Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

INS

Information Systems

INformation Systems

Automatic optimization of array queries

A.R. van Ballegooij, R. Cornacchia, A.P. de Vries

REPORT INS-E0507 APRIL 2005

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization 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)

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

Automatic optimization of array queries

ABSTRACT
Non-trivial scientific applications often involve complex computations on large multi-dimensional datasets. Using relational database technology for these datasets is cumbersome since expressing the computations in terms of relational queries is difficult and time-consuming. Moreover, query optimization strategies successful in classical relational domains may not suffice when applied to the multi-dimensional array domain. The RAM (Relational Array Mapping) system hides these issues by providing a transparent mapping between the scientific problem specification and the underlying database system. This paper focuses on the RAM query optimizer which is specifically tuned to exploit the characteristics of the array paradigm. We detail how an intermediate array-algebra and several equivalence rules are used to create efficient query plans and how, with minor extensions, the optimizer can automatically parallelize array operations.

1998 ACM Computing Classification System: H.2.3 Languages; H.2.4 Systems
Keywords and Phrases: array query optimization

Automatic Optimization of Array Queries

Alex van Ballegooij
Alex.van.Ballegooij@cwi.nl

Roberto Cornacchia
R.Cornacchia@cwi.nl

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

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

ABSTRACT

Non-trivial scientific applications often involve complex computations on large multi-dimensional datasets. Using relational database technology for these datasets is cumbersome since expressing the computations in terms of relational queries is difficult and time-consuming. Moreover, query optimization strategies successful in classical relational domains may not suffice when applied to the multi-dimensional array domain. The RAM (Relational Array Mapping) system hides these issues by providing a transparent mapping between the scientific problem specification and the underlying database system. This paper focuses on the RAM query optimizer which is specifically tuned to exploit the characteristics of the array paradigm. We detail how an intermediate array-algebra and several equivalence rules are used to create efficient query plans and how, with minor extensions, the optimizer can automatically parallelize array operations.

1. Introduction

Efficiently managing collections of e.g. scientific models or multimedia archives, is beyond the capabilities of most database systems since indexing and retrieval functionality are tuned to completely different workloads. Existing database systems support the relational model and they do not support multidimensional arrays as a first class citizen. Maier and Vance have argued for long that the mismatch of data models is the major obstacle for the deployment of relational database technology in computation oriented domains (such as multimedia analysis) [MV93]. While storage of multidimensional objects in relations is possible, it makes data access awkward and provides little support for the abstractions of multidimensional data and coordinate systems. Support for the array data model is a prerequisite for an environment suitable for computation oriented applications.

The RAM (Relational Array Mapping) system [vB04] bridges the gap caused by the mismatch in data-models with a mapping layer between them. It provides the user with an array oriented query language and data-model, which are internally mapped onto relational queries over relations representing arrays. This way data storage and query evaluation can be delegated to an existing database system.

Past experience with the implementation of multimedia retrieval and analysis in a database setting, see [Nes01], has proved the potential value of relational bulk processing for multimedia analysis. Similarly, given an effective query optimizer, the RAM system has shown the potential to rival the performance of specialized solutions [CvBdV04]. However, with large collections, the amount of computations required for a single query is prohibitive for querying to be interactive. This calls for the parallel query evaluation, fortunately the structured nature of the array paradigm helps with the automatic distribution of complex queries.

This paper presents the basic equivalence rules utilized by the RAM query optimizer and a discussion on extending the optimizer for automatic query distribution.

2. The RAM System

The RAM system adds multidimensional array structures to existing database systems by internally mapping array structures and operations to relations and relational queries respectively. Storing the

Table 1: Basic Array Operations

Operation	**Meaning**
$const(\mathcal{S}, c)$	$[c\|\bar{\imath} < \mathcal{S}]$
$grid(\mathcal{S}, j)$	$[i_j\|\bar{\imath} < \mathcal{S}]$
$map(f, A1, \ldots, Ak)$	$[f(A1(\bar{\imath}), \ldots, A(\bar{\imath}))\|\bar{\imath} < \mathcal{S}_A]$
$apply(A, I1, \ldots, Ik)$	$[A(I1(\bar{\imath}), \ldots, Ik(\bar{\imath}))\|\bar{\imath} < \mathcal{S}_I]$
$choice(C, A, B)$	$[if(C(\bar{\imath}))\ \ then\ \ A(\bar{\imath})\ \ else\ \ B(\bar{\imath})\|\bar{\imath} < \mathcal{S}_C]$
$aggregate(g, j, A)$	$[g([A(x_0, \ldots, x_{j-1}, i_j, \ldots, i_{n-1})\|x_0, \ldots, x_{j-1}])\|i_j, \ldots, i_{n-1}]$, where $n = \|A\|$
$concat(A, B)$	$A + +B$

array data as relations instead of a proprietary data structure allows the full spectrum of relational operations to be applied to the array data. This indirectly guarantees complete query and data-management functionalities, and, since the array extensions naturally blend in with existing database functionalities, the RAM front-end can focus solely on problems inherent to the array domain.

The use of relational mapping is a distinguishing aspect of RAM as most array oriented database efforts rely on proprietary data-structures. For example, the AQL language depends on a custom prototype back-end [LMW96], and languages such as AML [MS97], designed for image processing, and AQuery [LS03], designed for ordered data in the business domain, are realized by stand-alone applications. A notable exception is the RasDaMan system [Bau99], which does use proprietary data structures, but is implemented as an extension module for an object oriented database system.

2.1 Query Language

The RAM system provides an array database query language based on comprehension syntax (see [BLS+94]), inspired by the AQL language detailed in [LMW96]. Array comprehensions allow users to specify new arrays by declaring its dimensions and a function to compute the value for each of its cells. The array constructor has the following form:

$$A = [f(i_0, \ldots, i_{(n-1)})|i_0 < S_A^0, \ldots, i_{(n-1)} < S_A^{(n-1)}],$$

which specifies an array A with shape $\mathcal{S}_A$.

An n-dimensional array is defined by specifying its shape $\mathcal{S}_\mathcal{A}$ and associating its indexes $\bar{\imath} = (i_0, \ldots, i_{n-1})$ with their cell values $f(\bar{\imath})$. Function f may apply the operators defined on the base type in the database layer to values indexed in previously defined arrays, the index values themselves and constant values. The array indexes are defined as consecutive ranges of natural numbers starting from 0, hence the shape of the array is defined completely by giving its *index generators*, $i_j < S_A^j$: $\{i_j|i_j \in \mathbb{N}_0, i_j < S_A^j\}$.

The semantics of the comprehension syntax are simple. For example, expression $[x + 10 * y|x < 3, y < 3]$ defines an array with shape $[3, 3]$, binds the (result of) function $x + 10 \cdot y$ to each of its cells, and thus results in the following array:

0	1	2
10	11	12
20	21	22

2.2 RAM Array Algebra

The RAM front-end translates the high level comprehension style queries into an intermediate array-algebra before the final transformation to the relational domain.

The RAM array algebra consists of 6 basic operators summarized in Table 1. In addition a separate concatenation operator exists. This operator is superfluous, but its common occurrence in queries

warrants an efficient, direct, mapping to algebra for this operator.

The first two operators generate new arrays given a shape. The *const* operator fills a new array with a constant value, whereas the *grid* operator creates an array with numbers taken from its index values. The next pair of operators deals with function application. The *map* operator applies a function to the elements of one or more arrays, whereas the *apply* operator applies an array, essentially a stored function, to arrays of indexes. Finally, the *choice* operator allows elements from two arrays to be merged based on a boolean condition and the *aggregate* operator applies an aggregation function over axes of an array.

Translation of array comprehensions into the algebraic form is achieved by substitution of the patterns detailed in Table 1 with their algebraic equivalent. Using these equivalences complex comprehensions can be translated:

Example 1 *(Translating array comprehension)* Consider the following array comprehension expression:

$$[f(i_j, c)|\bar{\imath} < \mathcal{S}].$$

This comprehension can be decomposed into three elementary parts, the use of an axis variable, a constant value, and a function application: these correspond to the *grid*, *const*, and *map* operators respectively.

$$\begin{array}{lcll} A & = & [i_j|\bar{\imath} < \mathcal{S}] & \longmapsto \quad grid(\mathcal{S}, j) \\ B & = & [c|\bar{\imath} < \mathcal{S}] & \longmapsto \quad const(\mathcal{S}, c) \\ & & [f(A, B)|\bar{\imath} < \mathcal{S}] & \longmapsto \quad map(f, A, B) \end{array}$$

These translation patterns lead to this algebraic expression:

$$map(f, grid(\mathcal{S}, j), const(\mathcal{S}, c)).$$

For further details on the RAM data-model, query language, and translation see [vBdVK03].

3. Optimizer

Optimization techniques typically focus on those issues and patterns inherent to the expected query load. Array processing results in patterns significantly different from typical relational queries. Also, the process of mapping array queries into the foreign relational domain results in an inevitable loss of context information.

Example 2 *(Loss of context)* Consider example query

$$[f(A(x))|x],$$

that specifies an array whose values correspond to the function f applied to the values in array A. The system translates it into the following array algebra expression:

$$map(f, apply(A, grid(\mathcal{S}_A, 0))).$$

By representing array as sets of tuples consisting of *index/value* pairs (i, v), this can subsequently mapped to the following relational query:

$$\pi_{I.i, f(A.v)}(A \bowtie_{A.i=I.v} (I = grid(\mathcal{S}_A, 0))),$$

Many properties of the data can be communicated effectively to a relational system: for example, it is known that the index-columns of an array-relation are key, and that the values in array I have

a foreign key relation to the index values of array A [1]. Nevertheless, some contextual information is inevitably lost.

In this case, the property information is insufficient for the relational system to discard the join operation. The array algebra expression however, is easily recognized as an identity transformation of array A and can immediately be reduced to:

$$map(f, A),$$

which maps to this relational query:

$$\pi_{A.i,f(A.v)}(A).$$

This loss of context, caused by relational mapping, makes that relational engines may perform sub-optimally on non-relational domains. A specialized query optimizer can remedy this problem by exploiting domain-specific knowledge to produce a better relational query.

In the relational domain the selectivity of a particular operation is often difficult to estimate reliably, this results in complex, expensive, and, inaccurate cost-models. In contrast, for arrays, the exact dimensionality and size of each intermediate result is cheap to compute. This provides an array specific optimizer with exact statistics about alternative query plans that are cheap to determine.

The RAM query optimizer operates natively in the array domain at the internal array-algebra level. A set of equivalence rules allow it to reformulate the original query and generate alternative query plans. In addition, several simple heuristics direct the rewriting by indicating the applicability of individual rules in specific situations. Ultimately, the cheapest among alternative query plans is chosen by a cost-model.

3.1 Optimizing array queries

The most basic equivalence rules provided to the RAM optimizer deal with the special case of constant arrays. For example, a function performed over an array with constant values can be performed just once over the constant value:

Equivalence 1

$$map(f, const(\mathcal{S}, c)) \rightsquigarrow const(\mathcal{S}, f(c))$$

Another example is the application of a constant array to a set of indexes:

Equivalence 2

$$apply(const(\mathcal{S}, c), I1, \ldots, Ik) \rightsquigarrow const(\mathcal{S}_{I1}, c)$$

A similar result can be obtained for constant transformations. The result of an identity transformation is by definition identical to the original. Many identity transformations are easily identified – for example persistent arrays, used in index-expressions, complicate detection – and such array applications can be removed:

Equivalence 3

$$apply(A, I1, \ldots, Ik) \rightsquigarrow A$$
$$\text{when}$$
$$k \equiv |\mathcal{S}_A|, I1 \equiv grid(\mathcal{S}_A, 0), \ldots, Ik \equiv grid(\mathcal{S}_A, k-1)$$

[1] Many more properties are known, such as the ranges of index values.

In the relational domain, it is often beneficial to 'push selections down' through a query expression tree to reduce data volume as soon as possible. Similar reasoning leads to the following rule in the array domain:

Equivalence 4

$$apply(map(f, A), I1, \ldots, Ik)$$
$$\leftrightsquigarrow$$
$$map(f, apply(A, I1, \ldots, Ik))$$

Notice that this rule is bi-directional: it allows for the *apply* operator to be pushed both up and down through other function applications. In general one can reason that if $size(A) > size(I)$ it is beneficial to push down (apply the rule from left to right) and vice versa.

An interesting aspect of array application is the fact that arrays are stored functions. For any array expression applied to indexes it is possible to perform the application – evaluate the array function – directly through substitution:

Equivalence 5

$$apply(A, I) \rightsquigarrow A'$$
where any $grid(S_A, i)$ in A' is substituted by I_i

Example 3 *(Substitution)* In the following expression,

$$[[f(z)|z < 4](x + y)|x < 3, y < 3]$$

we could substitute the index used in the array-application and obtain the result directly:

$$[f(x + y)|x < 3, y < 3].$$

The inverse of this can be used to reduce array expressions that are constant along some of the axes it is defined for. Suppose there is an array (sub-)expression which is independent of some of the axes in this shape. The expression can be evaluated over only the axes it depends on and extended afterward to the desired shape:

Equivalence 6

$$A \rightsquigarrow apply(A', I)$$
where
$$\mathcal{S}_I = \mathcal{S}_A, |A'| < |A|, apply(A', I) \equiv A$$

Example 4 *(Dependencies)* The calculus expression

$$[f(x)|x < 3, y < 4]$$

defines an array of shape $[3, 4]$, however the values depend only on the x axis. If f is an expensive function, it may be cheaper to re-formulate the query as follows:

$$[[f(x)|x < 3](x)|x < 3, y < 4],$$

where f is first evaluated for all values of x and subsequently duplicated for all values of y.

The RAM system has no direct control over low-level details such as memory usage, which depends on the specific execution strategies decided by the relational back-end system. Nevertheless, the way in which the query is formulated can assist the back-end system in formulating an efficient execution plan. Intermediate results can require the relational back-end to materialize big tables, posing severe memory management issues. Fortunately, predictable access patterns in array queries offer opportunities for rewriting rules that allow for management of system resources. The evaluation of aggregation functions turned out to be critical with respect to maximum memory usage. The following equivalence can be (repeatedly) applied to any commutative and associative aggregate [2]:

Equivalence 7

$$\begin{gathered} aggregate(\sum, A) \\ \rightsquigarrow \\ map(+, aggregate(\sum, A_1), aggregate(\sum, A_2)) \\ \text{where } concat(A_1, A_2) \equiv A \end{gathered}$$

The importance of the transformations being driven by heuristics is clear in this equivalence: as it does not reduce the size of the intermediate result, it would not be applied if only the cost-model were taken into account.

3.2 Distributing array queries

In the field of high performance computing array computations are usually captured in complex algorithms carefully designed to exploit parallelization. This is viable as the complexity of the operations provides enough work to supply multiple CPUs with a sufficient workload in-between the inevitable data exchange operations.

RAM queries are composed of many primitive operators, that are too simple to warrant a parallelized implementation: the amount of work represented by a single operator is too small for the benefits of distributed evaluation to outweigh communication overhead. However, the workload generated by a complex query is vast enough to consider distributed evaluation at a higher granularity. This results in parallelism through the concurrent evaluation of a number of queries each formulated to produce a part of the complete result.

Distribution of RAM queries over multiple machines involves discovery of a suitable location in the query plan to split it into disjunct sub queries that can be executed in parallel. In an algebra disjunct sub-expressions are by definition independent: in the expression $f(E_A, E_B)$ sub-expressions E_A and E_B have no side effects and can potentially be evaluated in parallel. Any operator, with multiple arguments, is an opportunity to split the query and parallelize sub queries.

However, when simply using those opportunities readily available in an existing query plan it is hard to achieve a balanced query load across nodes: it is rare to find sub expressions that are equally expensive to compute. Fortunately, the structured nature of array queries allows the creation of new, balanced opportunities to split the query for distribution.

A straightforward approach to distribute a query over multiple nodes is to fragment the result space in disjunct segments and compute each of those parts individually. This approach is simply mimicked in RAM, generating a series of queries that each yield a specific fragment, and concatenating those resulting fragments to produce a single result:

Equivalence 8

$$\begin{gathered} map(f, A) \\ \leftrightsquigarrow \\ concat(map(f, A_1), map(f, A_2)) \\ \text{where } concat(A_1, A_2) \equiv A \end{gathered}$$

[2] Equivalence 7 deals with the summation, rules for other commutative and associative aggregates are similar.

Aggregations are also a suitable operation for the creation of balanced sibling sub queries. Equivalence 7 shows how an aggregate can be split into fragments to be combined afterward. Again, a new opportunity for balanced query distribution is introduced.

Rewriting the query plan like this introduces a new operator in the query, which represents a new opportunity to split the query for distribution. Moreover, since the size of the various fragments created can be controlled it is possible to ensure the costs are balanced.

The RAM optimizer is easily extended to include distribution of fragmented queries. The *distribute* pseudo-operator distributes its arguments (sub-queries) over multiple machines and collects the results:

Equivalence 9

$$\begin{array}{lcl} map(f, A_1, A_2) & \leadsto & map(f, distribute(A_1, A_2)) \\ concat(A_1, A_2) & \leadsto & concat(distribute(A_1, A_2)) \end{array}$$

The term pseudo-operator is used to indicate that it does not operate on the data, instead it manipulates the query execution itself. Notice that it performs a role similar to that of the *exchange* operator introduced in the classical Volcano system [Gra94]. Balanced sub-queries can be created using rules as Equivalence 7 and Equivalence 8.

3.3 Estimating Query Cost

The query optimizer uses a cost-model to determine the best (cheapest) alternative of the query plans it produces with the equivalence rules.

The RAM cost-model is simple, it estimates the cost of an array expression based on the sum of all intermediate result sizes. The underlying assumption of this cost-model is that the total volume of data processed is the dominant factor in overall costs, not the type of processing required.

The cost-model computes a score for an expression by recursively adding the size of all intermediates produced by sub-expressions of an operator to the size of its own result.

Example 5 *(Cost function)* The cost of this expression:

$$map(+, grid(\mathcal{S}, 0), const(\mathcal{S}, 1))$$

is computed as follows. The *map* produces an array of shape $\mathcal{S}$ thereby inducing a cost of $|\mathcal{S}|$. Both arguments of the *map* operator, the sub-expressions $grid(\ldots)$ and $const(\ldots)$, produce intermediate results, in this case also of shape $\mathcal{S}$. Thus the total estimated cost for the expression is $3|\mathcal{S}|$.

There is one exception, the *distribute* pseudo-operator. The *distribute* pseudo-operator gets assigned only the maximum cost among its children, as they are evaluated in parallel, and an additional a cost factor related to the data volume to be communicated.

4. Summary and Discussion

We presented a short overview of the RAM system. The paper focuses on the RAM query optimizer and in particular on the equivalence rules used to rewrite query plans. The effectiveness of various optimizations are convincingly shown in [CvBdV04]. In this case study, we showed that the optimized RAM query plan performed on-par with the custom-built baseline application, while the naive query plan, with all the optimizations disabled, suffered a performance gap of factor 16.

The distribution strategies presented are based on the assumption that each node has full access to the complete data set to allow a focus on problems inherent to query distribution without data-placement issues. Preliminary experiments in this setting on the parallelization of array queries indicate that the RAM system can produce effective parallel query plans. In those cases examined a

near-linear speed-up is achieved by automatically distributing a RAM query plan over a small cluster of machines.

Open issues that remain for future work include the identification of more transformation rules, the validation of the cost-model and the development of more advanced tactics for reducing the search space of the optimizer.

References

[Bau99] P. Baumann. A database array algebra for spatio-temporal data and beyond. In *Next Generation Information Technologies and Systems*, pages 76–93, 1999.

[BLS+94] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[CvBdV04] R. Cornacchia, A.R. van Ballegooij, and A.P. de Vries. A case study on array query optimisation. In *First International Workshop on Computer Vision meets Databases (CVDB 2004)*, pages 3–10, Maison de la Chimie, Paris, France, June 2004. ACM Press. In cooperation with ACM SIGMOD.

[Gra94] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[LMW96] L. Libkin, R. Machlin, and L. Wong. A query language for multidimensional arrays: Design, implementation, and optimization techniques. In *ACM SIGMOD 1996*, pages 228–239. ACM Press, June 1996.

[LS03] Alberto Lerner and Dennis Shasha. AQuery: Query Language for Ordered Data. In *Proceedings of the International Conference on Very Large Databases VLDB*, pages 345–356, 2003.

[MS97] A.P. Marathe and K. Salem. A language for manipulating arrays. In *Proceedings of the 23rd VLDB Conference*, pages 46–55, 1997.

[MV93] D. Maier and B. Vance. A call to order. In *Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems*, pages 1–16. ACM Press, 1993.

[Nes01] N. Nes. *Image Database Management Systems - Design Conciderations, Algorithms and Architecture*. PhD thesis, University of Amsterdam, December 2001.

[vB04] A.R. van Ballegooij. RAM: A Multidimensional Array DBMS. In *Proceedings of the ICDE/EDBT 2004 Joint Ph.D. Workshop*, pages 169–174, 2004.

[vBdVK03] A.R. van Ballegooij, A.P. de Vries, and M. Kersten. Ram: Array processing over a relational dbms. Technical Report INS-R0301, CWI, March 2003.