

Object-Oriented Design of a Database Engine for Multidimensional Discrete Data

P. Furtado¹, R. Ritsch, N. Widmann, P. Zoller, P. Baumann
FORWISS (Bavarian Research Center for Knowledge-Based Systems)
Munich, Germany

Abstract

Multidimensional discrete data (MDD), i.e. arrays of arbitrary size, dimension and base type, occur in a variety of application fields. The object-oriented DBMS RasDaMan² provides domain-independent management of MDD. In the design and development of the RasDaMan system, an informed assessment of current solutions to the management of persistent MDD led to the identification of major limitations remaining in object and object relational DBMSs and to the proposal of alternative approaches. In this paper, we present a discussion of those issues and report on the main design decisions taken for the RasDaMan system.

1 Introduction

Raster data has become one of the most often occurring types of data in computer systems. Examples of raster data objects range from common 1-D sound sequences and 2-D images, to domain-specific objects originated from sampling natural phenomena, like a 4-D climate simulation, or from artificial sources such as simulators and business data analyzers, e.g. an 8-D OLAP datacube. Even though these objects differ greatly, they share the same basic properties and requirements. Each raster data object is a multidimensional array of cells of some base type, hence the term Multidimensional Discrete Data or MDD. Although MDD forms a very well-defined category of data structures, for a long time it has received surprisingly little attention among research communities on object-orientation and database issues.

Investigation in MDD application areas such as medical imaging/PACS, geographic information systems, and OLAP/data mining have shown that there is a common need for MDD services, provided this functionality can be decoupled from the legion of application specific data formats in use. Functions such as subcube extraction or projection, and aggregation along specified dimensions play an important role in all these application fields [2]. Even content-based retrieval methods require base MDD operations that only consider pixel level information, with no interpretation of contents.

¹PhD work sponsored by a PRAXIS XXI scholarship.

²sponsored by the European Commission in the ESPRIT Domain 4: Long-Term Research under grant no. 20073.

In the RasDaMan system, an object-oriented approach to the modeling, storage, manipulation, and retrieval of MDD in databases is followed which is domain-independent. As described in [2], this approach responds to the particular requirements identified for MDD management. Classical DBMS features available on alphanumeric data such as declarativeness, orthogonality, optimizability, and data independence will be made available on MDD, too. An MDD object is an array, a generic collection category, parametrized with its base type, a C++ type, and spatial domain consisting of the lower and upper bounds for each dimension. Every boundary can be left variable, allowing the MDD to grow and shrink during instance lifetime. A C++ application programming interface, RasLib, is offered which extends the ODMG-93 standard [5] binding with MDD functionality.

The RasDaMan query language, RasQL, is used by applications to query collections of MDD objects stored in the database. It provides a set of high-level primitives for executing advanced operations on MDD. RasQL extends standard SQL with MDD operators which can be roughly categorized as follows. Among operations changing the spatial domain of an MDD there are rectangular cutouts (trimming) and extraction of lower-dimensional subarrays (projection) of MDD. Other operations simultaneously change cells values, leaving MDD geometry unaffected. With “local” operations, for each cell value v a new value $f(v)$ is derived; with “global” operations, additionally some neighborhood of a cell is used for computation of the derived cell value (e.g., filtering and warping). Condensers, a general (second-order) concept generalizing the relational aggregation operators, allow to selectively derive summary information. Array traversal sequence is left undefined, thereby opening up space for internal query optimization. MDD expressions can be used in the `select` part of a query and, if the outermost expression result type is scalar, in the `where` part.

The remainder is organized as follows. In Section 2, we discuss the state of the art in MDD database technology. The following sections are dedicated to the main components of the system. Section 5 presents conclusions.

2 Related Work

Current DBMSs do not support MDD directly. In order to store such data, the classical approach is to revert to unstructured BLOBs, Binary Large Objects. In this approach, the DBMS does not know anything about the application semantics, but treats the multidimensional data item as a one-dimensional, encoded byte string. This solution is inadequate both from the point of view of performance as well as from that of the data model and operations supported. In the meanwhile, many relational and object-relational DBMSs enable the implementation of new abstract data types (ADTs). In object DBMSs (ODBMSs) [11] the data model allows implementation of any user defined classes which is done in client code. In object-relational and relational DBMSs [7], definition of new types is done by adding extensions to the core engine.

In ODBMSs, seamless integration of persistent data in the programming language is a major goal [13]. For this reason, the properties and operations of the object models they support are represented in object programming languages such as C++ or Smalltalk. The shortcomings of the programming models are then directly reflected in the object model supported by the DBMS. Regarding MDD, very basic modeling is provided by object-oriented programming languages. An array is a 1-D entity by default. Higher dimensional arrays are modeled as arrays of arrays while their internal representation remains linear. No high-level array operations or specialized storage structures are supported in object-oriented programming languages of today, like Java [8], C++ [19], and Smalltalk [9]. Basically, only primitive operations like access to a cell or to the whole array are possible.

ODBMSs rely on the extensibility of the underlying programming language data model to support more complex data types. Even though, in applications using only transient MDD objects, higher level support for MDD functionality may be implemented with reasonable effort as a class, such a solution is not feasible if persistent MDD objects are to be managed. If MDD functionality is required in an application, it has to be fully implemented by the programmer. Because the application is executed at the client side, it is not possible to optimize operations on persistent MDD or adopt specialized storage structures, which is important to enhance performance when dealing with typically large MDD objects. Without those features, much time is spent on transmission of unnecessary data between client and server.

Recently, several DBMSs have started supporting extensible engines. This approach allows more flexibility in the definition of complex types and execution of operations on those types through extensions to the DBMS engine. Nevertheless, extensible types have to be implemented by very specialized DBMS developers and the amount of implementation effort is enormous, particularly if advanced storage structures are to be used. In addition, in those systems it is not possible to extend the query language with new syntax, since only functions can be called on newly defined data types. One of the most prominent systems in this area is Illustra [18] which supports extensions through so-called DataBlades. There are already individual DataBlade extensions for time-series and temporal data, images, as well as 2-D and 3-D spatial and location data, but up to now no DataBlade has been defined for MDD of arbitrary dimensions. This is a major limitation since operations on multidimensional data often involve operands or results of different dimensionalities (for example, a 3-D object may have to be created from a set of 2-D ones, or an n -D MDD object may result from the projection of an $n+1$ -D object).

Specialized support for OLAP applications is provided by some relational DBMSs [14]. Relations are used to store multidimensional OLAP data consisting of sparsely (e.g., 5%) populated arrays in multidimensional domains. As a consequence, performance and storage utilization in those systems is not acceptable for generic MDD application areas dealing with dense multidimensional arrays. Other application specific DBMSs are dedicated to high-level operations on data for particular application areas. Paradise [15] is an example

of a DBMS designed for handling 2-D MDD in GIS applications. 2-D arrays are modeled as ADTs in the object-relational model of SHORE [4]. Efficient storage of the raster ADTs is provided in Paradise by tiling the data into a set of SHORE objects. Paradise does not support MDD of more than two dimensions, nor does it provide a general MDD query language.

3 Storage Management

Storage management in RasDaMan aims at providing efficient access to MDD objects or parts of them and transparent support for various storage devices. Due to the typically large sizes of MDD objects and the type of operations most often performed on them, specialized storage management is required to manage those objects efficiently.

3.1 Storage Structure

An MDD object is stored as a set of multidimensional rectangular subarrays, *tiles*, each one stored in a different object of the base DBMS. The cells of one tile are stored as a linear array of bytes, i.e. a BLOB. The system supports arbitrary tiling in that the tiles belonging to an object can have different sizes and can be unaligned, as illustrated on Figure 1 for a 2-D object. This allows for more flexible choice of the adequate tiling for each object. A more in-depth discussion of tiling of MDD objects in RasDaMan can be found in [3].

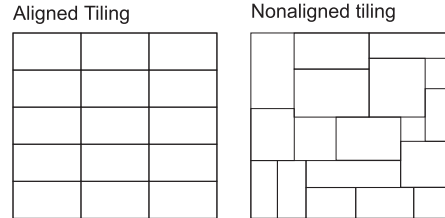


Figure 1: Different MDD Tiling Schemes.

For each object, a spatial index is created which maintains information about the tiles of the object and corresponding spatial information consisting of the coordinates for each tile and the current spatial domain for the object (which may change during the object's lifetime). Different index structures can be created for different objects, for example, a directory structure for a small object with aligned tiles, or an R+-tree [16] for another object with a more complex tiling scheme.

Further metadata, such as a reference to the cell type and the definition spatial domain for the object, also has to be stored. Each persistent MDD object is represented in the base DBMS by an instance of the `DBMDDObject` class which gathers all the information about the object. This instance contains

the metadata and a reference to the object's index which, in turn, references the object's tiles.

3.2 Base DBMS Interface

In the design of RasDaMan, an early decision was made for adopting an ODBMS as the storage system. This decision was driven by four main factors:

- an ODBMS allows fast navigational access, which is of prime importance for the implementation of tree-based indexes;
- the DBMS functionality already provided by the ODBMS - for instance, transaction processing and recovery - is already supported and can be used, whereas it would have to be fully implemented if another simple storage system were used;
- the object-oriented MDD functionality can be smoothly integrated in the ODBMS interface;
- using an ODBMS, the RasDaMan system, including the lower level storage management modules, can be designed and implemented in an object-oriented approach, leading to clear design and high quality extensible code.

In order to choose the base ODBMS for RasDaMan, an assessment of several systems, including a small benchmark, was undertaken. The most important choice criteria was the performance of the system when dealing with large sets of tiles which have to be stored as BLOBs. In addition, we considered that it would be positive to have an ODMG conformant system, so that RasLib could be plugged with the ODMG interface of the base system. The commercial ODBMS O₂ [1] was chosen since it proved to be more efficient than the other candidates for the typical MDD data sets tested in the benchmark and it provides an ODMG binding.

Even though the current version was implemented using the O₂ system, care was taken to design the RasDaMan Storage Manager in a way to allow easy portability to another base DBMS or storage system. This is achieved through the Base DBMS Interface layer. The classes in this layer are responsible for all accesses to persistent data. Their interfaces to upper level classes in the system are kept as small and simple as possible. Porting of RasDaMan to another base storage system only requires porting this layer to the new system. The Base DBMS Interface layer provides storage of index structures, tiles, catalog data, and MDD objects and collections of MDD objects, as well as general database functionality.

Implementation of the storage management using the O₂ ODMG C++ binding brought the well-known advantages of seamless integration provided by ODBMSs. The binding allowed ease of implementation of persistent classes according to an adequate object-oriented design. Persistent classes are defined

for MDD objects, collections of MDD objects, BLOB tiles, indexes and index nodes. Instances of those classes are directly used in the C++ code with no need for translation between the database and the programming language data models, as it would be needed had another base storage system been adopted. It also facilitated the use of late binding in the support for the same functionality for persistent and transient data. Due to this, many steps in query evaluation are implemented independently of whether the operands are persistent or transient. At the same time, data may be accessed efficiently through direct navigation in the data structure. This is particularly important in the access to tiles through the index structure.

Other aspects in the adoption of the used version of the O₂ binding were not so positive. One major difficulty regarded error handling. The binding provides no means to detect and deal with error conditions. For instance, if an attempt is made to open a non-existent root object, an error is generated and the application is simply aborted. Even though exception handling is assumed in the ODMG standard, it is not implemented in the O₂ system. The solution to this problem was the usage of lower level routines from the O₂ Engine to test for error conditions.

More flexibility or transparency in some features of the O₂ ODMG binding would be useful in the development of the storage manager, for example, locking, memory management and object identifiers. Explicit locks are not supported by O₂, but are convenient for more advanced applications and, in particular, would be useful in the development of RasDaMan.

The second issue concerns automatic memory management. A counter of references to persistent objects in main memory kept by O₂ allows automatic deallocation when an object is no longer referenced. It is difficult for the application programmer to have information about whether memory is still allocated for an object or not. In an application requiring much memory like ours, this makes implementation more difficult.

Finally, object identifiers (OIDs) are hidden from the user in the current version of O₂. A new implementation of OIDs was therefore required in order to uniquely identify MDD objects at the client side. It is our opinion that the support for visible logical OIDs is important to allow some advanced functionality, e.g., for interdatabase references. In fact, such functionality is announced for the new version of O₂, even though not at the ODMG interface level.

3.3 Index Manager

The Index Manager is responsible for providing all information on access to tiles of persistent MDD objects. The functionality supported by the Index Manager includes identification of tiles affected by a spatial access to an MDD object, calculation of tiles access costs, and determining the most efficient access sequence to a set of tiles. Whenever classes responsible for query evaluation require access to part of an object - typically a multidimensional subinterval of the object's spatial domain - they send a message to the `MDDObject` to request the set of tiles intersecting the area of interest. The `MDDObject` then calls the

appropriate method from the index member object to identify the tiles affected. Search is performed using navigation in the persistent index structure and the tiles of interest are returned to the caller in a collection.

In order to allow different objects to have different index structures, polymorphism is used. A super class defines the common interface for MDD objects indexes which can then have several implementations as subclasses. The existence of different indexes allows the most appropriate type of index to be adopted for each object when it is first created. This is of interest due to the arbitrary tiling: a complex spatial index may be required for an object stored using nonaligned tiling, whereas the same index structure may lead to bad storage utilization in another object having a simple aligned tiling. Having a common indexing interface allows execution of the operations on MDD objects without having to care about the type of index for each object. In addition, this implementation allows us to easily add new index techniques to the Index Manager.

4 Query Processing

Processing of queries begins with the translation of the textual query statement into an internal tree-based representation. On this data structure, semantic analysis and optimization take place before an evaluation plan is generated, which finally is executed at the tile level. Figure 2 illustrates the different data structures occurring in this process in rounded boxes and operations on them in angular ones. The following subsections account for the object-oriented design chosen for the different structures.

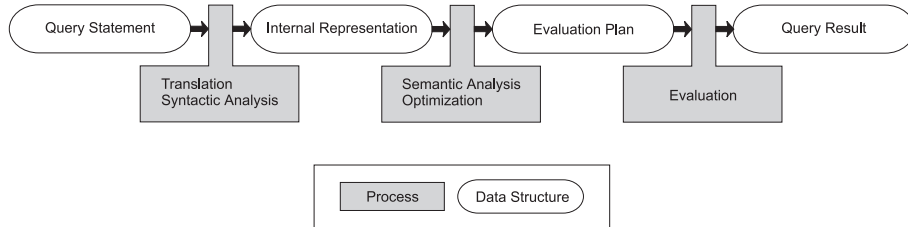


Figure 2: Query Processing Steps.

4.1 Construction and Optimization of the Query Tree

The internal representation is an operator-based query tree which is a procedural description of the query based on operator and dataflow statements. Inner nodes of the tree stand for operators and leaf nodes for data sources which, in our case, are collections of MDD objects. From a data modeling point of view, for each operation, be it a relational one, e.g. selection, or an MDD operation, e.g. projection, a subclass of the node base class exists which inherits functionality like management of descendants or knowing about its parent. Therefore,

they all support the same interface and a query tree representing any query statement can be set up using just the methods of the node base class.

Traversal of the tree is described with the type checking process. A pure virtual declaration of the type checking method is declared in the common superclass of all operator classes. This method then is implemented for each specific operator; it takes care that type checking first is propagated to its subtree and then the types of its own operands are verified. At the end, a result is passed to its parent. Usage of a common interface (pure virtual declaration in a type generalization) and encapsulation of operator specific type checks within the operators allow for type checking without knowing about the structure of the operator tree, making extensions of new operators simple and less error-prone.

Query rewrite is a process in which general-purpose heuristics drive semantics preserving transformations. These transformations need to rearrange or exchange operator nodes or even whole subtrees. According to the fact that all operators share the node interface definition, modifications of the tree structure are mostly carried out with the methods of the node base class. Operator specific information is just needed to ensure equivalence and whether the type of a node is to be changed.

The evaluation model abstracts from special operators so that each operation is seen as a processing unit which consumes data from one or more input streams, carries out a specific operation, and produces data sent into exactly one output stream. By combining these processing units in such a way that an output stream from one unit serves as one input stream to the following unit, an evaluation tree is built. The structure is similar to the query tree, so the operator nodes used for internal query representation are also used as elements in the evaluation tree.

The evaluation strategy of the tree is demand driven and based on the ONC protocol [10], according to which each processing unit supports the methods `open()`, `next()` and `close()`. A processing unit (data producer) is first initialized by sending an `open()` message to the object, then data items are received by invoking `next()` as long as no out of data signal is received, and finally the producer is shut by calling `close()`. To evaluate a query, the output stream of the root object is read using the ONC protocol. Invocations of `open()` and `close()` are propagated through the whole tree. If `next()` of a processing unit is called, then it is only passed to those input streams which actually provide data necessary for computation of the next data item.

The protocol is implemented through inheritance of the stream functionality and ad hoc polymorphism. The protocol methods are declared virtual in a common superclass. The methods can provide default implementations which propagate `open()` and `close()` to the descendants and just pass items from the input stream to the output stream within the method `next()`. Each processing unit type can then redefine the implementations according to its specific functionality and late binding takes care of choosing the right implementation.

4.2 Execution Engine

The Execution Engine is responsible for executing basic operations used in the operator nodes of the query plan. The functionality of the Execution Engine is part of the Storage Manager modules where the classes for data sources are defined. The basic data source is a collection of MDD, on which selection operators can be applied to retrieve MDD objects. Finally the basic operations are carried out on tiles, which are retrieved from the MDD objects. In the following, the implementation of the usual arithmetic operations on numeric base types or access to elements of a structured base type will be discussed.

While persistent C++ as supported by the base DBMS O_2 offers the full power of the object-oriented programming paradigm for specifying operations on types, we did not use the C++ type system. One reason for this is that we use the base DBMS only as a storage manager (see Section 3.2); tiles are stored as arrays of C++ `chars`. Another reason is that it should be possible to define new structured base types for cells at runtime without shutting down or even recompiling the RasDaMan server. With a strongly typed and compiled language like C++ new types can be used only after recompilation.

To support base types other than `char`, the RasDaMan DBMS has to provide its own internal type system. We implemented our type system in an object-oriented way, supporting the ODMG-93 primitives like `ULong` or `Bool` and structures. Our type system implementation uses metaclasses as in dynamic object oriented languages like Smalltalk [9] or CLOS [12]. For each base type, a C++ object is created which is an instance of a metaclass. For simple base types, just one instance of the corresponding metaclass exists. For structured base types, one instance of the corresponding metaclass is created for each user defined structured base type, which stores all information needed for the structured type (e.g., name and type of its elements) in member variables.

Each metaclass knows how to carry out operations on cells of its type using a group of `chars` as operands. These operations are returned as function objects if needed. Function objects or functors [6] overload the function call operator `operator()` and can be used to implement higher order functions, i.e. functions returning functions as a result, like provided in LISP [17]. Whenever the executor carries out an operation on a tile, it requests a function object for the operation from the metaclass. This has to be done only once for each tile, the function object is then applied to each cell of the tile.

Given these function objects, the executor calls a method of the tile to carry out an operation on all of its cells or a part specified as a multidimensional interval. It is the responsibility of the executor to break down MDD operations on tiles, specifying, for each tile, the areas which are affected when carrying out an operation involving two MDDs. The schema information, i.e. all instances of metaclasses representing the simple base types supported by the system and the user defined structured base types currently in use, is stored in the RasDaMan DBMS. C++ classes are used to do this which are made persistent with O_2 .

5 Conclusions

Compared to existing DBMSs, RasDaMan offers several innovative features regarding MDD management. An MDD object is modeled as an array with specified base type and multidimensional spatial domain. The conceptual MDD model and its ODMG conformant binding provide seamless integration of persistent MDD objects in C++ applications. A query language allows execution of complex, computation intensive operations on the RasDaMan server. A specialized storage structure for large MDD objects is adopted which is composed of multidimensional tiles and spatial indexes on the tiles. Query evaluation is performed at tile-level. Inheritance and ad hoc polymorphism are used to implement the query tree evaluation protocol. Operations on tiles are executed using function objects, which allow support for the RasDaMan provided MDD operations also on MDD objects with user defined cell types.

Although the first version of the system does not fully exploit the optimization potential, initial performance tests showed that both client-server data transmission and storage structure have high impact on the execution performance of MDD operations. A first test case assessed the influence of data transmission on the overall performance. The same MDD operation was performed both on the client and on the server running in two hosts located in a local network, executing a simple trimming operation on the images of a collection. The collection had four 2.5 megabytes (MB) satellite images corresponding to the same geographical area and the trimmed subimages had 750 kilobytes (KB) each. A performance gain higher than two was obtained for this case. A second test case showed the impact of the tiling scheme on the speed of execution of operations on MDD. In this case, a 40 MB 3-D object, consisting of tomogram slices, was stored using two different tiling schemes: cubes (100 KB each) and slices of thickness one along the x-direction. The time required for executing a z-projection on the sliced data was five times that required for the cubed object. Such performance gains can only be achieved if internal support for MDD is provided by the DBMS at the server side.

The implementation of basic storage management using the C++ ODMG binding of O₂ allowed a design of the database engine where the main entities of the system like MDD objects, collections of MDD objects, and tiles, are modeled directly as classes starting from the storage level up to the communication module. The architecture of the system was designed to keep the interface between modules small. This is particularly important in the Base DBMS Interface layer, in order to allow easy porting of the RasDaMan system to another base storage system, and in the communication module. In this later case, the efficacy of the design was already proven when a necessary re-implementation of the communication layer using a different RPC system was done with little effort. In the future, we intend to explore more advanced optimization techniques and support new data types. Implementation of further operations such as affine transformations (e.g., scaling and rotation) is also planned.

References

- [1] F. Bancilhon, C. Delobel, P. Kanellakis: **Building an Object-Oriented Database System**. Morgan Kaufmann Publishers, San Mateo-California, 1992.
- [2] P. Baumann, P. Furtado, R. Ritsch, N. Widmann: Geo/Environmental and Medical Data Management in the RasDaMan System. **Proceedings of the 23rd VLDB Conference**, Athens-Greece, 1997.
- [3] P. Baumann, P. Furtado, R. Ritsch, N. Widmann: The RasDaMan Approach to Multidimensional Database Management. **Proceedings of the 1997 ACM Symposium on Applied Computing**, San Jose-California, February 1997, pp. 166-173.
- [4] M. Carey et al.: Shoring up Persistent Objects. **Proceedings of the 1994 ACM-SIGMOD International Conference on Management of Data**, Minneapolis-Minnesota, 1994, pp. 383-394.
- [5] R. Cattell: **The Object Database Standard: ODMG-93**. Morgan Kaufmann Publishers, 1996.
- [6] J. Coplien: **Advanced C++ Programming Styles and Idioms**. Addison Wesley, 1992.
- [7] J. Davis: **INFORMIX-Universal Server: Extending The Relational DBMS To Manage Complex Data**. DataBase Associates International, Informix, 1996.
- [8] D. Flanagan: **Java in a Nutshell**. O'Reilly & Associates, Inc., 1996.
- [9] A. Goldberg, D. Robson: **Smalltalk-80: the Language**. Addison Wesley, 1989.
- [10] G. Graefe: Query Evaluation Techniques for Large Databases. **ACM Computing Surveys**, 1993, vol. 25, no. 2, pp. 73-170.
- [11] W. Kim, Ed.: **Modern Database Systems: the Object Model, Interoperability, and Beyond**. ACM Press, New York, 1995.
- [12] G. Kiczales, J. des Rivieres, D. Bobrow: **The Art of the Metaobject Protocol**. The MIT Press, 1991.
- [13] M. Loomis: **Object Databases: The Essentials**. Addison-Wesley, 1995.
- [14] Oracle: **Oracle7 Spatial Data Option Overview**. Oracle Corporation, 1996.
- [15] J. Patel et al.: Building a Scalable GeoSpatial Database System: Technology, Implementation, and Evaluation. **Proceedings of the 1997 ACM-SIGMOD International Conference on Management of Data**, Tucson-Arizona, 1997.
- [16] T. Sellis, N. Roussopoulos, C. Faloutsos: The R+-tree: A Dynamic Index for Multidimensional Objects. **Proceedings of the 13th VLDB Conference**, Brighton-England, 1987, pp. 507-518.
- [17] G. Steele Jr.: **CommonLISP: The Language**. Digital Press, 1984.
- [18] M. Stonebreaker, D. Moore: **Object-Relational DBMSs: The Next Great Wave**. Morgan Kaufmann Publishers, 1996.
- [19] B. Stroustrup: **The C++ Programming Language**. Addison-Wesley, 1991.