MonetDB/X100 at the 2006 TREC TeraByte Track

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

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

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' [8]: 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.

To investigate the feasibility of running terabyte scale information retrieval tasks on top of a relational engine, our team from CWI participated in the 2006 TREC Terabyte Track, using its experimental MonetDB/X100 database system [3, 11]. This system, is designed for high performance on data-intensive workloads, whereof TREC-TB is an excellent example. Furthermore, we believe that standard relational algebra provides enough flexibility to express most IR retrieval models, and show 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.

This notebook is organized as follows. 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 indexing the TREC-TB collection, and the resulting relational schema. This is followed by a description of the TREC-TB runs we submitted, together with the hardware platforms used to run them. Effectiveness and efficiency results for these runs are then presented in Sections 6 and Section 7, respectively, before concluding in Section 8.

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* [11] 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 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 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

Text Retrieval Conference (TREC) November 2006, Gaithersburg, Maryland, USA.

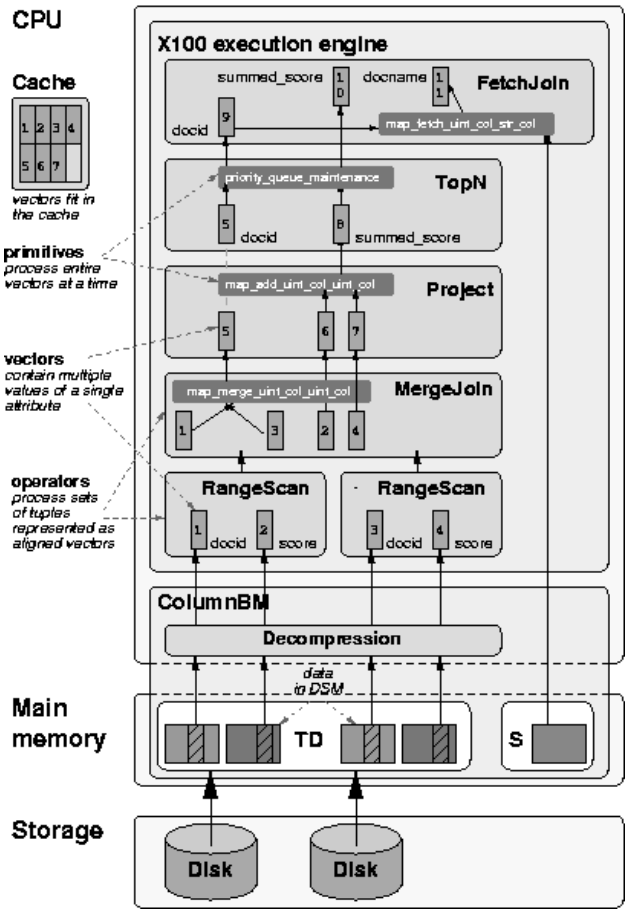

Figure 1: MonetDB/X100 architecture

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 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.

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 of several gigabytes per second. To reach such speeds, we recently introduced three novel compression algorithms, PFOR, PFOR-DELTA and PDICT [11], 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.

3. INDEXING

Indexing the 426GB TREC-TB document collection, consisting of roughly 25 million web documents, entails three main phases: parsing, constructing an inverted index structure using relational tables, and index compression. Parsing is done using an external program, that scans the collection and filters out markup and stopwords. For the remaining text, the parser returns (docid, term) pairs (DT) for each term it encounters, with terms being stemmed using a Porter stemmer [9], converted to lowercase, and scrambled into 64bit integers. Our scrambling function produces a one-to-one mapping from its input string to its integer representation, as long as no other input string has the same thirteen character long prefix. This is achieved by iterating over the input string, and on each iteration multiplying the current integer result by twenty-seven and adding the i'th characters offset from the character 'a' + 1. Furthermore, the parser generates a unique *docid* identifier for each document encountered, and outputs it, together with the documents name and length (in number of terms).

To index the data, we used the *inverted list* data-structure, which can be easily represented as a relational table. To build this index from the DT output of the parser, the following relational query, which sorts and then aggregates on (term, docid) pairs, was used:

```
# TD computation using DT
  Aggr(
    Sort(
      Scan(DT, [docid, term] ),
      [ term, docid ] ),
    [ term, docid ],
    [ tf = count() ] )
```

The *[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, and per term information *[term, ftd]* in table T. The relational table layout, together with the amount of storage each field occupies, is summarized in table 1.

3.1 Compression

As Table 1 shows, the full index (the D, T and TD tables), occupies approximately 29 GB uncompressed when we ignore the *term* column in TD, and replace it with a range index of negligible size. We applied MonetDB/X100's PFOR and PFOR-DELTA light-weight column compression algorithms [11] to reduce the total size of this index to roughly 9GB. The benefit of this is twofold. First of all, due to the

Table 1: Database tables and constants used

symbol	column name	semantic	sorted type		compression scheme	bits
DT – 12.3 Gtuples, output of parsing						
D	docid	document id	int	Y	none	32
T	term	term code	long	N	none	64
TD – 3.5 Gtuples, document-level index						
T	term	term code	long	Y	$\text{PFD}_{b=1}$	2.13
D	docid	document id	int	Y	$\text{PFD}_{b=8}$	11.98
$f_{D,T}$	tf	frequency of T in D	int	N	$\text{PF}_{b=5}$	5.91
$\omega_{D,T}$	score	score of T in D	float	N	none	32
$\omega'_{D,T}$	scoreQ	quantized score	int	N	$\text{PF}_{b=8}$	8.00
D – 25 Mtuples, output of parsing, per-document information						
D	docid	document id	int	Y	none	32
	docname	document name	str	Y	none	88
$\|D\|$	doclen	document length	int	N	none	32
T – 12 Mtuples, per-term information						
T	term	term code	long	Y	none	64
$f_{T,D}$	ftd	#documents with T	int	N	none	32
Global constants						
k_1	k1	BM25 parameter (0.8)				
b	b	BM25 parameter (0.3)				
f_D	numdocs	number of documents ($25M$)				
$avgdl$	avgdoclen	average document length (491)				
compression: **PF**=PFOR, **PFD**=PFOR-DELTA, all with base=0						

minimal decompression overhead of the compression algorithms, I/O bandwidth utilization is improved, as the data gets decompressed only when it is used, upon crossing the RAM-CPU Cache boundary. Second, the compressed index requires less memory to make it fully main-memory resident. Even if it does not fit fully, more data can be cached in RAM in compressed form, improving overall performance.

In short, Patched Frame of Reference (PFOR) compression stores a column of n-bit integer values as a b-bit integer offset from an arbitrary *base* value, with $b < n$. All values in the range $[base, base + 2^b - 1]$ are stored in b bits, with b being minimized. Values outside this range, are stored in uncompressed form, to make PFOR robust against outliers that would unnecessarily increase b. PFOR-DELTA is similar to PFOR, with the difference that PFOR-DELTA operates on the differences (deltas) between subsequent values in a column. This makes PFOR-DELTA well-suited for clustered or (partially) sorted columns. The rightmost column of Table 1 shows the reduction in storage requirements after applying compression.

Both PFOR and PFOR-DELTA are relatively generic database compression mechanisms; neither has been optimized for IR needs. Furthermore, a single column is always fully compressed with a fixed set of compression parameters, often resulting in suboptimal compression ratios. We are looking into ways of making compression more adaptive, and better suited for the IR domain, without loosing too much of our decompression efficiency.

3.2 BM25 Score Materialization

The *[term,docid,tf]* (TD) structure presented in Section 3, together with per-document and per-term information, is in itself sufficient to compute a ranked document list for a given query. In our experiments, we use the Okapi BM25 formula for document ranking:

$$S^{(D)}_{BM25} = \sum_{T \in Q} \omega_{D,T} \tag{1}$$

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

Evaluation of this formula is rather expensive in terms of CPU time, and can be avoided. The score for each *(term, docid)* pair is independent of a query, and can thus be precomputed. This means, that we can extend the TD table with a *score* column, which contains the precomputed scores. MonetDB/X100 uses column-wise storage, and only reads those columns from disk that are actually needed for a query. So, in case the *score* column is used, the *tf* column is left untouched. However, as the BM25 scores are 32 bit floating point numbers, as opposed to the 5.91 bit *tf* values, the storage requirements for our index increase from 18 bits to 44 bits per tuple. To avoid such an increase in index size, we decided to replace the floating point scores by so-called *score ranks* [2], which quantize the range of floating point numbers into small, compressed integer numbers. We used 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. The formula produces integer numbers between 1 and q. We chose q to be equal to 256, resulting in 8-bit integer scores, as this provided the best trade-off in precision, and run-time efficiency. This resulted in our final index occupying roughly 10GB, as the quantized scores occupy somewhat more space than the compressed *tf* values. We aim to investigate possibilities to generalize floating point quantization into a dictionary based compression scheme that provides bounds on the error introduced when mapping floating point ranges to integer numbers.

4. QUERYING

4.1 Relational Query Plan

Keyword search in a DBMS boils down to retrieving all the documents in which some or all of the query terms occur, and then ranking this list of documents. Given that we have our pre-computed, materialized and quantized BM25 scores, the ranking is simply a summation of these scores for each *(queryterm, docid)* pair. In relational algebra, this could look as follows for a two term query:

```
TopN(
  Project(
    MergeOuterJoin(
      RangeSelect( TD1=TD, TD1.termid=10 ),
      RangeSelect( TD2=TD, TD2.termid=42 ),
      TD1.docid = TD2.docid),
    [ S.docid = MAX(TD1.docid,TD2.docid),
      score = TD1.scoreQ + TD2.scoreQ ]),
  [ score DESC ], 20)
```

First, for each query term, a `RangeSelect` is used on the TD table, to retrieve the list of documents the query term appears in, together with its score on each document. These lists of *(docid, scoreQ)* pairs are then joined, producing the relation *[TD1.docid, TD1.scoreQ, TD2.docid, TD2.scoreQ]*.

One should note that, in case a document contains only one of the query terms, `MergeOuterJoin` pads the other side of the join result with NULL values, thereby producing a disjunctive boolean restriction on query term presence, i.e. a document should contain one or more of the query terms to propagate into the join result. On the other hand, a regular `MergeJoin` would only propagate those documents that contain *all* query terms, thereby producing a boolean conjunctive evaluation. The output of the join is then fed into the `Project` operator, which sums the scores for both query terms, on a per document basis. Finally, `TopN` filters out the N documents with the highest overall score, in descending order.

4.2 Two-Pass Evaluation

By default, The BM25 retrieval model scores each document, regardless the number of matching query terms. This coincides with the query plan that uses `MergeOuterJoin`, introduced in Section 4.1. 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 those documents that contain more query terms are likely to obtain a better score [4], 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 N results, we execute a second pass using the less restrictive *MergeOuterJoin*. On the full efficiency run, such a second pass was required in only 15.5% of the queries.

5. EXPERIMENTAL SETUP

We evaluated the performance of our DBMS driven IR engine on top of three different hardware architectures, resulting in the following four configurations:

DISK1 A mid-end server architecture, consisting of a single 3.0GHz Intel Xeon CPU, 4GB RAM, and a 10 disk RAID0 I/O subsystem, capable of storing 1TB of data. This configuration is intended to evaluate I/O dominated performance, as the whole index does not fit main memory.

MEM1 A main-memory oriented high-end (but relatively old, 2003) server architecture, containing four 1.4GHz AMD Opteron CPUs and 16GB of RAM. For experiments on this machine, data was first loaded into RAM over the network from the raid connected to the DISK1 system. In this setup, only a single query stream was used.

MEM4 Exactly the same setup as MEM1, with the only difference that this configuration always uses four query streams instead of one. This configuration was added to evaluate scalability of our system, by utilizing all four CPUs, without network interference.

DIST8 A cluster of eight modern desktop machines, built from off-the-shelf hardware components. Each node contains a 2GHz AMD64 X2 dual-core 3800+ CPU, has 2GB RAM, and 2 disks configured for RAID0. The document collection is partitioned over all 8 nodes, with each node indexing its own partition, in parallel. Queries are submitted to a *broker* program, which broadcasts each query to all eight nodes, and merges the local document rankings returned by each node into a global ranking. As the partial indices of each node fit into main-memory, these experiments don't involve any I/O. In this setting, we used four query streams to hide network latencies induced by communication with the broker.

All listed systems run the Linux operating system (Fedora Core 4).

We participated in both the ad-hoc task, measuring effectiveness, and in the efficiency task.

Table 2: Effectiveness Results

Label	P@10	P@20	MAP	InfAP
DISK1ah-50	0.5340	0.4780	0.2770	0.2299
DISK1ah-150	0.5591	0.5171	0.2952	NA
DIST8ah-50	0.5380	0.4750	0.2766	0.2308
DIST8ah-150	0.5577	0.5138	0.02952	NA
DISK1ah*-50	0.5440	0.4690	0.2677	0.2183

Table 3: Efficiency Results, with indexing time in minutes, and query times in seconds

Label	Index		Query		
	Size	Time	Streams	Avg. Time	Total Time
DISK1	10GB	1000	1	0.1971	19708
MEM1	10GB	1000	1	0.0790	7914
MEM4	10GB	1000	4	0.0805	2052
DIST8	10GB	185	4	0.0132	539

6. EFFECTIVENESS RESULTS

We submitted two different runs for the ad-hoc task. One to evaluate single node effectiveness, and one to investigate any differences in a distributed setting. If some documents outside the TopN obtain the same score as the TopN document, there can be slight variations in the results returned by these configurations, since the distributed run relies on the broker to merge the per-node local TopN's into a global TopN. As Table 2 shows, the differences between the DISK1 and DIST8 runs are minor. The table furthermore distinguishes between efficiency results obtained on both the 50 2006 topics and on the 150 combined topics of 2004, 2005 and 2006.

The final run in Table 2, labeled DISK1ah*, was not submitted officially. This run was conducted after the relevance judgments were released. As can be seen in table 1, our officially submitted runs used BM25 parameters different from the commonly used $k_1 = 1.2$ and $b = 0.5$, as these turned out to work better on the 2004 and 2005 topics. The DISK1ah* run uses these more common parameters. As the results show, our modified parameters performed worse on P@10. However, P@20 and MAP scores are better using the modified parameters.

7. EFFICIENCY RESULTS

For the efficiency runs, we measured both index creation time and index size. Furthermore, we evaluated average query response time (*latency*) for returning the top-20 results on all 100.000 efficiency topics and the total time needed to execute all queries (reflecting query *throughput*). Results are summarized in Table 3.

The comparison between MEM1 and MEM4 shows good system scalability. The fact that MEM4 uses four query streams, keeps all four CPUs in the system busy. Average query response time hardly suffers, and total query execution time is roughly divided by four, as we hoped. This also means, that using four CPUs, the system does not suffer from any contention of the memory bus, which can become an issue when increasing SMP parallelism. Note that our use of database tables that are kept compressed in RAM and are only decompressed in a small vector-granularity (in the CPU cache), helps keep to the memory bandwidth usage down (and increases the amount of inverted list data that can be cached). In the DIST8 configurations, we used four parallel query streams to hide any network latencies, thereby increasing query throughput significantly.

To have a point of reference, we also participated in the comparative TREC-TB task. We chose to do this on the Zettair system [10], as we expected this system to be the most competitive in terms of efficiency, which was our main focus. On our DISK1 architecture, Zettair achieved an indexing time of 460 minutes, which beats us by more than a factor two. However, it has to be mentioned that our indexing times include generation of many optional index structures that are not needed for the runs submitted to TREC, and should therefore be lower if this is cleaned out. We plan to do this for the final version of this notebook. In terms of query processing speed, Zettair required 9304 seconds to execute the 10.000 comparative queries. Of this total, 8094 seconds was CPU time. This means that the system was almost CPU bound, and therefore almost at its peak performance on our 3.0GHz Intel Xeon CPU. Comparing our 0.1971s average query execution time against Zettairs 0.9304s, we beat them by a factor 4.7.

7.1 Precision of Efficiency Runs

After submission of our final efficiency results, we noticed a slight deviation with respect to the official rules. The rules state, that the top-20 as returned by the efficiency runs, should be the same as the twenty topmost results in the top-10000 submitted for the ad-hoc runs. Due to our two-pass policy (see section 4.2), this is not the case though. For the ad-hoc runs, we executed a second (disjunctive) pass only if the first (conjunctive) pass did not fill the full top-10000. For the efficiency runs, the second pass was triggered in case of less than 20 results. This means, that the ad-hoc results are more often based on the disjunctive query plan, as the second pass is triggered more often. The effect of this is, that the top-20s of the ad-hoc and the efficiency runs can show slight differences. For the 150 combined ad-hoc topics, a second pass was needed 100 times, while for efficiency runs on the same 150 topics, it has been triggered only three times. Overall, for the full efficiency run with 100,000 topics, a second pass was needed in 15.5% of the cases, with the conjunctive plan returning 86838 documents on average.

Table 4: Precision of efficiency runs

Label	P@10	P@20
MEM1-50	0.5360	0.4720
MEM1-150	0.5651	0.5168

We believe, that in a real-world system, execution of only the conjuctive plan is sufficient to satisfy an end-user's need. The system can first present the conjunctive results to the end-user, and only execute the disjunctive plan in case the user desires to see more results than the amount returned by the conjunctive plan. One should note that in a database system like X100 it easy to write a query that complies with the rules, namely, one can return $C \cup (D \setminus C)$, where C and D are the results of the conjunctive and disjunctive runs, respectively. To confirm that our efficiency runs did not compromise precision, we present precision scores in table 4. These numbers are very similar to the ad-hoc effectiveness scores in Table 2, confirming that the efficiency runs we submitted are capable to satisfy the end-users need.

8. CONCLUSION

By participating in the 2006 TREC TeraByte Track, we have shown that it is possible to run terabyte scale information retrieval tasks on top of a relational database engine, and that such an approach can rival customized IR systems in terms of performance. This work presents a step towards the integration of DB and IR systems, identifying some of the key ingredients needed to achieve this result being: MonetDB/X100's raw speed, light-weight data compression, and distributed execution. In the future, we plan to investigate the effect of light-weight compression schemes that are better suited for IR tasks, such as improved inverted list compression and generalized floating point quantization. A closely related research direction investigates how an array database system with an IR-researcher friendly query language [6] can generate these highly efficient MonetDB/X100 query plans automatically from a declarative specification of the retrieval model[7].

9. 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. Simplified Similarity Scoring Using Term Ranks. In *Proceedings of the International Conference on Information Retrieval (ACM SIGIR)*, pages 226–233, Salvador, Brazil, 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] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the Conference of Information and Knowledge Management (CIKM)*, pages 426–434, New Orleans, LA, USA, 2003.

[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] R. Cornacchia and A. P. de Vries. A declarative DB-powered approach to IR. In *Proceedings of the European Conference on IR Research (ECIR)*, 2006.

[7] R. Cornacchia, S. Héman, M. Zukowski, A. P. de Vries, and P. A. Boncz. Flexible and efficient IR using Array Databases. Submitted for publication, 2006.

[8] 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.

[9] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[10] Y. Bernstein et al. RMIT University at TREC 2005: Terabyte and Robust Track. 2005.

[11] 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.