RAM: array database management through relational mapping

van Ballegooij, A.R.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 5

Optimization

Database query languages are usually declarative, which means that queries specify the desired result, not how it is computed. In general, a database management system has multiple options at producing a valid evaluation plan for a given query. These plans are equivalent in the sense that they produce the same result, but may differ vastly in resources required for execution (time, memory space, etc.). Picking the most appropriate query plan from the alternatives is the primary task of the query optimizer component in a DBMS.

This chapter investigates how classic relational optimizer technology can be used to optimize array queries. The RAM optimizer is part of a frontend system designed to delegate physical evaluation of the queries to a back-end system. Nevertheless, as argued in Chapter 4, there is a need for array query optimization. The loss of domain knowledge and contextual information caused by the translation of an array-expression into a relational query makes it hard to identify array-centric optimization opportunities in the relational query plan.

5.1 The RAM Optimizer

The task of a query optimizer is twofold: First, at a logical level the processing pattern is optimized to best suit the constraints imposed on the query processor; second, the most appropriate physical implementations of operators are determined. These tasks serve the goal of the query optimizer, which is to find a suitable query plan with (near) lowest estimated costs. The RAM optimizer follows the now classic structure of a query optimizer as produced by the Volcano optimizer generator [1]. The optimizer consist of three parts [2]: First, a query optimizer contains methods to generate alternative query plans; second, a query optimizer has an efficient method to navigate through the space of possible alternatives; and; finally, a query optimizer has a cost model that estimates the cost of a given query plan.

The logical phase of query optimization includes simple heuristics, symbolic manipulation of expressions, and semantic-based optimization. Symbolic optimization
is the manipulation of a query expression by replacing patterns with potentially more efficient, functionally equivalent alternatives. Semantic optimization is more involved and uses integrity constraints on the data, in addition to domain knowledge about the query language, to simplify queries.

Query optimizers often rely on heuristics to prune the search space. Pruning is necessary because the search space (the number of alternative execution plans) is large and (relational) query optimization is known to contain problems that are NP-complete [3]. As query optimization is NP-complete, optimizers have to settle for a good query plan, rather than an optimal plan. A good query plan is a plan that is likely to be near optimal. An example of a heuristic found in relational optimizers is push-select-down, which moves selective operations over constructive operands to reduce data volume as early as possible during execution [4].

The RAM array-query optimizer deals with the logical phase of the optimization process. The array algebra it operates on is not a physical language: Ultimately, the optimizer has no control over how the query is physically executed. The physical phase of the optimization process is delegated, either to the relational back-end system, or a specialized array algebra translator such as those proposed for MIL and X100 in Chapter 4. Isolation of the logical phase into a separate component builds on the concept of a layered optimizer design where each layer performs a specific task in the overall process. In this context, the RAM optimizer fills the role of the “strategic” optimizer that formulates the best possible abstract query plan for the next layer to work with [5].

5.1.1 Query Transformations

An important part of any query optimizer is the ability to generate alternative evaluation plans for a given query. The RAM optimizer uses equivalence rules to generate these alternative plans.

The basic 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 5.1.

\[ \text{map}(f, \text{const}(S, c)) \sim \text{const}(S, f(c)) \]

Another example is the application of a constant array to a set of indexes, which produces a vector.

Equivalence 5.2.

\[ \text{apply}(\text{const}(S, c), I_1, \ldots, I_k) \sim \text{const}(S_{I_1}, c) \]

A similar result can be obtained for constant transformations. The result of an identity transformation is by definition identical to the original.
5.1. The RAM Optimizer

Equivalence 5.3.

\[ apply(A, I_1, \ldots, I_k) \leadsto A \]

when

\[ k \equiv |S_A|, I_1 \equiv \text{grid}(S_A, 0), \ldots, I_k \equiv \text{grid}(S_A, k - 1) \]

In some cases it is not trivial to detect these identity transformations. For example, when persistent arrays are used in index expressions, it would be necessary to analyze its content to prove equivalence to an array axis. The implementation of the RAM optimizer is limited to simple symbolic analysis of the index expressions, which in practice proved sufficient to identify and eliminate many occurrences of identity transformations.

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 5.4.

\[ apply(\text{map}(f, A), I_1, \ldots, I_k) \]

\[ \leadsto \text{map}(f, apply(A, I_1, \ldots, I_k)) \]

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, 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 effectively stored functions. For any array-expression applied to indexes it is possible to perform the application – evaluate the array function – directly through substitution.

Equivalence 5.5.

\[ apply(E, I_1, \ldots, I_k) \leadsto E' \]

where any occurrence of grid(S_A, 1) in expression E is substituted by I_1,

\[ \ldots \]

any occurrence of grid(S_A, k) in expression E is substituted by I_k

Example 5.1 (Substitution). In the following expression,

\[ [(f(z)|z < 6)(x + y)|x < 3, y < 3] \]

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

\[ [f(x + y)|x < 3, y < 3]. \]
The inverse substitution can be used to reduce array expressions that are constant along some of the axes. Suppose there is an array (sub-)expression that is independent of some of the axes in this shape. This expression can then be evaluated over the dependent axes only and extended afterward to the desired shape:

**Equivalence 5.6.**

\[ A \rightsquigarrow \text{apply}(A', I) \]

where

\[ S_I = S_A, \ |A'| < |A|, \ \text{apply}(A', I) \equiv A \]

**Example 5.2 (Dependencies).** The calculus expression

\[ \{ f(x) \mid x < 3, y < 4 \} \]

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:

\[ \{ f(x) \mid x < 3 \}\{ x < 3, y < 4 \}, \]

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

### 5.1.2 Search Strategy

Searching through the space of alternative query plans is a challenging task. Ideally an optimizer would consider all alternatives and pick the single best option. However even for small queries the search space is very large. Fully traversing all alternatives is infeasible and optimizers resort to pruning the search space by enumerating only those plans that appear promising according to some heuristics.

The RAM optimizer first reduces the search space by dividing the transformation rules into two rule sets.

The first set of rules is aimed at simplifying queries by removing unnecessary operations. It consists of transformation Rules 5.1, 5.2, and, 5.3. These rules match patterns that constitute unnecessary work: computations over arrays filled with constant values and identity transformations. Each application of one of these rules directly reduces the amount of data processed.

The second set of rules allows for more complex transformations that are not (heuristically) guaranteed to produce a “better” plan for each application. It consists of the remainder of transformation rules: Rules 5.4, 5.5, and, 5.6. Application of these rules alters the order in which certain operations are applied throughout the query. Individual applications of these rules may or may not directly improve the cost measure for the query. However, a common problem with query optimization is that transformations that initially increase the cost of a query may lead to a better solution later in
the optimization process. Recall, however, that the search space of all alternatives is very large: There is a practical need to reduce this search space.

The RAM optimizer first reduces the search space by applying both rule sets to the problem in separate phases of the optimization process.

In the first phase the optimizer traverses the query graph applying the first rule set wherever possible until none of the rules apply any longer.

In the second phase the RAM optimizer applies the second rule set while further reducing the search space by allowing only \( n \) random rule applications to the query graph. After \( n \) rules have been applied, it picks the best alternative (the plan with the lowest estimated cost) and repeats the process. This strategy is a naive hill-climbing approach: The value of \( n \) is arbitrary. While for infinite \( n \) the approach considers every single alternative, in practice \( n \) will be a relatively small value to guarantee a timely result. The exact value of \( n \) to be used depends on a variety of parameters, such as the size of the query graph and the resources available to the optimizer.

### 5.1.3 Cost Model

Relational optimizers contain complex cost models to accurately estimate the costs related to a particular query plan. The complexity of these models stems from the fact that it is often difficult to reliably estimate the selectivity of a particular operation. For maximal accuracy of these estimations, database systems accumulate statistical data on their relations. Despite a high degree of sophistication, the cost models remain based on estimators: The exact size of any result produced can only be determined by performing the operation. In contrast, for the RAM array queries the exact dimensionality and size of each intermediate result can easily be computed: The array-specific optimizer can compute exact statistics on alternative query plans.

**Intermediate Shapes**

The RAM optimizer estimates the cost of a query plan based on (intermediate) array size. The size of an array follows directly from its shape, which is easy to derive for any RAM array-algebra expression. For any of the RAM array-algebra operators, the shape of the result can be derived given its parameters. For example, for both the \texttt{const} and \texttt{grid} operators the shape of the array produced is directly specified by one of the arguments:

\[
S_{\text{const}}(s,c) \equiv s_c, \\
S_{\text{grid}}(s,g,i) \equiv s_g.
\]

For the \texttt{map}, \texttt{apply}, and, \texttt{choice} operators the shape of the result array is determined by the shape of the arrays that the operators work on:

\[
S_{\text{map}}(f, A_1, \ldots, A_k) \equiv S_{A_1} \equiv \ldots \equiv S_{A_k}, \\
S_{\text{apply}}(A, I_1, \ldots, I_k) \equiv S_{I_1} \equiv \ldots \equiv S_{I_k},
\]
\[ S_{\text{choice}}(C,A,B) \equiv S_C \equiv S_A \equiv S_B. \]

The shape of the results produced by both the \emph{aggregate} and \emph{concat} operators is also determined by the shape of the arrays these operators work on. However, in these cases computing the result shape is marginally more complex:

\[ S_{\text{aggregate}}(g,j,A) \equiv [S_{A_j}, \ldots, S_{A_n}], \text{where } n = |S_A| - 1, \]
\[ S_{\text{concat}}(A,B) \equiv [S_{A_0}, \ldots, S_{A_{(n-1)}}, S_{A_n} + S_{B_n}], \text{where } n = |S_A| - 1. \]

Given these rules the exact shape, and thus size, of each (intermediate) result array in a query plan can be determined.

**Cost Measures**

The cost model for the RAM optimizer computes two statistics for a query plan. The first statistic is the volume of data produced, which corresponds to the total size of the query result itself and all intermediate results. The second statistic is the size of the largest shape that occurs in the query plan. Both statistics are measured in array elements, ignoring (for the time being) differences in storage requirements for the various base types.

The total volume measure is computed by counting the number of array elements in both the final and intermediate results in a complete query plan. This computation is achieved by traversing a query graph and computing the size of the array produced by all operators in the expression:

\[
\begin{align*}
C_{\text{const}}(S_A,c) & = |A| \\
C_{\text{grid}}(S_A,i) & = |A| \\
C_{\text{map}}(f,A_1,\ldots,A_k) & = |A_1| + C_{A_1} + \ldots + C_{A_k} \\
C_{\text{apply}}(A,I_1,\ldots,I_k) & = |I_1| + C_A + C_{I_1} + \ldots + C_{I_k} \\
C_{\text{aggregate}}(g,j,A) & = \prod_{i=j}^{n} S^i_A + S_A \\
C_{\text{choice}}(C,A,B) & = |A| + C_C + C_A + C_B \\
C_{\text{concat}}(A,B) & = (|A| + |B|) + C_A + C_B \\
C_A & = 0
\end{align*}
\]

Note that the size of a persistent array is not counted in this measure, only the volume of data produced during query evaluation is counted. Exclusion of persistent arrays in the measure is a choice: While actual query evaluation cost may differ for alternative usage patterns of a persistent array, the contribution of a persistent array to this measure (its size) is a given.
5.1. The RAM Optimizer

Example 5.3 (Total volume). Consider the following example expression:

\[
\text{apply}(A, \text{grid}([3, 3], 0)),
\]

where \( A \) is an array with shape \([100]\). The expression produces and array with 9 elements as a result and the intermediate array produced by the grid operator also contains 9 elements: the total volume measure for this query is 18.

\[
C_{\text{apply}(A, \text{grid}([3, 3], 0))} = |\text{grid}([3, 3], 0)| + C_A + C_{\text{grid}([3, 3], 0)} = 9 + 0 + 9 = 18
\]

The second measure, the footprint of a query plan, reflects the largest size of any (intermediate) array in a query plan. It is computed again by traversing the query graph and computing the size of each intermediate result:

\[
\begin{align*}
FP_{\text{const}(S_A, c)} &= |A| \\
FP_{\text{grid}(S_A, i)} &= |A| \\
FP_{\text{map}(f, A_1, \ldots, A_k)} &= \max(FP_{A_1}, \ldots, FP_{A_k}) \\
FP_{\text{apply}(A, I_1, \ldots, I_k)} &= \max(FP_A, FP_{I_1}, \ldots, FP_{I_k}) \\
FP_{\text{aggregate}(g, j, A)} &= FP_A \\
FP_{\text{choice}(C, A, B)} &= \max(FP_C, FP_A, FP_B) \\
FP_{\text{concat}(A, B)} &= \max((|A| + |B|), FP_A, FP_B) \\
FP_A &= 0
\end{align*}
\]

Note that in many cases the size of the array produced by the operator itself is not considered for the maximum footprint. In these cases, the result size of an operator is guaranteed to be equal to or smaller than the size of its largest argument. For example, the array produced by an aggregation operation is by definition smaller than the source array.

Example 5.4 (Footprint). Consider again example expression 5.3: The footprint measure for this query is 9.

\[
FP_{\text{apply}(A, \text{grid}([3, 3], 0))} = \max(FP_A, FP_{\text{grid}([3, 3], 0)}) = \max(0, 9) = 9
\]

Notably, neither of these heuristic measures consider differences in processing costs among different operators: Only the volume of the data processed is used. This simplification is motivated by the characteristics of our primary target domain (multimedia) and the known characteristics of the relational backends used. In the multimedia domain, the large inherent data volumes dictate that query-evaluation cost is usually dominated by I/O.

As shown in Chapter 4, both the MIL and X100 relational translations of the array-algebra operators consist of relational primitives with linear complexity. This includes
the specific physical implementation of the join operator used: the fetch-join, which can be used due to the perfect index based on positional information provided by the array domain. Given that memory access is a major bottleneck for database performance [6], in the case of our primary target platform MonetDB in particular, processing cost is dominated by memory access cost and not the CPU [7].

5.1.4 Discussion

The optimization goal for the optimizer presented here is relatively modest. Its focus is the removal of unnecessary operations (such as identity transformations) and the reduction of the amount of data to be processed. Choice of the most efficient low-level operators to evaluate the query remains delegated to the (optimizer of the) back-end system. The effectiveness of the cost models stems from the data model, which allows exact sizes to be computed cheaply. The combination of these characteristics allow the system to be effective regardless of its simplicity: This effectiveness is shown in Chapter 6.

5.2 Optimizer Extensions

This section investigates four possibilities to tune the optimization process to cater to the characteristics of specific back-end systems. First, we discuss how aggregation operations can be exploited to reduce the footprint of a query. The reduction of the footprint of a query is specifically beneficial for main-memory oriented back-ends, whereas it can be counter-productive for pipelined systems. Second, we examine how the optimizer can be adapted to consider distributed query processing in a parallel computing environment. Thirdly, we focus on the recognition of specific patterns that indicate higher-level operations. These higher-level operations can potentially be exploited by specialized translations that are more efficient than the generic solution. Finally, we examine how array domain knowledge about concepts such as shapes and axes can be used to filter unnecessary joins from generated relational query plans.

5.2.1 Unfolding array queries

The RAM optimizer 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 deriving an efficient execution plan.

Intermediate results can require the back end to materialize big tables, posing severe memory-management issues. Fortunately, predictable access patterns in array queries offer opportunities to rewrite these queries to optimize management of system resources.
The evaluation of aggregation functions is critical with respect to maximum memory usage. The following equivalence can be (repeatedly) applied to any commutative and associative aggregate\(^1\). It splits an aggregate over a (large) array into two aggregates over disjoint parts of that array. We call this technique unfolding.

**Equivalence 5.7.**

\[
\text{aggregate}(\sum, A, j) \\
\sim \\
\text{map}(+, \text{aggregate}(\sum, A_1, j), \text{aggregate}(\sum, A_2, j))
\]

where \(\text{concat}(A_1, A_2) \equiv A\)

**Query Cost for Unfolding**

The default measure used by the RAM optimizer is based on the total data volume for all (intermediate) results in a query plan. This measure does not suffice for the unfolding optimization as the additional step introduced into the query by transformation almost always increases that particular measure. As a result, the optimizer will opt not to apply unfolding by default. The purpose of the unfolding strategy, however, is not overall cost reduction, but reduction in memory consumption instead. The transformation has the potential to significantly reduce the footprint of a query as measured by the footprint cost-function.

The optimizer repeatedly triggers the unfolding strategy for a query that violates an a priori specified limit imposed on the footprint measure. This limit is user imposed and should reflect the limits of the memory resources available to the back-end.

**Search Strategy**

The opportunities for unfolding in a particular query plan are limited to occurrences of the aggregation operator. When triggered because the query’s footprint estimate exceeds the limit, the optimizer attempts application of the rule to each aggregate and picks the single occurrence that yields the best cost-model score. The score used in this case is a combination of the primary objective of reducing the footprint measure and a secondary objective of minimizing the induced increase for the total volume cost measure of the query plan.

**Fragmentation of array queries**

The unfolding strategy suggests a similar, query-footprint reducing, 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 the arguments do not have the same shape, the arrays need

\(^1\) Equivalence 5.7 deals with the summation, rules for other functions are similar.
to be “aligned” before mapping-function $f$. In many cases this alignment may result in the replication of smaller arrays. Consider, as an example, the following RAM expression, where $A$ is a $[2, 100]$ array and $B$ is a $[100]$ array:

$$C = [A(i,j) + B(j) \mid i<2, j<100]$$

In this example, array $C$ stores the cell-by-cell sum between both columns of $A$ and the single column of $B$. For evaluation, array $B$ is transformed to match the shape of array $A$.

A more efficient way of realizing this “shapes alignment” may be to fragment the larger array $A$, rather than expanding the smaller array $B$. By fragmenting the query, less data per array is kept in memory at the same time and the smaller array does not need to be replicated:

$$C = [A(0,j) + B(j) \mid i<1, j<100] + [A(1,j) + B(j) \mid i<1, j<100]$$

We observe that the cost of on-the-fly fragmentation, as shown for array $A$ in the example, depends on the physical representation of the initial data. Since the fragmentation results in a series of selections from the original array, an appropriate organization of the initial data minimizes this cost. Naturally, re-organisation of array storage before starting the actual computation is most effective under the assumption that the array is to be used (frequently) by the query its storage is optimized for.

Note that, in the current RAM prototype, this technique has not been implemented, because, at present, the optimizer is limited to the optimization of single queries at a time: It does not include reorganization of persistent data.

### 5.2.2 Distributing array queries

In the field of high performance computing, array computations are usually captured in complex algorithms carefully designed to exploit parallelization. This kind of parallelization is achieved by analyzing imperative algorithms at a low-level and exploiting opportunities for fine-grained parallelism. Viability of this approach requires a complexity of the operations that 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 in isolation 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. Such distributed evaluation 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 disjoint sub-queries that can be executed
in parallel. In an algebra, disjoint 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, using those opportunities 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 for distribution.

The RAM optimizer considers query-driven distribution only it is assumed that data is fully replicated at each site. In the case that the data itself is fragmented and stored in a distributed manner, factors other than computation cost come into play. These factors have been well studied in the context of distributed relational databases [8]. When only part of the data is available at a given node, the most-efficient query plan with respect to parallelism may no longer be viable.

A straightforward approach to distribute a query over multiple nodes is to fragment the result space in disjoint 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:

\[ \text{Equivalence 5.8.} \]

\[
\text{map}(f, A) \\
\sim \\
\text{concat(\text{map}(f, A_1), \text{map}(f, A_2))}
\]

where \( \text{concat}(A_1, A_2) \equiv A \)

Aggregations are also a suitable operation for the creation of balanced sibling sub-queries. Equivalence 5.7 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 plans in this manner introduces a new operator in the query that 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 \textit{distribute} pseudo-operator distributes its arguments (sub-queries) over multiple machines and collects the results:

\[ \text{Equivalence 5.9.} \]

\[
\text{map}(f, A_1, A_2) \sim \text{map}(f, \text{distribute}(A_1, A_2))
\]

\[
\text{concat}(A_1, A_2) \sim \text{concat}(\text{distribute}(A_1, A_2))
\]
The term pseudo-operator is used to indicate that it does not operate on the data, but 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 [9]. Balanced sub-queries can be created using rules as Equivalence 5.7 and Equivalence 5.8 (other ways to create such opportunities can be imagined).

**Estimating Query Cost**

The cost of the distribute pseudo-operator is estimated differently from the normal array algebra operators. For normal operators, the cost is recursively determined by adding a cost for the operator itself to the sum of its children’s costs. 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.

\[
C_{\text{distribute}}(A_1, A_2) = \max(C_{A_1}, C_{A_2}) + c \times \sum(|A_1|, |A_2|),
\]

where \(c\) is a constant representing the discrepancy between in-memory data movement and cross-network data transfer.

This treatment of the operator guarantees that distributed plans are preferred over sequential plans, whereas the constant \(c\) allows for the cost model to be tuned to the characteristics of a specific platform.

**Search Strategy**

The benefit of splitting and distributing the query is likely to be greater closer to the top of the operator tree: The higher up in the tree a query is distributed, the larger the fraction of the query that is evaluated in parallel. This heuristic is incorporated in the optimization process by performing the search for the best parallelization opportunities through a top-down traversal of the query tree.

During the search, creation of distribution opportunities is considered at each point by attempting a uniform fragmentation of the query over the available nodes. Search for an optimal query-distribution scheme is stopped as soon as the cost of the static part of the query, the non-distributed part, is greater than the total cost of the best plan identified so far.

**5.2.3 Alternative Translations**

The RAM algebra operators each capture a small bit of functionality; more complex data processing is achieved through the combination of operators. This approach guarantees that everything that can be expressed in the RAM query language can also be
5.2. Optimizer Extensions 113

evaluated on a back-end that supports the small set of RAM algebra operators. However, combinations of individual simple operators may not always be optimal.

As shown in Section 4.2 the process of translation from the array domain to the relational domain inevitably results in a loss of context. This loss of context can obfuscate the nature of compound operations, hiding alternative evaluation patterns. For example, consider the following expression, where array $A$ has the shape $[3,3]$:

$$apply(A, [grid([3, 1], 0), grid([3, 1], 1)]),$$

the expression translates to a series of relational join operations:

$$(X = grid([3, 1], 0) \bowtie X.i_0 = Y.i_0 \ Y = grid([3, 1], 1)) \bowtie X.v = A.i_0 \land Y.v = A.i_1 \ A,$$

that requires an understanding of exactly what both grids and $A$ represent to optimize. Upon closer examination of the RAM expression it becomes apparent that the query as a whole merely selects all values in a dense rectangular subrange of the domain of $A$. The query essentially performs a range selection on the indexes of $A$ and could be translated as such:

$$\sigma_{A.i_1 < 1} A$$

The difference between this example and, for example, the identity transformation Rule 5.3, is that this particular optimization cannot be performed solely by manipulating the internal RAM array-algebra expression. The array algebra expression has to be extended for the optimizer to be able to express the notion of “selection”.

This section explores two ways of using domain knowledge to produce more efficient query plans for the back-end. The first approach is aimed at capturing common patterns that can be translated efficiently at the array level. The second approach examines ideas on how to approach the problem from the relational domain.

New Operators And Rules

Many large array queries contain elements that are essentially manipulations of existing arrays. The generative nature of the RAM language requires these operations to be achieved through array application, which translates internally to the generation of indexes and a relational join operation. In many cases, this method to compute the result is more expressive than one based on a selection paradigm (instead of generation).

Pivoting An array manipulation that is relatively expensive to express through the array algebra is the reordering of array axes. While in the array algebra, the expression that transposes a matrix requires indices to be created and joined against the original array, the operation can be handled in the relational domain by simply reordering the index columns of the table representing the array.

Definition 5.1 (Operator: pivot). The pivot operator alters the order of the axes of an array:
The operator takes a permutation vector $\bar{p}$ that indicates how to reorder the axes of an array $A$:

$$\text{pivot}(A, \bar{p}) \rightsquigarrow \pi_{i_0=_{p_0}, \ldots, i_k=_{p_k}}.vA$$

What remains is a transformation rule that enables the optimizer to recognize opportunities to introduce this operator in the query plan:

**Equivalence 5.10.**

$$\text{apply}(A, I_0, \ldots, I_k) \rightsquigarrow \text{pivot}(A, \bar{p})$$
when

$$S_{I_0} =_{\text{perm}} S_A,$$
$$k \equiv |S_A| - 1,$$
$$I_0 \equiv \text{grid}(S_{I_0}, p_0), \ldots, I_k \equiv \text{grid}(S_{I_0}, p_k)$$

It is apparent that a relational query consisting of a single projection operation over an array's index columns is likely to be more efficient than the alternative, which generates and uses a join-index to reorder the array elements.

**Range Selection** Another example of an operation that can be realized efficiently in the relational domain is range selection. Range selections occur frequently in RAM queries, especially when queries are partitioned as a result of optimization strategies such as unfolding and distribution discussed earlier. Selection is not part of the array paradigm because performing a selection over array values does not guarantee a valid array as a result. Range selection over array indexes however always produces a dense and rectangular result.

**Definition 5.2** (Operator: index_range_select). The index range select operator performs a range selection over the axes of an array and adjusts the indexes selected such that a valid array, with indexes starting at the origin, is produced:

The operator takes an offset-vector $\bar{o}$ and the shape $S$ of the desired region to be selected from an array $A$:

$$\text{index_range_select}(A, \bar{o}, S)$$

$$\rightsquigarrow \pi_{i_0=_{o_0}, \ldots, i_k=_{o_k}}.v (\sigma_{o_0 \leq i_0 < (S^0 + o_0), \ldots, o_k \leq i_k < (S^k + o_k)}.A)$$

Detecting opportunities to deploy this range selection operation is less straightforward than the pivot, as it requires the reverse engineering of more complicated index-generating expressions in the apply operator.

**Equivalence 5.11.**

$$\text{apply}(A, I_0, \ldots, I_k) \rightsquigarrow \text{index_range_select}(A, \bar{o}, S_{I_0})$$
when

$$S_{I_0} \subset S_A,$$
$$k \equiv |S_A| - 1,$$
$$I_0 \equiv \text{grid}(S_{I_0}, 0) + \text{const}(S_{I_0}, o_0), \ldots, I_k \equiv \text{grid}(S_{I_0}, k) + \text{const}(S_{I_0}, o_k)$$
Like in the case of the pivot operator, at the relational level it is obvious that the new query plan, a single range selection, makes more sense than the generation of a series of indexes to be joined with the array. This improvement is equally notable at the array level: The existing cost model will register the reduced data volume for the query as a whole due to the absence of indexes to create.

The RAM language requires arrays to be aligned before values in multiple arrays can be combined. For example, adding a vector to each column in a matrix, \( [A(x, y) + B(y)]|x, y| \), results in a query plan where the vector is first replicated to form a matrix of matching size before the addition is performed: \( [A(x, y) + [B(x)|x, y|(x, y)|x, y]| \). This replication pattern occurs frequently and can be translated directly to an elegant and efficient relational query plan.

**Definition 5.3** (Operator: replicate). The replicate operator creates a new array with a shape equal to its argument with one added axis. The original array is replicated to fill all slots in the result.

The operator takes the length of the desired axis \( n \):

\[
\text{replicate}(A, n) \rightsquigarrow \pi_{A.i_0, \ldots, A.i_{|S_A|-1}, A.i_{|S_A|}=N.i_0, v}(N = \text{grid}([n], 0) \times A)
\]

The replication query pattern is easily recognized in array algebra.

**Equivalence 5.12.**

\[
apply(A, I_0, \ldots, I_k) \rightsquigarrow \text{replicate}(A, j, S_{I0}^{k+1})
\]

when

\[
k \equiv |S_A| - 1, \\
|S_{I0}| = |S_A| + 1, \\
I_0 \equiv \text{grid}(S_{I0}, 0), \ldots, I_k \equiv \text{grid}(S_{I0}, k)
\]

Expanding beyond these three operators, other operators can be imagined for special cases of the apply operator. For example, an operator akin to the FORTRAN reshape operator, which alters the shape of an existing array by serializing it using the polynomial indexing function and subsequently deserializing the array into its new shape. The essential component of these types of special-purpose operators is that they approach a given situation from the opposite perspective of what the apply operator would. Instead of approaching the problem from the perspective of the result array by defining the values of its cells, these operators produce the same result by manipulating indexes of the source array.

There are interesting commonalities among the operators introduced in this section: First, all three operators are special cases of the apply operation; second, the optimizations proposed cannot be made at the relational level alone as they require in-depth knowledge on the semantics of both the grid and apply operators; and, finally, at the relational level the optimizations lead to elimination of generated grids and relational join operations.
5.2.4 Avoiding Join Operations

From the notable commonalities among the special purpose apply operators two elements clearly stand out: the elimination of grids and their associated relational joins. Normally, the relational back-end lacks the domain knowledge to achieve these results given the original relational query plan. However, given sufficient insight into the array domain, optimization at the relational-algebra level could achieve similar results.

This discussion is limited to fragments of array-expressions where the shape is constant, including the index part of the apply operator, the map operator, the grid operator, and, the const operator. These partial queries are linked with the remainder of the query by treating sub-queries of different shape as opaque: We assume that the results of other operators in the query plan are materialized; these are the aggregate, concat, and the choice operators as well as the source arrays for apply.

Following the basic translation rules, these simplified expressions are translated to relational plans with a simple structure: a projection over join operations for the map and apply operations, or a projection over the base function for the grid and const operations. The resulting relational query plans consist of many projection and join operations over a fixed number of tables. Most of these join operations will be joins that combine aligned arrays, while the remaining join operations correspond to the evaluation of the apply operations.

A common factor in these relational array-expressions is the shape of the result base(S_R): Every data source is either of this shape or manipulated to match it via the apply operator. Data sources only come in two forms, either a materialized array (either a persistent array, or the result of an opaque sub-expression), or the result of the base function.

Focusing solely on the map, grid, and const operators for the moment, a naive relational query plan produced from a RAM query can be improved significantly. Only three possibilities exist for data sources of a (binary) join: first, a join between two projections over base(S_R); second, a join between a projection over base(S_R) and a materialized array; and finally a join between two materialized arrays. In the first case, the join operation can be eliminated at the relational level by noting that the source data on both sides is identical and joined over its key:

\[
\pi_{(i,f(v_1,v_2))}(\pi_{(i,v_1=...)}(base(S_R))) \bowtie \pi_{(i,v_2=...)}(base(S_R)) \\
\Rightarrow \pi_{(i,f(v_1,...,v_2=...))}(base(S_R))
\]

Optimization of the second case, a join between a projection over base(S_R) and a materialized array, requires knowledge from the array domain. It requires information on the shape of the materialized data source (A). Given that the shape of the array is equal to the result shape, \( \pi(i).A = base(S_R) \), the same optimization can be applied:

\[
\pi_{(i,f(A.v,v_1))}(A \bowtie \pi_{(i,v_1=...)}(base(S_R))) \Rightarrow \pi_{(i,f(A.v,v_1=...))}A
\]
Unfortunately, the join in the third case, a join between two materialized arrays, cannot be avoided. To ease further reasoning, we can however treat the result of a join between two materialized data sources as a unit: a new (materialized) data source. With this abstraction and the two patterns discussed so far, any query consisting of a nested sequenced of \textit{map}, \textit{grid}, and \textit{const} operators is reduced to a single projection over either a \textit{base} operator, or joined materialized arrays:
\[
\pi_{(i,\ldots)}(A \times B \times \ldots).
\]

For the most part this optimization can be performed by a purely relational optimizer, the only additional knowledge required is the equality between the shape of materialized arrays and the result shape as built by the \textit{base} function: \(\pi_{(i)} A \equiv \text{base}(S_R)\).

The \textit{apply} operator is somewhat more complex as it introduces a join between a set of index relations of the result shape with an (assumed materialized) array of different shape. This join operation is introduced according to a fixed pattern:
\[
\pi_{(I,\ldots,A,v)}((I_0 \times I_1 \times \ldots) \times_{I_0.v=A.i_0,I_1.v=A.i_1\ldots} A)
\]

The sub-queries that make up the various index relations can be merged, following the logic discussed above, resulting in a single projection over a sequence of joins (potentially eliminating duplicate data sources):
\[
\pi_{(I,A,v)}((I = \pi_{(\ldots \times \ldots)}) \times_{I_0.v=A.i_0,I_1.v=A.i_1\ldots} A)
\]

Unfortunately, optimization of the \textit{apply} operator in isolation does not allow for the elimination of the join operation it introduces. In combination with other operations however, shape analysis provides the potential for further optimization as demonstrated by the following example.

\textbf{Example 5.5 (Matrix multiplication).} The advantages of shape analysis are clearly demonstrated using the matrix-multiplication example:

\[
\text{[sum([A(i, k) \times B(k, j)]|k]|i, j].}
\]

Excluding the aggregation operation, this query translates to the following array algebra expression:

\[
\text{map}(\ast, \text{apply}(A, \text{grid}(S, 1), \text{grid}(S, 0)), \text{apply}(B, \text{grid}(S, 0), \text{grid}(S, 2))),
\]

and subsequently to the following relational query plan:

\[
\pi_{(i_0=v_0,i_1=v_1,i_2=v_2,v=\text{A.v} \ast \text{B.v}}((A \times_{A.i_0=v_1,A.i_1=v_0} (\pi_{v=A.v \ast base(S)}) \times_{A.i_0=B.i_0,A.i_1=B.i_1,A.i_2=B.i_2} (B \times_{B.v=B.i_0,B.v=A.i_2} ((\pi_{v=0=10 base(S)}) \times_{A.i_0=B.i_0,A.i_1=B.i_1,A.i_2=B.i_2} (base(S))))))
\]
Using the ideas outlined in this section, the join operations are merged to simplify this expression to:

\[ \pi_{i_0=v_0, i_1=v_1, i_2=v_2, v=A \times B.v} (\pi_{A.v=B.v}( A \times A.i_0=v_1, A.i_1=v_0 \times ( \pi_{v_1=i_1 base(S)} \times ( \pi_{v_2=i_2 base(S)} )) \times v_0=B.i_0, v_2=B.v_2 )) \]

Finally, the base operations are eliminated completely by analysis of the base shape and the array axes used. In this case the base axes are combined with identical array axes from \(A \) and \(B\), \(S \equiv \pi_{i_0=B.i_0, i_1=A.i_0, i_2=B.i_1} (A \times A.i_1=B.i_0 \ B)\), therefore the join operation with \(base(S)\) can be omitted:

\[ \pi_{i_0=B.i_0, i_1=A.i_0, i_2=B.i_1, v=A.v=B.v} (A \times A.i_1=B.i_0 \ B) \]

This expression is correct, yet it uses only a single one-way join versus five two-way and three-way joins in the initial expression: clearly a significant improvement.

**Discussion**

Section 5.2.3 proposes three new operators. The operators capture specific forms of the application operator and translate directly to existing relational operators, which is different from the operation in the last example: matrix multiplication. Matrix multiplication is a common operation in the computational domains the RAM system is aimed at, and efficient implementations of this operation are known. Therefore, addition of an efficient matrix multiplication implementation might provide a significant performance increase. Unfortunately, the possibilities are endless in both the number of additional operators and the complexity of those operations: For the RAM system we have chosen to maintain the original design criterion of minimal additions to the back-end. Additionally, implementation of complex operations, such as matrix multiplication, in isolated functions might impair the freedom of a query optimizer to optimize the query as a whole: a well known problem for object-oriented databases [10]. A possible alternative is for the optimizer to identify costly sub-queries, and compile at runtime a low-level function to evaluate that sub query: The RAM to C++ generator discussed in the previous chapter provides the functionality required.

As argued, many of the optimizations discussed in this section take place in, or on the boundaries of, the relational domain, yet they require knowledge about the characteristics of the data: arrays. The relation between the optimizations and the relational domain leads to the question of where these optimizations should take place. They could be implemented as part of the relational mapping process as the translator has explicit knowledge about both the array domain and the generated relational queries. Alternatively, given limited additional knowledge about the array domain, a relational optimizer could perform the optimizations as well.
Bibliography


