USENIX Association

Proceedings of the Third USENIX Conference on File and Storage Technologies

San Francisco, CA, USA
March 31–April 2, 2004

Diamond: A Storage Architecture for Early Discard in Interactive Search

Larry Huston,[†] Rahul Sukthankar,[†•] Rajiv Wickremesinghe,[†‡] M. Satyanarayanan,[†•]
Gregory R. Ganger,[•] Erik Riedel,[⋆] Anastassia Ailamaki[•]
[†]*Intel Research Pittsburgh,* [•]*Carnegie Mellon University,* [‡]*Duke University,* [⋆]*Seagate Research*

Abstract

This paper explores the concept of *early discard* for interactive search of unindexed data. Processing data inside storage devices using downloaded *searchlet* code enables Diamond to perform efficient, application-specific filtering of large data collections. Early discard helps users who are looking for "needles in a haystack" by eliminating the bulk of the irrelevant items as early as possible. A searchlet consists of a set of application-generated filters that Diamond uses to determine whether an object may be of interest to the user. The system optimizes the evaluation order of the filters based on run-time measurements of each filter's selectivity and computational cost. Diamond can also dynamically partition computation between the storage devices and the host computer to adjust for changes in hardware and network conditions. Performance numbers show that Diamond dynamically adapts to a query and to run-time system state. An informal user study of an image retrieval application supports our belief that early discard significantly improves the quality of interactive searches.

1 Introduction

How does one find a few desired items in many terabytes or petabytes of complex and loosely-structured data such as digital photographs, video streams, CAT scans, architectural drawings, or USGS maps? If the data has already been indexed for the query being posed, the problem is easy. 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. Although attributes such as the author, date, or other context of data items can restrict the search space, the user is still left with an enormous number of items to examine. Today, scanning such a large volume of data is so slow that it is only performed in the context of well-planned data mining. This is typically a batch job that runs overnight and is only rarely attempted interactively [15].

Our goal is to enable interactive search of non-indexed data, where the user wishes to retrieve a small set of important items buried in a large collection. For instance, consider a surveillance scenario where an analyst is monitoring satellite imagery for interesting activity around oil tankers. Current image processing algorithms may be able to automatically discard images that do not contain oil tankers, but they cannot detect "interesting activity". Filtering the data allows the analyst to focus attention on the promising candidates by significantly reducing the number of irrelevant items. To make such systems practical, new techniques for scanning large volumes of data are needed. We believe that the solution lies in *early discard*, the ability to discard irrelevant data items as quickly and efficiently as possible (*e.g.*, at the storage device rather than close to the user). We have developed a storage architecture and programming model called *Diamond* that embodies early discard. Diamond has been designed to run on an active disk [1, 20, 25] platform, but does not depend on the availability of active storage devices. It can be realized using diverse storage back ends ranging from emulated active disks on a general-purpose cluster to storage nodes on a wide-area network.

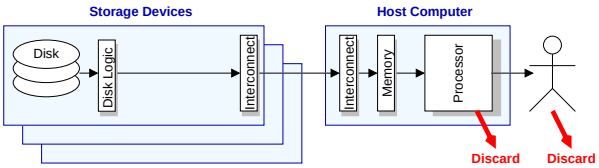
This paper focuses on *pure brute-force* interactive search (*i.e.*, where all of the data is processed for each query). Studying this extreme case enables us to determine the feasibility of early discard in a worst-case setting. Future Diamond implementations could incorporate performance optimizations such as caching results from previous queries and exploiting indices to reduce the search time.

This paper is organized as follows. Section 2 introduces early discard. Section 3 presents the Diamond architecture. Section 4 describes a proof-of-concept application and an informal user study. Section 5 discusses implementation details. Section 6 presents experimental results. Section 7 summarizes related work. Finally, Section 8 concludes the paper.

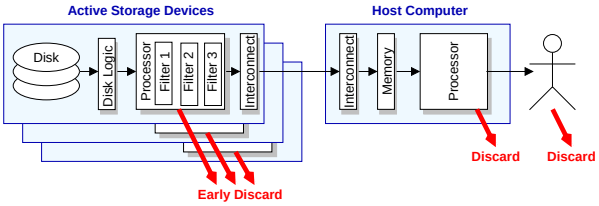
2 Background and Motivation

2.1 Limitations of Indexing

The standard approach to efficient interactive search is to create an offline index of the data. Indexing assumes that the mapping between the user's query and the relevant data can be pre-computed, enabling the system to efficiently organize data so that only a small fraction is accessed during a particular search. Unfortunately, indexing complex data remains a challenging problem for several reasons. First, manual indexing is often infeasible for large datasets and automated methods for extract-



(a) Traditional storage architecture



(b) Diamond architecture

Figure 1: **Early Discard** - Unlike traditional architectures for exhaustive search, where all of the data must be shipped from to the host computer, the Diamond architecture employs early discard to efficiently reject the bulk of the irrelevant data at the active storage device.

ing the semantic content from many data types are still rather primitive (the *semantic gap* [23]). Second, the richness of the data often requires a high-dimensional representation that is not amenable to efficient indexing (a consequence of the *curse of dimensionality* [6, 9, 33]). 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 the user's needs in a usable form can be extremely difficult (*e.g.*, "I need a photo of an energetic puppy playing with a happy toddler"). All of these problems limit the usability of *interactive data analysis* [15] today.

2.2 Importance of Early Discard

Figure 1(a) shows the traditional architecture for exhaustive search. Each data item passes through a pipeline from the disk surface, through the disk logic, over an interconnect to the host computer's memory. The search application can reject some of the data before presenting the rest to the user. Two problems with this design are: (1) the system is unable to take full advantage of the parallelism at the storage devices; (2) although the user is only interested in a small fraction of the data, all of it must be shipped from the storage devices to the host machine, and the bulk of the data is then discarded in the final stages of the pipeline. This is undesirable because the irrelevant data will often overload the interconnect or host processor.

Early discard is the idea of rejecting irrelevant data as early in the pipeline as possible. For instance, by exploiting active storage devices, one could eliminate a large fraction of the data before it was sent over the interconnect, as shown in Figure 1(b). Unfortunately, the storage device cannot determine the set of irrelevant objects *a priori* — the knowledge needed to recognize the useful data is only available to the search application (and the user). However, if one could imbue some of the earlier stages of the pipeline with a portion of the intelligence of the application (or the user), exhaustive search would become much more efficient. This is supported by our experiments, as described in Section 6.3.

For most real-world applications, the sophistication of user queries outpaces the development of algorithms that can understand complex, domain-dependent data. For instance, in a homeland security context, state-of-the-art algorithms can reliably discard images that do not contain human faces. However, face recognition software has not advanced to the point where it can reliably recognize photos of particular individuals. Thus, we believe that a large fraction of exhaustive search tasks will be interactive in nature. Unlike a typical web search, an interactive brute-force search through a large dataset could demand hours (rather than seconds) of focused attention from the user. For example, a biochemist might be willing to spend an afternoon in interactive exploration seeking a protein matching a new hypothesis. It is important for such applications to consider the human as the most important stage in the pipeline. Effective management of the user's limited bandwidth becomes crucial as the size and complexity of the data grows. Early discard enables the system to eliminate clearly useless data items early in the pipeline. The scarcest resource, *human attention*, can be directed at the most promising data items.

Ideally, early discard would reject all of the irrelevant data at the storage device without eliminating any of the desired data. This is impossible in practice for two reasons. First, the amount of computation available at the storage device may be insufficient to perform all of the necessary (potentially expensive) application-specific computations. Second, there is a fundamental trade-off [9] between false-positives (irrelevant data that is not rejected) and false-negatives (good data that is incorrectly discarded). Early discard algorithms can be tuned to favor one at the expense of the other, and different domain applications will make different trade-offs. For instance, an advertising agency searching a large collection of images may wish to quickly find a photo that matches a particular theme and may choose aggressive filtering; conversely, a homeland security analyst might wish to reduce the chance of accidentally losing a relevant object and would use more conservative filters (and accept the price of increased manual scanning). It is important to note that early discard does not, by itself, impact the accuracy of the search application: it simply makes applications that filter data more efficient.

2.3 Self-Tuning for Hardware Evolution

The idea of performing specialized computation close to the data is not a new concept. Database machines [7, 17] advocated the use of specialized processors for efficient data processing. Although these ideas had significant technical merit, they failed, at the time, because designing specialized processors that could keep pace with the sustained increase in general-purpose processor speed was commercially impractical.

More recently, the idea of an *active disk* [1, 20, 25], where a storage device is coupled with a general-purpose processor, has become popular. The flexibility provided by active disks is well-suited to early discard; an active disk platform could run filtering algorithms for a variety of search domains, and could support applications that dynamically adapt the balance of computation between storage and host as the location of the search bottleneck changes [2]. Over time, due to hardware upgrades, the balance of processing power between the host computer and storage system may shift. In general, a system should expect a heterogeneous composition of computational capabilities among the storage devices as newer devices may have more powerful processors or more memory. The more capable devices could execute more demanding early discard algorithms, and the partitioning of computation between the devices and the host computer should be managed automatically. Analogously, when the interconnect infrastructure or host computer is upgraded, one may expect computation to shift away from the storage devices. To be successful over the long term, the design needs to be *self-tuning*; manual re-tuning for each hardware change is impractical.

In practice, the best partitioning will depend on the characteristics of the processors, their load, the type and distribution of the data, and the query. For example, if the user were to search a collection of holiday pictures for snowboarding photos, these might be clustered together on a small fraction of devices, creating hotspots in the system.

Diamond provides two mechanisms to support these diverse storage system configurations. The first allows an application to generate specialized early discard code that matches each storage device's capabilities. The second enables the Diamond system to dynamically adapt the evaluation of early discard code, and is the focus of this paper. In particular, we explore two aspects of early discard: (1) adaptive partitioning of computation between the storage devices and the host computer based on run-time measurements; (2) dynamic ordering of search terms to minimize the total computation time.

2.4 Exploiting the Structure of Search

Diamond exploits several simplifications inherent to the search domain. First, search tasks only require read access to data, allowing Diamond to avoid locking complexities and to ignore some security issues. Second, search tasks typically permit stored objects to be examined in any order. This order-independence offers several benefits: easy parallelization of early discard within and across storage devices, significant flexibility in scheduling data reads, and simplified migration of computation between the active storage devices and host computer. Third, most search tasks do not require maintaining state between objects. This "stateless" property supports efficient parallelization of early discard, and simplifies the run-time migration of computation between active storage device and host computer.

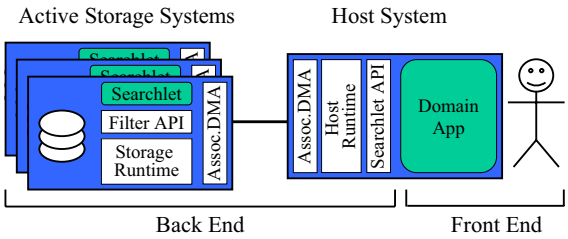


Figure 2: **Diamond Architecture**

3 Diamond Architecture

Figure 2 illustrates the Diamond storage architecture. Diamond provides a clear separation between the *front end*, which encapsulates domain-specific application code on the host computer, and the *back end*, which consists of a domain-independent infrastructure that is common across a wide range of search applications.

Diamond applications aim to reduce the load on the user by eliminating irrelevant data using domain-specific knowledge. Query formulation is domain-specific and is handled by the search application at the front end. Once a search has been formulated, the application translates the query into a set of machine executable tasks (termed a *searchlet*) for eliminating data, and passes these to the back end. The searchlet contains all of the domain-specific knowledge needed for early discard, and is a proxy of the application (and of the user) that can execute within the back end.

Searchlets are transmitted to the back end through the *searchlet API*, and distributed to the storage devices. At each storage device, the runtime system iterates through the objects on the disk (in a system-determined order) and evaluates the searchlet. The searchlet consists of a set of filters (see Section 5), each of which can independently discard objects. Objects that pass through all filters in a searchlet are deemed interesting, and made available to the domain application through the searchlet API.

The domain application may perform further processing on the interesting objects to see if they satisfy the

user's request. This additional processing can be more general than the processing performed at the searchlet level (which was constrained to the independent evaluation of a single object). For instance, the additional processing may include cross-object correlations and auxiliary databases. Once the domain application determines that a particular object matches the user's criteria, the object is shown to the user. When processing a large data set, it is important to present the user with results as soon as they appear. Based on these partial results, the user can refine the query and restart the search. Query refinement leads to the generation of a new searchlet, which is once again executed by the back end.

3.1 Searchlets

3.1.1 Searchlet Structure

The searchlet is an application-specific proxy that Diamond uses to implement early discard. It consists of a set of *filters* and some configuration state (*e.g.*, filter parameters and dependencies between filters). For example, a searchlet to retrieve portraits of people in dark business suits might contain two filters: a color histogram filter that finds dark regions and an image processing filter that locates human faces.

For each object, the runtime invokes each of the filters in an efficient order, considering both filter cost and selectivity (see Section 5.2). The return value from each filter indicates whether the object should be discarded, in which case the searchlet evaluation is terminated. If an object passes all of the filters in the searchlet, it is sent to the domain application.

Before invoking the first filter, the runtime makes a temporary copy of the object. This copy exists only until the object is discarded or the search terminates, allowing the filters to share state and computation without compromising the stored object.

One filter can pass state to another filter by adding attributes (implemented as name-value pairs) to the temporary object being searched. The second filter recovers this state by reading these attributes. If the second filter requires attributes written by the first filter, then the configuration must specify that the second filter depends on the first filter. The runtime ensures that filters are evaluated in an order that satisfies their dependencies.

The filter functions are sent as object code to Diamond. This choice of object code instead of alternatives, such as domain-specific languages, was driven by several factors. First, many real-world applications (*e.g.*, drug discovery) may contain proprietary algorithms where requiring source code is not an option. Second, we want to encourage developers to leverage existing libraries and applications to simplify the development process. For instance, our image retrieval application (described in Section 4) relies heavily on the OpenCV [8] image processing library.

Executing application-provided object code raises serious security and safety implications that are not specifically addressed by the current implementation. Existing techniques, such as processes, virtual machines, or software fault isolation [31], could be incorporated into future implementations. Additionally, Diamond never allows searchlets to modify the persistent (on-disk) data.

3.1.2 Creating Searchlets

Searchlets can be generated by a domain application in response to a user's query in a number of ways. The most straightforward method is for domain experts to implement a library of filter functions that are made available to the application. The user specifies a query by selecting the desired filters and setting their parameters (typically using a GUI). The application determines filter dependencies and assembles the selected functions and parameters into a searchlet. This works well for domains where the space of useful early discard algorithms is well spanned by a small set of functions (potentially augmented by a rich parameter space). These functions could also be provided (in binary form) by third-parties.

Alternately, the domain application could generate code on-the-fly in response to the user's query. One could envision an application that allows the user to manually generate searchlet code. We believe that the best method for searchlet creation is highly domain-dependent, and the best way for a human to specify a search is an open research question.

3.2 Key Interfaces

The Diamond architecture defines three APIs to isolate components: the searchlet API, the filter API and associative DMA. These are briefly summarized below.

- The *searchlet API* provides the interface that applications use to interact with Diamond. This API provides calls to query device capabilities, scope a search to a specific collection of data, load searchlets, and retrieve objects that match the search.
- The *filter API* defines the interface used by the filter functions to interact with the storage run-time environment. This API provides functions to read and write object contents, as well as functions to read and write object attributes to share state among filters. Any changes only affect the temporary version of the object.
- *Associative DMA* isolates the host and the storage implementations. This API abstracts the transport mechanism and flow control between host and storage run-time systems. Associative DMA also provides a common interface that enables Diamond to employ diverse back-end implementations without modifications to the host runtime.

3.3 Host and Storage Systems

The host system is where the domain application executes. The user interacts with this application to formulate searches and to view results. Diamond attempts to balance computation between the host and storage systems. To facilitate this, storage devices may pass *unprocessed* objects to the host runtime, due to resource limitations or other constraints. The host runtime evaluates the searchlet, if necessary; if the object is not discarded, it is made available to the domain application. The storage system provides a generic infrastructure for searchlet execution; all of the domain-specific knowledge is completely encapsulated in the searchlet. This enables the same Diamond back-end to serve different domain applications (simultaneously, if necessary).

Diamond is well-suited for deployment on active storage, but such devices are not commercially available today, nor are they likely to become popular without compelling applications. Diamond provides a programming model that abstracts the storage system, enabling the development of applications that will seamlessly integrate with active storage devices as they become available.

Diamond's current design assumes that the storage system can be treated as object storage [30]. This allows the host to be independent of the data layout on the storage device and should allow us to leverage the emerging object storage industry standards.

4 Diamond Applications

Diamond provides a general framework for building interactive search applications. All of the domain-specific knowledge is contained in the front-end application and in the searchlets. To illustrate the process, consider the problem of drug candidate design.

Given a target protein, a chemist must search through a large database of 3D ligand structures to identify candidates that may associate strongly with the target. Since accurate calculation of the binding free energy of a particular ligand is prohibitively expensive, such programs could benefit from user input to guide the search in two ways. First, the chemist could adjust the granularity of the search (trading accuracy for speed). Second, the chemist could test hypotheses about a particular ligand-protein interaction using interactive molecular dynamics [14]. In Diamond, the former part of the search could be downloaded to the storage device while the latter could be performed on the chemist's graphical workstation. Early discard would reject hopeless ligands from consideration allowing the chemist to focus attention on the more promising candidates. If none of the initial candidates proved successful, the chemist would refine the search to be less selective. This example illustrates some of the characteristics that make an application suitable for early discard. First, that the user is searching for *specific instances* of data that match a query rather than aggregate statistics about the set of matching data items. Second, that the user's criteria for a successful match is often subjective, potentially ill-defined, and typically influenced by the partial results of the query. Third, that the mapping between the user's needs and the matching objects is too complex for it to be captured by a batch operation. An everyday example of such a domain is image search; the remainder of this section presents SnapFind, a prototype application for this domain built using the Diamond programming model.

4.1 SnapFind Description

SnapFind was motivated by the observation that digital cameras allow users to generate thousands of photos, yet few users have the patience to manually index them. Users typically wish to locate photos by semantic content (*e.g.*, "show me photos from our whale watching trip in Hawaii"); unfortunately, computer vision algorithms are currently incapable of understanding image content to this degree of sophistication. SnapFind's goal is to enable users to interactively search through large collections of unlabeled photographs by quickly specifying searchlets that roughly correspond to semantic content.

Research in image retrieval has attracted considerable attention in recent years [11,29]. However, prior work in this area has largely focused on whole-image searches. In these systems, images are typically processed off-line and compactly represented as a multi-dimensional vector. In other systems, images are indexed offline into several semantic categories. These enable users to perform interactive queries in a computationally-efficient manner; however, they do not permit queries about local regions *within* an image since indexing every subregion within an image would be prohibitively expensive. Thus, whole-image searches are well-suited to queries corresponding to general image content (*e.g.*, "find me an image of a sunset") but poor at recognizing objects that only occupy a portion of the image (*e.g.*, "find me images of people wearing jeans"). SnapFind exploits Diamond's ability to exhaustively process a data set using customized filters, enabling users to search for images that contain the desired content only in a small patch. The remainder of this section describes SnapFind and presents an informal validation of early discard.

SnapFind allows users to create complex image queries by combining simple filters that scan images for patches containing particular color distributions, shapes, or visual textures (detailed in a technical report [19]). The user can either select a pre-defined filter from a palette (*e.g.*, "frontal human faces" or "ocean waves") or create new filters by clicking on sample patches in other images (*e.g.*, creating a "blue jeans" color filter

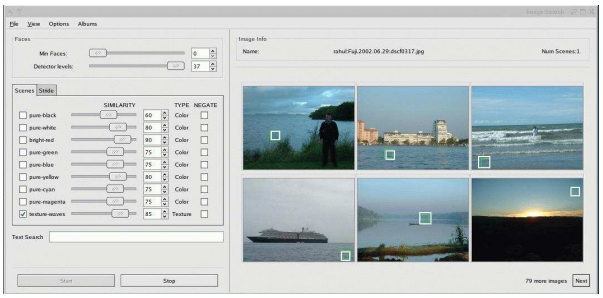


Figure 3: **SnapFind** – a proof-of-concept image search application designed using the Diamond programming model. Users can search a large image collection using customized filters. Here, the user has filtered for regions containing water texture (marked with white boxes). The filter correctly identifies most of the water in the images, but occasionally makes errors; for example, the sky in the bottom right image is incorrectly labeled as water.

by giving half a dozen examples). Indexing is infeasible for two reasons: (1) the user may define new search filters at query time; (2) the content of the patches is typically high-dimensional. When the user submits the query, SnapFind generates a searchlet and initiates a Diamond search. Diamond typically executes a portion of the query at the storage device, enabling early discard to reject many images in the initial stages of the pipeline.

4.2 SnapFind Usage Experience

We designed some simple experiments to investigate whether early discard can help exhaustive search. Our chosen task was to retrieve photos from an unlabeled collection based on semantic content. This is a realistic problem for owners of digital cameras and is also one that untrained users can perform manually (given sufficient patience). We explored two cases: (1) purely manual search, where all of the discard happens at the user stage; (2) using SnapFind. Both scenarios used the same graphical interface (see Figure 3), where the user could examine six thumbnails per page, magnify a particular image (if desired) and mark selected images.

Our data set contained 18,286 photographs (approximately 10,000 personal pictures, 1,000 photos from a corporate website, 5,000 images collected from an ethnographic survey and 2,000 from the Corel image CD-ROMs). These were randomly distributed over twelve emulated active storage devices. Users were given three minutes to tackle each of the following two queries: (S1) find images containing windsurfers or sailboats; (S2) find pictures of people wearing dark business suits or tuxedos.

For the manual scenario, we recorded the number of images selected by the user (correct hits matching the query) along with the number of images that the user viewed in the alloted time. Users could page through the

	MANUAL		DIAMOND			
	hits	user seen	hits	user seen	system seen	early discard
S1						
a	7	684	46	396	18,286	97.8%
b	8	774	39	396	18,286	97.8%
c	13	966	46	396	18,286	97.8%
S2						
a	29	600	29	78	15,286	99.5%
b	18	612	29	74	15,362	99.5%
c	24	630	29	74	15,198	99.5%

Table 1: **SnapFind user study** - Results of an informal interactive search experiment using SnapFind. Users (a,b,c) were given three minutes to locate photographs matching each query (S1 and S2) in a collection of 18,286 images.

images at their own pace, and Table 1 shows that users scanned the images rapidly, viewing 600–1,000 images in three minutes. Even at this rate of 2–5 images per second, they were only able to process about 5% of the total data.

For the SnapFind scenario, the user constructed early discard searchlets simply by selecting one or more image processing filters from a palette of pre-defined filters, configured filter parameters using the GUI, and combined them using boolean operators. Images that satisfied the filtering criteria (*i.e.*, those matching a particular color, visual texture or shape distribution in a subregion) were shown to the user. Based on partial results, the user could generate a new searchlet by selecting different filters or adjusting parameters. As in the manual scenario, the user then marked those images that matched the query. For S1, the early discard searchlet was a single "water texture" filter trained on eight 32×32 patches containing water. For S2, the searchlet was a conjunction of a color histogram filter combined with a face detector. Table 1 shows these results, and searchlets are detailed in Table 2.

For S1, SnapFind significantly increases the number of relevant images found by the user. Diamond is able to exhaustively search through all of the data, and early discard eliminates almost 98% of the objects at the storage devices. This shows how early discard can help users find a greater number of relevant objects.

For S2, the improvement, as measured by hits alone, is less dramatic, but early discard shows a different benefit. Although Diamond fails to complete the exhaustive search in three minutes (it processes about 85% of the data), the user achieves approximately as many hits as in the manual scenario while viewing 88% fewer images. For applications where the user only needs a few images, early discard is ideal because it significantly decreases the user's effort. By displaying fewer irrelevant

items, the user can devote more attention to the promising images.

5 Prototype Implementation

Our Diamond prototype is currently implemented as user processes running on Red Hat Linux 9.0. The searchlet API and the host runtime are implemented as a library that is linked against the domain application. The host runtime and network communication are threads within this library. We emulate active storage devices using off-the-shelf server hardware with locally-attached disks. The active storage system is implemented as a daemon. When a new search is started, new threads are created for the storage runtime and to handle network and disk I/O. Diamond's object store is implemented as a library that stores objects as files in the native file system. Associative DMA is currently under definition; Diamond uses a wrapper library built on TCP/IP with customized marshalling functions to minimize data copies.

The remainder of this section details Diamond's two primary mechanisms for efficient early discard: run-time partitioning of computation between the host and storage devices, and dynamic ordering of filter evaluation to reject undesirable data items as efficiently as possible.

5.1 Dynamic Partitioning of Computation

As discussed in Section 2, bottlenecks in exhaustive search pipelines are not static. Diamond achieves significant performance benefits by dynamically balancing the computational task between the active storage devices and the host processor.

The Diamond storage runtime decides whether to evaluate a searchlet locally or at the host computer. This decision can be different for each object, allowing the system to have fine-grained control over the partitioning. Thus, even for searchlets that consist of a single monolithic filter, Diamond can partition the computation on a per-object basis to achieve the ratio of processing between the storage devices and the host that provides the best performance. The ability to make these fine-grained decisions is enabled by Diamond's assumption that objects can be processed in any order, and that filters are stateless.

If the searchlet consists of multiple filters, Diamond could partition the work so that some filters execute on the storage devices and the remainder execute on the host; the current implementation does not consider such partitionings. Diamond could also detect when there are many objects waiting for user attention and choose to evaluate additional filters to discard more objects.

The current implementation supports two methods for partitioning computation between the host and the storage devices. The effectiveness of these methods in practice is evaluated in Section 6.3.

5.1.1 CPU Splitting

In this method, the host periodically estimates its available compute resources (processor cycles), determines how to allocate them among the storage devices, and sends a message to each device. The storage device receives this message, estimates its own computational resources, and determines the percentage of objects to process locally. For example, if a storage device estimates that it has 100 MIPS and receives a slice of 50 MIPS from the host, then it should choose to process 2/3 of the objects locally and send the remaining (unprocessed) objects to the host. CPU splitting has a straightforward implementation: whenever the storage runtime reads an object, it probabilistically decides whether to process the object locally.

5.1.2 Queue Back-Pressure

Queue Back-Pressure (QBP) exploits the observation that the length of queues between components in the search pipeline (see Figure 1) provide valuable information about overloaded resources. The algorithm is implemented as follows.

When objects are sent to the host, they are placed into a work queue that is serviced by the host runtime. If the queue length exceeds a particular threshold, the host refuses to accept new objects. Whenever the storage runtime has an object to process, it examines the length of its transmit queue. If the queue length is less than a threshold, the object is sent to the host without processing. If the queue length is above the threshold, the storage runtime evaluates the searchlet on the object. This algorithm dynamically adapts the computation performed at the storage devices based on the current availability of downstream resources. When the host processor or network is a bottleneck, the storage device performs additional processing on the data, easing the pressure on downstream resources until data resumes its flow. Conversely, if the downstream queues begin to empty, the storage runtime will aggressively push data into the pipeline to prevent the host from becoming idle.

5.2 Filter ordering

A Diamond searchlet consists of a set of filters, each of which can choose to discard a given object. We assume that the set of objects that pass through a particular searchlet is completely determined by the set of filters in the searchlet (and their parameters). However, the filter order dramatically impacts the efficiency with which Diamond processes a large amount of data.

Diamond attempts to reorder the filters within a searchlet to run the most promising ones early. Note that the best filter ordering depends on the set of filters, the user's query, and the particular data set. For example, consider a user who is searching a large image collection for photos of people in dark suits. The application may

determine that a suitable searchlet for this tasks includes two filters (see Table 3): a face detector that matches images containing human faces (filter F1); and a color filter that matches dark regions in the image (filter F3). From the table, it is clear that F1 is more selective than F3, but also much more computationally expensive. Running F1 first would work well if the data set contained a large number of night-time photos (which would successfully pass F3). On the other hand, if the collection contained a large number of baby pictures, running F1 early would be extremely inefficient.

The effectiveness of a filter depends upon its selectivity (pass rate) and its resource requirements. The total cost of evaluating filters over an object can be expressed analytically as follows. Given a filter, Fi, let us denote the cost of evaluating the filter as $c(Fi)$, and its pass rate as $\mathrm{P}(Fi)$. In general, the pass rates for the different filters may be correlated (*e.g.*, if an image contains a patch with water texture, then it is also more likely to pass through a blue color filter). We denote the *conditional pass rate* for a filter Fi that is processing objects that successfully passed filters Fa, Fb, Fc by $\mathrm{P}(Fi|Fa,Fb,Fc)$. The average time to process an object through a series of filters $F0 \ldots Fn$ is given by the following formula:

$$C = c(F0) + \mathrm{P}(F0)c(F1) + \mathrm{P}(F1|F0)\mathrm{P}(F0)c(F2) + \mathrm{P}(F2|F1,F0)\mathrm{P}(F1|F0)\mathrm{P}(F0)c(F3) + \cdots$$

The primary goal of choosing a filter order is to minimize this cost function. To perform this optimization, the system needs the costs of the different filters and the conditional pass rates. Diamond estimates these values during a searchlet execution by varying the order the filters are evaluated and measuring the pass rates and costs over a number of objects.

5.2.1 Partial Orderings

Allowing filters to use results generated by other filters enables searchlets to: (1) use generic components to compute well-known properties; (2) reuse the results of other filters. For instance, all of the color filters in SnapFind (see Section 4) rely on a common data structure that is generated by an auxiliary filter. Filter developers can explicitly specify the attributes that each filter requires, and these dependencies are expressed as *partial ordering* constraints. Figure 4 shows an example of a partial order. The forward arrows indicate an 'allows' relationship. For example, "Reader" is a prerequisite for "Histogram" and "WaterTexture", and "Red" and "Black" are prerequisites for "ColorTest". The filter ordering problem is to find a *linear extension* of the partial order. Figure 5 shows one possible order. Note that finding a path through this directed acyclic graph is not sufficient; all of the filters in the searchlet still need to be evaluated.

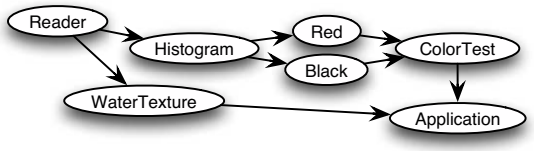


Figure 4: **Example partial ordering** - "Reader" must be executed before "Histogram" and "WaterTexture". "Histogram" must be evaluated before "Red" and "Black".

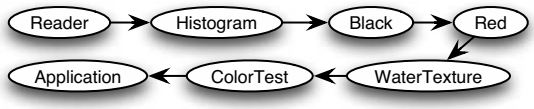


Figure 5: **Linear Extension** - a possible ordering for the filters shown in Figure 4.

5.2.2 Ordering Policies

The *filter ordering policy* is the method that Diamond uses for choosing the sequence for evaluating the filters. We describe three policies below.

- **Independent:** assumes that there is no correlation between $\mathrm{P}(Fi)$ and $\mathrm{P}(Fj)$, or $c(Fi)$ and $c(Fj)$. Using this assumption we can find an optimal sequence by sorting on $c(Fi)/\mathrm{P}(Fi)$ [24]. In practice the correlations between filters may cause this policy to perform poorly.
- **Hill climbing (HC):** picks a random sequence from the space of all legal linear extensions. The policy attempts to iteratively improve the order by swapping adjacent filters until a local minimum is reached. Multiple random restarts are used to reduce sensitivity to the starting point.
- **Best filter first (BFF):** iteratively expands a list of valid sub-sequences to find the optimal filter sequence. BFF initializes a list with the set of single-element sub-sequences consisting of the filters that have no dependencies. BFF then removes the cheapest sub-sequence from the list, computes all valid sub-sequences that are one filter longer, and reinserts them into the list. BFF terminates when the cheapest sub-sequence is complete; this is an optimal sequence. The algorithm is motivated by the observation that later filters typically have less impact on the average cost than earlier filters, because the overall pass probability drops as one goes deeper in the filter chain.

6 Experimental Evaluation

This section presents empirical results from a variety of experiments using SnapFind running on the Diamond implementation described in Section 5. The active storage devices were emulated using rack-mounted computers (1.2 GHz Intel® Pentium® III processors, 512 MB RAM and 73 GB SCSI disks), connected via a 1 Gbps Ethernet switch. The host system contained a 3.06 GHz Intel® Pentium® Xeon™ processor, 2 GB RAM, and a 120 GB IDE disk. The host was connected via Ethernet to the storage platforms. We varied the link speed between 1 Gbps and 100 Mbps depending on the experiment. Some experiments required us to emulate slower active storage devices; this was done by running a real-time task that consumed a fixed percentage of the CPU. These experiments employ homogeneous backends.

6.1 Description of Searchlets

We evaluate Diamond using the set of queries enumerated in Table 2. These consist of real queries from SnapFind searches supplemented by several synthetic queries. The searchlets are described in Table 2, and the filters used by these searchlets are listed in Table 3.

The Water (S1) and Business Suits (S2) queries match the tasks we used in Section 4. The Halloween (S3) query is similar to Business Suits with an additional filter. The three synthetic queries (S4–S6) are used to test filter ordering and the two Dark Patch queries (S7, S8) are used to illustrate bottlenecks for dynamic partitioning.

Table 3 provides a set of measurements summarizing the discard rate and the computational cost of running the various filters. We determined these filter characteristics by evaluating each filter over the objects in our image collection (described in Section 4). The overall discard rate is the fraction of objects dropped divided by the total number of images, and the cost is the average number of CPU milliseconds consumed. Filters F0–F5 are taken from SnapFind. The other filters were synthetically generated with specific characteristics.

The searchlets S5 and S6 were designed to examine the effect of filter correlation. F14, F15 and F16 are correlated: $\mathrm{P}(F14, F15, F16) \neq \mathrm{P}(F14)\mathrm{P}(F15)\mathrm{P}(F16)$. F17, F18 and F19 are uncorrelated: $\mathrm{P}(F17, F18, F19) = \mathrm{P}(F17)\mathrm{P}(F18)\mathrm{P}(F19)$.

6.2 Disk and Host Processing Power

Our first measurements examine how variations in system characteristics (number of storage devices, interconnect bandwidth, processor performance, queries) affect the average time needed to process each object. For each configuration, we measure the completion time for a different static partitioning between the host and storage devices. A particular partition is identified by percentage of objects that are evaluated at the storage devices. Remaining objects are passed to the host for processing.

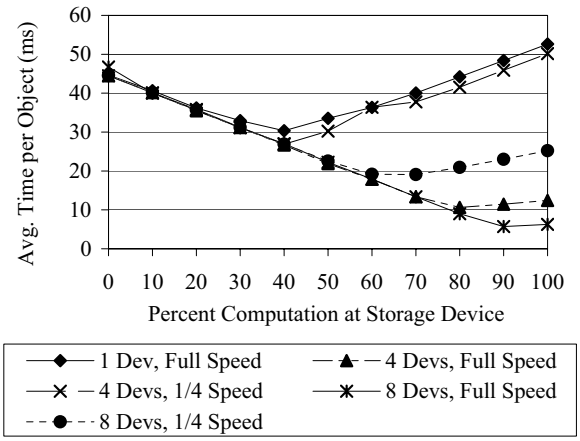


Figure 6: **Compute Limited** - This graph shows how the average time spent processing an object varies with the percentage of the objects evaluated at the storage system when the CPU is the bottleneck. The average time is computed as the total search time divided by the number of objects searched.

In these experiments, each storage device has 5,000 objects (1.6 GB). As the number of storage devices increases, so does the total number of objects involved in a search. For each configuration, we report the mean time needed for Diamond to process each object (averaged over three runs). Our data set was chosen to be large enough to avoid startup transients but small enough to enable many different experiments. Using a larger data set would give the same average time per object, but will increase the overall completion time for a search.

The first set of experiments (see Figure 6) shows how variations in the relative processing power of the host and storage devices affect search time for CPU-bound tasks. These experiments use searchlet S3 to find pictures of a child in a Halloween costume.

We observe that, as the number of storage devices increases, more computation is moved to the storage devices. This matches our intuition that as the aggregate processing power of the storage devices increases, more of the overall processing should be done at the storage devices.

When there is no processing at the storage devices, this is equivalent to reading all of the data from network storage. On the left-hand side of the lines, we see linear decreases as processing is moved to the storage devices, reducing the load on the bottleneck. When most of the processing moves to the storage device, the bottleneck becomes the storage device and we see increases in average processing time. The best case is the local minimum; this corresponds to the case where the load be-

Query		Searchlet Description	CPU Cost
Water - regions containing water waves	S1	Uses texture filter trained on water samples.	Low
Business Suits - images of people in dark business suits	S2	Uses face detector and color histogram trained on dark patches of color.	High
Halloween - images of a child in Halloween costume	S3	Uses face detector and color histograms trained on red patches and dark patches of color.	High
Synthetic	S4	Synthetic filters with inversely (non-linearly) related pass rate and cost.	Med
Synthetic	S5	Three filters with correlated pass rate and constant cost.	Low
Synthetic	S6	Three filters with independent pass rate (same as S5 overall) and constant cost.	Low
Dark Patch A - searchlet with high selectivity	S7	Uses color histogram trained on black sample patch; has a high threshold so few images match.	Low
Dark Patch B - searchlet with low selectivity	S8	Uses color histogram trained on black sample patch; has a low threshold so many images match.	Low

Table 2: **Test Queries** - The queries and associated searchlets used to evaluate the Diamond prototype.

Filter	Searchlet	Discard rate	CPU (ms)
F0 - Reader (required)	S1,2,3,7,8	0	5
F1 - Face Detect	S2,3	99%	530
F2 - Histogram	S2,3,7,8	0	20
F3 - Black (req. F2)	S2	83%	2
F3a - Black (req. F2)	S7	99%	2
F3b - Black (req. F2)	S8	78%	2
F4 - Red (req. F2)	S2	99%	2
F5 - Wave Texture	S1	95%	14
F6 - Synthetic		20%	2
F7 - Synthetic		22%	4
F8 - Synthetic		26%	8
F9 - Synthetic	S4	29%	16
F10 - Synthetic		31%	32
F11 - Synthetic		33%	64
F12 - Synthetic		36%	128
F13 - Synthetic		36%	256
F14		50%	1
F15 - Synthetic	S5	40%	8
F16		30%	8
F17		50%	1
F18 - Synthetic	S6	40%	8
F19		30%	8

Table 3: **Filters** - The discard rate is over a collection of 18,286 images. F0 and F2 are helper filters that do not discard data. The filters in S5 are correlated; those in S6 are uncorrelated.

tween the host and the storage devices is balanced. Note that Diamond benefits from active storage even with a small number of storage devices.

Our next measurements (see Figure 7) examine the network-bound case using searchlets S7 and S8. Both searchlets look for a small dark region and are relatively cheap to compute. S7 rejects most of the objects (highly selective) while S8 passes a larger fraction of the objects (non-selective).

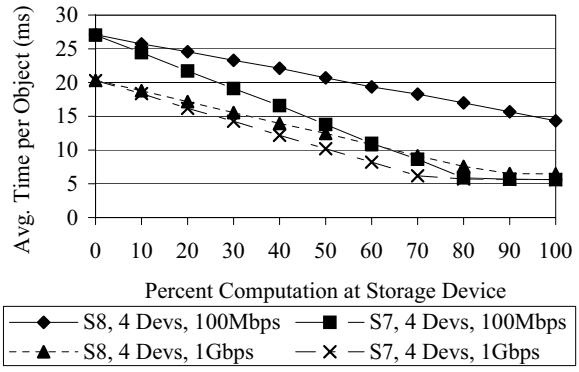


Figure 7: **Network Limited** - This graph shows how the average time per object varies with the percentage of objects evaluated at the storage system when the network is the bottleneck. The average time is computed as the total search time divided by the number of objects searched.

These experiments demonstrate that as the network becomes the limiting factor, more computation should be performed at the storage device. We also see that these lines flatten out at the point where reading the data from the disk becomes a bottleneck. The upper two lines show S7 and S8 running on a 100 Mbps network. We see that S8 is always slower, even when all of the computation is performed at the storage device. This is because S8 passes a large percentage of the objects, creating a data transfer bottleneck in all cases.

6.3 Impact of Dynamic Partitioning

This section evaluates the effectiveness of the dynamic partitioning algorithms presented in Section 5. As a baseline measurement, we manually find the ideal partitioning based on the results from the previous section. We then compare the time needed to complete

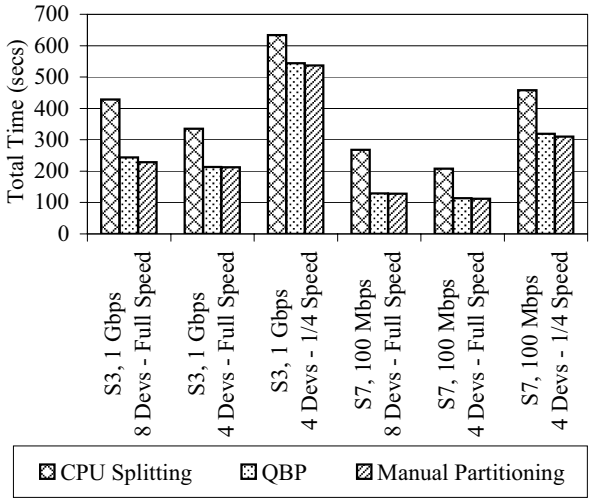


Figure 8: **Dynamic Partitioning** - This graph compares the performance of two automated partitioning algorithms against a hand-tuned manual partitioning.

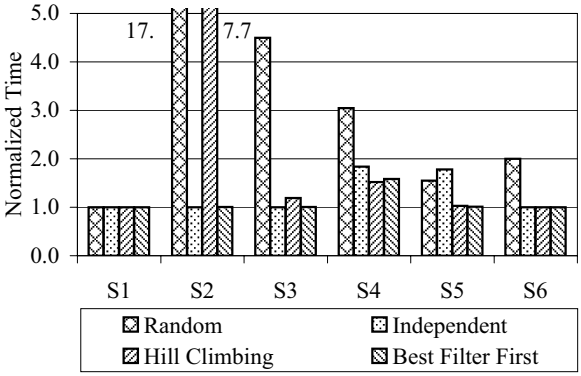


Figure 9: **Filter Ordering** - Execution time for evaluating searchlets using different ordering policies, normalized to the Offline Best policy.

the search using this manual partitioning to those for the two dynamically-adjusting schemes: *CPU Splitting* and *Queue Back-Pressure* (QBP).

For these tests, we use both a CPU-bound task (searchlet S3) and a network-bound task (searchlet S7). We run each task in a variety of configurations and compare the results as shown in Figure 8.

In all of these cases, the QBP technique gives similar performance to the Best Manual technique. CPU Splitting does not perform as well in most of the cases for two reasons. First, in the network-bound task (searchlet S7), the best results are obtained by processing all objects at the storage devices. CPU Splitting always tries to process a fraction of the objects on the host, even when sending data to the host is the bottleneck. QBP detects the network bottleneck and processes the objects locally. Second, relative CPU speeds are a poor estimate of the time needed to evaluate the filters. Most of these searchlets involve striding over large data structures (images) so the computation tends to be bound by memory access, not CPU. As a result, increasing the CPU clock rate does not give a proportional decrease in time. It is possible that more sophisticated modeling would make CPU Splitting more effective. However, given that the simple QBP technique works so well, there is probably little benefit to pursuing that idea.

6.4 Impact of Filter Ordering

This section compares the different policies described in Section 5.2.2, and illustrates the significance of filter ordering. We use searchlets S1–S6, which are composed of the filters detailed in Table 3. This experiment eliminates network and host effects by executing entirely on a single storage device and compares different local optimizations. Total time is normalized to the *Offline Best* policy; this is the best possible static ordering (computed using an oracle), and provides a bound on the minimum time needed to process a particular searchlet. *Random* picks a random legal linear order at regular intervals. This is the simplest solution that avoids adversarial worst cases without extra state, and would be a good solution if filter ordering did not matter.

Figure 9 shows that completion time varies significantly with different filter ordering policies. The poor performance of *Random* demonstrates that filter ordering is significant. There is a unique legal order for S1, and all methods pick it correctly. *Independent* finds the optimal ordering when filters are independent, as in S6, but can generate expensive orderings when they are not, as in S5. *Hill Climbing* sometimes performs poorly because it can get trapped in local minima. *Best Filter First* is a dynamic techniques that works as well as *Independent* (it has a slightly longer convergence time) with independent filters, and has good performance with dependent filters. The dynamic techniques spend time exploring the search space, so they always pay a penalty over the *Offline Best* policy. This is more pronounced with more filters, as in S4.

The next experiment examines Diamond performance when dynamic partitioning and filter ordering are run concurrently. For our baseline measurement, we manually find the best partitioning and the best filter ordering for each configuration. We then compare the time needed to execute searchlet S3 against two test cases that use dynamic adaptation. The first uses dynamic partitioning (QBP) and the filter ordering (BFF); the second uses dynamic partitioning (QBP) and randomized filter orders. Figure 10 shows the results of these experiments. As expected, the combination of dynamic partitioning and dynamic filter ordering gives us results that are close to the best manual partitioning. Random filter ordering performs less well because of the longer time needed to

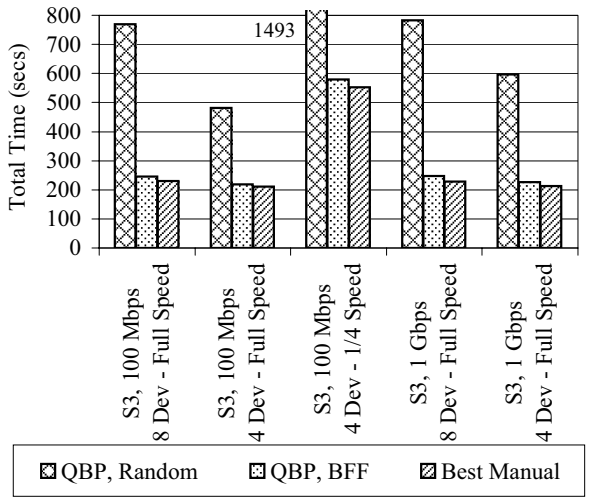


Figure 10: **Dynamic Optimizations** - Execution times for evaluating searchlets using a combination of dynamic partitioning and filter ordering, compared against a hand-tuned algorithm.

process each object.

6.5 Using Diamond on Large Datasets

To better understand the impact of Diamond on real-world problems, we consider a typical scenario: how much data could a user search in an afternoon? The results from Figure 10 show that Diamond can process 40,000 objects (8 storage devices with 5,000 objects each) in 247 seconds. Thus, given four hours, the user should be able to search through 2.3 million objects (approximately 0.75 TB) using the same searchlets. In the case of searchlet S3, this would imply that the user should see about 115 objects. However, since the number of objects seen by the user is sensitive to search parameters and the distribution of data on the storage device, it could vary greatly from this estimate.

Although raw performance should scale as disks are added, the limitations imposed by the user and the domain application are less clear. For instance, in the drug discovery application described in Section 4, the user's think-time may be the limiting factor even when Diamond discards most of the data. Conversely, in other domains, the average computational cost of a searchlet could be so high that Diamond would be unable to process all of the data in the alloted time. These questions are highly domain-dependent and lie beyond the scope of this paper.

As we discussed in the introduction, the current implementation is focused on pure brute-force search, but other complimentary techniques can be used to improve performance. One technique would be to use pre-computed indices to reduce the number of objects searched. For example, filter F1 from Table 3 could be used to build an index of pictures containing faces. Using this index would reduce the search space by 99% for any searchlets that use filter F2.

Another complimentary technique is to take advantage of cached results. In certain domains, a user may frequently refine a searchlet based on partial results in a manner that leaves most of its filters and their parameters unchanged. For instance, in SnapFind, the user may modify a search by adding a filter to the existing set of filters in the searchlet. When re-executing a filter with the same parameters, Diamond could gain significant computational benefits by retrieving cached results. However, caching may provide very little benefit for other applications. For instance, a Diamond application that employs relevance feedback [16] typically adjusts filter arguments at each iteration based on user-provided feedback. Since the filter arguments are different with each search, the use of cached information becomes more difficult. We plan to evaluate the benefits of caching as we gain more experience with other Diamond applications.

7 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* [15] 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.

In more traditional database research, advanced indexing techniques exist for a wide variety of specific data types including multimedia data [10]. Work on data cubes [13] takes advantage of the fact that many decision support queries are well-known 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 [27], which also details theoretical and practical bounds on the (often high) cost of indexing.

Work on *approximate query processing*, recently surveyed in [5], complements these efforts by observing that users can often be satisfied with approximate answers when they are simply using query results to iter-

ate through a search problem, exactly as we motive in our interactive search tasks.

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* [6, 9, 33]. Efficient algorithms for *approximate* nearest neighbor in certain high-dimensional spaces, such as locality-sensitive hashing [12], are available. However, these require the similarity metric 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.

In systems research, our work builds on the insight of active disks [1, 20, 25] 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 [21, 22, 26, 32]. The work of Abacus [2], Coign [18], River [3] and Eddies [4] 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 [4] 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 [28]. 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.

Recent efforts to standardize object-based storage devices (OSD) [30] 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.

8 Conclusion

Diamond is a system that supports interactive data analysis of large complex data sets. This paper argues that these applications require applying brute-force search to a portion of the objects. To efficiently perform brute-force search the Diamond architecture uses *early discard* to push filter processing to the edges of the system — executing semantic data filters directly at storage devices, and greatly reducing the flow of data into the central bottlenecks of a system. The Diamond architecture also enables the system to adapt to different hardware configurations by dynamically adjusting where computation is performed.

To validate our architecture, we have implemented a prototype version of Diamond and an application, SnapFind, that interactively searches collections of digital images. Using this system, we have demonstrated that searching large collections of images is feasible and that the system can dynamically adapt to use the available resources such as network and host processor efficiently.

In the future, we plan to work with domain experts to create new interactive search applications such as ligand screening or satellite imagery analysis. Using these applications, we plan to validate our approach to interactive search of large real-world datasets.

Acknowledgments

Thanks to: Derek Hoiem and Padmanabham (Babu) Pillai for their valuable help with the Diamond system; Genevieve Bell and David Westfall for contributing data for the SnapFind user study; Ben Janesko and David Yaron for useful discussions on applying Diamond to computational chemistry problems; our shepherd, Christos Karamanolis for all his help; and the anonymous reviewers for feedback on an earlier draft of the paper.

References

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

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

[3] 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).

[4] AVNUR, R., AND HELLERSTEIN, J. Eddies: Continuously adaptive query processing. In *Proceedings of SIGMOD* (2000).

[5] BABCOCK, B., CHAUDHURI, S., AND DAS, G. Dynamic sample selection for approximate query processing. In *Proceedings of of SIGMOD* (2003).

[6] BERCHTOLD, S., BOEHM, C., KEIM, D., AND KRIEGEL, H. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of PODS* (May 1997).

[7] BORAL, H., AND DEWITT, D. Database machines: an idea whose time has passed? A critique of the future of database machines. In *Proceedings of of the International Workshop on Database Machines* (1983).

[8] BRADSKI, G. Programmer's tool chest: The OpenCV library. *Dr. Dobbs Journal* (November 2000).

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

[10] FALOUTSOS, C. *Searching Multimedia Databases by Content*. Kluwer Academic Inc., 1996.

[11] 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* (1995).

[12] GIONIS, A., INDYK, P., AND MOTWANI, R. Similarity search in high dimensions via hashing. In *Proceedings of VLDB* (1999).

[13] 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).

[14] GRAYSON, P., TAJKHORSHID, E., AND SCHULTEN, K. Mechanisms of selectivity in channels and enzymes studied with interactive molecular dynamics. *Biophysical Journal 85* (2003).

[15] 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).

[16] HOIEM, D., SUKTHANKAR, R., SCHNEIDERMAN, H., AND HUSTON, L. Object-based image retrieval using the statistics of images. Tech. Rep. Intel Research IRP-TR-03-13, November 2003.

[17] HSIAO, D. Database machines are coming, database machines are coming. *IEEE Computer 12*, 3 (1979).

[18] HUNT, G., AND SCOTT, M. The Coign automatic distributed partitioning system. In *Proceedings of OSDI* (1999).

[19] HUSTON, L., SUKTHANKAR, R., R. WICKREMESINGHE, M. S., GANGER, G., RIEDEL, E., AND AILAMAKI, A. Diamond: A storage architecture for early discard in interactive search. Tech. Rep. IRP-TR-2004-02, Intel Research, January 2004. `http://www.intel-research.net/pittsburgh/publications.asp`.

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

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

[22] 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).

[23] MINKA, T., AND PICARD, R. Interactive learning using a society of models. *Pattern Recognition 30* (1997).

[24] PRUESSE, G., AND RUSKEY, F. Generating linear extensions fast. *SIAM Journal on Computing 23*, 2 (April 1994).

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

[26] 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.

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

[28] 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).

[29] SMEULDERS, A., AND WORRING, M. Content-based image retrieval at the end of the early years. *IEEE Transactions on Pattern Analysis and Machine Intelligence 22*, 12 (2000).

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

[31] 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 System Principles* (December 1993).

[32] 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).

[33] YAO, A., AND YAO, F. A general approach to D-Dimensional geometric queries. In *Proceedings of STOC* (May 1985).