Fusing Data Management Services with File Systems[*]

Scott Brandt Carlos Maltzahn Neoklis Polyzotis Wang-Chiew Tan
University of California, Santa Cruz
{scott,carlosm,alkis,wctan}@cs.ucsc.edu

ABSTRACT

File systems are the backbone of large-scale data processing for scientific applications. Motivated by the need to provide an extensible and flexible framework beyond the abstractions provided by API libraries for files to manage and analyze large-scale data, we are developing Damasc, an enhanced file system where rich data management services for scientific computing are provided as a native part of the file system.

This paper presents our vision for Damasc, a performant file system that would allow scientists or even casual users to pose declarative queries and updates over *views* of underlying files that are stored in their native bytestream format. In Damasc, a configurable layer is added on top of the file system to expose the contents of files in a logical data model through which views can be defined and used for queries and updates. The logical data model and views are leveraged to optimize access to files through caching and self-organizing indexing. In addition, provenance capture and analysis to file access is also built into Damasc. We describe the salient features of our proposal and discuss how it can benefit the development of scientific code.

1. INTRODUCTION

The increasing rate of information generation has led to an explosion in the amount of data managed in file systems [14], with some predicting that the data volume of experimental science and simulations is growing 10 to 100 times faster than the compute speeds of the systems managing that data [18] . Yet, most file systems are still using the POSIX IO interface, an interface that was designed in the mid-1960s [11] when high-end file systems stored less than 100MB [32]. Today's personal file systems are 3-4 orders of magnitude larger and high-end file systems in super-computing environments and search engine and social network companies are up to 7-9 orders of magnitude larger, resulting in numbers of data items for which POSIX abstractions are quite inadequate [36]. One important reason for the remarkable stability of this interface is its role as a shield for long-lived I/O libraries and other applications against changes in the underlying storage system. Another is that software above the POSIX interface was perceived to sufficiently compensate for the interface's shortcoming: examples range from file format-specific data management applications and libraries in high-performance computing environments [26, 1, 16, 10] to functionality in personal computing [3, 4, 23].

Recent evidence has shown that the price paid by maintaining the POSIX interface can be high. As an example, PLFS [5] makes the storage layer cognizant of a particular workload pattern and thereby reduces the runtime of many important scientific simulations by nearly three orders of magnitude. PLFS demonstrates two things: (1) the potential performance benefit of making the storage layer aware of workloads is significant enough to merit changes in long-lived applications and interfaces, and (2) despite many man-years of effort, application designers were not able to realize this potential without detailed knowledge of how workloads affect the performance of particular storage system architectures.

We therefore argue that file systems should take over the role of characterizing workloads and data formats, automatically optimizing their internal data structures accordingly. Relieving application developers of the need for detailed knowledge about the performance characteristics of storage systems is a similar to what database designers successfully performed in the 1970s with the introduction of the relational data model and its declarative interfaces. A declarative interface allows application programmers to specify *what* to do based on a logical data model as opposed to *how* to do it based on assumptions about the physical layout.

Motivated by these observations, we are developing the novel Damasc file system that includes mechanisms for translating declarative access into well-performing physical access patterns. In Damasc, we augment the file system with data management services that are important for the types of applications seen today. Specifically, the Damasc file system interface will support the following additional functionality: (1) declarative queries, (2) views, (3) provenance, and (4) automatic indexing. Realizing these services hinges upon the enrichment of file metadata with information on content structure–the file system becomes more "aware" of the data that resides inside files instead of treating them simply as bytes.

Since relational databases already offer mature implementations of declarative interfaces, one possibility would be to replace file systems technology with database technology (similar to [28]). However, scientific data applications consist of tightly cooperating clients performing structured access of very large data volumes that are orders of magnitude greater than any known conventional database management system has ever stored. Conventional database management systems do not scale sufficiently since they were designed for the storage of small, mostly independent data items on memory-starved systems that are randomly accessed by non-cooperating clients which expect strong consistency semantics [31, 38]. Moreover, it would take a major effort to migrate the existing file-based infrastructure to a database management system. Maintaining and tuning the latter would also require an expe-

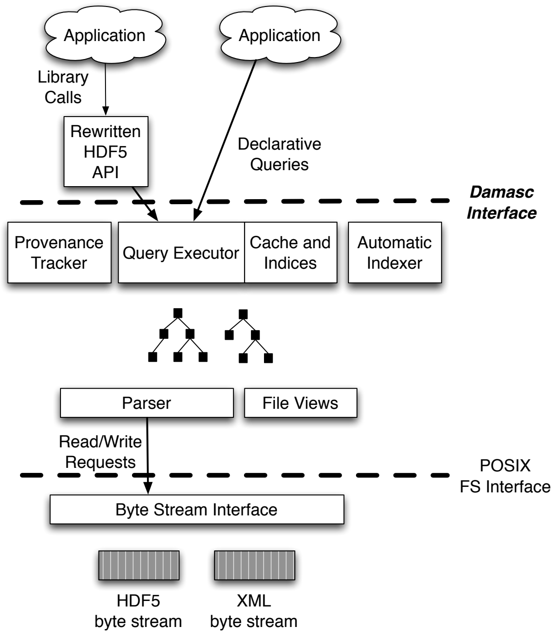

Figure 1: Schematic of proposed file system architecture.

rienced administrator, thus adding to the total cost of ownership. We therefore conclude that real-world applications are unlikely to adopt conventional databases.

Given the history of failed efforts trying to extend or change file system interfaces [27, 9, 30] it is justified to ask why we are convinced that the time to change the file system interface has arrived. A number of key technologies have become available that provide a particularly conducive atmosphere for change: (1) the success of HPC middleware that provides data management functionality missing in the POSIX I/O interface [1, 16, 10] has established API standards that can be used for abstract file system interfaces with little impact on existing applications; (2) recent advances in end-to-end virtualization and performance management [33, 17, 7] allow better scheduling of competing data management activities on storage nodes; (3) scientific simulations process ever larger volumes of data and are increasingly forced to use out-of-core data management (e.g. [21]); (4) recent advances in automatic indexing in databases provides important automatic tuning mechanisms for file systems [35]; (5) recent advances in parallel file systems which shows that greater intelligence at storage nodes leads to greater scalability—some of that intelligence is already used for data management [39]. In short, we believe that the time is right for some data management techniques to be added to file systems to support added services such as declarative querying and updates, better optimization and caching strategies, as well as provenance capture and analysis.

2. OVERVIEW OF Damasc

Figure 1 provides an illustration of Damasc's design. In short, the file system is augmented with a *Parser* module that exposes the data stored in files in a structured logical model (described below). The logical model is equipped with high-level operators for querying and updating the data , which in turn provide the foundation for *declarative queries*. This concept (popularized in the databases domain) implies the following: The application specifies solely *what* data to retrieve from a file, but not *how* to retrieve it; the *Query Executor* module becomes responsible for transparently optimizing the retrieval of the requested data, e.g., using indices or other auxiliary data structures. Damasc also includes an interface that allows applications to perform their own query optimizations, bypassing the system optimizer.

The combination of declarative queries and a logical model also allows the definition of *views*, that is, files whose contents are not materialized but computed on-the-fly based on a query expression. This concept has proven very powerful in databases and, as we argue later, it has important applications in file systems as well.

Finally, Damasc includes two important services that are built over the foundation of the logical data model: *self-organizable indexing* that automatically indexes file contents based on the applications' access patterns, and *provenance tracking* that automatically maintains information on the lineage of the data stored in files.

In the following sections, we discuss each of the aforementioned components in more detail.

Parser Module. At a high level, the *Parser* module maps the unstructured byte stream of a file to a logical model that exposes structural information. The logical model in turn becomes the foundation on which we develop data management services.

More specifically, the module is a repository of parsers for different file formats (see also Figure 1). A parser receives as input a handle to the byte stream of a file of a specific format and outputs an encoding of the contents in the *semi-structured data model*, i.e., as a graph of labeled nodes. For instance, the parser for a bibliography file may output a graph of "publication" nodes, where each node is linked to other nodes describing its data (e.g., "title" nodes, "author" nodes, etc.), and publication nodes may be linked to each other to denote citations. We choose the semi-structured data model due to its flexibility to represent heterogeneous information and also because it is the native format for several scientific applications, e.g., for biological and chemical data [25].

The parsing module performs a similar function to format-specific libraries that also output a structured representation of a file's contents. A key difference of our approach is that every file is parsed to the same logical data model, thus allowing applications to use the same abstractions in order to access and combine data from files of different formats.

It is important to note that the logical data model is not used to physically store information—it serves only as a conceptual representation of files that enables the *formulation* of high-level queries. The *evaluation* of these queries is based on physical operators that match closely the "native" structure of the files' contents. For instance, a query that selects a publication from a bibliography file will be evaluated by reading the relevant extent in the corresponding byte stream, without parsing the complete file or materializing its logical representation. This optimization requires specific query processing mechanisms, which we discuss next.

Declarative Queries. Damasc couples the common logical data model with a language that enables applications to extract and update data in files. The language is declarative, i.e., an expression in this language describes what data to retrieve or update, but not how to do so; the actual evaluation and optimization of the expression is left to the file system, and specifically to the *Query Executor* module. Declarative querying is a staple feature of database systems that has proven crucial for the management of large data sets.

More concretely, we envision a language that comprises algebraic operators. The input and output of an operator is in the semi-structured logical model that we described earlier. As an example, a *path access* operator will take as input the semi-structured representation of a file and output the parts that correspond to a specific path. For instance, given the path[1] *publication[/title contains 'file' 'systems' 'databases']/abstract* over a bibliography file, the operator will return the parts of the file corresponding to abstracts such that specific keywords appear in the corresponding title. We also expect to have operators for filtering the input based on complex conditions, retaining only parts of the input, applying set operations (i.e., union, intersection, difference), and modifying the input (i.e., update operators). The development of these operators can leverage previous work in query languages for semi-structured data, but should also take into account the query operators of existing file formats (e.g., HDF5, NetCDF, and others).

The algebraic property of operators implies that they can be combined to form complex expressions. For instance, an expression may chain a path access operator to a filtering operator to an update operator, in order to modify information in specific parts of the file that satisfy the filtering condition. Another feature of the language is that it is closed under the logical model, i.e., the results of any expression will be a semi-structured graph. Conceptually, this implies that queries take as input files and produce as output other files. In turn, this leads to the following interesting idea of "reverse-parsing": given an instance of the data model, which may result from a query, translate it to a byte stream according to a specific format. Implementing this idea involves several challenges, as it is necessary to reason about the matching of a semi-structured graph to a specific format.

Query Optimization. As described above, a query expression only describes what data to retrieve or update. The efficient evaluation of such a declarative expression is the responsibility of the *Query Executor* module, which takes as input an expression comprising logical operators and outputs an equivalent expression of *physical operators*. The physical operators are also algebraic, and each physical operator has a counterpart in the logical query language. The difference is that it employs a specific algorithm to perform the computation described by the logical counterpart.

To illustrate this concept, let us consider the logical path access operator described above. A straightforward physical implementation may parse the complete file and return only the portions that correspond to the path. However, this physical operator can be very inefficient if the file is large and the path is selective, since it will perform wasteful work. As an alternative, we plan to develop a physical path access operator based on the concept of *fragmented parsing*. The basic idea is to record metadata describing the schema of the file contents and the byte extents associated with the different parts of the schema. The schema metadata can be obtained by examining the header of the file, doing a limited parsing of the file contents, or summarizing and compressing the information resulting from a full parse of the file. The physical operator can match the path against the schema, and then invoke the parser to selectively parse the subset of the byte stream that is relevant for the path. Returning to our previous example, the operator will only parse the parts of the file corresponding to publication titles and abstracts.

Another relevant idea is the use of indexing. This approach is already used in file systems that support keyword queries over files, but in a more limited context. In Damasc, indexing is applied on the structure of file data and not just on keywords. Moreover, index accesses may be combined with parsing in order to provide the final results. For instance, an index may identify the offsets of publication records whose title contains certain keywords, and then these records may be parsed in order to retrieve the corresponding abstracts. This is again an application of the general idea of fragmented parsing, but with a different implementation.

[1]We use the syntax of the XPath [6] language for path expressions.

As mentioned earlier, the Query Executor is responsible for transforming a logical query to an equivalent expression of physical operators. Essentially, this module examines alternative physical expressions that are equivalent to the input query and selects one of low execution cost. This optimization paradigm is common in database systems, but it will require the development of rewrite rules and cost models for physical operators that are specific to the context of accessing and parsing data in a file system.

As a final note, we mention that applications will have the option of bypassing the optimizer and submitting directly an expression of physical operators to the Query Executor. This can be useful when applications wish to perform domain-specific optimizations that are not captured in the general *Damasc* framework. Of course, there is also the option of issuing hybrid expressions comprising both physical and logical operators. In this case, the Query Executor will attempt to optimize only the logical part of the expression without affecting the physical operators specified by the application.

Views. The combination of the logical data model with a query language enables the powerful mechanism of *views*. A view is loosely defined as a file whose contents are the results of evaluating a query over another file. For instance, a scientific application may define a view of an HDF5 file that contains solely a slab of the original data matrix. Views also provide a convenient mechanism for aggregating information from several files. As an example, a view can union publication records from different files and present them as a single file to applications.

A view is a first class citizen inside the file system, treated equal to any file. Thus, it is possible to issue queries on views, or to define views on top of views. The distinguishing difference is that a view may remain virtual, i.e., its contents may not correspond to a physical byte stream, in which case an access to the view is satisfied by generating its contents (i.e., evaluating the defining query) on-the-fly. One possible optimization is to "fuse" the access over the view with the query that defines the view in order to avoid generating the complete contents of the view. This optimization is again performed transparently in the Query Executor.

Views enable the key feature of *logical independence*. This is best described using a simple example. Suppose that an application defines a view over an HDF5 file and performs its data processing based solely on the view. Even if the format of the underlying file changes (e.g., due to an upgrade in the HDF format, or a conversion to a different format), the application can continue operating on the same view as long as the latter is redefined over the modified file. In other words, the view creates a layer of independence between the application and the specifics of the underlying file. We note that logical independence is another important feature offered by database systems, from where the concept of views is borrowed. Our goal is to bring the same properties to applications that rely on the file system for persistence of data.

Self-Organizable Indexing. We expect that indexing will be critical for the optimized evaluation of declarative queries. However, selecting the appropriate indices is a difficult optimization task that must balance the benefit of indexing against the overhead in storage and computation (indices have to be kept up-to-date with respect to modifications in the indexed file). To avoid delegating this problem to applications, we extend Damasc with a service that continuously monitors the querying patterns of applications and automatically installs indices to speed up query processing. The set of constructed indices may thus evolve over time to match the queries issued by applications. We term this paradigm *self-organizable indexing*.

Clearly, we expect indices to be created based on the access patterns of recently evaluated queries. This points to the idea of *hy-*

brid, *partial indexing*, where different parts of the same file may be indexed using different structures. Let us consider a bibliography file as an example. The file system may choose to build an inverted-list index on the keywords appearing in title entries of publications, and a path index [22] that records the paths found in the semi-structured representation of the file. The two indices may be combined in the same query plan or used in isolation—this decision depends on the query and is left again to the Query Executor. Overall, Damasc's indexing paradigm is significantly different than existing approaches in file systems [2, 15, 37, 19], which typically index all information in a file using a single type of physical structure. At the same time, it introduces an additional challenge in the problem of online index selection, requiring the development of novel solutions.

Indices bring important benefits to query processing but they also incur overhead when files are updated. The self-organizable indexing module will take such overheads into account when the indices are created. A different approach is to apply updates to indices only if the system load permits it. This implies that certain parts of the index may become out-of-date temporarily and thus unavailable for query processing. This can also be viewed as a case of partial indexing, except that partialness refers now to the validity of information in the index.

Support for Provenance. In recent years, there has been significant research on how to capture provenance at the level of operating systems (see for instance, [8, 24]). Similar to prior systems, our goal is to transparently and effectively store the necessary provenance information in order to enable provenance tracking functionalities. Unlike prior work, we plan to provide an *end-to-end* provenance tracking service, where data in the result of a view can be seamlessly traced to data in the underlying files, and back to (other) view data.

To give an example, our provenance service in Damasc would be able to handle the following temporal query: Who has accessed any part of matrix E (in a view) in the past two days? With appropriate bookkeeping, the view that produced E is first identified. Accesses to E in the view in the past two days could be traced. In addition, E is traced to the relevant data in the underlying files that was used to construct E through the view. After this, information about which read OS calls accessed these relevant data, as well as which (other) views triggered some of these calls in the past two days are determined. Information about who or which views issued these calls is then returned. Such integrated management of provenance at both levels (i.e., views and OS) can provide valuable semantics, which is a feature that is largely absent from prior systems. In particular, our file system is able to automatically determine the provenance of a *semantic object* (e.g., a matrix), whose information may have been constructed from several underlying files that have undergone several version changes. Since provenance capture and analysis is built-into our file system at both levels, it can be seen as a solution towards automatically bridging the gap between between *disclosed provenance* (i.e., provenance disclosed by an application) and *observed provenance* (i.e., provenance observed by the OS) [8].

Some research issues include the design of appropriate data structures for storing provenance, algorithms for placing provenance information at appropriate nodes in our file system so that the cost of a given workload is optimized, as well as techniques for efficient provenance retrieval through prefetching, caching, and indexing.

3. REALIZATION IN A PARALLEL FS

We plan to implement Damasc by extending the Ceph petascale distributed object-based file system [39]. Ceph's design leverages the intelligence present in OSDs to distribute the complexity surrounding data access, update serialization, replication and reliability, failure detection, and recovery. Ceph utilizes a highly adaptive distributed metadata cluster architecture that dramatically improves the scalability of metadata access, and with it, the scalability of the entire system.

A key component of mapping logical data models to physical data models is a file's *striping strategy*. Our modified implementation of Ceph will employ *format-aligned striping*, where the stripe unit is expressed not in terms of bytes but in terms of structural elements defined in the file's semi-structured representation. More generally, the striping strategy can also take into account the patterns of declarative queries in order to map files to distributed data structures defined in terms of object-level operators.

We plan to adapt distributed query processing to parallel file systems by installing a query processor inside every OSD. When a client receives a query, it breaks it in sub-queries that are shipped to several OSDs, using file system metadata such as striping strategies and other attributes and statistics of physical storage. Each OSD uses its local query processor to parse files locally and evaluate the received sub-queries. Following our previous discussion, the query processor in each OSD can maintain its own cache and indices to speed up query evaluation over the locally stored files. An interesting challenge is to coordinate caching among clients and OSDs in order to avoid redundant caching. We will leverage the rich literature on distributed and parallel database systems [12, 29] in order to address this issue.

4. APPLICATIONS & Damasc

Scientists rely on file systems to store and manipulate large volumes of data. Since the POSIX interface provides little support for structured data (beyond directories and files), scientists typically rely on external data management libraries or APIs (e.g., HDF5) that provide higher level abstractions to support access and updates to structured data in the development of scientific applications.

Damasc can be leveraged by existing applications without rewriting the application code, as long as data management libraries (e.g., NetCDF [20], NetCDF-4 [34], HDF5 [13], MPI-IO [10]) are rewritten to interface Damasc. That is, rewrite only the external libraries to issue declarative queries instead of reading/writing byte extents. For instance, HDF API can be implemented on top of Damasc. In this way, any existing application that makes use of the HDF API can interface with Damasc without any changes to its code.

A consequence of having a common interface (i.e., Damasc) to multiple applications is that Damasc can synthesize access information across these applications. The aggregated access pattern across applications can be analyzed to derive a more effective layout and storage of data for the access pattern. For instance, the striping strategy for a file may take into account the corresponding query patterns in order to improve locality of access within stripes. This type of optimization becomes very difficult when the file system only observes low-level information such as reads and writes to byte extents.

Damasc can also be used to "emulate" (and to some extent, improve) services provided by external libraries. We consider the recently proposed PLFS [5] framework as an example. PLFS aims to provide an efficient checkpointing mechanism for distributed applications. It provides the abstraction of a single file for storing checkpoint data, but internally associates each writer process of the distributed application with a disjoint region of the file. Thus, concurrent write requests can proceed concurrently without the need for expensive synchronization. Our proposed file system can achieve a similar functionality by using views. More concretely, each writer process updates a separate file (without the need for synchronization), and the disjoint files are then union-ed in a view which presents the abstraction of a single file. Recall that the view is not materialized, and any access to the view is translated to accesses over the constituent files. Moreover, the file system can automatically index the contents of the view, thus improving the performance of data retrieval. This is a definite advantage over PLFS, which cannot support efficiently random-access reads over the single file.

We conclude. The amount of data stored is growing 1-2 orders of magnitude faster than the speeds of the computers accessing that data and thus moving the data to the processors is becoming a significant overhead to large-scale data processing. User-level libraries and databases provide much-needed functionality to applications, but suffer from performance overheads. User-level libraries requires data to be moved to the host processors and hide important data layout and access pattern information from the file system, limiting the opportunity for effective prefetching, intelligent data layout, and other possible optimization. Databases provide important functionality, but cannot provide the raw throughput required of high-performance data intensive applications. Damasc will address these shortcomings by adding a layer of data management functionality to file systems. This has several key benefits over existing file systems and databases. It will provide additional information to the storage system, allowing intelligent data-structure and access-pattern aware data layout and storage management. It will also facilitate in-place processing of the data using knowledge of the data layout. This will initially be used for data management operations such as indexing, querying, and provenance management. Eventually, we hope this will lay the foundation for future full-scale distributed processing over the storage system. Our eventual goal is to provide standard interfaces to support arbitrary applications running on remote processors or directly over storage system.

5. REFERENCES

[1] Network Common Data Form. `http://www.unidata.ucar.edu/software/netcdf/papers/`.

[2] Spotlight. http://www.apple.com/macosx/features/spotlight/.

[3] Apple Developer Connection. Working with Spotlight. http://developer.apple.com/macosx/tiger/spotlight.html, 2004.

[4] Beagle Project. About beagle. http://beagle-project.org/About, 2007.

[5] J. Bent, G. Grider, B. McClelland, J. Nunez, M. Wingate, G. Gibson, M. Polte, and P. Nowoczynski. Plfs: A checkpoint filesystem for parallel applications. Technical Report LA-UR 09-02117, LANL, 2009.

[6] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. Xml path language (xpath) 2.0 w3c recommendation. W3C Working Draft, January 2007.

[7] S. A. Brandt, C. Maltzahn, A. Povzner, R. Pineiro, A. Shewmaker, and T. Kaldewey. An integrated model for performance management in a distributed system. In *(OSPERT 2008)*, July 2008.

[8] Uri Braun, Simson L. Garfinkel, David A. Holland, Kiran-Kumar Muniswamy-Reddy, and Margo I. Seltzer. Issues in automatic provenance collection. In *IPAW*, pages 171–183, 2006.

[9] P. F. Corbett and D. G. Feitelson. The vesta parallel file system. *ACM Trans. Comput. Syst.*, 14(3):225–264, August 1996.

[10] P. F. Corbett et al. Overview of the MPI-IO parallel I/O interface. In *Input/Output in Paralell and Distributed Computer Systems*, volume 362, chapter 5, pages 127–146. Kluwer Academic, 1996.

[11] R. C. Daley and P. G. Neumann. A general-purpose file system for secondary storage. In *AFIPS '65*, pages 213–229, 1965.

[12] J. Dewitt, D., S. Ghandeharizadeh, A. Schneider, D., A. Bricker, I. Hsiao, H., and R. Rasmussen. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, 1990.

[13] M. Folk, A. Cheng, and K. Yates. HDF5: A file format and I/O library for high performance computing applications. In *Proceedings of SC'99*, Portland, OR, November 13-19 1999.

[14] J. F. Gantz, D. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva. The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011. An idc white paper - sponsored by emc, IDC, March 2008.

[15] Google. Google desktop - features. http://www.google.com/enterprise/gsa/index.html, 2006.

[16] Hierarchical Data Format. `http://www.hdfgroup.org/HDF5/`.

[17] T. Kaldewey, A. Povzner, T. Wong, R. Golding, S. A. Brandt, and C. Maltzahn. Virtualizing disk performance. In *RTAS 2008*, St. Louis, Missouri, April 2008.

[18] P. Kogge and et al. Exascale computing study: Technology challenges in achieving exascale systems. Technical Report TR-2008-13, DARPA, September 28 2008.

[19] A. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *FAST'09*, San Jose, CA, February 2009.

[20] J. Li, W.-K. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *SC '03*, 2003.

[21] Grant Mackey, Saba Sehrish, John Bent, Julio Lopez, Salman Habib, and Jun Wang. Introducing map-reduce to high end computing. Austin, TX, November 2008.

[22] T. Milo and D. Suciu. Index structures for Path Expressions. In *Proceedings of ICDT*, pages 277–295, 1999.

[23] MSDN. Indexing service. http://msdn.microsoft.com/en-us/library/aa163263.aspx, 2008.

[24] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference, General Track*, pages 43–56, 2006.

[25] P. Murray-Rust. Chemical markup language. *World Wide Web Journal*, pages 135–147, 1997.

[26] netcdf operator. `http://nco.sourceforge.net/`.

[27] N. Nieuwejaar and D. Kotz. The galley parallel file system. *Parallel Computing*, 23(4-5):447–476, June 1997.

[28] M. A. Olson. The design and implementation of the Inversion file system. In *Winter 1993 USENIX Technical Conference*, January 1993.

[29] M. Tamer Ozsu and P. Valduriez. *Principles of distributed database systems*. Prentice-Hall, Inc., 1991.

[30] P. Corbett et al. Proposal for a common parallel file system programming interface 1.0. Technical Report CMU-CS-96-193, Carnegie Mellon University, 1996.

[31] J. L. Pfaltz, R. F. Haddleton, and J. C. French. Scalable, parallel, scientific databases. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, pages 4–11, July 1998.

[32] James N. Porter. Five decades of disk drive industry firsts. http://www.disktrend.com/5decades2.htm, 2005.

[33] A. Povzner, T. Kaldewey, S. A. Brandt, R. Golding, T. Wong, and C. Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. In *Eurosys 2008*, Glasgow, Scottland, March 31 - April 4 2008.

[34] R. Rew, E. Hartnett, and J. Caron. netcdf-4: Software implementing an enhanced data model for the geosciences. In *22nd International Conference on Interactive Information Processing Systems for Meteorology, Oceanograph, and Hydrology*, 2006.

[35] K. Schnaitter, N. Polyzotis, and L. Getoor. Index interactions in physical design tuning: Modeling, analysis, and applications. In *VLDB '09 (also available as Tech. Rep. UCSC-CRL-09-23)*, 2009.

[36] M. Seltzer and N. Murphy. Hierarchical file systems are dead. In *HotOS XII*, Monte Verita, Switzerland, 2009.

[37] C. A. N. Soules and G. R. Ganger. Connections: using context to enhance file search. In *SOSP '05*, New York, NY, USA, 2005.

[38] M. Stonebraker, J. Becla, D. Dewitt, K.-T. Lim, D. Maier, O. Ratzeberger, and S. Zdonik. Requirements for science data bases and SciDB. In *CIDR 2009*, Monterey, CA, January 4-7 2009.

[39] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. Seattle, WA, November 2006.