Pantheon: Exascale File System Search for Scientific Computing

Joseph L. Naps, Mohamed F. Mokbel, David H. C. Du

Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA
{naps, mokbel, du}@cs.umn.edu

Abstract. Modern scientific computing generates petabytes of data in billions of files that must be managed. These files are often organized, by name, in a hierarchical directory tree common to most file systems. As the scale of data has increased, this has proven to be a poor method of file organization. Recent tools have allowed for users to navigate files based on file metadata attributes to provide more meaningful organization. In order to search this metadata, it is often stored on separate metadata servers. This solution has drawbacks though due to the multi-tiered architecture of many large scale storage solutions. As data is moved between various tiers of storage and/or modified, the overhead incurred for maintaining consistency between these tiers and the metadata server becomes very large. As scientific systems continue to push towards exascale, this problem will become more pronounced. A simpler option is to bypass the overhead of the metadata server and use the metadata storage inherent to the file system. This approach currently has few tools to perform operations at a large scale though. This paper introduces the prototype for Pantheon, a file system search tool designed to use the metadata storage within the file system itself, bypassing the overhead from metadata servers. Pantheon is also designed with the scientific community's push towards exascale computing in mind. Pantheon combines hierarchical partitioning, query optimization, and indexing to perform efficient metadata searches over large scale file systems.

1 Introduction

The amount of data generated by scientific computing has grown at an extremely rapid pace. This data typically consists of experimental files that can be gigabytes in size and potentially number in the billions. Tools for managing these files are built upon the assumption of a hierarchical directory tree structure in which files are organized. Data within this tree are organized based on directory and file names. Thousands of tools, such as the POSIX API, have been developed for working with data within this hierarchical tree structure.

The POSIX API allows for navigation of this hierarchical structure by allowing users to traverse this directory tree. While the POSIX API is sufficient for directory tree navigation, its ability to search for specific files within the

directory tree is limited. Within the confines of the POSIX API, there are three basic operations that one is able to use to search a directory hierarchy for desired information: *grep*, *ls*, and *find*. Each one of these operations searches for data in their own way, and come with their own requirements and limitations. The *grep* operation performs a naïve brute force search of the contents of files within the file system. This approach presents an obvious problem, namely its lack of scalability. A single *grep* search would need to be performed over gigabytes, or even terabytes of information. As *grep* is not a realistic solution for data at current scale, it clearly will not be a solution for future scale. A better possible route is to use the POSIX operation *ls*, that is instead based on file names. The *ls* operation simply lists all files that are within a given directory. In order to facilitate a more efficient file search, scientists used *ls* in conjunction with meaningful file names. These files names would contain information such as the name of experiments, when such experiments were run, and parameters for the experiment. By using such names, along with the wild-card(*) operator, one would perform a search for desired information based on file names. This solution also had its own problems. First, this technique is dependent on a consistent application of conventions between file names. Even something such as parameters being in different orders could prevent such a search from returning the needed information. Second, as experiments grow in complexity, more parameters must be maintained, resulting in long file names that are difficult to remember and work with. The POSIX operation *find* allows navigation of files via metadata informaiton, a much more attractive file search option than either *grep* or *ls*. Metadata represents information about the file, as opposed to information within the file. Such information includes items such as file owner, file size, and time of last modification. Unfortunately, the *find* is not sufficient for the large scale searches needed for the scientific computing community.

To solve the limitations imposed by simple POSIX commands, research began to develop full featured metadata search tools at an enterprise (e.g. Google [1], Microsoft [2], Apple [3], Kazeon [4]) as well as the academic level (e.g. Spyglass [5]). These tools added to the richness of current metadata searching capabilities, but also possessed their own limitations. Enterprise solutions typically index their data by using a standard database management system. This stores all metadata information as flat rows, thus losing the information that can be inferred from the directory hierarchy itself. Also, the need for a single metadata server can cause scalability problems in large, distributed computing systems. Relationships between files based on their location in the hierarchy are lost. Spyglass [5] exploited these hierarchical structure relationships, but at the expense of losing the query optimization and indexing powers of a database management system.

In this paper, we present the prototype of the Pantheon system. Pantheon is a file system search tool designed for the large scale systems used within the scientific computing community. Pantheon combines the query optimization and indexing strategies of a database management system with the ability to exploit the relationships between files based on locality used in current file system

search. For our initial prototype, we focused on the effects of basic database style query optimization and indexing when implemented over a more tailored file system partitioning scheme. To this end, we implemented a detailed partitioning algorithm with simple query optimization and indexing built on top.

Pantheon's core is separated into three primary components: partitioning, query optimizer, and indexing. The partitioning component is responsible for separating the directory hierarchy into disjoint partitions. We present a general partitioning algorithm, but any custom algorithm may be used. Partitioning is needed in order to avoid a system-wide bottleneck. Without partitioning, all searches would be forced to go through a single set of indexes. This would create an obvious bottleneck that would severely limit the scalability of the system. Query optimization is a well known technique from database management systems [6]. The Pantheon optimizer collects statistics on a per-partition basis, and evaluates query plans using a basic cost model. This strategy selects predicates that will prune the largest number of possible files from the result. This simple technique results in a significant performance boost over picking predicates at random. The indexing component maintains B^+-Tress and hash tables, also on a per-partition basis. A single index is kept for every metadata attribute that can be searched. Taking this approach gives Pantheon two distinctive advantages. First, this indexing method ties in very well with our query optimization. Second, the use of individual indexes allows Pantheon to quickly adapt should attributes be added to the system, as could be the case with extended attributes.

The rest of the paper is organized as follows. Section 2 discusses work related to Pantheon. Section 3 gives a high level overview of the Pantheon system. Section 4 details the partitioning system used in Pantheon. Section 5 discusses the Pantheon query optimizaer and interface. Section 6 gives an overview of the indexing system used in Pantheon. Section 7 looks at the experimental evaluation of the Pantheon system. The paper is concluded with Section 8.

2 Related Work

At an enterprise level, numerous products have been developed allowing for metadata search [1–4]. At an academic level, the closest work to Pantheon is Spyglass [5]. Spyglass uses a technique known as hierarchical partitioning [7], which is based on the idea that files that are close to each other within the directory tree tend to be searched together often. Pantheon presents an algorithm that exapnds upon this idea in two primary ways. First, many modern large scale systems use storage architectures that involve multiple tiers of storage. In such systems, data is moved between multiple layers of storage in a dynamic fashion. From the standpoint of the file system, this results in sudden changes to the directory hierarchy that must be accounted for. Pantheon's partitioning algorithm is able to adapt to data being migrated into the directory hierarchy. Second, Pantheon monitors query patterns and allows for the partition structure to be changed based on changes to query loads. For indexing, Spyglass uses a single KD-Tree [8] built over each partition. This approach to indexing has

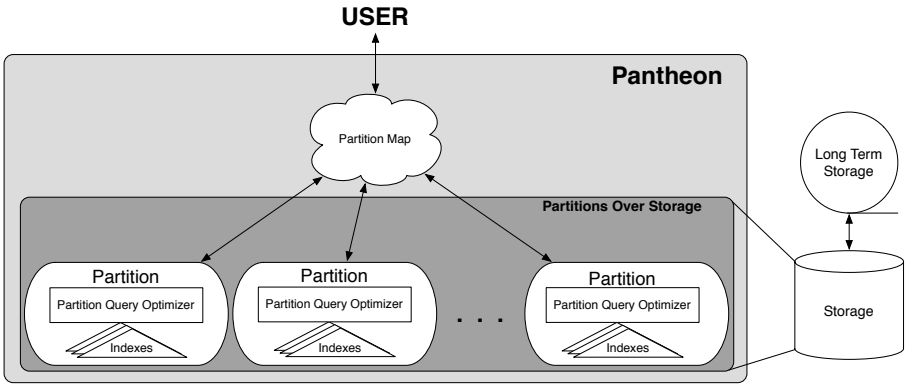


Fig. 1. Pantheon System Architecture

several drawbacks. First, using a multi-dimensional index limits the performance scalability of the system if the number of attributes being indexed were to grow very large. By splitting attributes into multiple indexes Pantheon is able to adapt in the case that additional attributes are introduced to the system more gracefully. Second, having a single index per partition means that Spyglass is unable to take advantage of the attribute distribution of a partition. Spyglass also lacks any form of a query optimizer. Using query optimization, in conjunction with a richer set of indexing structures, Pantheon is able to make intelligent decisions when dealing with queries to the file system.

3 System Architecture

Figure 1 gives the architecture for the Pantheon system. Partitions exist over the storage layer of the system, and within each partition we have the query optimizer, where distribution statistics are stored, as well as the partition indexes. The figure also gives the basic flow of a query within the Pantheon system. The query begins at the partition map. This map simply determines which partitions must be accessed in order to respond to the query. Each required partition produces the result of the query, which is then passed back to the user.

Pantheon also uses a modular design in its operations. Each of the three primary components are totally independent of one another. Only basic interfaces must remain constant. We believe this to be important for two primary reasons. First, by modularizing the design, we give scientists the ability to quickly add custom features based on their individual needs. Such examples of this could include a custom partitioning module or a different indexing structure that may be more suited to their data and querying properties. Second, a modular design will make it easier for additional components to be added to the system to adapt to new storage and architecture paradigms.

4 Partitioning

The partitioner is the heart of the Pantheon system. Without the partitioner, we would be forced to construct a single set of indexes over the data space that we wish to search. This will create a massive bottleneck that would slow down all aspects of the system, and severely limit system scalability. Similar techniques can be seen in distributed file systems [9–12].

A common pattern found in studies on metadata [5, 13–15] is that of spatial locality of metadata. Spatial locality is the general concept that files that are located close to one another in the directory hierarchy tend to have significantly more similarities in metadata values and tend to be queried together more often. This is typically the result of how files tend to be organized by users within the directory hierarchy. So, files that are owned by a user u will tend to reside close to one another in the directory tree, i.e. they will tend to reside in u*'s* home directory. When lookinging possible algorithms for partitioning our directory tree we explored works that looked into disk page based tree partitioning [16]. The general idea of our partitioning algorithm is as follows. We begin with the root of the directory tree R and proceed to find all leaves of the directory tree. From this point we place each leaf into its own partition and mark it as being processed. We then proceed up the tree processing all interior nodes such that all of their children have been marked as processed. If the interior, parent, node is able to be merged into a partition with all of its children, we merge them. In the event that there is not enough room, we create a new partition with only this interior node. Following this step, the interior node is marked as processed. This work continues all the way up the tree until we get to the root node. For more specifics about this process, refer to the pseudocode presented in Algorithm 1.

Initially, the entire directory tree must be partitioned in this manner. Since this process may take some time, it is run as a background process so that normal system function may continue.

5 Query Optimizer

Query optimization is a well studied problem in the field of database management systems. Database systems typically use a cost based model [6] to estimate optimal query plans. The gain in bringing the idea of the query optimizer from databases to file systems is significant. Query optimization research is on of the primary reasons that database systems have been able to perform so well in real world environments.

Formally, the job of Pantheon's query optimizer is a follows: Given a query Q and a series of predicates $P_1, \ldots, P_n$, the query optimizer finds a plan for evaluation of these predicates that is efficient. Using the query optimizer, indexes are used to prune the possible result set. From there a scan can be performed over this pruned data space. If done properly, this pruned data space will be significantly smaller than the original. This results in a scan that can be done very quickly. More so, this scan can be performed as a pipelined process that is done as results are being returned from the index.

Algorithm 1 Pantheon Partitioning Algorithm

```
1: Pantheon-Tree-Partition(T)
2:    Input: A tree rooted at T
3:    Output: A mapping from nodes in T to partitions
4: while There are nodes in T not yet processed do
5:    Choose a node P that is a leaf or one where all children have been processed
6:    if P is a leaf node then
7:       Create a new partition C containing node P
8:    else
9:       Let P_1, ..., P_n be the children of P
10:      Let C_1, ..., C_n be the partitions that contain P_1, ..., P_n.
11:      if Node P and the contents of the partitions C_1, ..., C_n can be merged into
         a single partition then
12:         Merge P and the contents of C_1, ..., C_n into a new partition C, discarding
            C_1, ..., C_n.
13:      else
14:         Create a new partition C containing only P
15:      end if
16:   end if
17: end while
```

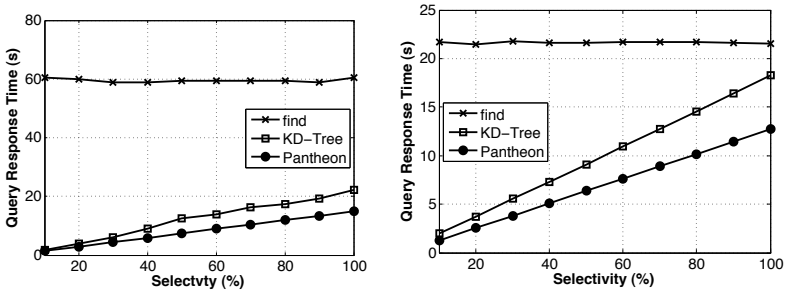
For Pantheon's query optimization, the decisions is based primarily on the selectivity of a given predicate. The selectivity represents how effective that predicate is at reducing the overall data set. A predicate will low selectivity percentage will prune more extraneous items, while a predicate with high selectivity percentage will prune less such values. This distinction is important, as not choosing the proper index when evaluation a query can lead to a significant decrease in query response time.

To track the selectivity, we need to keep basic statistics about the files on a per partition basis. For each attribute within a partition, we construct a histogram. Given an input value, these histograms quickly return an estimate as to the percentage of values that will satisfy that query.

6 Indexing

The initial indexing implementation uses simple and well known indexing structures as a baseline evaluator. We use multiple single dimensional indexes over the attributes in conjunction with query optimization. In the event that a new attribute is added to the file system, we simply construct an index over that new attribute, and continue operation as normal.

The metadata attributes being indexed are those typically found in a standard file system including: file mode, file owner, file group, time of last access, time of last modification, time of last status change, and file size. These attributes are then separated into two groups. One group represents those attributes for which range queries make sense. This includes all of the time attributes as well as file size. The remaining attributes are those where only equivalence queries make



(a) Nondistributed (b) Distributed

Fig. 2. Query Response Time vs Selectivity

sense. These include file mode, owner, and group. Each attribute that has been deemed an equivalence attribute is indexing using hash table. Each attribute that will be searched over a range is indexed using a B^+-Tree. These indexes were chosen due to the fact that each handle their own respective query types very well.

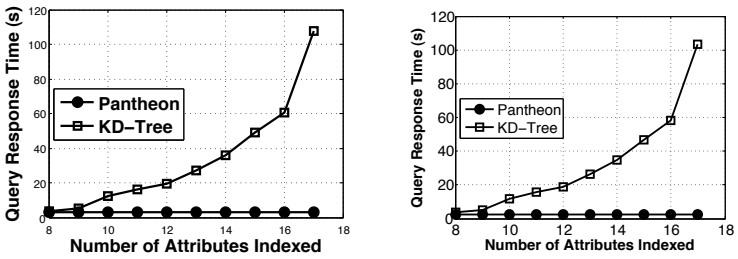
7 Experimental Evaluation

Experimental evaluation is meant to provide us a baseline for which future work may be compared. There are two other techniques that we test Pantheon against. The first is the POSIX operation *find*. This is simply to show Pantheon's viability over the naïve method. The second is testing Pantheon's processing over that of a KD-Tree. This is the indexing used by the Spyglass system, and provides a good competitor to examine Pantheon's strengths and weaknesses.

Pantheon is implemented as a FUSE module [17] within a Linux environment over an ext4 file system. For the default partition cap size we used 100,000. This was the same cap used in [5] and we see no reason to change this for experiments where partition size is held constant. The default selectivity used is 20% unless otherwise noted. The default number of attributes indexed was 8.

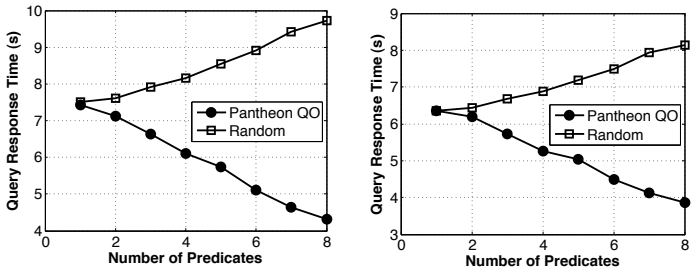
Experimentation was done over two different system configurations. The first of which is refered to as the nondistributed configuration. This was done on a single node system consisting of a dual-core 3 GHz Pentium 4 with 3.4 GB of RAM. The distributed tests were done on a 128 node cluster. Each node in the cluster consisted of two processors at 2.6 GHz with 4 GB of RAM.

In Figure 2 we see the effect on query response time when we vary the selectivity of a single query predicate. First, It shows that *find* is not any competition to Pantheon. As such, it will not be considered in future experiments. Second, it shows that using selectivity as a metric for the query optimizer is a good idea. In both cases we see that if Pantheon evaluates based on the most selective index, there is an improvement in query response time over that of a KD-Tree.



(a) Nondistributed (b) Distributed

Fig. 3. Query Response Time vs Number of Attributes Indexed



(a) Nondistributed (b) Distributed

Fig. 4. Query Response Time vs Number of Predicates

Figure 3 relates query response time to the number of attributes being indexed. Here is where Pantheon shows significant improvement over a KD-Tree based approach. As the number of dimensions increases without the partition size changing, the performance of the KD-Tree suffers greatly. Due to the fact that Pantheon indexes attributes separately, it does not show any noticeable change as the number of attributes increases.

Figure 4 displays how the Pantheon query optimizer is able to improve query performance as the number of predicates increases. Here, we generated query predicates for random attributes with random values. These results strengthen the case for using query optimization. If predicates are chosen at random, we see is significant increase in the overall time needed to response to queries.

8 Conclusion

Here we have presented the foundational work for the Pantheon indexing system. Pantheon represents a combination of ideas from both file system search and database management systems. Using these ideas Pantheon plays on the strength of each of the two fields to accomplish its goal. We have shown through

experimentation that Panteon is either competitive or outperforms current file system indexing strategies. We intend to use this prototype as a test bed for future work in aspects of partitioning, query optimization, and indexing within the context of file system search.

References

1. G. Inc., "Google enterprise," http://www.google.com/enterprise.
2. M. Inc., "Enterprise search from microsoft," http://www.microsoft.com/enterprisesearch.
3. Apple, "Spotlight server: Stop searching, start finding," http://www.apple.com/server/macosx/features/spotlight.
4. Kazeon, "Kazeon: Search the enterprise," http://www.kazeon.com.
5. A. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. Miller, "Spyglass: Fast, scalable metadata search for large-scale storage systems," in *Proccedings of the 7th conference on File and storage technologies*. USENIX Association, 2009, pp. 153–166.
6. P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM, 1979, pp. 23–34.
7. S. Weil, K. Pollack, S. Brandt, and E. Miller, "Dynamic metadata management for petabyte-scale file systems," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2004, p. 4.
8. J. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.
9. J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, "The Sprite network operating system," *Computer*, vol. 21, no. 2, pp. 23–36, 1988.
10. S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, p. 320.
11. B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "NFS version 3 design and implementation," in *Proceedings of the Summer 1994 USENIX Technical Conference*, 1994, pp. 137–151.
12. J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Smith, "Andrew: A distributed personal computing environment," *Communications of the ACM*, vol. 29, no. 3, p. 201, 1986.
13. N. Agrawal, W. Bolosky, J. Douceur, and J. Lorch, "A five-year study of file-system metadata," *ACM Transactions on Storage (TOS)*, vol. 3, no. 3, p. 9, 2007.
14. J. Douceur and W. Bolosky, "A large-scale study of file-system contents," *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 1, p. 70, 1999.
15. A. Leung, S. Pasupathy, G. Goodson, and E. Miller, "Measurement and analysis of large-scale network file system workloads," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. USENIX Association, 2008, pp. 213–226.
16. A. Diwan, S. Rane, S. Seshadri, and S. Sudarshan, "Clustering techniques for minimizing external path length," in *Proceedings of the International Conference on Very Large Data Bases*. Citeseer, 1996, pp. 342–353.
17. FUSE, "File system in user space," http://fuse.sourceforge.net.