



Centrum voor Wiskunde en Informatica

REPORTRAPPORT

INS

Information Systems



Information Systems

Flexible and efficient IR using array databases

R. Cornacchia, S. Héman, M. Zukowski, A.P. de Vries,
P.A. Boncz

REPORT INS-E0701 JANUARY 2007

Centrum voor Wiskunde en Informatica (CWI) is the national research institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organisation for Scientific Research (NWO). CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

Copyright © 2007, Stichting Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-3681

Flexible and efficient IR using array databases

ABSTRACT

The Matrix Framework is a recent proposal by IR researchers to flexibly represent all important information retrieval models in a single multi-dimensional array framework. Computational support for exactly this framework is provided by the array database system SRAM (Sparse Relational Array Mapping) that works on top of a DBMS. Information retrieval models can be specified in its comprehension-based array query language, in a way that directly corresponds to the underlying mathematical formulas. SRAM efficiently stores sparse arrays in (compressed) relational tables and translates and optimizes array queries into relational queries. In this work, we describe a number of array query optimization rules and demonstrate their effect on text retrieval in the TREC TeraByte track (TREC-TB) efficiency task, using the Okapi BM25 model as our example. It turns out that these optimization rules enable SRAM to automatically translate the BM25 array queries into the relational equivalent of inverted list processing including compression, score materialization and quantization, such as employed by custom-built IR systems. The use of the high-performance MonetDB/X100 relational backend, that provides transparent database compression, allows the system to achieve very fast response times with good precision and low resource usage.

2000 Mathematics Subject Classification: 68U35 Information systems (hypertext navigation, interfaces, decision support, etc.)

1998 ACM Computing Classification System: H.3.3 Information Search and Retrieval, H.2.4 Systems

Keywords and Phrases: information retrieval, array databases, query optimization, database compression

Flexible and Efficient IR using Array Databases

Roberto Cornacchia
Roberto.Cornacchia@cwi.nl

Sándor Héman
Sandor.Heman@cwi.nl

Marcin Zukowski
Marcin.Zukowski@cwi.nl

Arjen de Vries
Arjen.de.Vries@cwi.nl

Peter Boncz
Peter.Boncz@cwi.nl

CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

The Matrix Framework is a recent proposal by IR researchers to flexibly represent all important information retrieval models in a single multi-dimensional array framework. Computational support for exactly this framework is provided by the array database system SRAM (Sparse Relational Array Mapping) that works on top of a DBMS. Information retrieval models can be specified in its comprehension-based array query language, in a way that directly corresponds to the underlying mathematical formulas. SRAM efficiently stores *sparse arrays* in (compressed) relational tables and translates and optimizes array queries into relational queries. In this work, we describe a number of array query optimization rules and demonstrate their effect on text retrieval in the TREC TeraByte track (TREC-TB) efficiency task, using the Okapi BM25 model as our example. It turns out that these optimization rules enable SRAM to automatically translate the BM25 array queries into the relational equivalent of inverted list processing including compression, score materialization and quantization, such as employed by custom-built IR systems. The use of the high-performance X100 relational backend, that provides transparent database compression, allows the system to achieve very fast response times with good precision and low resource usage.

Keywords and Phrases: information retrieval, array databases, query optimization, database compression

1. INTRODUCTION

Information Retrieval researchers develop methods to assess the degree of relevance of data to user queries. While ideally such a retrieval model could be considered ‘just’ a (somewhat complicated) query for a database system, in practice the researcher attempting to deploy database technology to information retrieval will stumble upon two difficulties. First, database implementations of IR models are still inefficient in runtime and resource utilization if compared to hand-built solutions. Published accounts of using DBMS software for IR tasks reported either disappointing performance (the only TREC-TB database result, on MySQL, achieved a query time of 5 seconds and low precision [CCS04]) and/or disk resource usage much higher than custom-built IR solutions [GBS04]). Second, the set-oriented query languages provided by relational database systems provide a fairly poor abstraction in expressing information retrieval models. Specifically, the lack of explicit representation of ordered data has long been acknowledged as a severe bottleneck for developing scientific database applications [MV93], and we believe the same problem has hindered the integration of databases and information retrieval.

1.1 Approach & Contributions

The main contributions of our research are: (i) SRAM, the first system that implements the “Matrix Framework for IR”, a recent IR proposal to express retrieval models flexibly and uniformly in terms of matrix operations, (ii) the design of a new array mapping to relational database systems that allows to instantiate and query *sparse* array data structures, (iii) a number of array query translation

and *optimization* rules that generate efficient relational database queries for array formulas, (iv) a demonstration of the benefits of generic data compression by the DBMS to IR applications running on top of it, (v) existence proof in case of the TREC TeraByte track (TREC-TB) efficiency task that an IR application on top of a DBMS *can* rival and even beat custom-built IR systems in terms of query throughput, while achieving equivalent precision using minimal hardware resources.

The Matrix Framework for IR [RTK05] maps IR concepts to matrix spaces and matrix operations, providing a convenient logical abstraction that facilitates the design of IR systems. To get a flavor, this framework describes the occurrence of terms $t \in T$ in documents $d \in D$ as a two-dimensional matrix $DT_{d,t}$. The IR researchers that recently proposed this framework have shown that all popular IR retrieval strategies such as TF/IDF, BIR, Language modeling and Probabilistic Logical modeling can be easily expressed in matrix formulas. One of the stated goals of the Matrix Framework is to be able to easily experiment with and compare these IR models. As such, a usable *implementation* of the Matrix Framework constitutes a highly valuable workbench for IR researchers, allowing to test variations of IR retrieval models without engineering effort, by just changing the IR model specifications, expressed in mathematical notation.

Sparse Array Databases. A natural implementation of the Matrix Framework is based on the array abstraction, which can be mapped onto relational database systems. However, this appears to be a viable solution only with specific support for processing *sparse* arrays. The data representation that matrix spaces offer is very redundant compared to a set-based representation (although documents contain only a small fraction of the possible terms, all the presence/absence combinations are represented explicitly). This results in matrices that are typically very large and extremely sparse (the density of the $LD_{l,d}$ binary matrix, which encodes *collection_location-document* associations, is lower than 0.000005% for the TREC-TB collection). One interesting observation about the relational mapping of the sparse document-term matrix is that the resulting physical data structure is the equivalent of an inverted list – a best practice in custom-built IR systems.

Array Query Optimization. Sparse array representations offer many opportunities for query optimization, in particular with regard to strategies that allow direct computations on the sparse relational representations (i.e. those that avoid materializing sparse matrices into dense intermediates). For the main SRAM operations on sparse arrays, such as function mapping, array reshaping, aggregation, and top-N, we present optimization rules that map these onto efficient relational query plans. We also discuss how *aggregate unfolding* can transform aggregate computations into efficient relational plans equivalent to inverted list merging.

Efficient DBMS with Compression. A rather mundane drawback hindering the adoption of the DBMS as a component in IR systems has been its mediocre expression evaluation performance, especially when compared with hand-written programs. We make use of the X100 relational engine that is specifically designed to tackle this issue, using a database architecture that is tuned to modern computer architecture, limiting database interpretation overhead using “vectorized execution” and enforcing good use of parallel execution capabilities and cache memories of modern CPUs. X100 in addition offers *compressed* column-wise storage, using a variety of high-performance database compression schemes. These compression schemes are transparent to users and queries, and transform our sparse document-term matrices into compressed inverted lists – another best practice in custom-built IR systems. It is noteworthy that X100 was developed for data warehousing and OLAP and is purely relational and not specifically targeted to IR applications.

The TREC TeraByte track efficiency task is our main example scenario and evaluation test case. We use SRAM to specify the Okapi BM25 retrieval model, and show how our subsequent query optimizations improve query performance. The resulting query throughput actually exceeds that of all previously reported results with custom-built IR systems on comparable hardware, with identical precision (most TREC-TB participants use BM25 nowadays). In this, we see proof of the practical viability of the Matrix Framework for IR in general, and the SRAM sparse array approach in particular.

1.2 Outline

Section 2 introduces the main concepts of SRAM, and explains its mapping of sparse arrays and array queries onto relational databases. Section 3 details the experimental setup we used for TREC-TB. This both encompasses the retrieval model created using SRAM, as well as some physical characteristics of the TREC-TB dataset. With regards to the latter, we focus on the compression features of the X100 relational database engine and their effects on the TREC-TB dataset. Section 4 describes the experiments on TREC-TB with a number of processing variants and SRAM array optimizations. Related research in IR and DB is discussed in Section 5, before we conclude and outline future work in Section 6.

2. SPARSE RELATIONAL ARRAY MAPPING

Although sparse array computations are not a settled topic in the numerical analysis research field, a large amount of literature and software implementations exists, in particular for two-dimensional arrays (matrices) [GL83]. However, such solutions focus on the optimization of single operations rather than full expressions (for example, many different implementations exist for matrix multiplication), assuming a specific data encoding, or even tuned for specific hardware. This is in contrast with the ‘database approach’ proposed here, which turns the numerical problem into a query optimization problem, providing the following potential benefits:

- *Data independence*: relational expressions are transparent to the physical organization of data. The data access optimization problem is taken care of by the relational engine, rather than being bound to specific numerical algorithms.
- *Resource utilization*: modern database engines are tuned for effective exploitation of the (possibly limited) hardware resources, and use cache-conscious, CPU-friendly algorithms, becoming more and more attractive for computational-intensive applications.
- *Open ‘black boxes’*: instead of providing ad-hoc implementations of, say, matrix multiplication or matrix transposition algorithms, express them as a combination of native database primitives (join, selection, etc.). This allows the query optimization process to take such operations into account within a larger problem and look for the overall best query plan.

2.1 The SRAM system

The SRAM (Sparse Relational Array Mapping) system is a prototype tool for mapping sparse arrays to relations and array operations to relational expressions. While SRAM syntax allows to express array operations on an element by element basis, the system translates such element-at-a-time operations to collection-oriented database queries, suited for (potentially more efficient) bulk processing. Although SRAM is RDBMS-independent, supporting translation to SQL queries, in this paper we present relational queries expressed in the query language of X100.

The life-cycle of array queries through the SRAM architecture can be summarized by the following sequence of transformations:

1. Array comprehension syntax \mapsto Array algebra
2. Array algebra \mapsto Relational algebra
3. Relational algebra \mapsto Relational plan

Both array and relational algebra expressions are rewritten by a traditional rule-based optimizer.

The next three sections introduce the array-syntax of the SRAM front-end, the storage of sparse arrays in relational databases, and a (not comprehensive) set of mapping rules from array-algebra to relational algebra, respectively. Table 1 summarizes the basic notation elements used throughout this paper for both arrays and relations. In particular, notice that arrays and their associated relations are denoted by a different text font: an array A is stored in the database as a relation A .

Table 1: Notation for arrays and relations

| Symbolic | Meaning |
|--|--|
| A | n -dimensional array A |
| $A_{i_0, \dots, i_{n-1}}$ | array A with dimension variables i_0, \dots, i_{n-1} |
| $\mathcal{S}_A \in \mathbb{N}^n$ | shape of A (vector of dimension lengths) |
| $A[\mathcal{S}]$ | n -dimensional array A of shape \mathcal{S} |
| \mathcal{S}_A^i | length of the i -th dimension of A |
| $ A = \prod_{i=0}^{n-1} \mathcal{S}_A^i$ | size of array A |
| \mathcal{D}_A | domain of A values |
| $\mathbf{i} \in \mathbb{N}^n$ | index values vector (i_0, \dots, i_{n-1}) |
| $A(\mathbf{i})$ | array value indexed by \mathbf{i} |
| ε | default value (of A) |
| non- ε (non- ε_A) | a value $v \neq \varepsilon$ ($v \neq \varepsilon_A$) |
| \mathbf{A} | relation \mathbf{A} |
| $ \mathbf{A} $ | size of relation \mathbf{A} |
| $\bar{\mathbf{A}}$ | $Grid(\mathcal{S}_A) \setminus \pi_{\mathbf{A}, \bar{\mathbf{i}}}(\mathbf{A})$ |
| $Grid(\mathcal{S})$ | populates a relation $\mathbf{G}(i_0, \dots, i_{n-1})$ with the enumeration of all index values in \mathcal{S} |
| $\mathbf{A}, \bar{\mathbf{i}}$ | relation attributes $(\mathbf{A}, i_0 \dots \mathbf{A}, i_{n-1})$ corresponding to array indices (i_0, \dots, i_{n-1}) |
| $\mathbf{A}, \bar{\mathbf{v}}$ | all \mathbf{A} attributes but $\mathbf{A}, \bar{\mathbf{i}}$ |

2.2 Comprehension syntax

The SRAM language defines operations over arrays declaratively in comprehension syntax [BLS⁺94], which allows to declare arrays by means of the following construct:

```
A := [ <array-cell value> | <array axes> ]
```

The section `<array axes>` specifies the *shape* \mathcal{S}_A of array A , namely the number of dimensions and the domain of each dimension, in the following form: $i_0 < I_0, \dots, i_{n-1} < I_{n-1}$. The value of each dimension variable i_j ranges from 0 to $I_j - 1$. Dimension variables must be specified and named. However, their explicit domain specification can be omitted when this is clear from the context.

The section `<array-cell value>` assigns a value to each cell indexed by the index values enumerated by the `<array axes>` section. For example, the expression $B = [\log(A(y,x)) \mid x < 5, y < 6]$ defines a new array $B[5,6]$, where the value of each cell is computed by applying the function *log* to cells of array A taken in transposed order.

A second constructor enumerates all the array elements explicitly. The definition of a one-dimensional explicit array $A[5]$ looks like this:

```
A := [ 10 42 0 3 1 ]
```

Explicit arrays of higher dimensionality can be specified by array nesting. The following expression defines the explicit array $A[2,5]$:

```
A := [ [10 42 0 3 1] [7 4 19 5 6] ]
```

Aggregations over any array dimension are also supported (`sum`, `prod`, `min`, `max`). For example, the summation of array B over its second axis is expressed as:

```
C := [ sum([ B(x,y) | x ]) | y ]
```

The shape \mathcal{S}_C of array C is easily identified by the rightmost axis y .

Retrieving the *top-N* values is allowed for one-dimensional arrays only. However, the resulting array does not contain sorted array values, but positions of the sorted values in the original array, which would otherwise be lost during the sorting. The following construct returns a *dense* array T , with $\mathcal{S}_T = [N]$:

```
T := topN(C, N, <ASC|DESC>)
```

The actual values can subsequently be fetched by dereferencing the original array: $D = C(T)$.

The SRAM syntax allows the definition of functions, implemented as macros expanded symbolically by a preprocessor at every occurrence, such as `myfunc(a,b) = (a / b) * 2`. More useful functions for the scope of this paper include matrix transposition and multiplication:

```
mxTrnsp(A) = [ A(j,i) | i,j ]
mxMult(A,B) = [ sum([ A(i,k) * B(k,j) | k ]) | i,j ]
```

An interesting remark is that arrays can be seen as mathematical functions that map points from their index space to the values addressed by those points. Therefore, functions can be used in SRAM syntax as non-stored arrays. An explicit assignment to a variable name, stores an array in the database:

```
<array name> := <comprehension expression>
```

The next section details about storage of sparse arrays in a relational database.

2.3 Storage of sparse arrays in a RDBMS

Many storage schemes have been proposed for sparse multi-dimensional arrays, with strong emphasis on the special case of two-dimensional arrays [DDRvdV00]. Most of them are tuned for specific data access patterns (e.g. Compressed Row/Column Storage, Compressed Diagonal Storage, Skyline Storage) for a particular application.

A generic choice for storing sparse arrays in relational databases maps every array to one relation, where array-cells are represented as tuples. The optimization of different data access patterns can be achieved by means of standard relational indexing structures on top of such relations, or by explicit tuple clustering/sorting. In the current storage scheme, any n -dimensional sparse array A , with ε_A denoting its default value, can be mapped to a relation A where each tuple enumerates the index values and cell value explicitly, and only those cells for which $A(\mathbf{i}) \neq \varepsilon_A$ need to be physically stored in the relation:

$$A \mapsto A(i_0, \dots, i_{n-1}, v) = \{(i_0, \dots, i_{n-1}, A(\mathbf{i})) \mid A(\mathbf{i}) \neq \varepsilon_A\}.$$

Notice that index columns together form a primary key for such a relation. SRAM uses the policy to store all persistent sparse arrays in relations that are clustered on this primary key (e.g. using index-organized tables). This means that the order in which the array dimensions are specified matters, and puts the tuples in the underlying database relation in lexicographical dimension order.

Because the nominal shape \mathcal{S}_A of the array A is known, the domain of the index columns in the associated relational table A is known as well, which makes the non-stored portion of the table, denoted as \tilde{A} , always computable as: $\tilde{A} = \pi_{\mathbf{i}, v=\varepsilon_A}(Grid(\mathcal{S}_A) \setminus \pi_{\mathbf{i}}(A))$. The function $Grid(\mathcal{S}_A)$ creates a relation $G(i_0, \dots, i_{n-1})$ filled with the enumeration of all index values in the \mathcal{S}_A domain. Although no standard relational operator can generate such a relation, most modern RDBMS can easily be extended to provide such a functionality. Support for *table functions*, i.e. external functions that can return a table, is also included in the SQL-2003 standard [EMK⁺04].

This storage scheme naturally extends to dense arrays, for which the ε value is not specified, and therefore all the tuples are physically stored. In case of persistent dense arrays, all values in the primary key are present and thus form a computable sequence. As an optimization, columns (i_0, \dots, i_{n-1}) could be omitted from the dense representation, as the full table can be produced using a view that regenerates these columns with *identity-columns*, as supported by SQL-99.

The sparse array representation may also be seen as a compression mechanism. In principle, this can be further exploited, by recursively removing frequent values and managing a list of missing elements $(\varepsilon_A^1, \varepsilon_A^2, \dots)$ from the initial table. However, we discuss here only the simpler case of a single level of compression. One special case worth mentioning are *boolean* sparse arrays. Here, the value stored in column v is always the same ($v = \neq$), hence for sparse boolean arrays v can be omitted from the physical representation.

The decision of whether to use the sparse or dense representation for an array A depends on many factors, including the particular application, the storage space available, and the density of the sparse array (defined as the fraction of the non- ε_A values). In the current incarnation of SRAM, sparse is the default array representation.

2.4 From Array Algebra to Relational Algebra

The first translation step of SRAM queries generates an array-algebra tree, which represents the sequence of operations performed on the stored arrays. The following list briefly describes the main operators of this algebra:

Array(A, \mathcal{S}) binds an array of shape \mathcal{S} to its relational representation A .

Grid(\mathcal{S}, i) creates an array of shape \mathcal{S} , whose values are the index values of its i -th dimension.

Apply(A, I_0, \dots, I_{n-1}) dereferences the n -dimensional array A , using n arrays whose values are index values for A .

Pivot(A, \mathcal{P}) permutes the dimensions of A following the axis order permutation specified in \mathcal{P} (e.g. $\mathcal{P} = [1, 0]$ for matrix transposition).

RangeSel($A, \mathcal{O}, \mathcal{S}$) selects from A the sub-array identified by offset \mathcal{O} from the origin and shape \mathcal{S} .

Replicate(A, n) increases the number of dimensions by 1, by replicating array A , n times.

Map(f, A, B, \dots) maps the function f to corresponding cells of arrays A, B, \dots .

Aggregate(f, j, A) collapses the first j dimensions by applying the aggregation function f over the remaining dimensions.

TopN($A, n, \langle \text{ASC} | \text{DESC} \rangle$) for vectors only, it returns the indices of the first n values in the desired order.

We classify these operations based on the type of processing required on input arrays: *shape-only* and *content-shape* operations. For the most complex operators, we present formal translation rules into relational algebra, using inference rule syntax: premises and conclusions appear above and below the horizontal line, respectively.

Shape-only array operations Shape-only array operations operate on the structure of arrays (e.g. dereferencing, replicating, pivoting, slicing). This translates to relational expressions that only involve the manipulation of index columns.

Apply. Dereferencing operations, i.e. the selection of array values based on their array position, are performed by the *Apply* operator:

$$\text{Apply}(A, I_0, \dots, I_{n-1})$$

Each of the I_j arrays, called index arrays, contains the j -th dimension coordinates of the values to select in A . In the current implementation, index arrays are limited by the following constraints: they must be dense and their value must be of integer type. As shown in Rule (APPLY), this operator is implemented by performing a series of primary key joins between relations $I_0 \dots I_{n-1}$, and a foreign-primary key join to retrieve the correct values from relation A .

$$\frac{\begin{array}{l} A; \mathbf{A} \quad I_0 \dots I_{n-1}; \mathbf{I}_0 \dots \mathbf{I}_{n-1} \\ \mathcal{S}_{I_0} = \dots = \mathcal{S}_{I_{n-1}} \quad \forall j \in [0 \dots n] : I_j \text{ is dense} \\ B = \text{Apply}(A, I_0, \dots, I_{n-1}) \end{array}}{\begin{array}{l} \mathcal{S}_B = \mathcal{S}_{I_0} \quad \varepsilon_B = \varepsilon_A \\ B \mapsto \mathbf{B} = \pi_{\bar{\mathbf{i}}=\mathbf{X}, \bar{\mathbf{i}}, \mathbf{v}=\mathbf{A}, \mathbf{v}} (\mathbf{X} \bowtie_{\bar{\mathbf{x}}=\mathbf{A}, \bar{\mathbf{i}}} \mathbf{A}) \\ \mathbf{X} = \mathbf{I}_0 \bowtie_{\bar{\mathbf{i}}} \dots \bowtie_{\bar{\mathbf{i}}} \mathbf{I}_{n-1} \end{array}} \quad (\text{APPLY})$$

In principle, all the supported reshaping operations (**Pivot**, **RangeSel**, **Replicate**) may be implemented by the *Apply* operator. First, a result space of the desired shape is produced for a given operation, using the *Grid* operator to generate the index arrays I_0, \dots, I_{n-1} ; second, each cell of the result array is filled in, fetching the correct values from the input array, by the expression $Apply(A, I_0, \dots, I_{n-1})$. For example, if $A = [1 \ 2 \ 3 \ 4 \ 5]$, the selection of the first 3 elements can be obtained by: $Apply(A, Grid([3]))$.

When evaluating sparse arrays however, this strategy becomes not viable in practice for these operators, although still correct. Specifically, the creation of the whole theoretical result space leads to a considerable waste of system resources. Index values ought to be manipulated ‘in-place’ for sparse arrays. As an example, the transposition of a matrix A can be mapped to a simple projection: $\pi_{i_0=i_1, i_1=i_0, v}(A)$, which swaps the two index columns. This specific operation can be captured by the following operator: $Pivot(A, [1, 0])$. Also *RangeSel* and *Replicate* translate to more efficient relational expressions than the generic *Apply* approach, involving selection and projection of index columns and cross products, respectively.

Content-shape array operations Content-shape array operations operate on both the structure and the content of arrays. In the following, we present the translation rules for operators *Map*, *Aggregate*, and *TopN*.

Map. Mapping a function over two or more arrays is a common content-shape array operation, captured by the array-algebra operator *Map*:

$$C = Map(f, A, B, \dots).$$

The simple case of a function applied to one array, $B = Map(f, A)$, translates to the relational expression $B = \pi_{i, v=f(A, v)}(A)$ for both dense and sparse evaluation. For the latter however, the value of the default value must be updated: $\varepsilon_B = f(\varepsilon_A)$.

Consider now the *dense* arrays $A[X, Y]$ and $B[Y, Z]$ and a function f that takes two input values. The expression $C = [f(A(x, y), B(y, z)) \mid x, y, z]$ defines a new array $C[X, Y, Z]$, where the function f has been applied to cells $A(x, y)$ and $B(y, z)$ for each possible combination of indices x, y, z . Notice that the shape \mathcal{S}_C of the result array C is defined as the union of all the input array shapes. The same operation on relational tables A and B basically maps to a join over the shared index columns, followed by a projection on the value columns:

$$\pi_{A.i \cup B.i, v=f(A.v, B.v)}(A \bowtie_{A.i \cap B.i} B) \quad (2.1)$$

Sparse arrays require a more complex relational mapping, in order to take care of missing tuples in the relations (i.e. non-stored default values ε_A and ε_B), denoted as \tilde{A} and \tilde{B} . The generic relational translation of the operator *Map* for two arrays is described by the translation Rule (MAP):

$$\frac{A; A \quad B; B \quad f: \mathcal{D}_A \times \mathcal{D}_B \rightarrow \mathcal{D}_C}{\mathcal{S}_C = \mathcal{S}_A \cup \mathcal{S}_B \quad \varepsilon_C = f(\varepsilon_A, \varepsilon_B)} \quad (MAP)$$

$$C \mapsto \mathcal{C} = \bigcup_{j=1}^4 \mathcal{C}_j$$

$$\begin{aligned} \mathcal{C}_1 &= \pi_{A.i \cup B.i, v=f(A.v, B.v)}(A \bowtie_{A.i \cap B.i} B) \\ \mathcal{C}_2 &= \pi_{A.i \cup \tilde{B}.i, v=f(A.v, \varepsilon_B)}(A \bowtie_{A.i \cap \tilde{B}.i} \tilde{B}) \\ \mathcal{C}_3 &= \pi_{\tilde{A}.i \cup B.i, v=f(\varepsilon_A, B.v)}(\tilde{A} \bowtie_{\tilde{A}.i \cap B.i} B) \\ \mathcal{C}_4 &= \pi_{\tilde{A}.i \cup \tilde{B}.i, v=f(\varepsilon_A, \varepsilon_B)}(\tilde{A} \bowtie_{\tilde{A}.i \cap \tilde{B}.i} \tilde{B}) \end{aligned}$$

In general, the *Map* between two arrays corresponds to relational join, but as sparse arrays consist of two parts (stored,omitted), there are four combinations, (stored,omitted)x(stored,omitted) and the *Map* consists of the union of four relational joins.

Aggregate. Array aggregation is supported by the operator *Aggregate*, with the following syntax:

$$B = \text{Aggregate}(f, j, A)$$

The first j dimensions are dropped by aggregating over the remaining $n-j$ dimensions. This semantics simplifies the translation, while not compromising the possibility of aggregating over any dimension, by first performing a (almost no-cost) *Pivot* operation. The supported aggregation functions f on array values are: *sum*, *prod*, *min*, *max*. The relational evaluation of this operator on sparse arrays can be mapped as the corresponding dense version (a standard relational aggregate), subsequently patched for all the non-stored tuples in each group. For example, if a given one-dimensional sparse array A , with $\varepsilon_A = 2$, is stored as a relation \mathbf{A} that misses x tuples, the summation over \mathbf{A} needs to be patched adding $2 * x$ to the result. This naturally extends to multi-dimensional arrays. Before presenting the formal mapping rule, we define the binary functions f_- and f_+^+ as follows:

$$\begin{aligned} f = \text{sum} &\Rightarrow f_- = +, & f_+^+ &= * \\ f = \text{prod} &\Rightarrow f_- = *, & f_+^+ &= \text{pow} \\ f = \text{min/max} &\Rightarrow f_- = \text{min/max}, & f_+^+ &= \text{id}_1 \end{aligned}$$

where the function id_1 always returns its first argument.

In Rule (AGGR), gs denotes the nominal group size, and relations \mathbf{G} , \mathbf{M} , and \mathbf{B} , compute the dense aggregation, the number of missing tuples per group, and the final sparse aggregation, respectively.

$$\begin{array}{c} A; \mathbf{A} \quad gs = \prod_{k=0}^{j-1} \mathcal{S}_A^k \quad B = \text{Aggregate}(f, j, A) \\ \hline \mathcal{S}_B = \mathcal{S}_A^{[j, \dots, n-1]} \quad \varepsilon_B = f_-^+(gs, \varepsilon_A) \\ B \mapsto \mathbf{B} = \pi_{\mathbf{i}, v=f_-(G.v, f_+^+(\varepsilon_A, M.v))}(\mathbf{G} \bowtie_{\mathbf{i}} \mathbf{M}) \\ \mathbf{G} = \pi_{(i_0=i_j, \dots, i_{n-1}=i_{n-1}, v)}((i_j, \dots, i_{n-1}) \mathcal{G}_{v=f(v)} A) \\ \mathbf{M} = \pi_{(i_0=i_j, \dots, i_{n-1}=i_{n-1}, v=gs-v)}((i_j, \dots, i_{n-1}) \mathcal{G}_{v=\text{count}(\ast)} \mathbf{A}) \end{array} \quad (\text{AGGR})$$

TopN. Top-N operations are not defined in standard relational algebra. However, most modern database systems implement such a functionality. SRAM uses an extended relational algebra that supports Top-N operations, whose syntax is:

$$\tau^n [\text{attr1} \langle \uparrow \mid \downarrow \rangle ; \text{attr2} \langle \uparrow \mid \downarrow \rangle ; \dots](\mathbf{A})$$

where n is the number of desired tuples in the result, $\text{attr1}, \text{attr2}, \dots$, are the ranking attributes, and $\langle \uparrow \mid \downarrow \rangle$ denotes $\langle \text{ASC} \mid \text{DESC} \rangle$ order.

Rule (TOPN-DESC) describes the relational translation of the array operator $\text{TopN}(A, n, \text{DESC})$. A similar rule exists for ascending order.

First, the top n candidates are selected from the stored relation \mathbf{A} . Then, a second set of n candidates \mathbf{M} is created for the non-stored portion of the relation, representing ε_A values. Finally, a regular top-N is performed on the union of these two set of candidates, and the index column of the result relation is projected as an array value. Notice that because the result array is defined as dense, it does not require a materialized index column itself.

$$\begin{array}{c} A; \mathbf{A} \quad |\mathcal{S}_A| = 1 \quad T = \text{TopN}(A, n) \quad n \leq |A| \\ \hline \mathcal{S}_T = [n] \\ \mathbf{B} = \pi_{v=i_0}(\tau^n[v \downarrow](\mathbf{T} \cup \mathbf{M})) \\ \mathbf{T} = \tau^n[v \downarrow](\mathbf{A}) \quad \mathbf{M} = \sigma_{\text{id} < n}(\pi_{\text{id}=\#\text{rowid}, v=\varepsilon_A}(\tilde{\mathbf{A}})) \end{array} \quad (\text{TOPN-DESC})$$

The computation of $\tilde{\mathbf{A}}$ (the non-stored indices), requires the difference of the full $Grid(|A|)$ with \mathbf{A} , which in turn could trigger generation of the full grid. However, the above selection on the identity column `#rowid` performs what in SQL is a `LIMIT n` on the generation of $\tilde{\mathbf{A}}$. This limits the amount of materialized grid tuples to a maximum of $|A| + n$, and in case of a sparse array it is likely to be very near n only.

2.5 Simplification rules

Several rules are considered in order to simplify the generic translation of function mapping and aggregation operations. Expression \mathbf{C}_4 in Rule (MAP) is always removed by Rule (EMPTY- π), as the value $f(\varepsilon_A, \varepsilon_B)$ coincides by definition with the default value ε_C of the new array, which requires no storage.

$$\frac{Y; \mathbf{Y} = \pi_{\bar{\mathbf{i}}, v=f(\bar{\mathbf{v}})}(\mathbf{X}) \vdash f(\bar{\mathbf{v}}) = \varepsilon_Y}{\mathbf{Y} \equiv \{\}} \quad (\text{EMPTY-}\pi)$$

This rule, which has a great impact on the simplification of both *Map* and *Aggregate* operations, is activated by the *arithmetic optimization*, discussed in Section 2.6.

A *Map* operation between *dense* arrays A and B , for which default values ε_A and ε_B are not defined, translates to (2.1) by applying Rule (EMPTY- $\tilde{\mathbf{A}}$), which removes both expressions \mathbf{C}_2 and \mathbf{C}_3 . The same rule removes either \mathbf{C}_2 or \mathbf{C}_3 in case of a combination of dense and sparse input arrays.

$$\frac{A; \mathbf{A} \quad \nexists \varepsilon_A}{\tilde{\mathbf{A}} \equiv \{\}} \quad (\text{EMPTY-}\tilde{\mathbf{A}})$$

A further simplification is performed when arrays A and B have the same shape, i.e. when $\mathcal{S}_A = \mathcal{S}_B$. When this condition holds, the missing index values identified by $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$ can be found in \mathbf{B} and \mathbf{A} respectively. Rule (SAME- \mathcal{S}) removes the expensive computation of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$ by means of difference operations on tables \mathbf{A} and \mathbf{B} .

$$\frac{A; \mathbf{A} \quad B; \mathbf{B} \quad \mathcal{S}_A = \mathcal{S}_B}{\begin{array}{l} \mathbf{A} \bowtie_{\mathbf{A}.\bar{\mathbf{i}} \cap \tilde{\mathbf{B}}.\bar{\mathbf{i}}} \tilde{\mathbf{B}} \equiv \mathbf{A} \setminus_{\mathbf{A}.\bar{\mathbf{i}} \cap \mathbf{B}.\bar{\mathbf{i}}} \mathbf{B} \\ \mathbf{B} \bowtie_{\mathbf{B}.\bar{\mathbf{i}} \cap \tilde{\mathbf{A}}.\bar{\mathbf{i}}} \tilde{\mathbf{A}} \equiv \mathbf{B} \setminus_{\mathbf{B}.\bar{\mathbf{i}} \cap \mathbf{A}.\bar{\mathbf{i}}} \mathbf{A} \end{array}} \quad (\text{SAME-}\mathcal{S})$$

By combining rules (MAP) and (SAME- \mathcal{S}), a new translation Rule (ALIGNED-MAP) can be derived for mapping a function f over shape-aligned sparse arrays A and B , which uses the outer join relational operator:

$$\frac{\begin{array}{l} A; \mathbf{A} \quad B; \mathbf{B} \quad \mathcal{S}_A = \mathcal{S}_B \\ C = \text{Map}(f, A, B) \quad f: \mathcal{D}_A \times \mathcal{D}_B \rightarrow \mathcal{D}_C \end{array}}{\begin{array}{l} \mathcal{S}_C = \mathcal{S}_A \quad \varepsilon_C = f(\varepsilon_A, \varepsilon_B) \\ C \mapsto \mathbf{C} = \pi_{h(\mathbf{A}.\bar{\mathbf{i}} \cup \mathbf{B}.\bar{\mathbf{i}}), v=f'(\mathbf{A}.\mathbf{v}, \mathbf{B}.\mathbf{v})}(\mathbf{A} \bowtie_{\mathbf{A}.\bar{\mathbf{i}} \cap \mathbf{B}.\bar{\mathbf{i}}} \mathbf{B}) \\ h(\mathbf{A}.\bar{\mathbf{i}} \cup \mathbf{B}.\bar{\mathbf{i}}) = \forall \bar{\mathbf{i}} \in \bar{\mathbf{i}}: \mathbf{A}.\bar{\mathbf{i}} \vee \mathbf{B}.\bar{\mathbf{i}} \\ f'(\mathbf{A}.\mathbf{v}, \mathbf{B}.\mathbf{v}) = f(\mathbf{A}.\mathbf{v} \vee \varepsilon_A, \mathbf{B}.\mathbf{v} \vee \varepsilon_B) \end{array}} \quad (\text{ALIGNED-MAP})$$

Notice that the removal of either or both expressions \mathbf{C}_2 and \mathbf{C}_3 in Rule (MAP) corresponds to the substitution of the outer join in Rule (ALIGNED-MAP) by left/right outer join or inner join operations respectively. The formal rules used to obtain such translations are similar to the ones shown above.

2.6 Arithmetic optimization

During the translation from array-algebra to relational algebra, a simple arithmetic analysis takes place. Common patterns such as $(x * 0) = 0$, $(x * 1) = x$, $(x/1) = x$, $\log(1) = 0$, and so on, are identified and simplified. This activates Rule (EMPTY- π), which removes predictable computations from all translation rules, most importantly from Rule (MAP) and Rule (AGGR).

Such an arithmetic analysis is particularly important in that it provides the main mechanism for limiting the increased complexity of generic sparse array evaluation. As an example, consider the matrix multiplication in the expression $\mathbf{C} = [\text{sum}([\mathbf{A}(\mathbf{x},\mathbf{y}) * \mathbf{B}(\mathbf{y},\mathbf{z}) \mid \mathbf{y}]) \mid \mathbf{x},\mathbf{z}]$, where A and B are both sparse arrays with $\varepsilon_A = \varepsilon_B = 0$. It is easily verified that computations on the non-stored values normally required for sparse arrays can be simply ignored, as they do not alter the final result. In particular, this example requires no extra processing with respect to a dense array evaluation, because of arithmetic simplifications $(x * 0) = 0$, which simplifies the multiplication part, and $(x + 0) = x$, which simplifies the summation part.

In addition, it is a crucial mechanism to avoid the introduction of ε elements in intermediate results, which would make the *physical* density of such arrays higher than the nominal one and therefore make the optimization process less reliable.

3. EXPERIMENTAL SETUP

We used SRAM to build an IR application that runs the TREC-TB efficiency task. Section 3.1 describes the set-up of this IR application, including the (best-practice) IR methods we selected for it. Our main purpose with this application is to demonstrate the flexibility and efficiency with which IR retrieval models can be specified and implemented using SRAM.

In Section 3.2 we detail on the data preparation process for the TREC-TB dataset. Parsing and stemming phases are performed by a separate program, which generates compressed relations for the two matrices needed to bootstrap the collection indexing as described by the Matrix Framework for IR. Thereafter, in Section 3.3 we describe the most important features of our X100 relational database backend. Here, we will give special attention to the database compression features offered by X100.

3.1 TREC-TB

TREC TeraByte track [CSS05] has introduced a task to evaluate IR system efficiency on ranking a large web-crawled collection documents (the GOV2 collection). This data set consists of 25 million web documents, with a total size of 426GB. System efficiency is measured by total execution time of 50,000 queries. Effectiveness is evaluated by early precision ($p@20$) on a subset of 50 preselected queries for which relevance judgments are available.

Okapi BM25 as an Array Query We selected the top-scoring Okapi BM25 [RWB98] formula as the IR retrieval model for these experiments. Note that we select BM25 just for its ubiquity and aptness for TREC-TB.

The BM25 document scoring formula is:

$$S_{BM25}^{(D)} = \sum_{T \in Q} \omega_{D,T} \quad (3.1)$$

$$\omega_{D,T} = \log\left(\frac{f_D}{f_{T,D}}\right) \cdot \frac{(k_1 + 1) \cdot f_{D,T}}{f_{D,T} + k_1 \cdot ((1 - b) + b \cdot \frac{|D|}{\text{avgdl}})} \quad (3.2)$$

The constants used in this formula are shown in Table 2 and all arrays used in Table 3. One can observe that the SRAM expression for the BM25 document score is an almost direct transcription of the mathematical formula to ASCII characters, which demonstrates the intuitiveness of array comprehensions as an IR query language:

```
s(d) = sum( [ w(d,Q(t)) | t ] )
w(d,t) = log( $Ndocs / F(t) )
* (( $k1 + 1 ) * TD(t,d))
/ (TD(t,d) +
  $k1 * ((1 - $b) + ($b * S(d) / $avgdl)))
```

Table 2: Global constants used for TREC-TB

| symbol | meaning |
|---------|-------------------------------|
| k_1 | BM25 parameter (1.2) |
| b | BM25 parameter (0.5) |
| f_D | number of documents (25M) |
| $avgdl$ | average document length (491) |

Table 3: SRAM Arrays used for TREC-TB

| name | symbol | meaning | array(type, ϵ) | size |
|------------|-------------------|--------------------|--------------------------|-----------|
| $LD_{l,d}$ | $\exists d_{L,D}$ | doc at location? | sparse(bool,0) | 12.3Gx25M |
| $LT_{l,t}$ | $\exists t_{L,T}$ | term at location? | sparse(bool,0) | 12.3Gx12M |
| $TD_{t,d}$ | $f_{T,D}$ | freq of (term,doc) | sparse(int,0) | 25Mx12M |
| S_d | $ D $ | size of doc | dense(int) | 12M |
| F_t | f_T | doc-freq of term | sparse(int,0) | 25M |
| Q_t^i | T | term id | dense(long) | variable |

The above array expressions correspond to (3.1) and (3.2), respectively. Notice that they are defined as functions, i.e. as non-materialized arrays.

The arrays F_t , $TD_{t,d}$, and S_d contain document frequency for terms t , within-document term frequency for (t,d) pairs, and document size (length) of documents d , respectively. The dense array Q_t^i enumerates the query terms for each query Q^i . These query vectors are temporary objects, typically instantiated inside the query, using the explicit array constructor syntax defined in Section 2.2. Finally, the following expression selects the best 20 scored documents:

```
D20 := topN( [s(d) | d<20], DESC )
S20 := [s(d) | d](D20)
```

First, the indices of the 20 highest scored documents are retrieved from the array $[s(d) | d]$ by the `TopN()` construct and materialized as array $D20$. Then, the scores of those documents are fetched by dereferencing the array $[s(d) | d]$ with $D20$.

Application-Level IR Optimizations In our TREC-TB experiments we applied three additional best-practice IR application-level optimizations: two-pass, score quantization and distribution.

Two-Pass. The BM25 retrieval model scores each document, regardless the number of matching query terms. Broder et al. [BCH⁺03] have proposed a two-pass processing strategy based on the *heuristic* that documents that contain more query terms are likely to obtain a better score. The first pass ranks only documents that contain a set of terms with summed weights exceeding a pre-defined threshold. This may reduce significantly the number of documents considered, improving execution time. Only when the first pass does not return enough documents, a second pass is performed that considers all documents.

We experimented with a simplified version of this approach, where the first pass ranks only documents containing *all* query terms. As this strategy does not compute the true BM25 score, we use the term BM25C for it. The C-suffix stands for “Conjunctive”, as we look for documents that have the conjunction of all query terms (similarly, we may call the real BM25 computation “Disjunctive”). Thus, in TREC-TB, our IR application first computes the top 20 documents using the conjunctive expression, and only if $|D20| < 20$, it actually computes the top-N of the disjunctive expression. If a query has fewer terms, if these terms are more frequent, or if these are likely to co-occur, the probability of the conjunctive expression finding enough documents increases. In the TREC-TB query set, the second pass turns out to be necessary in only 15% of the queries.

The conjunctive scoring formula multiplies the BM25 score with a (0,1) boolean that is only 1 for

those documents that have all terms:

$$S_{BM25C}^{(D)} = \prod_{T \in Q} (\omega_{D,T} > 0) * \sum_{T \in Q} \omega_{D,T}$$

This is transcribed in SRAM syntax as follows:

$$\begin{aligned} \mathbf{s}(d) = & \text{prod} ([\mathbf{w}(d, Q(t)) > 0 \mid \mathbf{t}]) \\ & * \text{sum} ([\mathbf{w}(d, Q(t)) \quad \mid \mathbf{t}]) \end{aligned}$$

Multiplications of arrays with such sparse (boolean) matrices can significantly improve the running time of a query. The underlying reason is that such multiplications can make the result sparser, and thus smaller, when considering its representation as a relational table. This is the reason why the conjunctive variant, being a multiplication on the disjunctive (normal) BM25 formula, turns out to be much faster, and makes the two-pass strategy beneficial (see Section 4).

Score Materialization and Quantization. The BM25 score for document d is the sum of all $w(d, t)$ term-document scores for all terms t in the query. The SRAM array function $w(d, t)$ is quite compute-intensive, as it contains four floating-point multiplications and additions, three divisions, and one logarithm. It is a best-practice in IR to pre-compute (materialize) such partial scores. In principle, SRAM could actually use compiler techniques similar to *strength reduction*, to automatically extract query-independent parts from a scoring formula, and materialize these. We consider this further work. For these experiments, we just declared $w(d, t)$ as a persistent array instead of an array function.

A drawback of materializing floating-point $\omega_{D,T}$ scores, is that, unlike the small integer numbers $f_{D,T}$, floating point numbers are much harder to compress (see Section 3.3 for more on compression). An alternative to storing the $\omega_{D,T}$ values is to replace these by so-called *score ranks* [AM05a] – small integer values that are the result of quantization of floating point scores. For example, the *linear Global-By-Value* quantization

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

where L and U are the minimum and maximum values of $\omega_{D,T}$ in the entire collection, produces integer values between 1 and q and ϵ is a small threshold (not a default value!). Quantization with too small integers can lead to precision loss in retrieval, and after some experiments we settled for 8-bits integers (bytes).

Distributed Execution. Our IR application is a program that receives a full-text query, stems the keywords into terms, and uses SRAM to generate and run a database query (or sometimes two queries, in the two-pass strategy), and finally formats and returns the names of the top-N documents. A common technique to improve IR query throughput (as measured in TREC-TB) is distributed IR query processing [BDH03, GBS04]. Distribution is relatively easy in IR, as one can simply use multiple IR retrieval setups that each index only a partition of the full document collection. In our case, we use multiple array database servers on separate machines, and the IR application sends queries to all these servers in parallel, and combines the top-N results in a single top-N. Notice that in the two-pass strategy, this can happen twice; the queries for the second phase are only sent if the total amount of scored documents returned by all databases in the first phase is less than N.

We can expect the *average* query times of the different servers to be identical, as the server hardware is identical and the dataset is perfectly partitioned, However, query times from individual servers naturally exhibit a certain variance, and the final query answer is as slow as the slowest response. This variation in query times may be caused by many factors (OS scheduling, I/O interference, network events, etc.) and can never be fully avoided. Thus, the database servers will be idle sometimes, waiting for a slower server, or waiting for the network while receiving the next query or while sending a result. The latest edition of TREC-TB therefore allows multiple queries (from multiple *query streams*) to be

issued simultaneously to the IR system. The net effect in a distributed architecture is that instead of spending time waiting, all servers can always be kept busy, such that throughput is optimized. In principle, IR system throughput then scales perfectly with parallelism, which is confirmed in Section 4. We implemented multiple query streams by simply running a different instance of our IR application for each stream.

3.2 TREC-TB Data Preparation

To parse the TREC-TB GOV2 collection, we used a program that performed standard Porter stemming [Por80] and stop word removal (the 19 most common words were used as stop words). It sequentially scans all files in the collection, and for each term it encounters writes out a *term* identifier and a *document* identifier, to two separate files. We can view the resulting single-column files as two-column [location,term] and [location,document] relations, if we attach to each value a (virtual) densely increasing sequence number *location*. Our relational backend has support for identity columns (#rownum), as well as tables stored in binary files, such that these files are treated by it as database relations. These two binary relations, in turn, represent the two sparse boolean matrices location-term $LT_{l,t}$ and location-document $LD_{l,d}$, as used in the Matrix Framework for IR [RTK05]. As mentioned previously, sparse boolean arrays do not need to materialize the value column because it is known to be the negated default value. Thus, the relational representations LD and LT of sparse boolean matrices $LD_{l,d}$ and $LT_{l,t}$ just consist of their dimension columns (which together form the primary key).

As described in the Matrix Framework for IR, $TD_{t,d}$ is computed from these in SRAM by a simple matrix multiplication:

$$TD := \text{mxMult}(\text{mxTrnsp}(LT), LD)$$

S_d is computed as a summation over LD :

$$S := [\text{sum}([LD(l,d) \mid l]) \mid d]$$

Similarly, the computation of F_t requires a summation over TD , whose values are first converted from term-document frequency to term-document presence/absence:

$$F := [\text{sum}([\min(TD(t,d), 1) \mid d]) \mid t]$$

Table 4 shows the resulting database schema, with suggestive column names for clarity of presentation.

Since we make the sparse arrays listed above persistent, SRAM needs them clustered on their primary key (i.e. sorted on that order). The most expensive step in data preparation is doing so for TD . Sorting its underlying relational table corresponds exactly with creating an inverted list from a table of postings. In all, our Pentium4 server took 7 hours for all TREC-TB data preparation.

3.3 X100

X100 is an experimental relational database engine, optimized for high performance data warehousing and OLAP workloads. It relies on the concept of *vectorized in-cache* query execution to achieve good CPU utilization [BZN05], and a column-oriented storage manager that provides transparent *light-weight data compression* [ZHNB06] to improve I/O-bandwidth utilization. An overview of the system architecture is presented in Figure 1.

X100 offers a language based on standard relational algebra, providing operators such as `Scan`, `ScanSelect`, `Project`, `Aggr`, `TopN`, `Sort`, `Join`.

Figure 1 shows an operator tree, being evaluated within X100 in a pipelined fashion, using the traditional `open()`, `next()`, `close()` interface from the Volcano [Gra94] iterator model. However, each `next()` call within X100 does not return a single tuple, as is the case in most traditional DBMSs, but

Table 4: X100 Relations resulting from Table 3

| symbol | column name | meaning | data type | compression scheme | bits |
|--|---------------|---------------------|-----------|--------------------|-------|
| $LD_{l,d}$ LD table – 12.3 Gtuples (output file of parsing) | | | | | |
| l major-key | LD.location | location id | long | #rownum | 0 |
| d minor-key | LD.docid | document id | int | none | 32 |
| $LT_{l,t}$ LT table – 12.3 Gtuples (output file of parsing) | | | | | |
| l major-key | LT.location | location id | long | #rownum | 0 |
| t minor-key | LT.termid | term id | long | none | 64 |
| $TD_{t,d}^*$ TD table – 3.5 Gtuples, term-document info | | | | | |
| t major-key | TD.termid | term id | long | PFD _{b=1} | 2.13 |
| d minor-key | TD.docid | document id | int | PFD _{b=8} | 11.98 |
| $f_{T,D}$ | TD.tf | frequency of T in D | int | PF _{b=8} | 8.13 |
| $\omega_{T,D}$ | TD.score | score of T in D | float | none | 32 |
| $\omega'_{T,D}$ | TD.scoreQ | quantized score | int | PF _{b=8} | 8 |
| S_d D table – 25 Mtuples, document info | | | | | |
| d key | D.docid | document id | int | #rownum | 0 |
| $ D $ | D.doclen | document length | int | none | 32 |
| F_t T table – 12 Mtuples, term info | | | | | |
| t key | T.termid | term id | long | #rownum | 0 |
| f_T | T.ftd | doc frequency | int | none | 32 |
| $Q_t^1..Q_t^{50000}$ transient Q^i tables – avg. 4 tuples (query len) | | | | | |
| t | Q^i .termid | term id | long | none | 64 |
| Compression: PF =PFOR, PFD =PFOR-DELTA, for all base=0 | | | | | |

a *vector* of tuples. A vector is a unary memory array, containing a small slice of a single column. Vectorization of the iterator pipeline allows 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 X100 query evaluation to be an order of magnitude faster than existing technology on data- and query-intensive workloads [BZN05].

The processing power of 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, 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 X100 we take the point of I/O-bandwidth utilization even further, by integrating ultra light-weight column compression, that happens on the boundary between RAM in the CPU cache (whereas other compressed database systems typically target the disk-RAM boundary). These compression schemes are integrated into the DBMS in such a way, that data blocks are stored in compressed form in memory (such that more data fits the buffer cache), 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.

For compression to be beneficial to speed at all in the context of RAID storage systems that

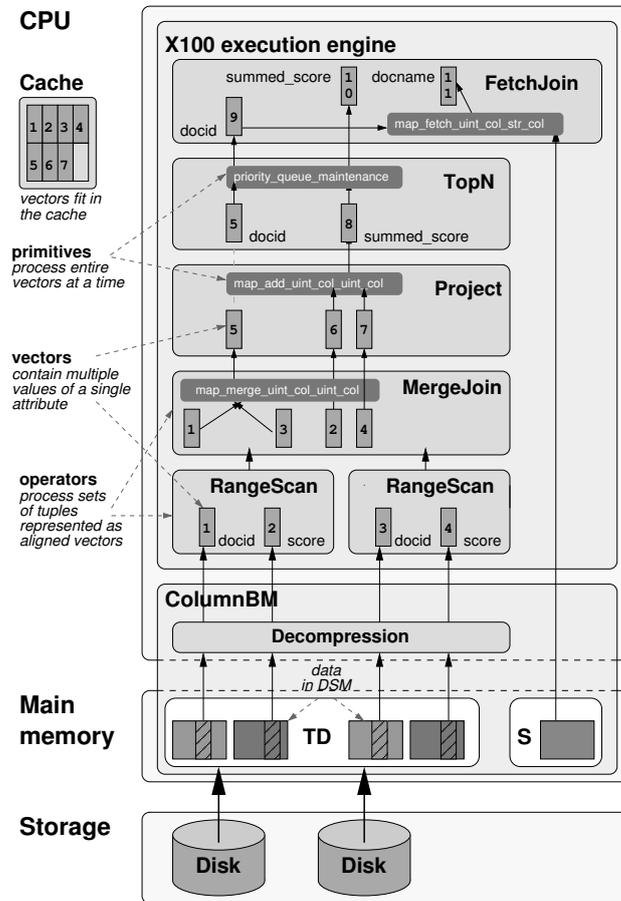


Figure 1: X100 architecture (IR query example)

are capable of delivering data at several hundreds of megabytes per second, 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 the compression algorithms PFOR and PFOR-DELTA [ZHNB06], that are designed to sacrifice some performance in terms of compression ratio, in exchange for fast decompressibility.

FOR, PFOR and PFOR-DELTA Frame Of Reference (FOR) is a database compression method that stores numerical table columns in a disk-block as the increment to a certain base value. The increments are represented in a small integer, with a fixed bit width b . Each disk-block may define a different base value and b . We recently proposed PFOR (Patched FOR), an extension of FOR [ZHNB06] that stores values as either a *code* or an *exception*. In PFOR, codes are small integer increments to a base, like in FOR. Exception values are stored in uncompressed form at the end of a disk block walking backwards. PFOR-DELTA is PFOR on the differences between subsequent column values.

PFOR and PFOR-DELTA can handle data distributions with outliers better than FOR, because they can represent outliers as exceptions, allowing b to stay low. This makes them better suited to compress *inverted lists*, which consist of increasing integer document identifiers that contain a term (or *gaps*, the differences between subsequent identifiers). Custom-built IR systems routinely employ compression in inverted lists [WMB99]. Recently in IR there is a trend towards compression schemes that sacrifice some compression ratio for better decompression speed. Carryover-12 is a recent example

| | PFOR-DELTA | | | carryover-12 | | |
|--------------|------------|-----------|----------|--------------|-----------|----------|
| | comp ratio | comp MB/s | dec MB/s | comp ratio | comp MB/s | dec MB/s |
| INEX | 1.75 | 679 | 3053 | 2.12 | 49 | 524 |
| TREC fbis | 3.47 | 788 | 3911 | 4.26 | 98 | 740 |
| TREC fr94 | 3.12 | 682 | 3196 | 3.49 | 84 | 689 |
| TREC ft | 3.13 | 761 | 3443 | 3.47 | 84 | 704 |
| TREC latimes | 2.99 | 742 | 3289 | 3.30 | 79 | 683 |

Table 5: PFOR-DELTA vs carryover-12

of such a compression scheme [AM05b].

Table 5 shows that on a number of IR datasets, PFOR-DELTA is 5-6 times faster than Carryover-12, at a cost of 15-20% lower compression ratio. Given the bandwidth provided by modern multi-disk hardware (i.e. 300-600MB/s), and IR compression ratios of around 3, a CPU needs to achieve a decompression bandwidth of 1-2GB/s just to keep up with the disks. As the CPU needs to compute the IR ranking as well, decompression bandwidth must actually be significantly higher to prevent the IR system from being I/O bound. The main drivers behind the high speed in PFOR and PFOR-DELTA are their *vectorizable* algorithms, i.e. they decompress column values without control dependencies (if-then-else), using a technique called “patching”. By avoiding if-then-elses, modern CPUs are not slowed down by *branch-misprediction* events, and the vectorization exposes data parallelism which modern super-scalar CPUs can exploit using *loop pipelining* and *speculative execution* to reach high IPC (Instructions Per Cycle) [ZHNB06].

We enabled compression when creating the DT table in X100 and Table 4 shows that this significantly reduced its storage size (from 128 to 22 bits per tuple).

4. TREC-TB EXPERIMENTS

We now report on experiments running the TREC-TB 2005 efficiency task with our IR application on top of SRAM and X100.

For these experiments, we used a dual-CPU 3GHz Pentium Xeon (only 1 CPU used for processing) with 4GB of main memory, and a software-RAID system consisting of 12 disks. The estimated system cost is EUR 4000. In all tests we run each query twice, which allows presenting results of *cold* and *hot* runs. The *hot* run represents pure processing time for a query, while the *cold* run gives insight into the overhead of fetching the data from disk. In all experiments presented in this section we used a full Terabyte TREC dataset and a subset of 1000 randomly chosen queries.

In the following, we describe a number of IR and array optimization techniques, of which Table 6 displays the successive results.

4.1 Basic BM25 Query

The array-query presented in Section 3.1 for the BM25 retrieval model results in the following physical query plan for the X100 database system:¹

```

TopN(
  DenseAggr(
    Project(
      FetchJoin(
        FetchJoin(
          MergeJoin(Scan(Q), TD, TD.termid = Q.termid),
            F, F.termid = Q.termid),
          S, S.docid = TD.docid),
        [ S.docid, scores = BM25(TD.tf,S.doclen,F.ftd)]),
      [ score = sum(scores) ]),
    [ score DESC ], 20)

```

¹For clarity, we simplify these relational plans to take just the top-N on the relation representing the sparse array. Rule (TOPN-DESC) would add some elaborate (yet cheap) operators to take care of default values entering a top-N.

Here, the computation of partial scores, as in (3.2), has been replaced by the macro `BM25()` for sake of clarity.

The above plan can be viewed as a logical relational plan by substituting `DenseAggr` for `Aggr`, and `MergeJoin` and `FetchJoin` for `Join`.

Since `termid` is the first attribute of the primary key, it is ordered, the join with the (tiny) query table `Q`. can be handled with a `MergeJoin`. This join has a sequential access pattern and only touches those disk blocks actually needed; thus is quite fast. The fetch-joins perform a foreign key-join with an identity column (both `F.termid` and `S.docid` are of type `#rownum`) of the small, memory-resident, tables `S` and `F`. This particular kind of join is especially efficiently implemented in X100 as a pointer-based lookup.

Some additional performance analysis showed that of the 1.58 seconds this query takes (see Table 6) in the hot run, almost 1.35 are spent in aggregation. Dense aggregation can be applied if the `GROUP-BY` is a single cardinal attribute with a known domain size. In this case, `docid` indeed is a cardinal between 0 and 25M. Internally, dense aggregation creates an in-memory array, used to keep all aggregate totals, that can be accessed by position. Even though the minor ordering of the `TD` table on `docid` will cause “nice” sequential passes over this array for each term in the query, the required initialization of all 25M floating point aggregate totals to a value of 0.0, and the processing of this 25M result by the top-N operator, makes dense aggregation a costly operator.

We also experimented with hash aggregation, but doing so made the queries twice slower. In the TREC-TB queries, hash aggregation inserts on average 1.4M documents (from the total 25M) into a hash table, and each aggregate value is processed 3 to 4 times (i.e. as many terms as a query has). The inefficiency of the hash aggregation is explainable as relational implementations of this operator tend to be optimized for the inverse usage pattern (i.e. updating few aggregate values many times). For this reason, we stuck to dense aggregation.

4.2 Aggregate Unfolding

Consider the following SRAM query, where terms are enumerated explicitly, and dereferenced in an aggregate:

```
Q := [ 10 42 ]
s(d) = sum( [ w(d,Q(t)) | t ] )
```

The very low cardinality of array Q_t (i.e. few query terms), together with the availability of explicit query term id’s in the SRAM expression, make the array optimizer consider an alternative plan, where the summation over the query terms is *unfolded* to a series of additions:

```
s(d) = w(d,10) + w(d,42)
```

Apart from avoiding the aggregate and transforming it into a computation (`Project`), this approach has the additional benefit that the join between `TD` and the query `Q` is transformed into $|Q| - 1$ `MergeOuterJoins`. What is more, because `TD` has `termid` as the major column of its primary key, all selected tuples appear consecutively and the RDBMS can use an efficient clustered `RangeSelect` operator per term.

```
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)
```

Because the addition $w(d,10) + w(d,42)$ represents a $Map(+, A, B)$ operation between shape-aligned arrays, Rule (ALIGNED-MAP) could join the corresponding tables by means of an `MergeOuterJoin` operation.

Table 6: SRAM and X100 on TREC-TB

| Run name (+ added feature) | p@20 | avg | |
|---|-------|------------|-----------|
| | | ms cold | ms hot |
| BM25 | 0.546 | 1826 | 1584 |
| BM25U (+Unfolding) | 0.546 | 440 | 342 |
| BM25UC (+Compression) | 0.546 | 431 | 351 |
| BM25UCM (+Materialization) | 0.546 | 155 | 29 |
| BM25UCMQ (+Quantization) | 0.543 | 118 | 28 |
| BM25UCMQC (+Conjunctive) | 0.538 | 112 | 12 |
| BM25UCMQC2 (+2-Pass) | 0.547 | 125 | 19 |
| 12-disk 3GHz Pentium4 4GB RAM server | | | |
| BM25UCMQC2D (+Distributed 8-way) | 0.547 | - | 11.3 |
| BM25UCMQC2D8 (+8 Streams) | 0.547 | - | 3.2 |
| 8 dual-core 2GHz AthlonX2 2GB RAM workstations | | | |

Table 6 clearly shows the effect of eliminating the aggregation from the query plan, which accounted for 1.35 seconds both cold and hot: performance improves to 440 ms (cold) and 342 ms (hot).

The query plan obtained by unfolding the sum aggregate coincides with the well known *Document-At-A-Time* (DAAT) IR processing model [TJ95]. It is interesting to note that the system does not need to be explicitly instructed to implement such a strategy. It comes as the result of a more generic rewriting rule, which can be applied to a variety of patterns that do not necessarily find an equivalent in the IR application domain. The equivalence with the DAAT processing model is completed by the observation that the clustered storage of arrays discussed in Section 2.3 makes the *TD* array act as an inverted list for this processing model.

4.3 Compression, Materialization and Quantization

Both in the BM25 as well as in the BM25U run, we used uncompressed columns. Thus, for these experiments, the last column of Table 4 should be ignored, as the data types `int` and `long` occupy 32- and 64-bits per value, respectively. When instead using the compressed version of the TREC-TB dataset as described in Table 4, the width of the *TD* table is reduced from 128 to 36.11 bits.

Table 6 shows, however, that using the compressed *DT* table does not significantly improve the cold run, and even makes the hot run slightly slower (due to decompression cost). The reason is that the unfolded query is fully computation-bound on the BM25 expression. Thus, when we instead use the materialized $DT_{D,T}^{\omega}$ with partial BM25 scores, we see a significant performance enhancement. With materialization, there is again a clear (126ms) difference between the hot and cold run. Now we can better appreciate compression, as the original BM25 run, that touched the same data, had a difference between hot and cold (i.e. the I/O time) of 242ms.

One drawback of the materialized representation is that `TD.score` is a 32-bits floating-point, which is not compressible. As X100 stores relations in a column-wise fashion, only those columns that are used must be read from disk.² Thus, for the compressed queries, X100 must load `TD.termid`, `TD.docid` and `TD.score`, for a total of 36.11 bits per tuple.

Quantization substitutes `TD.score` for the 8-bits `TD.scoreQ`, which reduces the tuple size to 22.11 bits. This size reduction saves another 37 ms in the cold run, bringing it down to 118 ms. One should note that quantization slightly reduces the precision.

4.4 Conjunctive and 2-Pass

In Section 3.1, we described a conjunctive strategy that multiplied the document scores with a boolean (0,1) constant that is only 1 if that document contains all terms:

²Column-wise storage also has disadvantages, as it increases the number of I/O requests (by the amount of columns). In the future, we will use the PAX [ADHS01] storage scheme to combine the benefits of column- and row-wise storage.

Table 7: 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 |
| zetdir | 0.5410 | 1 | 231 |

$$s(d) = \text{prod} ([w(d, Q(t)) > 0 \mid t]) \\ * \text{sum} ([w(d, Q(t)) \mid t])$$

Although this expression is a superset of the one for the normal BM25 query, we describe here a number of optimizations that make this formula much faster to compute. By unfolding both the aggregates, as explained in the previous section, the expression above can be rewritten as:

$$s(d) = (w(d, \$t1) > 0) * (w(d, \$t2) > 0) * \dots \\ * (w(d, \$t1) + w(d, \$t2) + \dots)$$

The arithmetic optimization and rules (MAP), (EMPTY- π), and (EMPTY- \tilde{A}) translate the multiplications in the expression above to projections on top of join operators. In contrast, for the additions in the second line, according to Rule (ALIGNED-MAP), full outer join operators are needed.

If W^{tj} denotes the relation corresponding to each partial score $w(d, \$tj)$, the join sequence for a 3-term expression becomes as follows (ignoring projections):

$$(W^{t1} \bowtie_d W^{t2} \bowtie_d W^{t3}) \bowtie_d (W^{t1} \text{=} \bowtie \text{=} _d W^{t2} \text{=} \bowtie \text{=} _d W^{t3})$$

Notice that all join and outer-join equality conditions are on the attribute d , which is the only dimension of the corresponding array, thus the key attributes for all the relations W^{tj} . This allows the simplification of outer-join operators to join operators [GLR97], as all the NULL-padded tuples produced by outer-joins will be subsequently discarded by the joins on the same key:

$$(W^{t1} \bowtie_d W^{t2} \bowtie_d W^{t3}) \bowtie_d (W^{t1} \bowtie_d W^{t2} \bowtie_d W^{t3})$$

The same conditions allow to verify that the expression above contains redundant key equi-join operations. Standard join elimination techniques [CGM90] reduce it to the following expression:

$$W^{t1} \bowtie_d W^{t2} \bowtie_d W^{t3}$$

Finally, standard join order optimization [SMK93] can schedule the execution of the most selective join operations first, to reduce as soon as possible the cardinality of intermediate results. In IR words, this means computing the score of the less frequent terms first, exploiting the typical skew of the Zipfian term distribution to discard document candidates early.

Replacing the outer-joins by joins in the order of cardinality (i.e. term frequency) has as effect that the number of candidate documents quickly decreases. Whereas the top-N in the default plan chooses among 1.4M documents on average, in the conjunctive case the average amount of candidates is down to 62K, such that the query executes much quicker in the hot run (from 28 to 12 ms). The number of candidates still varies widely, and in 15% of the TREC-TB queries, the conjunctive strategy yields in fact *less* than the 20 documents required for P@20. This explains the deterioration in precision to .53, visible in Table 6. This deterioration is mitigated using the 2-pass strategy, settling our cold run at 125 ms and the hot run at 19 ms.

4.5 Distributed Experiments

For the distributed experiments, we used our LAN with 8 workstations, all dual-core 2GHz Athlon64X2 CPUs with 2GB RAM and two 200GB SATA disks. These Linux PCs (each worth around \$800), are slightly slower than the 3GHz Xeon server in the hot runs: it averages 23 ms (vs. 19). The main advantage of the distributed setup is that only the hot run matters: thanks to database compression, the entire dataset (9GB) fits in the combined RAM buffer pool of the 8 PCs and I/O is eliminated as a performance factor.

However, as our results show, the *speedup* with 8 machines is far from perfect, (it decreases from 23 only to 11 ms). This is caused by a quite significant variance in server response times. With 8 servers, the slowest one, which determines the overall query latency, takes twice as long as the fastest (11 vs. 5.5 ms). 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. In the TeraByte TREC efficiency task of 2006, this is mimicked by the ability to run multiple queries (“streams”) concurrently. The last line in Table 6 shows that with 8 concurrent streams 8 servers are able to process around 300 queries per second, taking an amortized 3.2 msec per query only (vs. 23 msec for one server).

4.6 Discussion

Table 7 summarizes the performance and precision of the leading systems in the 2005 efficiency run of TREC-TB. As many of these systems use BM25, the precision results obtained are quite comparable. The efficiency results are harder to compare due to differences in the hardware platforms being used. Still, when comparing our single server cold run with the best 1-CPU result (zetdir), the X100 powered system slightly leads (150 vs 230ms), and it clearly leads in the 8-way distributed case (3.2 ms vs 24ms).

X100. We see these results as proof that a read-oriented compressed column-store with strong attention paid to expression evaluation efficiency (the vectorization in X100) can achieve the same raw performance as a custom-built IR system. Thus, we do think that the extreme take in X100 on optimizing database architecture to the needs of modern hardware caters to the needs of a number of application areas where more traditional database architectures currently simply fail on raw speed, and IR is one of those areas. Sticking to the topic of database architecture, we have shown that generic light-weight database compression can enhance I/O based IR performance. On serious disk subsystems, such as our 12-disk RAID with its 350MB/s read bandwidth, the multi-GB/s decompression bandwidths of schemes like PFOR-DELTA can accelerate I/O (traditional compression methods such as lzip are too slow for this). Database compression is even more important in the distributed IR runs, as it allows a cluster of cheap machines to keep the main indices resident in the aggregate RAM of the cluster (the “Google” approach [BDH03]). When operating inside RAM, with I/O removed from the performance equation, fast decompression has an even more direct impact on overall query throughput. Just like with efficient expression evaluation thanks to vectorization, the IR application benefits from the low-level idea of compression between RAM and CPU cache boundaries transparently offered by a DBMS, without any low-level computer architecture-conscious engineering required.

SRAM. The relational DAAT query plans that perform inverted list merging, used here to achieve our highest performance results, are not new at all, but we do consider it a major success that we were able to generate these plans automatically, basically starting from the mathematical formula that defines BM25. We are convinced that the array paradigm proposed in the Matrix Framework for IR can only be implemented using *sparse* arrays (materializing arrays like *LT* or *DT* is simply infeasible on non-trivial collections). The SRAM approach of mapping its generic array expression language, onto a relational representation of sparse arrays, turned out to be a fruitful source of a large collection of query optimization strategies. This collection is by no means complete, and we are currently working on ways to incorporate strategies like max-score pruning and *Term-At-A-Time* processing. One can foresee that the rules described in this paper can accelerate a wider family of IR models, e.g. all that depend on the aggregation (summation) of a number of scores. It is our objective to verify this claim

in the near future and express and experiment with an ever wider collection of IR models and retrieval strategies, exactly coinciding with the original intention of the Matrix Framework.

5. RELATED WORK

The idea to use DBMS technology as a building block in an IR system is pursued e.g. in [GFHR97], where the authors store inverted lists in a Microsoft SQLServer and use SQL queries for keyword search. Similarly, in [GBS04] IR data is distributed over a PC cluster, and an analysis of the impact of concurrent updates is provided. Our approach arrives at similar strategies as this previous work, but rather than hand-crafting a database schema and query to one particular information retrieval model, we show how to generate these automatically from high-level array formulas. In this sense, we extend the state of the art with respect to flexibility.

In this work, we also try to push the state of the art of DB+IR efficiency. The physical DB techniques for powering our IR application and their effectiveness/performance trade-off are demonstrated here on a much larger collection (500GB TeraByte TREC vs. 500MB in [GBS04]) and show significantly faster retrieval performance, using a relatively modest hardware infrastructure. In the TREC benchmark there were a few attempts to use database technology, e.g. [MKD99]. 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 [CCS04], but its precision and speed (5 sec per query) were disappointing compared to other participants.

Our experiments show the importance of database compression for IR applications, both to reduce disk I/O as well as memory footprint, such that (in a distributed system) the entire index can become memory-resident. Several commercial database systems use compression; especially node pointer prefix compression in B-trees (e.g. supported by DB2) can reduce the size IR tables if they e.g. use (`termid,docid`) as primary key. Our PFOR extends FOR (Frame Of Reference), a light-weight database compression scheme for compressing correlated integers [GRS98]. Compression is an important topic in IR, and the superiority of inverted lists over signature files is credited to its effectiveness [WMB99]. Early IR compression work focused on exploiting the specific characteristics of gap distributions to achieve optimal compression ratio (e.g. using Huffman or Golomb coding tuned to the frequency of each particular term with a local Bernoulli model [Huf52]). More recently, attention has been paid to schemes that trade compression ratio for higher decompression speed [AM05b, Tro03], a point taken to the extreme by our PFOR-DELTA.

While IR applications are commonly developed as custom-built applications or (less commonly) implemented on top of database systems, we are not aware of previous work using the array data-model as a “gluing layer” for DB+IR. Prior work by our own group has explored use of arrays in DB+IR, using dense arrays only [vBdVK03]. The formalism proposed by the Matrix Framework for IR [RTK05] is infeasible with a dense array implementation, hence our current interest in sparse arrays. Sparse matrix operations have also been described in [KPS97] as a query optimization problem, confirming the potential of database technology for array processing, but there the objective was to compile these into a standalone program, rather than storing and querying sparse arrays in a DBMS.

There has been prior work on array databases, either by integrating arrays as first-class citizens in the relational data model, or by using an ADT/blob approach.

The array query language AQL[LMW96] has been an important contribution toward the integration of arrays in the relational model. AQL is a functional array language geared toward scientific computation. The authors show that inclusion of array support to their nested relational language entails the addition of two functions: an operator to produce aggregation functions and a generator for intervals of natural numbers. The AQuery system [LS03], targeting financial stock analysis, uses the concept of “arrables”, ordered relational tables, and an extension of SQL based on the clause `ASSUMING ORDER`. However, it only supports uni-dimensional arrays.

The ADT/blob approach has been pursued in [HM04], where an algebra for the manipulation of irregular topological structures is applied to the natural science domain. The RasDaMan DBMS is a domain-independent array database system, implemented as an abstract data type in the O_2 object

oriented DBMS [Bau99, FB99]. Its RasQL query language is a SQL/OQL like query language based on a low level array algebra.

6. CONCLUSIONS AND FUTURE WORK

We presented SRAM, an array database system that works on top of the X100 relational database engine. We described its array comprehension-based query language, and explained a number of mapping and optimization rules for translating queries on *sparse* arrays into efficient queries on relational tables. Our claim that array comprehensions are a flexible way for IR researchers to express ranking formulas, is aligned with the recently proposed Matrix Framework for IR, and is demonstrated here in case of the BM25 model in the context of the TREC TeraByte track (TREC-TB). It turns out that our mapping and optimization rules are able to automatically generate an efficient relational plan equivalent to the DAAT strategy for processing inverted lists, right from the BM25 formula. The top performance and precision on TREC-TB, that rivals custom-built IR systems, further shows that it *is* feasible to use generic DB technology for IR. The X100 relational engine originally developed for data warehousing, with its database architecture specifically designed to match modern computer architecture (i.e. taking into account CPU parallelism, caches and branch prediction) was able to provide the raw muscle normally achieved by hand-written programs (such as IR systems). Also, the feature of transparent database compression with a number of high-performance compression schemes, was a good match for the sparse array relations.

In future work, we will further extend SRAM to automate a number of tasks that were left here to the IR application, such as score materialization and distribution. We will also test SRAM on a series of retrieval tasks (text, video, XML) to see how far the successes achieved on BM25 extend to other retrieval models. As for X100, we are investigating in detail the further possibilities of various compression schemes, both in the sense of architecture-conscious performance study and optimization, as well as in the sense of extending its compression functionality towards lossy compression schemes that could be used for automatic score quantization with quality guarantees.

References

- [ADHS01] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 169–180, Rome, 2001.
- [AM05a] 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.
- [AM05b] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.
- [Bau99] P. Baumann. A database array algebra for spatio-temporal data and beyond. In *Next Generation Information Technologies and Systems*, pages 76–93, 1999.
- [BCH⁺03] 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.
- [BDH03] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, 2003.
- [BLS⁺94] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.
- [BZN05] 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.
- [CCS04] 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.
- [CGM90] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based approach to semantic query optimization. *ACM Trans. Database Syst.*, 15(2):162–207, 1990.
- [CSS05] 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.
- [DDRvdV00] James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. *Templates for the solution of algebraic eigenvalue problems: a practical guide*. Society for Industrial and

- Applied Mathematics, Philadelphia, PA, USA, 2000.
- [EMK⁺04] Andrew Eisenberg, Jim Melton, Krishna Kulkarni, Jan-Eike Michels, and Fred Zemke. Sql:2003 has been published. *SIGMOD Rec.*, 33(1):119–126, 2004.
- [FB99] P. Furtado and P. Baumann. Storage of multidimensional arrays based on arbitrary tiling. In *Proc. of the 15th International Conference on Data Engineering, ICDE99*, pages 408–489, March 1999.
- [GBS04] Torsten Grabs, Klemens Bhoem, and Hans-Jorg Schek. PowerDB-IR: scalable information retrieval and storage with a cluster of databases. *Knowledge and Information Systems*, 6(4):465–505, 2004.
- [GFHR97] 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.
- [GL83] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, Maryland, 1983.
- [GLR97] Cesar Galindo-Legaria and Arnon Rosenthal. Outerjoin simplification and re-ordering for query optimization. *ACM Trans. Database Syst.*, 22(1):43–74, 1997.
- [Gra94] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE TKDE*, 6(1):120–135, 1994.
- [GRS98] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *Proceedings of the International Conference of Data Engineering (IEEE ICDE)*, Orlando, FL, USA, 1998.
- [HM04] B. Howe and D. Maier. Algebraic manipulation of scientific datasets. In *VLDB*, pages 924–935, 2004.
- [Huf52] D.A. Huffman. A method for construction of minimum redundancy codes. In *Proc. of the IRE*, volume 40, pages 1098–1101, 1952.
- [KPS97] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. A relational approach to the compilation of sparse matrix programs. In *Euro-Par*, pages 318–327, 1997.
- [LMW96] L. Libkin, R. Machlin, and L. Wong. A query language for multidimensional arrays: Design, implementation, and optimization techniques. In *Proceedings of ACM SIGMOD International Conference on Managing Data*, pages 228–239. ACM Press, June 1996.
- [LS03] Alberto Lerner and Dennis Shasha. Aquery: Query language for ordered data, optimization techniques, and experiments. In *VLDB*, pages 345–356, 2003.
- [MKD99] 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.
- [MV93] David Maier and Bennet Vance. A call to order. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 25-28, 1993, Washington, DC*, pages 1–16. ACM Press, 1993.
- [Por80] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.
- [RTK05] T. Roelleke, T. Tsirikika, and G. Kazai. A general matrix framework for modelling information retrieval. *IP&M*, 42(1):4–30, 2005.
- [RWB98] 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.
- [SMK93] M. Steinbrunn, G. Moerkotte, and A. Kemper. Optimizing join orders. Technical report, University of Passau, 1993.
- [TJ95] H. Turtle and J.Flood. Query evaluation: strategies and optimizations. *Information*

- Processing and Management*, 31(6):831–850, 1995.
- [Tro03] Andrew Trotman. Compressing inverted files. *Inf. Retr.*, 6(1):5–19, 2003.
- [vBdVK03] A.R. van Ballegooij, A.P. de Vries, and M. L. Kersten. RAM: Array processing over a relational DBMS. Technical Report INS-R0301, CWI, 2003.
- [WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images*. Morgan Kaufmann Publishers Inc., San Francisco, USA, 1999.
- [ZHNB06] M. Zukowski, S. Heman, 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.