. focus their research on keyword search, proposing in their work a generalized *inverted index* for keyword search (Ginix) that merges consecutive IDs in inverted lists (Wu et al. 2013). This method reduces the storage space, disk I/O time and also CPU processing time.

To further improve the access to the *inverted index* and achieve fast response times to user's queries, a compression can be applied. *inverted index* compression has been common for some time. One common practice while storing a posting list is to use δ-gaps where possible, e.g. to record the differences between monotonically increasing components (such as IDs or positions) instead of their actual values (Jin et al. 2008; Sun et al. 2014; Witten et al. 1999). By not using binary word-aligned notations, smaller numbers lead to smaller representations (Catena et al. 2014). Glory et. al propose a new integer compression technique, Extended Golomb Code, to reduce the size of the inverted index (Glory et al. 2013).

The *Porter Stemmer* is an algorithm for suffix stripping. It effectively works by considering complex suffixes as compounds made up of ones, and removes the them step by step. At each step the removal of a suffix is made to depend upon the form of the remaining stem, which usually involves a measure of its syllable length (Porter 1980). By using a stemming algorithm, the space allocated to store an *inverted index* is minimized. A lemmatization algorithm can be used instead of a stemmer, but the information gain, although it exists, is not that great (Kettunen et al. 2005).

3. MODEL AND IMPLEMENTATION

3.1 Index Model

The *inverted index* is constructed as a dictionary - a collection of keys and values (Figure 1). The key of an element is the *stem* and the value is a list. An element in the list has two components: a *docID* and *count*. The *docID* stores the document's unique identifier (ID). The count keeps the number of occurrences of the *stem* in that particular document. Only documents where the stem appears are included in the list. Stop words are not taken into account.

```
{
    "term": "mansion",
    "docIDs" : [
        { "docID": 3,   "count" : 3 },
        { "docID": 17,  "count" : 2 },
        { "docID": 59,  "count" : 6 },
        { "docID": 175, "count" : 5 },
        { "docID": 391, "count" : 8 },
        ...
        { "docID": 965, "count" : 1 },
        { "docID": 993, "count" : 3 }
    ]
}
```

Fig. 1. Inverted Index element

3.2 Implementation

The *inverted index* is created using four different approaches. Queries with constraints are issued at the search engine layer to retrieve text documents from the database. The databases used are *MongoDB* [1] and *Oracle* [2]. *Oracle* Database is the most used Relational Database Management System (RDBMS) to date as seen in the ranking done by *DB-Engines Ranking* [3]. *MongoDB* uses SQL-like query syntax and, according to the same ranking, it is the most used Document Oriented Database (DODB) of its kind. Each approach is implemented by an application. These applications are: i) a single thread application using MongoDB, ii) a *MapReduce* application using multithreading, iii) an application using the *MapReduce* framework feature

[1] MongoDB Documnetation http://docs.mongodb.org/
[2] Oracle Documentation http://docs.oracle.com/en/
[3] DB-Engines Ranking http://db-engines.com/en/ranking

of *MongoDB*, and iv) an Oracle application using *Pipelined Table Functions* to simulate *MapReduce*.

Three different programming languages are used to implement these applications: two applications are in Python using the *MongoDB* database; one implementation is in JavaScript and is stored in *MongoDB*; the last one is in PL/SQL and is stored as an *Oracle* package.

Documents are stored in *MongoDB* as a collection. Each element of the collection contains the document ID and a field with the document. In *Oracle* the structure is implemented as a table with two columns. The first column is the primary key and uniquely identifies the documents. The second column is a text field containing the document. All four implementations execute queries in the database for getting the document, processing the text and constructing the inverted index for the *stems* within it.

All the implementations use *Porter Stemmer*, an algorithm that removes suffixes from words in English, resulting a new terms called *stems*. Search results are improved because queries use *stems* instead of words (Bird et al. 2009). For the single thread application and the *MapReduce* application written in Python the NLTK [4] implementation of this algorithm was used. NLTK's stemmers handle a wide range of irregular cases. All the stop words are also removed when parsing the text. This is the text processing step.

When using *MapReduce*, the *Map* function provides the *Reduce* function with a *stem*, a document ID and a counter. In the reducer, if the *stem* is found for the first time in the document then a counter - which is the number of occurrences of the *stem* in the document - is initialized.

Algorithm 1 Append Stem Function

Input: a stem (*stem*), document ID (*docId*), dictionary with existing stems and IDs (*docDictionary*)
Output: updates the *docDictionary*

```
1: if stem in dictionary then
2:   documents = docDictionary.get(stem);
3:   if docId in documents then
4:     docDictionary.get(stem)[docId] += 1;
5:   else
6:     create a new dictionary newDictionary
7:     newDictionary[docId] = 1
8:     docDictionary.get(stem).update(newDictionary)
9:   end if
10: else
11:   create a new dictionary newDictionary
12:   newDictionary[docId] = 1
13:   docDictionary.get(stem).add(newDictionary)
14: end if
```

3.3 Single Thread Application

The single thread implementation constructs the *inverted index* using a procedural approach. After querying the database, each text is striped of punctuation and special characters using regular expressions. The stemming algorithm is then applied on the cleaned text. Using the *Append Stem Function* (Algorithm 1) the indexing structure is constructed as a dictionary. When the dictionary does not contain the stem, a new record is added where the key is the stem and the value is the list. The list stores a dictionary structure containing the unique document ID and the *stem*'s first appearance in the document. If the *stem* exists in the dictionary then the document's ID is searched through the list. If the ID is found in the list, the number of co-occurrences is incremented. Otherwise, a new pair is added with the number of co-occurrences initialized with 1. After all the documents pass through this process, the dictionary containing the inverted index is stored in the *MongoDB* database as a collection.

[4] NLTK Documentation http://www.nltk.org

3.4 Multithreading MapReduce Application

The second application for constructing an *inverted index* uses the MapReduce programming paradigm. This algorithm is implemented using a multithreading approach to simulate the workers. Each document is sent to the *Map* function (Algorithm 2) for processing the text and stripping it of punctuation using regular expressions. Then each word is stemmed and the *Map* function emits the *stem* with the document ID and the number of appearances of the *stem* in text, initialized with 1.

Algorithm 2 Map Function

Input: a dictionary of documents (*documents*) with each element containing a document ID (*docId*) and the documents content (*docContent*)
Output: emits to the worker a stem and a dictionary with document ID and a counter for the stem

```
1: for each docId, docContent in documents do
2:   docContent.words =
           tokenize(docContent.strip(punctuation))
3:   for word in docContent.words do
4:     stem = PorterStemmer.getStem(word)
5:     emit(stem, docId: 1)
6:   end for
7: end for
```

The *Reduce Function* (Algorithm 3) receives the *stem*, the document ID and the counter, and starts to construct the *inverted index* as a dictionary.

Algorithm 3 Reduce Function

Input: a stem (*stem*), and a dictionary (*oldDictionary*) containing document IDs with stem counts
Output: the *stem* and a new dictionary *newDictionary* with an updated stem *counter* for each document

```
1: create a new dictionary newDictionary
2: for each docId, counter in oldDictionary do
3:   if docId in newDictioanry then
4:     newDictionary[docId] += counter
5:   else
6:     newDictionary[docId] = counter
7:   end if
8: end for
9: return (stem, newDictioanry)
```

If the *stem* does not exist in the dictionary then it is added as key, and the pair (document ID, *stem* number of appearances in text) as value. If the *stem* is already in the dictionary then the *Reduce* function checks if the

document's ID appears among the existing pair values. If it is found then the value for the number of appearances in text is incremented, else a new pair value is added to the dictionary. The function returns the *stem* with a dictionary of document ID and *stem* counts.

This algorithm uses a multiprocessing resource pool to construct the *inverted index*. When a *MapReduce* worker finishes the job of adding a new value to the dictionary, then it receives a new value to process. By default the number of workers is equal to the number of CPUs available on the host machine.

3.5 MongoDB Application

The *MongoDB* application uses the *MapReduce* framework available with the database distribution. In this case, the *Map* and *Reduce* functions are implemented in JavaScript and are stored at the database layer in *MongoDB* together with the *Porter Stemmer* algorithm. The algorithm for constructing the *inverted index* is similar to the one used in the Python *MapReduce* application, but it takes advantage of the *MapReduce* framework implemented at the DBMS layer. The text is processed and striped of punctuation, then each word is stemmed and the *Map* function emits a structure containing the *stem*, the document ID and the number of appearances for the word in the text.

The *Reduce* function receives two parameters: the first one is the word and the second one is a dictionary containing the document ID as key and the number of occurrences for the word in text as value. The function follows the same steps as described for the *Reduce* function in the Python implementation of *MapReduce*.

3.6 Oracle Application

The *Oracle* database has three tables: *Documents*, *Stems*, and *InvertedIndex* (Figure 2). The *Documents* table stores the document's text and an ID. The *Stem* table stores the *stems* with a corresponding ID. This table is updated by the *PorterStemmer* tokenization package. The *InvertedIndex* table stores the document ID, the steam ID and the number of appearances of the *stem* in the document.

Fig. 2. Normalized database

The *Oracle* application uses *Pipelined Table Functions* to construct the *inverted index* (Moore 2014). At a logical level, a *Table Function* is a function that can appear in the FROM clause and therefore acts as a table, returning a stream of rows. The *Table Functions* can also take a stream of rows as input. Since *Pipelined Table Functions* are embedded in the data flow, they allow data to be 'streamed' to an SQL statement avoiding intermediate materialization in most cases. Additionally, *Pipelined Table Functions* can be parallelized. Using this feature, the *MapReduce* algorithm is simulated. The *Map* function uses a cursor to loop through the *Documents* table and extract the words. Each word is then processed by the *PorterStemmer* tokenization package (Algorithm 4 lines 20 and 25). This package extracts the word's *stem*, inserts it, if it does not exist, in the *Stems* table and returns, through the *getStemID* function, the *stem* ID. A new record (*stemRecord*) containing the document ID and the *stem* ID is created after the use of the *PorterStemmer.getStemID* function. This record - whom is a new row in a *Pipelined Table Function* - is added to a pipe.

Algorithm 4 PL/SQL Map Function

Input: a cursor that iterates over the documents, a list of separator characters
Output: a *stemRecord* (record with docID and stemID) is added to the pipe

```
1: function mapper(documentsCursor in sys_refcursor,
          separator in varchar2)
2:   return stemsTable
3:   pipelined parallel_enable (partition
          documentsCursor by any)
4:   is
5:       doc documents%rowtype;
6:       sp number; // Start Position
7:       cp number; // Current Position
8:       docLen number; // Document length
9:       sRec stemRecord; // Stem Record
10:  begin
11:   loop
12:     fetch documentsCursor into doc;
13:     exit when documentsCursor%notfound;
14:     sp := 1;
15:     docLen := length(doc.content);
16:     while sp <= docLen
17:       loop
18:         cp := instr(doc.content, sep, sp);
19:         if cp = 0 then
20:           sRec.stemID :=
                    PorterStemmer.getStemID(
                    substr(doc.content, sp));
21:           sRec.docID := doc.docID;
22:           pipe row (sRec);
23:           sp := docLen + 1;
24:         else
25:           sRec.stemID :=
                    PorterStemmer.getStemID(
                    substr(doc.content, sp, cp - sp));
26:           sRec.docID := doc.docID;
27:           pipe row (sRec);
28:           sp := cp + 1;
29:         end if;
30:       end loop;
31:  end loop;
32:  return;
33: end mapper;
```

The *Reduce* function (Algorithm 5) computes the number of appearances of the words in text. This function then emits the record which contains the *stem*, the document ID and the counter. The *Reduce* function is an aggregation Table Function. It receives the *stem* as key and a set of document IDs and counters as value. For each *stem*, the set is iterated and it sums the counters for each document.

The *Map* function uses a *pipe row* to insert the *stem*, the document unique identifier and the counter as values for

a new row in the *Pipelined Table Functions*, instead of using an *Emit* . The *Reduce* function, an aggregation *Table Function*, simply extract the information from the *pipe* and sums all the counters for a given *stem* and document ID. At the end of this process the *InvertedIndex* and *Stems* tables are populated with all the correct information.

Algorithm 5 PL/SQL Reduce Funtion

Input: a *stemRecord* - record with docID and stemID
Output: recalculates the number of appearances for each stem

```
1: function reducer(inStemsCursor in stemsCursor)
2:   return stemssCountTable
3:   pipelined parallel_enable (partition inStemsCursor
                        by hash(stemID))
4:   cluster inStemsCursor by (stemID)
5:   is
6:         stemCR stemsCountRecord; // Record for storing
               the number of co-occurrences for a stem
7:         stemN stemRecord; // Next Stem
8:   begin
9:     stemCR.count := 0;
10:    loop
11:      fetch inStemsCursor into nw;
12:      exit when inStemsCursor%notfound;
13:      if stemCR.stemID is null then
14:        stemCR.stemID := stemN.stemID;
15:        stemCR.docID := stemN.docID;
16:        stemCR.count := stemCR.count + 1;
17:      else
18:        if stemN.stemID <> stemCR.stemID then
19:          pipe row (stemCR);
20:          stemCR.stemID := stemN.stemID;
21:          stemCR.docID := stemN.docID;
22:          stemCR.count := 1;
23:        else
24:          if stemCR.docID = stemN.docID then
25:            stemCR.count := stemCR.count + 1;
26:          else
27:            pipe row(stemCR);
28:            stemCR.docID := stemN.docID;
29:            stemCR.count := 1;
30:          end if;
31:        end if;
32:      end if;
33:    end loop;
34:    if stemCR.count <> 0 then
35:      pipe row (stemCR);
36:    end if;
37:    return;
38:  end reducer;
```

4. PERFORMANCE TESTING

The performances of the applications are tested by measuring the response time for constructing the *inverted index* for five different sets of input data. The tests are done for 10, 50, 100, 500 and 1 000 documents. A document has approximately 4 000 words. The performance tests for the multi-thread applications use 4 threads. All the tests are done on virtual machines. These machines are created using the VMware virtualization software and have the following hardware specifications: 4GB of RAM, a 40GB HDD and one processor with 2 cores with a frequency of 2.4GHz each. The operating system for the machine with *MongoDB* is Ubuntu 12.04LSTx64 and the operating system for the *Oracle* database is Windows 7 Professional x64. *MongoDB 2.6.3* and *Oracle 11gR2* are used.

Algorithm 6 JavaScript function for calculation the inverted index construction time

Input: the number of tests
Output: inverted index construction time

```
1: function creationInvertedIndex(n){
2:   for (var i = 0; i < n; i++) {
3:     db.inverted_index.remove();
4:     var start = new Date();
5:     db.runCommand({
            "mapreduce" : "documents",
            "map" : map,
            "reduce" : reduce,
            "out" : "inverted_index"}
       );
6:     var end = new Date();
7:     print(end-start);
8:   };
9: };
```

Algorithm 7 PL/SQL block for calculation the inverted index construction time

Input: the number of tests
Output: inverted index construction time

```
1: declare
2:   timeStart number;
3:   timeSnd number;
4:   sep varchar2(20) := '[^0-9a-zA-Z+#]+';
5: is
6: begin
7:   for i in 1..&n
8:     loop
9:       execute immediate 'truncate table
                  InvertedIndex';
10:      timeStart := dbms_utility.get_time();
11:      insert into InvertedIndex
           select docID, stemID, sum(count)
           from table(MR.reducer(
           cursor(select value(mr).stemID stemID,
           value(mr).docID docID from table(
           MR.mapper(cursor(select docID,
           lower(regexp_replace(content, sep , ' '))
           from documents), ' ')) mr)))
         group by stemID, docID
         order by 1,2;
12:      timeEnd := dbms_utility.get_time();
13:      dbms_output.put_line(timeEnd-timeStart);
14:    end loop;
15: end
```

The application that uses the *MongoDB MapReduce* framework uses a JavaScript function for testing (Algorithm 6). The Python applications use a bash script to test the creation of the inverted index. The Python *MapReduce* application uses a number of 4 threads for both databases.

Finally, the Oracle implementation uses a PL/SQL anonymous block, presented in Algorithm 7.

The algorithm that builds the *inverted index* using *MapReduce* and *Pipelined Table Functions* in *Oracle* is the slowest as shown in Table 1 and for that reason it is not a viable solution, although it improves the allocated space for the index, due to the index structure. For *Oracle*, it is recommended to use the multi thread solution because it runs faster and improves the creation time considerable.

MongoDB can safely and quickly handle very large datasets using horizontal sharding by value ranges (Redmond et al. 2012). The last tests are designed to see if the construction of the *inverted index* can be improved in a distributed environment. To achieve this, a three node configuration in a shared everything architecture is used (Figure 3).

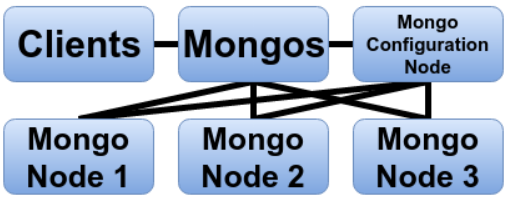

Fig. 3. MongoDB distributed architecture

The configuration in Figure 3 uses three *MongoDB* nodes for storing information. The information is divided between the nodes based on a shard key. The *Mongo Configuration Node* is the server that keeps track of the keys. The *Mongos* server is the single point of entry for the clients. This server uses the *Mongo Configuration Node* to get the sharding keys so it can redirect the clients to the correct server where the data is stored. The implementation uses a shared everything architecture because the nodes share the same memory address space for read/write access between them. The following commands are used to configure the distributed architecture.

```
$mkdir MongoNode1 MongoNode2 MongoNode3 MongoConfig
$mongod –shardsvr –dbpath MongoNode1 –port 27101
$mongod –shardsvr –dbpath MongoNode2 –port 27102
$mongod –shardsvr –dbpath MongoNode3 –port 27103
$mongod –configsvr –dbpath MongoConfig –port 27110
$mongos –configdb localhost:27110 –port 27120
$mongo localhost:27120/admin
mongos>db.runCommand({addshard:"localhost:27101" })
mongos>db.runCommand({addshard:"localhost:27102"})
mongos>db.runCommand({addshard:"localhost:27103"})
mongos>db.runCommand({enablesharding:"DocumentsDB"})
```

The best time performance for the *inverted index* is achieved by the application that uses *MongoDB MapReduce* framework in a distributed environment (Figure 4). *Oracle* implementation of *MapReduce* using *Pipelined Table Functions* has the worst time performance. The mean execution time for the Python implementation of *MapReduce* is very close to the time measured for the *MongoDB* implementation. When the volume of documents increases, the mean execution time grows exponentially. The mean execution time is shown in Table 1.

An index provided by the database can be used to index a text e.g. Context index in *Oracle* (Shea 2014) or Text Index in MongoDB. This could be used only when needed by a word search, but when it is needed to search by document this approach is not recommended because the index is not flexible. In Text Mining the words are used as values for attributes, and using an index provided by the database will not fulfill this requirement.

Table 1. Inverted Index creation time in seconds

	No. Documents				
Algorithm	**10**	**50**	**100**	**500**	**1000**
Single Thread	3.15 ±0.02	13.60 ±0.13	26.89 ±0.49	131.42 ±0.89	257.62 ±0.68
Multithread MongoDB	2.88 ±0.01	12.27 ±0.03	23.50 ±0.13	120.14 ±0.43	246.86 ±0.83
Multithread Oracle	3.15 ±0.03	9.78 ±0.04	18.33 ±0.12	90.90 ±1.17	230.39 ±3.65
MongoDB MapReduce	1.48 ±0.02	7.80 ±0.04	15.96 ±0.09	81.26 ±0.27	160.92 ±0.26
Oracle MapReduce	39.08 ±0.35	189.38 ±0.99	381.45 ±0.71	1 924.08 ±2.08	3 693.19 ±3.89
MongoDB Distributed	1.53 ±0.02	1.94 ±0.01	7.92 ±0.08	21.17 ±0.19	80.98 ±0.33

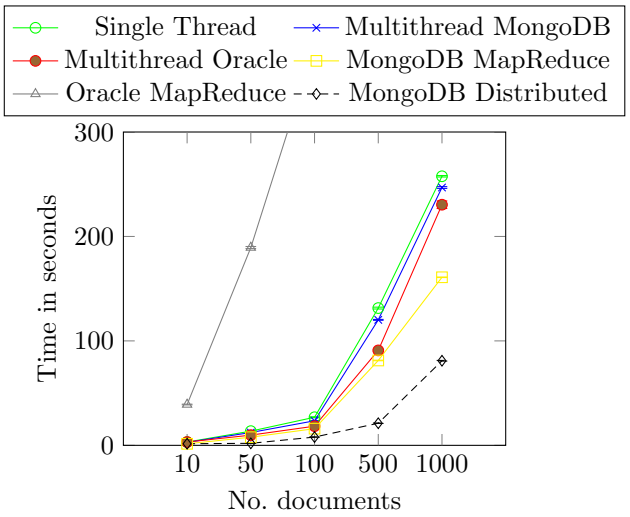

Fig. 4. Inverted Index creation time in seconds

Tables 2 and 3 present the query performance time when using no index, a database build-in index and the *inverted index* presented in this paper. The 1 000 documents corpus was used for this set of tests. The results show that using an inverted index instead of a build-in index improves query performance for both databases (Figure 5 and Figure 6).

Table 2. Oracle query performance in milliseconds

No. Words	TST	TSTWCI	TSTWIV
1 word	1 340.00±14.04	1 247.50±36.88	841.00±32.13
2 words	1 469.00±26.01	1 334.75± 6.75	948.00±10.33
3 words	1 612.00±45.66	1 455.00±33.27	1 040.00±27.08
4 words	1 769.00±60.08	1 550.00±17.92	1 159.00±23.31
5 words	1 985.00±34.08	1 649.75±27.51	1 252.00±29.36

TST - Text Search Time without inverted index;
TSTWCI - Text Search Time with Context Index;
TSTWIV - Text Search Time with inverted index

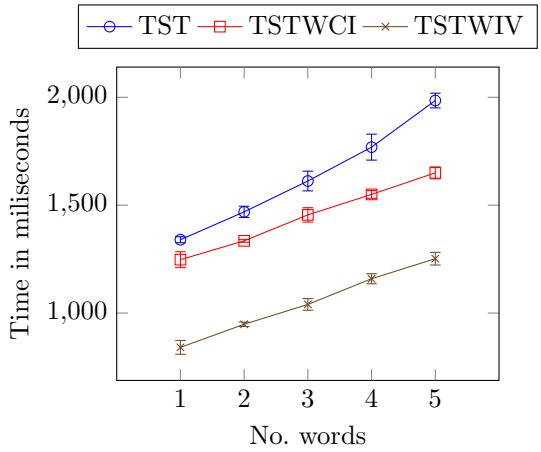

Fig. 5. Oracle query performance in milliseconds

Table 3. MongoDB query performance in milliseconds

No. Words	TST	TSTWTI	TSTWIV
1 word	213.5±0.85	3.2±0.42	4.9±0.32
2 words	376.9±0.99	42.2±0.42	7.0±0.00
3 words	551.4±0.85	77.5±0.53	9.0±0.00
4 words	1 621.4±2.41	92.3±0.48	11.3±0.48
5 words	1 824.5±4.40	105.0±0.94	14.1±0.57

TST - Text Search Time without inverted index;
TSTWTI - Text Search Time with Text Index;
TSTWIV - Text Search Time with inverted index

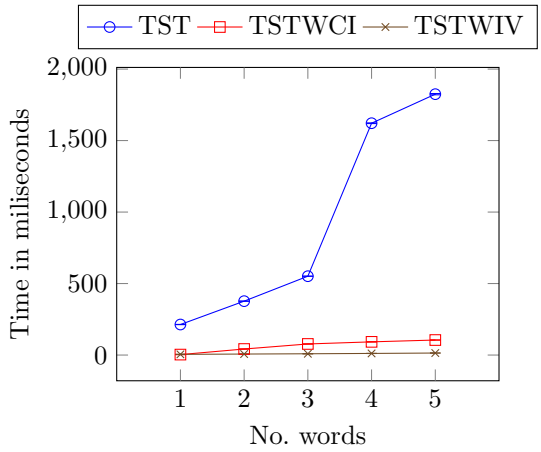

Fig. 6. MongoDB query performance in milliseconds

Algorithm 8 presents the JavaScript function used for MongoDB TSTWIV tests (Table 3). This function receives as a parameter an array of words and consists of two steps. The first step extract the stem for each words and creates a stem array. The second step uses a MongoDB Pipeline Aggregation function which unwinds the stem array, finds the documents that contain each individual stem and aggregates the response so that only the documents that contain all the stems are retrieved.

Tests are highly dependent on the hardware architecture. Performance is influenced by the size of the internal memory: a higher amount of RAM directly impacts the number of records that the DBMS can store here. Also the CPU plays a major role because the speed for processing queries is influenced by the processor frequency and by the number of cores (Truică et al. 2014).

Algorithm 8 MongoDB TSTWIV tests JavaScript function

Input: the number of tests, a list of words
Output: documents that contain the list of words

```
1:  function findDocuments(n, wordsArray){
2:    var stemsArray = new Array();
3:    for (var word in wordsArray) {
4:      var stem = PorterStemmer.getStem(word);
5:      stemArray.push(stem);
6:    };
7:    for (var i = 0; i < n; i++) {
8:      var start = new Date();
9:      var elems = db.inverted_index.aggregate(
                { $match: { "stem": { $in: stemsArray } } },
                { $unwind: "docIDS"},
                {
                    $group: {
                        _id: 'docIDs.docID',
                        stems: { $addToSet: "stem" }
                    }
                },
                { $match: { stem: { $all: stemArray } } },
                { $project: {_id: 1}}
          );
10:     var end = new Date();
11:     print(end-start);
12:   };
13: };
```

5. CONCLUSIONS

MongoDB is a flexible, schema-less database which allows documents to be nested inside other documents (Truică et al. 2013). These features have an impact on the way data is stored; in *MongoDB* the inverted index is stored as one document. The best performance for *MongoDB* is achieved when using the *MapReduce* framework inside the DBMS in a distributed environment (Table 1).

Oracle Table Functions are a robust scalable way to implement *MapReduce* within the *Oracle* database and leverage the scalability of the *Oracle* Parallel Execution framework. Using this in combination with SQL provides an efficient and simple mechanism for database developers to use the *MapReduce* functionality within the environment they understand and with the languages they know (Shankar et al. 2009).

The use of the *Porter Stemmer* algorithm and the removal of the stop words considerably minimize the dimensions of the index. The stop words are dropped because their relevance is minimal when doing a full text search. This *Porter Stemmer* algorithm has its flaws - the *stems* are not always accurate. A better approach would be to use a lemmatization algorithm. The reason why such an algorithm was not used is because it is hard to integrate in a database. For applications that do not process any text in the database, a lemmatization algorithm is recommended. If lemmas are used instead of stems, the index is more

accurate during full text search and the query results have more meaningful content.

Inverted indexes can be used in semantic search. The *Porter Stemmer* algorithm can be applied on the search phrase resulting a list of *stems*. This list can then be used to search inside the *inverted index* to find the relevant documents that contain the searched words, instead of searching for the entire phrase inside the documents.

The number of appearances of each word in a document was also kept when the *inverted index* is built. The number of occurrences can be used to classify the documents using an unsupervised classification algorithm or for Text Mining. Moreover, this information can be used to compute measures used in Information Retrieval, e.g. *term frequency (TF)*, *inverted document frequency (IDF)*, *TFIDF*, etc.

The experimental validation shows that, for *MongoDB*, constructing the index at the DBMS layer using the *MapReduce* framework is the best approach. Moreover, the results show that constructing the index at the application layer is the best approach when using *Oracle*. The best time performance for full text search is achieved when using the proposed *inverted index* instead of no index or a build-in index provided by the DBMS for both databases.

The work carried out in this paper has revealed many promising areas and further research in optimizing the creation of an *inverted index* for full text search in databases. An area that is worthy of further investigation is the implementation of a lemmatization algorithm at the DBMS layer. Another interesting approach could be the use of a *GPU MapReduce framework* (He et al. 2008) to construct *inverted indexes* because it implies a serious amount of parallel processing (Munteanu et al. 2015). Another area of interest would be to implement clustering algorithms inside the DBMS for unsupervised classification of text and for data and text mining. Lastly, it is worth mentioning that a text search engine that uses an optimized inverted index would be appropriate to be implemented inside the database for semantic search.

REFERENCES

Babenko, A.; Lempitsky, V. (2012). The inverted multi-index. *IEEE International Conference on Computer Vision and Pattern Recognition*, 3069–3076.

Bai, Q; Ma, C.; Chen, X. (2012). A new index model based on inverted index. *International Conference on Software Engineering and Service Science*, 157–160.

Bird, S.; Loper, E.; Klein, E. (2009). *Natural Language Processing with Python*. O'Reilly Media Inc.

Catena, M.; Macdonald, C.; Ounis, I. (2014). On inverted index compression for search engine efficiency. *Advances in Information Retrieval*, 359–371.

Dean, J.; Ghemawat, S. (2010). MapReduce: a flexible data processing tool, *Communications of the ACM*, 53, 1, 72–77.

Glory, V.; Domnic, S. (2013). Inverted index compression using Extended Golomb Code, *International Conference on Advances in Engineering, Science and Management*, 20–25.

He, B.; Fang, W; Luo, Q.; Govindaraju, N.K.; Wang, T. (2008). Mars: a MapReduce framework on graphics processors. *International Conference on Parallel Architectures and Compilation Techniques*, 260–269.

Ilic, M.; Spalevic, P.; Veinovic, M. (2014). Inverted index search in data mining, *Telecommunications Forum Telfor (TELFOR)*, 943–946.

Jin, L; Xiao-Fang, Z.; Shu-Liang, L; Fa-Yong, Z.; Xin-Cai, W. (2008). Design and Compressed Storage of inverted index Based on Mixed Word Segmentation, *International Workshop on Knowledge Discovery and Data Mining*, 334–338.

Kettunen, K.; Kunttu, T.; Järvelin, K. (2005). To stem or lemmatize a highly inflectional language in a probabilistic IR environment?, *Journal of Documentation*, 61, 4, 476–496.

Liu, F.; Yu, C. T., Meng, W.; Chowdhury, A. (2006). Effective keyword search in relational databases, *ACM SIGMOD*, 563–574.

Liu, X. (2010). Efficient maintenance scheme of inverted index for large-scale full-text retrieval. *International Conference on Future Computer and Communication*, 565–570.

Moore, S. (2014). *Oracle Database PL/SQL Language Reference 11g Release 2 (11.2)*.

Munteanu, G.; Mocanu, Ș.; Saru, D. (2015). GPGPU optimized parallel implementation of AES using C++AMP. *Journal of Control Engineering and Applied Informatics*, 17, 2, 73–81.

Porter, M. (1980). An algorithm for suffix stripping. *Program*, 14, 130–137.

Redmond, E.; Wilson, J. R. (2012). *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*. Pragmatic Bookshelf.

Shankar, S.; Dijcks, J.P. (2009). *An Oracle White Paper: In-database Map-Reduce*.

Shea, C. (2014). *Oracle Text Application Developer's Guide 11g Release 2 (11.2)*.

Sun, D.; Wang, X. (2014). inverted index Compression Using Multi-codes, *International Symposium on Computational Intelligence and Design*, 2, 96–99.

Truică, C.O.; A. Boicea, A.; Trifan, I. (2013). CRUD operations in MongoDB. *International Conference on Advanced Computer Science and Electronics Information*, 347–350.

Truică, C.O.; Boicea, A.; Rădulescu, F. (2014). Performance time for e-learning applications with multiple databases. *International Scientific Conference eLearning and Software for Education Bucharest*, 121–128.

Vishwakarma, S.K.; Lakhtaria, K.I.; Bhatnagar, D.; Sharma, A.K. (2014). An efficient approach for inverted index pruning based on document relevance. *International Conference on Communication Systems and Network Technologies*, 487–490.

Witten, I.H.; Moffat, A.; Bell, T.C. (1999). *Managing gigabytes: compressing and indexing documents and images* Morgan Kaufmann Publishers.

Wu, H.; Li, G.; Zhou, L. (2013). Ginix: Generalized inverted index for keyword search. *Tsinghua Science and Technology*, 18, 1, 77–87.