

Maitri: Format Independent Data Management for Scientific Data

Rishi Rakesh Sinha

Soumyadeb Mitra

Marianne Winslett

Department of Computer Science, University of Illinois at Urbana-Champaign

{rsinha, mitra1, winslett}@uiuc.edu

Abstract

*Today's scientific applications are very data intensive, and their data management requirements can no longer be met by special-purpose libraries for particular scientific data formats or by traditional database management systems. This paper proposes **Maitri**, a data-format-independent, loosely-coupled, application-tailorable set of libraries that provides a holistic data management framework for scientists. A scientific programmer can write applications that access scientific data in its native format, while using Maitri's APIs for buffering, indexing, metadata management, and concurrency control, plus a small amount of format-specific code written by the programmer. Experiments to date with Maitri's buffer and index managers show that this format independence can be obtained at an acceptable runtime cost.*

1 Introduction

Scientific applications can be broadly classified into observation/experiment driven, simulation driven, and information driven [5]. All three types can easily produce or consume terabytes of data. For example, observational applications like MODIS, one of many instruments on board the EOS Aqua satellite, produces 144 MB/hour just for its calibrated radiances 5km sub-sampled data. At the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois at Urbana-

Champaign (UIUC), a complex simulation run produces hundreds of gigabytes of data. Other UIUC scientists use 7GB working sets of NASA satellite data to study vegetation and climate patterns in the Appalachians [7]. Future scientific applications will be even more data-intensive, due to improved observation-gathering technology and more complex and larger-scale simulation codes.

Some scientists have turned to commercial database management systems (DBMSs) to provide easy and efficient access to their data: scientific tools submit declarative queries to the DBMS, which optimizes and executes the queries. For example, the USD [6] and Tioga [1] systems for scientific data access were built on top of standard relational or object-relational [11] DBMSs. These works were developed specifically for visualization tools (an important need for scientists) and did not try to address the broader aspects of scientific data management. As another example, the bioinformatics community initially standardized on the Oracle relational DBMS, but has started looking for alternatives. More generally, it is well known that traditional DBMSs are not suited to meet the broader needs of scientific data management [5]. For example, traditional DBMSs require significant performance tuning for scientific applications, a task that scientific programmers are not well prepared or eager to address. Since most scientific data are write once, and read-only thereafter, traditional relational concurrency control and logging mech-

anisms are an expensive liability. The pricing structure of traditional DBMSs is ill suited for scientists, who commonly use multiple platforms spread over a wide geographic area, often with different architectures and operating systems. For the scientists' codes to be easily portable to all their platforms, the same DBMS needs to be running on all the platforms, which is very expensive. Parallel database software is especially expensive, and is not available for the leading-edge parallel machines popular in scientific computing centers.

Today most scientific data is stored in binary files whose formats were developed specifically for scientific data, and accessed through user-level libraries. These formats range from the generic (e.g., HDF (hdf.nsa.uiuc.edu) and netCDF (my.unidata.ucar.edu/content/software/netcdf)) to the domain-specific (e.g., ROOT (root.cern.ch) for high energy physics and FITS (heasarc.gsfc.nasa.gov/docs/heasarc/fits.html) for astronomy). These systems store data together with their associated metadata, and provide APIs for efficient storage and retrieval of the data.

A major problem with these formats is that they require scientific programmers to write code using the APIs specific to that format. The programmer must program the details of how to navigate through the data to find the items of interest, as in 1960s commercial DBMSs [2]. Moreover, the resulting applications are format-specific, and must be rewritten if the format changes. In addition, these libraries are oriented towards operations on a single file at a time, while scientists usually have a huge number of files. Thus scientists must manage the file-level metadata themselves, typically through a complex file-naming scheme. Further, the libraries do not provide buffering, caching, and indexing facilities that work across multiple files. Finally, scientists often need to access data from a variety of sources, each with a different storage format; their application code must use a separate API for each format.

To address these shortcomings, scientists need

a unified query interface that supports queries over data and metadata stored in a variety of formats, and provides flexible support for buffering, caching, indexing, metadata management, and concurrency control. In this paper we propose Maitri, a set of *data-format-independent, application-tailorable* libraries to provide these facilities. Maitri provides the scientific application developers with a set of lightweight, mix-and-match components that the developers can combine as needed to provide holistic data management facilities for a particular application. Scientific application developers can use the Maitri components they need and omit the remainder, which simplifies application development, flattens the Maitri learning curve, and avoids unnecessary runtime overheads.

2 System Architecture

Figure 1 shows the architecture of the Maitri system, which has six main components: the query, metadata, concurrency control, index, buffer, and block managers. Scientists access their data through tools that call Maitri, such as visualization programs, simulation codes, and data transfer tools.

For example, the scientist may fill out a form in a visualization tool, asking to see all parts of a rocket that reached a temperature of 2000K in the first three seconds after ignition. These user-supplied parameters become values in a query the tool issues to Maitri. If the tool uses all of Maitri's components, then the query first goes to Maitri's query manager, which converts the query into an abstract syntax tree for execution. The query manager determines which indexes are available and devises a query execution plan that takes advantage of the relevant indexes. The index manager and metadata manager work together to determine which data blocks must be read, and the concurrency control manager can be consulted to acquire locks on the blocks being read, if needed. The query manager calls the buffer manager to

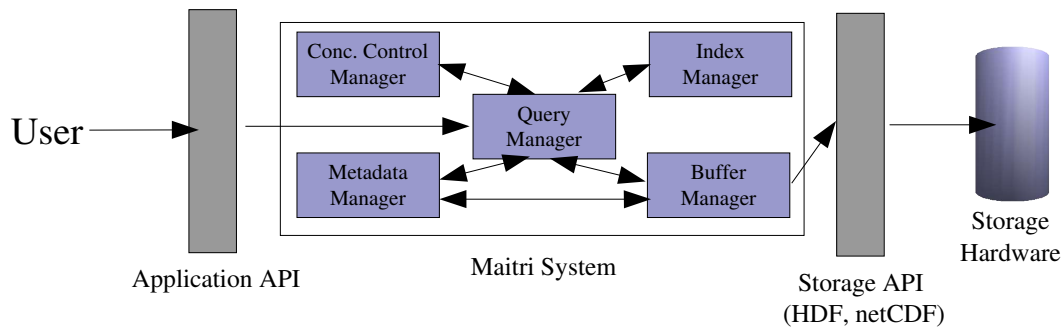


Figure 1. System architecture. The arrows show the flow of control for a query that a scientific tool has submitted to Maitri.

fetch the blocks it needs; if the buffer manager has not already cached a requested block, it calls the block manager to fetch the block.

So far, the Maitri description fits many conventional DBMSs. Maitri departs from tradition in its block manager. In traditional DBMSs, a block is a physical entity consisting of one or more logically contiguous disk blocks of a file; but in Maitri a block is a user-defined aggregate of data objects, which may or may not correspond to a sequence of logically consecutive disk blocks. All the knowledge of how to manipulate a block (reading and writing the block, extracting its attribute metadata and its data values) is isolated in the block manager, whose key routines must be implemented by the scientific tool writer who wants to use Maitri. For example, the tool developer can define each file from the output of a time step during a simulation as a separate block. Alternatively, the developer can define the temperature data from all snapshots of a run as a single block. For observational data, the developer can define each individual data set from a file containing one snapshot of data gathered by a satellite as one block, for example. The other components of Maitri operate on these logical blocks without any understanding of the underlying storage format. This feature provides Maitri with format independence, at the cost of the application developer having to choose the granularity of the blocks and implement the block access interface of the block manager. The block manager will read/write a block by making ordinary file system

read/write requests, which in turn fetch/store ordinary file system disk blocks. The goal is that with a small amount of code from the tool developer, Maitri can provide DBMS-like facilities while still being format independent.

A Maitri database is a hierarchy of objects. At the highest level is the database itself, which consists of a list of *superblocks*, an abstraction for storing metadata that is common to a set of blocks. Each superblock contains a list of the IDs of all the blocks belonging to this superblock, each of which must have the same type, i.e., use the same block manager. Thus the blocks in a single Maitri database may reside in thousands of different files, using a variety of different underlying storage formats, and possibly spread across multiple sites. The superblock also contains a list of the indexes constructed for its blocks.

A Maitri superblock is analogous to a traditional relational table, and a Maitri block is analogous to a page of the table. After reading the page, the DBMS needs to extract the tuples stored on that page, and find the values of certain tuple attributes. Similarly, Maitri must provide facilities to retrieve individual records from the block and to retrieve attribute values from records. Just as in a relational system, the developer of a tool that will use Maitri must first define a schema—Maitri’s superblocks—and define the fields in the records in that superblock. Then the developer can implement the block manager interface for each superblock. With these facilities in place, Maitri can read and write data in the database.

2.1 The Block Manager Interface

The block manager interface defines the following methods:

```

INTERFACE BLOCK {
  Block(BlockID id);
  read();
  write();
  Record getInstantiations(Attributes);
  Record getInstantiations();
}
CLASS BLOCKID {
  Hashtable storage_info;
  int id;
  BlockID(int id);
  BlockID(Hashtable table);
  read();
  write();
}

```

A BlockID object (described below) must be initialized before the corresponding block can be read or written using the block manager's *read* and *write* methods. Once a block has been read into memory, the block manager's *getInstantiations()* method can be called to read part or all of the data in the block, and place the data in newly created Maitri Record objects. The developer of a tool that will use Maitri must implement the *getInstantiations* method. Each Record object contains the values of the attributes for a particular object that resides in the fetched block.

A BlockID object contains the metadata that the read and write methods need to read/write a particular block from/to disk. The BlockID contains an integer identifier that is unique inside of the block's superblock. The remainder of the BlockID contains the metadata required by the block read and write methods supplied by the tool developer; this implementation-dependent metadata resides in the BlockID's generic "table" data structure, which is currently implemented as an extensible hash table. The BlockID's metadata is initially created when a data set is bulk loaded into Maitri for the first time (by a user-supplied routine in the database initializer) or when new data

is added to a pre-existing database. A subsequent call to the BlockID's *write* method will cause the metadata manager to store the metadata permanently on disk. Subsequent runs can initialize a BlockID object with a unique identifier, and the metadata manager uses the unique ID to find or generate the metadata to read the block.

3 Maitri System Modules

Our long-term intent is to have a variety of implementations of the major Maitri modules. Current module implementations include a very simple query manager and no concurrency manager (since our scientists use read-only data). In the remainder of the paper, we will discuss Maitri's current metadata, index, and buffer managers.

3.1 Metadata Manager

Many researchers have addressed metadata management for scientific data (e.g., [9, 10, 4]). Most of these researchers use a database engine to store the metadata, with the actual data residing in scientific-format binary files. This guarantees fast file I/O for data and sophisticated data management capabilities for metadata. However, as scientists move between platforms, they either must have their metadata DBMS available on all platforms (costly and annoying) or else use remote calls to access the metadata DBMS (very slow at run time). Maitri's metadata manager is loosely coupled with the other Maitri modules, so that any existing metadata manager implementation can be wrapped and used in Maitri; the metadata could be stored in a DBMS, special-format binary files, simple ASCII files (most portable), or even as mathematical formulas or lookup tables to map a block identifier to the metadata required to read, write, and parse the associated block.

3.2 Buffer Manager

Maitri's buffer manager is a modified and extended version of the Godiva buffer manager [8],

which uses a relatively simple API to allow the programmer to specify blocks of data that can be read and stored in memory together. Godiva also allows the application developer to specify prefetching and buffering hints, which Godiva will use in deciding when to fetch/write a block and when to evict a block from its cache. Godiva fetches and writes blocks in a background thread. In Godiva, the block manager is an integral part of the buffer manager. In contrast, a Maitri block is an independent entity that can be read and written without a buffer manager; all Maitri modules must call the appropriate Maitri block manager when they need to read or write blocks on disk, and when they need to extract information from blocks in memory. A detailed description of Godiva and a performance study of Godiva with visualization tools can be found in [8].

3.3 Index Manager

Our recent efforts in Maitri have focused on the index manager. Scientific data, with its huge dimensionality and staggering volume, requires good indexing schemes for fast retrieval of relevant data, yet this topic has received almost no attention in the research community. B-trees and hashing, the popular schemes for indexing in traditional databases, are for single attributes, and are not very helpful for answering multidimensional queries. The numerous proposals for indexing multidimensional data all seem to have problems with high-dimensionality data sets. Unfortunately, scientific data sets routinely include hundreds of attributes, and a query may involve conditions on as many as 40 attributes [12]; yet R-trees, to pick one example, degrade severely in performance after 15 dimensions.

We know of only one proposal that addresses these issues in a manner that may be suitable for high-dimensional scientific data sets; its authors suggest the use of bitmap indexes [12]. A bitmap index for an attribute value v is a sequence of n bits, where the n th bit is 1 if the n th object in the

data set has value v for that attribute. A set of bitmaps can be combined using bitwise AND and OR to quickly find all the objects that have a particular combination of attribute values. Bitmap indexes are not used in Online Transaction Processing systems as updates are very common in these systems, and bitmap index updates are slow. Since most scientific databases are written once and are not updated thereafter, bitmap indexes are potentially very suitable for scientific data.

Maitri uses an extended version of bitmap indexes proposed by [12] that can index floating point data, which are common in scientific data sets, as well as categorical data. Floating point data are first binned, and then one bit vector is created for each bin. In contrast to other proposals for bitmap indexes, Maitri indexes blocks, rather than individual objects inside blocks. This invalidates optimizations developed to support fast query responses like range and interval encoding [3], which encode the bit vectors in the bitmap so that any query can be answered by referencing at most two bit vectors. On the other hand, block-level indexing means Maitri bitmaps are much shorter than would otherwise be the case.

Block-level indexing also complicates query processing, because a block satisfying a conjunctive condition $p_1 \wedge p_2$, may contain no single object that satisfies both p_1 and p_2 . Thus the block must be parsed, and individual objects examined, to determine whether any individual object satisfies $p_1 \wedge p_2$. We have recently been evaluating the overhead of this screening step using real-world data sets and queries supplied by the scientists who use those data sets. When the overhead is too high, we plan to have Maitri use multiple resolutions of blocks to provide multi-level indexing.

4 Experiments

Maitri is implemented in C++, with each manager as a separate independent library adhering to a standard API (with the exception that all other managers depend on the block manager library).

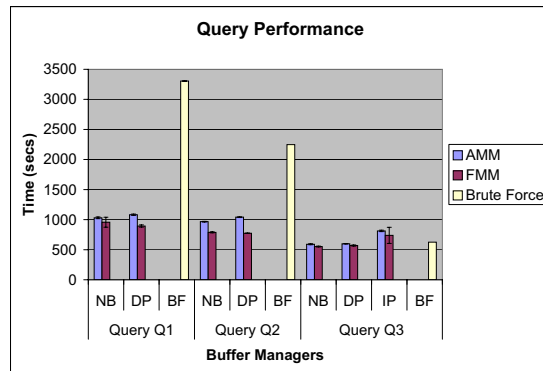


Figure 2. Run times for three queries on vegetation index data. NB = No buffer manager. DP = Delayed prefetch. IP = Immediate prefetch. BF = Brute force sequential search through all the blocks containing the queried attributes.

In this paper, we describe our preliminary performance results for the bitmap index manager, with and without the buffer manager described earlier. (For details regarding the performance of the buffer manager without an index manager, we refer the reader to a performance study of Godiva with visualization tools [8].) These experiments are a first step toward our long-term goals of demonstrating that the format independence afforded by Maitri can be achieved with acceptable performance, bitmap indexing works well for a wide variety of scientific applications, and Maitri's plug and play architecture has an acceptable impact on performance.

The experiments employed a data set used by UIUC scientists studying vegetation patterns in the United States. The data come from the NASA EOS Data Gateway (edcim-www.cr.usgs.gov/pub/imswelcome), by selecting the VI files collected on 8/2/2003 and 8/16/2003 over the Appalachians. The resulting 3.4 GB data set has a simple schema with 11 attributes, each of which is stored in a separate HDF dataset (a 2D array of 4800 * 4800 points) in one

of seven HDF files, with each file corresponding to a different geographical region. All attributes were stored as short integers.

The experiment evaluates the performance of three queries: (Q1) SELECT LOCATION WHERE NDVI IN [2000, 5000], EVI IN [5000, 7000], MIR Reflectance IN [1000, 5000], Sun Zenith Angle IN [1000, 2000]. (Q2) SELECT LOCATION WHERE NDVI IN [1000, 2000], EVI IN [1000, 5000], MIR Reflectance IN [1000, 5000], Sun Zenith Angle IN [1000, 2000]. (Q3) SELECT LOCATION WHERE NDVI IN [2000, 3000]. Maitri stored indexes for each of the where-clause attributes, using equi-width binning with 100 bins for each attribute. The selectivity of the queries, in terms of locations, is .03% for Q1, .03% for Q2, and 1.0% for Q3; and in terms of blocks containing a location that satisfies at least one of the query conditions, the selectivity with respect to each queried attribute is 8.84% for Q1, 11.02% for Q2, and 69.94% for Q3. Because the data's native format stores each attribute for a location separately, Maitri must read separate blocks for each of the indexed attributes in the query to find the locations that satisfy all of the query conditions. In other words, the queries implicitly involve a join across the attribute files. Four superblocks are involved in the queries (one for each attribute), and each block contains 1200 points of data for a single attribute.

To evaluate the interaction of the index manager with the buffering strategy, the experiments use three buffering strategies: no explicit buffering; buffering with immediate prefetch; and buffering with delayed prefetch. In all three cases, the query manager first invokes the index manager to obtain a list containing the IDs of every block whose objects satisfy one of the query conditions. With immediate prefetch, the buffer manager is asked to fetch all the blocks that the index manager listed, and Maitri checks the corresponding fetched blocks to find all locations that satisfy all of the query conditions. As soon as Maitri determines that a particular location does not satisfy

the second condition (say) of a query, it deletes the blocks containing the data for conditions three and four from the prefetch list. In the delayed prefetch policy, Maitri asks the buffer manager to fetch blocks from only the first two indexed attributes. If none of the locations in these blocks satisfy the conditions of the query, then the buffer manager is not asked to fetch the blocks containing the remaining attributes of these locations, because these locations cannot possibly satisfy the query. In the experiments with no buffer manager, the application code directly reads in blocks, using ordinary HDF calls; as with the delayed prefetch experiments, only the minimal number of blocks is read.

To evaluate the performance impact of different metadata management approaches, the experiments were performed both with an ASCII file for storing metadata (AMM) and a formula based system (FMM) in which the metadata was generated using a formula provided by the scientists. The AMM explicitly stores the file name, HDF dataset identifier, and region inside the HDF dataset for each Maitri block, while the FMM computes that information from the block identifier. The experiments were run on a 1.4 GHz Pentium 4 CPU with 512 MB RAM, with data on a local disk with up to 100 MB/sec transfer rate (quoted from the Maxtor web site).

Query performance results are summarized in figure 4. All reported numbers are the average of three warm runs, and error bars show the 95% confidence interval for the mean. CPU costs to buffer, parse, and extract data from blocks are the dominant factor in the query run times (and in the cold runs also, which are not shown in the figure). As expected, FMM is slightly faster than AMM, due to the time saved in reading and parsing the ASCII file to find the metadata, as opposed to quickly calculating the values. Moreover, the use of indexes allows us to retrieve only a fraction of the total data and hence speeds up the performance considerably, as can be seen by the speed up over the brute force algorithm of all except the

“Immediate Prefetch” buffer manager.

Interestingly, the number of blocks fetched is almost identical for the immediate and delayed prefetching schemes, which implies that the query processor is deleting blocks from the prefetch list before they have actually been brought in from disk. Instead, the poor performance of immediate prefetch is due to Godiva’s original data structures for the prefetch block lists, which require sequential search to find blocks in the list; we will change the data structure to support fast deletions and insertions. With immediate prefetch, Q1 and Q2 ran for a dozen hours, so we do not report those numbers in figure 4.

We also see that the scheme with no buffer manager is slightly faster than with delayed prefetching. This is because in the delayed prefetching scheme, there is only a short time between the point where it becomes known that the block must be fetched and the time that it absolutely must be fetched, in which case the buffer manager adds overhead and offers no benefit compared to directly calling HDF to read the block. (If a needed block had been accessed previously, or if the same block was accessed more than once, then the buffer manager might offer a performance benefit.)

We will follow up on these experiments by working to minimize Maitri’s CPU overhead before proceeding to a full evaluation of the performance of bitmap indexing on a wider range of scientific data and queries.

5 Conclusion and Future Work

The goal of Maitri is to provide scientists with a set of easy to use, lightweight, plug and play components (metadata manager, buffer manager, query manager, concurrency manager, and index manager) without requiring scientists to give up the data storage formats they are accustomed to and for which many supporting tools have already been developed. We have proposed the use of bitmap indexes as a viable option for indexing

data in a block oriented database like Maitri. Our preliminary performance results with the indexing and buffering subunits of Maitri system show that the performance of the system is acceptable, in spite of the extra overhead introduced by a loosely coupled architecture.

Maitri requires a great deal more development and evaluation before we can claim that it will effectively support a wide range of scientific applications. In the near future, we will be experimenting with additional data sets, developing the multilevel indexing scheme described earlier, a user interface for developing a formula-based metadata manager, binary-file-based metadata managers, upgrades to the buffer manager to support new data structures, and an exploration of the effect of bin size and block size on performance.

6 Acknowledgments

The research is funded by the U.S. Department of Energy through the center for Simulation of Advanced Rocks at UIUC under subcontract number B341494, and through the centre for Programming Models for Scalable Parallel Computing under DOE DEFC02-01ER25508, and by a NSF grant to PI Laxmikant V. Kale and co-PI Marianne Winslett.

References

- [1] A. Aiken, J. Chen, and M. Stonebraker et al. Tioga-2: A direct manipulation database visualization environment. In *ICDE*, 1996.
- [2] C. Bachman. The programmer as navigator. *Comm. ACM*, 16(11):653–658, 1973.
- [3] C. Chan and Y. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *SIGMOD Conference*, 1999.
- [4] A. Choudhary, M. Kandemir, and H. Nagesh et al. Data management for large-scale scientific computations in high performance distributed systems. In *International Symposium on High Performance Distributed Computing*, pages 263–272. IEEE Computer Society Press, 1999.
- [5] The Department of Energy Office of Science Data Management Challenge. <http://www.sc.doe.gov/ascr/Final-report-v26.pdf>, 2004.
- [6] R. Johnson, M. Goldner, and M. Lee et al. USD - a database management system for scientific research. In *SIGMOD Conference*, page 4, 1992.
- [7] P. Kumar, P. Bajcsy, and D. Tchong et al. Using D2K data mining platform for understanding the dynamic evolution of land-surface variables. In *Earth-Sun System Technology Conference*, 2005.
- [8] X. Ma, M. Winslett, and J. Norris et al. Godiva: Lightweight data management for scientific visualization applications. In *ICDE*, pages 732–744, 2004.
- [9] J. No, R. Thakur, and A. Chaudhary. Integrating parallel file I/O and database support for high-performance scientific data management. In *Proceedings of SC2000: High Performance Networking and Computing*. IEEE Computer Society Press, 2000.
- [10] B. Plale, J. Alameda, and B. Wilhelmson et al. Active management of scientific data. In *IEEE Internet Computing*, 2005.
- [11] M. Stonebraker and L. Rowe. The design of Postgres. In *SIGMOD Conference*, pages 340–355, 1986.
- [12] K. Wu, E. J. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *VLDB*, pages 24–35, 2004.