On the Design and Implementation of the Multidimensional CUBEStore Storage Manager

Wolfgang Lehner, Wolfgang Sporer
Department of Database Systems
Friedrich-Alexander University of Erlangen-Nuremberg
Martensstr. 3
D-91058 Erlangen, Germany
{lehner, wgsporer}@immd6.informatik.uni-erlangen.de
Tel: +49 9131 85-7800
Fax: +49 9131 85-8854

Abstract: CUBEStore is a storage manager designed within the CUBESTAR project to fit the special needs of multidimensional applications as found in the area of Statistical and Scientific DataBases (SSDB). Some of the goals being achieved within this project were built-in support for multiple dimensions and good performance for range queries regardless of the dimensionality of the data. The general characteristics of SSDB-applications, the derived requirements with regard to a storage management system and the specific open architecture as well as the reference implementation of CUBEStore are described in this paper.

1. Introduction

With the advent of "Online Analytical Processing" (OLAP, [3]) and "Data Warehousing" as its data delivering platform, the well-known database research field of "Statistical and Scientific Databases" (SSDB, [18]) comes to a rejuvenation. Typically, SSDBs are used to store and retrieve satellite weather data or the numerical results of large physical experiments. They mostly still stick to tables for presenting results of a query. OLAP applications show the same data access and analysis characteristics, just replacing statistical tables by multidimensional data cubes ([19]).

In certain points, the characteristics of SSDBs differ remarkably from standard "Online Transaction Processing" (OLTP) database applications nowadays commonly built on top of relational database systems. Therefore, proprietary systems tailored to fit the special needs of SSDB applications are quite likely to be found in this area. Some of the points to be taken into regard are:

- efficient access to very large multidimensional data volumes
- only a limited amount of data is needed per request
- the physical storage structure should in some way reflect the logical, i.e. multidimensional structure of the data

The overall goal of the CUBESTAR project is to build a reference system matching these requirements. According to the ANSI/Sparc 3-Level Database Architecture ([21]), the CUBESTAR project works on all three levels of this architecture, thus trying to maximize usability and performance without the limitations of other projects approaching just one of these levels.

On the *conceptual level*, the *CROSS-DB data model* ([12]) is used to represent raw facts and figures as well as the basic structures supporting the data analysis phase. The description of different users' views of the data corresponds to the *external level* and is performed by queries formulated in the CUBEQueryLanguage *(CQL*, [1]). In general the distinction of several application-oriented external views on the same application-independent conceptual schema provides *logical data independence*.

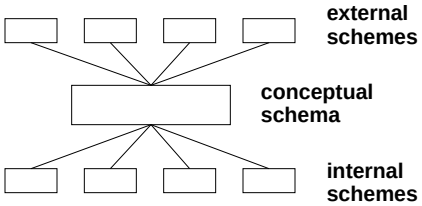

Figure 1. ANSI/Sparc 3-Level DB Architecture

CUBEStore, as to be described here, represents an important part of the *internal level*, providing CUBESTAR with adequate support for physically storing and retrieving data. The separation of a conceptual level from an internal level provides *physical data independence*, meaning that the model of the mini world is independent from its implementation. The most important means to make CUBEStore more efficient for SSDBs than standard database architectures are *distribution* and *replication.* In some way, CUBESTAR can be viewed as some kind of MiddleWare ([2]), providing efficient access to aggregated information based on large data volumes distributed over different servers and over different storage media. Thus, from an user's point of view, CUBESTAR implements a fast '*Information Everywhere'* service, with CUBEStore as the technical basis for data distribution and replication.

Referring more specifically to CUBEStore, the next section covering related work presents some other approaches of how to overcome the shortcomings of traditional database systems for multidimensional databases. The third and fourth section describe the characteristics and the requirements of multidimensional data respectively in more detail. Afterwards, the design of CUBEStore is presented, and finally, interesting features of the implementation are highlighted. The paper concludes with a short summary.

2. Related Work

In general, two streams of implementing multidimensional data models can be distinguished in scientific literature as well as in commercial products ([4]). The 'relational'-OLAP approach (ROLAP) transforms each query formulated in the multidimensional view into an SQL-statement, to be processed on a relational database engine. On the other side, the 'multidimensional'-OLAP approach (MOLAP) implements the multidimensional model directly on the physical layer. Since we (and probably the majority of the database research community) believe that there is no unique winner in this discussion, both directions should be supported by a database system providing efficient data access for OnLine Analytical Processing. Hence, CUBEStore is designed to be an *open* storage management system, capable of using relational *and* multidimensional drivers. As our outline of the basic CUBEStore architecture especially emphasizes the pure multidimensional way, we are now going to have a short look at other projects dealing with open storage management in general and multidimensional storage techniques in detail.

Supporting database systems by a sophisticated storage subsystem is a long story in database research ([20]). Furthermore, much work was done by Sarawagi, incorporating tertiary storage media *directly* into the database system ([17]). A different, *indirect* way of

extending the capabilities of existing database systems is described in [14]. A virtual disk storage manager called Daisy is offered, operating similar to a virtual memory manager by transparently ‘swapping’ data blocks between different storage media, e.g. hard disk and tape drives. This is an interesting method to enhance existing systems, but one drawback of its transparency is the lack of integration into the database system, making it more difficult to control the activities of the storage manager.

Addressing the problem of multidimensional storage techniques, we will sketch two different ways to tackle this problem. The NASA’s *CommonDataFormat* (CDF) and its extension *NetCDF* [6] are similar to CUBESTAR in the fact that they stand for a new, self-contained system adopted to the needs of multidimensional data as good as possible. But in contrast to the aim of the CUBESTAR project to build a full-featured database system as explained before, CDF mainly consists of a combination of the I/O-library and utilities to use this library. NetCDF extends the CDF data format to become more flexible and machine-independent. Thus, NetCDF might become a single module of the CUBEStore system in the future.

Relational database vendors are aware of the need to provide multidimensional data access. Oracle for example adds a new data type called *HHCODE* to their relational database system ([16]). This way offers the ability to integrate relational and multidimensional data in one system, a fact that might be outstanding to many possible users, as it allows a soft migration. On the other hand, it will be hard to adopt such a system fully to the needs of multidimensional applications. For instance, with an HHCODE-type field, it is not possible to query for exact data, only ranges can be requested, so an exact reproduction of the original values is rather hard. In general, adding multidimensional extensions to relational database systems (like the DataBlade technique of the Informix Universal Server [9]) might solve some users’ needs rather comfortable, but it seems not appropriate for building a fully multidimensional application.

3. CUBEStore-Characteristics

This section tries to make the reader more familiar with the characteristics of SSDBs in general and some of the consequences for a storage management system. Afterwards, a model for redundant storage of information will be sketched.

3.1. Sample Scenario

One of the roots of the CUBESTAR project lays in a series of projects with our industrial partner, the worldwide operating retail research company “GfK”. Their task can be shortly characterized by the following analysis steps: in a first phase, sales figures of single articles in single shops (mostly from Point-of-Sales systems) are gathered, classified and periodically added to a common raw database. Based on these data, typical statistical analyses like distributions, top-n calculations, etc. are performed. In general, the GfK-database is a good example for the basic properties determining the character of SSDBs [10]:

- multidimensional data structure
- data access according to dimensional structures
- very large amounts of data
- periodic bulk updates with an append-only semantic for the vast majority of data

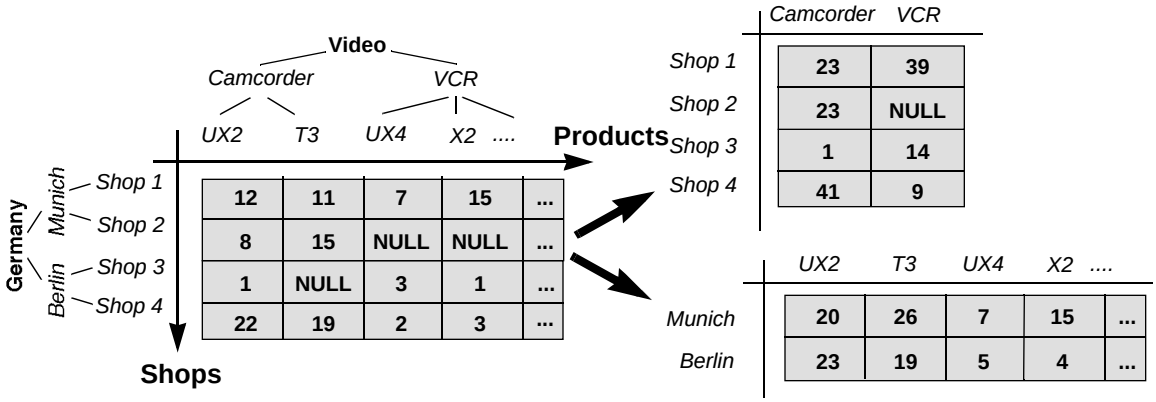

Figure 2. Sample Sales Figures of Video Equipment in Germany

For example, the left side of figure 2 illustrates some sample sales figures in a two-dimensional context. The hierarchically organized dimensional structures of the dimensions "Products" and "Shops" prescribe data access during the analysis process. Another interesting feature of multidimensional data which can also be seen in this example is the fact that data density is often very low, i.e. most of the possible multidimensional values are not existent. In our example densities between 4% and 20% were observed.

The right side of figure 2 shows two possible derivations: the upper one holds summarized values according to different product families subsumed by the notion of 'Video'; the lower one reflects the result of an aggregation process in the geographic direction. The raw database has large data volumes (10 GBytes in total for sales figures for German shops in a single gathering period for the current retail research scenario). Hence, for answering queries efficiently, data has to be *aggregated* according to different dimensions and at many different levels of detail, or *granularity*, even differing between the dimensions within one query. Typically, sales figures like Camcorder versus VCRs are requested not for single shops, but for instance summed up for the whole of Germany. On the other hand, the area manager of a retail chain running many stores might be interested in how his shops in Munich compare to those in Berlin.

Therefore, the system has to provide additional mechanisms to be able to cope with such requests. The problem becomes even worse, as all the data is kept for at least two years and used for time-oriented sequence and long-term analyses. Thus, time as a third dimension, missing in figure 2, must be added in the model and the data volumes must be multiplied by a corresponding factor (24 for two years on a monthly reporting frequency).

3.2. A Redundancy Model supporting SSDB-Applications

The append-only characteristic of SSDB-applications leads to the natural conclusion to use *redundancy* as a means to make such database systems faster. A project with GfK, published in [11], showed that an impressive speedup can be gained by using aggregated data instead of raw sales figures, and if the basis for those aggregated data stays rather stable, they can be reused quite often.

From a more general point of view, redundancy in the context of SSDB can be seen as representing one logical object x by n physical objects x_i, (i=1,...,n), with n being the number of physical representations. Each physical object x_i is generated by applying a function $x_i = f_i(x)$ to a logical object. According to the applied function f, two different kinds of redundancy can be distinguished:

- *Horizontal Redundancy (= Replication):*
  Using the identity function $x_i = f(x) = id(x)$, the physical object corresponds to the logical object. This case, with two or more physical objects[1], represents the classical way of replicating data to increase local availability and local access performance of remote data.
- *Vertical Redundancy (= Aggregation):*
  In this case, f corresponds to an aggregation function like $x_i = SUM(x)$ or $x_i = AVG(x)$. The derived physical objects can be seen as persistent results of a computation on a different object, probably of finer structure. Again, this is done mainly for performance reasons, to prevent often needed data to be recalculated from mostly unchanged raw data.

4. Requirements for an SSDB-Storage System

Consequently, three major requirements were identified for CUBEStore, influenced by the characteristics described above and the use of redundancy to improve query performance:

- being able to support different storage media
- being freely configurable, especially *what*, *where* and *how* data is stored
- reflecting the multidimensional data structure physically

The reasons for integrating different ways of storing data are now described, followed by an explanation of the need for a flexible way to configure a multidimensional storage manager, including the ability to reflect the multidimensional data structure physically.

4.1. Support for Different Storage Media

As SSDB applications are often used for several years, sometimes even more than a decade, the amount of data to be administered by CUBEStore can easily become magnitudes larger than other databases and reach several terabytes. Obviously, a limitation to magnetic hard disks as single mass storage medium is rather inappropriate here. Accordingly, the *open* concept of CUBEStore supports, by means of its *abstract, media-independent* interface, almost any storage medium, thus giving the user the possibility to decide which one fits his needs best. Organized in a hierarchic pyramid, the well-known principle of using cheaper but slower media for less often needed data can thereby be applied (figure 3). This includes, of course, the use of network drivers for horizontal redundancy.

Nevertheless, at least from within the system it should be possible to query the costs of accessing data on different storage media. This will enable the query optimizer to decide which volume to choose, if different options are available. In general, these costs can be

1. Without going in to much detail, it must be noted that at least one $x_i = id(x)$ must be present in the system

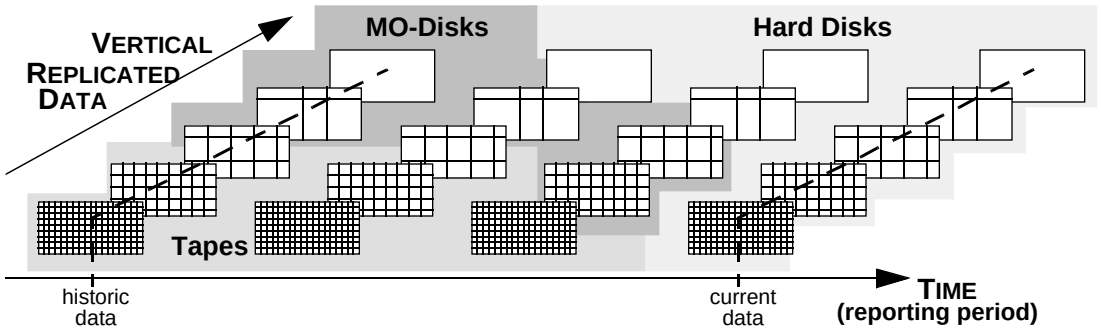

Figure 3. Data Distribution within the Storage Hierarchy

divided into *media access costs* and *transfer costs*. The former describe the costs for gaining access to a CD-ROM in a juke box, for instance, independent of how much data is needed. The latter tells the system about the actual transfer costs, normally of course dependent on the amount of data to be transferred. There are several other interesting characteristics about mass storage media that can be taken into regard. In general, the system needs to *classify* its media, and as a consequence of the former paragraph, this information has to be maintained separately from the original data in the so-called *Meta Information System*. It must be possible to decide which media to use *before* actually accessing the chosen medium.

4.2. Flexible Configuration

In addition to the claimed support for different storage media, their usage should be fully *configurable*, to adopt a system to the changing needs of its users. A global *Allocation Manager*, supported by local *Allocation Agents* might then move older or less often used data to a cheaper medium according to certain preset parameters, maybe on a different site. Their tasks include to decide what and how many physical data partitions should be used for one logical data partition. For instance, this can be determined according to access patterns, given priorities or granularities. Another part of their work is, as said before, to decide which sites should replicate frequently used data. Figure [4] shows what such a configuration could be like. It includes an example for horizontal replication, or duplication of data on a different site and for vertical replication or aggregating data.

Not only needs the system to be open to different types of storage media, equally important is its capability to support different *strategies* and *algorithms* transparently, for example with regard to the way the multidimensional data is linearized and compressed. In particular the chosen sequential ordering is a key to improve or slow down query perfor-

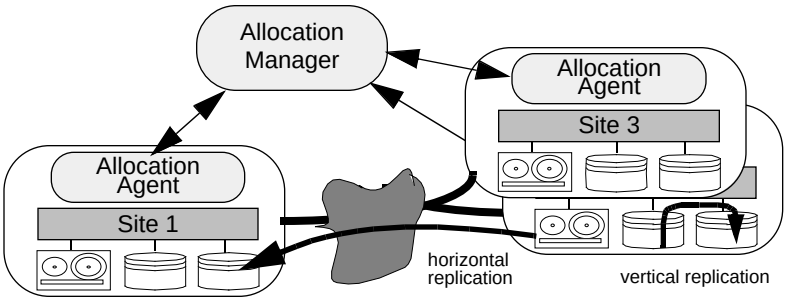

Figure 4. Allocation of Mass Data at Different Sites and on Different Storage Media

mance. Therefore, the physical representation of the multidimensional data must be chosen very carefully and it should be possible to adopt it to the needs of different applications. Furthermore, as a consequence of the low data density, intelligent compression techniques are a real necessity for efficient data storage.

To put it into a nutshell, CUBEStore provides, as a result of these requirements, an open, configurable data storage system for handling large volumes of multidimensional data with a potential high sparsity. Support for horizontal and vertical replication of data with regard to geographic distribution as well as distribution over inhomogeneous storage media, makes CUBEStore the ideal basic technique for a database management system fitting SSDB-applications' needs.

5. Design

Having explained the requirements that determined the design of CUBEStore, first of all the CUBEObjects as the core entities of CUBEStore are described. This is followed by a basic scheme for the possible division of work between the drivers within a CUBEObject. Finally, two possible configurations are presented. Some of the ideas used in this approach can be found in [5], [8], [6] and [16].

5.1. The CUBEObjects

According to the requirements described before, design and architecture of the CUBEObjects, the most important components of CUBEStore, are presented in this section. A CUBEObject is a logical representation of a data partition. Figure 5 illustrates how its main parts work together:

- CUBEDrv: driver inside a CUBEObject
- CUBEData: data objects for these drivers
- CUBEIniFiles: configuration files of the CUBEObjects

A CUBEObject is something like a black box offering a well-defined interface to its users to store and retrieve data. The data exchanged between them and a CUBEObject or its CUBEDrv is called CUBEData. The CUBEDrv form the core of a CUBEObject, they determine what and how it performs. A driver queue contains a list of drivers according to the configuration found in the CUBEIniFile belonging to that object. Getting CUBEData in the standard format described later, a CUBEObjects involves its first driver asking him to process the CUBEData. The CUBEDrv transforms the CUBEData according to the requirements of the next driver in the queue, and this goes on until the last one succeeds in storing CUBEData, or in retrieving the requested data from the storage media and handling the CUBEData back upwards the driver queue.

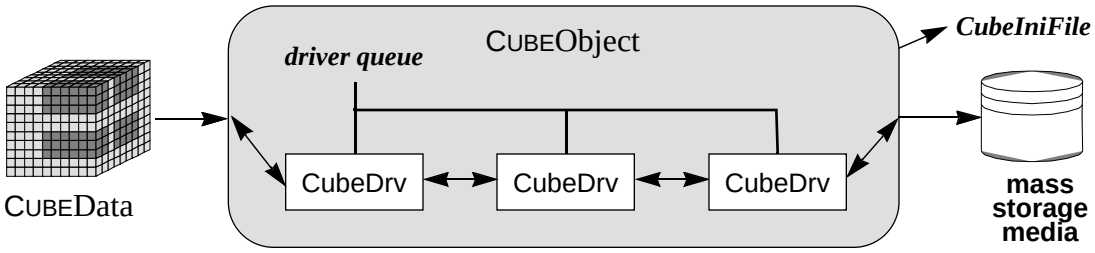

Figure 5. Architecture of a CUBEObject and its components

The division into several CUBEDrv ensures modularization and reusability. A new compression technique, for instance, can easily be added by implementing just a compressing driver and changing the CUBEIniFile as to load this driver to its place in the driver queue. Or, to use the recovery capabilities of a standard database system, only a simple driver transforming CUBEData into an SQL query is necessary. The same argument holds for network drivers in a distributed system or, e.g. for dummy drivers to overcome the limitation to one CUBEObject, thus enabling clustering of several logical data partitions data into one physical partition.

5.2. A Basic Scheme

The first step was to develop a basic scheme on how to divide work between the different drivers. According to this scheme, the task of storing multidimensional data partitions on a storage medium consists of dividing the partitions into a number of data blocks. Secondly, a format for storing (and compressing) data within these blocks has to be found. And finally, the data blocks have to be written to or read from the medium. In other words, we have got three different kinds of drivers (figure 6):

- a *block directory driver* responsible for splitting the partition into blocks
- a *block access driver* determining the storage format within these blocks
- a *block driver* reading and writing blocks to an external medium

One drawback of splitting a data partition into several blocks is the fact that additional administration has to be done by the block directory driver. On the other hand, to maintain a minimum of flexibility, in particular with respect to write performance, e.g. when adding or correcting some data, some kind of division is needed anyway.

5.3. Two Configurations

Using the basic scheme of the last section, two possible ways to interpret this scheme are described now. The first aims at using storage media of a well-known fixed block size, whereas the second one is more appropriate to devices working with sequential access. Finally, some remarks are made about using tape drives.

5.3.1. Storage Media with Fixed Block Size

This configuration is the first choice when storing data on normal hard disks. This implies a fixed block size and random access to all blocks. It is now explained using figure 7:

To start at the bottom of the driver queue, the task of the block driver is rather simple. Block number and fixed block size enable it to find the requested block quite easily. The same holds for replacing one block with another or adding new blocks.

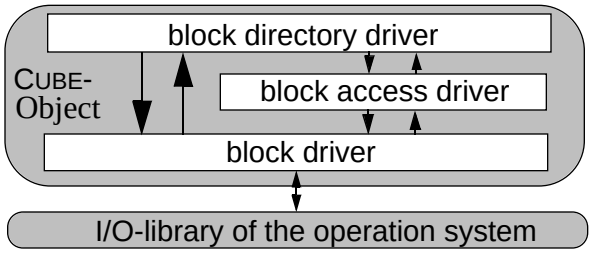

Figure 6. A Basic Scheme for Configuring a CUBEObjects

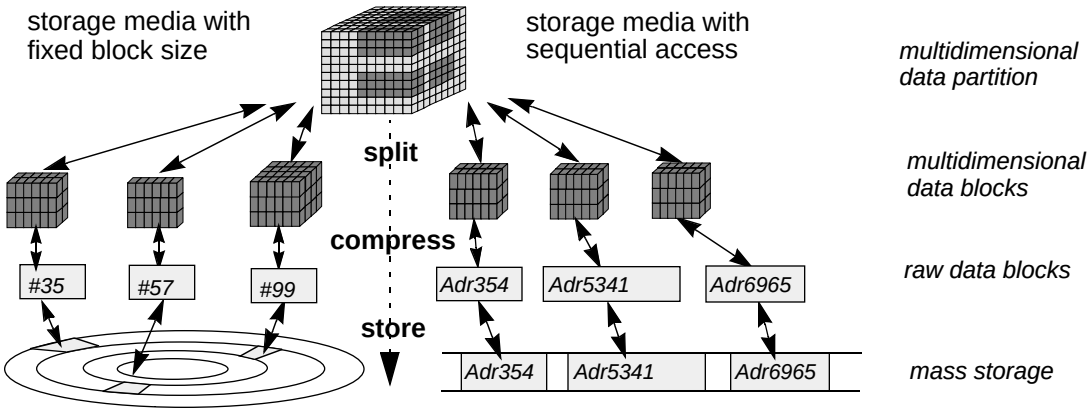

Figure 7. Two Sample Driver Configurations for Different Storage Media

The block access driver is responsible for compressing multidimensional data into a raw data block of given size. This implies that there might be some space left, or that two raw blocks might be needed.

The most complex driver in this configuration is the block directory driver, working on top of the block access driver with blocks of variable size. Therefore, further directory information is necessary to divide the data partition as good as possible and to determine the raw data block number quickly. On the other hand, removing the block access driver, thus enabling the block directory driver to use fixed size blocks as well cannot be a solution with regard to the low data density mentioned earlier.

5.3.2. Storage Media with Sequential Access

If, in contrast to the former section, the block size of the storage medium cannot be fixed in advance, this configuration might be a replacement. It uses the stream paradigm, viewing the medium as a single stream without bigger units of access. Again, it is explained using figure 7:

In this case, the data partition can be divided according to a standard mapping into fixed size blocks, resulting in a rather simple block directory driver. Now the block access driver gets blocks of a fixed size and compresses them as good as possible, resulting in variable sized blocks. Consequently, the block driver is becoming slightly more complex, as it has to maintain information about addresses and sizes of the raw data blocks stored on the storage medium. Therefore, this model is not appropriate if blocks are often moved or change their size.

As tape drives are often used in juke boxes or something similar, long access times are typically for this medium. Therefore, data on tapes are often compressed once more as a whole, as the time for doing so is rather short compared to access time. This can easily be done by adding another driver at the end of the driver queue and hence no extra configuration is needed for tape drives.

6. Implementation

Following the first of the two possible configurations for block devices with random access, the implementation of a prototype is now described. Again, the requirements are presented first. Thereafter, the implementation of a CUBEObject is explained. MemCubes, the means of exchanging data with a CUBEObject, get a section of their own, followed by a short overview over the different drivers of this implementation.

6.1. Analysis and Requirements

The standard requirements for modern software like flexibility, portability, extensibility and reusability must be inherent properties of an open system like CUBEStore, capable of adopting to different and changing needs. Nevertheless, the overall performance goal of being faster than standard database systems has to be reached, as this is the key factor for its existence. In addition to this, the traditional solution of using more or faster hardware might be slightly harder with data volumes reaching several terabytes, as said before. Hence, the C++ programming language has been chosen for the implementation. Its object-oriented features offer almost everything necessary for modern program design. Static type checking and nothing like a runtime garbage collection enable performance. Moreover, the availability on almost all modern computer architectures is the basis for portability.

The relatively clear distinction between the production and query phase in the lifetime of multidimensional SSDB data leads to the decision to look at fast read-only access as the more important feature. A lot of emphasis was put on ensuring good read performance regardless to the requested dimensions. This implies that a traditional primary key linearization could not be used. The implementation was simplified by the restriction to numerical data, leaving coding and decoding to a separate module. The units administrated by CUBEStore are data partitions. Dividing and joining these units is the task of the already mentioned Allocation Manager. As the query optimizer can be multithreaded, the CUBEStore code has to be reentrant, but not necessarily multithreaded itself.

Error handling, at least for unexpected errors, is done almost completely by throwing exceptions, hierarchically organized and named by a unique combination of major and minor error number. For handling standard tasks, the class library GNU libg++ was used.

6.2. The CUBEObjects

As said before, a CUBEObject is the logical representation of a CUBEStore data partition. Data can be accessed only if the requested CUBEObject has been opened before. This is done according to the description in the CUBEIniFile, so the name of a CUBEIniFile identifies a CUBEObject. It also serves as a simple locking mechanism. A more sophisticated name and access administration is to be implemented in future versions of CUBEStore. On the other hand, file system methods enable e.g. the transparent use of network devices. Each CUBEDrv to be loaded for a CUBEObject has its own paragraph in the *driver section* of the CUBEIniFile, with the content being dependent on the driver. The *cost section* in the CUBEIniFile describes access and transfer costs as explained at the beginning of the design section.

To prevent the CUBEObjects from having to know the type of each CUBEDrv it uses, thus making it impossible to use drivers written after the compilation of a CUBEStore application, the drivers are generated by a separate, global or local CUBEDrvFactory with a standard interface for all drivers. As a result, only this driver factory needs to be updated if new driver classes are developed. Accessing data in a CUBEObject is done by two basic methods, Read and Write. They transfer data to and from CUBEObjects in the MemCube format described later. Actually, as shown above, they just order the first driver in the queue of the CUBEDrv in a CUBEObject to start working.

```
Bool Read(MemCubeData<T>& cube_data) {
  return (drv_queue.First())->Read(cube_data);
}
```

```
Bool Write(MemCubeData<T>& cube_data) {
  return (drv_queue.First())->Write(cube_data);
}
```

Data is held in main memory using the multidimensional MemCube class. Speaking more exactly, the MemCube module contains a group of classes: Firstly, there is the class CUBEIndex describing a multidimensional point by its coordinates. Two points form a convex ScanRegion. The class MemCubeContainer is responsible for storing data in a format similar to C-arrays. The MemCube class itself offers a comfortable interface to access data in a multidimensional way, using the three helper classes just mentioned. Among others, it allows to compare, replace or resize MemCubes as a whole.

6.3. The Drivers

This section corresponds to the largest portion of the implementation effort, the CUBEDrv. Based on the configuration for random access block devices, a variety of drivers was developed. Their hierarchical structure is shown in figure 8. Corresponding to each layer of the basic scheme explained in the design section, there is a base class, containing an abstract interface definition of the service offered by this layer: BlockDirDrv, BlockAccessDrv and BlockDrv.

6.3.1. The Block Layer

To start bottom-up, the block layer is the most low-level layer providing others drivers with the capability to read and write a raw data block of a fixed size as specified in FixedSizeBlockDrv. The twins SyscallBlockDrv and StreamBlockDrv perform the task of transferring data to and from disk, currently using system calls or C++ streams. Management of

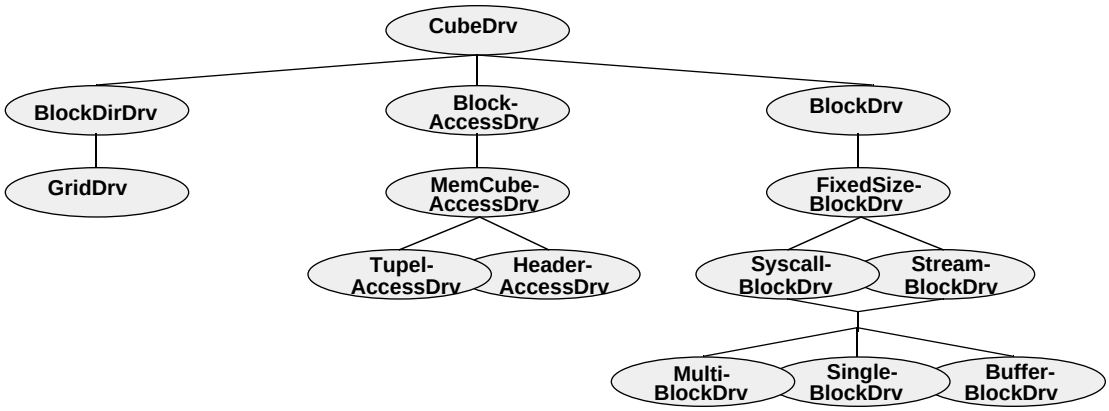

Figure 8. Hierarchy of Driver Classes

the dynamic main memory used for the blocks read is done in three different ways by MultiBlockDrv, SingleBlockDrv and BufferBlockDrv. The first one does almost no optimization, the second one tries to reuse memory as far as possible, whereas the third one serves as a front end to a global system buffer implemented in a separate module. Thus, the BufferBlockDrv demonstrates of how to overcome the borders of a CUBEObject, useful e.g. for implementing a storage format clustering several logical partitions into one physical.

Their base class is, as shown by the following short code extract, only given as a template. By using this enhanced mechanism, it became possible to unify the use of two different I/O -drivers with three different dynamic memory management drivers in quite an elegant way:

```
template <class T>
class MultiBlockDrv : public T
{
        public:
                MultiBlockDrv(CubeObject* cubeP, const int size, const String& file_name, const Mode mode=OPEN)
                        : T(cubeP, size, file_name, mode) {}
}
```

6.3.2. The BlockAccess Layer

In this configuration, it is the task of the block access layer to ensure that the multidimensional data are stored in blocks of fixed size, and as a consequence, the block access layer determines the format within these blocks. As it can be seen in figure 8, there are two implementations of block access drivers, namely HeaderAccessDrv and TupelAccessDrv. The latter stores a multidimensional value just by adding its coordinates. Even this simple format incorporates a compression of data, as long as the data density is lower than $1/(1+n)$ with n being the number of dimensions. Changing data, including adding or deleting values, is rather easy with this format.

A more sophisticated format is implemented in the HeaderAccessDrv: This driver uses the SingleCountHeader method according to [7]. With this method, one *distinct* value can be removed from an array, here the NULL standing for 'data not available'. To do this, a header noticing the distance between non-NULL values is calculated. Standard C linearization is used to create the array needed for compression out of the multidimensional data, so this format is not as stable to operations resizing the multidimensional data block. On the other hand side, since no space is needed for unavailable data, resize operations should not be necessary too often, an advantage of this format over alternative methods like a bitmap header.

6.3.3. The Directory Layer

The most sophisticated driver in this model is the block directory driver, whose task is to split the multidimensional partitions into smaller blocks. To fulfill the requirement of being rather independent to the dimension used for access, the grid file format originally proposed in [15] has been used. On average, its performance over all dimensions is quite good, although a method using the primary key for linearizing might be better with respect to this special dimension. An additional benefit of this format is its local character, with

only limited changes after adding or deleting data. Furthermore, its open character makes it an ideal partner for different block access drivers.

A grid directory consists of two different parts: A grid scale for each dimension and a multidimensional grid array. The latter reveals the data block numbers where the requested data can be found, and the scales indicate the upper and lower bounds of the data contained in a block. Using the example given in figure 9, it can be seen that all values with coordinates between 13 and 21 in dimension 1 and between 66 and 81 in dimension 2 are contained in block number 27.

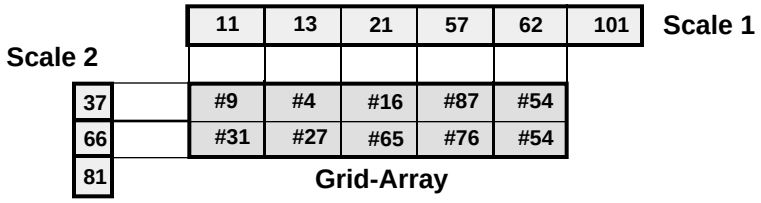

Figure 9. Sample Grid Directory

Consequently, especially *range queries* asking for data within some upper and lower bounds, quite important in multidimensional applications, can be answered straightforward.

The implementation of the grid directory is divided into two classes, a driver class and a directory administration class. Furthermore, the dimension chosen for splitting if the directory has to be extended can be determined according to different strategies like for example round-robin, fixed or toggle, therefore allowing to distinguish between more often needed dimensions with higher granularity scales and less often needed dimensions with lower granularity scales.

7. Summary

Basically, the development of the CUBEStore storage manager included two big tasks: Firstly, a design flexible enough to fit the needs of different SSDB applications had to be developed. This design was presented in section four. To prevent the description from becoming too abstract, two possible ways of realizing such a configuration were added. Nevertheless, further work in this area, especially in defining a more sophisticated partition administration than the basic CUBEIniFile mechanism might be helpful, but has to be done in close connection with the development of the CUBESTAR run time system ([13]).

The second part was to show, in the implementation section, that this design could be realized. As this implementation was only a prototype, several interesting concepts could not be realized. In this context, a completely relational database interface in addition to the special purpose design presented here might be useful.

Despite these remarks, the design and implementation of CUBEStore, a multidimensional storage manager, has been described, which was developed with respect to the needs of SSDB applications. CUBEStore offers a useful basis for further developments in this area.

References

[1] Bauer, A.; Lehner, W.: The Cube-Query-Language for Multidimensional Statistical and Scientific Database Systems, in: *5th International Conference on Database Systems For Advanced Applications* (DASFAA'97, Melbourne, Australia, April 1-4, 1997), pp. 263-272

[2] Bernstein, P.A.: Middleware, in: *Communications of the ACM, 39(1996)2*, pp. 86-98

[3] Codd, E.F.; Codd, S.B.; Salley, C.T.: *Providing OLAP (On-line Analytical Processing) to User Analysts: An IT Mandate*, White Paper, Arbor Software Corporation, 1993

[4] Colliat, G.: OLAP, Relational, and Multidimensional Database Systems, in: *SIGMOD Record, 25(1996)3,* pp. 64-69

[5] Cabrera, L.-F.; Steiner, S.; Penner, M.; Rees, R.; Hineman, W.: *ASDM: A Multi-Platform, Scalable, Backup and Archive Mass Storage System*, Research Report RJ 9936, IBM Almaden Research Center, San Jose, CA, 1995

[6] Davis, G.; Rew, R.: NetCDF: An Interface for Scientific Data Access, in: *IEEE Computer Graphics and Applications,* 1990, pp. 76-82

[7] Eggers, S. J.; Olken, F.; Shoshani, A.: A Compression Technique for Large Statistical Databases, in: *7th International Conference on Very Large Data Bases* (VLDB'81, Cannes, France, 1981), pp. 424-434

[8] Fortner, B.: *The Data Handbook*, Springer Verlag, New York, 1995

[9] N.N.: Informix Universal Server, Product Information, Informix Corp., (http://www.informix.com/), 1997

[10] Lehner, W.; Ruf, T.; Teschke, M.: Data Management in Scientific Computing: A Study in Market Research, in: *International Conference on Applications of Databases* (ADB'95, Santa Clara, California, Dec. 13-15, 1995), pp. 31-35

[11] Lehner, W.; Ruf, T.; Teschke, M.: Improving Query Response Time in Scientific Databases using Data Aggregation - A Case Study, in: *7th International Conference and Workshop on Database and Expert Systems Applications* (DEXA'96, Zürich, Switzerland, Sept. 9-13, 1996), pp. 201-206

[12] Lehner, W.; Ruf, T.; Teschke, M.: CROSS-DB: A Feature-extended multi-dimensional Data Model for Statistical and Scientific Databases, in: *5th International Conference on Information and Knowledge Management,* (CIKM'96, Rockville, Maryland, Nov. 12-16, 1996), pp. 253-260

[13] Lehner, W.; Teschke, M.: On the Architecture of a Multidimensional Database System for 'Decision Support' Applications, *Technical Report TR-97-132*, University of Erlangen-Nuremberg, Erlangen, 1997 (in German)

[14] Menon, J.; Treiber, K.: *Daisy: Virtual Disk Hierarchical Storage Manager*, Research Report RJ 10075, IBM Almaden Research Center, San Jose, CA, 1997

[15] Nievergelt, J.; Hinterberger, H.; Sevcik, K.C.: The Grid File: An Adaptable, Symmetric Multikey File Structure, in: *ACM Transactions on Database Systems 9(1984)1*, pp. 38-71

[16] N.N.: *Oracle Spatial Cadridge*, Product Information, Oracle Corp., (http://www.oracle.com/st/cartridges/spatial/), 1997

[17] Sarawagi, S.: Query Processing in Tertiary Memory Databases, in: *21th International Conference on Very Large Data Bases* (VLDB'95, Zurich, Switzerland, Sept. 11-15, 1995), pp. 585-596

[18] Shoshani, A.: Statistical Databases: Characteristics, Problems, and Some Solutions, in: *8th International Conference on Very Large Data Bases* (VLDB'82, Mexico City, Mexico, Sept. 8-10, 1982), pp. 208-222

[19] Shoshani, A.: OLAP and Statistical Databases: Similarities and Differences, in: *16th Symposium on Principles of Database Systems* (PODS'97, Tuscon, Arizona, May 12-14, 1997), pp. 185-196

[20] Stonebraker, M.: The Design of the POSTGRES Storage System, in: *13th International Conference on Very Large Data Bases* (Brighton, England, Sept. 1-4, 1987), pp. 289-300

[21] Tsichritzis, D.C.; Klug, A.: The ANSI/X3/SPARC DBMS framework report of the study group on database management systems, *Information Systems 3(1978)3*, pp. 173-191