The Architecture of SciDB

Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman

Paradigm4, Inc.
186 Third Avenue
Waltham, MA 02451

Abstract. SciDB is an open-source analytical database oriented toward the data management needs of scientists. As such it mixes statistical and linear algebra operations with data management ones, using a natural nested multi-dimensional array data model. We have been working on the code for two years, most recently with the help of venture capital backing. Release 11.06 (June 2011) is downloadable from our website (SciDB.org).

This paper presents the main design decisions of SciDB. It focuses on our decisions concerning a high-level, SQL-like query language, the issues facing our query optimizer and executor and efficient storage management for arrays. The paper also discusses implementation of features not usually present in DBMSs, including version control, uncertainty and provenance.

Keywords: scientific data management, multi-dimensional array, statistics, linear algebra.

1 Introduction and Background

The Large Synoptic Survey Telescope (LSST) [1] is the next "big science" astronomy project, a telescope being erected in Chile, which will ultimately collect and manage some 100 Petabytes of raw and derived data. In October 2007, the members of the LSST data management team realized the scope of their data management problem, and that they were uncertain how to move forward. As a result, they organized the first Extremely Large Data Base (XLDB-1) conference at the Stanford National Accelerator Laboratory [2]. Present were many scientists from a variety of natural science disciplines as well as representatives from large web properties. All reported the following requirements:

Multi-petabyte amounts of data. In fact a recent scientist at a major university reported that 20 research groups at his university had more than a quarter of a petabyte each [3].

A preponderance of array data. Geospatial and temporal data such as satellite imagery, oceanographic data telescope data, telematics data and most simulation data all are naturally modeled as arrays. Genomics data generated from high throughput sequencing machines are also naturally represented as arrays.

Complex analytics. Traditional business intelligence has focused on simple SQL aggregates or windowing functions. In contrast, scientists need much more sophisticated capabilities. For example, satellite imagery can be reported at various resolutions and in different co-ordinate systems. As a result, earth scientists need to regrid such imagery in order to correlate the data from multiple satellites. In addition, most satellites cannot see through cloud cover. Hence, it is necessary to find the "best" cloud-free composite image from multiple passes of the satellite. These are representative of the complex operations required in this application area.

A requirement for open source code. Every scientist we have talked to is adamant about this requirement. Seemingly, the experience of the Large Hadron Collider (LHC) project [4] with one proprietary DBMS vendor has "poisoned the well". Hence, scientists require the option of fixing bugs and adding their own features, if the vendor of their chosen solution is unable, unwilling, or just slow to respond. In effect, only open source software is acceptable.

A requirement for no overwrite. Scientists are equally adamant about never throwing anything away. For example, large portions of the earth are currently not very interesting to earth scientists. However, that could change in the future, so discarding currently uninteresting data is not an option. Also, they wish to keep erroneous data that has been subsequently corrected. The reason for this is to redo analyses on the data as it existed at the time the original analysis was done, i.e. they want auditability for their analyses. This is related to the provenance discussion below, and requires that all data be kept indefinitely.

A requirement for provenance. If a data element looks suspicious, then scientists want to be able to trace backward through its derivation to find previous data that appears faulty. In other words, trace the error back to its source. Similarly, they would then want to find all of the derived data that came from this faulty item. In other words, they want the ability to do forward and backward derivation efficiently.

One reason for this requirement is assistance in the error correction noted above. A second reason is to facilitate sharing. Different scientists generally cannot make use of derived data unless they know the algorithm that was used to create it. For example, consider the "best" cloud free image discussed above. There is no universal way to choose the best composite image, and any scientist who wants to use a composite image must know what algorithm was used to construct it. They want to find this information by exploring the provenance of the data of interest.

A requirement for uncertainty. After all, every bit of scientific data comes with error bars. Current DBMSs were written to support the business market, and assume the data is perfect. Obviously, enterprises must know accurate salaries, in order to write pay checks. Essentially all information collected from sensors (nearly 100% of science data) does not have this property. Furthermore, scientists may want to propagate error data through a sequence of calculations.

A requirement for version control. There is no universal agreement on the cooking algorithms, which turn raw data into derived data sets. Hence, scientists would like to re-cook raw data for their study areas, retaining the conventional derivations for the rest of the data set. Although they can construct a complete copy of the data, with the required characteristics, it is wildly more efficient to delta their copies off of the conventional one, so the common data only appears once. Version control software has been supporting this functionality for years.

At XLDB-1, there was a general feeling that RDBMSs would never meet the above requirements because they have:

- The wrong data model,
- The wrong operators, and
- Are missing required capabilities.

Moreover, the RDBMS vendors appear not to be focused on the science market, because the business enterprise market is perceived to be larger. Hence, there was skepticism that these shortcomings would ever be addressed.

A second theme of the meeting was the increasing difficulty of meeting big science requirements with "from the bare metal up" custom implementations. The software stack is simply getting too large. Several of the web properties indicated the scope of their custom efforts, and said "we are glad we have sufficient resources to move forward". Also, there was frustration that every big science project re-grows the complete stack, leading to limited shared infrastructure. The Sloan Digital Sky Survey [5] was also noted as a clear exception, as they made use of SQLServer.

In effect, the community was envious of the RDBMS market where a common set of features is used by nearly everybody and supported by multiple vendors. In summary, the mood was "Why can't somebody do for science what RDBMS did for business?"

As a result, Dave Dewitt and Mike Stonebraker said they would try to build a from-the-ground-up DBMS aimed at science requirements. Following XLDB-1, there were meetings to discuss detailed requirements and a collection of use cases written, leading to an overall design. This process was helped along by the LSST data management team who said, "If it works, we will try to use it".

We began writing code in late 2008, with a pick-up team of volunteers and research personnel. This led to a demo of an early version of SciDB at VLDB in Lyon, France in Sept 2009 [6]. We obtained venture capital support for the project, and additional assistance from NSF in 2010. This has allowed us to accelerate our efforts. We have recently released SciDB 11.06 and are working toward a full featured high performance system in late 2011. We have been helped along the way by the subsequent annual XLDB meetings [7, 8, 9] where SciDB issues have been discussed in an open forum.

This paper reports on the SciDB design and indicates the status of the current system.

2 SciDB Design

In this section we present the major design decisions and our rationale for making them the way we did. We start with system assumptions in Section 2.1, followed by a data model discussion in Section 2.2. The query language is treated in Section 2.3.

The optimizer and storage management components are treated respectively in Section 2.4 and 2.5. Other features, such as extensibility, uncertainty, version control and provenance are discussed at appropriate times.

2.1 System Assumptions

It was pretty obvious that SciDB had to run on a grid (or cloud) of computers. A single node solution is clearly not going to make LSST happy. Also, there is universal acceptance of Linux in this community, so the OS choice is easy. Although we might have elected to code the system in Java, the feeling was that C++ was a better choice for high performance system software.

The only point of contention among the team was whether to adopt a shared-disk or a shared-nothing architecture. On the one hand, essentially all of the recent parallel DBMSs have adopted a shared nothing model, where each node talks to locally attached storage. The query optimizer runs portions of the query on local data. In essence, one adopts a “send the query to the data” model, and strives for maximum parallelism.

On the other hand, many of the recent supercomputers have used a shared-disk architecture. This appears to result from the premise that the science workload is computation intensive, and therefore the architecture should be CPU-focused rather than data focused. Also, scientists require a collection of common operations, such as matrix multiply, which are not “embarrassingly parallel”. Hence, they are not obviously faster on a shared-nothing architecture.

Since an important goal of SciDB is petabyte scalability the decision was made that SciDB would be a shared nothing engine.

2.2 Data Model

It was clear that we should select an array data model (rather than a table one) as arrays are the natural data object for much of the sciences. Furthermore, early performance benchmarks on LSST data [10] indicated that SciDB is about 2 orders of magnitude faster than an RDBMS on a typical science workload. Finally, most of the complex analytics that the science community uses are based on core linear algebra operations (e.g. matrix multiply, covariance, inverse, best-fit linear equation solution). These are all array operations, and a table model would require a conversion back and forth to arrays. As such, it makes sense to use arrays directly.

Hence, SciDB allows any number of **dimensions** for an array. These can be traditional integer dimensions, with any starting and ending points or they can be unbounded in either direction. Moreover, many arrays are more natural with non-integer dimensions. For example, areas of the sky are naturally expressed in polar co-ordinates in some astronomy projects. Hence, dimensions can be any user-defined data type using a mechanism we presently describe.

Each combination of dimension values defines a **cell** of an array, which can hold an arbitrary number of **attributes** of any user-defined data type. Arrays are **uniform** in that all cells in a given array have the same collection of values. The only real decision was whether to allow a nested array data model or a flat one. Many use cases, including LSST, require nested arrays, so the extra complexity was deemed well worth it. Also, nested arrays support a mechanism for hierarchical decomposition of cells, so that systematic refinement of specific areas of an array can be supported, a feature often cited as useful in HDF5 [11].

Hence, an example array specification in SciDB is:

```
CREATE ARRAY example <M: int, N: float> [I=1:1000, J=1000:20000]
```

Here, we see an array with attributes M and N along with dimensions I and J.

2.3 Query Language

SciDB supports both a functional and a SQL-like query language. The functional language is called AFL for array functional language; the SQL-like language is called AQL for array query language. AQL is compiled into AFL.

AFL, the functional language includes a collection of operations, such as filter and join, which a user can cascade to obtain his desired result. For example, if A and B are arrays with dimensions I and J, and c is an attribute of A, then the following utterance would be legal:

```
temp = filter (A, c = value)
result = join (B, temp: I, J)
```

Or the composite expression: `result = join (B, filter (A, c = value), I, J)`

Such a language is reminiscent of APL [12] and other functional languages and array languages [13, 14].

For commercial customers more comfortable with SQL, SciDB has created an array query language, AQL, which looks as much like SQL as possible. Hence, the above example is expressed as:

```
select *
from A, B
where A.I = B.I and A.J = B.J and A.c = value
```

We have had considerable discussion concerning two aspects of the semantics of AQL, namely joins and non-integer dimensions, and we turn to these topics at this time.

Consider the arrays A and B from above, and suppose A has attributes c and d, while B has attributes e and f. The above example illustrated a dimension join, i.e., one where the dimensions indexes must be equal. The result of this operation has dimensions I and J, and attributes c, d, e and f. In essence this is the array version of a relational natural join. It is also straightforward to define joins that match less than all dimensions.

Non equi-dimensional joins are also reasonably straightforward. For example the following join result must be defined as a three dimensional array, I (from A), I (from B) and J.

```
select *
from A, B
where A.I > B.I and A.J = B.J and A.c = value
```

The problem arises when we attempt to define attribute joins, e.g.,

```
select *
from A, B
where A.c = B.e and A.d = B. f
```

In effect, we want to join two arrays on attribute values rather than dimensions. This must be defined as a four dimensional result: I (from A), J (from A), I (from B), and J (from B).

To understand array joins, it is useful to think of an array as having a relational representation, where the dimensions are columns as are the cell values. Then any array join can be defined as a relational join on the two argument tables. This naturally defines the semantics of an array join operation, and SciDB must produce this answer. In effect, we can appeal to relational semantics to define array joins.

A second semantic issue is an offshoot of the first one. It is straightforward to change attribute values in AQL with an update command. For example, the following command increments a value for a specific cell:

```
update A set (d = d+1)
where A.I = value and A.J = value
```

Obviously, it must be possible to manipulate SciDB dimensions, and an update command is not the right vehicle. Hence, SciDB has included a new powerful command, **transform**, to change dimension values. The use cases for transform include:

- Bulk changes to dimensions, e.g. push all dimension values up one to make a slot for new data,
- Reshape an array; for example change it from 100 by 100 to 1000 by 10,
- Flip dimensions for attributes, e.g. replace dimension I in array A with a dimension made up from d.
- Transform one or more dimension, for example change I and J into polar co-ordinates.

Transform can also map multiple dimension or attribute values to the same new dimension value. In this case, transform allows an optional aggregation function to combine the multiple values into a single one for storage. In the interest of brevity we skip a detailed discussion of the transform command and the interested reader is referred to the online documentation on SciDB.org, for a description of this command.

Non-integer dimensions are supported by an index that maps the dimension values into integers. Hence, an array with non-integer dimensions is stored as an integer array mapping index.

2.4 Extensibility

It is well understood that a DBMS should not export data to an external computation (i.e. move the data to the computation), but rather have the code execute inside the DBMS (move the computation to the data). The latter has been shown to be wildly faster, and is supported by most modern day relational DBMSs. The norm is to use the extension constructs pioneered by Postgres more than 20 years ago [15].

Since science users often have their own analysis algorithms (for example examining a collection of satellite passes to construct the best cloud-free composite image) and unique data types (e.g. 7 bit sensor values), it is imperative to support user-defined extensibility. There are four mechanisms in SciDB to support user extensions.

First, SciDB supports **user-defined data types**. These are similar to Postgres user defined types as they specify a storage length for a container to hold an object of the given type. User-defined types allow a (sophisticated) user to extend the basic SciDB data types of integer, float, and string. Hence, the attribute values in a SciDB cell can be user-defined.

Second, a user must be able to perform operations on new data types. For example, a user could define arbitrary precision float as a new data type and then would want to define operations like addition and subtraction on this type. **User-defined functions** are the mechanism for specifying such features. These are scalar functions that accept one or more arguments of various data types and produce a result of some data type. Again, the specification is similar to Postgres, and right now such functions must be written in C++.

Third, SciDB supports **user-defined aggregates**, so that conventional aggregates can be written for user-defined types. As well, science-specific aggregates can be written for built-in or user-defined data types. An aggregate requires four functions, along the lines of Postgres [16]. Three of the functions are the standard Init (), Increment (), and Final () that are required for any single node user-defined aggregate calculation. Since SciDB is a multi-node system, these three functions will be run for the data at each node. Subsequently, a rollup () must be specified to pull the various partial aggregates together into the final answer.

The last extension mechanism in SciDB is **user-defined array operators**. These functions accept one or more arrays as arguments and usually produce an array as an answer. Although Join is a typical example, the real use case is to support linear algebra operations, such as matrix multiply, curve fitting, linear regression, equations solving and the like. Also in this category are data clustering codes and other machine learning algorithms.

There are two wrinkles to array functions that are not present in standard Postgres table functions. As will be discussed in Section 2.6 SciDB decomposes storage into multi-dimensional chunks, which may overlap. Some array functions are **embarrassingly parallel**, i.e. they can be processed in parallel on a collection of computing nodes, with each node performing the same calculation on its data. However, some array functions can only be run in parallel if chunks overlap by a minimum amount, as discussed in more detail in Section 2.6. Hence, a user-defined array function must specify the minimum overlap for parallel operation.

Second, many array operations are actually algorithms consisting of several steps, with conditional logic between the steps. For example, most algorithms to compute the inverse of a matrix proceed by iterating a core calculations several times. More complex operations may perform several different kinds of core operations, interspersed with conditional logic. Such logic may depend on the size or composition of intermediate results (e.g. an array column being empty). As such, a user-defined array operation must be able to run other operations, test the composition of intermediate results and control its own parallelism. To accomplish this objective, SciDB has a system interface that supports these kinds of tasks.

It should be noted that writing user-defined array operations is not for the faint of heart. We expect experts in the various science disciplines to write libraries to our interface that other scientists can easily use in AQL, without understanding their detailed composition. This is similar to ScaLAPACK [17], which was written by rocket scientists and widely used by mere mortals.

2.5 Query Processing

Users specify queries and updates in AQL, and the job of the optimizer and executor is to correctly solve such queries. We have several guiding principles in the design of this component of SciDB.

First, we expect a common environment for SciDB is to run on a substantial number of nodes. As such, SciDB must scale to large configurations. Also, many science applications are CPU intensive. Hence, the user-defined functions that perform the complex analytics usually found in this class of problems are often CPU bound. Also, many are not “embarrassingly parallel”, and entail moving substantial amounts of data if one is not careful. Thus, the three guiding principles of query processing in SciDB are:

Tenet 1: aim for parallelism in all operations with as little data movement as possible.

This goal drives much of the design of the storage manager discussed in Section 2.6. Also, if an operation cannot be run in parallel because the data is poorly distributed, then SciDB will redistribute the data to enable parallelism. Hence, SciDB is fundamentally focused on providing the best response time possible for AQL utterances.

Second, the optimizers in relational DBMSs often choose poor query plans because their cost functions entail predicting the size of intermediate results. If a query has three or four cascading intermediate results, then these size estimates become wildly inaccurate, resulting in a potentially poor choice of the best query plan. Because SciDB queries are expected to be complex, it is imperative to choose a good plan.

To accomplish this goal, the SciDB optimizer processes the query parse tree in two stages. First, it examines the tree for operations that **commute**. This is a common optimization in relational DBMSs, as filters and joins are all commutative. The first step in the SciDB optimizer is to push the cheaper commuting operation down the tree. In our world, we expect many user defined array operations will not commute. For example, re-gridding a satellite imagery data set will rarely, if ever, commute with operations above or below it in the tree. Hence, this tactic may be less valuable than in a relational world.

The next step is to examine the tree looking for **blocking** operations. A blocking operation is one that either requires a redistribution of data in order to execute, or cannot be pipelined from the previous operation, in other words it requires a temporary array to be constructed. Note that the collection of blocking operations separates a query tree into sub-trees.

Tenet 2: Incremental optimizers have more accurate size information and can use this to construct better query plans.

The SciDB optimizer is **incremental**, in that it picks the best choice for the first sub-tree to execute. After execution of this sub-tree, SciDB has a perfect estimate for the size of the result, and can use this information when it picks the next sub-tree for execution.

Of course, the downside is that SciDB has a run-time optimizer. Such run-time overhead could not be tolerated in an OLTP world; however, most scientific queries run long enough that optimizer overhead is insignificant.

Tenet 3: Use a cost-based optimizer.

This third principle is to perform simple cost-based plan evaluation. Since SciDB only plans sub-trees, the cost of exhaustive evaluation of the options is not onerous.

Right now the optimizer is somewhat primitive, and focuses on minimizing data movement and maximizing the number of cores that can be put to work, according to tenet 1.

In summary, the optimizer/execution framework is the following algorithm:

```
Until no more {
        Choose and optimize next sub-plan
        Reshuffle data, if required
        Execute a sub-plan in parallel on a collection of local nodes
        Collect size information from each local node
}
```

2.6 Storage of Arrays

Basic Chunking

It is apparent that SciDB should **chunk** arrays to storage blocks using some (or even all) of the dimensions. In other words, a **stride** is defined in some or all of the dimensions, and the next storage block contains the next stride in the indicated dimensions. Multiple dimension chunking was explored long ago in [18] and has been shown to work well. Equally obviously, SciDB should chunk arrays across the nodes of a grid, as well as locally in storage. Hence, we distribute chunks to nodes using hashing, range partitioning, or a block-cyclic algorithm.

In addition, chunks should be large enough to serve as the unit of I/O between the buffer pool and disk. However, CPU time can often be economized by splitting a chunk internally into **tiles** as noted in [19]. In this way, subset queries may be able to examine only a portion of a chunk, and economize total time. Hence, we support a two level chunk/tile scheme.

One of the bread-and-butter operations in LSST is to examine raw imagery looking for interesting celestial objects (for example, stars). Effectively this is a data clustering problem; one is looking for areas of imagery with large sensor amplitude. In other areas of science, nearest neighbor clustering is also a very popular operation.

For example, looking for islands in oceanographic data or regions of snow cover in satellite imagery entails exactly the same kind of clustering.

To facilitate such neighborhood queries, SciDB contains two features. First, chunks in SciDB can be specified to **overlap** by a specific amount in each of several dimensions. This overlap should be the size of the largest feature that will be searched for. In this way, parallel feature extraction can occur without requiring any data movement. As a result, unlike parallel RDBMSs, which use non-overlapping partitions, SciDB supports the more general case.

At array creation time, stride and overlap information must be specified in the create array command. Hopefully, overlap is specified to be the largest size required by any array function that does feature extraction. Also, every user-defined array operation specifies the amount of overlap it requires to be able to perform parallel execution. If insufficient overlap is present, then SciDB will reshuffle the data to generate the required overlap.

Fixed or Variable Size Chunks

A crucial decision for SciDB was the choice of fixed or variable size chunks. One option is to fix the size of the stride in each dimension, thereby creating logically fixed size chunks. Of course, the amount of data in each chunk can vary widely because of data skew and differences in compressibility. In other words, the first option is fixed logical size but variable physical size chunks.

The second option is to support variable logical size chunks. In this case, one fills a chunk to a fixed-size capacity, and then closes it, thereby a chunk encompasses a variable amount of logical array real estate.

Variable chunk schemes would require an R-tree or other indexing scheme to keep track of chunk definitions. However, chunks would be a fixed physical size, thereby enabling a simple fixed size main memory buffer pool of chunks. On the other hand, fixed size logical chunks allow a simple addressing scheme to find their containers; however, we must cope with variable size containers.

We have been guided by [19] in deciding what to do. The "high level bit" concerns join processing. If SciDB joins two arrays, with the same fixed size chunking, then they can be efficiently processed in pairs, with what amounts to a generalization of merge-sort. If the chunking of the two arrays is different, then performance is much worse, because each chunk in the first array may join to several chunks in the second array. If chunking is different, then the best strategy may be to rechunk one array to match the other one, a costly operation as noted in [19].

This argues for fixed chunking, since frequently joined arrays can be identically chunked. That will never be the case with variable chunking. Hence, SciDB uses fixed logical size chunks. Right now, the user executing the Create Array command specifies the size of these chunks. Obviously, a good choice makes a huge difference in performance.

In summary, chunks are fixed (logical) size, and variable physical size. Each is stored in a container (file) on disk that can be efficiently addressed. The size of a chunk should average megabytes, so that the cost of seeks is masked by the amount of data returned.

There are several extensions to the above scheme that are required for good performance. These result from our implementation of versions, our desire to perform skew management, and our approach to compression. These topics are addressed in the next three sections.

Version Control

There are three problems which SciDB solves using version management. First, there is a lot of scientific data that is naturally temporal. LSST, for example, aims its telescope at the same portion of the sky repeatedly, thereby generating a time series. Having special support for temporal data seems like a good idea.

Second, scientists **never** want to throw old data away. Even when the old data is wrong and must be corrected, a new value is written and the old one is retained. Hence, SciDB must be able to keep everything.

The third problem deals with the "cooking" of raw data into derived information. In LSST, raw data is telescope imagery, and feature extraction is used to identify stars and other celestial objects, which constitute derived data. However, there is no universal feature extraction algorithm; different ones are used by different astronomers for different purposes. As such, LSST supports a "base line" cooking process, and individual astronomers can recook portions of the sky that they are interested in. Hence, astronomers want the base line derived information for the whole sky, except for the portions they have recooked. Such versioning of data should be efficiently supported.

To support the no overwrite model, all SciDB arrays are versioned. Data is loaded into the array at the time indicated in the loading process. Subsequent updates, inserts or bulk loads add new data at the time they are run, without discarding the previous information. As such, new information is written at the time it becomes valid. Hence, for a given cell, a query can scan particular versions referenced by timestamp or version number.

We now turn to the question: "How are array versions stored efficiently?" As updates or inserts occur to a chunk, we have elected to keep the most up-to-date version of the chunk stored contiguously. Then, previous versions of the chunk are available as a chain of "deltas" referenced from the base chunk. In other words, we store a given chunk as a base plus a chain of "b**ackwards deltas"**. The rationale is that users usually want the most current version of a chunk, and retrieval of this version should be optimized. The physical organization of each chunk contains a reserved area, for example 20% additional space, to maintain the delta chain.

Arrays suffixed with a timestamp can be used in scan queries. Since we expect queries of the state of the array at a specific time to be very popular, we allow the select arrayname@T shorthand popularized in Postgres. If no specification is made, the system defaults to select arrayname@now.

We turn briefly to support for **named versions**. A user can request a named version to be defined relative to a given materialized array at time T. At this point, no storage is allocated, and the time T is noted in the system catalogs. As updates to the named version are performed, new containers for stored chunks are allocated and updates recorded in the new chunks. Multiple updates are backwards chained, just like in normal arrays. Over time, a **branch** is constructed, which is maintained as a chain of deltas based on the base array at time T. Clearly a tree of such versions is possible.

Query processing must start with the named version looking for data relevant to a given query. If no object exists in the version, its parent must be explored, ultimately leading back to the stored array from which the version was derived. This architecture looks much like configuration management systems, which implement similar functionality. A more elaborate version management solution is described in [20], and we may incorporate elements of this system into SciDB in the future.

Skew Management

Data in SciDB arrays may be extremely skewed for two reasons. As noted above, update traffic may be skewed. In addition, the density of non-null data may also be skewed. For example, consider a population database with geographic co-ordinates. The population density of New York City is somewhere around 1000000 times that of Montana.

There are two skew issues which we discuss in this section: what to do with chunks that have too little data, and what to do with chunks that have too much data.

Decades of system design experience dictates that it is advantageous to move data from disk to main memory in fixed size blocks (pages) versus variable size blocks (segments). The universal consensus was that fixed size blocks were easier to manage and performed better. Hence, SciDB has a fixed-size block model, where the main memory buffer pool is composed of a collection of fixed size slots containing "worthy" fixed size disk blocks. As noted above, this block size must be at least several megabytes.

If the user specifies a chunk size that results in a chunk containing more than B bytes, then the chunk must be split. We cycle through the chunking dimensions, splitting each in turn. As such, actual chunks will be some binary tree refinement of the user-specified chunk size. Unlike [19] which reports experiments on two chunk sizes, SciDB supports an arbitrary number of splits to keep the chunk size below B.

If a chunk is too small, because it is sparsely populated with data, then it can accommodate many updates before it fills. In the meantime, it can be co-located in a disk block of size B with neighboring sparse chunks. The storage manager current performs this "bin packing".

Compression

All arrays are aggressively compressed on a chunk-by-chunk basis. Sparse arrays can be stored as a list of non-null values with their dimension indexes, followed by prefix encoding. Additionally, value encoding of many data types is also profitable. This can include delta encoding, run-length encoding, subtracting off an average value, and LZ encoding. The idea is that the compression system will examine a chunk, and then choose the appropriate compression scheme on a chunk-by-chunk basis.

In addition, if the chunk is subject to intensive update or to small geographic queries, then it will spend much overhead decompressing and recompressing chunks to process either modest queries or updates. In this case, it makes sense to divide a chunk into **tiles**, and compress each tile independently. In this way, only relevant tiles need to be decompressed and recompressed to support these kinds of queries and updates. Hence, tiling will result in better performance on workloads with many small updates and/or small geographic queries. On a chunk-by-chunk basis, the compression system can optionally elect to tile the chunk.

Also, we have noted that some SciDB environments are CPU limited, and compressing and decompressing chunks or tiles is the “high pole in the tent”. In this case, SciDB should switch to a lighter weight compression scheme.

The compression system is inside the storage manager and receives a new chunk or tile to encode. After encoding, the result is obviously variable sized, so the compression engine controls the splitting of chunks described above as well as the packing of small chunks into storage blocks mentioned in the previous section.

Uncertainty

Essentially all science data is uncertain. After numerous conversations with scientists, they pretty much all say:

Build in the common use case (normal distributions) to handle 80% of my data automatically.

My other 20% is specific to my domain of interest, and I am willing to write error analysis code in my application to handle this.

As such we have implemented both uncertain and precise versions of all of the common data types. Operating on precise data gives a precise answer; operating on uncertain data yields an uncertain answer. The uncertain versions of SciDB operations “do the right thing” and carry along errors in the internal calculations being performed. Moreover, a challenge to the compression system is to be smart about uncertainty. Specifically, most uncertain values in a chunk will have the same or similar error information. Hence, uncertainty information can be aggressively compressed

Notice that SciDB supports uncertain cell values but not uncertain dimensions. That functionality would require us to support approximate joins, which is a future extension.

Provenance

A key requirement for most science data is support for provenance. The common use case is the ability to point at a data value or a collection of values and say “show me the derivation of this data”. In other words, the data looks wrong, and the scientist needs to trace backwards to find the actual source of the error. Once, the source has been identified, it should be fixed, of course using the no-overwrite processing model. Then, the scientist wants to trace forward to find all data values that are derived from the incorrect one, so they can also be repaired.

In other words, SciDB must support the ability to trace both backward and forward. Some systems support coarse provenance (for example at the array level) that allow this functionality only for arrays, not cells. Since SciDB expects some very big arrays, this granularity is unacceptable. Other systems, e.g. Trio [21] store provenance by associating with each output value the identifier of all input values that contributed to the calculation. This approach will cause the data volumes to explode. For example, matrix multiply generates a cell from all the values in a particular source row and source column. If an array is of size M, then the provenance for matrix multiply will be of size M **3. This is obviously not an engineering solution.

Our solution is to allow database administrators to specify the amount of space they are willing to allocate for provenance data. The SciDB provenance system chooses how to best utilize this space, by varying the granularity of provenance information, on a command-by-command basis. The details of this system are discussed in [22].

Discussions with LSST personnel indicate a willingness to accept approximate provenance, if that can result in space savings or run time efficiency. For example, many LSST operations are "region constrained", i.e. the cell value that results from an operation comes from a constrained region in the input array. If true, approximate provenance can be supported by just recording the centroid of this region and its size. Often, the centroid is easily specified by a specific mapping from input to output, thereby further reducing the amount of provenance information that must be kept. The details of our approximate provenance are also discussed in [22].

In-situ Data

Most of the scientists we have talked to requested support for in-situ data. In this way, they can use some of SciDB's capabilities without having to go through the effort of loading their data. This would be appropriate for data sets that are not repeatedly accessed, and hence not worth the effort to load.

We are currently designing an interface (wrapper) that will allow SciDB to access data in other formats than SciDB natively understands. The details of how to do this as well as how to make the optimizer understand foreign data are still being worked out.

3 Summary, Status Performance, and Related Work

3.1 Related Work

SciDB is a commercial, open-source analytical database oriented toward scientific applications. As such, it differs from RDBMSs, which must simulate arrays on top of a table data model. The performance loss in such a simulation layer may be extreme [6]. The loss of performance in linear algebra operations may be especially daunting [23]. Also, most RDBMSs have trouble with complex analytics, because they are expressed on arrays, not tables. SciDB implements such operations directly, whereas RDBMSs, such as GreenPlum and Netezza, must convert a table to an array inside user-defined functions, then run the analytic code, and convert the answer back to a table to continue processing. Such out-and-back conversion costs do not need to be paid by SciDB. A similar comment can be made about interfaces between R [24] and RDBMSs. In addition, RDBMSs do not support multi-dimensional chunked storage, overlapping chunks, uncertainty, versions or provenance.

MonetDB [25] has an array layer [26], implemented on top of its column store table system. All of the comments in the previous paragraph apply to it. Similarly RasDaMan [27] is an array DBMS. However, it is implemented as an application layer that used Postgres for blob storage. As such, it is implementing multi-dimension chunking in an application layer external to the DBMS. It also lacks overlapping chunks, version control, uncertainty and provenance.

There are a myriad of statistical packages, including R [24], S [28], SAS [29], ScaLAPACK [17], and SPSS [30]. All of these perform complex analytics, often on a single node only, but perform no data management. SciDB is an integrated system to provide both data management and complex analytics.

Status and Summary

At the time of the SSDBM conference, SciDB version 11.06 will be available for download. SciDB development is backed by the commercial company Paradigm4 who will provide support as well as offer extensions for the commercial marketplace (monitoring tools, proprietary function libraries, etc.)

Development is proceeding with a global team of contributors across many time zones. Some are volunteers but at this early stage, most are employees of Paradigm4, including the engineering manager and chief architect. QA is being performed by volunteers in India and California. User-defined extensions are underway in Illinois, Massachusetts, Russia, and California.

References

1. `http://arxiv.org/abs/0805.2366`
2. Becla, J., Lim, K.-T.: Report from the First Workshop on Extremely Large Databases. Data Science Journal 7 (2008)
3. Szalay, A.: Private communication
4. Branco, M., Cameron, D., Gaidioz, B., Garonne, V., Koblitz, B., Lassnig, M., Rocha, R., Salgado, P., Wenaus, T.: Managing ATLAS data on a petabyte-scale with DQ2. Journal of Physics: Conference Series 119 (2008)
5. Szalay, A.: The Sloan Digital Sky Survey and Beyond. In: SIGMOD Record (June 2008)
6. Cudre-Mauroux, P., et al.: A Demonstration of SciDB: a Science-oriented DBMS. VLDB 2(2), 1534–1537 (2009)
7. Becla, J., Lim, K.-T.: Report from the Second Workshop on Extremely Large Databases, `http://www-conf.slac.stanford.edu/xldb08/`, `http://www.jstage.jst.go.jp/article/dsj/7/0/1/_pdf`
8. Becla, J., Lim, K.-T.: Report from the Third Workshop on Extremely Large Databases, `http://www-conf.slac.stanford.edu/xldb09/`
9. Becla, J., Lim, K.-T.: Report from the Fourth Workshop on Extremely Large Databases, `http://www-conf.slac.stanford.edu/xldb10/`
10. Cudre-Maroux, P., et al.: SS-DB: A Standard Science DBMS Benchmark (submitted for publication)
11. `http://www.hdfgroup.org/HDF5/`
12. `http://en.wikipedia.org/wiki/APLprogramming_language`
13. `http://en.wikipedia.org/wiki/Functional_programming`
14. `http://kx.com/`
15. Stonebraker, M., Rowe, L.A., Hirohama, M.: The Implementation of POSTGRES. IEEE Transactions on Knowledge and Data Engineering 2(1), 125–142 (1990)
16. `http://developer.postgresql.org/docs/postgres/xaggr.html`
17. `http://www.netlib.org/ScaLAPACK/`

18. Sarawagi, S., Stonebraker, M.: Efficient organization of large multidimensional arrays. In: ICDE, pp. 328–336 (1994), http://citeseer.ist.psu.edu/article/sarawagi94efficient.html
19. Soroush, E., et al.: ArrayStore: A Storage Manager for Complex Parallel Array Processing. In: Proc. 2011 SIGMOD Conference (2011)
20. Seering, A., et al.: Efficient Versioning for Scientific Arrays (submitted for publication)
21. Mutsuzaki, M., Theobald, M., de Keijzer, A., Widom, J., Agrawal, P., Benjelloun, O., Das Sarma, A., Murthy, R., Sugihara, T.: Trio-One: Layering Uncertainty and Lineage on a Conventional DBMS. In: Proceedings of the 2007 CIDR Conference, Asilomar, CA (January 2007)
22. Wu, E., et al.: The SciDB Provenance System (in preparation)
23. Cohen, J., et al.: Mad Skills: New Analysis Practices for Big Data. In: Proc. 2009 VLDB Conference
24. http://www.r-project.org/
25. http://monetdb.cwi.nl/
26. van Ballegooij, A., Cornacchia, R., de Vries, A.P., Kersten, M.L.: Distribution Rules for Array Database Queries. In: Andersen, K.V., Debenham, J., Wagner, R. (eds.) DEXA 2005. LNCS, vol. 3588, pp. 55–64. Springer, Heidelberg (2005)