Bulletin of the Technical Committee on

Data Engineering

December, 1995 Vol. 18 No. 4

IEEE Computer Society

Letters

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems.

Letter from the Editor-in-Chief

About this issue

To a very large extent, the problems of commercial data processing have been solved by relation database systems. But there is a vast array of non-formatted data that has yet to be captured within systems that are capable of high speed storage and retrieval. Even text data, which has a longer history than relational databases, is mostly not captured by database systems and its retrieval and update pose significant problems.

This issue focuses on video (or continuous media) data, how to organize it on disk, how to deliver it to clients, how to index it, etc. So called ”video on demand” has received much press coverage. This issue explains some of the underlying technology that makes it possible. Shahram Ghandeharizadeh has assembled four articles for your reading pleasure, describing several aspects of the technology. I am sure you will find this of real interest, and I thank Shahram for pulling it together for us.

State of the Bulletin

Once again, through the generosity of the IEEE Computer Society, we are distributing another issue of the Data Engineering Bulletin. And once again, we do not know whether this will continue in the next year, now 1996. I will keep you posted on the funding status via emailings. I hope we can avoid some of the confusion that surrounded this year’s funding.

We are in the process of setting up a home page to make the Bulletin available on the web. I had hoped that I could report in this issue that the site is up and running. Unfortunately, it is still under construction as I have been distracted by other events. My revised plan is to make the site usable as a source for issues of the Bulletin by late January, 1996. I will send an announcement to the membership of the TC when the web site comes on line.

We continue to make the Bulletin available electronically via the Microsoft FTP server. The procedure for this is to log on to

```
ftp.research.microsoft.com
```

as ”anonymous” and give as your password your email address. Change directories as follows-

```
cd  pub
cd debull
```

You can then do a ”dir” or ”ls” to determine what files are present. Read the README file in the debull directory for more complete information.

Having selected the issue you wish, and the format you desire, select the appropriate file with the ftp ”get” command, e.g.

```
get sept95-letfinal.ps
```

to have the file delivered to your system. The files are all in postscript so you will need a postscript viewer and/or printer to be able to read them. *Note that the files now come with the file type “.ps”*.

David Lomet
Editor-in-Chief

Letter from the Special Issue Editor

If "object-oriented" was the buzz word of the 1980s, "multimedia" is almost certainly emerging as the catch phrase of the 1990s. A multimedia information system manages complex data types such as images, audio and video clips, 3-D objects, structured presentations consisting of 3-D objects (e.g., animation), etc. One does not have to search long to find tens of books on a variety of topics related to multimedia. While one book might focus on how to generate a multimedia document using a specific software package, another might focus on the concept of fair-use and its complexities with the introduction of multimedia documents. Why is there so much excitement about multimedia? From the users' perspective, the excitement is due to the ability of these systems to convey information naturally by exercising our audio/visual senses (a picture says a million words). Moreover, they provide tools that enable a user to generate valuable data with ease. From a technical perspective, these systems are worth investigating because their requirements push the current technology (both hardware and software) to its limits. The focus of this issue is on techniques to realize a multimedia information system.

A multimedia information system should meet several criteria. First, a user requesting the display of either an image or a video clip should not be forced wait for a long time. Ideally, the wait time should be a fraction of a second. Second, the data is valuable and many users might want to access it simultaneously. Due to its value, many complex social, cultural and legal issues arise with the use of data, e.g., copy-right, fair-use, etc. The ignorance of these topics by this special issue should not be interpreted as their lack of importance. To the contrary, the technical researchers can play a pivotal role in this arena. Third, the volume of available data is already overwhelming and is continuing to increase. While browsing techniques are effective, query processing techniques are needed to enable a user to pose queries to locate relevant information. Fourth, some of the data types, namely the continuous media data type (audio, video), require their data to be delivered at a pre-specified bandwidth in support of their display. If the system does not render the data at the pre-specified rate, the user might observe disruptions and delays. With audio, for example, delays would turn a music clip to a collection of random noises (the impact of delays on video is less disruptive). Avoiding such delays is a challenging task because (1) continuous media data types are large in size, e.g., a two hour movie might be several gigabytes in size, and (2) resources (e.g., memory, disk bandwidth) must be scheduled intelligently. And finally, the design of a system should be both economically viable and *scalable*. By scalable, we mean that the system should be able to grow as the performance requirements of an application changes. To illustrate, assume a data provider that supports 100 simultaneous users accessing a terabyte of data with an expected delay of 0.1 second. If the demand on this system grows such that the vendor desires to support 1000 simultaneous users, the vendor should be able to increase the amount of available resources proportionally to meet the new demand.

The focus of this special issue is on the design of a multimedia storage manager that realizes the listed requirements. While the first three papers of this issue focus on techniques to support continuous display of video objects, the fourth paper by Christos Faloutsos describes a general purpose technique to search multimedia documents by content. The first paper by Ozden, Rastogi and Silberschatz analyzes striping as a technique to realize a scalable server. It explains the storage requirements of video objects to motivate the use of magnetic disks for realizing an economical system. The second paper by Golubchik, Lui and Muntz describes techniques to share the retrieval of either an audio or a video clip (termed a stream) from disks among multiple displays in order to increase the number of simultaneous users serviced. The third paper by David Andersen focuses on the VCR functionalities and how they can be realized using MPEG encoding technique.

While this issue focuses on stream-based video, structured video [1] is starting to emerge as an important research direction. With structured video, a video clip consists of a sequence of scenes. Each scene consists of a collection of background objects, actors (e.g., 3 dimensional representation of Mickey Mouse, dinosaurs, lions), light sources that define shading, and the audience's view point. Spatial constructs are used to place objects that

[1] S. Ghandeharizadeh, Stream-based Versus Structured Video Objects: Issues, Solutions, and Challenges. In *Multimedia Database Systems: Issues and Research Directions*, Editors: S. Jajodia and V.S. Subrahmanian, Springer Verlag, 1995

constitute a scene in a rendering space while temporal constructs describe how the objects and their relationships evolve as a function of time. The structured video provides adequate information to support both effective query processing techniques and re-usability of information. Structured video raises a host of research topics worth investigating: How is the data presented at a physical level? How should the system represent temporal and spatial constructs to enable a user to author complex relationships (e.g., chasing, hitting)? What are the specifications of a query language that interrogates the temporal and spatial relationships between objects? What techniques should be employed to execute queries? What indexing techniques can be designed to speedup the retrieval time of a query? How should a system employ content-based retrieval techniques proposed for stream-based video to construct its structured presentation? These novel research directions require further investigation.

I want to conclude by thanking the authors for both contributing quality articles to this issue and serving as reviewers for each others manuscript. In addition, I want to thank David Lomet for providing me with the opportunity to put together this special issue and generating the final draft of this issue using Latex.

Shahram Ghandeharizadeh
Computer Science Department
University of Southern California
Los Angeles, CA 90089

Disk Striping in Video Server Environments

Banu Özden Rajeev Rastogi Avi Silberschatz

AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974

Abstract

A growing number of applications need access to video data stored in digital form on secondary storage devices (e.g., video-on-demand, multimedia messaging). As a result, video servers that are responsible for the storage and retrieval, at fixed rates, of hundreds of videos from disks are becoming increasingly important. Since video data tends to be voluminous, several disks are usually used in order to store the videos. A challenge is to devise schemes for the storage and retrieval of videos that distribute the workload evenly across disks, reduce the cost of the server and at the same time, provide good response times to client requests for video data.

In this paper, we present schemes that are based on striping videos (fine-grained as well as coarse-grained) across disks in order to effectively utilize disk bandwidth. For the schemes, we show how an optimal-cost server architecture can be determined if data for a certain pre-specified number of videos is to be concurrently retrieved. The schemes provide good response times in environments in which the number of concurrent client requests is not much larger than the number of videos that can be concurrently retrieved by the server.

1 Introduction

In recent years, we have witnessed significant advances in both networking technology, and technologies involving the digitization and compression of video. It is now possible to transmit several gigabits of data per second over optic fiber networks. Also, with compression standards like MPEG-1, the bandwidth required for the transmission of a video is as low as 1.5 Mbps. These advances have resulted in a host of new applications involving the transmission of video data over networks. Examples include video-on-demand, on-line tutorials, training and lectures, multi-media messaging, interactive TV, games, etc.

One of the key components in the above applications is the *video server* which is responsible for the storage and transmission of videos. Depending on the application, a video server may be required to store hundreds of videos, and may be required to concurrently transmit to clients, data for a few hundred to a few thousand videos. Furthermore, the data for every video must be transmitted at a fixed rate depending on the compression technique employed (for MPEG-1, the rate is about 1.5 Mbps).

Due to the voluminous nature of video data (a 100 minute long MPEG-1 video requires approximately 1.125 GB of storage space), and the high cost ($40/MB) of *random access memory* (RAM), storing videos in RAM is prohibitively expensive. A less expensive way of storing videos is to utilize disks, which are much cheaper than RAM (storage on disks costs less than $1/MB). Thus, a video server would be more cost-effective if it relied on disks instead of RAM for the storage of videos. However, modern disks have limited storage capacity (1 GB-9 GB) and relatively low transfer rates (30 Mbps - 60 Mbps). As a result, in order to be able to store hundreds of videos as well as support the retrieval of hundreds of videos concurrently, a video server would need to store the videos on several disks. Schemes for laying out the videos on multiple disks are crucial to distributing the load uniformly across the various disks and thus, utilizing the disk bandwidth effectively. For example, certain videos may be more popular than others, and a naive approach that stores every video on an arbitrarily chosen disk could result in certain disks being over-burdened with more requests than they can support, while other disks remain idle.

Another characteristic of disks is that they have a relatively high latency for data access (typically between 10-20 ms). As a result, if very little data is retrieved during each disk access, then the effective transfer rate of the disk is much lower, and thus the number of videos that can be concurrently retrieved is much less. On the other hand, retrieving large amounts of data during each disk access reduces the impact of disk latency and increases the throughput of the disk. However, beyond a certain point, increasing the amount of data retrieved is not cost-effective. The reason for this is that in order to buffer the large amounts of data retrieved, a large amount of RAM is required, and this results in an increase in the cost of the server. What is needed, is an effective method for computing the optimal amount of data to be retrieved during a disk access that reduces the cost of retrieving data per video. Thus, the challenge is to design a low-cost video server that can transmit several videos at the required rate concurrently, utilize the disk bandwidth effectively, and provide low response times to requests for videos.

In this paper, we present schemes based on fine-grained as well as coarse-grained striping in order to distribute the workload evenly across disks. For each of the schemes, we compute the optimal amount of data to be retrieved during each disk access that maximizes the number of videos that can be concurrently transmitted. We show that the schemes based on coarse-grained striping can concurrently retrieve data for more videos than those based on fine-grained striping. For the schemes, we determine the optimal-cost server architecture that supports the concurrent retrieval of data for a certain pre-specified number of videos. For coarse-grained striping, we present a starvation-free scheme that minimizes the wasted disk bandwidth when servicing client requests.

Due to space constraints, we do not present the proofs of our theorems; these can, however, be found in [3].

2 System Model

The *video-server* is a computer system containing one or more processors, RAM, and multiple disks to store videos. Videos are stored on disks in compressed form and need to be displayed at a certain rate, denoted by r_{disp}. The server is connected to clients via a high speed network over which videos are transmitted at the rate r_{disp}. We denote by l_i, the length (in bits) of a video V_i.

Clients make requests for videos to the server which maintains a *request list*. Each of the entries in the request list contains an identifier for the requested video (e.g., name) and the set of clients who have requested the video. Requests are added to the request list as follows. If an entry for the requested video is already contained in the request list, then the client is simply added to the set of clients in the entry. If, on the other hand, an entry for the requested video is not contained in the request list, then a new entry containing the video and the client is appended to the end of the request list.

In general, it is not possible to service client requests immediately upon arrival since videos reside predominantly on disk, and they need to be retrieved into RAM before being transmitted to a client. In this paper, we present schemes that the server employs in order to determine the set of videos to be concurrently retrieved from disk at any given time. For each of the videos being retrieved concurrently, the server allocates a certain amount of buffer space in RAM, and retrieves portions of the video into the buffer at certain time intervals such that the entire video is retrieved at a rate r_{disp}. The size of the portions retrieved depends on the scheme; thus, the server transmits videos to clients without loading the entire video into RAM. To start a video, a certain initial amount of the video data must be retrieved into RAM. Once this is accomplished, the transmission of the video to all the clients with outstanding requests for the video can commence. We refer to the continuous transfer of a video

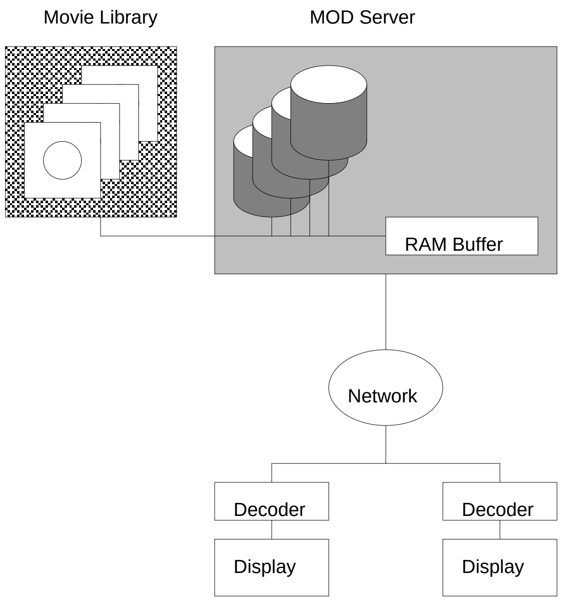

Figure 1: System architecture for VOD services.

Inner track transfer rate	r_{disk}	45 Mbps
Settle time	t_{settle}	0.6 ms
Seek latency (worst-case)	t_{seek}	17 ms
Rotational latency (worst-case)	t_{rot}	8.34 ms
Total latency (worst-case)	t_{lat}	25.5 ms
Cost	C_d	\$1500
Capacity		2 GB

Figure 2: Disk Parameters for a Commercially Available Disk

from disk to RAM at rate r_{disp}, as a *stream*.

For the development of the schemes, it is important to understand the characteristics of disks, which we now describe. Data on disks is stored in a series of concentric circles, or *tracks*, and accessed using a disk head. Disks rotate on a central spindle and the speed of rotation determines the transfer rate of disks. Data on a particular track is accessed by positioning the head on (also referred to as *seeking* to) the track containing the data, and then waiting until the disk rotates enough so that the head is positioned directly above the data. Seeks typically consist of a coast during which the head moves at a constant speed and a settle, when the head position is adjusted to the desired track. Thus, the latency for accessing data on disk is the sum of seek and rotational latency. In the table of Figure 2, we present the notation and values employed in the paper for the various disk characteristics (the values are for a commercially available disk). In this paper, we assume that $r_{disp} < r_{disk}$, for the videos.

As mentioned earlier, a video server that stores hundreds of videos would require a large number of disks to hold the video data. A naive storage scheme in which an entire video is stored on a single disk could result in an ineffective utilization of disk bandwidth. The reason for this is that not all videos are accessed with the same frequency; clients may request to view certain videos more often than others, thus causing certain disks to be busy while others, with less frequently requested videos, remain idle. Also, unless a video is replicated on several disks, the number of concurrent streams of the video that can be supported is bounded above by the bandwidth of the disk storing the video.

In this paper, we store videos using *disk striping*, a popular technique in which consecutive logical data units (referred to as *stripe units*) are distributed among the disks in a round-robin fashion [1,4,5]. Disk striping, in addition to distributing the workload uniformly across disks, also enables multiple concurrent streams of a video to be supported without having to replicate the video.

3 Related Work

A number of schemes for retrieving video data continuously from disks have been proposed in the literature. Most of them, however, only consider the problem of retrieving videos from a single disk, and do not address the issue of evenly distributing the workload in the case that multiple disks are employed to store the videos.

Among striping-based schemes, *staggered striping* was proposed in [1] to utilize disk bandwidth effectively for both high bandwidth (that is, those for which $r_{disp} \geq r_{disk}$) as well as low bandwidth (those for which $r_{disp} < r_{disk}$) videos with different rate requirements. For the high bandwidth videos, each video object is divided into subobjects and each subobject is further declustered across $\lceil \frac{r_{disp}}{r_{disk}} \rceil$ disks, from which it is retrieved in parallel. The data for a subobject on a disk is referred to as a *fragment*, and the distance between disks containing the first fragment for consecutive subobjects is referred to as the *stride*. The authors show that by choosing the stride to be relatively prime with respect to the number of disks, the workload can be uniformly distributed across the disks. The authors also show that a fragment size of about 2 cylinders wastes only 10% of the bandwidth, and at the same time provides a reasonable display latency time.

The striping schemes and results presented in [4,5] are more general, and are not specifically tailored for video server environments in which workloads typically comprise of accesses (to large files) that are sequential in nature, and have timing requirements.

4 Fine-grained Striping

In fine-grained striping, the striping unit is typically a bit, a byte or a sector [4]. Thus, if there are m disks, then every retrieval involves all the m disk heads, and the m disks behave like a single disk with bandwidth $m \cdot r_{disk}$. This striping technique is used in the RAID-3 data distribution scheme [5]. In the following subsections, we describe a scheme for retrieving data for multiple concurrent video streams in case fine-grained striping is used to store videos.

4.1 Servicing Requests

In the fine-grained striping scheme, the server maintains a *service list* that is different from the request list, and contains the videos for which data is being retrieved. The server retrieves data for the videos in the service list in *rounds*, the number of bits retrieved for a video during a round being d. The d bits retrieved for a video during a round follow the d bits retrieved for the video during the previous round. The value of d is computed based on the amount of RAM, D, and the number of disks, m, such that the number of concurrent streams that can be supported is maximized. In addition, it is a multiple of $m \cdot su$, where su is the stripe unit size (we describe the method used for computing d at the end of this subsection). A video can thus be viewed as a sequence of portions of size d; each portion can further be viewed as a sequence of sub-portions, each of size $m \cdot su$. Each sub-portion is striped across the m disks; the stripe units contained in a sub-portion are at the same position on their respective disks. In addition, stripe units on a disk, belonging to consecutive sub-portions of a portion, are stored contiguously on the disk one after another.

At the start of a round, the server sorts the videos in the service list based on the positions of the tracks on disk, of the d bits to be retrieved for each video. It then retrieves d bits from disk for each of the videos in the order of their appearance in the service list. Note that sorting the videos ensures that the disk heads move in a single direction when servicing videos during a round. As a result, random seeks to arbitrary locations are eliminated (this is similar to the known C-SCAN disk scheduling algorithm [6]). Since the time to transmit d bits of video is $\frac{d}{r_{disp}}$, in order to ensure that data for every video is retrieved at rate r_{disp}, we require every round to begin $\frac{d}{r_{disp}}$ after the previous round. As a result, we require the time to service videos during a round to never exceed $\frac{d}{r_{disp}}$, the duration of a round. Since, during a round, disk heads travel across the disk at most twice, and retrieving data for each video, in the worst case, incurs a settle and a worst-case rotational latency overhead, we require the following equation to hold (assuming that $V_1, V_2, \ldots, V_q$ are the videos in the service list):

$$2 \cdot t_{seek} + q \cdot (\frac{d}{m \cdot r_{disk}} + t_{rot} + t_{settle}) \leq \frac{d}{r_{disp}} \tag{1}$$

At the start of a round, before the server sorts the videos in the service list, it adds videos at the head of the request list to the service list as long as Equation 1 holds for videos in the service list after each of the videos is added (videos that are added to the service list are deleted from the head of the request list). For every video added to the service list, a buffer of size $2 \cdot d$ is allocated. The d bits for the video are retrieved into the buffer during successive rounds. Transmission of a video's bits to clients is begun only at the end of the round in which the first d bits of the video have been retrieved into it's buffer. The reason for this is that since new videos may be added to the service list, and videos are sorted at the start of each round, the times (relative to the start of a round) at which the d bits for a video are retrieved, in two consecutive rounds, may not be the same. Thus, by ensuring that a video's buffer contains at least d bits at the start of each round, we ensure that bits for the video can be transmitted at a rate r_{disp} irrespective of when during the round the next d bits are retrieved for the video. Finally, a video is deleted from the service list at the end of the round in which all the data for it has been retrieved. The above approach to retrieving data for videos is similar to the *disk multitasking* scheme proposed in [2] and the *streaming* RAID scheme proposed in [7].

We now address the issue of computing a value for d based on the amount of RAM, D, and the number of disks, m, such that the number of concurrent streams that can be supported is maximized. From Equation 1, it follows that, for a given value of d, the maximum number of streams that can be supported is $\frac{\frac{d}{r_{disp}} - 2 \cdot t_{seek}}{\frac{d}{m \cdot r_{disk}} + t_{rot} + t_{settle}}$. Thus, since $r_{disp} < m \cdot r_{disk}$, as the value of d increases, the number of streams that can be supported also increases. However, since the buffer per stream is $2 \cdot d$ and the total buffer requirements must not exceed D, increasing d beyond a certain value results in a decrease in the number of streams. Thus, the value of d that supports the maximum number of streams, can be obtained by solving the following equation:

$$\frac{2 \cdot d \cdot (\frac{d}{r_{disp}} - 2 \cdot t_{seek})}{\frac{d}{m \cdot r_{disk}} + t_{rot} + t_{settle}} = D$$

Let d_{calc} be the maximum value for d due to solving the above equation. Let $q_{calc} = \frac{\frac{d_{calc}}{r_{disp}} - 2 \cdot t_{seek}}{\frac{d_{calc}}{m \cdot r_{disk}} + t_{rot} + t_{settle}}$ be the maximum number of streams that can be supported with $d = d_{calc}$. Note that $2 \cdot d_{calc} \cdot q_{calc} = D$. The problem is that d_{calc} may not be a multiple of $m \cdot su$. For all values of d greater than d_{calc}, the maximum number of streams is $\frac{D}{2 \cdot d}$; on the other hand, for values of d less than d_{calc}, the maximum number of streams is $\frac{\frac{d}{r_{disp}} - 2 \cdot t_{seek}}{\frac{d}{m \cdot r_{disk}} + t_{rot} + t_{settle}}$. Thus, the optimal value for d is either $\lceil \frac{d_{calc}}{m \cdot su} \rceil \cdot m \cdot su$ or $\lfloor \frac{d_{calc}}{m \cdot su} \rfloor \cdot m \cdot su$ depending on which of the following two is greater: (1) $\frac{D}{2 \cdot \lceil \frac{d_{calc}}{m \cdot su} \rceil \cdot m \cdot su}$ or (2) $\frac{\frac{d}{r_{disp}} - 2 \cdot t_{seek}}{\frac{d}{m \cdot r_{disk}} + t_{rot} + t_{settle}}$ with $d = \lfloor \frac{d_{calc}}{m \cdot su} \rfloor \cdot m \cdot su$.

Thus, if $D = 2$ Gb and $m = 50$, then the optimal value for d is 7.1 Mb, and the maximum number of MPEG-1 streams that can be supported is 139 (the values for disk parameters are those presented in Figure 2 and su is

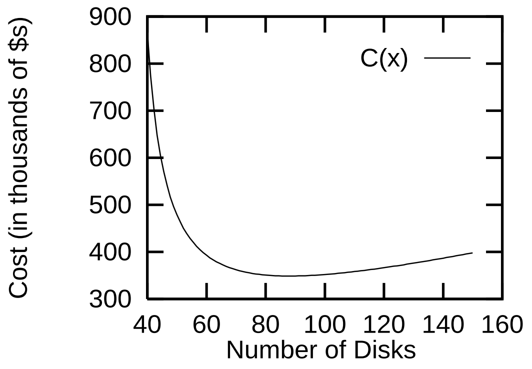

Figure 3: Cost of Server as a function of Number of Disks

chosen to be 512 bytes). For simplicity, in the schemes presented in this section and later in the paper, video streams are begun only based on the availability of disk bandwidth. As a result, we select d to be $\lfloor \frac{d_{calc}}{m \cdot su} \rfloor \cdot m \cdot su$. The schemes can be extended to take into account even the availability of buffer space, in which case, an optimal value for d can be selected as described above. However, we do not discuss these extensions in this paper.

4.2 Designing an Optimal-cost Video Server

Ideally, the architecture of a video server must be such that it can support a certain pre-specified number of concurrent streams and its cost is as low as possible. In the following, we compute the amount of RAM, D, and the number of disks, m, that a video server must be configured with if it is to support Q concurrent streams, and its cost is to be optimal.

Let C_r denote the cost of RAM per Mb and C_d be the cost of a single disk. For a given value of m, the minimum amount of RAM required to support Q streams is $2 \cdot Q \cdot d$, where $d = \lceil \frac{\hat{d}}{m \cdot su} \rceil \cdot m \cdot su$ and

$$\hat{d} = \frac{(Q \cdot (t_{rot} + t_{settle}) + 2 \cdot t_{seek}) \cdot m \cdot r_{disk} \cdot r_{disp}}{(m \cdot r_{disk} - Q \cdot r_{disp})}$$

is the minimum value of d required to support Q streams, obtained as a result of solving Equation 1. Thus, $D = 2 \cdot Q \cdot d$, and $C(m)$, the minimum cost of the server for a given m, is

$$C(m) = C_r \cdot 2 \cdot Q \cdot d + C_d \cdot m$$

Substituting for d in the above equation, we obtain the following value for $C(m)$.

$$C(m) = C_r \cdot 2 \cdot Q \cdot \lceil \frac{(Q \cdot (t_{rot}+t_{settle})+2 \cdot t_{seek}) \cdot r_{disk} \cdot r_{disp}}{su \cdot (m \cdot r_{disk} - Q \cdot r_{disp})} \rceil \cdot m \cdot su + C_d \cdot m$$

From the above cost equation, it follows that it may not be cost-effective to architect the server with the minimum number of disks required to support Q streams (that is, m is such that $m \cdot r_{disk} > Q \cdot r_{disp} \geq (m-1) \cdot r_{disk}$). The reason for this is that the size of the buffer per stream, $2 \cdot d$, is inversely proportional to $r_{disk} - \frac{Q \cdot r_{disp}}{m}$. Thus, at $m = \frac{Q \cdot r_{disp}}{r_{disk}}$, the buffer size is infinity, and subsequent increases in the number of disks, m, cause the size of the buffer per stream to decrease; the decrease in the buffer size is larger for smaller values of m. Thus, since RAM is more expensive than disks, the reduction in the cost of the buffer for the streams due to increasing the number of disks beyond $\frac{Q \cdot r_{disp}}{r_{disk}}$, could more than offset the cost of the additional disks. In Figure 3, we show

how the cost of the server, $C(m)$, that supports 1000 MPEG-1 streams varies with the number of disks, m (the values for disk parameters are those presented in Figure 2, C_r is \$5/Mb, and su is 512 bytes). As the number of disks increases, initially the cost decreases due to the large decreases in buffer sizes. However, beyond a certain number of disks, the cost of an additional disk exceeds the decrease in the cost of RAM as a result of the decrease in buffer sizes, and thus the cost of the server increases.

Our goal is to compute the value of m for which $C(m)$ is minimum, subject to the constraint that $m > \frac{Q \cdot r_{disp}}{r_{disk}}$. We begin by computing the value of m for which $C(m)$ without the $\lceil\,\rceil$s, which we denote by $\hat{C}(m)$, is minimum. Solving $\frac{d}{dm}(\hat{C}(m)) = 0$, we obtain the following value for m, which we denote by m_{calc}, that satisfies the constraint and at which $\hat{C}(m)$ is minimum.

$$\frac{Q \cdot r_{disp}}{r_{disk}} + \sqrt{\frac{2 \cdot Q^3 \cdot C_r \cdot r_{disp}^2 \cdot (t_{rot} + t_{settle}) + 2 \cdot t_{seek}}{C_d \cdot r_{disk}}}$$

Since $\frac{d}{dm}(\frac{d}{dm}(\hat{C}(m))) > 0$ at $m = m_{calc}$, it follows that for increasing values of m beyond m_{calc} and decreasing values of m below m_{calc} (and above $\frac{Q \cdot r_{disp}}{r_{disk}}$), the value of $\hat{C}(m)$ increases. The optimal value of m for which $C(m)$ is minimum can now be computed as follows.

We refer to values of m for which $\hat{C}(m) = C(m)$ as breakpoints. At increasing breakpoints beyond m_{calc}, the value of $C(m)$ increases. Furthermore, in between two successive breakpoints beyond m_{calc}, the value of $C(m)$ increases linearly with m. As a result, the optimal value of m is bounded above by $\lceil m_{calc} \rceil$. Furthermore, since for a given value of m, $C(m)$ is always greater than or equal to $\hat{C}(m)$, the lower bound for the optimal value of m can be obtained as follows. Let $C_{\lceil calc \rceil}$ be the value of $C(m)$ at $m = \lceil m_{calc} \rceil$. By solving $\hat{C}(m) = C_{\lceil calc \rceil}$, two values of m for which $\hat{C}(m) = C_{\lceil calc \rceil}$ are obtained. One of these is to the right of m_{calc}, and the other is to the left of m_{calc}, which we denote by m_{left}. Since $\hat{C}(m)$ increases to the left of m_{left} and $C(m)$ is at least as high as $\hat{C}(m)$, the optimal value of m is an integer in the interval between m_{left} and $\lceil m_{calc} \rceil$ for which $C(m)$ is minimum.

The optimal value of m for a server that supports 1000 MPEG-1 streams is 88. The amount of RAM required is 43.3 Gb and the cost of the the server is \$348,689. Also, the value for d is 21.6 Mb, and the duration of a round is 14.4 s.

5 Coarse-grained Striping

In coarse-grained striping, the size of stripe units is much larger; it is the amount of data typically retrieved during contrast to fine-grained striping in which all disk heads are involved in data retrieval during each access, in coarse-grained striping, usually, only a single disk is involved. For large requests and for sequential accesses to data, coarse-grained striping distributes the workload evenly among the various disks. This striping technique is used in the RAID-5 data distribution scheme [5]. In the following subsections, we present a scheme for retrieving data for multiple concurrent video streams in case coarse-grained striping is used to store videos.

5.1 Servicing Requests

In the coarse-grained striping scheme, the size of a stripe unit is d, the number of bits retrieved for a video during each round (we show how d is computed in the next subsection). Every video is assumed to have a length that is a multiple of d[1]. The various videos are first concatenated to form a *super-video*. Consecutive stripe units of size d belonging to the *super-video* are then stored on the m disks in a round-robin fashion. We denote the disk on which the first stripe unit of a video V_i is stored by $disk(V_i)$. Note that, with the coarse-grained striping

[1]This can be achieved by appending advertisements or padding videos at the end.

scheme, for two different videos V_i and V_j, $disk(V_i)$ may be different from $disk(V_j)$. We now describe schemes for servicing client requests when videos are laid out using coarse-grained striping.

In contrast to the fine-grained striping scheme, in the coarse-grained striping scheme, a separate service list is maintained for every disk. The service list for a disk contains the videos for which data is being retrieved from the disk during the current round. The data for a video is retrieved into a buffer of size $2 \cdot d$ for the video. As before, at the beginning of a round, the videos in each service list are sorted based on the positions of the tracks on disk, of the d bits to be retrieved for each of the videos during the round. Following this, the d bits for videos in the service list for every disk are retrieved from the disk in the sorted order. Furthermore, data for videos in different service lists is retrieved in parallel. As a result, in order to retrieve data at a rate r_{disp}, since rounds must begin at intervals of $\frac{d}{r_{disp}}$, for every service list (let $V_1, \ldots, V_q$ be the videos in the service list), we require the following equation to hold:

$$q \cdot (\frac{d}{r_{disk}} + t_{rot} + t_{settle}) + 2 \cdot t_{seek} \leq \frac{d}{r_{disp}} \qquad (2)$$

At the end of a round, videos for which no more data needs to be retrieved are deleted from the service lists. Following this, the service list for every disk is set to the service list of the disk preceding it – the data to be retrieved for a video in the service list being the d bits in the video that follow the d bits retrieved for the video in the previous round from the preceding disk. Thus, data for a video during successive rounds is retrieved from successive disks. Also, at the end of the round in which the first d bits have been retrieved for a video, transmission of data to clients that requested it is begun. The above approach to storing and retrieving data is similar to the staggered striping scheme for low bandwidth objects, proposed in [1].

We now present schemes that address the issue of when to begin retrieving data for videos in the request list or alternately, when to insert videos contained in the request list into the service list for a disk. All of the schemes are employed by the server just before the service lists are sorted at the beginning of a round, and after the service list for every disk is set to that of its preceding disk, at the end of the previous round. Note that if q is the number of videos serviced at a disk during a round, then the *unutilized* (also referred to as *wasted* or *available*) bandwidth on the disk, during the round, is at least $\frac{d}{r_{disp}} - q \cdot (\frac{d}{r_{disk}} + t_{rot} + t_{settle}) - 2 \cdot t_{seek}$.

The simplest scheme is to perform the following action for every request in the request list, beginning with the first request.

> If for the requested video V_i and the videos contained in the service list for $disk(V_i)$, Equation 2 holds, then the request is deleted from the request list and added to the service list for $disk(V_i)$.

The problem with the scheme is that, even though it utilizes the disk bandwidth fairly well, certain requests in the request list may never be serviced; thus causing them to wait forever.

Example 1: Consider a server with 100 disks numbered 0 through 99. Consider videos $V_1, \ldots, V_{999}$, each of which begins at disk 0 and terminates at disk 99, and video V_{1000} that begins at disk 50. Now, suppose a request for video V_{1000} arrives at a time when the bandwidth of all the disks is being used to retrieve data for videos in $\{V_1, \ldots, V_{999}\}$. Furthermore, suppose that additional requests for videos in $\{V_1, \ldots, V_{999}\}$ keep arriving at the server. Every time data retrieval for a video in $\{V_1, \ldots, V_{999}\}$ completes, bandwidth is available on disk 0. Thus, since $disk(V_{1000}) = 50$ while videos in $\{V_1, \ldots, V_{999}\}$ begin at disk 0, the above scheme would allocate the available bandwidth to one of the videos in $\{V_1, \ldots, V_{999}\}$, for which a request is contained in the request list. □

A variant of the above scheme that adds a video to a service list only after videos preceding it in the request list have been added to service lists, eliminates the above problem. However, a problem with the approach is that

1. For every reserved list and free list entry e, $e(1)$ is set to $(e(1)+1) \bmod m$; (thus, the disk in each entry is set to the disk following it).

2. For every video for which the last d bits were retrieved in the previous round from disk i, we add the bandwidth available on disk $(i+1) \bmod m$ to the free list as follows. Two variables, cur_entry and cur_disk are maintained; cur_entry is initially set to the first entry in the reserved list and cur_disk is set to $(i+1) \bmod m$.

   (a) If cur_entry is nil, then add $\frac{d}{r_{disk}} + t_{rot} + t_{settle}$ to the available bandwidth of the free list entry for cur_disk, and go to Step 3.

   (b) If $dist(\text{cur\_disk}, disk(\text{cur\_entry}(2)))$ is less than $dist(\text{cur\_entry}(1), disk(\text{cur\_entry}(2)))$, then the values of $\text{cur\_entry}(1)$ and cur_disk are swapped.

   (c) cur_entry is set to the next entry in the reserved list. Go to Step 2(a).

3. Once all the available disk bandwidth has been added to the free list, it is then reserved for requests in the request list as follows.

   (a) If the request list is empty or for every disk, the available bandwidth in the free list entry for the disk is less than $\frac{d}{r_{disk}} + t_{rot} + t_{settle}$, go to Step 4. Else, set cur_video to the video contained in the first request in the request list, and delete the request from the request list.

   (b) Let disk j be the disk for which available bandwidth in the free list is at least $\frac{d}{r_{disk}} + t_{rot} + t_{settle}$, and $dist(j, disk(\text{cur\_video}))$ is minimum. Append $(j, \text{cur\_video})$ to the end of the reserved list.

   (c) Decrement $\frac{d}{r_{disk}} + t_{rot} + t_{settle}$ from the available bandwidth for disk j in the free list. Go to Step 3(a).

4. For all entries e in the reserved list such that $e(1) = disk(e(2))$, delete e from the reserved list and insert $e(2)$ into the service list for $e(1)$ (the data to be retrieved for $e(2)$ is the first d bits).

Figure 4: A Starvation-free Scheme

it could result in disk bandwidth being unnecessarily wasted. In the server of Example 1, suppose that at the start of a round, there are outstanding requests for videos V_1 and V_{1000} (the one for V_1 preceding the one for V_{1000} in the request list), and bandwidth is available only on disks 50 and 98. With the modified scheme, since V_1 can be added to the service list of disk 0 only when bandwidth becomes available on disk 0 (after two rounds), V_{1000} cannot be added to the service list for disk 50 even though bandwidth is available on it. Thus, the bandwidth on disk 50 is wasted.

In the following, we present a scheme that prevents requests from starving and at the same time, utilizes disk bandwidth effectively. In order to prevent starvation, the scheme reserves available disk bandwidth for videos based on the order in which they appear in the request list. The assignment of reserved disk bandwidth to videos is such that the amount of disk bandwidth that is wasted is minimized. The scheme requires two additional data structures to be maintained. One of the data structures is a list of (disk,video) ordered pairs which stores information about the disk on which bandwidth for a video is reserved. We refer to this list as the *reserved list*. The other data structure utilized by our scheme is a *free list*, which stores information relating to the available bandwidth on each disk; the free list entries are (disk, available bandwidth) ordered pairs. For each reserved and free list entry e, $e(1)$ and $e(2)$ denote the first and second elements of e, respectively. We assume the m disks are numbered 0 through $m-1$. The distance from disk i to disk j is denoted by $dist(i,j)$. If $i \leq j$, then $dist(i,j)$ is simply $j-i$; else $dist(i,j)$ is $m-i+j$. Note that $dist(i,j)$ may not be equal to $dist(j,i)$. The distance between the disk on which available bandwidth is reserved for a video V_i and $disk(V_i)$ is the number of rounds for which $\frac{d}{r_{disk}} + t_{rot} + t_{settle}$ portion of a disk's bandwidth would be unutilized due to video V_i. It is also the number of rounds after which data retrieval for V_i is begun. The available bandwidth for each disk in the free list is $\frac{d}{r_{disp}} - 2 \cdot t_{seek}$ initially. The reserved list is initially empty.

The scheme, for inserting videos from the request list into service lists at the start of a round just before service lists are sorted, is as shown in Figure 4. In Step 1, the bandwidth available on a disk is set to the bandwidth available on the disk preceding it. It may be possible that for newly available bandwidth on disk $(i+1) \bmod m$ (due to the completion of data retrieval for a video on disk i), swapping it with the disk assigned to some video in the reserved list could result in better bandwidth utilization. Thus, in Step 2, for a video V_j in the reserved list, if disk $(i+1) \bmod m$ is closer to $disk(V_j)$ than the disk currently assigned to V_j, then disk $(i+1) \bmod m$ is assigned to V_j. The available bandwidth on the disk that was previously assigned to V_j is then similarly assigned to some subsequent video in the reserved list; the assignments are repeated until the end of the reserved list is reached. In Step 3, available bandwidth in the free list is assigned to requests at the head of the request list, and finally, in Step 4, videos in the reserved list for which data retrieval can begin, are added to the service lists. The following example illustrates the actions performed by the various steps of the scheme.

Example 2: Consider the server in Example 1 with 100 disks and videos $V_1, \ldots, V_{1000}$. Suppose, at the start of a round, say r_0, the request list contains requests for videos V_1 and V_{1000} (the one for V_1 preceding the one for V_{1000}), the reserved list is empty, and the only disks with available bandwidth greater than or equal to $\frac{d}{r_{disk}} + t_{rot} + t_{settle}$ in the free list are disks 30 and 48. Due to Step 3, since $dist(48, disk(V_1)) < dist(30, disk(V_1))$ ($disk(V_1)$ is 0), $(48, V_1)$ is first appended to the reserved list, followed by $(30, V_{1000})$. Suppose, during round r_0, data retrieval for video V_2 completes at disk 99. At the start of round r_0+1, the available bandwidth on disk 0 is added to the free list as follows. During the first iteration of Step 2, since $dist(0, disk(V_1)) < dist(49, disk(v_1))$, cur_disk is set to 49 and the first reserved list entry becomes $(0, disk(V_1))$. During the second iteration, since $dist(49, disk(V_{1000})) < dist(31, disk(V_{1000}))$, cur_disk is set to 31 and the second reserved list entry becomes $(49, V_{1000})$. At the end of Step 2, $\frac{d}{r_{disk}} + t_{rot} + t_{settle}$ is added to the available bandwidth of the free list entry for disk 31. Due to Step 4, since $disk(V_1) = 0$, V_1 is inserted into the service list for disk 0 at the start of round r_0+1; furthermore, since $disk(V_{1000}) = 50$, V_{1000} is inserted into the service list for disk 50 at the start of round r_0+2. $\square$

Note that, in Step 2, since a video V_i in the reserved list is not re-assigned to a different disk unless the disk is closer to $disk(V_i)$, data retrieval for no video in the request list is delayed forever; thus, the scheme is starvation-free. Also, the scheme incurs very little overhead. The availability of new disk bandwidth results in a comparison being performed for every entry in the reserved list. Furthermore, the addition of a request into the reserved list requires a comparison with every disk entry in the free list (in order to determine the closest disk with the required available bandwidth).

We next show that the assignment of available bandwidth to videos in the reserved list due to the scheme results in as good or better disk utilization than any other assignment of the available disk bandwidth to the same videos. Note that, for an entry e in the reserved list, $dist(e(1), disk(e(2)))$ is the number of rounds that bandwidth for a stream is wasted. Thus, we show that the scheme minimizes the wasted bandwidth by showing that the sum total of $dist(e(1), disk(e(2)))$, for all entries e in the reserved list, is less than or equal to the total distance due to any other assignment of videos in the reserved list to disks with available bandwidth in the reserved and free lists. We begin by showing that the following invariant always holds.

For every entry e in the *reserved* list, for all disks k contained in (1) entries following e in the reserved list, and (2) entries in the free list such that the available bandwidth is at least $\frac{d}{r_{disk}} + t_{rot} + t_{settle}$, the following is true: $dist(e(1), disk(e(2))) \leq dist(k, disk(e(2)))$.

The above invariant is trivially preserved when an entry e is appended to the end of the reserved list in Step 3, since among disks with available bandwidth in the free list, $dist((e_1), disk(e(2)))$ is minimum. When bandwidth on a disk becomes available, every iteration in Step 2 of the scheme also preserves the invariant, and in addition, the following property: for entries e preceding cur_entry, $dist(e(1), disk(e(2))) \leq dist(\mathsf{cur\_disk}, disk(e(2)))$. We show that Step 2(b) preserves both the invariant and the property. The invariant and property is preserved for all entries e preceding cur_entry as a result of Step 2(b), since before Step 2(b), $dist(e(1), disk(e(2))) \leq dist(\mathsf{cur\_disk}, disk(e(2)))$ and $dist(e(1), disk(e(2))) \leq dist(\mathsf{cur\_entry}(1), disk(e(2)))$, and the only action that Step 2(b) can perform is swap cur_entry(1) and cur_disk. Furthermore, since due to Step 2(b), $dist(\mathsf{cur\_entry}(1), disk(\mathsf{cur\_entry}(2)))$ can only decrease, and $dist(\mathsf{cur\_entry}(1), disk(\mathsf{cur\_entry}(2))) \leq dist(\mathsf{cur\_disk}, disk(\mathsf{cur\_entry}(2)))$ at the end of Step 2(b), the invariant and property are preserved for cur_entry.

We now show that for any two entries (i, V_j) and (k, V_l) in the reserved list ((i, V_j) preceding (k, V_l)), assigning disk i to V_l and disk k to V_j does not cause the total distance to decrease. The reason for this is that, due to the invariant, $dist(i, disk(V_j)) \leq dist(k, disk(V_j))$, and since distances between disks are linear, it follows that $dist(i, disk(V_j)) + dist(k, disk(V_l)) \leq dist(i, disk(V_l)) + dist(k, disk(V_j))$. The following theorem, thus, follows from the above invariant.

Theorem 1: Consider an arbitrary assignment A of videos in the reserved list to disks with available bandwidth in the reserved and free lists ($A(V_i)$ is the disk assigned to video V_i). The sum of $dist(A(V_i), disk(V_i))$ for all videos assigned in A is greater than or equal to the sum of $dist(e(1), disk(e(2)))$ for all entries e in the reserved list. $\Box$

5.2 Determining the Optimal Stripe Unit Size

We now address the issue of determining an optimal value for d such that the number of concurrent streams that can be supported by the server is maximized. For a given value of d, the maximum number of streams that can be supported by each disk is $\frac{\frac{d}{r_{disp}} - 2 \cdot t_{seek}}{\frac{d}{r_{disk}} + t_{rot} + t_{settle}}$, the maximum value for q obtained as a result of solving Equation 2 (and by the server, it is m times the amount). Furthermore, since the buffer per stream is $2 \cdot d$, and the total buffer requirements cannot exceed D, the optimal value for d can be obtained by solving the following equation:

$$2 \cdot m \cdot d \cdot \left(\frac{\frac{d}{r_{disp}} - 2 \cdot t_{seek}}{\frac{d}{r_{disk}} + t_{rot} + t_{settle}}\right) = D$$

Let d_{calc} be the maximum value of d obtained as a result of solving the above equation. Let q_{calc} be the maximum number of streams that can be supported by each disk for $d = d_{calc}$. Note that $2 \cdot d_{calc} \cdot q_{calc} \cdot m = D$. The problem is that q_{calc} may not be an integer. As a result, the maximum number of concurrent streams that can be supported by the server is $\lfloor q_{calc} \rfloor \cdot m$. Thus, since the RAM required to support $\lfloor q_{calc} \rfloor \cdot m$ streams may be substantially less than D, if $d_{\lceil calc \rceil}$ is the minimum value of d in order to support $\lceil q_{calc} \rceil$ streams from a disk, then by choosing the value of d to be $d_{\lceil calc \rceil}$, it may be possible for the server to support a larger number of concurrent streams. Since $d_{\lceil calc \rceil} \geq d_{calc}$, if $d = d_{\lceil calc \rceil}$, then the maximum number of streams that can be supported by the server is $\lfloor \frac{D}{2 \cdot d_{\lceil calc \rceil}} \rfloor$. Depending on which of $\lfloor q_{calc} \rfloor \cdot m$ or $\lfloor \frac{D}{2 \cdot d_{\lceil calc \rceil}} \rfloor$ is greater, the value of d is set to either d_{calc} or $d_{\lceil calc \rceil}$.

Thus, if $D = 2$ Gb and $m = 50$, then the optimal value of d is .9 Mb, and the maximum number of streams that can be supported is 1011. This is almost 7.5 times the maximum number of streams that can be supported by

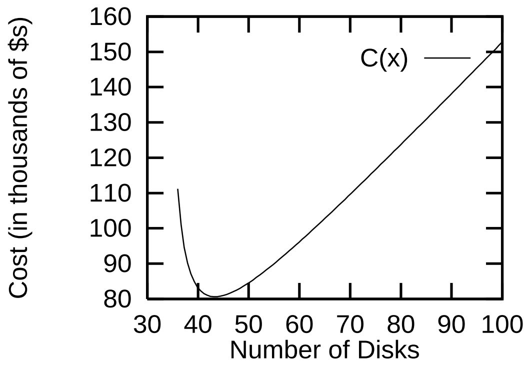

Figure 5: Cost of Server as a function of Number of Disks

the fine-grained striping scheme for the same values of D and m. The reason for this is that the latency overhead per data access for the fine-grained scheme is m times the overhead per data access for the coarse-grained scheme. As a result, the value of d_{calc} computed for the fine-grained scheme. is larger than that computed for the coarse-grained scheme.

For simplicity, in the schemes presented in this section and later in the paper, video streams are begun only based on the availability of disk bandwidth. As a result, we select d to be d_{calc}. The schemes can be extended to take into account even the availability of buffer space, in which case, an optimal value for d can be selected as described above. However, we do not discuss these extensions in this paper.

5.3 Designing an Optimal-cost Video Server

We next address the issue of architecting an optimal-cost video server that supports a certain pre-specified number, Q, of concurrent streams. If the server has m disks, then in order to support Q streams from the m disks, d must be such that data for $\lceil \frac{Q}{m} \rceil$ concurrent streams can be retrieved from each of the disks. Thus, from Equation 2, it follows that $d = \frac{(\lceil \frac{Q}{m} \rceil \cdot (t_{rot} + t_{settle}) + 2 \cdot t_{seek}) \cdot r_{disk} \cdot r_{disp}}{r_{disk} - \lceil \frac{Q}{m} \rceil \cdot r_{disp}}$, and the minimum amount of RAM required, $D = 2 \cdot Q \cdot d$. As a result, the cost of the server, in terms of the number of disks, obtained as a result of substituting for d in $C_r \cdot 2 \cdot Q \cdot d + C_d \cdot m$ is as follows.

$$C(m) = \frac{C_r \cdot 2 \cdot Q \cdot (\lceil \frac{Q}{m} \rceil \cdot (t_{rot} + t_{settle}) + 2 \cdot t_{seek}) \cdot r_{disp}}{1 - \lceil \frac{Q}{m} \rceil \cdot \frac{r_{disp}}{r_{disk}}} + C_d \cdot m$$

In Figure 5, we show how $C(m)$ varies with the number of disks, m, for a server that uses coarse-grained striping and is required to support 1000 MPEG-1 streams. An optimal integral value for m at which $C(m)$ is minimum, subject to the constraint that $m > \frac{Q \cdot r_{disp}}{r_{disk}}$, can be computed as described earlier in Section IV.

The optimal value of m for a server that supports 1000 MPEG-1 streams is 44. The amount of RAM required is 3.08 Gb and the cost of the server is $81,404. Thus, the cost of the server when coarse-grained striping is used to store videos is almost a quarter of the cost if fine-grained striping is used to support 1000 MPEG-1 streams. Also, the value of d is 1.5 Mb and the duration of a round is 1s.

Acknowledgements:

We would like to thank Bob Holt and B. Narendran for sharing with us certain known mathematical results that helped improve our understanding of certain results presented in this paper.

References

[1] S. Berson, S. Ghandeharizadeh, R. Muntz and X. Ju. Staggered striping in multimedia information systems. In *Proceedings of ACM-SIGMOD 1988 International Conference on Management of Data,* 1994.

[2] S. Ghandeharizadeh and L. Ramos, Continuous retrieval of multimedia data using parallelism. *IEEE Transactions on Knowledge and Data Engineering*, 5(4):658–669, August, 1993.

[3] B. Özden, R. Rastogi and A. Silberschatz. Disk striping in video server environments. Technical Report 113830-950220-01, AT&T Bell Laboratories, Murray Hill, 1995.

[4] G. R. Ganger, B. L. Worthington, R. Y. Hou and Y. N. Patt. Disk arrays: high-performance, high-reliability storage subsystems. *Computer*, 27(3):30-36, March 1994.

[5] D. Patterson, G. Gibson and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of ACM-SIGMOD 1988 International Conference on Management of Data*, 1988.

[6] A. Silberschatz and P. Galvin. *Operating System Concepts.* Addison-Wesley, 1994.

[7] F. A. Tobagi, J. Pang, R. Baird and M. Gang. Streaming RAID: A disk storage system for video and audio files. In *Proceedings of ACM Multimedia, Anaheim, CA*, pages 393-400, 1993.

I/O Stream Sharing for Continuous Media Systems

Leana Golubchik[†] John C.S. Lui[‡] Richard Muntz[§]

Abstract

Recent technological advances have made multimedia on-demand services, such as home entertainment and home-shopping, feasible. One of the most challenging aspects of such systems is providing access either instantaneously or within a small and reasonable latency upon request. In this paper, we discuss various data sharing techniques which reduce the aggregate I/O demand on the multimedia storage server and thus improve its overall performance.

1 Introduction

Recent technological advances in information and communication technologies have made multimedia on-demand services, such as movies-on-demand, home-shopping, etc., feasible. Information systems today can not only store and retrieve large multimedia objects, but they can also meet the stringent real-time requirements of continuously providing objects at a constant bandwidth, for the entire duration of that object's display. Already, multimedia systems play a major role in educational applications, entertainment technology, and library information system.

An example of a video-on-demand storage server is depicted in Figure 1; such a server archives many objects of long duration, e.g., movies, music videos, educational training material, etc. The storage server consists of a set of disks ($D_1 \ldots D_N$), a set of processors ($P_1 \ldots P_K$), buffer space, and a tertiary storage device. The entire database resides on tertiary storage, and the more frequently accessed objects are cached on disks. It is reasonable to assume that a request for an object must be serviced from the disk sub-system; the size of the objects (on the order of 4.5 GB for a 100 minute MPEG-2 encoded movie) precludes them from being stored in main memory, and the long latency and high bandwidth cost of tertiary storage[1] precludes objects from being transmitted directly from tertiary devices. If the requested object is not disk-resident, then it has to be retrieved from the tertiary store and placed on disks before its display can be initiated[2]; this could result in one or more objects being purged from disks, due to lack of space. A disk-resident object can be displayed by scheduling an I/O stream and reading the data from the appropriate disks.

One of the most challenging aspects of such systems is providing *on-demand* service to multiple clients simultaneously, thus realizing economies of scale; that is, users expect to access objects, e.g., movies, within a small and "reasonable" latency, upon request. We define the latency for servicing a request as the time between

[†]Department of Computer Science, Columbia University (leana@cs.columbia.edu). This research was supported in part by the IBM graduate fellowship and was partially done while the author was a graduate student at UCLA.

[‡]Department of Computer Science, The Chinese University of Hong Kong (cslui@cs.cuhk.hk). This research was supported in part by the CUHK Direct Grant and the Croucher Foundation.

[§]Computer Science Department, UCLA (muntz@cs.ucla.edu). This research was supported in part by Hewlett Packard through an equipment grant.

[1]The seek latency for a 1.3GB tape on a \$$1000$ tape drive can be on the order of 20 seconds [6], whereas a similarly priced disk, of a similar capacity, has a maximum seek time on the order of 35 milliseconds and more than 16 times the transfer rate. Tape systems with significantly higher transfer rates and tape capacities although not with much lower seek latency do exist, but at a cost \$$40,000$-\$$300,000$.

[2]Techniques exist for starting playback of an object before it is entirely retrieved from tertiary storage; however, we will not discuss these here and will assume for the remainder of the paper that the entire object is retrieved before playback is initiated.

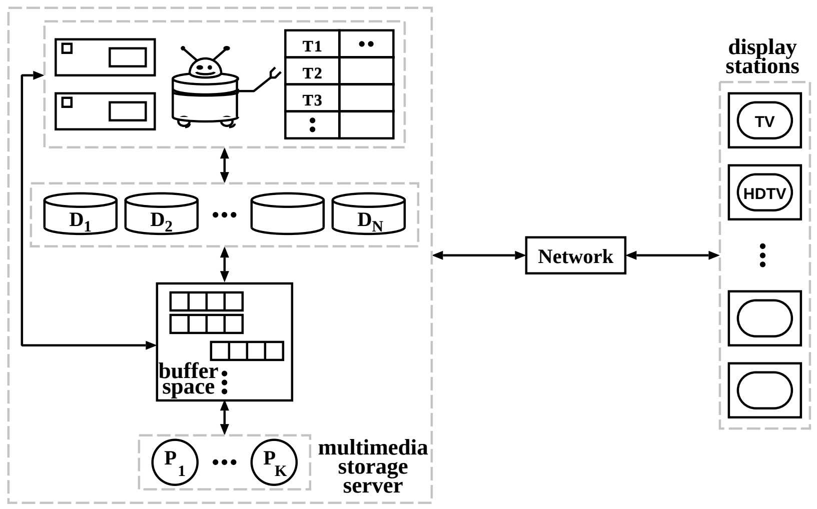

Figure 1: Multimedia Storage Server Architecture.

the request's arrival to the time the system initiates the reading of the object (from a disk); for the purposes of our discussion, we consider the additional delay until data is actually delivered to the display device relatively negligible. Latency can be attributed to: a) insufficient bandwidth for servicing the request, b) insufficient buffer space for scheduling its reading from the disks, or c) insufficient disk storage, i.e., the object in question may not be disk-resident and hence may have to be retrieved from tertiary storage before it can be scheduled for display.

For ease of exposition, we can assume that the server, depicted in Figure 1, can be described by the following three parameters: 1) total available I/O bandwidth, 2) total available disk storage space, and 3) total available buffer space[3]. These parameters, in conjunction with data layout and scheduling schemes, determine the cost of the server as well as the "quality of service" it can offer; although quality of service is a somewhat ambiguous term, the *latency*, in servicing a video request, is one useful measure. In general, the more video streams a system can support simultaneously, the lower is the average latency for starting the service of a new request (at least for the disk resident objects).

There are many architectures that can be used for constructing a video-on-demand server [2, 16, 12, 1, 10]. The distinctions between these architectures can be (mostly) attributed to the data layout and scheduling techniques used. Let us consider one such system[4], where the workload can be described by $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_K)$, where λ_i is the arrival rate of requests for object i and K is the total number of objects available on the storage server (including the non-disk-resident objects). Informally, we expect a skewed distribution of access frequencies with a relatively small subset of objects accessed very frequently, and the rest of the objects exhibiting fairly small access rates[5]. In such a system, it is fair to assume that there is sufficient disk storage to at least hold the popular objects; moreover, it is very likely that I/O bandwidth is the critical resource which contributes to increases in latency. One way to reduce the latency is to simply purchase more disks. A more interesting and more economical approach might be to either attempt to improve the data layout and scheduling techniques or to reduce the I/O demand of each request in service through "sharing" of data between requests for the same object.

There are several approaches to reducing the I/O demand on the storage server through data sharing, or, in

[3]We will not consider the characteristics of the tertiary device in this paper.

[4]In this work, we focus on the benefits that can be gained from I/O stream sharing. These techniques can be applied to almost any VOD server architecture, although we do not consider a specific implementation here; of course, the actual amount of performance improvement will depend on a particular implementation.

[5]For instance, a movie server would have such characteristics, where a small subset of popular movies (for that week, perhaps) is accessed simultaneously by relatively many users; furthermore, we can assume that the change in access frequency is relatively slow, e.g., the set of popular movies should not change more often than once per week.

effect, increasing the number of user requests which can be served simultaneously. For example:

1. *batching*: delaying requests for up to T_i time units in hopes of more requests, for the same object i, arriving during the batching interval and servicing the entire group using a single I/O stream

2. *buffering*: closing the temporal "gaps" between successive requests through the use of buffer space, i.e., holding data read for a "leading" stream and servicing "trailing" requests out of the buffer rather than by issuing another I/O stream

3. *adaptive piggybacking*: adjusting display rates of requests *in progress* (for the same object) until their corresponding I/O streams can be "merged" into one

Note that, these techniques are *not* mutually exclusive; for example, the results of using adaptive piggybacking in conjunction with batching are presented in [7]. Note also, that sharing of data between requests for the same object, results in additional complications when attempting to provide VCR functionality, for instance, allowing one user in a batch to pause and then to eventually resume. This problem is a difficult one and is investigated in [12, 4, 18], to name a few.

In general, the following parameters can be used to improve the number of simultaneous requests that a system can serve: 1) delay time (for batching), 2) merging policy (for adaptive piggybacking), 3) buffer allocation policy, and 4) display rate altering techniques (see Appendix 6 for more details), where (as already mentioned) reduction in the I/O bandwidth consumed by the aggregate requests for a movie is considered to be the main goal of these policies. While other resources are affected, disk bandwidth is likely to be the most important and costly. This will remain so for the foreseeable future since disk capacity is increasing at a faster rate than disk bandwidth.

In this paper we describe each of the *data sharing* techniques, as well as their advantages and disadvantages, in turn. The remainder of the paper is organized as follows. In Section 2 we describe batching approaches as well as the tradeoffs associated with batching of streams. In Section 3 we describe buffering approaches, and in Section 4 we present adaptive piggybacking. Finally, Section 5 presents our concluding remarks.

2 Batching

A *batching procedure* is defined to be an I/O scheduling policy which, on a per object basis, delays the initiation of I/O streams for the purpose of grouping requests and using a single I/O stream to service the entire group. Grouping requests can save system resources by allocating one I/O stream for each batch of requests instead of one I/O stream per request; however, it can also result in higher average latency for initiating a display of an object. Hence, there is an obvious tradeoff between initial delay experienced by each request after arrival and the number of I/O streams that a system can support. In this paper, we are interested in controlling the utilization of the I/O subsystem; for reasonably busy systems (the only really interesting case), the lower utilization a system has, the lower is its response time for servicing requests.

There are several ways to batch requests into a single I/O stream. We discuss the following basic batching policies: (1) batching by size and (2) batching by timeout. For the purposes of this discussion, we assume that the request arrival process, for a particular object j, is Poisson with rate λ_j.

Batching by size
Let B_j be the pre-defined batching size and λ_j be the arrival rate of requests for object j. The system issues an I/O request to the storage server only when B_j such requests accumulate in the system. Let $E[N_j]$ be the expected reduction (due to batching) in the number of I/O streams issued; then

$$E[N_j] \quad = \quad B_j - 1 \qquad (3)$$

Let L_j be a random variable denoting the latency experienced by each request for object j; then the expected latency is:

$$E[L_j] = \frac{1}{B_j} \sum_{i=1}^{B_j} \frac{B_j - i}{\lambda_j} = \frac{B_j - 1}{2\lambda_j} \tag{4}$$

Although this policy reduces the I/O demand on the storage server, it can result in long delays for requests, particularly for low to moderate arrival rates[6].

Batching by timeout
Another policy is "batching by timeout". The timer is set when a request arrives to the storage server and there exists no other outstanding request for the same object j. The system issues an I/O request to the storage server T_j time units after the initiation of the timer. Any request, for the same object, arriving during these T_j time units is *batched* and serviced when the timer expires. Let N_j be a random variable denoting the number of I/O streams saved due to batching, and p_k^j be the probability of k arrivals during time T_j; since the arrival process is Poisson, we have:

$$E[N_j] \;\; = \;\; \lambda_j T_j \tag{5}$$

To evaluate the expected latency experienced by each request, we can view the system as an $M/G/1$ queue with a constant setup time (where the setup time is the duration of the timer T_j) and a deterministic service time distribution with a mean of zero. The expected latency for this type of a system can be found in [15] as:

$$E[L_j] \;\; = \;\; \frac{T_j(2 + \lambda_j T_j)}{2(1 + \lambda_j T_j)} \tag{6}$$

Since $E[N_j] = \lambda_j T_j$, the I/O demand on the secondary storage can be reduced tremendously under moderate to high request arrival rates. This is also a reasonable policy for movie-on-demand applications because each requests, for object j, does not experience more than T_j units of delay due to batching. Figure 2 depicts the expected latency curves for the two batching policies described in this section.

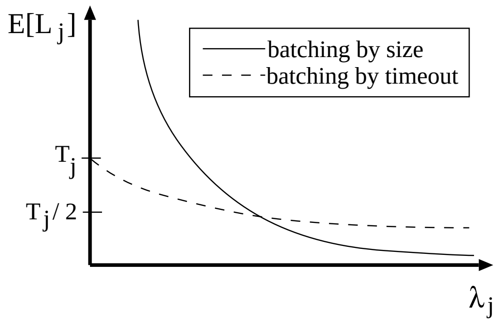

Figure 2: Exp. Latency for Batching Policies (for object j).

There has been considerable work done in investigating the use of batching techniques. We describe several of these below. In [5], the authors consider several policies for deciding on which movie to multicast (or which requests to batch), where the objective is to reduce the average waiting time for starting the service of a request as well as to reduce the probability of a request reneging, i.e., there is some non-zero probability that a request

[6]Depending on the network characteristics, it might be wise to batch by size, since it can result in lower network traffic; however, we do not consider network characteristics in this paper.

may decide to leave the system if it does not receive service within some ("reasonable") amount of time; note that these are often conflicting goals. Fairness is also considered as the third objective of this optimization problem. The two policies that are studied in the paper are FCFS and MQL (maximum queue length). The FCFS policy is the recommended policy, especially if the reneging probability is a function of the waiting time.

In [4], the authors also consider batching as a resource conservation technique, and they correctly point out that such techniques complicate provision of VCR functionality; more specifically, when one of the clients in a batched group pauses and then eventually attempts to resume, the system must provide a new I/O stream in order to serve the resumed client. (Of course, if no streams are available when a resume is requested, the client is forced to wait.) The authors consider the following tradeoffs when analyzing the batching policies. Firstly, the longer is the batching interval, the greater are the savings from batching, but also the greater is the probability of reneging of a client. Secondly, to address the VCR functionality problem, the authors consider reserving a pool of channels in order to provide quicker response to future resume requests; these are termed contingency channels. In this work, an analytic model is developed for predicting the reneging probability and the expected resume delay; this model is used to allocate channels for batching, contingency channel pool, and on-demand (i.e., non-batched) service.

In [12], the authors propose an architecture for a movies-on-demand (MOD) system, which is efficient for multiple concurrent retrievals of data for servicing requests for the same movie. The basic scheme works as follows. A movie is divided into p phases, where the duration of a phase is determined by the bandwidth requirements of the movie as well as the bandwidth capability of the disk or disk array that stores that movie. The data layout and scheduling schemes are such that the system architecture can continuously support multiple simultaneous retrievals of the same movie, separated by one phase. Thus a request for a particular movie can belong to one of p groups that are supported by the p concurrent streams (separated in time by the duration of a phase) being retrieved for that movie. One important advantage of the schemes presented in this work is that they make it fairly simple to provide VCR functionality (see [12] for details). Note that, the "granularity" of the VCR functions, e.g., how long of a delay a user might incur after pushing the resume button, the frame rate of the fast forward function as compared to normal display, etc. depends on the duration of the phase; finer granularity can be provided, but at the cost of a (possible) storage/bandwidth mismatch. In [12], the authors propose several solutions to the "granularity" problem.

In [18], the authors are mainly concerned with providing pause/resume functionality in a system where batching techniques are used to conserve resources. They propose an interesting solution to this problem, termed look-ahead scheduling with look-aside buffering, which can be briefly described as follows. The idea of look-ahead scheduling is to "back up" each display (in case of a pause/resume request) with a stream that is being used for a different display and is near completion, rather than backing up each display with "real" stream capacity. The look-aside buffering is used to support the display's pause/resume needs until the appropriate look-ahead stream runs to completion and becomes available. Note that, although this scheme does result in significant throughput improvements, this is accomplished through the use of a very large buffer pool; thus, it remains to be seen at which point it is more cost-effective to purchase additional disks (i.e., increase the bandwidth of a system) rather than continue increasing the buffer pool. This of course, depends on the future trends of buffer to disk cost rations.

3 Buffering

A *buffering procedure* is defined to be a buffer scheduling policy which attempts to bridge the temporal gap between two requests, for the same object, as follows. Rather than deallocating the buffer space after transmitting data for a particular request, the system can retain the information in the buffer so that later requests, for the same object, can extract this data directly from the buffers rather than issue another I/O request to the secondary storage. The tradeoff here is between I/O bandwidth utilization and buffer space utilization. Retaining even a few minutes of a movie in main memory can be an expensive proposition[7] and might only be cost effective for the very popular objects.

Considerably less studies have been performed on buffering techniques. We briefly describe a few of these below. In [9, 8] the authors consider the use of buffering techniques to improve the efficiency of use of the disk I/O bandwidth. In this work they discuss schemes for caching continuous media data. More specifically, the authors propose and analyze heuristics for determining when data, that has been played back for a particular request, should be kept in buffers for the purpose of reuse (or data sharing) with future requests for the same objects. Their results indicate significant performance improvements, when data sharing is possible.

In [3], the authors also consider caching techniques for improving the efficiency of the I/O subsystem. They study the cost impact of varying buffer sizes, disk utilization, as well as disk characteristics on the capacity of the system. A simulation study is performed to determine the benefits of interval caching, i.e., exploiting temporal locality rather than just caching the hottest objects. The cost-effectiveness of such buffering techniques is, of course, highly affected by the fluctuations in the workload.

In [13], the authors present novel demand paging algorithms which reduce I/O bandwidth requirements through the use of buffer space. A "pinning" demand paging technique is given to insure that the real-time constraints of displaying video objects are satisfied. Furthermore, algorithms are presented for eliminating disk bandwidth limitations, given that there is a sufficient amount of buffer space. Finally, the problem of optimizing the total buffer space requirements is considered, in order to increase the number of streams that can be serviced simultaneously.

4 Adaptive Piggybacking

An *adaptive piggybacking procedure* [7] is defined to be a policy for altering display rates of requests *in progress* (for the same object), for the purpose of *merging* their respective I/O streams into a single stream, which can serve the entire group (of merged requests). The idea is similar to that of batching, with one notable exception. The grouping is done *dynamically* and while the displays are *in progress*, i.e., no latency is experienced by the user. Note that, the reduction in the I/O demand is not quite as high as in the case of batching, since some time must pass before the streams can merge[8]; hence, the tradeoff (between these two techniques) is between latency for starting the service of a request and the amount of I/O bandwidth saved.

Consider a storage system, where for each request for an object there exists a display stream and a corresponding I/O stream. The processing nodes use the I/O streams to retrieve the necessary data from disks, possibly modify the data in some manner, and then use the display streams to transmit the data (through the network) to appropriate display stations (e.g., in Figure 3, display streams 1 and 2 are serviced using the corresponding I/O streams 1 and 2). The I/O demand on the storage server can be reduced by using a single I/O stream to service several display streams corresponding to requests for the same object (e.g, in Figure 3, display streams 3 and 4 correspond to requests for the same object and are serviced using a single I/O stream[9], 3). This can be done in a *static manner*, i.e., by batching requests (see Section 2), and in a *dynamic* or *adaptive* manner.

A dynamic approach initiates an I/O stream, for each display stream, on-demand, and then allows one display stream to *adaptively piggyback* on the I/O stream of another display stream (for the same object). We can also view this as a *dynamic merging* of two I/O streams into one. Before the merge, there were two I/O streams, each serving one (or more) display stream(s), where the display streams correspond to two temporally separated displays of the same object. Of course, the benefits are that after the merge, there is only one I/O stream, which

[7]For instance, a 1 min segment of an MPEG-2 stream may occupy as much as 45 MB of buffer space; of course, the duration of time for which it is needed is of importance.

[8]The display adjustment must be gradual (or slow) enough to insure that it is not noticeable to the user; we assume that altering the quality of the display (as perceived by an "average" user) is not an acceptable solution.

[9]Depending on the network characteristics, it might be wiser to delay "splitting" display streams 3 and 4 until the last possible moment, i.e., transmit them through the network as a single stream for as long as possible. However, we do not consider network characteristics in this paper; hence, we shall not consider alternative transmission policies here which can reduce network bandwidth utilization.

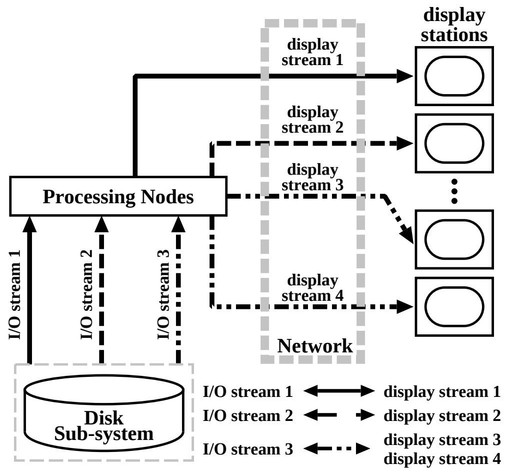

Figure 3: Simplified View of the System.

can service both display streams, and furthermore the corresponding displays are then “in synch”. This merging can be accomplished by adjusting requests’ display rates, i.e., rather than displaying each request at the “normal” rate, the system can adjust the display rate of each request (see Appendix 6), either to a “slower” rate or a “faster” rate, in order to close the temporal gap between the displays[10].

Consider an analogy of servicing video requests, for a particular movie, to a collection of bugs sitting on a moving conveyor belt (refer to Figure 4). The conveyor belt represents one particular movie; its length corresponds to the duration of the movie’s display, and the rate at which the conveyor moves corresponds to the *normal* display rate of the movie (e.g., 30 frames/sec for U.S. television). Each bug represents a single I/O stream, servicing one or more display requests for that movie; the position of the bug on the conveyor belt represents the part of the movie being displayed by the corresponding I/O stream. If a bug chooses to remain still on the conveyor belt, then the corresponding stream displays the movie at the normal rate. If the bug chooses to crawl forward (at some speed), then the corresponding movie is displayed at a slightly higher rate. Similarly, if the bug chooses to crawl backwards (at some speed), then the corresponding movie is displayed at a slightly lower rate.

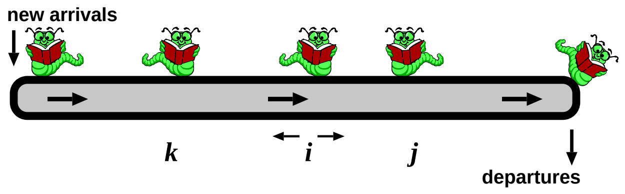

Figure 4: Conveyor Belt Analogy.

If two bugs, one crawling forward and one crawling backward, are able to “merge” at time t, before either one falls off the conveyor belt, then starting at time t the system is able to support both displays using only a single I/O stream. Consider for the moment bug i in Figure 4, which must make a decision, namely, whether to crawl forward, toward bug j, and piggyback on its I/O stream or whether to crawl backward, toward bug k, and instead piggyback on its stream. If i crawls forward, then it will take less time to merge; however, after the merge, a smaller portion of the movie will remain (to be displayed), and hence the benefits of merging would not

[10]In Appendix 6, we discuss different approaches to altering display rates; of course, an appropriate choice depends on the particular implementation of the VOD server and involves such considerations as effect on: data layout techniques, scheduling, etc. (see [7] for details).

be as great. On the other hand, if i crawls backward, toward k, then it will take longer to merge; however, greater benefits might be reaped from that merger, if it can be achieved at an earlier portion of the conveyor belt.

Thus, one can view the duration of the object's display as a *continuous* line of finite length and consider the problem of adaptive piggybacking as a decision problem; given the global state of the system, i.e., the position (relative to the beginning of the display) of each display stream in progress, one must choose a display rate for each of these requests, such that the total average I/O demand on the system is minimized[11]. In [7], several merging policies, in conjunction with batching policies, are investigated and evaluated with respect to reduction in I/O bandwidth utilization. The results indicate convincingly that small variations in the delivery rate can enable enough merging of I/O streams that significant reduction of I/O bandwidth is realized.

5 Conclusions

On demand video servers present some interesting performance problems. Part of the effort is simply to understand the constraints and goals well enough to appreciate what is possible. In this paper we have presented several techniques for sharing data between requests for the same object in a multimedia storage servers, namely, batching, buffering, and adaptive piggybacking; we also described several studies in each catagory. The results of these studies indicate that significant benefits can be gained from sharing data; however, as expected, these are highly workload dependent.

6 Appendix A — Altering Video Display Rates

As stated in Section 4, adaptive piggybacking is a viable technique for reducing I/O demand on a video storage server, if the storage server has the capability to dynamically alter the display rate of a request, or, rather, to dynamically *time compress* or *time expand* some portion of an object's display[12]. In this section we discuss how this can be done.

If one makes the basic assumption that the display units being fed by the storage server are NTSC standard and display at a rate of 30 frames per second (fps), then any time expansion or contraction should be done at the storage server. Slow down in the effective display rate can be done by adding additional frames to the video since the display device displays at a fixed rate. For example, if 1 additional frame is added for every 10 of the original frames, the effective display rate (orig-frames/sec) will be $30 \times \frac{10}{11}$. Similarly, by removing frames the effective display rate can be increased. There is ample evidence that effective display rates that are $\pm 5\%$ of the nominal rate can be achieved in such a way that it is not perceivable by the viewer (see [7] for further details). For instance, a movie shot on film is transferred to video using a *telecine* machine which adapts to the 30 fps required for the video from the 24 fps which is standard for films; this is done using a 3-2 pulldown algorithm [14, 11], which for every 4 movie frames creates 5 video frames, where two of the five frames produced are interpolations of a pair of the original frames. A similar type of interpolation could be used in systems using adaptive piggybacking.

There are two approaches to actually providing the altered stream of frames to be transmitted to the display stations:

- The altered version of the video can be created on-line. In this case the I/O bandwidth required from the disk varies with the effective display rate. There are two possible disadvantages of the on-line alteration: (1) the layout of the data on disk is often tuned to one delivery bandwidth and having to support multiple

[11] Note that in [7] the minimization of the average I/O demand is taken as the objective. Such reductions, if small, would not necessarily be a good measure of how latency is decreased; however it is shown in [7] that large reductions are obtainable, and therefore the reduction in I/O bandwidth requirements will translate directly to latency reduction.

[12] We do not discuss it in detail here, but necessary time adjustments can be performed on the audio portion of an object, using techniques such as audio pitch correction [17]; clearly, the rate of this adjustment must be chosen accordingly to insure the necessary synchronization [14] with the video portion of the object.

bandwidths can complicate scheduling and/or require additional buffer storage and (2) to support on the fly modification may require the expense of specialized hardware to keep up with the demand.

- The altered version of the video is created off-line and stored on disk with the original version. The obvious disadvantage of this approach is the additional disk storage required.

Based on the above discussion, it is reasonable to assume that one can alter the effective display rate by $\pm 5\%$ without sacrificing video quality, and one can consider both the on-line generation approach to providing the altered stream of frames and the off-line approach[13]. For the latter, one must include additional considerations in the scheduling policies that are motivated by the desire to limit the amount of additional disk space required for storing replicates of a video (see [7] for details).

References

[1] S. Berson, L. Golubchik, and R. R. Muntz. Fault Tolerant Design of Multimedia Servers. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 364–375, San Jose, CA, May 1995.

[2] Steven Berson, Shahram Ghandeharizadeh, Richard R. Muntz, and Xiangyu Ju. Staggered Striping in Multimedia Information Systems. *SIGMOD*, 1994.

[3] A. Dan, , D. Dias, R. Mukherjee, D. Sitaram, and R Tewari. Buffering and Caching in Large-Scale Video Servers. In *Proc. of COMPCON*, 1995.

[4] A. Dan, P. Shahabuddin, D. Sitaram, and D. Towsley. Channel Allocation under Batching and VCR Control in Movie-On-Demand Servers. Technical Report RC19588, IBM Research Report, Yorktown Height, NY, 1994.

[5] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling Policies for an On-Demand Video Server with Batching. In *Proc. of the 2nd ACM International Conference on Multimedia*, October 1994.

[6] A. L. Drapeau and R. Katz. Striped Tape Arrays. In *Proc. of the 12th IEEE Symposium on Mass Storage Systems*, pages 257–265, Monterey, California, April 1993.

[7] L. Golubchik, J. C.-S. Lui, and R. R. Muntz. Reducing I/O Demand in Video-On-Demand Storage Servers. In *Proceedings of the ACM SIGMETRICS and Performance*, pages 25–36, May 1995.

[8] M. Kamath, K. Ramamritham, and D. Towsley. Continuous Media Sharing in Multimedia Database Systems. In *Proc. of the 4th Intl. Conference on Database Systems for Advanced Applications (DASFAA '95)*, Singapore, April 10-13, 1995.

[9] M. Kamath, K. Ramamritham, and D. Towsley. Buffer Management for Continuous Media Sharing in Multimedia Database Systems. Technical Report 94-11, University of Massachusetts, Feb. 1994.

[10] S. W. Lau and J. C. S. Lui. A Novel Video-On-Demand Storage Architecture for Supporting Constant Frame Rate with Variable Bit Rate Retrieval. In *Proc. of the 5th Intl. Conf. on Network and Operating System Support for Digital Audio and Video*, April 1995.

[11] T. Ohanian. *Digital Nonlinear Editing: new approaches to editing film and video*. Focal Press, 1993.

[12] B. Ozden, A. Biliris, R. Rastogi, and A. Silberschatz. A Low-Cost Storage Server for Movie on Demand Databases. *Proc. of the 20th Intl. Conf. on Very Large Data Bases*, Sept. 1994.

[13] B. Ozden, R. Rastogi, A. Silberschatz, and C. Martin. Demand Paging for Video-on-Demand Servers. *Proc. of the IEEE Intl. Conference on Multimedia Computing and Systems*, May 1995.

[13]In either case one can assume that when frames are inserted, the additional frames are some interpolation of existing frames (not simply duplicates). Similarly, when a frame is deleted, the preceding and succeeding frames are altered to reduce the abruptness of the change (e.g., each becomes an interpolation of the original and the deleted frame).

[14] M. Rubin. *Nonlinear: a guide to electronic film and video editing*. Triad Publishing Co., 1991.

[15] H. Takagi. *Queueing Analysis: A Foundation of Performance Evaluation. Volume 1: Vacation and Priority Systems, Part 1*. North-Holland, 1991.

[16] F. A. Tobagi, J. Pang, R. Baird, and M. Gang. Streaming RAID - A Disk Array Management System For Video Files. *ACM Multimedia Conference*, pages 393–399, 1993.

[17] Personal Communication with Mr. Rich Igo and Mr. Bill Carpenter at Ampex Corp.

[18] P. S. Yu, J. L. Wolf, and H. Shachnai. Design and Analysis of A Look-ahead Scheduling Scheme to Support Pause-Resume for Video-on-Demand Applications. Technical Report RC19683, IBM Research Report, Yorktown Height, NY, 1995.

A Proposed Method for Creating VCR Functions using MPEG streams

David B. Andersen
Hewlett-Packard Company

1 Introduction

The development of Video-On-Demand (VOD) systems for movie delivery requires that the user be able to perform VCR functions over a broadband network system. These functions include Play, Pause, Fast Forward, and Fast Rewind. No standard method exists between content developers, server manufacturers, and client applications to provide these functions. This paper will propose a standard method for implementing these functions using MPEG streams and discuss some of the important tradeoffs.

The encoding and distribution of content has become one of the most important issues facing video information providers. Today, in the case of movies, every service provider must encode the material for the specific equipment being deployed in the network. Therefore, the ease of use and speed of the algorithms employed to encode the material are extremely important. In the future, the creator of the content may encode the material once and distribute it to the service providers in compressed form, but this is not the case today due to the lack of standards.

2 Data Management Issues

There are several important data management issues to consider when deciding on how to implement MPEG VCR functions. The two most important are buffer management at the client and the network architecture requirements for bandwidth allocation. In addition, the efficiency of the encoding process is also a data management concern. Many tradeoffs must be resolved in deciding these issues and standards must be adopted in order for broadband interactive television to be successful.

The two types of network architectures for interactive video are constant bit rate (CBR) and variable bit rate (VBR). In a CBR network, the server must ensure that every client receives an isochronous video stream. This implies that no handshaking is performed between the server and the client. The advantage of this type of network is that it can minimize the amount of buffer memory required at the client since it is assumed that the data will be delivered at a constant rate. In a VBR network, the client manages the data buffer in FIFO fashion and requests more data as necessary to guarantee the frame rate required. This type of buffer management is more suitable in networks that cannot allocate isochronous bandwidth. The disadvantage is that VBR systems usually require larger client buffers. The algorithm used to provide VCR functions should take these network architecture issues into account.

Once the network architecture and client buffer issues are understood, the algorithms for creating the MPEG video streams must be considered. Here the issues are efficiency and total data storage requirements. While it is not the intent of this paper to detail these issues, several future research directions shall be described.

3 Broadband Interactive Television systems for Video-On-Demand

There are several possible methods to provide VCR functions. In broadband VOD systems the most important constraint for any method is that of Constant Bit Rate (CBR). Interactive television requires isochronous data streams at all times. In addition, broadband networks employ a ”push” architecture that does not allow handshaking to perform client buffer management. This implies that even in Play mode the bit rate must be carefully controlled by the server and the network without any client intervention.

4 VCR Functions

Pause is the simplest of the VCR functions. It is typically implemented at the client by freezing the last picture.

Fast Forward and Fast Rewind can be implemented in many different ways. It is assumed for this discussion that the picture should remain viewable during these functions to distinguish them from skip ahead functions. The nature of the MPEG compression algorithm suggests that selective sampling of the I frames would achieve a fast forward effect that could be varied by the sampling frequency. The problem with this approach is that the aggregate network bit rate does not remain constant. This is due to the increased number of bits found in an I frame picture compared to P and B frames. This can lead to buffer overruns at the client. It also requires real-time processing at the server to perform the selective sampling.

An alternate method for implementing Fast Forward and Fast Rewind is to preprocess the content and create special VCR mode files. When the user requests Fast Forward mode, the server switches the output stream to the appropriate Fast Forward file. This has the advantage that the content creator can choose the algorithm to create the VCR mode file, perform the preprocessing once, and distribute the complete package to many service providers.

The VCR mode files can be created in one of two ways. This paper describes the Fast Forward file, but the Fast Rewind file is almost identical with the frames in reverse order. The first method requires that the original video material be sub-sampled before compression. For a ten times fast forward speed, every tenth video frame would be stored to a suitable device before MPEG encoding. This material would then be MPEG encoded in the normal fashion and stored in a separate file in the video server. This method results in very high quality Fast Forward viewing with very smooth motion, but requires intermediate storage of the sub-sampled uncompressed video.

The Fast Forward file can also be created in the MPEG compressed domain. Here, I frames are extracted and a valid MPEG stream is produced from the results. Each MPEG GOP (one I frame only, typically) is wrapped in a Sequence Header. The desired speed up rate is achieved by selecting one or more of the following strategies:

1. Select every I frame only from input play stream. Wrap this single I frame with a GOP header. The speed up rate is a function of the input I frame rate. If the input GOP size was, say, 15 pictures per GOP then the speed up would be 15x. If the GOP size was 12 then the speed up rate is 12x. This method limits the speed up rate to the GOP size.

2. Select certain I frames (from play stream) to meet a desired speed up rate. If the desired speed up rate is, say, 30 and the GOP size is 15 then build a stream with every other I frame captured. This method works for speed up rates greater than the GOP size. By choosing which I frames to capture, this technique can realize any desired average speed up rate (¿ GOP size).

3. Duplicate certain I frames (from play stream) to meet a speed up rate. This is needed when the speed up rate is less than the GOP size. For example, if GOP size is equal to one and a factor of five speed up is desired then every play I frame is captured and duplicated. The resulting picture exhibits a sample-and-hold look since every captured frame is held. By choosing which I frames to capture then any average

speed up rate (¡ GOP size) should be met. This effect may be achieved without actually duplicating the I frame. Whenever an I frame needs a duplicate, place a template P frame in its place. This P frame has no non-zero coefficient data so the previous I frame is held, in effect.

While these methods will create files with the desired speed up rate, the aggregate network bit rate will still increase. Therefore, the rate of each I frame needs to be reduced so that the new picture sequence has a reasonably low data rate. This can be accomplished by zeroing out higher frequency DCT coefficients in individual I frames. This method is computationally efficient and yields an acceptable picture sequence. For example, assume a 10X speed up for FF and FR and a 4:1 reduction in spatial resolution (fewer non-zero DCT coefficients). Then the FF and FR tracks only consume 5the server. This method can even result in reduced network bandwidth requirements for VCR functions if the spatial resolution is reduced sufficiently.

5 Video Server Implications

The server must store separate video tracks called the Fast-Forward and Fast-Reverse tracks. These tracks are jumped to and viewed when a user requests fast-forward and fast-reverse functions respectively. These tracks are MPEG-2 transport streams that contain the speeded up version of the standard track.

For the FF track, one out of every Nth picture in the STD track is represented in the track, where N is a predetermined ratio number. This ratio is attached to the FF and FR tracks as a property in a Table of Contents file for the content package.

To implement FF and FR functions, the capability must be provided to jump from one track to another. To facilitate this, the STD, FF, and FR tracks are required to have locations that allow random access into the track. These locations are known as Random Access Points (RAPs).

RAPs are defined to be at the start of the NULL Transport Stream Packet (TSP) that immediately precedes a video TSP whose random_access_indicator is set. The following must be true for this video TSP:

1. There shall be a NULL TSP immediately preceding it.
2. It shall have its payload_unit_start indicator set and shall contain a payload.
3. It shall contain an adaptation field.
4. The random_access_indicator shall be set in the adaptation field.
5. The PCR_flag shall be set in the adaptation field.
6. The video TSP shall contain a PCR.
7. The payload of the video TSP shall be the start of a PES packet.
8. The PES packet header shall contain a PTS
9. The first byte of the PES packet payload shall contain the first byte of a video sequence.

A STD track should provide a RAP at least every 500 millisecond (half a second). FF and FR tracks must provide a RAP at each picture. When jumping from one track to another, the video server needs to know where to start playing the new stream. This is determined by performing lookups into index files called Random Access Point Index Files. These files contain information about each possible point of entry into each stream. The RAP Index file is a binary file that contains a series of records describing the RAPs. This information includes the STD track picture number, the MPEG-2 Presentation Time Stamp, and the 64 bit byte offset into the stream.

6 Conclusion

A standardized method for creating VCR functions using MPEG streams has been presented. The advantage of this method is that content creators can specify and develop the desired quality and effects appropriate to the content at hand. Service providers do not have to be involved in the content encoding and quality process. In addition, the network bandwidth required to deliver the VCR functions can be controlled in a computationally efficient manner.

Several data management issues deserve further investigation. The video server buffer management for CBR vs. VBR networks is not well understood. When VCR functionality is added to the set of server features, what is the impact on the server buffer management? What is the most efficient pipeline algorithm for MPEG encoding if VCR files are to be created? What are the network bandwidth allocation considerations for VBR networks providing VCR functions and what are the advantages of VCR files in these networks? These are a few of the many issues that must be resolved for the widespread deployment of video as a network data type.

Acknowledgments

The author would like to acknowledge the work of Al Kovalick and John Shelton of the Hewlett-Packard Video Communications Division for inputs and review of this paper.

Fast Searching by Content in Multimedia Databases

Christos Faloutsos[*]
Department of Computer Science
and Institute for Systems Research (ISR)
University of Maryland at College Park

Abstract

We describe a domain-independent method to search multimedia databases by content. Examples of these searches include 'find all images that look like this graphic drawing (which is a photograph of a sunset)*' in a collection of color images;* 'find stocks that move like Motorola's*' in a collection of stock price movements; and* 'find patterns with stripes containing red and white*' in a collection of retail catalog items.*

In all these applications, we assume that there exists a distance function, which measures the dissimilarity between two objects. Given that, the idea is to extract f numerical features from each object, effectively mapping it into a point in f-dimensional space. Subsequently, any spatial access method (like the R-trees) can be used to search for similar objects (that is, nearby points in the f-d space). Comparing the features corresponds to a 'quick and dirty' test, which will help us exclude a large number of non-qualifying objects. The test could allow for false alarms, but no false dismissals. This implies that the mapping from objects to f-d points should preserve the distance, or, as we show, it should lower-bound *it.*

We briefly show how this idea can be applied to achieve fast searching for time sequences and for color images. Experiments on real or realistic databases show that it is much faster than sequential scanning, while not missing any qualifying objects, as expected from the lower-bounding lemma. Thus, this approach can be used for any *database of multimedia objects, as long as the lower-bounding lemma is satisfied.*

Keywords: *image databases; indexing; spatial access methods; time sequence matching.*

1 Introduction

The problem we focus on is the design of fast searching methods that will search a database of multimedia objects, to locate objects that match a query object, exactly or approximately. Objects can be 2-dimensional color images, time sequences, video clips etc.

Specific applications include medical image databases in 2-d or 3-d (eg., MRI brain scans[4]); financial, marketing and production time sequences (eg., stock-market time sequences [24]); scientific databases with vector fields [9]; audio and video databases [26], DNA/genome databases [3], etc. In such databases, typical queries would be '*find companies whose stock prices move similarly*', or '*find images that have colors similar to a sunset photograph*', or '*find medical X-rays that contain something that has the texture of a tumor*'.

[*]On leave at AT&T Bell Laboratories, Murray Hill, NJ; His research was partially funded by the National Science Foundation under Grants IRI-9205273 and IRI-8958546 (PYI), with matching funds from EMPRESS Software Inc. and Thinking Machines Inc. E-address: `christos@research.att.com, christos@cs.umd.edu`

Searching for similar patterns in such databases as the above is essential, because it helps in predictions, computer-aided medical diagnosis and teaching, hypothesis testing and, in general, in 'data mining' [2] and rule discovery.

To solve these problems, the distance of two objects has to be quantified. We rely on a domain expert to supply such a distance function $\mathcal{D}()$:

Definition 1: Given two objects, O_1 and O_2, the distance (= dis-similarity) of the two objects is denoted by

$$\mathcal{D}(O_1, O_2) \tag{7}$$

For example, if the objects are two (equal-length) time series, the distance $\mathcal{D}()$ could be their Euclidean distance (sum of squared differences).
Similarity queries can be classified into two categories:

Whole Match: Given a collection of N objects $O_1, O_2, \ldots, O_N$ and a query object Q, we want to find those data objects that are within distance ϵ from Q. Notice that the query and the objects are of the same type: for example, if the objects are 512×512 gray-scale images, so is the query.

Sub-pattern Match: Here the query is allowed to specify only part of the object. Specifically, given N data objects (eg., images) $O_1, O_2, \ldots, O_N$, a query (sub-)object Q and a tolerance ϵ, we want to identify the parts of the data objects that match the query. If the objects are, eg., 512×512 gray-scale images (like medical X-rays), in this case the query could be, eg., a 16×16 sub-pattern (eg., a typical X-ray of a tumor).

Additional types of queries include the '*nearest neighbors*' queries (eg., '*find the 5 most similar stocks to IBM's stock*') and the '*all pairs*' queries or '*spatial joins*' (eg., '*report all the pairs of stocks that are within distance ϵ from each other*'). Both the above types of queries can be supported by our approach: As we shall see, we reduce the problem into searching for multi-dimensional points, which will be organized in R-trees; in this case, nearest-neighbor search can be handled with a branch-and-bound algorithm (eg., [33]), and the spatial-join query can be handled with recent, highly fine-tuned algorithms [7]. Thus, we do not focus on nearest-neighbor and 'all-pairs' queries.

For all the above types of queries, the ideal method should fulfill the following requirements:

- it should be *fast*. Sequential scanning and distance calculation with each and every object will be too slow for large databases.
- it should be '*correct*'. In other words, it should return all the qualifying objects, without missing any (i.e., no 'false dismissals'). Notice that 'false alarms' are acceptable, since they can be discarded easily through a post-processing step.
- the proposed method should require a small space overhead.
- the method should be dynamic. It should be easy to insert, delete and update objects.

As we see next, the heart of the proposed approach is to use f feature extraction functions, to map objects into points in f-dimensional space; thus, we can use highly fine-tuned database spatial access methods to accelerate the search. The remainder of the paper is organized as follows. Section 2 gives some background material on past related work, on image indexing and on spatial access methods. Section 3 describes the main ideas for the proposed, generic approach to indexing multimedia objects. Section 5 summarizes the conclusions and lists problems for future research.

2 Survey

As mentioned in the abstract, the idea is to map objects into points in f-d space, and to use multi-attribute access methods (also referred to by *Spatial Access Methods* (SAMs)) to cluster them and to search for them. There are two questions: (a) how to derive good features and (b) how to organize them in SAMs.

In terms of features to use, research in image databases benefits from the large body of work in machine vision on feature extraction and similarity measures see e.g., [5, 10]. There is also a lot of work from the database end (see, eg., [21]). In many cases, the article emphasizes either the vision aspects of the problem, or the indexing issues. Several papers (eg., [27]) comment on the need for increased communication between the vision and the database communities for such problems. See [11] for a survey of papers on feature extraction from the machine vision research.

From the database end, there is a wide variety of multidimensional indexing methods (or Spatial Access Methods - 'SAM's). The prevailing methods form three classes [23]: (a) R-trees [18] and related tree-based methods (k-d-B-trees [31], cell-trees [17], the BANG file [15], hB-trees [25], packed R-trees [32, 22], R^+-trees [34], R^*-trees [6], Hilbert-R-trees [23] etc.) (b) linear quadtrees [16] or, equivalently, the z-ordering [30] or other space filling curves [12] [20], and (c) grid-files [19, 28].

The upcoming, proposed method, can use *any* of the above spatial access methods (SAMs), exactly because it will treat the SAM as a 'black box': The SAM should be able to store multi-dimensional points, and, upon a range query, the SAM should quickly return the data points inside the query region.

Our personal preference is the R-tree family, because the R-tree based methods seem to be most robust for higher dimensions [11]. In the implementations reported in [13, 11] we used the R^*-tree.

3 Proposed method

To illustrate the basic idea, we shall focus on 'whole match' queries. There, the problem is defined as follows:

- we have a collection of N objects: $O_1, O_2, \ldots, O_N$
- the distance/dis-similarity between two objects (O_i, O_j) is given by the function $\mathcal{D}(O_i, O_j)$, which can be implemented as a (possibly, slow) program
- the user specifies a query object Q, and a tolerance ϵ

Our goal is to find the objects in the collection that are within distance ϵ from the query object. An obvious solution is to apply sequential scanning: For each and every object O_i ($1 \leq i \leq N$), we can compute its distance from Q and report the objects with distance $\mathcal{D}(Q, O_i) \leq \epsilon$.
However, sequential scanning may be slow, for two reasons:

1. the distance computation might be expensive. For example, the editing distance in DNA strings requires a dynamic-programming algorithm, which grows like the product of the string lengths (typically, in the hundreds or thousands, for DNA databases).
2. the database size N might be huge.

Thus, we are looking for a faster alternative. The proposed approach is based on two ideas, each of which tries to avoid each of the two disadvantages of sequential scanning:

- a 'quick-and-dirty' test, to discard quickly the vast majority of non-qualifying objects (possibly, allowing some false-alarms)
- the use of Spatial Access Methods, to achieve faster-than-sequential searching, as suggested by Jagadish [21].

The case is best illustrated with an example. Consider a database of time series, such as yearly stock price movements, with one price per day. Assume that the distance function between two such series S and Q is the Euclidean distance

$$\mathcal{D}(S,Q) \equiv \left(\sum_{i=1} (S[i] - Q[i])^2 \right)^{1/2} \tag{8}$$

where $S[i]$ stands for the value of stock S on the i-th day. Clearly, computing the distance of two stocks will take 365 subtractions and 365 squarings in our example.

The idea behind the 'quick-and-dirty' test is to characterize a sequence with a single number, which will help us discard many non-qualifying sequences. Such a number could be, eg., the average stock price over the year: Clearly, if two stocks differ in their averages by a large margin, it is impossible that they will be similar. The converse is not true, which is exactly the reason we may have false alarms. Numbers that contain some information about a sequence (or a multimedia object, in general), will be referred to as '*features*' for the rest of this paper. Using a good feature (like the 'average', in the stock-prices example), we can have a quick test, which will discard many stocks, with a single numerical comparison for each sequence (a big gain over the 365 subtractions and squarings that the original distance function requires).

If using one feature is good, using two or more features might be even better, because they may reduce the number of false alarms (at the cost of making the 'quick-and-dirty' test a bit more elaborate and expensive). In our stock-prices example, additional features might be, eg., the standard deviation, or, even better, some of the discrete Fourier transform (DFT) coefficients (see [13]).

The end result of using f features for each of our objects is that we can map each object into a point in f-dimensional space. We shall refer to this mapping as $\mathcal{F}()$ (for 'F'eature):

Definition 2: Let $\mathcal{F}()$ be the mapping of objects to f-d points, that is $\mathcal{F}(O)$ will be the f-d point that corresponds to object O.

This mapping provides the key to improve on the second drawback of sequential scanning: by organizing these f-d points into a spatial access method, we can cluster them in a hierarchical structure, like the R^*-trees. Upon a query, we can exploit the R^*-tree, to prune out large portions of the database that are not promising. Such a structure will be referred to by *F-index* (for 'Feature index'). Thus, we do not even *have* to do the quick-and-dirty test on all of the f-d points!

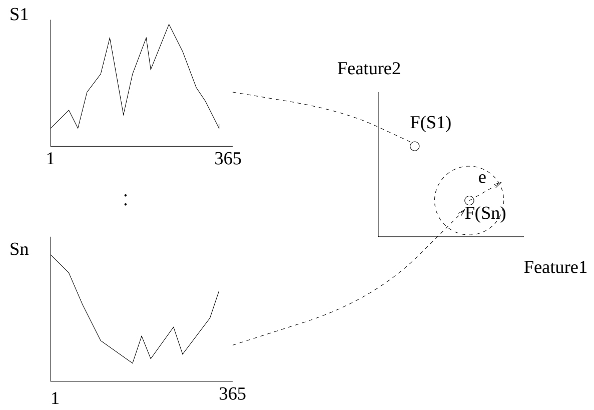

Figure 1: Illustration of basic idea: a database of sequences S1, ... Sn; each sequence is mapped to a point in feature space; a query with tolerance ϵ becomes a sphere of radius ϵ.

Figure 1 illustrates the basic idea: Objects (eg., time series that are 365-points long) are mapped into 2-d points (eg., using the average and the standard-deviation as features). Consider the 'whole match' query that requires all the objects that are similar to S_n within tolerance ϵ: this query becomes an f-d sphere in feature space, centered on the image $\mathcal{F}(S_n)$ of S_n. Such queries on multidimensional points is exactly what R-trees and other SAMs are designed to answer efficiently. More specifically, the search algorithm for a whole match query is as follows:

Algorithm 1: Search an F-index:

1. map the query object Q into a point $\mathcal{F}(Q)$ in feature space
2. using the SAM, retrieve all points within the desired tolerance ϵ from $\mathcal{F}(Q)$.
3. retrieve the corresponding objects, compute their actual distance from Q and discard the false alarms.

Intuitively, an F-index has the potential to relieve both problems of the sequential scan, presumably resulting into much faster searches. The only step that we have to be careful with is that the mapping $\mathcal{F}()$ from objects to f-d points does not distort the distances. Let $\mathcal{D}()$ be the distance function of two objects, and $\mathcal{D}_{feature}()$ be the (say, Euclidean) distance of the corresponding feature vectors. Ideally, the mapping should preserve the distances exactly, in which case the SAM will have neither false alarms nor false dismissals. However, requiring perfect distance preservation might be difficult: For example, it is not obvious which features we have to use to match the editing distance between two DNA strings. Even if the features are obvious, there might be practical problems: for example, in the stock-price example, we could treat every sequence as a 365-dimensional vector; although in theory a SAM can support an arbitrary number of dimensions, in practice they all suffer from the 'dimensionality curse', as discussed in the survey section.

The crucial observation is that we can guarantee that the 'F-index' method will not result in any false dismissals, if the distance in feature space matches or underestimates the distance between two objects. Intuitively, this means that our mapping $\mathcal{F}()$ from objects to points *should make things look closer* (ie., it should be a contractive mapping).

Mathematically, let O_1 and O_2 be two objects (e.g., same-length sequences) with distance function $\mathcal{D}()$ (e.g., the Euclidean distance) and $\mathcal{F}(O_1)$, $\mathcal{F}(O_2)$ be their feature vectors (e.g., their first few Fourier coefficients), with distance function $\mathcal{D}_{feature}()$ (e.g., the Euclidean distance, again). Then we have:

Lemma 3: To guarantee no false dismissals for whole-match queries, the feature extraction function $\mathcal{F}()$ should satisfy the following formula:

$$\mathcal{D}_{feature}(\mathcal{F}(O_1), \mathcal{F}(O_2)) \leq \mathcal{D}(O_1, O_2) \tag{9}$$

Proof: Let Q be the query object, O be a qualifying object, and ϵ be the tolerance. We want to prove that if the object O qualifies for the query, then it will be retrieved when we issue a range query on the feature space. That is, we want to prove that

$$\mathcal{D}(Q, O) \leq \epsilon \Rightarrow \mathcal{D}_{feature}(\mathcal{F}(Q), \mathcal{F}(O)) \leq \epsilon \tag{10}$$

However, this is obvious, since

$$\mathcal{D}_{feature}(\mathcal{F}(Q), \mathcal{F}(O)) \leq \mathcal{D}(Q, O) \leq \epsilon \tag{11}$$

Thus, the proof is complete. $\square$

We have just proved that lower-bounding the distance works correctly for range queries. Will it work for the other queries of interest, like 'all-pairs' and 'nearest neighbor' ones? The answer is affirmative in both cases: An 'all-pairs' query can easily be handled by a 'spatial join' on the points of the feature space: using a similar

reasoning as before, we see that the resulting set of pairs will be a superset of the qualifying pairs. For the nearest-neighbor query, the following algorithm guarantees no false dismissals: (a) find the point $\mathcal{F}(P)$ that is the nearest neighbor to the query point $\mathcal{F}(Q)$ (b) issue a range query, with query object Q and radius $\epsilon = \mathcal{D}(Q, P)$ (ie, the actual distance between the query object Q and data object P.
In conclusion, the proposed generic approach to indexing multimedia objects for fast similarity searching is as follows (named '*GEMINI*' for *GEneric Multimedia object INdexIng*):

Algorithm 2 ('GEMINI'): GEneric Multimedia object INdexIng approach:

1. determine the distance function $\mathcal{D}()$ between two objects
2. find one or more numerical feature-extraction functions, to provide a 'quick and dirty' test
3. prove that the distance in feature space *lower-bounds* the actual distance $\mathcal{D}()$, to guarantee correctness
4. use a SAM (eg., an R^*-tree), to store and retrieve the f-d feature vectors

The first two steps of GEMINI deserve some more discussion: The first step involves a domain expert. The methodology focuses on the *speed* of search only; the quality of the results is completely relying on the distance function that the expert will provide. Thus, GEMINI will return *exactly the same* response-set (and therefore, the same quality of output, in terms of precision-recall) with what the sequential scanning of the database would provide; the only difference is that GEMINI will be faster.

The second step of GEMINI requires intuition and imagination. It starts by trying to answer the question (referred to as the '*feature-extracting*' question for the rest of this work):

> **'Feature-extracting' question:** *If we are allowed to use only one numerical feature to describe each data object, what should this feature be?*

The successful answers to the above question should meet two goals: (a) they should facilitate step 3 (the distance lower-bounding) and (b) they should capture most of the characteristics of the objects.

Next, we present briefly two case studies.

4 Case studies

We have used the above approach for retrieval by content in two environments, time sequences and 2-d color images.

Time Sequences: There [1, 13], we used the Euclidean distance as the dis-similarity measure, which has been widely used for time-series forecasting [8]. For features, we used the first few coefficients for of the Discrete Fourier Transform (DFT) [29], and we showed that they lower-bound the actual distance, thanks to Parseval's theorem. Specifically, for a signal $\vec{x} = [x_i]$, $i = 0, \ldots, n-1$, let X_F denote the n-point DFT coefficient at the F-th frequency ($F = 0, \ldots, n-1$). Parseval's theorem [29] states that the DFT preserves the energy of a signal (sum of squares of its entries), as well as the Euclidean distance between two signals:

$$\sum_{i=0}^{n-1} |x_i - y_i|^2 = \sum_{F=0}^{n-1} |X_F - Y_F|^2 \tag{12}$$

where $\vec{X}$ and $\vec{Y}$ are the Fourier transforms of $\vec{x}$ and $\vec{y}$ respectively. Thus, if we keep the first $f(\leq n)$ coefficients of the DFT as the features, we lower-bound the actual distance:

$$\mathcal{D}_{feature}(\mathcal{F}(\vec{x}), \mathcal{F}(\vec{y})) = \sum_{F=0}^{f-1} |X_F - Y_F|^2 \leq \sum_{F=0}^{n-1} |X_F - Y_F|^2 = \sum_{i=0}^{n-1} |x_i - y_i|^2 \equiv \mathcal{D}(\vec{x}, \vec{y}) \qquad (13)$$

because we ignore positive terms from Equation 8. Thus, there will be *no false dismissals*, according to Lemma 3.

Timing experiments showed that the proposed method outperforms the sequential scanning for ‘whole-match’, ‘all-pairs’ and sub-pattern match queries. As expected, our method introduces some false alarms; however, the cumulative time to traverse the F-index and to clean-up the false alarms is still much smaller than the response time of the sequential scanning.

Color Images: We applied GEMINI on color image databases [11], as part of the QBIC project of IBM [14]. There, the dis-similarity measure was the so-called *color-histogram* distance: After deciding on a number k of colors (eg., k=256), the color histogram of an image is a k-dimensional vector, where each entry is the count of pixels that have the specific color. Once these histograms are computed, one method to measure the distance between two histograms ($k \times 1$ vectors) $\vec{x}$ and $\vec{y}$ is given by

$$d^2_{hist}(\vec{x}, \vec{y}) = (\vec{x} - \vec{y})^t \mathcal{A}(\vec{x} - \vec{y}) = \sum_{i}^{k} \sum_{j}^{k} a_{ij}(x_i - y_i)(x_j - y_j) \qquad (14)$$

where the superscript t indicates matrix transposition, and the color-to-color similarity matrix $\mathcal{A}$ has entries a_{ij} which describe the similarity between color i and color j.

The problem with the color-histogram distance is that it involves ‘cross-talk’ between the features: For example, the ‘bright-red’ color is similar to the ‘orange’ color, as well as to the ‘pink’ color etc.; thus, when comparing two color histograms, we have to take these similarities into account. This ‘cross-talk’ among features precludes the use of spatial access methods.

We solved the problem by asking the ‘feature-extracting’ question. Since we have three color components, (eg., Red, Green and Blue), we considered the average amount of red, green and blue in a given color image. We showed that the Euclidean distance between the average RGB vectors lower-bounds the color-histogram distance [11], and we presented timing experiments on a collection of 924 color images: There, sequential scanning requires roughly 10 seconds, while our method requires from a fraction of a second up to 4 seconds.

5 Conclusions

We have presented a generic method (the ‘*GEMINI*’ approach) to accelerate queries by content on image databases and, more general, on multimedia databases. Target queries are, eg., ‘*find images with a color distribution of a sunset photograph*’; or, ‘*find companies whose stock-price moves similarly to a given company’s stock*’.

The method expects a distance function $\mathcal{D}()$ (given by domain experts), which should measure the dis-similarity between two images or objects O_1, O_2. We mainly focus on *whole match* queries (that is, *queries by example*, where the user specifies the ideal object and asks for all objects that are within distance ϵ from the ideal object). Extensions to other types of queries (nearest neighbors, ‘all pairs’ and sub-pattern match) are briefly discussed. The ‘*GEMINI*’ approach combines two ideas:

- The first is to devise a ‘*quick and dirty*’ test, which will eliminate several non-qualifying objects. To achieve that, we should extract f numerical features from each object, which should somehow describe the object (for example, the first few DFT coefficients for a time sequence, or for a gray-scale image). The key question to ask is *‘If we are allowed to use only one numerical feature to describe each data object, what should this feature be?’*

- The second idea is to further accelerate the search, by organizing these f-dimensional points using state-of-the art spatial access methods ('SAMs') [21], like the R^*-trees. These methods typically group neighboring points together, thus managing to discard large un-promising portions of the address space early.

The above two ideas achieve fast searching. We went further, and we considered the condition under which the above method will be not only fast, but also *correct*, in the sense that it will not miss any qualifying object (false alarms are acceptable, because they can be discarded, with the obvious way). Specifically, we proved the *lower-bounding* lemma, which intuitively states that the mapping $\mathcal{F}()$ of objects to f-d points should *make things look closer*.

We briefly discussed how to apply the method for a variety of environments, and specifically, 2-d color images and 1-d time sequences. Experimental results on real or realistic data confirmed both the correctness as well as the speed-up that our approach provides.

Future work involves the application of the method in other, diverse environments, like voice and video databases, DNA databases, etc.. The interesting problems in these applications are to find the details of the distance functions in each case, and to design features that will lower-bound the corresponding distance tightly.

References

[1] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. Efficient similarity search in sequence databases. In *Foundations of Data Organization and Algorithms (FODO) Conference*, Evanston, Illinois, October 1993. also available through anonymous ftp, from olympos.cs.umd.edu: ftp/pub/TechReports/fodo.ps.

[2] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. *Proc. of VLDB Conf.*, pages 487–499, September 1994.

[3] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. A basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[4] Manish Arya, William Cody, Christos Faloutsos, Joel Richardson, and Arthur Toga. Qbism: Extending a dbms to support 3d medical images. *Tenth Int. Conf. on Data Engineering (ICDE)*, pages 314–325, February 1994.

[5] D. Ballard and C. Brown. *Computer Vision*. Prentice Hall, 1982.

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. *ACM SIGMOD*, pages 322–331, May 1990.

[7] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. Multi-step processing of spatial joins. *ACM SIGMOD*, pages 197–208, May 1994.

[8] M. Castagli and S. Eubank. *Nonlinear Modeling and Forecasting*. Addison Wesley, 1992. Proc. Vol. XII.

[9] Mathematical Committee on Physical and NSF Engineering Sciences. *Grand Challenges: High Performance Computing and Communications*. National Science Foundation, 1992. The FY 1992 U.S. Research and Development Program.

[10] R.O. Duda and P.E. Hart. *Pattern Classification and Scene Analysis*. Wiley, New York, 1973.

[11] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *Journal of Intell. Inf. Systems*, 3(3/4):231–262, July 1994.

[12] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. *Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 247–252, March 1989. also available as UMIACS-TR-89-47 and CS-TR-2242.

[13] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. *Proc. ACM SIGMOD*, pages 419–429, May 1994. 'Best Paper' award; also available as CS-TR-3190, UMIACS-TR-93-131, ISR TR-93-86.

[14] Myron Flickner, Harpreet Sawhney, Wayne Niblack, Jon Ashley, Qian Huang, Byron Dom, Monika Gorkani, Jim Hafner, Denis Lee, Dragutin Petkovic, David Steele, and Peter Yanker. Query by image and video content: the qbic system. *IEEE Computer*, 28(9):23–32, September 1995.

[15] Michael Freeston. The bang file: a new kind of grid file. *Proc. of ACM SIGMOD*, pages 260–269, May 1987.

[16] I. Gargantini. An effective way to represent quadtrees. *Comm. of ACM (CACM)*, 25(12):905–910, December 1982.

[17] O. Gunther. The cell tree: an index for geometric data. Memorandum No. UCB/ERL M86/89, Univ. of California, Berkeley, December 1986.

[18] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Proc. ACM SIGMOD*, pages 47–57, June 1984.

[19] K. Hinrichs and J. Nievergelt. The grid file: a data structure to support proximity queries on spatial objects. *Proc. of the WG'83 (Intern. Workshop on Graph Theoretic Concepts in Computer Science)*, pages 100–113, 1983.

[20] H.V. Jagadish. Linear clustering of objects with multiple attributes. *ACM SIGMOD Conf.*, pages 332–342, May 1990.

[21] H.V. Jagadish. A retrieval technique for similar shapes. *Proc. ACM SIGMOD Conf.*, pages 208–217, May 1991.

[22] Ibrahim Kamel and Christos Faloutsos. On packing r-trees. *Second Int. Conf. on Information and Knowledge Management (CIKM)*, November 1993.

[23] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An Improved R-tree Using Fractals. In *Proceedings of VLDB Conference,*, pages 500–509, Santiago, Chile, September 1994.

[24] Blake LeBaron. Nonlinear forecasts for the s\&p stock index. In M. Castagli and S. Eubank, editors, *Nonlinear Modeling and Forecasting*, pages 381–393. Addison Wesley, 1992. Proc. Vol. XII.

[25] David B. Lomet and Betty Salzberg. The hb-tree: a multiattribute indexing method with good guaranteed performance. *ACM TODS*, 15(4):625–658, December 1990.

[26] A. Desai Narasimhalu and Stavros Christodoulakis. Multimedia information systems: the unfolding of a reality. *IEEE Computer*, 24(10):6–8, October 1991.

[27] Wayne Niblack, Ron Barber, Will Equitz, Myron Flickner, Eduardo Glasman, Dragutin Petkovic, Peter Yanker, Christos Faloutsos, and Gabriel Taubin. The qbic project: Querying images by content using color, texture and shape. *SPIE 1993 Intl. Symposium on Electronic Imaging: Science and Technology, Conf. 1908, Storage and Retrieval for Image and Video Databases*, February 1993. Also available as IBM Research Report RJ 9203 (81511), Feb. 1, 1993, Computer Science.

[28] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM TODS*, 9(1):38–71, March 1984.

[29] Alan Victor Oppenheim and Ronald W. Schafer. *Digital Signal Processing*. Prentice-Hall, Englewood Cliffs, N.J., 1975.

[30] J. Orenstein. Spatial query processing in an object-oriented database system. *Proc. ACM SIGMOD*, pages 326–336, May 1986.

[31] J.T. Robinson. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. *Proc. ACM SIGMOD*, pages 10–18, 1981.

[32] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. *Proc. ACM SIGMOD*, May 1985.

[33] Nick Roussopoulos, Steve Kelley, and F. Vincent. Nearest Neighbor Queries. *Proc. of ACM-SIGMOD*, pages 71–79, May 1995.

[34] T. Sellis, N. Roussopoulos, and C. Faloutsos. The r+ tree: a dynamic index for multi-dimensional objects. In *Proc. 13th International Conference on VLDB*, pages 507–518, England,, September 1987. also available as SRC-TR-87-32, UMIACS-TR-87-3, CS-TR-1795.

CALL FOR PAPERS

Fourth International Conference on PARALLEL AND DISTRIBUTED INFORMATION SYSTEMS

December 18-20, 1996, Eden Roc Resort & Spa, Miami Beach, Florida
Sponsored by: IEEE Computer Society, IEEE Technical Committee on Data Engineering, ACM (pending approval)

Parallel and distributed database technology is at the heart of many mission-critical applications such as online transaction processing, data warehousing, business workflow management, interoperable information systems, and information brokering in global networks. While commercial systems in this arena are gradually maturing, new challenges are posed by the growing demand for large-scale, enterprise-wide solutions and the proliferation of services on the "information superhighway." Future parallel and distributed information systems will have to support millions of clients and will face tremendous scalability challenges with regard to massive data volume, performance, availability, and also administration and long-term maintenance.

The scope of this conference includes all aspects of parallelism and distribution in information systems. Submissions are solicited particularly but not exclusively on the following topics:

* online analytical processing and data mining
* parallel query optimization and scheduling
* parallel and distributed object management
* parallel and distributed document management
* storage system administration and tuning
* data management on networks of workstations
* extended transaction models and management
* interoperability and heterogeneous systems
* directory services and information discovery
* distributed real-time and active database systems
* applications on parallel and distributed systems
* data distribution and replication
* multimedia databases
* business workflow management
* tertiary storage management
* distributed middleware
* WWW-based query processing
* information and resource brokering
* mobile and disconnected clients
* highly available large-scale systems

Submission Instruction:

Original papers on the above topics are invited. These should be no longer than 25 double-spaced pages with no smaller than 11 point fonts.

To promote cross-fertilization between works in progress, short synopses of substantial projects are invited as well. These should be no longer than 5 double-spaced pages with no smaller than 11 point fonts. A synopsis should contain enough information for the program committee to understand the scope of the problem being addressed and to evaluate the novelty of the proposed solution. The current status of the project should also be indicated. Time and space will be provided in the program and proceedings to discuss innovative aspects of selected projects.

Please submit six copies of an original paper or a project synopsis to program co-chair Jeffrey Naughton no later than May 1, 1996 (firm "drop-dead" deadline). For further information, please contact the general chair Wei Sun, one of the program co-chairs, Jeffrey Naughton and Gerhard Weikum, or browse http://panda.cs.binghamton.edu/pdis96.html.

In addition, the authors should send electronic mail by March 21, 1996 to pdis@cs.wisc.edu with the title of the manuscript, authors' names, contact author and address (include email and fax information), and an abstract not to exceed 80 words. Authors will be interested to know that a special issue of the Distributed and Parallel Database Journal containing best papers from the PDIS conference is being planned.

Important Dates

Email Submission of Abstract:	March 21, 1996
Submission Deadline:	May 1, 1996 (firm deadline)
Notification of Authors:	August 15, 1996
Tutorials and Conference:	December 18-20, 1996

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398