Efficient and Flexible Information Retrieval Using MonetDB/X100

Sándor Héman, Marcin Zukowski, Arjen de Vries, Peter Boncz

CWI
Kruislaan 413
Amsterdam, The Netherlands
{Firstname.Lastname}@cwi.nl

ABSTRACT
Today's large-scale IR systems are not implemented using general-purpose database systems, as the latter tend to be significantly less efficient than custom-built IR engines. This paper demonstrates how recent developments in hardware-conscious database architecture may however satisfy IR needs. The advantage is flexibility of experimentation, as implementing a retrieval system on top of a DBMS boils down to relational query formulation, rather than system programming. We demonstrate in the context of the TeraByte TREC efficiency task that our experimental MonetDB/X100 database system provides highly competitive results both regarding precision and speed. We analyze the two innovations in MonetDB/X100 that most contributed to this successful application of DB technology in IR, namely *vectorized in-cache* processing and the use of two new *light-weight compression* schemes that work between the RAM and CPU cache memory levels.

1. INTRODUCTION
Requirements of database management (DB) and information retrieval (IR) systems overlap more and more. Database systems are being applied to scenarios where features such as text search and similarity scoring on multiple attributes become crucial. Many information retrieval systems are being extended beyond plain text, to rank semi-structured documents marked up in XML, or maintain ontologies or thesauri. In both areas, these new features are usually implemented using specialized solutions limited in their features and performance.

Full integration of DB and IR has been considered highly desirable, see e.g. [5, 1] for some recent advocates. Yet, none of the attempts into this direction has been very successful. The explanation can be sought in what has been termed the 'structure chasm' [11]: database research builds upon the idea that all data should satisfy a pre-defined schema, and the natural language text documents of concern to information retrieval do not match this database application scenario. Still, the structure chasm does not explain why IR systems do not use database technology to alleviate their data management tasks during index construction and document ranking. In practice however, custom-built information retrieval engines have always outperformed generic database technology, especially when also taking into account the trade-off between run-time performance and resources needed.

Proceedings of the Conference on Innovative Database Research January 2007,Asilomar, California, USA.

The aim of proposed demonstration is twofold. First, the demonstration shows the advantage, in terms of *flexibility*, of using standard relational algebra to formulate IR retrieval models. Secondly, it shows that, by employing a *hardware-conscious* DBMS architecture, it is possible to achieve performance, both in terms of *efficiency* and *effectiveness*, that is competitive with leading, customized IR systems. The demonstration takes place in the context of our experimental MonetDB/X100 database system [3, 17], running a terabyte-scale information retrieval task consisting of the TREC TeraByte track (TREC-TB) [7]. Running TREC-TB on top of a DBMS efficiently, is something that has never been demonstrated to be realizable before, and could therefore be seen as a step towards closing the gap between DBMS and IR systems.

Section 2 describes the distinguishing features of MonetDB/X100 that allow it to run large-scale data processing tasks efficiently. Section 3 then explains the process of running TREC-TB on top of a DBMS, followed by an overview of the experiments we plan to demonstrate using MonetDB/X100 in Section 4.

2. MonetDB/X100 OVERVIEW
MonetDB/X100 is an experimental relational database kernel, optimized for high performance on data- and query-intensive workloads. It relies on the concept of *vectorized in-cache* query execution to achieve good CPU utilization [3], and a column-oriented storage manager that provides transparent *light-weight data compression* [17] to improve I/O-bandwidth utilization. An overview of the system architecture is presented in Figure 1.

Figure 1 shows an operator tree, being evaluated within MonetDB/X100 in a pipelined fashion, using the traditional *open()*, *next()*, *close()* interface. However, each *next()* call

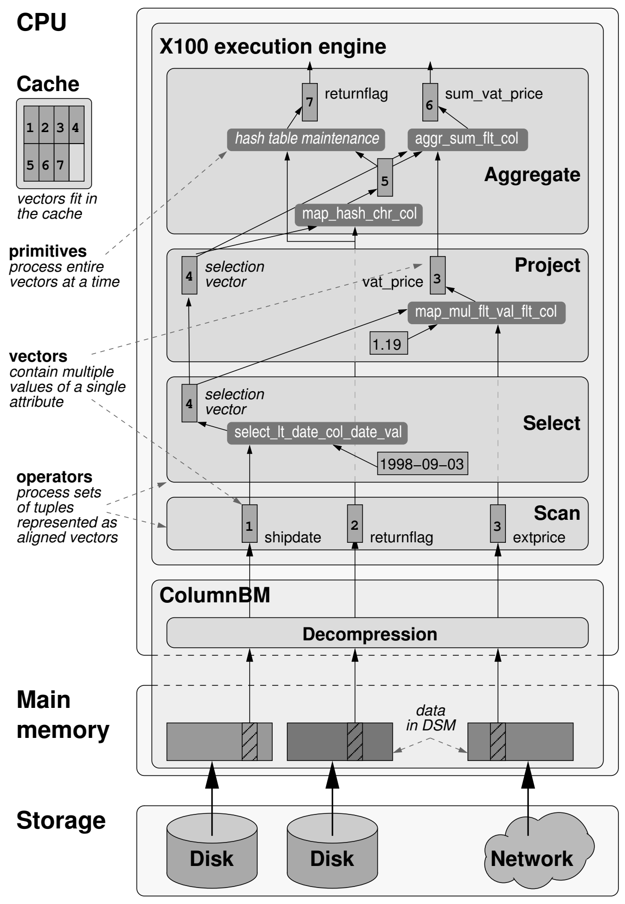

Figure 1: MonetDB/X100 architecture

within MonetDB/X100 does not return a single tuple, as is the case in most traditional DBMSs, but a *vector* of tuples. A vector is a unary array, containing a small slice of a single column. Vectorization of the iterator pipeline allows MonetDB/X100 *primitives*, which are responsible for computing core functionality such as addition and multiplication, to be implemented as simple loops over vectors. This results in function call overheads being amortized over a full vector of values instead of a single tuple, and allows the compiler to produce data-parallel code that can be executed efficiently on modern CPUs. Furthermore, the size of a vector is chosen in such a way, that all vectors needed by a query fit the CPU cache. This way, we avoid materialization of tuples that are being passed from one operator to the next, minimizing main memory access overheads. Such a *vectorized in-cache* architecture allows MonetDB/X100 query evaluation to be orders of magnitude faster than existing technology on data- and query-intensive workloads.

The processing power of MonetDB/X100 can make the system extremely I/O-hungry on certain queries. If the database does not fit main memory, the only solution to this problem is to increase the available I/O bandwidth. This can be done by adding more hardware, or by optimizing the DBMSs buffer manager for bandwidth utilization. With respect to the latter, MonetDB/X100 employs a buffer manager, called ColumnBM, that relies on a column-oriented

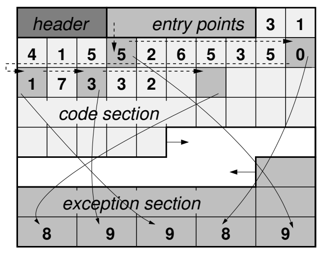

Figure 2: Compressed data layout (encoding the digits of π: 31415*9*26535*8979*32).**

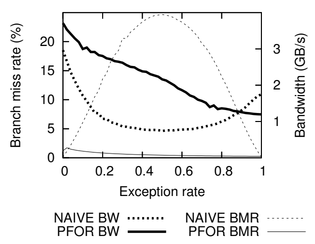

Figure 3: Branch Miss Rate (BMR) and decompression bandwidth versus exception rate

storage scheme, to avoid reading unnecessary columns from disk. Further, the granularity of disk accesses is in blocks of several megabytes, to optimize for fast sequential I/O.

2.1 RAM-CPU Cache Compression

In MonetDB/X100, we take the point of I/O-bandwidth utilization even further, by integrating ultra light-weight *RAM-CPU cache compression* into our system. The idea is, that by reading compressed blocks from disk, we can increase the perceived I/O bandwidth, as the actual data size of a block after decompression is assumed to be larger than the compressed block read from disk. For such an approach to be applicable even in the context of RAID storage systems that are capable of delivering data at several hundreds of megabytes per second, it should be clear that we need decompression routines that are capable of producing uncompressed data at speeds in the order several gigabytes per second. To reach such speeds, we recently introduced three novel compression algorithms, PFOR, PFOR-DELTA and PDICT [17], that are designed to sacrifice some performance in terms of compression ratio, in exchange for fast decompressibility. Furthermore, these compression schemes are integrated into the DBMS in such a way, that data blocks are stored in compressed form in RAM, and data is only decompressed on-demand, at vector granularity, directly into the CPU cache, where it is fed directly into the operator pipeline, without writing the uncompressed data back to main memory, as can be seen in Figure 1.

The two compression algorithms that are relevant for this work are PFOR and PFOR-DELTA. These take the trend in IR to move away from space-optimal compression towards light-weight compression [16, 2] to the extreme. A crucial difference is that our schemes are *vectorizable*, which means that the operations for (de)compressing subsequent values must be independent and expressible as a simple loop without any `if-then-else`. Our new compression schemes store input values as either *coded* or *exception* values. Coded values are small integers, with bit-widths b that may vary $1 \leq b \leq 24$. Exception values are stored in uncompressed form, thus they should be infrequent in order to still achieve a good compression ratio. In PFOR (Patched Frame-of-Reference), the small integers are positive offsets from a *base* value. Per disk block, one (possibly negative) *base* and one bit-width b is used. PFOR-DELTA (PFOR on deltas) encodes the *differences* between subsequent values in a column with PFOR.

In a compressed disk block (see Figure 2), the *code section* is forward-growing and densely packed, while the *exception section* grows backwards and stores uncompressed values. An *entry point section* holds for every 128 values the offset to the next exception point in the code section, and the corresponding location in the exception section. This allows fine-granularity access and skipping, which is especially useful during merging of inverted-lists.

The naive way to implement these simple schemes, is to use a special code (`MAXCODE`) to *mark* exception positions:

```
for(i=j=0; i<n; i++) /* NAIVE approach to decompression */
  if (code[i] < MAXCODE)
    output[i] = DECODE(code[i]);
  else
    output[i] = exception[--j]);
```

In this pseudo code, we abstract from algorithmic differences using the following macros: *(i)* `int ENCODE(ANY)`, that transforms an input value into a small integer, and *(ii)* `ANY DECODE(int)`, that produces the encoded input value given a small integer code.

The problem with this NAIVE approach is that it violates our guideline to avoid `if-then-else` in the inner loop. This hinders loop pipelining by the compiler, and also causes branch mispredictions when the else-branch is taken. Figure 3 demonstrates how NAIVE decompression throughput rapidly falls as the exception rate gets nearer to 50%. This is due to branch mispredictions[1] on the `if-then-else` test for an exception, that becomes impossible to predict for the CPU. To avoid this, we use the following "patch" approach for decompression:

```
for(int i=j=0; i<n; i++) /* LOOP1: decode regardless */
  output[i] = DECODE(code[i]);

for(int i=entry[0]; i<n; i+=code[i]) /* LOOP2: patch it up */
  output[i] = exception[--j];
```

Decompression is now split in two tight loops without any `if-then-else` statements, that all can be loop-pipelined by a compiler. Looking at Figure 2, we can see the block contains the integer sequence of π stored using PFOR$_{b=3}$ ($base = 0$). The exception values (i.e. digits ≥ 8) use their code value to store an offset to the next exception, forming a *linked list.* LOOP1 simply decodes all values, which will generate incorrect values for the exceptions. LOOP2 then **patches up** the incorrect values by walking the linked exception list and copying the exception values into the output array.

Figure 3 shows that the patched decompression variant clearly outperforms the naive variant and achieves a bandwidth of 3.5 GB/s without exceptions (LOOP1 only). With increased exception rate, patching work (LOOP2) increases linearly, and bandwidth diminishes accordingly.

3. IR ON TOP OF MonetDB/X100

3.1 TREC-TB Setup

Lately, the Text REtrieval Conference (TREC) introduced the "TeraByte Track" [7] as a large-scale text retrieval testbed.

[1]We collected IPC, cache misses, and branch misprediction statistics using *CPU event counters.*

Table 1: Top results for TREC-TB 2005

Run	p@20	CPUs	Time per query (ms)
MU05TBy3	0.5550	8	24
uwmtEwteD10	0.3900	2	27
MU05TBy1	0.5620	8	42
zetdist	0.5300	8	58
pisaEff4	0.3420	23	143

The TeraByte Track (TREC-TB) consists of the GOV2 document collection, together with tasks to evaluate system performance in terms of both effectiveness and efficiency. The data set consists of 25 million web documents, crawled from the *.gov* domain, with a total size of 426GB. System efficiency is measured by total execution time of 50,000 keyword-search queries. Effectiveness is evaluated by early precision (*p@20*) on a subset of 50 preselected queries for which relevance judgments are available. Table 1 shows the performance of the leading systems on the 2005 TREC-TB efficiency task.

We ran the 2005 TREC-TB on top of MonetDB/X100, using a single 3GHz Pentium Xeon CPU, 4GB of RAM, and a software RAID system consisting of 12 disks. For the distributed experiments in Section 3.4, we used a LAN of 8 machines, equipped with dual-core, 2GHz Athlon64X2 CPUs and 2GB RAM. To index the data, we used an *inverted list* data-structure, represented by a relational table. This *[term,docid,tf]* (TD) table, holds for each term, the ids of the documents the term appears in (*docid*), and the number of times the term occurs within a given document (*tf*). The table is ordered on *(term,docid)*, which allows the *term* column to be replaced by a range index onto *[docid,tf]*, and allows the occurrence lists of two arbitrary terms to be combined efficiently using merge-join. Additionally, per-document information is kept in a separate *[docid, name, length]* document table D, with *length* being measured in number of terms, and per-term global frequency counts (*ftd*) in table *T[term, ftd].*

3.2 Keyword Search Using Relational Algebra

Keyword search in a DBMS boils down to retrieving all the documents in which some or all of the query terms occur. Such a boolean retrieval approach can be formulated in relational algebra as a series of join operations over inverted lists, with boolean AND and OR mapping to *Join* and *OuterJoin* respectively. For example, a query *"information AND (storing OR retrieval)"* can be translated to:

```
Join(
  ScanSelect( TD1=TD, TD1.term="information" ),
  OuterJoin(
    ScanSelect( TD2=TD, TD2.term="storing" ),
    ScanSelect( TD3=TD, TD3.term="retrieval" )))
```

As the results for the runs **BoolAND** and **BoolOR** from Table 2 show, simple boolean queries without ranking result in very low precision. To address the low effectiveness, we present results with the Okapi BM25 [14] retrieval model.

Table 2: MonetDB/X100 TREC-TB Experiments

Run name (+ added feature)	p@20	Avg.query time (ms), cold data	Avg.query time (ms), hot data
BoolAND	0.0130	76	12
BoolOR	0.0000	133	80
BM25	0.5460	440	342
BM25T (+Two-pass)	0.5470	198	72
BM25TC (+Compression)	0.5470	158	73
BM25TCM (+Materialization)	0.5470	155	29
BM25TCMQ8 (+Quant.8-bit)	0.5490	118	28

The document score of a given query is expressed as:

$$S_{BM25}^{(D)} = \sum_{i=1}^{|Q|} \omega_{D,T_i} \quad (1)$$

$$\omega_{D,T} = \log(\frac{f_D}{f_{T,D}}) \cdot \frac{(k_1 + 1) \cdot f_{D,T}}{f_{D,T} + k_1 \cdot ((1-b) + b \cdot \frac{|D|}{avgdl})} \quad (2)$$

Given a query with $|Q|$ terms, the score of each document $S_{BM25}^{(D)}$ is a sum of scores of each query term for this document $\omega_{D,T}$. The per-term document scores $\omega_{D,T}$ are computed as a function of the total number of documents (f_D), the number of documents containing term T ($f_{T,D}$), the frequency of T within D ($f_{D,T}$), the document length ($|D|$), and the average document length (*avgdl*) over the whole collection. Variables k_1 and b represent two predefined constants. A relational query that finds the top 20 documents using this formula for a 2-term query could look like:

```
TopN(
  Project(
    Join(
      OuterJoin(
        ScanSelect( TD1=TD, TD1.term=t1_term ),
        ScanSelect( TD2=TD, TD2.term=t2_term ),
        TD1.docid=TD2.docid),
      Scan( D ),
      D.docid=MAX(TD1.docid,TD2.docid))
    [ D.docname, score=BM25(TD1.tf,D.doclen,t1_ftd)
      +BM25(TD2.tf,D.doclen,t2_ftd) ]),
  [ score DESC ], 20)
```

where `score=BM25(TD2.tf,D.doclen,t2_ftd)` is used as a shorthand, the real query contains the full BM25 formula.

Running above BM25 query requires a significant amount of processing time, as run **BM25** in Table 2 illustrates. This is not strange, however, considering that the average length of the 50.000 TREC-TB queries is 2.3 terms, with each term occurring in 775 thousand documents on average. There is, however, still room for improvement in terms of average query execution time.

3.3 Performance Optimizations

In this section we show a set of representative IR optimization techniques that allowed MonetDB/X100 to be competitive with the leading TREC participants.

First of all, The BM25 retrieval model scores each document, regardless the number of matching query terms. Given the observation that we are only interested in the top-N most relevant documents, we can refrain from computing the score for documents that are highly unlikely to make it into the top-N. Relying on a heuristic that documents that contain more query terms are likely to obtain a better score, we can obtain a significant performance improvement by following a **two-pass** strategy. In the first pass, we retrieve only the documents that contain *all* query terms, using a conventional *MergeJoin* instead of a *MergeOuterJoin*. Only if the first pass does not return enough results, we execute a second pass using the less restrictive *MergeOuterJoin*. Run **BM25T** in table 2 illustrates the performance gain, where roughly 15% of the 50.000 queries required a second pass.

Second, a large part of the cost of processing inverted lists is related to reading these from disk. Most IR systems use data compression to reduce this time. Using MonetDB/X100's built in **compression**, we were able to reduce the sizes of the *docid* and tf columns, which constitute the major part of total I/O, from 32 to 11.98 and 8.13 bits per tuple, respectively. To compress the partially ordered *docid* column, we used PFOR-DELTA compression with a codeword size of 8 bits. For the small integer tf values, we used PFOR, also with an 8-bit codeword size. The **BM25TC** run in Table 2 shows that this significantly improves the cold run, where all data needs to be read from disk. The time of the hot run did not change significantly, thanks to the high performance of the decompression routines implemented in MonetDB/X100.

The BM25 retrieval model aggregates the $\omega_{D,T}$ scores, which are query independent (see Eq. 1 and 2). Once its tuning parameters k_1 and b have been fixed, $\omega_{D,T}$ values may be precomputed, and can be stored as a separate *score* column in the TD table. This **score materialization** not only saves the cost of computing the per-term document score, it additionally allows us to perform a join with the document table only for the top-n documents, since the per-document properties are not needed for score computation anymore, only to retrieve the names of the top-ranked documents. As the results of the **BM25TCM** run show, this reduces the in-memory processing time significantly. However, the *cold* run did not improve, since the I/O overhead increased, as we now read 32-bit floating point $\omega_{D,T}$ values instead of compressed 8.13-bit term frequencies $f_{D,T}$. We were able to **quantize** the range of floating point scores into 8-bit integer numbers, without loss of precision, using the following *linear Global-By-Value* quantization

$$\omega'_{D,T} = \left\lfloor q \cdot \frac{\omega_{D,T} - L}{U - L + \epsilon} \right\rfloor + 1,$$

where L and U are the minimum and maximum values of $\omega_{D,T}$ in the entire collection. This produces integer values between 1 and q. We used a value of $q = 256$, significantly reducing the data size, and therefore the amount of I/O, which resulted in the performance results labeled **BM25TCMQ8**.

3.4 Distributed Execution

Text retrieval lends itself well for distributed execution, as we can easily split up the document collection into N partitions, and let each partition be indexed by its own server node. An incoming query can then be broadcast to all indexing nodes, with each of them returning its local top-N documents for that query. These per-node results can then

Table 3: Performance of the distributed runs

	Average query time (ms)		Average per-query server response time (ms)		
	absolute	amortized	min	average	max
Full TREC-TB run (hot data)					
Sequential	23.1				
8 servers	11.26		5.50	6.39	11.00
Using less servers (1 stream, fixed partition size)					
4 servers	9.21		5.92	6.78	9.06
2 servers	7.30		6.46	6.83	7.20
1 server	7.41		7.34	7.34	7.34
Increasing the concurrency (8 servers)					
1 stream	11.24	11.26	5.50	6.39	11.00
2 streams	9.61	4.86	5.56	6.92	9.36
4 streams	14.30	3.64	5.81	8.56	13.99
8 streams	25.46	3.26	6.21	12.28	25.07

be merged into a global top-N to produce the final result.

To investigate the scalability of our system, we ran distributed experiments on our LAN, using 8 partitions, each indexed by a separate machine. Thanks to MonetDB/X100's data compression, the whole index (10GB), could be kept in RAM, so that I/O is eliminated as a performance factor.

However, as the results in 3 show, the *speedup* using 8 machines is far from perfect (it decreases from 23.1 only to 11.26 msec). The main cause for the non-linear speedup is load imbalance, as is shown in the right half of Table 3: with increased number of database servers, the average per-server running times start to vary significantly. With 8 servers, the slowest one (which determines the overall query latency) takes twice as long as the fastest (11 vs. 5.5 msec). In a real IR system, however, such load imbalance affects latency but not throughput, as the system will be handling multiple queries continuously and differences even out. This is currently modeled in the TREC terabyte efficiency track by submitting a limited number of concurrent query "streams" to the system. The lower part of Table 3 shows that with an increased amount of concurrency, latency deteriorates much less than linear (i.e. throughput improves). As a result, throughput does scale linearly, as 8 servers are able to process more than 300 queries per second, taking an amortized 3.26 msec per query only (vs. 23 msec for one server).

4. TO BE DEMONSTRATED

We propose to demonstrate our system from two different angles: *basic search* and *performance.*

basic search Provides the user with a google-like search interface to enter keyword queries and browse the ranked result documents, which are stored in the DBMS as well. The user is allowed to select the search strategy and optimizations to be used during the search. To get a deeper understanding of what is going on in the system, given the chosen search strategy, alongside with the query results, we display the relational query plan that was executed, annotated with profiling information.

performance To demonstrate the raw performance of the system, we evaluate batches of official TREC-TB queries under any of the settings presented in Table 2. For distributed runs, we analyze performance as a function of the number of nodes (keeping data sizes fixed) and the number of simultaneous query input streams. We present run-time statistics, such as average query latency over nodes, best node latency, worst node latency, CPU utilization, average query length, average data volume processed per query, by sampling system state at regular intervals and displaying the statistics to the user in a graphical interface. To demonstrate the impact of MonetDB/X100 design decisions on performance, we also run benchmarks using varying MonetDB/X100 parameters, such as the vector size used in the execution pipeline.

5. RELATED WORK

Integration of the DB and IR processing has recently been discussed in [5]. The authors present a set of motivating examples, discuss different approaches of combining area-specific processing techniques, and propose an extension to the relational algebra that provides various IR features, e.g. a top-k operator. They also discuss the new challenges this algebra brings for the relational query optimizers.

A good summary of the DB-community view on integration with IR technology was presented during the recent SIGMOD panel [1]. While most researchers discuss DB and IR integration within a DBMS, our approach is to rather provide IR applications with features necessary for the efficient execution of their tasks. In this sense it is similar to [10], where the authors store inverted lists in a Microsoft SQLServer and use SQL queries for keyword search. Similarly, in [9] the data is distributed over a PC cluster, and an analysis of the impact of concurrent updates is provided. Our approach extends this previous work, by showing how a much wider series of IR optimization techniques can be translated to database queries. These techniques and their effectiveness/performance trade-off are further demonstrated on a much larger collection (500GB TeraByte TREC vs. 500MB in [9]) and show significantly faster retrieval performance.

In the TREC benchmark there were a few attempts to use database technology, e.g. [12]. However, most of these systems used a DBMS for effectiveness tasks only, where the system efficiency was not an issue. Only one TeraByte TREC submission used a system built on top of the MySQL DBMS [6], but its precision and speed (5 sec per query) were disappointing compared to other participants.

There still is a large group of IR efficiency optimization techniques not discussed in this paper. For example, Buckley [4] presented an optimization technique in which, during *term-at-a-time* top-r search, execution stops when the score difference of the r-th and $r+1$-th document is larger than the sum of the maximum scores of the remaining attributes. Another pruning approach is the well-known Fagin algorithm [8], recently extended with probabilistic pruning [15]. The final interesting group of optimizations exploit word positions for improved retrieval effectiveness (e.g. [13]). We believe all these methods can be implemented on top of a DBMS using techniques similar to the ones presented in this paper.

6. CONCLUSION

In our demonstration, we will show that it is possible to run terabyte scale information retrieval tasks on top of a relational database engine. Besides, it will be demonstrated that we can easily implement several standard IR optimization techniques, and that these techniques allow us to rival customized IR systems in terms of performance. This work presents a step towards the integration of DB and IR systems, with some of the key ingredients needed to achieve this result being: MonetDB/X100's raw speed, light-weight data compression, and distributed execution.

7. REFERENCES

[1] S. Amer-Yahia. Report on the DB/IR Panel at Sigmod 2005. *SIGMOD Record*, 34(4):71–74, 2005.

[2] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.

[3] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of the Conference of Innovative Database Research (CIDR)*, pages 225–237, Asilomar, CA, USA, 2005.

[4] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *Proceedings of the International Conference on Information Retrieval (ACM SIGIR)*, pages 97–110, Montreal, Canada, 1985.

[5] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR Technologies: What is the Sound of One Hand Clapping? In *Proceedings of the Conference of Innovative Database Research (CIDR)*, pages 1–12, Asilomar, CA, USA, 2005.

[6] C. L. A. Clarke, N. Craswell, and I. Soboroff. Overview of the TREC 2004 Terabyte Track. In *Proceedings of the Text Retrieval Conference (TREC)*, Gaithersburg, MD, USA, 2004.

[7] C. L. A. Clarke, F. Scholer, and I. Soboroff. The TREC 2005 Terabyte Track. In *Proceedings of the Text Retrieval Conference (TREC)*, Gaithersburg, MD, USA, 2005.

[8] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. pages 102–113, 2001.

[9] T. Grabs, K. Bhoem, and H.-J. Schek. PowerDB-IR: scalable information retrieval and storage with a cluster of databases. *Knowledge and Information Systems*, 6(4):465–505, 2004.

[10] D. A. Grossman, O. Frieder, D. O. Holmes, and D. C. Roberts. Integrating structured data and text: A relational approach. *JASIS*, 48(2):122–132, 1997.

[11] A. Y. Halevy, O. Etzioni, A. Doan, Z. G. Ives, J. Madhavan, L. McDowell, and I. Tatarinov. Crossing the structure chasm. In *Proceedings of the Conference of Innovative Database Research (CIDR)*, 2003.

[12] K. Mahesh, J. Kud, and P. Dixen. Oracle at TREC 8: a lexical approach. In *Proceedings of the Text Retrieval Conference (TREC)*, Gaithersburg, MD, USA, 1999.

[13] D. Metzler and W. B. Croft. A markov random field model for term dependencies. In *Proceedings of the International Conference on Information Retrieval (ACM SIGIR)*, pages 472–479, Salvador, Brazil, 2005.

[14] S. E. Robertson, S. Walker, and M. Beaulieu. Okapi at TREC-7: automatic ad hoc, filtering, VLC and interactive track. In *Proceedings of the Text Retrieval Conference (TREC)*, pages 143–167, Gaithersburg, MD, USA, 1998.

[15] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 648–659, Toronto, Canada, 2004.

[16] A. Trotman. Compressing inverted files. *Inf. Retr.*, 6(1):5–19, 2003.

[17] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the International Conference of Data Engineering (IEEE ICDE)*, Atlanta, GA, USA, 2006.