Rethinking Storage for Discard-Based Search

Lily Mummert[†], Steve Schlosser[†], Mike Mesnier[†], M. Satyanarayanan[‡]

December 2007
CMU-CS-07-176

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[†]Intel Research Pittsburgh, [‡]Carnegie Mellon University

Abstract

The workload characteristics of content-based image retrieval are poorly suited to existing storage architectures. This is particularly true for *discard-based* search, where images are filtered on-demand, potentially at the storage devices themselves (an active disk technique referred to as *early discard*). In general, discard-based search is a highly concurrent, sequential, and read-only workload, making it ideally suited for JBOD ("just a bunch of disks"), as opposed to a more familiar RAID configuration. Further, as with most image databases, no specific order is imposed on the retrieved images. In the context of discard-based search, such any-order semantics introduce a variety of opportunities to the storage system designer (e.g., an I/O coalescing technique called "bandwagon synchronization").

This paper examines the storage workloads of discard-based search, and discusses the implications for a new storage system specifically designed for content-based image retrieval. In addition, representative synthetic workloads are used to demonstrate the efficiency of JBOD over RAID, and to quantify the benefits of bandwagon synchronization. We show that JBOD achieves 80–90% of an array's bandwidth (depending on the level of I/O concurrency and the average object size), compared to 40–80% for RAID-0.

This research was partly supported by the National Science Foundation (NSF) under grant number CNS-0614679. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, Carnegie Mellon University, or Intel Corp.

Keywords: Image processing, distributed search, OpenDiamond®, active storage, active disk, RAID, JBOD, caching, indexing

1 Introduction

Discard-based search, a new approach to interactive search of complex data, was introduced by Huston *et al* in 2004 [16] in the context of a prototype called *Diamond.* Instead of precomputing index data structures for anticipated queries over an entire corpus, which is the approach used by search engines and databases today, Diamond formulates and executes a customized computation in response to each query. As discussed below in Section 2.2, this query-specific customization yields a number of important benefits for an interactive user when searching rich data of high dimensionality such as medical images. An optimization called *early discard* is the key to making discard-based search practical on large volumes of data. In this optimization, application-specific code called a *searchlet* is executed close to storage with the goal of rejecting irrelevant data as cheaply and efficiently as possible.

There is growing evidence that discard-based search can lead to new kinds of domain-specific applications in which hypothesis formation and hypothesis validation proceed hand-in-hand in a tightly-coupled and iterative sequence. This inherently human-centric activity is called *interactive data exploration (IDE)*. The published literature on Diamond spans many IDE examples, including use of personal and professional digital photograph collections [15], tracking of suspicious entities using surveillance cameras [14], similarity search in mammogram interpretation [32], and adipocyte quantitation in automated cell microscopy [11]. While it is premature to speculate on the eventual significance of IDE, it appears likely that discard-based search is here to stay.

This paper examines the storage workloads induced on servers by discard-based search. We show that these workloads contradict many commonly-held assumptions that are implicit in the design of storage systems today:

- Striping proves to be less effective than a JBOD ("just a bunch of disks") organization.
- Aborting work in progress turns out to be the norm rather than the exception.
- Speculation in the form of additional computation imposes no additional cost on the system, and can yield performance benefits.
- Some applications are not particular about order of storage accesses.
- Avoiding seeks is not always a dominant performance consideration.

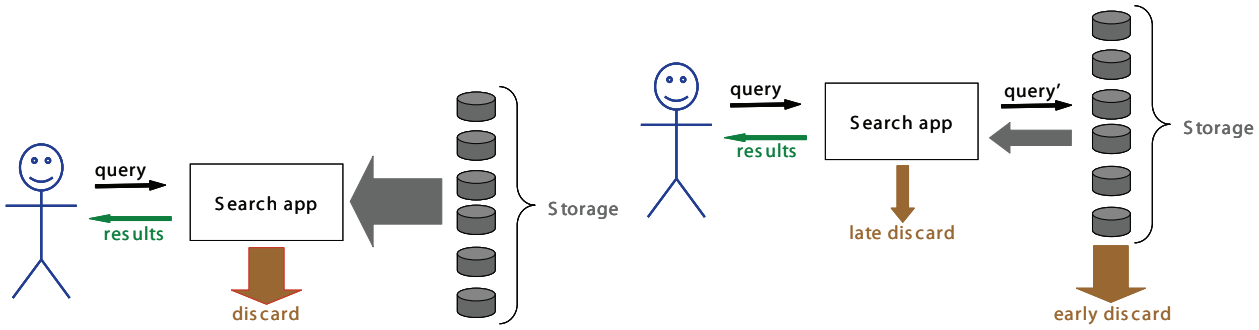
These counter-intuitive observations suggest a fundamental rethinking of storage design for discard-based search applications. We begin in Section 2 by providing background information on discard-based search and systems that use it. Next, Section 3 presents the distinct characteristics of server storage workloads induced by discard-based search. Sections 4 to 9 then show how these workload characteristics impact some widely-held beliefs about storage today. From these observations, Section 10 proposes a metaphor for the ideal design of server storage to support discard-based search. We conclude in Section 12 with a summary of the main points of the paper.

2 Background

2.1 Indexed Search

Today, the term "search" is almost synonymous with "indexed search" because of its phenomenal success in many contexts, ranging from Web search engines to relational databases. An indexed search strategy rests on three assumptions. First, the entire space of queries can be anticipated in advance. Second, it is possible to preprocess data to create indexes. Third, the preprocessing can label the data using query-relevant index terms (also called "tags," "meta-data," or "annotations"). Historically, text has been the data type most thoroughly studied from an indexing viewpoint.

In spite of extensive research in systems such as QBIC [9], automated indexing of complex data such as images remains a challenging problem for several reasons. First, automated methods for extracting semantic content from many data types are still rather primitive. This is referred to as the *semantic gap* [20] in information retrieval. Second, the richness of the data often requires a high-dimensional representation that is not amenable to efficient indexing. This is a consequence of the *curse of dimensionality* [5, 8, 33]. Third, realistic user



(a) Basic Concept (b) Early Discard Optimization

Figure 1: Discard-based Search

queries can be very sophisticated, requiring a great deal of domain knowledge that is often not available to the system for optimization. Fourth, expressing a user's vaguely-specified query in a machine-interpretable form can be difficult. These problems constrain the success of indexed search for complex, multi-dimensional, loosely-structured data.

2.2 Discard-based Search

Diamond explores a radically different approach to searching complex data, called *discard-based search.* This approach does not attempt to preprocess all data in advance of future queries. Rather, it performs *content-based computation on demand in response to a specific query.* The purpose of the computation is to eliminate objects that are clearly not results for the query. An optimized version of this concept is called *early discard.* The optimization rejects most of the irrelevant data as early as possible in the pipeline from storage to user. This improves scalability since it eliminates a large fraction of the data from most of the pipeline. Figure 1 illustrates the concepts of discard-based search and the early-discard optimization.

Since the knowledge needed to recognize irrelevant data is domain-specific, discard-based search requires domain-specific algorithms to be executed on data objects during a search. In Diamond, these algorithms are embodied in searchlets. Early discard requires searchlets to be executed close to storage. Ideally, discard-based search would reject all irrelevant data without eliminating any desired data. This is impossible in practice because of a fundamental trade-off between false-positives (irrelevant data that is not rejected) and false-negatives (relevant data that is incorrectly discarded) [8]. The best one can do in practice is to tune a discard algorithm to favor one at the expense of the other. Different search applications and queries may need to make different trade-offs in this space.

2.3 Relative Merits

The strengths and weaknesses of indexed search and discard-based search complement each other. As discussed below, speed and security favor indexed search. However, discard-based search offers other advantages. These include flexibility in tuning between false positives and false negatives, dynamically incorporating new knowledge, and better integration of human expertise. The growing interest in discard based search suggests that the latter attributes offer high value in some important domains.

Search Speed: Because all data is preprocessed, there are no compute-intensive or storage-intensive algorithms to be run during an indexed search. It is therefore much faster than discard-based search, possibly by many orders of magnitude.

Server security: The early-discard optimization, which is essential for scaling discard-based search to large data volumes, requires searchlet code to be run close to servers. The closer to storage that searchlets can be executed, the more effective the optimization. While a broad range of sandboxing techniques [28], language-based techniques [29], and verification techniques [21] can be applied to reduce risk, the essential point remains that potentially untrusted code may need to run on trusted infrastructure during a discard-based search. This

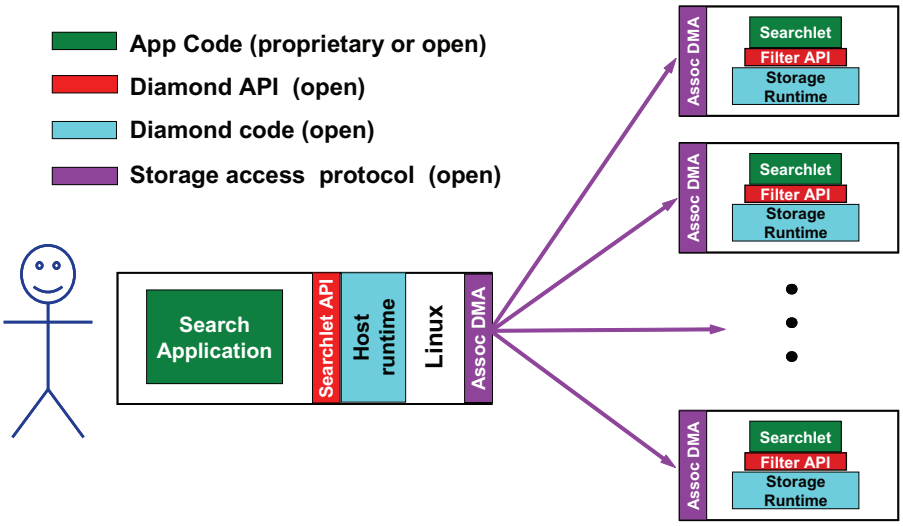


Figure 2: Diamond System Architecture

risk does not exist with indexed search, where the preprocessing is done offline by trusted code on trusted infrastructure.

Precision and Recall: The preprocessing for indexed search represents a specific point on a precision-recall curve, and hence a specific design choice in the tradeoff space between false positives and false negatives. In contrast, discard-based search can change this tradeoff dynamically as a search progresses through many iterations. An expert user with extensive domain-specific knowledge may tune searchlets toward false positives or false negatives depending on factors such as the purpose of the search, the completeness of the search relative to total data volume, and the user's expert judgement of results from earlier iterations in the search process. It is also possible to return a clearly-labelled sampling of discarded objects during a discard-based search to alert the user to what he might be missing, and hence to the likelihood of false negatives.

New knowledge: The preprocessing for indexing can only be as good as the state of knowledge at the time of indexing. New knowledge may render some of this preprocessing stale. In contrast, discard-based search is based on the state of knowledge of the user at the moment of searchlet creation or parameterization. This state of knowledge may improve even during the course of a search. For example, the index terms used in labelling a corpus of medical data may later be discovered to be incomplete or inaccurate. Some cases of a condition that used to be called "A" may now be understood to actually be a new condition "B." Short of re-indexing the entire corpus, this new knowledge cannot be incorporated into indexed search. Note that this observation is true regardless of how the index terms were obtained, including game-based human tagging approaches such as ESP [27].

User expertise: Discard-based search takes better advantage of the expertise and judgement of the user conducting the search. There are many degrees of freedom, including searchlet creation and parameterization, through which this expertise and judgement can be expressed. In contrast, indexed search limits even the most expert user to the intrinsic quality of the preprocessing that produced the index.

2.4 Diamond Architecture

As shown in Figure 2, Diamond cleanly separates domain-specific application code from a generic runtime system. The user interacts via a GUI with a domain-specific application on a client machine. To handle a user query, the application constructs a searchlet out of individual components called *filters*. For example, an application might construct a searchlet using color filters, texture filters, and face detection filters. The parameter values to each filter serve to tune it: for example, the RGB parameter values to a color filter determine what color it detects. The searchlet is presented by the application through the Searchlet API to the Diamond runtime system, which then distributes it to all the servers involved in the search task.

At each server, Diamond iterates through the locally-stored objects in a system-determined order and presents them to filters for evaluation through the Filter API. Each filter can independently discard an object. Diamond is ignorant of the details of filter evaluation, only caring about the scalar return value which is thresholded to determine whether the object should be discarded or passed to the next filter. Only those objects that pass through all of the filters in the searchlet are transmitted to the client. For each object, the order of filter evaluation is determined by runtime cost-benefit statistics that are dynamically re-evaluated as a search proceeds. Each server performs persistent caching of filter execution results, using object identfier and a cryptographic hash of filter code and parameters as the tag value for a cache entry. Huston *et al.* [16] describe Diamond in more detail.

3 Discard-based Search Workloads

The APIs through which discard-based search clients and servers interact with storage are based on an object model. An object is viewed as a linear sequence of bytes and is named by a unique identifier. Due to the nature of discard-based search, these applications exhibit certain distinct storage access characteristics on servers:

- *Read-only, whole object:* After initial provisioning, a corpus of data is never modified. Although the runtime performs persistent caching of filter executions, the storage updates for persistence occur in data areas distinct from the corpus itself. In a typical discard-based search application, the early-discard computation performed by a filter involves the entire contents of an object. While it is theoretically possible for early-discard to be based only on part of an object's content, we have not seen this in practice.
- *Any-order semantics:* The API through which a filter requests the next object to be examined uses an iterator model. In other words, a filter says "Get next object" rather than "Get object X." Hence, the storage subsystem on a server is free to return any object that has not been presented before in the current search. All that is guaranteed by the API is exactly-once semantics within a search: no object is repeated and, unless a search is aborted, every object is presented once. This offers the storage subsystem a degree of freedom that is rarely available in non-discard-based search applications.
- *Frequent aborts:* As mentioned earlier, discard-based search facilitates interactive exploration of complex, non-indexed and loosely-organized data. *Iterative query refinement* is a key characteristic of this style of user interaction. A user issues a query, gets back a few results, and then uses these exemplary results to refine the query. The user is, in effect, conducting two interleaved searches: one on the query space and the other on the data space. This continues until the user finds the desired results or gives up. Almost never does a user wait for the entire corpus of data to be processed by a query. Instead, there is high probability that current query may be aborted by the user at any moment.
- *Embarrassing parallelism:* It is hard to imagine a workload more friendly to server CPU and storage parallelism than a typical discard-based search application. Each object is processed independently and in its entirety, with no concurrency control or ordering constraints across objects. Since image objects are typically large, the net effect is to provide ample opportunity for coarse-grained and easy-to-exploit parallelism.

In Sections 5 to 9 we show how these workload characteristics are at odds with some key tenets of storage design today. We use the term *tenet* here in accordance with its dictionary definition [19]: "a principle, belief, or doctrine generally held to be true; especially one held in common by members of an organization, movement, or profession." The storage community has evolved these tenets over the course of many years based primarily on workloads from databases, file systems, web servers, scientific computing and personal computing. The central message of this paper is that discard-based search workloads require a rethinking of these tenets. We begin by describing our experimental setup in the next section.

4 Experimental Setup

4.1 Testbed

All experiments were run on an Intel SSR212CC storage system, which is a 3U server with a 2.8GHz Intel Xeon CPU with hyperthreading, 1GB of main memory, and twelve internal disks. The disks were 200GB Seagate Barracuda 7200.10 drives (ST3200820AS), which were connected to two Intel SRCS28X SATA RAID controllers. Each controller had 128MB of memory and SATA ports running at 3Gb/s. Five disks on one controller were used in a JBOD configuration, while five disks on the other controller were configured as a striped RAID array with 64KB stripe units using the hardware RAID. The remaining two drives were used as spares. The operating system on the server was Ubuntu 7.04 (Fiesty Fawn) server edition, with Linux kernel version 2.6.20.

While the disk drives we used support SATA Native Command Queueing (NCQ), we found that NCQ was disabled in the RAID controllers due to a hardware bug. However, NCQ is not critical to discard-based search workloads, which are almost entirely sequential.

4.2 Workload generator

All of our experiments used a synthetic benchmark that generates a discard-based search storage workload. Given a number of objects and file size (or range of file sizes), the benchmark creates a synthetic repository in which the objects are laid out sequentially, nose to tail, on one or more storage devices. Whole object reads are issued using direct device I/O. The repository can be scanned sequentially, or specific objects may be accessed randomly given additional settings such as the query pass rate and the probability that results for an object are cached. Passing objects are assumed to be distributed uniformly throughout the repository. Repositories may be read concurrently using a specified number of threads per device. Multiple threads synchronize on a work queue consisting of the list of files in the repository, their sizes and locations.

5 Tenet: “Think Striping”

When considering a storage system design for an application such as discard-based search, it would seem that its embarrassingly parallel, read-only workload would be ideally suited to a RAID array. In particular, the widely known performance advantages of RAID-0 make it an obvious first choice. The only questions remaining relate to the stripe factor (number of disks) and the stripe unit (amount written to each disk).

It is tempting to immediately proceed down the RAID path and overlook the simple JBOD solution: “just a bunch of disks.” However, earlier work on medical image retrieval found little performance improvement from the use of RAID [12], and even earlier work on RAID [6] suggests that during times of high concurrency (the expected case for discard-based search), one should reduce the number of disks that each I/O is required to access, by employing “narrow striping.” For example, with four disks and a stripe unit of 64 KB, a 256 KB request must access all four disks (a stripe width of 4). However, by increasing the stripe unit to 128 KB, a 256 KB request will access only 2 disks (a narrower stripe).

In effect, JBOD represents the narrowest striping, where each request touches exactly one disk. JBOD sets the stripe unit sizes to be equal to the size of each individual data object, meaning that each object is stored entirely on a single disk. Redundancy is achieved easily with mirroring.

5.1 Justifying JBOD

The goal of discard-based search is to filter disk images (objects) on-demand. Because there is no particular order imposed on the objects, they can be read and discarded in any order, and the most efficient way of processing objects is to read them sequentially from disk. Given a single disk drive and a collection of images, one can store the images sequentially and then process them in the same sequential order.

Of course, the best performance one can expect from any single disk drive is that of its full streaming bandwidth, where the disk’s rotational speed is the only limiting factor. In the case of reads, one can achieve

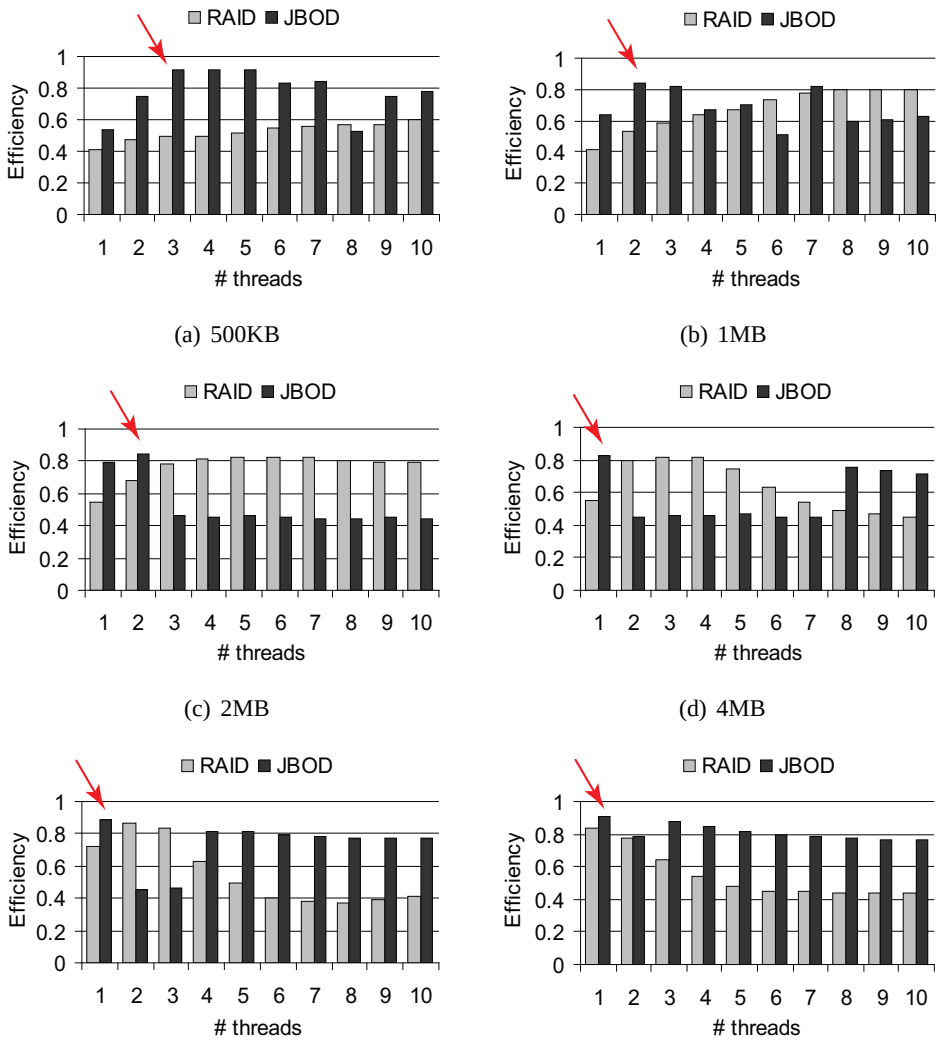


Figure 3: Comparison of RAID and JBOD

such performance by issuing sequential I/O requests. Unless the disk drive has read caching, one must also ensure that there are at least two outstanding I/Os at the disk controller (one being serviced and a second waiting to be serviced). Otherwise, a full disk rotation will be incurred for each I/O.

Let us now consider the multi-disk case. There are two options for discard-based search. One would be to manage the disks separately using JBOD, and another would be to use RAID. Again, because there is no particular order to the objects stored on disk, with JBOD, the runtime can independently manage each disk. For example, a worker thread can be assigned to each disk, so as to sequentially scan that disk's objects and pass them up to the filters. In the case of a cached retrieval, each worker can be given a list of objects to retrieve (objects that have previously passed a filter). Moreover, there could be multiple workers assigned to the same disk, each retrieving cached results on behalf of a particular client. Because the I/O scheduling is under control of the runtime in a JBOD configuration, the number of worker threads (level of concurrency) can be dynamically adjusted so as to make the most efficient use of each disk (e.g., by monitoring disk utilization and spawning/killing worker threads as necessary).

Although RAID presents a simpler programming model (i.e., it looks just like a single disk) and can provide data redundancy, there are no performance benefits relative to JBOD for sequential scans (both JBOD and RAID-0 will can see at most the full streaming bandwidth of the disks), and RAID can decrease performance

when concurrency is high (because of disk contention). The only case where a RAID-0 array will potentially outperform JBOD is in the case of low-concurrency during a cached retrieval (i.e., random I/O). Consider the case where one client is randomly retrieving previously filtered objects (whose object and disk IDs are stored in the result cache). With JBOD, the random I/O may be directed to only a subset of the disks. With RAID-0, the I/O will naturally be load-balanced over the entire disk array, and the low-concurrency will limit any disk contention among the clients. However, because high concurrency is the common case for discard-based search, this is opportunity will not occur often. Moreover, a JBOD configuration, with a bit more effort, can effectively achieve the same load-balancing of the random I/O, simply by selecting which disks to retrieve cached objects from (i.e., order does not matter).

In summary, our hypothesis is that discard-based search is comparable to other highly concurrent workloads, and can therefore achieve the best performance by using JBOD.

5.2 Results

To test our hypothesis, we ran the following experiments. Our goal was to find a configuration that keeps all of the disks busy at their full sequential bandwidth under our synthetic discard-based search workload. Thus, our figure of merit is efficiency: the ratio of the achieved bandwidth to the maximum sequential bandwidth of the array. We measured that each of our disks can sustain a maximum transfer rate of 74MB/s, so our aggregate maximum bandwidth is 370MB/s for the five-disk array.

We ran our synthetic discard-based search benchmark on our array configured as JBOD. In order to see the impact of concurrency, we varied the number of outstanding requests at each disk. We also tested various object sizes. In each case, we ran the benchmark to completion, fetching every object in the array.

The results appear in Figure 3. Each graph shows the efficiency of JBOD for a specific file size with increasing concurrency from left to right. The arrow shows the maximum efficiency achieved for the given file size. For each file size, the synthetic discard-based search workload was able to achieve a disk efficiency of at least 80% (for at least one of the levels of concurrency). In general, we found that the best efficiency occurred with one or two threads. Note, because the disks have read-ahead caches, it is not a requirement that each disk have more than 1 outstanding I/O in order to reach its full streaming bandwidth.

As a point of comparison, we ran the same tests with the disks configured as RAID-0 (results also shown in Figure 3). In most (65%) of the cases, JBOD outperformed RAID-0 when compared at the same level of concurrency, thereby supporting our hypothesis that JBOD is a more efficient disk organization for discard-based search.

6 Tenet: "Abort Is Rare"

As described in Section 3, the vast majority of discard-based search queries never run to completion, meaning that request abort is the rule rather than the exception. The usage pattern is one of iterative query refinement: a user issues a query, gets a few results, and then uses these results to refine the query. This process continues until the user has found the desired results or gives up.

Iterative query refinement is illustrated in Figure 4. A user wishes to find pictures of a friend's wedding from a shared photo collection. She begins her search by noting that wedding pictures are likely to contain faces. While the returned images contain faces, many of them are not from the wedding. She adds a color filter to match the wedding dress. Unfortunately, the color is not accurate and the new query returns no relevant images. The user then remembers that some of the wedding photos were taken outdoors. She replaces the wedding dress filter with a new green color filter to match grass. The new query returns a photo that includes the bride standing in front of the church. Based on the texture of the stone on the wall of the church, the user then creates another filter that, together with the face and grass filters, produces a satisfactory set of images from the wedding.

In experiments with users running search tasks over repositories of consumer digital photographs, only 16% of queries actually ran to completion [10]. Two of the search tasks emphasized recall, or fraction of relevant

Query Iterations

Q1 Face-detector
Q2 Face-detector ∧ Wedding-dress-color
Q3 Face-detector ∧ Grass-color
Q4 Face-detector ∧ Grass-color ∧ Stone-texture

Figure 4: Example of Iterative Query Refinement

objects retrieved against the number of relevant objects in the repository, implying an exhaustive search. Even for recall-oriented search, only 42% of the queries ran to completion. The remaining searches emphasized precision, or fraction of relevant objects retrieved against the number of retrieved objects. For these precision-oriented searches, 7% of the queries ran to completion. On average, users aborted their queries after viewing 36 objects, and the queries processed less than 10% of the objects in the repository.

7 Tenet: "The App Knows Best"

As described in Section 3, discard-based search applications access objects by collection, using an iterator model. Rather than reading a specific object, a discard-based search application simply requests the next object that passes the query. The runtime is free to present any such object that has not already been presented for the current search. Any-order semantics provides the runtime the flexibility to process objects in whatever order is most efficient for the storage device. In this section we describe and analyze a mechanism for shaping the I/O workload of discard-based search to achieve this efficiency.

7.1 Shaping Storage Traffic

The best I/O throughput can be achieved by reading objects sequentially starting with the object closest to the current position of the storage device. If multiple queries are started on the same collection of data, any-order semantics allows their read operations to be coalesced into a single request stream, thus avoiding the effects of contention at the storage device. This concept, called *bandwagon synchronization*, is illustrated in Figure 5.

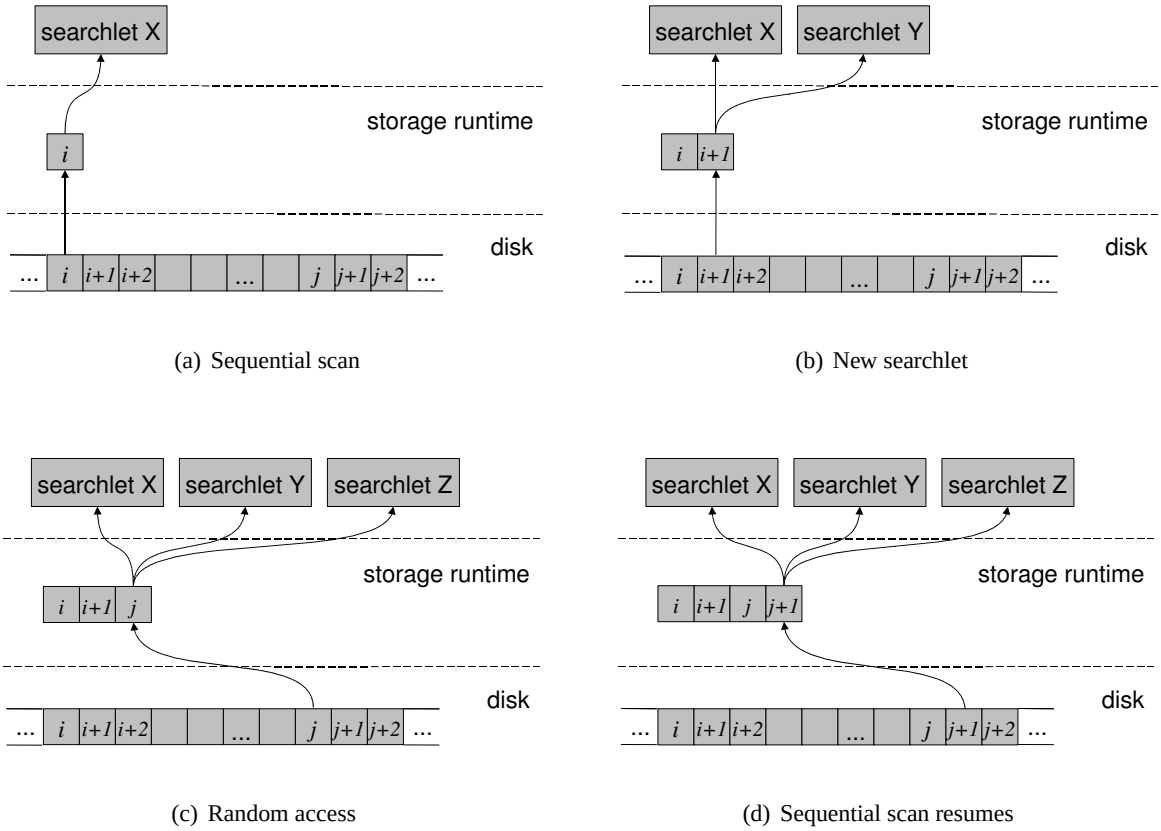
In Figure 5a, a query embodied in searchlet X processes objects retrieved from a sequential scan of the collection. In Figure 5b, a new query is started on the same collection, appearing on the server as searchlet Y. Because objects may be processed in any order, searchlet Y need not begin with the first object in the collection. Instead, it jumps on the I/O bandwagon at object $i+1$, retrieved for searchlet X. Figure 5c shows another query joining the fray, appearing as searchlet Z. Previously cached results indicate that object j passes searchlet Z. Any-order semantics allows the runtime to proceed directly to object j to satisfy the new query, and simultaneously make the object available to searchlets X and Y. In Figure 5d, the sequential scan resumes with object $j+1$.

7.2 Evaluating Bandwagon Synchronization

Figures 6 and 7 are conceptual illustrations of the number of objects delivered to the application by two queries, X and Y. The experiment is simple: query X starts at t_x and query Y starts at t_y. Query Y can either introduce its own I/O requests, as in Figure 6, or can synchronize (jump on the bandwagon) with query X, as in Figure 7. The graphs show the number of results produced as a function of time.

The time to deliver a result is determined by the sequential and random bandwidth the storage system can provide, the computational demands of the the searchlet, and the selectivity, or pass rate, of the query. The pass rates for queries X and Y are denoted p_x and p_y. In this example, $p_x \neq p_y$. Given a maximum sequential read rate of B bytes per second, the retrieval time for an object of size f is simply $t_{seq} = f/B$. Accessing data nonsequentially (randomly) will incur an average seek and rotational latency period, t_{rp}. When accessing data randomly, the time to fetch an object is $t_{rand} = t_{rp} + f/B$.

Figure 6 shows the case in which the two queries are not synchronized. Before t_y, query X reads data sequentially. After query Y begins, requests from both queries will incur a random seek (t_{rand}) for each access,



(a) Sequential scan

(b) New searchlet

(c) Random access

(d) Sequential scan resumes

Figure 5: Bandwagon Synchronization

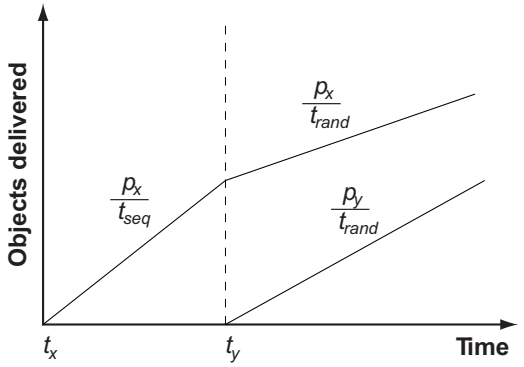


Figure 6: Independent Request Streams

lowering the bandwidth. Neither X nor Y is able to read sequentially at that point. Figure 7 illustrates the two queries with bandwagon synchronization. That is, query Y is able to share the objects that query X fetches. In this case, both queries can proceed at the sequential read rate with no mutual interference.

7.3 Results

We compared independent request streams and bandwagon synchronization using the experimental setup described in Section 4. The workload generator ran on a single disk. Two queries were started five seconds apart. The queries had equal selectivity, but did not pass the same objects. In the independent synchronization case, each query employed its own reader thread that scanned the repository from the beginning. In the bandwagon synchronization case, the queries were serviced from a single reader thread that scanned the repository from the

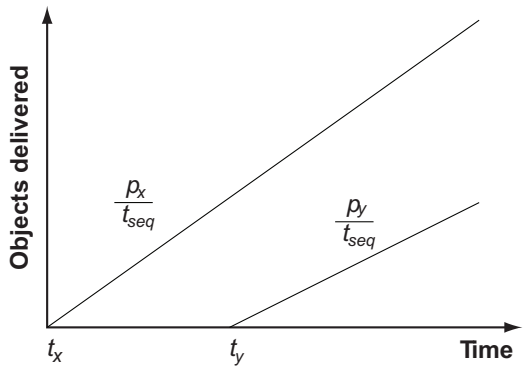


Figure 7: Synchronized Request Streams

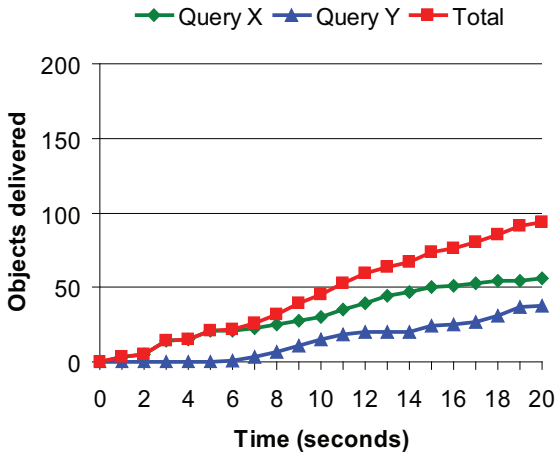


Figure 8: Independent Request Streams

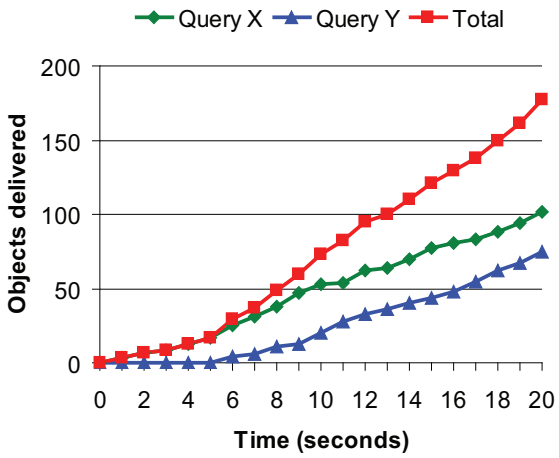


Figure 9: Bandwagon Synchronization

beginning.

Figures 8 and 9 show the total objects returned as a function of time for independent request streams and bandwagon synchronization, respectively, for 1MB objects, query pass rates of 10%, and no searchlet computation time. In Figure 8, the rate of return for query *X* flattens after query *Y* is started. Both queries then return objects at essentially the same rate, limited by random storage bandwidth. In Figure 9, both queries are able to

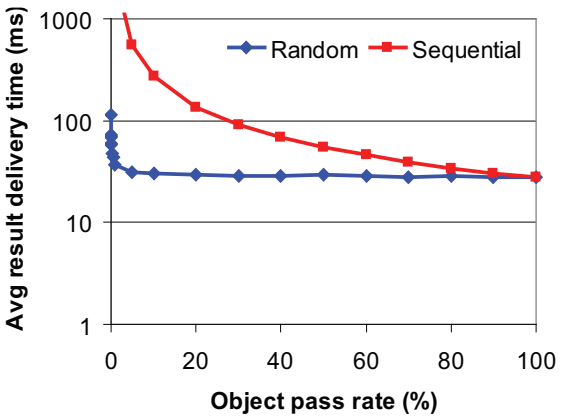


Figure 10: Response Time for Sequential and Random Scans

take advantage of sequential bandwidth, resulting in a twofold improvement over independent request streams. In the next section, we will introduce a variant of bandwagon synchronization and present results for additional file sizes and pass rates.

8 Tenet: "Seeks Are Evil"

Conventional wisdom holds that the best performance from storage is achieved by making I/O requests as sequential as possible. Random access is to be avoided because of the overhead associated with seeks and rotational delay. The any-order semantics of discard-based search allow data to be searched by sequentially reading and filtering images from the storage system. However, results of previous searches are also stored in a result cache. If a search or part of a search is re-run, these results can be read directly (randomly), rather than waiting for passing data to be delivered via sequential scan.

In this section, we examine whether or not it is worth interrupting sequential accesses for opportunistic random accesses to objects known to satisfy a query according to the result cache. We examined two aspects of this question. First, what is the relative cost of sequential and random access? Second, how does random access impact bandwagon synchronization?

8.1 Sequential vs. Random

Because discard-based search is intended to support interactive data exploration, the dominant performance consideration is response time rather than throughput. Users need to see results before they can judge the success of the their queries and determine how to refine them. Figure 10 compares the average time to retrieve objects satisfying a query as a function of pass rate using sequential and random scans of storage. The workload is a search over 10MB objects stored in the JBOD described in Section 4. Response times are shown on a log scale and reflect only the storage system's performance. They do not include processing time for the filters in the searchlet. At a pass rate of 100%, random and sequential accesses are identical, and so achieve the same performance of around 28ms per object. As the pass rate decreases, the time for both sequential and random scans to retrieve objects increases. For sequential access, the response time increases for the simple reason that fewer objects pass the searchlet. As the pass rate drops below 1%, response time increases from 2.7 seconds to over 18 seconds for a pass rate of 0.1%.

The time to retrieve objects using a random scan also increases as pass rates decrease, but not for the same reason as the sequential scan. The random scan retrieves objects that are known to satisfy the query according to the result cache. Passing objects are uniformly distributed over the entire repository. Therefore, the pass rate determines how long a seek is required to reach the next pass. The curve tips upward only at the smallest pass rates, to 113ms for a rate of 1 in 50000. This is essentially the time of the longest possible seek and the time to transfer the data.

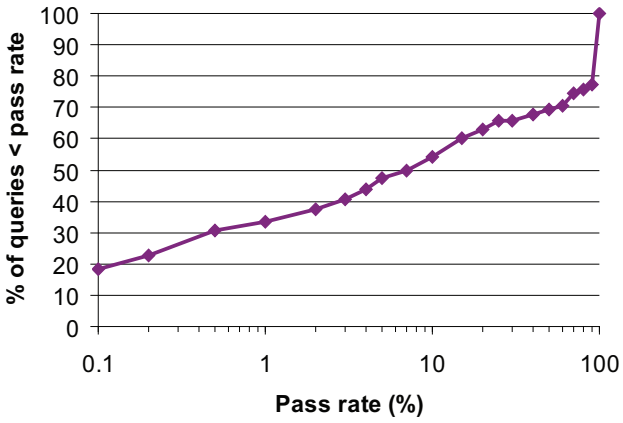


Figure 11: Selectivity of Queries for Digital Photo Searches

Such low pass rates are not unusual. Figure 11 shows the pass rates for user queries from an empirical study of digital photo searches [10]. The graph is read as follows. For a given pass rate P on the X axis (shown on a log scale), the Y value is the percentage of filters defined that have pass rates less than P. The jump at 100% reflects browsing activity. The figure shows that over half of the queries had pass rates of less than 10%, and nearly 20% of the queries had pass rates less than 0.1%.

For searches that are selective, it may be faster to seek to objects that are known to satisfy the query. Such objects must still be retrieved for presentation to the user. Presentation could involve the data itself (e.g., an image that matches the query), or a derivation of the data (e.g., locations of interesting features).

8.2 Seeks and Bandwagon Synchronization

From Figure 10, we can conclude that if cached results are available, it is always worth the cost of a random access to retrieve an object known to satisfy a query. The storage access pattern for a query then proceeds as follows. When the query starts, the filter cache is scanned to determine if there are any passing objects, and such objects are retrieved. The remaining objects are then scanned sequentially starting from the last passing object as indicated by the result cache.

Figure 12 incorporates greedy retrieval of known passes into bandwagon synchronization. For simplicity, the greedy retrieval phase is shown only for query Y. When query Y begins, its passing objects are retrieved in a period between t_y and t_c lasting $n_c \cdot t_{rand}$, where n_c is the number of cached passes for query Y. These objects are also available to query X, generating an additional $p_x \cdot (t_c - t_y)/t_{rand}$ passing objects. Once the greedy retrieval phase completes, both queries resume sequential scans.

The greedy retrieval phase for a new query imposes a penalty on the queries in progress N of

$$(t_c - t_y) \sum_{i=1}^{N} (p_i/t_{seq} - p_i/t_{rand}).$$

For the example in Figure 12, the gain from greedy retrieval exceeds the penalty when

$$p_x/t_{rand} + 1/t_{rand} > p_x/t_{seq} + p_y/t_{seq}.$$

In other words, the selectivity of p_y must be sufficiently small for the passes generated in the greedy retrieval phase to outweigh the decrease in passes for the other queries because of the overhead of random access. File size and the relative performance of seek and transfer rates for the storage device determine the extent of this overhead.

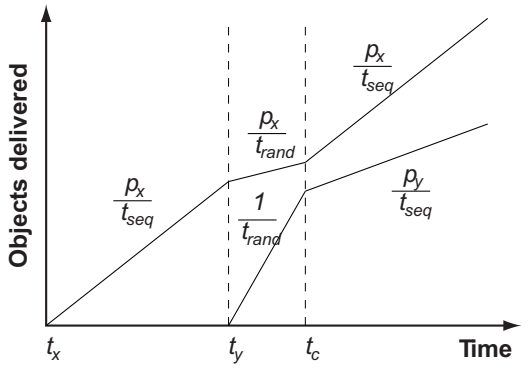


Figure 12: Bandwagon Synchronization with Greedy Retrieval

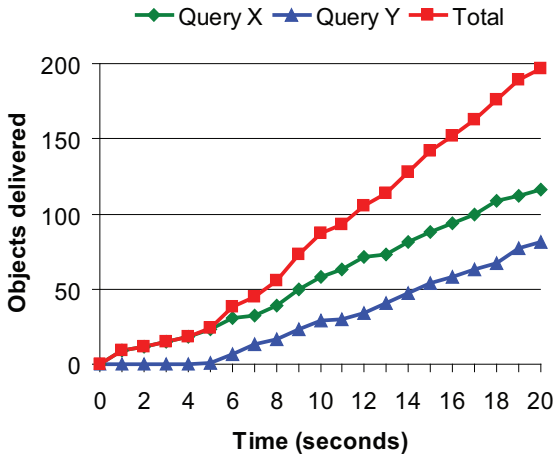


Figure 13: Bandwagon Synchronization with Greedy Retrieval

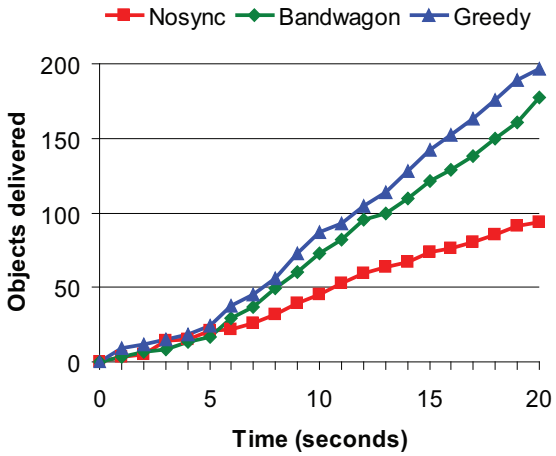
8.3 Results

To compare greedy retrieval to the strategies described earlier, we used the same experimental setup as in Section 7.3. This time, at the start of each query, the reader thread performed a random retrieval of ten passing objects distributed uniformly over a 200GB disk. Figure 13 shows the total objects returned as a function of time for bandwagon synchronization with greedy retrieval. As in Figures 8 and 9, the object size was 1MB, and both queries had pass rates of 10%. The improvement from greedy retrieval is visible as bump in the number of objects retrieved at times 0 and 5.

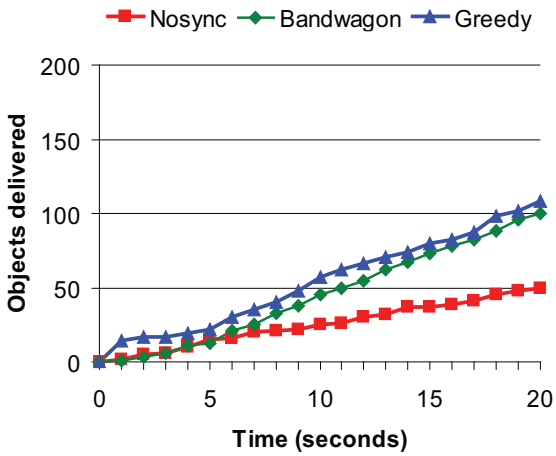
Figure 14 shows the total objects retrieved for all three strategies at five different object sizes and a pass rate of 10%. Greedy bandwagon synchronization performs best in all cases except for the 4MB object size. In this case, we found that after the greedy retrieval phase, the sequential scan started deep into the disk, where transfer rates are lower because of zoned bit recording (ZBR). For the other strategies, reads always start at the outer tracks of the disk. This positioning is an artifact of our experimental setup. In practice, independent and bandwagon retrievals will be distributed more evenly over the disk.

As object size increases, so does the time to retrieve passing objects. Transfer time dominates retrieval of large objects, diminishing the effect of seeks for both independent streams and queries already in progress during greedy retrieval. The effect of greedy retrieval is most visible on the 16MB graph at times 0 and 5.

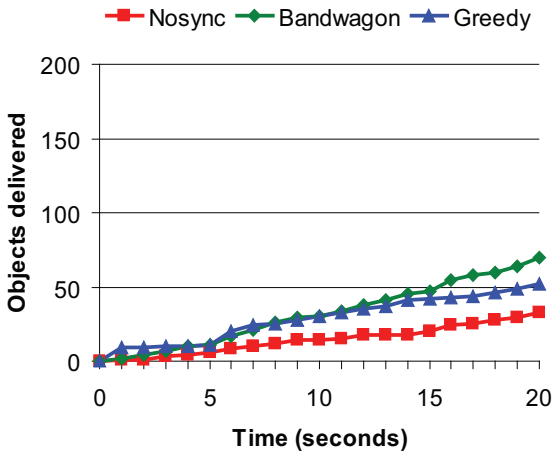
Figure 15 shows the effect of varying the query selectivity for 1MB files. At high pass rates, the benefits of greedy retrieval are diminished, because sequential scan is likely to produce objects in the time it takes to seek to the location of a known pass. Greedy retrieval is most valuable at low pass rates, as shown in Figure 15b, where it produces the fast response times needed for discard-based search.



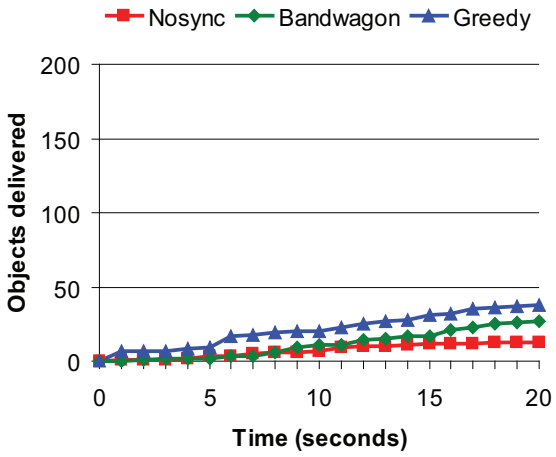
(a) 1MB objects, pass rate 10%



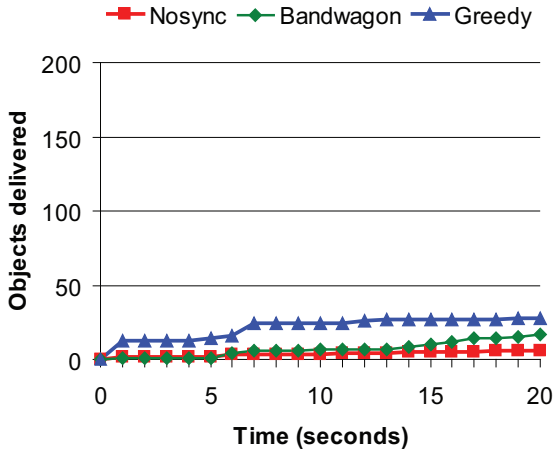
(b) 2MB objects, pass rate 10%



(c) 4MB objects, pass rate 10%

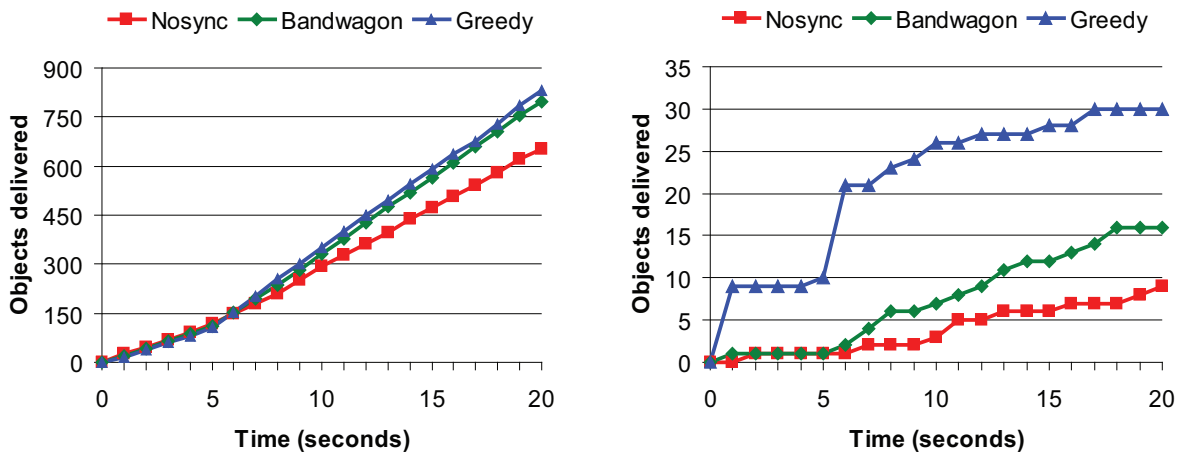


(d) 8MB objects, pass rate 10%



(e) 16MB objects, pass rate 10%

Figure 14: Comparison of Synchronization Strategies



(a) 1MB objects, pass rate 50%

(b) 1MB objects, pass rate 1%

Figure 15: Effect of Query Selectivity

9 Tenet: "Real Work Beats Speculation"

It is conventional wisdom that speculative work should always take a backseat to foreground work. This tenet is especially true in storage systems, as the cost of mis-speculation is very high: usually at least the cost of a disk rotation. As a result, storage systems tend to be quite conservative when it comes to speculation.

Discard-based search can benefit from several forms of speculative execution [10]. Intra-query speculation uses periods of user think time to perform additional processing on objects. Inter-query speculation performs queries during idle periods consisting of predicates likely to benefit future queries. If there is communal locality among queries, then these speculation schemes can produce results likely to be used in future queries.

Intra-query speculation does not generate any additional I/O - it generates additional processing for objects already retrieved. Inter-query speculation does generate additional I/O, but only during idle periods. Since objects are processed one at a time, there are many convenient points for interrupting speculative work. Furthermore, bandwagon synchronization allows demand queries to synchronize on the objects retrieved by inter-query speculation. From the perspective of the storage system, this kind of computational speculation poses no risk, and is limited only by the CPU parallelism available.

10 Kaiten-zushi Storage

Given the workload characteristics described in Section 3, and their implications discussed in Sections 5 to 9, what would be an ideal storage design for discard-based search? We suggest that an apt metaphor for such a design is that of a Japanese "conveyor belt sushi" or "kaiten-zushi" restaurant, as described in Wikipedia [30]:

> *Kaiten-zushi is a sushi restaurant where the plates with the sushi are placed on a rotating conveyor belt that winds through the restaurant and moves past every table and counter seat. Customers may place special orders, but most simply pick their selections from a steady stream of fresh sushi moving along the conveyor belt.*

The strongly sequential nature of a discard-based search workload gives rise to the metaphor, as keeping disk requests sequential is straightforward. We assign to each disk in the system a single reader thread that continually accesses each object in order. This sequential fetch process is the equivalent of the conveyor belt carrying sushi to patrons. As objects are fetched from disk, filters for the active queries are applied to each object. CPU-bound filters (the equivalent of patrons who eat sushi more slowly than it arrives) will process objects less quickly than they are fetched from disk, but will always have a new object available to process

when they complete. I/O-bound filters (the equivalent of patrons who can sushi more quickly than it arrives) will be able to process each object as it is fetched from disk.

Queries for which there are cached results can be serviced by seeking the disk directly to the location of those results. At first glance, it would seem that a seek-based fetch process that favors cached results would result in better performance than a sequential fetch process. However, if a query is not highly selective, then it is often more efficient to process sequentially-fetched objects for which there are no cached results as they are fetched from disk rather than seek to the locations of cached results. In the metaphor, processing cached results is equivalent to placing special sushi orders rather than waiting for a particular piece to arrive on the conveyor belt. (A more extreme version would be a patron getting up from his seat and quickly moving to another seat in front of which is his desired piece of sushi.) If a patron is very selective, a particular piece of sushi could arrive more quickly by special order. On the other hand, if a patron is less selective, then he can be satisfied more quickly by just waiting for sushi from the conveyor belt.

11 Related work

Multimedia information retrieval (MIR) encompasses a variety of disciplines which relate to storing and retrieving digital content, particularly images and video. Content-based image retrieval (CBIR), one such sub-discipline, focuses on retrieval techniques which filter or otherwise process digital content. Such techniques are advantageous when text annotations are either incomplete, or nonexistent.

A variety of content-based image retrieval systems have been developed in recent years. Lew *et al.* provide an excellent survey of CBIR and the current state-of-the-art [18]. CANDID [17], QBIC [9], and ASSERT [26] are notable examples. In each of these systems, images are preprocessed in order to efficiently satisfy subsequent queries. More specifically, in the CANDID system, images are searched by providing an example image. A distance metric is calculated between each image in the database (e.g., by color, shape, and texture) and the example image. Similarly, ASSERT uses computer vision and imaging processing algorithms to automatically extract image attributes, and a multi-dimensional index is built, based on these attributes. In the QBIC system, users provide example images, selected color and texture patterns, and even sketches or drawings that are representative of the desired images.

Diamond, in contrast to existing CBIR systems, attempts to build an image repository that enables efficient interactive search of large volumes of complex, non-indexed data. As previously discussed, it is the unique workload characteristics of Diamond that suggests a rethinking of the appropriate storage system design.

Naturally, matching application workload characteristics to storage system designs has a long and rich history in computer systems. Early examples include the use of empirical observations of file sizes and lifetimes [24] in the design of AFS [13, 25], the use of workload data reported by Ousterhout *et al.* [23] in the design of the Sprite file system [22], and optimizations for read-only workload characteristics in the design of the FileNet file system [7]. More recently, systems have begun to automatically make storage configuration decisions normally assigned to an administrator, including the selection of the proper RAID configuration for a storage array [4, 31], the assignment of workloads to a collection of storage arrays [2, 3], and the selection of a data distribution policy for a given workload (e.g., replication versus erasure coding)[1].

This paper follows in this long tradition by identifying the unique workload characteristics of content-based image retrieval. More precisely, it explores the new class of discard-based search applications and the implications for a new storage system design.

12 Conclusion

This paper discussed the unique workload characteristics of discard-based search and how one must re-think the proper storage system design for such a workload. It was shown that the highly concurrent, sequential, and read-only characteristics of discard-based search make JBOD preferable over RAID. Experiments with representative workloads confirmed that a JBOD organization can achieve at least 80% of a disk array's potential

bandwidth. Further, because there is no specific order imposed on the retrieved images, a newly introduced I/O coalescing technique, bandwagon scheduling, can be used to substantially improve the effective throughput of the system. The benefits of bandwagon synchronization were quantified in the context of various images sizes, filter selectivity, and object access patterns (sequential versus random). In addition, discard-based search introduces additional opportunities in terms of speculative filtering. Unlike most storage applications, the characteristics of discard-based allow one to speculate for free (i.e., without affecting foreground traffic).

Acknowledgements

OpenDiamond is a registered trademark of Carnegie Mellon University.

References

[1] ABD-EL-MALEK, M., II, W. V. C., CRANOR, C., GANGER, G. R., HENDRICKS, J., KLOSTERMAN, A. J., MESNIER, M., PRASAD, M., SALMON, B., SAMBASIVAN, R. R., SINNAMOHIDEEN, S., STRUNK, J. D., THERESKA, E., WACHS, M., AND WYLIE, J. J. Ursa Minor: versatile cluster-based storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies* (San Francisco, CA, December 2005), The USENIX Association.

[2] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. Hippodrome: running circles around storage administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)* (Monterey, CA, January 2002), The USENIX Association.

[3] ANDERSON, E., KALLAHALLA, M., SPENCE, S., SWAMINATHAN, R., AND WANG, Q. Quickly finding near-optimal storage system designs. *ACM Transactions on Computer Systems (TOCS) 23*, 4 (November 2005), 337–374.

[4] ANDERSON, E., SWAMINATHAN, R., VEITCH, A., ALVAREZ, G. A., AND WILKES, J. Selecting RAID levels for disk arrays. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)* (Monterey, CA, January 2002), The USENIX Association.

[5] BERCHTOLD, S., BOEHM, C., KEIM, D., KRIEGEL, H. A Cost Model for Nearest Neighbor Search in High-Dimensional Data Space. In *Proceedings of the Symposium on Principles of Database Systems* (Tucson, AZ, May 1997).

[6] CHEN, P., AND PATTERSON, D. Maximizing performance in a striped disk array. In *Proc. 17th Annual Int'l Symp. on Computer Architecture, ACM SIGARCH Computer Architecture News* (1990), p. 322.

[7] DAVID A. EDWARDS, M. S. M. Exploiting read-mostly workloads in the filenet file system. In *Proceedings of the 12th ACM Symposium on Operating System Principles (SOSP 89)* (Litchfield Park, AZ, December 3–6 1989), ACM Press.

[8] DUDA, R., HART, P., STORK, D. *Pattern Classification.* Wiley, 2001.

[9] FLICKNER M, SAWHNEY H, NIBLACK W, ASHLEY J, HUANG Q, DOM B, GORKANI M, HAFNER J, LEE D, PETKOVIC D, STEELE D, YANKER P. Query by Image and Video Content: The QBIC System. *IEEE Computer 28*, 9 (September 1995).

[10] GIBBONS, P., MUMMERT, L., SUKTHANKAR, R., SATYANARAYANANA, M. Just-In-Time Indexing for Interactive Data Exploration. Tech. Rep. CMU-CS-07-120, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, April 2007.

[11] GOODE, A., CHEN, M., TARACHANDANI, A., MUMMERT, L., SUKTHANKAR, R., HELFRICH, C., STEFANNI, A., FIX, L., SALTZMANN, J., SATYANARAYANAN, M. Interactive Search of Adipocytes in Large Collections of Digital Cellular Images. In *Proceedings of the 2007 IEEE International Conference on Multimedia and Expo (ICME07)* (Beijing, China, July 2007).

[12] HAUSER, S. E., BERMAN, L. E., AND THOMA, G. R. Is the bang worth the buck?: a raid performance study. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems and Technologies* (College Park, MD, September 1996 1996), IEEE.

[13] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS) 6*, 1 (February 1988), 51–81.

[14] HUSTON, L., SUKTHANKAR, R., CAMPBELL, J., AND PILLAI, P. Forensic Video Reconstruction. In *Proceedings of International Workshop on Video Surveillance and Sensor Networks* (2004).

[15] HUSTON, L., SUKTHANKAR, R., HOIEM, D., AND ZHANG, J. SnapFind: Brute force interactive image retrieval. In *Proceedings of International Conference on Image Processing and Graphics* (2004).

[16] HUSTON, L., SUKTHANKAR, R., WICKREMESINGHE, R., SATYANARAYANAN, M., GANGER, G.R., RIEDEL, E., AILAMAKI, A. Diamond: A Storage Architecture for Early Discard in Interactive Search. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (San Francisco, CA, April 2004).

[17] KELLY, P. M., CANNON, M. T., AND HUSH, D. R. Query by image example: the comparison algorithm for navigating digital image databases (candid) approach. In *Proceedings of the Conference on Storage and Retrieval for Image and Video Databases (SPIE 1995)* (San Diego, CA, February 24–27 1995), The International Society for Optical Engineering (SPIE).

[18] LEW, M. S., SEBE, N., DJERABA, C., AND JAIN, R. Content-based multimedia information retrieval: State of the art and challenges. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP) 2*, 1 (February 2006), 1–19.

[19] MERRIAM-WEBSTER. Merriam-webster online search, 2007. [Online; accessed 8-September-2007].

[20] MINKA, T., PICARD, R. Interactive Learning Using a Society of Models. *Pattern Recognition 30* (1997).

[21] NECULA, G. C., AND LEE, P. Safe Kernel Extensions Without Run-Time Checking. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 1996).

[22] NELSON, M. N., WELCH, B. B., AND OUSTERHOUT, J. K. Caching in the sprite network file system. *ACM Transactions on Computer Systems (TOCS) 6*, 1 (February 1988), 134–154.

[23] OUSTERHOUT, J. K., COSTA, H. D., HARRISON, D., KUNZE, J. A., KUPFER, M. D., AND THOMPSON, J. G. A trace-driven analysis of the unix 4.2 bsd file system. In *Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP 85)* (Orcas Island, WA, December 1–4 1985), ACM Press.

[24] SATYANARAYANAN, M. A study of file sizes and functional lifetimes. In *Proceedings of the 8th ACM Symposium on Operating System Principles (SOSP 81)* (Pacific Grove, CA, December 14–16 1981), ACM Press.

[25] SATYANARAYANAN, M., HOWARD, J. H., NICHOLS, D. A., SIDEBOTHAM, R. N., SPECTOR, A. Z., AND WEST, M. J. The itc distributed file system: Principles and design. In *Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP 85)* (Orcas Island, WA, December 1–4 1985), ACM Press.

[26] SHYU, C.-R., BRODLEY, C. E., KAK, A. C., KOSAKA, A., AISEN, A. M., AND BRODERICK, L. S. ASSERT: A physician-in-the-loop content-based retrieval system for HRCT image databases. *Computer Vision and Image Understanding (CVIU) 75*, 1–2 (July/August 1999), 111–132.

[27] VON AHN, L., AND DABBISH, L. Labeling images with a computer game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (April 2004).

[28] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient Software-based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, NC, December 1993).

[29] WALLACH, D. S., BALFANZ, D., DEAN, D., AND FELTEN, E. W. Extensible Security Architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems and Principles* (Saint-Malo, France, October 1997).

[30] WIKIPEDIA. Conveyor belt sushi — Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 3-September-2007].

[31] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems (TOCS) 14*, 1 (February 1996), 108–136.

[32] YANG, L., JIN, R., SUKTHANKAR, R., ZHENG, B., MUMMERT, L., SATYANARAYANAN, M., CHEN, M., AND JUKIC, D. Learning Distance Metrics for Interactive Search-Assisted Diagnosis of Mammograms. In *Proceedings of SPIE Medical Imaging* (2007).

[33] YAO, A., YAO, F. A General Approach to D-Dimensional Geometric Queries. In *Proceedings of the Annual ACM Symposium on Theory of Computing* (May 1985).