Distribution Rules for Array Database Queries

Alex van Ballegooij, Roberto Cornacchia, Arjen P. de Vries,
and Martin Kersten

CWI, INS1, Amsterdam, The Netherlands
{Alex.van.Ballegooij, R.Cornacchia, Arjen.de.Vries,
Martin.Kersten}@cwi.nl

Abstract. Non-trivial retrieval applications involve complex computations on large multi-dimensional datasets. These should, in principle, benefit from the use of relational database technology. However, expressing such problems in terms of relational queries is difficult and time-consuming. Even more discouraging is the efficiency issue: 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 difficulties by providing a transparent mapping between the scientific problem specification and the underlying database system. In addition, its optimizer is specifically tuned to exploit the characteristics of the array paradigm and to allow for automatic balanced work-load distribution. Using an example taken from the multimedia domain, this paper shows how a distributed real-word application can be efficiently implemented, using the RAM system, without user intervention.

1 Introduction

Efficiently managing collections of e.g. images or scientific models, is beyond the capabilities of most database systems, since indexing and retrieval functionality are tuned to completely different workloads. Relational database systems 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) [1]. 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 bridges the gap caused by the mismatch in data-models between problems in ordered domains and relational databases, by providing a mapping layer between them. This approach has shown the potential to rival the performance of specialized solutions given an effective query optimizer [2]. This paper investigates how to improve performance by distributing the query over a cluster of machines. We hypothesize that the array

K.V. Andersen, J. Debenham, and R. Wagner (Eds.): DEXA 2005, LNCS 3588, pp. 55–64, 2005.

paradigm allows RAM to automatically distribute the evaluation of complex queries. Extending the case study presented in [2], we detail how a non-trivial video retrieval application has been moved from a stand-alone application to a distributed database application, using the RAM system for automatic query-distribution. The execution time of the ranking process has been significantly improved without requiring manual optimizations of query processing.

2 The RAM Approach

The RAM system is a prototype array database system designed to make database technology useful for complex computational tasks [3]. Its innovation is the addition of multidimensional array structures to existing database systems by internally mapping the array structures to relations. Array operations are also mapped to the relational domain, by expressing them as relational queries. This contrasts with earlier array database systems; for example, array processing in RasDaMan was implemented as an extension module for an object oriented database system [4]. By storing array data as relations instead of a proprietary data structure, the full spectrum of relational operations can be performed on that 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 RAM system provides a comprehension based array-query language, to express array-specific queries concisely (comprehension syntax is explained in [5]). Array comprehensions allow users to specify a new array by declaring its dimensions and a function to compute the value for each of its cells. The RAM query language is inspired by the language described in the AQL proposal [6]. Due to space limitations we omit a detailed discussion of the RAM syntax, which can be found in [7], but its basic construct is the comprehension: $[f(x, y)|x < 2, y < 3]$. This defines an array of shape 2×3, with axes named x and y, where the value of each cell in the array is determined by an expression over its axes: $f(x, y)$. Support for this language is isolated in a front-end that communicates with the DBMS by issuing relational queries.

The RAM front-end translates the high level comprehension style queries into an intermediate array-algebra before the final transformation to the relational domain. This intermediate language is used by a traditional cost driven rule based optimizer specifically geared toward optimization of array queries. The RAM optimizer searches for an optimal query plan through the application of equivalence rules. A second translation step expresses the array algebra plan in the native query language of the database system. This can be not only a standard query language (SQL), but also a proprietary query language, such as the MonetDB Interface Language (MIL) for MonetDB [8], the database system used for the experiments in this paper.

3 Case Study

The case study used in this paper is a RAM based implementation of the probabilistic retrieval system that our research group developed for the search task of TRECVID 2002-2004 [9].

In this retrieval system, the relevance of a collection image m given a query image is approximated by the ability of its probabilistic model ω_m (a mixture of Gaussian distributions) to describe the regions $\mathcal{X} = (\boldsymbol{x_1}, \ldots, \boldsymbol{x_{N_s}})$ that compose the query image. Such a score $P(\boldsymbol{\mathcal{X}}|\omega_m)$ is computed using the following formulas (the interested reader is referred to [10] for more details):

$$P(\boldsymbol{x_s}|\omega_m) = \sum_{c=1}^{N_c} P(C_{c,m}) \frac{1}{\sqrt{(2\pi)^{N_n} \prod_{n=1}^{N_n} \sigma_n^2}} e^{-\frac{1}{2}\sum_{n=1}^{N_n} \frac{(x_n-\mu_n)^2}{\sigma_n^2}}. \quad (1)$$

$$P(\boldsymbol{x_s}) = \sum_{\omega_m=1}^{N_m} P(\omega_m) * P(\boldsymbol{x_s}|\omega_m). \quad (2)$$

$$P(\mathcal{X}|\omega_m) = \sum_{s=1}^{N_s} log(\lambda P(\boldsymbol{x_s}|\omega_m) + (1-\lambda)P(\boldsymbol{x_s})). \quad (3)$$

These mathematical formulas map nicely to concise RAM expressions:

Expression 1

```
p(s,m) = sum([P(c,m) * (1.0/(sqrt(pow(2*PI,Nn))*prod([S2(n,c,m)|n<Nn])))
    * exp(-0.5 * sum([pow(Q(n,s)-Mu(n,c,m),2)/S2(n,c,m)|n<Nn])) | c<Nc])
p(s)   = (1 / Nm) * sum([ p(s,m) | m<Nm ])
Scores = [ sum( [ log( l*p(s,m)+ (1-l)*p(s) ) | s<Ns ] ) | m<Nm ]
```

For a more in depth discussion of this implementation of the retrieval system in RAM, the interested reader is referred to [2].

Comparison between Equations 1, 2, 3, and RAM Expression 1 clearly shows that the mathematical description of the ranking problem maps almost 1-on-1 to RAM. We postulate that the RAM query language, thanks to its array based data model, remedies many of the *interfacing hurdles* encountered when implementing computation oriented algorithms in a database system. However, the retrieval problem inherently deals with complex computations and large volumes of data. This means that even though the efficiency of the RAM query evaluation is competitive with the native Matlab [11] implementation of the retrieval algorithm, the costly computations make it infeasible to use the system in an interactive setting.

4 Distributing Array Queries

The inherent parallelism of distributed systems may be exploited to overcome the computational limitations of a single machine. By distributing queries over

multiple nodes, more complex queries than normally possible can be evaluated, having each node compute only part of the answer. Query distribution for relational databases is well studied and offers more than just efficiency (distributed database technology is discussed for example in [12] and [13]). In case of the RAM array database system however, the primary concern is to speed-up query evaluation.

This paper investigates the distribution of RAM queries over multiple computational nodes by discovering a suitable location in the query plan to split it into disjunct sub-queries. When simply using those opportunities readily available in an existing query plan it is hard to achieve a balanced query load across all nodes: it is rare to find sub-expressions that happen to be equally expensive to compute. Fortunately, the structured nature of array queries allows the creation of new, balanced opportunities to split the query for distribution.

4.1 Query Fragmentation Patterns for Array Queries

The straightforward approach to distribute an array query 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, for example[1]:

$$[f(x)|x < n] \Rightarrow [f(x)|x < n/2]{+}{+}[f(x + n/2)|x < n/2]. \tag{4}$$

Rewriting the query plan like this introduces an operator in the query, the array concatenation '++', which represents a new opportunity to split the query in balanced sub-queries.

It is possible to ensure that the new sub-queries are balanced thanks to the context knowledge that is available in the array domain. All the array operations are position-based rather than value-based. This absence of selective operations on values means that the size of all the intermediate results of a query plan is known in advance with no uncertainty.

Aggregations are also suitable for the creation of balanced sub-queries. Commutative and associative aggregates are equivalent to a series of binary operations, which means that such aggregates can be split into any number of parts, for example:

$$sum([f(x)|x < n]) \Rightarrow sum([f(x)|x < n/2]) + sum([f(x + n/2)|x < n/2]). \tag{5}$$

Targeting distribution at aggregations specifically seems sensible since large aggregates are frequent in the typical query-load. This is reflected by distributed evaluation technology developed for information retrieval applications [14]. In many cases aggregation constitutes a large fraction of the total query cost.

[1] Note that the RAM system rewrites queries at the internal algebra level. However, we present the patterns denoted in RAMs high-level query language for readability.

4.2 Automatic Array Query Distribution

The RAM system is extended to include distribution of these fragmented queries by the introduction of a *distribute* pseudo-operator. The term pseudo-operator is used because it does not operate on the data, instead it manipulates the query execution itself: it distributes sub-queries over multiple computational nodes and collects the results. Notice that it performs a role similar to that of the *exchange* operator in the Volcano system [15].

This new pseudo-operator is introduced into query plans through the addition of tailored equivalence rules in the optimizer. These new rules implement the patterns identified in Section 4.1, Equations 4 and 5. For example, the pattern depicted in Equation 4 produces an expression formed by the concatenation of partial results: $E \Rightarrow concat(E_A, E_B)$, where *concat* is the algebra operator equivalent to the array concatenation, '++', in the RAM syntax. The rewriter includes the *distribute* pseudo-operator to indicate that the various query fragments should be distributed: $concat(distribute(E_A, E_B))$.

The query optimizer performs a top-down search for the best distribution plan. At each opportunity, uniform fragmentation of the query over all available nodes is attempted. The search is terminated as soon as the cost of non-distributed part of the next generated query plan surpasses the total cost of the best plan found earlier.

The RAM cost model is designed to steer the minimization of the data volume to be processed. Its cost function computes a score based on the total amount of data generated by a query. For normal operators the cost is recursively determined by assigning a cost for the operator itself to the sum of its children's costs:

$$cost(op(E_1 \dots E_n)) = cost\_function(op) + \sum_{i=1}^{n} cost(E_i).$$

The *distribute* pseudo-operator is treated differently: it gets assigned only the maximum cost among its children, as they are evaluated in parallel, and a cost factor related to node-communication. Assuming data volume to be the dominant factor in data transfer leads to this cost estimate function:

$$cost(distribute(E_1 \dots E_n)) = max(cost(E_1) \dots cost(E_n)) + \sum_{i=1}^{n} c|E_i|, \qquad (6)$$

where c is a constant tuned to the speed of the network connecting the nodes.

Although the current cost model may be further improved by taking more variables into account, it is shown to be reliable. The reason for this is not to be found in its complexity, but rather in the choice of the domain where the costs are estimated. Performing optimizations and deciding fragmentation strategies in the array domain allows for the exploitation of properties usually not available in the relational domain. Cost models tend to be complex and of limited reliability in the relational domain because the size of intermediate results can only be estimated. In the array domain, the exact knowledge on this factor justifies the usage of simple cost models and ensures more reliable results.

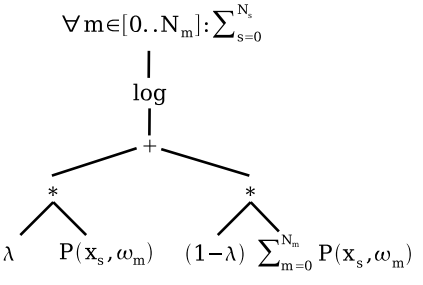
5 Experiments

This section presents experimental results aimed at verifying the effectiveness of the distribution strategies outlined in Section 4. The goal is to show that not only the query optimizer produces viable distribution plans, it also has the means to reliably select the most efficient strategy.

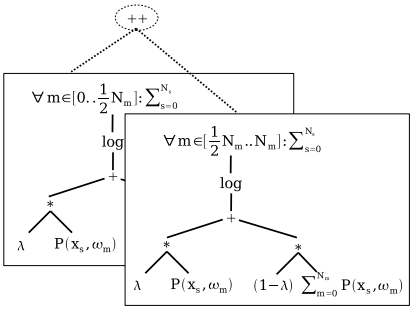
Figure 1 visualizes three possible strategies for the query derived from equation 3. These query plans are referred to as Query B, Query C, and Query D in the remainder of this paper. The original non-distributed query, as it is in Expression 1, is referred to as Query A.

These examples are just three of the many possible alternative distribution plans that are considered by the system. There is no need for users to study the formulas by hand to decide on a suitable query fragmentation strategy: the RAM system derives the most suitable strategy automatically. The experiments are focused on these three variants since they represent patterns that can be intuitively explained.

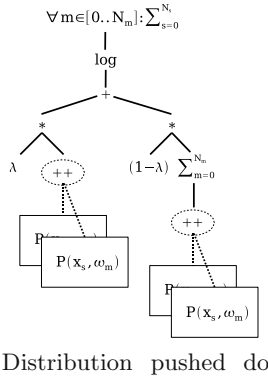
Query A, the original non-distributed query plan, is visualized in Figure 1(a).



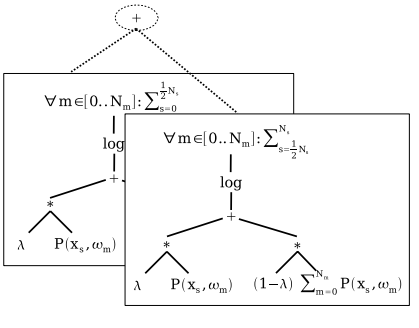
(a) Original query



(b) Query on disjunct sub-collections



(c) Distribution pushed down in the query tree



(d) Partial queries on the whole collection

Fig. 1. Query fragmentation strategies

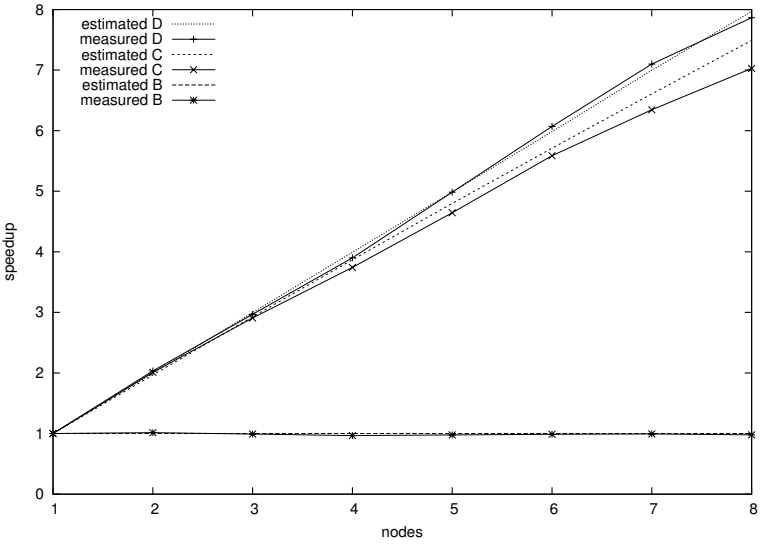


Fig. 2. Speed-up of Queries B, C, D when using an increasing number of nodes

Query B, depicted in Figure 1(b), represents the pattern where disjunct subsets of the collection are independently scored by different nodes. This approach is not effective in our case study, because of the aggregation over all documents that is part of the score for each individual document: when performed naively, this query-fragmentation strategy results in a situation where each node still needs to compute the values the entire collection.

One solution around this problem is to move fragmentation further down the query expression, as depicted in Figure 1(c) and implemented by Query C. In this case, the large intermediate matrix of individual sample probabilities per model is computed in fragments, assembled, and used to compute the final scores. This approach has two downsides however: this matrix has $N_s \times N_m$ elements which may result in considerable communication overhead, and a significant part of the computation is no longer performed in parallel as the collected data requires non-trivial post-processing.

Finally we consider Query D, as depicted in Figure 1(d), which fragments the query along another dimension of its problem space. The query is fragmented along the direction of the aggregation, each node computes a partial score for each document and these partial scores are subsequently combined into a single value. This approach results in a more manageable communication overhead compared to the previous solution, of N_s elements per node.

5.1 Measurements

Experiments have been conducted on a cluster composed of 8 slave nodes and one master node[2] Calibration tests have shown that, for this cluster of machines,

[2] Each configured with an Opteron 1.4GHz, 2GB of main memory, Gigabit network interface, and MonetDB [8] version 4.4.0.

Table 1. Execution statistics for query distribution over 8 nodes

Query	master % time	slave % time	network % time	network data volume
B	0.004	99.993	0.003	$c * Nm$
C	11.6	88.1	0.3	$c * Nm * Ns$
D	0.005	99.993	0.002	$c * Nm * \#nodes$

$c \approx 120$ is a suitable value for the constant c introduced in Equation 6. Query A has been executed on the single master node, and queries B, C, and D have been distributed over a number of nodes ranging from 2 to 8 slaves and a master-node.

Figure 2 summarizes the results in terms of speed-up with respect to the non-distributed Query A. For each of the three distributed variants, the estimated speed-up and the measured speed-up are shown.

The first observation is that in all cases the measured speed-up is in-line with the predicted values (Figure 2). This shows that, for these queries, the cost-model proposed in Section 4 reliably estimates the relative costs of alternative query plans.

The second observation is that Query D exhibits the best scalability, as predicted by the RAM optimizer and our intuitive expectations. Query C also exploits the increasing number of nodes, although not as effectively as Query D does, whereas Query B does not provide any significant speed-up. As shown in Table 1, for both queries B (worst) and D (best), nearly all time is spent computing the distributed part of the query. Here, time spent on communication and post-processing of the data on the master node is negligible. The reasons why Query C is less scalable than Query D are explained clearly by Table 1: first, since distribution is pushed down in the query tree, a significant fraction of the query (11%) is not parallelized, but executed sequentially by the master node; second, because Query C requires more data to be transferred than the other distributed queries, its communication overhead is more noticeable.

A final observation is that, in the experimental setting, communication costs are not really an issue. Only in the case of Query C, communication overhead is noticeable but still hardly significant for the overall cost. There are two apparent reasons for this: first, the workload represented by the query in our example case is large and by far the predominant factor in the overall execution costs; second, experiments were performed on a tightly coupled cluster of machines with a fast interconnecting network. While both aspects seem reasonable in light of our target domain – large scale scientific problems inherently bring costly computations and are usually processed on dedicated machinery – it remains to be seen how the results translate to different environments.

6 Discussion and Conclusions

The experimental results clearly indicate that distributed database technology is applicable to complex scientific queries, such as the ones issued by recent information retrieval research.

However, a correct reading of these results goes beyond a mere quantitative analysis of the problem at hand. Although the identification of the most efficient fragmentation strategy is an important issue for the practical usage of the system, we would like to emphasize that many strategies other than those presented are possible, and that it is not our intention to analyze their efficiency in this paper. The valuable information for the scope of our research is in the comparison between the estimated and the measured costs. Figure 2 shows that the RAM system is not only capable of fragmenting queries efficiently, but it appears to be particularly reliable in estimating the costs associated to different strategies.

In Section 4 we detail about the reasons of such an accuracy. Mapping array queries into the relational domain does not imply that all the optimization issues are also mapped to the new domain, where they could be effectively solved by the available solutions. We showed, for instance, how the size of intermediate results, that plays a crucial role in the optimization and in the distribution of a query, is known with no uncertainty in the array domain. The mapping process, however, results in an inevitable loss of such context information, which can then be only estimated by the target relational system. The novelty introduced by the RAM system is combining the exploitation of the context knowledge in the array domain with the usage of the advanced solutions available in the relational domain.

In this light, RAM appears to be an attractive solution for scientific computations on large data volumes. Its data model provides advantages at different levels: it drastically reduces user effort required to implement database-powered solutions by tackling problems within the array domain, and its structure allows a reduced complexity of the optimization process without sacrificing effectiveness. Likewise, query distribution benefits from its realization at the array level.

References

1. Maier, D., Vance, B.: A call to order. In: Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems, ACM Press (1993) 1–16
2. Cornacchia, R., van Ballegooij, A., de Vries, A.: A case study on array query optimisation. In: First International Workshop on Computer Vision meets Databases (CVDB 2004), Maison de la Chimie, Paris, France, ACM Press (2004) 3–10 In cooperation with ACM SIGMOD.
3. van Ballegooij, A.: RAM: A Multidimensional Array DBMS. In: Proceedings of the ICDE/EDBT 2004 Joint Ph.D. Workshop. (2004) 169–174
4. Baumann, P.: A database array algebra for spatio-temporal data and beyond. In: Next Generation Information Technologies and Systems. (1999) 76–93
5. Buneman, P., Libkin, L., Suciu, D., Tannen, V., Wong, L.: Comprehension syntax. SIGMOD Record **23** (1994) 87–96
6. Libkin, L., Machlin, R., Wong, L.: A query language for multidimensional arrays: Design, implementation, and optimization techniques. In: ACM SIGMOD 1996, ACM Press (1996) 228–239

7. van Ballegooij, A., de Vries, A., Kersten, M.: Ram: Array processing over a relational dbms. Technical Report INS-R0301, CWI (2003)
8. Amsterdam, C., of Amsterdam, U.: Monetdb. (`http://sourceforge.net/projects/monetdb/`)
9. Ianeva, T., Boldareva, L., Westerveld, T., Cornacchia, R., de Vries, A., Hiemstra, D.: Probabilistic approaches to video retrieval. In: TREC Video Retrieval Evaluation Online Proceedings. (2004)
10. Westerveld, T.: Using generative probabilistic models for multimedia retrieval. PhD thesis, University of Twente (2004)
11. Inc., T.M.: Matlab. (http://www.mathworks.com)
12. Ceri, S., Pelagatti, G.: Distributed Databases. McGraw-Hill Book Company, (Singapore)
13. Ozsu, M.T., Valduriez, P.: Principles of distributed database systems (2nd ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1999)
14. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. OSDI (2004)
15. Graefe, G.: Volcano - an extensible and parallel query evaluation system. IEEE Trans. Knowl. Data Eng. **6** (1994) 120–135