Towards Efficient Search on Unstructured Data : An Intelligent-Storage Approach

Aravindan Raghuveer, Meera Jindal, Mohamed F. Mokbel, Biplob Debnath, David Du
DTC Intelligent Storage Consortium (DISC), University of Minnesota
Minneapolis, Minnesota, USA
{aravind, jindal, mokbel, du}@cs.umn.edu, debna004@umn.edu

ABSTRACT

Applications that create and consume unstructured data have grown both in scale of storage requirements and complexity of search primitives. We consider two such applications: exhaustive search and integration of structured and unstructured data. Current block-based storage systems are either incapable or inefficient to address the challenges bought forth by the above applications. We propose a storage framework to efficiently store and search unstructured and structured data while controlling storage management costs. Experimental results based on our prototype show that the proposed system can provide impressive performance and feature benefits.

Categories and Subject Descriptors: H.3.0 [Information Storage and Retrieval]: General

General Terms: Algorithms, Design, Performance

1. INTRODUCTION

Storage systems for scientific applications are becoming extremely complex due to the scale and variety of data. For instance, experiments at Mayo clinic (Rochester, Minnesota) have an average storage requirement of 1TB per 9 days [8]. Applications that require complex search primitives on unstructured data are also becoming increasingly common. For instance, query-by-example and data summarization applications exhaustively search through all files because either the query model is fuzzy or indices are extremely expensive to maintain for high dimensional data. Supporting exhaustive search is not among the primary design criteria of today's filesystems and database systems. Therefore the problem of *where* and *how* to store unstructured data in order to search it efficiently, needs a fresh re-assessment. Furthermore, unstructured data is often created, updated and queried in the context of structured data. Although, there is a lot of research in providing a unified access at the query level, there is still lack of research in providing a scalable storage platform for integrating structured and unstructured data. In the presence of petabytes of data and Information Lifecycle Management (ILM) regulations like HIPAA, controlling storage management costs is extremely challenging. In this paper, we present one complete solution to tackle three highly interconnected sub-problems: efficiently storing and searching a wide variety of unstructured data, integrating unstructured and structured data, controlling storage management costs.

The block interface, widely prevelant among todays' systems, does not allow for any form of communication between the application and the underlying storage subsystem. With complex applications operating on petabyte scale storage, this disconnect leads to scalability problems, performance degradation and high cost of storage management. Object-based storage [4, 6] addresses all the drawbacks of block-based storage by revamping the narrow block interface with an extensible and expressive object interface. In object-based storage devices (OSD, for short), the storage management component of the filesystem or the database is migrated to the storage device. Therefore, the tasks of managing free space and mapping object IDs to physical disk locations is delegated to the storage system. This division of labor enhances scalability, enables communication through a richer interface and empowers the storage to perform smarter data-specific optimizations to improve application performance.

Custom storage systems have been researched in the past to overcome the limitations of block storage. The Google Filesystem [2] uses an object-based storage system to provide high storage performance through parallel, direct access I/O paths. The Active Disk [5] project demonstrated that large scale data mining and multimedia applications benefit from embedding search functionalities into the storage system.

Storage systems that integrate structured and unstructured data can be broadly classified into three categories. (1) *The Database-only approach:* unstructured data is stored as binary large objects (BLOBs) within a database. (2) *The database-plus-filesystem approach:* structured data is stored in a database, unstructured data is stored in a filesystem and path names are used to link the data together (3) *Custom Architectures:* The Google Bigtable [1] integrates structured data and text. It uses the google filesystem beneath to store and manage data. Native XML stores can also be used to integrate structured data and text.

Our solution tackles the three subproblems (namely: storing and searching unstructured data, integration with structured data and controlling storage management costs)in two steps. First, we we propose an application-aware storage system called an Intelligent Storage Node (ISN, in short) that can efficiently store and search unstructured data. The underlying principle behind the proposed ISN is that when the storage system is aware of the application, it can map *application access patterns* to *storage-device specific optimizations* to improve application performance. We demonstrate this principle through a storage embedded exhaustive search algorithm that performs upto six times better than conventional techniques.

Second, we use ISNs as a building block to build a system called

CIKM'07, November 6–8, 2007, Lisboa, Portugal.

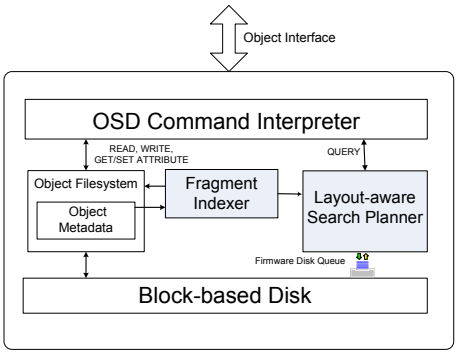

Figure 1: Architecture of an Intelligent Storage Node: Shaded boxes represent the new modules that an ISN adds to a traditional OSD

SQUAD that enables seamless and storage-centric integration of structured and unstructured data. SQUAD stores unstructured data on ISNs and structured data in a traditional database. An entity called Metadata Server abstracts the ISNs and the database to provide a unified, query-able view to the client.

2. INTELLIGENT STORAGE FOR UNSTRUCTURED DATA

In this section, we present the concept of an *Intelligent Storage Node* and show that through application-awareness at the storage level, an ISN can improve the performance of exhaustive search. An intelligent storage node is an object-based storage device with storage-device-specific software to make exhaustive search efficient. An ISN could just be a rack server or a commodity PC that exposes a standardized object interface like OSD T-10 [6]. The ISN supports a new command `OSD_QUERY` in addition to the OSD T-10 standard command set to submit exhaustive search queries to the storage device.

In the filesystem-based approach, exhaustive search is implemented by recursively exploring the filesystem namespace. With filesystem aging and fragmentation, recursive exploration can end up being a random scan operation at the disk level leading to extremely poor performance. This clearly shows that the lack of communication between the filesystem and the storage system leads to poor performance. Another drawback with current systems is the lack of performance isolation. Exhaustive search, being a long-running, I/O intensive task, can increase the response times of other concurrent filesystems applicatons. Todays' block-based storage systems cannot differentiate between requests from two host applications to provide any form of performance isolation. We show that through application-awareness at the storage level, the ISN boosts performance and provides performance isolation. We discuss these two features of the ISN in the next two subsections: Search Strategy and Search Suspend-and-Resume.

2.1 Search Strategy

The principle of our search approach is to exploit application-awareness at the storage device to transform application requests to an access pattern that is most efficient at the storage device. For the exhaustive search application, we read all the objects in the ISN in a specific order that gives near-sequential performance.

An ISN stores data objects as a set of non-contiguous fragments. Each fragment is a contiguous run of logical blocks on disk that are assigned to a particular object. In an exhaustive search operation, all object fragments on the ISN need to be examined. Therefore the order in which they are retrieved is not important. We leverage on this characteristic feature of the exhaustive search application to plan the search operation such that we visit the fragments in an order that minimizes random seeks. By minimizing random seeks, we get close to sequential performance. Another critical point to note here is that since we operate at the granularity of a fragment, the extent of fragmentation on the ISN does not affect the performance of the exhaustive search significantly. On the other hand, filesystem level exhaustive search will be severely affected by fragmentation since it operates at the granularity of files or objects. Our approach assumes that the fragments of the objects can be visited in any order. With large main memory available in modern storage devices, this assumption is valid for most unstructured data like text, video, images and audio. Also, since objects that do not satisfy the query are not returned back to host, interconnect bandwidth is not wasted.

Figure 1 gives the architecture of the proposed ISN with embedded exhaustive search functionality. The shaded boxes represent the *application-aware intelligence* that we introduce to a regular object-based storage device. In the front-end, we have an *OSD command interpreter* which exposes a standard object interface [6]. The *object filesystem* performs disk-space management routines and maps a given object-ID to the fragments it occupies on the disk. The *Fragment Indexer* maintains the fragments in efficient visit order. The fragment indexer interacts with the object filesystem to update the index whenever object fragments are created, updated or deleted. The *Search Planner* module is the heart of the exhaustive search system. Given the current position of the disk head, the search planner consults the fragment indexer to find the most efficient order to visit the fragments. The search planner operates at a higher granularity when compared to the scheduler that resides on the disk firmware. Based on an estimate of the disk's current head position, the planner selects the set of fragments closest to head and queues requests to those fragments inside the disk firmware queue. The disk may re-order these requests to find the most efficient order to visit them. So the search planner works in concert with the native command queuing available at the disk to discover the best possible search plan.

The proposed exhaustive search technique shows *how* to efficiently search a large disk drive and not *what* to search. Therefore, any application that needs to read the entire filesystem fits our requirement for a search application. We assume that storage is the bottleneck in the system and the application can process the data close to sequential disk bandwidth. Recent trends indicate that storage blades with extremely fast processors and large buffer memory are becoming common. The reader is referred to the Diamond project [3] for exhaustive search applications and a storage-level programming model for such applications.

2.2 Search Suspend-and-Resume

Upon receiving a real-time request, an ongoing exhaustive search process is suspended. It is resumed after servicing the real-time request. When a suspended search operation is resumed, the search resumes exactly from where it left off. For instance, assume the disk head is over fragment number f_i and the next fragment in the search plan is f_{i+1}, when a real-time request for block b arrives at the disk. The search planner first finishes retrieving fragment f_i. It then suspends the search and repositions the disk head at logical block b for servicing a real-time request. On resumption, the search proceeds from fragment f_{i+1} irrespective of the physical distance between f_{i+1} and b. Therefore, on every resume operation, the static plan mode may incur costly seek operation. However the resume operation is very straightforward to implement and has very

little state information to be maintained, namely, last fragment visited. Also the search plan is computed just once and it remains static until the search is completed.

3. INTEGRATING STRUCTURED AND UNSTRUCTURED DATA

Structured and unstructured data have entirely different storage access techniques. Structured data is best placed within the slotted page structure of databases while filesystems have been proven to handle unstructured data better [7]. This *impedance mismatch* forms the underlying challenge in integrating structured and unstructured data at the storage level.

To address the above challenge, we propose SQUAD: a unified framework for storing and querying structured and unstructured data. In SQUAD, structured data is stored in a traditional database system and the unstructured data lives in the ISNs as objects. Relationships between structured and unstructured data are maintained through object identifiers. The Metadata Server (MDS) forms a wrapper around the database and the ISNs to expose a single, query-able integrated data store. The SQUAD client uses a set of APIs exposed by the Metadata Server to provide intuitive tools for the user to seamlessly store and search mixed data. The client also has an object filesystem that exposes a traditional hierarchical namespace while internally using object devices to store the data. In the rest of the section we explain how the simple SQUAD framework provides integration at the storage and query dimensions.

Integration at the Storage Level: The database uses object IDs to keep track of the unstructured data related to a particular tuple. Since object identifiers are application, filesystem path and location independent, integration is generic. Furthermore, the extra indirection provided by object IDs enables loose coupling between the database and the ISNs. Consequently, the database is relieved from handling filesystem artifacts (e.g., path names and access control lists) and the filesystem is relieved from handling database features (e.g., integrity constraint violation checks). All tasks that fall in the intersection of filesystem and databases are now delegated to the MDS. The MDS is responsible for ensuring that any changes in the filesystem's state do not render the database inconsistent and vice versa. For instance, the MDS commits a delete only when the operation does not violate any integrity constraints of tuples associated with the object. However, read/write of objects from/to the ISNs happens without the intervention of the MDS. This clear separation of control and data paths provides high parallel access performance [2, 4, 6].

Integration at the Query Level: In SQUAD, ISNs and the database form a distribution query execution framework. The database executes queries on structured data while the ISNs execute exhaustive search queries on unstructured data. Exhaustive search is one example to show how application awareness at storage can be used to improve performance. Concepts like extended attributes and sessions in object-based storage systems can be used to provide differentiated service based on the type of data and application [4, 6]. In SQUAD, the client submits a mixed query to the MDS. The MDS then decomposes the query into its structured and unstructured components, dispatches sub-queries to the database and ISNs respectively, stitches the results together and relays it back to the client.

4. PROTOTYPE IMPLEMENTATION

We have built a prototype SQUAD system with all the components shown in Figure 2. In the following subsections, we explain in detail, some critical implementation aspects of our prototype.

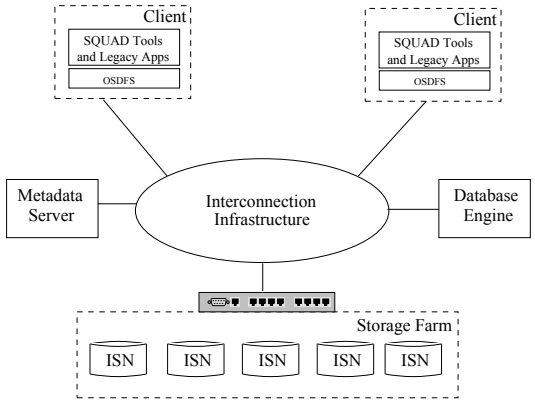

Figure 2: The SQUAD framework

An ISN is an enhanced version of an object-based storage device. We extended an open source implementation of an object target available from the DISC-OSD project [9]. The DISC-OSD target is a user-level program, that exposes the T10 object interface over iSCSI transport. The DISC-OSD target internally stores objects as files on the Linux extended filesystem (ext3). We implemented two modules within the target to support exhaustive search : *fragment indexer* and *search planner*. The *fragment indexer* maintains efficient search order of object fragments as they are created, updated or deleted. We use a persistent B-Tree to maintain fragment information (namely, object ID and fragment number) in increasing order of logical block addresses (LBA). The *search planner* consults the fragment index to construct a search plan that sweeps from start to the end of the disk.

To compare the performance of our approach versus a filesystem level exhaustive search, we construct search plans for both cases. The filesystem search plan visits each object in the order as seen by a filesystem (e.g., alphabetic order). The search plan constructed by our technique visits the objects at a fragment granularity and in an order that is as close to sequential as possible. We have implemented the *Query Executor* which reads the entire filesystem contents based on the two search plans. The Query Executor uses the SCSI generic (sg) driver [10] to construct `SCSI READ` commands and queue them at the disk.

To observe the effect of fragmentation on the performance of our scheme, we implemented a synthetic *aging tool* that performs file create, delete and append operations. We use a generalized version of *storage age* metric introduced in [7] to quantify the extent of fragmentation of a storage system. The storage age of a volume is defined as the ratio between the number of bytes of modified data (written/deleted) and the number of bytes in use on the volume. We first bulk load a filesystem with binary data to the required occupancy level of $l\%$. Since the filesystem is new and unfragmented, the age is zero. Then, we start the aging tool and fragment the filesystem to the required age. We ensure that while aging, the size of the filesystem is always $(l \pm 0.5)\%$.

5. PERFORMANCE ANALYSIS

In this section, we evaluate the performance of the proposed ISN and the SQUAD framework using our prototype system.

The ISN is set up on a Dell PowerEdge 2650 Server that has two Intel Xeon 2GHz processors, 2 GB RAM and three direct-attached Seagate Cheetah 15K rpm, 73G SCSI hard drives (ST 373454LC). The PostgreSQL database is set up on another Dell PowerEdge 2650 Server with the same hardware configuration as above. The

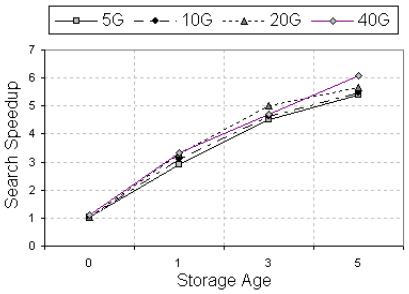

Figure 3: Speedup provided by an ISN for the exhaustive search operation compared to a filesystem-level search.

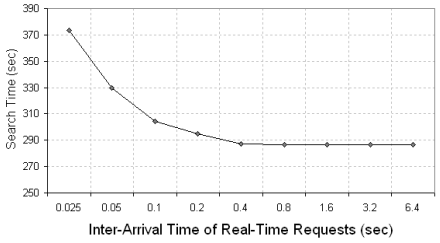

Figure 4: Suspend-Resume exhaustive search on ISN (filesystem size=20G, age=5)

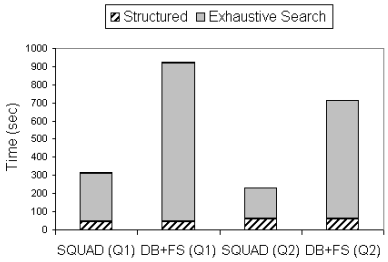

Figure 5: Time taken to execute queries Q_1^{icon} and Q_2^{icon} on SQUAD, DB+FS.

MDS code is set up on a Dell Power Edge 2950 that has two, dual core Intel Xeon 3GHz processors, 2GB RAM and three Fujitsu 15K rpm, 73G direct-attached SAS hard drives (MAX3073RC). The client is set up on a commodity PC that has an Intel Pentium III, 1 Ghz processor, 512 MB RAM and a direct attached Seagate 30G SATA hard drive. All the above components are connected through a gigabit Ethernet network.

5.1 Intelligent Storage Node

In the first set of experiments, we examine the performance of an ISN for exhaustive search queries and compare it with the traditional filesystem approach. We set up an ext3 filesystem on a partition of size 63GB. We use the aging tool mentioned in Section 4 to control the extent of storage fragmentation. We populate the filesystem with random binary objects with an average of 256KB.

In the first experiment, we vary both the age of the storage system and filesystem occupancy and find the times taken to perform exhaustive search using filesystem level search and ISN-based layout-aware search (see Figure 3). The vertical axis shows the search speedup which is the ratio of Filesystem Search Time to ISN Search Time. We see that that when the filesystem is new (age = 0), both the filesystem level search and the ISN-based search have the same performance (i.e., speedup = 1). But as the filesystem ages, the performance of the filesystem level exhaustive search degrades. ISN-based search however is able to maintain the performance by re-ordering search requests to obtain near-sequential performance, consequently leading to upto six times speedup.

Figure 4 gives the time taken to perform exhaustive search on an ISN (filesystem size=20G, age=5) in the presence of real-time traffic. The real-time requests were `READ` operations on one sector randomly chosen within the filesystem boundary. The horizontal axis gives the inter-arrival time between two real-time requests. Therefore as we move to the right, the real-time requests are less frequent. The vertical axis gives the exhaustive search time. We see that when the real-time traffic is high, the performance of the exhaustive search suffers. This can be attributed to the incurred random seek operations.

5.2 The SQUAD Framework

We now evaluate the performance of the SQUAD framework for mixed queries. The mixed queries are of the form ($Q = Q_s \cap Q_u$), where Q_s is a SQL query on structured data and Q_u is an exhaustive search query. We use the following schema in our experiments: *Wiki(pageID, date, HTML, icon)*. *pageID* is an indexed attribute and *date* is a non-indexed attribute. The *icon* field stores a pointer to data stored outside the database. Depending on the whether we use the *database-plus-filesystem* (DB+FS) approach or SQUAD, this pointer is a filesystem path name or objectID. Exhaustive search is used for queries on *icon*. *pageID* and *date* form the structured component of the database while *HTML* and *icon* constitute the unstructured component. We populate the table with roughly half million rows. We use the 5G wikipedia dump for the HTML attribute.

We evaluate the performance of SQUAD for two mixed queries. The structured component Q_s has two variations, Q_1 and Q_2; where $Q_1 = (date > 10/3/2000)$ and $Q_2 = (date > 10/3/2000) AND (objectID > 4000)$. Therefore we have two mixed queries that are represented as : Q_1^{icon} and Q_2^{icon}.

We compare the performance of SQUAD with the DB+FS approach for the queries Q_1^{icon} and Q_2^{icon} (See Figure 5). In both approaches, the database first executes Q_s. On the resulting set of objectIDs from Q_s, Q_u is executed on either the filesystem or the ISN. We observe that SQUAD performs better than the DB+FS approach for both queries. The performance advantage is because of the layout-aware search available at the ISNs. However in this case, the ISNs do not perform an exhaustive search but instead search a subset as found by Q_s. From the above experiments, we can conclude that for mixed queries involving exhaustive search, SQUAD has significant better performance than the DB+FS approach. The performance advantage stems from the fact that the SQUAD design embeds search functionalities in components based on where they can be performed best. Specifically, executing Q_s is embedded in the database and executing Q_u is embedded in the ISN.

6. REFERENCES

[1] F. Chang, J. Dean, S. Ghemawat, and et. al. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM SOSP*, pages 29–43, 2003.

[3] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A Storage Architecture for Early Discard in Interactive Search. In *Proceedings of the International Conference on File and Storage Technologies, FAST*, 2004.

[4] M. Mesnier, G. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, August 2003.

[5] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 62–73, 24–27 1998.

[6] *SCSI Object-Based Storage Device Commands -2 (OSD-2)*. Project T10/1721-D, Revision 0, October 2004.

[7] R. Sears and C. van Ingen. Fragmentation in Large Object Repositories. In *CIDR*, 2007.

[8] N. Spillers. Storage Challenges in the Medical Industry. In *The 4th Intelligent Storage Workshop, Digital Technology Center, University of Minnesota*, 2006.

[9] The DISC-OSD T10 Reference Implementation. `http://sourceforge.net/projects/disc-osd`.

[10] The Linux SCSI Generic (sg) Driver . `http://sg.torque.net/sg/`.