# The OpenDiamond® Platform for Discard-based Search

M. Satyanarayanan[†], Rahul Sukthankar[‡], Adam Goode[†],
Larry Huston, Lily Mummert[‡], Adam Wolbach[†],
Jan Harkes[†], Richard Gass[‡], Steve Schlosser[‡]

May 2008
CMU-CS-08-132

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

## Abstract

Interactive exploration of large distributed collections of complex, non-text data such as medical images is a challenging task because of the difficulty of creating useful indexes. To handle such tasks, we introduce a new approach to search called *discard-based search.* In contrast to classic search strategies that precompute indexes for all anticipated queries, discard-based search is an *on-demand strategy* that performs *content-based computation in response to a specific query.* This simple change in strategy turns out to have deep consequences for flexibility and user control, while also enabling easy exploitation of CPU and storage parallelism on servers. This paper presents the design and implementation of the OpenDiamond platform for discard-based search, describes some of the applications that have been built with it, and offers experimental evidence that its workloads exhibit easily-exploitable storage parallelism.

This research was 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 or Carnegie Mellon University.

**Keywords:** middleware, open source, medical images, image processing, pattern recognition, self-tuning, load balancing, result caching, just-in-time indexing, interactive search, database, anomaly detection, similarity search, pathology, radiology, mammograms, whole-slide image, active disk, smart storage, object storage, coarse-grain parallelism, ImageJ

## 1 Introduction

*Discard-based search* is the basis of a new approach to interactive exploration of complex, non-indexed data. Examples of such data include large distributed repositories of digital photographs, medical images, surveillance images, speech clips or music clips. The emphasis on "interactive" is important: this work assumes that the most precious resource is the time and attention of the person conducting the search rather than system resources such as network bandwidth, CPU cycles or disk bandwidth. That person is assumed to be a high-value expert such as a doctor, pharmaceutical researcher, military planner, or law enforcement official, rather than a mass-market consumer.

In contrast to classic search strategies that precompute indexes for all anticipated queries, discard-based search uses an *on-demand computing strategy* that performs *content-based computation in response to a specific query.* As discussed in Section 2, this simple change in strategy has deep consequences for flexibility and user control. Server workloads in discard-based search exhibit coarse-grained storage and CPU parallelism that is easy to exploit. Further, discard-based search can take advantage of result caching at servers. This can be viewed as a form of just-in-time indexing that is performed incrementally at run time rather than in bulk *a priori.*

We have been exploring this new search paradigm since late 2002 in the Diamond project, and have gained extensive experience with both software infrastructure and domain-specific applications. This experience has helped us to cleanly separate the domain-specific and domain-independent functionality. We have encapsulated the latter into Linux middleware called the *OpenDiamond platform.* Based on standard Internet component technologies, it is distributed in open-source form (`http://diamond.cs.cmu.edu/`) under the Eclipse Public License.

For ease of exposition and for historical reasons, 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" is always used in a precise technical sense to refer to the open-source middleware.

This paper presents the design and implementation of the OpenDiamond platform, describes some of the applications that have been built with it, and offers experimental evidence to confirm that discard-based search workloads exhibit easily-exploitable storage parallelism.

## 2 Background

### 2.1 Need for Discard-based Search

Automated indexing of complex data such as images remains a challenging problem today 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* [29] 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* [7, 10, 43]. 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 problems constrain the success of indexed search for complex, multi-dimensional, loosely-structured data.

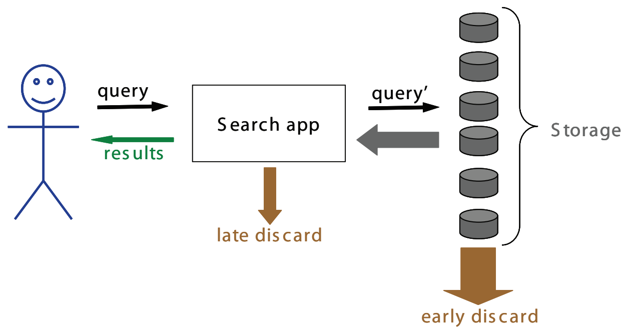


**Figure 1: Discard-based Search with Early Discard Optimization**

In contrast, discard-based search does not attempt to preprocess all data in advance of future queries. Rather, it dynamically performs computation to eliminate objects that are clearly not results for the query. An optimized version of this concept, called *early discard,* rejects most of the irrelevant data as early as possible in the pipeline from storage to user. This improves scalability by eliminating a large fraction of the data from most of the pipeline. Figure 1 illustrates this concept.

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 code components called *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) [10]. 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.

## 2.2 Relative Merits of Indexed and Discard-based Search

The strengths and weaknesses of indexed search and discard-based search complement each other, as discussed below. Speed and security favor indexed search, but 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 it offers high value in several 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 can therefore be much faster than discard-based search. In practice, this speed advantage tends to be less dramatic because of result caching by the OpenDiamond platform. Discard-based searches that have some overlap in their queries with previous searches will benefit from the result caching. Over time, cache entries will be created for many objects on frequently-used combinations of filters and filter parameters, thus reducing the speed differential with respect to indexed search.

*Server security:* The early-discard optimization requires searchlet code to be run close to servers. Although a broad range of sandboxing techniques [39], language-based techniques [40], and verification techniques [30] can be applied to reduce risk, the essential point remains that user-generated code may need to run on trusted infrastructure during a discard-based search. This is not a concern with indexed search, since preprocessing is done offline. The higher degree of scrutiny and trust that tends to exist within an enterprise suggests that initial uses of discard-based search are most likely to be within the Intranets of enterprises.

*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-labeled sampling of discarded objects during a discard-based search to alert the user to what she 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 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." Short of re-indexing the entire corpus, this new knowledge cannot be incorporated into indexed search. Note that this observation is true even if index terms were obtained by game-based human tagging approaches such as ESP [38].

*User expertise*: Discard-based search better utilizes the user's expertise and judgement. There are many degrees of freedom in searchlet creation and parameterization through which this expertise and judgement can be expressed. In contrast, indexed search limits even experts to the intrinsic quality of the preprocessing that produced the index.

# 3 Design and Implementation

## 3.1 Diamond Architecture

As Figure 2 illustrates, the Diamond architecture cleanly separates domain-specific application code from a 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 that spans Layers 5–7 of the OSI model. On a client machine, the user interacts via a GUI with a particular search application.

To handle a user query, the application constructs a searchlet out of individual components called *filters*. A filter consists of executable code combined with some specific parameters. As an example, a content-based image search application might construct a searchlet using color histogram filter code plus "dark red" color parameters, Gabor visual texture filter code plus "stone" and "fur" texture parameters, and face detection code plus some default face detection parameters. The parameters in each filter serve to tune it: the parameters to a color histogram filter determine which colors are detected. The searchlet is composed and presented by the application through the *Searchlet API* to the Diamond runtime system, which then distributes it to all of 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

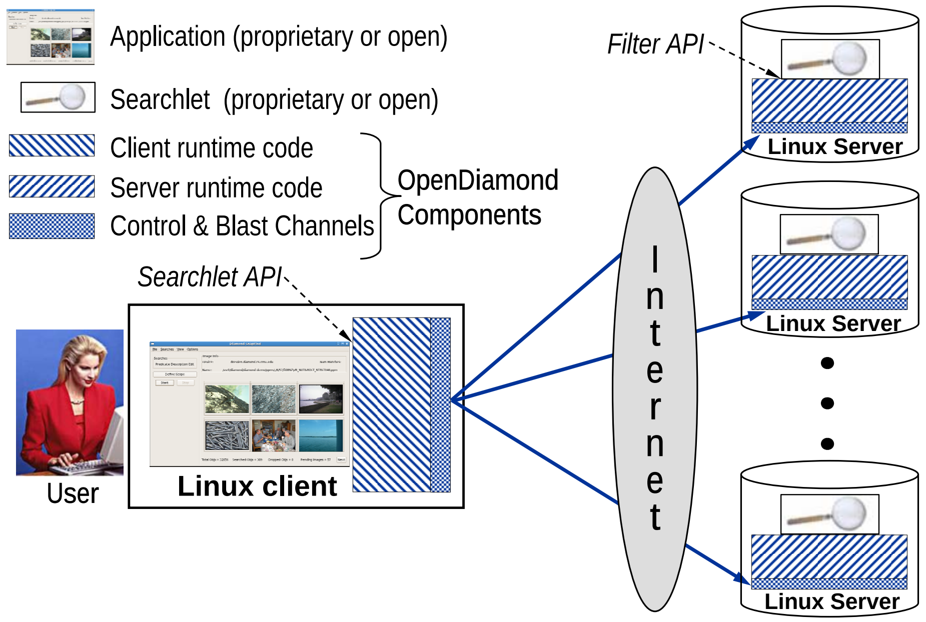


**Figure 2: Diamond System Architecture**

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 all of the filters in the searchlet are transmitted to the client[1].

A key architectural constraint of Diamond is that servers do not communicate directly with each other; they only communicate with clients. The primary factor driving this design decision is the simplification it achieves in the logistics of 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. A secondary factor is the simplification and decomposability that it achieves in the server code structure. Our experience with the applications described in Section 4 confirm that this architectural constraint is a good tradeoff. Only in one instance (the online anomaly detection application in Section 4.4) have we found a need for even limited sharing of information across servers during a search. Even in that case, the volume of sharing is small: typically, a few hundred bytes to a few kilobytes every few seconds. This is easily achieved through the use of *session variables* in the APIs described in Section 3.2.

## 3.2 Searchlet and Filter APIs

The OpenDiamond platform consists of two APIs: the Searchlet API and the Filter API. The Searchlet API defines the programming interface for the application code (typically GUI-based) that runs on the client. The Filter API defines the programming interface for filter code that runs on a server.

Tables 1 through 4 list the calls of the Searchlet API, grouped by logical function. For brevity, these tables omit the prefix “`ls_`” (for “libsearchlet”) from the names of calls. Table 1 includes the calls for initialization and scoping of a search session (discussed in Section 3.5).

[1]An exception to this is when the load balancing mechanism described in Section 3.6.2 offloads execution of some filters from a slow or heavily-loaded server to a fast client.

| | |
|---|---|
| `init_search` | Allocate and initialize the global search object and threads. |
| `set_searchlist` | Set global search list to a set of 64-bit object disk group identifiers. |

**Table 1: Initialization and Scoping (Searchlet API)**

| | |
|---|---|
| `set_searchlet` | Load and parse a searchlet specification file to set up the current searchlet. Also load a binary file containing some of the executable code needed at the server. |
| `add_filter_file` | Load an additional binary file into the current searchlet. Needed to complete the searchlet based on the specification. |
| `set_blob` | Set the binary argument for a particular filter. Normal arguments can only consist of strings of printable characters (except space). A filter's blob argument can be any binary data. |

**Table 2: Defining a Searchlet (Searchlet API)**

Table 2 lists the calls used by an application to define searchlets and filters. The filter code provided through these calls is transmitted by the OpenDiamond platform to each server involved in the current scope. Table 3 shows the calls used by the application to control a search. After issuing a `start_search`, the application calls `next_object` repeatedly as a result iterator. When the user aborts the current search and goes back to the step of selecting a new filter or changing the parameters of an existing one, the calls in Table 2 again become relevant. The calls `get_dev_session_variables` and `set_dev_session_variables` in Table 3 allow the client to obtain a small amount of search-specific data from each server and to disseminate them to all servers. Table 4 shows calls that are typically used for debugging applications and for performance analysis.

Tables 5 through 7 present the Filter API. For brevity, these tables omit the prefix "`lf_`" (for "libfilter") from the names of calls. The OpenDiamond runtime code on a server iterates through objects within scope in an unspecified order, giving the storage subsystem an important degree of freedom for future performance optimizations. Although we do not yet exploit this opportunity, the Filter API design ensures that even applications written today are already able to cope with any-order storage semantics. Each filter provides the set of callback functions shown in Table 5, and the OpenDiamond runtime code invokes these functions at appropriate times.

The `filter_eval` callback function is invoked once for each object in scope. Within this function, the filter code can use the calls in Table 6 to obtain the contents of the object and the calls in Table 7 to get and set *attributes* associated with the object. Attributes typically encode intermediate results: for example, an image codec will read compressed image data and write out uncompressed data as an attribute; an edge detector will read the image data attribute and emit a

| | |
|---|---|
| `start_search` | Start a search. |
| `next_object` | Get the next object from the result queue. If result queue is empty, return immediately. |
| `num_objects` | Get the number of pending objects in the current processing queue. |
| `release_object` | Free a previously returned object. |
| `terminate_search` | Tell servers to stop processing objects. Actual termination occurs asynchronously at each server. |
| `get_dev_session_variables` | Get the names and values of the session variables as stored on a server. Session variables are used to accumulate application-specific values that can be combined and distributed across all servers. |
| `set_dev_session_variables` | Set a server's session variables to particular values given here. After this call completes, all session variables of `get_dev_session_variables` are set to zero. |

**Table 3: Controlling a Search (Searchlet API)**

new attribute containing an edge map. As an object passes through the filters of a searchlet, each filter can add new attributes to that object for the benefit of filters that are further downstream.

## 3.3 Result and Attribute Caching on Servers

Caching in the OpenDiamond platform takes two different forms: *result caching* and *attribute caching*. Both are implemented entirely on the servers, and are 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 other's search activities without any coordination or awareness of each other. The sharing of knowledge within an enterprise, such as one member of a project telling his colleagues what filter parameter values worked well on a project-related search task, can give rise to significant communal locality in filter executions. One can thus view result caching as a form of incremental indexing that occurs as a side-effect of normal use.

Result caching allows a server to remember the outcomes of object–filter–parameter combinations. Since filters consist of arbitrary code and there can be many parameters of diverse types, we use a cryptographic hash of the filter code and parameter values to generate a fixed-length cache tag. The cache implementation uses the open-source SQLite embedded database [17] rather than custom server data structures. When a filter is evaluated on an object during a search, the result is

| | |
|---|---|
| get_dev_list | Get the list of servers involved in current search. |
| dev_characteristics | Get performance characteristics of a single server. |
| get_dev_stats | Get search statistics for a single server. Statistics include number of dropped objects, number of searched objects, and total number of objects. |
| set_user_state | Tell servers to log an integer value into their trace logs for later analysis of user state timings. |
| terminate_search_extended | Same as `terminate_search`, but tell each server the number of "queued" and "presented" objects as reported in an application-specific way. |

**Table 4: Debugging and Tuning (Searchlet API)**

| | |
|---|---|
| filter_init | This callback is called once at the beginning of a search. It provides arguments to the filter and a way for the filter to specify some data to be passed along with each object. |
| filter_eval | This callback is called once per object. It provides a handle to the current object and the filter data created in the init call. |
| filter_fini | This callback is called once at the end of the search. It gives the filter one last chance to do any necessary cleanup. |

**Table 5: Callback Functions (Filter API)**

| | |
|---|---|
| next_block | Read data from the object. |
| skip_block | Skip over some data in the object. |

**Table 6: Object Access (Filter API)**

entered with its cache tag in the SQLite database on that server. When that object–filter–parameter combination is encountered again on a subsequent search, the result is available without re-running the potentially expensive filter operation. Note that cache entries are very small (few tens of bytes each) in comparision to typical object sizes.

Attribute caching is the other form of caching in the OpenDiamond platform. Some intermediate attributes can be costly to compute, while others are cheap. Some attributes can

| | |
|---|---|
| `read_attr` | Return a copy of a particular attribute. |
| `ref_attr` | Return a direct reference to a particular attribute. |
| `write_attr` | Create a new attribute. Existing attributes should not be modified. Attributes cannot be deleted. |
| `omit_attr` | Mark a flag on this attribute to hint that this attribute does not need to be sent over a network connection. |
| `first_attr` | Get a pointer to the first attribute and its data. |
| `next_attr` | Given an attribute, get a pointer to the next attribute and its data. |
| `get_session_variables` | Get the values of a subset of session variables. |
| `update_session_variables` | Take a list of session variable names, values, and updater functions of type (double, double) $\rightarrow$ double. Atomically update the given session variables using the updater functions and values. |

**Table 7: Attribute & Session Variable Manipulation (Filter API)**

be very large, and some can be small. It does not make sense to cache attributes that are large and cheap to compute, since this wastes disk space and I/O bandwidth for little benefit. It is best to cache attributes that are small but expensive to generate. To implement this policy, the OpenDiamond platform dynamically monitors filter execution times and attribute sizes. Only when an attribute is below a certain space-time threshold (currently one MB of attribute size per second of computation) is it retained in the SQLite database. During subsequent searches, hits in the attribute cache reduce server load and improve performance.

## 3.4 The OpenDiamond Network Protocol

The OpenDiamond platform uses a network protocol that logically separates control from data. 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 the critical attribute on the control channel; in contrast, high throughput is the critical attribute on the data channel, which we refer to as the *blast channel*. Since many individual searches in a search session tend to be aborted long before completion, the ability to rapidly flush now-useless results in the blast channel would be valuable. This will improve the crispness of response seen by the user, especially on a blast channel with a large bandwidth-delay product. Although this is not feasible on today's Internet, we have structured the system to easily accomodate future networking improvements.

Two separate TCP connections are used for the control and blast channels, thus resulting in a pair of TCP connections between a client and each server involved in a search. Each control/blast pair is associated via a nonce that is created on the server, passed to the client through one connection, and returned on the other. Work is in progress to encrypt these connections.

The control channel uses an RPC library to provide the client synchronous control of various aspects of a search. These controls, such as starting, stopping, or modifying a search, map directly to RPC calls between the OpenDiamond client and server. For example, when the user clicks a button to start a search, this results in an RPC call to each server. We currently use Sun RPC, but are replacing it with a new RPC mechanism (`http://minirpc.cs.cmu.edu/`) that supports encryption and IPv6.

The blast channel works asynchronously, since a single search can generate many results spread over a long period of time. This TCP connection does not use a RPC mechanism, but instead uses a very simple whole-object protocol.

## 3.5 Scoping and Access Control of a Search Session

When determining the subset of objects relevant to a particular search session, the OpenDiamond platform is designed to take advantage of available rich metadata sources. The current implementation uses the concept of *object collections.* The search scoping can be done manually in advance by creating OpenDiamond configuration files, or at runtime by connecting to a web server and downloading these files (possibly generated dynamically by the web server).

Object collections are given semantically meaningful names when they are created by an administrator. Each name is mapped to a *group-ID* that is 16 hex digits long. Each group-ID is associated with one or more servers that contain objects in that group. When a client wants to search through a collection, she must specify the name of the collection, the collection's group ID, and the servers in the group.

Since this is obviously a tedious bookkeeping exercise, we provide a tool that allows this configuration information to be created at runtime from up-to-date server information kept on a *scope server.* A PHP web application, called the *OpenDiamond Gatekeeper,* runs on the scope server and provides a visual interface for selecting collections. It also provides access control, allowing administrators to restrict access to collections by individual users. User authentication is implemented using the extensible access control system provided by the Apache HTTPD Server.

Planned future work includes removing the concept of static object groups entirely, and instead allowing a scope to be specified as the result of a query to a SQL database or other source of structured data such as a patient record system. This architecture is shown in Figure 3. The scope server handles all the details of determining the objects to search, the permissions of each object, and the server location of each object. The scope server hands back an opaque cryptographic token called a *scope cookie* that is effectively a time-limited capability for access to specific objects on specific content servers. This scope cookie is presented to the content servers and remains as implicit context for the search session until the user changes scope or the time limit expires.

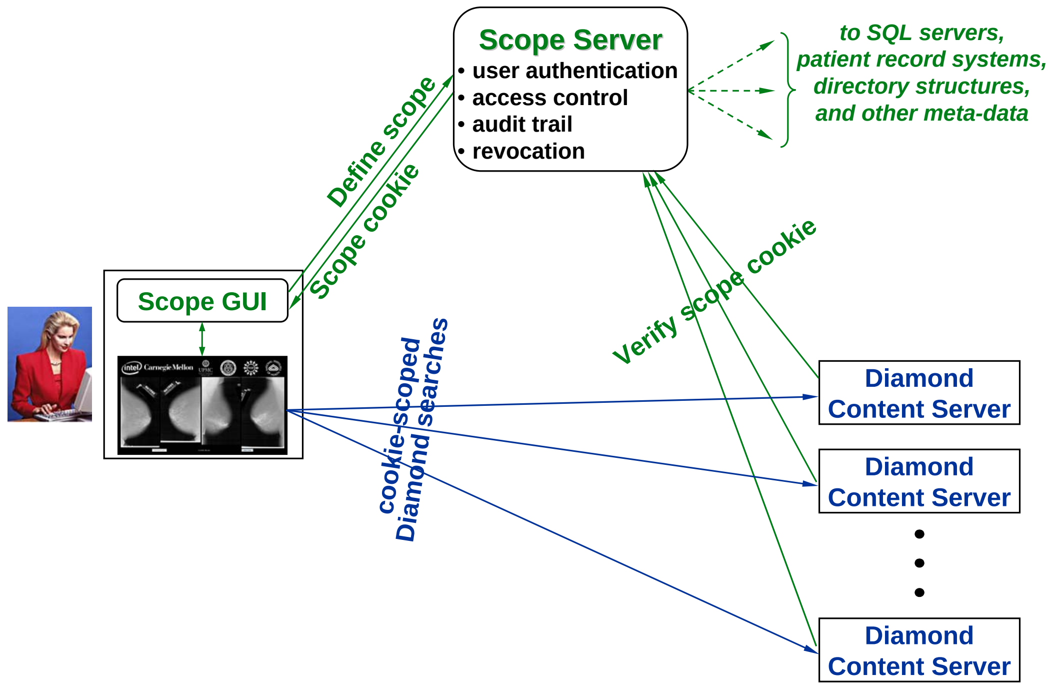


**Figure 3: Scoping a Diamond Search**

## 3.6 Self-Tuning

A key goal of Diamond is to offer good performance by dynamically adapting to changes in data content, client and server hardware, network load, server load, etc. We refer to this concept as *self-tuning*. By authoring an application on this platform, the developer benefits from self-tuning because she is relieved of the need to provide application support for dealing with this complexity of the environment. Self-tuning was one of the earliest areas of investigation in Diamond, and the results have been reported in earlier papers [20, 19]. We therefore provide only a brief summary here.

### 3.6.1 Filter Ordering

Filters in a Diamond search have only partial dependencies on each other. This means that, for example, while both a texture filter and a face detection filter must run after an image decoding filter, the texture filter can run before, after, or in parallel with the face detection filter. Diamond can take advantage of this independence to order the filters to run most efficiently. The filter ordering code of Diamond attempts to order filters so that the cheapest and most discriminating filters will run first. This is achieved in a completely application-independent way by maintaining dynamic measurements of both execution times and discard rates for each filter. This approach is robust with respect to upgrading hardware or installing hardware performance accelerators for specific filters. Further details can be found in an earlier paper [20].

### 3.6.2 Load Balancing

There may be some situations in which it may be advantageous to perform some or all of the processing of objects on the client. For example, if a fast client is accessing an old, slow, heavily-loaded server over an unloaded gigabit LAN, there may be merit in executing some filters on the client. Dynamic load balancing in Diamond is based on queue backpressure, and is thus application-independent. Details can be found in earlier papers [20, 19].

## 4 Applications Built on the OpenDiamond Platform

We have implemented a number of applications on the OpenDiamond platform that give us confidence that this indeed a versatile base for discard-based search. Most of these applications have been built in close collaboration with domain experts from the medical and pharmaceutical areas. We briefly describe a sampling of these applications in the rest of this section.

### 4.1 Searching Unorganized Digital Photographs

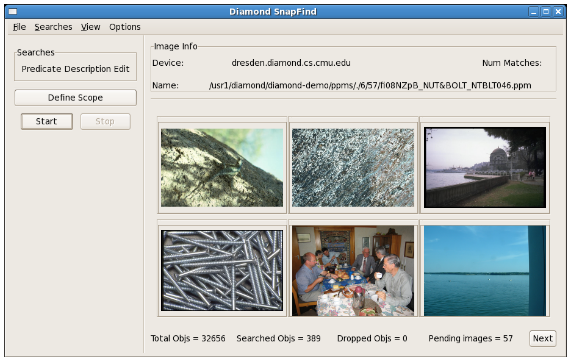
*SnapFind,* which was the very first Diamond application, enables users to interactively search large collections of unlabeled photographs by quickly specifying searchlets that roughly correspond to semantic content. Users typically wish to locate photos by semantic content (for example, “Show me the whale watching pictures from our Hawaii vacation”), but this level of semantic understanding is beyond today’s automated image indexing techniques. As shown in Figure 4(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 pre-defined filter (for example, “frontal human faces”) or create new filters by clicking on sample patches in other images (for example, a “blue jeans” color filter).

### 4.2 Investigating Masses 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 Diamond interactive search. Radiologists can browse cases in the standard four-image view, as shown in Figure 4(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 masses on a mammogram when given a center point to start from. Once a mass is identified, a Diamond search can be invoked to search for similar masses. Distance metrics [42] and visual similarity search are used 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 to help categorize an unknown mass based on similarity to images in an archive.

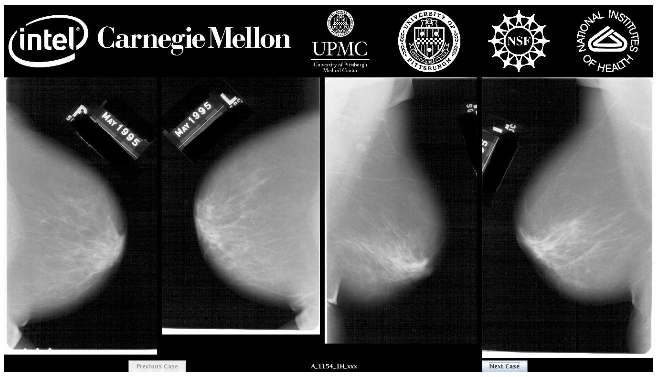
### 4.3 Digital Pathology Search

Based on analysis of expected workflow by a typical pathologist, a Diamond tool called *PathFind* has been developed. As shown in Figure 4(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. The PathFind interface allows the pathologist to identify regions of interest on the slide at any level of magnification and then search for similar regions

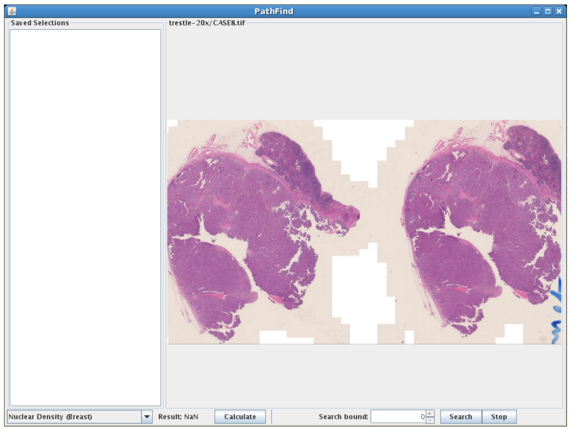


Device: dresden.diamond.cs.cmu.edu
Num Matches:
Name: /usr1/diamond/diamond-demo/ppms/./6/57/fi08NZpB_NUT&BOLT_NTBLT046.ppm
Total Objs = 32656 Searched Objs = 389 Dropped Objs = 0 Pending images = 57

(a) SnapFind

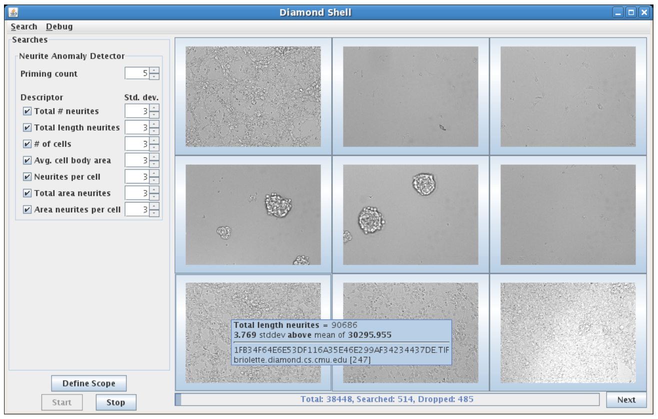


(b) MassFind



trestle-20x/CASE8.tif
Nuclear Density (Breast)
Result: NaN
Search bound: 0

(c) PathFind

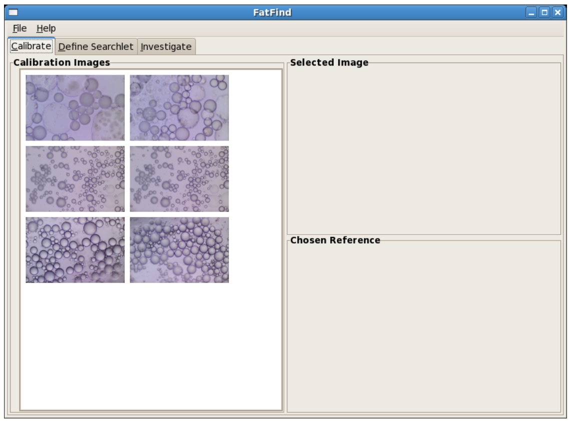


Neurite Anomaly Detector
Priming count 5
Descriptor Std. dev.
✓ Total # neurites 3
✓ Total length neurites 3
✓ # of cells 3
✓ Avg. cell body area 3
✓ Neurites per cell 3
✓ Total area neurites 3
✓ Area neurites per cell 3

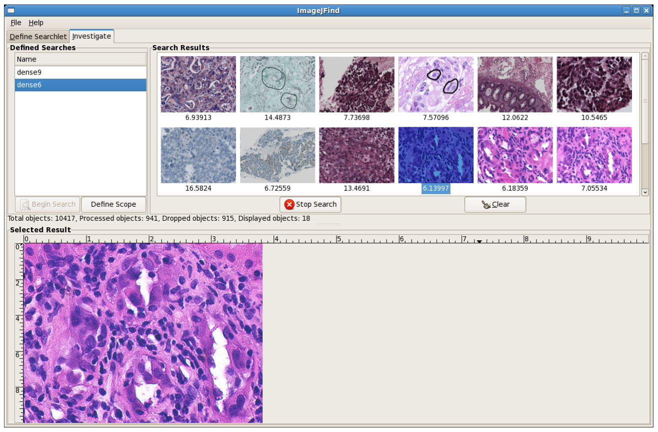
Total length neurites = 90686
3.769 stddev above mean of 30295.955
1F834F64E6E53DF116A35E46E299AF34234437DE.TIF
briolette.diamond.cs.cmu.edu [247]

Total: 38448, Searched: 514, Dropped: 485

(d) StrangeFind



(e) FatFind



Name
dense9
dense6
6.93913 14.4873 7.73698 7.57096 12.0622 10.5465
16.5824 6.72559 13.4691 6.13997 6.18359 7.05534
Total objects: 10417, Processed objects: 941, Dropped objects: 915, Displayed objects: 18

(f) ImageJFind

**Figure 4: Screenshots of Some Diamond Applications**

via Diamond across multiple slide formats. The results returned by Diamond can be viewed and compared with the original image, and the case data for each result can also be retrieved.

## 4.4 Online Anomaly Detection in Automated Cell Microscopy

*StrangeFind* is a Diamond application that does online anomaly detection across different modalities and types of data. It was developed for assisting pharmaceutical researchers in automated cell microscopy, where very high volumes of cell imaging are typical. Figure 4(d) 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) and brightfield neurite images (where the image processing analyzes counts, lengths, and sizes of neurite cells). Since StrangeFind is an online anomaly detector, it does not require a preprocessing step or a predefined statistical model. Instead, it builds up the model as it examines the data. While this can lead to a higher incidence of false positives early in the analysis, the benefits of online detection outweigh the additional work of screening false positives [14].

## 4.5 Quantitating Adipocytes

In the field of lipid research, the measurement of adipocyte size is an important but diffcult problem. A Diamond tool called *FatFind* enables 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 4(e) 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 (based on [24]) to locate the adipocytes in the image, and the investigator chooses one of these as the reference image. He can then define 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.

## 4.6 Leveraging ImageJ Macros as Searchlets

ImageJ is a public domain image processing tool that is supported by the National Institutes of Health. It is widely used as an investigative tool by researchers in cell biology, pathology and other areas. The ability to easily add Java-based plugins, and the ability to record macros of user interaction are two valuable features of the tool. *ImageJFind* is a Diamond application that enables an investigator to use an ImageJ macro as a Diamond searchlet. This enables the researcher to create the macro on a small sample of images, and then apply it to a large collection of images. Figure 4(f) shows a typical screenshot of this application. The implementation of ImageJFind requires a copy of ImageJ to be running at each server, and this process is connected to the OpenDiamond platform as a heavyweight filter. A similar approach has been used to integrate the widely-used MATLAB® tool with Diamond, resulting in a tool called *MATLABFind.*

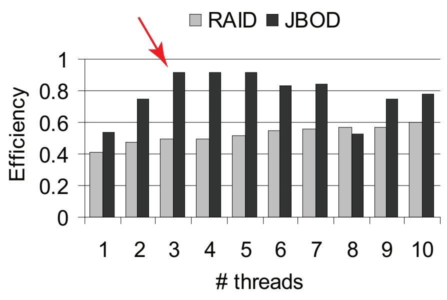
## 5 Storage-friendly Parallelism

The server workloads generated by Diamond applications exhibit high degrees of parallelism that are compatible with storage-embedded processing. There has long been speculation regarding the value of “intelligent storage” or “active disks” (e.g., [27, 28, 41]) in which application-visible processing capability is integrated with persistent storage. Although conceptually promising, the commercial viability of such hypothetical devices has faced a Catch-22: without a large established base of applications that can exploit their novel capability, there is no market for such devices; conversely, software developers have no incentive to create such applications in the absence of supporting hardware.

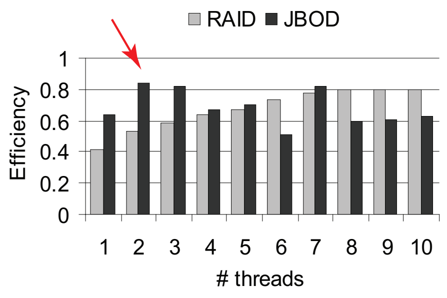
Diamond has the potential to break this deadlock. Software developers are motivated to create new search applications on the OpenDiamond platform because they provide valuable new capabilities to end users. These applications can run on off-the-shelf commodity hardware and operating systems that are available today. Yet, without conscious programmer intent or effort, the resulting server workloads have attributes that are attractive to storage-embedded processing with a high degree of parallelism. As the deployment and use of Diamond applications becomes widespread, it has the potential to create a market for specialized hardware that improves performance in terms of discards per second. A promising first step in this direction is the recent emergence of a commercial product called Netezza [31] that was designed independently of Diamond. As a performance accelerator for streaming data applications on relational databases, Netezza provides application-visible processing close to disk storage. We hope to explore potential synergies between Diamond and Netezza in the future.

To illustrate this distinct character of Diamond workloads, consider two alternative organizations of server storage. One organization is striping, in which a large file is broken into fixed-size units (whose size is the “stripe unit”) and these units are distributed round-robin across a fixed number of disks (this number is the “stripe factor”). Striping, which is the basis of RAID storage, is the dominant server storage organization today. Unfortunately, striping is unfriendly to storage-embedded processing because the embedded processors do not have access to entire objects. The other organization is to simply place each object in its entirety on a single disk, and to map a set of objects round-robin on to the available disks. This is referred to as a JBOD organization (“just a bunch of disks”). JBOD is much friendlier to storage-embedded processing because each object is available in its entirety to the embedded processor. In the context of Diamond, this means that searchlet algorithms do not require radical revision for use in storage-embedded processors.

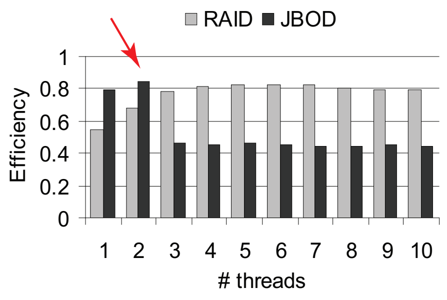
Since discard-based search imposes no particular ordering on objects, they can be read and discarded in the most efficient order for the storage system. On a server with multiple disks and multiple CPUs, the most flexible organization is to have a pool of worker threads assigned to each disk. Since the I/O scheduling is under control of the runtime software 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., server storage looks like a single giant disk), JBOD offers better coupling of storage parallelism to CPU parallelism. This is especially valuable in an early discard system, where there tends to be high variance in processing time across different objects: some objects may be rejected very quickly, others may survive a number of filter stages before rejection, and the very few that pass the entire gauntlet of filters require considerable processing.



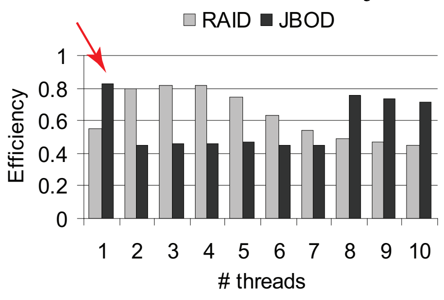
(a) 500 KB objects



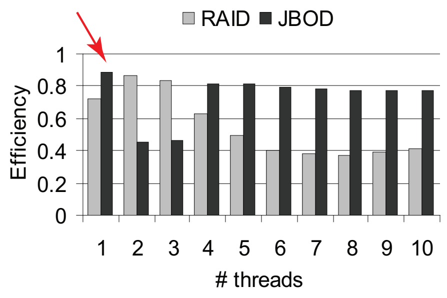
(b) 1 MB objects



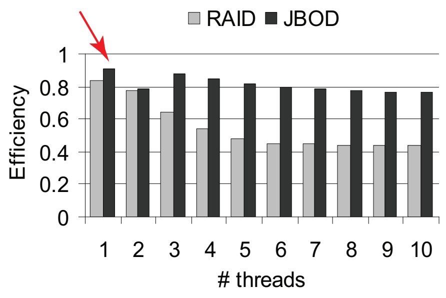
(c) 2 MB objects



(d) 4 MB objects



(e) 8 MB objects



(f) 16 MB objects

**Figure 5: Comparison of RAID and JBOD for Discard-based Search**

Based on our observations of the applications described in Section 4, we designed a synthetic benchmark that generates a discard-based search storage workload. Using an object count and an object size distribution as input parameters, the benchmark creates a synthetic storage repository in which files corresponding to 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 objects in the repository, their sizes and locations.

We ran a series of experiments with this benchmark on an Intel® SSR212CC storage system. The hardware consisted of a 2.8GHz Intel® Xeon™CPU with hyperthreading, 1GB of main memory, and 12 disks each of 200 GB capacity. Five of these disks on one controller were used in a JBOD configuration, while five disks on an identical controller were configured as a RAID array with 64KB stripe units. The operating system on the server was Ubuntu 7.04 with Linux kernel version 2.6.20. We explored object sizes ranging from 500 KB to 16 MB. Our figure of merit in these experiments is I/O efficiency: the ratio of the achieved bandwidth to the maximum sequential bandwidth of the five-disk array. Each of our disks can sustain a maximum transfer rate of 74MB/s, so our aggregate maximum bandwidth is 370MB/s for a five-disk array.

Figure 5(a) through 5(f) present our results for different object sizes. Each graph compares the I/O efficiency of JBOD and RAID at increasing levels of CPU concurrency from left to right. In each graph, the arrow shows the maximum efficiency achieved for a given object size. The results show that across all object sizes, the highest I/O efficiency was achieved with JBOD rather than RAID. This maximum efficiency was attained at a concurrency level of one or two threads. The amount of concurrency needed to saturate the storage system decreases as object size increases. JBOD achieves a higher I/O efficiency at the peak operating point in part because RAID is limited by the speed of the slowest disk.

These results strongly suggest that one does not have to sacrifice I/O efficiency in order to use a configuration(JBOD) that is friendly to storage-embedded processing. This key attribute of discard-based workloads may help to catalyze future innovation in intelligent storage systems.

## 6 Related Work

To the best of our knowledge, Diamond is the first attempt to build a system that enables efficient interactive search of large volumes of complex, non-indexed data. While unique in this regard, Diamond does build upon many insights and results from previous work.

Recent work on *interactive data analysis* [16] outlines a number of new technologies that will be required to make database systems as interactive as spreadsheets — requiring advances in databases, data mining and human-computer interaction. Diamond and early discard are complementary to these approaches, providing a basic systems primitive that furthers the promise of interactive data analysis.

Effective tools exist for analyzing workloads in relational databases to determine those indexes that might be most beneficial e.g., [2, 9]. Indexing methods for text applications are well-known (e.g., [34]). Feature-space indexes (e.g., QBIC [12]) have shown some success in content-based retrieval of whole images and recent work on low-level feature detectors and descriptors (e.g., SIFT [26]) has led to efficient schemes for index-based sub-image retrieval (e.g., [22]. However, all of these methods rely on the hope that the user's semantic needs can be sufficiently characterized by the system's limited concept of keypoints.

In more traditional database research, advanced indexing techniques exist for a wide variety of specific data types including multimedia data [11]. Work on data cubes [15] takes advantage of the fact that many decision support queries are well-known; such queries can be used to pre-process a database and then perform queries directly from the more compact representation. The developers of new indexing technology must constantly keep up with new data types, and with new user access and query patterns. A thorough survey of indexing and the outline of this tension appear in a recent dissertation [35], which also details theoretical and practical bounds on the (often high) cost of indexing.

In addition, in high-dimensionality data (such as feature vectors extracted from images to support indexing), sequential scanning is often competitive with even the most advanced indexing methods because of the *curse of dimensionality* [43, 7, 10]. Efficient algorithms for *approximate* nearest neighbor in certain high-dimensional spaces, such as locality-sensitive hashing [13], are available. However, these require the similarity metric to be known in advance (so that the data can be appropriately pre-indexed using the proximity-preserving hashing functions) and that the similarity metric satisfy certain properties. Diamond addresses searches where neither of these constraints is satisfied.

Work on *approximate query processing*, recently surveyed in [6], complements these efforts by observing that users can often be satisfied with approximate answers when they are simply using query results to iterate through a search problem. There has also been some recent interest in applying link analysis techniques (e.g., [8]) to large collections of unstructured data, such as images on the web [21, 25]. These techniques require significant offline analysis of a large number of documents but the extracted information could complement the Diamond approach. In particular, the existence of multiple semantically-disjoint clusters in real-world data highlights the applicability of efficient discard based search strategies.

In systems research, our work builds on the insight of active disks [1, 23, 32] where the movement of search primitives to extended-function storage devices was analyzed in some detail, including for image processing applications. Additional research has explored methods to improve application performance using active storage [27, 28, 33, 41]. The work of Abacus [3], Coign [18], River [4] and Eddies [5] provide a more dynamic view in heterogeneous systems with multiple applications or components operating at the same time. 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 [5] adaptively reshapes dataflow graphs to maximize performance by monitoring the rates at which data is produced and consumed at nodes. The importance of filter ordering has also been the object of research in database query optimization [36]. The addition of early discard and filter ordering bring a new set of semantic optimizations to all of these systems, while retaining the basic model of observation and adaptation while queries are running.

Standardization efforts in object-based storage devices (OSD) [37] provide the basic primitives on which we build our semantic filter processing. In order to most efficiently process searchlets, active storage devices must contain whole objects, and must understand the low-level storage layout. We can also make use of the attributes that can be associated with objects to store intermediate filter state and to save filter results for possible re-use in future queries. Offloading space management to storage devices provides the basis for understanding data in the more sophisticated ways necessary for early discard filters to operate.

## 7 Conclusion

The past decade has witnessed great improvements in many aspects of disk technology such as capacity, cost per bit, rotational speed, and aggregate bandwidth. Ways to rapidly fill empty disk space have also grown. The emergence of digital photography allows any consumer to easily generate large volumes of image data on his or her personal computer. More importantly, industrial applications of imaging technologies have grown tremendously in scale and importance.

A daunting problem arises from this ease of data creation and storage. How does one find a few vaguely-specified items in many terabytes or petabytes of complex and loosely-structured data such as digital photographs, video streams, CAT scans, AutoCAD drawings, or USGS maps? If the data has already been indexed for the query being posed, the solution is well-understood. Unfortunately, a suitable index is often not available and a user has no choice but to perform an exhaustive search over the entire volume of data. While author, date, and other meta-data can be used to restrict the search space, the user still has an enormous number of items to examine.

Without a system like Diamond, scanning such a large volume of data would be so slow that it could only performed in the context of well-planned data mining. Enabling this kind of search allows users to discover a small set of relevant items buried in a huge collection. And as discussed in this paper, aggressive filtering of the data is essential to permit the user to focus his or her limited attention on the most promising candidates. To enable the creation of applications for interactive exploration of non-indexed data, we have created the OpenDiamond platform. Instead of precomputing indexes for all anticipated queries, the OpenDiamond platform embodies support for discard-based search. This approach performs content-based computation in response to a specific query. Our rethinking of search from first principles has favorable consequences for flexibility and user control, while also enabling easy exploitation of CPU and storage parallelism on current and future hardware.

## Acknowledgements

This research was 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. OpenDiamond is a registered trademark of Carnegie Mellon University.

We thank Casey Helfrich, Limor Fix, Mei Chen, Phil Gibbons, and Le Zou for their contributions to Diamond. The MassFind application described in Section 4.2 and the PathFind application described in Section 4.3 were developed with support from the Clinical and Translational Sciences Institute of the University of Pittsburgh, with funding from the National Center for Research Resources under Grant No. 1 L1 RR024153. We thank Bin Zheng, Ty Bae, Drazen Jukic, Jonathan Ho, and Laura Drogowski for their contributions to these applications. Bin Zheng implemented the mass contour tool in MassFind that is mentioned in Section 4.2. The StrangeFind application described in Section 4.4 and the FatFind application described in Section 4.5 were developed in collaboration with Merck & Company, Inc. We thank Jeff Saltzman, Anil Tarachandani, and Dave Ross for their contributions to these applications.

## References

[1] ACHARYA, A., UYSAL, M., AND SALTZ, J. Active disks: Programming model, algorithms and evaluation. In *Proceedings of ASPLOS* (1998).

[2] AGRAWAL, S., CHAUDHURI, S, NARASAYYA, V. R. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of VLDB* (2000).

[3] AMIRI, K., PETROU, D., GANGER, G., AND GIBSON, G. Dynamic function placement for data-intensive cluster computing. In *Proceedings of USENIX* (2000).

[4] ARPACI-DUSSEAU, R., ANDERSON, E., TREUHAFT, N., CULLER, D., HELLERSTEIN, J., PATTERSON, D., AND YELICK, K. Cluster I/O with River: Making the fast case common. In *Proceedings of Input/Output for Parallel and Distributed Systems* (1999).

[5] Avnur, R., and Hellerstein, J. Eddies: Continuously adaptive query processing. In *Proceedings of SIGMOD* (2000).

[6] Babcock, B., Chaudhuri, S., and Das, G. Dynamic sample selection for approximate query processing. In *Proceedings of of SIGMOD* (2003).

[7] 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).

[8] Brin, S., and Page, L. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems 30*, 1–7 (1998).

[9] Chaudhuri, S., Narasayya, V. AutoAdmin "What-if" Index Analysis Utility. In *Proceedings of ACM SIGMOD 1998* (June 1998).

[10] Duda, R., Hart, P., Stork, D. *Pattern Classification*. Wiley, 2001.

[11] Faloutsos, C. *Searching Multimedia Databases by Content*. Kluwer Academic Inc., 1996.

[12] Flickner, M. and Sawhney, H. and Niblack, W. andAshley, J. and Huang, Q. and Dom, B. and Gorkani, M. and Hafner, J. and Lee, D. and Petkovic, D. and Steele, D. and Yanker, P. Query by Image and Video Content: The QBIC System. *IEEE Computer 28*, 9 (1995).

[13] Gionis, A., Indyk, P., and Motwani, R. Similarity search in high dimensions via hashing. In *Proceedings of VLDB* (1999).

[14] Goode, A., Sukthankar, R., Mummert, L., Chen, M., Saltzman, J., Ross, D., Szymanski, S., Tarachandani, A., and Satyanarayanan, M. Distributed Online Anomaly Detection in High-Content Screening. In *Proceedings of the 2008 5th IEEE International Symposium on Biomedical Imaging* (Paris, France, May 2008).

[15] Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., and Venkatrao, M. Data Cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery 1* (1997).

[16] Hellerstein, J., Avnur, R., Chou, A., Hidber, C., Raman, V., Roth, T., and Haas, P. Interactive data analysis: The CONTROL project. *IEEE Computer* (August 1999).

[17] Hipp, D. R., and Kennedy, D. SQLite. http://www.sqlite.org/.

[18] Hunt, G., and Scott, M. The Coign automatic distributed partitioning system. In *Proceedings of OSDI* (1999).

[19] Huston, L., Nizhner, A., Pillai, P., Sukthankar, R., Steenkiste, P., and Zhang, J. Dynamic load balancing for distributed search. In *HPDC '05: Proceedings of the High Performance Distributed Computing, 2005. HPDC-14. Proceedings. 14th IEEE International Symposium* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 157–166.

[20] 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).

[21] Jing, Y., and Baluja, S. PageRank for product image search. In *Proceedings of WWW* (2008).

[22] Ke, Y., Sukthankar, R., and Huston, L. Efficient near-duplicate and sub-image retrieval. In *Proceedings of ACM Multimedia* (2004).

[23] Keeton, K., Patterson, D., and Hellerstein, J. A case for intelligent disks (IDISKs). *SIGMOD Record 27*, 3 (1998).

[24] Kim, E., Haseyama, M., and Kitajima, H. Fast and Robust Ellipse Extraction from Complicated Images. In *Proceedings of IEEE Information Technology and Applications* (2002).

[25] KIM, G., HEBERT, M., AND FALOUTSOS, C. Unsupervised modeling of object categories using link analysis techniques. In *Proceedings of IEEE Computer Vision and Pattern Recognition* (2008).

[26] LOWE, D. Distinctive image features from scale-invariant keypoints. *International Journal on Computer Vision* (2004).

[27] MA, X., AND REDDY, A. MVSS: An Active Storage Architecture. *IEEE Transactions On Parallel and Distributed Systems 14*, 10 (2003).

[28] MEMIK, G., KANDEMIR, M., AND CHOUDHARY, A. Design and evaluation of smart disk architecture for DSS commercial workloads. In *International Conference on Parallel Processing* (2000).

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

[30] 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).

[31] NETEZZA CORPORATION. Netezza. http://www.netezza.com/.

[32] RIEDEL, E., GIBSON, G., AND FALOUTSOS, C. Active storage for large-scale data mining and multimedia. In *Proceedings of VLDB* (August 1998).

[33] RUBIO, J., VALLURI, M., AND JOHN, L. Improving transaction processing using a hierarchical computing server. Tech. Rep. TR-020719-01, Laboratory for Computer Architecture, The University of Texas at Austin, July 2002.

[34] SALTON, G., AND MCGILL, M. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[35] SAMOLADAS, V. *On Indexing Large Databases for Advanced Data Models*. PhD thesis, University of Texas at Austin, August 2001.

[36] SELINGER, P., ASTRAHAN, M., CHAMBERLIN, D., LORIE, R., AND PRICE, T. Access path selection in a relational database management system. In *Proceedings of SIGMOD* (1979).

[37] T10 TECHNICAL COMMITTEE. ANSI T10/1355-D: SCSI Object-Based Storage device commands (OSD). http://www.t10.org/ftp/t10/drafts/osd/, September 2003.

[38] 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).

[39] 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).

[40] 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).

[41] WICKREMISINGHE, R., VITTER, J., AND CHASE, J. Distributed computing with load-managed active storage. In *In IEEE International Symposium on High Performance Distributed Computing (HPDC-11)* (2002).

[42] 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).

[43] 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).