A Case Study on Array Query Optimisation

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

ABSTRACT

The development of applications involving multi-dimensional data sets on top of a RDBMS raises several difficulties that are not directly related to the scientific problem being addressed. In particular, an additional effort is needed to solve the mismatch existing between the array-based data model typical for such computations and the set-based data model provided by the RDMBS. The RAM (Relational Array Mapping) system fills this gap, silently providing a mapping layer between the two data models. As expected though, a naive implementation of such an automatic translation cannot compete with the efficiency of queries written by an experienced programmer. In order to make RAM a valid alternative to expensive and time-consuming hand-written solutions, this performance gap should be reduced. We study a real-world application aimed at the ranking of multimedia collections to assess the impact of different implementation strategies. The result of this study provides an illustrative outlook for the development of generally applicable optimisation techniques.

1. INTRODUCTION

Most multimedia retrieval efforts use database systems. However, it rarely entails the entire retrieval process. Usually, the database system serves only as a persistent store for (derived) meta data for multimedia. Only the most rudimentary retrieval functionality, such as keyword search on transcripts, is handled by the database system: complex tasks like multimedia indexing are performed by proprietary external tools. This phenomenon is not limited to the multimedia domain: application areas beyond (simple) administrative tasks are dominated by custom-built solutions.

Using database technology for multimedia applications is a challenge that has an impact on the requirements of such a system. First, it shifts priority toward runtime performance; while high performance has always been one of the requirements of a database system, in a business setting this requirement has been overshadowed by criteria such as reliability and security. Also, in domains like multimedia, the order of data objects plays a very important role, such that the set-based data model underlying relational database technology may no longer suffice. 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]. It causes an unnatural encoding of (multimedia) objects in the relational data model, encouraging users to implement their processing code client-side. The result is that the DBMS is still used merely as a persistent data store.

Our research builds upon past experience with the implementation of multimedia retrieval and analysis in a database setting, see e.g. [2]. While the set-oriented (bulk) query processing techniques common in relational query processing have proven their potential value for multimedia analysis, the efficient manipulation of the (inherently ordered) media data requires non-trivial data storage schemes. The resulting query plans are rarely intuitive, while deriving such plans is a laborious and error-prone process: it is difficult to keep track of the relationship between the operations in the relational query and the steps in the original algorithm.

This paper presents a case study of how to apply the database approach successfully to a multimedia retrieval problem, overcoming the issues discussed above. We detail how a non-trivial video retrieval application has been moved from a stand-alone application to a client-server database application. The case study relates to the video retrieval system that our research group has applied for both TRECVID 2002 and 2003 [3, 4]. The core of its retrieval process computes the likelihood of an example image query for each indexed shot, using a probabilistic retrieval model based on Gaussian mixture models.

The retrieval system has been originally implemented in Matlab [5], precisely because of the aforementioned mismatch in data models and the (resulting) lack of efficiency. In this report, we focus specifically on the strategies applied to overcome the problems encountered when implementing our retrieval algorithm directly on top of a relational database system. For each individual strategy, we discuss its generality or its limitations with respect to its application in different settings.

2. THE RAM APPROACH

Expressing the main algorithm of our case study using a standard set-based database query language is not trivial. As detailed in Section 3, our multimedia retrieval algo-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CVDB'04, June 13, 2004, Paris, France.
Copyright 2004 ACM 1-58113-917-9/04/06 ...$5.00.

rithm requires multi-dimensional data structures to handle the image representation in feature space as well as the parameters of the Gaussian Mixture models. Using these data structures in a set-oriented query language (such as SQL) requires the developer to make the dimensions that are referred to explicit, resulting in a rather lengthy SQL query. Besides the added complexity, the result is also undesirable from the perspective of the DBMS engine: large complex SQL queries pose a problem for the query optimiser.

We seek a solution to this problem of expressiveness of standard relational query languages for multi-dimensional data by introducing array data structures in the database query language. The RAM (Relational Array Mapping) system is a prototype array database system, that aims to disclose database technology to users with complex computational tasks [6]. A discussion of the prototype RAM implementation is presented in [7]. Instead of developing an array database system from scratch, arrays are added to existing database systems, by mapping internally the array structures to relations. This way, array extensions naturally blend in with existing database functionality.

The RAM system provides a comprehension based array-query language (see [8]), that allows to express array-specific queries concisely. Array comprehensions allow users to specify (new) arrays by declaring it's dimensions and a function to compute the value for each of its cells. The RAM query language is similar to the one described in the AQL proposal [9]. Support for this language is isolated in a separate front-end that communicates to the DBMS by issuing relational queries. This approach matches the layered design of MonetDB [10], the primary target system for RAM. MonetDB supports various front-ends, each providing a different query language, that implement their query processing strategies on top of a generic relational kernel.

The front-end does not translate the array comprehensions directly into the back-end's relational query language. Queries are translated into an intermediate array-algebra before the final transformation to the relational domain. This intermediate language is utilised by a traditional System-R style optimiser [11], specifically geared toward the optimisation of array queries. Through application of rewriting rules the RAM optimiser searches for a (more) optimal query plan by minimising intermediate result sizes. The actual optimised query expressed in the target query language is the result of the last transformation step. Figure 1 depicts the life-cycle of an array query through the different layers of the RAM system, as described above.

By adding the array functionality into a relational DBMS, we can reuse existing storage and evaluation primitives. By storing array data as relations the full spectrum of relational operations can be performed on that array data. This indirectly guarantees complete query and data-management functionality: the RAM front-end focuses solely on problems inherent to the array domain.

The challenge being addressed by our current and future research activity is to effectively exploit the opportunities introduced by the RAM system in the multimedia information retrieval domain.

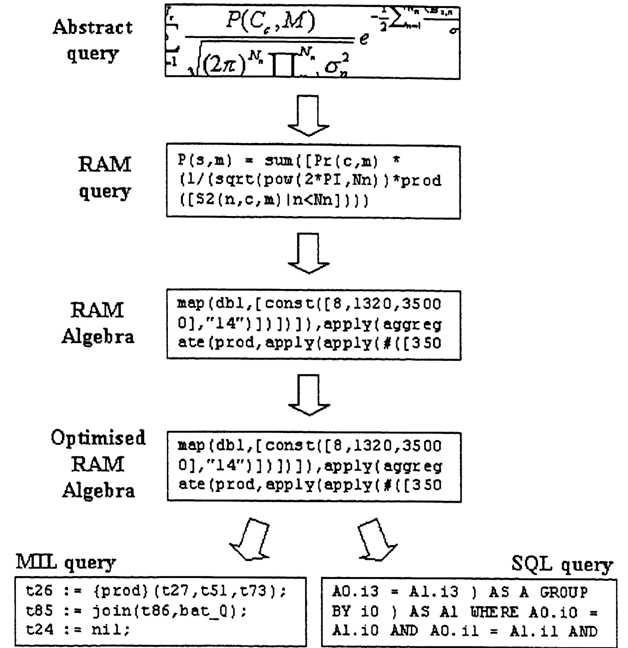

Figure 1: Life-cycle of a query evaluated by RAM.

3. CASE STUDY

The main contribution of this paper is a case study to investigate the feasibility of a RAM implementation of the probabilistic retrieval system that our research group developed for the search task of TRECVID 2002 and 2003: the retrieval of relevant shots of video material given a query image. The probabilistic retrieval method used to rank video shots is a generative model. Using generative models for information retrieval (IR) follows the so-called 'language modelling approach' to IR (see e.g. [12]). Applying this idea to image retrieval has been pioneered in [13]. Now, before changing our focus to the database aspects of this retrieval problem, this section presents concisely the visual part of the multimedia retrieval system studied. The interested reader is referred to [14] for more details.

Image documents are first decomposed as bags of samples (8 by 8 pixel blocks), described by their DCT coefficients. These bags of samples are subsequently modelled as probability distributions, by fitting a Gaussian Mixture model. The relevance of a collection image given a query image is then assumed to be approximated by the ability of its mixture model ω_m to describe the samples $\mathcal{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{N_s})$ of the query image:

$$P(\mathcal{X}|\omega_m) = \prod_{s=1}^{N_s} P(\boldsymbol{x}_s|\omega_m). \quad (1)$$

The probability $P(\boldsymbol{x}_s|\omega_m)$ for a single sample $\boldsymbol{x}_s$ is obtained by summing the contribution of each component of the mixture model, altered by its a priori probability $P(C_c)$:

$$P(\boldsymbol{x}_s|\omega_m) = \sum_{c=1}^{N_C} P(C_{c,m})\mathcal{G}(\boldsymbol{x}_s, \boldsymbol{\mu}_{c,m}, \boldsymbol{\Sigma}_{c,m}). \quad (2)$$

Here, the probability density function for each component is defined as a multivariate Gaussian distribution in N_n dimensions:

$$\mathcal{G}(\boldsymbol{x}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^{N_n}|\boldsymbol{\Sigma}|}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})}. \quad (3)$$

Assuming that the Gaussian models have a diagonal covari-

ance matrix (i.e. $(\Sigma)_{ij} = \delta_{ij}\sigma_j^2$) simplifies equation 3 to:

$$\mathcal{G}(x, \mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^{N_n} \prod_{n=1}^{N_n} \sigma_n^2}} e^{-\frac{1}{2}\sum_{n=1}^{N_n} \frac{(x_n - \mu_n)^2}{\sigma_n^2}} . \quad (4)$$

We switch to log-space to avoid precision problems in computing the complete ranking formula:[1]

$$P(\mathcal{X}|\omega_m) = \prod_{s=1}^{N_s} P(x_s|\omega_m) =_{\text{rank}} \sum_{j=1}^{N_s} log(P(x_s|\omega_m)). \quad (5)$$

Formula 5 maps almost directly to the RAM syntax. We first define a function p corresponding to Formula 2.

```
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 ])
```

Q is an array containing Ns samples from the query image; P, Mu and S2 are arrays containing the prior, mean and covariance values of a Gaussian mixture model, each consisting of Nc components over a Nn dimensional feature space.

Function p is applied in the creation of an array that contains a score for each of the Nm Gaussian mixture models in the collection:

```
Scores = [ sum( [ log( p(s,m) ) | s<Ns ] ) | m<Nm ]
```

The RAM queries given may seem far from trivial, but recall that they express a non-trivial problem to start with. It should be clear from a comparison to Equations 4 and 5 that the mathematical description 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.

4. EXPERIMENTS

The RAM prototype system implements a rather simple and straightforward translation scheme to transform the declarative RAM expressions into relational query plans. This simplicity results in a (naive) query execution plan that provides dissatisfying results with respect to performance, at least for the case study at hand; the query plan generated by the prototype was more than an order of magnitude slower than the original Matlab application. Also, scalability proved an issue, because it generates and materialises all intermediate stages in the computation process.

We have analysed the specific bottlenecks in the initial query plan, and developed several variants of the GMM computation query that was generated by the RAM prototype system. Fortunately, the well structured nature of array queries together with their highly predictable access patterns have opened up a wide variety of effective optimisations. Here, we present a series of experiments that improves the efficiency of the database implementation of the retrieval system, such that the final results are actually faster than the original Matlab code. The purpose of these experiments has been twofold: firstly, to prove that

[1]Here, the symbol '$=_{\text{rank}}$' indicates equivalent document ranking.

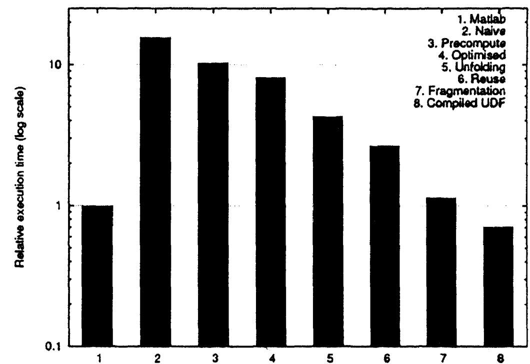

Figure 2: Query evaluation time relative to Matlab. (Optimisations are applied incrementally)

the problem of our case study can be addressed efficiently using a database application, and, secondly, to identify additional *patterns* in the hand optimised variants that can be utilised by the RAM system. Some of these optimisations have already been integrated in the RAM query compiler, while others are currently realised by manual intervention in the automatic query generation process.

Figure 2 shows the performance of each version of the query plan relative to a baseline. This baseline is given by the performance of our reference implementation: the Matlab script used in our actual TRECVID participations, hand-written and optimised for performance.

The presentation of the experiments is ordered by the abstraction level of the optimisation strategies employed, ranging from a high-level algorithmic point of view to the exploitation of some low-level DBMS specific features. The RAM-based solutions are translated into MonetDB's query language (called MIL [15]). Each improvement is added incrementally, thus enhancing the overall performance of the query plan with each step: the final query plan incorporates all earlier improvements as well. The relative timings in Figure 2 refer to the ranking of a collection of 2500 images (Gaussian mixture models), using a query image composed of 1320 samples.

4.1 Pre-computation

At the highest level we observe that the probability estimation function contains a part that is independent of the sample for which the probability is being estimated. The middle part of the query only depends on parameters of the mixture model used:

```
1.0/(sqrt(pow(2*PI,Nn)) * prod([S2(n,c,m) | n<Nn]))
```

This means that this value only needs to be computed once for each of the models and can be reused in the probability estimation of all samples.

By isolating this portion of the query and explicitly materialising its results for subsequent use in the remainder of the computation, re-computation of the same value is avoided. As Figure 2 shows, this minor change in the query plan results in a 35% reduction in query execution time.

Automatic identification of these kinds of sub-queries, subqueries independent from a subset of the dimensions, is straightforward. In a RAM query specific variables are used to refer to specific axes of arrays which represent dimensions of the problem space involved in the computation. The oc-

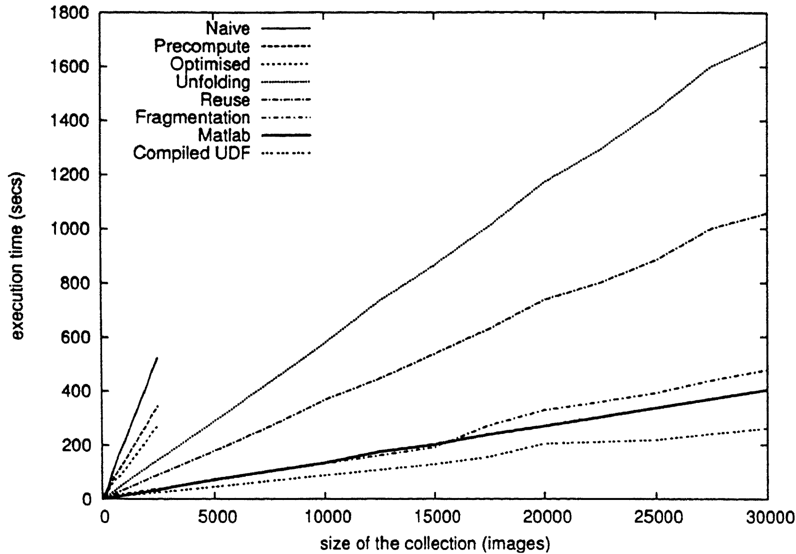

Figure 3: Query evaluation time for different collection sizes.

currence of such reference variables in a sub-query identifies a dependency of that sub-query on the dimension referenced, conversely absence of such variables in that sub-query imply an Independence from those axes. Deciding between either pre-computing and materialising such a sub-query or the alternative of bluntly performing the redundant computations is less trivial. This decision can only be made based upon reliable cost estimation for both solutions.

4.2 Algebraic optimisation

After the mathematical analysis of RAM queries, the system translates the comprehension type queries into an intermediate array algebra. This algebra serves primarily as an intermediate stage, to simplify the translation of RAM expressions to query plans for the relational back-end. However, it *also* provides an excellent opportunity to optimise array queries.

Like most query optimisers developed for relational systems, the RAM optimiser rewrites the query-plan based on equivalence rules. Currently, the rule-base consists of rules aimed at reducing the amount of work performed:

- rules that eliminate identity transformations,
- rules that reduce computations over arrays with constant values to constant expressions, and,
- rules that avoid computation of unused portions of intermediate arrays.

As Figure 2 shows, application of these optimisation rules to the query plan already results in an additional 21% reduction in query execution time. This step in the experiments is already fully automatic.

4.3 Aggregates unfolding

A problem that remains however is that the query plan does not scale well: aside from its execution speed, many (often large) intermediate results are materialised, causing the system to fail on a shortage of storage space. As can be read from Figure 3 [2], the query plans obtained so far have failed at collection sizes of over 3000 images.

The obvious pragmatic solution to solve the scalability issues is to simply divide the dataset into smaller chunks and compute results one chunk at a time. Bluntly chopping up the input data into uniform chunks may result in satisfying performance in some cases, but could also result in iterative query plans that repeatedly need to reference the same chunks. However, the GMM-based scoring function is representative of many algorithms involving multi-dimensional spaces: the target result is an aggregation over a computation involving the Cartesian product of all input dimensions. This pattern allows to derive a suitable fragmentation strategy that matches the access pattern.

Most aggregation functions are basically repeated application of a binary operator to subsequent elements, e.g.: the *sum* aggregation operator can be written as a series of additions. Consider a query that generates some 2-dimensional matrix, and subsequently computes the sum of each of its columns, ultimately producing one 1-dimensional vector of aggregate values. By default the RAM system explicitly translates this into a query that produces an array representing the intermediate first and then applies the aggregation function $\sum$ to the (materialised) intermediate result. In such cases, a big multi-dimensional arrays is created, some computation is performed on the whole array at one time, and finally it is collapsed back to few dimensions by an aggregation function. As an example, consider part of the GMM computation (see Section 3):

```
Scores = [ sum( [ log( p(s,m) ) | s<Ns ] ) | m<Nm ]
```

Here, the naive approach would first create a `[Ns,Nm]` array, compute the `log( p(s,m) )` expression on each cell, and

[2]Figure 3 shows wall-clock timings from experiments run on a machine with a 1400MHz AMD Opteron CPU and 16GB of internal memory. The collection used for these experiments consisted of Gaussian mixture models composed of 8 components over a 14 dimensional feature space. Finally, the query was a collection of 1320 samples (14 dimensional feature vectors) taken from an example image.

finally collapse all the columns into one by computing the `sum` aggregate.

A natural approach to reduce the intermediate storage a query demands for this is to make use of the fact that a summation is essentially a sequence of additions: rewrite the query such that only one column is computed at once and these columns are added together to incrementally produce the desired result. The advantage is that we know that the query plan does not require values from multiple columns at a time: each column is independently added to the incrementally constructed result accumulator. In addition, each column is added exactly once to the accumulator and can thus be discarded as soon as it is used, minimising the need to retain the entire matrix of values.

By applying this optimisation once to our original query plan, *unfolding* the outer-most summation in the computation, overall performance is significantly improved, a 47% speed-up, as shown in Figure 2. The improvement is due to reduced memory consumption, which avoids the swapping of data between main memory and disk. It is important to realise however, that this increased performance is actually achieved as a side effect of the main goal of this strategy: improved scalability. While earlier versions of the query have failed for shortage of memory on datasets of approximately 3000 images, the one used in this experiment had no problem scaling up to the entire TREC-2003 dataset of 30000 images: indeed, this is the first step toward a real scalable solution.

4.4 Reuse of materialised intermediates

The naive translation of RAM expressions into the DBMS query language has adopted a simple (and rather conservative) policy regarding intermediate results: release an intermediate results' allocated memory as soon as its operator completes. Although this policy provides an effective basic rule for limiting memory usage, a closer look at the code generated by RAM has revealed many opportunities for safely reusing such materialised intermediate results. This is particularly true for algorithms like the GMM computation, where some basic computation is repeated many times (in this case, Equation 2).

In addition to those intermediate arrays introduced by the algorithm directly, many index arrays are created: RAM array operators are position based rather than value based, they use arrays of cell indexes as input. These indexes are generated on the fly every time they are needed. Especially in case of unfolded aggregates, repetitive query patterns may cause the same indexes to be produced many times: caching and reusing those arrays can drastically improve query efficiency, in this case execution time was reduced by 38%.

4.5 Array fragmentation

Provided that the scalability issue can be solved by unfolding a sufficient number of aggregate functions (see Section 5.1 for further details), the unfolding strategy suggests a similar improvement in the evaluation of *mapping operators*. In RAM, a function f can be *mapped* (applied cell by cell) to n arrays $A_1, \ldots, A_n$ if the arrays have exactly the same *shape* (number and size of dimensions). If this is not the case, RAM "aligns" the arrays to the biggest one before mapping function f, using replication of columns when needed. Consider, as an example, the following RAM expression, where `A` is a $[100, 20]$ array and `B` is a $[100]$ array:

```
C = [A(i,j) + B(i) | i<100 , j<20]
```

which stores in `C` the cell by cell sum between every column of `A` and the single column of `B`.

When such an operation has to be evaluated, a more efficient way of implementing the "shapes alignment" mechanism is to *fragment* the array A, rather than expanding the array B. This way only one column per array is kept in memory at the same time and the smaller array does not need any replication of data.

As a further optimisation, we observe that the cost of frequent on-the-fly fragmentations like the one in the example depends on the physical representation of the initial data. Since each fragmentation ends up in a selection from the original array, a "smart" organisation of the initial data would minimise this cost. In principle, the system has all the information needed to perform a sort of pre-fragmentation of tables before starting the actual computation, under the assumption that these tables are to be used by the query they were optimised for. However, for the purpose of our experiment, we manually fragmented the initial data.

Figure 2 and Figure 3 show a performance improvement of 58% achieved by the *array fragmentation* strategy. Notice that the sequence of query processing strategies applied so far has removed almost all overhead with respect to the baseline, that was introduced in the naive translation of the original RAM expression to its corresponding relational query plan.

4.6 UDF compilation

Modern database systems allow the user to extend the database query language by introducing *user defined functions* expressed in some external programming language. Lack of expressiveness is a first possible reason to use such a technique: consider, as an example, an SQL query involving some multi-column complex computation in a multimedia or financial domain. Nevertheless, the user may resort to external languages even in those cases when functionality provided is in principle sufficient, simply for the sake of improved runtime performance.

Unfortunately, encouraging users to construct complex queries as external libraries to solve specific problems partially defeats the purpose of a DBMS. It forces them to manually define data processing techniques at implementation level in an imperative language, creating "black-boxes" opaque to the system. Shifting part of the query to a general-purpose language decreases the chances of formulation of a consistent and complete optimisation strategy by the DBMS engine: characteristics of the operation implemented are unknown to the optimiser, and it is impossible to change the physical representation of the data it consumes. For these reasons, one should use UDFs only sporadically: for few, reusable, and performance-critical functions, and, only after all higher level optimisation strategies have been exhausted.

In our case, when profiling the execution of the so far optimised GMM query, we found that more than 75% of the whole execution time was spent on the computation of a small part of Formula 4: the Mahalanobis distance function, given by $\frac{(x_n - \mu_n)^2}{\sigma_n^2}$ (for a single dimension n). This computation of the Mahalanobis distance is a perfect candidate to be compiled into a UDF: it is performance-critical, it is a small part of the query, its implementation is trivial, and, it can be reused by several applications.

Figure 2 shows that implementing the Mahalanobis distance as a user defined function squeezed out another 38% reduction of query execution time. In fact, this final version of the query evaluates faster than the manually optimised baseline Matlab implementation of the algorithm. However, we should be careful drawing conclusions from this observation: resorting to compiled UDFs is an extreme optimisation measure and it remains to be seen whether it is feasible (and desirable) to automatically discover suitable candidates for UDF compilation.

5. DISCUSSION

The various optimisation techniques discussed in the previous section have shown to be effective as a combined package. These optimisations have been applied in a particular order for a reason: subsequent optimisations are performed at decreasing level of abstraction.

Analysis of the experimental results raises two main issues: first, why is the unfolding technique so effective at increasing scalability; second, are these optimisations independent improvements, or are they only effective in combination?

5.1 Scalability

The GMM computation algorithm examined in this case study has an inherent linear complexity with respect to the size of the data. Therefore, regardless of the optimisations applied, query execution time will remain dependent on the collection size. Nevertheless, lowering the hidden constant factor of the performance curve can bring the waiting time closer to a range of 'acceptable' values. As clearly visible in Figure 3, all the techniques presented provide an effective improvement.

The *Aggregates Unfolding* strategy plays a particularly important role. Aside from the performance improvement it provides, its major contribution is scalability of the system: the ability to deal with larger datasets without failure.

The naive RAM implementation literally translates the whole problem space to temporary arrays over all the axis: a natural consequence of the algorithm expressed. The sizes of such (intermediate) arrays increase dramatically the chance of running out of main memory, introducing the need for materialisation of intermediate results on disk even for relatively small datasets.

The obvious solution to limit the size of the intermediate results, and therefore the amount of memory required, is to split the original problem in smaller, independent problems and evaluate them separately. The presence of aggregation functions is an excellent opportunity to perform such query fragmentation. Rewriting of an aggregate as a sequence of binary operations (e.g. the sum as a sequence of additions) creates N sub-queries, where N is the size of the axis on which the aggregate is computed. Each of this sub-queries can be evaluated separately in a problem space that is N times smaller than the original one.

Note that aggregation functions appear very frequently in meaningful queries defined over multi-dimensional problem spaces. This makes aggregation operations a good candidate for generally applicable optimisation: the fragmentation performed by the *Aggregates Unfolding* strategy represents the natural optimisation for an operator that decreases the dimensionality of the problem space.

The *Aggregates Unfolding* strategy is already fully integrated as part of the optimiser in the current RAM prototype. The decision on whether or not to apply the unfold optimisation to a particular aggregation function currently relies on simple heuristics. Unfolding is always applied, unless:

- the memory consumption of the naive execution plan is (already) below a certain threshold;
- the optimisation would produce a large number of very small sub-queries, where the disadvantage of introduced overhead is expected to outweigh the advantage of the optimisation.

It is worthwhile to notice two things. First, the memory consumption of a particular query plan is not *estimated* by the optimiser; as a consequence of the array data model the actual size of all intermediate results is known in advance. Second, the currently implemented heuristics for the unfolding strategy (and for other optimisations) are simple; more advanced heuristics could add flexibility and improve the reliability of the optimisation choices made. One example of more advanced application of the unfolding strategy is to apply it partially, where, for instance, a summation is not directly split into a long sequence of binary additions but into several smaller aggregations.

After the optimisation step at the array-algebra level, the actual realisation of the array-algebra operators, including the *unfold* operator, depends on the target DBMS used for evaluation.

Note that our reference platform, Matlab, tends to compute operations on matrices similarly to the naive RAM approach in conjunction with MonetDB as a back-end. Intermediate results are materialised in memory and consequently Matlab also runs out of memory. For this reason, our baseline Matlab script bounded its memory usage explicitly by applying - by hand - some of the same strategies that have now been integrated in the RAM optimiser. For comparison purposes, we plan to implement a layer that translates the RAM algebra to a Matlab script: effectively using Matlab as a back-end instead of a relational DBMS. In the comparison between the Matlab query script generated by RAM and the one written and optimised by hand, we expect to observe similar effectiveness of the various optimisations and resulting performance.

A last consideration suggests further opportunities for improving the system scalability. The *Aggregates Unfolding* strategy, when applied to commutative and associative aggregates (e.g. $\sum$ and $\prod$), provides query plans with additional desirable properties, on top of reducing the sizes of intermediate results:

- sub-queries created by rewriting of aggregates as sequences of binary operations are independent, and,
- the order in which the results of these sub-queries are subsequently aggregated does not matter.

These properties indicate that those sub-queries can be executed in parallel. Although this feature is not part of the current RAM implementation, these patterns for parallel execution are readily provided by the unfolding strategy. Given a target DBMS capable of intra-operator parallelism it is straightforward to exploit this opportunity. Our current target DBMS, MonetDB, provides native parallel execution

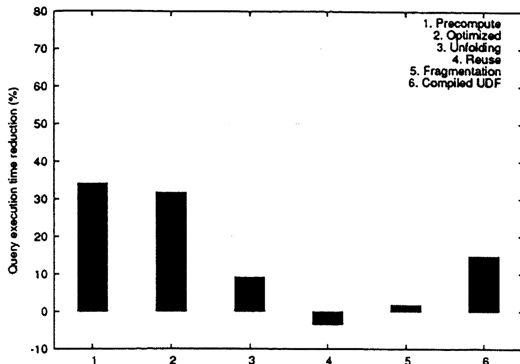

Figure 4: Contribution of individual optimisations when independently applied.

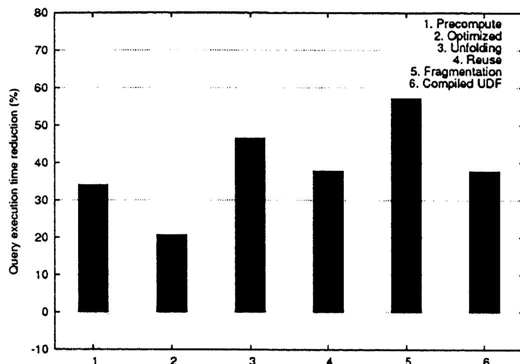

Figure 5: Contribution of individual optimisations when applied in combination.

support. Preliminary experiments, performed as a simple post-processing to the MIL (the MonetDB query language) query generated by the RAM translator, showed near linear speed-up of the query with respect of the number of CPUs deployed.

Although beyond the scope of this paper, it is apparent that parallelisation is a promising direction for future experimentation.

5.2 Optimisation Dependencies

The combined effectiveness of the proposed optimisation strategies has been shown in Section 4. However, these results raise a question: to what extent the contribution of a single optimisation strategy has been influenced by the application of other optimisations? By applying each of the strategies to the naive query plan independently, while disabling all the others, we gain some insight on the dependencies among the various optimisations.

Figure 4 shows the improvement with respect to the naive query plan for each individually applied optimisation strategy. For example, the exploitation of the UDF function (strategy 6 in Figure 4) reduces the execution time of the naive query plan by roughly 16%. Figure 5 shows the contribution of the same optimisation strategies in the setting of the previous set of experiments, where optimisation strategies were applied incrementally. In comparing both figures it is apparent that the various optimisations are not independent: the effectiveness of optimisations performed in isolation or in combination with the other optimisations differs significantly.

Techniques 3, 4 and 5, for example, perform poorly when applied in isolation, while they are among the most effective in the combined scenario. These three techniques have in common that they result in more complex query plans: this inadvertently introduces some degree of overhead. In the combined case, other optimisations (rewriting at the array algebra level in particular) compensate for most of this induced overhead, while in the isolated case this same overhead cancels out most advantages of these techniques. The various optimisations need to be applied in combination for maximum effectiveness.

Combination of optimisations does not only keep induced overhead under control; in some cases one optimisation technique creates opportunities for another. This is the case for the combination of unfolding and reuse: in the isolated case there is little opportunity to reuse intermediates and the overhead induced by the optimisation outweighs its benefits. However, the fragmentation performed as part of the unfolding technique introduces many more opportunities to reuse intermediate results (many identically shaped array fragments are processed with the same patterns) and the effectiveness of intermediate-reuse is dramatic.

Naturally, different optimisations can also adversely affect each-other. For example, the algebraic optimisation reduces overall execution time in the isolated case by more than 30%; in the combined case however, this contribution is closer to 20% (a reduced effectiveness of almost 50%). This effect can be explained by the fact that the previously applied reuse optimisation reduced the freedom of the optimiser to rewrite the query. Note that in this case the cumulative result of the combined optimisations still outperforms the single optimisation.

6. CONCLUSIONS

This paper presented an overview of the RAM system, aimed to reduce the mismatch between the need for array-based processing in handling multi-dimensional data (e.g. multimedia data retrieval) and the standard relational database interface. RAM maps declarative array-expressions to relational query languages. While it improves expressiveness of the resulting database system, a significant performance gap between a naive translation of RAM expressions and optimised hand-crafted applications (partially) remained to be filled. We report upon a non-trivial case study used to identify the most promising directions for the development of efficient query processing for RAM expressions. The resulting system provides an proper alternative to time-consuming and "unnatural" custom solutions.

We analyse six different implementation strategies, showing their impact on the total performance, and compare them (including the naive translation) to the performance of an optimised query script implemented on top of Matlab. The choice of Matlab as a baseline for our tests is mainly due to the observation that it is particularly suitable for computation on multi-dimensional arrays, with respect to both execution speed and language expressiveness. These characteristics make it a good choice for the quick implementation of different prototype algorithms.

Pre-computation and *Algebraic optimisation* techniques provide some important high-level optimisations that re-

sult in better performance. Also, as demonstrated in Section 5.2, these techniques influence subsequent optimisations significantly. The turning point about the scalability issue has been achieved with the proposed *Aggregates unfolding* strategy, which reduces the memory requirements of our experiment dramatically. *Reuse of materialised intermediate results* and *Array fragmentation* provide further improvements by removing some of the overhead introduced in the transformation of RAM expressions to relation query plans.

A final experiment demonstrated the potential of *UDF Compilation.* Automatic or semi-automatic recognition and compilation of crucial UDFs is in principle possible. Nevertheless, we still consider this possibility as an extreme optimisation strategy, addressing our speculations on its desirability more than on its feasibility.

The good results obtained in this case study leave various insights to be carefully verified. Further investigations concern the inclusion in the RAM optimiser of those techniques that proof generically applicable. Particular emphasis will be put in recognising abstract patterns that can be applied at a high-level (RAM algebra), in order to take advantage of the multi-layer structure of the RAM system. Shifting part of the developers expertise to an automatic process is a guarantee of exhaustive and quick exploration of the optimisation opportunities space. Of course, the quality of the result of such a process also depends on the accuracy of the implemented heuristics. Therefore, more research is needed to develop improved heuristics for the existing optimisations.

7. REFERENCES

[1] 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.

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

[3] T.Westerveld, A.P. de Vries, and A.van Ballegooij. CWI at the TREC-2002 Video Track. In *E.M. Voorhees and D.K. Harman, editors,* The Eleventh Text REtrieval Conference (TREC-2002), 2002.

[4] T. Westerveld, T. Ianeva, L. Boldareva, A.P. de Vries, and D. Hiemstra. Combining information sources for video retrieval. In *TRECVID 2003 Workshop*, 2003.

[5] The MathWorks Inc. Matlab. http://www.mathworks.com.

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

[7] 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.

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

[9] 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.

[10] CWI Amsterdam and University of Amsterdam. Monetdb. http://sourceforge.net/projects/monetdb/.

[11] M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, and V. Watson. System R: relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, 1976.

[12] D. Hiemstra. *Using language models for information retrieval.* PhD thesis, Centre for Telematics and Information Technology, University of Twente, 2001.

[13] N. Vasconcelos. *Bayesian Models for Visual Information Retrieval.* PhD thesis, Massachusetts Institute of Technology, 2000.

[14] T. Westerveld, A.P. de Vries, A.van Ballegooij, F.M.G. de Jong, and D.Hiemstra. A probabilistic multimedia retrieval model and its evaluation. *EURASIP Journal on Applied Signal Processing*, 2:186–198, 2003.

[15] P.A. Boncz and M.L. Kersten. Mil primitives for querying a fragmented world. *The VLDB Journal*, 8(2):101–119, October 1999.