JACOBS UNIVERSITY

Peter Baumann

Object-Oriented or Object-Relational? An Experience Report from a High-Complexity, Long-Term Case Study

Technical Report No. 20
July 2010

School of Engineering and Science

Object-Oriented or Object-Relational?

An Experience Report from a High-Complexity, Long-Term Case Study

Peter Baumann

School of Engineering and Science
Jacobs University Bremen gGmbH
Campus Ring 1
28759 Bremen
Germany
E-Mail: p.baumann@jacobs-university.de
`http://www.jacobs-university.de/`

Summary

The discussion between object-oriented and object-relational DBMS technology seems to be decided since some time, in favor of the second candidate. We unroll this question based on our experience with the design and implementation of the array DBMS *rasdaman* which offers storage and query language retrieval on large, multi-dimensional arrays such as 2-D remote sensing imagery and 4-D atmospheric simulation results. This information category is sufficiently far from both relational tuples and object-oriented pointer networks to achieve a ”fair” comparison where no approach has an immediate advantage.

The *rasdaman* system is implemented in a strictly object-oriented manner. We discuss *rasdaman* on model, interface, and implementation level and contrast our experience with concepts and concrete systems of object-relational technology. To underpin and justify *rasdaman* design decisions we also present *rasdaman* performance results. The *rasdaman* system is in operational use and available in open source, so our results can easily be reproduced.

Contents

1 Introduction

Ever since object-relational database systems have entered the scene to compete with (and, on the long run, commercially outperform) object-oriented DBMSs the discussion is ongoing on which approach is better fulfilling the promise of flexibly adding user-defined structures and operations to databases. Meantime both technologies are around since more than ten years, hence can be considered mature enough for a fair comparison. Object-relational evangelists [29] have reported success stories in manifold non-standard situations, emphasizing both functionality and performance of the approach [7][30][10][12]. Likewise, however, up to today proponents of the object-oriented field continue making a convincing case for versatility and performance of such databases [28][11][18][25].

With this paper we want to contribute a new facet to this discussion. We unroll the question based on our experience with the design and implementation of the array DBMS *rasdaman* which offers storage and query language retrieval on large, multi-dimensional arrays. This information category is sufficiently far from both relational tuples and object-oriented pointer networks to no not give any of the approaches an immediate advantage; hence, we believe it constitutes a ”fair” comparison ground.

In fact, no general-purpose DBMS, be it relational, object-relational, or object-oriented, supports the concept of large, multi-dimensional arrays. Such arrays appear in OLAP applications – which is not our focus here – and in technical/scientific applications [16][4][6][21][19][2][20]. 1-D arrays resemble timeseries as encountered in geo sensor webs and life science lab measurements. The most common 2-D data type is the raster image, as encountered in large volumes, e.g., in remote sensing; microarray data in life science form another relevant class. By building time series over 2-D data we end up with 3-D image time series. In ocean and atmospheric simulation the full spectrum of spatio-temporal dimensions is leveraged, ending up with 4-D data. This is not the end, though. Scientific applications frequently make use of non-spatio-temporal dimensions, such as pressure, spectral frequencies, simulation-related additional time axes, etc. This commonly leads to 5-D and higher. Statistical databases with many more such domain-dependent dimensions, to drive the story further, often range from nine to twelve dimensions, and high-dimensional feature spaces can encompass thousands of dimensions.

A special case occurs when the array contains values at only very few positions; such arrays are called ”sparsely populated” as opposed to ”densely populated” arrays like images where every pixel contains a color value. For sparse arrays, typically up to a data content of about 5%, and their main query types (”OLAP” = Online Analytical Processing) both dedicated (”MOLAP” = Multi-dimensional OLA) and relational technology (”ROLAP” for Relational OLAP) is in use. None of this technology is suitable for handling, say, multi-Terabyte dense arrays.

Systems offering dedicated support for such arrays are called *array DBMSs*. They pose quite some challenges on tuple-based technology due to the substantially different structure and operations. While typically several tuples fit into one database page, the converse is true for arrays: one array normally requires a large number of database pages, which may well range into the millions.

The *rasdaman* system is such an array DBMS, offering support for both sparse and dense multi-dimensional arrays. Development started at the heydays of object-oriented databases, and so it was natural to use pertaining concepts, standards [9], and technology wherever possible. A few years later object-oriented DBMSs more and more disappeared, and *rasdaman* was adapted, over a number of years and depending on demand, to a number of relational DBMSs. Today, both an open-source (see www.rasdaman.org) and a commercial variant (see www.rasdaman.com) are available and in active use, mainly in geo applications. EarthLook (www.earthlook.org) is a publicly accessible demo site of *rasdaman* .

In this paper we attempt a comparison of design decisions and their effects by focusing on the object-oriented paradigm (which has been used by *rasdaman*) and object-relational technology. The latter we pick because we have been asked many times why we do not use (or switch to) the object-relational approach which was perceived as superior by its proponents. To this end, we inspect conceptual model, interfaces, and implementation of *rasdaman* . The remainder of this paper is organized as follows. In Sections 2, 3, and 4, array modeling, interfaces, and server implementation are addressed, respectively. Relevant performance results are presented in Section 5. Finally, Section 7 gives conclusions and outlook.

2 Array Modeling and Retrieval

Arrays represent functions from some Euclidean space (its *domain*) to a value set (its *range* [16][6][19]. Array domains must have axis-parallel boundaries. At each array position, a *cell* is located which holds a range value. Arrays are typed, and *collections* (the ODMG correspondent of tables) hold arrays of a particular type. Actually, *rasdaman* uses specialized collections where one attribute is a system-generated OID and the other one is an array. Cell, array, and collection types are defined at server runtime.

Arrays are stored partitioned into multi-dimensional subarrays called *tiles* whereby each tile goes into a BLOB (binary large object). This partitioning can be chosen arbitrarily, controlled by a storage layout language which is part of the `insert` statement. Further, *rasdaman* maintains some tables for catalog information, indexes, and the like. Storage overhead for all these is negligible in face of the large array objects themselves.

In the remainder of this section we inspect array type definition, engine implementation, and APIs with a focus on comparing object-oriented and object-relational concepts.

2.1 Array Type Definition

We recall that an array, being a function, is a higher-order construct. Similar to a set, a stack, etc., an array is not a mere data type, but a parametrized data type constructor ("templates" in programming languages like C++) whose operators are functionals. All of these have in common that they need to be instantiated with the type of the values they hold; in the case of arrays, instantiating them requires provision of a cell data type and a domain extent.

In *rasdaman* , a type definition language is provided which is based on ODMG's Object Definition Language, ODL [9]. The extension mainly is a template type which needs to be instantiated with a C/C++ like atomic or composite data type and a domain. "Open" bounds (i.e., runtime-varying limits) can be defined on each dimension's lower or upper bound. A typical definition might look like this, it introduces $10,000 \times 10,000$ `LandsatImages`:

```
typedef marray<
    struct{ unsigned char b1, b2, b3, b4, b5; },
    [0:9999,0:9999]
> LandsatImage;
```

The *rasdl*utility can create C++ header file definitions to sync program and database types.

Object-relational database systems (ORDBMSs) normally allow only runtime definition of data types, not of data type constructors. Hence, ORDBMSs with array support are constrained to a few selected data types; Oracle 11g, for example, supports only 2-D arrays over hand-picked pixe l types while *rasdaman* allows n-D arrays over flexibly defined cell types. One notable example to this is Predator [23] – however, at the expense of server-side programming so that, in the end, Predator offers some selected hand-programmed array data types, but no general array concept nor array type runtime definition via DDL like in *rasdaman* .

2.2 Array Retrieval

We introduce *rasql*only briefly, with a walk-through guided by didactics rather than by completeness; see [24] for a more comprehensive and formal treatment. Like SQL, a query returns a set of items (in this case: either arrays or, in summarization queries, scalars). Subsetting of arrays includes *trimming* (rectangular cutouts) and *slicing* (extraction of lower-dimensional sub-arrays). The following query retrieves a 2000x3000 cutout (trim) from every Landsat satellite image in collection `LandsatImages`:

```
select ls[ 1001:3000, 1001:4000 ]
from   LandsatImages as ls
```

This works for any number of dimensions, and also allows wildcards to refer to the array boundaries. For each operation available on the raster cell type, a corresponding so-called *induced operation* is provided which applies the operation to all cells of an array simultaneously. Both unary operations (e.g., record access, cast operations, or constant multiplication for contrast enhancement) and binary operations (e.g., masking an image) can be induced. An example is "*Band 3 of all Landsat images, with intensity reduced by a factor of 2*":

```
select (char) (ls.band3 / 2)
from   LandsatImages as ls
```

Not only arithmetic, but also boolean operators can be induced. The query below masks out all non-green pixels from a Landsat remote sensing image; the boolean result of the parenthesis expression is nterpreted as 0 or 1, resp., so that a *false* value maps to black and a *true* value retains the original color value:

```
select (ls.green > 130 and ls.red < 110 and ls.blue < 140) * ls
from   LandsatImages as ls
```

In general, array expressions can be used in the `select` part of a query and, if the outermost expression result type is `boolean`, also in the `where` part. Therefore, we need a means to "collapse" raster-valued expressions, such as induced comparisons, into scalar boolean values. This is accomplished through so-called *condensers* which summarize over the array values, thus representing the counterpart to SQL aggregation. Based on a higher-order generalized construct the usual aggregates are defined, such as `count_cells`, `average_cells`, `min_cells`, `max_cells`, `all_cells`, and existence quantifiers. For example, the following query retrieves the "*percentage of green area per region in Landsat scenes*".

```
select   count_cells( ls.green > 127 and r )
       / count_cells( ls and r )
from   LandsatImages as ls, Regions as r
```

The general second-order condenser allows to define `add_cells`, etc., as in this example:

```
cond +
over x in [0:255], y in [0:255]
values ls[x,y]
```

Another second-order operator, `marray`, constructs a new array with some given extent and a contents determined by a scalar expression used to assign a value into each raster cell position. To this end, it defines locally scoped cell coordinate iteration variables. For example, a 256x256 checker-board array can be defined as

```
marray x in [0:255], y in [0:255]
values mod( x + y, 2 )
```

In practice, `marray` frequently appears in conjunction with condensers. The following example illustrates this, deriving a histogram from 16-bit brain scans of unknown dimension:

```
select marray n in [0:16535]
       values count_cells(b) = n
from   BrainScans as b
```

Advanced applications of this pattern include filter kernels and general convolutions, up to the Fast Fourier Transform. Interesting parallels also can be found with the relational `group by` / `having` clause. A theoretical analysis of the expressiveness and complexity of array indexing expressions is given in [20].

Let us turn to object-relational systems. While they allow definition of new data types they generally do not support definition of parametrized data types (”templates” in object-oriented languages like C++), that is: second-order constructs. Just like stacks or lists, which have to be instantiated to obtain a concrete data type, arrays are parametrized with cell type, dimension, and extent as we have seen with *rasdl*. Consequently, all object-relational implementations of large arrays uniformly offer only a selection of hardwired array types. 2-D greyscale and RGB arrays are common, with sometimes further concrete types added.

Predator [27] is a database system supporting ”Enhanced Abstract Datatypes” (E-ADTs). In theory, the high degree of extensibility in Predator should make array support feasible. Still, the array support implemented by the Predator group only offers selected array types, not a generic array constructor. This observation is supported by the fact that the R*-trees implemented only support 2-D arrays. Further, the effort of writing an E-ADT seems prohibitively high for most practical purposes.

Another research prototype in this field is Paradise [13][14]. Arrays are supported indeed, even with a comparatively simple tiling scheme for partitioned array storage. However, the extension of SQL with only functions, but not second-order entities like arrays, precludes an easy-to-read raster query language. For example, 2-D raster images come with only three different cell types: 8 bit, 16 bit, and 24 bit. The set of operations supported focuses on simple cutout or subaccess operations; even access to components of composite cells is nontrivial. As with Predator, Paradise developers report a significant learning curve for familiarization [8]: "One reason for this is that too much expertise ended up being required to use them". In *rasdaman* , conversely, *rasdl*allows to define new array types during runtime using a simple C-style language.

3 Array DBMS Programming Interfaces

A central claim of ODBMSs is their seamless coupling with object-oriented programming languages. At the time *rasdaman* development started a number of ODBMSs was on the market which, however, had very divergent and absolutely incompatible APIs. Any decision for a particular system, therefore, would involve a total commitment to the product. There was a standard, ODMG [9], which, however, was supported seriously by only one product, O2 [3], although other vendors claimed that, too. Actually, the lack of uniform interfaces allowing implementers to switch from one product to another without major application redesign arguably is one of the major drawbacks of ODBMS technology overall. Given the criticality of this decision the *rasdaman* team performed a thorough evaluation and, ultimately made its decision in favor of open standards and, hence, O2.

3.1 The C++ API

The *rasdaman* C++ API, which is conformant with ODMG, consists of two packages: `raslib`and `rasodmg`. The `rasodmg`package implements all ODMG provided classes, such as for databases, transactions, collections, result iteration, and the type system, while in the `raslib`package all array specific classes are gathered, such as arrays, format converters, storage layout and tiling definition.

At the very core of `rasodmg`is sending queries and receiving results, for which ODMG foresees the `oql_execute()` function which is freestanding, that is: it does not belong to any class. As, at the time of crafting *rasdaman* , namespaces where not yet known in C++ world this concept had to be emulated and so all `raslib`functions are prefixed with `r_`, such as in `r_oql_execute()`.

Instances of class `r_OQL_Query` represent *rasql*queries. Following the set-oriented paradigm of ODMG (which it, in turn, borrows from the relational model) a query result consists of a set of persistent pointers, depending on the query result type either to scalar values or to arrays. To correctly handle query results depending on their type, a class hierarchy for runtime typing is included. For iteration over query results the template class `r_Iterator<T>` is provided.

Class `r_Marray<T>` is the central class representing with multi-dimensional arrays (Figure 1). This template is instantiated, depending on the query result, with either a C++ primitive type or a user-defined structure. Spatial domains are provided to instances; to achieve flexibility for danymically varying arrays this part of the second-order array construct is not modeled via templates. For accessing and manipulating arrays the template class a rich, high-level interface is provided with `raslib`.

Among the further functionality provided by is an overloaded array access operator, `operator[]` for trimming and slicing of objects. In the special case that this operator receives only a single point coordinate it coincides in semantics with the usual C/C++ single-element array accessor.

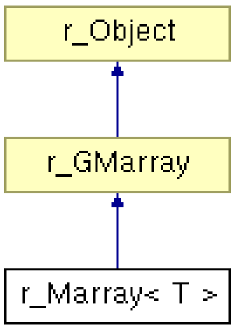

Figure 1: C++ API class hierarchy for arrays

The following code snippet retrieves a result set containing three elements, as sample collection `mr` - as provided in the *rasdaman* distribution - contains three objects. The `r_OQL_Query` constructor serves to establish a query object which subsequently is shipped to the server by means of `r_oql_execute`. This method accepts a query object and returns a set of smart pointers of some type not specified at compile time – this reflects the fact that the result type is constructed on the fly, depending on the actual query.

```
r_OQL_Query myQuery =
    new r_OQL_Query( "select (char)sqrt(mr) from mr" );
r_Set<r_Ref<r_GMarray>> rasterSet;
r_oql_execute(query, rasterSet);
void r_oql_execute( r_OQL_Query query, r_Set<r_Ref_Any> result);
```

Result arrays obtained thereby can be readily processed in C++. In the example below we first declare an iterator over the result set, in our case 2-D greyscale images; this type is known because the *rasdl*operator has created an include file with the pertaining definitions. Array data are delivered in the C++ memory layout and endianness of the client's CPU, a service provided by the client/server communication layer of *rasdaman* which performs translation where necessary. Therefore, we can directly print out, for each image delivered, all of its pixel values:

```
r_Iterator<r_Ref<GreyImage>> iter = rasterSet.create_iterator();
for (iter.reset(); iter.not_done(); iter++)
{
    GreyImage myArray = (GreyImage) r_Ref<r_GMarray>(*iter);
    int loX = myArray->spatial_domain()[0].low(),
        hiX = myArray->spatial_domain()[0].high(),
        loY = myArray->spatial_domain()[1].low(),
        hiY = myArray->spatial_domain()[1].high();
    cout << endl << "pixel data: " << hex;
    for (int x = loX; x<= hiX; x++)
        for (int y = loY; y <= hiY; y++)
            cout << myArray->get_array()[x,y] << " ";
}
```

As can be observed, data transition between database and application code is smooth and respects all C++ mechanisms like templates, overloading, and correct data representation. The principle shown readily generalizes to accessing arrays of any dimensionality and any cell type. This is just half of the story, though. Client-side utility functions like the overloaded array operator `[]` have turned out an asset making application development much easier[1].

Compare this with the code for retrieving a GeoRaster object from an Oracle database. The sample code shown has been taken from an Oracle White Paper [22]. It is written in PL/SQL to retrieve a 2-D raster cutout:

```
declare
g sdo_georaster;
b blob;
begin
select raster into g from uk_rasters where id = 4;
dbms_lob.createTemporary(b,true);
sdo_geor.getRasterSubset(
georaster => g,
pyramidlevel => 0,
window => sdo_number_array(0,0,699,899),
bandnumbers => '0',
rasterBlob => b );
end;
```

Roughly, this query corresponds to a *rasql*rewrite as follows:

```
select g.0 [0:699,0:899]
from   uk_rasters as g
where  oid(g) = 4
```

The main difference is not in the syntactic complexity of the Oracle code; much more important, raster data have to be shuffled over to C++ prior to using them in the way shown with `raslib`. Aside from expected performance difficulties this leaves client-side coding in the responsibility of the programmer, without honoring object-oriented programming and without security support like include file generation. Such mechanisms, however, are not foreseen in object-relational interfaces.

3.2 The Java API

For reasons of completeness, let us briefly glance at the *rasdaman* Java interface, `rasj`. In C++, the second-order array construct is mapped straightforward to templates. As these do not exist in Java, a substitute had to be found. The "implementation" feature of java turned out helpful to derive array types from a generic class, `RasGlobalDefs`. Below is an excerpt from the javadoc generated class hierarchy:

```
class java.lang.Object
+--class rasj.RasImplementation
| | (implements org.odmg.Implementation)
| +--class rasj.RasODMGInterface
```

[1]these `raslib`functions are also heavily used in the server code.

```
+--class rasj.odmg.RasObject
   | (implements rasj.RasGlobalDefs)
   +--class rasj.RasGMArray
   | (implements rasj.RasGlobalDefs)
   +--class rasj.RasMArrayByte
   | (implements rasj.RasGlobalDefs)
   ...
   +--class rasj.RasMArrayFloat
     (implements rasj.RasGlobalDefs)
```

In this hierarchy, a separate subtype exists for every cell type. Atomic types are prefabricated, other ones have to be created by the programmer. Obviously, the fact that the implementation class strategy cannot capture cell type parametrization of the array construct is a severe drawback as compared to C++.

On instance level, however, handling is relatively similar to C++. Just as with the `raslib`and `rasodmg`twins encountered with C++, *rasdaman* Java applications need to import `rasj.*` and `org.odmg.*`. In the code example below note that the Java API is more serious in its naming, it reveals that the result collection is a bag rather than a set.

```
OQLQuery myQu = myApp.newOQLQuery();
myQu.create( "select (char)sqrt(mr) from mr" );
DBag resultSet = (DBag) myQu.execute();
Iterator iter = resultSet.iterator();
while (iter.hasNext())
{
    RasGMArray result = (RasGMArray) iter.next();
    byte[] pixelfield = result.getArray();
}
```

3.3 The Language Decision Revisited

C++ has shown significant advantages as server implementation language: a direct translation of the object-oriented design into classes; maintainability and modularization elasticity during experimentation phases involving substantial redesign of this or that component; and all this coming with highly efficient code.

On the other hand, the team also faced severe problems. Most of all, at the time implementation started C++ compilers were far from being stable. Templates, which are extensively used in the *rasdaman* code, caused compilers to simply crash or come up with surprising ideas. Sometimes compile times reached geologic scales. Further, there was no standardization across platforms, and STL (the C++ Standard Template Library) was just another library and *in statu nascendi*, far from being canonical.

This is different today. At the time of this writing C++ programming has matured, except maybe from small surprises with the GNU C++ compiler when a minor release change brings along cinompatible changes so that the source code has to be adapted. Overall, a strictly object-oriented design as well as higher level concepts such as patterns and idioms has proven advantageous.

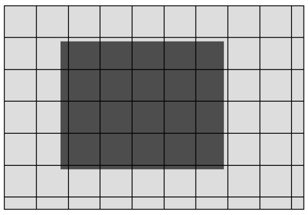

Figure 2: Query box partially overlapping border tiles

4 Array Engine Implementation Aspects

The core design goals of *rasdaman* appear not so different from other large-scale information system projects:

- Portability between different operating systems (Solaris, HP-UX, Irix, AIX, Linux) and base DBMSs (initially O2 and now PostgreSQL, MySQL, Oracle, IBM DB2, IBM Informix).
- Code maintainability and extensibility, as the system is continuously being improved and extended.
- Modularisation with a concise, easy to use interface to enable parallel development of different modules.
- Efficiency in dealing with huge amounts of data and concurrent access.

4.1 An Architecture for Array Query Evaluation

The *rasdaman* architecture overall follows a classical client/server DBMS approach, however, with each component engineered specifically towards array processing. Client APIs in C++ and Java represent the stubs for submitting queries. On server side, queries are unmarshalled, parsed, optimized, and executed, based on auxiliary modules like catalog manager, index manager, cache and transaction manager, and a persistent storage manager. The latter actually is an adapter, it mainly is in charge of mapping storage accesses to the underlying DBMS (called ”base DBMS”). In the case of O2 this was as straightforward as can be – the smart pointer to the object (such as a tile, a catalog descriptor, etc.) was passed to O2 which guaranteed to write through until commit time. For the relational adapters, the adaptation code had to be written following the well-known tuple access pattern. This made the storage manager code effectively grow by a factor of 2.5 to 3. A complication arose because several relational systems do not include BLOB handling in their SQL, but require extra API programming. This has led to another 20% to 30% code growth, as can be seen, e.g., in module `relblobif`.

Query processing follows the usual steps of parsing, syntactic and semantic checking, optimization, and query tree execution. The latter is based on the paradigm of *tile streaming*: each node reads a tile, processes it, and passes a result tile or scalar on to the next higher node. Only a few operators are blocking, such as the `scale` operation. Of particular interest in the context of this paper is query operator evaluation, on this we will concentrate next.

Class `Tile` represents tiles. Its methods know how to process a tile, and, hence, these are invoked during operator tree evaluation. In the simplest case of access tile data are just copied.

No knowledge about tile extent and cell type is necessary, a byte copy turns out sufficient. Cell type information becomes necessary whenever unary operations are to be executed, such as `abs` or `ln`. Still, iteration can be kept straightforward by incrementing the target pointer in units of cell size, regardless of the tile's dimension. Common in real life, however, is partial access to a tile as subsetting limits frequently do not align with tile borders (Figure 2). In this case, a multi-dimensional iteration is required indeed. Interestingly, it turns out that a naive implementation of tile iteration via nested loops has severe disadvantages. The number of loops nested is fixed during compile time, and it is not straightforward to implement nesteds loops of arbitrary depth with a decision at runtime. During implementation of a mechanism allowing for d-dimensional dynamic iteration it further turned out that the loop housekeeping activities cause a substantial overhead. Widmann [31], therefore, has proposed a method below which allows to efficiently execute d-dimensional loops with d chosen at runtime: Performance analysis using a profiler

Algorithm 1 TILEITERATION

```
Require: A binary function $f$ with non-empty input tiles $T_1, T_2$ and result tile $T_{res}$ all
    of which have the same extent.
 1: initialize tile iteration vectors: $x_1 = get\_origin(T_1), x_2 = get\_origin(T_2), x_{res} =$
    $get\_origin(T_{res})$
 2: initialize current dimension inspected: $d_{curr} = 0$
 3: while $d_{curr} < dim(T_{res})$ do
 4:     execute cell operation by addressing via the offsets obtained
 5:     increment position in $x_1, x_2, x_{res}$ for dimension $d_{curr}$
 6:     if $x_{res}$ exceeds upper bound of extent in $d_{curr}$ then
 7:         reset position in current dimension $d_{curr}$ to lowest index
 8:         $d_{curr} += 1$
 9:     end if
10: end while
```

reveals that most of the time is spent in calculating the offsets in step 4. Increment computation can be improved greatly by providing incorements a priori in a setup phase, and also by avoiding multiplications.

Assuming linearised storage of array A, the one-dimensional distance in cells between vector $x = (x_0, \ldots, x_i, \ldots, x_{d-1}$ and $x' = (x_0, \ldots, x_i + 1, \ldots, x_{d-1}$ is a constant integer value. If linearisation is done along dimension 0 first, this increment vector $\Delta = (\Delta_0, \ldots, \Delta_{d-1}$ can be predetermined. The offset in bytes, then, is obtained by multiplying the Δ_i with the length of the cell type in bytes. As iteration now works directly on the offset the costly offset computation inside the loop is avoided. Still, however, invocation to `r_Minterval` and `r_Point` are required to check the iteration boundaries. We, therefore, repeat the principle of precalculation and determine the number of repetitions ahead of the loop, which is given by $r_i = hi(dom(T(, i) - lo(dom(T), i) + 1$ per dimension i. All this is combined into a modified version of our iteration vector, Δ', which now holds, for each dimension, the total number of repetitions, the increment per repetation (as given by the linearisation function), and the current repetition.

One key target in the design of the *rasdaman* system was support for user defined composite base types in the DBMS. Structured base types are common when looking at the typical application areas of multidimensional arrays. Examples are colour images with a red, a green, and a blue component or remote sensing images containing different wavelength bands.

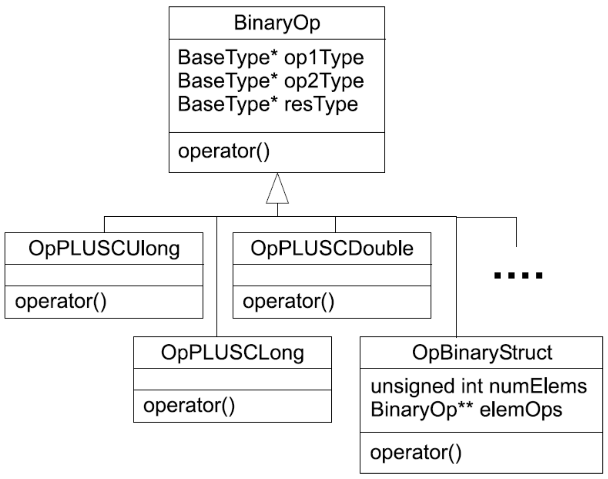

Figure 3: Class hierarchy for binary function objects

With a server knowing about the substructure of a cell it can offer extraction of components, as well as dynamic recomposition. Not only is this convenient for the users of such a system, it moreover bears excellent potential for better bandwith exploitation. Read the last one as: better performance. By transmitting only one band of interest out of a 255-band Hyperion hyperspectral satellite image data set the gain in transmission speed (which turns out to be one of the overall performance determining factors in array DBMSs, see Section 5) is immediately visible. Recombination, on the other hand, can save processing and transmission time – think of the same hyperspectral satellite image where one single image can be retrieved by the client which may consist red, green, and blue bands in case of the visible spectrum or near-infrared, red, and green for false-color images.

Earlier we had stated that object-relational mechanisms lack the necessary data type constructors. Now we encounter one point where this gets manifest in implementation. To make operation selection separate from operation execution, a function which returns a function is needed; with its *function objects* C++ supports a direct implementation of such a design pattern. Function objects are classes with an overloaded `operator()` allowing objects of these classes to be used in a function call syntax, thereby blurring the distinction between a function and an object. This allows for a very clean and comprehensible syntax for applying operations on cells, as demonstrated by the following example which describes part of the `execBinaryOp()` method internals in pseudo code. Inside the method, a typical invocation of the cell evaluation function `myOp` might look like this pseudo code:

```
BinaryOp* myOp;
myOp = Ops::getBinaryOp( op, resType,
    op1Tile->getType(), op2Tile->getType() );
for each result cell cellRes in the result tile area:
    (*myOp)(cellRes, cellOp1, cellOp2);
```

In the first line, the operation is identified; due to the overloading of the query language operators this lookup is based on the complete operation signature. The parser delivers the `opCode` as an enumeration value while input and result types of the operation are provided in `op1Tile-> getType()`, `op2Tile->getType()`, and `resType`, resp. Subsequently, execution iterates over the result array area identified and inspects each cell by calling function pointer `*op` passing pointers to the result cell location, `cellRes`, and the input operand cells, `cellOp1` and `cellOp2`. For each of the previously discussed categories - unary, binary, and condense operations - there is one type of function in class `Tile`, represented by abstract classes defining the basic cell evaluation operations. Figure 3 shows that excerpt of the class hierarchy which is in charge of binary induced operations. Method `operator()` is central to this modeling. In `BinaryOp` it remains abstract, a variety of subclasses implement the concrete query operators for different type combinations.

Using abstract base classes as the interface opens up freedom for operation implementation: by relying on the type system's conversion functions it is easy to implement an operator with broad applicability over all the types supported. By providing specialized classes for particular type combinations, important use cases can be tuned individually. Internal complexity is effectively shielded from both query evaluation and operation execution and only relevant locally, in the Catalog Manager module. For example, introducing a new primitive type minimizes code changes required in the execution engine. This has proven advantageous, for example, when complex numbers have been introduced into the *rasdaman* type system.

This function object idiom can be used again when it comes to operations on composite cell types. The function objects for operations on structures retrieve function objects for each structure component and then simply apply these objects on all elements. Hence, it is not necessary to implement function objects specifically for each user defined structure.

So far we get the impression of quite compact code. However, there is a drawback given by sheer combinatorial multitude of signatures. ODMG defines nine primitive types [9], and each binary operation has the choice of type for two input types and one result type. This yields a total of 9,477 possible operation signatures. While a dedicated implementation for each input type combination certainly would lead to an extremely efficient implementation it is utterly impractical to have that many classes – even if generated automatically the consequence would be huge, unmaintainable code and excessive compile and link times. An alternative is to use a dynamic conversion of operands to the corresponding C++ types during execution.

This is the approach adopted for the *rasdaman* type system. Function objects for all operations are provided for the three basic type categories supported, namely floating point numbers, signed integers, and unsigned integers. During execution both operands and result have to be converted into the highest precision type of the respective category.

The conversion functions are invoked prior to operation execution on the base types of operands and result, which both are stored in the internal state of the function object representing the operation. Using dynamic conversion, implementing the 13 binary operations supported in *rasdaman* requires 39 classes for the function objects. For frequently used types optimised implementations can be provided. Currently this is implemented for 8-bit unsigned integers used, for example, in 24-bit RGB images.

In summary, combining function objects with the object-oriented design separates application of functions to cells from the multi-dimensional iteration. Further, operation selection can be performed independently by encapsulating it in the Catalog Manager. This opens up vistas for optimizing specific cell types without affecting the operation execution module, such as the abovementioned support for 8-bit integers. The virtual function calls occurring with the function object approach can be resolved efficiently by the compiler. Finally, hardcoded support for specific situations, such as greyscale and RGB cell types, can be added in a straightforward

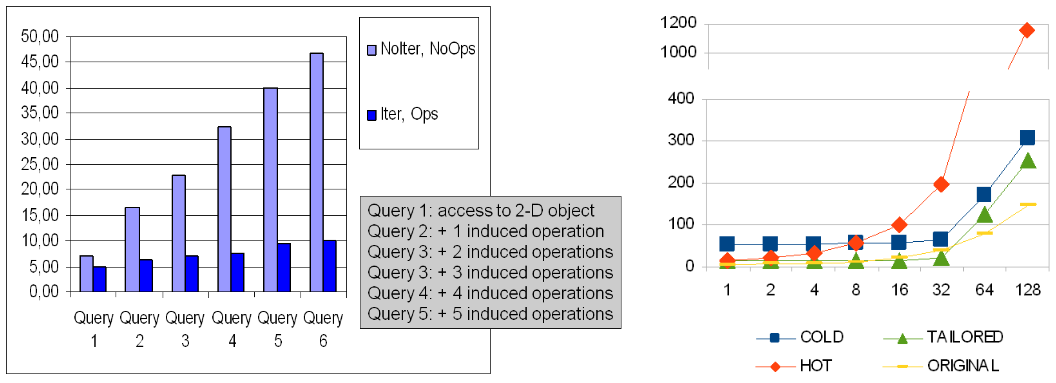

Figure 4: Performance gain through object-oriented engine implementation (left) and advanced optimization (right)

manner by specializing class `Tile`.

5 Performance

Due to space limitations only a brief glance at performance data is possible. The effect of the object-oriented implementation of the *rasdaman* core engine is shown in Figure 4 (left). As can be seen the gain in using function pointers and optimized multi-dimensional iteration over a naive nested loop implementation is substantial as soon as only a few operations are involved in a query.

In Figure 4 (right) the effect of an advanced optimization can be seen, namely just-in-time compilation (JIT) [17]. In this approach, suitable fragments are extracted from incoming queries, transformed into C code, compiled, and dynamically linked into the server. The diagram shows processing time of a 512^2 tile when applying n floating-point multiplications on each pixel. ”ORIGINAL” refers to unoptimized execution, ”COLD” to JIT-based query execution including compilation, ”HOT” to the case that the corresponding code is already linked in from a previous query, ”TAILORED” constitutes a hand-optimized version for comparison.

On www.earthlook.org a publicly accessible showcase with 1-D to 4-D geo-spatiotemporal data sets is available to test *rasdaman* performance under real-life conditions. Typically, even complex rendering queries take abuot 400 ms.

6 Related Work

In the domain of array databases, AQL [19] is one important representative offering a powerful array query language. Its implementation relies on SML, a general-purpose functional programming language, and NetCDF files. RAM [2] implements an array model on top of a relational model and comes with an array query sub-language embedded into SQL. RAM is implemented as part of the MonetDB database system by mapping the algebra to relational algebra, i.e., ”hard-coding” arrays in the core. In particular, object-relational concepts are not used either. AML is an array algebra implemented on top of MatLab for processing of array expressions. None of these system is reported to use an object-relational platform.

Functions which return functions to the best of our knowledge are not supported by OR-DBMSs, hence there is no counterpart to the *rasdl*type definition language and *rasql*second-order functions. Predator [27] and Paradise, two object-relational systems with array support, have been introduced in a previous section. As discussed, they do not offer generic array support, but only hand-picked array types. Additionally, implementation of such data types is reported to be prohibitively complex. The *rasdaman* system, on the other hand, has been reported as the ”most comprehensive implementation” of an array DBMS [20].

7 Conclusions

We have studied application of object-oriented and object-relational techniques for a specific domain where a mapping to relational tuples, as usually done in object-relational modeling, is not feasible. Likewise, object-oriented technology per se does not offer any suitable solution off the shelf. Hence, we feel that this is an interesting situation for investigating flexibility and adaptiveness of both concepts.

Object-orientation has been pursued in both model, interface, and implementation. The array model makes use of existing concepts of the ODMG 3.0 standard; introducing multi-dimensional intervals and arrays appeared possible seamlessly, extending ODMG’s ODL. Client interfaces are crafted along the C++ and Java APIs of the ODMG 3.0 standard. While C++ templates allowed a natural modelling of the array type constructor, the concept of implementation classes offered by Java turned out a good substitute. During implementation of the engine, smart pointers as we encountered them in O2 turned out advantageous in terms of efficiency of programming, runtime efficiency, and flexibility; the latter in particular when it came to an own implementation of smart pointers during the switch to relational systems. The kernel optimizations for multi-dimensional iteration and the dynamic type system could be implemented elegantly and efficiently using the function pointer idiom.

Object-relational technology, on the other hand, comes with some drawbacks. On model level, it allows only introduction of new data types, but not of data type constructors. Consequently, all object-relational systems offering array support are constrained to a few hand-picked, hardwired data types. When it comes to implementation, it has turned out prohibitively tedious to implement new types and, in particular, optimizations for them – a fact reported by the object-relational proponents themselves. In the end, the major part of the code that comprises, e.g., the *rasdaman* engine would need to be reimplemented for such an architecture. We seriously doubt that efficiency would be comparable.

Implementation work reported has been performed between 1995 and 2010 in steps and with several generations of developers involved. Feasibility of the *rasdaman* implementation approach has been proven many times, up to operational installations. For example, the French National Geographic Institute (IGN-F) maintains, since more than five years, a dozen-Terabyte airborne image in a stack consisting of (from bottom to top) PostgreSQL, rasdaman, and a servlet implementing an OGC Web Map Service (WMS). EarthLook is a public accessible demonstration site accessible at www.earthlook.org. Query examples drawn from the Earth Sciences showcase 1-D to 4-D retrieval and processing through the raster query language of the OGC Web Coverage Processing Service (WCPS) standard [5]. Further, array service stacks using *rasdaman* have been implemented for gene expression analysis [26] and human brain imaging [15]. Hence, the array DBMS architecture and the ramifications of our design decisions for efficient implementation are quite well understood, which reinforces validity of our results. To the best of our knowledge, this research resembles the largest study ever undertaken in terms of both development and use time (14 years) and project size (around 1.5 million lines of code).

For the future, we plan continue array DBMS research in various aspects. A core issue, given the advanced state of research in array DBMSs, is development of an array benchmark which allows to thoroughly compare different implementations, including vendor-specific optimizations. As for *rasdaman* , one line of research is to investigate further optimizations, such as OLAP-style preaggregation, where we have encounraging first result [1]. Another thread is to extend the array concept to general spatio-temporal objects, including point clouds and triangulated irregular networks (TINs), to name but two. We feel optimistic that the strictly object-oriented design will aid us in adding features to the *rasdaman* engine.

References

[1] Peter Baumann Angelica Garcia-Gutierrez. Computing aggregate queries in raster image databases using pre-aggregated data. In *Proc. ICCSA'08*, pages 84–89, 22-24 October, 2008.

[2] Alex Van Ballegooij, Arjen P. De Vries, and Martin Kersten. Ram: Array processing over a relational dbms, 2003.

[3] Francois Bancilhon, Claude Delobel, and Paris Kanellakis. *Building an Object-Oriented Database System*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[4] Peter Baumann. On the management of multi-dimensional discrete data. *VLDB Journal 4(3)1994, Special Issue on Spatial Database Systems*, 4(3):401–444, 1994.

[5] Peter Baumann, editor. *Web Coverage Processing Service (WCPS) Implementation Specification*. Number 08-068. OGC, 1.0.0 edition, 2008.

[6] Peter Baumann. A database array algebra for spatio-temporal data and beyond. In *Proc. NGITS'99*, volume LNCS 1649, pages 76 – 93. Springer Verlag, July 5-7, 1999.

[7] M. Carey, D. Chamberlin, D. Doole, S. Rielau, N. Mattos, S. Narayanan, B. Vance, and R. Swagerman. O-o, what's happening to db2? *SIGMOD Rec.*, 28(2):511–512, 1999.

[8] Michael J. Carey and David J. DeWitt. Of objects and databases: A decade of turmoil. In *Proc. VLDB'96*, 1996.

[9] Rick Catell and R. G. G. Cattell. *The Object Data Standard*, 3.0 edition, 2000.

[10] Vo Thi Ngoc Chau and Suphamit Chittayasothorn. A temporal object relational sql language with attribute timestamping in a temporal transparency environment. *Data and Knowledge Engineering*, 67(3):331–361, 2008.

[11] Vincent Coetzee and Robert Walker. Experiences using an odbms for a high-volume internet banking system. In *Proc. OOPSLA'03*, pages 334–338, 2003.

[12] Shirley Cohen, Patrick Hurley, Karl W. Schulz, William L. Barth, and Brad Benton. Scientific formats for object-relational database systems: a study of suitability and performance. *SIGMOD Records*, 35(2):10–15, 2006.

[13] David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel, and Jie-Bing Yu. Client-server paradise. In *Proc. VLDB'94*, pages 558 – 569, 1994.

[14] Jingesh M. Patel et al. Building a scalable geospatial database system: Technology, implementation, and evaluation. In *Proc. ACM SIGMOD'97*, 1997.

[15] Per Roland et al. A database generator for human brain imaging. *Trends in Neurosciences*, 24(10):562 – 564, 2001.

[16] Trenchard More Jr. Axioms and theorems for a theory of arrays. *IBM Journal of Research and Development*, 17(2):135–175, 1973.

[17] Constantin Jucovschi, Peter Baumann, and Sorin Stancu-Mara. Speeding up array query processing by just-in-time compilation. In *Proc. SSTDM'08*, pages 408 – 413, December 15, 2008.

[18] Jiang Li. Hierarchical land cover information retrieval in object-oriented remote sensing image databases with native queries. In *Proc. 45th ACM-SE*, pages 467–472, New York, NY, USA, 2007. ACM.

[19] Leonid Libkin, Rona Machlin, and Limsoon Wong. A query language for multidimensional arrays: Design, implementation, and optimization techniques. In *Proc. ACM SIGMOD'96*, pages 228–239, 1996.

[20] Rona Machlin. Index-based multidimensional array queries: safety and equivalence. In *Proc. ACM PoDS'07*, pages 175–184, 2007.

[21] Arunprasad P. Marathe and Kenneth Salem. A language for manipulating arrays. In *Proc. VLDB'97*, pages 46–55, 1997.

[22] n. n. Oracle spatial 11g raster georaster. www.oracle.com/technology/products/spatial/pdf/11g/spatial-11g-georaster-whitepaper.pdf. accessed March 13, 2009.

[23] n. n. Predator object-relational database system. www.distlab.dk/predator. accessed March 13, 2009.

[24] n.n. *rasdaman query language guide*, 8.1 edition, 2009.

[25] Nezihe Burcu Ozgura, Murat Koyuncub, and Adnan Yazicia. An intelligent fuzzy object-oriented database framework for video database applications. *Fuzzy Sets and Systems*, 160(15)2009:2253 – 2274, 2009.

[26] Andrei Pisarev, Ekaterina Poustelnikova, Maria Samsonova, and Peter Baumann. Mooshka: a system for the management of multidimensional gene expression data in situ. *Information Systems*, 28:269–285, 2003.

[27] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The case for enhanced abstract data types. In *Proc. VLDB'97*, pages 66 – 75, 1997.

[28] Jim Smith, Sandra Sampaio, Paul Watson, and Norman W. Paton. The design, implementation and evaluation of an odmg compliant, parallel object database server. *Distrib. Parallel Databases*, 16(3):275–319, 2004.

[29] Michael Stonebraker, Dorothy Moore, and Paul Brown. *Object-Relational DBMSs: Tracking the Next Great Wave*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[30] Juan M. et al. Vara. Model transformation for object-relational database development. In *Proc. SAC'07*, pages 1012–1019, 2007.

[31] Norbert Widmann. *Efficient Operation Execution on Multidimensional Array Data*. Phd thesis, TU Muenchen, 2000.