Searching Complex Data Without an Index

Mahadev Satyanarayanan[†], Rahul Sukthankar[‡], Adam Goode[⋆], Nilton Bila[•], Lily Mummert[‡], Jan Harkes[†], Adam Wolbach[†], Larry Huston, Eyal de Lara[•]
[†]Carnegie Mellon Univ., [‡]Intel Labs Pittsburgh, [⋆]Google, Inc., [•]Univ. of Toronto

We show how *query-specific content-based computation* can be used for interactive search when a pre-computed index is not available. Rather than text or numeric data, we focus on complex data such as digital photographs and medical images. We describe a system that can perform such interactive searches on stored data as well as live Web data. The system is able to narrow the focus of a non-indexed search by using structured data sources such as relational databases. It can also leverage domain-specific software tools in search computations. We report on the design and implementation of this system, and its use in the health sciences.

Keywords: Non-indexed Search, Complex Data Query, Interactive Image Search, OpenDiamond Project

1. INTRODUCTION

Indexing through preprocessing is the basis of search today. Unfortunately, as described in Section 2, there are real-world situations where we don't know how to automate the creation of an index. In addition, it is not possible to preprocess live sources of data such as a collection of webcams on the Internet. How can one search such data?

In this paper, we show how *query-specific content-based computation* can be used for interactive search when a pre-computed index is not available. Our emphasis on interactive search makes *user attention* more precious than any system resource. Rather than text or numeric data, our focus is on complex data such as digital photographs, medical images, and speech recordings.

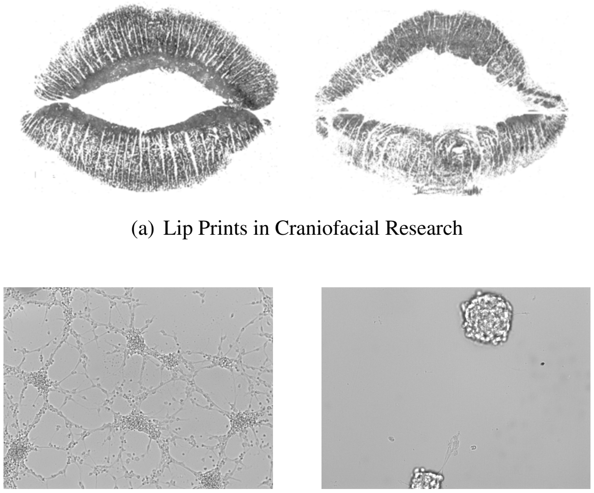
We have been exploring this approach since late 2002 in the context of the *Diamond* project. An early description of Diamond was published in 2004 [Huston et al. 2004]. Since then we have gained considerable experience in applying our approach to real-world problems in the health sciences. This experience has led to extensive evolution and redesign of our early prototype, resulting in open-source middleware called the *OpenDiamond ® platform.* For ease of exposition, we use the term "Diamond" flexibly: 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.

We begin in Section 2 with two motivating examples drawn from our actual experience. Our early Diamond prototype is summarized in Section 3, and its transformation through our collaborations is presented in Section 4. We describe the design and implementation of our current system in Section 5, then validate it in two parts: versatility in Section 6.1, and interactive performance in Section 6.2. Section 7 discusses related work, and Section 8 closes with a discussion of future work.

2. MOTIVATING EXAMPLES

Faced with an ocean of data, how does an expert formulate a crisp hypothesis that is relevant to his task? Consider Figure 1(a), showing two examples of lip prints from thousands collected worldwide by craniofacial researchers investigating the genetic origins of cleft palate syndrome. From genetic and developmental reasoning, they conjecture that even asymptomatic members of families with the genetic defect will exhibit its influence in their finger prints and lip prints. Of the many visual differences between the left image (control) and the right image (from a family with cleft palate members), which are predictive of the genetic defect? What search tools can the systems community offer a medical researcher in exploring a large collection of lip prints to answer this question?

Another example pertains to the pharmaceutical industry. Long-running automated experiments for



(a) Lip Prints in Craniofacial Research

(b) Neuronal Stem Cell Growth

Figure. 1: Examples of Hard-to-Index Data

investigating drug toxicity produce upwards of a thousand high-resolution cell microscopy images per hour for many days, possibly weeks. Monitoring this imaging output for anomalies, often different from previous anomalies, is the task of human investigators. Figure 1(b) shows two examples of neuronal stem cell images from such an experiment. The left image is expected under normal growth conditions. The right image is an anomaly. After discovering this anomaly (itself a challenging task), an investigator has to decide whether it is a genuine drug effect or if it arises from experimental error such as loss of reagent potency or imaging error. For some errors, aborting the entire experiment immediately may be the right decision. What search tools exist to help discover such anomalies in real time and to see if they have occurred before?

3. PRELIMINARY PROTOTYPE

Our early thinking on this problem was strongly influenced by the possibility of using specialized hardware. 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. Work published in the late 1990s on *active disks* [Acharya et al. 1998; Keeton et al. 1998; Memik et al. 2000; Riedel et al. 1998] suggested that application-level processing embedded within storage was feasible and offered significant performance benefits. Extending this approach to searching complex non-indexed data appeared to be a promising path.

As a first step, we built a software prototype to emulate active disks that were specialized to the task of searching complex data. Our primary goal was to gain an understanding of the mechanisms that would be needed for a hardware implementation. A secondary goal was to verify that interactive search applications could indeed be built on an active disk interface. *Early discard,* or the application-specific rejection of irrelevant data as early as possible in the pipeline from storage to user, emerged as the key mechanism required. It improves scalability by eliminating a large fraction of the data from most of the pipeline. We refer to the application-specific code to perform early discard as a *searchlet,* and this overall search approach as *discard-based search.*

The focus of our 2004 paper [Huston et al. 2004] was a detailed quantitative evaluation of this prototype. Qualitatively, those results can be summarized as follows:

- ✦ Queue back-pressure can be effectively used for dynamic load balancing in searchlet execution between the back-end (storage) and front-end (user workstation). Such load balancing can partly compensate for slower improvement in embedded computing hardware performance, relative to desktop hardware.

✦ Application-transparent runtime monitoring of the computational cost and selectivity of searchlet components is feasible, and 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.

4. EXPERIENCE-DRIVEN EVOLUTION

In the years since our 2004 paper, we have gained considerable experience in applying our approach to real-world problems. The insights we acquired changed the strategic direction of the project and led to the re-design and re-implementation of many aspects of our system. Foremost among these changes was a re-thinking from first principles of the need for specialized hardware. Based on the positive outcome of our preliminary prototype, the natural next step would have been for us to build active disk hardware. However, actual usage experience with commodity server hardware gave pause to our original assumption that user experience would be unacceptable.

In an example application to search digital photographs, we found user think time to be sufficiently high that it typically allowed servers to build up an adequate queue of results waiting to be displayed to the user. Think time often increased in the later stages of an iterative search process, as a user carefully considered each result to distinguish true positives from false positives. Search tasks rarely required the corpus of data to be searched to completion; rather, the user quit, once she found enough hits for her query. Only rarely was a user annoyed because of slow rate of return of results.

Even with a compute-intensive searchlet, such as one incorporating image processing for face detection, the presence of multiple servers working in parallel typically yielded a result rate that avoided user stalls after a brief startup delay. Discard-based search is embarrassingly parallel because each data object is considered independently of all others. It is therefore trivial to increase search throughput by adding more servers. This affinity for CPU parallelism and storage parallelism aligns well with today's industry trend towards higher numbers of processing cores per chip and the improvements in capacity and price/performance of storage.

Based on these insights, we decided to defer building hardware. Instead, we continued with a software-only strategy and sought collaborations with domain experts to address real-world problems. While collaborators from many domains expressed interest, the strongest responses came from the health sciences. Extensive revision of many aspects of our system resulted from our multi-year collaborations. Today, our system cleanly separates application-specific and application-independent functionality, with the latter encapsulated into Linux middleware based on standard Internet component technologies. The key considerations underlying this evolution were as follows:

✦ *Exploit temporal locality in searchlets*
We often observed a distinctive usage pattern that we call *interactive data exploration.* In a typical search session, a user's formation and validation of hypotheses about the data is interleaved in a tightly-coupled, iterative sequence. This leads to significant overlap in searchlet components as a search progresses. Caching execution results at servers can exploit this temporal locality. To take advantage of partial overlap of searchlets, the cache can be maintained at the granularity of searchlet components. Over time, cache entries will be created for many objects on frequently-used combinations of searchlet components and parameters, thus reducing the speed differential with respect to indexed search. This can be viewed as a form of just-in-time indexing that is performed incrementally at run time.

✦ *Unify indexed and discard-based search*
A recurring theme in our collaborations was the need to use information stored in a structured data source (such as a relational database) to constrain the search of complex data objects. Consider, for example, "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.

✦ *Enable use of domain-specific tools for searchlets*
We observed several instances where a particular tool was so widely used that it was the basis of discourse among domain experts. For example, *ImageJ* is an image processing tool from the National

Institutes of Health (NIH) that is widely used by cell biology researchers. Another example is the use of MATLAB for computer vision algorithms in bioengineering. Enabling searchlets to be created with such tools simplifies the use of our system by domain experts, and also leverages the power of those tools.

✦ *Streamline result transmission*
Initially, we did not pay attention to the efficiency of result transmission from server to client since this took place over a LAN. Over time, our thinking broadened to include Internet-wide searches. This required protocol changes for efficiency over WANs, as well changes in the handling of results. Rather than always shipping results in full fidelity, we now ship results in low fidelity. The full-fidelity version of an object is shipped only on demand. Since "fidelity" is an application-specific concept, this required end-to-end changes in our system design.

✦ *Balance versatility and customization*
Another recurring theme was the tension between quick incremental extension of existing applications, and the creation of new applications with more domain-specific support and a more natural workflow for domain experts — attributes that proved to be important in successful collaborations. We learned how to separate searchlet development (involving, for example, image processing) from the user interaction and workflow aspects of an application. Prioritizing searchlet development often exposed deep contextual issues and assumption mismatches early in the collaboration process. We thus arrived at an approach in which a single generic application (described in Section 6.1.1) acts as a kind of "Swiss army knife" for searchlet development. Only when searchlet development has proceeded far enough to be confident of success, do we begin the effort to design the rest of an application.

✦ *Enable search of live data sources*
Until recently, our focus was solely on searching stored data. The growing importance of live data sources on the Web, such as traffic monitoring cameras, suggested that it might be valuable to extend our system to search live data. While significant back-end extensions were necessary to enable this functionality, we were gratified that no application-level changes were required. To a user, searching a live stream of images from the Web appears just like searching images on server disks.

✦ *Re-engineer the code base*
As a proof-of-concept artifact, our initial prototype had numerous limitations. Over time, many aspects of the system required extensive re-writing in order to improve efficiency, robustness, portability and maintainability. For example, we replaced the entire communication layer for these reasons.

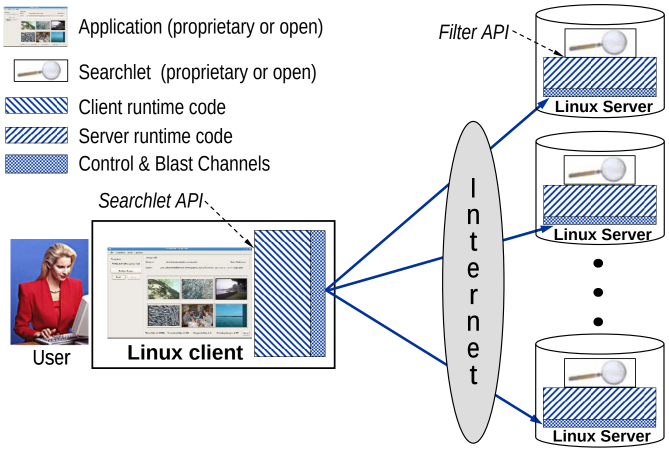


Figure. 2: System Architecture

5. DESIGN AND IMPLEMENTATION

As shown in Figure 2, our system architecture cleanly separates a domain-independent runtime platform from domain-specific application code. The runtime platform consists of client and server code, as well as a TCP-based network protocol. User interaction typically occurs through a domain-specific GUI.

For each search, the user defines a searchlet from individual components called *filters.* A filter consists of executable code plus parameters that tune the filter to a specific target. For example, an image search for women in brown fur coats might use a searchlet with three filters: a color histogram filter with its color parameter set to the RGB value of the desired shade of brown; a texture filter with examples of fur patches as its texture parameter; and a face detection filter with no parameters. The searchlet is submitted by the application via the *Searchlet API,* and is distributed by the runtime system to all of the servers involved in the search task. Each server has a persistent cache of filter code, to avoid retransmissions.

Each server iterates through its local objects in a system-determined order and presents them to filters for evaluation through the *Filter API.* Each filter can independently discard an object. The details of filter evaluation are totally opaque. The scalar return value 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 are transmitted to the client.

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-realm searches. If a user has privileges to search servers individually in different authentication realms, 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 6.1 confirm that this architectural constraint is a good tradeoff. Only in one instance (the online anomaly detection application in Section 6.1.5) 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 5.1.

5.1 Programming Interface

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. We provide details of each below.

5.1.1 *Searchlet API.* Table I lists the calls of the Searchlet API, grouped by logical function. For brevity, we omit the calls for initialization and debugging. An application first defines filters and searchlets through the calls in Table I(a). These are transmitted to each server involved in the current search. Next, in response to user interaction, the application initiates and controls searches using the calls in Table I(b). The application first defines what it means by a low-fidelity result by calling `set_push_attrs( )`. Then, after issuing `start_search( )`, the application calls `next_object( )` repeatedly as a result iterator. At any point, typically in response to a user request, the full-fidelity version of a result can be obtained by calling `reexecute_filters( )`. When the user aborts the current search and goes back to selecting a new filter or changing parameters, the calls in Table I(a) again apply.

The calls in Table I(c) allow the client to obtain a small amount of side effect data from each server and to disseminate them to all servers. As mentioned in the previous section, this was motivated by online anomaly detection but can be used in any application that requires a small amount of periodic information sharing across servers during a search.

5.1.2 *Filter API.* Table II present the Filter API. Each filter provides the set of callback functions shown in Table II(a), and runtime code on the server invokes these functions once for each object. Within `filter_eval( )`, the filter code can use the calls in Table II(b) to obtain the contents of the current object. It can use the calls in Table II(c) to get and set *attributes* associated with the object. Attributes are name-value pairs that 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

set_searchlet()	Define current searchlet by loading and parsing a specification.
add_filter_file()	Load a binary file corresponding to a filter in the current searchlet.
set_blob()	Set binary argument for a filter.

(a) Defining a Searchlet

set_push_attrs()	Indicate which attributes are to be included in the low-fidelity results returned on the blast channel.
start_search()	Start a search.
next_object()	Get the next object from the result queue.
num_objects()	Get the number of pending objects in the current processing queue.
reexecute_filters()	Return full-fidelity version of specified object, after re-executing all filters on it.
release_object()	Free a previously returned object.
terminate_search()	Abort current search.

(b) Controlling a Search

get_dev_session_vars()	Get names and values of session variables on a server.
set_dev_session_vars()	Set a server's session variables to particular values given here.

(c) Session Variable Handling

Table I: Searchlet API

image data attribute and emit a 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. Early discard strives to eliminate an object from this pipeline after the smallest possible investment of total filter execution time.

The calls in Table II(d) allow a filter to examine and update session variables on a server. The use of these variables is application-specific, but the intent is to provide a low-bandwidth channel for annotational information that is continuously updated during a search.

The runtime code on a server iterates through objects in an unspecified order. This *any-order semantics* gives the storage subsystem an important degree of freedom for future performance optimizations. For example, it could perform hardware-level prefetching or caching of objects and have high confidence that those optimizations will improve performance. In contrast, a classical I/O API that gives control of object ordering to application code may or may not benefit from independent hardware optimizations. This aspect of the Filter API design ensures that applications written today are ready to benefit from future storage systems that exploit any-order semantics.

5.2 Result and Attribute Caching

Caching on servers takes two different forms: *result caching* and *attribute caching.* Both are application-transparent, and invisible to clients except for improved performance. Both caches are persistent across

filter_init()	Called once at the start of a search.
filter_eval()	Called once per object, with its handle and any data created in the init call.
filter_fini()	Called once at search termination.

(a) Callback Functions

next_block()	Read data from the object.
skip_block()	Skip over some data in the object.

(b) Object Access

read_attr()	Return value of specified attribute.
ref_attr()	Return reference to specified attribute.
write_attr()	Create a new attribute. Attributes cannot be modified or deleted.
omit_attr()	Indicate that this attribute does not need to be sent to client.
first_attr()	Get first attribute-value pair.
next_attr()	Iterator call for next attribute-value pair.

(c) Attribute Handling

get_session_vars()	Get the values of a subset of session variables.
update_session_vars()	Atomically update the given session variables using the updater functions and values.

(d) Session Variable Handling

Table II: Filter API

server reboots and are shared across all users. Thus, users can benefit from each others' 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 search task, can give rise to significant communal locality in filter executions. As mentioned earlier, result caching can be viewed 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 [SQLite] rather than custom server data structures. When a filter is evaluated on an object during a search, the result is 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 comparison to typical object sizes.

Attribute caching is the other form of caching in our system. 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. The most valuable attributes to cache are those that are small but expensive to generate. 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 one MB of size per second of computation) are cached. As processor speeds increase, certain attributes that used to be cached may no longer be worth caching.

5.3 Network Protocol

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 the critical attribute on the control channel, while 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.

The control channel provides the client synchronous control of various aspects of a search. The library includes calls for starting, stopping, modifying a search, requesting a full-fidelity object, and so on. The blast channel works asynchronously, since a single search can generate many results spread over a long period of time. This TCP connection uses a simple whole-object streaming protocol. Each object in the blast channel is tagged with a search id, to distinguish between current results and obsolete results.

5.4 Self-Tuning

At runtime, our platform dynamically adapts to changes in data content, client and server hardware, network load, server load, etc. This relieves an application developer of having to deal with this complexity of the environment.

Data content adaptation occurs by transparent filter reordering, with some hysteresis for stability. In a typical searchlet, filters have partial dependencies on each other. For example, a texture filter and a face detection filter can each be run only after an image decoding filter. However, the texture filter can run before, after, or in parallel with the face detection filter. The filter ordering code 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 (such as hardware for face detection or recognition) for specific filters.

As mentioned earlier, dynamic load balancing in our system is based on queue backpressure, and is thus application-independent. There may be some situations in which it is 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 even though it violates the principle of early discard.

5.5 External Scoping of Searches

Many use cases of our system involve rich metadata that annotates the raw data to be searched by content. In a clinical setting, for example, patient record systems often store not only the raw data produced by laboratory equipment but also the patient's relevant personal information, the date and time, the name of the attending physician, the primary and differential diagnoses, and many other fields. The use of prebuilt indexes on this metadata enables efficient selection of a smaller and more relevant subset of raw data to search. We refer to this selection process as *scoping a discard-based search.* Effective scoping can greatly improve search experience and result relevance.

Our early research focused exclusively on discard-based search, and treated indexed search as a solved problem. Hence, the original architecture shown in Figure 2 ignored external metadata sources. Figure 3 shows how that architecture has been modified for scoping. The lower part of this figure pertains to discard-based search, and is unmodified from Figure 2. It can be loosely viewed as the "inner loop" of an overall search process. No changes to application code, searchlets, or the server runtime system are

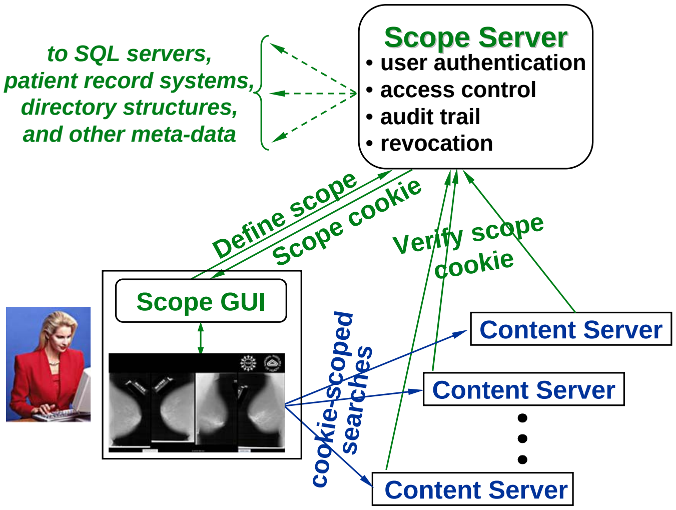


Figure. 3: Scoping a Discard-based Search

needed in moving from Figure 2 to Figure 3 — only a few small changes to the client runtime system.

The system extensions for scoping recognize that many valuable searches may span administrative boundaries. Each administrative unit (with full autonomy over access control, storage management, auditing, and other system management policies) is represented as a *realm* in our architecture. Realms can make external business and security arrangements to selectively trust other realms.

Each realm has a single logical *scope server,* that may be physically replicated for availability or load-balancing using well-known techniques. A user must authenticate to the scope server in her realm at the start of a search session. For each scope definition, a scope server issues an encrypted token called a *scope cookie* that is essentially a capability for the subset of objects in this realm that are within scope. The fully-qualified DNS hostnames of the content servers that store these objects is visible in the clear in an unencrypted part of the scope cookie. This lets the client know which content servers to contact for a discard-based search. However, the list of relevant objects on those content servers is not present in any part of the scope cookie. That list (which may be quite large, if many objects are involved) is returned directly to a content server when it presents the scope cookie for validation to the scope server. Figure 3 illustrates this flow of information. Scope cookies are implemented as encrypted X.509 certificates with lifetimes determined by the scope server.

For a multi-realm search, there is a scope cookie issued by the scope server of each realm that is involved. In other words, it is a federated search in which the issuing and interpretation of each realm's scope cookies occur solely within that realm. A client and its scope server are only conduits for passing a foreign realm's scope cookie between that realm's scope server and its content servers. A user always directs her scope queries to the scope server in her realm. If a query involves foreign realms, her scope server contacts its peers in those realms on her behalf.

A user generates a metadata query via a Web interface, labeled "Scope GUI" in Figure 3. The syntax and interpretation of this query may vary, depending on the specific metadata source that is involved. The scope cookie that is returned is passed to the relevant domain-specific application on the client. That application presents the cookie when it connects to a content server. The cookie applies to all `start_search( )` calls on this connection. When a user changes scope, new connections are established to relevant content servers. Thus, the "inner loop" of a discard-based search retains the simplicity of Figure 2, and only incurs the additional complexity of Figure 3 when scope is changed.

When the full functionality of external metadata scoping is not required, our system can also be set up to scope at the coarse granularity of *object collections.* A much-simplified scope server, implemented as a PHP application, provides a Web interface for selecting collections.

5.6 Live Data Sources

Support for searching live data sources was a simple extension of the system for searching stored data. No changes to applications or to the client runtime system were necessary. The only modification to the server runtime system was a change to the mechanism for obtaining object identities. Rather than reading names of objects from a file we now read them from a TCP socket.

Separate processes, called *data retrievers,* are responsible for acquiring objects from arbitrary sources, saving the objects locally, and providing content servers with a live list of these objects. With this design, it is easy to incorporate new sources of data. All that is required is the implementation of a data retriever for that data source. We have implemented the following data retrievers, each requiring less than 250 lines of Python code:

- **File:** This simple retriever takes a list of files and provides them to the search system without further processing. It mimics the old behavior of the system, before support for live data was added.
- **Web image crawl:** This retriever crawls a set of URLs, extracting images as input to the search.
- **Video stream:** This retriever takes a set of live video stream URLs and saves periodic frame snapshots of the video for searching. The original videos are also reencoded and saved locally so that searches can refer back to the original video which may otherwise be unavailable.
- **Webcam stream:** This retriever is a simpler version of the video stream retriever. It is designed to work with a URL that returns a new static video frame each time it is retrieved.

The processing flow of live data retrieval is simple. First, a master process with a list of initial data sources is started. The master spawns and controls a set of data retriever processes as workers. Each worker pulls one or more data sources off the work list, processes them, and generates one or more objects in the local file system. The names of these new objects is added to the list awaiting discard-based search. Optionally, a worker may add more work to the work list. For example, in the case of Web crawling, the work list would contain a list of URLs; the worker would fetch each URL, save images pointed to by `<img>` and `<A>` tags, and add new URLs back to the work list. Workers continue until the work list is empty; in some cases, as with a webcam, this may never happen.

6. VALIDATION

Two broad questions are of interest to us:

- *How versatile is the system?*
  Is it feasible to build a wide range of applications with it? How clean is the separation of domain-specific and domain-independent aspects of the system? Does it effectively support the use of domain-specific tools, interfaces and workflows?
- *How good is interactive performance?*
  Can users easily conduct interactive searches of non-indexed data? Are they often frustrated by the performance of the system? Are they able to search data from a wide range of sources? Do they easily benefit from the additional system resources such as servers?

6.1 Versatility

Over a multi-year period, we have gained confidence in our approach to searching complex data by implementing diverse applications. The breadth and diversity of these applications speaks for the versatility of our system. As explained in Section 4, it was the process of working closely with domain experts to create these applications that exposed limitations in our system and in our thinking about discard-based search, and guided us through extensive evolution to the current system described in Section 5. 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 our system to interactive search of large collections of virtual machine images. To encourage such collaborations, we have made our system available open-source under the Eclipse Public License.

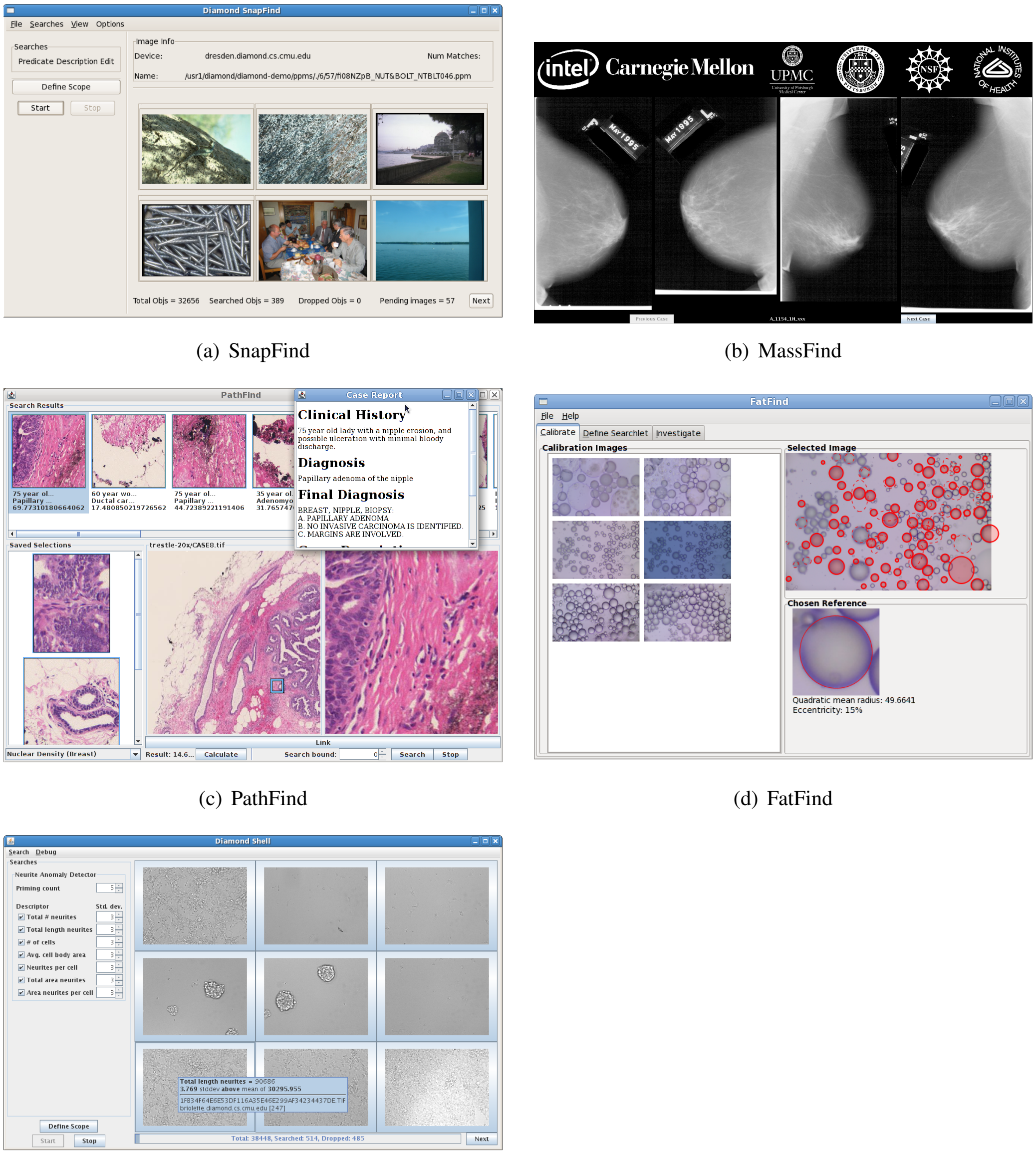
6.1.1 *Unorganized Digital Photographs. SnapFind,* which was the only application available at the time of our 2004 paper, enables users to interactively search large collections of unlabeled photographs by quickly creating searchlets that roughly correspond to semantic content. Users typically wish 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). Details of this image processing have been reported elsewhere [Huston et al. 2004].

Our extensive modifications to SnapFind since 2004 have created a new tool called *HyperFind* that supports a much wider range of filters, an improved GUI, and streamlined result shipping as described in Section 4. HyperFind supports filters created as ImageJ macros. As an NIH-supported image processing tool, ImageJ 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 HyperFind 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 our server runtime system. A similar approach has been used to integrate the widely-used MATLAB tool. This proprietary tool is an interpreter for matrix manipulations that are expressed in a specialized programming language. It is widely used by researchers in computer vision and machine learning. 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 the versatility of our system, but an efficient implementation is likely to be challenging because of the overhead of VM boundary crossings.

Today, we use HyperFind in the early stages of any collaboration that involves some form of imaging. Since the GUI is domain-independent, customized filters for the new domain can be written in ImageJ or MATLAB, and rapidly tested without building a full-fledged application with customized GUI and workflow. Only after early searchlet testing indicates promise does that overhead have to be incurred.

6.1.2 *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 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 lesions on a mammogram when given a center point to start from. 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 [Yang et al. 2010], 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.

6.1.3 *Digital Pathology.* Based on analysis of expected workflow by a typical pathologist, a 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 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.



Device: dresden.diamond.cs.cmu.edu

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

75 year ol... Papillary ... 69.77310180664062

60 year wo... Ductal car... 17.480850219726562

75 year ol... Papillary ... 44.72389221191406

35 year ol... Adenomyo... 31.765747

trestle-20x/CASE8.tif

Clinical History

75 year old lady with a nipple erosion, and possible ulceration with minimal bloody discharge.

Diagnosis

Papillary adenoma of the nipple

Final Diagnosis

BREAST, NIPPLE, BIOPSY:
A. PAPILLARY ADENOMA
B. NO INVASIVE CARCINOMA IS IDENTIFIED.
C. MARGINS ARE INVOLVED.

Result: 14.6...

Search bound: 0

(c) PathFind

Quadratic mean radius: 49.6641
Eccentricity: 15%

(d) FatFind

Priming count 5

☑ 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

(e) StrangeFind

Figure. 4: Screenshots of Example Applications

6.1.4 *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 4(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 [Goode et al. 2007; Kim et al. 2002] to locate the adipocytes in the image. 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.

6.1.5 *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 Section 2: assisting pharmaceutical researchers in automated cell microscopy, where very high volumes of cell imaging are typical. Figure 4(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. 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. Further details on this application can be found elsewhere [Goode et al. 2008].

6.1.6 *Discussion.* Through our extensive collaborations and from our first-hand experience in building the above applications, we have acquired a deeper 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 Section 5.2. 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 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. Because of the higher degree of scrutiny and trust that tends to exist within an enterprise, we expect that use of discard-based search is more likely within the intranets of enterprises rather than in mass-market use.

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.

Tunable 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 judgement 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. Interactive data exploration requires at least a modest rate of return of results even if they are not of the highest quality. The user cannot progress to the next iteration of a search session by re-parameterizing or redefining the current searchlet until she has sufficient clues as to what might be wrong with it. Consideration of false negatives may also be important: 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 a 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 whether the sampling of discarded objects should be uniform or biased towards the discard

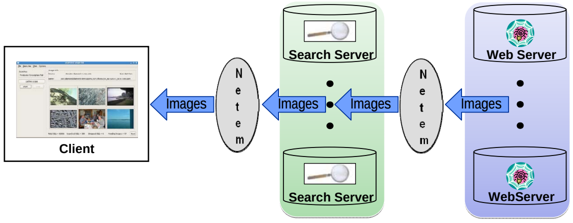


Figure. 5: Experimental Setup.

threshold (i.e., “near misses”).

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 [von Ahn et al. 2004].

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

6.2 Interactive Performance

While answering questions about the versatility of the system is relatively straightforward, answering questions about its performance is much harder. The heart of the complexity lies in a *confounding of system-centric and domain-centric effects.* Classical measures of system performance such as number of results per second fail to recognize the quality of those results. A bad searchlet may return many false positives, and overwhelm the user with junk; the worst case is a searchlet that discards nothing! The user impact of a bad searchlet is accentuated by good system infrastructure, since many more results are returned per unit time. Conversely, an excellent searchlet may return only a few results per unit of time because it produces very few false positives. However each result may be of such high quality that the user is forced to think carefully about it, before deciding whether it is a true positive or a false positive. The system is not impressive from a results per second viewpoint, but the user is happy because this rate is enough to keep her cognitively engaged almost continuously. Large think times are, of course, excellent from a systems perspective because they give ample processing time for servers to produce results.

Clearly, from the viewpoint of the user, it is very difficult to tease apart system-centric and domain-centric effects in our system. We therefore adopt a much less ambitious validation approach by fixing the user trace of interactive data exploration. This corresponds to a fixed sequence of searchlet refinements, each made after viewing the same number of results as in trace capture. Think time per result is also distributed as in the original trace. For such a trace, we show that the distribution of *user response time* (that is, unproductive user time awaiting the next result) is insensitive to network bandwidth and improves significantly with increased parallelism. We also show that response times are comparable on LAN-based server farms and live Web search. In other words, users can interactively search live Web data.

6.2.1 *Experimental Setup.* Figure 5 shows our experimental setup. All experiments involve a single client that replays a trace of search requests. These traces are described in Section 6.2.2. The experiments described in Sections 6.2.3 and 6.2.4 involve one to eight search servers that process data stored on their local disks. These experiments do not involve the Web servers shown in Figure 5. To emulate live Web search, the experiments in Section 6.2.5 use two Web servers. A *Netem* emulator varies network quality between client and search servers. Another *Netem* emulator varies network quality between search servers and Web servers. Except for the *Netem* machines, all computers, were 3GHz Intel Core^{TM} 2 Duo with 3 GB of RAM running Debian Linux 5.0. The Web servers ran Apache HTTP Server 2.2.9. The *Netem* machines were 3.6GHz Pentium 4 with 4GB of RAM.

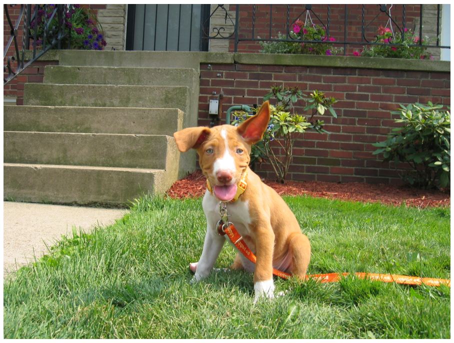


Figure. 6: Picture Shown to Users

User	Trace sequence	Images viewed	Elapsed time (s)
1	Browse to find photo containing green grass. Set a color filter and a texture filter matching the grass to find photo of dog playing in yard. Finally, set color filters matching the dog's brown and white patches.	166	598
2	Browse to find image of grass and image of brick wall. Set grass color and brick wall color and texture. Drop brick wall and go with grass color until image of dog is found. Set grass color and dog's colors filter. Finally, drop grass color and use only dog colors.	492	972
3	Browse to find image with grass. Used grass color filter to find image of brown deer. Used filters based on grass colors and deer color to find white hat and added white color to search. Found dog. Set color and texture filter based on dog. Found another image of the dog and created another brown color filter. Used first dog's white filter and new brown filter. Revert to dog's white color, fur texture and brown color filter. Finally revert to just dog white color and brown color filter.	289	1221

Table III: User Trace Characteristics

	Mean (s)	Median (s)	σ (s)
User 1	1.23	0.81	2.23
User 2	2.04	1.00	6.74
User 3	1.01	0.75	1.29
All Users	1.35	0.82	3.93

Table IV: User Think Times

6.2.2 *User Traces.* The traces used in our experiments were obtained from three members of our research team. Using the SnapFind client described in Section 6.1.1, their task was to find five pictures of the dog shown in Figure 6. The corpus of images they searched contained about 109,000 images that were downloaded from the Flickr [Flickr .] photo sharing Web site. We uniformly interspersed 67 images of the dog, taken over a year as it grew from puppy to adult, to the set of downloaded images. To allow us to accurately measure think times, we modified the SnapFind application to display one thumbnail at a time (rather than the usual six thumbnails). We obtained our traces at 100 Mbps, with four search servers. Table III summarizes the traces, while Table IV shows the mean and standard deviations of user think times. Our experiments replayed these traces on a SnapFind emulator.

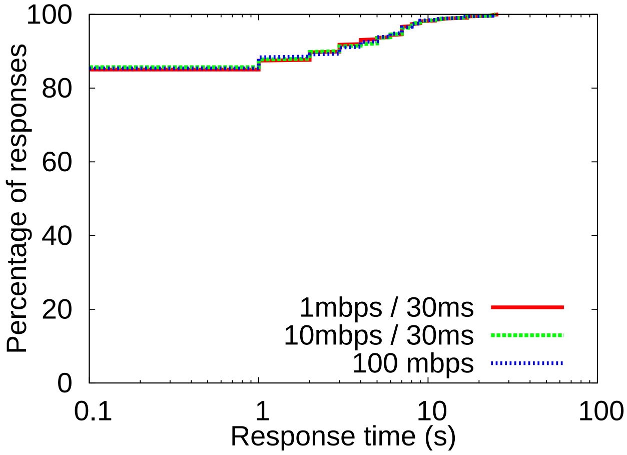


Figure. 7: Impact of Network Quality

We present our results in Figures 7 through 10 as cumulative distribution functions (CDFs) of measured response times over 3 runs of the User 1 trace. The User 2 and User 3 traces were only used to obtain think times. Note that the X axis in all these graphs is in log scale, to better illustrate the areas of most interest.

6.2.3 *Impact of Network Quality.* Figure 7 shows the CDFs of response times when we vary the network conditions between the client and 8 search servers from a well connected LAN environment to typical WAN links. Under the network conditions studied, the user experience provided by the system remains unaffected. Changes in bandwidth have little to no effect on the user experience because the system transfers minimal data during a search. The default behaviour of the system is to return low fidelity thumbnails of the image results instead of returning the full images. The user requests the full fidelity version of only those images he finds of interest. Robustness to network latency is achieved by buffering results as they arrive at the client. Often, requests for results are serviced from the buffer.

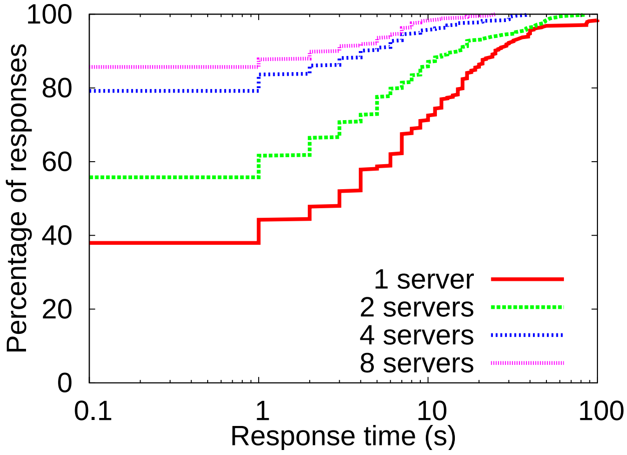


Figure. 8: Impact of Parallel Computation

6.2.4 *Impact of Parallel Computation.* Figure 8 shows the CDFs of response times as the number of search servers varies between one and eight. The results reported were obtained on a network consisting of the search servers and a client that connects to the servers through a simulated 1 Mbps WAN link with 30 ms RTT. Images are uniformly distributed across the search servers.

With a single search server, fewer than half of the user requests were serviced within one second. With four servers nearly 84% of the requests are serviced within one second. Eight servers increase the portion of requests serviced within a second to 87%. Thus, we conclude that design of the system enables users to easily exploit opportunities for parallel computation, and take advantage of additional processors without any application-specific support for parallelism.

6.2.5 *Searching the Live Web.* We evaluated the performance delivered by the system when conducting searches on content downloaded from the Web. For this experiment, the image collection is hosted on two remote Web servers accessible through a simulated 10 Mbps WAN link with RTT of 30 ms. The client connects to the search servers through a 100 Mbps LAN with no added latency. We varied the number of search servers from one to eight. Each search server ran five instances of the Web image retriever. We found that running five instances of this process strikes a balance between fully utilizing the bandwidth available and sharing the processor time with the search process.

Figure 9 compares the performance of Web search with the performance of a local search, where the data is stored on disks on the search servers. The figure shows that for a small number of servers, the task is CPU bound and the performance of searching web based images is comparable to that of a local search. However, with four or more search servers, the performance gap widens as the network link saturates. To conduct the search over the Web, the system downloaded, on average, 3.86 GB of image data. The magnitude of the download makes it impractical to perform such search interactively over the 1 Mbps link. Conversely, Figure 10, shows that increasing the link capacity to the Web servers improves the performance of Web search on large clusters to levels that are comparable to local search.

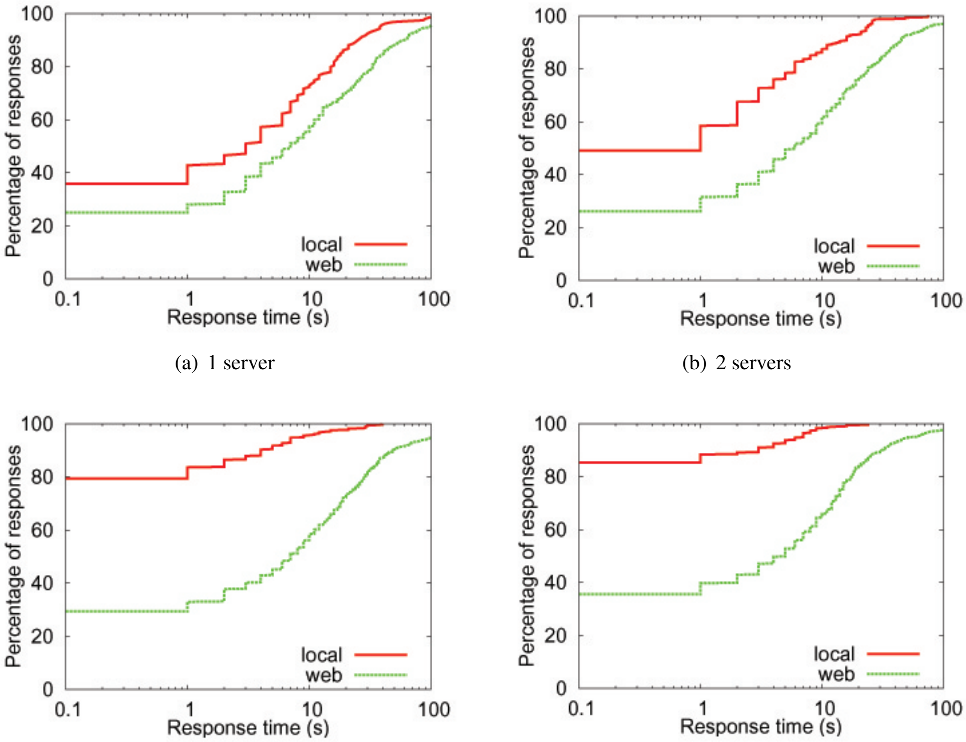
It is not surprising that response times are sensitive to the capacity of the link to the Web servers. Unlike the client, data retrievers download full resolution images since Web servers do not usually support negotiation of image fidelity. However, provided that there is sufficient bandwidth to keep all search servers busy, the performance of searching Web content is comparable to that of searching local content. We expect sites deploying clusters of the system to be well connected to the Internet, thus mitigating any network effect.

7. RELATED WORK

Diamond is the first system to unify the distinct concerns of interactive search and complex, non-indexed 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 filter execution site, separation of control and blast channels in the network protocol, and any-order semantics in server storage accesses. Although no other system addresses these dual concerns, some individual aspects of our design and implementation overlap previous work.

The dissemination and parallel execution of searchlet code at multiple servers bears some resemblance to the execution model of MapReduce [Dean et al. 2004] and Hadoop [Hadoop]. Both models address roughly the same problem, namely, going through a large corpus of data for identifying objects that match some search criteria. Our execution model was developed in 2002-2003 in the context of active disks, before we became aware of MapReduce, so it is interesting to note the similarity in our independent solutions. In both models, execution happens as close to data as possible. Of course, there are considerable differences at the next level of detail. 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 mechanisms for crisp user interaction that were mentioned in the previous paragraph have counterparts in the MapReduce model. Fault tolerance is important in MapReduce because it is intended for long-running batch executions; searchlet execution, in contrast, ignores failures since most executions are likely to be aborted by the user in at most a few minutes.

Aspects of filter execution in Diamond bear resemblance to the work of Abacus [Amiri et al. 2000],



(a) 1 server

(b) 2 servers

(c) 4 servers

(d) 8 servers

Figure. 9: Web at 1 Mbps with 1 to 8 Search Servers

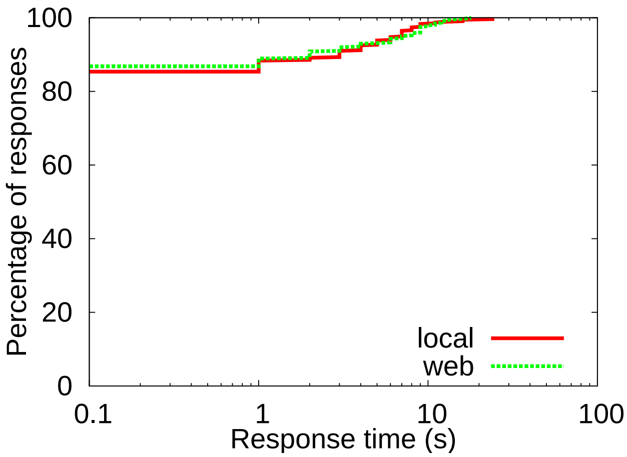


Figure. 10: Web at 100 Mbps with 8 Search Servers

Coign [Hunt et al. 2000], River [Amiri et al. 2000] and Eddies [Avnur et al. 2000]. 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. The importance of filter ordering has long been a topic of research in database query optimization [Selinger et al. 1979].

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 [Flickner et al. 1995]. More recently, low-level feature detectors and descriptors such as SIFT [Lowe 2004] and PCA-SIFT [Ke et al. 2004] have led to efficient schemes for index-based sub-image 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" [Minka et al. 1997] 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 [Berchtold et al. 1997; Duda et al. 2001; Yao et al. 1985]. 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.

8. CONCLUSION

Diamond aims to help domain experts creatively explore large bodies of complex non-indexed data. Through Diamond, 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. 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 pre-compute 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. That gap is Diamond's sweet spot, where it will offer the greatest value.

Acknowledgements

This research was supported by the National Science Foundation (NSF) under grant number CNS-0614679, and an IBM Open Collaborative Research Grant. Development of the MassFind and PathFind applications described in Section 6.1 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 Sections 6.1 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, IBM, Merck, Google, University of Toronto, or Carnegie Mellon University. OpenDiamond is a registered trademark of Carnegie Mellon University. Parts of this paper (totalling less than 25%) have some overlap with "The unique strengths and storage access characteristics of discard-based search" that was published under Open Access in the Journal of Internet Services and Applications, Volume 1, Number 1 in February 2010.

REFERENCES

Flickr. `http://www.flickr.com`.

Hadoop. `http://hadoop.apache.org/core/`.

SQLite. `http://www.sqlite.org/`.

ACHARYA, A., UYSAL, M., AND SALTZ, J. 1998 Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the International Conference on Architectural Support for Programming Langugages and Operating Systems* (1998).

AMIRI, K., PETROU, D., GANGER, G., AND GIBSON, G. Dynamic Function Placement for Data-Intensive Cluster Computing. In *Proceedings of the USENIX Technical Conference* (2000).

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).

AVNUR, R., AND HELLERSTEIN, J. Eddies: Continuously Adaptive Query Processing. In *Proceedings of SIGMOD* (2000).

BERCHTOLD, S., BOEHM, C., KEIM, D., AND 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).

DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (San Francisco, CA, 2004).

DUDA, R., HART, P., AND STORK, D. *Pattern Classification*. Wiley, 2001.

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

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).

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

HUNT, G., AND SCOTT, M. The Coign Automatic Distributed Partitioning System. In *Proceedings of OSDI* (1999).

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

HUSTON, L., SUKTHANKAR, R., WICKREMESINGHE, R., SATYANARAYANAN, M., GANGER, G.R., RIEDEL, E., AND 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).

KE, Y., SUKTHANKAR, R., AND HUSTON, L. Efficient near-duplicate and sub-image retrieval. In *Proceedings of ACM Multimedia* (2004).

KEETON, K., PATTERSON, D., AND HELLERSTEIN, J. A case for intelligent disks (IDISKs). *SIGMOD Record 27*, 3 (1998).

KIM, E., HASEYAMA, M., AND KITAJIMA, H. Fast and Robust Ellipse Extraction from Complicated Images. In *Proceedings of IEEE Information Technology and Applications* (2002).

LOWE, D. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal on Computer Vision* (2004).

MEMIK, G., KANDEMIR, M., AND CHOUDHARY, A. Design and Evaluation of Smart Disk Architecture for DSS Commercial Workloads. In *Proceedings of the International Conference on Parallel Processing* (2000).

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

RIEDEL, E., GIBSON, G., AND FALOUTSOS, C. Active Storage for Large-Scale Data Mining and Multimedia. In *Proceedings of the International Conference on Very Large Databases* (August 1998).

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).

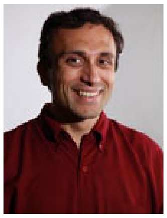
VON AHN, L., AND DABBISH, L. Labeling images with a computer game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vienna, Austria, April 2004).

YANG, L., JIN, R., MUMMERT, L., SUKTHANKAR, R., GOODE, A., ZHENG, B., HOI, S. C., AND SATYANARAYANAN, M. A Boosting Framework for Visuality-Preserving Distance Metric Learning and Its Application to Medical Image Retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence 32*, 1 (January 2010).

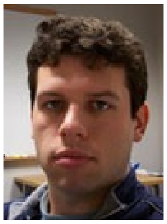
YAO, A., AND YAO, F. A General Approach to D-Dimensional Geometric Queries. In *Proceedings of the Annual ACM Symposium on Theory of Computing* (May 1985).

Prof. Mahadev Satyanarayanan (Satya) is an experimental computer scientist who has pioneered research in mobile and pervasive computing. One outcome is the open-source Coda File System, which supports distributed file access in low-bandwidth and intermittent wireless networks through disconnected and bandwidth-adaptive operation. Many Coda concepts can be found in the hotsync capability of PDAs today. Key ideas from Coda have been incorporated by Microsoft into the IntelliMirrorcomponent of Windows 2000 and the Cached Exchange Mode of Outlook 2003. Satya is a co-inventor of many supporting technologies relevant to mobile and pervasive computing, such as data staging, lookaside caching, translucent caching and application-aware adaptation. Satya is the Carnegie Group Professor of Computer Science at Carnegie Mellon University. From May 2001 to May 2004 he served as the founding director of Intel Research Pittsburgh, one of four university-affiliated research labs established worldwide by Intel to create disruptive information technologies through its Open Collaborative Research model. Satya received the PhD in Computer Science from Carnegie Mellon, after Bachelor's and Master's degrees from the Indian Institute of Technology, Madras. He is a Fellow of the ACM and the IEEE, and was the founding Editor-in-Chief of IEEE Pervasive Computing.

Rahul Sukthankar is a senior principal research scientist at Intel Labs Pittsburgh and adjunct research professor in the Robotics Institute at Carnegie Mellon. Rahul received his Ph.D. in Robotics from Carnegie Mellon and his B.S.E. in Computer Science from Princeton. His current research focuses on computer vision and machine learning, particularly in the areas of object recognition, video event detection and information retrieval.



This work was done while **Adam Goode** was a project scientist in the Diamond project at the School of Computer Science at Carnegie Mellon University. Adam received the bachelors degree in computer science and psychology from Rensselaer Polytechnic Institute and the masters degree from the Human-Computer Interaction Institute at Carnegie Mellon University. He currently works at Google Pittsburgh.



Nilton Bila is a Ph.D. candidate at the University of Toronto. His research interests include cloud computing, mobile systems and ubiquitous computing. Presently, his focus is in novel applications of cloud computing environments to reduce energy use of desktop computers. Nilton has a M.Sc. in Computer Science from the University of Toronto, Ontario, Canada.

Lily Mummert is a Research Scientist at Intel Labs Pittsburgh working in the area of distributed systems. Before joining Intel in 2006, she was a Research Staff Member at IBM TJ Watson Research Center, where she worked on problems in enterprise system management and contributed to several products. She earned a Ph.D. in Computer Science from Carnegie Mellon University in 1996. Her current research is focused on systems issues in high performance computational perception, enabling interactive applications that process data from heterogeneous, potentially high data rate sensors such as video and audio. She spends most of her time on the Everyday Sensing and Perception project. Previous projects include ISADS, for which she served as co-PI, and Diamond.

Jan Harkes is a senior project scientist at Carnegie Mellon University. His research interests include distributed systems, networking, and operating systems. Harkes has an MS in computer science from the Vrije University, Amsterdam. Contact him at jaharkes@cs.cmu.edu.

Adam Wolbach graduated from Carnegie Mellon University in 2008 with a Master's in Computer Science. He presently works in research and development at Epic in Verona, WI.

From 2002 to 2006, **Larry Huston** was a researcher at Intel Labs Pittsburgh where he was one of the co-founders of the Diamond project. Larry previously worked as a software architect for Intel's Network Processor Division, and at NetBoost and Ipsilon Networks. He received his Ph.D. in Computer Engineering from the University of Michigan in 1995. He also holds MSE and BSE degrees in Computer Engineering and Aerospace Engineering from the University of Michigan.

Eyal de Lara is an associate professor at the University of Toronto. His research interests lie in systems-level support for mobile and cloud computing. He has a PhD in electrical and computer engineering from Rice University. Contact him at delara@cs.toronto.edu.