Efficient Execution of Operations in a DBMS for Multidimensional Arrays

Norbert Widmann[1], Peter Baumann
Bavarian Research Centre for Knowledge-Based Systems (FORWISS)
Orleansstr. 34, D-81667 Munich, Germany
{widmann, baumann}@forwiss.tu-muenchen.de

Abstract

In the RasDaMan project a database system for management of multdimensional arrays is being built. It offers a declarative query language extending SQL-92 with operations on arrays of arbitrary base types and a C++ programming interface. Integrating arrays in the query language enables the system to process complex queries on high-volume multidimensional data in the database server close to physical data storage. Storage of arrays is done in tiles of arbitrary size. Operations on arrays are transformed into operations on tiles during query optimization and execution. These operations are then executed on tiles loaded from mass storage.

This paper describes the underlying formal model for tile based operations on multidimensional arrays and its efficient implementation in C++ as part of the RasDaMan system.

1. Introduction

Multidimensional arrays have been used in scientific computing for a long time as a data structure for modeling and analysing scientific phenomena. General database technology, however, still focuses mainly either on structured data for business applications modeled as records of simple base types like numbers or on unstructured strings ignoring arrays as a data structure. In conventional DBMSs large amounts of data can only be stored as references to external files loosing concurrency control and recovery or as unstructured binary large objects (BLOBs). In this form, the semantics of a multidimensional array is lost and the DBMS cannot execute queries on the arrays or optimize access. Only the application programs know the interpretation as arrays, which forces them to unnecessarily access data because selection of relevant data can only be done by the application. As the applications usually are clients of a database server, this involves transfer of large amounts of data over a network.

RasDaMan is a DBMS managing array data at a high semantic level. Because arrays are usually seen as a low level programmming language construct closely tied to their implementation as main memory pointers[2] the term "Multidimensional Discrete Data" (MDD) was introduced to the database community [13]. This term stresses the arbitrary dimensionality and separates this quantised information from multidimensional vector data.

RasDaMan supports general MDD functionality independent from specific application areas. Application systems on top of RasDaMan are being implemented for management of geographical and medical data. Besides its traditional use in scientific computing, management of multidimensional discrete data is also important for business applications such as OLAP or data mining.

An MDD is defined by a spatial domain and a base type. The spatial domain specifies the size of the MDD in all dimensions. MDD is more expressive than arrays in most programming languages in specifying the spatial dimension: the dimensional boundaries do not have to start with zero and any lower or upper bound can be defined as being variable. The base type specifies a type for the contents of an MDD. Both the usual primitive types, such as CHAR, FLOAT or ULONG, and user defined structures can serve as MDD base types.

In the RasDaMan system, the RasDL (Raster Definition Language) is used to specify types for MDD. RasDL is an extension of ODMG ODL [7], a standard data definition language for ODBMSs. For retrieval and analysis of MDD the RasDaMan system provides the declarative query language RasQL (Raster Query Language) extending SQL-92 [6] with constructs for specifying operations on MDD or parts thereof.

In RasQL high level operations on MDD can be specified, which are evaluated on the RasDaMan server. These operations include operations to modify the spatial domain

[1] sponsored by the European Commission in the ESPRIT Domain 4: Long-Term Research under grant no. 20073.

[2] "The built-in arrays are a major source of errors – especially when they are used to build multidimensional arrays" [26]

of an MDD and operations on cell values. The condense operation, a second-order construct, iterates over a subset of cells in an MDD, combining all values through an commutative and associative operation. Condensers are useful to retrieve summarized information from an MDD (eg, the sum of all columns) substantially reducing the amount of data transfered over a network. These operations are fully orthogonal to the query language and can also be used in retrieval conditions. A declarative query language on MDD gives the system a large degree of freedom for query optimization.

RasQL and RasDL present a logical view on MDD data independent from the actual physical representation on storage media. To enable efficient processing of queries, tiled storage is utilized [12]. Tiles of an MDD in RasDaMan can be of arbitrary size with the only restriction that all borders of multidimensional tiles have to be perpendicular to one coordinate axes. This opens up possibilities for storage optimization like putting frequently accessed data into one tile to minimize the number of disk accesses.

During query processing, operations on MDD are broken down to tiles. A set of operations provided on tiles forms the basis for executing operation in queries. A formal specification of the interface to operation execution and a clear definition of preconditions for applicability of operations is needed for processing MDD queries. An efficient implementation is important for the overall performance of the system.

The remainder of the paper will discuss implementing operation execution in detail. Section 2 gives a survey of the RasDaMan system, its interface to application programmers and the demands on operation execution. A theoretical foundation for operation execution on tiles is developed in Section 3, forming the basis for a discussion of the implementation in Section 4. Related Work is presented in Section 5 and Section 6 concludes the paper.

2. The RasDaMan System

This Section gives a short description of the RasDaMan client/server system. The system is fully implemented and both server and client are running under HP-UX and Solaris. Windows NT can be used as platform for clients.

2.1 The RasDaMan Architecture

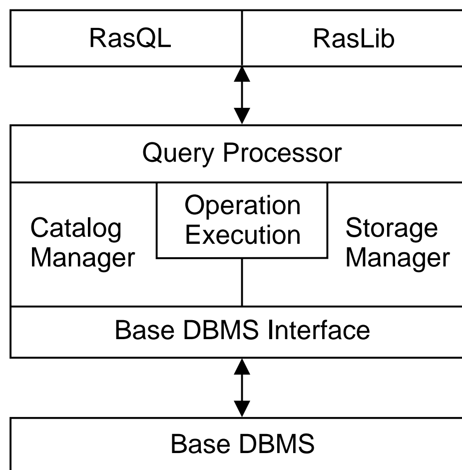

Figure 1. Architecture of the RasDaMan DBMS.

Operation Execution is integrated in the RasDaMan architecture as shown in Figure 1. The client uses the declarative query language *RasQL* and the C++ API *RasLib* to access MDD on the server. The *Query Processor* at the server parses queries and transforms them into operator based query trees. In query trees MDD collections are used as data sources for MDD objects. The actual data is stored on mass storage in tiles. During query processing logical and physical optimizations of the query are done [22]. The *Query Processor* module executes operations specified in the query on tiles using functionality of the *Operation Execution* module.

The *Catalog Manager* is used by the *Query Processor* and *Operation Execution* to retrieve type information on MDD collections, MDD objects and base types of tiles. The *Storage Manager* is responsible for the physical storage of and efficient access to the actual MDD collections, MDD objects and tiles. Both these modules use the *Base DBMS Interface* to store their data in persistent storage. The commercial ODBMS O_2 [1] is used as a base DBMS providing transactions and recovery. The interface to the base DBMS was designed to be very small to minimize the effort involved in changing the base DBMS. All upper layers work with classes implemented independently of O_2.

2.2 RasDL and RasQL

In the following a short survey of the use of RasDL and RasQL will be given using examples. Detailed explanations of RasQL can be found in [4, 5, 27]. The user defines MDD types using RasDL as in the following example:

```
typedef struct {
  unsigned char band1, band2, band3,
                band4, band5, band6,
                band7 } LSPx;

typedef d_Marray<[0:5759,0:7019],LSPx>
  LSImg;
```

Added to OMDG ODL is the possibility to specify the spatial domain of an MDD in C++ template syntax. `LSImg` is defined as a type for MDD of size 5780×7020 to hold Landsat images. The second parameter to the template is the base type which in the example is a user defined structure `LSPx` consisting of seven unsigned 8-bit integer values, one for each Landsat sensor. The RasDL preprocessor imports this information into a RasDaMan database and creates a C++ include file to use these types in client applications.

Once these data definitions are imported into the RasDaMan server, the application program can process MDD utilizing the RasDaMan C++ API. This API, called RasLib, is the ODMG-92 C++ binding extended with classes for managing persistent MDD. The class `r_Marray<>` is a template class for MDD parameterized with the basetype (eg, `LSPx`). Spatial domains are stored as objects of class `r_Minterval` providing multidimensional interval arithmetics. Further classes enable the user to iterate through collections of MDD or to begin, abort, commit transactions. RasLib seamlessly integrates persistent MDD into the C++ programming language.

Declarative RasQL queries can be executed using RasLib and results retrieved as C++ objects. An example query on a collection `myLSImgSet` of Landsat images follows:

```
SELECT img[712:1467,112:968].band7+10
FROM   myLSImgSet as img
WHERE  all(
  img[712:1467,976:1762].band1 > 127)
```

This query in plain language says: "Return band 7 in my area of interest with intensity raised by 10 of all those Landsat images where the intensity of every pixel of band 1 in the area of interest is greater than 127." Only the image areas selected in the query are sent over the network to the client machine. Due to tiled storage only tiles in the area of interest have to be read from mass storage. In the above query this is only about 1.5% of total data and this data is stored closely together on disk in tiles as opposed to being dispersed over the whole MDD if BLOBs would be used.

To evaluate this query, operations have to be applied on MDD. The following operations are executed:

- `[712:1467,112:968]`: select a 2-D part from a 2-D MDD.
- `.band7` and `.band1`: access an element of a structure for each cell of an MDD. The result has the same spatial domain as the original MDD and the type of the accessed element as base type (`unsigned char` for elements `band1` and `band7` of a LSPx structure).
- `+10`: add a constant to every cell of an MDD.
- `>127`: apply this comparison to every cell. The result has the same spatial domain and base type `bool`.
- `all`: condense a boolean MDD to a boolean value. The result is TRUE if all cells of the MDD are TRUE.

During query processing, Operation Execution is used to compute these operations on tiles.

2.3 Operation Execution

Operation Execution is a central component in the RasDaMan system. RasDaMan has specific requirements on its implementation. Most conventional database queries result in random disk access, because data is accessed in index order over non-clustered indexes. Random access to a disk is several orders of magnitude slower than the execution speed of the CPU. Using sequential access to data on the other hand, a few disks accessed in parallel can easily saturate a processor so that it does nothing else than just copying data in main memory [21]. If a DBMS executes more complex operations than just reading data, system performance is limited by the time the CPU needs to carry out these operations.

In the RasDaMan system this situation is very common: Tiled storage and query optimization are designed to get a high percentage of sequential accesses to data on mass storage. As operations in RasDaMan are executed on large amounts of data, CPU overhead becomes a very important factor for system performance. An efficient implementation of operation execution is therefore crucial for any DBMS working with MDD at a high semantic level. OLAP systems [8] utilizing a high amount of table scans face the same problem.

RasDaMan enables the user to define his own structured base types dynamically at runtime of the database server. Execution of operations therefore has to deal with the additional difficulty of dynamically handling new types not known at compilation time. The selection of applicable operations can only be done at runtime of the system. For efficient operation execution, this overhead has to be minimzed.

3. Formal Model

As all MDD operations are transfered to operations on tiles which themselves can be regarded as MDD, the theoretical foundation for MDD treatment as defined in [2, 3] can also be used as the formal basis for query transformation and tile-based query execution. A summary of the work presented there is given in the following Subsection to define the semantics of operations on MDD. The later Subsections develop a formal model as basis for implementing operation execution on tiles.

3.1 MDD Algebra

The coordinate set of an MDD is called its spatial domain. A spatial domain sd consists of a set of integer vectors $x = (x_1, \dots, x_d) \in Z^d$ bounded by two vectors $l = (l_1, \dots, l_d)$ and $h = (h_1, \dots, h_d)$:

$$sd = \{x | \forall i : l_i \leq x_i \leq h_i\}$$

We use the following notatation for specifying sd through l and h:

$$sd = l_1 : h_1, \dots, l_d : h_d$$

An MDD m is defined as a function $m : sd \rightarrow T$ mapping a vector $x \in sd$ to the corresponding cell value $m(x) \in T$. T is called the base type of the MDD. Two of the basic operation types defined on MDD are spatial operations modifying the spatial domain of an MDD and induced operations working on cell values of MDD.

Trimming is an operation reducing the spatial domain of an MDD in a dimension s by setting new boundaries l' and h' with $l_s \leq l' \leq h' \leq h_s$. The *section* operation reduces the dimensionality of a spatial domain by one by fixing a dimension s to a value t with $l_s \leq t \leq h_s$. The section operation results in a new MDD m' with spatial domain sd' which is defined as follows:

$$sd' = l_1 : h_1, \dots, l_{s-1} : h_{s-1}, l_{s+1} : h_{s+1}, \dots, l_d : h_d$$
$$m'((x_1, \dots, x_{d-1})) = m((x_1, \dots, x_{s-1}, t, x_s, \dots, x_{d-1}))$$

Induced operations are functions applied on the cell values in T. There are three kinds of induced operations: unary operations, binary operations and condense operations. Applying an unary induced operation $f : T \rightarrow T$ on an MDD m results in a new MDD m' with the same spatial domain. It is defined as follows:

$$m'(x) = f(m(x))$$

Binary operations $g : T \times T \rightarrow T$ can be defined similarly on two MDD m_1 and m_2. Condense operations are a higher level construct applying a commutative and associative operation $c : T \times T \rightarrow T$ on all cell values accumulating the result. If c is written in infix notation $t_1 \circ t_2$, the result r of condensing c over MDD m with $sd = \{x_1, \dots, x_n\}$ is defined as follows:

$$r = m(x_1) \circ m(x_2) \circ \dots \circ m(x_n)$$

An example for a condense operation is the operation `all` in Section 2.2. Here the boolean operation AND is applied on all cell values of an MDD with base type BOOL, resulting in TRUE if all cell values of the MDD are TRUE, FALSE otherwise.

3.2 Operations on Tiles

In the previous Subsection a formalism defining the mathematical semantics of operations was presented. The formal model for executing operations on tiles concentrates on specifying a formalism as a basis for implementing an easy-to-use interface used by query processing. For data types like integers a simple formalism is enough: $r = m_1 \text{ OP}_{binary} \, m_2$. But operations on tiles have to take into account two fundamental properties every tile of an MDD has: spatial domain and base type. There is no useful definition for applying an unary operation on an operand tile storing the result in a tile with a different dimensionality. The base types of tiles in an operation are relevant for deciding if this operation is applicable. In the following, a formalism including these two elements will be developed and applicability of operations will be defined.

3.2.1 Spatial Domain

As defined in Section 3.1 the spatial domain of an MDD can be specified as a set of integer pairs specifying lower and higher bound of the indexes in the respective dimension. While in the data type definition of an MDD, one or both of the boundaries can be left open, for every existing MDD both upper and higher bounds are fixed and can only be modified by inserting new tiles into or deleting tiles from the MDD. Every tile has a fixed spatial domain which cannot be modified during the lifetime of the tile. The spatial domain of a tile is unique in an MDD, no other tile of the same MDD can overlap this spatial domain.

The following functions are defined on spatial domains $sd = l_1 : h_1, \dots, l_d : h_d$:

$$\begin{aligned} \text{dim}(sd) &:= d \\ \text{low}(sd, i) &:= l_i \\ \text{high}(sd, i) &:= h_i \\ \text{extent}(sd, i) &:= h_i - l_i + 1 \end{aligned}$$

Using these functions applicability of unary operations on operand t_{op} with spatial domain sd_{op} resulting in t_{res} with spatial domain sd_{res} is given if the following conditions hold:

$$\text{dim}(sd_{res}) = \text{dim}(sd_{op}) \quad (1)$$
$$\forall i : \text{extent}(sd_{res}, i) = \text{extent}(sd_{op}, i) \quad (2)$$

These conditions can be easily extended to binary operations. Note that no conditions have to hold for condense operations as they only return a value and have only one operand. This will change in the future as it is planned to

implement condense operations which do not return values but MDD. An example would be to sum up the columns of a 2-D MDD and return a 1-D MDD containing the sums of its columns.

It is very common, however, that these conditions do not hold. Examples are the addition of a 2-D tile and a section of a 3-D tile or building a large tile as the negation of four small tiles covering the same area. In these cases the tiles have to be adapted in such a way that the conditions are fulfilled. In the following elements for specifying these adaptions are introduced. How they are actually done is decided by higher level modules in the RasDaMan system.

If Condition 2 does not hold, then the spatial domain in the tiles on which to execute the operation has to be specified. To apply operations only on parts of tiles, the element $t[sd_{new}]$ is introduced into the formalism. For this construct to be legal the following conditions have to hold (sd_t being the full spatial domain of the tile):

$$\begin{aligned} \dim(sd_t) &= \dim(sd_{new}) && (3) \\ \forall i : \text{low}(sd_t, i) &\leq \text{low}(sd_{new}, i) && (4) \\ \forall i : \text{high}(sd_t, i) &\geq \text{high}(sd_{new}, i) && (5) \end{aligned}$$

That is, the domain sd_{new} must be contained in sd_t. Now the way of writing down operations is extended to the following:

$$\begin{aligned} t_{res}[sd_{res}] &= \text{OP}_{unary}\ t_{op}[sd_{op}] \\ t_{res}[sd_{res}] &= t_{op1}[sd_{op1}]\ \text{OP}_{binary}\ t_{op2}[sd_{op2}] \\ r &= \text{OP}_{condense}\ t_{op}[sd_{op}] \end{aligned}$$

To fulfill Condition 1 there must be a way to modify the dimensionality of a tile. Note that a spatial domain of [5:5, 1:10] still is 2-D, only the thickness of the first dimension is reduced to 1. To reduce the dimensionality of a tile, a special operation is added which builds a new tile of reduced dimensionality by fixing the index of one dimension p:

$$t_{new} = t_{op}(p)[sd_{op}]$$

For this operation the following conditions have to hold:

$$\begin{aligned} \dim(sd_{op}) &> 1 \\ 1 \leq p &\leq \dim(sd_{op}) \\ \text{low}(sd_{op}, p) &= \text{high}(sd_{op}, p) \\ \dim(sd_{new}) &= \dim(sd_{op}) - 1 \end{aligned}$$

These two extensions to the formalism are the spatial operations on MDD defined in Section 3.1: trimming and section. An example for applying operations on tiles using the formalism is the execution of the following query on two collections *coll1* and *coll2* containing one MDD each:

```
SELECT a[50:150,0:100] + b[0:100,50:150]
FROM coll1 as a, coll2 as b
```

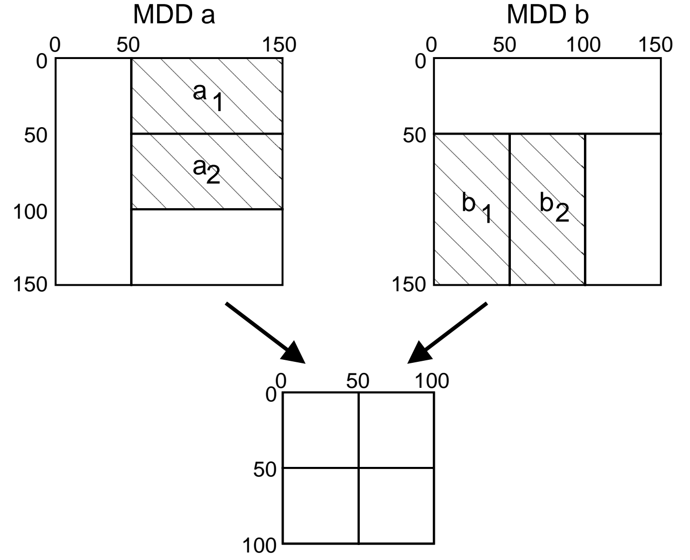

Figure 2. Example for operation on two MDD.

The result shall be one tile t_{res} with a spatial domain of [0:100,0:100] which is then sent as an MDD to the client. The tiling of MDD a and b is shown in Figure 2. The binary operation + on MDD a and b is broken down into 4 operations on tiles a_1, a_2, b_1 and b_2:

$$\begin{aligned} t_{res}[0:50, 0:50] &= a_1[50:100, 0:50] + \\ &\quad b_1[0:50, 50:100] \\ t_{res}[0:50, 50:100] &= a_2[50:100, 50:100] + \\ &\quad b_1[0:50, 100:150] \\ t_{res}[50:100, 0:50] &= a_1[100:150, 0:50] + \\ &\quad b_2[50:100, 50:100] \\ t_{res}[50:100, 50:100] &= a_2[100:150, 50:100] + \\ &\quad b_2[50:100, 100:150] \end{aligned}$$

These operations can be executed in any order giving the optimizer the flexibility to choose the best way to access the tiles on disk. The subdivision of one operation on MDD into four operations on tiles offers possibilities for parallelization of operation execution.

3.2.2 Base Type

In addition to a spatial domain, each tile has a base type. The base type is necessary to decide if operations are appli-

cable on tiles, eg, it is not possible to apply an AND operation on a float. The RasDaMan DBMS supports the following primitive base types as defined in the ODMG standard:

- floating point numbers: DOUBLE, FLOAT
- signed integers: LONG, SHORT, OCTET
- unsigned integers: ULONG, USHORT, CHAR, BOOL

As opposed to C/C++ the bit length of this data types is fixed: LONG and ULONG are 32 bit, SHORT and USHORT are 16 bit and OCTET and CHAR are 8 bit. BOOL is a logical TRUE/FALSE value. In addition to these primitive base types, user defined structured base types are allowed. They can be defined as combinations of primitive base types or other structured base types. In the formalism, numbers are used to access the elements. The type of a structure S_i can be defined as a list of types (P being the set of all primitive types and S the set of all structured types defined by the user):

$$\begin{aligned} S &= \{S_1, \ldots, S_n\} \\ S_i &= [T_1, \ldots, T_n] \\ T_i &\in P \cup \{S_j : j \neq i\} \end{aligned}$$

To make this fully correct, the following rule has to be added: No element of a structure S_i (directly or indirectly) contains S_i. For two structured types $R = [R_1, \ldots, R_m]$ and $S = [S_1, \ldots, S_n]$ to be equivalent ($R \equiv S$) the following conditions have to hold:

$$m = n \tag{6}$$

$$\forall i : R_i \equiv S_i \tag{7}$$

Two primitive types are equivalent if they are the same type: FLOAT $\equiv$ FLOAT, DOUBLE $\equiv$ DOUBLE, but FLOAT $\not\equiv$ DOUBLE. The introduction of structured base types makes it necessary to extend the formalism. It must be possible to execute an operation only on one element of structure, eg, to increment the red channel of an RGB image by 10. For this purpose, the common dot syntax from programming languages is used. The numbers $.e_i$ are used to reference the elements in the formalism. The $.e_i$ part is optional and only applicable to tiles with structured base types. If it is omitted, the operation is executed on the whole structure. The new syntax for operations looks like this:

$$\begin{aligned} t_{res}.e_{res}[sd_{res}] &= \mathrm{OP}_{unary}\ t_{op}.e_{op}[sd_{op}] \\ t_{res}.e_{res}[sd_{res}] &= t_{op1}.e_{op}[sd_{op1}]\ \mathrm{OP}_{binary} \\ &\quad\ t_{op2}.e_{op}[sd_{op2}] \\ r.e_x &= \mathrm{OP}_{condense}\ t_{op}.e_{op}[sd_{op}] \\ t_{new}.e_{new} &= t_{op}.e_{op}(p)[sd_{op}] \end{aligned}$$

To discuss the applicability of operations on type combinations, first, the operations currently implemented in the system are explained. The set of operations supported will be extended in the future. They are classified in the basic groups of operations as defined in 3.1. Two unary operations are supported: NOT does a bitwise or logical negation and IDENTITY copies a value. Binary operations are further subdivided:

- arithmetic operations: MINUS, PLUS, DIV and MULT.
- bit operations: AND, OR, XOR.
- comparison operations: EQUAL, LESS, LESSEQUAL, NOTEQUAL, GREATER, GREATER-EQUAL.

Two condense operations are supported: ALL returns TRUE if all cell values are TRUE (see 3.1) and SOME returns TRUE if at least one cell value is TRUE.

The applicability of these two condense operations is easy to define: The base types of the operand tile and the result value have to be BOOL. The unary operation NOT is applicable to signed and unsigned integers doing bitwise negation. If both operand and result are of base type BOOL, NOT does a logical NOT. IDENTITY is applicable to all operand types. The RasDaMan implementation follows C/C++ in the way it defines legal operations: Even if the result type is of less precision than the operands, the operations are applicable. The result then is not the mathematical result expected, but has to take overflows and signed/unsigned conversions into account. It is not legal to have float operands and integer results. Table 1 shows the applicability of operation IDENTITY.

Result	Operand
unsigned	unsigned
signed	unsigned
unsigned	signed
signed	signed
float	float
float	unsigned
float	signed

Table 1. Applicability of IDENTITY

Applicability of binary operations is different for the three groups of operations. Arithmetic operations follow the same rules as IDENTITY, the result must be float if one of the operands is float. Bit operations are only applicable to booleans, signed or unsigned integers. Comparison operations are applicable to all types, the result always has to be BOOL.

Operations can also be applied to user defined structures, binary operations additionally to one structured and one primitve operand. The operation is then applied to all elements of the structure. For an operation to be applicable to STRUCT types two conditions have to hold:

1. The result type and the type of the structured operands must be equivalent.

2. The operation must be applicable to all elements of the structures.

While the rules for applicability of operations may seem trivial to a person experienced with C/C++ it is nevertheless essential to formally write them down before starting implementation. Applicability of operations depending on spatial domain and type of the operands has to be defined in order to specify the functionality of the Operation Execution module in the RasDaMan system to be used by Query Execution.

When using the Operation Execution module, it is a common case that the base type of the result tile is not given, eg, if a tile is created as a temporary result during query processing. To choose a meaningful base type which avoids loosing precision as much as possible, rules for determining a result type from the operand type have to be set. For unary and condense operations this problem is trivial, the result should have the same base type as the operand.

For binary operations the problem is more difficult. Comparison operations always have result type BOOL. For arithmetic and bit operations, if one of the operands is a structure, the result must be an equivalent structure, so the type of this operand can be used as a result type. If the operands have the same primitive type, it is used as the result type.

In the case of two different primitive types the “stronger” type is used as a result type to minimize the loss of precision in converting the result to this type. The rules for determining the result type are as follows:

1. If one of the operand types is DOUBLE, the result is DOUBLE.

2. If one of the operand types is FLOAT, the result is FLOAT.

3. If one of the operand types is signed, the result is also signed. The bit length of the result is the maximum of the bit length of the operands.

4. If both operands are unsigned, then the result has the unsigned type with the longer bit length.

Rule 3 says that, eg, addition of a USHORT and a LONG should result in a ULONG. Note that overflows are still possible when executing operations.

4. Implementation

In the previous Section, a formal model for applying operations on tiles of an MDD utilizing the base type and the spatial domain has been developed, and preconditions for executing an operation on a set of operands have been defined. The semantics of the operations has not been defined, but in the implementation the operations are mapped to C++ operations thereby defining the semantics. This is possible as the ODMG-92 type system used by RasDaMan is closely related to the C++ type system. The actual execution of operations on binary data is done by the C++ compiler. The mapping of RasDaMan operations to C++ operations is explained in Section 4.2. The problem of applying operations in spatial domains is discussed in 4.3. In the following Subsection, the implementation of operation execution system is discussed. A general overview on the OO design of the whole RasDaMan system was given in [12].

4.1 Object Oriented Design

The RasDaMan DBMS is a complex software system being developed by a team over more than two years. To manage development of this system, state of the art software engineering techniques were applied. Important goals in the design of the system were the following:

- Portability between different OS platforms.
- Maintainability and extensibility of the code, as the system will be refined and improved over time.
- Modularization into units with a concise, easy to use interface.
- Efficiency when dealing with huge amounts of data.

The system was designed in an object oriented way and implemented in C++, making it possible to map the object oriented concepts used in design directly to the implementation. To leverage the use of C++, not only the basic C++ features were used, but also higher level concepts like design patterns [14] or idioms [9] were applied. Data structures from the Standard Template Library were used [20] and the developers were familiar with common C++ programming techniques [18, 19].

For designing the Operation Execution module, the formal model defined in Section 3 was used as a basis. The main data structure in operation execution are tiles, so it was obvious that there should be a class Tile. The spatial domain and the base type are stored within the Tile objects. To follow the formalism closely, Tile should offer a member function for executing operations. This member function is called on the result tile. One alternative would be to provide a member function for every possible operation like

execPLUS or execNOT. This was not done for following reasons:

- The addition of a new operation to the system would involve modification of the Tile class.
- There is a great amount of code reuse between the different exec methods, eg, the multidimensional iteration described in 4.3. This code would have been factored out in a general method anyway.
- The code for implementing operations is closely connected to the data types supported and has to be modified if new data types are added to the system.

After taking this into consideration it was decided to offer a set of generic execution functions getting the operation to execute as a parameter. The minimum set that is needed are functions for unary, binary and condense operations. Other parameters needed are the operand tiles and the spatial domains in which the operation has to be executed. The member function for executing an unary operation is the following:

```
virtual void execUnaryOp(
  Ops::OpType op,
  const r_Minterval& areaRes,
  const Tile* opTile,
  const r_Minterval& areaOp );
```

`r_Minterval` is a class for spatial domains offering the functions defined in the formal model. This signature can be extended for binary and condense operations. Binary operations need a second operand, condense operations need a cell value to accumulate the result. A call to execute an unary operation looks like this:

```
t_res.execUnaryOp( OP_NOT, sd_res,
  t_op, sd_op );
```

This is basically a direct mapping of the formalism defined in the previous Section to C++. These function calls are the main interface for other modules to Operation Execution, all other things explained in the following are internal to this module. Inside the execution functions the operation parameter is mapped to a function object executing the operation on cell values depending on the base types of result tile and operands as explained in Section 4.2. This function object then is applied to all cell values in the spatial domains as explained in Section 4.3.

Object oriented design was also used in modeling the type system in the Catalog Manager, which is used extensively in Operation Execution. The relevant part of the class hierarchy for types is shown in Figure 3. Base types of tiles are subclasses of class BaseType containing information such as the size or the name of the type. There is one class for every primitive type of which one global persistent instance exists. One persistent instance of class StructuredType is created for each user defined type storing information like name and type of the elements in the structure. Tiles use references to these objects for storing information about their base type.

4.2 Function Objects

The member functions of class Tile executing operations have to apply the correct C++ operation depending on the types of result and operands. As the Catalog Manager manages type information, this selection process should be implemented there. For this purpose a function which returns a function is needed, ie a higher level function as supported for example in Lisp [23]. A standard C++ idiom used for this purpose are function objects [9, 16]. Function objects are classes with an overloaded `operator()` so objects of these classes can be used in a function call syntax.

In C higher level functions could be implemented using function pointers. One advantage of function objects is that they keep a state as they have member variables. In RasDaMan the member variables of a function object contain pointers to the base types of result and operands. Function objects are similar to closures [16] from functional programming languages in this respect.

The application of function objects is syntactically very simple. In the execution functions of class Tile the selection process for the operation and its application is very easy to read:

```
BinaryOp* myOp;
myOp = Ops::getBinaryOp( op, resType,
         op1Tile->getType(),
         op2Tile->getType() );
...
(*myOp)(cellRes, cellOp1, cellOp2);
```

The application of the function object `myOp` is done inside a multidimensional iteration. Using higher level functions, the selection process for the appropriate function to be applied has to be done once for each operation on tiles only, not once for each cell.

The type of the function object (BinaryOp, UnaryOp or CondenseOp) is the only knowledge the class Tile has to have about the operations it executes. All the other information needed is in the Catalog Manager stored together with the closely related information on types in the system. This design minimizes code changes to modules directly affected by, eg, adding a new primitive type to the RasDaMan system.

Using the function object idiom, execution of operations on structures needs minimal additional code. The function objects for operations on structures simply apply function

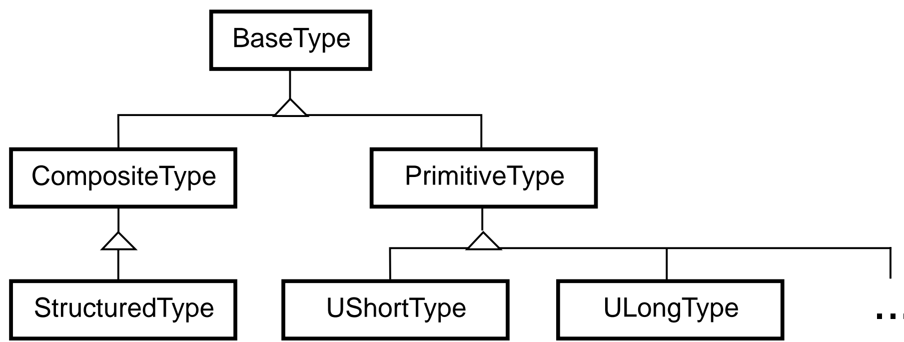

Figure 3. Class model for types.

objects on primitive types to all elements of the structure. It is not necessary to implement function objects specific for each structure.

Inside the function objects, operations are executed in C++ statements on the appropriate C++ types. Considering the primitive types supported in the RasDaMan system, there are a lot of possible type combinations on which to execute operations. There are nine primitive types, every binary operation takes 3 types, every unary or condense operation 2 types. So there are 9^3 possible type combinations each for 13 binary operations and 9^2 combinations for 4 unary and condense operations making a total of 9801 combinations. It is not practical to implement that many functions in a program. Even if the source code would be generated, the object code would get an unhandy size and compilation and linking would take extremely long.

To reduce the number of possible combinations operations are executed on three C++ types only: double, long and unsigned long. These are the highest precision types in the three groups of primitive types: floating point number, signed and unsigned integers. So no precision is lost when using these types for executing operations. Efficiency does not degenerate, because current CPUs are optimized for the execution of 32bit operations. Every primitive type knows how to convert a cell of its type to a C++ type according to Figure 4.

Using this technique every operation needs to have three implementations for the different C++ base types, eg, OpPLUSCDouble, OPPlusCLong, OPPlusCULong. Implementing the 17 RasDaMan operations then needs a maximum of 51 C++ functions carrying out the work. In each of these functions, member functions of the types in the type hierarchy (see Figure 3) are executed to convert a cell value to the appropriate one of those three types.

4.3 Multidimensional Iteration

To execute the function objects on all relevant cells the execution functions have to iterate through a spatial domain. As RasDaMan supports arbitrary dimensionalities of tiles, the dimensionality has to be a loop variable also. The basic algorithm for multidimensional iterations is very simple:

1. Increment the lowest dimension.
2. If the highest index in the current dimensions is not exceeded, we are done.
3. Otherwise increment the next higher dimension and set the previous dimension to the lowest index. Check 2 again.
4. If the dimension we want to increment is outside the dimensionality, we have iterated through everything.

For every iteration a 1-D offset into the tile has to be calculated from the multidimensional coordinate. This process involves iterating through the dimensions again and calculating differences to the origin of the tile and multiplying them. All these operations are done using a the class `r_Minterval`.

The first versions of the RasDaMan system used a straight forward implementation of these algorithms as described above. Performance analysis done on operation execution showed that most of the time used was spent in the multidimensional iteration. Setting the optimization of the compiler to a higher level gave better results, but still not satisfying. Further optimization was done by making critical functions inline to save the function call overhead. Another optimization is to not recalculate the 1-D offset if only the lowest coordinate has been incremented, but to just increment the offset. Only after these steps an acceptable performance level was reached.

Compare this to a standard C solution iterating through a 2-D array: The code usually fits in the processor cache, there are no function calls or loops to do the offset calculation. Current compilers optimize this code very well, the programmer does not have to tune its code carefully for good performance. This is different if more complex problems are tackled with the more expressive C++ language.

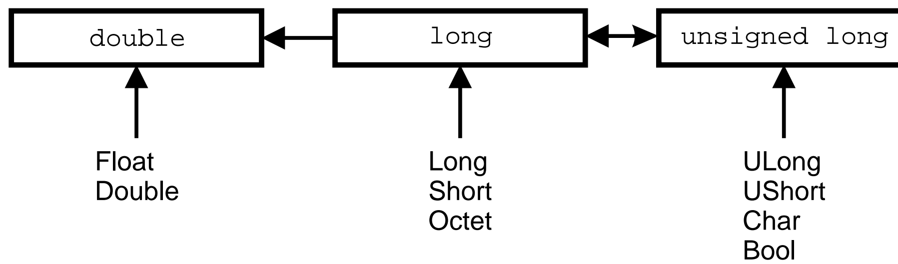

Figure 4. Converting RasDaMan types to C types.

In this case it is the responsibility of the developer to use the features of the C++ language in a way that does not drastically worsen the performance of the generated code. The higher expressiveness of C++, which is of great advantage when dealing with complex problems, comes at a cost of giving more responsibility regarding performance to the programmer.

5. Related Work

Executing operations on data is almost a definition of computer science. Considered relevant for the work described in this paper is the work done in programming languages and database systems. Compiled and interpreted programming languages are examined, database systems are subdivided into relational systems (RDBMS), object database management systems (ODBMS) and object-relational systems (ORDBMS).

Programming languages which can be compiled into efficient code are usually statically typed, ie the type of every variable is known at compile time. The correct code for executing operations then can be generated at compile time, the efficiency of this step is not relevant to the runtime performance of the program. In C++ [11, 25, 26] runtime polymorphism is supported, but still all types are known at compile time and polymorphic operations can be resolved by just adding a level of indirection (the virtual function table). Interpreted languages usually are dynamically typed like Smalltalk [15] or Lisp [23]. The selection of code to execute an operation is done at runtime, as the types of variables are not known before. This process must be done every time an operation is executed, only recently systems employ just-in-time compilation to reduce this overhead. In general, interpreted systems are not adequate for executing operations on large amounts of data.

RDBMSs [10] support a limited set of data types defined in the SQL-92 standard [6], support for these data types is compiled into the kernel. The system does not have to deal with user defined types and can be tuned very well for efficiency. By far, most commercial ODBMSs [17] are persistent C++ systems. The ODBMS server just stores the attributes of objects, he cannot execute user defined operations on them. Execution of operations is completely done at the client, which means that all data to be processed has to be sent to the client.

ORDBMSs [24] extend the relational model with user defined data types. The interface between the data types and the declarative query language is small, only functions and access to elements of the new data types are possible. Extending the syntax of the query language like done in RasQL is not possible. User defined functions are usually interfaced with an ORDBMS by dynamically linking them to the DBMS server. This has the severe disadvantage that user code can corrupt the server code as it is running in the same adress space.

The aim in the RasDaMan project was to build a DBMS for managing MDD offering an extensible type system. The implementation effort necessary to offer compilation of operations in queries to efficiently execute them is much too high in this context. Execution of operations at the client should be avoided, as huge amounts of data are examined in typical applications working on MDD. The work done in ORDBMS is relevant for RasDaMan, but the limitations of the object relational concept are too restrictive for a DBMS supporting general MDD indepedent of a specific application area. Therefore RasDaMan chose to use object-oriented design and C++ constructs to efficiently implement operation execution with dynamic user defined types.

6. Conclusion

In this paper operation execution on tiles of multidimensional arrays in the RasDaMan system was discussed. This is a necessary precondition for the implementation of scientific database systems handling multidimensional arrays at a high semantic level. The RasDaMan system differs from work done on operation execution in two important points: First it supports user defined structures which can be added at runtime to a database system and second it supports a declarative query language embedding multidimensional arrays into SQL-92. A theoretical model has been developed for the execution of operations on tiles. This model

used the work already done on operations on MDD for RasDaMan and was designed with object-oriented implementation in mind.

Efficiency was a key design target for the implementation, as the operations have to be applied to large amounts of data. Another design goal was to offer an easy to use and small interface to other modules in the RasDaMan system and at the same time keep the system extensible towards new operations and primitive types. All of these targets could be achieved to a high degree by using the C++ programming language. The object oriented features allowed to translate the theoretical model almost directly to the implementation. The interface is small, but flexible enough to support an arbitrary number of operations and data types. Extension of the system is possible, code changes are localized to a few classes. The function object idiom could be used to minimize the runtime overhead of executing operations in a dynamically typed system. The compiler optimization and inline functions helped to make critical functions more efficient.

Using object oriented implementation techniques, it was possible to evaluate the efficiency of the implementation very early in the development. First, performance evaluations have been done using tiles in main memory and the interface for executing operations stand alone. During these tests, some performance bottlenecks have been discovered. But the RasDaMan system as a running system offers the possibility to evaluate the implementation in a complete system using MDD stored in the database and evaluating RasQL queries as opposed to executing single operations only. Performance measurements done on the whole system will result in possibilities for further optimizing operation execution and also the interaction between query optimization, operation execution and access to the base DBMS.

Acknowledgements

The authors would like to thank Andreas Dehmel, Paula Furtado and Roland Ritsch, all working in the RasDaMan team at FORWISS.

References

[1] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System*. Morgan Kaufmann Publishers, 1992.

[2] P. Baumann. On the management of multidimensional discrete data. *VLDB Journal*, 4(3):401–444, 1994.

[3] P. Baumann. An algebra for domain-independent array management in databases. submitted for publication, 1998.

[4] P. Baumann, P. Furtado, R. Ritsch, and N. Widmann. Geo/environmental and medical data management in the rasdaman system. In *Proceedings of the VLDB'97 Conference, Athens, Greece*, 1997.

[5] P. Baumann, P. Furtado, R. Ritsch, and N. Widmann. The rasdaman approach to multidimensional database management. In *Proceedings of the 1997 ACM Symposium on Applied Computing, San Jose, California*, pages 166–173, February 1997.

[6] S. Cannan and G. Otten. *SQL - The Standard Handbook*. McGraw-Hill, 1993.

[7] R. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, 1997.

[8] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1):65–74, 1997.

[9] J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[10] C. Date. *An Introduction to Database Systems, Sixth Edition*. Addison-Wesley, 1995.

[11] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[12] P. Furtado, R. Ritsch, N. Widmann, P. Zoller, and P. Baumann. Object-oriented design of a database engine for multidimensional discrete data. In *Proceedings of the OOIS '97 Conference, Brisbane, Australia*, November 1997.

[13] P. Furtado and J. Teixeira. Storage support for multidimensional discrete data in databases. *Computer Graphics Forum - Special Issue on Eurographics'93 Conference*, 12(2):89–100, September 1993.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[15] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[16] T. Kühne. The function object pattern. *C++ Report*, 9(9), 1997.

[17] M. E. S. Loomis. *Object Databases: The Essentials*. Addison-Wesley, 1995.

[18] S. Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 1992.

[19] S. Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley, 1996.

[20] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.

[21] E. Riedel, C. vam Ingen, and J. Gray. Sequential i/o on windows nt 4.0 - achieving top performance. 1997.

[22] R. Ritsch and P. Baumann. Optimization and evaluation of array queries. submitted for publication, 1998.

[23] G. L. Steele. *Common Lisp: The Language, 2nd Edition*. Digital Press, 1990.

[24] M. Stonebreaker and D. Moore. *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers, 1995.

[25] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[26] B. Stroustrup. *The C++ Programming Language Third Edition*. Addison-Wesley, 1997.

[27] N. Widmann and P. Baumann. Towards comprehensive database support for geoscientific raster data. In *Proceedings of the ACM-GIS 97, Las Vegas, Nevada, USA*, November 1997.