DataSeries: An efficient, flexible data format for structured serial data

DataSeries Technical Documentation
software@cello.hpl.hp.com

August 15, 2017

Note: some of the experiments described are based on old versions of the DataSeries code, prior to many of the later improvements. This tr snapshot built from the 2008-02-27 version of the source with this note added.

Abstract

In this paper we describe DataSeries, an on-disk data format and run-time library that is optimized for analyzing *structured serial data*, which we define as a series of records that share a common structure. The need to maintain and analyze such data occurs in a large number of scientific fields. We discuss how DataSeries has been optimized to be extremely space and CPU efficient, while providing the flexibility to accommodate a very wide range of record structures. We then validate our claims on a variety of storage trace data. In particular, we show data compression rates several times better than existing, specialized formats, processing rate improvements of up to an order of magnitude, storage of hundreds of billions of records and that we can store and process these traces on very modest equipment. Finally, DataSeries software is open source, enabling others to take advantage of these benefits.

1 Introduction

Traces, recordings and measurements taken from computer systems, networks and scientific infrastructure are vitally important for a large variety of tasks. In every area of computer system design, traces from existing systems have been used to validate hypotheses, test assumptions and estimate performance. This is true of I/O subsystems [3, 15, 30], processor systems [21], network systems [16] and memory systems [26], among others. Traces and logs are also extremely useful for fault-finding, auditing and debugging purposes [22]. Traces composed of failure data have been used to determine system reliability [8, 25, 23]. Trend analyses of performance information is a core operation of various management tools [12]. Specific to the area of I/O and storage systems alone, we found that almost 60% of papers published in the File and Storage Technologies (FAST) conferences have used traces of one sort or another. Scientific and medical instrumentation can also generate large amounts of data [4], which also needs to be stored, filtered and analyzed.

The data stored in each of these diverse uses is *structured serial data*, which we define as a series of records, each record having a specified structure (i.e., containing the same set of variables). Structured serial data has four defining characteristics: its structure is record-oriented; it is typically written only once, not modified afterward, and is read many times; it is usually ordered in some manner, e.g., chronologically; and it is typically read in a sequential manner. Traditionally, researchers and developers have accomplished the tasks of collecting, storing, and analyzing this type of data using formats, libraries and software which are customized to the particular task at hand. Unfortunately, such approaches significantly limit both flexibility and reusability, and often performance. For instance, if a binary format is used, it may be difficult to add new items of information or remove obsolete information. More flexible formats (e.g., text or XML) are not amenable to efficient analysis or storage. Traditional databases generally store data without compression, requiring large amounts of disk space for storage, and are typically not optimized for the specific types of processing used on structured serial data. One of the key contributions of this work is the specific storage format and optimizations that make it possible to very efficiently store and analyze this type of data, without requiring excessive amounts of disk storage or computational resources.

Another, often overlooked advantage of having a one-size-fits-all trace format is the ease of use provided in data analysis. In our own work, we have combined disparate trace types (block level I/O, process accounting, system call, NFS, batch scheduler, and system performance traces) in various combinations. Having a single software system and trace format that can work with each of these trace types through merge and analysis operations greatly facilitates the ease of analysis. A related advantage is scientific; reproducibility is enhanced and interpretation is simplified. The authors have experienced, both personally and anecdotally, the difficulty of reproducing results with "old" trace files and software (often just finding software and getting it to compile can be extremely problematic). We observe that of the 101 papers published in the history of FAST, 57 used traces in some way; the traces used in these papers were of at least 45 different formats. Several papers even used 4-5 different types (e.g., [20, 7]).

There are five key properties that are required of a data format and analysis system for structured serial data:

1. **Storage efficiency**: the data should be stored in as few bytes as possible.

2. **Access efficiency**: accessing, interpreting and encoding trace data, whether reading or writing, should make efficient use of CPU and memory resources.

3. **Flexibility**: adding additional fields should not affect users of the trace data. Removing or modifying data fields should only affect users who use those fields and the system should support catching incorrect usage. Further, the format should not constrain the type of data being stored, and should allow multiple record types in a single file.

4. **Self-describing**: the data set should contain the metadata that describes the data.

5. **(Re)Usability**: the data format should have an associated programming interface that is both expressive and easy to use.

Although numerous tracing and measurement systems have been developed over the last 20-30 years, we are not aware of any that meet all of these requirements. We analyze some of these in our description of related work (section 2).

We provide four primary contributions in this paper. First, we introduce DataSeries, a data format and associated library, which was specifically designed to meet the five key properties discussed above. Second, we discuss how DataSeries can support very large datasets (e.g., hundreds of billions of records) on modest systems. Third, we describe how we have used DataSeries in practice to store a wide variety of data types. Fourth, we demonstrate the performance and storage efficiency of DataSeries in a set of controlled experiments, using empirical data sets. We show that the performance of DataSeries exceeds the performance of common trace formats and databases by at least a factor of two, and in some cases up to an order of magnitude. DataSeries also requires far less disk space (factors vary from 4X to 8X in test workloads).

Since DataSeries software is publicly available (under a BSD software license), and given the benefits of DataSeries that we demonstrate, we argue that DataSeries should be considered for use by any application that needs to store large amounts of structured serial data. Indeed, a storage industry group[1] has chosen DataSeries as a standard format for I/O trace data, and is currently specifying the semantics of the fields.

The remainder of this paper is organized as follows. Section 2 describes the strengths and weaknesses of existing storage technologies relative to DataSeries. Section 3 describes the design of DataSeries, including on-disk and in-memory formats. Section 5 describes the programming interface for DataSeries. Section 6 presents empirical and benchmark results from our use of DataSeries to illustrate and quantify the benefits of DataSeries. Section 7 describes our experiences with using DataSeries, and Section 8 concludes the paper with a summary of our work and a list of future directions.

2 Related Work

We classify the related work into three categories: those that use a customized binary format, those that use a text-based format, and relational database systems.

A large number of serial data formats use a custom binary format, in which the in-memory structures or objects representing records are directly written to or read from disk. The primary advantage is that access can be very efficient (on read, memory structures are typically converted using a pointer cast on the raw data read from disk). There are three main disadvantages with this approach. First, it can be difficult to add or remove data fields. Second, most of these formats are designed for a particular data type; for example libpcap [17] for network packet data, SRT [28] for block disk I/O traces and JCAMP for spectroscopy data [5]. Consequently, there is no easy way to re-use the format, or any of its associated software, for another data type. Third, the software to manipulate binary formats may be nonportable unless the developer is careful to avoid endian, word size and alignment issues. One semi-general binary format is CDF [9]. CDF is designed to support random reads and writes in a portable manner on multi-dimensional arrays. Because of the need to support writes within the array, CDF has limited support for compression and variable length data.

DataSeries addresses all of these disadvantages. Each DataSeries file contains a description of the data structure(s) stored within, which enables those structures to be

[1] name withheld for blinding purposes

easily changed, without affecting DataSeries software or clients using it. DataSeries can easily be used with multiple data types, even within a single file, and handles endian and similar issues internally, and transparently to the user. These features of DataSeries are described in section 3. A custom binary format may still be appropriate during initial capture if it is naturally produced by the system, for example pcap [17] for capturing network packets.

A common alternative to binary formats is to use text, typically one record per line with fields in a record separated by spaces or commas (i.e., CSV). Using text has the advantages of being portable across platforms and readable by many tools, but introduces three problems. First, it consumes more space than the equivalent binary format. Second, it takes substantially more time to process. Third, text formats have difficulty storing strings that contain the field separator or storing binary data. Compressing the text format can reduce its space needs, but it generally does not work as well as compression in DataSeries because DataSeries can do type specific transforms (see section 3). We find that DataSeries typically gets a factor of 2x better compression over the equivalent compressed text, and is 7-25x more efficient to decode (see section 6 for experimental results). A CSV format may be appropriate for interchange between systems, or for small datasets.

One increasingly popular variation on text encoding is to use XML. XML provides a large degree of flexibility, coupled with a variety of well-implemented parsers. XML unfortunately takes up even more space than CSV and is much slower to process. The primary advantage of XML is its self-describing format. DataSeries is both flexible and self-describing but is significantly more efficient (for both storage and access) than XML. An XML format may be appropriate for interchange between systems, for small datasets, or for cases where the data to be stored requires an arbitrary hierarchical structure. DataSeries uses XML internally to describe the types of data that it stores, because those descriptions are small and can leverage XML's self-describing format.

Relational databases provide an extremely flexible and easy to query system for storing structured serial data, and provide a high-level data manipulation language (SQL). Databases have four main limitations. First, the databases that can handle hundreds of billions of rows in their data [31] run only on high-end SMPs [6] in large clusters [2]. Second, databases usually store the data uncompressed, which means it consumes much more storage space than necessary. Third, while SQL is general, queries that cannot be expressed in SQL, such as the stack-distance analysis for caching, require that the application retrieve all of the rows, which is very inefficient. Fourth, most database files are not intended to be portable between systems in their raw format which means to move data between researchers the data has to be exported. DataSeries has handled hundreds of billions of rows on low end servers and small RAID arrays, it stores its data compressed, provides efficient access to all of the rows, and DataSeries files are portable between machines. However, DataSeries currently has very limited support for generic queries, so for moderate sized datasets and questions that can be expressed in SQL, a database can be an excellent solution.

Some recent research on column stores [29] holds the promise for more efficient databases operating on structured serial data, although the SQL limitations are still present. We examine the performance of one such system (C-Store) in section 6 and find that DataSeries can outperform C-Store, and is significantly more flexible.

3 Design

DataSeries is intended to provide streaming access to structured serial data. Corresponding to the first four properties[2] described in the introduction, DataSeries was designed with the following goals in mind. First, it should be very storage efficient. Second, it must be efficient to encode, decode and interpret the data. Third, the format should not constrain the types of information to be stored. Fourth, the internal data must be self-describing, i.e., the names and types of the data stored have to be determined by the contents of the file itself, rather than externally.

The file structure and type definitions are described in the user guide.

3.1 Extent types and options

Extent types and options are now described at <https://github.com/dataseries/DataSeries/wiki/Defining-types>.

3.2 Design summary

The DataSeries file format was designed to allow for flexibility (through the use of a self-contained and extensible type description for extents) and performance (through extensive use of compression and a data layout that allows for direct access to data values). Section 6 describes experiments using DataSeries that quantitatively validate these claims.

[2]The fifth property, an expressive programming interface, is described in Section 5.

4 File format specification

The file format is described on the DataSeries wiki at <https://github.com/dataseries/DataSeries/wiki/Dataseries-file-format>

5 Programming

Programming in dataseries is now described as part of the DataSeries User Guide <https://github.com/dataseries/DataSeries/wiki/Dataseries-user-guide>.

6 Performance Results

We performed various experiments to measure the effectiveness of DataSeries' compression techniques, and then further compared other types of data encoding and analysis tools for compression and execution speed. We first describe the experimental setup, then the workloads, the benchmarks, and finally our results.

6.1 Experimental setup

The test-bed we used to perform most of the quantitative benchmarks for this work was a cluster of 18 servers configured for batch processing of single server jobs. Each server had one or two dual core Opteron 280 2.4GHz processors. Each processor had 64KB of L1 D-cache and 1024KB of L2 cache. Additionally, each server was configured with 4GB of main memory and could access a 10TB NFS filesystem over 1Gb/s Ethernet, the underlying storage being RAID6 in the form of HP MSA20 and MSA60 disk arrays. The cluster was configured with RedHat Enterprise Linux 4 and each server was running the 2.6.9 SMP x86_64 kernel version.

Finally, our comparison with C-Store [29] was performed on a single machine with two dual-core Intel Pentium 4 3.0GHz Xeon processors, each with 16KB L1 D-cache and 2048KB L2 cache. This machine was configured with 5 GB of RAM, running Debian Etch 4.0 with a 2.6.21.3 SMP-Bigmem Linux kernel. The system also had a single 160GB Samsung HD160JJ Serial ATA hard drive.

6.2 Data set descriptions

Our data sets included the cello disk traces ("disk") from HP Labs [28], which we converted into DataSeries from a custom binary format, NFS traces collected from a busy enterprise file server ("NFS"), and file system call data from [27] ("system call"). Having three trace formats each with very different extent types and associated data values provides an indication of the performance and flexibility of DataSeries in general. For all experiments to have a minimum of 10 extents with an extent size of 128MB (the largest we measured) all formats were transcoded into 1.2 GB (when uncompressed) files. The smallest data set had six of these files.

For most of our experiments, the results from all three data sets were similar, so we will only present detailed results and details on the disk results. We discuss our use of the NFS traces in more detail in section 7, as it is by far our largest dataset ($\approx$ 5TB).

The disk traces contain entries that correspond to operating system level read and write requests for blocks in a storage system. Each request contains three time fields (stored as doubles) describing when that request was submitted to the device driver (enter_driver), when the request returned from the storage device (return_to_driver), and when the request was returned to the calling process (leave_driver). Additionally, the size (number of bytes read/written) of each request, the logical volume identifier and the device number are recorded as 32 bit integers. There are 28 boolean fields, eight 32-bit fields (including those mentioned above), two 64-bit fields and the three double time fields.

The disk trace data set included six data files. For the DataSeries analysis these files were compressed using lzf compression to an average size of 320MB. For the CSV analysis, they were converted to CSV format using a DataSeries to CSV converter. For the MySQL analysis, the files were further converted to the MySQL bulk-load format and loaded into a single MySQL database table. The six files comprise our "small" data set, while the combination of all six into a single file comprise our "large" data set. The final data sizes in all cases are shown in Table 1.

6.3 Benchmarks

DataSeries is optimized for, and performs very well on, queries which operate on scans of data or ranges of data. We executed several encoding and decoding microbenchmarks to demonstrate the performance and tunability of DataSeries as a trace storage and processing format.

Additionally, to provide a comparative analysis versus other known techniques for data processing, we generated nine related queries to run against a portion of the disk data set. Unfortunately, C-Store could not perform the set of queries generated, so we performed a single simple query to compare C-Store and DataSeries.

We performed experiments with data sets of two different sizes. We performed warm-cache experiments with

Trace Name	Avg. CSV Size	DataSeries Size	MySQL Table Size
small disk trace	2.3GB	320MB	1.5GB
big disk trace	14GB	1.9GB	8.5GB

Table 1: Trace data sizes in CSV, DataSeries and MySQL formats.

the small data set since it could fit in main memory. We performed cold-cache experiments with the large data set.

6.3.1 Compression microbenchmark

DataSeries currently supports four different compression algorithms (bzip2 [1], gzip [10], lzf [18] and lzo [19]), and an arbitrary extent size for record data. Empirical knowledge and algorithm author data seem to indicate that the algorithms are optimized for different usages. For example, bzip2 is commonly believed to compress better than gzip, albeit more slowly. Also, all compression algorithms in common use today use a compression window, giving the impression that compression ratio and perhaps compression and decompression rate are optimized for files above a certain size (i.e., the window size). We evaluated the compression ratio, compression rate, and decompression rate for each of these algorithms using various extent sizes.

Several of the compression algorithms (bzip2 and gzip) have tunable parameters, which trade off compression rate for increased compression ratio. We evaluated a range of settings for each of these algorithms. For the remainder of this section, we utilize bzip2 level 9 as the representative for bzip2, and gzip level 6 as the representative for gzip. We found these levels provide reasonable tradeoffs between the compression rate and ratio for each algorithm respectively.

Extent size determines the maximum window of data a compression algorithm can look at. For very small extent sizes, we expected to see poor compression ratios because redundancy that would have been within any algorithm's window size was artificially being blocked. For very large extent sizes, we expected to see differentiation among algorithms based on their window sizes.

The microbenchmark consisted of reading each DataSeries file and recompressing with the compression algorithm under study. The uncompressed data size was divided by the CPU time for the compression operation to compute *Compression Rate*. Next, the same file was decompressed and its content thrown away. The uncompressed data size was divided by the CPU time to compute the *Decompression Rate*. Finally, the compressed DataSeries file size was divided by the uncompressed DataSeries file size to determine the *Compression Ratio*. The decompression and compression operations were performed three times for each data file in each dataset. All microbenchmark measurements were taken with checksum validation enabled.

Next, we examined the performance of each algorithm, one metric at a time. For publishing traces or in archival situations the compression ratio will be the dominant metric to consider. Figure 1 shows the performance of the tested compression algorithms on the disk trace data. Bzip2 is the clear winner, achieving an average compression ratio of about 10:1, for extent sizes 1 MB or larger. gzip and lzo performed similarly, achieving a maximum compression ratio of about 6:1, for extent sizes larger than 128 KB. lzf had the poorest compression ratio on this data set, achieving a maximum compression ratio of about 3:1.

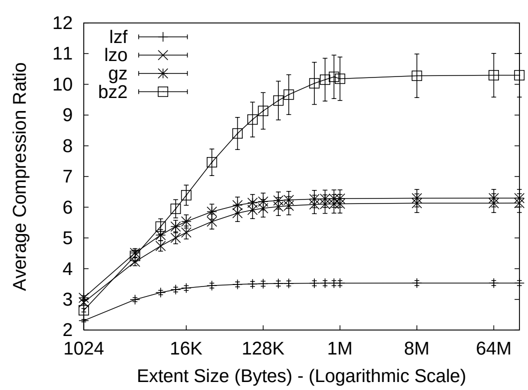


Figure 1: Compression ratio versus extent size results for disk trace data.

For online generation of DataSeries files, the compression rate will be the dominant metric to consider. Figure 2 shows the compression rates achieved by each of the algorithms for the disk trace data. lzf dominates in terms of compression speed, achieving a peak compression rate of 90 MB/s, over four times that of the next best algorithm (gzip). The extent size appears to have only a marginal effect on the compression rate achieved by the tested algorithms. The "no compression" (none) curve indicates the cost imposed by the checksumming and data transforms. The cost increases above 128KB as the data no longer remains in the L2 cache between the transform and compression operations.

For trace analysis, the decompression rate will be the dominant metric to consider, followed by compression ratio. Figure 3 shows that the lzo algorithm has the highest decompression rate, exceeding lzf while also achieving 2x

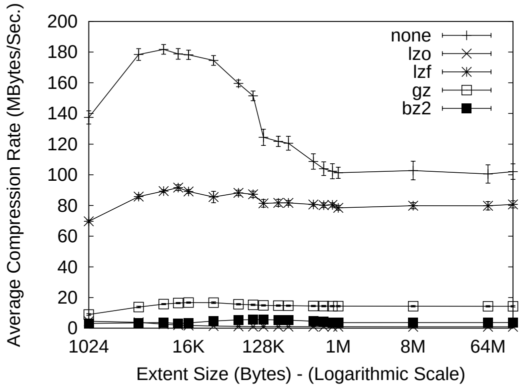


Figure 2: Compression rate (logarithmic scale) versus extent size results for disk trace data.

more compression.

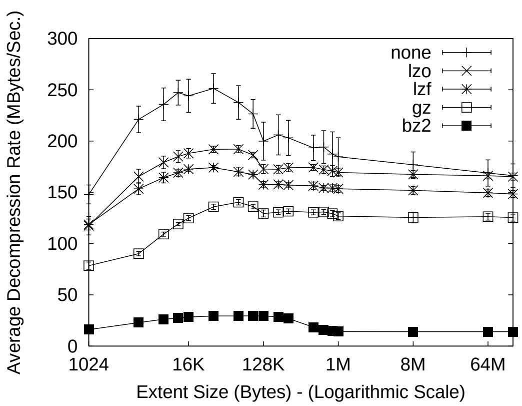


Figure 3: Decompression rate versus extent size results for disk trace data.

A different view of the data can clarify these tradeoffs. For online creation of DataSeries files we care about the compression rate and the compression ratio. Figure 4 compares these metrics with one point for each extent size. The compression rate is shown in log-scale because the different algorithms have vastly different rates. This figure reinforces the previous graph showing that lzf dominates with regard to compression rate, but gzip is a good tradeoff between compression ratio and rate, sacrificing 10x the rate to get 2x the compression. bzip2 is useful if very high compression ratios are desired, while lzo is dominated by all others on this graph. Neither bzip2 nor lzo is likely to be suitable for online creation.

Figure 5 compares the algorithms by the compression ratio and decompression rate metrics. This is important for repeated analysis of data, a very common use case for DataSeries. In this case, lzo dominates the other algorithms in terms of decompression rate (~175 MB/s), while still keeping a reasonable compression ratio (~6:1). lzo is

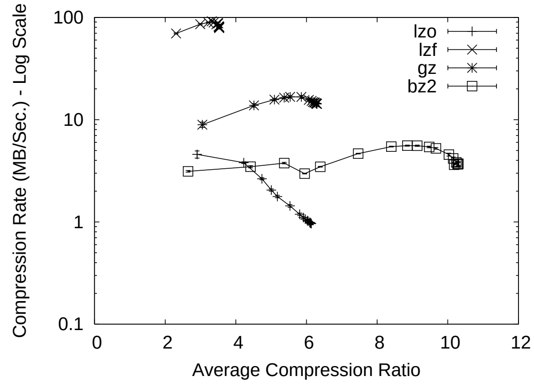


Figure 4: Compression rate versus compression ratio results for disk trace data.

strictly superior to lzf. bzip2 achieves 2x increase in compression ratio, but at a 10x reduction in decompression rate. gzip achieves negligibly higher compression ratios at a 1.3x reduction in decompression ratio. Thus, gzip might only be considered if lzo's compression time is too excessive.

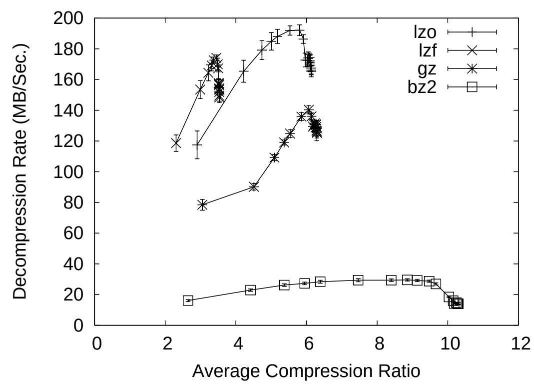
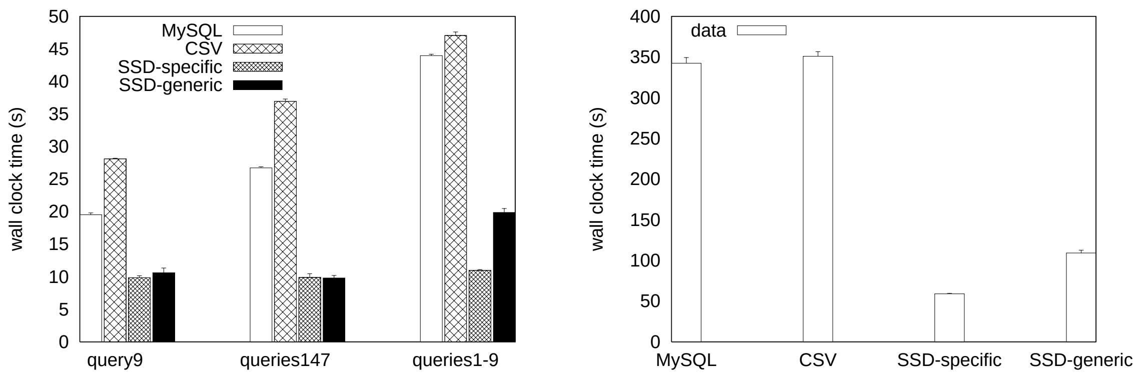


Figure 5: Decompression rate versus compression ratio results for disk data.

6.3.2 Comparison with CSV, MySQL

This set of benchmarks compared a hand-coded, type-specific DataSeries module (DS-specific), a command line DataSeries interface using DSStatGroupByModule (DS-generic), the MySQL database, a custom application for parsing and processing Comma Separated Value (CSV) files and C-Store, a research column-store database. These three additional analyses give a sense for how well DataSeries performs versus common alternatives. [3]

[3] These experiments were performed with lzf compressed files. We plan to re-run the experiments with lzo compressed files and expect to see improved performance.



(a) (b)

Figure 6: Query processing times for three sample queries using MySQL, our custom CSV engine, and DataSeries. Standard deviations for all data are smaller than 5% of the average value: (a) 2.4GB disk trace File; (b) 14GB disk trace file.

While the authors are not database researchers, we felt using MySQL as our representative database was a fair comparison because it is open source (and thus an option for any researcher), provides the necessary SQL parsing engine, is widely used for data analysis tasks, and has reasonable performance. It also provides an easy comparison point for others to use when evaluating relative performance of their current data analysis setup versus what they would gain by using DataSeries. We believe for our experiments MySQL was suitably tuned as the results from the large data set experiment were consistent with the run being disk bound, and the results for the small trace file were consistent with the run being CPU bound.

The first set of queries compute count, average, standard deviation, minimum and maximum over the difference of each of the three time fields, selecting for and grouping by each of the three non-time fields. This leads to nine possible queries.

The compute time of these queries is relatively small so performance should be dominated by the scan time of the data. Ideally, only a single scan of the data should be sufficient to compute the results for these queries. We attempted to optimize DataSeries, MySQL and CSV parsing to extract the fastest query response times possible.

We optimized DataSeries by creating a type-specific version of the queries, thereby eliminating the run-time type checking present in the general purpose DSStatGroupByModule. We also disabled checksum validation to further improve performance.

We optimized SQL by minimizing the number of queries we issued. Instead of issuing nine separate queries, we combined the queries when they were grouping by the same field, resulting in only three queries to compute nine underlying SQL queries.

We optimized the CSV parsing by tuning the program, carefully parsing the lines, caching conversions from strings to doubles, and only converting fields that were being used. Profiling showed we still spent 80% of the instructions in these operations with the remainder in the statistics calculation.

Each complex query was run seven times with the file system cache warm for the 2.3GB data set for each system. The results are plotted in Figure 6(a). The single query takes an average of 22.1 seconds with MySQL and 28.1 seconds with CSV, while DataSeries processes the same query in an average of 9.85 seconds, or 2-3X faster. Data processing rates are 2.3GB/22.1sec = 108MB/sec, 85MB/sec and 243MB/sec respectively. When three queries are combined, the MySQL data processing rate drops to 2.3GB/32.1sec = 74.4MB/sec, CSV drops to 64.7MB/sec while DataSeries remained relatively unchanged at 242MB/sec. Per operation overhead with DS-generic is much higher than DS-specific, therefore, when all 9 queries are run, DS-generic statistics computation dominates processing time, while DS-specific runtime continues to be dominated by decompression.

Figure 6(b) demonstrates the benefit of the compression in DataSeries. In this experiment the large disk trace is used, so the trace must be read from disk rather than from the file system buffer cache. As a result, the MySQL data processing rate has dropped to 41.9MB/sec and CSV is at 40.9MB/sec, as both are disk bound. However, DataSeries continues to process data at 243MB/sec (because of the use of compression). While investment in a faster disk subsystem could improve MySQL and CSV performance, DataSeries is well balanced for modern desktops, compute clusters and laptops. A modern 1U server might have 2 or 4 drives and 8 cores, the number

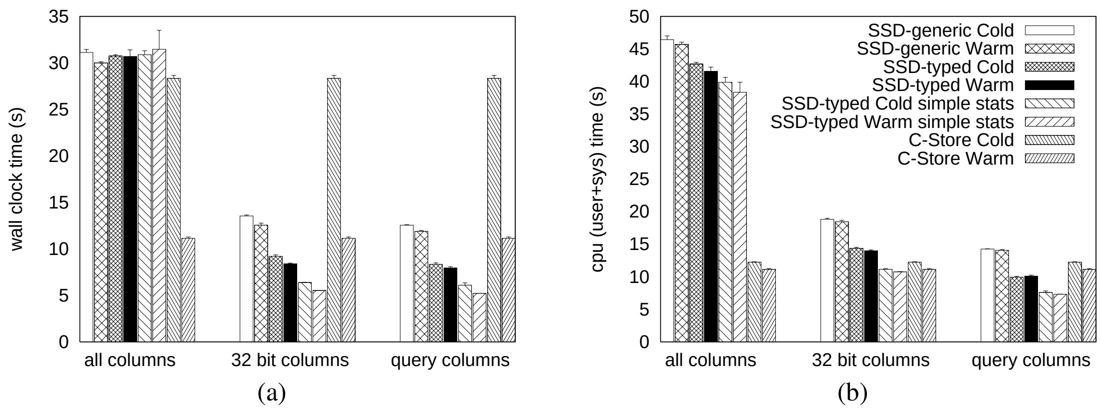


Figure 7: Query processing times for one simple query using C-Store and DataSeries. Standard deviations for all data are smaller than 5% of the average value: (a) Wall clock time (sec); (b) CPU time (sec).

of drives is unlikely to increase, but the number of cores continues to go up.

6.3.3 Comparison with C-Store

As discussed in section 2, C-Store has recently been developed as a more efficient DBMS for read-mostly data. Thus, we wish to compare its performance to DataSeries. Unfortunately, the open source C-Store implementation does not support any data type except 32-bit integers, does not support expressions, and does not support multiple aggregates in a single query. Therefore, we could not compare C-Store in the same manner that we evaluated MySQL and CSV. Instead, for the C-Store comparison, we chose a simple query that C-Store could support, the average I/O size in bytes grouped by device number, on the large disk trace. We used the default configuration of C-Store. The simple query was run five times for each configuration of DataSeries and C-Store, with both a warm and cold file system cache. We used the generic DataSeries program and the type specific version from the previous comparison. We also created a special case type-specific version that only calculated the one statistic used in our simple query.

The advantage of C-Store is that it is only reading the columns that it needs in order to perform the calculation, whereas DataSeries has to read all of the columns. Indeed, Figure 7(a) shows that when operating on all columns, the warmed C-Store has a lower wall clock time than any of the DataSeries configurations. C-Store's advantage when cold is quite small; this is a result of the lack of (functioning) compression in C-Store.[4] However, if we prune the DataSeries file to just the 32 bit integer columns supported in C-Store, the performance of DataSeries can be better than C-Store. In particular, the wall clock times for the type specific and the one-statistic versions of the DataSeries programs both run faster than the C-Store queries for both the warm and cold cases. The CPU-time results shown in Figure 7(b) indicate that only the one-statistic version of DataSeries is using less CPU time; the much better wall clock time shows the benefit of overlapping the decompression and statistics calculations. This result is somewhat surprising as [11] showed a column store needed to access 70-80% of the columns in a row to use more CPU or wall clock time than a row store, but we are showing that 25% (2 of 8 int32 columns) is sufficient for the row store to be faster. This shows the efficiency of the programming interface in DataSeries. As a final comparison we prune the files to just the columns used in the query. In this case, the CPU time for the one-statistic version of the DataSeries program drops to 2/3 of the C-Store CPU time.

The limited functionality of the C-Store implementation make it unusable for generic trace storage and analysis, but the results show that some of the column store techniques to avoid processing un-needed columns may benefit DataSeries provided they can be implemented without sacrificing the efficiency of the DataSeries implementation. In our experience with DataSeries, we usually run multiple queries (different modules) at the same time when analyzing data and the combination of those modules often accesses most of the columns. If this usage is common, the advantages of column oriented storage would be reduced.

[4] C-Store is supposed to support compression but we were unable to get it to work.

6.4 Ellard Traces

In an effort to experiment with using DataSeries to represent and analyze traces generated by other people, we converted Daniel Ellard's Harvard traces[7] into DataSeries. The Ellard traces were originally stored as compressed text files, one record per line. The first part of each line is a series of fixed fields, followed by a set of key-value pairs, and finally some debugging information. As part of the tools, there is also a scanning program which reads the trace files and outputs summary information for a trace.

Our evaluation came in two parts. First, we wrote programs that converted between the two formats. The reversable conversion guaranteed that we were properly preserving all of the information. We found that the DataSeries files were on average 0.74x the size of the original files when both were compressed using gzip. Second, we wrote an analysis program that implemented the first three examples in the README that came with the tools. We found that our analysis program ran about 76x faster on those data files than the text analysis program that came with the distribution. We also found that if we utilized lzo compression, which decompresses more quickly than gzip, our analysis program ran about 107x faster, in exchange for slightly larger (1.14x) data files. Other options in the space-time tradeoff are described as part of the experiments.

6.4.1 Conversion to and from DataSeries

The conversion to DataSeries was interesting for two reasons. First, it stretched DataSeries in the direction of supporting many nullable fields, which we had previously resisted going. Second, it turned out to be a case study in the difficulties posed by ad-hoc text formats.

DataSeries was designed to follow a relational database model. As such it supports null fields, although we have not put any special optimizations in place to handle them (null fields were previously just an extra boolean and a test). While there was a slight space penalty for having a few null fields, this penalty was mostly removed by the compression algorithms. However, when there are many null fields, it could be much more space efficient to explicitly remove the null values and create variable length rows before using a generic compression algorithm. We call this process null compaction. We had previously considered and decided against this feature because it encourages people to choose schemas that do not follow normal form. In the Ellard traces, this manifests as not knowning which columns will be valid given a particular operation type when it is likely that the operation type and valid fields are fixed.

In our initial conversion of the data, it appeared that the duplicate elimination performed by the pack_unique option would compensate for the additional space used by storing a value for all of the fields. However, once we had identified all of the fields we needed to store, the naive implementation resulted in larger DataSeries files than text files. There were two options at this point, first, null compaction, and second, normalizing the data design. Normalizing the data design would involve separating out the rows into multiple tables potentially with keys to have a common and optional tables. After further consideration we decided that the most faithful representation would be a single table, and hence to get small files we implemented null compaction.

We implemented a reverse conversion program so that people could continue to use existing scripts, and so we could verify that the conversion worked properly. This turned out to be very important, as the files had a number of glitches in them. This is a common problem with under-specified text formats: it is very easy to generate a file which appears to conform to a specification, but doesn't. The same problem can occur with XML, as it is easy to generate invalid XML. Related to this problem is the lack of a specification; without a specification, users have no idea what information may be present. Similarly, in XML without a document type definition (DTD), it is difficult to understand the meaning of a parsable document. We solved the problem of unparsable lines by introducing a "garbage" field into the DataSeries output that stores unparsable lines. We currently have code that detects all of the unparsable lines, but if we had known there would be as many as there were, we would have instead written the code to throw an exception on parsing errors and store unparsable lines as garbage.

We experienced a number of these problems in parsing and converting the Ellard data. We plan to make checksums (`sha1` and `md5`) available so that people can validate they are working with the same input data we used. We categorize these problems as follows:

- **Duplicate keys.** The Ellard traces have key-value pairs on each line. We initially asumed that the keys were unique. However, we learned that this assumption is incorrect, as a subset of the keys can occur multiple times on a single line. Inspection of the code that ships with the Ellard traces indicates it handles this case by detecting the duplicate key and silently appending a "-2" to the field name in the in-memory representation. We translate these fields as `_dup` to make them clearly separate from the Ellard translation of *field*2 for some duplicated fields.

- **Unknown keys.** There is no explicit list of the keys used in the Ellard traces, hence we had to dynami-

Trace Set	gz-64k	gz-128k	gz-512k	lzo-64k	lzo-128k	bzip2-16M
overall	0.9459x	0.8531x	0.7721x	1.1387x	1.0437x	0.76x
deasna	0.9800x	0.8856x	0.7996x	1.1535x	1.0552x	0.8064x
deasna2	1.0003x	0.8976x	0.8051x	1.1680x	1.0614x	0.8148x
home02	0.9111x	0.8204x	0.7440x	1.1252x	1.0335x	0.7084x
home03	0.9059x	0.8170x	0.7422x	1.1197x	1.0301x	0.7073x
home04	0.8974x	0.8094x	0.7358x	1.1148x	1.0261x	0.6979x
lair62	0.9663x	0.8949x	0.8271x	1.1153x	1.0427x	0.8145x
lair62b	0.9859x	0.8883x	0.8009x	1.1597x	1.0579x	0.8438x

Table 2: Compression ratios for the different options. The gz and lzo columns are compared relative to the gzip compressed text files. The bzip2 results are compared to bzip2 compressed text files. The results exclude the 8 files with zero blocks in them. The sizes after the compression ratio is the extent size used for the DataSeries files.

cally build up the list of keys that could be present by attempting to parse a file and generating an error if we found a new key. The DataSeries files include (as part of the extent type) the list of all unique fields observed. We later learned that except for the duplicate key issue and the few keys with 2 on the end of the names, the key names follow the xdr spec, so could be inferred from there.

- **Keys with identical meaning and different names.** The Ellard traces parse NFSv3 file handles as a field named `fh`, but NFSv3 commit file handles into a field named `file`. Similarly, file names are parsed as `fn` for v2 and `name` for v3, and offset is parsed as `offset` for v2 and `off` for v3. The DataSeries files document these inconsistencies in a comment for that field. We could have removed the inconsistency, but that would have been less faithful to the original files. This inconsistency is present as a result of Ellard's converter following the xdr spec which uses different names for fields with the same meaning. The intent was to make it easy to map the traces back to the xdr, we would have chosen to make the field names consistent to make it easier to write analysis.

- **Format changes.** The Ellard traces document that at some point they changed the semantics of the `acc` field from a character to a bitmask. To make conversion from DataSeries to the text format work, we had to determine the date for the change so we could generate different output depending on the date. The DataSeries files always use a bitmask, but if we had encountered this problem, a version change would clearly indicate the format used in each file. The earlier acc values are inaccurate and shouldn't be trusted, the DataSeries files have a comment indicating the time at which the switch occurred.

- **Format inconsistencies.** The Ellard traces document a series of fixed fields at the beginning of each line. However, for the null operation, the reply format is missing one of those fixed fields; we had to special case parsing null fields. Similarly, different operations print out the fields in different orders. While this is valid and correct, it meant we had to special case the conversion from DataSeries to text to print fields in the appropriate order.

- **Garbage times.** The Ellard traces specified that times were in microseconds since the unix epoch, consistent with how NFS represents these times. In particular, times were printed as the regular expression [0-9]+
.[0-9]6, i.e. a series a digits, a period and then 6 more digits. Unfortunately, in a number of cases, the lines did not match that format; 185 of these cases we explicitly listed, and two numbers showed up sufficiently often that we checked for them explicitly. While the number of garbage times is a small fraction of the total number of lines, it still is worrisome. We also subjected the times to a check that they were in a reasonable range of 9 or ten digits for the seconds. We identified 17 special numbers that we can't prove are invalid, but some of them are likely to be network parsing errors; however since we couldn't tell, we parse them as if valid. The specific values can be found in the `ellardnfs2ds.cpp` source file.

- **Garbage trailer.** The Ellard traces end with some debugging information that has been converted from numbers to XXX's in the anonymous traces. Unfortunately, when the line was short, the cleanup for the trailer was done incorrectly and the debugging information was left in. Initially, it looked like the debugging information was all identical for short packets, but later we found some cases where it wasn't, so we passed it through as garbage. Interestingly, the documentation says that the debugging fields can't be removed because it would break scripts. This is an advantage of a format like DataSeries wherein analysis that don't need those fields would not care if they were removed.

- **Non-data lines.** A few lines started with "XX Funny line" We pass these lines through as garbage.

- **Unknown errors.** There were 36 lines which had some sort of random error in them. Most of the errors look like a number of characters were inserted or deleted at random combining or splitting multiple lines. A few of them look like the underlying packet data was bad, but an output line was still generated as the stable field is listed as '?'.

- **Zero blocks.** Eight of the compressed files have long (multi-MB) blocks of null characters ('\0') in them. We suspect this came from a toolchain error before we got the files, we re-downloaded one of the files and verified that it had the block of nulls in it. This confused our parser since it saw a line that just happened to not end with a newline, but thought it had reached the end of the file as we were using `fgets`. We eventually decided not to try to translate these files, although in theory we could update our program so that it would properly parse them, and pass them through as garbage. The affected files are listed in ellardnfs2ds.cpp.

6.4.2 Compression comparison

Table 2 shows the DataSeries compression results relative to the original Ellard traces. The compression difference using bzip2 compression is slightly lower than with gzip, the DataSeries files are 0.76x smaller than the text files compressed using bzip2. We compared the lzo files to the gzip compressed files since for the text files compression with lzo would offer no benefit, the files would be larger and the wall time for analysis would be the same. For gzip we tested with extent sizes of 64k, 128k and 512k. For lzo, we tested with extent sizes of 64k and 128k. We didn't bother to calculate compression ratios for lzo-512k because the performance is no better than gzip, and the compression would be worse. Interestingly, the ratios are not constant across the different trace sets:

We have not investigated what causes the difference in the compressed file sizes. We have observed that for the small extent sizes (64k/128k) the compressed extents are very small (5-10k), which means that some of the DataSeries per-extent overhead may be contributing to the larger size, as well as the compression algorithms may not have enough data to even fill their window.

6.4.3 Performance comparison

For the performance comparison, we implemented a subset of the analysis performed Ellard's `nfsscan` program. In particular one that can perform the first three of the five example questions presented in the EXAMPLES file that comes with Ellard's trace tools. This analysis turned out to be very simple, it is just counting the number of requests performed of each client of each type. We chose to implement this over the integer operation id, rather than the string, and so wrote a short table that converted NFSv2 and NFSv3 operation id's into a common space. The performance comparison was done using DataSeries revision 61f07e212acb972da6c603bed82ab2ec5ca1b731 from 2008-01-21.

Our initial implementation did not perform as expected. In particular, we expected to see the CPU utilization exceed 100% during execution because the analysis and the decompression steps were overlapped. Further investigation using `oprofile` indicated that the analysis module was only using 4% of the total CPU time; 96% of the application's time was going into decompressing the extents. We therefore decided it was time to implement parallel decompression so that we could take suitable advantage of our multi-core machines.

The implementation on multi-core machines appeared as if it would be straightforward, we implemented a standard pipeline, with one thread for prefetching extents off disk, and n threads for decompressing extents, defaulting n to the number of CPU cores. Each stage in the pipeline had a maximum memory capacity.

Experiments with this scheme indicated that the performance had high variance. After studying the problem we identified two issues. First, the analysis thread could be pre-empted by a decompression thread, and second, the thread decompressing the first extent in the decompression queue could be pre-empted by any of the other threads. Either of these situations could result in stalls in processing, and instability in the performance. Eventually we decided to detect these two conditions, the second by noting that the first extent in the decompression queue is not ready, and the current decompression thread is working on an extent far down in the queue. In either of these two cases, the decompression thread will call `sched_yield` to try to get the more important thread running again.

This use of `sched_yield` is an inferior solution because it can result in many system calls that end up doing nothing, or simply transfer us from running one thread that doesn't matter to a different thread that doesn't matter. With the existing threading interface, the only other option to try would be to use priorities, however it is not clear that priorities are sufficiently pre-emptable across

CPUs to have a useful effect. If we were to extend the kernel threading interface, there are two obvious possibilities for an improved interface. The first is what we call a directed yield, it would be a variant of `sched_yield`, but would specify a thread that should start running, the call would transfer control to that thread if it is not running, or do nothing otherwise. The second possibility is process local priorities, this would allow us to increase and decrease the priorities dynamically during a run (which is currently not allowed as threads usually can't increase their priority), and it would isolate this process from other processes so that decreasing a thread's priority would not cause other processes to run in preference to the thread with lowered priority. It is unclear which of these solutions would work best, or how they could be made properly composable so that in more complicated pipeline graphs the "right thing" can still happen.

All of our experiments were performed on a DL365g1 with 8 GB memory, 2x 2.4GHz dual-core Opteron 2216 HE. Data was stored on nfs. Ellard's `nfsscan` program was run `zcat (or bzcat) | nfsscan -t0 -BC -`, so we get separate times for the decompression and nfsscan execution, but a single elapsed time. The detailed measurements can be found in the DataSeries distribution in `doc/tr/ellard-details.tex`.

We compare below the performance of running `nfsscan` either with gzip or bz2 inputs, and the performance of running DataSeries with bz2, gzip, and lzo inputs. Interestingly, for the gzip inputs, the scheduler chose to keep the gunzip and the `nfsscan` processes on the same CPU. For bzip2, it used different CPUs, which meant that `nfsscan` ran somewhat slower, presumably because the data had to be copied between CPUs, bzip2 has a larger block size, and the buffering was insufficient to smooth out the difference.

Table 3 presents the summary results, showing the impressive speedup and reduction in CPU time that can be achieved by using DataSeries. The different sizes specified after the compression algorithm for the DataSeries rows are the extent sizes. The substantial increase in system time for dealing with large extents for bzip2 is a result of glibc's use of mmap/munmap for large allocations. Every extent results in a separate pair of mmap/munmap calls to the kernel and hence a substantial about of page zeroing in the kernel. The detailed measurements can be found in the DataSeries distribution in `doc/tr/ellard-details.tex`.

6.5 1998 World cup traces

The 1998 World cup traces[14] are one of the largest publically available web traces. They were collected from the many web servers for the 1998 World Cup games. We converted the traces from the special raw format used for them into DataSeries, and also created a version of the `checklog` program that comes with the traces as a comparison point for analysis. For these traces, we found that conversion to DataSeries files were on average 0.93× the size of the original files. The analysis program used about 1.17× less CPU time, and ran about 3.46× faster on the complete data set.

The analysis program used about 1.04× less CPU time, and ran about 2.60× faster on the complete dataset in the original version. After optimization, the analysis program used about 1.17× less CPU time, and ran about 3.46× faster. Compared on a subset of data that fit entirely in memory,

6.5.1 Conversion to and from DataSeries

The conversion to and from DataSeries was quite straightforward. The traces use a simple fixed record format of 4 32 bit integers and 4 8 bit integers. The integers are unsigned in the original format, but since DataSeries only supports signed 32 bit integers, we converted to a signed format, and convert backwards during analysis.

As with most of our converters, we convert both backwards and forwards to verify that the conversion was correct. The conversion results turned out to be very strange; while overall DataSeries compressed 2% better, some of the files ended up being smaller while others ended up being larger. Further examination identified five factors that were affecting the file size:

- **Compression level**. The original gzipped World Cup files were compressed using `gzip -6`, as that is the default compression level. DataSeries files default to `gzip -9`. We simply re-compressed all of the original files as level 9. However, this points out the importance of defaults as people have been downloading and using needlessly oversized files since they were released.
- **Endianness**. The original World Cup files were stored in big-endian format. DataSeries files are stored in the native endianness of the host, which on x86 processors is little-endian.
- **Field order**. The original World Cup files were stored with the four 32-bit fields first and then the 4 8-bit fields. DataSeries happened to store the files with all the 8-bit fields first followed by the 32-bit ones.
- **Field padding**. The original World Cup files included no padding. DataSeries padded the files to an 8 byte boundary, so added in an extra 4 bytes of zeros to each record.

algorithm	mean user (s)	mean system (s)	mean CPU (s)	CPU speedup	mean wall (s)	Wall time speedup
ellard-gz	537.58	7.80	545.38	1.0x	545.71	1.0x
ellard-bz2	638.48	12.68	651.16	0.836x	571.49	0.955x
ds-gz-512k	22.91	3.62	26.53	20.557x	7.16	76.186x
ds-gz-64k	21.45	1.14	22.59	24.147x	5.81	93.945x
ds-gz-128k	23.30	1.19	24.49	22.268x	6.30	86.604x
ds-bz2-16M	94.38	11.82	106.20	5.136x	27.66	19.732x
ds-lzo-64k	18.71	1.14	19.85	27.472x	5.10	106.897x
ds-lzo-128k	21.15	1.10	22.25	24.514x	5.74	95.022x
ds-lzo-512k	24.07	4.07	28.14	19.382x	7.40	73.762x

Table 3: Summary of performance results for the two analysis programs operating on a variety of input files. The analysis was run over the anon-home04-011118-* files. For the ellard `nfsscan` program the text files were compressed with either gz or bz2. For the DataSeries `ellardanalysis` program, the dataseries files were compressed with either gz, bz2, or lzo, and used various extent sizes as specified. CPU and wall time are both relative to ellard-gz.

- **Relative field packing**. The original World Cup files stored the time field as absolute seconds. DataSeries includes an option to store fields calculated relative to the previous value.

Table 4 shows the overall compression of the different options. The {le,be}-{bts,stb}-gz files are the original data files stored in little or big endian format (le/be) and big to small or small to big (bts/stb) ordering. The original traces are be-bts-gz. The DataSeries file without any options is ds-base, and then options can be added on for reducing the field padding to the maximum column size (mcs), packing the fields in big to small order (bts), or packing the time field self-relative (tsr).

dataset	size (MiB)	ratio to ds-base	ratio to be-bts-gz
ds-mcs-bts-tsr	7648.98	1.133	1.077
ds-mcs-tsr	7722.78	1.122	1.067
ds-bts-tsr	7890.12	1.098	1.044
ds-tsr	8065.99	1.074	1.021
le-bts-gz	8152.47	1.063	1.011
ds-mcs-bts	8183.98	1.059	1.007
be-bts-gz	8239.15	1.052	1.000
le-stb-gz	8391.09	1.032	0.982
be-stb-gz	8411.46	1.030	0.980
ds-mcs	8423.56	1.029	0.978
ds-bts	8439.72	1.027	0.976
ds-base	8663.78	1.000	0.951

Table 4: Compression options for the data files sorted from the smallest to the largest. Gzipped files can be stored little or big endian (le/be) and small to big or big to small (stb/bts). DataSeries files can be stored with the fields padded only to the maximum column size (mcs), with the fields stored big to small (bts), or with the time field packed self relative (tsr). be-bts-gz is the original 1998 World Cup trace format.

6.5.2 Original file compression experiments

Figure 8 shows the four possible options for packing the data before compression in the original data format. Since the dataset includes some empty fields, we always get some files that compress the same regardless of the packing (to 20 bytes, the minimal gzip file size).

The top two lines compare the small to big and big to small ordering options holding the endianness constant. Since both lines are entirely positive, this shows us that we always prefer the big-to-small ordering for this dataset (if the ratio is > 1 then the denominator is smaller and hence the preferrable choice), regardless of endianness. However, we note that changing the ordering makes more of a difference if the files are little endian.

The bottom two lines compare the endianness choices holding the field ordering constant. Since the lines are mostly below zero, we in gneneral prefer the little endian files to the big endian files, and the preference is more pronounced for the big-to-small ordering.

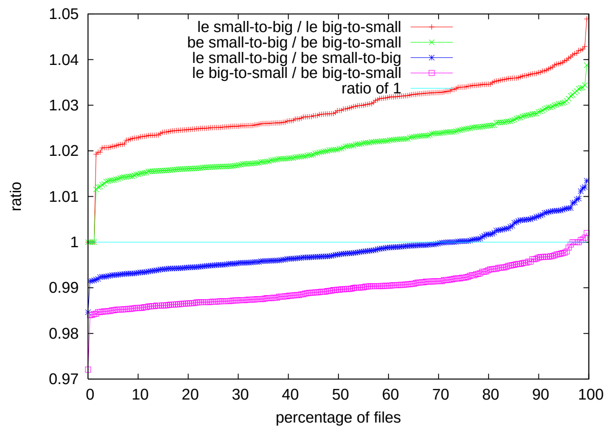


Figure 8: Reordering options for the original data files. The data can be stored in little or big endian format (le/be); it can also be stored with the small fields before the big ones or the reverse (stb/bts)

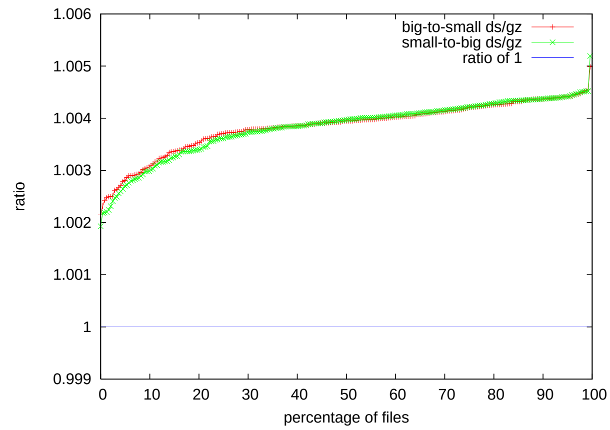


Figure 9: Overhead imposed by DataSeries. The record format is constrained to be identical to the original data. DataSeries imposes a 0.2%-0.5% overhead.

6.5.3 DataSeries file compression experiments

The first DataSeries graph looks at the overhead imposed by the DataSeries file format. For this experiment, we pruned out the 5 tiny files (4 empty, and 1 <4k compressed) in the dataset as DataSeries imposes a large overhead on very tiny files since it includes the type information in each file. In particular, for an empty file, the overhead is about 80x as at DataSeries files are 1.6k and the original files are 20 bytes. Figure 9 shows the overhead imposed by DataSeries with both the big-to-small and the small-to-big orderings for the files, and using 10^6 byte extent sizes. We can see that DataSeries imposes a 0.2%-0.5% overhead on the files if the underlying data is identical. We manually verified that a few files were in identical except for the type extent at the beginning of the file, the extent headers that occurred at the middle of the files, and the trailer at the end.

The second graph shown in figure 10 looks at the effect of the field packing (mcs) and field ordering (bts) options for DataSeries. We compare the file sizes measured by turning on each of the options individually, and then both at the same time. The results are much as would be expected; eliminating the extra padding (mcs) gives us a flat 2-3% compression improvement. Switching to big-to-small (bts) field ordering gives a 1-4%. The options combine to give us a 5-7% compression improvement over the base DataSeries case. The one oddness is the couple of points that are below 1. These come from the empty files. We changed the options to the packing by changing the XML option from pack_* to xack_* so as to minimize the change in the size of the type extent. However, having all three options specified as xack_* compresses slightly better (4 bytes) than having one or two options as pack_* and the others as xack_*, resulting in the negligably below 1 ratio.

The third graph shown in figure 11 looks at the effect of the time self-relative packing option and then combines it with the field packing (mcs) and field ordering (bts) options. Time self relative gives a much more substantial improvement in some cases – by itself it can give a 20% compression improvement. However it does not operate nearly as independently as the first two options, in particular turning on both bts and mcs packing is not much of an improvement over just mcs packing.

The fourth graph shown in figure 12 looks at just the effect of turning on the big-to-small packing option with the field compaction (mcs) and time self-relative (tsr) options already turned on. This graph shows why the benefit of big-to-small is so minor once the other options are on. The big-to-small option interferes in some cases with the time self-relative option. This figure leads to the idea that different extents could be packed using different field orderings as in DataSeries, each extent boundary could be stored in different ways.

The fifth graph shown in figure 13 looks at whether or not there is any size correlation to the improvment provided by the compression options. Visually the graph shows no significant correlation between size and compression ratio as the values are just scattered within a range.

6.5.4 Compression conclusions

The work in this section showed that the field compaction option had a small but consistent effect on the compressibility of the data. We would recommend that any extent

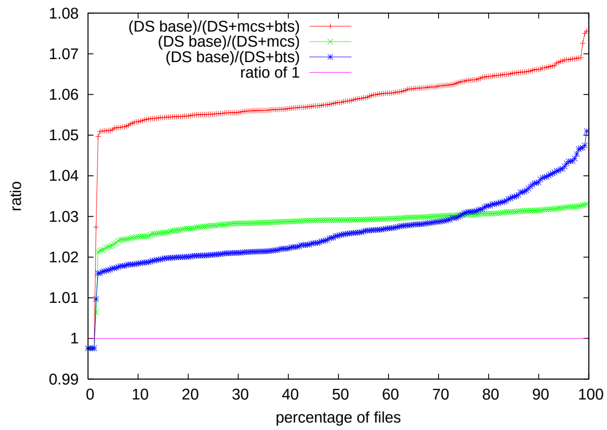


Figure 10: Turning on the field compaction (mcs) and ordering (bts) options in DataSeries. The compression options are mostly independent, so the resulting compression of turning both on is multiplicative.

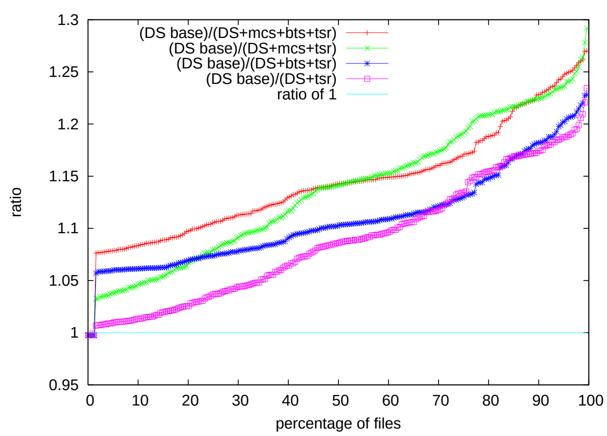


Figure 11: Turning on the time self-relative (tsr) option and then the field compaction (mcs) and ordering (bts) options in DataSeries. With time self-relative packing, the options are no longer independent, and overall time self-relative gives a much more substantial improvement than the other options.

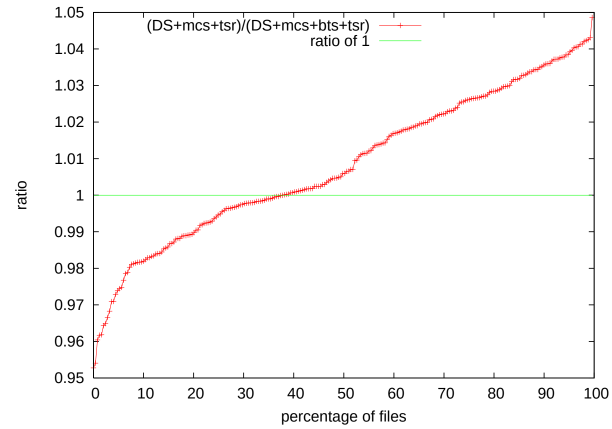


Figure 12: Comparing just the field ordering option (bts) with both the time self-relative (tsr) and field compaction (mcs) options enabled. Packing big to small is a win, but not by much.

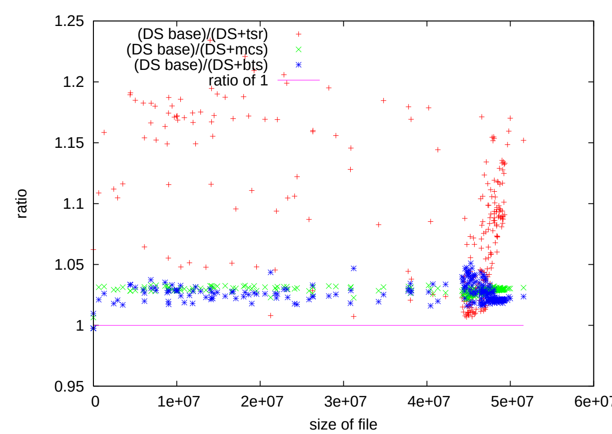


Figure 13: Determining if any of the compression options correlate with size of the file. Visually they do not appear to.

type that does not have any 8 byte fields always turn on this option. In fact, the option could be turned on in general, but it will have no effect (other than consuming a small amount of space in the extent type) if there are 8 byte fields in the record type, or if the record size is otherwise a multiple of 8.

The work in this section also shows that field ordering can have a substantial effect on the compressibility of an extent. On every extent boundary, the fields that access the extent get a chance to re-calculate their offsets into the data. This means that for graphs like figure 12 we could in theory make it so that all files are compressed as well as just having the mcs+tsr options enabled. Indeed, just choosing a per-file field ordering that would match the best ordering for the file would improve the compression. However, we could potentially do even better as each extent in a single file could choose the best ordering. Moreover, we have only tested a subset of the possible ordering options. For this dataset, there are actually $5! * 4!$ possible orderings that don't introduce additional padding. This comes from there are 5 32-bit units (4 actual and 4 8-bit) that could have any order, and then the 4 8-bit values could be ordered in any order. We hypothesize that correlations between fields will correlate with the amount of compression that could be achieved, and hence it may be possible to work out much better field orderings without evaluating an exponential number of values.

6.5.5 Analysis performance

We re-implemented the `checklog` program that comes with the 1998 World Cup traces. That program calculates a couple of the values in multiple ways. We duplicated this calculation because we wanted the programs to be comparable, even though performing the calculation is unnecesary. When running the program we found that the DataSeries implementation ran 2.60× faster. This result was expected as the analysis performed was fairly simple, and as DataSeries used multiple CPUs for decompression, it was able to run faster. However, we also found that the implementation used about 1.04× less CPU. This was unexpected given that DataSeries was using a virtual function call for every row, was applying additional if statements to test for null fields on every access, and was having to process the extents separately. We examined the CPU utilization using oprofile and valgrind and found that the original code had four problems:

- **Default to non-optimized compilation**. The `checklog` program defaulted to non-optimized compilation. This led to a much larger initial difference measured between the programs. The numbers reported above are after this was fixed in the Makefile. DataSeries defaults to optimized builds for exactly this reason.
- **Non-integrated decompression**. The `checklog` program used a separate process to perform the decompression, this led to slightly increased system time to pass the data over the pipe.
- **Very slow byte-swapping routine**. The `checklog` program had to byte-swap every record, and the implementation used for byte-swapping was very slow. Moreover, the implementation copied the data rather than doing the byte-swapping in-place, further increasing the overhead.
- **Slow per-record fread**. The `checklog` program called fread for each record. While we didn't expect this to be slow, it turned out that the implementation is slow, and so calling fread for each record is a poor choice.

It turned out to be an artifact that caused the DataSeries analysis program to use slightly less cpu time than the `checklog` program. Examining the number of instructions showed that `checklog` used only 959 million instructions to perform the analysis, while DataSeries was taking 1,775 million instructions to perform the same analysis. Optimizing the `checklog` program so that it performs better would result in it using less CPU time than DataSeries. Parallelizing decompression would be more difficult since we could only parallelize on a per-file basis and each individual file is fairly large.

However, we did choose to examine how we could optimize the DataSeries program. The analysis indicated that there were two sources of slowdown: if-tests for possibly null fields, and virtual function calls. We modified the `wcanalysis` program to have a special version of the fields that skipped the null test, and expanded out the getExtent function so that there were no per-row virtual function calls. This led to the measurements shown in figure 5 that show a 1.6× reduction in instructions and a potential 1.1× improvement in runtime. Most of the runtime improvement came from skipping the nullable tests, so we implemented a templatized version of the extent fields so that the nullable check could be turned of if the analysis can not deal with null fields. The templatized version performed identically to the special case versions of the fields written for the direct-{int32,byte} fields use by the world cup analysis.

For consistency in our measurements, we measured the performance of the tools for the full data-set under RHEL4 using the same system our other measurements ahve been taken on, a 2.4GhZ dual-cpu, dual-core DL365 with opteron 2216 HE processors. Given our experience

measurement	user time (s)	ratio	billion instructions total	for analysis	analysis ratio
gunzip	1.92	-	n/a	n/a	-
checklog	1.02	-	3.060	0.960	-
original	1.92	1.000	5.134	1.775	1.000
direct-int32	1.83	1.049	4.784	1.424	1.247
+direct-byte	1.77	1.085	4.588	1.228	1.445
+non-virtual	1.72	1.116	4.440	1.081	1.642
gcc-4.3	-	-	-	0.875	2.028

Table 5: Measurements for the world cup analysis. gunzip and checklog were use in a pipeline. Except for the gcc-4.3 measurement which was taken on a separate debian (lenny) machine, the timings and instructions were measured on a HP nw9440 T2600 cpu running debian etch. Instruction counts were within 100,000 instructions between the two debian machines if we used gcc-4.1 on both. Ratios are relative to the original measurement in both cases; the direct-byte and no-virtual rows are cumulative optimizations with the previous rows.

above, we measured the instruction counts using valgrind, and found that `checklog` was using 800m instructions for the analysis whereas `wcanalysis` was using 1,357m instructions. Compiling with gcc-4.3, and using the non-virtual row processing would drop the instruction count to 801m, indicating that gcc-3.4 on RHEL4 has a higher abstraction penalty than gcc-4.1 and 4.3. Since this would not model the performance that would be measured on stock RHEL4, we chose to take the measurements for the full tests using gcc-3.4 compilation, templatized extent fields, and the virtual function call per row since this is the recommended implementation pattern (or at least will be once the templatized implementation is well tested).

We ran all of the programs over the full data set, and then ran them over a subset of the data that would fit in memory as we discovered that when compressed with lzo, we were being limited by the disk read rate for performing our analysis. The results are shown in figure 6. They show that we are limited in the analysis performance by the speed of the analysis, hence the switch to gcc-4.3 should bring a substantial improvement. We found that the analysis on the full dataset might have been disk limited, but established that it only was for the lzo compression algorithm, the gz algorithm compressed just barely well enough to not be disk limited. We expect that with gcc-4.3 we would be disk limited for this analysis.

7 Discussion

The original motivation for developing DataSeries was our need to store various types of I/O traces. For over a decade we used a binary data format for block level traces, but found this untenable for two reasons. Firstly, various fields in the traces had been added or deleted over the years, and worse, some had changed their meaning. This resulted in significant software engineering overhead to maintain the multiple internal record structures, and confusion on the part of those who had to write analyses. Secondly, it was not easily extensible to some of the new types of data we wished to store. As a consequence, we designed and built DataSeries, completing the first release in August of 2003.

Although we have stored many different types of information in DataSeries, very few changes to the initial version have been required. We have not had to add in any additional data types beyond the ones described in section 3, all of which were in the initial version of DataSeries. We did add the pack_scale option in a subsequent version of DataSeries, as we determined it was necessary for improving the compression of data stored in doubles. The pack_unique option, which was included in the initial version, has proven to be very useful in improving compression rates.

The vast majority of the analyses we have performed have been a scan over a time range of the data in a single extent type. We have only wanted joins in a few cases; for example, to report on the applications with the most I/O utilization by joining traces of block I/O and Unix process information. Because all of the extents are sorted in chronological order, we have been able to use a simplified variant of the sort-merge join; we maintain a short re-order buffer to allow us to process either based on the request time or the response time using a standard priority queue, and then we perform the expected merge between the two tables. The pseudo-sortedness of the extents means we can use a single-pass priority-queue+merge algorithm rather than a two pass sort+merge algorithm.

We have implemented fairly few generic operations because we have not found them to be useful. Most of our analyses are more complicated than could be easily represented in standard SQL, although they have been straight-

measurement	user	system	cpu	ratio	wall	ratio	cpu-util
gunzip,checklog	364.88	45.30	410.18	1.00	428.93	1.00	96
original-full-gz	352.35	41.46	393.81	1.04	165.03	2.60	239
template-full-gz	308.76	42.10	350.86	1.17	123.81	3.46	283
checklog-subset	184.56	19.75	204.32	1.00	208.22	1.00	98
template-subset-gz	152.30	20.67	172.97	1.18	60.63	3.43	285
template-subset-lzo	117.66	21.31	138.97	1.47	60.36	3.45	230

Table 6: Performance measurements over the full world cup dataset, or a subset of the data (files matching wc_day[12345]∗). The dramatic wall clock time improvement between the original and template version shows that we are being limited by analysis performance. This observation is repeated by the complete lack of wall clock time improvement shown by switching to the lzo decompression algorithm. Measurements using the lzo compression algorithm on the full dataset were slower because we were limited by the disk read performance, and so were not further measured. All measurements were taken with 1million byte extent sizes to match the sizes use in the compression comparison.

forward to implement in C++ as a streaming calculation with some amount of buffering (e.g., the sort-merge join described above). The two generic operations we have implemented are indexing and statistics over an expression grouped by a column.

The largest dataset we have is the NFS data. The primary extent type in this data is the common records which store information about each of the 200 billion request and reply messages. We have secondary tables that store information about each packet captured, operations that included file attributes, read and write requests and mount requests. The total dataset is about 5TB in size.

As a demonstration of the real-life performance of DataSeries, consider the following example. Utilizing a trace of NFS traffic from a LAN, we performed an analysis of the throughput effects if servers were instead accessed across a WAN. The analysis read in 45.5 GB of data (406 GB when uncompressed), and processed 7.6 billion records (each record corresponds to an NFS transaction). Using a four year old two-way 2.8 GHz Xeon server, the entire data set was processed in 11,263 wall clock seconds (about 3 hours), or roughly 675,000 rows per second performing a set of complex analyses.

As a second demonstration of real life performance we used DataSeries to build a large set of reports and graphs from LSF data. Some of our reports were similar to ones already being created through queries to a commercial relational database. Report generation in DataSeries ran over 50× faster than the database report generation. While we did not investigate precisely why the database version was slower, it appeared to come from two sources. First, DataSeries is entirely targeted at analysis, and hence runs those calculations very efficiently. Second, the desired calculations would require many SQL queries to generate the same results as the single pass DataSeries calculation, and it is likely the queries were executed sequentially for simplicity in the program generating the report.

8 Conclusions

We have described DataSeries, a data format that enables the efficient and flexible storage of structured serial data. This type of data is used in numerous applications in all areas of computing and science, and researchers have developed almost as many ways to store and process it as there are applications. Unfortunately, there are many limitations to these formats. In contrast, DataSeries offers five major advantages:

1. DataSeries improves *storage efficiency*, by incorporating compression algorithms and related techniques. Using DataSeries, we have been able to store and analyze large datasets on a significantly smaller server and storage system than is typically used with databases.

2. DataSeries provides much better *access efficiency* than other formats. This enables the timely analysis of very large datasets.

3. DataSeries is *flexible*, in that it can handle a wide variety of different types of data, multiple types of data in the same file, and is easily extensible to handle new data types without changing the format. In fact, in almost four years of use, and an increasing number of data types, including I/O traces (disk block and NFS), batch cluster logs, system call traces, performance measurements and email content, we have not had to update the format once.

4. DataSeries is *self-describing*; that is, relevant metadata is retained with the data.

5. DataSeries has an API to improve the *usability* of the format by others.

We have demonstrated these advantages by describing our experiences using DataSeries in a variety of situations and through a series of experiments designed to show how DataSeries compares to other types of solutions. Based on these results, we believe that DataSeries will be useful to other researchers who collect and analyze structured serial data. We also think that DataSeries has the potential to be used for more than just traces and measurements of computer systems. For example, data retention is becoming an increasingly important topic. Legislation such as the "Internet Stopping Adults Facilitating the Exploitation of Today's Youth (SAFETY) Act of 2007" [24] seeks to require ISPs to retain information on their subscribers for extended periods of time. In addition, many businesses are realizing a competitive advantage from collecting and analyzing a wide range of data [13]. The number of businesses utilizing data for such purposes will increase over time, as others try to narrow the gap. DataSeries can potentially assist with these emerging trends.

There are a number of ways that DataSeries could be enhanced in the future. The first method would be to increase the functionality DataSeries provides. A SQL interface would allow for the easy expression (without program development) of some types of queries and analysis. Similarly, extending the set of generic operations that are supported (e.g., sort-merge join) would make DataSeries more appealing to some. Several further performance enhancements are possible. For example, different hash algorithms could be used for compressed and uncompressed data. Similarly, fewer sanity checks could be applied (e.g., disable checksumming of uncompressed data except when repacking). In other words, DataSeries could provide more options to the user to tune it to their particular performance and coherence needs.

DataSeries is open source that can be downloaded from `http://tesla.hpl.hp.com/opensource/`

References

[1] bzip2 compression library, http://www.bzip.org/, accessed September 2007.

[2] F. Chang, J. Dean, S. Ghemawhat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, November 2006.

[3] Z. Chen, Y. Zhang, H. Scott, and B. Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *ACM SIGMETRICS*, pages 145–156, June 2005.

[4] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23(3):187–200, July 2000.

[5] A. Davies and P. Lampen. JCAMP-DX for NMR. *Applied Spectroscopy*, 47:1093–1099, 1993.

[6] http://www.research.att.com/~daytona/inuse.php, accessed September 2007.

[7] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST 2003)*, pages 203–216, San Francisco, CA, 2003. USENIX.

[8] A. Ganapathi and D. Patterson. Crash data collection: a windows case study. In *Dependable Systems and Networks*, pages 280–285, July 2005.

[9] G. Goucher and J. Matthews. The national space science data center common data format, December 1994.

[10] gzip compression library, http://www.gzip.org/, accessed September 2007.

[11] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 487–498. VLDB Endowment, 2006.

[12] E. Hoke, J. Sun, and C. Faloutsos. Intemon: Intelligent system monitoring on large clusters. In *VLDB*, pages 1239–1242, September 2006.

[13] M. Hurd and L. Nyberg. *The Value Factor: How Global Leaders Use Information for Growth and Competitive Advantage*. Bloomberg Press, 2004.

[14] http://ita.ee.lbl.gov/html/contrib/WorldCup.html, accessed July 2008.

[15] M. Ji, A. Veitch, and J. Wilkes. Seneca: remote mirroring done write. In *USENIX Technical Conference*, pages 253–268, June 2003.

[16] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the self-similar nature of ethernet traffic. *IEEE/ACM Transactions on Networking*, 2(1):1–15, February 1994.

[17] http://www.tcpdump.org/, accessed September 2007.

[18] lzf compression library, http://www.goof.com/pcg/marc/liblzf.html, accessed September 2007.

[19] lzo compression library, http://www.oberhumer.com/opensource/lzo/, accessed September 2007.

[20] N. Megiddo and D. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST 2003)*, pages 115–130, San Francisco, CA, 2003. USENIX.

[21] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *ACM SIGMETRICS*, pages 216–227, June 2006.

[22] R. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. *SIGPLAN Notices*, 28(12):1–11, 1993.

[23] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST*, pages 17–28, February 2007.

[24] Information on SAFETY Act of 2007, http://www.theregister.com/2007/02/12/congress_isp_data_retention_push/, accessed September 2007.

[25] B. Schroeder and G. A. Gibson. Disk failures in the real world: what does an MTTF of 1,000,000 hours mean to you. In *FAST*, pages 1–16, February 2007.

[26] S. Sohoni, R. Min, Z. Xu, and Y. Hu. A study of memory system performance of multimedia applications. In *ACM SIGMETRICS*, pages 206–215, June 2001.

[27] C. Soules and G. Ganger. Connections: using context to enhance file search. In *Symposium on Operating Systems Principles (SOSP)*, pages 119–132, April 2005.

[28] http://tesla.hpl.hp.com/public_software/; SRT and trace data sections; accessed September 2007.

[29] M. Stonebraker, D. 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 *International Conference on Very Large Databases (VLDB) 2005*, pages 553–564, September 2 2005.

[30] M. Uysal, A. Merchant, and G. Alvarez. Using MEMS-based storage in disk arrays. In *Conference on File and Storage Technologies (FAST)*, pages 89–102, April 2003.

[31] http://www.wintercorp.com/VLDB/2005_TopTen_Survey/2005TopTenWinners.pdf, accessed September 2007.