Evaluating DataSeries for use with a Job Centric Monitoring Service

R. Curry R. Simmonds
University of Calgary
Calgary, AB

E. Anderson M. Arlitt
HP Labs
Palo Alto, CA

Abstract—
The role of monitoring data in IT environments is changing. Once reserved primarily for debugging purposes, monitoring data is increasingly being integrated into the daily use and management of IT systems. These new uses place different requirements on the storage of and access to the monitoring data. We believe that such requirements are not met by existing data providers.

In this paper we propose using DataSeries, a generic trace format developed at HP Labs, to address these emerging requirements. We examine the data access and storage requirements of a Job Centric Monitoring (JCM) service developed at the Grid Research Centre (GRC), and explain how DataSeries could fulfill these requirements. We then explore the performance of DataSeries for both the access and storage requirements, using empirical data to help us illustrate the tradeoffs that exist. As future work the GRC intends to integrate DataSeries into the JCM service.

Keywords— **High Performance Computing, Monitoring, Data Storage, Data Compression**

I. Problem Statement

IN the daily use and operation of high performance (HPC) and grid computing environments, monitoring data can be useful in addressing many needs. For example, end users may want to know the current status of their submitted jobs; system administrators need to know the operational status of the computing resources; and organizational executives may want to understand how effectively the computing environment is being utilized. In enterprise IT environments, monitoring data can be used for many similar roles. Monitoring data will be vital in utility computing, to enable the automation of IT management tasks. Unfortunately, in many situations the data that is required to address such needs is either not (systematically) available, or not (programmatically) accessible.

In order to provide a better understanding of this high-level problem, we focus on a specific scenario - monitoring the complete lifecycle of a job in an HPC/grid environment. By combining data gathered from the batch (i.e., job) scheduler with data collected on each computational node, information can be composed that addresses the needs of numerous groups, including: end users who want to tune their job submission strategies to improve their individual response times; system administrators attempting to improve the throughput of an entire cluster; and organizational executives who need to understand how effectively provisioned resources are being utilized, in order to make allocation decisions.

Researchers at the Grid Research Centre (GRC) at the University of Calgary have developed a Job Centric Monitoring (JCM) service and portal to make such information available to and accessible by the individuals who require it [1]. Much of the information composition process has been automated, and is used within the WestGrid [3] environment. This service enables the organization and its users to work more effectively, without adding additional workload on the IT staff that supports the grid infrastructure.

A challenge for the JCM service is providing (sustainable) access to the data of interest that has been gathered from the scheduler and the computational nodes. There are multiple attributes to this challenge. First, when a request is received, the service must quickly find the relevant pieces of data, and then compose the requested information. Second, the retrieval (and composition) must be done efficiently, so that the service can support many concurrent users. Third, in order to satisfy as many requests as possible, each piece of data must be retained for (at least) its entire life span (i.e., for as long as the data is useful). Given the large volume of monitoring data that could be required in an HPC/grid environment, efficiently storing the data is very important.

The first two attributes described above are usually addressed when a monitoring infrastructure is designed. The third attribute, however, is typically paid less attention, either as a trade off against improving response times or concurrency, or because its importance is underestimated. Although the life span of a piece of data may be months or years, the provisioning of most monitoring systems today does not directly consider the data life span. Instead, (older) data may be discarded when a disk reaches capacity (or a high water mark), when a database table exceeds a specified number of rows, or when the query response time of the database violates a specified threshold.

II. Solution

Our solution to the aforementioned problem is to use DataSeries [4], an open source, generic trace format developed at HP Labs, to store the data required by the JCM service. We selected DataSeries because we believe it adequately addresses all three attributes of the problem at hand.

DataSeries was initially developed to generalize beyond the block I/O trace specific format developed and used at HP Labs. DataSeries was first used to capture block I/O and process data, followed by NFS traces, LSF information, and `sar` data. Such traces can be quite large, so compression was integrated into DataSeries. At the same time, the users of the data (HPL researchers) required the ability to quickly read selected fields within each trace. In addition, the format needed to be extensible, both to handle new fields in records as well as new data types.

In order to better understand the benefits of using DataSeries, we briefly describe how it organizes and stores data. Initially, the data is placed in an in-memory data structure called an *extent*. Each extent consists of two arrays: an array of fixed-size records (e.g., a structure for types such as integer and floating point values), and an array of variable-sized data (e.g., strings). When the extent is ready to be written to disk, the data is packed (based on type definition), checksummed, compressed, checksummed again, and then written. Several different compression algorithms can be used, depending on the desired optimization; to minimize storage space, the bz2 algorithm [5] could be used; to minimize read time, the lzo algorithm [6] could be utilized; or to minimize compression time, the lzf algorithm [7] could be applied. Reading an extent from disk involves checksumming the packed data, uncompressing the fixed and variable parts into memory, and checksumming the unpacked data. The checksums ensure the integrity of the data.

DataSeries incorporates several other optional mechanisms for improving the compression. For example, many large floating point values have mostly random lower bits. By rounding the least significant bits before packing, compression can be improved. DataSeries can also pack values relative to other values. For example, the cpu utilization value may be similar from one interval to the next. By storing relative rather than absolute values, more zeros occur in the fields, which also leads to better compression. Initial tests at HP Labs found these techniques improved compression by a factor of up to four over compression without these techniques. A third technique for improving compression is to store each unique string only once, and use a common index to retrieve the appropriate string when reading the data.

Besides the choice of compression algorithm, there are several techniques for reducing the time to retrieve values from a DataSeries file. First indexing can be used to reduce the search space. Second, the size and contents of each extent can be designed to improve retrieval times. In the next section we examine how these techniques can be used to customize DataSeries for the JCM service.

DataSeries was designed to match the performance of a system that uncompressed the files and then cast the record type. Given the data-specific compression, the performance is sometimes even better. On a mid-range computer (e.g., a two-way 2.8 GHz Xeon server), simple scan-type queries can run as fast as the extents can be uncompressed (≈100 MB/s scan rate on the uncompressed data).

DataSeries permits us to make more efficient use of the storage space devoted to monitoring data. While this alone does not guarantee that we will meet our objective of retaining each piece of data for its entire life span, it is a significant step in that direction. Until we have a better understanding of the life span of data (beyond knowing that we need to retain data longer than has traditionally been the case), we assume that providing the ability to efficiently read compressed data sufficiently addresses the life span requirement. In the next section, we evaluate how effective DataSeries is, in terms of storage space, read latency, and concurrency, for job-related data.

III. Evidence the Solution Works

Our initial motivation for considering DataSeries is that it has been used effectively within HP Labs over the past several years, for a variety of situations. The common programming interface and trace format makes it easier for researchers to create analysis tools to work with different types of data. For example, DataSeries has been used to store and analyze ≈100 billion NFS request or response records [2]. Several types of supporting tables were used in this case: read/write only tables, and tables of per-packet information ($\approx 3 - 5\times$ larger). DataSeries has also been used within HP Labs to store `sar` statistics, collected every one second, on a 500 node cluster for almost one year. Each observation included 2 four byte integer timestamps and 129 eight byte double metrics. For 280 days, the size of the raw, uncompressed data for a single node is 23.4 GB; the equivalent data stored in DataSeries is 1.1 GB.

We have also conducted an initial assessment of DataSeries for storing the node and process monitoring data required by the JCM service. The goals of our study were to understand the effect of a set of factors on the average query time. Section III-A describes the data used by the service, and lists some of the issues that we evaluate. Section III-B discusses our experimental design and methodology. Section III-C provides our results. Section III-D summarizes what we have learned from this initial study.

A. Overview

In this preliminary investigation, we examine the ability of DataSeries to provide responsive access to two types of data required by the JCM service: per node metrics such as cpu utilization, and per-process metrics, such as cpu time used by the process. For this study we used a day-long data set collected from a 128 node cluster in WestGrid. The measurements in this trace were recorded every 15 seconds. This data set was originally stored as a set of text files, totaling 320 MB in size (uncompressed), and 61 MB when compressed with `gzip`. In total there were 2,611,107 records in the data set.

The JCM service requires (near) real-time access to the node and process data. Ideally, we would like the service to have access to the data within a few minutes of when the data was recorded on each node (shorter times are better). DataSeries was designed for long-term storage and offline analysis of data. Currently, DataSeries does not support simultaneous reading and writing of the same DataSeries file. As a result, we would need to close a file for writing before we can begin to read from it. We therefore need to vary the duration of time for which data is stored in individual DataSeries files, in order to control how quickly the data becomes available for the JCM service (or other services with similar real-time needs).

There are numerous tradeoffs to consider. For example, reducing the time per file will make the data available more quickly, but it will also increase the number of files and extents, which can reduce compression of the overall data set. The increased number of files or extents will increase the index time, which can also affect the query time. Restricting the number of nodes per file or extent, or altering the size of each extent are other parameters that can be adjusted. In the remainder of this section, we examine the effects of such parameters on the query times and the level of compression achieved.

Factor	Level
Time Per File (TPF)	60,300,900,3600,86400 seconds
Extent Size (ES)	1,16,64, 256 KB; 1, 2 MB; 1 GB
Nodes Per Extent (NPE)	1, 16, 64, 128
Nodes Per File (NPF)	1, 16, 64, 128

TABLE I
FACTORS AND LEVELS FOR OUR EXPERIMENTS

B. Experimental Design and Methodology

For our evaluation we consider the factors and levels that are shown in Table I.

The first factor we consider is Time Per File (TPF). This parameter determines the duration of time represented in each DataSeries file. We vary this from 60 seconds (e.g., 12:00-12:01am, 12:01-12:02am, etc.) to 86,400 seconds (i.e., the entire day in one file).

A second factor we consider is the maximum size of an extent (ES). Once an extent reaches this size, it is compressed and written to disk, and a new extent is opened (if needed). We vary this setting from 1 KB (very small, and likely to negatively impact compression) to 1 GB (very large, likely to improve compression but affect access time).

The Nodes Per Extent (NPE) factor represents the number of nodes stored in each extent in a DataSeries file. $NPE = 1$ means the data for each distinct node is stored in a separate extent, while $NPE = 128$ means the data from all nodes is stored in the same extent.

The Nodes Per File (NPF) factor determines the number of nodes that data is stored for in each DataSeries file. With $NPF = 1$, the observations for each unique node are stored in separate files. For $NPF = 128$, observations for all nodes are stored in the same file. We consider this factor as it may be necessary use multiple files when aggregating data for a large cluster, when timely access to the data is needed.

We conduct each experiment in three steps.

1. **Refactor Original Data Set.** The initial (text) data set has $NPF = 128$ and $TPF = 1800$. This step creates a new set of compressed (using gzip) flat text files with the specified NPF and TPF levels. In a production environment, this step could be eliminated by recording the original data set in DataSeries using the desired level settings for each factor.
2. **Convert Refactored Files in DataSeries.** Each flat file from step 1 is mapped 1:1 to a DataSeries file. The NPE and ES levels are set in this stage. The resulting set of DataSeries files has the four factor settings desired. This step could also be eliminated in a production environment.
3. **Measure the time to perform a set of queries.** In this step we perform a predetermined set of queries against the set of DataSeries files created in step 2, and measure the average response time. For our preliminary evaluation, the set of queries is based on the jobs whose execution occurred during the data collection period. For each job, two queries are generated, one for node data and one for process data. The amount of data returned by a query varies, depending on the length of the job, the number of nodes it used, the number of processes it spawned, and the amount of overlap with the recorded data.

C. Experimental Results

In this section we present our initial results. It is important to understand that it is the relative performance of each configuration that is significant, not the absolute query response time values. For example, the current analysis tools are written in perl; we could improve the absolute performance by re-writing the tools in C++, to directly access the DataSeries files. However, we expect that the relative performance trends would be similar.

Time Per File. Figure 1 shows the results for varying the duration of time in each file. In this case the nodes per file and nodes per extent were held constant at 128. The extent size was also held constant, at 1 GB. This value was selected to be large enough that there would never be more than one extent per file. As expected, the best query time occurs when we have smaller files to search (e.g., one minute of time per file), while the best compression occurs when we include all data in a single extent (one day of time per file). With a time per file of 300 seconds (five minutes), we achieve a reasonable query response time, while achieving a level of compression almost as good as the best case.

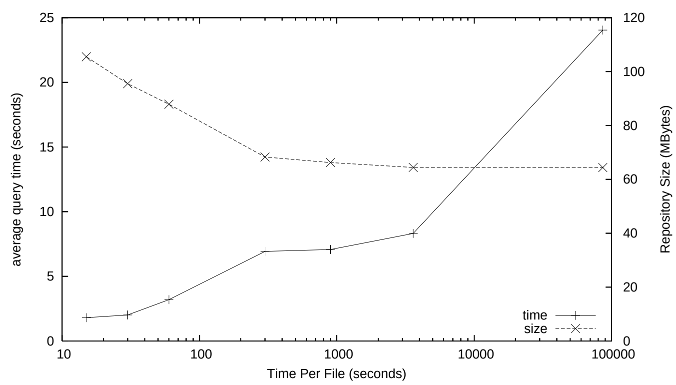

Fig. 1. Varying time per file with 128 nodes per file, 128 nodes per extent, and extent size of 1GB

Extent Size. Our next experiment examined the effect of extent size on the average query time. The results are shown in Figure 2. The results indicate that if the extent size is too small (e.g., 1 KB), the query time increases significantly. This occurs for two reasons. First, there are many small extents to open and close. Second, the lookup time increases as the index becomes much larger. This accounts for part of the query time in this case. With an extent size of 1KB, the total storage size is quite large, as the fragmented data cannot be compressed as effectively. If we make the extents very large (e.g., 2 MB), the query time is still high, as there is more data to uncompress and search through in each extent. However, as we anticipated having more data in each extent improves the compression. An extent size in the 16 KB to 256 KB range offers a reasonable tradeoff between query time and compression rate.

Nodes Per Extent. Figure 3 provides the results for changing the number of nodes per extent. In this case, the lowest average query time and best compression occurs when fewer nodes are stored in each extent. These results are intuitive. First, since most queries request data about only a subset of the nodes, less filtering of the records occurs when the extents contain only (or

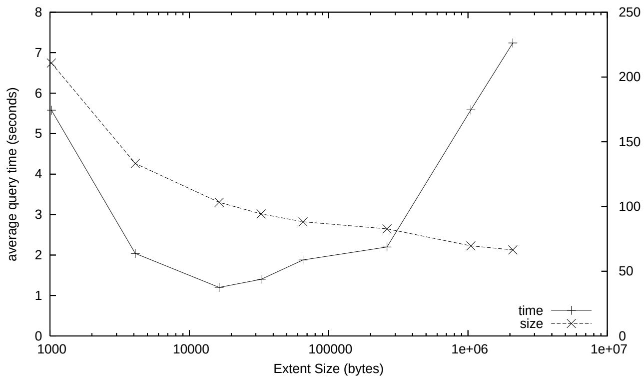

Fig. 2. Varying extent size with 128 nodes per file, 24 hours time per file, and 128 nodes per extent

primarily) records for the node(s) of interest. In addition, having data from fewer nodes in an extent increases the similarity of the data, which results in the improved compression.

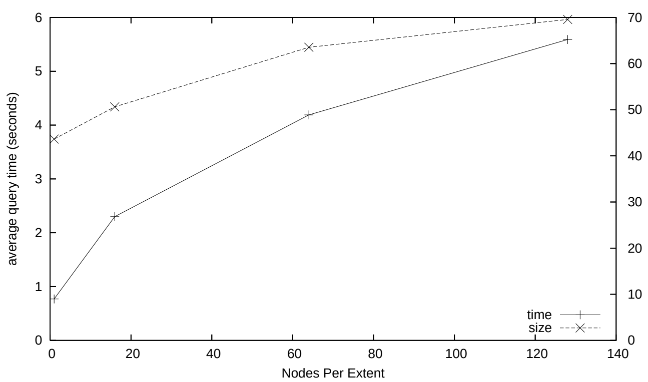

Fig. 3. Varying nodes per extent with 128 nodes per file, 24 hours time per file, and extent size of 1MB

Nodes Per File. Figure 4 shows the results for the nodes per file analysis. As expected, altering the number of nodes per file has a similar effect to varying the number of nodes per extent. Figure 5 shows that if we use one extent per node, we can include multiple nodes per file without impacting the query time or the compression. Depending on how the data is aggregated from each of the computational nodes, it may be desirable to have fewer nodes per file, or at least not have all nodes in a single file. These results are helpful for determining the level of multiplexing to use.

D. Summary

To date, DataSeries has primarily been used for long-term storage of data intended for offline analysis. As our preliminary results indicate, DataSeries offers methods to tune its performance for more real-time access, while still achieving good levels of compression. Our results showed that for our particular situation, we can improve performance and achieve good compression of the data by breaking the nodes into groups, either in separate files or extents (or combinations thereof). This approach reduces the amount of filtering that needs to occur.

These initial results suggest that DataSeries will provide us with responsive access to data within the constraints of the JCM

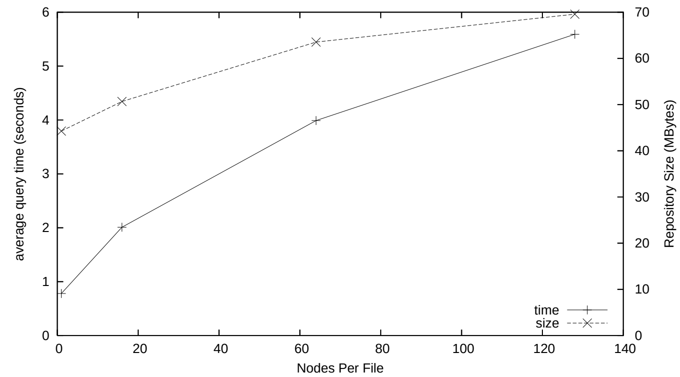

Fig. 4. Varying nodes per file with 1 extent per file, 24 hours time per file, and extent size of 1MB

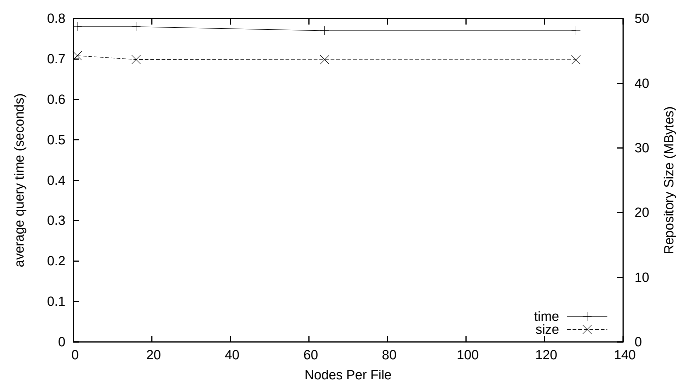

Fig. 5. Varying nodes per file with 1 extent per node, 24 hours time per file, and extent size of 1MB

service (i.e., 1 minute access latency). In particular, if the service is re-written to access the DataSeries files directly, we believe the indexing provided in DataSeries will give us quicker query times than is possible with compressed text files. The compression achieved in this configuration enables DataSeries to store data for substantially longer periods of time than could a database, which addresses one of our primary goals.

IV. Competitive Approaches

Although we believe DataSeries is a format well suited to the JCM service's needs, it certainly is not the only solution that could be used. In this section we examine some alternative solutions, and consider there advantages and disadvantages.

The initial version of the JCM service stored the data retrieved from the job scheduler and the collectors on each computational node in flat text files. The obvious advantage of this approach is its simple implementation (the raw data is already available as text). We expect the primary disadvantages compared to DataSeries (once the integration is completed) will be reduced manageability and read performance. Although text files can be compressed, the optional features of DataSeries may compress the data even further.

Another common format for storing data is XML. XML's self-describing format makes it portable across different platforms. DataSeries files are architecture independent, so they are also portable. In addition, off-the-shelf tools are available for reading and writing XML, which can reduce development time.

The primary disadvantages of XML are side effects of its self-describing format; XML is less space efficient and much less read efficient than DataSeries. The simple tests done during the development of DataSeries found compressed XML to be 1.5 to 2 times larger than the equivalent DataSeries files, and up to $10\times$ slower for accessing the data.

In many situations, databases are the preferred tool for providing data to users. The most desirable aspect of databases is their query interface, which provides simple yet powerful access to the data. Currently, a limitation of DataSeries is the lack of a query interface. Another reason databases are commonly used is they are available as off-the-shelf products. The main disadvantages of databases are they do not use storage space as efficiently as other mechanisms, they may not scale cost-effectively, and they may not be portable. DataSeries does not suffer from these characteristics.

NCSA has developed the Hierarchical Data Format for storing scientific data [8]. We are not familiar with any work that uses HDF to store monitoring data.

There are also alternative approaches that we could attempt. For example, the most recently collected data could be stored in an SQL database, while older data could be archived in DataSeries. Another approach would be to store the most recently collected data in DataSeries with a short Time Per File, and to re-pack older data (e.g., once per day) into larger extents and with a more space efficient compression algorithm). Both of these approaches could reduce the time required to make current data available, and increase the duration of time that data could be retained. We may consider one or both of these approaches as we implement a permanent solution.

V. Current Status

JCM is currently used to monitor Matrix, a 128 node dual processor HP XC Opteron cluster; part of the Westgrid resource pool. Once DataSeries has been integrated into JCM, we will start using it to monitor several other Westgrid clusters.

The node and process monitoring component of JCM collects samples from each node approximately every 15 seconds. The node and process monitor component is currently in the prototype stage; it stores observations for all 128 nodes in a compressed flat text file. A new file is created every half hour. The current system uses DataSeries to back up existing data files. We have not yet made the switch to a complete DataSeries solution as we are resolving issues related to real time access to the data. When these issues are addressed the node and process monitoring tool will be modified so that it writes directly to DataSeries files rather than an intermediate compressed text file.

The main issue to be addressed before DataSeries is adopted as the primary storage format for node and process monitoring within the context of JCM, is real time access to data. In many situations, the JCM service provides access to monitoring data while a job is still executing. A short delay of 5min is probably acceptable for most clients; while an hour delay would be too long. As described in this document, this issue can be partially addressed by varying the amount of time stored in each DataSeries file. Files might need aggregation at certain intervals to keep the number of files from becoming to large and maintaining good compression.

VI. Next Steps

There are numerous ways in which we can extend this experimental work. For example, we could look at data sets from larger clusters and for longer durations of time, to determine the impact of each on the tuning of DataSeries. We will also more closely examine the compression options that DataSeries offers, to determine if we can improve the compression beyond that achieved for the raw text files while still maintaining acceptable query times.

Upon completion of (or in parallel to) the experimental work, we intend to integrate DataSeries into the JCM service, to better support the access to job-related data. Experience gained from this work will be used to address the data access requirements for other data services under development at the Grid Research Centre, such as Replay [9], and potentially with projects at HP Labs. As part of our on going collaboration, additional features could be added to DataSeries, such as a query interface or support for additional data types.

In this paper we argued that monitoring data tends to be discarded prematurely. Our current solution addresses this indirectly, as we simply use existing storage space more efficiently in order to keep data for a longer period of time. A better solution would be to quantify the life span of different data, so that a replacement policy other than FIFO could be used if storage space must be reclaimed.

Acknowledgments

References

[1] R. Curry and R. Simmonds, “Job centric cluster monitoring”, *12th International Conference on Parallel and Distributed Systems (ICPADS 2006)*, Minneapolis, MN, July 2006.

[2] Y. Zhou, T. Kelly, J. Wiener and E. Anderson, “An extended evaluation of two-phase scheduling methods for animation rendering”, *11th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2005)*, Cambridge, MA, June 2005.

[3] `http://www.westgrid.ca/`

[4] `http://tesla.hpl.hp.com/public_software`

[5] `http://www.bzip.org`

[6] `http://www.oberhumer.com/opensource/lzo/`

[7] `http://www.goof.com/pcg/marc/liblzf.html`

[8] `http://hdf.ncsa.uiuc.edu/`

[9] D. Krishnamurthy, C. Kiddle, J. Rolia, R. Simmonds, “Replay: A Model-based Service for Supporting Transparent Cluster Analysis Tools”, *the IEEE International Conference on Cluster Computing*, Barcelona, Spain, September 2006.