Overview of SciDB

Large Scale Array Storage, Processing and Analysis

The SciDB Development Team http://www.scidb.org

ABSTRACT

SciDB [4, 3] is a new open-source data management system intended primarily for use in application domains that involve very large (petabyte) scale array data; for example, scientific applications such as astronomy, remote sensing and climate modeling, bio-science information management, as well as commercial applications such as risk management systems in the financial services sector, and the analysis of web log data. In this talk we will describe our set of motivating examples and use them to explain the features of SciDB. We then provide a snapshot of the project 'in flight', describing our novel storage manager, array data model, query language, and extensibility frameworks.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]:

General Terms

design performance

Keywords

ACM SIGMOD Industrial Proceedings Scientific Data Management

1. INTRODUCTION

In this paper we introduce and describe a new data management system; SciDB. We begin by (1) explaining the motivations for SciDB, which lie in serving the needs of the community of scientific users with very large scale, and algorithmically sophisticated, data management problems. Then we (2) move on to describe the features and functionality of our system: our data model, query language, extensibility framework, and (3) our overall architecture. In brief, SciDB is built to support an *array* data model and query language with facilities that allow users to extend our system with new scalar data types and array operators. We then proceed to (4) sketch our system's architecture and highlight certain design features. We envision SciDB as a massively parallel storage manager that is able (where possible) to parallelize large scale array processing algorithms. We conclude the paper (5) with a review of our project in-flight.

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

2. BACKGROUND AND MOTIVATION

Due to the way modern science makes extensive use of *sensor arrays* to measure whatever physical phenomenon is the subject of study, much modern scientific data differs from business data in at least three important respects.

First, as their name suggests, sensor arrays consist of rectangular *arrays* of individual sensors. For example, the Hubble Space Telescope's third generation Wide Field Camera (WFC3) consists of two UV/visible detecting charged-couple detectors, each capturing 2048x4096 pixels. An image from this camera covers an area of the sky about 8.5% the diameter of the moon. [10] Within each WFC3 image there are almost certain to be a large number of distinguishable features corresponding to light from distinct objects such as stars, galaxies and nebula. Each of these observable features spans a block of co-located pixels. Because it is ultimately derived from such sensor arrays scientific data has a necessary and implicit ordering; for each element or data value there are other values left, right, up, down, next, previous, or adjacent to it.

Second, in contrast to most commercial information management applications, scientific analysis typically requires mathematically and algorithmically sophisticated data processing methods. Sensor data is 'noisy'. Typically, it must be heavily pre-processed to be cleaned before scientists can use it, and the cleaned data is input to complex analytic processing. Scientists as consumers of data management technologies are comfortable with tools for statistical analysis such as R, MatLab or SAS. Indeed, the goal of many scientific research projects is to develop novel techniques for data processing and analysis.

And third, because sensor arrays can be manufactured on a large scale, and re-used multiple times (to search for changes over time, such as the short lived appearance of super-nova), data generated by modern scientific instruments is extremely large. For example, the Large Hadron Collider will produce approximately 15 petabytes of data annually [2], and a single experimental run of a modern gene sequencing machine will produce on the order of a terabyte of image data that is reduced through subsequent processing to several gigabytes of short read data.

J \ I	[0]	[1]	[2]	[3]	[4]
[0]	(2, 0.7)	(5, 0.5)	(4, 0.9)	(2, 0.8)	(1, 0.2)
[1]	(5, 0.5)	(3, 0.5)	(5, 0.9)	(5, 0.5)	(5, 0.5)
[2]	(4, 0.3)	(6, 0.1)	(6, 0.5)	(2, 0.1)	(7, 0.4)
[3]	(4, 0.25)	(6, 0.45)	(6, 0.3)	(1, 0.1)	(0, 0.3)
[4]	(6, 0.5)	(1, 0.6)	(5, 0.5)	(2, 0.15)	(2, 0.4)

Figure 1: Simple Two Dimensional SciDB Array.

J \ I	[0]	[1]	[2]	[3]	[4]
[0]	(2, 0.7)	(5, 0.5)	(4, 0.9)	(2, 0.8)	(1, 0.2)
[1]	(5, 0.5)	(3, 0.5)	(5, 0.9)	(5, 0.5)	
[2]	(4, 0.3)	(6, 0.1)		(2, 0.1)	(7, 0.4)
[3]	(4, 0.25)	(6, 0.45)	(6, 0.3)	(1, 0.1)	
[4]		(1, 0.6)	(5, 0.5)	(2, 0.15)	(2, 0.4)

Figure 2: Sparse Array with Jagged Edges and Holes

Unlike the relational model where the concept of implicit ordering is anathema to the model's set-theoretic underpinnings, or SQL DBMS systems where ordering information must be explicitly defined as part of the schema and stored in the database, an *array data model*, with implicit ordering, and notions of 'adjacency' or 'neighborhood', is more desirable in scientific domains. 'Order' in scientific sensor data is not a question of representation: it is fundamental to the semantics of the problem domain. Furthermore, to cope with the complexity of the data processing, scientific users require a much more flexible (ie. extensible) data management platform than those currently available. These factors suggest that a different kind of DBMS is called for.

However, in contrast to tools like the statistical software packages mentioned earlier, the SciDB architecture draws heavily on the scalability lessons learnt by commercial DBMS vendors. SciDB is first and foremost a system for the storage, processing and analysis of data. Our system will rely on other software to handle user-interface, data visualization and so forth.

3. FEATURES AND FUNCTIONALITY

In this section we review the features of the SciDB data model. As we mentioned earlier, SciDB adopts an *array* data model. The properties of this model reflect common scientific use-cases. SciDB database are organized as collections of n-dimensional arrays. *Cells* in a SciDB array each contain a *tuple* of *values*, and individual values in a tuple are associated with a distinguishing *attribute name*.

3.1 Data Definition

To create an array in SciDB, the user would issue the following command:

```
CREATE ARRAY Example
  ( A::INTEGER, B::FLOAT ) [ I=0:4, J=0:4];
```

Figure 1 illustrates what what such an array might look like. SciDB arrays may be *sparse*. A nominally rectilinear array can have jagged edges, and can even have islands of 'empty' cells surrounded by cells containing actual values. In Figure 2, the cells at [4,1], [2,2], [4,3] and [0,4] are all shown as being *empty*.

Drawing on several of our scientific use cases SciDB distinguishes between two classes of 'missing' information conflated in SQL's handling of NULL values. Empty cells of the kind blacked out in Figure 2 are simply ignored for the purpose of any data manipulation operations. Yet scientific applications often employ some mechanism for handling values which are 'out of bound' codes, and are treated differently depending on the operation being undertaken. For example, in remote sensing applications, 'clouds' are encoded differently than 'pixel missing due to camera malfunction'. However, neither of these case mean the pixel 'empty'. An 'empty' cell might occur when several images are stitched together but there are gaps between them.

Values in SciDB attributes can be of any of the expected numerical or (currently) fixed length string data types. From our work with the science community it is clear that these types aren't sufficient to cater to every requirement. For example, a scientific measurement is often accompanied by error bars, or even expressed as a probability distribution function. For these reasons SciDB will support an extensible type system similar to Postgres' user-defined types [8] (see below).Further, the SciDB data model is *nested*; a cell in a SciDB array can itself contain another SciDB array.

3.2 Data Manipulation

Users employ a declarative query language when working with data in a SciDB database . Underlying our query language is a small collection of algebraic primitives which operate on arrays. These primitives can generally be characterized based on whether or not the operations manipulates an array in terms of it's *structure*–the array's rank, and dimension indices–or by addressing the array's *contents*–the data values in attributes in cells. Some do both.

A complete, formal description of the operators in our data model is beyond the scope of this paper. Instead, we provide examples that illustrate several operators and show how they are combined. Three of our structural operators appear below. Figure 3 illustrates the output each of these operations produces.

```
Slice ( Example,  I = 2);
```

Slice() projects an array along a particular index value in single dimension. In this case the Slice() operator will extract from the Example array a single column, corresponding to cells where the column index value is 2.

```
Subsample (
  Example,
  I BETWEEN 1 AND 3 AND J BETWEEN 2 AND 4 );
```

Subsample() is a generalization of Slice(). Instead of a single 'slice' through the array Subsample() extracts a region of the array, where the region is specified by a conjunctive predicate over the dimension indices.

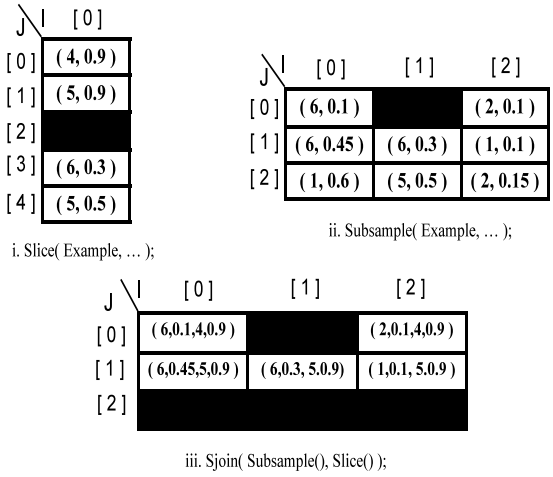

Figure 3: Structural Operator Examples

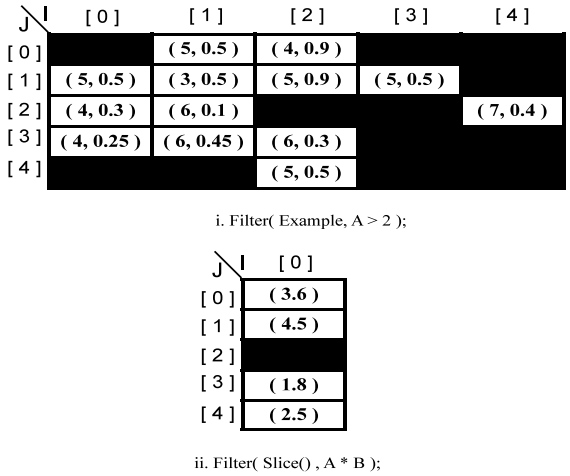

Figure 4: Content Operator Examples

```
SJoin (
  Subsample (
      Example,
      I BETWEEN 1 AND 3 AND J BETWEEN 2 AND 4 ),
  Slice ( Example,  I = 2)
 );
```

SJoin() is a binary array operator that combines cells from two input arrays. It combines attributes from cells with the same dimensional index values. SJoin()'s input arrays need not have identical dimension structure, as we see in this example.

The following pair of query fragments illustrate how users can perform operations on the data contents of the array. Figure 4 illustrates the output each of these query fragments produces.

```
Filter( Example, A > 2 );
```

Filter() applies a predicate to the attribute values in the input array, and produces as output another array of the same size–same rank and dimension. Cells where the predicate is found false are set to empty.

```
Apply( Slice ( Example,  I = 2), A * B );
```

Apply() applies a calculation to attribute values in the input array, and produces a new array with cells populated with new values.

All of the operators in our algebra are *composable*. The second example query above illustrates how to combine two SciDB operators, one a structural operator and the other a content based operator. The list of potential array operators is very large but we currently anticipate implementing about a dozen, basic building blocks and relying on our users to prioritize for us what they require.

Although it elevates arrays as its fundamental organizational method, it is important to note that the SciDB operators have only an approximate correspondence to more rigorous operators of linear algebra. We expect that users will come to require these powerful analytics methods and we are working to introduce them into our system.

3.3 Extensibility

We envision that SciDB will be highly extensible. To satisfy user requirements for non-business data types–complex numbers, values with error bars–SciDB will provide Postgres style user-defined data type(UDT) and user-defined function (UDF) extensibility of the type found in most modern SQL DBMS products. Users will be able to add their own, domain specific data types to a SciDB instance, and these new types will inter-operate with SciDB's own types and array operators.

In addition, SciDB will support a novel class of *operator extensibility*. Our experience working with scientific use-cases suggests that scientific researchers often wish to apply their own highly specific algorithms to data stored in SciDB in conjunction with some pre-processing handled by SciDB's built-in operators. We encourage users to add their own *user-defined array operators* written in C/C++. New operators will take SciDB array input(s), and return array output(s).

For example, one extremely common kind of operation in scientific data management involves examining a region or neighborhood of values in an array. Standard procedure when dealing with image data is to 'smooth' or 'blur' the data to remove irrelevant detail and noise. One popular filtering method is usually referred to as *Gaussian Smoothing* which produces, for each cell in the array, a 'weighted average' of the cell's neighborhood, with the average weighted more towards the value of closer cells. [1] Such an operator might appear in SciDB's query language as follows:

```
Smooth(
  Subsample (
      Example,
      I BETWEEN 1 AND 3 AND J BETWEEN 2 AND 4 )
      );
```

[1]Or. more interestingly, when studying the phenomenon of gravitational lensing (where light from distant objects is distorted or 'lensed' around a closer, massive and invisible object) astronomers want to compare the spectral signatures of multiple glints of light in an image, were each glint spans a block of adjacent pixels.

Obviously this degree of extensibility complicates query planning enormously. The SciDB optimizer will have very limited insight into the logical or performance properties of the operators it is working with. In the case of the Smooth() operator introduced above, the SciDB engine may have to materialize the result of the intermediate Subsample(). For other kinds of operators, however, it might be possible to pipe-line the results of an inner sub-expression directly.

4. ARCHITECTURE

SciDB adopts a *shared nothing* design for its overall system architecture. We envision a SciDB instance being deployed over a network of computers, each with its own locally attached storage. Each compute/storage node runs a semi-autonomous instance of a SciDB engine, providing communications, query processing and a local storage manager. SciDB processes running on each node share access to a (logically) centralized system catalog database that stores information about the nodes, data distribution, user-defined extensions, and so forth. Our design is rather less tightly coupled than most shared nothing systems, and is influenced by the design of modern distributed computing systems that use a Map/Reduce[5] model. We expect this looser architecture will make it easier for us to build more flexible provisioning and reliability. Physical nodes may come and go, but unless a query addresses data stored on a node that is off-line, the SciDB instance is unaffected. Adding nodes amounts to adding new entries in our system catalog: the only central point of failure in the system.

Over the next few sections we review some of the features of our system's implementation.

4.1 Storage Manager

The design of our storage manager draws on features from a number of commercial DBMSs, but includes a number of novel features reflecting the requirements of array data processing.

SciDB implements a distributed, *no-overwrite* storage manager. Data in a SciDB array is not update-able. New array data can be appended to a SciDB database, or the results of a SciDB query can be written back to the storage manager. We plan to implement only the 'A' and 'D' of transactional ACIDity.

In addition to our own purpose built storage manager we anticipate that SciDB will provide access to external data *in situ* through our extensible operator mechanism. For example, scientific users with large collections of NetCDF [9] or HDF [6] files will able to address that data without importing it, and export data from SciDB in these *de facto* standard file formats.

4.1.1 Chunking, and Vertical Partitioning

Figure 5 presents an outline of our approach to storage management; how SciDB maps data in logical arrays into physical storage.

First, in common with the so-called *column-store* storage managers[1, 7] , we vertically partition the data in our arrays. The SciDB storage manager splits attributes in a single logical array and handles values for each attribute separately. In other words, all low level operations in SciDB deal with arrays that contain a single value in each cell. The motivation here is the same as it is for column-store systems. Scientific users often focus their attention, in a particular query, on a sub-set of attributes in the logical array. Vertical partitioning therefore reduces I/O costs.

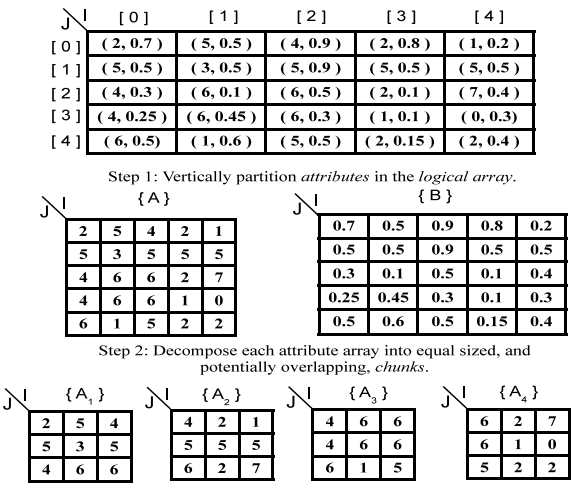

Figure 5: SciDB Storage Manager

Second, our storage manager takes each attribute's data, and further decomposes the array into a number of equal sized, and potentially overlapping, *chunks*. In SciDB chunks are our physical unit of I/O, processing, and inter-node communication. Chunks are quite large: on an order of 64 megabytes. Within the SciDB system catalog we store, for each chunk, the chunk's location–as a range of dimensional indices–within the logical array.

4.1.2 Overlapping Chunks

The motivation for our decision to *overlap* chunks deserves more detailed discussion.

Recall the Gaussian Smoothing operation introduced in Section 3.3. To compute the new, smoothed value for each cell, the algorithm needs to consult attribute values in the surrounding region. The size of the region that contributes to the smoothing operation varies from application to application but is typically fairly small, relative to the overall data. If the array data was partitioned into non-overlapping chunks, SciDB would be obliged to 'stitch together' adjacent chunks to reconstruct 'boundary' regions.

By segmenting our arrays into overlapping chunks, and picking the right 'overlap' extent, we are able to parallelize operations like Gaussian Smoothing. All of the data needed to compute the filter is available within the same chunk. The downside of this strategy is that it increases our storage management overhead, and presents a configuration challenge to SciDB DBAs.

Consider the illustration in Figure 6. Here, an 11x5 array 'A' has been decomposed into three, overlapping 5x5 chunks. With this scheme, an operation that requires the examination of any complete 3x3 subsample of A need only consult the data in a single chunk and the operation can be parallelized three ways. The darker grey areas are regions of the array that are replicated; a particular attribute value in a cell stored in more than one chunk. Lighter gray areas represent regions of the array which are 'non-core' in the sense that any algorithm considering a 3x3 subsample of 'A' will be unable to compute results for these boundary cells.

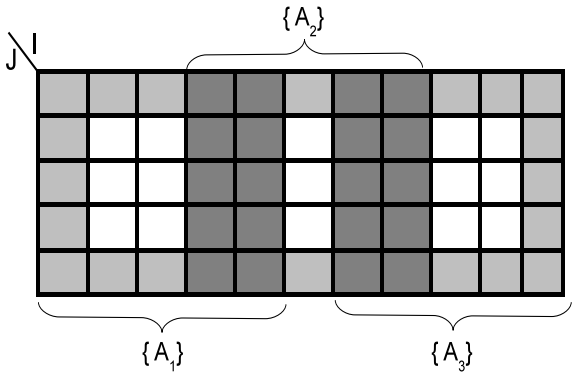

Figure 6: Chunk Overlap Motivation

This discussion of overlapping chunks is a good place to point out another feature of our design. Earlier, we described the SciDB data model's selection of array operators. As physically implemented, these operators are actually applied on a chunk-at-a-time basis. Thus we typically do not have to compute the entire result of an operator applied to an array before going on to the next operator. For example, the chunks produced by the physical implementation of Slice(Example, I=2) from Section 3.2 can be passed immediately to the Apply() operator in a pipeline.

4.1.3 Distribution

Initially we expect that we will distribute data in a simple, consistent fashion. Chunks are assigned to nodes in such a way that chunks containing data values for a particular cell of the logical array are co-located on the same node. The idea is to optimize for queries that combine data from single cells. For example, the following query:

```
Filter (
  Subsample (
      Example,
      I BETWEEN 1 AND 3 AND J BETWEEN 2 AND 4 ),
  A * B > 5 );
```

can be executed by pipe-lineing a chunk-at-a-time through the operator tree, and does not require any communication between nodes.

SciDB records in its system catalogs an entry for each chunk that informs the query processor what region of the logical array each chunk spans and what attribute it contains. We are using the Postgres open-source DB as a system catalog repository manager because this allows us to use R-Trees to quickly identify which chunks contain data relevant to a particular subsample of the array, thereby facilitating fragment elimination and improving query planning.

4.1.4 Compression

Earlier we mentioned that we plan on setting the SciDB chunk size to something reasonably large: 64 megabytes or so. In many of our scientific use-cases, we have observed that *compression* is used very frequently to cut down storage requirements. Consequently we compress each chunk individually when writing it to disk, and uncompress it as we read it in. For some operators it is possible to work on uncompressed data directly, depending on the compression scheme, and we are actively investigating this optimization. Compression is also used in network operations to minimize the bandwidth requirements of our Scatter/Gather operation (see below).

4.2 Query Processing

In contrast with the relational model, where algebraic operators are widely commutable, the SciDB engine has much less flexibility when organizing query plans. Some operations can be re-organized for efficiency, but the semantics of many of them preclude this. For example, we may not know the precise dimensions of the output of an an SJoin() or CrossProduct() operator until the size of it's input arrays is known. Consequently our initial approach to query processing will be very simple. We describe it in this section.

4.2.1 Parsing and Plan Generation

SciDB's extensibility requires that our parser and plan generator consult the system catalogs to determine the semantics of query expressions. At this point we do not envision a type system as rich and sophisticated as the one provided by Postgres, but our decision to support user-defined operators–such as Smooth()–complicates parsing. For each syntactic token we need to determine the operator it corresponds to, and to check that the schema's semantics match the operator's requirements.

4.2.2 Optimization

For each client query, SciDB will perform initial processing–parsing, semantic checks, type checking, lookup of user-defined extensions–at a coordinator node. The coordinator node then parcels out the execution of fragments of the query plan to the distributed nodes.

Because we anticipate that our query planning problem problem will be enormously complex, we plan to use *adaptive* methods. Where possible, the query planner on the coordinator node will produce a complete physical plan from the logical plan tree it is presented with. But where this is impossible, the planner will identify a portion of the logical plan that can be collected into a tree of pipe-lineable operators and then schedule the parallel execution of these physical sub-plans, one per node that contains relevant data.

The result of each of these sub-plans can be thought of as a new (temporary) array. Depending on the result of these physical sub-plans–the temporary array's size, rank and dimensions–the optimizer on the coordinator is then free to perform another processing step.

4.2.3 Query Executor

Physical sub-plans consist of pipe-linable fragments (called *segments*). The SciDB coordinator passes segments to local nodes where they are executed against locally stored data. We anticipate being able to pipe-line chunk-at-a-time through the segment, and storing chunks which make up results locally. Segment executors are localized to each node. One of the complications our design implies is that the shared libraries containing user-defined extensions must be linked into each executable, on each node. In fact, we will be storing these shared library modules as data in our system catalogs.

4.2.4 Scatter/Gather and Inter-Node Messaging

Not all queries can be easily parallelized. We anticipate that, from time to time, SciDB will be required to dynamically re-distribute data over the nodes of the cluster. To cope with these situations we have implemented a special, sophisticated operator pair we call the *Scatter/Gather* (S/G) operator, and a special purpose messaging system designed with our needs in mind. The properties of the SciDB S/G operator are similar to the corresponding functionality in MPP relational systems.

When the query processor determines that it needs to perform a dynamic re-distribution of data, it informs the participating nodes of how the *logical* re-distribution of data needs to occur. That is, it informs each node about the new 'chunk map' for the array; the core data, and the extent of the overlap. Then for each local chunk, the S/G operator assigns the data it contains to a buffer associated with the target node's ID. Once that chunk-sized buffer is full, the S/G operator pushes it to the target node.

The symmetric gather operator receives chunks from other nodes, and may need to *merge* data from multiple remote nodes.

5. PROJECT STATUS

We debuted SciDB at VLDB 2009 [3], were we presented a demonstration system that we have used as a vehicle for exploring user requirements in more detail. At this time (March, 2010) we are in the process of developing a Version 1 release. Our release builds on the lessons of our simple prototype and represents a considerably more flexible and sophisticated implementation of these ideas.

SciDB is being managed as an open-source development project with all code released under the Gnu Public License Version 3.0. We are actively working with three 'early adopter' customers in both scientific and commercial domains. Currently SciDB is being developed by an international pick-up team of engineers. We anticipate release of Version 1 of our system, with most of the functionality described in this document, in the second quarter of 2010.

6. CONCLUSION

We have presented SciDB: a novel, array-centric DBMS designed to cater to large scale, query centric scientific workloads. We motivated our system by describing how in modern, Big Science projects the wide spread use of sensor arrays causes data to be structured in a way that makes it awkward to manage efficiently using relational DBMS approaches. We explored some of the features of our system: it's data model, query language and approach to extensibility. We also described our system's architecture: it's shared nothing architecture, approach to query processing and query execution. We finished the paper with an overview the project's status.

Readers interested in learning more, or volunteering to help, are encouraged to visit our web site:

http://www.scidb.org

7. ADDITIONAL AUTHORS

The SciDB Development Team are: J. Rogers (Brown U.) R. Simakov (NIISI RAS) E. Soroush (U Wash.) P. Velikhov (NIISI RAS) M. Balazinska (U Wash.) D. DeWitt (Microsoft) B. Heath (VoltDB), D. Maier (PSU), S. Madden (MIT), J. Patel (UWisc), M. Stonebraker (MIT), S. Zdonik (Brown U.) A. Smirnov (NIISI RAS) K. Knizhnik (NIISI RAS) and Paul G. Brown (Zetics)

8. REFERENCES

[1] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.

[2] http://public.web.cern.ch/public/en/lhc/computing-en.html.

[3] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. A demonstration of scidb: a science-oriented dbms. *Proc. VLDB Endow.*, 2(2):1534–1537, 2009.

[4] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. Scidb at cidr. *Proc. CIDR.*, 2009.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[6] http://www.hdfgroup.org/documentation/.

[7] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[8] M. Stonebraker and L. A. Rowe. The design of postgres. In *IEEE Transactions on Knowledge and Data Engineering*, pages 340–355, 1986.

[9] http://www.unidata.ucar.edu/software/netcdf/docs/.

[10] Hubble space telescope servicing mission 4 fact sheet, 2007.