Requirements for Science Data Bases and SciDB

Michael Stonebraker, MIT
Jacek Becla, SLAC
David Dewitt, Microsoft
Kian-Tat Lim, SLAC
David Maier, Portland State University
Oliver Ratzesberger, eBay, Inc.
Stan Zdonik, Brown University

***Abstract*:** For the past year, we have been assembling requirements from a collection of scientific data base users from astronomy, particle physics, fusion, remote sensing, oceanography, and biology. The intent has been to specify a common set of requirements for a new science data base system, which we call SciDB. In addition, we have discovered that very complex business analytics share most of the same requirements as "big science".

We have also constructed a partnership of companies to fund the development of SciDB, including eBay, the Large Synoptic Survey Telescope (LSST), Microsoft, the Stanford Linear Accelerator Center (SLAC) and Vertica. Lastly, we have identified two "lighthouse customers" (LSST and eBay) who will run the initial system, once it is constructed.

In this paper, we report on the requirements we have identified and briefly sketch some of the SciDB design.

I INTRODUCTION

XLDB-1 in October 2007 brought together a collection of "big science" and commercial Internet users with extreme data base requirements. Also present were a collection of vendors and David DeWitt and Michael Stonebraker. The users complained about the inadequacy of current commercial DBMS offerings. DeWitt and Stonebraker countered with the fact that various researchers in the DBMS community have been working on science databases for years and have even built prototypes (e.g. Sequoia 2000 with Postgres [1], Paradise [2], the Sloan Digital Sky Survey [3], and extensions to MonetDB [4]). Moreover, they also said "if you can define a common set of requirements across several science disciplines, then we will try to build it."

The result was a meeting at Asilomar in March 2008 between a collection of science users and a collection of DBMS researchers to define requirements, followed by a more detailed design exercise over the summer. Additional use cases were solicited, and parallel fund raising was carried out.

This paper presents the results of this requirements exercise in Section 2 and sketches some of the SciDB design. Intermingled are a collection of research topics that require attention. It concludes with summary of the state of the project in Section 3.

II REQUIREMENTS

These requirements come from particle physics (the LHC project at CERN, the BaBar project at SLAC and Fermilab), biology and remote sensing applications (Pacific Northwest National Laboratory), remote sensing (University of California at Santa Barbara), astronomy (Large Synoptic Survey Telescope), oceanography (Oregon Health & Science University and the Monterey Bay Aquarium Research Institute), and eBay.

There is a general realization in these communities that the past practice (build custom software for each new project from the bare metal on up) will not work in the future. The software stack is getting too big, too hard to build and too hard to maintain. Hence, the community seems willing to get behind a single project in the DBMS area. They also realize that science DBMSs are a "zero billion dollar" industry. Hence, getting the attention of the large commercial vendors is simply not going to occur.

2.1 Data Model

While most scientific users can use relational tables and have been forced to do so by current systems, we can find only a few users for whom tables are a natural data model that closely matches their data. Few are satisfied with SQL as the interface language. Although the Sloan Digital Sky Survey has been very successful in the astronomy area, they had perhaps the world's best support engineer (Jim Gray) helping them. Also, a follow-on project, PanSTARRS, is actively engaged in extending the system to meet their needs [5].

The Sequoia 2000 project realized in the mid 1990s that their users wanted an array data model, and that simulating arrays on top of tables was difficult and resulted in poor performance. A similar conclusion was reached in the ASAP prototype [6] which found that the performance penalty of simulating arrays on top of tables was around two orders of magnitude. It appears that arrays are a natural data model for a significant subset of science users (specifically astronomy, oceanography, fusion and remote sensing).

Moreover, a table with a primary key is merely a one-dimensional array. Hence, an array data model can subsume the needs of users who are happy with tables.

Seemingly, biology and genomics users want graphs and sequences. They will be happy with neither a table nor an array data model. Chemistry users are in the same situation.

Lastly, users with solid modelling applications want a mesh data model [7] and will be unhappy with tables or arrays. The net result is that "one size will not fit all", and science users will need a mix of specialized DBMSs.

Our project is exploring an array data model, primarily because it makes a considerable subset of the community happy and is easier to build than a mesh model. We support a multi-dimensional, nested array model with array cells containing records, which in turn can contain components that are multi-dimensional arrays.

Specifically, arrays can have any number of *dimensions*, which may be named. Each dimension has contiguous integer values between 1 and N (the *high-water-mark***)**. Each combination of dimension values defines a *cell*. Every cell has the same data type(s) for its value(s), which is one or more scalar values, and/or ones or more arrays. An array A with dimensions I and J and values x and y would be addressed as:

A[7, 8] — indicates the contents of the $(7, 8)^{th}$ cell
A[I = 7, J = 8] — more verbose notation for the $(7, 8)^{th}$ cell
A[7, 8].x — indicates the x value of the contents of the $(7, 8)^{th}$ cell

Like SQL an array can be *defined*, and then multiple instances can be *created***.** The basic syntax for defining an array is:

define ArrayType ({name =Type-1}) ({dname})

The dimensions of the array, which must be integer-valued, are inside the second (…). The value(s) of the array are inside the first (…). Each value has a name and a data type, which can be either an array or a scalar.

For example, consider a 2-D remote sensing array with each element consisting of 3 different types of sensors, each of which generates a floating-point value.

This can be defined by specifying:

define Remote (s1 = float, s2 = float, s3 = float) (I, J)

A physical array can be created by specifying the high water marks in each dimension. For example, to create Remote as an array of size 1024 by 1024 one would use the statement

create My_remote **as** Remote [1024,1024]

It is acceptable to create a basic array that is *unbounded* in one or more dimensions, for example

create My_remote_2 **as** Remote [*, *]

Unbounded arrays can grow without restriction in dimensions with a * as the specified upper bound.

Enhanced arrays, to be presently described, will allow basic arrays to be scaled, translated, have irregular (ragged) boundaries, and have non-integer dimensions. To discuss enhanced arrays, we must first describe user-defined functions.

SciDB will also support POSTGRES-style user-defined functions (methods, UDFs), which must be coded in C++, which is the implementation language of SciDB. As in POSTGRES, UDFs can internally run queries and call other UDFs. We will also support user-defined aggregates, again POSTGRES-style.

UDFs can be defined by specifying the function name, its input and output signatures and the code to execute the function. For example, a function, Scale10, to multiply the dimensions of an array by 10 would be specified as:

Define function Scale10 (integer I, integer J)
returns (integer K, integer L)
file_handle

The indicated file_handle would contain object code for the required function. SciDB will link the required function into its address space and call it as needed.

UDFs can be used to *enhance* arrays, a topic to which we now turn. Any function that accepts integer arguments can be applied to the dimensions of an array to enhance the array by transposition, scaling, translation, and other co-ordinate transformations.

For example, applying Scale10 to a basic array would be interpreted as applying the function to the dimension values of each cell to produce an enhanced array. Hence:

Enhance My_remote **with** Scale10

has the intended effect. In this case the (I, J) co-ordinate system of the basic array, My_remote, continues to work. In addition the (K, L) co-ordinate system from Scale10 also works. To distinguish the two systems, SciDB uses [..] to address basic dimensions and { …} to address enhanced ones. Hence, the basic ones are addressed:

A [7, 8] or A[I = 7, J = 8]

while the enhanced ones are addressed as:

A{20, 50} or A {K = 20, L = 50}.

Some arrays are *irregular*, i.e. they do not have integer dimensions, and some are defined in a particular co-ordinate system, for example Mercator geometry. Enhancing arrays with more complex UDFs can deal with both situations.

Consider a UDF that accepts a vector of integers and outputs a value for each dimension of some data type, T. If, for example, a one-dimensional array is irregular, i.e. has co-ordinates 16.3, 27.6, 48.2, …, then a UDF function can convert from the contiguous co-ordinate system in a basic array to the irregular one above.

Hence, an array can be enhanced with any number of UDFs. Each one adds *pseudo-coordinates* to an array. Such co-ordinates do not have to be integer-valued and do not have to be contiguous. Our array model does not dictate how pseudo-

coordinates are implemented. Some possibilities are as part of the cell data, as a separate data structure, or with a functional representation (if the pseudo-coordinate can be calculated from the integer index).

Addressing array cells of an irregular array can use either the integer dimensions:

A [7, 8] or A[I = 7, J = 8]

or the mapped ones:

A{16.3, 48.2} or A {name-1 = 16.3, name-2 = 48.2}

Lastly, if the dimension is in some well-known co-ordinate system, e.g. Mercator-latitude, then the output of an array enhancement is simply this data type.

When dimensions have "ragged" edges, we can enhance a basic array with a *shape* function to define this class of arrays. A shape function is a user-defined function with integer arguments and a pair of integer outputs.

A shape function must be able to return low-water and high-water marks when one dimension is left unspecified, e.g. shape-function (A[I, *]) would return the maximum high-water mark and the minimum low-water mark for dimension I. In addition, the shape of an individual slice can be returned by specifying *shape-function (A[7,*]).*

To find out whether or not a given cell (e.g., [7,7]) is present in a 2-dimensional array A, we use the function *Exists? [A, 7, 7]* which returns **true** if [7,7] is present and **false** otherwise.

An array can be enhanced with a shape function with the following syntax

Shape array_name **with** shape_function.

Notice that a shape function can define "raggedness" both in the upper and lower bounds. Hence, arrays that digitize circles and other complex shapes are possible. If shape functions can be simplified to have only upper bound raggedness, then the syntax can be simplified somewhat. In addition, it is not possible to use a shape function to indicate "holes" in arrays. If this is a desirable feature, we can easily add this capability.

Every basic array can have at most one shape function, and SciDB will come with a collection of built-in shape functions. In many applications the shape function for a given dimension does not depend on the value for other dimensions. In this case shape is *separable* into a collection of shape function for the individual dimensions. Hence shape-function is really (shape-function (I), shape-function (J)). In this case, the user would have to define a composite shape function that encapsulates the individual ones.

2.2 SciDB Operations

In this section, we describe some of the operators that accompany our data model. Since our operators fall into two broad categories, we illustrate each category with a few examples rather than listing all of them.

2.2.1 Structural Operators

The first operator category creates new arrays based purely on the structure of the inputs. In other words, these operators are data-agnostic. Since these operators do not necessarily have to read the data values to produce a result, they present opportunity for optimization.

The simplest example in this category is an operator that we call *Subsample*. Subsample takes two inputs, an array A and a predicate over the dimensions of A. The predicate must be a conjunction of conditions on each dimension independently. Thus, the predicate "X = 3 and Y < 4" is legal, while the predicate "X = Y" is not. Subsample then selects a "subslab" from the input array. The output will always have the same number of dimensions as the input, but will generally have a smaller number of dimension values.

As an example, consider a 2-dimensional array F with dimensions named X and Y. We can write Subsample(F, even(X)) which would produce an output containing the slices along the X-dimension with even index values. The slices are concatenated in the obvious way and the index values are retained.

Reshape is a more advanced structural operator. This operator converts an array to a new array with a different shape that can include more or fewer dimensions, possibly with new dimension names, but the same number of cells. For example, a 2x3x4 array can become a 2x6x2 array, or an 8x3 array or a 1-dimensional array of length 24.

For example, if G is a 2x3x4 array with dimensions X, Y and Z, we can get an 8x3 array as:

Reshape(G, [X, Z, Y], [U = 1:8, V = 1:3])

The second and third arguments are lists of index specifiers. The first one says that we should first imagine that G is linearized by iterating over X most slowly and Y most quickly. The second list says that we take the resulting 1-dimensional array and form 8 groups of 1-dimensional arrays of length-3 (contiguous pieces from the linearized result), with dimensions named U and V.

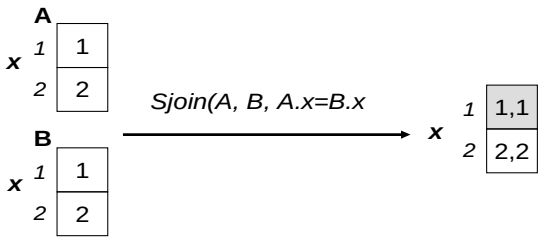

Figure 1 - Example Sjoin

Finally, we define a Structured-Join (or Sjoin) operator that restricts its join predicate to be over dimension values only. In the case of an Sjoin on an *m*-dimensional array and an *n*-dimensional array that involves only *k* dimensions from each of the arrays in the join predicate, the result will be an (*m* + *n* – *k*)-dimensional array with concatenated cell tuples wherever

the JOIN-predicate is **true**. Figure 1 shows an Sjoin over two 1-dimensional arrays. The result is also a 1-dimensional array with concatenated data values in the matching index positions.

Other structural operators include *add dimension*, *remove dimension*, *concatenate*, and *cross product*.

2.2.2 Content-Dependent Operators

The next category involves operators whose result depends on the data that is stored in the input array.

A simple example of this kind of operator is *Filter*. *Filter* takes an array A and a predicate P over the data values that are stored in the cells of A. The argument list to the predicate must be of the same type as the cells in A. *Filter* returns an array with the same dimensions as A. If v is a vector of dimension values, A(v) will contain A(v) if P(A(v)) evaluates to **true**, otherwise it will contain **NULL**.

Aggregate is another example of a data dependent operator. Aggregate takes an *n*-dimensional array *A*, a list of *k* grouping dimensions *G*, and an aggregate function *Agg* as arguments. Note that the Aggregate function take an argument that is an (n-k)-dimension array since there will one such array for each combination of the grouping dimension values.

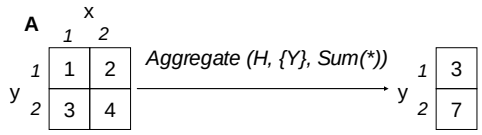

Figure 2 - An Aggregate Operation

As an example, the left side of Figure 2 shows a 2-dimensional input array A to which we apply an aggregate operator that groups on y and returns the SUM of the values in the non-grouped dimensions, in this case *x*. As shown in the figure, the operation *Aggregate (H, {Y}, Sum(*))* will produce the array on the right side of the figure. Note that the data attributes cannot be used for grouping since it is not always possible to determine a meaningful assignment of index values for the result.

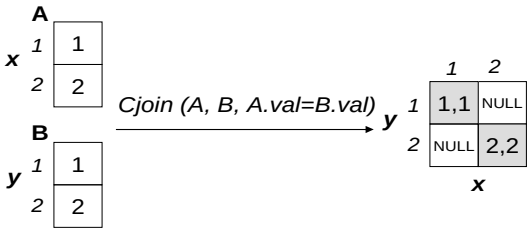

Figure 3 - Example Cjoin

We also define a content-based Join (or Cjoin) that only that restricts its join predicate to be over data values only. In the case of a Cjoin on an *m*-dimensional array and an *n*-dimensional array, the result will be an (*m* + *n*)-dimensional array with concatenated cell tuples wherever the JOIN-predicate was **true**.

Figure 3 shows an example of a Cjoin on the same input arrays as in the previous Sjoin example (see Figure 1). In this case, the result is a 2-dimensional array with multiple index values corresponding to the source dimension values from the original arrays. Thus, cell [1,1] in the result corresponds to data that came from dimension value 1 in both of the inputs. The contents is a concatenated tuple for those cases where the Cjoin predicate is **true**. For cases in which this predicate is **false**, the result array contains a NULL.

Other examples of data-dependent operators include *Apply* and *Project*.

2.3 Extendibility

The key science operations are rarely the popular table primitives, such as Join. Instead, science users wish to regrid arrays and perform other sophisticated computations and analytics. Hence, the fundamental arrays operations in SciDB are user-extendable. In the style of Postgres, users can add their own array operations. Similarly, users can add their own data types to SciDB.

2.4 Language Bindings

There is no consensus on a single programming language for access to a DBMS. Instead, there are many users who want persistence of large arrays in C++, and are partial to the interfaces in object-oriented DBMSs such as Object Design and Objectivity. Other users are committed to Python and want a Python-specific interface. Furthermore, both MATLAB [8] and IDL [9] are also popular.

To support these disparate languages, SciDB will have a parse-tree representation for commands. Then, there will be multiple language bindings. These will map from the language-specific representation to this parse tree format. In the style of Ruby-on-Rails [10], LINQ [11] and Hibernate [12], these language bindings will attempt to fit large array manipulation cleanly into the target language using the control structures of the language in question.

In our opinion, the data-sublanguage approach epitomized by ODBC and JDBC has been a huge mistake by the DBMS community, since it requires a programmer to write lots of interface code. This is completely avoided by the language embedding approach we advocate.

2.5 No Overwrite

Most scientists are adamant about not discarding any data. If a data item is shown to be wrong, they want to add the replacement value and the time of the replacement, retaining the old value for provenance (lineage) purposes. As such, they require a no-overwrite storage manager, in contrast to most commercial systems today, which overwrite the old value with a new one.

Postgres [13] contained a no-overwrite storage manager for tables. In SciDB, no-overwrite is even easier to support. Specifically, arrays can be optionally declared as *updatable*. All arrays can be loaded with new values. Afterwards, cells in

an updatable array can receive new values; however, scientists do not want to perform updates in place. To support this concept, a *history* dimension must be added to every updatable array.

An initial transaction adds values into appropriate cells for history = 1. The first subsequent SciDB transaction adds new values in the appropriate cells for history = 2. New values can be either updates or insertions. A delete operation removes a cell from an array and in the obvious implementation based on deltas, one would insert a deletion-flag as the delta, indicating the value has been deleted. Thereafter, every transaction adds new array values for the next value of the history dimension.

It is possible to enhance the history dimension with a mapping between the integers noted above and wall clock time, so that the array can be addressed using conventional time, and SciDB will provide an enhancement function for this purpose.

As such, a user who starts at a particular cell, say [x=2, y=2, history=1] and travels along the history dimension by incrementing the history dimension value ([x=2, y=2, history=2], etc.) will see the history of activity to the cell [2,2]. Enhancement functions for updatable arrays must be cognizant of this extra dimension.

A variant of Remote, capable of capturing a time series of measurements, might be:

define updatable Remote_2
(s1 = float, s2 = float, s3 = float) (I, J, history)
create my_remote_2 **as** Remote_2 [1024, 1024, *]

Of course, the fact that Remote is declared to be updatable would allow the system to add the History dimension automatically.

2.6 Open Source

It appears impossible to get any traction in the science community unless the DBMS is open source. The main reasons why the scientific community dislikes closed-source software include (a) a need for multi-decade support required by large science projects, (b) an inability to recompile the entire software stack at will and (c) difficulties with maintaining closed-source software within large collaborations encompassing tens or even hundreds of institutes.

Seemingly, the LHC project was so badly burned by its experience in the 1990's with commercial DBMSs, that it has "poisoned the well". We view the widespread application of closed-source DBMSs in this community as highly unlikely.

As such SciDB is an open source project. Because the science community wants a commercial strength DBMS, we have started a non-profit foundation (SciDB, Inc.) to manage the development of the code.

2.7 Grid Orientation

LSST expects to have 55 petabytes of raw data. It goes without saying that a DBMS that expects to store LSST data must run on a grid (cloud) of shared-nothing [14] computers. Conventional DBMSs such as Teradata, Netezza, DB2, and Vertica have used this architecture for years, employing the horizontal partitioning of tables that was first explored in Gamma [14]. Gamma supported both hash-based and range-based partitioning on an attribute or collection of attributes. Hence, the main question is how to do partitioning in SciDB.

For example, LSST and PanSTARRS have a substantial component of their workload that is to survey the entire sky on a regular basis. For these applications, dividing the co-ordinate system for the sky into fixed partitions will probably work well.

Most satellite imagery has the same characteristic; namely the entire earth is scanned periodically. Again, a fixed partitioning scheme will probably work well.

In contrast, any science experimentation that is "steerable" will be non-uniform. For example, it is generally recognized that the mid-equatorial pacific is not very interesting, and many studies do not consider it. On the other hand, during El Nino or La Nina events, it is very interesting.

It would obviously be much easier to use a fixed partitioning scheme in SciDB. However, there will be a class of applications that cannot be load-balanced using such a tactic. Hence, in SciDB we allow the partitioning to change over time. In this way, a first partitioning scheme is used for time less than T and a second partitioning scheme for time > T.

Like C-Store [15] and H-store [16], we plan an automatic data base designer which will use a sample workload to do the partitioning. This designer can be run periodically on the actual workload, and suggest modifications.

Although dynamic partitioning makes joins harder because data movement to support the execution of the join is more elaborate, the advantages of load balancing and data equalization across nodes seems to outweigh the disadvantages.

One research problem we plan to consider is the *co-partitioning* of multiple arrays with a common co-ordinate system. Such arrays would all be partitioned the same way, so that comparison operations including joins do not require data movement.

2.8 Storage Within a Node

Within a node, the storage manager must decompose a partition into disk blocks. Most data will come into SciDB through a streaming bulk loader. We assume that the input stream is ordered by some dominant dimension – often time. SciDB will divide the load stream into site-specific substreams. Each one will appear in the main memory of the associated node. When main memory is nearly full, the storage manager will form the data into a collection of rectangular buckets, defined by a **stride** in each dimension, compress the bucket and write it to disk. Hence, within a

node an array partition is divided into variable size rectangular buckets. An R-tree [18] keeps track of the size of the various buckets. In a style similar to that employed by Vertica, a background thread can combine buckets into larger ones as an optimization.

Optimization of the storage management layer entails deciding:

- When to change the partitioning criteria between sites
- How to form an input stream into buckets
- How and when to merge disk buckets into larger ones
- What compression algorithms to employ

The SciDB research team will address these research issues in parallel with an implementation.

2.9 "In Situ" Data

A common complain from scientists is "I am looking forward to getting something done, but I am still trying to load my data". Put differently, the overhead of loading data is very high, and may dominate the value received from DBMS manipulation.

As such, SciDB must be able to operate on "in situ" data, without requiring a load process. Our approach to this issue is to define a self-describing data format and then write adaptors to various popular external formats, for example HDF-5 [19] or NetCDF [20]. If an adaptor exists for the user's data or if he is willing to put it in the SciDB format mentioned above, then he can use SciDB without a load stage.

Of course, "in situ" data will not have many DBMS services, such as recovery since it is under user control and not DBMS control.

2.10 Integration of the Cooking Process

Most scientific data comes from instruments observing a physical process of some sort. For example, in remote sensing applications, imagery is collected from satellite or airborne observation. Such sensor readings enter a *cooking process* whereby raw information is *cooked* into finished information. Cooking entails converting sensor information into standard data types, correcting for calibration information, correcting for cloud cover, etc.

There are two schools of thought concerning cooking. One school suggests loading raw data into a DBMS and then performing all cooking inside the DBMS. In this way accurate provenance information can be recorded. The other school of thought suggests cooking the data externally, employing custom hardware if appropriate. In some applications, the cooking process is under the control of a separate group, with little interaction with the storage group. With this sort of organization, cooking external to the DBMS is often chosen. However, when there is a single group in control of both processes, then cooking in the DBMS has many advantages.

The goal of SciDB will be to enable cooking inside the engine if the user desires. All that is required is a strong enough data manipulation capability so that it is possible.

2.11 Named Versions

A requirement of most science users is the concept of *named versions*. Consider the following example. The cooking algorithm of most remote sensing data sets includes picking an observation for a particular "cell" of the earth's surface from those available from several passes of the satellite. In other words, a single composite image is constructed from several satellite passes. Often, the observation selected is the one with least cloud cover. However, a scientist with a particular study area and goal might want a different algorithm. For example, he might want the observation when the satellite is closest to being directly overhead. In other words, he wants a different cooking step for part of the data. Scientists desiring a different calibration algorithm for an instrument have essentially the same use case.

Such users want a data set that is the same as a "parent" data set for much of the study region, but different in a portion. The easiest way to support this functionality is with named versions. At a specific time, T, a user will be able to construct a version V from a base array A with a SciDB command. A new array is defined for V and the time T is recorded. At time T, the version V is identical to A. Since V is stored as a delta off its parent A, it consumes essentially no space, and the new array is empty.

Thereafter, any modifications to V go into this array, and the delta will record the divergence of V from A. When the SciDB execution engine desires a value of a cell in V, it will first look in the delta array for V for the most recent value along the history dimension. If there is no value in V, it will then look for the most recent value along the history dimension in A.

In turn, if A is a version, it will repeat this process until it reaches a base array. In general, hanging off any base array is a tree of named versions, each with its delta recorded.

Notice that, as described previously, any updatable array is time-travelled using an extra history dimension. In this way, the history of values of any cell is accessible. Named versions extend this capability by supporting the time travel of a tree of named alternatives to a given cell.

2.12 Provenance

A universal requirement from scientists was repeatability of data derivation. Hence, they wish to be able to recreate any array A, by remembering how it was derived. For a sequence of processing steps inside SciDB, one merely needs to record a log of the commands that were run to create A. For arrays that are loaded externally, scientists want a metadata repository in which they can enter programs that were run along with their run-time parameters, so that that a record of provenance is available.

The search requirements for this repository and log are basically:

1. For a given data element D, find the collection of processing steps that created it from input data.
2. For a given data element D, find all the "downstream" data elements whose value is impacted by the value of D.

When a scientist notices a data element that he suspects is wrong, he wants to track down the cause of the possible error. This is the first requirement, i.e. trace backwards through the derivation process to find where the error originated.

Assuming the scientist ascertains that the data element is wrong and finds the culprit in the derivation process, then he wants to rerun (a portion of) the derivation to generate a replacement value or values. Of course, this re-derivation will not overwrite old data, but will produce new value(s) at the current time. Then, the scientist needs to ascertain how far "downstream" the errant data has propagated, so he can perform downstream re-derivation. The second requirement is used to find out how far a mistake has spread, once D is found to be faulty.

Recording the log and establishing a metadata repository is straightforward. The hard part is to create a provenance query language and efficient implementation. Although one could use Trio [21] as an exemplar, the space cost of recording item-level derivations is way too high.

An alternate approach for backward derivation is to look at the time of the update that produced the item in question. That identifies the command that produced the item from the provenance log. One can then rerun the update in a special executor mode that will record all items that contributed to the incorrect item. Repeating this process will trace backwards to produce the desired result.

Tracing forward is less efficient. If one wants to find all items that are derived from the contents of a specific cell, C, with dimension index values V1, …Vk, then one can run subsequent commands in the provenance log in a modified form. Specifically, one wants to add the qualification

And dimension-1 = V1

And dimension-2 = v2

…

And dimension-k = Vk.

to the first command. This will produce a collection of directly affected values. For each such value, one must run the next command in the provenance log to find the next set of values. This process must be iterated forward until there is no further activity. A named version can be created to hold the results of these updates.

This solution requires no extra space at all, but has a substantial running time. Of course, one can cache these named versions in case the derivation is run again at a later time. This amounts to storing a portion of the Trio item level data structure and re-deriving the portions that are not stored. An interesting research issue is to find a better solution that can easily morph between the minimal storage solution above and the Trio solution.

2.13 Uncertainty

Essentially all scientific data is imprecise, and without exception science researchers have requested a DBMS that supports uncertain data elements. Of course, current commercial RDBMSs are oriented toward business users, where there is a much smaller need for this feature. Hence, commercial products do not support uncertainty.

In talking with many science users, there was near universal consensus on requirements in this area. They requested a simple model of uncertainty, namely normal distributions for data elements. In effect, they requested "error bars" (standard deviations) for data elements and an executor that would perform interval arithmetic when combining uncertain elements. Some researchers have requirements for a more sophisticated model, but there was no agreement on which one to use. Hence, we were repeatedly requested to build in support for normal distributions, leaving more complex error modelling to the user's application.

Hence, SciDB will support "uncertain x" for any data type x that is available in the engine. Of course, this requires two values for any data element, rather than one. However, every effort will be made to effectively code data elements in an array, so that arrays with the same error bounds for all values will require negligible extra space.

Over time, we will revisit this decision, and perhaps build in support for a more sophisticated definition of uncertainty.

There is another aspect to uncertain data, exemplified by the data base design for the PanSTARRS telescope project [5]. During the cooking process the "best" location of an observed object is calculated. However, this location has some error, and the actual object location may be elsewhere.

To deal with this kind of error, the PanSTARRS DBAs have identified the maximum possible location error. Since they have a fixed partitioning schema between nodes, they can redundantly place an observation in multiple partitions if the observation is close to a partition boundary.

In this way, they ensure that "uncertain" spatial joins can be performed without moving data elements. In SciDB, this would surface as uncertainty concerning what array cell an item was in. We expect to extend the SciDB error model to deal with this situation.

2.14 Non-Science Usage

A surprising observation is that the above requirements apply more broadly than just for science applications.

For example, eBay records a click stream log of relevant events from its websites. One of their use cases is "how relevant is the keyword search engine?" In other words, an

eBay user can type a collection of keywords into the eBay search box, for example "pre-war Gibson banjo". eBay returns a collection of items that it believes match the search request. The user might click on item 7, and then follow a sub-tree of links under this item. Returning to the top level, the user might then click on item 9 and follow a similar sub-tree under this item. After this, the user might exit the system or type another search request. From this log, eBay wishes to extract, the fact that items 7 and then 9 were touched, and that their search strategy for pre-war Gibson banjos is flawed, since the top 6 items were not of interest. Not only is it important which items have been clicked through, it is even more important to be able to analyse the user-ignored content. E.g., how often did a particular item get surfaced but was never clicked on?

As eBay's web pages and algorithms get more and more dynamic in nature, traditional weblog analysis cannot provide the required insight as no page or search result is static. As such, deep information about the content present, at the time the user visited, needs to be collected and processed during the analysis.

This application is nearly impossible in current RDBMSs; however, it can be effectively modelled as a one-dimensional array (i.e. a time series) with embedded arrays to represent the search results at each step. A collection of simple user defined functions complement the built-in search capabilities of SciDB to effectively support this use case.

Such time series analysis becomes multi-dimensional in nature. The combination of an array based data model coupled with constructs of uncertainty will provide a new platform for Web 3.0 type analytics on petabytes of information.

2.15 A Science Benchmark

Over the years there have been many benchmarks proposed for various application domains, including the Wisconsin benchmark for conventional SQL functionality [22], the Bucky benchmark for object-relational applications [23], the Linear Road benchmark for stream processing [24] and the various TPC benchmarks. To focus the DBMS community on science requirements, we are almost finished with a science benchmark. We expect to publish the specifications for this collection of tasks during Q1/2009.

III SUMMARY

We are committed to building SciDB and have commitments of resources from eBay, LSST, Microsoft, SLAC and Vertica. We will also try to get NSF to help out, and are in various stages of talks with others.

We have recruited an initial programming team, and have started a non-profit foundation to manage the project. At this time (December 2008), we are nearing a "critical mass" of financial support that will allow us to start building in earnest.

We expect to have a usable system for scientists within two years.

ACKNOWLEDGMENT

We would like to acknowledge the contributions of the rest of the SciDB technical advisory committee, which includes Magda Balazinska, Mike Carey, Ugur Cetintemel, Martin Kersten, Sam Madden, Jignesh Patel, Alex Szalay, and Jennifer Widom.

REFERENCES

[1] Jeff Dozier, Michael Stonebraker, James Frew: Sequoia 2000: A Next-Generation Information System for the Study of Global Change. IEEE Symposium on Mass Storage Systems 1994:47-56.

[2] David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel, Jie-Bing Yu: Client-Server Paradise. VLDB 1994:558-569.

[3] Alexander S. Szalay, Peter Z. Kunszt, Ani Thakar, Jim Gray, Donald R. Slutz, Robert J. Brunner: Designing and Mining Multi-Terabyte Astronomy Archives: The Sloan Digital Sky Survey. SIGMOD 2000:451-462.

[4] Milena Ivanova, Niels Nes, Romulo Goncalves, Martin L. Kersten: MonetDB/SQL Meets SkyServer: the Challenges of a Scientific Database. SSDBM 2007:13.

[5] Alex Szalay (private communication).

[6] Michael Stonebraker, Chuck Bear, Ugur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, Stanley B. Zdonik: One Size Fits All? Part 2: Benchmarking Studies. CIDR 2007:173-184.

[7] Bill Howe,"GRIDFIELDS: Model-Driven Data Transformation in the Physical Sciences", PhD Thesis, Portland State University, (2007) http://www.cs.pdx.edu/~howe/howe_dissertation.pdf

[8] Kermit Sigmon and Timothy A. Davis, "MATLAB Primer", 6th Edition, CRC Press, 2002, ISBN 1-58488-294-8.

[9] OMG: *IDL: Details, http://www.omg.org/gettingstarted/omg_idl.htm

[10] Michael Bächle, Paul Kirchberg: Ruby on Rails. IEEE Software (SOFTWARE) 24(6):105-108 (2007).

[11] LINQ tutorial: VS2008, LinqDataSource and GridView, http://www.codegod.de/webappcodegod/LINQ-tutorial-VS2008-LinqDataSource-and-GridView-AID452.aspx.

[12] Amr Elssamadisy: Review of "Hibernate: A J2EE Developer's Guide by Will Iverson", Pearson Education Inc., 2005, ISBN: 0-471-20282-7. ACM SIGSOFT Software Engineering Notes (SIGSOFT) 31(3):42-43 (2006).

[13] Michael Stonebraker: The Design of the POSTGRES Storage System. VLDB 1987:289-300.

[14] Donovan A. Schneider, David J. DeWitt: A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment, SIGMOD 1989.

[15] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, M. Muralikrishna: GAMMA – A High Performance Dataflow Database Machine. VLDB 1986:228-237.
[16] Michael Stonebraker, et al: C-Store a Column-oriented DBMS, VLDB 2005.
[17] Michael Stonebraker et al., The End of an Architectural Era (It's Time for a Complete Rewrite), VLDB 2007.
[18] Timos K. Sellis, Nick Roussopoulos, Christos Faloutsos: The R -Tree: A Dynamic Index for Multi-Dimensional Objects. VLDB 1987:507-518.
[19] HDF5: API specification reference manual. National Center for Supercomputing Applications (NCSA), http://hdf.ncsa.uiuc.edu/, 2004.
[20] NETCDF User's Guide, http://www.unidata.ucar.edu/software/netcdf/guide.txn_toc.html.
[21] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha U. Nabar, Tomoe Sugihara, Jennifer Widom: Trio: A System for Data, Uncertainty, and Lineage, VLDB 2006:1151-1154.
[22] David Dewitt et al: Benchmarking Database Systems: A Systematic Approach, VLDB 1993.
[23] Michael Carey et al., "The Bucky Object-Relational Benchmark, SIGMOD 1997.
[24] Arvind Arasu et al., Linear Road: A Stream Data Management Benchmark, VLDB 2004.