Network-Aware Partitioning of Computation in Diamond

Alex Nizhner[1] Larry Huston[2]
Peter Steenkiste Rahul Sukthankar[3]

June 2004
CMU-CS-04-148

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The Diamond storage architecture enables efficient interactive search of unindexed data by supporting the execution of application-specific filter binaries directly at storage devices. This functionality allows irrelevant objects to be discarded at the early stages of the search pipeline, thereby reducing the load on the interconnect and the user's workstation. In order to achieve efficient use of resources under dynamic conditions, Diamond adaptively partitions computation among the storage devices and the user's host.

In this paper, we explore the behavior of Diamond systems in network-bound configurations. We develop a performance model capturing the pertinent properties of a Diamond system; in particular, we characterize the amount of network traffic generated as a result of evaluating a Diamond query. Ultimately, we formulate a partitioning algorithm that provably minimizes the amount of traffic injected into the network during the execution of a search under CPU time constraints at processing stages.

[1]Alex Nizhner (nizhner@andrew.cmu.edu) is with the Information Networking Institute.
[2]Larry Huston is with Intel Research Pittsburgh.
[3]Rahul Sukthankar is with Intel Research Pittsburgh and the Robotics Institute.

Keywords: Active storage, dynamic partitioning, interactive search

1 Introduction

Diamond [4] is a storage architecture that enables interactive brute-force search of large collections of unindexed data, permitting the user to quickly retrieve a small fraction of desired objects. Under exhaustive search implemented with a traditional storage architecture, each data item passes from a storage device to the user's workstation over the interconnect; most of those items are subsequently discarded at the users host. The vast quantities of data involved typically overload the interconnect or the host processor. The Diamond architecture, on the other hand, is based on the concept of *early discard*, where the majority of irrelevant data are discarded at the early stages of the search pipeline. Early discard is made possible by the active storage abstraction and the execution of application-specific code directly at storage devices (Figure 1).

A principal feature of the Diamond architecture is its dynamic adaptation to changing run-time conditions precipitated by heterogeneous storage device configurations, hardware or interconnect infrastructure updates, competing searches executing concurrently, and other factors. The current Diamond prototype uses run-time measurements of pertinent system state to arrive at the most efficient ordering of query elements, and to adaptively partition computation between the storage devices and the host system. In this paper we concentrate on the latter problem.

The Diamond run-time continuously adjusts the amount of computation performed by processing stages: work is re-distributed among the storage devices and the host as necessary in order to ensure that processing elements are neither over-loaded nor under-utilized. In systems limited by the capacity of the interconnect, such dynamic partitioning presents an additional problem—namely, it becomes vital to minimize the amount of data exchanged between the storage devices and the host workstation. The set of objects discarded at the storage device, and consequently the amount of data transferred over the network, can vary drastically depending on which elements of a query are applied to objects in the storage subsystem. The goal of this work is a run-time mechanism for determining the elements of a query that are to be executed on a given object in order to minimize the amount of network traffic generated, coupled with dynamic adaptation of processing load. In the sections that follow, we derive an efficient solution to this problem.

The rest of this paper is organized as follows. Section 2 provides a brief discussion of related work. Section 3 presents an overview of an idealized Diamond architecture and introduces the assumptions and basic notation used throughout the paper. Section 4 builds on the notation introduced in Section 3 to develop an analytical framework for the problem at hand. In Section 5, we formally analyze the problem of network-aware partitioning, describe a partitioning algorithm that minimizes the communication cost associated with a search and prove its optimality. Finally, Section 6 summarizes the main contributions of this paper and discusses several directions for future work.

2 Related Work

The problem we consider in this paper falls in the more general domain of automated distributed partitioning. Here we mention some of the more relevant efforts in this area.

Abacus [1], a run-time system for data-intensive applications, automatically moves computation between storage servers and clients in a cluster environment. Migration decisions in Abacus are based on run-time observations and profiling of inter-object communication patterns and resource usage. Coign [3] profiles the inter-component communication patterns of COM-based applications and applies a graph-cutting algorithm offline to arrive at a static partitioning that minimizes

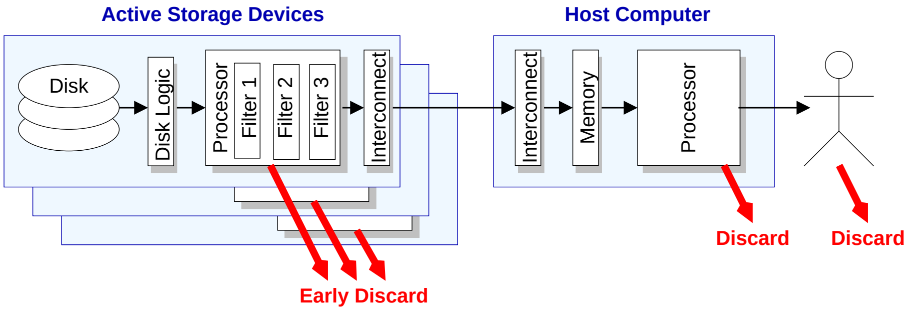


Figure 1: **Diamond Architecture**

communication latency. River [2] adaptively balances the data rates in producer-consumer pairs, allowing for efficient use of heterogeneous cluster resources.

Diamond shares many features of the above systems, particularly in the dynamic re-distribution of computational load across the storage devices and the user's host, and filter reordering based on run-time observations of system state. Features that set our work apart are Diamond's run-time environment and programming model, tailored specifically for interactive search. The resulting regularity of communication patterns in a Diamond system makes the hard problem of run-time distributed partitioning at least partially tractable and analyzable, and allows certain optimality guarantees to be made.

3 Diamond Overview

A Diamond query is represented as a *searchlet*—a piece of executable code comprised of a partially-ordered set of *filters*, each of which may independently choose to discard an object. The filters in a searchlet are evaluated sequentially in an order chosen by the Diamond run-time. The fraction of objects passed by any given filter depends on the filter's relative position in the evaluation order and the set of objects passed by previously-evaluated filters, as shown in Figure 2. This fraction is referred to as the filter's *conditional pass rate* in [4]; the Diamond system provides facilities for its run-time estimation. Our work assumes knowledge of the conditional pass rates of all filters in a fixed ordering.

For each filter, the run-time environment maintains an estimate of its average computation cost expressed in units of CPU time; we again assume knowledge of this average CPU cost in our analysis. Additionally, each filter may compute some amount of intermediate state for each object it passes, and communicate it to subsequent filters in the form of object attributes (in fact, filters are sometimes designed for the sole purpose of computing such state, and perform no discard at all). Likewise, a filter may remove a piece of state computed by an earlier filter. The Diamond run-time maintains a running estimate of the average amount of metadata each filter adds to objects it passes; below, we assume knowledge of this quantity as well.

Throughout our analysis, we focus our attention on an idealized Diamond pipeline shown in Figure 1, consisting of k identical storage devices, a host computer, and an interconnect with a constant bandwidth of R bytes per second. We assume, for simplicity, that the host computer can process objects at the same rate as any of the storage devices. Cases where this assumption does not hold are easily accounted for with appropriate adjustments to the constant k. Presently we do

Figure 2: **Searchlet Evaluation.** The filters are applied, in sequence, to each object in the collection. The order of evaluation is chosen by the Diamond run-time environment. In the ordering above, Filter 1 is evaluated first, and passes p_1 of the objects to which it is applied. Filter 2 is evaluated next, passing p_2 of the objects passed by Filter 1; similarly, Filter 3 executes last, with a conditional pass rate of p_3.

not consider multi-level hierarchies with intermediate processing nodes or heterogeneous storage device pools, although our results can naturally generalize to such configurations.

It is understood that the algorithms presented here are used in the context of a more general adaptive partitioning scheme whereby the computational resources of the system are kept appropriately utilized. Similarly, we rely on the existence of a scheme that determines an efficient filter ordering as described in [4], and assume this ordering to be fixed for the purposes of this work.

We now summarize the notation and terminology used in subsequent sections. Consider a *sequence* of filters $F = (F_i \mid 0 \leq i < n)$, which determines both a searchlet on n filters and a fixed order in which those filters are evaluated. We associate the following parameters with each filter F_i:

- c_i: The average computational cost or running time.
- p_i: The conditional pass rate (i.e., the fraction of passed objects among all those reaching F_i in their transit through F).
- M_i: The average amount of meta-data added to passing objects.
- D_i: The average amount of per-object data seen by F_i, given by $S + \sum_{k=0}^{i-1} M_i$, where S is the average object size.

These parameters (summarized in Table 1) determine the externally-visible behavior of a filter sequence—that is, the set of objects discarded, the average evaluation time, and the intermediate state carried by passing objects. The definition below will prove useful in the discussion that follows.

Definition 1 *Filter sequences F and F' are said to be* equivalent *if they have the the same observable effect on a given object, that is,*

- *both F and F' either pass the object or discard it;*
- *if F and F' pass the object, the amounts of meta-data generated by F and F' are the same;*
- *when executed to completion, F and F' exhibit identical running times.*

The Diamond run-time may manipulate a filter sequence (e.g., by combining adjacent filters into monolithic blocks) in order to reduce its network traffic footprint, as long as the externally-visible behavior of the sequence is unchanged. In other words, the sequence must remain equivalent to the original.

Table 1: **Essential Notation**

F	An ordered sequence on n filters
F_i	The filter at index i in F
c_i	The average running time of F_i
p_i	The conditional pass rate of F_i
M_i	The average metadata due to F_i
D_i	The average per-object data seen by F_i

4 Performance Model

In this section, we describe a generalized model of filter sequence evaluation, and introduce a collection of related analytical tools.

4.1 The Bypass Distribution

The partitioning algorithm described in [4] executes the filter sequence in its entirety on a fraction of objects, and transmits the remaining fraction to the host unprocessed. In general, the evaluation of a filter sequence on a given object need not complete at the storage device—it may be suspended upon the evaluation of any filter and resumed at the host. For the purposes of analysis, each filter can be treated as though configured to execute on some fraction of objects reaching it in their transit through the sequence. We formalize this notion in terms of a *bypass distribution*, shown conceptually in Figure 3.

Definition 2 *A* bypass distribution *is a sequence of fractions* $b_i \in [0, 1], \leq i < n$ *assigned to corresponding filters of a sequence* F *such that:*

- *A* $(1 - b_i)$ *fraction of objects reaching* F_i *are transmitted to the host along with their accumulated metadata; evaluation resumes at the host starting with* F_i, *and*
- F_i *is evaluated at the disk on the remaining* b_i *fraction of objects reaching it.*

It should be emphasized that this model does not prescribe a particular implementation of the Diamond run-time environment. Any evaluation strategy can be generalized in this fashion, including dynamic ones in which bypass fractions are determined implicitly as a result of per-object decisions.

As will become apparent in Section 5.4, bypass distributions under which contiguous subsequences of filters are executed atomically are of particular interest. The original partitioning algorithm in Diamond is a special case of this, with the entire searchlet executing to completion on any given object, that is, with $b_i = 1$ for $i > 0$.

Definition 3 *Given a sequence* F *and a bypass distribution* b, *the subsequence*

$$F_{km} = (F_k, F_{k+1}, \ldots, F_m \mid k \geq 0,\ k \leq m < n)$$

is said to comprise a filter group *or* execute atomically *under* b *if* $b_{k+1} = \cdots = b_m = 1$.

Note that we may construct a sequence F' equivalent to F by replacing F_{km} with a single filter which simply executes the filters in F_{km} sequentially; for F', the filters in F_{km} always execute atomically.

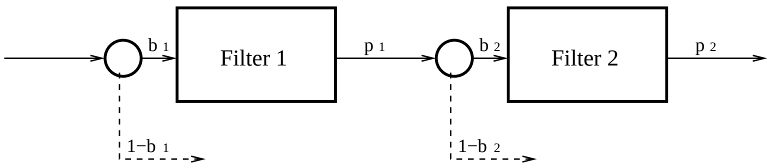


Figure 3: **Bypass-based Partitioning.** Filter 1 executes on the fraction b_1 of the ojects reaching it, and of all those passes the fraction p_1. The fraction $1 - b_1$ of all objects reaching Filter 1 are transmitted to the host for processing. Similarly, Filter 2 executes on the fraction $b_1 p_1 b_2$ of objects that reach Filter 1, while the remaining objects passing Filter 1 are transmitted to the host.

4.2 Cost Functions

The assignment of a bypass distribution to a filter sequence determines a collection of related cost metrics. Below, we express the on-disk and host computational costs, communication cost, and overall object throughput associated with evaluating a sequence F as functions of a bypass distribution b.

4.2.1 CPU Cost

We first derive the average CPU time required in order to fully evaluate the sequence F—that is, the expected time until an object is either discarded or presented to the user. Consider the fraction of objects in the entire collection on which a given filter is evaluated. Filter F_0 is applied to every object. Filter F_1, on the other hand, sees only p_0 of those: that is, only the objects that pass F_0. Similarly, each successive filter is evaluated only on the objects that pass its predecessor. C_F, the average computational workload associated with evaluating the sequence F, is then given by:

$$C_F = c_0 + p_0 c_1 + p_0 p_1 c_2 + \cdots + \left(\prod_{i=0}^{n-2} p_i \right) c_{n-1} \tag{1}$$

With the assignment of a bypass distribution, the execution frequency of a filter at a storage device is adjusted to account for its bypass fraction, and the bypass fractions of the preceding filters. Thus, F_0 executes on b_0 of the total number of objects in the collection, and F_1, accordingly, on $b_0 p_0 b_1$—that is, b_1 of the objects passed by F_0. The average "on-disk" running time of F under b is therefore expressed as follows:

$$C(b) \quad = \quad b_0 c_0 + b_0 b_1 p_0 c_1 + b_0 b_1 b_2 p_0 p_1 c_2 + \cdots + \left(\prod_{i=0}^{n-1} b_i \right) \left(\prod_{i=0}^{n-2} p_i \right) c_{n-1}, \tag{2}$$

where the i-th term in the summation corresponds to the incremental cost of evaluating F_i under b. Note that $C_F = C(b)$ for $b = (1, 1, \ldots, 1)$.

The average additional CPU time required to complete the evaluation of F at the host is simply the complement of the storage device CPU time $C(b)$:

$$H(b) = C_F - C(b) \tag{3}$$

4.2.2 Network Cost

Next, we derive the expected amount of data emitted by the disk per object as a result of evaluating F under b.

Let S be the average object size, and let $D_i, 0 \leq i \leq n$ be the amount of data that would be injected into the network if filter F_i were chosen for bypass; i.e., $D_i = S + \sum_{k=0}^{i-1} M_i$. (Recall from Section 3 that M_i represents the average amount of intermediate state added to objects by filter F_i.) Consider the quantity $N_i(b)$, the expected amount of data emitted given that $F_0, \ldots, F_{i-1}$ have executed and passed the object:

- With probability $(1 - b_i)$, the evaluation of F results in the emission of D_i bytes.
- With probability b_i, F_i executes, resulting in 0 bytes in the event of discard, and $N_{i+1}(b)$ bytes otherwise.

Thus,

$$N_i(b) = b_i p_i N_{i+1}(b) + (1 - b_i) D_i, \tag{4}$$

with $N_n(b) = D_n$. $N(b)$, the average per-object amount of data transferred over the interconnect, is then given by $N_0(b)$, which expands to

$$N(b) = \sum_{k=0}^{n-1} \left[\left(\prod_{i=0}^{k-1} b_i \right) \left(\prod_{i=0}^{k-1} p_i \right) (1 - b_k) D_k \right] + \left(\prod_{i=0}^{n-1} b_i \right) \left(\prod_{i=0}^{n-1} p_i \right) D_n$$

Note that the above expression consists of precisely $n + 1$ additive terms. Each of the first n represents the average amount of data injected into the network in the event that the on-disk evaluation of F stops at the given filter, weighted by the corresponding probability; in a similar fashion, the final term corresponds to the event that an object passes through all of F at the disk.

For the special case where $M_i \ll S, i \in \{0, \ldots, n-1\}$ (i.e., searchlets consisting of filters that compute little intermediate state), we have

$$\begin{aligned} N(b) &\simeq S \sum_{k=0}^{n-1} \left[\left(\prod_{i=0}^{k-1} b_i \right) \left(\prod_{i=0}^{k-1} p_i \right) (1 - b_k) \right] + S \left(\prod_{i=0}^{n-1} b_i \right) \left(\prod_{i=0}^{n-1} p_i \right) \\ &= SP, \end{aligned}$$

where the quantity

$$P = \sum_{k=0}^{n-1} \left[\left(\prod_{i=0}^{k-1} b_i \right) \left(\prod_{i=0}^{k-1} p_i \right) (1 - b_k) \right] + \left(\prod_{i=0}^{n-1} b_i \right) \left(\prod_{i=0}^{n-1} p_i \right)$$

represents the fraction of objects escaping the disk.

4.2.3 Throughput

Let a bypass distribution b be given. In a system where the data collection is evenly distributed among k identical storage devices, the storage subsystem emits k objects every $C(b)$ seconds on average, yielding a per-object time of $\frac{C(b)}{k}$. With a constant bandwidth of R bytes per second, the interconnect transfers the average object in $\frac{N(b)}{R}$ seconds. Finally, the host workstation takes an average of $H(b)$ seconds to process each arriving object.

Overall object throughput is then determined by that of the slowest stage of the search pipeline:

$$T(b) = \frac{1}{\max\left(\frac{C(b)}{k}, \frac{N(b)}{R}, H(b)\right)} \tag{5}$$

5 CPU-Constrained Partitioning

We now turn our attention to the problem of determining a throughput-maximizing partitioning of work among the storage devices and the host workstation, with a particular emphasis on configurations limited by the speed of the interconnect. A Diamond system attempts to arrive at an allocation of the workload to processing resources that ensures that neither the storage devices nor the host are idle, resulting in a typically-nontrivial fraction of the computational workload being assigned to each storage device. With run-time observation of transmit queue sizes as indicative of processing resource utilization, this fraction can be numerically estimated, or determined implicitly by means of per-object decisions. Below, we make the simplifying assumption that a numeric value of the computational workload is specified externally.

Definition 4 *We define the* computational workload *or* CPU time constraint *assigned to a storage device as the amount of CPU time the storage device is expected to devote to the average object. This quantity can be expressed as* $\beta C_F, \quad \beta \in [0,1]$*, where* C_F *is the expected total computational workload. The assigned computational workload is specified in terms of the fraction* β *alone.*

A storage device may satisfy this assignment in numerous ways, by choosing to evaluate certain filters on some objects but not others. However, distinct evaluation (i.e., partitioning) strategies that in the long run yield the same per-object computational expenditures can exhibit vastly different network behavior—an unsurprising phenomenon that is due to the differences in filter selectivities and the amounts of intermediate state produced. Clearly, in network-bound systems it is crucial that the amount of traffic generated by an evaluation strategy be minimized. Somewhat less obvious is the case of CPU-bound systems, where reducing the levels of network traffic is beneficial due to the nontrivial CPU costs associated with transmitting data.

In this section we focus on identifying CPU-constrained partitioning strategies that minimize the amount of data transferred over the interconnect. We begin with a look at the simple strategy described in [4], which we will call the Naive partitioning strategy. Under Naive partitioning, the Diamond run-time makes no per-filter decisions at all; instead, the entire filter sequence is evaluated atomically on a fraction of objects corresponding to the fraction of the computational workload assigned to the storage device. The remaining objects are processed similarly at the host. Intuition suggests that this technique does not always minimize the load on the interconnect. Consider, for example, a filter sequence in which a highly selective and computationally-inexpensive filter is followed by a very expensive one which happens to pass most objects it sees, while computing an intermediate representation of the objects for use by later filters in the sequence. As we will show, Naive partitioning is indeed suboptimal in such cases. Conversely, we will see that Naive partitioning can perform very well in the symmetric case, i.e., that of an expensive but not selective filter followed by a cheap and selective one.

In the rest of this section, after introducing some necessary formalism, we examine another partitioning strategy that sheds more light on our goal. We then derive a strategy that is optimal in terms of network traffic footprint and compare its behavior to that of the other strategies on a few representative filter sequences.

5.1 Filter Efficiency

Before proceeding further, we introduce some additional terminology in order to capture the notion of the relative "desirability" of filters at which we hinted above.

Definition 5 *We define* δ_i*, the* output ratio *of filter* F_i*, to be the per-object amount of data* F_i *offers to subsequent filters relative to the amount of data* F_i *accepts, in proportion to its normalized running time:*

$$\delta_i = \frac{p_i D_{i+1} - D_i}{\frac{c_i}{C_F}} = \frac{C_F}{c_i}\left(p_i D_{i+1} - D_i\right) \tag{6}$$

Recall that D_i represents the average amount of data carried by objects seen by F_i, and D_{i+1}, accordingly, is the amount of data carried by objects which F_i passes.

Low output ratios are thus more desirable than high ones—an inexpensive and selective filter is expected to have a large negative output ratio, in contrast to the positive output ratio of a filter that computes large amounts of intermediate state and discards few objects. As will become apparent shortly, sequences in which filters are ordered from most to least efficient deserve special recognition:

Definition 6 *A filter sequence* F *is said to be* concave *if* $\delta_i \leq \delta_{i+1}$ *for* $0 \leq i \leq n-2$*. Note that a filter sequence that is not concave must contain a pair* (F_k, F_{k+1}) *with* $\delta_k > \delta_{k+1}$*. Such filter pairs are referred to as* convex*.*

Filters in a non-concave sequence can be thought of as participating in a producer-consumer relationship, almost literally so in the case of filters designed for the purpose of computing intermediate state. One would expect that such convex pairs will be executed atomically under a network-optimal partitioning: naturally, data ought to be consumed close to where it is produced. In Section 5.4 we demonstrate that this is indeed the case.

5.2 The Bypass Distribution Function

In the discussion above, we had implicitly defined a filter sequence evaluation strategy satisfying a CPU time constraint as a mapping from a fraction of the total computational workload to a subsequence of filters evaluated on a given object at the storage device. In Section 4, we had argued that the latter is naturally generalized in terms of a bypass distribution. We formalize this concept as follows:

Definition 7 *A partitioning algorithm on a filter sequence* F *is defined as a function* B_F *that maps all* $\beta \in [0,1]$ *to bypass distributions, with the additional property that*

$$C(B_F(\beta)) = \beta C_F$$

We refer to the function B_F *as a* bypass distribution function *on the filter sequence* F*.*

The quantity βC_F represents the CPU time constraint. It is computed from β, the fraction of the total workload assigned to the storage device; accordingly, the host system is assigned the fraction $(1-\beta)$, resulting in an average per-object CPU time expenditure of $(1-\beta)C_F$. Thus, an assigment of $\beta = 0$ dictates that all computation be performed at the host (achieved with a bypass distribution in which the leading element is 0), whereas $\beta = 1$ requires that the storage device fully process every object, corresponding to the bypass distribution $b = (1, 1, \ldots, 1)$. In this manner we can express the Naive partitioning algorithm described above as the bypass distribution function B_F^N such that $\forall \beta \in [0, 1], \quad B_F^N(\beta) = (\beta, 1, 1, \ldots, 1)$.

Given β and a distribution function B_F, throughput is not limited by the network if and only if

$$\begin{aligned} \frac{N(B_F(\beta))}{R} &\leq \max\left(\frac{C(B_F(\beta))}{k}, H(B_F(\beta))\right) \\ &= \max\left(\frac{\beta C_F}{k}, (1-\beta)C_F\right) \\ &= \max\left(\frac{\beta}{k}, 1-\beta\right) C_F \end{aligned}$$

Restated more formally, our challenge is the optimization problem of computing distribution functions that minimize their associated network costs.

Definition 8 *A bypass distribution function B_F^* is called an* optimal distribution function *if $\forall B_F$ and $\forall \beta \in [0, 1]$,*

$$N(B_F^*(\beta)) \leq N(B_F(\beta))$$

Note that this definition implies neither the uniqueness nor the existence of B_F^* for an arbitrary filter sequence F. The latter will be demonstrated in Section 5.4.

5.3 Greedy Partitioning

We now revisit the hypothetical filter pair that we saw in the discussion of the Naive partitioning strategy, a pair comprised of a highly desirable filter followed by a highly undesirable one. Our intuition was that Naive partitioning was unlikely to perform well on such filter sequences: since the second filter provides meager (if any) benefits in terms of data reduction while carrying a nontrivial CPU time penalty, we could achieve much better results by executing the first filter on as many objects as possible under the CPU time constraint, and immediately transmitting the passing objects to the host. There we could evaluate the second filter without affecting the interconnect at all. We expect that a partitioning strategy with such properties would be optimal for this class of filter pairs; moreover, we expect that a similar approach of "greedily" executing filters would generalize to all sequences in which the filters are ordered from most to least desirable (that is, the concave sequences defined in Section 5.1). In this section, we formulate such a strategy and confirm our intuition.

Consider a bypass distribution function B_F^G (hereafter referred to as the Greedy distribution function) characterized by the property that $\forall \beta \in [0, 1]$ and $b = B_F^G(\beta), \quad b_i \neq 0 \Rightarrow b_{i-1} = 1$ for $i > 0$. The Greedy distribution function satisfies the disk CPU constraint with the shortest possible subsequence of the filter sequence F starting with filter F_0. It is easy to see that there exists a unique Greedy distribution function for any sequence F.

Let $\beta_i = \beta_{i-1} + \frac{c_{i-1}}{C_F} \prod_{k=0}^{i-2} p_k$ for $1 \le i \le n$, with $\beta_0 = 0$; the quantity $c_{i-1} \prod_{k=0}^{i-2} p_k$ corresponds to the incremental cost of F_{i-1} in the expansion of C_F. (Note that $\beta_n = 1$.) The fraction β_i represents the normalized cumulative CPU time associated with executing the sequence F until filter F_i is reached. Let $\beta \in [0, 1]$ be given, and let us choose F_k such that $(F_0, \ldots, F_k)$ is the shortest subsequence of F starting at F_0 which can satisfy the CPU time constraint. $B_F^G(\beta)$ then exhibits the following structure:

- $b_i = 1$ for $0 \le i < k$; that is, the filters $F_0, \ldots, F_{k-1}$ are evaluated at the storage device only.
- $b_k = \frac{\beta - \beta_k}{\beta_{k+1} - \beta_k}$. The fraction of objects on which the filter F_k is evaluated at the storage device is chosen so that the CPU time constraint is satisfied.
- $b_i = 0$ for $k < i < n$; that is, none of the remaining filters are evaluated at the storage device.

We now examine the shape of $N \circ B_F^G$, the network cost of the Greedy distribution function, illustrated in Figure 4. We observe that $N \circ B_F^G$ is piecewise-linear in β, with linear segments punctuated at points β_i for $0 \le i \le n$; furthermore, we note that $N(B_F^G(\beta_i)) = \left(\prod_{k=0}^{i-1} p_i\right) D_i$, where D_i, again, is the total amount of data associated with the average object immediately before the execution of filter F_i. Each segment of $N \circ B_F^G$ naturally corresponds to a filter in the sequence F: for $\beta < \beta_i$, filter F_i is never evaluated at the storage device, whereas for $\beta > \beta_{i+1}$ it is never evaluated at the host. Moreover, the following holds for the slope s_i of each segment of $N \circ B_F^G$:

$$\begin{aligned}
s_i &= \frac{N(B_F^G(\beta_{i+1})) - N(B_F^G(\beta_i))}{\beta_{i+1} - \beta_i} \\
&= \frac{C_F}{c_i \prod_{k=0}^{i-1} p_k} \left(\left(\prod_{k=0}^{i} p_i \right) D_{i+1} - \left(\prod_{k=0}^{i-1} p_i \right) D_i \right) \\
&= \frac{C_F}{c_i} \left(p_i D_{i+1} - D_i \right) \\
&= \delta_i
\end{aligned}$$

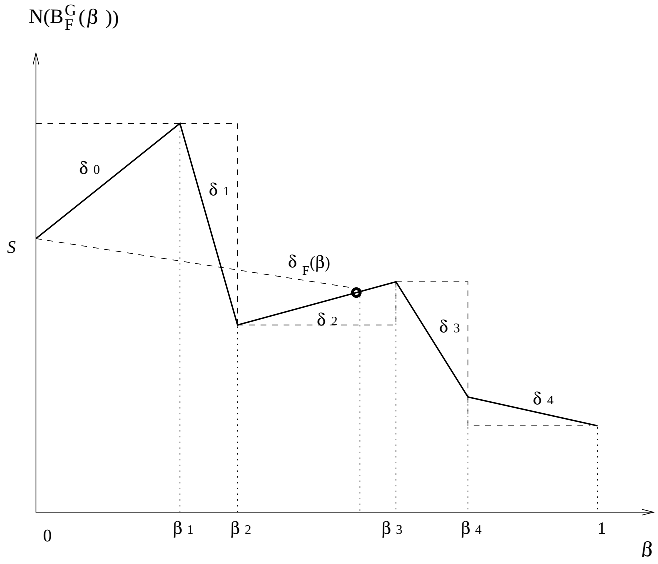


Figure 4: **The Greedy Distribution Function and Filter Output Ratios.** The slope of the segment of $N \circ B_F^G$ corresponding to filter F_i is precisely δ_i, the output ratio of F_i. The quantity $\delta_F(\beta)$ is defined as the slope of the line connecting $N(B_F^G(0))$ and $N(B_F^G(\beta))$.

The shape of $N \circ B_F^G$ thus permits a very natural graphical interpretation of the terminology we defined in Section 5.1. In particular, the terms "concave" and "convex" are given their proper geometric meaning, as shown in Figure 5.

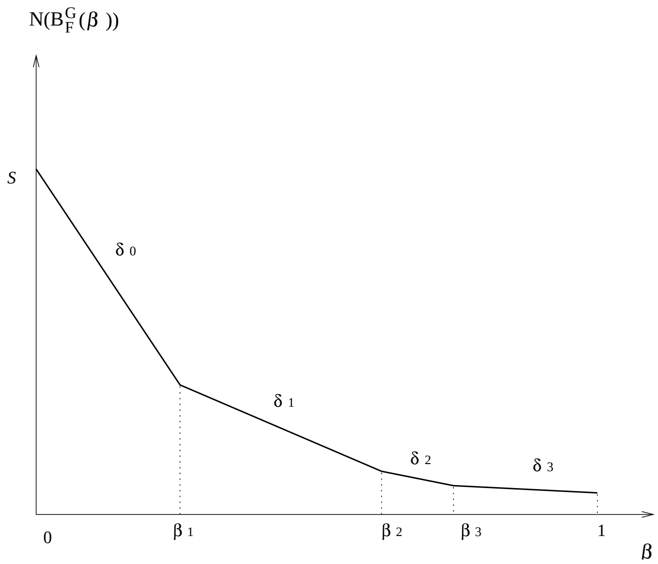
We are now ready to prove the main result of this section, namely the optimality of Greedy partitioning for concave filter sequences. As an analytical aid, we define the quantity $\delta_F(\beta)$ as the slope of $N \circ B_F^G$ over the range $[0, \beta]$:

$$\begin{aligned} \delta_F(\beta) &= \frac{N(B_F^G(\beta)) - N(B_F^G(0))}{\beta} \\ &= \frac{N(B_F^G(\beta)) - D_0}{\beta} \end{aligned}$$

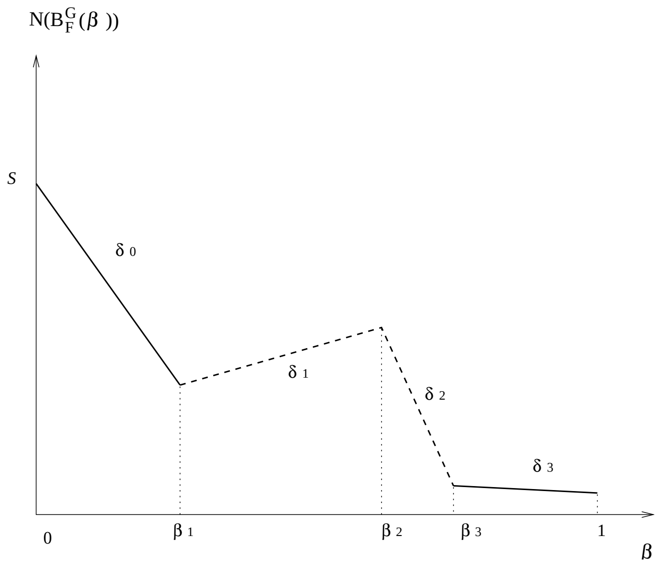
Claim 1 B_F^G *is an optimal distribution function for any concave filter sequence F.*

Proof By induction on n:

- Base Case. The result holds trivially for one-filter searchlets.
- Inductive Step. Let $\beta \in [0, 1]$ and a concave n-filter sequence F with $n > 1$ be given. Consider a bypass distribution b such that $C(b) = \beta C_F$. Of the total disk CPU budget βC_F, filter F_0 consumes $b_0 c_0$ units, with the remaining $\frac{\beta C_F - b_0 c_0}{b_0 p_0}$ units assigned to the subsequence $F' = (F_1, \ldots, F_{n-1})$ with total cost $C_{F'} = \frac{C_F - c_0}{p_0}$. The latter allocation translates into a workload fraction of $\beta' = \frac{\beta C_F - b_0 c_0}{b_0 p_0 C_{F'}}$. F' is a concave sequence on $n - 1$ filters; applying the inductive hypothesis yields that $B_{F'}^G$ is an optimal distribution function on F'. With



(a) A concave sequence



(b) A non-concave sequence with convex pair (F_1, F_2)

Figure 5: **Filter Sequence Classification.** The Greedy distribution function allows an intuitive graphical representation of filter sequence properties. The concave shape in (a) dictated by the monotonic ascending ordering of output ratios, and similarly the convex shape formed by the pair (F_1, F_2) in (b) give rise to the terminology used in this paper.

$N(B^*_{F'}(0)) = p_0 D_1$, we have $N(B^*_{F'}(\beta')) = p_0 D_1 + \beta' \delta_{F'}(\beta')$. Then:

$$
\begin{aligned}
N(b) &= b_0 N(B^*_{F'}(\beta')) + (1 - b_0) D_0 \\
&= b_0 (p_0 D_1 + \beta' \delta_{F'}(\beta')) + (1 - b_0) D_0 \\
&= \frac{\beta C_F}{p_0 C_{F'}} \delta_{F'}(\beta') + b_0 \left(D_0 + \frac{c_0}{C_F} \delta_0 - \frac{c_0}{p_0 C_{F'}} \delta_{F'}(\beta') \right) + (1 - b_0) D_0
\end{aligned}
$$

We show that $N(b)$ is a non-increasing function of b_0. Consider first the expression $\frac{\beta C_F}{p_0 C_{F'}} \delta_{F'}(\beta')$. Since F' is concave, $\delta_{F'}$ is a non-decreasing function of β'; β', in turn, is a monotonically decreasing function of b_0. Next, let us examine the quantity

$$
b_0 \left(D_0 + \frac{c_0}{C_F} \delta_0 - \frac{c_0}{p_0 C_{F'}} \delta_{F'}(\beta') \right) + (1 - b_0) D_0
$$

Since F is concave, δ_0 never exceeds $\delta_{F'}(\beta')$, scaled to account for the fractional cost of F' in F. More formally,

$$
\begin{aligned}
\delta_0 &\leq \frac{C_F}{p_0 C_{F'}} \delta_{F'}(\beta') \\
\frac{c_0}{C_F} \delta_0 &\leq \frac{c_0}{p_0 C_{F'}} \delta_{F'}(\beta') \\
\frac{c_0}{C_F} \delta_0 - \frac{c_0}{p_0 C_{F'}} \delta_{F'}(\beta') &\leq 0,
\end{aligned}
$$

from which we have

$$
D_0 + \frac{c_0}{C_F} \delta_0 - \frac{c_0}{p_0 C_{F'}} \delta_{F'}(\beta') \leq D_0
$$

The final two terms in the expression for $N(b)$ given above therefore also form a non-increasing function of b_0. It follows that $N(b)$ is minimized only when b_0 is maximized under the CPU time constraint—that is, assigned according to B^G_F.

5.4 Network-Optimal Partitioning

This section culminates our analysis with a description of a partitioning algorithm that minimizes the communication cost of executing a search under a CPU time constraint. Again, we begin by considering a sequence comprised of two filters with significantly different output ratios, e.g., a highly selective yet computationally inexpensive filter coupled with the opposite—a filter with a long running time that discards few objects. In the previous section, we have shown that Greedy partitioning achieves optimal results when such a sequence is concave, that is, when the cheap selective filter is evaluated first. Now we concentrate on the symmetric case of a convex pair. Intuitively, Greedy partitioning is precisely the wrong approach for such sequences. For any given object, evaluating both filters results in a much greater chance of discard than evaluating the first filter only, and since the second filter is comparatively inexpensive, evaluating both atomically on a slightly smaller fraction of objects generates a smaller overall amount of network traffic. As mentioned previously, the filters in a convex pair can be thought of as participating in a producer-consumer relationship, and separating the consumer from the producer is unlikely to be of value. Thus, we expect Naive partitioning to be optimal for convex two-filter sequences.

The arguments in this section proceed as follows. First, we formally prove the argument outlined above and establish that any network-optimal evaluation strategy must execute all convex pairs in a filter sequence atomically. We then recursively extend the notion of a convex pair, and demonstrate that all network-optimal evaluation strategies partition the filter sequence into atomically-executing filter groups (Definition 3) such that no convex pairs remain, and the resulting sequence of groups is concave. Finally, we apply the result of the preceding section to arrive at Group-Greedy partitioning—an evaluation strategy optimal for any filter sequence, concave or otherwise.

Before we begin, we must revisit the network cost function N defined in Section 4.2.2. The introduction of the filter output ratio δ_i permits an alternative formulation of N which will be useful in the discussion below:

$$\begin{aligned} N(b) &= (1-b_0)D_0 + b_0p_0(1-b_1)D_1 + \cdots + b_0\cdots b_{n-1}p_0\cdots p_{n-1}D_n \\ &= D_0 + b_0(p_0D_1 - D_0) + \cdots + b_0\cdots b_{n-1}p_0\cdots p_{n-2}(p_{n-1}D_n - D_{n-1}) \\ &= D_0 + b_0\frac{c_0}{C_F}\delta_0 + b_0b_1p_0\frac{c_1}{C_F}\delta_1 + \cdots + b_0\cdots b_{n-1}p_0\cdots p_{n-2}\frac{c_{n-1}}{C_F}\delta_{n-1}, \end{aligned} \tag{7}$$

where D_i, as defined in Section 3, stands for the overall amount of data associated with the average object immediately before the execution of filter F_i.

Claim 2 *Let F be a sequence containing a convex filter pair (F_i, F_{i+1}). There exists an equivalent sequence F' in which F_i and F_{i+1} comprise a filter group and $\forall\beta \in [0,1]$, $N(B^*_{F'}(\beta)) \leq N(B^*_F(\beta))$.*

Proof Let a sequence F with a convex pair (F_i, F_{i+1}) be given, and let b be an arbitrary bypass distribution on F under which F_i and F_{i+1} are not executed atomically—i.e., $b_i \neq 0$ and $b_{i+1} \neq 1$. We will show that it is possible to construct a bypass distribution that achieves the same CPU cost as b, at a lower network cost.

The quantity $c'_i = c_i + p_ic_{i+1}$ represents the contribution of (F_i, F_{i+1}) to C_F. Consider the bypass assignment b' defined as follows (the arguments below also hold for the case where $i = n-2$):

- $b'_k = b_k$ for $0 \leq k < i$
- $b'_i = \frac{b_ic_i + b_ib_{i+1}p_ic_{i+1}}{c'_i} = b_i\frac{c_i + b_{i+1}p_ic_{i+1}}{c'_i}$
- $b'_{i+1} = 1$
- $b'_{i+2} = \frac{b_ib_{i+1}b_{i+2}}{b'_i} = b_{i+2}b_{i+1}\frac{c'_i}{c_i + b_{i+1}p_ic_{i+1}}$
- $b'_k = b_k$ for $i + 2 < k < n$

We first demonstrate that the construction of b' is valid, that is, that b' satisfies $b'_i \in [0,1]$ for $0 \leq i < n$. Since $0 \leq b_{i+1} < 1$, we have $c_i + b_{i+1}p_ic_{i+1} < c'_i$, and consequently $0 \leq b'_i \leq b_i < 1$. For b'_{i+2}, we make the observation that the expression $b_{i+1}\frac{c'_i}{c_i + b_{i+1}p_ic_{i+1}}$ has no local extrema when viewed as a function of b_{i+1}, and, in particular, is monotonic in the range $[0, 1]$. At $b_{i+1} = 0$, the expression evaluates to 0; at $b_{i+1} = 1$, it evaluates to 1, and therefore, $0 \leq b'_{i+2} \leq b_{i+2} \leq 1$. All other bypass fractions in b' are valid by construction.

Next, we note that $b'_i c'_i = b_i c_i + b_i b_{i+1} p_i c_{i+1}$, and that $b'_i b'_{i+1} b'_{i+2} = b_i b_{i+1} b_{i+2}$: the former implies that the contribution of (F_i, F_{i+1}) (which is a filter group under b') to $C(b')$ is the same as the pair's joint contribution to $C(b)$, whereas the latter shows that a similar relationship holds among b and b' for all filters F_k where $k > i + 1$. Since the filters F_k, $0 \leq k < i$ clearly execute with the same frequency under b' as they do under b, we must have $C(b') = C(b)$.

Finally, consider the quantity $N_i(b') - N_i(b)$. Using Equation 7 and applying the fact that $b'_i b'_{i+1} b'_{i+2} = b_i b_{i+1} b_{i+2}$, we obtain

$$
\begin{aligned}
N_i(b') - N_i(b) &= b'_i \frac{c_i}{C_F} \delta_i + b'_i b'_{i+1} p_i \frac{c_{i+1}}{C_F} \delta_{i+1} - b_i \frac{c_i}{C_F} \delta_i - b_i b_{i+1} p_i \frac{c_{i+1}}{C_F} \delta_{i+1} \\
&= \frac{b_i}{C_F} \left(\frac{c_i + b_{i+1} p_i c_{i+1}}{c'_i} c_i \delta_i \right) - \frac{b_i}{C_F} \left(b_{i+1} p_i c_{i+1} \delta_{i+1} \right) - \frac{b_i}{C_F} c_i \delta_i + \\
&\quad \frac{b_i}{C_F} \frac{c_i + b_{i+1} p_i c_{i+1}}{c'_i} p_i c_{i+1} \delta_{i+1} \\
&= \frac{b_i}{C_F c'_i} c_i \delta_i \left(b_{i+1} p_i c_{i+1} - p_i c_{i+1} \right) - \frac{b_i}{C_F c'_i} p_i c_{i+1} \delta_{i+1} \left(c_i b_{i+1} - c_i \right) \\
&= \frac{b_i (1 - b_{i+1}) p_i c_i c_{i+1}}{C_F c'_i} \left(\delta_{i+1} - \delta_i \right)
\end{aligned}
$$

Since (F_i, F_{i+1}) is convex, we have $\delta_{i+1} < \delta_i$; consequently, $N_i(b') - N_i(b) < 0$, from which it follows that $N(b') < N(b)$. This in combination with the above shows that, for any bypass distribution b, one may construct a bypass distribution b' under which F_i and F_{i+1} comprise a filter group, with the additional properties that $C(b') = C(b)$ and $N(b') \leq N(b)$. The result follows.

Claim 2 thus implies that an optimal partitioning algorithm on F must execute all convex pairs in F atomically. We note that an atomically-executed pair is essentially a single filter, and consequently may form a convex pair with another filter or filter group of F.

Corollary For every filter sequence F, there exists an equivalent concave sequence F^c such that $\forall \beta \in [0, 1],\ N(B^*_{F^c}(\beta)) \leq N(B^*_F(\beta))$.

Proof Let a sequence F be given. The result holds trivially if F is concave. Otherwise, we may choose a convex filter pair (F_i, F_{i+1}); by Claim 2, we may collapse (F_i, F_{i+1}) into an atomically-executing filter group (as shown in Figure 6) to yield an equivalent sequence F' with one fewer filter than F and an optimal distribution function whose network cost never exceeds that of the optimal distribution function of F. Repeating this process eventually produces a concave sequence F^c—if none of the intermediate sequences are concave, the process terminates with a one-filter sequence that is concave by definition. Since the network cost of the optimal distribution function of each successive sequence is always below or equal to that of its predecessor, the result follows.

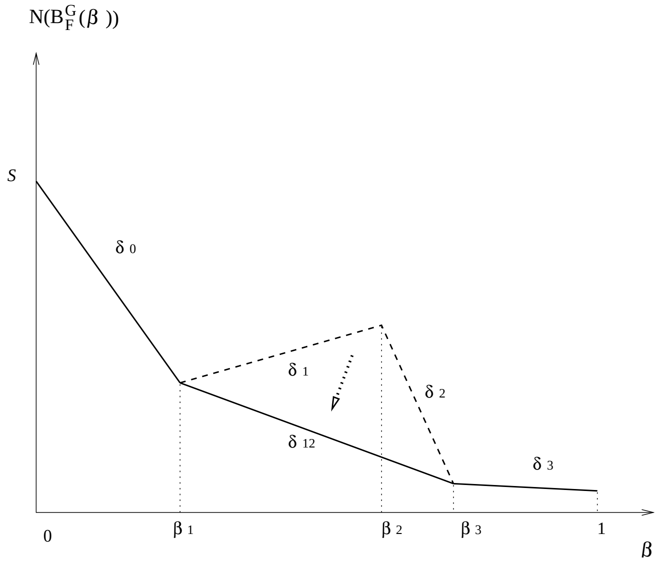


Figure 6: **Collapsing Convex Pairs.** The filters in a convex pair (F_1, F_2) are effectively "collapsed" into a single monolithic filter by the prescription of bypass assignments that ensure their atomic execution. Geometrically, the resulting filter is represented by a linear segment connecting the points corresponding to β_1 and β_3 on the plot of $N \circ B^G_F$. Since the pair (F_1, F_2) is convex, $N \circ B^G_F$ of the "new" sequence is strictly below that of the original on the interval (β_1, β_3).

Claim 2 and its Corollary imply that under any optimal distribution function, the filter sequence F is evaluated in such a way that collapsing designated filter groups into monolithic filters produces a concave sequence. Applying Claim 1, we see that every filter sequence F has an optimal distribution function given by $B^G_{F^c}$, obtained by applying the Greedy distribution function to the sequence of filter groups produced according to the Corollary. We will refer to it as the Group-Greedy distribution function, and denote it by B^U_F.

5.5 Group-Greedy Partitioning

In this section, we formulate an algorithm for computing the Group-Greedy distribution function at run time, and examine the behavior of Group-Greedy partitioning on a few representative filter sequences.

The Group-Greedy distribution function can be constructed by applying the Greedy distribution function to the concave sequence obtained via the algorithm outlined in the proof of the Corollary to Claim 2. Since it is precisely the proper grouping of filters that ensures the optimality of Group-Greedy partitioning, we concentrate on an efficient algorithm for determining this optimal grouping. We use the following property in its run-time implementation.

Claim 3 *For each filter group $F_{km} = (F_k, F_{k+1}, \ldots, F_m)$ of F that corresponds to a single filter of F^c, $\delta_{km} \leq \delta_{kj}$ for $k \leq j \leq m$.*

Proof The result can be seen to hold by a simple inductive argument. A filter group F_{km} of F corresponding to a single filter of F^c and consisting of more than one filter is comprised of two

```
for i = 0 to n − 1
    δ_min = ∞
    j_min = 0

    for j = i + 1 to n
        δ_ij = (N(B_F^G(β_j)) − N(B_F^G(β_i))) / (β_j − β_i)

        if δ_ij < δ_min
            δ_min = δ_ij
            j_min = j

    mark_group(i, j_min − 1)
    i = j_min
```

Figure 7: **Identifying the Optimal Filter Grouping**

sub-groups F_{kl} and F_{lm} that form a convex pair. It follows that $F_{km} < F_{kl}$; the argument is then applied to the sub-groups F_{kl} and F_{lm}.

Starting with $k = 0$, the filter group F_{km} is identified by selecting F_m such that $\delta_{km} \leq \delta_{ki}$ for $k \leq i < n$. The process repeats with $k = m + 1$ until $F_m = F_{n-1}$ is selected. The pseudo-code for this algorithm is given in Figure 7.

Given a partitioning of F into appropriate filter groups as specified above, $b = B_F^U(\beta)$ is constructed as follows. A filter group F_{ij} is chosen such that $\beta_i \leq \beta < \beta_{j+1}$. Then,

- $b_k = 1$ for $0 \leq k < i$; in other words, filters preceding the group F_{ij} in the sequence execute at the storage device only.
- $b_i = \frac{\beta - \beta_i}{\beta_{j+1} - \beta_i}$ (the group F_{ij} is evaluated at the storage device on a fraction of objects chosen so as to satisfy the CPU time constraint).
- $b_k = 1$ for $i < k \leq j$, in order to ensure the atomic execution of F_{ij}.
- $b_k = 0$ for $j < k < n$, i.e., none of the remaining filters are evaluated at the storage device.

Below, we qualitatively examine the network behavior of Group-Greedy partitioning, and compare it to that of the Greedy and Naive strategies.

5.5.1 Evaluation Methodology

We use the average number of bytes per object injected into the network by a storage device as the primary means of evaluating the performance of a partitioning strategy. The filter sequences we examine fall into three categories: all concave sequences, for which we've shown the Greedy policy to be optimal, non-concave sequences for which Naive partitioning achieves optimal results, and all remaining non-concave sequences.

In the latter two cases, we use filters written for the Diamond SnapFind application [4]. The two searchlets used were profiled on a set of 2000 digital images (with an average size of roughly

300 kilobytes) yielding estimates of selectivity, running time, and average amount of metadata generated for each of the filters. We focus only on the running-time-optimal filter ordering. The concave sequence used in the former case, on the other hand, is fully synthetic, with the pertinent parameters chosen so as to mimic the structure of a concave sequence. The average object size was arbitrarily set to 100 bytes, and no filter adds data to processed objects.

In all cases, we show the conditional pass rate, the average running time, and the average amount of per-object metadata generated for each of the filters. Comparisons are performed by simply computing $N \circ B_F$ (the average amount of data emitted by a storage device) for the Naive, Greedy, and Group-Greedy distribution functions on each of the sequences for a range of values of β.

5.5.2 Concave Sequences

The sequence used here is shown in Table 2, and the network footprint of the three evaluation strategies in Figure 8. As expected, Greedy partitioning is optimal for this sequence, and performs identically to Group-Greedy for all values of β. (Note that for concave sequences, Group-Greedy evaluation in fact reduces to Greedy.) Naive partitioning performs quite poorly; the shape of $N \circ B_F^N$ is always a straight line connecting the points corresponding to $\beta = 0$ and $\beta = 1$, and is guaranteed to lie above the graph of $N \circ B_F^G$ for concave sequences.

5.5.3 Non-Concave Sequences

Naive partitioning is network-optimal for the sequence listed in Table 3(a). This searchlet contains a color histogram detector trained on a black patch (the filter `pure_black`) and a face detector (`faces`); these are the only filters that perform discard. Both rely on the filter `RGB` to compute common RGB representations of the images (roughly equivalent in size to the images themselves). Additionally, `pure_black` uses another representation of the images computed by `HISTO_II`, while `faces` depends on the work performed by `INTEGRATE`; both `HISTO_II` and `INTEGRATE` add roughly three times the image size worth of metadata to images processed.

Figure 9(a) shows the performance of the three evaluation strategies on this sequence. We see that the first two filters roughly quadruple the size of the objects they process before any of those objects are discarded by the subsequent filters. (Note that this is precisely the reason why we did not use a SnapFind searchlet to illustrate the case of concave sequences: all SnapFind filters rely on `RGB`, making the first filter in any sequence decidedly unselective and the construction of a concave sequence impossible.) Because of this explosion of effective object size and the modest selectivities of `pure_black` and `faces`, evaluating the entire sequence atomically is more efficient than any other grouping. In practice, Naive partitioning can be nearly optimal for searchlets containing "helper" filters such as `RGB` that must be executed early in any ordering.

However, this is not the case with the three-filter sequence listed in Table 3(b). This sequence consists of a highly selective and comparatively inexpensive color histogram detector `bright_yellow`, and the considerably more expensive texture detector `grass`, both of which depend on the image representation computed by `RGB`. The optimal grouping determined by the Group-Greedy algorithm therefore separates `grass` from the atomically-executing pair `RGB` and `bright_yellow`. Neither the Greedy nor the Naive evaluation strategy can arrive at such a partitioning, and thus achieve sub-optimal results. As we can see in Figure 9(b), the Group-Greedy strategy outperforms both.

Table 2: **A Concave Sequence**

F_i	p_i	c_i	M_i
F0_01	0.4000	1.000	0.00
F1_01	0.5000	20.000	0.00
F2_01	0.7000	40.000	0.00
F3_01	0.9000	60.000	0.00

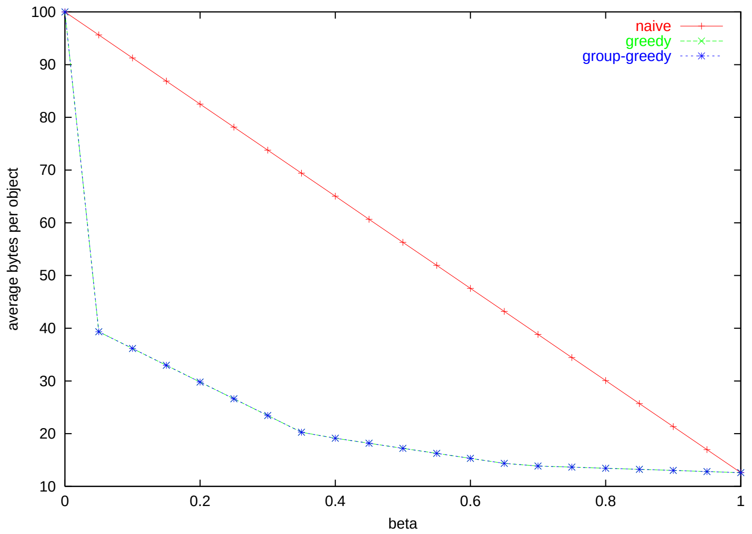


Figure 8: **A Concave Sequence.** The plot of bytes transmitted per object as a function of β for the Group-Greedy distribution function overlaps with that of Greedy for all values of β. This is expected, since the Greedy distribution function is optimal for concave sequences.

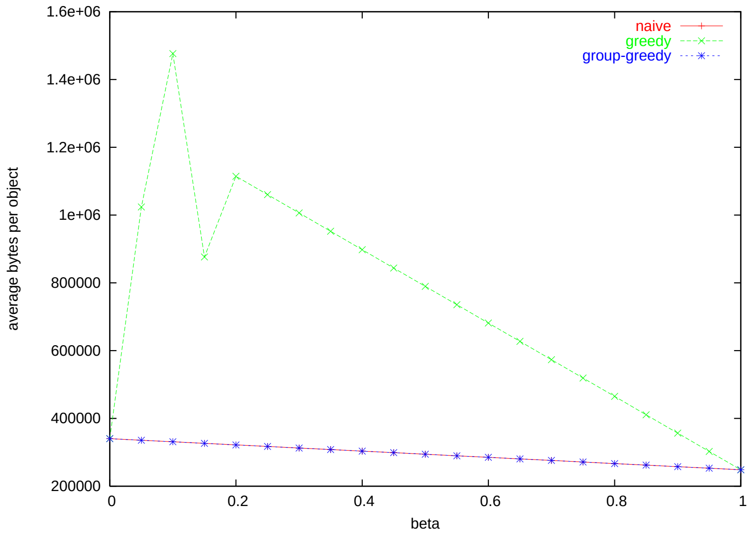
Table 3: **Non-Concave Sequences**

F_i	p_i	c_i	M_i
RGB	0.9948	3366.902	456544.83
HISTO_II	1.0000	12386.135	813215.17
pure_black	0.4531	2734.575	0.00
INTEGRATE	1.0000	13727.378	914023.46
faces	0.2184	251173.865	0.00
MERGE	1.0000	42.295	0.00

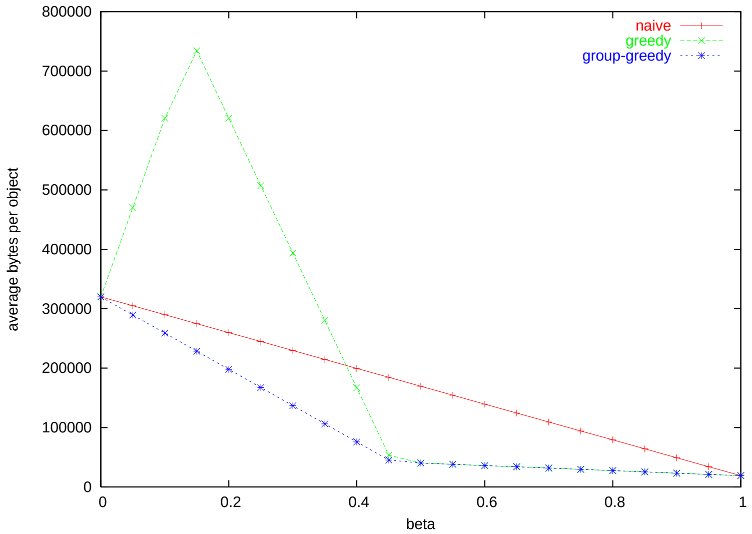
(a) A Naive-optimal sequence

F_i	p_i	c_i	M_i
RGB	0.9995	3042.961	430282.90
bright_yellow	0.0563	6640.882	0.00
grass	0.4495	206166.397	0.00

(b) A Naive- and Greedy-suboptimal sequence



(a) A Naive-optimal sequence



(b) A Naive- and Greedy-suboptimal sequence

Figure 9: **Non-Concave Sequences.** The Greedy distribution function is suboptimal for non-concave sequences. The sequence in (a) shows the case where Naive partitioning performs as well as Group-Greedy; this behavior is due to the vast amounts of meta-data generated by the first two filters in the sequence. Group-Greedy outperforms both Naive and Greedy partitioning on the sequence in (b), since only the nontrivial grouping of the first two filters yields optimal results.

6 Summary and Future Work

In this paper, we have explored the problem of network-sensitive partitioning of work in Diamond, an active-storage-based interactive search architecture. We have developed a performance model capturing the key run-time properties of Diamond query evaluation. Finally, we have presented a partitioning algorithm guaranteed to minimize the amount of data injected into the network during the execution of a search under CPU time constraints imposed on storage devices.

In the future, we plan to extend our model in support of heterogeneous storage device configurations, as well as multi-level distributed search hierarchies involving intermediate processing nodes. Additionally, we plan to investigate the impact of filter ordering on the network behavior of Diamond systems; so far, we had been assuming a fixed ordering optimized for CPU time. A related challenge is that of dynamically distinguishing bottlenecks in the search pipeline—specifically, the ability to identify network-bound searches. When the network is determined to be the bottleneck, the Diamond run-time could trade CPU time for bandwidth by discarding computed state, or adjusting the filter ordering policy appropriately.

References

[1] Amiri, K., Petrou, D., Ganger, G. R., and Gibson, G. A. Dynamic function placement for data-intensive cluster computing. In *2000 USENIX Annual Technical Conference: San Diego, CA, USA, June 18–23, 2000* (Berkeley, CA, USA, 2000), USENIX, Ed., USENIX, pp. 307–322.

[2] Arpaci-Dusseau, R. H., Anderson, E., Treuhaft, N., Culler, D. E., Hellerstein, J. M., Patterson, D., and Yelick, K. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems* (Atlanta, GA, May 1999), ACM Press, pp. 10–22.

[3] Hunt, G. C., and Scott, M. L. The coign automatic distributed partitioning system. In *Proceedings of the 3rd Symposium on Operating Systans Design and Implementation (OSDI-99)* (Berkeley, CA, Feb. 22–25 1999), Usenix Association, pp. 187–200.

[4] 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. In *Proceedings of USENIX File and Storage Technologies* (2004).