

The unique strengths and storage access characteristics of discard-based search

Mahadev Satyanarayanan · Rahul Sukthankar · Lily Mummert · Adam Goode · Jan Harkes · Steve Schlosser

Received: 26 January 2010 / Accepted: 2 February 2010 / Published online: 24 February 2010
© The Author(s) 2010. This article is published with open access at Springerlink.com

Abstract *Discard-based search* is a new approach to searching the content of complex, unlabeled, nonindexed data such as digital photographs, medical images, and real-time surveillance data. The essence of this approach is *query-specific content-based computation, pipelined with human cognition*. In this approach, query-specific parallel computation shrinks a search task down to human scale, thus allowing the expertise, judgment, and intuition of an expert to be brought to bear on the specificity and selectivity of the search. In this paper, we report on the lessons learned in the *Diamond project* from applying discard-based search to a variety of applications in the health sciences. From the viewpoint of a user, discard-based search offers unique strengths. From the viewpoint of server hardware and software, it offers unique opportunities for optimization that contradict long-established tenets of storage design. Together, these distinctive end-to-end attributes herald a new genre of Internet applications.

Keywords Data-intensive computing · Non-text search technology · Medical image processing · Interactive search · Computer vision · Pattern recognition · Distributed systems · ImageJ · MATLAB · Parallel processing · Human-in-the-loop · Diamond · OpenDiamond · Storage systems · I/O workloads · RAID

M. Satyanarayanan (✉) · A. Goode · J. Harkes
Carnegie Mellon Univ., Pittsburgh, PA, USA
e-mail: satya@cs.cmu.edu

R. Sukthankar · L. Mummert
Intel Labs Pittsburgh, Pittsburgh, PA, USA

S. Schlosser
Avere Systems, Pittsburgh, PA, USA

1 Introduction

Today, “search” and “indexing” are almost inseparable concepts. The success of indexing in Web search engines and relational databases has led to a mindset where search is impossible without an index. Unfortunately, there are situations where we do not know how to build an efficient index. This is especially true for complex data such as digital photographs, medical images, real-time surveillance images, and sound recordings. The need exists to search such data now rather than waiting for indexing technology to catch up.

For example, consider Fig. 1, showing two examples of lip prints from thousands collected worldwide by efforts to discover the genetic origins of cleft palate syndrome. Of the many visual differences between the left image (control) and the right image (from a family with cleft palate members), which are the features predictive of the genetic defect? What search tools can a medical researcher use today to explore a large collection of lip prints to answer this question?

Another example pertains to drug discovery in the pharmaceutical industry. Figure 2 shows two examples of neuronal stem cell images from a multi-week drug toxicity experiment that generates thousands of such cell microscope images per hour. The left image is expected under normal growth conditions. The right image is an anomaly, possibly



Fig. 1 Examples of lip prints in craniofacial research

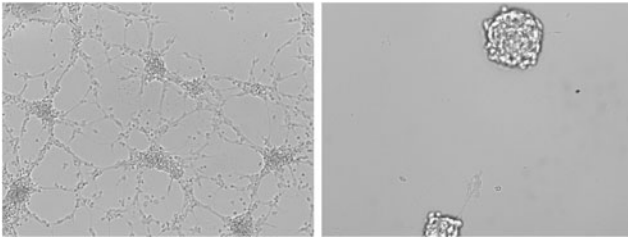


Fig. 2 Examples of neuronal stem cell growth

indicating experimental error. What tools can a researcher use to discover such anomalies in real time, to see if they have occurred before, and to possibly abort the experiment?

To address such real-world needs, we are exploring a new approach called *discard-based search*. The essence of this approach is *query-specific content-based computation, pipelined with human cognition*. More specifically, we use query-specific parallel computation on large collections of complex data spread across multiple Internet servers to shrink a search task down to human scale. The expertise, judgment, and intuition of the user performing the search can then be brought to bear on the specificity and selectivity of the current search. Our focus on interactive search, with a human expert such as a doctor, medical researcher, law enforcement officer, or military analyst in the loop, means that *user attention* is the most precious system resource. Making the most of available user attention is far more important than optimizing for server CPU utilization, disk traffic, network bandwidth, or other system metrics.

We have been exploring this new search paradigm in the *Diamond* project since late 2002 and have gained experience in applying it to the health sciences. We now have a deeper understanding of the unique attributes of discard-based search from a user perspective. We also realize that the server workloads induced by discard-based search invalidate many tenets of storage design. We focus on these lessons and insights in this paper.

Over time, we have learned how to cleanly separate the domain-specific and domain-independent aspects of discard-based search, encapsulating the latter into Linux middleware that is based on standard Internet component technologies. This open-source middleware is called the *OpenDiamond® platform* for discard-based search (<http://diamond.cs.cmu.edu>). For ease of exposition, we use the term “Diamond” loosely in this paper: as our project name, to characterize our approach to search (“the Diamond approach”), to describe the class of applications that use this approach (“Diamond applications”), and so on. However, the term “OpenDiamond platform” always refers specifically to the open-source middleware.

From an end-to-end Internet perspective, the OpenDiamond platform offers a new way to connect the owners of complex data and the domain experts who may derive value

from searching that data. At one end, as discussed in Sect. 4, it opens the door to search applications that take full advantage of the domain expertise of users. At the other end, as discussed in Sect. 5, it helps to overcome long-entrenched constraints of storage system design. Together, these distinctive attributes suggest the emergence of a new genre of Internet applications with valuable end-to-end properties.

2 Background and related work

2.1 Early discard

Without an index, brute-force search is the only way to separate relevant and irrelevant data. The efficiency with which data objects can be examined and rejected then becomes the key figure of merit. The Diamond approach was inspired by work published in the late 1990s on active disks [1, 16, 23]. These diverse investigations suggested that application-level processing close to storage offers significant performance benefits. In particular, they suggested the feasibility of *early discard* or the application-specific rejection of irrelevant data as early as possible in the pipeline from storage to user. This performance optimization, illustrated in Fig. 3, improves scalability by eliminating a large fraction of the data from most of the pipeline. Most discards are done by application-specific software at the extreme right, very close to where the data is stored (at the label “early discard”). Additional discards by application-specific software may happen closer to the user (at the label “late discard”).

The application-specific code in Diamond that performs early discard is called a *searchlet*. While a searchlet can be a monolithic piece of code, it is more typically composed of individual components called *filters*. For example, a photo search application may provide filters for face detection, color detection, and texture detection. Filters embody specific dimensions of domain-specific knowledge, while searchlets represent the combination of attributes being sought by the current search.

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

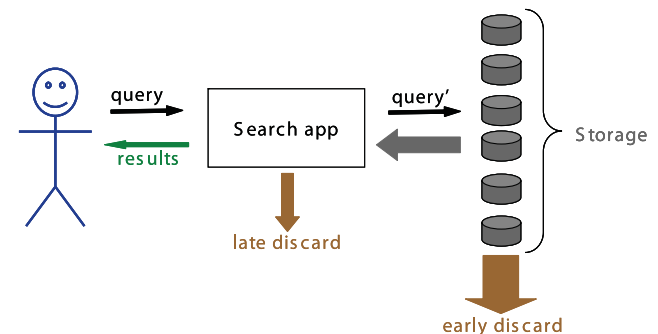


Fig. 3 Early discard optimization

false-positives (irrelevant data that is not rejected) and false-negatives (relevant data that is incorrectly discarded) [8]. The best one can do 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. A Diamond user has direct control over these trade-offs.

2.2 Diamond architecture

As Fig. 4 illustrates, the Diamond architecture cleanly separates domain-specific code from the domain-independent runtime system. The OpenDiamond platform consists of domain-independent client and server runtime software, the APIs to this runtime software, and a TCP-based network protocol. On a client machine, user interaction typically occurs through a domain-specific GUI.

The searchlet is composed and presented by the application through the *Searchlet API*, and is distributed by Diamond to all of the servers involved in the search task. A searchlet describes a precedence graph of filters that exposes the available degrees of freedom in scheduling and parallel execution to Diamond. 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 that is thresholded to determine whether a given object should be discarded or passed to the next filter. Only those objects that pass through the entire gauntlet of filters in the searchlet are transmitted to the client.

A key architectural constraint of Diamond is that servers do not communicate directly with each other. This simplifies access control in multi-enterprise searches. If a user has privileges to search servers individually in different enterprises, she is immediately able to conduct searches that span those servers.

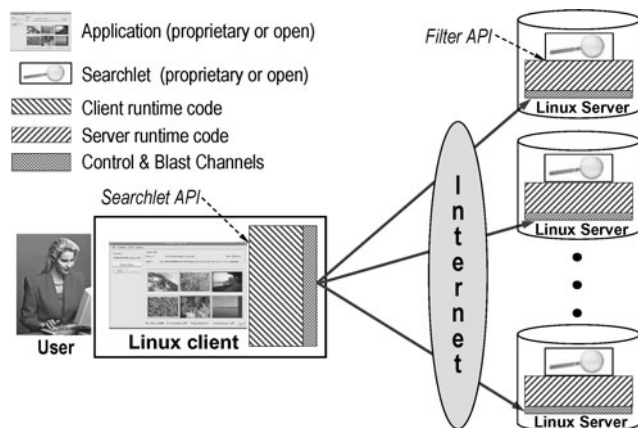


Fig. 4 System architecture

As discussed in our early Diamond paper [15], self-tuning mechanisms on servers can be used to improve the efficiency of discard-based search. First, queue back-pressure on the result stream can be used for dynamic load balancing in searchlet execution between server and client. Second, application-transparent runtime monitoring of the computational cost and selectivity of filters can be used for dynamic adaptation of searchlet execution. Such adaptation can help achieve earliest discard of data objects at least cost, without requiring data-specific or application-specific knowledge.

Domain-specific tools such as *ImageJ* and *MATLAB* can be used to create Diamond filters. ImageJ is an image processing tool that is widely used by cell biology researchers. MATLAB is a matrix language that is widely used in many domains. As Diamond matures, we expect to include other domain-specific tools.

Diamond allows information stored in a structured data source, such as a relational database, to constrain discard-based search. Consider a query such as “From women aged 40–50 who are smokers, find mammograms that have a lesion similar to this one.” Age and personal habits are typically found in a patient record database, while lesion similarity requires discard-based search of mammograms. We refer to this external constraining of data objects as *scoping a discard-based search*. Accurate scoping can greatly reduce the amount of work that has to be done by discard-based search.

The client–server network protocol separates control from data. For each server involved in a search, there is pair of TCP connections between that server and the client. This has been done with an eye to the future, when different networking technologies may be used for the two channels in order to optimize for their very different traffic characteristics. Responsiveness is critical on the control channel, while high throughput is critical on the data channel, referred to as the *blast channel*.

2.3 Result and attribute caching on servers

A typical Diamond search session exhibits significant overlap in filters and parameters across steps of the search. This temporal locality can be exploited by caching filter execution results at servers. Over time, cache entries will be created for many objects on frequently used combinations of filters and parameters. This reduces the speed differential with respect to indexed search and can be viewed as a form of *just-in-time indexing* that is performed incrementally at run time as a side-effect of discard-based searches. It is completely transparent to users, applications, and filters.

Caching on servers takes two different forms: *result caching* and *attribute caching*. An attribute is typically created as a side-effect of filter execution for the benefit of downstream filters or the application on the client. An example of an attribute is the set of locations of faces found by a

face detection filter that is upstream from a face recognition filter. Both forms of caching are application-transparent and invisible to clients except for improved performance. Both caches are persistent across server reboots and are shared across all users. Thus, users can benefit from each others' search activities without any coordination. The sharing of knowledge within an enterprise can give rise to significant communal locality in filter executions.

Result caching allows a server to remember the outcomes of object–filter–parameter combinations. Since filters consist of arbitrary code and parameters, we use a cryptographic hash of the filter code and parameter values to generate a fixed-length cache tag. Note that cache entries are very small (few tens of bytes each) in comparison to typical object sizes.

Attribute caching is the other form of caching in Diamond. Hits in the attribute cache reduce server load and improve performance. We use an adaptive approach for attribute caching because some intermediate attributes can be costly to compute, while others are cheap. Some attributes can be very large, while others are small. It is pointless to cache attributes that are large and cheap to compute, since this wastes disk space and I/O bandwidth for little benefit. An example of such an attribute is the decompressed bitmap of a JPEG image. The most valuable attributes to cache are those that are small but expensive to generate, such as the pixel coordinates of faces discovered by a face detection filter. To implement this policy, the server runtime system dynamically monitors filter execution times and attribute sizes. Only attributes below a certain space-time threshold (currently 1 MB of size per second of computation) are cached.

2.4 Relationship to MapReduce and other work

Diamond is the first system to unify the distinct concerns of interactive search and complex, nonindexed data. Data complexity motivates pipelined filter execution, early discard, self-tuning for filter execution order, the ability to use external domain-specific tools such as ImageJ and MATLAB, and the ability to use external meta-data to scope searches. Concern for crisp interaction motivates caching of results and attributes at servers, streamlining of result transmission, self-tuning of searchlets, and separation of control and blast channels in the network protocol.

The dissemination and parallel execution of searchlet code at multiple servers bears some resemblance to the execution model of MapReduce [6, 7]. At a high level of abstraction, both models appear to address the same problem: going through a large corpus of data for identifying objects that match some search criteria. In both models, execution happens as close to data as possible. However, the similarity is only superficial, and there are many differences at lower levels of abstraction. MapReduce is a batch processing model, intended for index creation prior to search execution. In contrast, Diamond searchlets are created and

executed during the course of an interactive search. None of the Diamond mechanisms for crisp user interaction that were mentioned earlier have counterparts in MapReduce. Fault tolerance is important in MapReduce because it is intended for long-running batch executions. In contrast, Diamond searchlet execution ignores failures since most executions are likely to be aborted soon by the user.

Aspects of Diamond filter execution bear resemblance to the work of Abacus [3], Coign [14], River [4], and Eddies [5]. Those systems provide for dynamic adaptation of execution in heterogeneous systems. Coign focuses on communication links between application components. Abacus automatically moves computation between hosts or storage devices in a cluster based on performance and system load. River handles adaptive dataflow control generically in the presence of failures and heterogeneous hardware resources. Eddies adaptively reshapes dataflow graphs to maximize performance by monitoring the rates at which data is produced and consumed at nodes.

From a broader perspective, indexed search of complex data has long been the holy grail of the knowledge retrieval community. Early efforts included systems such as QBIC [9]. More recently, low-level feature detectors and descriptors such as SIFT [18] have led to efficient schemes for index-based subimage retrieval. However, all of these methods have succeeded only in narrow contexts. For the foreseeable future, automated indexing of complex data will continue to be a challenge 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 [27]. Third, realistic user 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 deep problems will long constrain the success of indexed search for complex data.

3 Diamond applications

Over a multiyear period, we have implemented diverse applications on the OpenDiamond platform. The diversity of these applications speaks for the versatility of the Diamond approach and its embodiment in the OpenDiamond platform. Working closely with domain experts to create these applications has helped us to refine our thinking about discard-based search. It has also guided extensive evolution of the OpenDiamond platform.

We describe five of these applications below. Except for the first, they are all from the health sciences. Our concentration on this domain is purely due to historical circumstances. Researchers in the health sciences (both in industry and academia) were the first to see how our work could benefit them and helped us to acquire the funding to create these applications. We are confident that our work can also benefit many other domains. For example, we are collaborating with a major software vendor to apply discard-based search to large collections of virtual machine images.

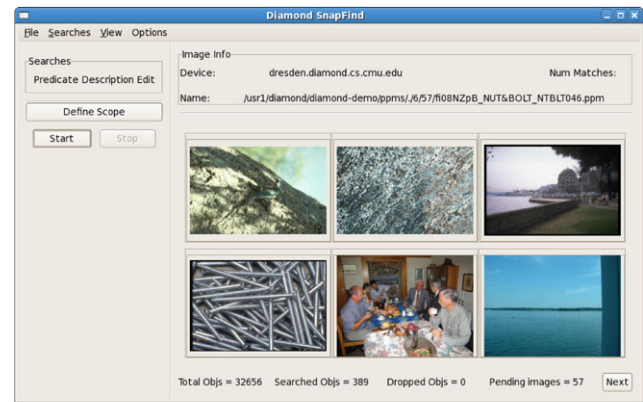
3.1 Unorganized digital photographs

SnapFind enables users to interactively search large collections of unlabeled photographs. Users typically wish to locate photos by semantic content (for example, “Show me the whale watching pictures from our Hawaii vacation”), but this level of semantic understanding is beyond today’s automated image indexing techniques. As shown in Fig. 5(a), *SnapFind* provides a GUI for users to create searchlets by combining simple filters that scan images for patches containing particular color distributions, shapes, or visual textures. The user can either select a predefined filter (for example, “frontal human faces”) or create new filters from patches in example images (for example, a “blue jeans” color filter).

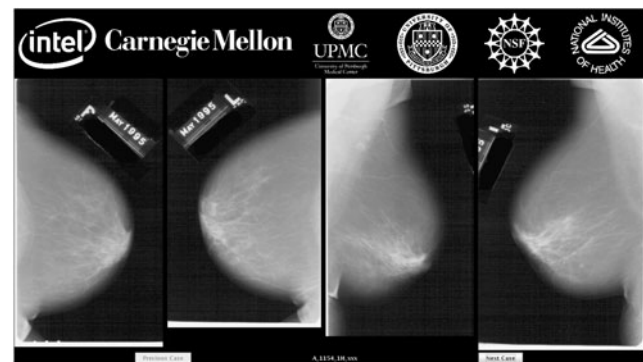
SnapFind supports filters created using ImageJ, a tool that is widely used by researchers in cell biology, pathology, and other medical specialties. The ability to easily add Java-based plugins and the ability to record macros of user interaction are two valuable features of the tool. An investigator can create an ImageJ macro on a small sample of images and then use that macro as a filter in *SnapFind* to search a large collection of images. A copy of ImageJ runs on each server to handle the processing of these filters and is invoked at appropriate points in searchlet execution by the OpenDiamond platform. A similar approach has been used to integrate the widely used MATLAB tool. Based on our positive experience with ImageJ and MATLAB, we plan to implement a general mechanism to allow VM-encapsulated code to serve as a filter execution engine. This will increase versatility, but an efficient implementation is likely to be challenging because of the overhead of VM boundary crossings.

3.2 Lesions in mammograms

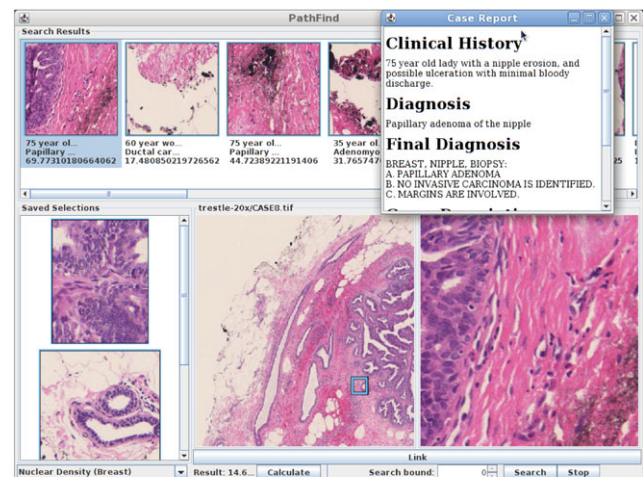
MassFind is an interactive tool for analyzing mammograms that combines a lightbox-style interface that is familiar to radiologists with the power of interactive search. Radiologists can browse cases in the standard four-image view, as shown in Fig. 5(b). A magnifying tool is provided to assist in picking out small detail. Also integrated is a semi-automated mass contour tool that will draw outlines around



(a) SnapFind



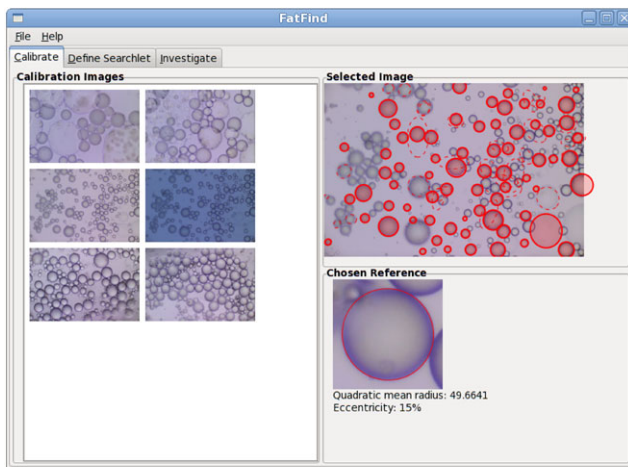
(b) MassFind



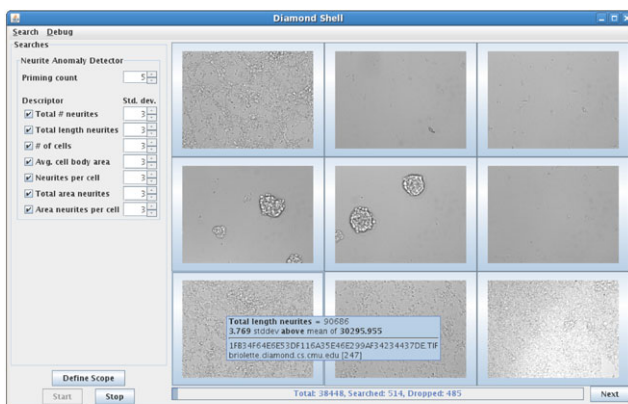
(c) PathFind

Fig. 5 Screenshots of example applications

lesions on a mammogram when given a center point from which to start. Once a mass is identified, a search can be invoked to find similar masses. We have explored the use of a variety of distance metrics, including some based on machine learning [26], to find close matches from a mass corpus. Attached metadata on each retrieved case gives biopsy results and a similarity score. Radiologists can use *MassFind*



(d) FatFind



(e) StrangeFind

Fig. 5 (Continued)

to help categorize an unknown mass based on similarity to images in an archive.

3.3 Digital pathology

Based on analysis of expected workflow by a typical pathologist, a tool called *PathFind* has been developed. As shown in Fig. 5(c), *PathFind* incorporates a vendor-neutral whole-slide image viewer that allows a pathologist to zoom and navigate a whole slide image just as he does with a microscope and glass slides today [13]. The *PathFind* interface allows the pathologist to identify regions of interest on the slide at any magnification and then search for similar regions across multiple slide formats. The search results can be viewed and compared with the original image. The case data for each result can also be retrieved.

3.4 Adipocyte quantitation

In the field of lipid research, the measurement of adipocyte size is an important, but difficult problem. We have built

a tool called *FatFind* for an imaging-based solution that combines precise investigator control with semi-automated quantitation. *FatFind* enables the use of unfixed live cells, thus avoiding many complications that arise in trying to isolate individual adipocytes. The standard *FatFind* workflow consists of calibration, search definition, and investigation. Figure 5(d) shows the *FatFind* GUI in the calibrate step. In this step, the researcher starts with images from a small local collection and selects one of them to define a baseline. *FatFind* runs an ellipse extraction algorithm to locate the adipocytes in the image [12, 17]. The investigator chooses one of these as the reference image and then defines a search in terms of parameters relative to this adipocyte. Once a search has been defined, the researcher can interactively search for matching adipocytes in the image repository. He can also make adjustments to manually override imperfections in the image processing and obtain size distributions and other statistics of the returned results.

3.5 Online anomaly detection

StrangeFind is an application for online anomaly detection across different modalities and types of data. It was developed for the scenario described as the second example of Sect. 1: assisting pharmaceutical researchers in automated cell microscopy, where very high volumes of cell imaging are typical [11]. Figure 5(e) illustrates the user interface of this tool. Anomaly detection is separated into two phases, a domain-specific image processing phase and a domain-independent statistical phase. This split allows flexibility in the choice of image processing and cell type, while preserving the high-level aspects of the application. *StrangeFind* currently supports anomaly detection of adipocyte images (where the image processing analyzes sizes, shapes, and counts of fat cells), brightfield neurite images (where the image processing analyzes counts, lengths, and sizes of neurite cells), and XML files that contain image descriptors extracted by proprietary image processing tools. As an online anomaly detector, *StrangeFind* does not require a predefined statistical model. Instead, it builds up the model as it examines the data.

4 Unique strengths of discard-based search

Through our extensive collaborations with domain experts, we have acquired a deep appreciation for the strengths of discard-based search relative to indexed search. These strengths were not apparent to us initially, since the motivation for our work was simply coping with the lack of an index for complex data.

Relative to indexed search, the weaknesses of discard-based search are obvious: *speed* and *security*. The speed

weakness arises because all data is preprocessed in indexed search. Hence, there are no compute-intensive or storage-intensive algorithms at runtime. In practice, this speed advantage tends to be less dramatic because of result and attribute caching by servers in our system, as discussed in Sect. 2.3. The security weakness arises because the early-discard optimization requires searchlet code to be run close to servers. Although a broad range of sandboxing techniques, language-based techniques, and verification techniques can be applied to reduce risk, the essential point remains that untrusted code runs on trusted infrastructure during a discard-based search. This is not a concern with indexed search, since preprocessing is done offline.

At the same time, discard-based search has certain unique strengths. These include: (a) flexibility in tuning between false positives and false negatives, (b) ability to dynamically incorporate new knowledge, and (c) better integration of user expertise.

4.1 Dynamic tuning of precision and recall

The preprocessing for indexed search represents a specific point on a precision-recall curve, and hence a specific choice in the tradeoff space between false positives and false negatives. In contrast, this tradeoff can be dynamically changed during a discard-based search session. Using domain-specific knowledge, an expert user may tune searchlets toward false positives or false negatives depending on factors such as the purpose of the search, its completeness relative to total data volume, and the user's judgment of results from earlier iterations in the search process.

It is also possible to return a clearly labeled sampling of discarded objects to alert the user to what she might be missing, and hence to the likelihood of false negatives. To improve a searchlet, a Diamond user needs at least a modest rate of return of results even if they are not of the highest quality. Sometimes, the best way to improve a searchlet is by tuning it to reduce false negatives, typically at the cost of increasing false positives. To aid in this, a planned extension of our system will provide a separate result stream that is a sparse sampling of discarded objects. Applications can present this stream in a domain-specific manner to the user, and allow her to discover false negatives. It is an open question at this time whether the sampling of discarded objects should be uniform or biased towards the discard threshold (i.e., "near misses").

4.2 Dynamic incorporation of 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 labeling 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." Note that this observation is true even if index terms were obtained by game-based human tagging approaches such as ESP [2].

4.3 Ability to leverage user expertise

Discard-based search better utilizes the user's intuition, expertise, and judgment. In contrast, indexed search limits even experts to the quality of the preprocessing that produced the index. There are many degrees of freedom in searchlet creation and parameterization through which user expertise can be expressed. In the most general case, the user can dynamically generate new code for a filter through programming or through machine learning from recent search results. More commonly, the parameters of an existing filter are modified.

As mentioned earlier, Diamond is very much a human-in-the-loop system. It is based on the premise that at small scale, there is no substitute for deep human knowledge of complex, domain-specific data. However, these uniquely human capabilities are easily overwhelmed by a large volume of data. The role of the query-specific content-based computation performed by Diamond is to reduce the apparent scale of the problem confronting a human expert: that is, to allow the expert's knowledge and skills to focus on relatively few objects (on the order of tens of objects rather than thousands, millions, or more). An important consequence of our human-in-the-loop assumption is that the content-based computation does not have to be perfect to be useful. In fact, one of the surprising outcomes of our experience with Diamond applications is that even simple and relatively unsophisticated domain-specific processing can be helpful in scale reduction.

5 Unique server storage characteristics

Searchlet execution on a Diamond server exhibits certain distinctive processing and I/O properties:

- *Read-only, whole object I/O*: After initial provisioning, a corpus of data objects is never modified. Storage for the persistent cache of filter execution results is distinct from the corpus. Filter execution in a typical Diamond application involves the entire contents of an object, rather than just part of it.

- *Any-order I/O semantics*: Since the Filter API uses an iterator model, a filter effectively says “Get next object” rather than “Get object X.” The storage subsystem on a server is hence free to return any object that has not been presented before in the current search. For example, an already-cached or already-prefetched object can be returned in preference to an object that would require blocking on I/O. 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 degree of freedom is rarely available outside the Diamond framework.
- *Interleaved search spaces*: Diamond applications exhibit a distinctive usage pattern that we call *interactive data exploration*. A user constructs an initial searchlet out of a set of filters, presents it to his Diamond application, gets back a few results, aborts the current search, and then modifies his searchlet (sometimes extensively) in the light of these results. This iterative process continues until the user finds what he is looking for, or gives up. The user is effectively conducting two interleaved and tightly coupled searches: one on the query space (the space of all possible searchlets) and the other on the data space (the contents of all data objects). This is consistent with the well-known observation that asking exactly the right question about complex data is often the key to a major insight. Of course, figuring out the right question to ask is itself a challenge!
- *Embarrassing parallelism*: It is hard to imagine a workload more friendly to server CPU and storage parallelism than a typical Diamond application. Each object is individually processed in its entirety, with no concurrency control or ordering constraints across objects. Since objects are typically large, the net effect is to provide ample opportunity for coarse-grained and easy-to-exploit parallelism.

In combination, these Diamond workload characteristics are at odds with 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. Diamond workloads force us to rethink these tenets, as discussed below in Sects. 5.1 to 5.5.

5.1 Invalid tenet: “Think Striping”

The I/O bandwidth advantages of disk striping are so well known today that use of RAID arrays in servers is almost instinctive for a system designer. The contrasting approach

of placing entire data objects on individual disks (referred to as the JBOD or “just a bunch of disks” solution) is rare. However, JBOD turns out to be the better fit for Diamond workloads. To understand why, consider the two alternatives in the context of discard-based search.

With JBOD, the OpenDiamond platform has full control over the independent management of 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 (objects already known from a previous search to have passed a filter), each worker can be given a list of objects to retrieve. 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 completely under control of the OpenDiamond platform, the number of worker threads (level of concurrency) can be dynamically adjusted so as to make the most efficient use of each disk.

Although RAID improves reliability and presents a simpler programming model, there are no performance benefits relative to JBOD for sequential scans. Both JBOD and RAID-0 will see at most the full streaming bandwidth of the disks. However, this bandwidth may be reduced in RAID-0 because of disk contention when concurrency is high. The only case where RAID-0 will potentially outperform JBOD is in the case of low concurrency during a random I/O retrieval. This would most typically happen when cache hits occur for all filters in the current searchlet, and the corresponding object is fetched for return to the client. With JBOD, this 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 opportunity will not occur often on Diamond servers. Moreover, with additional effort, a JBOD configuration can effectively achieve the same load-balancing of random I/O by reordering the retrieval of objects with cache hits.

The reasoning that JBOD is a better fit than RAID for Diamond workloads is confirmed by the experimental results shown in Fig. 6 for a synthetic discard-based search workload. These experiments were run on an Intel SSR212CC storage system, with a 2.8-GHz Intel Xeon[®] CPU with hyperthreading and 1 GB of main memory, running Ubuntu Linux 7.04. Ten 200-GB Seagate Barracuda 7200.10 disks were used; five were on one controller in a JBOD configuration, and the other five were on a second controller configured as a hardware RAID array with 64-KB stripe units. Each disk had an internal read-ahead cache and could reach its full streaming bandwidth even with a single outstanding I/O. The Y axis in the graphs of Fig. 6 is *I/O efficiency*:

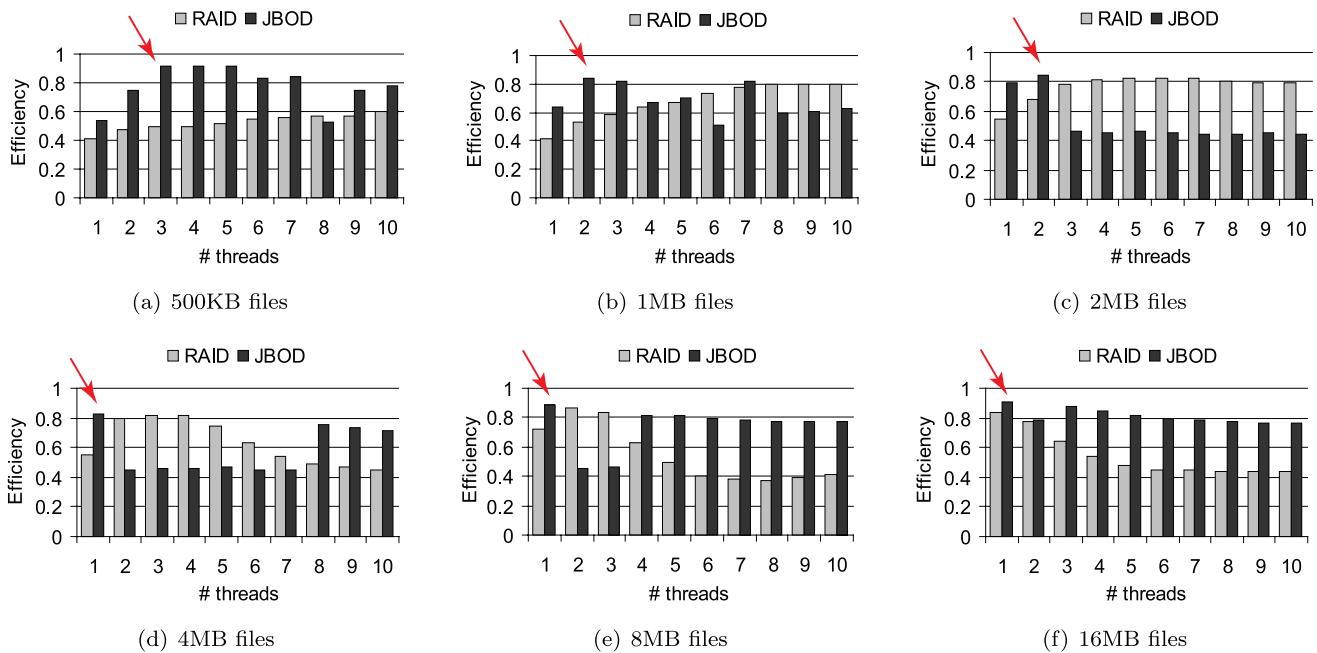


Fig. 6 Comparison of RAID and JBOD on Diamond servers

ratio of the measured bandwidth to the maximum theoretical aggregate bandwidth of 370 MB/s for the five-disk array. Each graph compares the I/O efficiency of JBOD and RAID for a specific file size, with increasing concurrency from left to right. The figure shows that JBOD outperforms RAID for most combinations of file size and concurrency level. In every case, the maximum efficiency (shown by the arrow) is achieved by JBOD.

5.2 Invalid tenet: “Abort Is Rare”

As mentioned earlier, typical Diamond use involves two interleaved search sequences, one on the query space and the other on the data space. Almost never does a user wait for his current searchlet to be applied to all data objects. Instead, the user typically aborts a search as soon as he has seen a sufficient number of results to guide him on how to improve the current searchlet.

High probability of abort during sequential scan is rare for storage systems. Scan access patterns typically arise in data mining and indexing workloads that run to completion on the entire dataset. Diamond’s unusual combination suggests that I/O optimization strategies that expect to amortize a high up-front cost over the large number of objects in a dataset are unlikely to be effective. For example, building up deep I/O and processing pipelines through prefetching can be counterproductive if flushing these deep pipelines on abort involves significant delays. This reasoning also applies to the client-server interconnect. A large number of results in flight from server to client instantly become junk when

a user aborts the current search. Until this junk is flushed, the user will not see results from the execution of his next searchlet. This problem is particularly severe in interconnects with large bandwidth-delay products (that is, “long, fat pipes”). The ability to rapidly flush junk from all interconnects between a user and data on server disks would be a valuable capability. This requires out-of-band communication, which is unfortunately not available on today’s Internet.

To confirm our understanding of this aspect of Diamond workloads, we studied a number of users performing different search tasks over collections of digital photographs [21]. We observed that, on average, users aborted their queries after viewing 36 objects and moved on to the next step of their search. At the point of abort, each query had processed less than 10% of the entire collection. As an illustrative example, consider how one user performed the task of finding pictures of a friend’s wedding. The user began her search by noting that wedding pictures are likely to contain faces. While the returned images contained faces, many of them were not from the wedding. She added a color filter to match the wedding dress. Unfortunately, the color was not accurate, and the new query returned no relevant images. The user then remembered that some of the wedding photos were taken outdoors. She replaced the wedding dress filter with a new green color filter to match grass. The new query returned a photo that included the bride standing in front of the church. Based on the color of the bride’s bouquet, the user created another filter. Combined with the face filter, this produced the highly relevant images shown in Fig. 7.

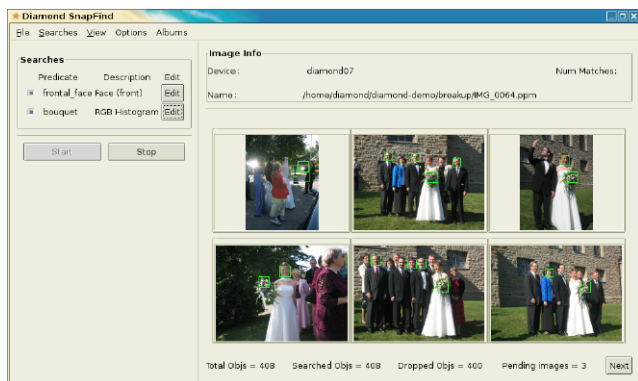


Fig. 7 Searching for wedding pictures

5.3 Invalid tenet: “The App Knows Best”

Conventional wisdom has long held that applications have deeper knowledge of their I/O access patterns than lower system layers, and are therefore better positioned to optimize the use of storage devices. As early as 1981, Stonebreaker articulated the case against management of database I/O by general purpose operating systems [24]. More recently, Patterson et al. [22] advocated application assistance for I/O prefetching. Today, hiding storage management and I/O device details from applications is justified primarily in terms of a simpler programming model and the need to enforce security constraints. Suboptimal performance is viewed as an acceptable price for these benefits.

Any-order semantics in Diamond leads to a powerful counterpoint to this conventional wisdom. Rather than reading a specific object, a Diamond application simply requests the next object that has not been discarded by the searchlet. This gives the OpenDiamond platform, operating system, and storage hardware the flexibility to process objects in whatever order is most efficient for those layers. In other words, I/O workload shaping is possible. The best disk I/O throughput is achieved by reading objects sequentially starting with the object closest to the current position of the disk head. If multiple queries are started, any-order semantics allows their read operations to be coalesced into a single request stream, thus avoiding device contention.

This concept, called *bandwagon synchronization*, is illustrated in Fig. 8. In Fig. 8(a), a query embodied in searchlet X processes objects retrieved from a sequential scan of the collection. In Fig. 8(b), 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 8(c) 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 Fig. 8(d), the sequential scan resumes with object $j + 1$. Bandwagon synchronization ensures that the storage subsystem receives a workload that is highly sequential, keeping disk seeks to a bare minimum.

Figures 9(a) and 9(b) conceptually illustrate the performance benefit of bandwagon synchronization by showing the number of objects delivered to the application by two queries, X and Y that start at times t_x and t_y , respectively. Query Y can either introduce its own I/O requests or can synchronize (jump on the bandwagon) with query X. 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 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 9(a) 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, lowering the bandwidth. Neither X nor Y is able to read sequentially at that point. Figure 9(b) illustrates the two queries with bandwagon synchronization: query Y is able to

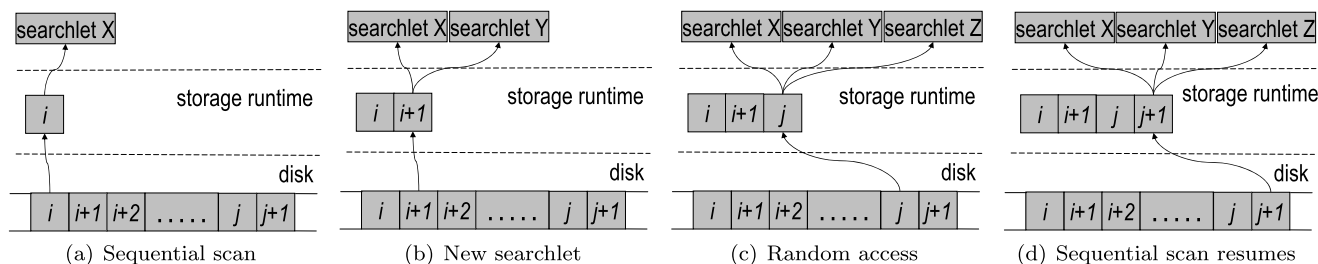
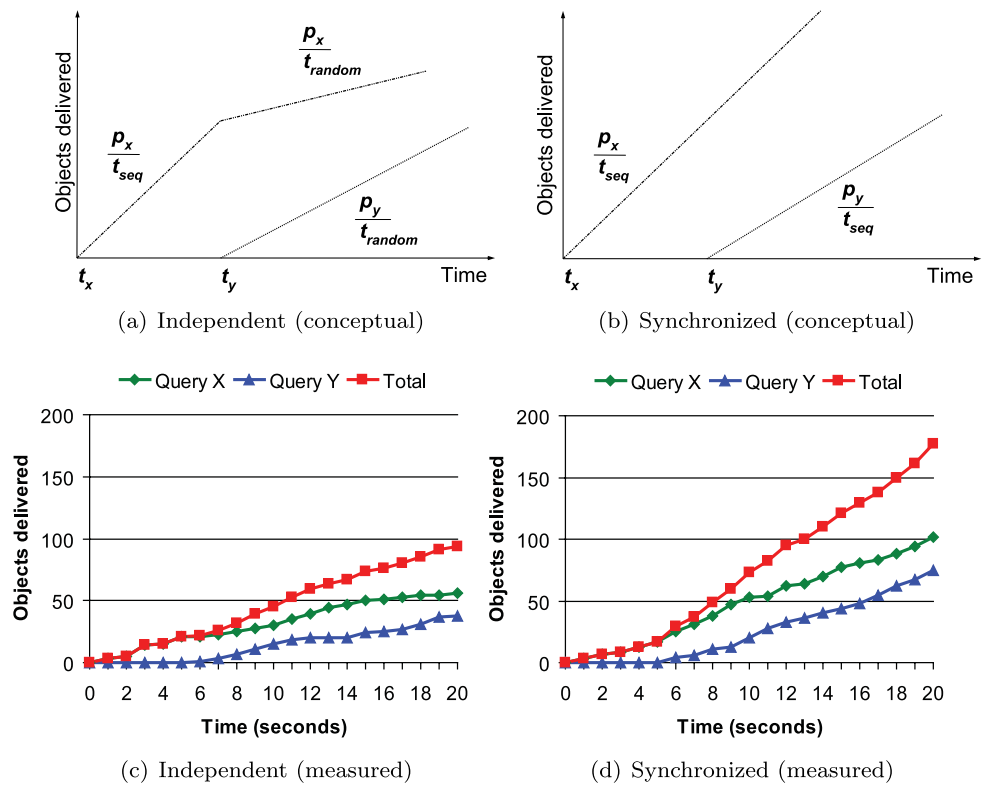


Fig. 8 Bandwagon synchronization

Fig. 9 Performance impact of bandwagon synchronization



share the objects fetched by X. Now, both queries proceed at sequential read rate without mutual interference.

Figures 9(c) and 9(d) show experimental validation of these concepts for a synthetic workload on a single disk. The graphs show the total objects delivered as a function of time. These experiments used 1 MB objects, query pass rates of 10%, and no searchlet computation time. Two queries were started five seconds apart. Although the queries had equal selectivity, they 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 beginning. In Fig. 9(c), 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 the bandwidth of random access. In Fig. 9(d), both queries are able to take advantage of sequential access bandwidth, resulting in a twofold improvement over Fig. 9(c).

5.4 Invalid tenet: “Seeks Are Evil”

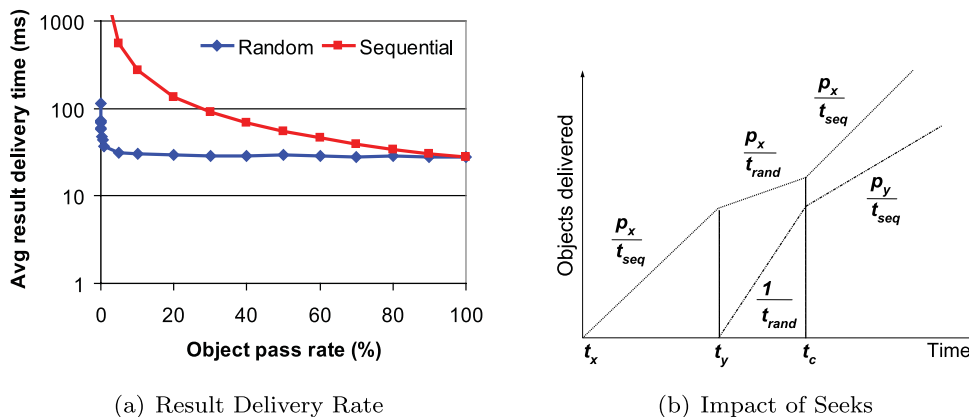
Conventional wisdom holds that random access on disks is to be avoided because of the performance overhead associated with seeks and rotational delay. As discussed in the previous section, bandwagon synchronization can achieve sequential access even when multiple users are concurrently executing searchlets on a Diamond server. However, result

caching muddies this clear picture. The fortunate user who gets a cache hit avoids searchlet execution on the corresponding object. However, I/O is still needed to fetch the object for transmission to the client. This may involve a seek that disrupts sequential access for other users. Should the seek be tolerated to benefit the fortunate user, or should that user be forced to wait until the bandwagon of other users reaches the object in question? How should this issue be resolved on a Diamond server that aims for fairness to all users?

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 a sufficient number of results before they can decide how to refine their searchlets. The *pass rate* of a searchlet is the probability that an object will survive the entire gauntlet of filters in the searchlet. It is a key factor in determining the rate at which a Diamond user sees results. Another factor is the average rate at which objects are presented to searchlets.

Figure 10(a) shows how these two factors interact. These experimental results were based on the same hardware described in Sect. 5.1. A synthetic workload generator referenced 10-MB objects that were stored on a five-disk JBOD array. The processing time of the filters in the searchlet is assumed to be negligible. For the curve labeled “Sequential,” the files on the disk were sequentially scanned, with the pass rate indicated on the X axis. For the curve labeled

Fig. 10 Prioritizing accesses to objects with cached results



(a) Result Delivery Rate

(b) Impact of Seeks

“Random,” we assume the existence of an in-memory index that allows direct seeks to objects whose filter execution results are already cached. The Y axis of Fig. 10(a) is in log scale and shows the average rate of results seen by users. At a pass rate of 100%, object data transfer is dominant. Hence, random and sequential access both achieve the same result interarrival time of roughly 28 ms. As the pass rate decreases, this time increases for both cases. For sequential access, the increase is due to fewer objects passing through the searchlet per time unit. Performance worsens rapidly as pass rate drops below 1%: result interarrival time increases from 2.7 seconds to over 18 seconds for a pass rate of 0.1%. For random access, lowering pass rate increases result interarrival time for a different reason. Assuming that result cache hits occur for objects that are uniformly distributed over the entire storage device, a low pass rate translates to a long average seek length because the accessed objects are more spread out. In Fig. 10(a), the maximum delay of 113 ms occurs when pass rates drop below 1 in 50000 (i.e., 0.002%). Our experience indicates that such low pass rates are not unusual in Diamond searches.

Based on this analysis, we can conclude that taking advantage of cached results is worthwhile even after accounting for the disruption of bandwagon synchronization that it implies. This leads to the storage management policy that was illustrated earlier in Figs. 8(c) and 8(d). Figure 10(b) conceptually illustrates the performance impact of this policy. The figure shows query X in sequential scan initially. When query Y arrives, all its cache hits are first retrieved. This occurs during the period from t_y to t_c , lasting $n_c \cdot t_{rand}$, where n_c is the number of cached hits for Y. These objects are also available to query X, generating $p_x \cdot (t_c - t_y) / t_{rand}$ additional passing objects. Once the cache hits of Y have been accessed, sequential scan resumes.

More generally, the cache hit phase for a new query imposes this penalty on the N other queries in progress:

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

For the example in Fig. 10(b), the gain exceeds the penalty when $(p_x / t_{rand} + 1 / t_{rand}) > (p_x / t_{seq} + p_y / t_{seq})$. In other words, this is a win only when the selectivity p_y is sufficiently small. Experimental results that confirm the reasoning presented here can be found in the technical report by Mummert et al. [21].

5.5 Invalid tenet: “Real Work Beats Speculation”

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

Discard-based search can benefit from several forms of speculative execution [10]. Intra-query speculation exploits user think time to perform additional processing on objects. Inter-query speculation exploits idle periods to execute 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. The caching of results and attributes means that even speculative work that appears to be wasted relative to the current search is not really wasted in a larger context.

6 Kaiten-zushi storage

What would a clean-sheet design of a storage system for Diamond servers look like? Based on the storage characteristics discussed in Sect. 5, 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 [25]:

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 objects in sequential order (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 eat 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.

7 Conclusion

Our ability to record real-world data has exploded. Huge volumes of medical imagery, surveillance imagery, sensor

feeds for the earth sciences, anti-terrorism monitoring, and many other sources of complex data are now captured routinely. The capacity and cost of storage to archive this data have kept pace. Sorely lacking are the tools to extract the full value of this captured data.

The goal of Diamond is to help domain experts creatively explore large bodies of complex, nonindexed data. We hope to do for complex data what spreadsheets did for numeric data in the early years of personal computing: allow users to “play” with the data, easily answer “what if” questions, and thus gain deep, domain-specific insights. In this paper, we have described how discard-based search can be a powerful tool for this task. We have discussed the unique strengths of discard-based search relative to indexed search and presented its unique server storage characteristics. We view pure indexed search and pure discard-based search as the extremes of a continuum; specific instances of result caching and scoping together define many intermediate points in this continuum.

The central premise of our work is that the sophistication of queries we are able to pose about complex data will always exceed our ability to anticipate, and hence precompute indexes for, such queries. While indexing techniques will continue to advance, so will our ability to pose ever more sophisticated queries—our reach will always exceed our grasp. It is in that gap that the Diamond approach will have most value.

Acknowledgements This research was supported by the National Science Foundation (NSF) under grant number CNS-0614679. Development of the MassFind and PathFind applications described in Sects. 3.2 and 3.3 was supported by the Clinical and Translational Sciences Institute of the University of Pittsburgh (CTSI), with funding from the National Center for Research Resources (NCRR) under Grant No. 1 UL1 RR024153. The FatFind and StrangeFind applications described in Sects. 3.4 and 3.5 were developed in collaboration with Merck & Co., Inc. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily represent the views of the NSF, NCRR, CTSI, Intel, Merck, or Carnegie Mellon University. OpenDiamond is a registered trademark of Carnegie Mellon University. Xeon is a registered trademark of Intel Corporation. All unidentified trademarks remain the properties of their owners.

Open Access This article is distributed under the terms of the Creative Commons Attribution Noncommercial License which permits any noncommercial use, distribution, and reproduction in any medium, provided the original author(s) and source are credited.

References

1. Acharya A, Uysal M, Saltz J (1998) Active disks: programming model, algorithms and evaluation. In: Proceedings of the international conference on architectural support for programming languages and operating systems
2. von Ahn L, Dabbish L (2004) Labeling images with a computer game. In: Proceedings of the SIGCHI conference on human factors in computing systems

3. Amiri K, Petrou D, Ganger G, Gibson G (2000) Dynamic function placement for data-intensive cluster computing. In: Proceedings of the USENIX technical conference
4. Arpaci-Dusseau R, Anderson E, Treuhaft N, Culler D, Hellerstein J, Patterson D, Yelick K (1999) Cluster I/O with river: making the fast case common. In: Proceedings of input/output for parallel and distributed systems
5. Avnur R, Hellerstein J (2000) Eddies: continuously adaptive query processing. In: Proceedings of SIGMOD
6. Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In: Proceedings of the USENIX symposium on operating systems design and implementation, San Francisco, CA
7. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. *Commun ACM* 51(1)
8. Duda R, Hart P, Stork D (2001) *Pattern classification*. Wiley, New York
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 (1995) Query by image and video content: the QBIC system. *IEEE Comput* 28(9)
10. Gibbons P, Mummert L, Sukthankar R, Satyanarayanan M (2007) Just-in-time indexing for interactive data exploration. Tech Rep CMU-CS-07-120, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA
11. Goode A, Sukthankar R, Mummert L, Chen M, Saltzman J, Ross D, Szymanski S, Tarachandani A, Satyanarayanan M (2008) Distributed online anomaly detection in high-content screening. In: Proceedings of the 2008 5th IEEE international symposium on biomedical imaging, Paris, France
12. Goode A, Chen M, Tarachandani A, Mummert L, Sukthankar R, Helfrich C, Stefanni A, Fix L, Saltzmann J, Satyanarayanan M (2007) 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
13. Goode A, Satyanarayanan M (2008) A vendor-neutral library and viewer for whole-slide images. Tech Rep CMU-CS-08-136, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA
14. Hunt G, Scott M (1999) The Coign automatic distributed partitioning system. In: Proceedings of OSDI
15. Huston L, Sukthankar R, Wickremesinghe R, Satyanarayanan M, Ganger GR, Riedel E, Ailamaki A (2004) 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
16. Keeton K, Patterson D, Hellerstein J (1998) A case for intelligent disks (IDISks). *SIGMOD Rec* 27(3)
17. Kim E, Haseyama M, Kitajima H (2002) Fast and robust ellipse extraction from complicated images. In: Proceedings of IEEE information technology and applications
18. Lowe D (2004) Distinctive image features from scale-invariant keypoints. *Int J Comput Vis*
19. Merriam-Webster (2007) Merriam-Webster online search. <http://mw1.merriam-webster.com/dictionary/tenet>
20. Minka T, Picard R (1997) Interactive learning using a society of models. *Pattern Recognit* 30
21. Mummert L, Schlosser S, Mesnier M, Satyanarayanan M (2007) Rethinking storage for discard-based search. Tech Rep CMU-CS-07-176, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA
22. Patterson RH, Gibson GA, Ginting E, Stodolsky D, Zelenka J (1995) Informed prefetching and caching. In: Proceedings of the fifteenth ACM symposium on operating systems principles, Copper Mountain, CO
23. Riedel E, Gibson G, Faloutsos C (1998) Active storage for large-scale data mining and multimedia. In: Proceedings of the international conference on very large databases
24. Stonebreaker M (1981) Operating system support for database management. *Commun ACM* 24(7)
25. Wikipedia (2007) Conveyor belt sushi. Wikipedia, The Free Encyclopedia. [Online: accessed 3-September-2007]
26. Yang L, Jin R, Mummert L, Sukthankar R, Goode A, Zheng B, Hoi SC, Satyanarayanan M (2010) A boosting framework for visuality-preserving distance metric learning and its application to medical image retrieval. *IEEE Trans Pattern Anal Mach Intell* 32(1)
27. Yao A, Yao F (1985) A general approach to D-dimensional geometric queries. In: Proceedings of the annual ACM symposium on theory of computing