Accelerating Computationally Intensive Queries on Massive Earth Science Data (System Demonstration)

Peter Baumann
Jacobs University
Campus Ring 1
28759 Bremen, Germany
+49-421-200-3178
p.baumann@jacobs-university.de

rasdaman GmbH
Hans-Hermann-Sieling-Str. 17
28759 Bremen, Germany
+49-421-6595777
baumann@rasdaman.com

ABSTRACT
We present a technique to speed up array query processing, just-in-time compilation, which is novel on arrays. Suitable query fragments are detected, compiled, and linked into the server dynamically. This technique has been integrated in an operational array DBMS where its impact on efficiency can be studied in a holistic environment.

We will demonstrate this on 1D to 4D real-life spatio-temporal raster data sets taken from the Earth Sciences. Performance results on computationally intensive queries are encouraging and can be examined interactively in a variety of demonstration examples.

Categories and Subject Descriptors
H.2.4 [Information Systems]: Systems – Query processing.

General Terms
Management, Performance, Languages.

Keywords
Array databases, just-in-time compilation, query caching.

1. INTRODUCTION
In face of the avalanche of scientific data pouring in from both natural phenomenon observation and lab simulations, value-adding services are increasingly demanded by e-Science user communities. This involves significantly more than just delivering data sets or subsets thereof to users; rather, both simple and complex ad-hoc search and processing are required.

Array DBMSs are streamlined to accomplish this for a large class of these extremely large data, namely multi-dimensional arrays. Advanced analytics on such arrays very quickly gets CPU-bound; hence, optimization of query operators has become an interesting and promising field.

Logical array query optimization is quite well understood; the rasdaman system [2][3][11][16] currently evaluates 150 rewriting rules routinely. Physical optimization, however, is far from being well understood in array databases. For example, on AML it is reported that handwritten C code performs 5 to 181 times faster than query interpretation [10]. Detailed reasons are not reported there. However, from experience with our system we find issues like interpretation overhead, too liberal use of copies for intermediary results, and sub-optimal execution of multi-dimensional loops where code cannot be generated at compile time [16]. An additional issue comes in with the dynamic type system of rasdaman which allows to define array structures of any C/C++ type (except arrays and pointers) during server runtime. Internally this is realized by C++ templates where the code generated contains significant overhead.

In an earlier paper we have introduced just-in-time (JIT) compilation as a way to accelerate array query processing [9]. In this approach, multiple array iterations are conflated into a larger one to reduce managerial overhead in query evaluation.

This method proves beneficial on in-core operations, that is: array processing, so it is most effective on long-running queries containing many operations and/or accessing many array objects. It does not contribute any speed-up on reading data from disk; on the other hand, it does not conflict with disk access optimization techniques like adaptive tiling [4]. As it is integrated with a query processing strategy of tile streaming it also applies to out-of-band workloads where array objects have to be loaded from disk and processed piecemeal.

Following the earlier proof of concept, JIT compilation in combination with loop fusion meantime has been integrated into the rasdaman array DBMS. The system is in several years of use with 1D to 4D datasets of sizes up to a dozen Terabytes, and with standing queries involving more than 50 operators, in operational environments of mapping agencies such as the French National Geographic Institute and Vattenfall Europe Mining.

This allows exercising arbitrary queries under JIT rather than only the lab example provided earlier, helping to determine practically the effectiveness of query caching (which was only assumed earlier). Further, this enables studies of the orchestration behavior of this optimization technique in the context of overall query processing. The demonstration described in this paper allows interested users to explore the dynamic behavior on 1D through 4D real-life data sets taken from Earth Sciences: time series, remote sensing imagery, and oceanography data.
2. QUERYING AND STORING ARRAYS
For our presentation of Array JIT in the next section we need to sketch the array model [2][3] and query language [11] used, together with some basics about the rasdaman tile streaming engine [16]. We do so by way of an example and refer the interested reader to for the concepts behind and to [11] for the complete language manual.

The rasdaman data model lends itself towards SQL which it extends with array-attached attributes. The conceptual model consists of typed arrays of any number of dimensions, extent, and cell type. Arrays are stored in tables called collections (following, for historical reasons, the ODMG model [6]) containing an OID column and an array column. This allows foreign key references to include arrays in the overall database schema in attribute positions, which by our experience is most convenient for real-life modelings.

The rasql language, crafted along SQL, offers expressions on such multi-dimensional array objects. The following example may serve to illustrate this: "A cutout of all those images from collection ModisScenes where the red channel average intensity exceeds 127; from these results, the product of the green channel with the bitmask RegionMask having oid=42, encoded in JPEG."

Expressed in rasql this reads as:

```sql
select jpeg( (m * r) [ x0:x1, y0:y1 ] )
from ModisScene m, RegionMask r
where avg_cells(m.red) > 127
and oid(r) = 42
```

The rasdaman query operators can roughly be classified into

- subsetting operations, delivering cut-outs (i.e., sub-arrays) of the same or a lower number of dimensions;
- induced cell operators where one operation is applied simultaneously on all cells independently;
- aggregate operators; these inspect the entire domain to produce a scalar result;
- two second-order generic operations, marray and condense. Marray is an array constructor and condense an aggregation operator. All of the above operations can be expressed through combinations of these, however, due to the generality of these operators they are harder to optimize than the special cases.

Internally, rasdaman partitions each array object into tiles which form the unit of storage access as well as query processing. Tiles are stored as BLOBs (binary large objects) in a relational database.

3. JUST-IN-TIME COMPILATION
Following the tile streaming paradigm, rasql query tree nodes fetch input data tile by tile according to the classic ONC (open-next-close) protocol. On the output side, tiles are passed upwards to the next processing node. Most of the rasql operators, like induced operations and aggregates, are non-blocking; only a few operators, such as scaling and block tile streaming, need more elaborate concepts.

Tile streaming allows processing of large arrays with limited main memory as tiles processed can be discarded immediately afterwards. On the other hand, even when designed carefully, materialization of intermediate tiles cannot always be avoided. Further, it has been observed that general multi-dimensional iteration through a tile can be performance relevant [16].

A simple example shows this. Consider a rasql query which combines Celsius and Fahrenheit arrays A and B, both of size n×m and with the same regular tiling assumed. A is converted to Fahrenheit before averaging over the pairwise differences:

```
select avg_cells( 1.8*A + 32 ) - B
from A, B
```

Semi-naive evaluation of this four-operation query (with tiles updated in place to avoid excessive copying) requires four iterations over each tile in memory. Hence, overall processing time can be approximated by

$$T = 2T_{\text{alloc}} + 4T_{\text{iter}} + 4nm \cdot T_{\text{op}}$$

where $T_{\text{alloc}}$ is the time needed to allocate $n \times m$ memory cells, $T_{\text{iter}}$ is needed for iterating over one array tile, and $T_{\text{op}}$ is the time needed to perform some operation on a pixel (such as addition and multiplication). As usual in database engines the server attempts to minimize buffer copying including allocation and deallocation [16], hence $T_{\text{alloc}}$ is only prefixed with a factor of 2 (for A and B, respectively). Bottom line, as A and B typically are huge, both $T_{\text{iter}}$ and $T_{\text{op}}$ have a substantial impact on the overall evaluation time. As has been shown in [16], $T_{\text{op}}$ is heavily affected by array index management during the iteration. Consequently, reducing the number of iterations seems worthwhile.

A skilful programmer setting out to create a tailored function for computing this formula would come up with C code like this:

```c
1. avg = 0;
2. for i in Dom(A)
3.   avg += A[i]*1.8 + 32;
4. end
5. avg /= size(A);
6. for i in Dom(B)
7.   result[i] = avg - B[i];
8. end
```

As can be seen this conflation of the “pixel” operations results in only two loops instead of four. Moreover, in the loop body intermediate values are already kept in registers and do not need to be reloaded from RAM. Further, CPU cache maintenance can be affected positively. All this register management is accomplished at compile time and, therefore, normally unavailable to interpreted queries.

Loop fusion addresses the first issue, minimizing the number of loops by re-aggregating operation clusters. The second issue gets tackled by just-in-time compilation aims by tasking the compiler with register optimization at query evaluation time.

3.1 Loop Fusion
We abandon the traditional view of “one operator, one query tree node” and introduce a distinct GroupIterator node type which allows holding a whole query subtree.

A GroupIterator is a tuple $G = (I, T)$ where $T$ is an operation tree and $I$ is a set of array data sources. Each node of tree $T$ is either some operation or, at the leaves, one of the input parameters from $I$. The algorithm conflates several single-operation query tree nodes into one GroupIterator. This is done in a bottom-up fashion by collecting connected tree fragments consisting only of nodes supporting this conflation, and replacing each such fragment by a single GroupIterator node. Nodes supporting loop fusion are those which receive array-valued input and work cell-wise. This
category encompasses, for example, induced operators and aggregates. Aggregates deliver scalars and, hence, can appear only at the root of some processing tree $T$.

More formally, in its bottom-up traversal the algorithm treats a node $X$ as follows (see Figure 1-3):

a) $X$ is a leaf node (array reference or a constant): $X$ is replaced with a GroupIterator node $G$ with $I=\{X\}$ and $T$ having one node $X$.

b) $X$ is an inner node with its operator node $O$ supporting loop fusion: If $O$ has $k$ child GroupIterators $G_1(I,T_1)$, $\ldots$, $G_k(I,T_k)$ then $O$ will be replaced by a GroupIterator $G=\{I,T\}$ where $I=\bigcup_{i=1}^{k} I_i$ with $T$ having root $O$ and children $T_n$ for $n=1\ldots k$. Otherwise, if $O$ is an aggregate operator then the GroupIterator will be locked against further extension.

c) $X$ is an inner node with its operator node $O$ not supporting loop fusion: a new GroupIterator $G$ with $I=\{G_1,\ldots, G_k\}$ is created from its child group iterators $G_1,\ldots, G_k$ and $T$ is a graph containing solely operator $O$ applied over its children. The GroupIterator gets locked.

In Figure 1-3, GroupIterators are displayed as named boxes around the grouped tree portion. A boxed “$X$” indicates that the group is locked and, hence, will not be extended further.

The resulting query tree by construction has at most the number of nodes the original tree has. Often, however, the resulting tree will be much smaller than the original, as in our example where four nodes are conflated into two.

The final tree is evaluated normally, except for GroupIterator nodes to which dynamic compilation is applied. These undergo the steps discussed next.

### 3.2 Code Generation

As the query tree, data types involved, etc. are known only at runtime we use dynamic compilation: From selected query fragments C code is generated, compiled, and linked as a shared object. This shared object, then, is loaded and executed on the input data.

The C code generation algorithm is implemented as a recursive function which first generates code for its children and then concatenates their codes in a way that the result is actually the result of current operation. The following code snippet shows the principle with a binary induced multiplication:

```c
function genCCode(node) {
  if (node.type == MULTIPLICATION) {
    (code1, var1) = genCCode(node.child(0));
    (code2, var2) = genCCode(node.child(1));
    result = genNewVariableName;
    code = code1 + code2;
    code += getResultType() + " " + result + " = " + var1 + " * " + var2 + " ; ";
    return(code, result);
  } else if ...
}
```

Consider the example of rendering an elevation or bathymetry image by classifying the floating-point depth/height values into different colors. The corresponding query contains expressions like

$$(T > -15 \text{ and } T < 0) \ast [10; 40; 100]$$

This involves checking the interval value (three operations) and multiplication of the resulting Boolean value with the RGB color triple (three more operations) to obtain either a zero (i.e., transparent) in case the original value is not in the interval, or the requested color otherwise. Finally, all these classification layers are added up; this is ignored here. The code generated essentially looks like this:

```c
void process( int units,
              void *data,
              void *result ) {
  void* dataIter = data;
  void* resIter = result;
  for (int iter=0; iter<units;
       ++iter, dataIter+=4, resIter +=12) { 
    float var0 = *(float*)dataIter;
    bool c = (var0>-15) && (var0<0);
    "*((int*)resIter) - 10*c;"
    "*((int*)resIter)+4) = 40*c;"
    "*((int*)resIter)+8) = 100*c;"
  }
}
```

Activating gcc compiler optimization will automatically remove the in-loop variables, so our code generating algorithm doesn’t
need to care about such improvements.

The final step is compiling the code into a shared library and linking it into the server executable. For future reuse by a similar incoming query, the GroupIterator query fragment is identified by some hash which ignores constant values so that different parameterizations of a query still can make use of the same pre-compiled code.

Technically, JIT is an optimization step performed following normalization, common subexpression elimination, and rewriting. Note that the first query evaluation step, normalization, already massages the tree into a canonical shape. After this step, equivalent rephrasings by the user still match the same GroupIterator code. For example, commutativity and associativity situations do not present a structural difference any longer.

4. PERFORMANCE

A first experiment has served to gain preliminary experience with the principle. To this end, a 512×512 tile of floating point numbers was used to multiply pixels with themselves in a query following this pattern:

```
select f * f from floatMatrix as f
```

Figure 2 shows times measured in ms for a varying number of such multiplications, the latter plotted logarithmically. The diamond symbols denote the reference run where all optimization has been disabled. Response time increases linearly with the number of operations. The squared line indicates JIT optimization on the same query workload using a cold operation cache. The constant overhead of (re-)generating the JIT code each time is clearly visible. In the upright triangle line a hot JIT cache is employed; response time is faster by the time of JIT code generation and linking, so the difference is constant. The downward triangle, finally, shows the hand-coded variant. A significant difference between the hand-optimized code and the generated code is visible only from about 32 multiplications onwards. Note that only processing time is measured, so it does not matter whether the database is hot or cold in the sense of buffer loading from disk, nor does operating system data caching have any impact.

Results indicate an overhead for dynamic compilation of about 50 ms. Up to about 32 multiplications, CPU usage is in the order of magnitude of a few ms for JIT code execution; only with high number of multiplications differences become more remarkable, and the hand-coded variant performs better.

These are only quite individual observations. While a comprehensive performance evaluation is on the agenda, interactive experiments can be carried out now that the complete DBMS integration is accomplished.

5. RELATED WORK

In the domain of supercomputing, array processing has a long tradition. Loop fusion is one of the techniques successfully applied there [7]. Further, optimization of main memory array operations has been investigated, e.g., in the context of APL [17]. Our approach is to adopt such techniques to cluster array iterations into maximal query fragments for subsequent native code generation at runtime as the number of dimensions of the target object and, hence, the number of nested loops, is known only at query time.

In databases, merging of operators is common in physical level query optimization by combining operations for increased efficiency. A typical example is combination of a nested-loop join with an index scan, resulting in an index-nested-loop join strategy. Such operators are hand-coded and static. In contrast to this, our approach generates the node operation sequence (and, hence, its semantics) dynamically during runtime and aggregates to unbounded complexity.

Extensible DBMSs allow injecting custom C++, C# or Java code into the server engine, such as in [14][1][15]. SciDB seems to plan a similar approach [13]. This approach ensures high performance and low interpretation overhead. The obvious disadvantage is that the user has to identify parts which might benefit from stored procedures, and then implement these manually. Typically, this task requires substantial knowledge about the database engine internals and skills in writing optimized code. Our method tasks the optimizer with identifying those parts of the query which can benefit from compilation to native code, generates the source code, and compiles it. Hence, it is fully automated.

More dynamic is query rewriting; while mostly heuristics focus on reordering, replacing, and precalculating, sometimes recombination of query nodes is supported [8][10]. Still, some authors, when comparing their results with tailored solution written in C or C++, find themselves 5-181 times slower [10] even though the strategies of computing are the same. Our grouping is only slightly outperformed by the hand-coded variants.

Query caching is also reported by MonetDB [12].

![Figure 2: Processing times of a floating-point tile with repeated multiplications applied](image)

In databases, merging of operators is common in physical level query optimization by combining operations for increased efficiency. A typical example is combination of a nested-loop join with an index scan, resulting in an index-nested-loop join strategy. Such operators are hand-coded and static. In contrast to this, our approach generates the node operation sequence (and, hence, its semantics) dynamically during runtime and aggregates to unbounded complexity.

Extensible DBMSs allow injecting custom C++, C# or Java code into the server engine, such as in [14][1][15]. SciDB seems to plan a similar approach [13]. This approach ensures high performance and low interpretation overhead. The obvious disadvantage is that the user has to identify parts which might benefit from stored procedures, and then implement these manually. Typically, this task requires substantial knowledge about the database engine internals and skills in writing optimized code. Our method tasks the optimizer with identifying those parts of the query which can benefit from compilation to native code, generates the source code, and compiles it. Hence, it is fully automated.

More dynamic is query rewriting; while mostly heuristics focus on reordering, replacing, and precalculating, sometimes recombination of query nodes is supported [8][10]. Still, some authors, when comparing their results with tailored solution written in C or C++, find themselves 5-181 times slower [10] even though the strategies of computing are the same. Our grouping is only slightly outperformed by the hand-coded variants.

Query caching is also reported by MonetDB [12].

![Figure 2: Processing times of a floating-point tile with repeated multiplications applied](image)

In databases, merging of operators is common in physical level query optimization by combining operations for increased efficiency. A typical example is combination of a nested-loop join with an index scan, resulting in an index-nested-loop join strategy. Such operators are hand-coded and static. In contrast to this, our approach generates the node operation sequence (and, hence, its semantics) dynamically during runtime and aggregates to unbounded complexity.

Extensible DBMSs allow injecting custom C++, C# or Java code into the server engine, such as in [14][1][15]. SciDB seems to plan a similar approach [13]. This approach ensures high performance and low interpretation overhead. The obvious disadvantage is that the user has to identify parts which might benefit from stored procedures, and then implement these manually. Typically, this task requires substantial knowledge about the database engine internals and skills in writing optimized code. Our method tasks the optimizer with identifying those parts of the query which can benefit from compilation to native code, generates the source code, and compiles it. Hence, it is fully automated.

More dynamic is query rewriting; while mostly heuristics focus on reordering, replacing, and precalculating, sometimes recombination of query nodes is supported [8][10]. Still, some authors, when comparing their results with tailored solution written in C or C++, find themselves 5-181 times slower [10] even though the strategies of computing are the same. Our grouping is only slightly outperformed by the hand-coded variants.

Query caching is also reported by MonetDB [12].

![Figure 2: Processing times of a floating-point tile with repeated multiplications applied](image)
processing-intensive query fragments. Based on 1-D through 4-D sample data mainly taken from the domain of Earth Sciences, queries applying different operators will be compared. Demonstration visitors can experiment with own query ideas. This way, the impact of JIT optimization can be studied interactively.

The following example illustrates this. It relies on a false color image where the three usual bands do not contain red/green/blue, but near-infrared/red/green. The Normalized Difference Vegetation Index (NDVI) derives a weight \( V \) to each pixel \( p \) according to

\[
V(p) = \frac{(p\text{.red} - p\text{nir})}{(p\text{.red} + p\text{nir})}
\]

This value is close to 1 for vegetation and, hence, can be classified as shown in Figure 3. For a threshold value of .6 the corresponding rasql query is

```sql
select (bool) (p\text{.red}-p\text{nir})/(p\text{.red}+p\text{nir}) > 0.6
from FalseColorImages as p
```

While the unoptimized query tree requires five nodes and, hence, five iterations over each tile, the JIT enabled version can perform evaluation in one run, benefiting also from better CPU cache utilization.

![false color satellite image](left) and thresholded vegetation detection (right).

Figure 3: False-color satellite image (left) and thresholded vegetation detection (right).

7. CONCLUSION

Array processing in databases bears close resemblance with vector-parallel supercomputer algorithms. Hence, it seems promising to investigate into further techniques which can be adopted, always with the caveat that array DBMSs convey a more dynamic incoming query behavior.

We have presented a technique which is particularly effective in situations with many operators and large data volumes processed. While it has the potential for speeding up such queries substantially, even worst-case queries are not slowed down. JIT integrates smoothly with rasdaman’s tile streaming approach. However, array joins where objects with incompatible tiling schemes are combined present a limitation currently.

JIT itself still needs a thorough performance evaluation under various aspects, such as tile size impact, interplay of logical and physical optimization, reuse percentage under realistic workloads, etc. To this end, JIT-enabled servers will be installed under operational conditions on large-scale data sets to gain further real-life experience.

8. ACKNOWLEDGMENTS

Kudos go to all rasdafa lots who have contributed since inception, and especially to Constantin Jucovschi for his JIT coding. Research funded by ESA as part of the VAROS project.

9. REFERENCES