SmartStore: A New Metadata Organization Paradigm with Semantic-Awareness for Next-Generation File Systems

Yu Hua[*] Hong Jiang[†] Yifeng Zhu[‡] Dan Feng[*✉] Lei Tian[*†]

[*]Huazhong Univ. of Sci. & Tech.
Wuhan, China
{csyhua, dfeng, ltian}@hust.edu.cn

[†]Univ. of Nebraska-Lincoln
Lincoln, NE, USA
{jiang, tian}@cse.unl.edu

[‡]Univ. of Maine
Orono, ME, USA
zhu@eece.maine.edu

ABSTRACT

Existing storage systems using hierarchical directory tree do not meet scalability and functionality requirements for exponentially growing datasets and increasingly complex queries in Exabyte-level systems with billions of files. This paper proposes semantic-aware organization, called SmartStore, which exploits metadata semantics of files to judiciously aggregate correlated files into semantic-aware groups by using information retrieval tools. Decentralized design improves system scalability and reduces query latency for complex queries (range and top-k queries), which is conducive to constructing semantic-aware caching, and conventional filename-based query. SmartStore limits search scope of complex query to a single or a minimal number of semantically related groups and avoids or alleviates brute-force search in entire system. Extensive experiments using real-world traces show that SmartStore improves system scalability and reduces query latency over basic database approaches by one thousand times. To the best of our knowledge, this is the first study implementing complex queries in large-scale file systems.

1. INTRODUCTION

Fast and flexible metadata retrieving is a critical requirement in the next-generation data storage systems serving high-end computing [1]. As the storage capacity is approaching Exabytes and the number of files stored is reaching billions, directory-tree based metadata management widely deployed in conventional file systems [2, 3] can no longer meet the requirements of scalability and functionality. For the next-generation large-scale storage systems, new metadata organization schemes are desired to meet two critical goals: (1) to serve a large number of concurrent accesses with low latency and (2) to provide flexible I/O interfaces to allow users to perform advanced metadata queries, such as range and top-k queries, to further decrease query latency.

In the next-generation file systems, metadata accesses will very likely become a severe performance bottleneck as metadata-based transactions not only account for over 50% of all file system operations [4, 5] but also result in billions of pieces of metadata in directories. Given the sheer scale and complexity of the data and metadata in such systems, we must seriously ponder a few critical research problems [6, 7] such as *"How to efficiently extract useful knowledge from an ocean of data?"*, *"How to manage the enormous number of files that have multi-dimensional or increasingly higher dimensional attributes?"*, and *"How to effectively and expeditiously extract small but relevant subsets from large datasets to construct accurate and efficient data caches to facilitate high-end and complex applications?"*. We approach the above problems by first postulating the following.

- First, while a high-end or next-generation storage system can provide a Petabyte-scale or even Exabyte-scale storage capacity containing an ocean of data, what the users really want for their applications is some knowledge about the data's behavioral and structural properties. Thus, we need to deploy and organize these files according to semantic correlations of file metadata in a way that would easily expose such properties.

- Second, in real-world applications, cache-based structures have proven to be very useful in dealing with indexing among massive amounts of data. However, traditional temporal or spatial (or both) locality-aware methods alone will not be effective to construct and maintain caches in large-scale systems to contain the working datasets of complex data-intensive applications. It is thus our belief that semantic-aware caching, which leverages metadata semantic correlation and combines pre-processing and prefetching that is based on range queries (that identify files whose attributes values are within given ranges) and top-k Nearest Neighbor (NN) queries[1] (that locate k files whose attributes are closest to given values), will be sufficiently effective in reducing the working sets and increasing cache hit rates.

Although state-of-the-art research, such as Spyglass [8], reveals that around 33% of searches can be localized into a subspace by exploiting the namespace property (e.g., home or project directory), it clearly indicates that a larger portion of queries must still be answered by potentially searching the entire file system in some way. The lack of effectiveness of exploiting spatial and temporal localities alone in metadata queries lies in the fact that such kind of localities, while generally effective in representing some static properties (e.g., directory and namespace) and access patterns of files, fail to capture higher dimensions of localities and correlations that are essential for complex queries. For example, after installing or updating software, a system administrator may hope to track and

[1]Given a clear context in the paper, we will simply use top-k queries in place of top-k NN queries.

find the changed files, which exist in both system and user directories, to ward off malicious operations. In this case, simple temporal (e.g., access history) or spatial locality (e.g., directory or namespace) alone may not efficiently help identify all affected files, because such requests for a complex query (range or top-k query) in turn need to check multi-dimensional attributes.

In a small-scale storage system, conventional directory-tree based design and I/O interfaces may support these complex queries through exhaustive or brute-force searches. However, in an Exabyte-scale storage system, complex queries need to be judiciously supported in a scalable way since exhaustive searches can result in prohibitively high overheads. Bigtable [9] uses a static three-level B$^+$-tree-like hierarchy to store tablet location information, but is unable to carry out and optimize complex queries as it relies on user selection and does not consider multiple replicas of the same data. Furthermore, the inherent performance bottleneck imposed by the directory-tree structure in conventional file system design can become unacceptably severe in an Exabyte-scale system. Thus, we propose to leverage semantic correlation of file metadata, which exploits higher-dimensional static and dynamic attributes, or higher-dimensional localities than the simple temporal or spatial locality utilized in existing approaches.

Semantic correlation [10] comes from the exploitation of high-dimensional attributes of metadata. To put things in perspective, linear brute-force approach uses 0-dimensional correlation while spatial/temporal locality approaches, such as Nexus [11] and Spyglass [8], use 1-dimensional correlation, which can be considered as special cases of our proposed approach that considers higher dimensional correlation. The main benefit of using semantic correlation is the ability to significantly narrow the search space and improve system performance.

1.1 Semantic Correlation

Semantic correlation extends conventional temporal and spatial locality and can be defined within a multi-dimensional attribute space as a quantitative measure. Assuming that a group $G_i(1 \leq i \leq t)$ from $t \geq 1$ groups contains a file f_j, semantic correlation can be measured by the minimum of $\sum_{i=1}^{t} \sum_{f_j \in G_i} (f_j - C_i)^2$ where C_i is the centroid of group G_i, i.e., the average values of D-dimensional attributes. The value of $(f_j - C_i)^2$ represents the Euclidean distance in the D-dimensional attribute space. Since the computational costs for all attributes are unacceptably high in practice, we use a simple but effective semantic tool, i.e., Latent Semantic Indexing (LSI) [12, 13] to generate semantically correlated groups as shown in Section 3.

The notion of semantic correlation has been used in many systems designs, optimizations and real-world applications. In what follows we list some examples from recent studies by other researchers and by our group, as well as our preliminary experimental results, to evidence the strong presence and effective use of semantic correlation of file metadata.

The semantic correlation widely existing in real systems has been observed and studied by a sizeable body of published work. Spyglass [8] reports that the locality ratios are below 1% in many given traces, meaning that correlated files are contained in less than 1% of the directory space. Filecules [14] reveals the existence of file grouping by examining a large set of real traces where 45% requests from all 11,568,086 requests visit only 6.5% files from all 65,536 files that are sorted by file popularity. Measurement of large-scale network file system workloads [15] further verifies that fewer than 1% clients issue 50% file requests and over 60% re-open operations take place within one minute.

Semantic correlation can be exploited to optimize system performance. Our research group has proposed metadata prefetching algorithms, Nexus [11] and FARMER [16], in which both file access sequences and semantic attributes are considered in the evaluation of the correlation among files to improve file metadata prefetching performance. The probability of inter-file access is found to be up to 80% when considering four typical file system traces. Our preliminary results based on these and the *HP* [17], *MSN* [18], and *EECS* [19] traces further show that exploiting semantic correlation of multi-dimensional attributes can help prune up to 99.9% search space [20].

Therefore, in this paper we proposed a novel decentralized semantic-aware metadata organization, called *SmartStore* [21], to effectively exploit semantic correlation to enable efficient complex queries for users and to improve system performance in real-world applications. Examples of the SmartStore applications include the following.

From a user's viewpoint, range queries can help answer questions like "*Which experiments did I run yesterday that took less than 30 minutes and generated files larger than 2.6GB?*"; whereas top-k queries may answer questions like "*I can not accurately remember a previously created file but I know that its file size is around 300MB and it was last visited around Jan.1, 2008. Can the system show 10 files that are closest to this description?*".

From a system's point of view, SmartStore may help optimize storage system designs such as de-duplication, caching and prefetching. Data de-duplication [22, 23] aims to effectively and efficiently remove redundant data and compress data into a highly compact form for the purpose of data backup and archiving. One of the key problems is how to identify multiple copies of the same contents while avoiding linear brute-force search within the entire file system. SmartStore can help identify the duplicate copies that often exhibit similar or approximate multi-dimensional attributes, such as file size and created time. SmartStore exploits the semantic correlations existing in the multi-dimensional attributes of file metadata and efficiently organizes them into the same or adjacent groups where duplicate copies can be placed together with high probability to narrow the search space and further facilitate fast identification.

On the other hand, caching [24] and prefetching [25] are widely used in storage systems to improve I/O performance by exploiting spatial or temporal access locality. However, their performance in terms of hit rate varies largely from application to application and heavily depends on the analysis of access history. SmartStore can help quickly identify correlated files that may be visited in the near future and can be prefetched in advance to improve hit rate. Taking top-k query as an example, when a file is visited, we can execute a top-k query to find its k most correlated files to be prefetched. In SmartStore, both top-k and range queries can be completed within zero or a minimal number of hops since correlated files are aggregated within the same or adjacent groups to improve cache accuracy as shown in Section 5.3.

1.2 SmartStore's Contributions

This paper makes the following key contributions.

- **Decentralized semantic-aware organization scheme of file system metadata**: SmartStore is designed to support complex query services and improve system performance by judiciously exploiting semantic correlation of file metadata and effectively utilizing semantic analysis tools, i.e., Latent Semantic Indexing (LSI) [13]. The new design is different from the conventional hierarchical architecture of file systems based on a directory-tree data structure in that it removes the latter's inherent performance bottleneck and thus can avoid its disadvantages in terms of file organization and query efficiency. Additionally and importantly, SmartStore is able to provide the existing services of conventional file systems while supporting new complex query services with high reliability and scalability. Our experimental results based on a SmartStore prototype implementation show that its complex query performance is more than one thousand times higher and its space overhead is 20 times smaller than current database methods with a very small false probability.

- **Multi-query services**: To the best of our knowledge, this is the first study to design and implement a storage architecture to support complex queries, such as range and top-k queries, within the context of ultra-large-scale distributed file systems. More specifically, our SmartStore can support three query interfaces for point, range and top-k queries. Conventional query schemes in small-scale file systems are often concerned with filename-based queries that will soon be rendered inefficient and ineffective in next-generation large-scale distributed file systems. The complex queries will serve as an important portal or browser, like the web or web browser for Internet and city map for a tourist, for query services in an ocean of files. Our study is a first attempt at providing support for complex queries directly at the file system level.

The rest of the paper is organized as follows. Section 2 describes the SmartStore system design. Section 3 presents details of design and implementation. Section 4 discusses some key issues. Section 5 presents the extensive experimental results. Sections 6 presents related work. Section 7 concludes the paper.

2. SMARTSTORE SYSTEM

The basic idea behind SmartStore is that files are grouped and stored according to their metadata semantics, instead of directory namespace, as shown in Figure 1 that compares the two schemes.

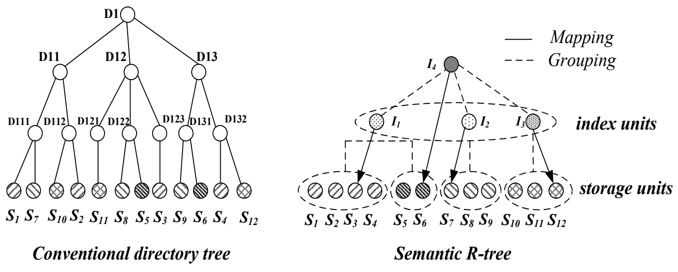

D1
D11
D12
D13
D111
D112
D121
D122
D123
D131
D132
S_1 S_7 S_{10} S_2 S_{11} S_8 S_5 S_3 S_9 S_6 S_4 S_{12}
Conventional directory tree
Mapping
Grouping
I_4
I_1
I_2
I_3
index units
storage units
S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{10} S_{11} S_{12}
Semantic R-tree

Figure 1: Comparisons with conventional file system.

This is motivated by the observation that metadata semantics can guide the aggregation of highly correlated files into groups that in turn have higher probability of satisfying complex query requests, judiciously matching the access pattern of locality. Thus, query and other relevant operations can be completed within one or a small number of such groups, where one group may include several storage nodes, other than linearly searching via brute-force on almost all storage nodes in a directory namespace approach. On the other hand, the semantic grouping can also improve system scalability and avoid access bottlenecks and single-point failures since it renders the metadata organization fully decentralized whereby most operations, such as insertion/deletion and queries, can be executed within a given group.

We further present the overview of the proposed SmartStore system and its main components respectively from user and system views with automatic configuration to match query patterns.

2.1 Overview

A semantic R-tree as shown on the right of Figure 1 is evolved from classical R-tree [26] and consists of index units (i.e., non-leaf nodes) containing location and mapping information and storage units (i.e., leaf nodes) containing file metadata, both of which are hosted on a collection of storage servers. One or more R-trees may be used to represent the same set of metadata to match query patterns effectively. SmartStore supports complex queries, including range and top-k queries, in addition to simple point query. Figure 2 shows a logical diagram of SmartStore that provides multi-query services for users while organizes metadata to enhance system performance by using decentralized semantic R-tree structures.

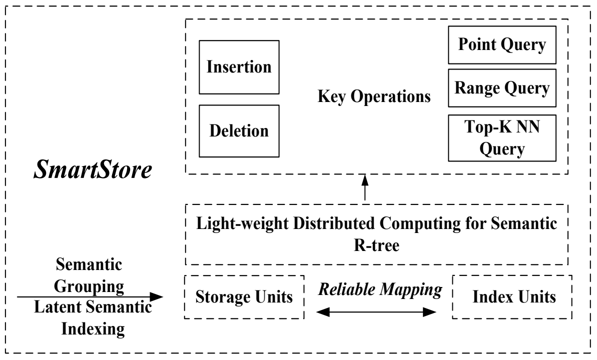


Figure 2: SmartStore system diagram.

SmartStore has three key functional components: 1) the grouping component that classifies metadata into storage and index units based on the LSI semantic analysis; 2) the construction component that iteratively builds semantic R-trees in a distributed environment; 3) the service component that supports insertion, deletion in R-trees and multi-query services. Details of these and other components of SmartStore are given in Section 3.

2.2 User View

A query in SmartStore works as follows. Initially, a user sends a query randomly to a storage unit, i.e., a leaf node of semantic R-tree. The chosen storage unit, called *home* unit for this request, then retrieves semantic R-tree nodes by using either an on-line multicast-based approach or an off-line pre-computation-based approach to locating the corresponding R-tree node. Specifically, for a point query, the home unit checks Bloom filters [27] stored locally in a way similar to the group-based hierarchical Bloom-filter array approach [28] and, for a complex query, the home unit checks the Minimum Bounding Rectangles (MBR) [26] to determine the membership of queried file within checked servers. An MBR represents the minimal approximation of the enclosed data set by using multi-dimensional intervals of the attribute space, showing the lower and the upper bounds of each dimension. After obtaining query results, the home unit returns them to the user.

2.3 System View

The most critical component in SmartStore is semantic grouping, which efficiently exploits metadata semantics, such as file physical and behavioral attributes, to classify files into groups iteratively. These attributes exhibit different characteristics. For example, attributes such as access frequency, file size, volume of “read" and “write" operations are changed frequently, while some other attributes, such as filename and creation time, often remain unchanged. SmartStore identifies the correlations between different files by examining these and other attributes, and then places strongly correlated files into groups. All groups are then organized into a semantic R-tree. These groups may reside in multiple metadata servers. By grouping correlated metadata, SmartStore exploits their affinity to boost the performance of both point query and complex queries.

Figure 3 shows the basic steps in constructing a semantic R-tree. Each metadata server is a leaf node in our semantic R-tree and can also potentially hold multiple non-leaf nodes of the R-tree. In the rest of the paper, we refer to the semantic R-tree leaf nodes as *storage units* and the non-leaf nodes as *index units*.

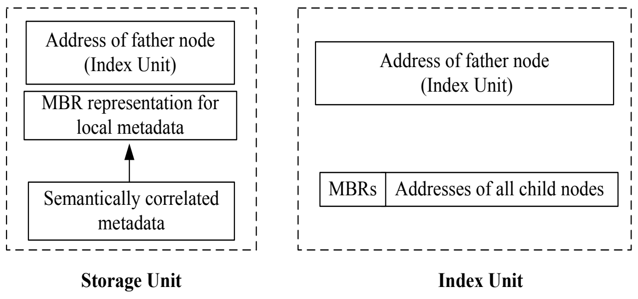

Address of father node (Index Unit)
MBR representation for local metadata
Semantically correlated metadata
Storage Unit
Address of father node (Index Unit)
MBRs
Addresses of all child nodes
Index Unit

Figure 3: Storage and index units.

2.4 Automatic Configuration to Match Query Patterns

The objective of the semantic R-tree constructed by examining the semantic correlation of metadata attributes is to match the patterns of complex queries from users. Unfortunately, in real-world applications, the queried attributes will likely exhibit an unpredictable characteristics, meaning that a query request may probe an arbitrary d-dimensional $(1 \leq d \leq D)$ subset of D-dimensional metadata attributes. For example, we can construct a semantic R-tree by leveraging three attributes, i.e., *file size, creation time and last modification time*, and then queries may search files according to their (*file size*), (*file size & creation time*), or other combinations of these three attributes. Although using a single semantic R-tree can eventually lead to the queried files, the system performance can be greatly reduced as a result of more frequently invoking the brute-force-like approach after each failed R-tree search. The main reason is that a single semantic R-tree representing three attributes may not work efficiently if queries are generated in an unpredictable way.

In order to efficiently support complex queries with unpredictable attributes, we develop an *automatic configuration* technique to adaptively construct one or more semantic R-trees to improve query accuracy and efficiency. More R-trees with each being associated with a different combination of multi-dimensional attributes provide much better query performance, but require more storage space. The automatic configuration technique thus must optimize the tradeoff between storage space and query performance. Our basic idea is to configure one or more semantic R-trees to adaptively satisfy complex queries associated with an arbitrary subset of attributes.

Assume that D is the maximum number of attributes in a given file system. The automatic configuration first constructs a semantic R-tree according to the available D-dimensional attributes to group file metadata, and counts the number of index units, $NO(I_D)$, generated in this R-tree. It then constructs another semantic R-tree using a subset (i.e., d attributes) and records the number of generated index units, $NO(I_d)$. When the difference in the number of index units between the two semantic R-trees, $|NO(I_D) - NO(I_d)|$, is larger than some pre-determined threshold, we conjecture that these two semantic R-trees are sufficiently different, and thus are saved to serve future queries. Otherwise, the R-tree constructed from d attributes will be deleted. We repeat the above operations on all subsets of available attributes to configure one or more semantic R-trees to accurately cover future query patterns. For a future query, SmartStore will obtain query results from the semantic R-tree that has the same or similar attributes. Although the cost of automatic configuration seems to be relatively high, we use the number of index units as an indicator to constrain the costs. Some subsets of available attributes may produce the same or approximate (by checking their difference) semantic R-trees and redundant R-trees can be deleted. In addition, the configuration operation occurs relatively infrequently on the entire file system.

These multiple R-trees covering different common subsets of all attributes will thus be able to serve the vast majority of queries. For the unlikely queries with attributes beyond these common subsets, the semantic R-tree constructed from D-dimensional attributes will be used to produce a superset of the queried results. The penalty is to further refine these results by either brute-force pruning or utilizing extra attributes that, however, are generally unknown in advance.

3. DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of SmartStore, including semantic grouping, system reconfigurations such as node insertion and deletion, and point and complex queries.

3.1 Semantic Grouping

3.1.1 Statement and Tool

STATEMENT 1 (**SEMANTIC GROUPING OF METADATA**). *Given file metadata with D attributes, find a subset of d attributes $(1 \leq d \leq D)$, representing special interests, and use the correlation measured in this subset to partition similar file metadata into multiple groups so that:*

- *A file in a group has a higher correlation with other files in this group than with any file outside of the group;*
- *Group sizes are approximately equal.*

Semantic grouping is an iterative process. In the first iteration, we compute the correlation between files and cluster all files whose correlations are larger than a predetermined admission constant ε_1 $(0 \leq \varepsilon_1 \leq 1)$ into groups. All groups generated in the first iteration are used as leaf nodes to construct a semantic R-tree. The composition of the selected d-dimensional attributes produces a `grouping predicate`, which serves as grouping criteria. The semantic grouping process can be recursively executed by aggregating groups in the $(i-1)$th-level into the ith-level nodes of the semantic R-tree with the correlation value $\varepsilon_i (0 \leq \varepsilon_i \leq 1, 1 \leq i \leq H)$, until reaching the root, where H is the depth of the constructed R-tree.

More than one predicate may be used to construct semantic groups. Thus, multiple semantic R-trees can be obtained and maintained concurrently in a distributed manner in a large-scale distributed file system where most files are of interests to arguably only one or a small number of applications or application environments. In other words, each of these semantic R-trees may possibly represent a different application environment or scenario. Our objective is to identify a set of predicates that optimize the query performance.

File metadata with D attributes can be represented as a D-dimensional **semantic vector** $S_a = [S_1, S_2, \cdots, S_D]$. Similarly, a point query can also be abstracted as $S_q = [S_1, S_2, \cdots, S_d]$ $(1 \leq d \leq D)$. In the semantic R-tree, each node represents all metadata that can be accessed through its children nodes. Each node can be summarized by a geometric centroid of all metadata it represents. The attributes used to form semantic vectors can be either physical ones, such as creation time and file size, or behavioral ones, such as process ID and access sequence. Our previous work [16] shows that combining physical and behavioral attributes improves the identification of file correlations to help improve cache hit rate and prefetching accuracy.

We propose to use Latent Semantic Indexing (LSI) [12, 13] as a tool to measure semantic correlation. LSI is a technique based on the Singular Value Decomposition (SVD) [29] to measure semantic correlation. SVD reduces a high-dimensional vector into a low-dimensional one by projecting the large vector into a semantic subspace. Specifically, SVD decomposes an attribute-file matrix A, whose rank is r, into the product of three matrices, i.e., $A = U\Sigma V^T$, where $U = (u_1, \ldots, u_r) \in R^{t\times r}$ and $V = (v_1, \ldots, v_r) \in R^{d\times r}$ are orthogonal, $\Sigma = diag(\sigma_1, \ldots, \sigma_r) \in R^{r\times r}$ is diagonal, and σ_i is the i-th singular value of A. V^T is the transpose of matrix V. LSI utilizes an approximate solution by representing A with a rank-p matrix to delete all but p largest singular values, i.e., $A_p = U_p \Sigma_p V_p^T$.

A metadata query for attribute i can also be represented as a semantic vector of size p, i.e., the i-th row of $U_p \in R^{t\times p}$. In this way, LSI projects a query vector $q \in R^{t\times 1}$ onto the p-dimensional semantic space in the form of $\hat{q} = U_p^T q$ or $\hat{q} = \Sigma_p^{-1} U_p^T q$. The inverse of the singular values is used to scale the vector. The similarity between semantic vectors is measured as their inner product. Due to space limitation, this paper only gives basic introduction to LSI and more details can be found in [12, 13].

While there are other tools available for grouping, such as K-means [30], we choose LSI because of its high efficiency and easy implementation. The K-means [30] algorithm exploits multi-dimensional attributes of n items to cluster them into $K(K \leq n)$ partitions. While the process of iterative refinement can minimize the total intra-cluster variance that is assumed to approximately measure the cluster, the final results' heavy dependence on the distribution of the initial set of clusters and the input parameter K may potentially lead to poor quality of the results.

The semantic grouping approach is scalable to support aggregation operations on multiple types of inputs, such as *unit vector* and *file vector*. Although the following sections mainly show how to insert/delete units and aggregate correlated units into groups, the approach is also applicable to aggregating files based on their multi-dimensional attributes that construct file vectors.

3.1.2 Basic Grouping

We first use LSI to determine semantic correlation of file metadata and group them accordingly. Next we present how to formulate and organize the groups into a semantic R-tree.

First, we calculate the correlations among these servers, each of which is represented as a leaf node (i.e., storage unit). Given N metadata nodes storing D-dimensional metadata, a semantic vector with d attributes $(1 \leq d \leq D)$ is constructed by using LSI to represent each of the N metadata nodes. Then using the semantic vectors of these N nodes as input to the LSI tool, we obtain the semantic correlation value between any two nodes, x and y, among these N nodes.

Next, we build parent nodes, i.e., the first-level non-leaf node (index unit), in the semantic R-tree. Nodes x and y are aggregated into a new group if their correlation value is larger than a predefined admission threshold ε_1. When a node has correlation values larger than ε_1 with more than one node, the one with the largest correlation value will be chosen. These groups are recursively aggregated until all of them form a single one, the root of R-tree. In the semantic R-tree, each tree node uses Minimum Bounding Rectangles (MBR) to represent all metadata that can be accessed through its children nodes.

The above procedures aggregate all metadata into a semantic R-tree. For complex queries, the query traffic is very likely bounded within one or a small number of tree nodes due to metadata semantic correlations and similarities. If each tree node is stored on a single metadata server, such query traffic is then bounded within one or a small number of metadata servers. Therefore, the proposed SmartStore can effectively avoid or minimize brute-force searches that must be used in conventional directory-based file systems for point and complex queries.

3.2 System Reconfigurations

3.2.1 Insertion

When a storage unit is inserted into a semantic group of storage units, the semantic R-tree is adaptively adjusted to balance the workload among all storage units within this group. An insertion operation involves two steps: group location and threshold adjustment. Both steps only access a small fraction of the semantic R-tree in order to avoid message flooding in the entire system.

When inserting a storage unit as a leaf node of the semantic R-tree, we need to first identify a group that is the most closely related to this unit. Semantic correlation value between this new node and a randomly chosen group is computed by using LSI analysis over their semantic vectors. If the value is larger than certain admission threshold, the group accepts the storage unit as a new member. Otherwise, the new unit will be forwarded to adjacent groups for admission checking. After a storage unit is inserted into a group, the MBR will be updated to cover the new unit.

The admission threshold is one of the key design parameter to balance load among multiple storage units within a group. It directly determines the semantic correlation, membership, and size of a semantic group. The initial value of this threshold is determined by a sampling analysis. After inserting a new storage unit into a semantic group, the threshold is dynamically adjusted to keep the semantic R-tree balanced.

3.2.2 Deletion

The deletion operation in the semantic R-tree is similar to a deletion in a conventional R-tree [26]. Deleting a given node entails adjusting the semantic correlation of that group, including the value of group vector and the multi-dimensional MBR of each group node. If a group contains too few storage units, the remaining units of this group are merged into its sibling group. When a group becomes a child node of its former grandparent in the semantic R-tree as a result of becoming the only child of its father due to group merging, its height adjustment is propagated upwardly .

3.3 On-line Query Approach

We first present on-line approaches to satisfying range, top-k and point query requests and then accelerate query operations by preprocessing.

3.3.1 Range Query

A range query is to find files satisfying multi-dimensional range constraints. A range query can be easily supported in the semantic

R-tree that contains an MBR on each tree node with a time complexity of $O(\log N)$ for N storage units. A range query request can be initially sent to any storage unit that then multicasts query messages to its father and sibling nodes in a semantic R-tree to identify correlated target nodes that contain results with high probability.

3.3.2 *Top-K Query*

A top-k query aims to identify k files with attribute values that are closest to the desired query point q. The main operations are similar to those of a range query. After a storage unit receives a query request, it first checks its father node, i.e., an index node, to identify a target node in the semantic R-tree that is most closely associated with the query point q. After checking the target node, we obtain a $MaxD$ that is used to measure the maximum distance between the query point q and all obtained results. $MaxD$ also serves as a threshold to improve the query results. Its value is updated if a better result is obtained. By multicasting query messages, the sibling nodes of the target node are further checked to verify whether the current $MaxD$ represents the smallest distance to the query point. This is to determine whether there are still better results. The top-k query results are returned when the parent node of the target node cannot find files with smaller distance than $MaxD$.

3.3.3 *Point Query*

Filename-based indexing is very popular in existing file systems and will likely remain popular in future file systems. A point query for filenames is to find some specific file, if it exists, among storage units. A simple but bandwidth-inefficient solution is to send the query request to a sequence of storage units to ascertain the existence and location of the queried file following the semantic R-tree directly. This method suffers from long delays and high bandwidth overheads.

In SmartStore, we deployed a different approach to supporting point query. Specifically, Bloom filters [27], which are space-efficient data structures for membership queries, are embedded into storage and index units to support fast filename-based query services. A Bloom filter is built for each leaf node to represent the filenames of all files whose metadata are stored locally. The Bloom filter of an index unit is obtained by the logical union operations of the Bloom filters of its child nodes, as shown in Figure 4. A filename-based query will be routed along the path on which the corresponding Bloom filters report positive hits, thus significantly reducing the search space.

A possible drawback of the above multi-query operations is that they may suffer from potentially heavy message traffic necessary to locate the most correlated nodes that contain queried files with high probabilities, since a query request is randomly directed to a storage unit that may not be correlated with the request. This drawback can be overcome by the following proposed off-line pre-processing.

3.4 Query Acceleration by Pre-processing

To further accelerate queries, we utilize a duplication approach to performing off-line pre-processing. Specifically, each storage unit locally maintains a replica of the semantic vectors of all index units to speed up the queries. Our motivation is to strike a tradeoff between accuracy and maintenance costs, as shown in Figure 5. We deploy the replicas of first-level index units, e.g., D, E, I, in storage units to obtain a good tradeoff, which is verified by our experiments presented in Section 5.5. After formulating each arrival request into a request vector based on its multi-dimensional attributes, we use the LSI tool over the request vector and semantic vectors of existing index units to check which index unit is the most closely correlated with the request. In this way we can discover the

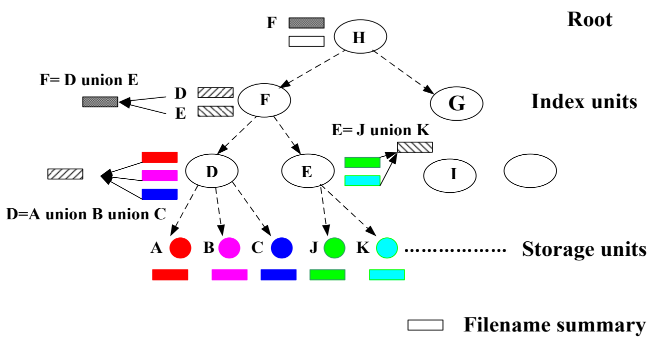


Figure 4: Bloom filters used for filename-based query.

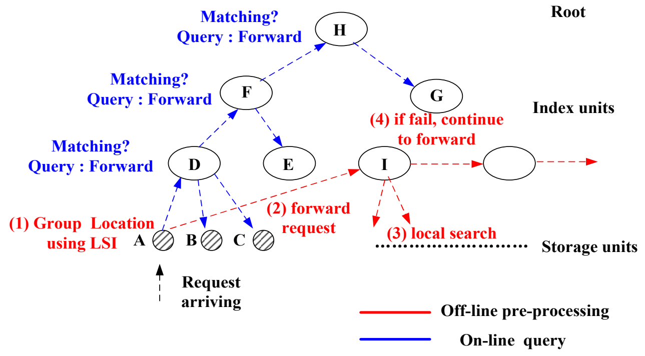


Figure 5: On-line and off-line queries.

target index unit that has the highest probability of successfully serving the request. The request is then forwarded directly to the target index unit, in which a local search is performed.

Off-line pre-processing utilizes lazy updating to deal with information staleness occurring among storage units that store the replicas of the first-level index units. When inserting or deleting files in a storage unit, its associated first-level index unit executes local update to maintain up-to-date information of storage units that it covers. When the number of changes is larger than some threshold, the index unit multicasts its latest replicas to other storage units.

4. KEY DESIGN ISSUES

This section discusses key design issues in SmartStore, including node split/merge, unit mapping and attribute updating based on versioning.

4.1 Node Split and Merge

The operations of splitting and merging nodes in semantic R-tree follow the classical algorithms in R-tree [26]. A node will be split when the number of child nodes of a parent node is larger than a predetermined threshold M. On the other hand, a node is merged with its adjacent neighbor when the number of child nodes of a parent node is smaller than another predetermined threshold m. In our design, the parameter m and M can be defined as $m \leq \frac{M}{2}$ and m can be tuned depending on the workload.

4.2 Mapping of Index Units

Since index units are stored in storage units, it is necessary and important to map the former to the latter in a way that balances the load among storage units while enhancing system reliability. Our mapping is based on a simple bottom-up approach that iteratively applies random selection and labeling operations, as shown in Figure 6 with an example of the process that maps index units to storage units. An index unit in the first level can be first randomly mapped to one of its child nodes in the R-tree (i.e., a storage unit from the covered semantic group). Each storage unit that has been mapped by an index node is labeled to avoid being mapped by another index node. After all the first-level index units have been mapped to storage units, the same mapping process is applied to the second-level index units that are mapped to the remaining storage units. This mapping process repeats iteratively until the root node of the semantic R-tree is mapped. In practice, the number of storage units is generally much larger than that of index units, as evidenced by experiments in Section 5.5, and thus each index unit can be mapped to a different storage unit.

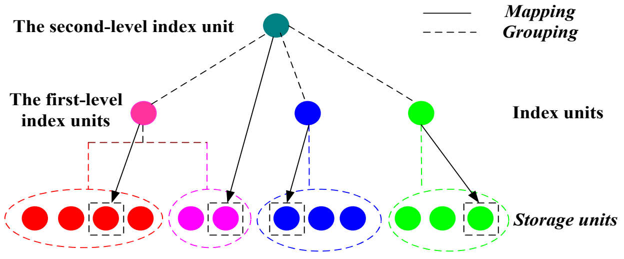


Figure 6: Mapping operations for index units.

Our semantic grouping scheme aggregates correlated metadata into semantic-aware groups that can satisfy query requests with high probability. The experimental results in Section 5 show that most of requests can obtain query results by visiting one or a very small number of groups. The root node hence will not likely become a performance bottleneck.

4.3 Multi-mapping of The Root Node

The potential single point of failure posed by the root node can be a serious threat to system reliability. Thus, we utilize a multi-mapping approach to enhancing system reliability through redundancy, by allowing the root node to be mapped to multiple storage units. In this multi-mapping of the root node, the root is mapped to a storage unit in each group of the storage units that cover a different subtree of the semantic R-tree, so that the root can be found within each of the subtrees.

Since each parent node in the semantic R-tree maintains an MBR to cover all child nodes while the root keeps the attribute bounds of files of the entire system (or application environment), a change on a file or metadata will not necessarily lead to an update on the root node representation, unless it results in a new attribute value that falls outside of any attribute bound maintained by the root. Thus, most changes to metadata in a storage unit will not likely lead to an update on the root node, which significantly reduces the cost of maintaining consistency among the multiple replicas of the root node that needs to multicast changes to the replicas in other nodes.

Mapping the root node to all semantic groups at a certain level of the semantic R-tree facilitates fast query services and improves system reliability. It can help speed up the query services by quickly answering query requests for non-existing files through checking the root to determine if the query range falls outside of the root range.

4.4 Consistency Guarantee via Versioning

SmartStore uses a multi-replica technique to support parallel and distributed indexing, which can potentially lead to information staleness and inconsistency between the original and replicated information for lack of immediately updating. SmartStore provides consistency guarantee among multiple replicas by utilizing a versioning technique that can efficiently aggregate incremental index updates. A newly created version attached to its correlated replica temporarily contains aggregated real-time changes that have not been directly updated in the original replicas. This method eliminates many small, random and frequent visits to the index and has been widely used in most versioning file systems [8, 9, 31].

In order to maintain semantic correlation and locality, SmartStore creates versions for every group, represented as the first-level index unit that has been replicated to other index units. At time t_0, SmartStore sends the replicas of the original index units to others and from t_{i-1} to t_i, updates are aggregated into the t_i-th version that is attached to its correlated index unit. These updates include insertion, deletion and modification of file metadata, which are appropriately labeled in the versions. In order to adapt to the system changes, SmartStore allows the groups to have different numbers and sizes of attached versions.

Versioning may introduce extra overhead due to the need to check on the attached versions in addition to the original information when executing a query. However, since the versions only maintain changes that require small storage overheads and can be fully stored in memory, the extra latency of searching is usually small. In practice, we propose to roll the version changes backwards, rather than forwards as in Spyglass [8], and a query first checks the original information and then its versions from t_i to t_0. The direct benefit of checking backwards is to timely obtain most recent changes since version t_i usually contains newer information than version t_{i-1}.

SmartStore removes attached versions when reconfiguring index units. The frequency of reconfiguration depends on the user requirements and environment constraints. Removing versions entails two operations. We first apply the changes of a version into its attached original index unit that will be updated according to these changes in the attached versions, such as inserting, deleting or modifying file metadata. On the other hand, the version is also multicast to other remote index units that have stored the replica of original index unit, and then these remote index units carry out the similar operations for local updating. Since the attached versions only need to maintain changes of file metadata and maintain small size, SmartStore may multicast them as replicas to other remote servers to guarantee information consistency while requiring not too much bandwidth to transmit small-size changes as shown in Section 5.6.

5. PERFORMANCE EVALUATION

This section evaluates SmartStore through its prototype by using representative large file system-level traces, including *HP* [17], *MSN* [18], and *EECS* [19]. We compare SmartStore against two baseline systems that use database techniques. The evaluation metrics considered are query accuracy, query latency and communication overhead. Due to space limitation, additional performance evaluation results are omitted but can be found in our technical report [20] and work-in-progress report [21].

5.1 Prototype Implementation

The SmartStore prototype is implemented in Linux and our experiments are conducted on a cluster of 60 storage units. Each storage unit has an Intel Core 2 Duo CPU, 2GB memory, and high-speed network connections. We carry out the experiments for 30 runs each to validate the results according to the evaluation guidelines of file and storage systems [5]. The used attributes display access locality and skewed distribution especially for multi-dimensional

Table 1: Scaled-up HP .

	Original	TIF=80
request (million)	94.7	7576
active users	32	2560
user accounts	207	16560
active files (million)	0.969	77.52
total files (million)	4	320

Table 2: Scaled-up MSN .

	Original	TIF=100
# of files (million)	1.25	125
total READ (million)	3.30	330
total WRITE (million)	1.17	117
duration (hours)	6	600
total I/O (million)	4.47	447

Table 3: Scaled-up EECS .

	Original	TIF=150
total READ (million)	0.46	69
READ size (GB)	5.1	765
total WRITE (million)	0.667	100.05
WRITE size (GB)	9.1	1365
total operations (million)	4.44	666

attributes.

In order to emulate the I/O behaviors of the next-generation storage systems for which no realistic traces exist, we scaled up the existing I/O traces of current storage systems both spatially and temporally. Specifically, a trace is decomposed into sub-traces. We add a unique sub-trace ID to all files to intentionally increase the working set. The start time of all sub-traces is set to zero so that they are replayed concurrently. The chronological order among all requests within a sub-trace is faithfully preserved. The combined trace contains the same histogram of file system calls as the original one but presents a heavier workload (higher intensity). The number of sub-traces replayed concurrently is denoted as the *Trace Intensifying Factor* (TIF) as shown in Table 1, 2 and 3. Similar workload scale-up approaches have also been used in other studies [28, 32].

We compare SmartStore with two baseline systems. The first one is a popular database approach that uses a B^+ tree [33] to index each metadata attribute, denoted as DBMS that here does not take into account database optimization. The second one is a simple, non-semantic R-tree-based database approach that organizes each file based on its multi-dimensional attributes without leveraging metadata semantics, denoted as R-tree. On the other hand, each Bloom filter embedded within an R-tree node for point query is assigned 1024 bits with $k = 7$ hash functions to fit memory constraints. We select MD5 [34] as the hash function for its relatively fast implementation. The value of an attribute is hashed into 128 bits by calculating its MD5 signature, which is then divided into four 32-bit values. We set the thresholds of 10% and 5%, respectively for the automatic configuration described in Section 2.4 and lazy updating of off-line pre-processing of Section 3.4.

While filename-based point query is very popular in most file system workloads, no file system I/O traces representing requests for complex queries are publically available. In this paper, we use a synthetic approach to generating complex queries within the multi-dimensional attribute space. The key idea of synthesizing complex quires is to statistically generate random queries in a multi-dimensional space. The file static attributes and behavioral attributes are derived from the available I/O traces. More specifically, a range query is formed by points along multiple attribute dimensions and a top-k query must specify the multi-dimensional coordinate of a given point and the k value. For example, a range query aiming to find all the files that were revised between time 10:00 to 16:20, with the amount of “read" data ranging from 30MB to 50MB, and the amount of “write" data ranging from 5MB to 8MB, can be represented by two points in a 3-dimensional attribute space, i.e., (10:00, 30, 5) and (16:20, 50, 8). Similarly, a top-k query in the form of (11:20, 26.8, 65.7, 6) represents a search for the top-6 files that are closest to the description of a file that is last revised at time11:20, with the amounts of “read" and “write" data being approximately 26.8MB and 65.7MB, respectively. Therefore, it is reasonable and justifiable for us to utilize random numbers as the coordinates of queried points that are assumed to follow either the Uniform, Gauss, or Zipf distribution to comprehensively evaluate the complex query performance. Due to space limitation, we mainly present the results of the Zipf distribution.

5.2 Performance Comparisons between SmartStore and Baseline Systems

We compare the query latency between SmartStore and the two baseline systems described earlier in Section 5.1, labeled DBMS and R-tree respectively. Table 4 shows the latency comparisons of point, range and top-k queries using the *MSN* and *EECS* traces. It is clear that SmartStore not only significantly outperforms but is also much more scalable than the two database-based schemes. The reason behind this is that the former's semantic grouping is able to significantly narrow the search scope, while DBMS must check each B^+-tree index for each attribute, resulting in linear brute-force search costs. Although the non-semantic R-tree approach improves over DBMS in query performance by using a multi-dimensional structure to allow parallel indexing on all attributes, its query latency is still much higher than SmartStore as it completely ignores semantic correlations.

Table 4: Query latency (in second) comparisons of SmartStore, R-tree and DBMS using *MSN* and *EECS* traces.

Query Types	**TIF**	*MSN* Trace			*EECS* Trace		
		DBMS	R-tree	SmartStore	DBMS	R-tree	SmartStore
Point Query	120	146.7	32.6	0.108	26.4	8.6	0.074
	160	378.6	122.5	0.179	168.9	42.1	0.136
Range Query	120	1516.5	242.5	1.63	685.2	126.3	1.56
	160	3529.6	625.7	3.41	1859.1	293.1	2.87
Top-k Query	120	4651.8	492.5	2.48	2076.1	196.8	2.25
	160	11524.6	1528.4	4.02	6519.3	571.7	3.47

We also examined the space overhead per node when using SmartStore, R-tree and DBMS, as shown in Figure 7. SmartStore consumes much less space than the R-tree and DBMS approaches, due to its decentralized scheme and multi-dimensional representation. SmartStore stores the index structure, i.e., semantic R-tree, across multiple nodes, while R-tree is a centralized structure. Additionally, SmartStore utilizes the multi-dimensional attribute structure, i.e., semantic R-tree, while DBMS builds a B^+-tree for each attribute. As a result, DBMS has a large storage overhead. Since SmartStore has a small space overhead and can be stored in memory on most servers, it allows the query to be served at the speed of memory access.

5.3 Grouping Efficiency

The grouping efficiency determines how effectively SmartStore can bound a query within a small set of semantic groups to improve the overall system scalability. Figure 8 shows that most operations, between 87.3% and 90.6%, can be served by one group, i.e., a 0-hop routing distance. This confirms the effectiveness of our semantic grouping. In addition, since the semantic vector of one group, i.e., the first-level index unit in the semantic R-tree, can accurately represent the aggregated metadata, these vectors are replicated to other storage units in order to perform fast and accurate queries locally as mentioned in Section 3.4. The observed results prove the feasibility of the off-line pre-processing scheme, which can quickly direct a query request to the most correlated index units.

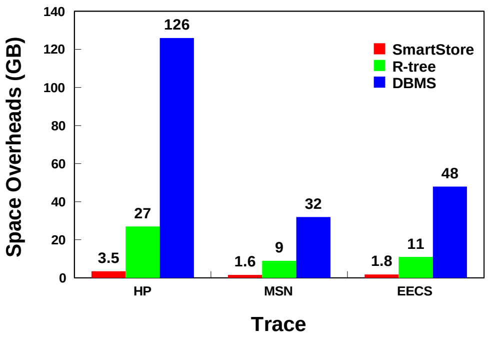


Figure 7: Space overheads of SmartStore, R-tree and DBMS.

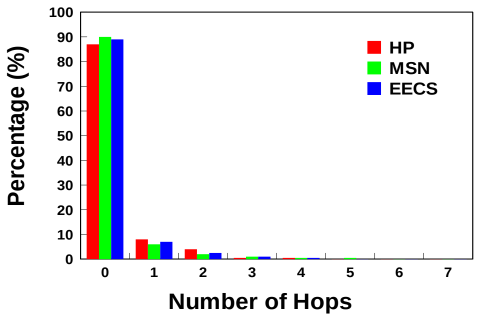


Figure 8: The number of hops of routing distance.

5.4 Query Accuracy

We evaluate the accuracy of complex queries by using the "Recall" measure and of point query by the false probability of Bloom filters.

5.4.1 *Point Query*

SmartStore can support point query, i.e., filename-based indexing, through multiple Bloom filters stored in index units as described in Section 3.3.3. Although Bloom filter-based search may lead to false positives and false negatives due to hash collisions and information staleness, the false probability is generally very small. In addition, these false positives and false negatives are identified when the target metadata is accessed. Figure 9 shows the hit rate for point query. It is observed that over 88.2% query requests can be served accurately by Bloom filters.

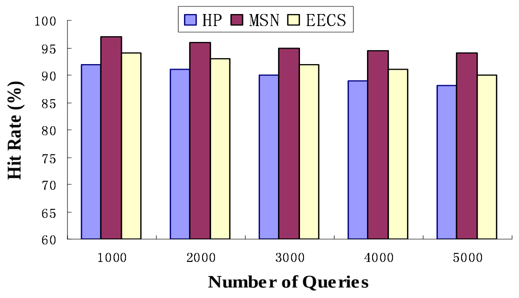


Figure 9: Average hit rate for point query.

5.4.2 *Complex Queries*

We adopt "Recall" as a measure for complex query quality from the field of information retrieval [35]. Given a query q, we denote $T(q)$ the ideal set of K nearest objects and $A(q)$ the actual neighbors reported by SmartStore. We define $recall$ as $recall = \frac{|T(q) \cap A(q)|}{T(q)}$.

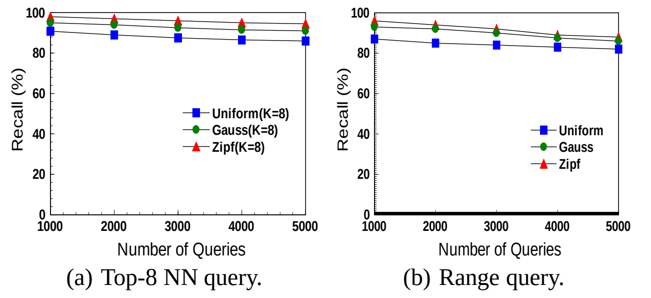


Figure 10: Recall of complex queries using *HP* trace.

Figure 10 shows recall values of complex queries, including range and top-k (k=8) queries, for the *HP* trace. We observe that a top-k query generally achieves higher recall than a range query. The main reason is that top-k query in essence is a similarity search, thus targeting a relatively smaller number of files. We also notice that requests following a Zipf or Gauss distribution obtain much higher recall values than those following a uniform distribution. This is because under a Zipf or Gauss distribution, files are mutually associated with a higher degree than under uniform distribution.

5.5 System Scalability

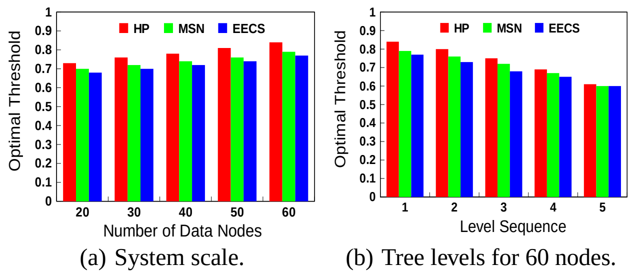
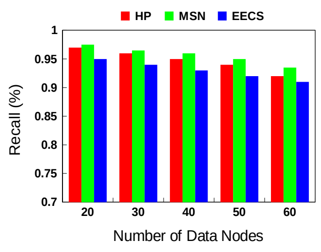


Figure 11: Optimal thresholds.

We study the impact of system size on the optimal thresholds, as shown in Figure 11. Recall that Section 1.1 defines a quantitative measure of semantic correlation, denoted by $\sum_{i=1}^{t} \sum_{f_j \in G_i} (f_j - C_i)^2$, that, when minimized using LSI-based computation, results in the corresponding optimal threshold. Figure 11(a) shows the optimal threshold as a function of the number of storage units. Figure 11(b) shows the optimal thresholds at different levels of the semantic R-tree. We examine the query accuracy by measuring the recall measure when executing 2000 requests composed of 1000 range and 1000 top-k queries, as show in Figures 12. These requests are generated based on the Gauss and Zipf distribution respectively. Experimental results show that SmartStore maintains a high query accuracy as the number of storage units increases, demonstrating the scalability of SmartStore.

We compare on-line and off-line query performance in terms of query latency and number of messages as a function of the system scale as shown in Figure 13. Figure 13(a) compares the query latency between two methods, as described in Section 3.4, under a Zipf distribution. The on-line method identifies the most correlated storage unit for the query requests by multicasting mes-


(a) Gauss distribution.

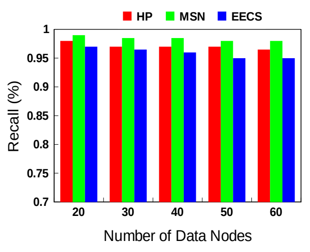
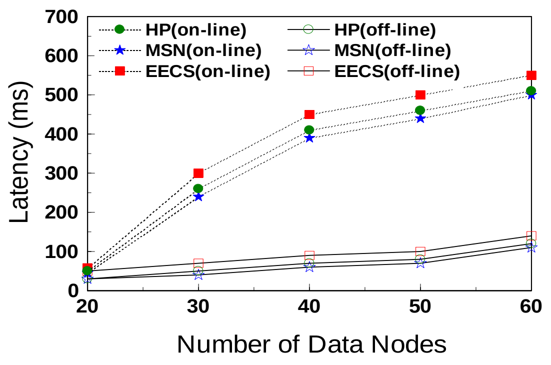

(b) Zipf distribution.

Figure 12: Recall as a function of system scale.

sages; whereas, the off-line method stores semantic vectors of the first-level index units in advance to execute off-line LSI-based preprocessing to quickly locate the most correlated index unit. Figure 13(b) compares the number of internal network messages produced by the on-line and off-line approaches when performing complex queries. We observe that the off-line approach can significantly reduce the total number of network messages.


(a) Average query latency.

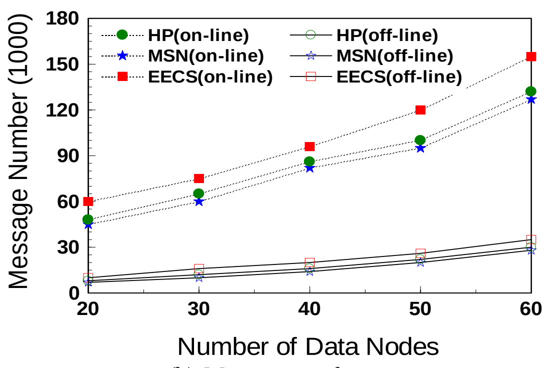
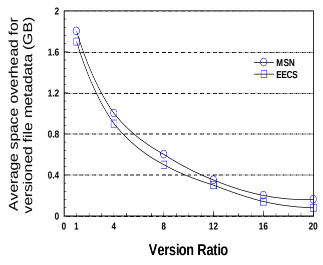

(b) Message numbers.

Figure 13: Performance comparisons using on-line and off-line.

5.6 Overhead and Efficiency of Versioning

Using versioning to maintain consistency among multiple replicas of the root and index nodes of the semantic R-tree, as described in Section 4.4, introduces some extra costs, i.e., extra space and latency, since SmartStore needs to store versions that are checked for quickly locating query results.

Similar to evaluating the versioning file systems [31], we adjust the version ratio, i.e., file modification-to-version ratio, to examine the overhead introduced by versioning. Figure 14 shows the versioning overhead in terms of required space and latency when checking the versions. Due to space limit, this paper only presents the performances under the MSN and EECS traces.


(a) Space overhead.

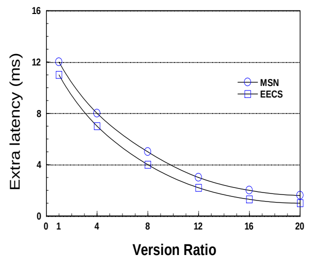

(b) Extra latency.

Figure 14: Versioning overhead in space and access latency.

Figure 14(a) shows the average required space in each index unit. The space overhead is tightly associated with the version ratio. If the ratio is 1, it is called a comprehensive versioning, and every change results in a version, thus requiring the largest storage space. When the ratio is increased, changes usually are aggregated to produce a version to reduce space overhead. The extra space overhead on the whole is acceptable since most existing computers can be expected to provide at least 2GB memory that is sufficient for versions.

Figure 14(b) shows the extra latency incurred verifying query results in the versions. Compared with the entire query latency, the additional versioning latency is no more than 10%. The reason is that all versions only need to record small changes stored in memory and we use rolling backward to reduce unnecessary checking on stale information.

SmartStore uses versioning and updates aggregated changes to maintain consistency and improve query accuracy. Tables 5 and 6 show the recalls of range and top-k queries with and without versioning, as a function of the number of queries, for the *MSN* and *EECS* traces. Experimental results confirm that SmartStore with versioning can significantly improve query accuracy.

Table 5: Recall of range and top-k queries using *MSN*.

		1000	2000	3000	4000	5000
Uniform	Range Query	86.2	85.7	84.5	83.2	82.8
	Versioning	93.5	92.7	92.2	91.6	91.1
	K=8	90.5	89.7	87.4	86.2	85.8
	Versioning	96.7	96.4	96.2	95.8	95.6
Gauss	Range Query	90.5	89.3	88.6	87.7	86.4
	Versioning	96.8	95.9	95.2	94.8	94.3
	K=8	95.8	94.2	93.5	92.4	91.6
	Versioning	100	99.6	99.3	99.1	98.8
Zipf	Range Query	91.2	90.5	89.3	88.7	87.3
	Versioning	100	99.2	98.8	98.6	98.5
	K=8	96.5	95.1	94.3	93.6	92.6
	Versioning	100	100	100	99.8	99.6

6. RELATED WORK

We compare SmartStore with state-of-the-art approaches in content-based search, directory subtree partitioning and database solution.

6.1 Content-based Search

One of the most prevalent metadata queries is content-based query by examining the contents and pathnames of files, such as attribute-based naming in the Semantic file system [36] and content-based search tool in Google Desktop [37]. However, the efficiency of

Table 6: Recall of range and top-k queries using *EECS*.

		1000	2000	3000	4000	5000
Uniform	Range Query	87.3	86.5	84.6	83.2	81.5
	Versioning	95.4	95.2	94.8	94.6	94.3
	K=8	91.5	90.2	89.8	87.4	85.6
	Versioning	97.6	97.3	97.1	96.6	96.2
Gauss	Range Query	89.7	88.2	87.5	85.5	83.1
	Versioning	96.6	96.3	96.1	95.7	95.5
	K=8	96.7	95.1	94.2	92.3	91.1
	Versioning	100	100	99.8	99.5	99.1
Zipf	Range Query	90.2	89.6	87.5	86.7	84.8
	Versioning	100	99.7	99.4	98.9	98.6
	K=8	97.3	96.2	94.8	93.5	92.7
	Versioning	100	100	100	100	99.7

content-based search heavily depends on files that contain explicitly understandable contents, while ignoring file context that is utilized by most users in organizing and searching their data [38]. Furthermore, typical techniques successful for the web search, such as HITS algorithm [39] and Google search engine [40], leverage tagged and contextual links that do not inherently, let alone explicitly, exist in large-scale file systems.

6.2 Directory-based Subtree Partitioning

Subtree-partitioning based approaches have been widely used in recent studies, such as Ceph [3], GIGA+ [41], Farsite [2] and Spyglass [8]. Ceph [3] maximizes the separation between data and metadata management by using a pseudo-random data distribution function to support a scalable and decentralized placement of replicated data. Farsite [2] makes the improvement on distributed directory service by utilizing tree-structured file identifiers that support dynamically partitioning on metadata at arbitrary granularity. GIGA+ [41] extends classic hash-tables to build file system directories and uses bitmap encoding to allow hash partitions to split independently, thus obtaining high update concurrency and parallelism. Spyglass [8] exploits the locality of file namespace and skewed distribution of metadata to map the namespace hierarchy into a multi-dimensional K-D tree and uses multi-level versioning and partitioning to maintain consistency. However, in its current form, Spyglass focuses on the indexing on a single server and cannot support distributed indexing on multiple servers.

In contrast, SmartStore uses bottom-up semantic grouping and configures a file organization scheme from scratch, which is in essence different from the above subtree-partitioning approaches that often exploit semantics of already-existing file systems to organize files. Specifically, SmartStore leverages semantics of multi-dimensional attributes, of which namespace is only a part, to adaptively construct distributed semantic R-trees based on metadata semantics and support complex queries with high reliability and fault tolerance. The self-configuration benefit allows SmartStore to flexibly construct one or more semantic R-trees to accurately match query patterns.

6.3 Database Solution

Researchers in the database field aim to bring database capacity to Petabyte scales with billions of records. Some database vendors developed parallel databases to support large-scale data management, such as Oracle's Real Application Cluster database [42] and IBM's DB2 Parallel Edition [43], by using a complete relational model with transactions. Although successful for managing relational databases, existing database management systems (DBMS) do not fully satisfy the requirements of metadata search in large-scale file systems.

- **Application Environments**: DBMS often assumes dedicated high-performance hardware devices, such as CPU, memory, disk and high-speed networks. Unfortunately, real-world applications, such as portable storage and personal devices, provide limited capacity to support complex queries for managing metadata.
- **Attribute Distribution**: DBMS treats file attributes equally and assumes uniform distribution of their values, ignoring skewed distribution of file metadata. A case in point is that DBMS considers file pathnames as a flat string attribute and ignores the locality of namespace.
- **Access Locality**: Database techniques generally cannot take full advantage of important characteristics of file systems, such as access locality and "hot spot" data, to enhance system performance.

Database research community has argued that existing DBMS for general-purpose applications would not be a "one size fit all" solution [44] and improvements may result from semantic-based designs [45].

7. CONCLUSION

The paper presents a new paradigm for organizing file metadata for next-generation file systems, called SmartStore, by exploiting file semantic information to provide efficient and scalable complex queries while enhancing system scalability and functionality. The novelty of SmartStore lies in it matches actual data distribution and physical layout with their logical semantic correlation so that a complex query can be successfully served within one or a small number of storage units. Specifically, this paper has three main contributions. (1) A semantic grouping method is proposed to effectively identify files that are correlated in their physical attributes or behavioral attributes. (2) SmartStore can very efficiently support complex queries, such as range and top-k queries, which will likely become increasingly important in the next-generation file systems. (3) Our prototype implementation proves that SmartStore is highly scalable, and can be deployed in a large-scale distributed storage system with a large number of storage units.

Acknowledgement

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant 60703046, National Basic Research 973 Program under Grant 2004CB318201, US National Science Foundation (NSF) under Grants 0621493 and 0621526, HUST-SRF No.2007Q021B and the Program for Changjiang Scholars and Innovative Research Team in University No. IRT-0725.

8. REFERENCES

[1] J. Nunez, "High End Computing File System and I/O R&D Gaps Roadmap ," *High Performance Computer Science Week, ASCR Computer Science Research*, August, 2008.

[2] J. R. Douceur and J. Howell, "Distributed Directory Service in the Farsite File System," *Proc. OSDI*, pp. 321–334, 2006.

[3] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system.," *Proc. OSDI*, 2006.

[4] D. Roselli, J. Lorch, and T. Anderson, "A comparison of file system workloads," *Proc. USENIX Conference*, pp. 41–54, 2000.

[5] A. Traeger, E. Zadok, N. Joukov, and C. Wright, “A nine year study of file system and storage benchmarking,” *ACM Transactions on Storage*, no. 2, pp. 1–56, 2008.

[6] A. Szalay, “New Challenges in Petascale Scientific Databases,” *Keynote Talk in Scientific and Statistical Database Management Conference (SSDBM)*, 2008.

[7] M. Seltzer and N. Murphy, “Hierarchical File Systems are Dead,” *Proc. HotOS*, 2009.

[8] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, “Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems,” *Proc. FAST*, 2009.

[9] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, “Bigtable: A distributed storage system for structured data,” *Proc. OSDI*, 2006.

[10] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. OąŕToole, “Semantic file systems,” *Proc. SOSP*, 1991.

[11] P. Gu, Y. Zhu, H. Jiang, and J. Wang, “Nexus: A Novel Weighted-Graph-Based Prefetching Algorithm for Metadata Servers in Petabyte-Scale Storage Systems,” *Proc. CCGrid*, 2006.

[12] S. Deerwester, S. Dumas, G. Furnas, T. Landauer, and R. Harsman, “Indexing by latent semantic analysis,” *J. American Society for Information Science*, pp. 391–407, 1990.

[13] C. Papadimitriou, P. Raghavan, H. Tamaki, and S. Vempala, “Latent Semantic Indexing: A Probabilistic Analysis,” *Journal of Computer and System Sciences*, vol. 61, no. 2, pp. 217–235, 2000.

[14] S. Doraimani and A. Iamnitchi, “File Grouping for Scientific Data Management: Lessons from Experimenting with Real Traces,” *Proc. HPDC*, 2008.

[15] A. Leung, S. Pasupathy, G. Goodson, and E. Miller, “Measurement and analysis of large-scale network file system workloads,” *Proc. USENIX Conference*, 2008.

[16] P. Xia, D. Feng, H. Jiang, L. Tian, and F. Wang, “FARMER: A Novel Approach to File Access coRrelation Mining and Evaluation Reference model for Optimizing Peta-Scale File Systems Performance,” *Proc. HPDC*, 2008.

[17] E. Riedel, M. Kallahalla, and R. Swaminathan, “A framework for evaluating storage system security,” *Proc. FAST*, 2002.

[18] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, “Characterization of storage workload traces from production Windows servers,” *Proc. IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[19] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer, “Passive NFS Tracing of Email and Research Workloads,” *Proc. FAST*, 2003.

[20] Y. Hua, H. Jiang, Y. Zhu, D. Feng, and L. Tian, “SmartStore: A New Metadata Organization Paradigm with Metadata Semantic-Awareness for Next-Generation File Systems,” *Technical Report, University of Nebraska- Lincoln, TR-UNL-CSE-2008-0012*, November, 2008.

[21] Y. Hua, H. Jiang, Y. Zhu, D. Feng, and L. Tian, “SmartStore: A New Metadata Organization Paradigm with Semantic-Awareness ,” *FAST Work-in-Progress Report and Poster Session*, February, 2009.

[22] B. Zhu, K. Li, and H. Patterson, “Avoiding the disk bottleneck in the data domain deduplication file system,” *Proc. FAST*, 2008.

[23] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, “Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality,” *Proc. FAST*, 2009.

[24] X. Liu, A. Aboulnaga, K. Salem, and X. Li, “CLIC: CLient-Informed Caching for Storage Servers,” *Proc. FAST*, 2009.

[25] M. Li, E. Varki, S. Bhatia, and A. Merchant, “TaP: Table-based prefetching for storage caches,” *Proc. FAST*, 2008.

[26] A. Guttman, “R-trees: A dynamic index structure for spatial searching,” *Proc. SIGMOD*, 1984.

[27] B. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[28] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, “Scalable and Adaptive Metadata Management in Ultra Large-scale File Systems,” *Proc. ICDCS*, 2008.

[29] G. Golub and C. Van Loan, *Matrix Computations*. Johns Hopkins University Press, 1996.

[30] J. Hartigan and M. Wong, “Algorithm AS 136: A K-means clustering algorithm,” *Applied Statistics*, pp. 100–108, 1979.

[31] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger, “Metadata efficiency in versioning file systems,” *Proc. FAST*, 2003.

[32] Y. Zhu, H. Jiang, J. Wang, and F. Xian, “HBA: Distributed Metadata Management for Large Cluster-based Storage Systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 4, pp. 1–14, 2008.

[33] D. Comer, “The ubiquitous B-tree,” *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, 1979.

[34] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, “Handbook of applied cryptography,” *CRC Press*, 1997.

[35] B. Piwowarski and G. Dupret, “Evaluation in (XML) information retrieval: Expected precision-recall with user modelling (EPRUM),” *Proc. SIGIR*, pp. 260–267, 2006.

[36] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O. Jr, “Semantic file systems,” *Proc. SOSP*, 1991.

[37] “Google Desktop,” *http://desktop.google.com/*.

[38] C. Soules and G. Ganger, “Connections: using context to enhance file search,” *Proc. SOSP*, 2005.

[39] J. KLEINBERG, “Authoritative Sources in a Hyperlinked Environment,” *Journal of the ACM*, vol. 46, no. 5, pp. 604–632, 1999.

[40] “Google,” *http://www.google.com/*.

[41] S. Patil and G. Gibson, “GIGA+ : Scalable Directories for Shared File Systems,” *Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-08-110*, 2008.

[42] “ORACLE.COM,” *www.oracle.com/technology/products/-database/clustering/index.html. Product page*.

[43] C. Baru, G. Fecteau, A. Goyal, H. Hsiao, A. Jhingran, S. Padmanabhan, G. Copeland, and W. Wilson, “DB2 parallel edition,” *IBM Systems journal*, vol. 34, no. 2, pp. 292–322, 1995.

[44] M. Stonebraker and U. Cetintemel, “One size fits all: an idea whose time has come and gone,” *Proc. ICDE*, 2005.

[45] M. Franklin, A. Halevy, and D. Maier, “From databases to dataspaces: a new abstraction for information management,” *ACM SIGMOD Record*, vol. 34, no. 4, pp. 27–33, 2005.