Adaptive Disk Scheduling in a Multimedia DBMS

by

Ketil Lund

A thesis submitted to Department of Informatics
Faculty of Mathematics and Natural Sciences, University of Oslo, Norway
in partial fulfillment of the requirements for the degree of
“doctor scientarum”

July 10, 2003

In memory of Marius

Thanks for teaching me what really matters in life

Abstract

In recent years, Internet access methods such as xDSL and cable modems have become much cheaper, and the result is that broadband network access is becoming more and more widespread. Consequently, online multimedia services like News-on-Demand (NoD), digital libraries and Learning-on-Demand (LoD) are becoming important elements of the information society.

The scenario of our work is a LoD-system, where a multimedia database management system (MMDBMS) realizes a large repository for multimedia-based learning material. Teachers create multimedia presentations consisting of combinations of multimedia objects, while the students search for and play back these presentations. The MMDBMS must handle a constantly shifting workload, with very diverse requirements; and because the large size of the multimedia data implies that secondary storage must be used, these requirements are also imposed on the storage subsystem.

In this thesis, we show that disk scheduling is an effective way of meeting these storage subsystem requirements. We have developed an adaptive disk scheduler for mixed-media workloads, called APEX, which is specifically targeted at the requirements of a MMDBMS-based LoD-system. APEX is able to provide a number of different service types, support QoS, and at the same time achieve very high disk utilization. This has been made possible through the use of the following four techniques: (1) dynamic queue management, which keeps the overhead of the scheduling framework low, and updates bandwidth reservations according to the requirements of the MMDBMS; (2) an extended token bucket model, which ensures accurate distribution of disk bandwidth; (3) a batch building principle, which submits disk requests in batches, and ensures high disk utilization; and (4) a work-conservation feature that re-distributes unused disk bandwidth without loss of disk efficiency.

We have implemented APEX together with two other disk schedulers in a simulation environment. Our measurements show that for best-effort disk requests, APEX achieves approximately 30% higher throughput and from 30% to 90% lower response times than comparable schedulers, while providing the equal QoS-guarantees for real-time requests. In addition, our analysis shows that APEX does not impose any higher computational complexity than other disk schedulers offering multiple service types.

From our simulations, it is also clear that APEX is able to handle the extensively shifting MMDBMS workload; it makes sure that the disk bandwidth is distributed according to the reservations that are made, and then re-distributes unused bandwidth to where it is needed. In addition, APEX is designed to take advantage of the intelligence of modern disks, leaving the final ordering of requests to the disk itself, and thereby further increasing the disk utilization.

Consequently, our disk scheduling framework is well suited for a demanding environment like a MMDBMS. Furthermore, due to its modularity and dynamic configurability, APEX is well suited for a wide range of other applications, whether they require real-time service, high-throughput, low-latency, or any combination of these.

Acknowledgements

Finally!

The road towards a doctoral degree has indeed been long and winding, and it has been filled with frustration, despair, and pain. However, it has also been a rewarding and broadening experience, and through the years of my work, I have received support from many people, whom I wish to thank.

Above all, I would like to thank my advisers, Professor Dr. Vera Goebel and Professor Dr. Thomas Plagemann, for your constant support, inspiration, enthusiasm and *patience*. You taught me to present my ideas in a clear, concise, and readable way, and you taught me the importance of being *accurate*. Thank you both for always having time for me, both at office and in your home, despite tight schedules.

A special thank you to my fellow student, Dr. Pål Halvorsen - my next door neighbor at UniK. We have had countless valuable discussions and conversations, many serious, and many not so serious, and you have provided much important feedback. We finally made the triple triple-twenty!

I also would like to thank the senior guest scientists in the project, Dr. Denise Ecklund and Dr. Earl F. Ecklund, for spending a lot of time and effort in discussions with me. They provided much inspiration and good feedback on my work. At the same time, I would like to express my gratitude to fellow student Chuanbao Wang for many valuable discussions, and to Professor Dr. Jonathan Walpole and Dr. Carsten Griwodz, who have provided much valuable feedback during the final phase of my work.

I did both my master thesis and my doctoral thesis at UniK (University Graduate Center, Kjeller), University of Oslo, and I am deeply grateful for all the support I have received there, especially during the last two years of my doctoral work. Without the flexibility and goodwill they displayed, finishing my work would have been considerably more difficult. UniK provides a nice environment to work in, and a very personal atmosphere. I wish to express a sincere thank you to all the people there who have supported me; Ivar, Hellfrid, Gerd, Trond, Arild, Kristin, Nina, and Anja.

Furthermore, I am thankful to my fellow project student Dr. Jan Øyvind Aagedal. We have had many interesting discussions, and you provided valuable insight into QoS management.

Last, but definitely not least, I would like to thank my family, and especially my wife Eva. I am grateful that you have been bearing with me when I sat night after night in my office. My two children, Mathea and Sigbjørn, made sure I got some much-needed breaks from my work, putting my mind onto something completely different. Finally, I would like to thank my parents; your support through these years has been invaluable.

Ketil Lund
UniK – University Graduate Center
Department of Informatics
University of Oslo
July 10, 2003

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Broadband network access is becoming more and more popular, as access methods such as xDSL and cable modems become cheaper. Consequently, online multimedia services like News-on-Demand (NoD), digital libraries and Learning-on-Demand (LoD) are becoming important elements of the information society.

These multimedia services are characterized by resource-intensive data, a high degree of user interaction, and need for quality of service (QoS) support. It is a challenge to make sure that all components involved in realizing such services are able to meet the application requirements. In this thesis, we focus on one of these components, namely the storage subsystem. Specifically, we aim to provide a disk scheduler that enables the storage subsystem to meet the requirements of a MMDBMS-based LoD-system.

1.1 Motivation and Background

This thesis is a part of the OMODIS[1] (Object-Oriented Modeling and Database Support for Distributed Multimedia Systems) project [36]. The focus of the project is integration of database system (DBS) technology into distributed multimedia systems, as well as QoS-issues for storage and retrieval of multimedia data. The primary target application areas of the OMODIS project are distance learning and LoD.

[1] The OMODIS Project is funded by the Norwegian Research Council, Distributed IT Systems (DITS) program to UniK and SINTEF, 1996-2001.

The focus of this thesis is QoS-support in the storage subsystem of a multimedia database management system (MMDBMS), and in particular, QoS-aware disk scheduling. The scenario of our work is LoD, and we consider a LoD-system with a server acting as a large repository for multimedia-based presentations, i.e., learning material such as recorded lectures, instructional videos, documentaries, etc. The presentations consist of combinations of multimedia objects, e.g., audio, video, subtitles, pictures, text documents, etc., all stored in the repository. These objects have been uploaded to the repository and then tagged with descriptive metadata. We assume that both syntactic metadata, describing the characteristics of the objects (format, bandwidth requirements, size, etc), and semantic metadata, describing the content of the objects (e.g., the theme of a video clip) are attached to the objects. The former type is required to be able to estimate resource requirements, and the latter to enable content searches and reuse of objects.

The teacher creates a presentation by selecting a set of such multimedia objects, and specifying the temporal and spatial relationships between them. For instance, a video can be accompanied by a separate window displaying documents that are referred to in the video. The author must specify when each document should be displayed, where on the screen, how large the window should be, and for how long the document should be displayed.

Consequently, in addition to storing the multimedia objects and their metadata, the repository must be able to store the temporal and spatial relationships that constitute the presentations. The LoD-server is also responsible for controlling the playback of the presentations. Thus, the server must be able to "understand" the relationships between the multimedia objects, and for this we need a data model that can handle such relationships.

A student typically searches the object- and presentation-metadata stored in the repository for relevant information, and then selects one of the resulting presentations for playback. During the playback, the student can pause, jump to other positions in the presentation, or abort it. Thus, the repository must provide effective query facilities, and it must be able to handle a high degree of interaction.

It is our view, that a LoD-system as described here can benefit greatly from using a MMDBMS to implement the repository (see Figure 1-1), since such systems provide efficient querying facilities, support for complex data models, and ability to control playback of multimedia presentations [8, 93]. Consequently, in the remainder of this thesis, the use of a MMDBMS will be the underlying assumption.

We have identified four types of interaction between the user and the LoD-system: uploading of multimedia objects, authoring of presentations, searching for content, and playback of presentations. These represent a very diverse workload, with very different requirements on the MMDBMS. For instance, uploading a multimedia object, such as a video, benefits from high throughput, but no QoS-guarantees are required. On the other hand, playing back presentations usually implies that continuous media data is streamed from the server to the client. Thus, both high throughput and QoS-guarantees are required, in order to ensure sufficient playback quality. In turn, this means that admission control is required, to allow resource reservation. We assume a centralized admission control for the LoD-system, which uses metadata in the MMDBMS to estimate the resource requirements of the presentation [90].

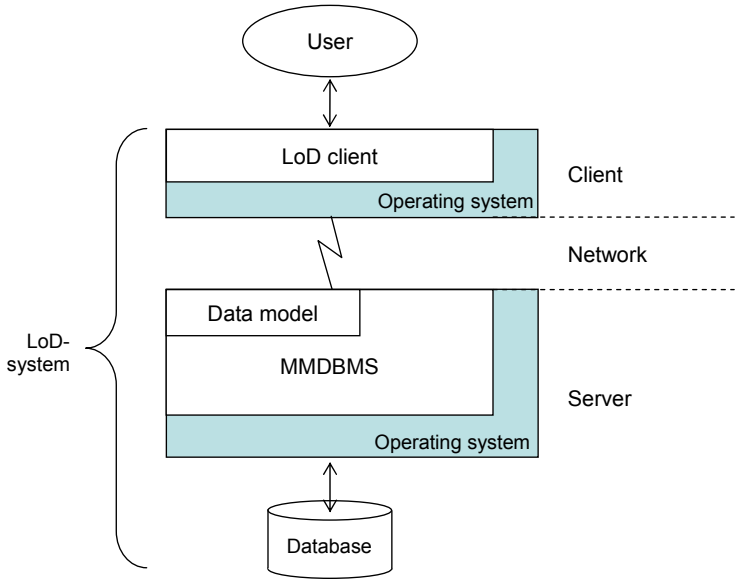

Figure 1-1: LoD-system architecture

The amount of data that such a system manages is very large; especially continuous multimedia data, like video, represents large amounts of data, which means that it must reside on secondary storage, i.e., one or more disks. Consequently, the requirements that apply to the MMDBMS in general, also apply to the storage subsystem, including the secondary storage itself.

1.2 Problem Statement

Given the description of the LoD-system in the previous section, we argue that we need a storage subsystem that provides multiple service types, such as:

- Different degrees of real-time guarantees for playback of continuous media data.

- Low-latency service for user-interaction and system-internal queries (e.g., retrieving index data from disk).
- High throughput for moving large multimedia objects into or out of the database.
- Traditional best-effort service for user queries, such as searching for content.

It is also very important that these different service types are isolated from each other, in order to prevent traffic of different service types to affect each other. For example, downloading a large video should not affect the playback of a presentation.

In addition to supporting different levels of service, the storage subsystem must ensure good utilization of disk bandwidth. This is necessary, since continuous media data often are bandwidth-demanding, and the disks still constitute a bottleneck in multimedia servers [102].

An efficient means of meeting these requirements is through disk scheduling. However, existing disk schedulers are not designed to meet the requirements of a MMDBMS. In particular, the lacking ability to handle large and rapid shifts in the workload is a problem. Even if detailed information about the requirements of the multimedia objects is available in the system catalog, resource planning in a complex system like a MMDBMS is a challenge. For instance, if two users play back the same presentation, with only a small time gap between them, disk bandwidth reserved for the second user may be left unused, because the data fetched for the first user is available in the buffer and can be used by the second user as well. If one of the users interact with the presentation (e.g., making a pause), the time gap between the two presentations may become too large, and the reserved bandwidth must be put to use. Thus, even with detailed knowledge of the nominal bandwidth requirements of the presentations, it is impossible to accurately predict the actual requirements. Consequently, it is necessary for the disk scheduler to be able to make constant adjustments, according to the actual bandwidth requirements, while still making sure that all bandwidth reservations are observed. This ability is usually not seen in existing disk schedulers, or it is realized with a significant loss of disk efficiency.

1.3 Contributions and Claims

The contribution of this thesis is a disk scheduling concept that submits disk requests in batches. By doing so, we achieve very high disk efficiency by leaving the final ordering of the disk requests to the components that has the best qualifications for optimizations, namely the disk and the disk controller. The concept is well suited for MMDBMS

environments with highly varying workload, and we achieve a good tradeoff between QoS-guarantees, low latency, and disk utilization.

We have realized the concept in a disk scheduling framework called APEX (AdaPtive disk schEduler for miXed-media workloads) [56], which is designed specifically for a MMDBMS environment, and which is able to exploit information derived from the system catalog of the MMDBMS, as well as handling the dynamic workload described above.

The thesis is built around four claims about disk scheduling in general and APEX in particular. These claims express the core of our work, and we will prove these claims through this thesis. Below, we present our claims, together with brief supplementary explanations.

Claim 1

It is possible to design, implement, and integrate a disk scheduler in a MMDBMS, in such a way that it can utilize metadata from the MMDBMS to optimize the scheduling of disk requests.

The OMODIS project focuses on using a MMDBMS for storing and managing multimedia data. Such a MMDBMS represents a demanding environment for the storage subsystem: it requires several, diverse service types from the disk, and the generated disk workload is highly varying. In this thesis, we demonstrate that our disk scheduling framework is capable of offering the required service types, as well as optimizing the scheduling of disk requests based on resource predictions made from MMDBMS metadata.

Claim 2

APEX is a highly configurable disk scheduling framework that can be used in a wide variety of contexts.

Different applications can have very different requirements to the disk subsystem. Existing disk schedulers usually offer one, or a few, fixed service types, and there is little room for reconfiguration. We demonstrate that APEX can be easily configured to meet a wide variety of application requirements, such as different levels of QoS-guarantees, different bandwidth allocation paradigms [92], request dropping, and priorities.

Claim 3

APEX offers a superior combination of QoS-support and high utilization of disk bandwidth.

QoS-guarantees and high disk efficiency are contradicting goals for a disk scheduler, and a trade-off is necessary. In addition, most disk schedulers that support more than one service class have a scheme for redistributing unused bandwidth (work-conservation), which usually further reduces disk utilization. APEX uses batching of disk requests, combined with a work-conservation scheme without efficiency loss, in order to achieve a very good trade-off between QoS-guarantees and disk utilization.

Claim 4

Disk scheduling is necessary for providing QoS-support in the storage subsystem, but existing QoS-aware disk schedulers do not exploit the capabilities of modern disks.

Modern, self-managing disks do their own internal scheduling of requests, in order to optimize performance. However, since the disk has no knowledge of different QoS-levels, and is unable to provide different service types to different requests, the internal scheduling is not sufficient to provide QoS-guarantees. Hence, external, host-based scheduling is also required. However, existing QoS-aware disk schedulers are based on detailed knowledge of the disk behavior, and they normally require full control of the order in which disk requests are served. Thus, the disk is left with little freedom to reorganize the requests, and reduced disk efficiency is the result. APEX, on the other hand, relaxes the need for detailed control, within the tolerance of the application, in order to take advantage of the intelligence of modern disks.

1.4 Approach

Our approach for proving that the four claims hold is twofold; with one theoretical part and one experimental part. The theoretical part consists of describing the scenario and the MMDBMS used in the LoD-system. We present the requirements of the user, the multimedia data, and the MMDBMS, and investigate the consequences of these requirements with respect to the storage subsystem. Furthermore, we consider the information that is available from the MMDBMS, to see what is relevant for the disk scheduler; and we analyze the behavior of modern disks. Based on this information, we analyze existing disk schedulers, to investigate their suitability in a MMDBMS context.

We then provide a detailed description of our disk scheduling framework, called APEX. We describe the scheduling principles, as well as how the information from the MMDBMS is used to optimize the distribution of disk bandwidth. Finally, we present the

implementation of APEX, in order to allow the user to assess the modularity and configurability of APEX.

The experimental part of the proof is performed using simulations. By implementing APEX in a simulation environment and applying a workload that is representative for a MMDBMS, the simulations serve as proof of concept for the scheduling principles of APEX.

In addition, we have implemented two existing disk schedulers, C-LOOK and Cello, in the same simulation environment, and using the same workload. For all three schedulers, we compare response times of disk requests, disk throughput, and the ability of the schedulers to maintain a given QoS-level. These comparisons serve to verify the theoretical proofs, and provide an indication in terms of absolute performance.

1.5 Outline

This thesis is organized as follows: Chapter 2 presents the LoD-application that constitutes the scenario, while the system architecture is described in Chapter 3. In particular, we focus on the MMDBMS, presenting both the layered architecture and the functional aspects of the system.

In Chapter 4, we analyze the requirements presented in the two previous chapters, and provide a requirements list for assessment of disk schedulers. Next, in Chapter 5, we use this requirements list to investigate existing disk schedulers.

Chapter 6 presents the scheduling principles of APEX on a conceptual level, and we discuss the influence that external factors in the storage subsystem have on our design. Then, in Chapter 7, we elaborate the details of our scheduling framework, by describing the implementation of APEX, using pseudo-code.

In Chapter 8, we describe our performance evaluation. We present the results of the simulations we have performed, and our analysis of the results. Finally, in Chapter 9, we conclude the thesis by doing a critical assessment of the four claims, and we describe future research directions for APEX.

Chapter 2

LoD Application Scenario

In this chapter, we present the application scenario of our work, namely an online, server-based Learning-on-Demand (LoD) application. We present the functionality of the application, the different types of user interactions, and how these map to operations in the server.

The purpose of this chapter is to provide the reader with sufficient background information about the scenario application to understand the requirements that are imposed on the server. This is important, since the LoD-application and its requirements on the server-side system forms the basis of our work.

2.1 Introduction

Multimedia applications exhibit a wide variety of characteristics, and can be classified based on several different criteria. At a high level, applications can be classified as either *synchronous*, such as videoconferencing, or *asynchronous*, like Video-on-Demand (VoD). The asynchronous applications can be further classified according to the degree of possible user interaction, leading to a range of different application classes. At one extreme, we have the non-interactive applications like broadcast video. This application is used by broadcasting companies, as a substitute for the ordinary video player [89], and for the end-users, i.e., the viewers, no interaction is possible when watching a programme.

VoD applications usually allow some user interaction. In the lower end, Near-VoD applications allow the users to start watching at specific times, while the most advanced

VoD-applications allow the users to start watching at any time, and also perform VCR-interactions.

Some of the most advanced multimedia applications, with respect to user interaction, are those based on hypermedia and multimedia databases [89]. Two examples of such applications are News-on-Demand (NoD) and Learning-on-Demand (LoD). These applications are characterized by a high degree of user interaction, both of VCR-type and for navigating between different (parts of) presentations, as well as searching for specific content.

For instance, in a NoD-application, a typical user browses through headlines or jumps from one story to another, maybe only watching the first few seconds to see if the story is of interest. The user may also skip parts of a story that are not of interest, or search for specific topics.

This class of applications is typically also the most advanced with respect to the content presented to the users. The stored multimedia data is often presented in the form of composite presentations, i.e., presentations consisting of several different multimedia elements (e.g., audio, video, text), which may be presented in a synchronized fashion, both sequentially and in parallel [4].

2.2 Terminology

In order to avoid misunderstandings, we present our understanding of a few central terms that will be used throughout the thesis. In addition, we give a more thorough definition of our understanding of quality of service (QoS), since this is a central notion in our work.

- *Multimedia data type (MMDT)*: Any data type used in a multimedia context is of some multimedia data type. Thus, data types like audio, video, text, images, and animations are all MMDTs. We distinguish between two main classes of MMDTs: (1) *Continuous* MMDTs: These are also known as time-dependent MMDTs, and include data types like audio, video, and animations. The main characteristic of this class is that the presentation of the data is time-dependent, i.e., it changes over time. Although objects of such types appear as continuous to the user, they normally consists of a number of individual elements, presented as a continuous sequence. For instance, video consists of a number of frames. (2) *Discrete* MMDTs: This class is also known as time-independent MMDTs. Typical examples are text and pictures, and common in all such data types is that their presentation is independent of time.

- A *multimedia object* is an object of some MMDT. For instance, video clips, text documents, and images are all multimedia objects. To separate between multimedia objects of discrete and continuous MMDTs, we use the terms *discrete multimedia object* and *continuous multimedia object*, respectively. Many multimedia objects, especially continuous multimedia objects, are complex, in the sense that they consist of a number of elements. For instance, a video object consists of a series of frames.
- A *presentation* is a set of multimedia objects that are presented to the user, with both temporal and spatial orchestration. A very simple example is a video clip with a corresponding audio track. A more complex presentation may consist of audio and video with subtitling, and a separate window showing documents that are being referred to in the video. All this is presented in a synchronized manner, such that, for instance, the correct document is shown at the time it is actually referred to in the video. Thus, a presentation can be defined as a specification of how a set of multimedia objects should be played back in a coordinated fashion.
- *Streaming*: Playback of a continuous multimedia object over a network. Instead of first downloading the object and then playing it back, the object is presented to the user as the data is received. The client normal buffers parts of the data, in order to smooth out jitter caused by the server and the network.
- *Metadata*: Data describing the multimedia objects. For instance, a video can have metadata describing the format of the video, the bandwidth requirements, and the content of the video clip.

2.2.1 Quality of Service

QoS is a central issue when discussing multimedia systems, and before we describe the application that constitutes the scenario of our work, we will present our understanding of QoS.

The concept of QoS was first used within the area of data communication, in order to describe technical characteristics, such as delay and error rate [97]. With the advent of multimedia applications, the QoS concept has been extended to cover end-systems as well, to enable an end-to-end control of the quality level.

There is still no commonly agreed upon definition of QoS, but we have chosen the definition from [97]:

Quality-of-Service represents the set of those quantitative and qualitative characteristics of a distributed multimedia system necessary to achieve the required functionality of an application.

Consequently, QoS concerns all parts of a distributed multimedia system: client system, network, and server. According to [97], there are five categories of QoS-parameters, and these are shown in Table 2-1.

Table 2-1: The five categories of QoS-parameters [97]

Category	Example parameters
Performance-oriented	End-to-end delay and bit-rate (bandwidth)
Format-oriented	Video resolution, frame rate, storage format, and compression scheme
Synchronization-oriented	Skew between the beginning of audio and video sequences
Cost-oriented	Connection and data transmission charges, and copyright fees
User-oriented	Subjective video and sound quality

QoS-parameters from all categories in Table 2-1 are relevant for a LoD-system. For the user, the user- and cost-oriented parameters are the ones that matter, but the user-oriented parameters are in turn dependent on the performance-, format-, and synchronization-oriented parameters. For instance, to perceive the subjective image quality of a video as being satisfactory, the video must be coded in a format and resolution that allows a sufficiently high image quality. Next, the system must be able to provide a sufficiently high bandwidth to support the bandwidth requirement of the video.

The performance-oriented QoS-parameters are typically associated with the lowest layers (i.e., closest to the physical resources) of a system, and therefore impact parameters in most other categories. In this thesis, we focus on disk scheduling, i.e., scheduling of a physical resource, and for this reason, the performance-oriented QoS-parameters are the most important for us. In Chapter 4, we therefore analyze how the user oriented parameters map down to the performance parameters.

2.3 The LoD-Application

Interactive distance learning (IDL) is an evolving paradigm of instruction and learning that attempts to overcome both distance and time constraints found in traditional classroom

learning. Learning-on-Demand (LoD) is one valuable approach for IDL and it extends conventional educational programs, as it introduces the idea of learning by doing. This creates an environment where learners have the necessary tools, computational or otherwise to explore different information spaces, and obtain contextualized and relevant information.

The scenario of our work is a LoD-system (see Figure 2-1), where students use a client application on their local computer to access a LoD-server that acts as a large repository for multimedia-based learning material such as recorded lectures, instructional videos, documentaries, etc. The students can search for relevant material, and play it back in the form of presentations

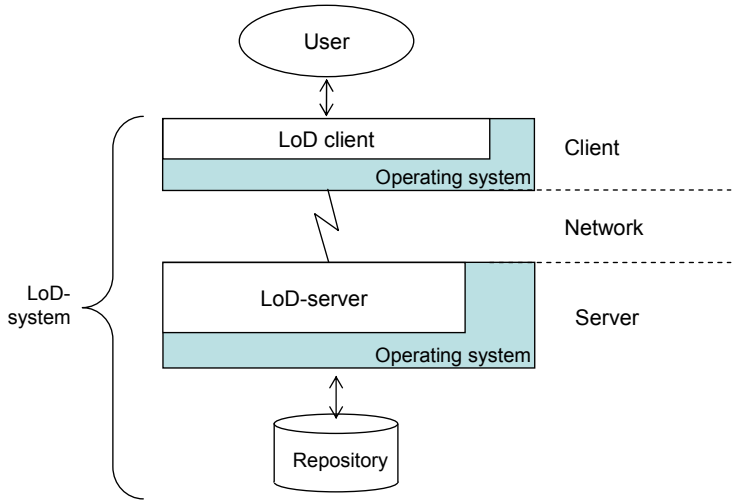

Figure 2-1: General LoD-system architecture

2.3.1 Functionality

We separate between two main user groups in such a LoD-system:

- *Content providers*, i.e., the teachers/instructors, who upload multimedia objects to the repository, review such objects, and combine them into presentations.
- *Content consumers*, i.e., the students, who search for relevant information and play back presentations.

Each user, whether it is a content provider or content consumer, uses his or her local machine, running a LoD-client application, to connect to the LoD-server over a network, as shown in Figure 2-1. The interaction that the two user groups have with the LoD-system can be classified into four main types: search for content, playback of content, authoring of

multimedia objects, and authoring of presentations. Below, we describe each of these operations in more detail:

Search for Content

Students and authors can search the repository for specific content, by specifying, for instance, a particular subject that they want information about. In practice, this operation implies searching the metadata associated with multimedia objects and presentations, and the result is a list of presentations and/or individual multimedia objects that all deals with the subject in question. In addition, the students can browse the repository, by following hyperlinks in the presentations.

Since these types of operations involve a lot of user interaction, they should preferably be served quickly, i.e., with low response times.

Playback of Content

When the student has found a presentation or an author has found an individual multimedia object of interest, he or she starts a playback. If continuous multimedia objects are involved in the playback, we assume that the data is streamed to the user.

During a playback, the user can interact with the presentation, such as pausing and resuming, fast forward and rewind, following hyperlinks to other parts of the presentation (or other presentations), and choosing between alternative sub-presentations. Thus, a playback can imply considerable user interaction.

It is important that the presentation is displayed in accordance with the specifications in the presentation and the requirements of the included multimedia objects. For instance, the presentation may specify that a text document should be displayed at a specific time during playback of a video, such as when a person in the video refers to the document. Thus, it is important that the LoD-system is able to transfer the document from the server to the client, and present it to the user at the correct time. In addition, objects of continuous MMDTs, such as video, require a minimum of resources, e.g., disk and network bandwidth, to be displayed properly.

Authoring of Multimedia Objects

Authoring of multimedia objects requires specialized tools, e.g., video editing programs and image editors, and we do not consider such tools as a natural part of the LoD-system. Thus, we assume that all such authoring takes place outside of the system. For instance, a

video is first transferred from video-tape to disk, and then edited, using a suitable program. When the editing is finished, the resulting video clip is uploaded to the repository.

In general, new multimedia objects are created with suitable tools, and then *checked in* (uploaded) to the repository. Likewise, if an existing multimedia object is to be edited, it is first *checked out* (downloaded) from the repository, edited with a suitable tool, and then checked back in. We assume that such check-in- and check-out-operations are performed as "data dumps", i.e., instead of streaming the data, it is transferred as fast as possible, but normally without QoS-requirements.

After a new or edited multimedia object has been checked into the repository, descriptive metadata must be added to the object. Parts of this metadata, such as indexing of text, as well as deriving characteristics like size, format, and average and maximum bandwidth requirements can be automatically added. Also more complex tasks can be automated, such as analyzing a continuous multimedia object with variable bit-rate, in order to derive bandwidth requirements as a function of time. Since these tasks are performed automatically, they can run as background jobs, with no QoS-requirements

Metadata that covers the semantic aspects for non-textual objects, e.g., descriptions of the content of a picture or a video, must normally be added manually. Thus, we assume that the author adds such data, after the check-in is completed. Being an interactive operation, the server should preferably respond quickly to the user-actions.

Authoring of Presentations

Creating a presentation means specifying temporal and spatial relationships between multimedia objects. Since all such objects are stored in a common repository, authors can reuse multimedia objects in different presentations.

Existing multimedia objects are selected by searching the metadata of the multimedia objects, similar to a content search. In addition, the resulting objects will often be played back for review, as well as finding the correct start- and end-points for use in the presentation. Obviously, this type of playback normally implies much user interaction, and as such, the operations should be served quickly by the server.

When the author has selected the relevant multimedia objects, the presentation is specified. This means that the author specifies the spatial layout of the multimedia objects, for instance, how a video window and a document window should be displayed on the screen. In addition, the temporal relationships are specified, i.e., *when* the different objects should be displayed, relative to each other. As opposed to multimedia object authoring, we

assume that presentation authoring is performed within the LoD-system. This is natural, since it is the LoD-system itself that coordinates the playback of presentations, based on the specifications made by the author, and since the specification refers to objects stored in the repository.

2.3.2 Server Operations

In the previous sub-section, we described the four main types of interaction between the user and the LoD-application. From this description, we saw that several of the interaction types involved the same operations on the server. We have identified four server operations that are required to handle the user operations, namely metadata retrieval, multimedia playback, multimedia authoring, and metadata authoring.

These four server operations together constitute the user-generated workload on the LoD-server. However, it is reasonable to expect far more content consumers (i.e., students) than providers accessing the system. Thus, we expect the metadata retrieval and multimedia playback operations to constitute the prevailing part of the workload.

Since the focus of our work is the behavior of the server, we will concentrate on the server operations in the remainder of the thesis. Below we describe these four operations in more detail.

Metadata retrieval

This operation normally occurs in authoring of presentations and in searches for content. In both cases, metadata stored in the repository is scanned, in order to find multimedia objects that match the given search criteria. The actual amount of data that needs to be searched depends on the storage structures used, for example, the extent to which metadata is indexed.

Multimedia playback

This operation usually takes place when a student plays back a presentation, but it is also used when content providers review individual multimedia objects, during authoring of presentations. The server is responsible for making the necessary data available to the client application at the correct time, according to the presentation specification, and the tolerance of the user. However, we assume that the final, fine-grained synchronization is performed on the client side.

Multimedia authoring

This corresponds to the check-in and check-out operations performed as part of authoring of multimedia objects. From the server point of view, the data in question should be transferred as fast as possible to or from the repository. The multimedia objects can be relatively large; a video clip, for example, can often reach a size of several hundred megabytes. Thus, the transfer will usually imply reading from or writing to disk, and disk throughput is therefore important.

Metadata authoring

This operation is used both when adding metadata to multimedia objects and when authoring presentations. We assume that the metadata consists of descriptive text and keywords, and therefore constitutes a relatively small amount of data. The operation itself is also small, since only one, or a few objects are updated at a time.

2.4 Summary

In this chapter, we have presented the LoD-application that forms the scenario of our work. We have described the four main types of user interaction, and described how these are mapped into four types of server interactions.

In the next chapter, we describe the architecture and functionality of the server-side of the LoD-system, using the functionality described in this chapter as basis.

Chapter 3

MMDBMS-Support for LoD

In this chapter, we focus on the architecture of the LoD-system, consisting of a client-application communicating with a LoD-server over a network. The server is realized using a multimedia database management system (MMDBMS), and the main focus of the chapter is the architecture and functionality of this MMDBMS.

We start by giving a short description of the client side, including our assumptions about the client capabilities. Next, we describe a reference DBMS architecture [44], which we use as a framework for describing the MMDBMS. Furthermore, we present the data model we use, which is called TOOMM (Temporal Object-Oriented MultiMedia data model) [35]. Finally, we present a thorough description of the MMDBMS architecture, and in particular the functional aspects, i.e., how the server operations are realized.

The purpose of this chapter is to provide the reader with an understanding of the composition of the LoD-system, while the presentation of the data model serves to clarify the modeling and structuring possibilities in the MMDBMS.

3.1 Introduction

The focus of the OMODIS project has been the integration of database systems (DBSs) into QoS-aware, distributed multimedia systems, and this also sets the background for the thesis. Thus, it is a fundamental assumption in our work that the repository of the LoD-system is realized using a MMDBMS [26] (see Figure 3-1). This assumption also provides us with several advantages with respect to the LoD-application [112]:

- Ability to handle an advanced data model like TOOMM. Thus, the MMDBMS is able to support explicit modeling and storage of the temporal and spatial relationships between multimedia objects.
- Support for both ad hoc queries (search for content) and pre-compiled queries (browsing).
- Concurrency control, which is necessary to allow multiple users, both authors and students, to access the system concurrently, in a controlled fashion.
- Fault tolerance through the use of transactions and error recovery.

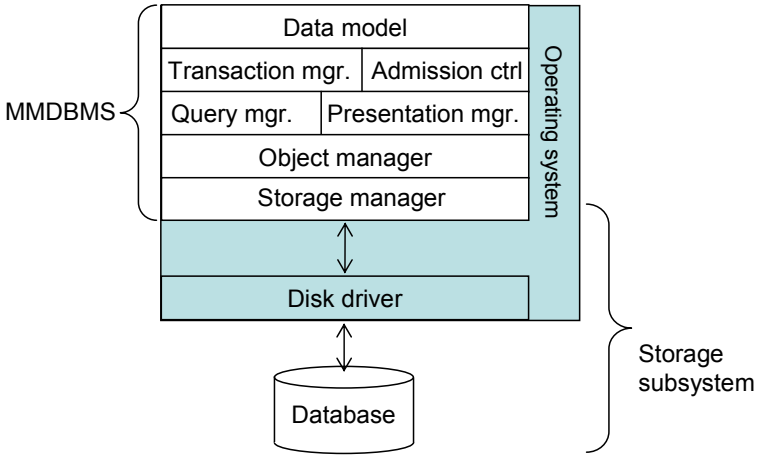

Figure 3-1: Overall architecture or the LoD-server

The database constituting the LoD-repository contains a large set of multimedia objects, which have been checked into (uploaded to) the database. In addition, descriptive metadata has been associated with each multimedia object. The authors, e.g., the teachers, create presentations by specifying the temporal and spatial relationships between multimedia objects, and these specifications are also stored in the database. When students search for information about a certain topic, the result is a set of presentations that contain multimedia objects covering the given topic.

When a user plays back a presentation, the MMDBMS is responsible for delivering the multimedia objects at the correct time. Thus, continuous multimedia objects are streamed over the network, while discrete multimedia objects are delivered in their entirety, shortly before they are to be presented. The LoD-client then handles the final coordination when displaying a presentation to the user. In other words, while the MMDBMS is responsible for the course-grained synchronization during data delivery, the LoD-client handles the fine-grained synchronization required when displaying the presentation.

3.2 Client Architecture

The client-system consists of a LoD-client application running on suitable hardware, such as a standard PC with multimedia capabilities. The client application has to enable users to interactively browse through lectures, to use pre-defined queries, to formulate ad hoc queries, and to specify QoS-requirements both as part of queries and as part of a general user profile. Furthermore, the client has the final responsibility for displaying the presentations to the user. Thus, it must support QoS and synchronized playback. In Figure 3-2, we show the architecture of the client-system [26].

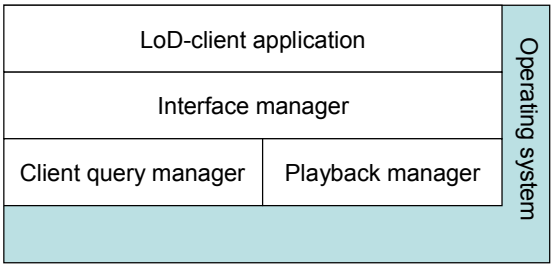

Figure 3-2: Layered architecture of the LoD-client

The interface manager is responsible for invoking MMDBMS interfaces and passing of input parameters and output results. The client query manager formulates complete multimedia queries, based on interactions with the end-user. Since a selection query may contain different types of multimedia data, this is done by integrating information from different multimedia devices on the client system into a single query. In addition, the client query manager is responsible for query refining, and checking that the queries are well-formed.

The playback manager receives data from the server, and presents it on the local client, in a best possible way, according to the capabilities of the client system. This includes responsibility for playback synchronization, and interfacing with the client's presentation devices, through the interface manager.

We also assume the presence of a client buffer, in which the data received from the server is temporarily stored before it is presented to the user. This client buffer serves an important role in masking jitter introduced by server and network, and is therefore an essential component when the final, fine-grained synchronization is performed. In addition, this buffer plays an important role in masking interaction latency, through the use of specialized buffering strategies. Examples of such strategies are Q-L/MRP [41] and MPEG-L/MRP [7].

3.3 Layered DBMS Reference Architecture

We base our discussion on a layered architecture that was originally proposed in [44] and later revised in [45]. The idea of the architecture is that each layer realizes a set of types and operations, based on the services offered by the layer below, similar to a network protocol stack. In other words, layer *n* uses the types and operations of layer *n-1* to realize its services. This principle is illustrated in Figure 3-3, where each layer is presented as a resource manager [54]. Each resource manager uses the resources on the layer below, in order to realize its own layer. Together, the resource managers transform physical resources into logical resources that are more appropriate for the application area, and on each layer, measures taken to eliminate the bottlenecks that may occur.

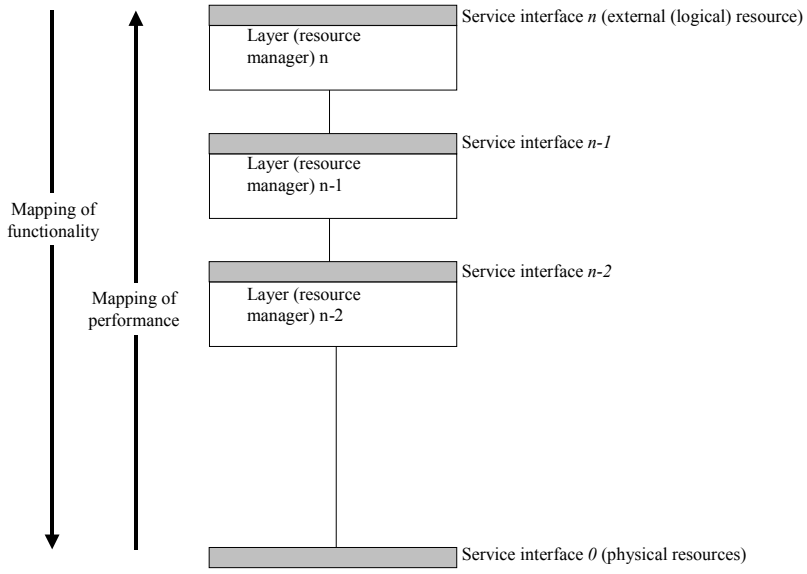

Figure 3-3: A layered architecture

Using this principle of layered architectures on a DBMS, we get the model shown in Figure 3-4 [54]. The physical resource of secondary storage is mapped into an external (logical) resource, namely an object-oriented or relational database, via three intermediate layers. Note that, for performance reasons, no concrete DBMS has fully realized this layered architecture, but the model serves as a good basis for comparing data management systems.

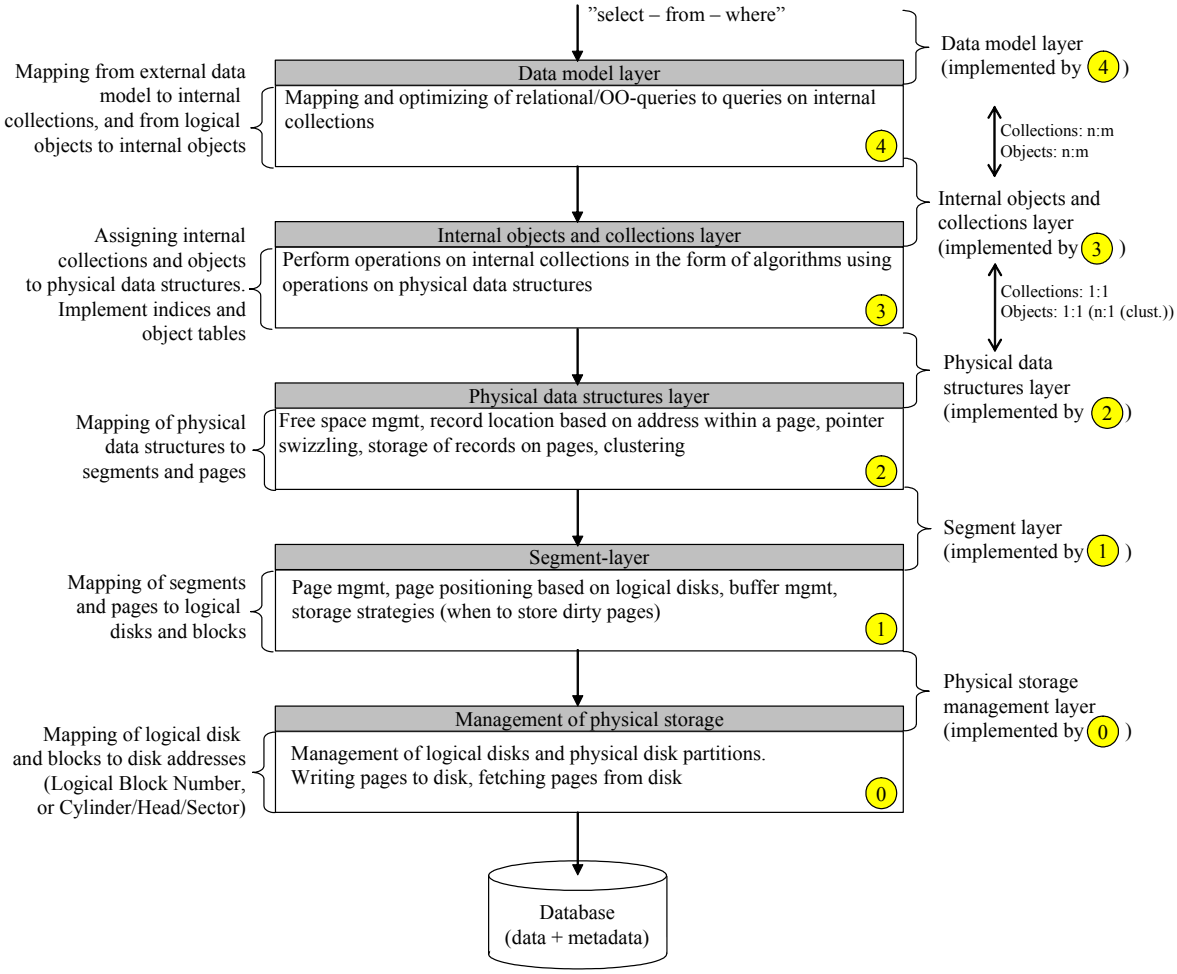

Figure 3-4: Layered DBMS Architecture

In the following sub-sections, we present each of the layers in Figure 3-4, starting from the bottom; and we explain how mapping of functionality and performance is realized. To avoid misunderstandings, we use the layer number (the number in the bottom right corner of each white box) when referring to the resource manager implementing a layer. Also note that we use the term *object* as a generic term for data element; it does not necessarily imply the use of object-orientation.

3.3.1 Physical Storage Management (Layer 0)

The physical database is stored on secondary storage, i.e., disks; and the lowest layer (layer 0) is dedicated to management of this physical storage. Physical disks consist of a number of disk platters, with one read/write head for each platter surface. Each platter consists of a number of tracks, and each track is divided into sectors (also called slots). A disk block, which is the unit of transfer between disk and main memory, consists of one or more

consecutive sectors. Typically, a sector contains 512 bytes, while a block contains 2 to 64 KB, i.e., 4 to 128 consecutive sectors [39]. Tracks with equal diameter constitute a cylinder, and since all read/write heads move together, all sectors within a cylinder can be read or written without moving the heads.

The addressing of a disk slot is dependent on how the disk "presents itself" to the disk driver. Many disks offer addressing by specification of the cylinder, head (i.e., surface) and sector that holds the requested block, so-called CHS-addressing (Cylinder, Head, Sector). The other alternative is to present the disk as an array of consecutively numbered blocks, known as LBN (Logical Block Numbers) addressing.

A disk consists of one or more physical disk partitions, where each partition consists of a contiguous set of tracks on one disk. A *logical disk* is assigned to one or more physical disk partitions, not necessarily on the same physical disk; and a physical disk partition may contain several logical disks. All disk partitions used to store one logical disk must have the same block size in order to keep the addressing scheme manageable, and the blocks of the logical disk (the logical blocks) are numbered consecutively, starting with zero.

The task of layer 0 is to hide the physical addressing (CHS or LBN), and instead present the disk as a collection of *logical disks*. Thus, when layer 0 receives a block request, e.g., read($\log.disk_A$, $\log.block_M$), from the next higher level, the address is mapped to a block number within the corresponding physical disk partition. This address is in turn mapped to a CHS or LBN address, which is submitted to the disk driver.

The performance measures taken on this layer are the storage of data on contiguous disk blocks, together with a large unit of transfer between disk and memory. Both these measures serve to reduce the non-productive work of the disk, i.e., disk head positioning.

3.3.2 Segment Management (Layer 1)

This layer offers a virtual storage containing *pages*. The storage is partitioned into *segments*, and the pages within a segment are numbered consecutively. The next-level layer (layer 2) requests pages using segment-ID and page number within the segment as address when it requests a page. When layer 1 receives such a request, it first checks whether the requested page is already present in one of the buffer frames. If so, a pointer to the page is returned immediately. If the page is not present, the (segment-ID, page number) address is mapped into a (logical disk ID, block number) address, and a request for the block with this address is submitted to the physical storage management layer. Note that, a

page has the same size as a disk block, and constitutes the unit of transfer between disk and memory.

A segment is stored on one logical disk, while a logical disk may contain several segments, thus, there is a 1:n relationship between segments and logical disks. The ability to distribute a segment over several logical disks is not necessary, since, in effect; this can be achieved on layer 0, by distributing a logical disk over several partitions (and possibly physical disks).

The main functionality of layer 1 is buffer management, and the performance measure taken on this layer is the efficient use of main memory, i.e., try to keep the most relevant pages in memory at all times, such that the disk traffic is minimized.

3.3.3 Physical Data Structures Mangement (Layer 2)

Layer 2 is responsible for the assignment of *physical records* to pages, and for the implementation of physical data structures that contain the records. Note that we use the term *record*, independent of whether it is an object-oriented or relational DBMS. Interfaces for accessing the records of a data structure are offered to the layer above, but different data structures support different access methods, and what interfaces to offer is therefore dependent on the data structure that is used. However, all structures support sequential access using iterators.

There are two main types of physical data structures that can be distinguished, based on how the placement of records within pages is controlled:

- Data structures with internal placement of records do their own record placements on pages, i.e., entire pages are allocated to the data structure. In general, these data structures do not allow direct access to its records based on physical address. Instead, object access is based on iteration, i.e., sequential access, on a key value (e.g., logical OID); or it is based on the relative position of the object. Examples of data structures with internal record placement are clustered lists, trees (B-trees, B*-trees, etc.), hash tables, and multi-dimensional data structures (grid-files [66], R-trees [40], etc.).
- Records with external record placement leave the placement of records to the common free space administration of layer 2. Therefore, records of different types may be stored on the same page, if several data structures share a segment. Each physical object has an address, a record-ID (RID), that can be used by other data structures. Key-based access is not supported, while iteration-based and position-based accesses are

supported. Examples of data structures with external record placement are linked lists and pointer arrays.

Depending on the physical data structure in question, when layer 3 requests a physical record from layer 2, the record is either requested based on a physical RID, using an iterator, or using one of the other access methods offered by the data structure. A RID typically consists of segment ID, page number within the segment, and an index into the page directory of the page. Upon receipt of such a request, layer 2 can locate the record directly, based on the RID. When the page is fixed in a buffer frame and the record is located, layer 2 returns a pointer to the record to the requester on layer 3.

In order to realize sequential access, layer 2 uses the next/previous pointers (RIDs) of each physical record, and maintains a cursor to keep track of the current position in the data structure.

Other functions performed by this layer are free space management for pages (not for data structures with internal record placement), pointer swizzling, and physical clustering. The performance measure taken on layer 2 is to control the placement of physical records on pages, such that the number of physical page references, i.e., pages that must be read from secondary storage, is minimized.

3.3.4 Internal Objects and Collections (Layer 3)

Layer 3 uses the physical data structures that layer 2 offers, in order to implement internal objects and collections, i.e., it realizes the internal database, using the content of the physical database. While the physical database consists of physical data structures and physical records, the internal database consists of (internal) collections of internal objects. Each internal collection maps to exactly one physical data structure and vice versa. Normally, there is also a 1:1 relationship between internal objects and physical records. However, in the case of physical clustering, one physical record may contain several internal objects.

The performance measure taken on layer 3 is to choose physical data structures for internal collections such that the access pattern for internal objects is reflected. For instance, if the objects of an internal collection are usually accessed in a particular order (e.g., based on an attribute), then the physical data structure should store the corresponding records in the same order, thereby reducing the access cost.

The actual execution of queries also takes place in this layer. Operations on internal collections are performed as algorithms that use operations on physical data structures. During query execution, indexes are used to speed up data access. Like other data structures, the indexes are implemented on layer 2, and no distinction is made between index data structures and regular data structures; they are both treated equally on layer 2.

3.3.5 Data Model (Layer 4)

The top layer is responsible for mapping between the logical objects and collections, and the corresponding internal objects and collections. While the logical database reflects the structures and relationships of the "real world" (universe of discourse), the internal database should reflect the access pattern on the logical objects, i.e., the objects that are frequently accessed together. Thus, the internal database represents a re-structuring of the logical database, in order to make the occurring operations on the data as efficient as possible.

The attributes of a logical object can be stored in one or more internal objects, and an internal object can contain attributes from several different logical objects. In the same way, a logical collection (e.g., a type extension) can be distributed over several internal collections. Thus, there is an n:m relationship, both between logical and internal objects, and between logical and internal collections.

There exist a number of techniques for mapping from logical to internal objects and collections [95]:

- The N-ary storage model (NSM) maps each logical collection to exactly one internal collection, and each logical object to exactly one internal object, which then contains all attributes of the logical object. Thus, there is a 1:1 relationship between the logical and the internal collections and objects. If complex objects are stored using this technique, there are two alternative techniques that can be used:
  - Normalized storage model: The complex object type is split into several internal object types, one for each logical object type, i.e., the super-object and every sub-object.
  - Direct storage model: the resulting internal objects consist of both the complex object and the sub-objects. A consequence of this storage model is that shared sub-objects are duplicated, i.e., a copy of the logical sub-object is stored in every internal object representing a complex object that references the sub-object.

- The Decomposition storage model (DSM) creates one internal collection for each attribute of the logical object type. If we assume a logical object type containing the attributes ‘name’ and ‘age’, in addition to an OID, the logical collection consists of all instances of this object type. If we use the decomposition storage model, the result is two internal collections, one consisting of internal objects of the type (OID, ‘name’), and one with objects of the type (OID, ‘age’).
- The Partial DSM is a combination of NSM and DSM. Affinities between object attributes are used as basis for the decomposition, such that attributes that are frequently used together are mapped into one internal object type. Thus, the access pattern determines how many fragments (i.e., internal object types) the logical object type is decomposed into.

The performance measure taken on layer 4 is in the structuring of the internal database. Thus, the structure of the internal database must reflect the access patterns into the logical database, such that the operations that are performed on the data are efficiently supported. For instance, objects that are frequently joined can be combined into one internal object, i.e., the join is pre-computed, in order to avoid joins during the execution of queries.

3.4 Data Model

Presentations usually consist of a combination of several multimedia objects, and it is important that the included objects are presented in a synchronized manner. Consequently, we need a data model that can handle these complex relationships, which include temporal information. In addition, the data model must support the metadata associated with the multimedia objects, and since objects can be reused in different presentations, the modeling of the multimedia objects themselves should be separated from the modeling of presentations.

As part of the OMODIS project, a data model called TOOMM (Temporal Object-Oriented MultiMedia data model) [35] has been developed to meet these requirements. This data model integrates temporal and spatial concepts into an object-oriented data model, in addition to modeling and handling of advanced multimedia-related metadata. Objects in TOOMM comprise the properties of traditional object-oriented data models and three different time dimensions: valid time, transaction time, and play time. The play time dimension places units of multimedia data, such as frames or audio samples, into a temporal structure for multimedia presentations.

In addition, TOOMM is based on the principle of separation of multimedia data from its presentation specification. This is in accordance with the modeling concept of independence between the way data is stored in the database, and how it is presented to the user. The advantage of the separation is that multiple presentations can be created, based on the same multimedia data, without having to replicate the data.

While objects instantiated from the *logical data model* contain the actual multimedia data, the objects from the *presentation model* specify how the multimedia data should be presented. We present both these models in the following sub-sections.

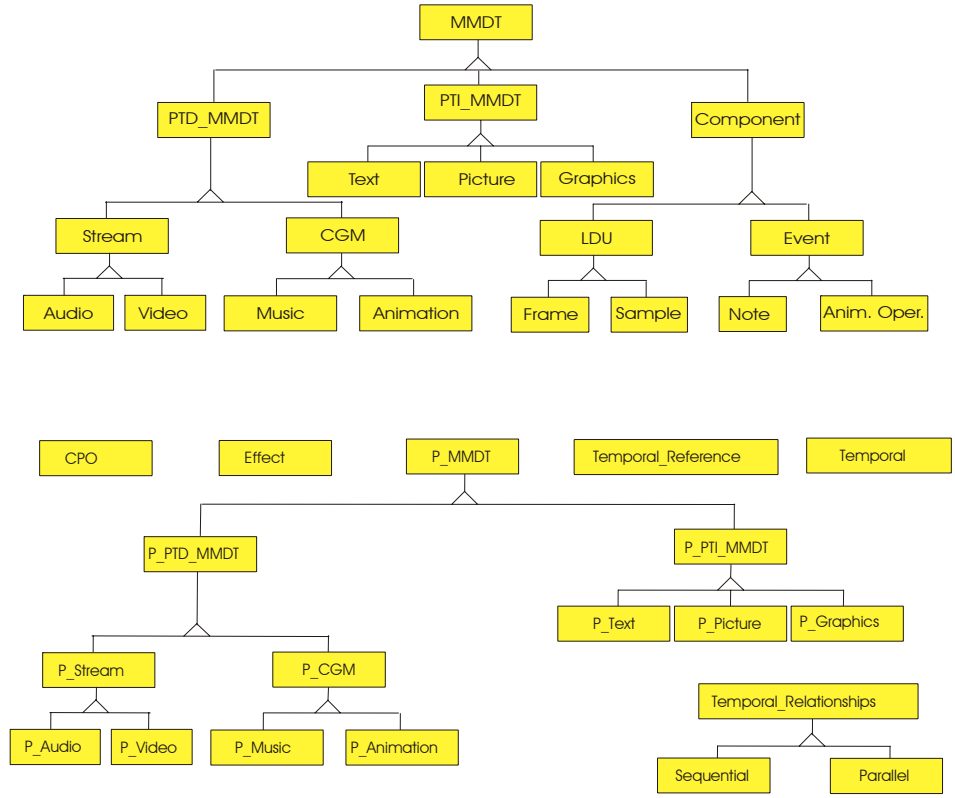

Figure 3-5: Type hierarchy of the TOOMM data model

3.4.1 Logical Data Model

The logical data model consists of the most common multimedia data types (MMDTs), such as video, audio, animation, music, and basic abstract data types (ADTs), organized as an extensible class hierarchy (see Figure 3-5).

In the logical data model, three main categories of MMDTs can be identified:

- *Play time dependent MMDTs* (PTD_MMDTs): Includes all types that have a temporal dimension in their presentation, and is the same as continuous MMDTs, introduced in

Section 2.2. Based on the temporal characteristics, these object types are further classified as either *stream* or *CGM* (computer-generated multimedia data).

- *Play time independent MMDTs* (PTI_MMDTs): Data types with no temporal dimension during presentation. Examples of this data type are text, images, and graphics. A PTI_MMDT is the same as a discrete MMDT, introduced in Section 2.2.
- *Component object types*: These are components of PTD_MMDTs, and they are classified according to the sub-category of the PTD_MMDT they belong to: components of stream-objects are called *LDUs* (logical data units), while components of CGM are called *event* object types.

In addition to the separation between multimedia data and its presentation specification, TOOMM also separates the temporal information inherent in all time-dependent multimedia data from the multimedia data itself. For instance, a video has a particular frame rate, and the frames have a temporal order. To maintain the original temporal order, TOOMM associates each component object with a play time stamp, via a time associator (TA) object. This is illustrated in Figure 3-6, where video frames constitute the component objects.

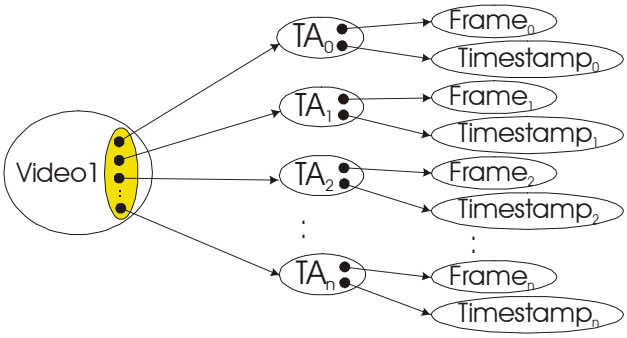

Figure 3-6: A video object modeled in TOOMM

This separation enables a possible reuse of multimedia data within different contexts. For instance, a video frame can be reused as a still image in another multimedia presentation; or a new video object can be defined, which uses only a subset of the frames of the original video.

3.4.2 Presentation Model

In order to support reuse of multimedia data in different presentations, TOOMM separates between the default temporal information inherently associated with the multimedia data (and which is stored within the logical data model), and the temporal information used for

a particular presentation of the multimedia data (see Figure 3-5). The latter information is kept in the presentation model, together with information on temporal relationships between different multimedia objects included in a presentation, as illustrated in Figure 3-7.

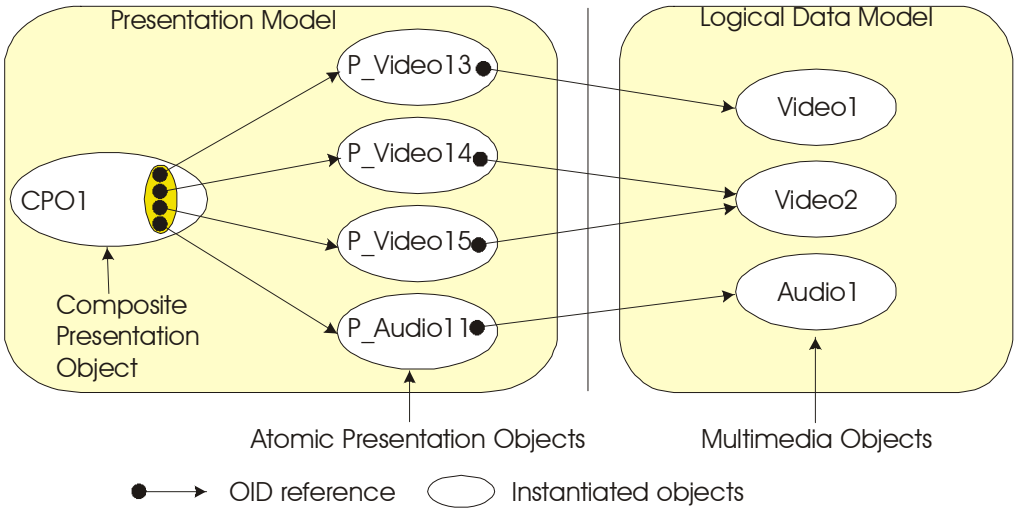

Figure 3-7: Relationship between logical data model and presentation model

The presentation model differentiates between two types of objects:

- *Atomic presentation objects* (APOs) describe the presentation of single multimedia objects, for instance, a video or an image.
- *Composite presentation objects* (CPOs) describe entire presentations. The main elements of a CPO are a set of APOs and a set of temporal relationships between these APOs (such as synchronization and order). TOOMM also allows recursive containment of CPOs, i.e., one CPO may contain other CPOs, such that a CPO tree is formed.

To illustrate the principles of TOOM, we have modeled a simple presentation. The presentation starts with a piece of text, followed by two alternative sub-presentations, each consisting of a picture, and then two consecutive video clips with sound and some text. In Figure 3-8 we show the timing of the presentation, while Figure 3-9 shows the presentation as it is modeled in TOOMM. Note that, in both figures, sub-presentation A is shown, since we assume that sub-presentation B is structurally equal.

In Figure 3-9, we see how the presentation is built from a hierarchy of CPO-objects. Each such object specifies the relationship between two sub-objects, each of which may be an APO or another CPO. We also see that the two video clips, P_Video_{A1} and P_Video_{A2}, actually point to different parts of the same multimedia object, namely $Video_A$.

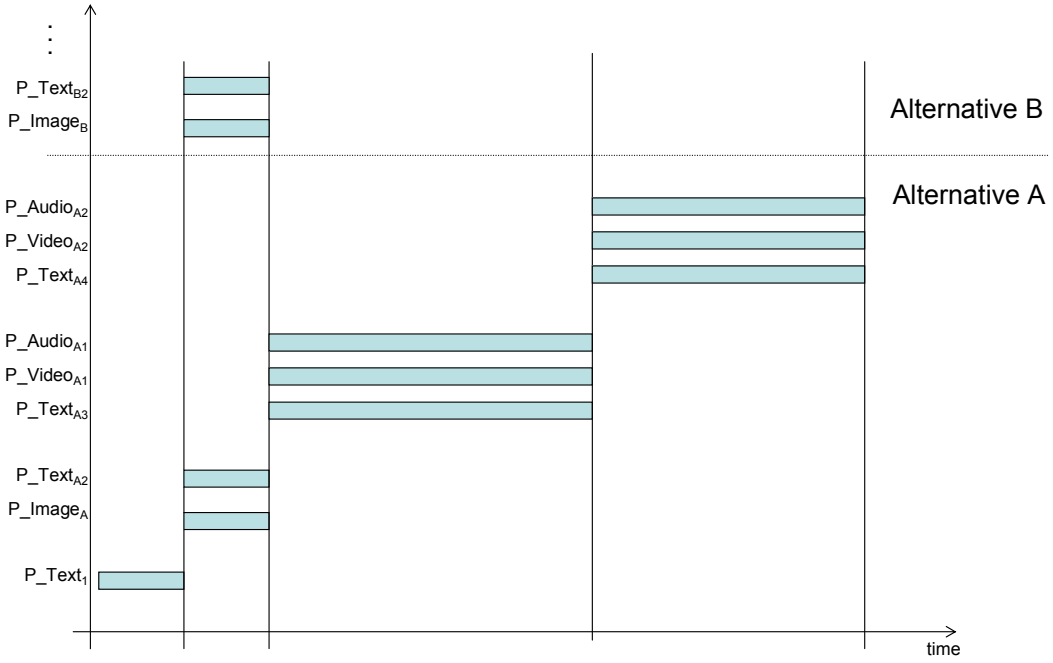

Figure 3-8: Timing of example presentation

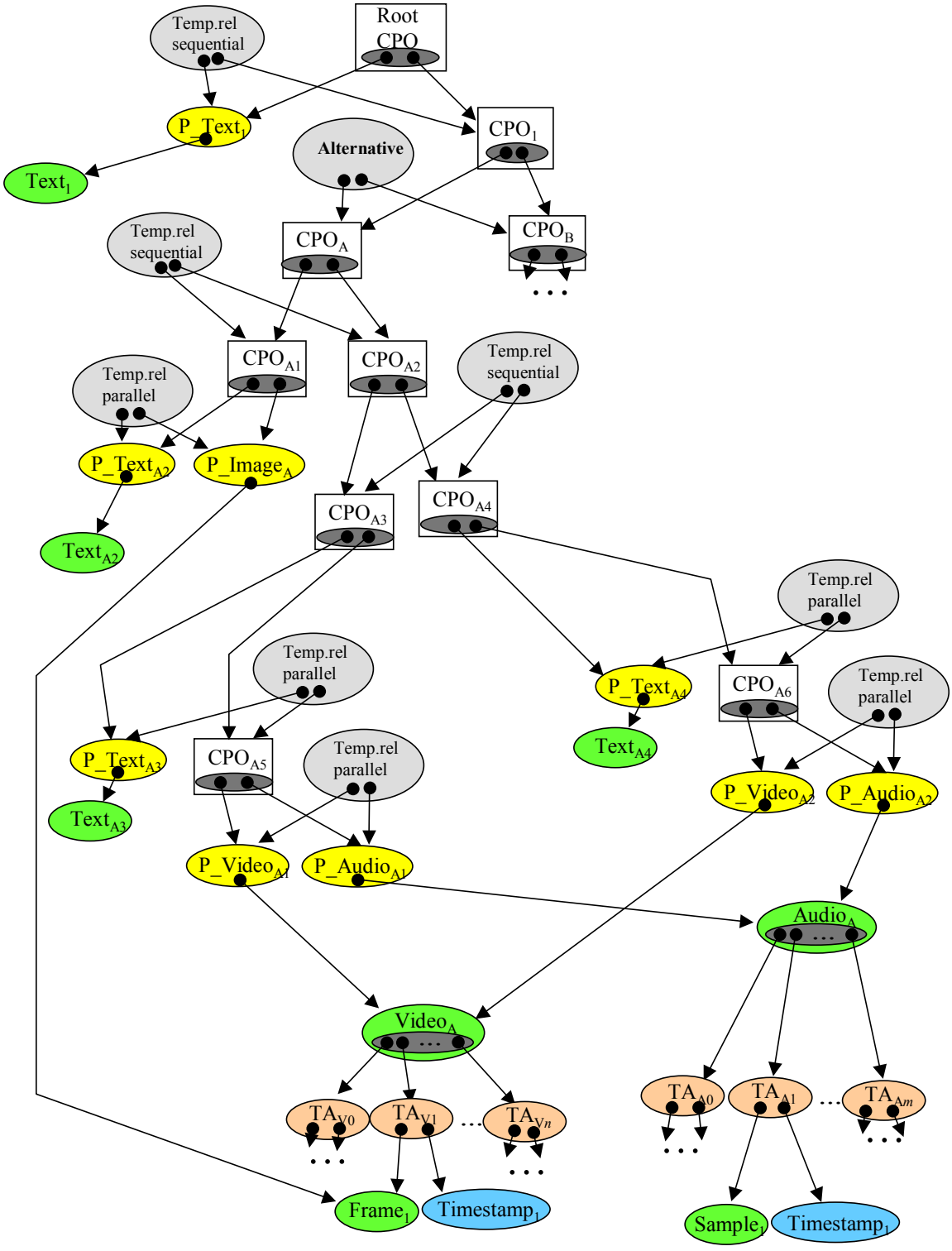

Figure 3-9: Modeling a presentation with TOOMM

3.5 Internal MMDBMS Architecture

We have now presented the layered reference architecture on which we base our description, as well as our data model. In this section, we use the reference architecture to describe the architecture of our MMDBMS.

In Figure 3-1, we presented the architecture of the LoD-server. If we take the layered architecture of the MMDBMS from this figure, and map it onto the layered reference model in Figure 3-3, the result is the architecture shown in Figure 3-10.

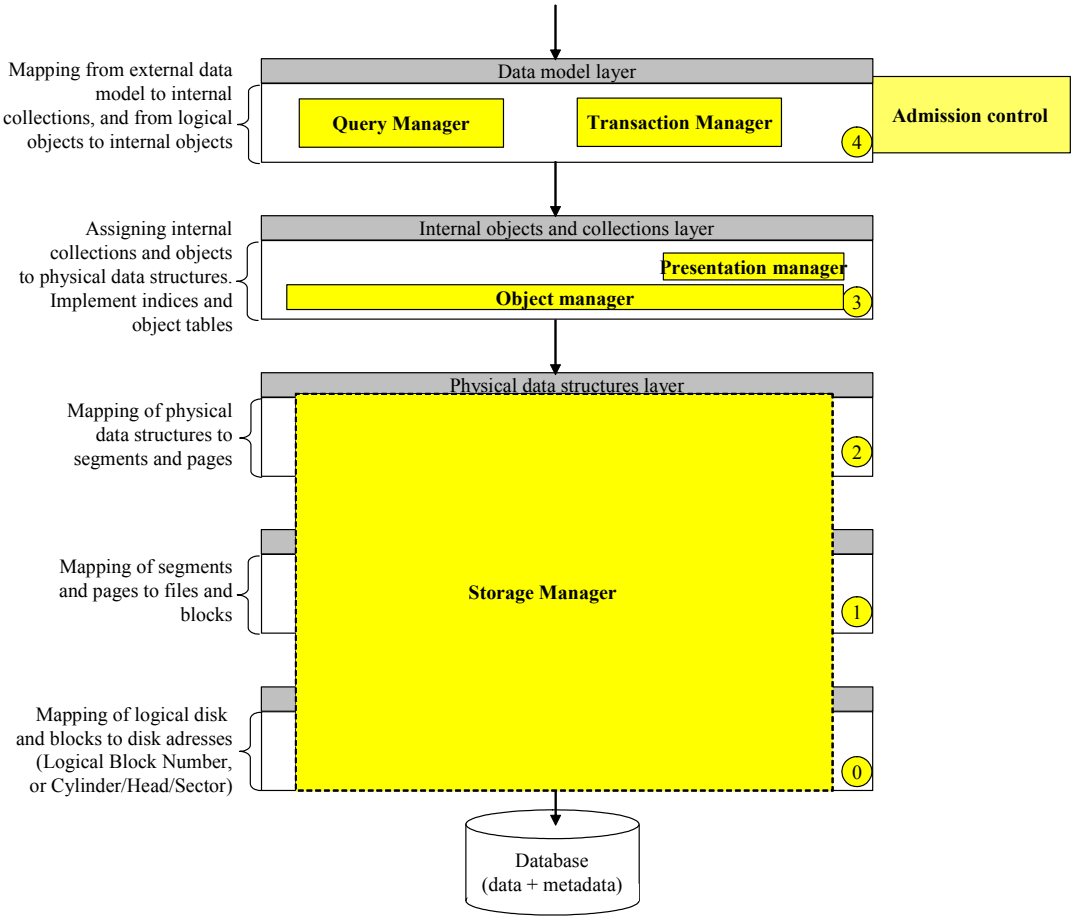

Figure 3-10: Layered MMDBMS architecture

The admission control component is not a part of the reference architecture, and must act independently of the layers. Thus, this component cannot be directly mapped onto the reference architecture, although it is closely related to the transaction manager. We also see that the storage manager covers the three lowest layers of the reference architecture, and as such, constitutes the largest component. In Figure 3-11, we show the internal components of the storage manager, and how these map to the reference architecture. In the following

sub-sections, we go through each of the MMDBMS components, and describe their functionality, using the reference architecture as a basis.

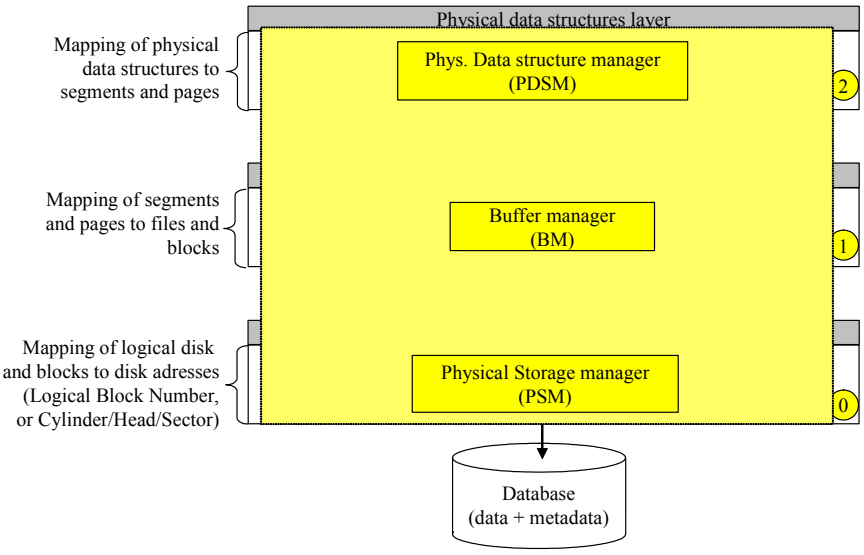

Figure 3-11: Layered architecture of the storage manager

3.5.1 Transaction Manager

The transaction manager (TM) is responsible for ensuring that the system meets the correctness criteria for all MMDBMS requests, and this task includes monitoring the active transactions. The TM defines, monitors, and modifies *transaction plans*, which are used to manage and control the system components. We assume a nested transaction model for the MMDBMS, and this is especially important for multimedia playback queries. In such a query, the transaction hierarchy reflects the CPO hierarchy, and ultimately, the playback of each individual multimedia object is controlled by a separate sub-transaction.

As explained earlier, it is reasonable to expect far more content consumers than content providers. Thus, the dominant usage of the LoD-server is students accessing the stored multimedia data through read-only playback queries for presentations, and for such use, the concurrency control requirements can be somewhat relaxed. Admittedly, during such a playback, students may link questions or other forms of annotations to a presentation (such questions or annotations may also be in the form of multimedia objects), and the teacher may add answers to questions. However, such update-operations do not affect the multimedia data itself. That only happens when the teacher creates a new presentation, or edits an existing one.

Still, there is a need for transaction management, in order to ensure that data is stored correctly and to provide support for concurrency control during writes. For instance, a new lecture may not be made available until it is completely stored in the DBS. An existing lecture, which is to be edited, is first checked out of the DBS and then edited externally. During the editing of a multimedia object, the original version of the object is still available for reading. This indicates the need for one more lock type, in addition to read- and write-locks, and we call this new type "checkout-lock". In Table 3-1, we show the dependencies between the different lock types.

When a multimedia object is checked out for editing, a checkout-lock is set. Subsequent read-locks are allowed for the same object, but requests for a write-lock must wait until the checkout-lock is released. How to handle a subsequent checkout-lock is policy-dependent. If multiple versions are allowed, the second checkout-lock is granted, if not, it must wait until the first lock is released.

Table 3-1: Relationship between lock types

First \ **Then**	Read-lock	Checkout-lock	Write-lock
Read-lock	OK	OK	Wait
Checkout-lock	OK	OK/wait	Wait
Write-lock	Wait	Wait	Wait

In addition, the TM is responsible for coordinating the admission control process, both with respect to authorization, and to the availability of resources. For example, a client request for a multimedia playback query starts with a QoS-negotiation between the server and the client, using the available information as a starting point:

- From the client, information like maximum tolerable interaction-latency and jitter, preferred QoS-guarantee level (deterministic, statistical, best-effort) and the willingness of the user to pay for a given guarantee level, as well as capabilities (client buffer size, codec capabilities, etc.) can be obtained.
- We also assume that information about the network, such as maximum latency and jitter, available bandwidth, and possible service types, is available.

Based on this information, a tentative transaction plan is created, which is used in the admission control process, by submitting it to the query manager. This plan specifies both

what should be retrieved (i.e., which multimedia objects) and *how* (i.e., QoS-parameters). If the query manager approves the request, the tentative transaction plan is converted into a regular plan, and the playback of the presentation can start.

3.5.2 Query Manager

The main task of the query manager (QM) is the translation and optimization of queries. We assume that (ad hoc) queries are entered in a declarative language, either directly by the user, or as a result of user interaction with the client application. In either case, the QM first translates the query into an algebraic expression on the logical data.

Next, it transforms this expression into a number of equivalent algebraic expressions on the internal data. The QM then evaluates the different expressions, using information from the system catalog (also known as the data dictionary [21]), such as physical data structures used, available indexes, etc. Based on this evaluation, the QM selects the optimal[2] expression, and translates this into a physical execution plan, by replacing the operators in the algebraic expression with algorithms that work on the physical data structures. Once the execution plan is ready, the QM hands it over to the object manager, which executes the code in the plan.

An additional challenge of query processing in a multimedia context is the heterogeneity of the data types. As a result, it is not possible to work out one single processing strategy that is optimal for all data types. Instead, different strategies must be used for each data type, and these strategies must then be combined while processing the data.

It is also important that queries, both pre-defined and ad hoc, can be easily formulated for all MMDTs. Since the queries can be made on multimedia data such as video and audio, i.e., content-based queries, it is necessary that the query language has mechanisms to support this. We assume that such content-based queries are supported through textual descriptions of the data, e.g., text describing the content of a scene in a video.

In our MMDBMS, the QM also takes part in admission control and resource reservation for multimedia playback queries [26]. Based on the tentative transaction plan received from the TM, the QM creates an admission request, consisting of a tentative query execution plan (created as described above) and submits it to the admission control

[2] In practice, the goal is to find an expression/plan that is as optimal as possible within the given time- and processing constraints.

component. If the request is admitted, i.e., there are sufficient resources to execute the plan and the reservations have been made, the QM returns a positive acknowledgement to the admission request from the TM. The TM then commits the admission request, and the tentative query execution plan is converted to a regular query execution plan.

3.5.3 Presentation Manager

The presentation manager (PM) is responsible for controlling the transfer of data from the MMDBMS to the client. Regardless of query type, the PM is responsible for the delivery of all data; and to do so, the PM must create a presentation plan. Such plans are created from templates, and modified based on the transaction plan received from the TM, metadata stored in the database, and information from the client about its capabilities.

Thus, while the QM is responsible for making the correct data available in the MMDBMS-buffer, the PM is responsible for transferring these data to the client at the correct time, and with the correct QoS. However, to be able to perform its task, the PM depends on the data being available in the MMDBMS-buffer at the correct time, which in turn requires that the QM requests the data and, thereby, the lower layers provide the data, in time.

3.5.4 Object Manager

The object manager (OM) is responsible for performing the operations in the query execution plan. Many of these operations consist of calls to operations implemented by the storage manager (SM) on layer 2. Thus, the actual work performed during a query is shared between the OM and the SM. A typical operation performed by the OM is to request an object, using an iterator, a key value, or the physical address of the object. In all three cases, the SM provides the access method at its interface to the OM, i.e., the OM uses operations offered by the SM, and the SM performs the actual work of providing the object, possibly fetching it from disk.

3.5.5 Storage Manager

In Figure 3-11, we see that the SM covers three layers of the reference architecture. We have included the sub-components of the SM, in order to illustrate the functionality of each layer. The main interface to the SM, used by the object manager, is the interface to the

physical data structures layer, which is implemented by the physical data structure manager (PDSM).

Physical Data Structure Manager

As described in Sub-section 3.3.3, the PDSM is responsible for mapping between records and pages, and for implementing the physical data structures used by the OM. In an MMDBMS, the multimedia objects, and especially video objects, are typically very large. Even if the multimedia object is structured, e.g., that each frame or set of consecutive frames constitutes a separate component object (i.e., a LDU), these objects are usually larger than a page. Thus, handling storage of large objects is an important task for the PDSM.

The Buffer Manager

The buffer manager (BM) is responsible for transferring pages between secondary and main memory. When a particular page is requested by the PDSM, the BM first checks if the page is already present in the buffer, and if so, a pointer is returned immediately. If the page is not in the buffer, a free buffer frame must be acquired, possibly by flushing the old contents of the frame to disk, before requesting the page from disk. Thus, page replacement is also an important task of the BM.

Since a MMDBMS handles very diverse data types, with correspondingly diverse access patterns, the buffer manager of an MMDBMS faces more challenges than the buffer manager of a traditional DBMS does [4]. For instance, system catalog data and TOOMM metadata requires a page replacement algorithm that tries to keep a “working set” of the most relevant data in memory at all times, such that subsequent accesses to the data can be served from the buffer, without having to go to disk. For multimedia objects, on the other hand, this is generally not feasible, since transactions typically access large amounts of sequential data. Instead, other page replacement algorithms are needed, that support the requirements of multimedia playback queries. One relevant technique for achieving this is known as *bridging* [48], i.e., pages used for one multimedia playback are kept in the buffer, in order to serve another, subsequent playback of the same multimedia object.

The Physical Storage Manager

The role of the PSM is the same in our MMDBMS as in the reference architecture. The BM submits disk requests to the PSM, which converts the logical addresses to physical

addresses according to the addressing scheme used by the disk. It then submits the requests to the disk driver in the operating system.

3.5.6 Admission Control

The LoD-server has a limited amount of resources, such as disk bandwidth, processor capacity, and amount of memory; and since the system is supposed to support multiple concurrent users, it is necessary to limit the admission to the system. If too many users access the system concurrently, the result is overloaded resources, and unsatisfactory QoS for the users.

As part of the OMODIS project, an admission control and resource reservation agent (ACRA) has been developed [90] as part of a QoS-framework [27]. In our work, we assume that this ACRA is utilized for admission control and resource reservation in the LoD-server. However, admission control in itself is outside the scope of our work.

The ACRA has been designed to meet the following criteria:

- *Multiple resource types*: For every admission request, the ACRA must estimate the resource requirements for processor, memory, and disk.
- *Presentations with multiple multimedia objects*: Presentations usually consist of a set of multimedia objects, which are presented both sequentially and in parallel. When estimating the resource requirements for a presentation, the ACRA must take the ordering of the objects into account, in order to compute the total resource requirements.
- *Multiple QoS-levels*: Depending on the capabilities of the client and the network, as well as the resource situation in the server, the ACRA must be able to support several QoS-levels. Typical levels are *deterministic* QoS-guarantees, where violations of the guaranteed service level should never occur; *statistical* (also known as *predictive*) QoS-guarantees, where some violations are allowed to occur; and *best-effort*, where no guarantees are provided [33]. We have also introduced an additional level, called *enhanced best-effort*. This QoS-level implies that a resource share, e.g., a certain amount of disk bandwidth, is reserved, but the amount is not necessarily in proportion to the requirement, and it may change over time.
- *Session-level admission control*: The ACRA must perform admission control on a session-level. For instance, it must perform admission control for a presentation

playback as a whole, instead of admitting the individual multimedia objects as they are requested during the presentation.

- *Support for user interaction*: As described in Chapter 2, we assume a relatively high degree of interaction during presentation playback, and the ACRA must be able to take this into account when performing admission control and resource reservation. To do so, the ACRA can use several information sources, such as statistical information about user interaction and metadata from the presentation itself, e.g., the number of alternative sub-presentations.
- *Support for all types of user operations*: In Sub-section 2.3.1, we described the four basic types of user operations, namely authoring of multimedia objects, authoring of presentations, search for content, and playback of content. Some of these operations require playback of time-dependent data, while others are time-independent. The ACRA must make sure that all operation types are handled fairly, and avoid starvation of, for instance, content searches.

A consequence of the criteria listed above is that the ACRA must have access to metadata describing the requirements of each multimedia object. Thus, before a multimedia playback query starts, the ACRA knows the resource requirements of the involved multimedia objects as a function of time. This is not a problem, since the multimedia objects are stored as structured objects, as explained in Section 3.4, and knowing the "object-rate", we can estimate the resource requirements over time.

However, even if detailed information about the requirements of the multimedia objects is available in the system catalog, resource planning in a complex system like a MMDBMS is a challenge. For instance, if two users play back the same presentation, with only a small time gap between them, disk bandwidth reserved for the second user may be left unused, because the data fetched for the first user is still available in the MMDBMS buffer and the BM applies the bridging-technique. If one of the users interacts with the presentation (e.g., making a pause), the time gap between the two presentations may become too large, and the reserved bandwidth must be put to use. Thus, even with detailed knowledge of the nominal bandwidth requirements of the presentations, it is impossible to accurately predict the actual requirements, and it must be possible to dynamically redistribute resource allocations according to the varying needs.

3.6 Functional Aspects

We have now described the architecture of our MMDBMS, together with the functions of each major component. Next, we present the MMDBMS from a functional point of view. In Sub-section 2.3.2, we presented four types of server operations, and these are executed by the MMDBMS:

- *Metadata retrieval query*: Retrieve application-visible metadata, i.e., data describing the stored multimedia objects, typically TOOMM-metadata. This query type is performed interactively.
- *Multimedia playback query*: Play back presentations, defined by CPOs. This query type is also performed interactively.
- *Metadata authoring query*: Create and/or update CPOs and indexes Depending on the actual operations performed, this query type is either performed interactively or it is automated.
- *Multimedia authoring query*: Checking multimedia objects into or out of the database. This query type is performed automated.

In addition, all these request types may cause the MMDBMS itself to query the system catalog. In many cases, data from the system catalog is required in order to proceed with a user-initiated request, e.g., find the physical address of an object. Consequently, the service level experienced by the system catalog query has a direct influence on the service level for the user-initiated request. In the following sub-sections, we give a brief description of the characteristics of these operations.

3.6.1 Metadata Retrieval Query

In general, users interactively submit metadata retrieval queries to the MMDBMS, to find relevant presentations. This type of query is quite similar to queries in traditional DBMSs, in the sense that object attributes of basic types (such as text or integer), and/or indexes are scanned to find objects that match the search criteria[3] in the query.

Queries of this type are generally short-lived; they scan through a set of objects as fast as possible, and return a list of matching objects to the user. Thus, the execution of such a query will typically cause a short burst of disk requests to be submitted.

[3] In the case of content-based queries, we assume that the multimedia objects have been analyzed (possibly automatically) in advance, and textual descriptions stored in the database.

We assume an extensive use of indexing on the content-related metadata. Thus, most of the work for the storage subsystem will consist of fetching pages containing such indexes. However, each index entry is usually very small, so many entries fit in a page (assuming clustering). Therefore, the disk traffic generated can be expected to be relatively modest, but short response times are advantageous, to keep the overall response time of the query down. On the other hand, queries involving object attributes that are not indexed can have a considerable influence on the extent of disk traffic, especially if the objects are large, i.e., there are few objects per page, and a storage model like NSM is used.

Since metadata retrieval queries are read-only, logging is, in principle, not required. In addition, the TM must check that there are no write-locks on the objects being requested, i.e., traditional lock management.

3.6.2 Multimedia Playback Query

A typical course of a user session is to first run a metadata retrieval query to find presentations of interest. The result of the query is one or more potentially relevant presentations (CPOs), and the user then selects one of these presentations, starting a multimedia playback query. In addition, content providers may use this query type to review multimedia objects during authoring.

Multimedia playback queries are often long-running transactions, since presentations frequently include continuous multimedia objects, which must be presented over time. A video, for instance, can typically last from 10 seconds to 60 minutes, and the transaction controlling the playback is obviously equally long.

Each stored CPO contains a relatively detailed specification of the playback of a multimedia object. Thus, much of the query execution plan (QEP) can be created in advance and stored as a template, together with a transaction plan template and a presentation plan template. When such a template is instantiated, only information regarding the chosen QoS-level needs to be added. Thus, the effort (and time) needed to start the playback of a presentation can be reduced.

During playback, the BM experiences a stream of requests for pages containing component objects, resulting from the playback of multimedia objects of the PTD_MMDT-type. These page requests arrive regularly, typically one or more every second. In addition, the BM may receive requests for pages containing multimedia objects of the PTI_MMDT-type, for instance, text documents to be displayed together with the

video. These requests are more sporadic, with maybe tens of seconds or more between each request.

3.6.3 Metadata Authoring Query

We assume two types of metadata authoring queries, namely automated (batched) and interactive. Automated authoring of metadata includes multimedia data analysis to create textual descriptions of the content, creation of indexes, resource requirement descriptions etc. Such queries can be run as background jobs, and best-effort service is therefore sufficient.

Interactive authoring, which includes operations like creation of presentations (CPOs) and manual additions of content-descriptive text, have much in common with metadata retrieval queries. However, an important difference is that authoring implies the use of write locks. Thus, an ongoing metadata authoring query may potentially block the start of a multimedia playback query, because some of the requested objects are write-locked. To avoid such situations, the playback transaction should try to acquire a read lock for all multimedia objects included in the presentation, before the playback starts.

Both types of metadata authoring queries can be long-running, automated transactions due to the large amount of work that must be performed, and interactive transactions due to "user think time". Automated authoring may generate extensive disk traffic, since entire multimedia objects must be analyzed. Interactive queries, on the other hand, imply small updates on individual multimedia objects, and the disk traffic is therefore modest.

Logging and locking for metadata authoring transactions are handled in the same way as for a traditional DBMS transaction, i.e., the log is written to disk before the data itself, and write-locks must be acquired before updates can be made.

3.6.4 Multimedia Authoring Query

Editing of existing multimedia objects is based on the check-out/check-in model. Thus, a multimedia object to be edited is checked out of the database, and edited externally. Once the editing is finished, the multimedia object is checked back into the database.

Checking a large multimedia object, e.g., a video, into or out of the database can represent a considerable load on the storage subsystem, and unless we limit the bandwidth usage, such operations may affect concurrent multimedia playback queries. However, since

multimedia authoring queries are non-interactive, they can be performed as a best-effort, background jobs, with limited bandwidth usage.

Since the objects being updated are checked out, leaving a copy in the database, performing an UNDO of a multimedia authoring transaction is very simple; just revert to the previous version, already stored in the database. Thus, to enable UNDO, it is sufficient to log what objects have been checked out, and what locks have been set. Enabling REDO for a check-out operation is equally simple, by using the same log-information as for UNDO. However, logging of a check-in operation requires special attention:

- Using traditional logging would mean that all data, including the multimedia data, must be written to the log, before being written to the database. This would require storing all the multimedia data twice, which is obviously not a good solution, especially for objects of the PTD_MMDT-type. Thus, traditional REDO based on the log is not feasible for multimedia data.
- Instead, we can first check the edited multimedia object into a separate "log-area", i.e., a temporary (but persistent) storage area owned by the MMDBMS, which functions as an intermediate store for large objects to be checked into the database. The MMDBMS then performs the check-in operation by copying data from this log-area. In addition, the check-in from the log-area can be split into several sub-transactions. Each time a sub-transaction ends, the OIDs of the component objects stored in the database during that sub-transaction are written to the log. If the system crashes during check-in, the log can be used to establish how far the check-in operation had come, and then re-start the check-in operation with the sub-transaction that was active when the system crashed

3.6.5 System Catalog Query

Common in all MMDBMS request types described so far is that during their execution, several components in the MMDBMS frequently have to access information in the system catalog. For example:

- The transaction manager must access type- and instance-specific information for multimedia objects in order to create the transaction plan.
- The query manager needs information about the mapping between the logical and the internal data model, in addition to information necessary for generation and optimization of query execution plans.

- The physical data structure manager must access addressing tables when the object manager requests objects stored on disk.

Every user-initiated request usually causes several such system catalog queries, which are used in order to get information that is necessary to complete the user-initiated request. Thus, the response time of each system catalog query affects the total response time of the user request.

In principle, a system catalog query is no different from a metadata retrieval query; in the physical database, no distinction is made between system catalog data and regular data, and the same physical data structures is used for storage. However, a system catalog query may have timing requirements, depending on the type of user request that generated it. For example, a requested object must be fetched within a certain deadline, but the mapping information necessary to find the address of that object (OID-RID table) is not resident. If the deadline of the object request is short, there may not be enough time to get first the mapping information from disk, and then the page containing the object, assuming the page has to be read from disk.

In general, one can expect much of the system catalog to be cached [29], so querying it will normally not represent a problem with respect to response time. However, if there is a possibility that system catalog data must be read from disk during execution of a user request with QoS-guarantees, then such situations must be accounted for.

In addition, some types of TOOMM metadata have the same characteristics as system catalog data. For instance, during playback of a video object, it is necessary first to consult objects of the Temporal-type in order to find the OID of the next Frame-object to fetch. Thus, in the remainder of the text, when we talk about system catalog queries, this also includes queries for this type of TOOMM metadata object. The remaining TOOMM metadata, which can be characterized as content-descriptive metadata, does not share these characteristics, and is not considered as being in the system catalog category.

We identify two main contexts in which system catalog queries take place, and timing requirements apply:

- *During the planning phase of a query*: This phase takes place before the actual execution of a user query. Thus, the execution times of the system catalog queries affect the response time of metadata retrieval queries, and the startup time of multimedia playback queries. Typical system catalog queries during this phase include finding information about physical data structures used, what indexes exist, blocking factor, and information about internal and physical schema. Since the planning phase

generally should take as short a time as possible, all disk requests generated, whether they are for system catalog data or regular data, have the same requirements, namely to be serviced as fast as possible. Thus, all disk requests can receive the same service from the storage subsystem; there is no need to differentiate between disk requests caused by system catalog queries and those caused by user operations.

- *During the run-time phase of a query*: As mentioned above, system catalog information, such as tables for mapping from logical OIDs to physical addresses, is needed during the execution of a user query. This information is often critical for the execution of the user query, i.e., if the system catalog information is not returned on time, the user query may not be able to fulfill its QoS-commitments. For instance, each video frame object must be fetched within a certain deadline during playback to be of any value. Therefore, the system catalog information necessary to identify and locate the frame object must be available some given amount of time before the deadline of the frame object. Thus, in this case we differentiate between the timing requirements for retrieval of regular objects and for retrieval of system catalog objects. In general, deadlines for requests for regular objects must be long enough to allow fetching system catalog data from disk before submitting the object request.

3.7 Summary

In this chapter, we have presented the MMDBMS constituting the core of our LoD-server. We first provided an overview of the system architecture, before describing the client-side architecture as well as the functionality of its major components. Next we presented the TOOMM data model, before introducing the layered reference architecture we use for describing our MMDBMS.

With respect to the claims we presented in Section 1.3, this chapter, together with Chapter 2, are targeted at Claim 1: Integration of a disk scheduler in a MMDBMS. We have investigated the MMDBMS, in order to clarify its behavior during the different types of user interactions, and how it affects the storage subsystem.

In the next chapter, we analyze the requirements imposed by the users, the application, and the MMDBMS, and we investigate the implications for the storage subsystem and in particular for disk scheduling. From this analysis, we derive a set of requirements, which we later use both to assess existing disk scheduling algorithms, and as a tool in the design of our own disk scheduling framework.

Chapter 4

Requirements Analysis

In this chapter, we review the requirements from the application, described in Chapter 2 and from the server and MMDBMS, described in Chapter 3. We analyze these requirements, and investigate their consequences for the storage subsystem in general, and for disk scheduling in particular.

The purpose of this chapter is to show that disk scheduling is an efficient means of meeting the storage subsystem requirements in a MMDBMS, and to work out a set of requirements that such a disk scheduler must be able to meet. In the subsequent chapters, these requirements are used both to analyze existing disk schedulers, and as design guidelines for our disk scheduling framework called APEX.

4.1 Introduction

In the previous two chapters, we have presented our LoD-application scenario, as well as the MMDBMS used to realize the LoD-server. We have described the operations that the users, i.e., content providers and content consumers, perform on the LoD-system, and how the server responds to these operations.

Knowing how the LoD-system is used enables us to map out the requirements that it must meet. On a high level, the LoD-system as a whole has to meet the requirements of the user. This, in turn, puts requirements on the individual components of the LoD-system. Thus, we map the user-level requirements down to physical resource requirements. Since we focus on QoS in the server-side storage subsystem in this thesis, our analysis will focus on the disk resource, and the requirements it must meet. In Section 4.5, we then show how

disk scheduling can be applied as an efficient means of making the storage subsystem meet its requirements.

4.2 Application and User Requirements

In Sub-section 2.3.1, we described the different user operations, and found that the requirements and expectations of the user varied from operation to operation. We also described the operations performed by the server in response to the user operations. Below, we have grouped these server operations according to the user's requirements to the corresponding user operation. For each group, we present a summary of the requirements:

- *Multimedia playback*: The user expects that the multimedia objects are presented smoothly, without glitches and hiccups. In addition, the multimedia objects included in a presentation must be presented with correct synchronization, e.g., that audio and video are presented with lip synchronization. Finally, the possibility of user interaction, such as pause, fast forward, and rewind, requires that the system responds quickly.
- *Metadata retrieval and interactive authoring*: The user expects the system to respond quickly to the operations. For example, the time from a metadata retrieval query is submitted, until the MMDBMS returns the result should be reasonably short.
- *Multimedia authoring*: This operation means checking a multimedia object into or out of the database. No interaction is required during the operation, and the response time requirements are therefore modest. However, it is important to avoid starvation, and thereby excessively long response times.
- *Automated authoring*: Such operations run without user interference, which means that they have no requirements at all. However, such operations should not be delayed indefinitely.

In Sub-sections 4.2.2 through 4.2.5, we analyze the requirements of each of the four groups above, and describe how the requirements affect the storage subsystem. However, we start by analyzing the requirements of different MMDTs, as these requirements have considerable influence on the storage subsystem, and require special attention.

4.2.1 MMDT Requirements

As mentioned in Section 2.2, we differentiate between two categories of MMDTs, namely discrete and continuous. Discrete MMDTs, like text and images, are elements without any

time-component. The processing of such MMDTs is not time-critical, since the validity of the data is independent of temporal conditions.

Continuous MMDTs, on the other hand, change over time. In other words, the time at which the information is presented influences the validity of the data; consequently, the processing of such MMDTs is time-critical. Typically, such MMDTs consist of a series of elements of discrete MMDTs that are presented sequentially. For instance, a video consists of a series of video frames, i.e., images, which are presented sequentially.

Discrete MMDTs do not differ significantly from traditional data types, and we will not focus on those here. For continuous MMDTs, on the other hand, there are normally three types of requirements that are common, namely space, real-time, and throughput. Below, we discuss each of these types.

Space

Continuous multimedia data normally requires considerable disk space. In Table 4-1, we show the space requirements for six different videos, and we see that even relatively low-resolution videos take up several hundreds of megabytes. In a LoD-repository containing a large number of videos, it is normally impossible to keep all data in main memory, and the data must normally be read from secondary storage. Thus, the storage subsystem must meet the requirements imposed by the multimedia data.

Table 4-1: Examples of space and bandwidth requirements for video (bandwidth requirements are measured with a resolution of one second)

	Format	Length (min)	Reso-lution	Rate (fps)	Size (MB)	Avg. bandw. KB/s	Max. bandw. KB/s
Video 1	MPEG-2	48	720x576	25	1668.1	582	768
Video 2	MPEG-1	60	320x240	25	348.6	98	192
Video 3	MPEG-2	46	352x288	25	416.1	154	256
Video 4	MPEG-1	29	480x360	25	355.8	212	320
Video 5	MPEG-1	11	720x576	25	221.1	345	448
Video 6	DVD	49	720x576	25	1203.7	421	1088

Except for the last video, which is a documentary recorded from television, all the example videos shown in Table 4-1 are used in LoD today, and as such, they constitute realistic examples[4].

Real-time

As mentioned above, continuous MMDTs are normally presented as a sequence of discrete elements. All the videos in Table 4-1 have a rate of 25 frames per second. Thus, each second, 25 discrete "images" are presented in a sequence, and each one is displayed for 40 ms. This means that there is a deadline for every frame, and if one or more frames are delayed, the user will observe a "hiccup" in the video. Alternatively, the delayed frames are dropped, and the user experiences a short dropout.

In [111], three types of deadlines are identified, namely hard, firm, and soft deadlines. Violating a hard deadline is catastrophic, and must never occur. With a firm deadline, a violation is not disastrous, but the data has no value once the deadline is passed. For soft deadlines, the requested data may still be of some value after the deadline, but this value is monotonically decreasing, and at some point it reaches zero. Depending on how delayed data is handled, the deadlines in multimedia playback are either firm or soft: if delayed frames in a video are dropped, the deadlines are firm, but if delayed frames mean that the entire playback is delayed, then the deadlines are soft.

On the storage subsystem level, individual frames are very rarely used as transfer unit. Instead, almost all storage subsystems that are used to store and play back continuous media data proceed in *rounds* [33]. In each round, the data to be played back during the following round is fetched. Thus, when data is read from disk during a round, all requested real-time data must be retrieved within the end of the round. For example, if a system uses a round time of one second, then at the beginning of each round one second worth of multimedia data is requested for each user, and the requested data has a deadline of one second, i.e., the end of the round. Note that we assume a constant time-length model (CTL), i.e., a fixed-length round, since (in a variable bit-rate environment) this has proven to be less resource-demanding than a constant data-length model, where a fixed amount of data is fetched in each round [17].

In Section 3.2, we described how the client-system buffers a small amount of data before starting the presentation. Such client-side buffering contributes to increasing the

[4] These videos are also used in our performance evaluation, which we return to in Chapter 8.

tolerance of the application playing back the multimedia data, since variations in the arrival time, i.e., jitter, is masked. Thus, while the data should still be delivered within the deadline, the client-side buffering provides the server with more freedom with respect to early delivery of data. In other words, the server is free to deliver the data ahead of the deadline, if that suits the operation of the server better. How much ahead of the deadline the data can be delivered is dependent on the buffering capabilities.

Throughput

The throughput requirements of continuous MMDTs can be relatively high. From our example videos in Table 4-1, we see that the average bandwidth requirements vary from 98 KB/s to 582 KB/s, depending on resolution and coding format. In addition, most modern video coding formats imply variable bit-rate. Thus, the actual bandwidth requirement may vary considerably over time. We see from the table that for some of the videos, the maximum bandwidth requirement is more than twice the average.

Note that, as described above, playing back multimedia data normally means retrieving the data from secondary storage (i.e., disks), and the use of buffering allows this retrieval to proceed in rounds. The round-principle, in turn, enables improvement of disk throughput, by allowing the ordering of disk requests to be optimized with respect to disk efficiency.

Other Requirements

The requirements described above are common to most continuous MMDTs. However, some MMDTs may have additional requirements. One example is layered video (see for instance [76]), where the different layers are stored separately. When data of such MMDTs is played back, requests sent to the storage subsystem may have different priorities, depending on which layer the requested data belongs to. Lower layers have higher priority than lower layers.

Another example is the SPEG coding scheme [51], where prioritized layering of the video frames is applied. Up to 16 priority levels are used to allow fine-grained adaptation of the video rate, by priority-based data dropping. If this is to be supported by the storage subsystem, it must be able to distinguish between the different priority-levels, and treat the disk requests accordingly.

4.2.2 Multimedia Playback

Multimedia playback is by far the most demanding server operation, imposing a number of different requirements on the LoD-system:

- As described in Sub-section 4.2.1, the individual multimedia objects displayed during the presentation have a number of requirements, such as real-time service and high throughput, which must be met.
- Since multiple multimedia objects are displayed during a presentation, both sequentially and in parallel, there are requirements for the synchronization between the objects.
- A multimedia playback is an operation that usually implies user interaction; and if so, the user requires low latency, for instance, similar to a DVD-player.
- Common in all three preceding requirement categories is that the user may require a certain degree of obligation from the LoD-system, i.e., QoS-guarantees.

The requirements of the MMDTs were described in Sub-section 4.2.1, and below, we discuss the remaining three groups of requirements, with focus on their implications for the storage subsystem.

Synchronization Requirements

As described in Section 2.3, the presentations stored in the LoD-server normally consist of several multimedia objects, presented both sequentially and in parallel. Thus, in addition to the requirements of the multimedia objects themselves, there are also timing requirements between different objects. The most typical example is synchronization between audio and video, but other MMDTs may also require synchronization. For instance, both subtitling and display of documents that are referred to in a video are examples of discrete MMDTs that must be synchronized with objects of continuous MMDTs. Thus, we see that also discrete MMDTs may have timing requirements similar to continuous MMDTs.

As described in Section 3.2, the client-system has the final responsibility for such synchronization, using the client buffer and the playback manager. All data received from the server is stored temporarily in the client buffer, from where the playback manager fetches the data.

However, the playback manager depends on that data being available in the buffer at the correct time, and it is the responsibility of the server to ensure that this happens. Assuming that all data, due to the large amount, normally must be read from disk, this

means that the storage subsystem on the LoD-server must provide functionality for delivering data within a given deadline. In other words, the storage subsystem must provide a real-time service.

Latency Requirements

The possibility of user interaction during multimedia playback necessitates short response times (i.e., low-latency). As described in Section 3.2, we assume that the responsibility for this is shared between client and server. In addition, we distinguish between *startup latency* and *interaction latency*, since the situations in which these two latencies occur are very different, seen from the server point of view.

Startup latency is the latency experienced by the user when a presentation is selected, i.e., the first time the play-button is pressed in a presentation playback session. This startup latency includes sending the request over the network from client to server; instantiating the transaction plan, presentation plan, and query execution plan; performing QoS-negotiation and admission control; filling the first buffer frames on the server side with data; submitting these data to the client; and filling the first buffer frames on the client side.

Interaction latency is the latency experienced during interaction throughout the playback, such as the time to resume playback after a pause, selecting a sub-presentation, etc. In this case, no plan instantiation or admission control is necessary, and the presentation data required for the start of the presentation can be expected to be present, either in the client buffer, or in the server buffer. As mentioned earlier, the client-side buffer manager can use an algorithm like MPEG/L-MRP [7], to make sure that the data most likely to be requested are kept in the client buffer.

The reason for introducing this separation is that the startup-phase of a playback query is very unpredictable, with respect to time consumption, since many components are involved, and a large number of operations must be performed. Because of this fact, it is difficult to give an accurate “promise” about startup latency, other than a general worst-case value, where we calculate with ample time. What we can (and should) do is make sure that sufficient resources are available for the startup phase, to avoid starvation. On the other hand, considerably fewer operations are required during interaction. Thus, it is more realistic to provide guarantees for interaction latency than for startup latency.

QoS-Guarantees

So far, we have focused on the requirements from the MMDTs, user, and application, which the LoD-system must meet. In addition, we need to take into account the extent to which the LoD-system obliges to meet these requirements. The degree of obligation, i.e., the *QoS-semantics*, is an important part of QoS-management. In [24], five types of QoS semantics are defined:

- *Best-effort QoS*: The weakest type of obligation. All components in the system do their best to meet the requested QoS-level, but no guarantees are given.
- *Guaranteed QoS*: The required QoS-level is guaranteed by the service provider, and is supported through allocation of the necessary resources.
- *Compulsory QoS*: Like guaranteed QoS, the QoS-level is guaranteed by the provider, through resource allocation. In addition, the QoS-parameters are monitored, and if the requested level can no longer be sustained, the connection is terminated.
- *Threshold QoS*: This is similar to compulsory QoS, but if the level can no longer be sustained, the user is notified, instead of terminating the connection.
- *Maximal QoS*: An upper limit to the QoS is provided, and the QoS-level should not exceed this. This semantic is typically used if there is a cost attached to the QoS-level, and the user does not want to pay more than a given price.

It is up to the user to select suitable QoS-semantics, based on personal preferences and perhaps earlier experiences with the system in question. For the storage subsystem, an important consequence of these semantics is the need for admission control and resource allocation. That is, for guaranteed, compulsory, threshold, and possibly maximum QoS, a certain share of the disk bandwidth and buffer space are reserved in order to fulfill the QoS-level agreed upon.

As described in Sub-section 3.5.6, we assume that the admission control is performed on a global level, and the result is either admittance of the query and subsequent reservation of resources; or the query is rejected, meaning either that the QoS-level must be renegotiated, or that the client must be rejected due to lack of system resources. In addition, except for best-effort QoS and guaranteed QoS, all semantics require that the actual QoS-level delivered by the storage subsystem is monitored.

Finally, we make a distinction between *statistical* and *deterministic* guarantees (see e.g., [61]). For the real-time requirement described in Sub-section 4.2.1, a deterministic guarantee implies that the requested data always will be delivered on time. However, this

level of guarantee is expensive, since we must reserve resources, in our case disk bandwidth, according to worst-case requirements. When variable bit-rate multimedia data is present, this means poor utilization of disk bandwidth.

A statistical guarantee means that a certain percentage of all requested data will be on time. For instance, for real-time disk requests, a statistical guarantee could state that 99.5% of all disk blocks will be delivered on time. The difference between the two guarantee levels lies primarily in the way admission control is performed: while a deterministic guarantee requires admission control and resource reservation according to worst-case requirements, a statistical guarantee is generally based on using some sort of average resource requirements (see, for instance, [107]). Thus, more clients can be admitted, and the resource is utilized better. The price to pay for this increased utilization is occasional resource shortage, since there may be times with peak load, where the resource requirement exceeds the amount reserved. However, the users of a statistical guarantee service are aware of this fact, and must be able to handle such violations of the service level agreement. For instance, deadline violations can be handled through increased buffering.

4.2.3 Metadata Retrieval and Interactive Authoring

These types of operations imply specifying a search criteria or a set of data to be written, submitting the query, and waiting for the result to be displayed. The requirement from the user is that the waiting time is “reasonably” short, where “reasonable” is defined by the LoD-application.

From the storage subsystem point of view, these operations mean moving a set of database pages (i.e., disk blocks) between the disk and the MMDBMS buffer. However, there are no timing requirements to the individual database page transfers, since nothing is presented to the user until the query is finished. Furthermore, the result of these operations is static; for metadata retrieval, the result is a list of presentations or multimedia objects and for interactive authoring a notification is given, stating whether the operation succeeded or not. Thus, since the presented data is not time-dependent, timing does not affect the quality of the result.

On the other hand, it is important that the storage subsystem does not starve requests generated by metadata retrieval and interactive authoring operations, and this means that it should be possible to reserve a certain amount of the disk resource for such operations. If

we assume that a share of the disk bandwidth and buffer space are reserved in advance, such that these resource shares are divided among all such queries, only traditional admission control, i.e., authorization control, is necessary. If bandwidth and buffer space are reserved per query, the admission control must be extended to check for available resources.

4.2.4 Multimedia Authoring

This type of operation is in principle equal to metadata retrieval and interactive authoring operations, but the amount of data transferred is considerably larger. Thus, on the one hand, the data should be transferred to or from disk as fast as the available disk bandwidth allows, but on the other hand, it is important to isolate such operations, to avoid reduced service for other, concurrent clients.

Consequently, the storage subsystem must be able to separate different operations from each other, ensuring that each user receives the amount of resources he or she is entitled to, and at the same time prevent them from occupying an unreasonable amount of resources.

4.2.5 Automated Authoring

Multimedia objects that are checked into the database are analyzed, in order to derive metadata for use during both searching and playback. Such automated analysis can imply reading a considerable number of database pages from disk, for instance when analyzing a video. However, being completely non-interactive, this type of operation can be run on a best-effort basis.

Similar to multimedia authoring, it may be necessary to limit resource usage, to avoid starvation of other clients. In addition, the progress of the operation should be monitored, to make sure that the operation do not starve, and thereby is delayed indefinitely.

4.3 MMDBMS Requirements

In Section 3.6, we investigated the different query types that occur in the MMDBMS, and in Section 4.2 we analyzed their requirements on the storage subsystem in more detail. From this analysis, we found that the different query types have very different requirements to the storage subsystem. In addition, it is important to consider two basic assumptions about the LoD-system:

- Multiple users are allowed to access the LoD-system concurrently.
- Interaction is an important element in the usage of the system.

Consequently, the MMDBMS must handle a workload that is constantly shifting as queries start and finish. Even the resource usage per individual user may change rapidly, due to user interaction, multimedia objects that start and finish as part of presentation playbacks, and variable bit-rate MMDTs. Obviously, this also applies to the storage subsystem, and we have identified three MMDBMS-specific requirements to the storage subsystem, assuming that data is normally read from disk:

- *Multiple service types, adapted to the needs of the different query types*: As we have described, the different query types present very different requirements to the storage subsystem. Since we assume a multi-user system, different query types can be expected to run concurrently. In addition, the individual user may require multiple service types, for example during a multimedia playback query. Consequently, we need a storage subsystem that is able to provide all required service types, and to provide them concurrently.
- *Isolation between transactions*: It is important that the system resources are divided fairly among the transactions, such that each user receives the service level that he or she is entitled to. Consequently, the storage subsystem, as well as other resources in the LoD-system, must isolate transactions from each other, such that no single transaction can behave in a way that affects the QoS for others. For instance, if one user downloads a large multimedia object, this download must not be allowed to use so much disk bandwidth that other users, playing back presentations, experience glitches or halts in the playback.
- *Redistribution of bandwidth (work-conservation)*: In Section 4.2, we described how admission control and resource reservation usually are necessary to ensure sufficient QoS during multimedia playback queries. However, we have also described how user interaction is an important part of using the LoD-application, and such user interaction is obviously hard to predict. Thus, reserved resources, including disk bandwidth may be left unused for shorter or longer periods. In addition, as described in Sub-section 3.5.5, the storage subsystem may invoke bridging whenever possible, which means that reserved disk bandwidth may suddenly become unused for periods. On the other hand, there are usually a number of best-effort queries running as well, which could use this

available bandwidth. Consequently, the storage subsystem must be able to efficiently re-distribute unused disk bandwidth.

4.4 Modern Disks

Magnetic disk drives are still the prevailing media for secondary storage of data, and the rest of the storage subsystem, as well as the MMDBMS as a whole must accommodate the characteristics of these devices. In this section, we investigate the consequences of this fact.

In Sub-section 4.2.1, we described real-time service as one of the requirements of multimedia playback queries. To be able to provide such real-time service, it must be possible to predict the behavior of the involved components. In our case, this means that the MMDBMS must be able to predict the service times of disk requests. If this is not possible, worst-case assumptions must be used, resulting in poor disk utilization.

We introduced the basic principles of a magnetic disk in Sub-section 3.3.1, and described how the disk is divided into platters, cylinders, tracks, and sectors. Thus, in principle, serving a disk request consists of three operations: (1) move the disk head to the correct track; (2) wait until the requested data rotates in under the read head; and (3) read the data and transfer it to main memory. Thus, by knowing the current disk head position, the service times can be predicted with a relatively high accuracy.

However, modern disk drives tend to hide all details about their internals, and they apply a series of techniques for performance improvement and self-management [74, 88, 105, 106]. Thus, the "intelligence" of modern disks makes their behavior unpredictable, and this represents a problem for application areas where predictability is essential. Below, we describe the most important techniques that are applied in modern disks, and investigate how they affect the behavior of the disk:

- *Disk block addressing*: As described in Sub-section 3.3.1, a disk uses either CHS (Cylinder, Head, Sector) addressing, or it uses logical block numbers (LBN)[5]. Using LBN means that we have no information about the geometry of the disk, and the mapping from LBN to physical address. For CHS-addressing, we do apparently know the geometry, but the problem is that for modern disks, this geometry is logical (see, for instance, [79]), and has nothing to do with the actual physical geometry. Thus, even

[5] LBN is also known as LBA: Logical Block Addressing.

if we know the current disk head position with respect to the logical geometry, we cannot estimate positioning times for the disk head.

- *Zone-bit recording*: Outer tracks on a disk are longer than the inner ones, and to maximize the storage capacity, a linear density of the stored data is used. Consequently, the outer tracks contain more sectors than the inner. Usually, the disk is divided into a number of zones, typically from three to 20, and within each zone, all tracks contains the same number of sectors. As a result of this zoning, the transfer rate that the disk can achieve is dependent on where on the disk data is read from. For instance, the Quantum Atlas 10K disk [58] has a transfer rate going from 18 MB/s at the innermost zone, up to 26 MB/s at the outermost zone. Combined with the lack of knowledge of physical disk geometry, this adds to the unpredictability of modern disks.
- *Sparing and slipping*: When a disk is initially low-level formatted, individual regions, either sectors or tracks, are left unused with regular intervals. This sparing is managed by the disk itself; it is transparent to the outside world, and the regions cannot be addressed from the outside. Instead, they are kept as spare regions, to replace regular regions that are damaged. If defects are discovered in a regular track or sector, it is remapped, or slipped, to a spare region. Thus, the disk geometry changes over time, making it even more difficult to know the physical geometry.
- *Caching and pre-fetching*: Modern drives contain an on-board memory of typically 2 - 8 MB [101]. This memory was originally used as a buffer for speed-matching between the physical drive and the transfer of data over the bus, but in modern drives, it is also used as a data cache [105]. One of the most common techniques used is so-called read-ahead. This means that once the disk has finished serving a request, it continues reading from the same track, storing the extra data in the on-board cache. If a future request addresses some of the data in the cache, this means that it is not necessary to access the physical disk, and the request can be served considerably faster. In addition, the cache can be used as a write-back buffer. Thus, instead of having to wait for the disk to write the data, the disk stores the data in the cache, and returns the write request immediately. However, using the on-board memory as a cache has two consequences: (1) it adds to the problem of determining disk head position, since the disk may be active even if it has no disk requests to service; and (2) the service time of disk requests become even more unpredictable, since it is reasonable to expect some requests to be served by the buffer.

- *Disk-internal scheduling*: Modern disks allow multiple outstanding requests, so-called command queuing [74]. This allows the on-board disk controller to reorder the requests, in order to optimize the request serving, for instance by minimizing disk head movement. Since the controller has full knowledge of the physical disk geometry, as well as the current head position, it is in a very good position to achieve an optimal order of the disk requests. However, the controller needs several requests to be able to perform any reordering, which means that the order in which the requests were submitted to the disk may be completely different from the order in which the requests are served. This is potentially a problem for a server offering real-time service, since the service time of the disk requests becomes more unpredictable. A possible solution is to avoid command queuing, submitting only one request at a time, but then we lose the performance gain of disk-internal scheduling. Consequently, the host-based scheduling of disk requests should cooperate with the disk-internal scheduling, in order to improve disk efficiency.

All the techniques described above contribute to an unpredictable disk behavior, and this represents a problem for any component that is dependent on knowing and/or predicting the behavior of a disk (see, for example, [55]).

On the positive side, disk performance is constantly improving; over the last 15 years, disk capacity has improved three orders of magnitude, while the transfer rate has improved 40 times [39]. A high-end disk, such as the Seagate X15 [80], offers an average seek time of 3.6 ms, and a rotational latency of 2 ms. The theoretical transfer rate is 51 MB/s during continuous read from the innermost (i.e., the "slowest") zone, and 69 MB/s from the outermost zone.

If we assume a block size of 64 KB, and one average seek time and rotational latency between each block read, the X15 is able to sustain a theoretical transfer rate of at least 146 blocks per second, i.e., 9.1 MB/s, from the innermost zone, and 9.6 MB/s from the outermost zone. Using 1 MB blocks instead, the X15 can theoretically deliver close to 40 MB/s from the innermost tracks. If we compare these figures with the bandwidth requirements in Table 4-1, we see that even a single disk is able to support a relatively large number of concurrent streams. However, as described in this chapter, playback of streams is only one of several tasks for the disks, and the bandwidth must be shared both among multiple clients and multiple different tasks. Thus, disk performance is still an issue.

4.5 Meeting the Requirements

We have now reviewed the requirements on the storage subsystem, imposed by the user, the application, the multimedia data, and the MMDBMS. We have also described some important characteristics of modern disks, and how these affect the behavior of the disk. Thus, the question is how the storage subsystem best can meet these requirements.

As described in Sub-section 4.2.1, multimedia data is often very large, and when multiple users concurrently play back presentations, search for data, and perform authoring, it is clear that having to read data from secondary storage is the normal situation. There will, of course, also be situations where data can be found in the MMDBMS-buffer, but we consider these as exceptions.

Consequently, the disk becomes the central component in the storage subsystem when trying to meet the requirements. Disk access is still several orders of magnitude slower than main memory access; thus, provided there is sufficient buffer space, the disk has by far the greatest impact on the behavior of the storage subsystem.

Below, we investigate the three central requirements, namely the need for multiple service types, isolation between transactions, and QoS-guarantees, and discuss how we best can meet these requirements.

Multiple Service Types

When multiple service types, such as real-time, low-latency, and high-throughput, are to co-exist in a storage subsystem, we face several, contradicting goals:

- High-throughput requires that the disk is given good working conditions, i.e., the disk requests are ordered such that the positioning work is minimized. This is achieved by careful scheduling of the disk requests, using the position of the requested data as the only sorting criteria. Note that this scheduling should involve the disk-internal scheduler, for reasons discussed above. In addition, there should always be a queue of pending requests, such that the disk is never idle. However, queuing requests causes waiting time, and thereby longer response times.
- A low-latency service requires that the disk requests are served immediately, without taking the position of the requested data into account. Consequently, this service is also realized through disk scheduling. However, since the position of the data is not considered, the disk may have to perform more positioning work, which is harmful for the disk efficiency, and thereby the throughput.

- Real-time disk requests require that the disk serves the request within a predictable time. Thus, the disk requests must be scheduled for service such that the disk has time to serve them. Since disk behavior is difficult to predict, as explained in Section 4.4, worst-case assumptions about service times are often required [33]. However, this affects the utilization of the disk.

We see that all service types are realized through disk scheduling. This is natural, since all these services require detailed knowledge of the traffic to and from the disk, and this knowledge is not available on the higher layers of the MMDBMS. Thus, it is clear that disk scheduling plays a crucial role in realizing a multi-service storage subsystem.

One possible solution for realizing multiple service types is to use separate disks for each service type. This way, each disk only has to relate to one service type, and the contradicting goals are avoided. However, we have described how the workload mix varies considerably: At one point in time, the workload may be dominated by multimedia playbacks, while shortly after, there may be mostly metadata retrieval queries. If we store data on separate disks, according to, for instance, MMDTs, we are likely to experience that one set of disks is heavily loaded, while another set is almost idle. In other words, we get poor resource utilization. Evidence of this can be found in [84], where it is shown that, especially under bursty workload, integrated storage is up to six times more efficient than separated storage. The main reason for this is that unused bandwidth for one data type can easily be transferred to the others.

In addition, we run into a classification problem. For example, when a video is played back, it is naturally treated according to its MMDT. However, when the same video is first checked into the database, it is usually treated as discrete data [22], with no QoS requirements. Thus, the same data requires different types of service, depending on the operations performed. Another example is video frames, being part of a continuous MMDT, that are used as still images, which are discrete MMDTs.

The advantages of integrated storage are also confirmed in [81] and [75]. Thus, we consider the use of integrated storage as a requirement for the storage subsystem of a MMDBMS.

Isolation

With the use of integrated storage, it becomes necessary to prevent the service types, as well as the transactions using the service types, from affecting each other. For instance,

checking a large video into the database could cause concurrent multimedia playback queries to run short of disk bandwidth, causing hiccups in the presentations.

There are two ways of preventing this from happening: (1) provide sufficient bandwidth, to ensure that there will always be enough for all transactions (over-provisioning); or (2) isolate service types, as well as transactions from each other, by limiting the amount of bandwidth each service type and transaction is allowed to use (isolation).

The over-provisioning alternative implies adding more disks, in order to increase disk bandwidth. Then we essentially have two ways of distributing the data over the disks:

- We can use striping, either course-grained or fine-grained. Note that this implies a very wide definition of striping, going from bit-striping, up to storing entire multimedia objects (e.g., videos) on a single disk. The latter is also known as localized placement [73].
- We can reserve certain disks for certain service types. For instance, a separate disk for low-latency requests.

However, the latter alternative is essentially separated storage, as described above; thus, this is not an option. Using striping means that data can be transferred in parallel from several disks, reducing transfer time. On the other hand, it is well known that parallel disk reads do not improve response time; on the contrary, they tend to increase it [73] [77]. Actually, the average seek time converge towards the full seek time, and the rotational latency converges towards the time for a full rotation, with an increasing number of disks [109]. The reason is that the more disks, the higher the probability that a disk head has to move longer, or a disk platter must rotate further, in order to get into the correct position.

The alternative, coarse-grained striping, introduces load-balancing issues. Thus, processing resources are needed for measuring the load on the disks, and computing on which disks to store which data. In addition, it may be necessary to move data from one disk to another, in order to reduce the load on hot-spot disks [78], adding to the load on the disks.

The alternative of over-provisioning is isolation, where we limit the amount of disk bandwidth that each service type and possibly each transaction is allowed to use, and thereby prevent them from affecting each other. Similar to the realizing of the service types themselves, this isolation requires detailed knowledge about the disk traffic, and disk scheduling is therefore a natural choice.

QoS-Guarantees

Realizing QoS-guarantees has similarities to the isolation issue, but while the isolation is concerned with preventing excess consumption of resources, the QoS-guarantees are a means to ensure a certain minimum of resource availability.

In theory, over-provisioning could be used to provide QoS-guarantees as well, but this approach would place restrictions on all transactions that are allowed use the storage subsystem. Thus, to make sure that each transaction receives its share of the disk bandwidth, the workload on the storage subsystem must be treated as a whole, meaning that the number of concurrent transactions allowed to generate disk requests must be limited. Consequently, it is up to the higher layers of the MMDBMS to control which and when transactions should be allowed access to the storage subsystem. The drawback of this solution is that the lack of detailed knowledge about the disk traffic makes it difficult to perform a sufficiently fine-grained control of the bandwidth usage.

On the other hand, if the storage subsystem is able to guarantee a minimum bandwidth on a per-service or per-transaction basis, it suffices to limit the number of transactions that require QoS-guarantees. Transactions that only require best-effort service can freely access the storage subsystem, as the bandwidth usage for such transactions will be limited automatically. This means that there is a higher probability that there are pending requests, which in turn means less idle time for the disk.

A further advantage of this solution is that it allows the storage subsystem to provide multiple guarantee levels, as described in Sub-section 4.2.2. Consequently, we consider the use of explicit QoS-guarantees through reservation a better solution than using over-provisioning. Furthermore, we argue that disk scheduling is a well-suited means of providing such QoS-guarantees, partly because it provides an effective way of controlling the bandwidth distribution of a disk, but also because it fits well with the approaches proposed for realizing multiple service types, as well as for isolation.

Putting it all Together

From the analyses above, we conclude that disk scheduling appears as an efficient and well-suited approach for meeting the requirements analyzed in this chapter. However, given the partly contradicting requirements, it is evident that we need a relatively advanced scheduler. In addition, it is important to realize that disk scheduling is not the one and only solution. It must be accompanied by supporting functionality, such as admission control and QoS-monitoring to ensure correct operation. Furthermore, the disk scheduling should

be supplemented with orthogonal techniques, such as suitable buffering schemes, to further improve the performance of the MMDBMS.

Based on our investigations above, as well as an analysis of disk scheduler requirements performed in [86], we have created a list of five requirements that should be met by a disk scheduler for use in a MMDBMS environment:

1. *Efficient disk read*: Playback of multimedia data can require a considerable disk bandwidth, and at the same time, other query types may generate large bursts of disk requests. Consequently, the disk scheduler must not induce substantial overhead during disk read, something which may easily happen if the scheduler does not consider the placement of data on disk. Instead, the scheduler should try to minimize the overhead, and make the servicing of requests as efficient as possible.
2. *Support for multiple service types and guarantee levels*: Different request types should receive service types that match their requirements. This includes deadline guarantees for real-time requests, support for interactivity through a low-latency service, high-throughput for multimedia authoring queries, and best-effort service for requests without QoS-requirements. Support for different priority levels, as well as request dropping, are also relevant functionalities, for example, in connection with layered video.
3. *Flexibility*: The load pattern on the storage subsystem may vary substantially, and the disk scheduler must be able to adapt to such changes. This means that the disk scheduler should be dynamically configurable, according to the service requirements of the data being retrieved from or written to disk at any given time. Otherwise, the result may be sub-optimal resource usage.
4. *Isolation*: A burst of best-effort requests, for instance, must not be allowed to affect the servicing of real-time requests with QoS-guarantees. Similarly, starvation should be avoided for best-effort services.
5. *Work-conservation*: As explained, the workload experienced by the storage subsystem may vary considerably and with a relatively high frequency, due to VBR streams, user interactions, appliance of bridging, and a varying mix of query types. Consequently, it is impossible to keep the distribution of disk bandwidth perfectly aligned with the needs of the different service types at all times. To compensate for this misalignment, work-conservation is required, in order to channel unused bandwidth to where it is needed. It is reasonable to assume extensive use of such work-conservation, thus, this functionality should be realized with a minimum of loss of disk efficiency.

These five requirements can be used both as a checklist when analyzing existing disk schedulers, and as guidelines in the design of new schedulers. The “ideal” disk scheduler should fully meet all these requirements; however, in practice, the contradictions in the requirements mean that trade-offs must be made. The goal is therefore to make these trade-offs as good as possible, minimizing the drawbacks.

4.6 Summary

In this chapter, we have analyzed the requirements imposed on the storage subsystem by the LoD-application, the users, the multimedia data, and the MMDBMS. We investigated the consequences of these requirements, and proposed possible approaches for meeting them. Out of the proposed approaches, we found disk scheduling to be an effective way of realizing the required storage subsystem functionality, and we presented a list of five requirements that must be met by a disk scheduler for use in our application scenario.

Like the two preceding chapters, this chapter has been targeted towards the first of our four claims presented in Section 1.3: the integration of a disk scheduler into a MMDBMS. We have shown that the requirements, when mapped down to the storage subsystem, can be met through disk scheduling, and what is required to do so.

In the next chapter, we introduce four disk scheduler classification parameters, which we use together with the five requirements introduced here, in order to investigate the suitability of existing disk schedulers for our application scenario.

Chapter 5
Related Work

In this chapter, we analyze existing disk schedulers, to investigate their suitability for our LoD-system. Our tools for doing so are the five LoD-specific requirements introduced in Section 4.5. In addition, we present a set of four parameters representing different aspects of disk scheduler services. These four parameters can be used to characterize a wide range of disk schedulers, and we use them as tools, together with the MMDBMS-specific requirements, to analyze existing disk schedulers.

The purpose of this chapter is to establish to what extent existing disk schedulers are suited for our application scenario. In addition, we select a small group of existing disk schedulers that we call the *reference group*, which consists of those schedulers that come closest to fulfilling our requirements. These schedulers will serve as a basis for comparison during the presentation and evaluation of our MMDBMS disk scheduler.

To appreciate this chapter, it is important that the reader is acquainted with the requirements analysis in Chapter 4.

5.1 Introduction

Originally, disk scheduling was employed to reduce latency, or increase throughput or efficiency [33]. However, with the advent of real-time and multimedia applications, additional requirements emerged, which the existing disk scheduling algorithms were unable to meet [63].

Consequently, disk scheduling has been an active research area for several decades, and there exists an extensive body of work [50]. This is also the case for disk scheduling in real-time and multimedia contexts (e.g., [3, 10, 11, 18, 23, 38, 50, 57, 62, 64, 86, 103]).

However, in the area of MMDBMSs, little has been done with respect to disk scheduling, although some work does exist (e.g., [42]). Thus, our purpose is to establish whether disk schedulers created for other application areas could be suitable for our MMBDMS-based LoD-system.

5.2 Characterizing Disk Schedulers

By analyzing existing disk schedulers, we have worked out a set of four parameters for describing their capabilities. Each of the parameters represents an aspect of the type or level of service that can be requested from a disk scheduler, and as we will show in this chapter, most existing disk schedulers can be described using some combination of these parameters.

The four parameters are: allocation paradigm, guarantee level, service type, and priorities. In the following sub-sections, we describe each of these parameters in detail. It should be noted that, in these descriptions we frequently use the term *queue*. Most disk schedulers that provide multiple service types do so by using queues for the disk requests, one queue for each service type. If multiple transactions require a particular service type, their disk requests usually share the same queue. Thus, in this section, we use queue as a generic term, representing an entity holding or generating disk requests that experience the effect of the different parameters.

5.2.1 Allocation Paradigm

When allocating a resource, the two most popular paradigms for doing so are *proportional-share* allocation and *reservation-based* allocation [92]. In a disk scheduling context, proportional share implies that each queue[6] reserves a relative share of the disk bandwidth, by specifying a *weight*. Thus, the bandwidth share received by a queue corresponds to the weight of the queue divided by the sum of the weights of all queues. This makes the allocation scheme fair, since each queue receives a share of the bandwidth that is in proportion to the other queues.

[6] Note that, in this sub-section, we have used the word *queue*, but we could just as well have used the word *client*, as it is done in [92].

The proportional-share allocation paradigm is also flexible, since there are no restrictions on adding or removing queues dynamically, or on changing the weights of queues. However, this means that proportional share allocation is state dependent, since the received bandwidth share varies with the level of competition [92]. Thus, if a new queue, n, with weight w_n, is added in a disk scheduler, each existing queue i will have its bandwidth reduced from $\frac{w_i}{W} \times BW$ to $\frac{w_i}{W + w_n} \times BW$, where W is the sum of all weights before queue n was added, and BW is the available disk bandwidth. Consequently, the weight of a queue only indicates the relative share of bandwidth *proportional to the other queues in the scheduler*; it says nothing about the actual bandwidth received. This means that, unless a the number of queues, as well as the weight of each queue is carefully controlled, a proportional share disk scheduler is unable to offer QoS-guarantees, since the bandwidth received with a given weight may vary arbitrarily.

The other paradigm, reservation-based allocation, means that an absolute share of the resource is reserved for a queue. This queue is then guaranteed to receive at least its reserved share, independent of competition from other queues. This allocation scheme is less flexible than proportional-share, since it requires admission control to avoid over-allocation (admission control can of course also be used with proportional share allocation, to ensure a lower bound on the shares). The scheme is also less fair, as the resource share reserved for a queue is fixed, independent of the requirements of other queues. To change the resource share, explicit reallocation must be performed.

On the other hand, guaranteeing a minimum bandwidth for a queue has the advantage of enabling QoS-guarantees. However, in a disk-scheduling context, a bandwidth guarantee alone is not a sufficient condition to ensure real-time guarantees. The disk scheduler must also ensure that each real-time request is scheduled for service in time to meet its deadline, through the ordering of the pending requests.

5.2.2 Guarantee Level

Based on the effects of the two allocation paradigms, we distinguish between four levels of guarantee, namely deterministic guarantees, statistical guarantees, enhanced best-effort, and best-effort.

Deterministic guarantees are absolute, i.e., violations of the guaranteed service level should never occur. For instance, offering deterministic real-time guarantees for disk

requests means that the requests will always meet their deadline. As described in Sub-section 4.2.2, this guarantee level is expensive, since worst-case assumptions about resource usage must be made.

Admittedly, there exist techniques that reduce the utilization problem by using so-called load traces, i.e., when a continuous multimedia object has been stored on disk, its bandwidth requirement is computed as a function of time (see, for instance, [103]). When a playback of a multimedia presentation is requested, the presentation is scheduled such that the overall resource requirements for all ongoing playbacks are reduced, for instance by delaying the presentation to avoid concurrent bandwidth peaks between presentations. The problem of this technique is that it constrains interactivity; thus, it is not well suited for our highly interactive environment.

In a mixed-media environment like ours, the cost of using deterministic guarantees can also be slightly reduced through work-conservation, since unused bandwidth from a deterministic guarantee queue can be transferred to other queues needing the extra bandwidth. However, during admission control, we must still use worst-case assumptions, so the total number of clients with deterministic guarantees that can be admitted does not increase with the use of work-conservation.

As described in Sub-section 4.2.2, statistical guarantees are in principle equal to deterministic guarantees, but imply a less rigid guarantee level. Resources are reserved according to some statistically estimated requirement, instead of worst-case; thereby the resources are utilized better, and more clients can be admitted.

Both deterministic and statistical guarantees require a reservation-based allocation paradigm, since proportional-share allocation is insufficient for this guarantee level, as explained in Sub-section 5.2.1. However, using proportional allocation still involves reserving some bandwidth. Thus, although the reserved bandwidth may have to be shared on an arbitrary number of queues, it is better than making no reservations at all, having only possible unused bandwidth at one's disposal.

We therefore introduce the term *enhanced best-effort* to describe the guarantee level offered by the "basic version" of the proportional-share allocation paradigm (as described in [92]), i.e., without restrictions on the access to disk bandwidth. Since the resource share can become arbitrarily, the service level is considered best-effort, but the queue in question will always receive a share that is in proportion to other proportional-share queues, which means that it is guaranteed fairness.

Finally, the best-effort “guarantee”-level means that no resource reservations are made, and the queue in question can only use whatever resource shares are available at any given time. In a disk scheduling context, this means bandwidth unused by other queues. Consequently, disk requests with a best-effort service level may experience starvation if there is no leftover bandwidth.

5.2.3 Service Types

From the point of view of the disk scheduler, there are three service types that can be offered, namely real-time, high-throughput, and low-latency. In addition, it is obviously possible to have no specified service type. The deadlines associated with real-time requests can be hard, firm, or soft, as described in Sub-section 4.2.1; hard deadlines can only be used in association with deterministic guarantee-level, while firm and soft deadlines can be used with any guarantee level.

A real-time guarantee implies that the disk requests are guaranteed to be serviced within a specified deadline (e.g., [64, 86, 103]). Thus, in practice, a real-time guarantee is also a throughput guarantee, as long as the query that causes the requests stays within the request rate set during admission control. A high-throughput guarantee, on the other hand, is not concerned with the service time of each individual disk requests, thus the service times may vary considerably. However, over time, a certain throughput is guaranteed [103].

In principle, a low-latency service could be recognized as a real-time service, but there are some important differences: a real-time service implies that the disk request is served within a specified time, but this is not necessarily a very short time. In a multimedia context, real-time disk requests typically operate with deadlines in the area of 1-2 seconds. Low-latency requests, on the other hand, should be served “immediately”, i.e., if possible, such requests are served before real-time requests [85].

In addition, while a real-time service usually is combined with some sort of QoS-guarantee, a low-latency service is normally only best-effort, possibly in combination with some sort of rate control [103]. The reason for this is that providing a low-latency service is expensive, and the arrival rate of such requests is very unpredictable [86, 103].

5.2.4 Priorities

Disk requests generated by one and the same transaction may have different priorities. For instance, the SPEG coding scheme [51] distinguishes between 16 different priority levels for the layered video frames, and this can be reflected in the storage subsystem, provided that the storage structures support this.

One way of handling priorities in a disk scheduler is by having one queue for each priority level (see, for instance, [15]). The queues are then serviced in order of priority, and no queue is serviced until all requests in higher priority queues are serviced. Alternatively, one queue can support multiple priorities, and sort the requests in the queue according to their priorities (other sorting criteria can be included as well) [49]. This solution is more flexible than the multi queue solution, since it supports an arbitrary number of priority levels.

However, there is a potential problem of priority-based scheduling that different transactions may have different understandings of the priorities [28]. For instance, in a disk scheduler offering two priorities, one transaction may specify most of its request as low priority requests, and then only a few special requests as high priority. Another transaction may use high priority as "default" for its request, and only specify some occasional requests as low priority. If these two transactions were treated equally with respect to priorities, the transaction with "default high priority" would use most of the bandwidth allocated to the queue, regardless of how the bandwidth was meant to be shared between the two transactions.

Consequently, unless the different transactions have the same understanding of the different priority levels, they cannot share queue(s). This problem has received very little attention, as existing work seems to assume that all transactions have the same understanding of the priority levels [15, 49].

5.2.5 Describing Disk Scheduler Services

By using the four parameters described in the preceding sub-sections, it is possible to describe a large number of different service levels and types that a disk scheduler may be required to offer. Obviously, not all possible combinations make sense; for instance, there is no point in combining priorities with deterministic real-time guarantees, since all requests are guaranteed to be served on time. In Table 5-1, we show some of the most relevant combinations.

From the table, we see that the real-time and high-throughput service types can be combined with deterministic and statistical guarantees, as well as enhanced best-effort; the choice of guarantee level is a question of how reliable one wants the service to be. In principle, a real-time service could also be combined with a best-effort guarantee level.

The best-effort guarantee level represents the lowest level of service. The requests are served without reserving resources, thus, choosing an allocation paradigm is not relevant. However, similar to enhanced best-effort, an arbitrary number of priority levels can be chosen.

In addition, we categorize the low-latency service type as best-effort, although reserving an absolute or relative share of the disk bandwidth is also possible. As explained in Sub-section 5.2.3, we do so because the request arrival rate to such a service is usually unpredictable, and the service type is expensive in use.

Table 5-1: The most relevant combinations of disk scheduler parameters

<table>
<tr><th>Guarantee level</th><th>Allocation paradigm</th><th>Service type</th><th>Priorities</th></tr>
<tr><td>Deterministic Guarantees</td><td>Reservation-based</td><td>Real-time (hard/firm/soft)</td><td>Not applicable</td></tr>
<tr><td rowspan="2">Statistical guarantees</td><td rowspan="2">Reservation-based</td><td>Real-time (firm/soft)</td><td rowspan="5">Any number</td></tr>
<tr><td>High-throughput</td></tr>
<tr><td rowspan="2">Enhanced best-effort</td><td rowspan="2">Proportional share</td><td>Real-time (firm/soft)</td></tr>
<tr><td>High-throughput</td></tr>
<tr><td rowspan="2">Best-effort</td><td>None</td><td>None</td></tr>
<tr><td>None</td><td>Low-latency</td></tr>
</table>

We can now use these parameters to describe the requirements of different application types, and below, we give a few examples:

- *Traditional real-time applications*: For traditional real-time applications, i.e., applications that only handle non-multimedia data, the real-time service type is the central issue. The choice of guarantee level and allocation paradigm is dependent on where the application is used. For instance, an application used to control a nuclear power station should use deterministic guarantees and hard deadlines, while for a POS terminal, statistical guarantees and soft deadlines may suffice.

- *VoD*: For VoD, a real-time service type with firm or soft deadlines is required. Again, the guarantee level is dependent on the requirements of the users. For instance, a service using enhanced best-effort can be offered at a low price, in addition to a higher-priced service using statistical guarantees. Depending on the coding of the videos, priorities may be required.
- *LoD*: As we have thoroughly described in the previous chapter, the LoD-system requires all the service types in Table 5-1. The guarantee level is dependent on factors such as the requirements and capabilities of the user and network.
- *Relational DBMS*: In this case, low-latency and high-throughput are the most relevant service types, and usually based on a best-effort guarantee-level. Priorities may be required.

5.3 Classification Criteria

In our study of related work, we have chosen to classify existing disk scheduling algorithms according to the purpose for which they are designed, and we have identified four main classes:

- *Performance-oriented disk scheduling algorithms*: These algorithms only focus on optimizing performance, i.e., increasing throughput and/or reducing latency.
- *Real-time disk scheduling algorithms*: This class of disk scheduling algorithms is intended for use in real-time environments, i.e., servicing disk requests within given deadlines.
- *Stream-oriented disk scheduling algorithms*: We introduce the term "stream-oriented" algorithms, since the algorithms in this class are primarily optimized for handling retrieval of continuous data streams.
- *Mixed-media disk scheduling algorithms*: The algorithms in this class recognize that different disk requests may have different requirements with respect to service types.

In addition, there exists a small body of priority-based disk scheduling algorithms, but support for priorities is orthogonal to the classification above, and we will therefore present those algorithms within the existing four classes.

5.4 Analysis of Existing Disk Schedulers

In this section, we examine each of the four disk scheduler classes, and investigate their suitability for our MMDBMS-environment. The schedulers in the first three classes are relatively homogeneous, and we analyze the schedulers in each of these classes as a whole. For the last class, the mixed-media schedulers, we analyze the schedulers individually.

5.4.1 Performance-Oriented Disk Scheduling Algorithms

As mentioned above, the original motivation behind disk scheduling was performance optimization. A number of scheduling algorithms with this purpose have been proposed, for instance, Shortest Seek Time First (SSTF), SCAN [25], LOOK [60], C-LOOK, VSCAN [31], and Shortest Access Time First (SATF) [47].

Common in all such pure performance-oriented schedulers is the lack of QoS-support. The order in which disk requests are selected for service is based solely on achieving the performance optimization goal. Consequently, the algorithm may choose to postpone the servicing of a real-time request (e.g., a video data request) and serve a best-effort request instead, if the latter request serves the optimization goal better.

Looking at the requirements from Section 4.5, we see that this class of disk schedulers only meets the efficiency requirement and, implicitly, the work-conservation requirement, since such disk schedulers will always schedule requests for service as long as there are any. From the list of service class parameters in Section 5.2, we see that this class only supports a best-effort guarantee level. Furthermore, this class does not support any particular service level, beyond the overall optimization goal of the scheduling algorithm, and priorities are not supported.

It is clear that pure performance-oriented disk scheduling algorithms are insufficient for our environment, due to the lack of QoS-support and multiple service types. However, as mentioned in Section 4.5, performance is still an important issue, and this class of algorithms can therefore be of interest if used in combination with other scheduling techniques.

5.4.2 Real-Time Disk Scheduling Algorithms

The simplest algorithm in this class, Earliest Deadline First (EDF), focuses on deadline only [53]. Due to the mechanical characteristics of a disk drive, i.e., variable service times

and a non-preemptive nature, EDF achieves poor efficiency when used as a disk scheduling algorithm. Newer algorithms of this class try to combine the real-time scheduling with optimized disk read, in order to achieve better efficiency. Examples are SCAN-EDF [62], "Shortest Seek and Earliest Deadline by Ordering/Value" (SSEDO/SSEDV) [19], and Priority SCAN (PSCAN) [15].

While algorithms of this class are able to support requests with real-time deadlines, their problem is that they focus on real-time requests only, i.e., best-effort requests are generally not an issue, and such requests therefore tend to suffer from starvation, if they are supported at all. If we compare with the requirements in Section 4.5, we see that these algorithms partly meet the efficiency requirement. In addition, they generally provide the disk with work as long as there are pending requests, so the work-conservation requirement is usually also fulfilled. However, the remaining requirements are not met.

From the analysis in Section 5.2, it is clear that the algorithms in this class offer real-time services with deterministic or statistical guarantees. The PSCAN algorithm supports priorities, by offering three service classes.

5.4.3 Stream-Oriented Disk Scheduling Algorithms

There exist a large body of "stream-oriented" disk scheduling algorithms, which are schedulers that are primarily optimized for handling retrieval of continuous data streams, i.e., for use in multimedia applications such as VoD. Compared to the real-time scheduling algorithms, these algorithms often focus less on deadlines, and instead rely on periodicity of the requests (the requests are typically served in fixed-length rounds), and fair allocation of disk bandwidth.

Some examples of algorithms in this class are Continuous Media File System (CMFS) [3], Pre-seeking Sweep algorithm [32], Quality Proportional Multi-subscriber Servicing (QPMS) [96], Grouped Sweep scheduling (GSS) [108], the scheduler in [110], BubbleUp [16], and T-scan [23]. Further examples are Batched SCAN (BSCAN) [50], Buffer-inventory-based dynamic scheduling (BIDS) [68], RSCAN [6], Greedy-but-Safe Earliest-Deadline-First (GS_EDF) [94], YFQ [11], Lottery scheduling [98], and Stride scheduling [99] (this algorithm is really a hybrid real-time and stream-oriented scheduler).

In [46], a framework called *anticipatory scheduling* is introduced. This framework is used as an addition to performance-oriented and stream-oriented disk schedulers, in order to avoid so-called deceptive idleness, i.e., the disk scheduler incorrectly assumes that a

disk request is not followed by any further requests. Thus, this is not a self-contained disk scheduler, but it is included here since the idea presented has been an inspiration in our work.

Compared to the requirements in Section 4.5, stream-oriented algorithms normally try to consider efficiency, and to a certain extent, work-conservation. The other requirements are generally not met. Looking at the list of characterization parameters, stream oriented algorithms typically do not offer an explicit real-time service. Instead, the lack of support for deadlines is compensated by careful load control, through admission control, and by the fact that the load is relatively uniform, since all requests are for video data. Furthermore, these schedulers normally operate with reservation-based allocation, in combination with statistical guarantees.

5.4.4 Mixed-Media Disk Scheduling Algorithms

During the last years, disk scheduling for mixed-media workloads has become an active research area, with some new designs for disk scheduling algorithms. Common in most of these algorithms is that they have a hierarchical (typically two-level) design, where one level ensures efficient usage of the disk, while the other handles QoS and differentiation of service types.

It should be noted that some of these algorithms rely on the proportional-share allocation paradigm, while still offering statistical or deterministic QoS-guarantees. This is possible since they use a fixed set of queues, and the weight of each queue is carefully controlled. Thereby, the relative share of the bandwidth for a queue becomes equal to the absolute share, and QoS-guarantees can be offered.

A very early attempt on mixed-media disk scheduling was the work in [72]. It is basically a stream-oriented disk scheduler with a certain fraction of the bandwidth reserved for non-real-time requests. Such requests are also allowed to use slack time during servicing of real-time requests, thus, the algorithm is work-conserving. It only supports two service classes, namely periodic real-time requests and non-real-time requests, but isolation between the two classes is provided. However, the two service types are fixed, thus, there is no flexibility; the scheduler cannot be configured according to varying needs.

Comparing the scheduler with the list in Section 5.2, we see that this algorithm provides deterministic guarantees for periodic real-time requests, in combination with an enhanced best-effort service. The latter service type is enhanced best-effort, since

bandwidth is reserved for the service class, which in turn means that progress is guaranteed. Priorities are not supported.

The Fellini storage system [57], supports two levels of service, a real-time service for retrieval of multimedia data, and a non-real-time service for other requests. Serving disk requests is based on a fixed service interval (rounds), where a fraction of each cycle is used for serving real-time requests, while the remainder of the interval is used for serving non-real-time requests. Thus, this scheduling algorithm supports a very limited number of service classes, and there is no flexibility with respect to reconfiguration. On the other hand, the two service classes are isolated from each other, and the algorithm is partly work-conserving, since non-real-time requests are allowed to utilize unused bandwidth from the real-time queue.

One of the first full-fledged mixed-media disk schedulers was Cello [85, 86], part of the Symphony multimedia file system [83]. This scheduler shares the total disk bandwidth among an arbitrary, but fixed, number of different *service classes* (i.e., queues) using a proportional-share allocation scheme. Thus, each class receives a share of the total disk bandwidth, proportional to its weight. The shares are proportional with respect to either time (share of the total round time) or amount of data (share of total number of bytes transferred in a round).

The *class-independent* scheduler of Cello, which selects disk requests from the different class queues according to their weights, only serves requests as they arrive. Thus, it is up to the *class-specific* schedulers to determine the order in which requests are served on disk. The result of this policy can be a substantial seek overhead, since each class scheduler only is concerned with optimized ordering of its own requests; there is no single component with the responsibility for the final ordering of disk requests. Consequently, disk efficiency can be a problem.

Cello is work-conserving, by distributing unused disk bandwidth among queues with pending requests. First, a set of queues eligible for utilizing the unused bandwidth is determined, according to weights or priorities, or both; and then requests are selected from these queues, according to their weights, in addition to trying to minimize seek time. Thus, the work-conserving property is essentially static, and Cello therefore only partially fulfills the work conservation requirement in Section 4.5.

The weights of the different queues are assigned statically, thus it is impossible to dynamically change the allocation of bandwidth to the different queues. Dynamic bandwidth allocation is defined as future work, but at the moment, Cello is not able to fully

meet the flexibility requirement from Section 4.5. The work-conserving property is not enough to compensate for the lack of dynamic allocation, since unused bandwidth is distributed using the same weights of the classes. This is illustrated by an example in Figure 5-1, which shows the situation immediately after all queues have been served in a round, and where we assume that there is unused bandwidth remaining. In this situation, queue A will receive twice as much of the unused bandwidth as queue B, since the weight ratio is two to one. Thus, it is clear that the bandwidth allocation does not match the workload.

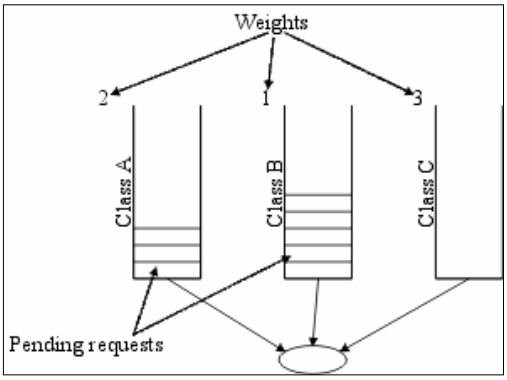

Figure 5-1: Work-conservation in the Cello disk scheduler

On the short term, i.e., within the round, this situation could be remedied by allocating more of the unused bandwidth to queue B. However, this is not possible in Cello, since the allocation of unused bandwidth is determined by the statically allocated queue weights.

If the situation described in Figure 5-1 persists, a good solution would be to reduce the weight of the queue C, and increase the weight of queue B. A possible alternative is to allocate unused bandwidth according to the length of the queues (the QoS-requirements associated with each queue must also be considered). However, neither of the solutions is possible in Cello; the former since the static determination of weights renders the reallocation of bandwidth impossible, and the latter since the same static weights are used for distributing unused bandwidth.

It should also be noted that Cello depends on a relatively detailed knowledge of the disk behavior, as it estimates service times based on the address of the requested data and the current disk head position. As we showed in Section 4.4, information about the physical disk geometry and the current disk head position is not necessarily available in modern disks, and this could represent a problem for Cello.

From the list in Section 5.2, we see that Cello provides a real-time service with deterministic or statistical guarantees, in addition to a high-throughput service with enhanced best-effort guarantees, and a best-effort low-latency service.

Another early approach was the disk scheduler of *MARS* (Massively-parallel And Real-time Storage) [13]. This disk scheduling algorithm maintains one non-real-time queue, and N real-time queues, and a fair queuing algorithm called Deficit Round Robin (DRR) [87] is used to allocate bandwidth to these queues. Each queue is assigned a service quantum (corresponding to a weight in Cello), and this quantum may be changed dynamically, through a system call (ioctl)[7].

As opposed to Cello, the class-independent scheduler, called *job selector* in MARS, is responsible for ordering the requests in the *work queue*, i.e., the queue containing the requests that are scheduled for service in the next round. The work queue is formed in the beginning of each round, and the requests are serviced using an elevator (SCAN) algorithm. Thus, the MARS scheduler meets the efficiency requirement from Section 4.5. However, a consequence of this solution is reduced support for a low latency service type, since no request is allowed to "jump into" the current work queue.

Another important difference between MARS and Cello is that the MARS disk scheduler allows the round length to vary. Implicitly, there is an upper bound on the round length, since the number of admitted clients must be limited, in order to avoid overload. This admission control requires assumptions about the performance of the disk, i.e., the disk bandwidth.

The work-conserving property is implicit, through the DRR scheduling, since the available bandwidth is proportionally allocated to the classes with pending requests; if a queue does not use all of its service quantum, the round becomes shorter (MARS uses proportional-byte allocation), instead of allocating the unused bandwidth to other queues with pending requests. The effect of MARS' work conservation is the same as in Cello, i.e., unused bandwidth is allocated according to the service quanta (weights) of the different queues.

It is difficult to evaluate the MARS scheduler both as a MMDBMS scheduling algorithm and against the list in Section 5.2, since the class-specific schedulers are not specified. By choosing appropriate schedulers, MARS has the potential of supporting

[7] Currently, MARS has only been implemented and evaluated with one real-time queue in addition to the best-effort queue, and without the possibility of dynamically changing the service quanta

several of the relevant combinations, but the lack of support for a low latency service is a problem.

In [64] several slightly different scheduler designs are discussed. The designs are all relatively simple, as only two service types are supported, discrete and continuous requests. The disk bandwidth is divided between the two classes, either statically, or dynamically, based on the service times of the discrete data requests. Disk efficiency varies between the different designs, as some use SCAN, while others use FCFS. The main problem of this approach is its limited flexibility. The only parameter that can be adjusted is the division of bandwidth between the two service types, i.e., the number of requests serviced or the time spent in each queue.

A more recent approach to mixed-media disk scheduling is the scheduler proposed in [103], part of the Prism server system architecture [102] (in the remainder of the thesis, we refer to this scheduler as the "Prism scheduler"). This approach is based on a two-level scheduling scheme, with three different service types: periodic, interactive, and aperiodic; with or without throughput guarantees. However, while Cello and MARS uses the class independent scheduler to allocate bandwidth to the queues, this algorithm makes each queue responsible for not using more bandwidth than allocated. This is achieved by having an admission controller for each queue that admits requests into the pool of requests scheduled for service, according to the allocated amount of bandwidth for the queue. Currently, only static bandwidth allocation is supported, and the scheduler is based on a constant round length.

At the start of each round, periodic requests, and aperiodic requests with minimum throughput guarantees are combined into SCAN order. These requests are then grouped into subgroups, based on disk location. If possible, interactive requests are grouped into the closest subgroup; otherwise they are served between the servicing of two subgroups, given that slack time allows it. If unused disk bandwidth still exists, it is used to serve interactive requests, and aperiodic requests without throughput guarantees, until the end of the round. Thus, this scheduler attempts to compromise between efficient disk accesses on the one hand, and support for interactive requests on the other. Still, the emphasis is on serving interactive requests, at the cost of disk efficiency.

The scheduling algorithm is partly work-conserving, in the sense that aperiodic and interactive requests are allowed to utilize all unused bandwidth; and there is no admission control for aperiodic requests without throughput guarantees. However, if there are no pending requests in the aperiodic and interactive queues, disk bandwidth remains unused.

Dynamic bandwidth allocation and adaptive performance guarantees are currently under investigation, but in its present version, the scheduler does not meet the flexibility requirement.

As opposed to Cello and MARS, the Prism scheduler also considers admission control. This admission control is based on load traces, i.e., a trace of the disk-demand of the stream. Thus, when a new stream requests admission, it presents its demand trace. The admission controller combines this trace with its current load trace, i.e., the sum of the demand traces of all admitted streams, to check if the new stream can be admitted. However, using this type of admission control represents a problem if users are allowed to interact with the playback of presentations, as is the case in our LoD-environment, since the admitted load trace will no longer be valid, as soon as a user pauses and restarts a presentation. This problem is not considered in the work.

From the list in Section 5.2, we see that, this scheduling algorithm provides a real-time service with deterministic and statistical guarantees for periodic requests, a best-effort low-latency service for aperiodic requests, as well as a high-throughput service with guaranteed bandwidth. Priorities are not supported.

In [49], a deadline-driven, priority-based disk scheduler is proposed. Each request has a deadline and a priority, and requests that will obviously miss their deadlines are considered lost, i.e., they are dropped. When the scheduler inserts a new request, it tries to satisfy the following conditions:

6. Insertion of a new request should not lead to deadline violations for higher priority requests already in the queue.
7. The number of requests that are lost should be minimal.
8. The new schedule should not violate the SCAN order.

This scheduling policy supports an arbitrary number of priority levels, and to the best of our knowledge, this is the only work on disk scheduling algorithms that acknowledges the fact that, within the same stream, different disk requests can have different importance. For playback of an MPEG video, for instance, a request for data constituting an I-frame is considered more important than a request for data constituting a B-frame. However, starvation is a potential problem, as is isolation between the classes. The policy is implicitly work-conserving, since the disk will never be idle as long as there are pending requests.

The most important problem of this approach is that there is really no support for bandwidth reservation. Combined with the attempt to compromise between priority,

deadline, and disk efficiency, this may lead to an unpredictable behavior, making it difficult to provide QoS-guarantees to the clients.

Using the service class parameter list from Section 5.2, we see that this algorithm provides a real-time service, but the guarantee level is restricted to something like enhanced best-effort. In addition, priorities are supported, but since all requests share the same queue, it is required that all clients have the same understanding of the priorities.

The ΔL disk scheduler in [10] is also deadline-driven. All admitted real-time tasks are guaranteed to meet their deadline, while best-effort requests are given priority, using slack time scheduling. This scheduler is quite similar to the scheduler in [72], and it shares most of the drawbacks.

In [38], a deadline sensitive SCAN (DSSCAN) scheduling algorithm is introduced. This algorithm supports periodic and aperiodic real-time requests, as well as interactive- and "ordinary" (i.e., non-interactive) best-effort requests. The algorithm selects the real-time request with the shortest deadline, and then picks as many interactive, and then real-time and best-effort requests, as possible without violating the deadline of the first real-time request. It is not possible to reserve bandwidth for any of the service classes, thus, this scheduler is in practice priority-based; real-time requests have the highest priority, then come interactive requests, and finally non-interactive best-effort requests.

This algorithm does allow efficient disk reads (to a certain extent), and it is work-conserving. However, the support for different QoS levels is too limited, as is the flexibility. In addition, the isolation between queues is not sufficient; under heavy load, the non real-time requests may not be served at all.

5.5 The Reference Group

We have now investigated four classes of existing disk scheduling algorithms. From these descriptions, it is clear that it is only algorithms of the mixed-media class that potentially have the qualities required for use in a MMDBMS-environment. However, as shown in Table 5-2, it is also clear that no existing scheduler in this class is ideal for such an environment.

Table 5-2: Summary of disk scheduler characteristics

("-" = not supported, "(X) = partly supported, "X" = supported)

	Efficiency	Multiple service types	Flexi-bility	Isolation	Work-conservation
[72]	-	(X)	-	X	(X)
Fellini [57]	(X)	(X)	-	X	(X)
Cello [85]	(X)	X	(X)	X	(X)
MARS [13]	X	(X)	(X)	X	(X)
[64]	-	(X)	-	X	(X)
PRISM [103]	(X)	(X)	(X)	X	(X)
[49]	(X)	(X)	-	-	(X)
ΔL [10]	-	(X)	-	X	(X)
DSSCAN [38]	(X)	(X)	-	-	(X)

In general, the main drawbacks of existing mixed-media disk scheduling algorithms are that they support few service types (typically two or three), they provide only static work-conservation and often with loss of disk efficiency, and they lack flexibility. Additionally, all algorithms provide a static set of queues, with a fixed set of service types. Thus, instead of the scheduling algorithm providing the services that the MMDBMS needs, the MMDBMS must accommodate to the service types offered by the scheduling algorithm. Only one algorithm supports priorities, but it does not allow different transactions to have different understandings of the priorities.

Among the existing scheduling algorithms for mixed-media workloads, we consider Cello, MARS, and the Prism scheduler to be the algorithms that come closest to the needs of our environment. All three schedulers have the potential ability to dynamically change the allocation of bandwidth to different queues, but only Cello has implemented this. In addition, they all (potentially) support multiple service types, and except for MARS, these include a low-latency service. However, none support dynamic creation and deletion of queues. Finally, the algorithms are all (partly) work-conserving.

For the remaining mixed-media scheduling algorithms, we consider these as being too limited in functionality and/or flexibility to be applicable in a MMDBMS-environment. As we introduce our disk scheduling algorithm in the next chapter, we will therefore use the three selected algorithms as a basis for comparison, i.e., they constitute the reference group.

Finally, an interesting observation is that, to the best of our knowledge, there exist no real-time, performance-oriented or mixed-media disk schedulers that support command

queuing. As explained in Section 4.4, the on-board controller on a modern disk is able to do its own internal scheduling. Through its first-hand knowledge of the physical disk geometry, the controller is able to achieve an optimal ordering of the requests, with respect to performance. However, existing disk schedulers allow only one request to be submitted to the disk at a time, and thereby lose the potential performance gain of disk-internal scheduling.

5.6 Summary

In this chapter, we have investigated existing disk schedulers, and evaluated their suitability for use in a MMDBMS. We found that only the mixed-media class of schedulers had the properties necessary for being considered as relevant. No scheduler in this class was found to have all the required properties, but we selected a reference group consisting of the three schedulers that met most of our requirements. This reference group will be used for comparison with our disk scheduling framework, which we present in the next chapter.

With respect to the four claims presented in Section 1.3, this chapter targets claim 4: Our descriptions of existing QoS-aware disk schedulers illustrate how these schedulers depend on detailed knowledge of the disk layout, as well as full control of the ordering of the disk requests. As described in Section 4.4, modern disks complicate these requirements, and reduced disk efficiency is the result.

Chapter 6
APEX Design

In this chapter, we present the design of the APEX disk scheduling framework. We assume that the reader is familiar with the description of the MMDBMS in Chapter 3, the disk scheduler requirements from Chapter 4, and the properties of the schedulers in the reference group that are described in Chapter 5.

This chapter serves two purposes: first, it presents our assumptions about the environment APEX is targeted for. This includes the components that surround, and interact with, APEX. Second, the chapter presents the design of APEX at a conceptual level, including data structures, interfaces, and components.

After this chapter, the reader should understand the principles of APEX; how it interacts with its environment, how queues and bandwidth are managed, and how the scheduling of requests is carried out. In the next chapter, we then describe the implementation of these principles.

6.1 Assumptions

APEX constitutes the bottom part of the storage manager, as shown in Figure 6-1. The buffer manager (BM) is the only component generating disk requests to APEX, and it does so in order to fetch pages requested by the PDSM, or to write dirty pages. These disk requests are submitted to APEX via the PSM, and each request is for a single page.

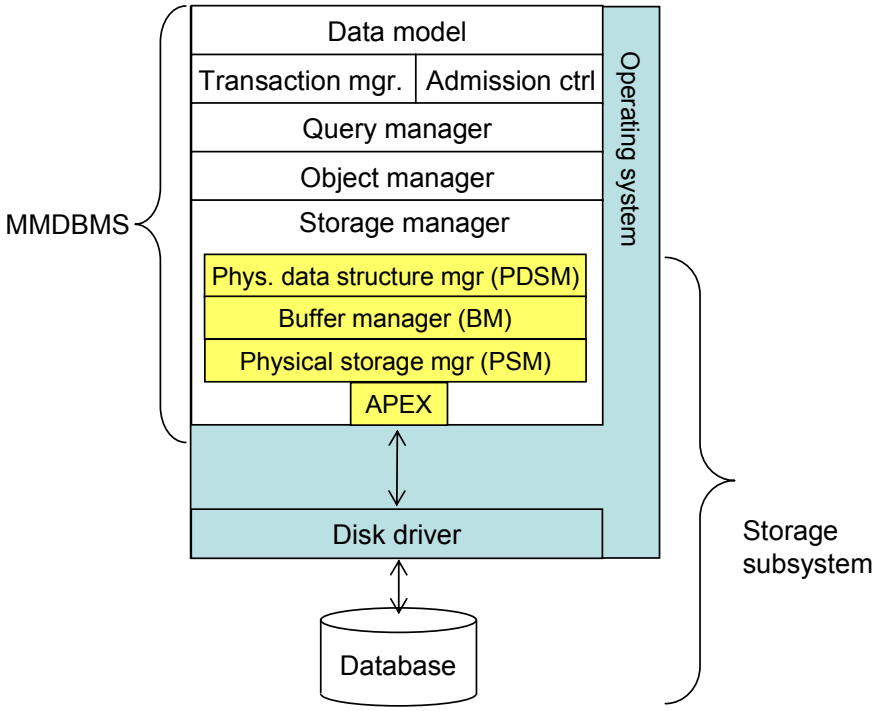

Figure 6-1: Server architecture

It is important to realize that as part of the APEX design, there are a set of underlying assumptions about the environment. These assumptions concern the data retrieval principles of the MMDBMS, how buffering and caching is performed, and how APEX interacts with higher-level components. In addition, we make assumptions about what functionality the operating system in general, and the disk driver in particular, offers to APEX. In the following sub-sections, we discuss each of these assumptions in more detail.

6.1.1 Round-Based Server

We assume that the server is round-based, i.e., at the beginning of each round, all real-time requests that are to be served by the disk during that round are submitted to APEX. In addition, we use the constant time-length (CTL) model [17], i.e., all rounds have the same length, typically between 0.5 and two seconds. It is important to note that, it is only the real-time requests that have to be submitted at the beginning of the rounds. All other disk requests can be submitted to APEX at any time during a round.

In round-based servers, continuous multimedia data (i.e., data retrieved by real-time requests) is usually transmitted from server to client using a double-buffering scheme [5, 65]. That is, the data that was read from disk during one round is transmitted to the client during the next round, and if a maximum of p pages are read from disk during a round, then $2p$ buffer frames must be reserved for the playback.

Longer rounds can improve disk efficiency, since the disk scheduler gets more disk requests to work with, and, consequently, more latitude to optimize the request ordering. However, longer rounds also mean that more buffer space is needed; consequently, choosing a round length becomes a trade-off between disk efficiency and memory usage. In addition, longer rounds mean higher latency, since the server needs to fill up one set of buffers before playback can start.

If necessary, APEX allows an intermediary solution, allowing real-time requests to have longer deadlines than the round time, but this requires that these extended deadlines are multiples of the round time. Alternatively, several different round times can be used, if the shortest round time is used as "master", and the other round times are multiples of this "master" round time.

6.1.2 Buffering and Caching

The BM of the MMDBMS tries to minimize the traffic to and from the disk, using page replacement algorithms. For non-multimedia data, this typically means using designated algorithms to try to keep the most relevant data in the buffer at all times. System catalog data is normally fetched from disk the first time it is needed, and then kept in memory for as long as the system is running [29].

For multimedia playback queries, we assume that the BM uses techniques like bridging (also known as buffering or caching [37, 48]), i.e., multimedia data used by one client is kept in the server buffer for later use by another client. However, we assume a high degree of interaction. Therefore, bridging can usually only be successfully applied during parts of presentations. For example, a user may make a pause in the presentation for a short period. When playback is resumed, another user, watching the same presentation, may be so close in time that bridging can be applied. However, as soon as one of the involved users do any further interaction, bridging may no longer be possible. Another example is two users "sharing" a presentation through bridging, but then, at some point, selecting different sub-presentations.

Since the techniques for reducing disk traffic are the responsibility of the BM, which is located above the PSM, the PSM itself, as well as the disk scheduler and the disk, are not aware of the usage of these techniques; it only experiences a reduction in workload. For example, if two clients are playing back the same presentation with a sufficiently large temporal distance, data must be fetched from disk for each of the presentations. One of the

clients then makes a pause in its presentation for a short time, and then resumes, such that the temporal distance between the presentations is reduced. The BM is now able to cache the data between the two playbacks, such that the presentation data only need to be fetched from disk once. However, to the PSM there is no difference between a client that has made a pause in its presentation and one that is using cached data; in both cases, the same reduction in workload is observed.

We also assume that neither the PSM nor APEX do pre-fetching, i.e., they only relate to the disk requests they have actually received. Pre-fetching is left to the higher layers of the MMDBMS, since these higher layers are in a much better position to know which data will be needed in the near future, due to their first-hand access to metadata. On the other hand, a modern disk may pre-fetch data into its internal buffer based on analysis of the incoming requests [106]. However, this pre-fetching is transparent outside of the disk controller, and it does not affect the functionality of APEX.

In addition, APEX assumes that the BM never submits a read request without having an available buffer frame in which to put the requested page. Thus, APEX is free to submit a disk request to the disk driver as soon as the request is received from the BM (via the PSM), independent of the length of a possible deadline of the request. Consequently, the BM may receive a requested page at any time between the submission of the page request (plus the time needed to actually serve the request) and the deadline of the request.

6.1.3 Interaction with MMDBMS Components

Like most other mixed-media disk schedulers, APEX implements the different service type using queues for the disk requests, one or more queues for each service type. All disk requests that are submitted to APEX must be associated with a queue; otherwise, APEX is unable to determine the service type that the requests should receive. Thus, for any query, the controlling transaction must have one or more queues assigned before it starts. Depending on the type of the transaction, this can be a pre-determined queue (for instance, one queue can be assigned to handle all disk requests from metadata retrieval queries), or a queue that is requested as part of the admission control for the transaction.

We assume that all existing queue types (i.e., service types) supported by APEX are registered in a table in the system catalog of the MMDBMS. Thus, as part of the admission control process of a user query, this table can be queried. When doing so, the requirements of the transaction are used as input, and the result of the query is the name of a suitable

queue type. This queue type name is included in a queue request to APEX, which then instantiates a queue of the requested type.

Requesting a queue from APEX is done as part of the admission control, and the request is only sent if the admission control component of the MMDBMS decides that there is sufficient bandwidth available. However, APEX is only one component among many on the server side, and until all relevant components on the server side have admitted the new transaction, the instantiation of the new queue is only tentative. We assume that the admission control component informs APEX about the result of the system-wide admission control, i.e., whether the queue request should be committed or not.

In the MMDBMS, transactions start and finish continuously, and in addition, active playback transactions may change their bandwidth requirements during playback. We assume that APEX is informed about such state changes for transactions. Thus, APEX can always keep the bandwidth distribution up to date, and this allows a very efficient distribution of disk bandwidth.

6.1.4 Interaction with the Operating System

Disk requests generated by the MMDBMS are handed over to the operating system (OS), which is responsible for serving the requests and transferring the data between MMDBMS buffer and disk. We assume that APEX constitutes the lowest layer of the MMDBMS (see Figure 6-1), and as such, APEX submits its scheduled disk requests to the OS (i.e., the disk driver). In other words, APEX is added as an extra layer below the PSM, which is the component that normally submits disk requests to the disk driver. Each request submitted to APEX from the PSM is a regular disk request, but in addition, we assume that the requests also contain the extra information needed by APEX, such as queue ID and possibly deadline. Given that the PSM is located in user space, such a configuration implies that the PSM uses raw disk access, i.e., it bypasses the file system in the OS. In other words, *APEX does not interact with the file system of the OS*, but instead, interacts directly with the disk driver.

In order to provide its services, APEX needs predictable access to the disk. This means that either APEX has the disk at its exclusive disposal, or that a fixed share of the total disk bandwidth is reserved for APEX and the rest may be used by the operating system. In the latter case, the disk scheduler located in the disk driver must be able to share the total disk bandwidth between the OS and the MMDBMS in such a way that the MMDBMS is

guaranteed to receive at least a certain share of disk bandwidth. This can be achieved by using, for instance, the current implementation of the MARS disk scheduler [13], which is implemented as an extension to the disk driver of NetBSD.

The functionality of APEX relies on the disk driver, or the disk itself, to sort the received disk requests in some optimized order (e.g., SCAN). Most existing operating systems, such as the BSD variants (NetBSD, FreeBSD, OpenBSD) [59], Linux [14], and Windows 2000 (with NTFS file system) [20] do this, and in any case, modern disks ultimately do the final scheduling of requests [88], as described in Section 4.4.

In addition, APEX relies on the existence of asynchronous I/O, i.e., when it submits a request to the disk driver, the call returns immediately, and when the serving of the request is finished, the disk driver gives a notification to APEX, either as a signal, or as a call to a specified function. This is also a reasonable assumption, since asynchronous I/O is supported by the POSIX standard, using the `aio_read()` and `aio_write()` system calls for single requests and the `lio_listio()` system call for multiple requests, and is implemented in, for example, GNU C library [34], and Windows 2000 [91].

6.1.5 Server Push vs. Client Pull

There are two main principles for moving data from the server to the client, namely server-push and client-pull. The server-push principle implies that a component on the server side (in the case of a MMDBMS, this is the presentation manager) is controlling the sending of data to the client. The client controls the presentation playback, by sending VCR-type commands to the server, such as play, pause, and stop.

Client-pull means that the client actively requests the multimedia data from the server, using a MMDBMS cursor. Thus, the client is in control of the data transfer, and the server can only try to make sure that the data is in the server buffer when requested.

Consequently, the main difference between server-push and client-pull is who controls the sending of data. However, for the OM and the components below it, there is no difference:

- The OM must make sure that the data is ready when requested, independently of whether the presentation manager (within the server) or the client is the requester.
- In both cases, the OM has a query execution plan which informs it about (probable) future operations.

- In both cases, the consumption of data from the MMDBMS buffer (i.e., sending the data over the network) may deviate from what is defined in the query execution plan, since user interaction may occur (this makes sense, since also in the server-push model, it is ultimately the client that controls the presentation manager). Therefore, the BM on the server side must adapt its disk block retrieval rate to the consumption rate of data in the buffer, both for server-push and for client-pull.

6.2 APEX Architecture

The basic architecture of APEX is that of a two-level disk scheduler, an architecture which is common in many mixed-media schedulers. In other words, there is at least one queue for each service type, and one queue for the requests scheduled for service. However, while the majority of other two-level schedulers, including Cello and Prism, are depending on complete control of the final scheduling, APEX leaves this task to the disk driver and/or the disk.

The basic principle of APEX is that instead of submitting one request at a time to the disk driver, a number of requests are assembled into a *batch*, and all requests in the batch submitted collectively to the disk driver. Thus, with APEX, the final scheduling of disk requests can be performed by the disk controller, taking into account low-level information that is not available on higher layers of software.

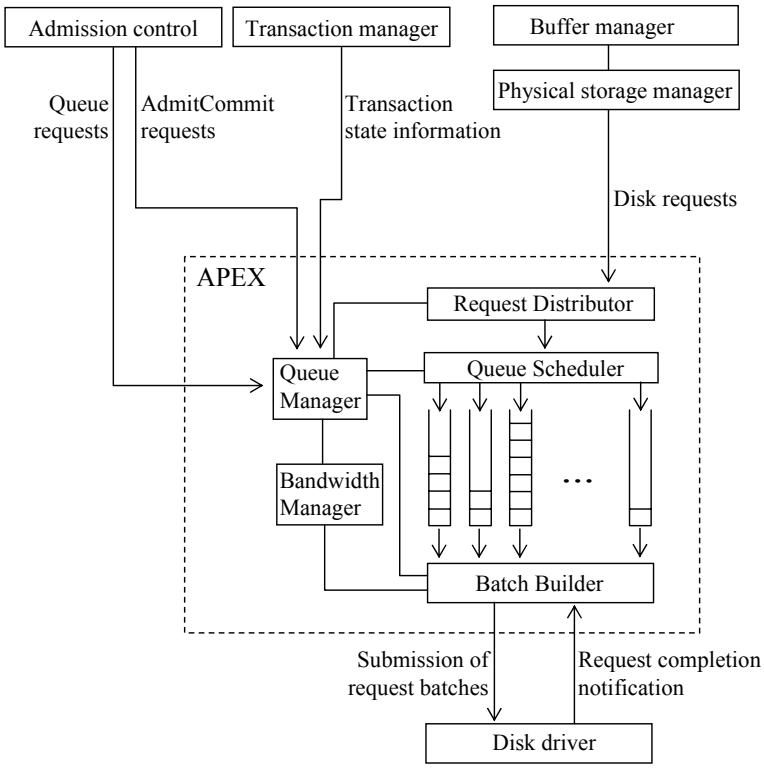

Figure 6-2: The APEX architecture in a MMDBMS context

6.2.1 Interfaces

APEX effectively functions as the interface between the MMDBMS and the disk driver. As shown in Figure 6-2, there are four interfaces between MMDBMS-components and APEX, as well as two interfaces between APEX and the disk driver:

- *Queue request interface*: Before a new query can start, an APEX-queue is needed, to which the disk requests of the query can be submitted. This queue request specifies the type of service that is needed, together with transaction ID, and possibly additional parameters such as the amount of bandwidth to be reserved. As mentioned in Sub-section 6.1.3, descriptions of the queue types (i.e., service types) offered by APEX are stored in the system catalog. Requesting queues is part of the admission control, and this interface is therefore used by the admission control component.
- *AdmitCommit interface*: APEX is informed about the completion of the global admission process through the AdmitCommit interface. The ID of the requesting transaction is included, and the return value is the ID of the queue (which is to be included with every disk request for that queue). This interface is used by the admission control component.

- *Transaction state interface*: This interface is used to notify APEX about the status of transactions, i.e., when they start and finish. In addition, this interface is used to change bandwidth reservations for running transactions. The transaction state interface is used by the transaction manager.
- *Disk request interface*: This interface is used by the PSM to submit disk requests to APEX. Associated with each disk request are a queue ID, a transaction ID, and, if the request is submitted to a real-time queue, a deadline. Calls to this interface are asynchronous, i.e., each call is returned as soon as the request is placed in a queue.
- *Batch submission interface*: This is the interface used by APEX to submit disk requests to the disk driver. The requests are submitted asynchronously to the disk driver.
- *Completion notification interface*: This is an interface used by the disk driver to notify APEX each time the disk is finished serving a disk request. Thus, if APEX submits a batch consisting of *n* disk requests, the disk driver will subsequently send a total of *n* completion notifications back.

From a functional point of view, APEX consists of two parts: the *request management* (see Section 6.3) is responsible for receiving and scheduling requests, while the *queue management* (see Section 6.4) is responsible for managing the queues, and allocating bandwidth to these. The queue request interface, AdmitCommit interface, and the transaction state interface are all part of the queue management, while the three remaining interfaces belong to the request management.

6.2.2 Components

From Figure 6-2, we see that APEX consists of five components:

- The *request distributor* receives disk requests from the BM (via the PSM), and determines the queue in which the request should be placed. This is done using the queue ID that is attached to each request.
- The *queue scheduler* receives the request from the request distributor together with a reference to the queue in which to place the request. It is the task of the queue scheduler to place the request in the correct position within that queue, and the actual position depends on the queue type. For instance, in a real-time queue the requests are ordered according to their deadlines, while in a best-effort queue, the requests may be ordered FCFS.

- The *batch builder* picks disk requests from the instantiated queues, assembles them into batches, and submits these to the disk driver. The number of requests picked from each queue depends on the amount of bandwidth assigned to that queue. In addition, the batch builder receives notification from the disk driver each time the disk is finished serving a request.
- The *queue manager* receives queue requests and AdmitCommit calls from the admission control component, as well as Transaction State calls from the transaction manager. Based on these calls, the queue manager makes sure that all required queues are present, instantiating and removing queues as necessary.
- The *bandwidth manager* distributes the available disk bandwidth between the instantiated queues, according to bandwidth reservations and transaction states.

The separation between request management and queue management mentioned in Sub-section 6.2.1 is also reflected by the components of APEX. The request distributor, queue scheduler, and batch builder together constitute the request management, while the queue manager and bandwidth manager constitute the queue management.

6.3 Queue Management

APEX is able to manage queues dynamically. Each offered service type is realized by means of a queue, which can be instantiated and removed dynamically, as needed. Each service type that APEX can provide is specified in the form of a queue template, which is a description of the service type, based on the parameters we discussed in Section 5.2.

The service type specification tells the queue scheduler how to order the requests in the queue. These specifications are also stored in a queue description table in the system catalog of the MMDBMS, each identified with a unique “name”. When the queue manager in APEX receives a queue name during a queue request call, the characteristics of that queue can be found in the queue templates. Note that, some service types require bandwidth reservation, and if so, the amount of bandwidth to be reserved is included in the request. However, no bandwidth reservation is actually made at this time.

The queue request call to APEX is made as part of an admission control process for the transaction in question. If this admission request is globally accepted in the LoD-system, APEX is then notified through the AdmitCommit interface. The queue manager in APEX instantiates a new queue if necessary, and returns a handle for the queue (i.e., a queue ID that uniquely identifies the instantiated queue) to the caller.

When the transaction is started, APEX is notified through the Transaction state interface. If a bandwidth reservation was requested in the queue request call, this reservation is now made. An END- or ABORT-message leads to the corresponding bandwidth reservation being released.

The transaction state interface can also be used to change bandwidth reservation for a running transaction. For example, if a user changes from ordinary playback to fast forward, this may require a change in required bandwidth, with a corresponding change in the bandwidth reservation.

Multimedia playback transactions can also be paused during runtime, causing reserved bandwidth to be left unused. How to re-distribute such bandwidth is policy-dependent, but regardless of the policy, APEX is able to re-distribute the bandwidth without efficiency loss.

When several transactions have requested the same queue type, these transactions can, in many cases, share a single queue. For instance, two transactions both requiring deterministic real-time guarantees, can use the same queue, which then needs a bandwidth reservation equal to the sum of the bandwidth requirements of the two transactions. On the other hand, since APEX supports an arbitrary and dynamic number of queues, we can also assign the two transactions to different queues, which allows us to efficiently isolate transactions from each other with respect to bandwidth usage.

It could be argued that dynamic queues are unnecessary, and just add overhead to APEX. However, as we will show in Section 7.4, the overhead of dynamic queues is low, and in addition, by removing unused queues, the overhead of the request management is reduced, since there are fewer queues to visit each time a batch is being assembled (obviously, this is dependent on the number of queues that are required). The use of dynamic queues actually enables APEX to support a “transaction-oriented” service model, with a separate queue for each transaction.

6.4 Request Management

APEX is basically a hybrid disk scheduler, in the sense that it is deadline-driven, while at the same time being round-based. The key principle is to postpone the submission of requests to the disk, in order to gather more disk requests, and if possible, submit these as a batch to the disk.

The number of requests that can be contained in a batch is dependent on the deadlines (if any) of the requests, and the time left in the current round: First, the pending request with the shortest deadline is found, by checking all real-time queues. We define this as the *controlling request* of the batch to be assembled. The set of instantiated queues that are allowed to provide the controlling request is called the *candidate queues*. Based on a pre-computed value for the service time of a disk request, we then compute how many requests B that can be served before the controlling request. We then pick B requests from the instantiated queues, and submit all the requests, including the controlling request, as a request batch to the disk driver.

Our rationale behind submitting requests in batches is that modern disks are very complex, as described in Section 4.4. Consequently optimizing the final disk schedule requires knowledge of low-level details such as bad block management, controller-level caching, etc., and this information is not available to higher level software. By submitting batches of requests to the disk, we leave the final optimization of the disk schedule to the component with the best knowledge of how to do so, namely the disk controller.

Normally, the cost of assembling requests into batches, instead of submitting them as they arrive, is increased response time. However, we reduce this effect through two measures: First, the batch assembly is, in practice, load dependent, such that in low load situations the scheduler will effectively function as an EDF/FCFS scheduler. Second, the batch builder supports a special low-latency service, provided on a best-effort basis.

6.4.1 Extended Token Bucket Model

When a new batch is being assembled, the batch builder visits each instantiated queue. The queues are always visited in the same order, and to avoid starvation of the backmost queues, it is therefore necessary to limit the number of requests taken from each queue. Our approach for doing so is based on the token bucket principle, but with a number of modifications to make it suitable for controlling disk bandwidth.

The token bucket model [67] is normally used for traffic shaping and characterization in networks. Two parameters are used to control the outgoing flow, namely rate r and maximum burst size b. Thus, b determines the "depth" of the bucket, while r is the rate at which the bucket is filled with tokens.

We use the token bucket model for controlling both the allocation of disk bandwidth to the different queues and the allowed burstiness within each queue. One token corresponds

to the serving of one disk request, i.e., in order to pick a pending request from a queue and put it into the batch being assembled, there must be a token available for that queue[8].

We assume that each disk request is for one disk block (corresponding to one database page), thus, the token rate *r* for a queue translates directly into disk bandwidth for that queue. For each queue, APEX keeps track of the token rate, as well as the last time the queue was visited by the batch builder. Thus, each time the batch builder visits a queue, the assigned rate is checked, and compared to the number of milliseconds passed since the last time the queue was visited. If sufficient time has passed, one or more tokens are added to the queue, a corresponding number of requests picked from the queue, and the time of the visit is registered for the queue. We refer to this procedure as *token update.*

For example, a queue is assigned a bandwidth of 10 pages/s, i.e., a new token should be assigned to the queue every 100ms. We assume that a token is added at *t*=2000ms (the assignment time is registered). At *t*=2090ms, the batch builder visits the queue. Only 90ms has passed since the last token was added, so no new token is added. The next visit is at *t*=2140ms. Now, 140 ms have passed since the last token assignment, so a new token is added, and the "last assignment time" is set to 2100. If the batch builder visits the queue again at *t*=2310ms, two new tokens are added, and the "last assignment time" is set to 2300.

However, availability of tokens is not the only requirement for serving a request. In addition, deadline requirements and service times are also taken into consideration. Thus, if adding another request to the batch means that a deadline may be violated, the request is not added, even if there are available tokens.

While *r* is the parameter for controlling the bandwidth allocation, *b* is used to fine-tune the bandwidth allocation to the characteristics of the workload for the queue, and to compensate for the fact that the token update operations not necessarily happen at regular intervals. In particular, we use *b* to limit the number of tokens that are allowed to accumulate. Consider, for instance, if the queue from the example above, has not received any requests for the last five seconds. If a large burst of requests then arrives in this queue, we could allow 50 requests to be picked from this queue. However, this may affect the following queues, and it may therefore be necessary to set *b* to a lower value.

Adjusting *r* and *b* according to the workload of the queue is done in two ways. Initially, the values are set based on the characteristics of the presentation and the requested QoS-

[8] This condition is relaxed during work-conservation, as we will show in Sub-section 6.4.3.

level. We assume that when multimedia objects are stored in the database, the characteristics, such as bandwidth requirements, are stored as metadata, and are used as basis for resource reservation in the MMDBMS. For example, a video stored in the database may have an average bandwidth requirement of three pages/s, and a maximum requirement of five pages/s. If the user wants a deterministic QoS-guarantee, then a bandwidth of five pages/s (i.e., five tokens/s) must be reserved for the playback, and with a bucket depth of five tokens. However, if the user requests a statistical QoS-guarantee, then it may be sufficient to reserve three pages/s, and then set the bucket depth to five tokens.

In addition, APEX keeps track of *r*, *b*, and the actual number of tokens available, for each queue. These variables enable the use of a controller with a feedback loop, which fine-tunes the bandwidth allocation. However, this has not yet been implemented.

6.4.2 Assembling Batches

Assembling a batch containing real-time requests, means that the batch builder needs an estimate of the time it takes to service a request. However, the unpredictability of modern disks complicates making such an estimate. In addition, since the final scheduling is left to the disk driver/disk, we do not know what the final schedule of a batch of requests will look like, making it even worse to estimate the service time.

Given this unpredictability, we use a fixed, predetermined estimate for this service time, called t_{es}. The main factors that determine t_{es} are disk performance, the disk block size, and the data placement scheme. In our current implementation of APEX, t_{es} is initially determined using manual adjustment, and then a very simple, automatic fine-tuning is applied, based on comparing the estimated batch service time with the actual time used.

An inaccurate setting of t_{es} is easily discovered by comparing the estimated and actual service times as described above. Normally, there will always be a small deviation between these two values, but by monitoring trends in the deviation a more advanced, feedback-based controller that automatically adjusts t_{es} is feasible to realize.

When assembling the batches, there are two conditions that should be met: First, the assembly of the next batch should be postponed for as long as possible, in order to wait for more disk requests to arrive. Second, as long as there are pending requests, the disk should never be idle. Therefore, the next batch must be ready for submission before the current batch is finished.

We meet these conditions by taking advantage of the fact that APEX is notified each time a disk request is finished: Since APEX knows both the original size of the current batch and how many requests remain to be served, it can always maintain an updated estimated finishing time, using t_{es}.

Based on a worst-case estimate for the time to assemble the batch, we know when to start the assembly, t_{bas}, relative to the number of remaining requests in the current batch. This is illustrated in Figure 6-3.

In the following, we assume that the assembly of a new batch starts when there is *one* request left to be served in the current batch. The batch assembly algorithm then works as follows:

1. *Find controlling request*: In order to find the controlling request, which has deadline t_{ed}, the batch builder checks the first request in each of the candidate queues. If there are no pending real-time requests, the end of the round is used as a substitute for the controlling request. If a request is found, token update procedure is run, the request is moved to the batch being assembled, and the number of tokens for the queue reduced by one.
2. *Compute batch size*: $B = (t_{ed} - t_{start}) / t_{es}$, where $t_{start} = t_{bas} + t_{es}$, and t_{bas} is the time at which the assembly started. Computing B in this way means that we always assume the worst-case condition that the controlling request is served last in the batch. The actual serving order of the request is dependent on how the disk or disk driver sorts the requests in the batch.
3. *Batch assembly*: Visit all instantiated queues, and for each queue, run the token update procedure, and reduce B according to the number of requests that were picked. Thus, a batch will normally contain requests from several queues, but the actual mix of requests is dependent on bandwidth allocations and the number of pending requests in each queue. Thus, the request mix will normally vary from batch to batch.
4. *Stop conditions*: The batch assembly continues until either B=0, or until all queues have been visited. Finally, the estimated finishing time of the batch is computed: $t_{ef} = B * t_{es}$.

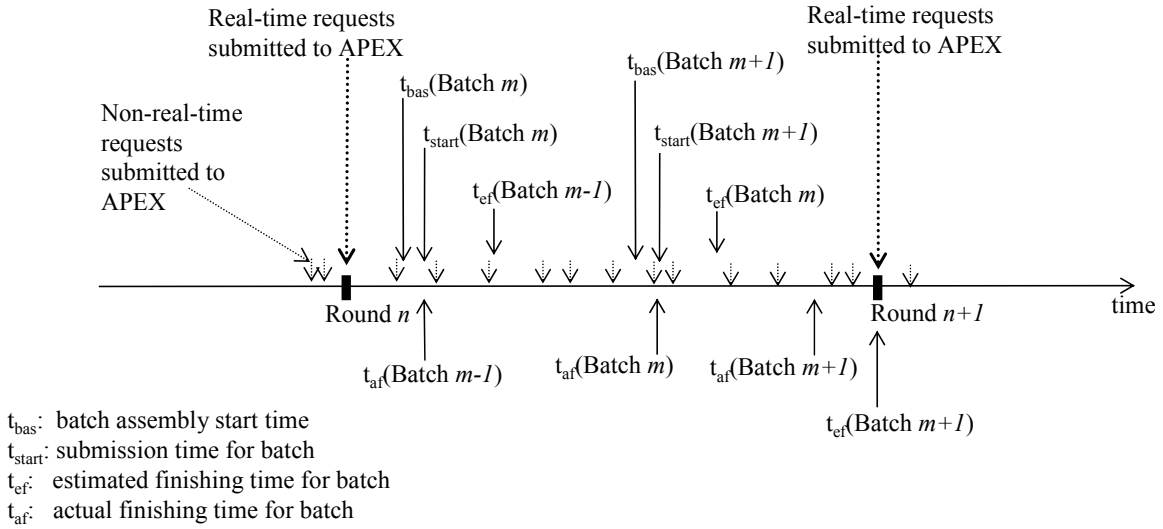

Figure 6-3: Batch assembly timing

Note that, with one global round time, the shortest deadline will always be the end of the round, and the principle of searching for a controlling request can therefore seem unnecessary. However, as described in Sub-section 6.1.1, APEX allows requests with different deadlines, or several round times in the same system, and this requires the use of a controlling request. Also note that, as shown in Figure 6-3, more than one batch may be assembled and served in each round.

We originally designed APEX as a pure deadline based scheduler, working without any notion of rounds. This required that APEX knew the minimum deadline that could occur in any queue, and the maximum size of such a request. When APEX computed the batch size, it always took into account that a request with this shortest deadline could arrive just after a batch was submitted. However, this solution limited the possible batch size, and thereby the disk utilization. This version of APEX was also implemented in our simulation environment (described in Chapter 8), and preliminary simulations confirmed the reduced performance.

As a consequence, we chose not to pursue this version of APEX for use in our LoD-scenario. However, the pure deadline version of APEX still outperforms the EDF disk scheduler, and in other scenarios, such as real-time environments that do not require continuous media playback, this version can be useful.

6.4.3 Work-Conservation

As we have already described, our scenario is characterized by variable bit-rate streams, a high degree of user-interaction, partial bridging of presentations, bursty user queries, and aperiodic real-time requests (e.g., still images synchronized with a video). All these factors contribute to a highly shifting workload for APEX, which, in turn, means that there will often be discrepancies between allocated bandwidth and the actual usage of bandwidth within queues. Typically, some queues can have pending requests, but no tokens, while other queues may have tokens but no pending requests.

To alleviate this, the batch assembly algorithm is extended with an active work-conserving facility that can re-distribute unused bandwidth. The principle is very simple: if a batch being assembled still has room for more requests after all queues have been visited, the batch builder starts selecting requests from the queues once more, but this time *without* taking tokens into consideration. These requests are then added to the batch that is currently being assembled, which means that the disk can serve these requests *with the same efficiency as ordinary requests*. In fact, adding more requests to the batch contributes to disk efficiency, since a large batch means more requests over which to amortize the positioning work of the disk head.

In addition, the work-conserving phase serves another important purpose: during the normal phase, only queues with tokens are visited, which means that best-effort queues are not served during this phase. Instead, requests in such queues must rely on the work-conserving phase.

In our current design, the order in which the queues are visited during the work-conserving phase is the same as during the normal batch assembly phase. However, as will be shown in the next chapter, APEX allows full control of the number of requests to pick from each queue during this phase. This means that the work-conservation of APEX can easily be adjusted to implement different policies:

- *General*: The bandwidth is distributed among all queues with pending requests, as described above.
- *Need-based*: The queues are re-visited in order of need, given some sorting criterion. For instance, using queue length as sorting criterion, the queue with most pending requests is re-visited first.
- *Dedicated*: The unused bandwidth is dedicated to one or a few queues. In this case, the extra bandwidth can either be used as a replacement for (some of) the explicitly

allocated bandwidth, or it serves as an extra reserve, e.g., for handling bursts. If the selected queue(s) is unable to use all the bandwidth, the rest is distributed using general or need-based distribution. One example of this policy is to dedicate all unused bandwidth to the best-effort queues, assuming that other queues receive sufficient bandwidth through their bandwidth reservations.

If we compare the work-conserving scheme of APEX with the schedulers in the reference group, we find that they all provide some form of work-conservation. Cello has, like APEX, a separate work-conservation phase, where queues "eligible for utilizing unused bandwidth" are re-visited, and requests possibly picked. MARS is implicitly work-conserving through its variable round-times: when all queues have been visited, the serving of the selected requests is started, even if the round is not "filled". Thus, the next round is started earlier, and the disk is kept busy.

The Prism scheduler dedicates all unused bandwidth to two queues (interactive queue and aperiodic queue), and as such use a "dedicated" policy. Cello and MARS, on the other hand, use a "general" policy, but for both schedulers, the bandwidth distribution scheme used during the work-conserving phase is the same as the scheme used during the normal phase.

6.4.4 Low-Latency Service

As described in Sub-section 6.4.2, we use an estimated service time t_{es}, when assembling batches. However, since this value is a slightly conservative estimate (in our simulations, 76% of the actual service times were smaller than t_{es}), the batch will normally finish earlier than the estimated finishing time, t_{ef}.

Since the batch builder knows both the remaining size and t_{ef} of the current batch, the slack time can easily be computed, using t_{es}. If this slack time is larger than t_{es}, the batch builder allows a request from a special low-latency queue to be inserted into the current batch. Since no guarantees can be given about available slack time, this service is offered on a best-effort basis, but by adjusting t_{es}, the actual service level can be adjusted; a larger t_{es} gives more slack time, but reduces the maximum batch size, and thereby the disk efficiency.

When a low-latency request is submitted by APEX to the disk driver, we assume that the request is added to the batch currently being served by the disk, and that the request is

added in sorted order. This means that inserting low-latency requests does not affect disk efficiency.

The service time of a low-latency request is estimated as follows: when a low-latency request is inserted, half the batch remains to be served, on average, and the low-latency request can be expected to be served in the middle of the remaining batch. This means that the average service time for a low latency request is approximately $B/4 * t_{es}$. In the worst-case, a batch has just been submitted, and the low-latency request is served last in the batch, giving a service time of approximately $B * t_{es}$.

If the disk is idle when a low-latency request arrives in APEX, the request is normally submitted to the disk immediately, without waiting for a new batch to start.

6.5 Different Contexts

Through the use of dynamic queues and a modular design, the APEX scheduling framework can be configured for a number of different environments. In Sub-section 5.2.5, we listed the requirements of some application areas, using the four characterization parameters. The service types of APEX are based on these four parameters, and APEX is therefore able to support all these application areas.

Furthermore, a MMDBMS represents a very demanding, dynamic environment that utilizes all functionality of APEX, and this makes it suitable as an example environment. Other environments may only require a subset of the functionality, but since APEX has a modular design, it is easy to remove or simplify components. For example, in an environment with a static, stable workload, most of the queue management can be left out. Instead, APEX just reads the required configuration data from a file, and instantiate the necessary queues.

It should also be noted that the core of APEX, which is the assembly of disk request batches, must always be present, regardless of configuration. Thus, a minimum requirement is that at least one queue is instantiated, such that the batch building can take place. Also the interaction between APEX and the disk driver is common in all configurations.

6.6 External Factors

There are several factors in the storage subsystem that can affect the functionality and performance of a disk scheduler. For instance, using long round times provides the scheduler with greater freedom to reorganize disk requests according to the optimization goal. In this section, we investigate four important external factors that can influence the disk scheduler, and discuss their impact on the scheduling principles used in APEX: data placement, disk block size, round time, and disk characteristics.

6.6.1 Data Placement

During assembly of a batch, APEX does not consider the placement of the data to be read of written. The idea behind this approach is that the unproductive positioning work performed during the serving of a batch is amortized over the requests in the batch. A large batch means more requests on which the unproductive work can be shared, which implies that the unproductive work constitutes a lower relative share of the overall work.

In addition, it is a basic assumption that the LoD-system is a multi-user system. Thus, multiple students and teachers access the system concurrently, working with different multimedia objects. Consequently, at any given time, we can expect the requests pending in the scheduler to address data all over the disk, regardless of data placement scheme.

However, if we consider the data requested for each individual transaction, the data placement scheme does have an impact. Especially for multimedia playback queries, more than one disk block (i.e., database page) is usually requested in each round. If we assume random data placement, and a transaction requests b blocks in a round, this means that b positioning operations have to be made. If, on the other hand, we use extent-based or contiguous allocation, only one such positioning operation is necessary. Consequently, reading the b blocks can be expected to go considerably quicker in the latter case.

In practice, this means that the estimated time for serving a request, t_{es}, must be lower when using extent-based or contiguous allocation, than when random allocation is used. As described in Sub-section 6.4.2, the actual setting of t_{es} is done based on comparing the estimated service time with the actual service time; consequently, it is easy to adjust APEX according to different data placement schemes.

6.6.2 Disk Block Size

In today's modern disks and operating systems, a disk block size between 2 KB and 64 KB is common, and the trend is towards larger block sizes, as disks become faster [39]. Determining the block size is a trade-off between disk efficiency, on the one hand, and latency on the other: a small block size means short transfer time, and the block can be read or written quickly. However, smaller blocks means that, for a given amount of data, more blocks must be fetched, leading to more positioning operations.

For instance, reducing the block size from 64 KB to 16 KB means a quadrupling of the arrival rate. On the other hand, the service time is *not* reduced to one-fourth, since only the transfer time is reduced, and not the seek time or rotational latency [63].

To APEX, reducing the block size would mean that each disk request is served slightly faster by the disk, since the transfer time is reduced. The average positioning time would not be changed, however, provided the data placement scheme is the same. Consequently, reducing the block size means that t_{es} should be slightly reduced.

6.6.3 Round Time

As mentioned in Sub-section 6.1.1, longer round times are beneficial for the disk scheduler, since it is given more requests to work with when setting up the schedules. For example, for a SCAN scheduler, longer round times means that more requests can be included in a sweep over the disk surface, which, in turn, means higher disk efficiency.

The drawback of longer round times is that the latency increases, since data retrieved during one round is sent to the user during the next. Thus, doubling the round time means doubling the latency. In addition, more data must be buffered before being sent to the user, meaning that the amount of buffer space must be increased.

For APEX, the length of the rounds directly affects the maximum size of the batches, since the round length affects the earliest deadline, t_{ed}. The size of the batches affects the efficiency of the disk, thus, the round length indirectly affects the disk efficiency. However, as mentioned above, this is not limited to APEX; practically all disk schedulers, with a few exceptions like FCFS and EDF, are exposed to this effect.

Furthermore, APEX needs to be able to accumulate requests for the batch building principle to work. If the round time becomes very short, we expect this principle to break down, and APEX will be reduced to a EDF or FCFS scheduler, depending on what types of queues that are instantiated. Assuming an average service time of a disk request in the

magnitude of 10 ms, this implies that a round time of perhaps 100 ms or less can cause problems. Since round-based servers normally use a much longer round time than this, typically 500 – 2000 ms, we do not consider this to be a problem.

6.6.4 Disk Characteristics

The most obvious characteristics of the disk are seeking time, rotational latency, and transfer time. Reducing these times means both lower latency and higher throughput for the disk. In addition, as discussed in Section 4.4, the ability of the disk to pre-fetch and cache the correct data also influences the performance.

When APEX submits a batch of requests to the disk, it is reasonable to expect some of these to be served by the disk-internal cache, while others have to be read from the disk surface, and the service times of the individual requests will therefore vary considerably within a batch. However, since a batch consists of many requests, the average service time for the requests in a batch will vary much less.

Consequently, the disk characteristics have little impact on the functionality of APEX. Similar to data placement scheme and disk block size, it is a question of adjusting t_{es} to match the disk.

6.6.5 Conclusion

As described in the previous sub-sections, the external factors do not represent any problem to APEX; it is simply a matter of adjusting t_{es} to match the premises set by these factors, and this can to a certain extent be done automatically.

Furthermore, it is important to notice that these factors normally are static: the disk characteristics only change if the disk is replaced, and once the system is running, data placement scheme, disk block size, and round time cannot easily be changed.

The batch building principle makes APEX relatively independent of these external factors, but this is to a certain extent dependent on the batch sizes. With a small average batch size, there may be large variations in the relationship between estimated (based on t_{es}) and actual service times for the batches, which, in turn, can cause problems for the automatic adjustment of t_{es}.

6.7 Example

In order to illustrate the functionality of APEX, we now present an example: A student is searching for information about a specific topic, and queries the LoD-database for it. The student enters the relevant search criteria, and submits them as a metadata retrieval query (see Sub-section 3.6.1). The disk requests generated by this query are sent to the common metadata retrieval query queue in APEX, which we assume is always present. The result of the query is a list of presentations that contain the requested information. We now assume that the student starts playback of one of these presentations, consisting of video, audio and subtitles.

The first step of the playback is admission control, and during this phase, the system catalog is queried, in order to find service types for the disk requests, which match the QoS-requirements of the student. These QoS-requirements can be provided by a profile associated with the user, or the user explicitly states the QoS-level he or she wants, before the playback starts. In addition, the capabilities of the network and the client system may affect the possible QoS-levels.

We assume that two suitable service types are found (one for audio/video disk requests, and one for disk requests for subtitle data), and the "names" of these service types are submitted to APEX through the queue request interface, together with IDs of the corresponding sub-transactions, and other relevant information. If the admission control component finds that there is sufficient disk bandwidth available, queue requests are submitted to APEX, which in turn registers the requests, and returns a positive acknowledgement for each of the requests. We assume that all server components admit the new playback request, and APEX is informed of this through the AdmitCommit interface. The requested queues are now instantiated (assuming no suitable queues already existed), and the IDs of the queues returned. The reserved bandwidth is still not associated with the queues, though, since the transactions have not yet started.

When the playback starts, APEX is informed through the transaction state interface. The bandwidth reservations are now made, and APEX is ready to receive disk requests generated by the multimedia playback query. If the bandwidth requirements of the playback change, for example because the student wants better QoS, APEX is notified through the transaction state interface, and updates the reservations correspondingly (we assume that the admission control component found that there were sufficient bandwidth available).

If the student makes a pause in the playback, APEX makes the unused bandwidth available to other transactions during the work-conserving phase, and when the playback ends, APEX is notified through the transaction state interface, the reserved bandwidth is released, and the queues are possibly removed.

6.8 Summary

The goal of this chapter has been to introduce the reader to APEX, at a conceptual level. We have described the assumptions we make about the environment of APEX, and how APEX interacts with it. Furthermore, we have described the overall architecture of APEX, its interfaces, and the functions of its components. We have also briefly explained how parts of APEX can be left out, depending on the application area.

However, the main focus of this chapter has been on the scheduling principles that APEX builds on. We first presented the queue management, showing how queues can be dynamically instantiated and removed, in order to keep the computational overhead at a minimum. Next, we described the principles behind the request management, with its use of an extended token bucket model to distribute bandwidth between the queues.

Then, we thoroughly described the principles behind the two key features of APEX, namely the assembly of request batches, to increase disk efficiency, and the work-conservation facility that redistributes unused bandwidth without loss of disk efficiency. Finally, we discussed the impact of external factors on APEX, such as data placement scheme and disk block size.

With respect to the claims presented in Section 1.3, we have shown how APEX creates batches of disk requests, which is our main argument for Claim 3 (Good combination of QoS-support and high utilization). In addition, the description of APEX used in different contexts, in Section 6.5, supports Claim 2 (configurability of APEX).

In the next chapter, we will describe the implementation of APEX, including data structures, interfaces, and components.

Chapter 7

Implementation of APEX

In this chapter, we present the implementation of APEX. All functions, except some minor ones, are presented in C-like pseudo-code, in order to provide an unambiguous explanation of the functionality. We also describe the data structures used, as well as the interfaces that APEX offers to its environment, and we analyze the complexity of APEX.

To understand this chapter, we assume that the reader knows the assumption and concepts described in Chapter 6. In addition, the reader should be familiar with the most relevant related work, described in Chapter 5, as well as the requirements analysis in Chapter 4.

7.1 Data Structures

APEX supports a number of different queues, which can be instantiated and removed dynamically. To realize this functionality, we have chosen to implement the queues as a linked list, as shown in Figure 7-1. The start of the list (`rootQ`) is a static data structure that is always present, and provides an entry point to the queue list. As shown in Figure 7-2, this structure is also used to store global state information that is frequently used, including the estimated request service time, t_{es}.

From Figure 7-1, we also see that each queue is controlled by a *queue header*. Each queue header contains a description of the queue, realized as a set of attributes (see Figure 7-3). The information in the queue header provides a full description of the queue, including the token bucket parameters r and b, and this information is used by several

components in APEX (this will be described in the following sub-sections). The order of the queue headers is fixed, determined by the order of the queue templates in the `ServiceTable`-array, and it is this order that determines the order in which the queues are serviced. Thus, except for Queue 0 (described below), real-time queues are always nearest to the queue root.

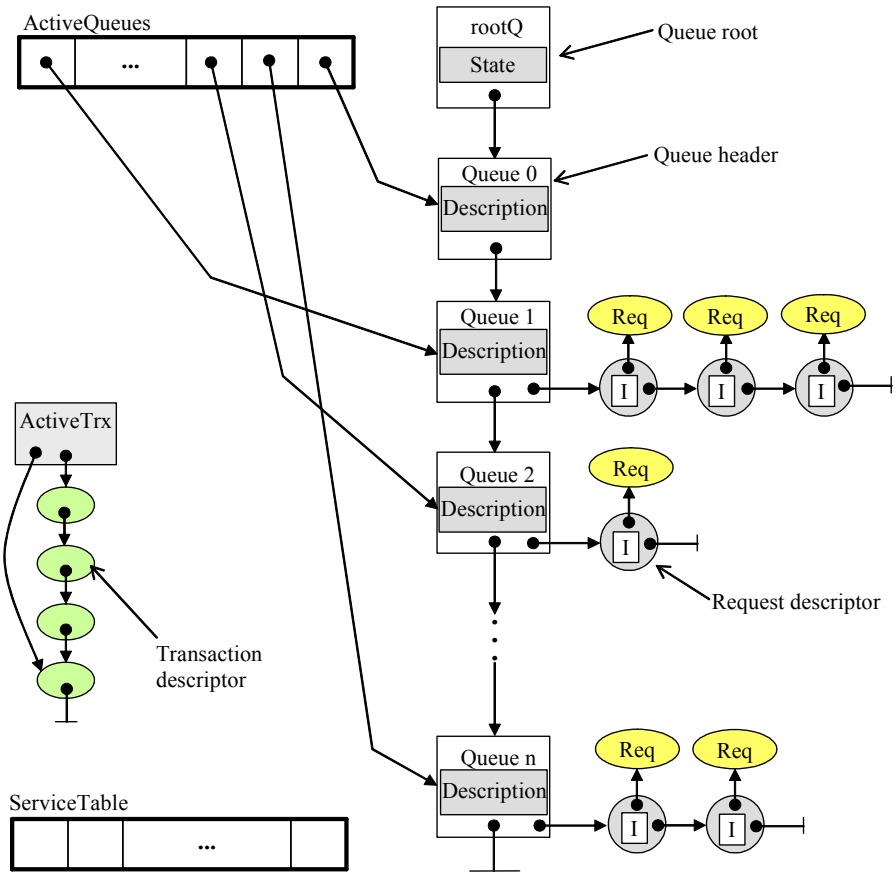

Figure 7-1: Structure of the queues in APEX

```
numQueues          //Number of active queues in APEX
ps_BW              //Bandwidth assigned to the queues using proportional-share allocation
totalW             //Sum of the weights of the proportional-share queues
t_es               //Estimated request service time, t_es
currDL             //Deadline (in absolute time) for the batch being served
nextDL             //Deadline (in absolute time) for the batch being assembled
currBSize          //Number of remaining requests in the current batch
nextBSize          //Number of requests in the new batch being assembled
bLimit             //Number of requests left when building of a new batch must start
*firstQ            //Pointer to the first queue (i.e., Q0) in the queue list
```

Figure 7-2: State information in the rootQ-structure

The first queue header, Queue 0, is instantiated at system startup time, and is always present. This queue serves as queue header for pending low-latency requests. As described in Sub-section 6.4.4, these requests are allowed to "jump the line", in the sense that they

are inserted into the batch currently being served by the disk, if there is sufficient slack time.

```
QID                 //The queue ID (i.e., the index in the ActiveQueues-array)
tokenRate           //The rate at which tokens should be assigned, i.e., the...
                    //... number of milliseconds between each token assignment
bw                  //The assigned bandwidth (pages/s). tokenRate=1/(bw/1000)
numTokens           //Number of tokens currently in the bucket
lastAssigned        //The (absolute) time when a token was last assigned
maxTokens           //Max number of tokens allowed to accumulate (bucket depth)
numRequests         //Number of requests in the queue
weight              //The weight of the queue (if proportional-share queue)
serviceType         //Describes the service level of the queue (index in ServiceTable-array)
candQ               //Can the queue provide controlling requests? (TRUE/FALSE)
*firstTrans         //Pointer to the first (oldest) transaction using the queue
*lastTrans          //Pointer to the last (newest) transaction using the queue
*firstReq           //Pointer to the first request descriptor in the queue
*nextQ              //Pointer to the next queue header
*prevQ              //Pointer to the previous queue header
```

Figure 7-3: Structure of the queue headers

The `serviceType`-attribute is an index to the `ServiceTable`-array, which contains the description of the queue (see Figure 7-4). Each element in this array is a structure with one attribute for each of the scheduler characteristics described in Section 5.2. In addition, the `reqDrop`-attribute indicates whether request dropping is used, the `share`-attribute shows whether the queue can be shared among several transactions, and the `candQ`-attribute indicates whether the queue is a candidate for providing controlling requests.

```
guaranteeLevel      //What guarantee level does the queue offer
allocParadigm       //The allocation paradigm
type                //Guarantee type (Real-time/throughput-only)
priority            //Priority? (TRUE/FALSE)
reqDrop             //Request dropping? (TRUE/FALSE)
share               //Can the queue be shared among several transactions (TRUE/FALSE)
candQ               //Can the queue provide controlling requests? (TRUE/FALSE)
```

Figure 7-4: Structure of the entries in the ServiceTable-array

As explained in Section 6.3, a queue may be shared by several transactions. Consequently, it is necessary to maintain information about each single transaction that uses a queue. This is done by maintaining a linked list of transaction descriptors for each instantiated queue (see Figure 7-5), which is pointed to by the `firstTrans` and `lastTrans`-attributes in the queue header. In addition, the transaction descriptors form a separate list, rooted in the variable `ActiveTrx`. This list is used to locate the queue header, given the transaction ID. When a transaction ends, the corresponding entry is removed from both lists, and the descriptor is destroyed. Note that in Figure 7-1, the transaction lists rooted in the queue headers are not shown.

```
TrxID               //Transaction ID
bandwidth           //The bandwidth (pages/s) requested by the transaction
weight              //Weight of the queue (for proportional-share queues)
QN                  //"Name" of the queue used by the transaction (i.e., index in the...
                    //...ServiceTable-array)
instQ               //Pointer to the instantiated queue header of the requested queue
currQ               //Pointer to the queue that the transaction is currently assigned to
nextTrxQ            //Pointer to the next transaction decriptor in the queue
nextTrxL            //Pointer to the next transaction decriptor in the ActiveTrx-list
```

Figure 7-5: Structure of the transaction descriptors

Since the disk requests must be handed over to the OS for service, the MMDBMS must format the requests according to the requirements of the OS. As explained in Sub-section 6.1.4, we assume raw disk access, which means that these requests must be formatted according to the requirements of the disk driver.

In many operating systems (e.g., BSD [59], Linux [14], and Windows [91]), a request to the disk driver is realized as a structure (or object), which contains the information needed by the disk driver to service the request (i.e., disk number, address on disk, buffer address, whether it is a READ or a WRITE operation, etc.). In the remainder of this thesis, we refer to this request structure as a *buf structure*, which is the term used in the BSD OSs [59].

The PSM submits disk requests to APEX using such buf structures, together with additional information that APEX needs to provide the correct service type. When APEX receives a disk request, the buf structure is placed in a "container", called a *request descriptor*, together with the information intended for APEX. The principle of handling the disk requests as buf structures internally in the disk scheduler is also used in Cello and MARS, but by wrapping the buf structure in a container, we are able to associate information with each request, without have to modify the buf structure itself.

```
TrxID               //The ID of the transaction that "caused" the request
deadline            //The deadline of the request (NULL if no deadline)
priority            //The priority of the request (NULL if no priority)
bufStruct           //Pointer to the buf structure received from the PSM
nextReq             //Pointer to the next request descriptor
```

Figure 7-6: Structure of the request descriptors

In Figure 7-6, we show the structure of the request descriptor. It contains three variables for the APEX-specific information (`TrxID`, `deadline`, and `priority`), a pointer to the buf structure, and a pointer to the next request descriptor in the queue. Thus, a queue of pending requests is realized as a linked list, which implies good flexibility with respect to the size of the queue. Each queue header contains a pointer to the linked list of

request descriptors for that queue, and when the batch builder picks requests for service, it always picks from the head of the queue, i.e., closest to the queue header.

Finally, there is a pointer array called `ActiveQueues`, which provides direct access to the different queue headers, using the queue ID (`QID`) as an index. The request distributor uses this array when new requests arrive, in order to quickly locate the correct queue into which the request should be inserted. When a new queue is instantiated, the first available position in the `ActiveQueues` -array is used. Thus, as Figure 7-1 indicates, there is no relationship between the order of the queue headers and the order of the pointers in the `ActiveQueues` -array.

7.2 Interfaces to APEX

As explained in Sub-section 6.2.1, APEX offers four interfaces to MMDBMS components; request submission interface, queue request interface, AdmitCommit interface, and transaction state interface. In this section, we give a short presentation of each of these interfaces, which parameters they take, as well as their return values.

Request Submission Interface

In our MMDBMS architecture, the physical storage manager (PSM) represents the traditional interface to the disk driver. Thus, without APEX, disk read/write calls would go directly from the PSM to the disk driver. With APEX as part of the MMDBMS, an extra layer between the disk driver and the PSM is effectively added. During a call, the request submission interface of APEX receives the same parameters as the disk driver would (i.e., the buf-structure), in addition to a set of APEX-specific parameters:

```
Schedule(*buf, QID, deadline, priority, TrxID)
```

When APEX receives a disk request through the `Schedule`-interface, an empty request descriptor is filled with the received data, and handed over to the queue scheduler for positioning within the queue given by the `QID`-parameter. As can be seen from Figure 7-6, the queue ID is not stored in the request descriptor. The reason is that this parameter is only needed to find the correct queue for the request. The request distributor uses the queue ID as an index in the `ActiveQueues`-array, which yields a pointer to the correct queue. Thus, with this pointer to the correct queue, the queue ID is no longer needed.

Not all the APEX-specific parameters are relevant for all types of requests. For instance, a non-real-time request does not use the `deadline`-parameter. In these cases, the unused parameters are set to `NULL`.

The request submission interface is synchronous, i.e., when the PSM uses the interface, the call returns as soon as the request has been placed in its queue. We have chosen this solution, because the process of finding the correct queue and placing the request is very simple and executes quickly. The return value is `OK` or `FAIL`, depending on whether the request could be placed in a queue or not.

Queue Request Interface

As mentioned, the queue names (i.e., names of service classes) are stored in the system catalog of the MMDBMS, and can be queried in order to find a suitable queue type (see Sub-section 6.1.3). Since the ServiceTable array contains descriptions of all supported queue types, such a queue name is nothing more than an index to the `ServiceTable` array.

When the admission control component requests a queue from APEX during the admission control phase, the queue name is included in the request. Thereby, APEX is able to determine the requested service type, and perform the necessary tentative reservations. The interface itself looks like the following:

```
RequestQ(TrxID, queueName, bandwidth, weight)
```

`TrxID` (i.e., the ID of the transaction that requests admission) and `queueName` are always included in the request. The `bandwidth` parameter states the amount of bandwidth (in pages/s) that should be reserved in the queue, on behalf of the transaction in question. Thus, this parameter is only used for queues that are based on explicit bandwidth reservation. The `weight` parameter is only used in requests for proportional-share queues, and states the weight of the queue, relative to other proportional-share queues. In other words, the `bandwidth` and `weight` parameters are mutually exclusive.

The result of the call is that a new transaction descriptor is instantiated, and the information received in the call is stored in this descriptor. The transaction descriptor is then placed in the `ActiveTrx`-list.

As described in Sub-section 3.5.1, we assume a nested transaction model. Thus, a query may include several transactions; one master-transaction, and several sub-transactions. For

instance, a multimedia playback query may have one sub-transaction controlling the playback of audio and video, while another controls the presentation of associated subtitles. Since these different elements of the query may require different service types (i.e., different queues in APEX), a separate `RequestQ`-call must be performed for each sub-transaction, and each call contains the ID of the corresponding sub-transaction.

The return value of this call is either "OK" or "FAIL", depending on whether APEX was able to provide the requested queue or not.

AdmitCommit Interface

When the global admission control for a transaction is completed, the admission control component of the LoD-system informs APEX about the result, through the AdmitCommit interface:

```
AdmitCommit(TrxId, result)
```

The call takes two parameters; the ID of the (sub-) transaction for which the admission control was made, and the result of the global admission process, i.e., either OK or FAIL. The transaction ID is necessary for APEX to be able to change the tentative reservation made after the `RequestQ`-call into an explicit one. Note that, if several `RequestQ`-calls were made, for instance for several sub-transactions constituting a playback, then a corresponding number of `AdmitCommit`-calls must also be made.

If the global admission control succeeded (i.e., the value of the `result`-parameter is "`OK`"), APEX returns a queue ID, which identifies the queue in which the requests should be placed. Later, when the query starts, and disk requests are submitted to APEX, this queue ID must be associated with every disk request that is sent to APEX through the request submission interface.

Transaction State Interface

In a dynamic environment like a MMDBMS, transactions continuously start and finish. Thus, it must be possible to notify APEX each time a transaction changes state, so the bandwidth distribution can be updated accordingly.

Consequently, we have implemented a transaction state interface that allows APEX to be notified each time a transaction starts, ends, or changes its bandwidth requirements:

```
TrxState(TrxID, state, newValue)
```

The `state` parameter can have one of the following values: START, END, ABORT, or CHANGE. For the first three values, the `newValue` parameter is set to 0, while if the `state` parameter is CHANGE, then the `newValue` parameter is set to the new reserved bandwidth (in pages/s) or new weight (depending on whether the queue in question is reservation-based or proportional share). The return value of the call is either "OK" or "FAIL".

Initially, it is not necessary to notify APEX about a transaction that makes a pause, since the work-conserving facility re-distributes the unused bandwidth. However, with the transaction state interface, it is possible to set the reserved bandwidth to zero for the transaction in question, and thereby release the reserved bandwidth. When the transaction resumes, another transaction state call can be made, to reclaim the reserved bandwidth.

7.3 Components in APEX

In Sub-section 6.2.2, we presented the five components of APEX (see also Figure 7-7). In this section, we provide a detailed description of each of these components, using pseudo-code. We first focus on the two queue management components, namely the queue manager and the bandwidth manager. Next, we describe the request management components, which are the request distributor, the queue scheduler, and the batch builder.

Notice that in order to limit the amount of code presented in the thesis, we have focused on the central functionality, and left out other parts of the code. For instance, small support functions are only described textually, and most of the error handling is left out.

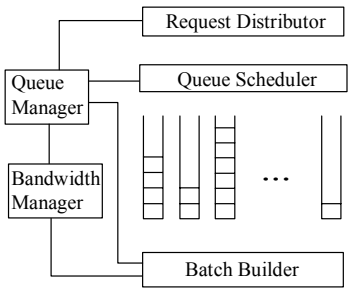

Figure 7-7: The components in APEX

7.3.1 Queue Management

The queue management is responsible for administering the queues in which the incoming disk requests are placed. New queues are instantiated as needed, and unused queues are

removed. In addition, distribution of bandwidth to the different queues is performed, based on the information received from the admission control component and the transaction manager.

The queue management components are only active when a transaction changes state, i.e., during admission control (when a queue is requested), when (if) it changes its bandwidth reservation, and when it ends. This means that these components are sleeping most of the time, and represents little processing overhead. Given the low overhead, we consider the use of dynamic queue management to be a better solution than a static solution where all queues that may be required are instantiated at system startup time.

Queue Manager

The queue manager is responsible for instantiating and deleting queues based on the needs of the active transactions (see Figure 7-8). When a queue is requested, using the `RequestQ()`-interface, the queue manager first creates a transaction descriptor structure, to store the information about the new transaction, and places this descriptor in the `ActiveTrx`-list. Thus, all necessary information about the new transaction is now stored within APEX.

```
RequestQ(Tx, Qname, bw, weight)
   TrxId Tx;
   int Qname;
   int bw;
   int weight;
{
   td = new(transactionDescr);  //Create a new transaction descriptor
   td->TrxID = Tx;              //Assign values to the attributes
   td->bandwidth = bw;
   td->weight = weight;
   td->QN = Qname;
   InsertTD(td);                //Insert the trx-descriptor in the ActiveTrx-list
}
```

Figure 7-8: Implementation of the `RequestQ`-interface of the queue manager

If the admission process failed, i.e., some component in the MMDBMS rejected the admission request, APEX receives an `AdmitCommit(TrxID, 'FAIL')`-message. The queue manager removes the transaction descriptor from the `ActiveTrx`-list, and then destroys this descriptor.

If the transaction was successfully admitted, APEX receives an `AdmitCommit(TrxID, 'OK')`-message. The queue manager must now make sure that the requested queue actually exists, creating a new queue header if necessary. This is described in Figure 7-9.

```
AdmitCommit(Tx, result)
   TrxId Tx;
   enum result;                              //Values: OK, FAIL
{
   if (result == 'FAIL') {                   //The MMDBMS admission process failed
      td=RemoveTD(tx);                       //Remove the transaction from the ActiveTrx-list...
                                             //...(returns a pointer to the transaction descriptor)
      destruct(td);                          //Destroy the transaction descriptor
   } else {                                  //The transaction was successfully admitted
      td = GetDescr(Tx);                     //Locates the trx-descr. in ActiveTrx, based on TrxId
      qh = NULL;
      if (ServiceTable [td->QN]->share)      //If queue-sharing is allowed, check ActiveQueues-
         qh = FindQH(td->QN]);               //To see if queue is instantiated (returns NULL if not)
      if (qh == NULL) {
          qh = new(queueHeader);             //A new queue header must be instantiated
          QID = GetID();                     //Locates a free entry in the ActiveQueues-array...
                                             //...and returns the index
          qh->QID = QID;
          qh->tokenRate = 0;                 //Rate is zero, since the queue is not yet active
          qh->weight = 0;                    //Weight is zero, since the queue is not yet active
          qh->serviceType = td->QN;          //Index to the ServiceTable-array
          if (ServiceTable[td->QN]->candQ)   //Candidate for providing controlling requests?
             qh->candQ='TRUE';
          InsertQ(qh);                       //Insert the queue header into the list of queues
          ActiveQueues[QID] = qh;            //Insert the queue header into the ActiveQueues-array
      }
      td->instQ = qh;
      return(QID);
   }
}
```

Figure 7-9: Implementation of the `AdmitCommit`-interface of the queue manager

When a transaction starts or finishes, APEX is informed through the `TrxState`-interface (see Figure 7-10). APEX then adds or removes the transaction from the queue in question, and possibly updates the bandwidth reservation for the queue. In addition, the queue is removed if no transactions are using it.

This interface is also used to change bandwidth reservations for transactions. If the transaction in question uses a reservation-based queue, the difference between the old and the new bandwidth reservation for the transaction is used to update the amount of bandwidth reserved for the queue. In addition, the amount of bandwidth available to the proportional-share queues is updated, using the `UpdatePS()`-function in the bandwidth manager.

For all transaction state changes, if the transaction uses a proportional-share queue, the weight for the queue in question is updated, as well as the total weight for all proportional-share queues. In addition, the bandwidth available for the proportional-share queues is re-distributed, using the `UpdatePS()`-function, based on the new weights.

```
TrxState(Tx, state, newValue)
   TrxId Tx;
   enum state;
   int newValue;
{
   td = GetDescr(Tx);                              //Get a pointer to the trx-descr. from the
                                                   //ActiveTrx-list
   if (state == 'START') {
      Move(td, td->instQ, 'ADD');                  //Add trx-descr. to the correct queue
   }
   if (state == 'CHANGE') {
      if (ServiceTable[td->QN]->allocParadigm == PS){    //If proportional-share queue
          td->currQ->weight += newValue - td->weight;    //Update weight in queue header
          rootQ.totalW += newValue - td->weight;         //Update total weight for alle PS-queues
          td->weight += newValue - td->weight;           //Update weight of trx
          UpdatePS(td, NONE, 0);                         //Update proportional-share queues
      } else {                                           //Reservation-based queue
          td->currQ->bw += newValue - td->bandwidth;     //Update queue bandwidth
          td->bandwidth += newValue - td->bandwidth;     //Update transaction bandwidth
          UpdatePS(td, UPDATE, newValue - td->bandwidth);//Update available prop.share bw
      }
   }
   if (state == 'END'|| state == 'ABORT') {
      Move(td, td->currQ, 'REMOVE');               //Remove the trx-descr from its queue
      RemoveTD(Tx);
      if (state == 'ABORT')
         RemoveReq(td);                            //Abort pending requests
      CheckQ(td);                                  //Remove the queue if no transactions
      destruct(td);
   }
}
```

Figure 7-10: Implementation of the `TrxState`-interface of the queue manager

In addition to implementing the queue management interfaces, the queue manager also contains several small support functions. These are relatively straightforward, and we therefore only describe them briefly:

- `InsertTD(*trxDescr)`: Inserts the transaction descriptor `trxDescr` into the `ActiveTrx` list. New transactions are inserted at the end of the list, using the pointer to the last transaction.
- `RemoveTD(TrxID)`: Traverses the `ActiveTrx` list to find the descriptor with transaction ID equal to `TrxID`. When the descriptor is found, it is removed from the list, and a pointer to the descriptor is used as return value of the call.
- `GetDescr(*queueHeader, TrxID)`: This function traverses the list of transaction descriptors in `queueHeader` and returns a pointer to the descriptor with transaction ID equal to `TrxID`. If `queueHeader` is NULL, the `ActiveTrx` list is traversed.
- `GetID()`: Searches the `ActiveQueues` array, and returns the index of the first unused element.
- `InsertQ(*queueHeader)`: Inserts `queueHeader` into the queue list. The order of the queues is given by the order of the elements in the `ServiceTable` array, i.e., the order of the queues is static.

- `CheckQ(*trxDescr)`: Checks if the queue header identified by the `instQ`-attribute of `trxDescr` has any transactions, by traversing the `ActiveTrx`-list. The purpose of this function is to avoid deleting a queue with transactions that have not yet started. If the queue has no transactions, the queue header is removed from the queue list and destroyed.
- `FindQH(queueName)`: Traverses the list of queue headers to check if a queue with the "name" (i.e., index in the `ServiceTable`-array) `queueName` is instantiated. If so, it returns a pointer to the queue header, otherwise it returns NULL.

Bandwidth Manager

While the queue manager is responsible for creating and deleting queues, the bandwidth manager distributes the available bandwidth among the instantiated queues. The `Move()`-function called by the queue manager is shown in Figure 7-11. This function relies on the `AddTrx()` and `RemoveTrx()`-functions described below to do the actual work of updating the queue headers.

```
Move(*td, *Q, op)
   trxDescr *td;
   queueHeader *Q;
   enum op;                                    //'ADD' or 'REMOVE'
{
   ap = ServiceTable[td->QN]->allocParadigm;  //Reservation-based or proportional-share
   if (op == 'ADD') {                          //A new transaction is added
      AddTrx(td, Q);                           //Add the transaction to queue Q
      if (ap == 'RES')                         //If reservation-based queue...
          UpdatePS(td, 'REVOKE', 0);           //...take bandwidth from proportional-share queues
      else if (ap == 'PS')                     //If proportional-share queue...
          UpdatePS(td, NONE, 0);               //... update weights for proportional-share queues
   } else if (op == 'REMOVE') {                //A transaction has ended
      RemoveTrx(td, Q);                        //Remove the transaction from queue Q
      if (ap == 'RES')                         //If reservation-based queue...
          UpdatePS(td, 'ADD', 0);              //...give bandwidth to proportional-share queues
      else if (ap == 'PS')                     //If proportional-share queue...
          UpdatePS(td, NONE, 0);               //... update weights for proportional-share queues
   }
}
```

Figure 7-11: Algorithm for the `Move()`-function in the bandwidth manager

The `Move()`-function distinguishes transactions using reservation-based queues from queues using proportional-share allocation. We do this because of the way bandwidth is distributed between the different queue types in APEX: As long as there are no queues using reservation-based allocation, all available bandwidth is reserved for the proportional-share queues, while best-effort queues only receive unused bandwidth. When a reservation-based queue is requested (in the `RequestQ()`-call), the required bandwidth is taken from the proportional-share queues using the `UpdatePS()`-function. Correspondingly, bandwidth is given back to the proportional-share queues when a

transaction using reservation-based allocation ends. When the available bandwidth for the proportional-share queues changes, all these queues are affected, according to their weights. This technique for supporting both reservation-based and proportional-share allocation is based on the work in [92].

```
AddTrx(*td, *Q)
   trxDescr *td;
   queueHeader *Q;
{
   ap = ServiceTable[td->QN]->allocParadigm;      //Get description of the queue
   td->currQ = Q;
   AddQHTD(td, Q);                                //Add td to Q's transaction-list
   if (ap == 'RES')                               //Reservation-based allocation
      UpdateBW(td, Q, ADD);                       //Add bandwidth & update tokenRate
   else if (ap =='PS') {                          //If proportional-share...
      Q->weight += td->weight;                    //...adjust the weight of the queue
      rooQ.totalW += td->weight;                  //Update total weight
   }
}
```

Figure 7-12: Algorithm for the `AddTrx()`-function in the bandwidth manager

When a transaction descriptor is added to a queue using the `AddTrx()`-function, the bandwidth allocation paradigm of that queue determines the course of the operation. For instance, we assume that transaction descriptor T_1 is added to queue Q_a. In all cases, T_1 is inserted into the list of descriptors pointed to by the `firstTrans/lastTrans`-attributes of the queue header of Q_a, using the `AddQHTD()`-call.

If T_1 has specified a queue with reservation-based allocation, the `tokenRate` and `bw`-attributes of the queue header of Q_a are updated, using the `UpdateBW()`-call. If, instead, T_1 has specified a proportional-share queue, the weight of the queue, as well as the total weight of all proportional-share queues is updated.

The `RemoveTrx()`-function, shown in Figure 7-13, is similar to the `AddTrx()`-function, but has the opposite effect.

```
RemoveTrx(*td, *Q)
   trxDescr *td;
   queueHeader *Q;
{
   ap = ServiceTable[td->QN]->allocParadigm;
   RemoveQHTD(td, Q);                             //Remove td from Q's transaction-list
   if (ap == 'RES')                               //If reservation-based allocation...
      UpdateBW(td, Q, REMOVE);                    //...remove bandwidth & update tokenRate
   else if (ap =='PS')  {                         //If proportional-share...
      Q->weight -= td->weight;                    //...adjust the weight
      rootQ.totalW -= td->weight;                 //Update total weight
   }
}
```

Figure 7-13: Algorithm for the `RemoveTrx()`-function in the bandwidth manager

In addition to the three main functions described above, the bandwidth manager provides four support functions: `UpdatePS()`, `UpdateBW()`, `AddQHTD()`, and `RemoveQHTD()`. The `UpdatePS()`-function is used for adjusting the bandwidth

allocations for the queues using proportional-share allocation. These queues share a certain bandwidth, and each queue receives a portion of this bandwidth according to its weight.

The bandwidth available for proportional-share queues changes each time a transaction using a reservation-based queue enters or leaves the system, or has its bandwidth changed. Thus, each time one of these events occur, bandwidth must be redistributed, and this is done by the `UpdatePS()`-function, described in Figure 7-14.

```
UpdatePS(*td, op, delta)
   trxDescr *td;
   enum op;                                        //ADD, REVOKE, UPDATE, or NONE
   int delta;                                      //If UPDATE, this is the difference in bandwidth
{
   if (op ==‘ADD’)                                 //Increase the amount of bandwidth at ...
      rootQ.ps_BW += td->bandwidth;                //... the disposal of the prop.share queues
   else if (op == ‘REVOKE’)                        //Decrease the amount of bandwidth at ...
      rootQ.ps_BW -= td->bandwidth;                //... the disposal of the prop.share queues
   else if (op == ’UPDATE’)                        //Update available PS-bandwidth after updating..
      rootQ.ps_BW += delta;                        //... a reservation-based queue
   qh = rootQ.firstQ;
   while (qh != NULL) {
      if (ServiceTable[qh->ServiceType]->allocParadigm == ‘PS’) {  //If prop.share queue...
         qh->bw = (qh->weight / rootQ. totalW) * rootQ.ps_BW;      //...compute bandwidth share
         updateBW(td, qh, NONE)
      }
      qh = qh->nextQ;
   }
}
```

Figure 7-14: Algorithm for the `UpdatePS()`-function in the bandwidth manager

The `UpdateBW()`-function (Figure 7-15) does the actual updating of bandwidth and token rate, and is called for each instantiated queue. The `op`-parameter determines whether the bandwidth of the included transaction descriptor should be added to or subtracted from the bandwidth reserved for the queue. If this parameter is set to “NONE”, the function only computes the token rate, without changing the bandwidth first, and this is used for updating the proportional-share queues each time bandwidth allocations are changed for a reservation-based queue.

```
UpdateBW(*td, *Q, op)
   trxDescr *td;
   queueHeader *Q;
   enum op;                              //ADD, REMOVE, or NONE
{
   if (op ==‘ADD’)                       //Transaction added to the queue
      Q->bw += td->bandwidth;
   else if (op == ‘REMOVE’)              //Transaction removed from the queue
      Q->bw -= td->bandwidth;
   Q->tokenRate = 1/(Q->bw / 1000);      //Compute token rate from bandwidth
}
```

Figure 7-15: Algorithm for the `UpdateBW()`-function in the bandwidth manager

The two remaining support functions are very simple, and we have chosen not to show them in pseudo-code:

- `AddQHTD(*trxDescr, *queueHeader)`: Inserts the transaction descriptor `trxDescr` into the linked list of transaction descriptors in `queueHeader`. New transaction descriptors are inserted at the end of the list, using the `lastTrans`-attribute in the queue header.
- `RemoveQHTD(*trxDescr, *queueHeader)`: Removes the transaction descriptor `trxDescr` from the linked list of transaction descriptors in `queueHeader`. It starts at the head of the list (pointed to by `firstTrans`), and works its way through the list.

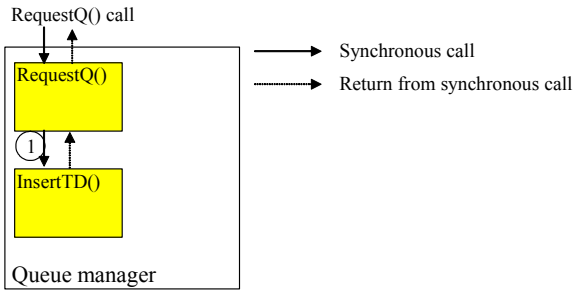

Figure 7-16: Functions involved in an `RequestQ()`-call

In Figure 7-16, we show the functions that participate in an `RequestQ()`-call, while Figure 7-17 shows the functions used in an `AdmitCommit()`-call. As can be seen from the figures, all calls are synchronous.

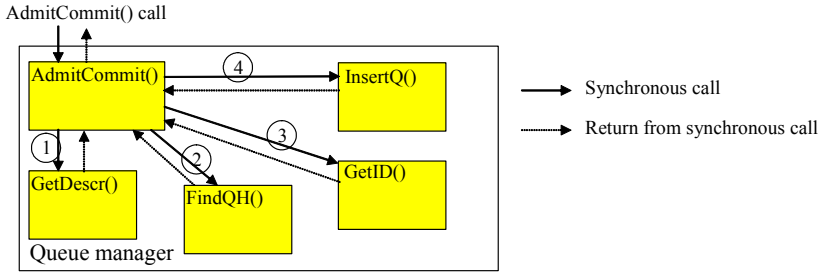

Figure 7-17: Functions involved in an `AdmitCommit()`-call (assuming transaction was admitted)

Figure 7-18 shows the functions involved in a `TrxState(TrxID, START, 0)`-call, i.e., a notification to APEX that a transaction has changed state. Again, all calls are synchronous.

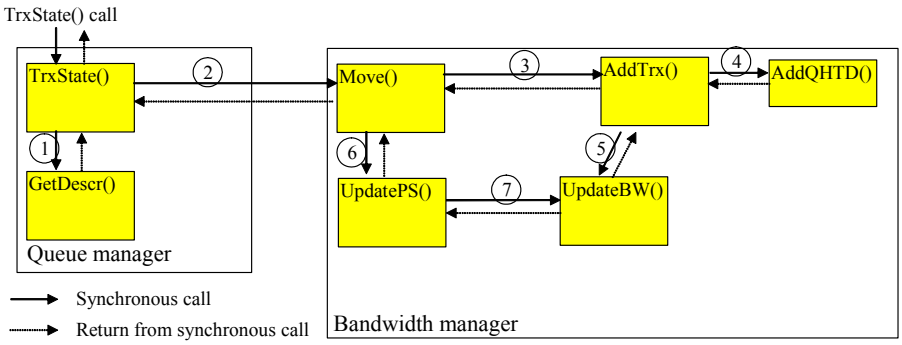

Figure 7-18: Functions involved in a `TrxState()`-call (assuming the state was changed to START for a reservation-based queue)

7.3.2 Request Management

The request management is responsible for all handling of disk requests. Thus, while the queue management controls the service type that each request should receive, the request management is responsible for realizing these service types.

All information that the request management components need for realizing the service types is available in the `rootQ`-structure, as well as the queue headers. Hence, these data structures effectively constitute the interface between the two parts of APEX.

Request Distributor

The request distributor implements the `Schedule()`-interface, where all disk requests from the MMDBMS arrives (see Figure 7-19). When the request distributor receives a request from the PSM, it extracts the queue ID, and uses this parameter to locate the correct queue (using the `ActiveQueues`-array) in which to put the request. More precisely, the queue ID is used as an index in this array, and the result is a pointer to the correct queue header. If the pointer does not exist (i.e., the requested queue does not exist), the request distributor returns the call from the PSM with an error message.

Next, the request distributor instantiates a request descriptor, fills in the attributes, and adds the request (i.e., the buf-structure) to the descriptor. Note that, in practice, we use a pool of empty request descriptors, since this is a more efficient solution than instantiating and destroying descriptors for each single request.

```
Schedule(*bp, Q, dl, pri, Tx)
   buf *bp;
   QID Q;
   deadline dl;
   priority pri;
   TrxID Tx;
{
   qh = ActiveQueues[Q];                //Find pointer to the correct queue header
   if (qh == NULL) return(FAIL);        //The requested queue does not exist
   rd = new(requestDescriptor);         //Instantiate a new request descriptor
   rd->TrxID = Tx;
   rd->deadline = dl;
   rd->priority = pri;
   rd->bufStruct = bp;
   PlaceRequest(rd, qh);                //Call the queue scheduler to insert the request
   return(OK);
}
```

Figure 7-19: Algorithm for the `Schedule()`-call

Finally, the request distributor calls the queue scheduler, with the request descriptor and a pointer to the queue header as parameters. If the request is successfully inserted into the queue, the queue scheduler returns the call from the request distributor, which, in turn, returns the call from the PSM with a positive acknowledgement.

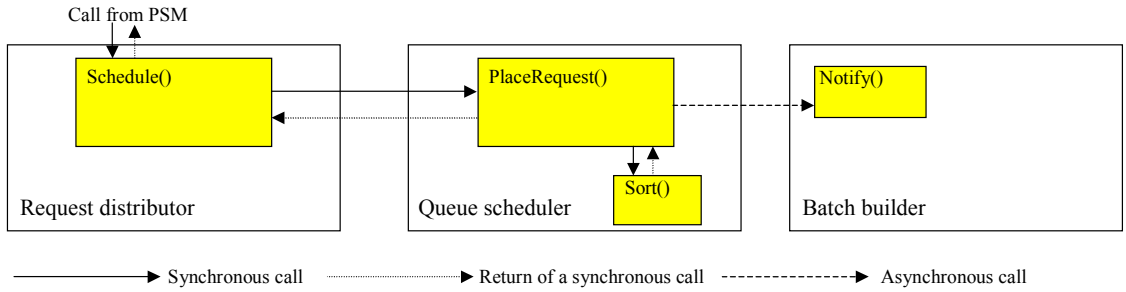

Figure 7-20: Functions involved in a normal `Schedule()`-call

In Figure 7-20, we show graphically how a `schedule()`-call from the PSM is handled by APEX. Note that, if a low-latency request is pending in Queue 0 or the batch builder is idle when the `Notify()`-function is called, the batch builder performs several additional actions (this is not shown in the figure).

Queue Scheduler

Given a request descriptor and a pointer to a queue header, the queue scheduler is responsible for placing the descriptor in the correct position in the queue held by the queue header. In order to do this, the queue scheduler uses the information in the queue header to determine where to place the request; for instance, if the queue is a real-time queue, the queue scheduler orders the requests according to their deadline.

```
PlaceRequest(r, qh)
   requestDescr r;
   queueHeader qh;
{
   qh->numRequests++;
   QC = ServiceTable[qh->serviceType];            //Determine the queue characteristics
   If (QC->Type == 'RT' && QC->priority == 'N')
          sort(qh, r, DEADLINE, POS);             //Sort on deadline first, then on position
                                          .
                                          .
                                          .
   If (QC-level == 'BE' && QC->priority == 'Y')
          sort(qh, r, PRIORITY, POS);             //Sort on priority first, then on position
   <call Notify-function in the batch builder>;   //This is to avoid deadlocks
}
```

Figure 7-21: Algorithm for the `PlaceRequest()`-function in the queue scheduler

Figure 7-21 shows the main principle of the algorithm. When a new request descriptor arrives from the request distributor, the queue scheduler first extracts the characteristics of the queue, using the `serviceType`-attribute of the queue header as index in the `ServiceTable`-array.

Based on the information from the `ServiceTable` element, the queue scheduler is able to add the new descriptor to the queue at the correct position. As can be seen from Figure 7-21, more than one sorting criteria can be used. For instance, real-time requests with a common deadline can be sorted based on the disk position of the requested data. We chose this approach, since it provides extensibility with respect to sorting criteria; if a new sorting method is required, it can easily be added to the scheduler.

In addition, the queue scheduler has two support functions:

- `RemoveReq(*trxDescr)`: The queue manager uses this function to remove pending requests if a transaction aborts; the function traverses the list of requests pointed to by the queue header indicated in the `instQ`-attribute of the transaction descriptor.
- `Sort(*queueHeader, *reqDescr, sortCrit1, sortCrit2)`: Inserts the request descriptor `reqDescr` into the request list of `queueHeader`, using `sortCrit1` and possibly `sortCrit2`.

Batch Builder

The batch builder is the component that performs the final scheduling of the pending disk requests. It picks requests from the queues in order to build request batches, and while doing so, it relies on the information in the queue headers. The batch builder is also responsible for submitting the request batches to the disk driver, and for receiving and

forwarding notifications from the disk driver, each time the disk is finished serving a disk request.

The core of the batch builder is the `Notify()`-function (see Figure 7-22), which is called by the disk driver each time a request finishes, and causes the batch builder to wake up. This function updates the number of outstanding requests, and when the number of requests left in the current batch goes below a given threshold, which is found in the variable `bLimit` in the `rootQ` structure (see Figure 7-2), the assembly of a new batch is started.

A potential problem of this approach is that, in a case where the disk is idle and requests suddenly starts to arrive, the batch builder would never be activated, since no notifications are received from the disk driver (which is idle). We avoid this by letting the queue scheduler call the `Notify()`-function with a NULL pointer each time a request is inserted in a queue, as shown in Figure 7-21.

```
Notify(*bp)
   buf *bp;
{
   if (bp==NULL) {                                  //The call comes from the queue scheduler
      if (Q0->numRequests>0) PostSubmit(Q0);        //Are there pending low-latency requests?
      if (rootQ.currBSize)==0) build(currTime);     //Run build() if the batch builder is idle
   } else {                                         //The call comes from the disk driver
      <notify PSM>;                                 //Notify PSM that a request is finished
      rootQ.currBSize--;                            //One request less in the batch
      if (Q0->numRequests>0) PostSubmit(Q0);        //Are there pending low-latency requests?
      if (rootQ.currBSize = rootQ.bLimit)           //Time to start building a new batch?
         batch = build(currTime + rootQ.currBSize * rootQ.servWC); //Estimate finishing time...
      else if (rootQ.currBSize == 0) {                             //...of current batch, t_ef
          rootQ.currDL = rootQ.nextDL;              //Next batch now becomes the active batch
          rootQ.currBSize = rootQ.nextBSize;
          submit(batch);                            //Submit new batch to the disk driver
      }
   }
   sleep();                                         //Go back to sleep
}
```

Figure 7-22: Algorithm for the `Notify()`-function in the batch builder

Before the assembly of a batch can start, the controlling request must be determined, as described in Sub-section 6.4.2. This is done by the function shown in Figure 7-23, which traverses the list of queue headers, and checks all queues that are candidates for providing the controlling request, i.e., the candidate queues.

```
FindCReq(*qh)
   queueHeader *qh;
{
   minDL = current_Time + <system default latency>;   //Start with a system-default deadline
   while (qh != NULL && qh->candQ) {                  //Only check candidate queues
      if (qh->firstReq != NULL && minDL > qh->firstReq->deadline)
          if (qh->firstReq->deadline >= (currTime + <system minimum deadline>))
              minDL = qh->firstReq->deadline;         //Update deadline
      qh = qh->nextQ;
   }
   return(minDL);
}
```

Figure 7-23: Algorithm for the `FindCReq()`-function in the batch builder

The function starts with the artificial system deadline, and then visits all candidate queues to check if there are requests with shorter deadlines. If such a request exists, and its deadline is not smaller than the system minimum deadline, the request is used as controlling request. The rationale behind the system minimum deadline is to take into account that the controlling request can be provided by a queue with statistical real-time guarantee. In such a queue, there is a chance that requests miss their deadlines, and if such a request is chosen as controlling request, then the batch building principle would break down, since a too short deadline would not allow additional requests to be added to the batch.

In a system with one global round time and where all real-time requests have deadlines equal to the end of the round, the `FindCReq()`-function is, strictly speaking, not necessary. However, having this function allows both real-time requests with deadlines that are multiples of the round, and several round times in the same system (these must be multiples of the smallest round time). In addition, the overhead of the function is small, since there are a limited number of candidate queues, and only the first request in each queue is checked.

```
Build(startTime)
   long startTime;                                  //The latest time at which the batch...
{                                                   //...will be submitted
   rootQ.nextBSize = 0;                             //Reset batch size
   endTime = FindCReq(Q0->nextQ);                   //Find the controlling request
   minTime = endTime-startTime;                     //Available service time for the batch
   numReq=floor(minTime/rootQ.t_es);                //Compute the maximum batch size
   if (numReq > MaxBatchSize) numReq = MaxBatchSize; //Limit maximum batch size
   currQ = Q0->nextQ;                               //First queue to visit
   while (numReq > 0 && currQ != NULL) {            //While available time and queues...
      if (currQ->numRequests>0){                    //If pending requests...
          tokenUpdate(currQ);                       //...update tokens
          if (currQ->numTokens>0) {                 //Visit queue if available tokens
             numReq=DeQueue(currQ, &first, &last, numReq, TRUE);  //Pick requests, reduce tokens
             if (batch == NULL) batch = first;      //Add requests to...
             else lastReq->nextReq = first;         //...the end of the batch
             lastReq = last;
          }
      }
      currQ = currQ->nextQ;
   }
   currQ = Q0->nextQ;                               //Start work-conserving phase
   while (numReq>0 && currQ!=NULL) {
      <same as above, but without checking for tokens,>
      <and the last parameter in the DeQueue-call is set to FALSE>
   }
   rootQ.nextDL = startTime + rootQ.nextBSize*rootQ.t_es; //Estimated finishing time
   return(&batch);
}
```

Figure 7-24: Algorithm for the `Build()`-function in the batch builder

While the `Notify()`-function controls the batch builder, the `Build()`-function has the overall responsibility for assembling the request batches. Given the latest time at which the serving of the batch will start, and the deadline of the controlling request, the

`Build()`-function selects as many requests as possible from the request queues. This is shown in Figure 7-24.

The assembly of a batch is performed in two phases. In the first phase, the ordinary selection of requests takes place, where the availability of tokens is taken into consideration. This means that only requests from reservation-based queues and proportional-share queues are selected during this phase, since best-effort queues are not assigned tokens. It is during this phase that we make sure that all queues with bandwidth-reservations actually get the bandwidth they are entitled to.

Next, if there is room for more requests in the batch, the work-conserving phase starts. Bandwidth that has not been reserved, as well as unused reserved bandwidth, is now distributed among the queues. The algorithm for this phase is dependent on the chosen work-conservation policy, as described in Sub-section 6.4.3). In Figure 7-24, we show the algorithm for a very simple policy; if there is still capacity left after all queues have been visited, each queue with pending requests is visited once more, and more requests selected.

Generally, tokens are not taken into consideration during the work-conserving phase. This approach is a consequence of the purpose of the extended token bucket principle:

- It is used to ensure that all reservation-based and proportional-share queues receive at least their share of the disk bandwidth.
- It prevents queues from starving other queues, by limiting the amount of bandwidth that a queue is allowed to use.

Once these requirements have been fulfilled, the tokens are no longer needed. Instead, the available time in the batch becomes the only criterion. Thus, the `numReq` parameter in the `DeQueue()`-call becomes the tool to realize the work-conserving policy. When there is no more capacity left for the batch (i.e., the `numReq`-variable reaches zero), or all queues have been visited once more (depending on what comes first), the work-conserving phase ends.

```
TokenUpdate(*qh)
   queueHeader *qh;
{
   elapsed = currTime - qh->lastAssigned;                    //How much time since last update
   tokens = floor(elapsed/qh->tokenRate);                    //How many tokens to add
   qh->numTokens += tokens;
   if (qh->numTokens > qh->maxTokens)                        //Bucket depth exceeded?
      qh->numTokens = qh->maxTokens;
   if (tokens > 0) qh->lastAssigned += tokens*qh->tokenRate; //Update time for last update
}
```

Figure 7-25: Algorithm for the `TokenUpdate()`-function in the batch builder

Before requests can be selected from a queue during the normal phase, a token update operation must be performed on the queue, in order to establish if, and how many, requests that can be selected. This is shown in Figure 7-25. Time is measured since the last token update, and tokens are added according to the token rate.

The `DeQueue()`-function is responsible for moving requests from a queue and into the batch being assembled (the batch is a linked list of request descriptors). The algorithm is shown in Figure 7-26. The principle of this algorithm is to return a linked list of requests, where the variables `firstR` and `lastR` points to the start and end of the list, respectively.

```
DeQueue(*qh, **firstR, **lastR, max, token)
   queueHeader *qh;
   requestDescriptor *firstR, *lastR;                    //Will point to first and last request
   int max;                                              //Max # of requests to pick
   boolean token;                                        //Indicates whether tokens are used
{
   *firstR = req = qh->firstReq;
   if (!token) qh->numTokens++;                          //For use in the work-conserving phase
   While (max > 0 && qh->numTokens > 0 && req != NULL) {  //Select requests
      if (token) qh->numTokens--;                        //Remove token if not work-cons. phase
      qh->numRequests--;                                 //Update # of pending requests
      rootQ.nextBSize++;                                 //Update batch size
      max--;                                             //Reduce "space" left in batch
      *lastR = req;
      req = req->nextReq;
   }
   qh->firstReq = req;                                   //De-queue the request(s)
   req = *lastR;
   req->nextReq = NULL;                                  //Detach the last selected request
   if (!token) qh->numTokens--;                          //Remove the extra token (for use...
   return(max);                                          //...in the work-conserving phase)
}
```

Figure 7-26: Algorithm for the `DeQueue()`-function in the batch builder

If tokens are not used (i.e., we are in the work-conserving phase), we add a token before the request picking start. This is just to make sure that there is at least one token available (otherwise the while-loop would not start). The extra token is removed before the function ends.

When the disk is serving the last request of a batch, APEX submits a new batch to the disk driver. This is done by the `Submit()`-function in the batch builder (see Figure 7-27), which takes a pointer to a list of request descriptors as input, extracts the requests from the request descriptors, and submits the requests to the disk driver. As mentioned earlier, we assume that the calls to the disk driver are non-blocking. The empty request descriptors are either destroyed, or returned to a pool of empty descriptors.

Note that the code shown in Figure 7-27 is based on using a "single-request"-call to the disk driver. If calls for submitting multiple requests at once are available, as described in Sub-section 6.1.4, the while-loop in the function can be dropped.

```
Submit(*batch)
   requestDescriptor *batch;
{
   while (batch != NULL) {
      Make_Request(batch->bufStruct);            //Call the disk driver (OS-dependent)
      rd = batch;
      batch = batch->nextReq;                    //Go to next request descriptor
      <release rd>;                              //Remove the request descriptor
   }
}
```

Figure 7-27: Algorithm for the `Submit()`-function in the batch builder

Finally, the `PostSubmit()`-function is responsible for handling low-latency requests. If there are pending requests of this type, the function is called each time the `Notify()`-function is called, be it by the queue scheduler or the disk driver.

```
PostSubmit(*qh)
   queueheader *qh;
{
   reqsLeft = floor((rootQ.currDL - currTime)/rootQ.t_es);  //How many requests can be...
                                                            //...served in the remaining time
   availReq = reqsLeft - rootQ.currBSize;                   //How many more can be inserted
   while (qh->firstReq != NULL && availReq > 0) {
      Make_Request(qh->firstReq->bufStruct);               //Call the disk driver (OS-dependent)
      rootQ.currBSize++;                                   //Update batch size
      rd = qh->firstReq;
      qh->firstReq = qh->firstReq->nextReq;                //Take the request out of the queue
      <release rd>;                                        //Remove the empty request descriptor
      qh->numRequests--;
   }
}
```

Figure 7-28: Algorithm for the `PostSubmit()`-function in the batch builder

In Figure 7-29, we show the course of a `Notify()`-call from the disk driver to APEX. For this particular example, we assume that the call makes the size of the current batch go below the threshold value (`rootQ.bLimit`), such that a call to the `Build()`-function is triggered.

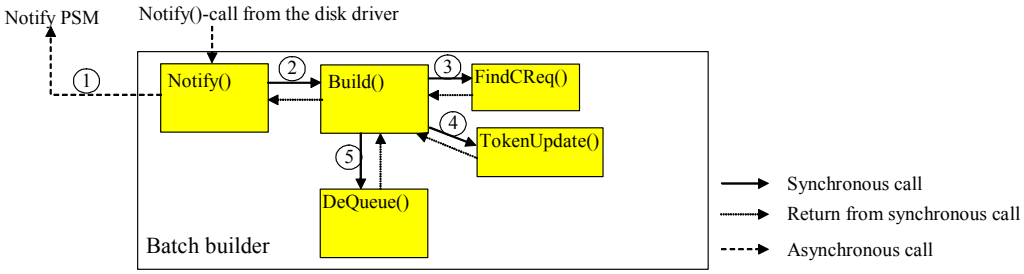

Figure 7-29: Functions involved in a `Notify()`-call from the disk driver, assuming that the call triggers a `Build()`-call

7.4 Complexity

Since the configuration of APEX shown in this thesis could appear as relatively complex, it is important to investigate the cost of using the scheduler. In this section, we examine the

computational complexity (i.e., the running time) of APEX, and for this, we use "big oh" notation [1].

As mentioned earlier, APEX consists of two functional parts: request management and queue management. The request management is the more critical part, since it is involved in every disk request submitted to the storage subsystem. The queue management is only active each time a queue is requested, or a transaction changes state (i.e., it starts, ends, or changes bandwidth requirement).

In the following sub-sections, we use a number of different parameters to analyze the running time of the different functions. We provide an overview of these parameters in Table 7-1, and in this table, we have tried to indicate a probable range for the values that each parameter may take. Note, however, that these ranges are by no means meant to be exact; they only indicate the magnitude of the values.

Table 7-1: Overview of the parameters used in the complexity analyses

Parameter	Description	#
E	Number of elements in the `ActiveQueues`-array (i.e., number of instantiated queues)	100
P	Average number of requests selected from each queue	1 – 20
Q	Number of queues	5 – 20
R_q	Number of requests in queue q	1 – 50
T_q	Number of transactions in queue q	1 – 5
T	Number of transactions in the system (i.e., the `ActiveTrx`-list)	1- 200

7.4.1 Request Management

When a disk request arrives, the location of the correct queue is done in a single step, given the queue ID associated with the disk request (Figure 7-19). Next, the request must be placed in the correct position within the queue. How much of the request list that must be traversed depends on the type of queue (i.e., how the pending requests are ordered in the queue), but in the worst case all requests in the queue in question must be traversed, until the correct position is found (Figure 7-21). Consequently, the worst-case running time is $O(R_q)$, where R_q is the number of pending requests in the queue, for each request that is submitted to the storage subsystem.

The other part of the request management, assembly of a batch, first requires finding the controlling request (Figure 7-23). This operation has a worst-case running time of

$O(Q)$, where Q is the number of queues. Note that this is the number of *candidate* queues, which is normally lower than the number of instantiated queues. During the batch assembly, all queues are checked, and in the worst case, each of these queues is visited twice: once during the normal batch assembly phase, and once during the work-conservation phase (Figure 7-24). Thus, the batch assembly operation has a worst-case running time of $O(Q+2QP)$, where P is the average number of requests picked per queue.

Finally, if we assume that each disk request is submitted separately to the disk driver, the batch submission operation has a running time of $O(QP)$, i.e., corresponding to the batch size. The total complexity of the batch assembly then becomes $O(3QP+Q)$. If we use the `lio_listio()`-call described in Sub-section 6.1.4, this is reduced to $O(2QP+Q)$, since all requests in the batch can be submitted with one call. However, in both cases, if we simplify according to the rules of the "big oh" notation, the resulting running time becomes $O(QP)$.

If we compare this to the schedulers in the reference group, we find that the first part, the placement of requests in the queues, is identical to APEX. A new request must be placed in the correct position within its queue, and this requires exactly the same operations as in APEX.

For the second part, Cello [86] visits all queues and moves as many requests as possible to the scheduled queue, i.e., a running time of $O(QP)$. If there is unused bandwidth, queues with pending requests are re-visited, and more requests picked, adding $O(QP)$. In addition, each request is submitted individually to the disk driver. Thus, the total running time for this part becomes $O(3QP)$.

Cello does not have the concept of a controlling request, therefore the $O(Q)$ time needed for locating this request is saved. On the other hand, the slack stealing technique used for inserting low-latency requests requires scanning the scheduled queue each time such a request is inserted, i.e., an $O(S)$-operation for every low-latency request, where S is the number of requests already in the scheduled queue. APEX, on the other hand, only relates to the *number* of outstanding requests in the current batch (Figure 7-28), i.e., an $O(1)$-operation. The combined running time of Cello becomes $O(QP)$, i.e., equal to APEX.

In MARS [13], each queue is visited only once in each round (i.e., there is no additional work-conserving phase), and requests are picked according to the allocated service quantum. Thus, the running time for moving requests from the queues to the

working queue (corresponding to the scheduled queue in Cello) is $O(QP)$, which is also the overall complexity.

The Prism scheduler [103], first selects requests from the "periodic" queue and the "aperiodic minimum-throughput" queue, taking $O(2P)$, and sort these in SCAN order. The sorting algorithm is not specified, but if we assume a fast algorithm like Quicksort, this takes $O(2P\log(2P))$. The scheduler then continues by selecting requests from the other two queues ("interactive" and "aperiodic best-effort"), and for each request selected, the already selected requests must be scanned to find an insertion point for the new request. This takes $O(2P2P)$, which is reduced to $O(P^2)$. Thus, the overall complexity becomes $O(P^2+2P\log(2P))$, which is reduced to $O(P^2)$.

From this short analysis, we conclude that the request insertion phase is equal for all schedulers (both APEX and the reference group). As for selecting requests from the queues and create a set of scheduled requests (a batch), APEX, Cello, and MARS performs equally ($O(QP)$). Given the value range of P and Q (see Table 7-1), the Prism scheduler will, in practice, also perform similarly to APEX.

Thus, our conclusion is that APEX does not have a higher degree of computational complexity than the schedulers in the reference group for this task (see Table 7-2).

Table 7-2: Comparison of complexity for APEX and the reference group

	Submit one request to storage subsystem	Select requests for a round (batch)
Cello	$O(R_q)$	$O(QP)$
MARS	$O(R_q)$	$O(QP)$
Prism	$O(R_q)$	$O(P^2)$
APEX	$O(R_q)$	$O(QP)$

7.4.2 Queue Management

The queue management functionality is unique to APEX. None of the reference schedulers describes similar functionality, and a comparison is therefore not meaningful. Still, it is important to investigate the complexity of this functionality, in order to get an idea of the overhead introduced.

Queue Request Interface

The `RequestQ()`-function is $O(1)$, as are `InsertTD()`, `Move()`, `AddTrx()`, `AddQHTD()`, and `UpdateBW()`. The `UpdatePS()`-function (Figure 7-14) is $O(Q)$, where Q is the number of instantiated queues. In total, the complexity of the queue request interface is therefore $O(Q)$.

AdmitCommit Interface

The `AdmitCommit()`-function itself (Figure 7-9) is $O(1)$. The `GetDescr()`-function must search the list of transactions, which is $O(T)$ in the worst case. Next, the `FindQH()`-function traverses the list of instantiated queues, i.e., a complexity of $O(Q)$.

If a new queue must be instantiated, the `GetID()`-function must traverse the `ActiveQueues`-array, an operation that is $O(E)$ in the worst case, where E is the number of elements in the array. However, since E is a fixed number, this is reduced to $O(1)$. Then the new queue descriptor must be inserted into the list of instantiated queues, which is done by the `InsertQ()`-function. This operation has a complexity of $O(Q)$. Consequently, the worst-case complexity of the `AdmitCommit`-interface is $O(T+2Q)$, which is reduced to either $O(T)$ or $O(Q)$, depending on which is the larger.

Transaction State Interface

Independent of what kind of state change takes place, the `TrxState()`- and `Move()`-functions (Figure 7-10 and Figure 7-11) have a complexity of $O(1)$. This also applies to `RemoveTrx()`, `UpdateBW()`, and `AddQHTD()`, while `GetDescr()` has a complexity of $O(T)$.

If the state change is 'START', the functions influencing the complexity are `GetDescr()` and `UpdatePS()`, the latter having a complexity of $O(Q)$. Thus, the complexity is $O(T+Q)$, which we reduce to either $O(T)$ or $O(Q)$, depending on which is the larger.

If the state change is to 'END' or 'ABORT' additional functions are executed, which have the following complexities: `RemoveTD()` is $O(T)$, `RemoveReq()` is $O(R_q)$, `RemoveQHTD()` is $O(T_q)$, and `CheckQ()` is $O(T+Q)$. Thus, the total complexity becomes $O(T_q+3T+2Q+R_q)$, which we reduce to one of the following values, depending on which is the largest: $O(T)$, $O(Q)$, or $O(R_q)$.

Finally, if the transaction state call is for a change in allocated bandwidth/weight, the involved functions are `TrxState()`, which is *O(1)*, and `UpdatePS()`, which is $O(Q)$. Thus, the total complexity becomes $O(Q)$.

Table 7-3: Complexities for the queue management functions in APEX

Queue request	$O(Q)$
AdmitCommit	$O(T) \vee O(Q)$
Transaction state (START)	$O(T) \vee O(Q)$
Transaction state (END/ABORT)	$O(T) \vee O(Q) \vee O(R_q)$
Transaction state (CHANGE)	$O(Q)$

In Table 7-3, we show an overview of the complexities for the queue management functions in APEX. From this table, it is clear that these functions introduce a relatively modest overhead, since all functions have linear growth rate for their running time, and since the magnitude of the variables (Q, T, and R_q) is relatively limited, as shown in Table 7-1. In addition, the queue management functions are only used when transactions change state, or new transactions are (tentatively) added. Thus, the invocation frequencies for the functions are much lower than for the request management functions.

7.5 Multi-Threading

Since APEX interacts both with MMDBMS-components and with the disk driver, we have chosen to split APEX into three separate threads (see Figure 7-30). Thread 1 handles disk requests received from the PSM, and insertion of these into queues. Thread 2 handles all remaining communication with MMDBMS-components, except notifying the PSM about completed requests, which is done by the batch builder. In addition, this thread controls creation and removal of queues, and distribution of bandwidth among the queues. Thread 3 handles communication with the disk driver, as well as assembling batches of requests, and submitting these to the disk driver. This means that thread 1 and 3 handles request management, while thread 2 is responsible for the queue management.

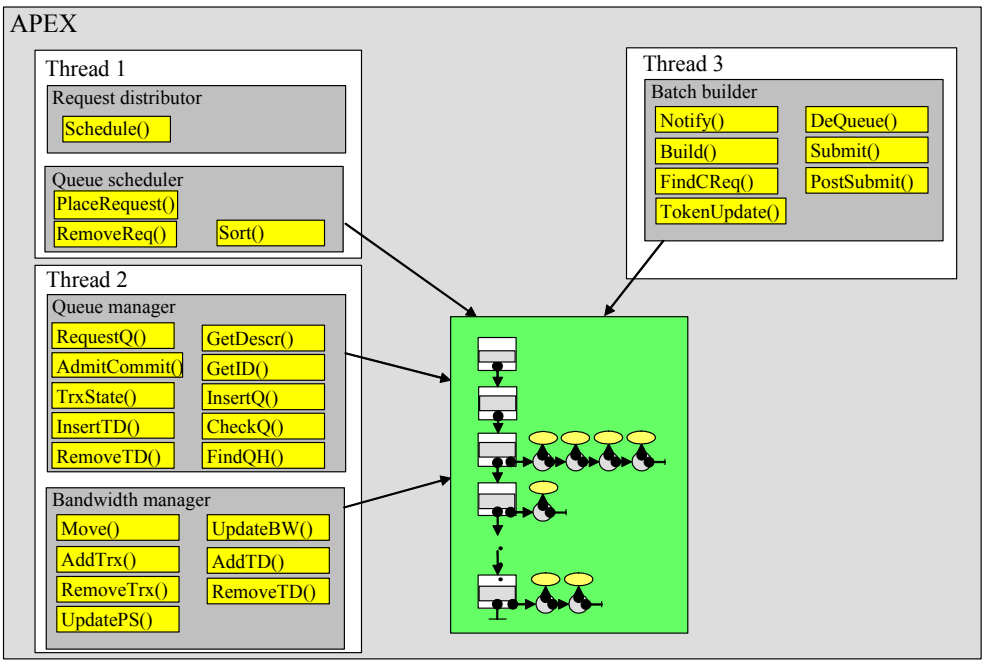

Figure 7-30: Multi-threading and usage of the main data structures in APEX

During normal execution, all threads access the main data structure of APEX, i.e., the queue list. For instance, the `Sort()`-function in the queue scheduler (thread 1) adds requests to the queues, while the `DeQueue()`-function (Figure 7-26) in the batch builder (thread 3) removes requests, possibly from the same queues. To avoid race conditions, it is therefore necessary to protect the data structure using, for instance, semaphores.

There are two entities with critical regions, namely the queue header structure (Figure 7-3) and the request descriptor structure (Figure 7-6). Some of the attributes in these structures are accessed by functions in different threads, and it is therefore necessary to protect, either the attributes or the entire structure, with locks, in order to avoid race conditions. In Table 7-4, we have listed the structures and attributes in question, together with the components accessing them.

Among the schedulers in the reference group, only Cello refers to the usage of threads. According to [85], Cello uses one thread for the class-independent scheduler, and one thread *for each* class-specific scheduler. Thus, with n class-queues, n+1 threads are required.

When a disk request arrives, a class-specific scheduler thread is responsible for receiving the request and place it in the correct position in its queue. The class-independent scheduler thread moves requests from the class-queues into the scheduled queue. To avoid race conditions, access to all queues are controlled by read-write locks.

Table 7-4: Attributes accessed by functions in different threads

Data structure	Attribute	Write	Read
Queue Header	`numRequests`	Queue scheduler (T1) Batch builder (T3)	Batch builder (T3)
	`firstReq`	Queue Scheduler (T1) Batch builder (T3)	Queue Scheduler (T1) Batch builder (T3)
	`nextQ`	Queue/Bw mgr (T2)	Queue/Bw mgr (T2) Batch builder (T3)
	`numTokens`	Queue/Bw mgr (T2) Batch builder (T3)	Batch builder (T3)
	`tokenRate`	Queue/Bw mgr (T2)	Batch builder (T3)
Request Descriptor	`nextReq`	Queue scheduler (T1) Batch builder (T3)	Queue scheduler (T1) Batch builder (T3)

7.6 Summary

In this chapter, we have presented the implementation of APEX. All major functions have been described using detailed pseudo-code, we have analyzed the complexity, and investigated the consequences of using multi-threading.

With respect to our four claims, we have demonstrated in this chapter that it is possible to design and implement a disk scheduler that uses metadata from a MMDBMS, i.e., Claim 1 has been our primary target. In addition, by thoroughly describing each component in APEX, the reader should be able to assess our disk scheduler framework against Claim 2 (configurability of APEX) and Claim 3 (good combination of QoS-support and high utilization).

In the next chapter, we describe how APEX, together with C-LOOK and Cello have been implemented in a simulation environment, and we present the results of the experiments performed on the three schedulers.

Chapter 8

Performance Evaluation

In this chapter, we present the performance evaluation of three disk schedulers, namely C-LOOK, Cello, and APEX. All three schedulers have been implemented in a simulation environment, and they are run with the same workload. We first present the implementation of the simulations and the tools used. Next, we describe the workload used as input to the simulations, before we present the configuration of the experiments. Finally, the results of the simulations are presented, together with an analysis of the results.

The purpose of this chapter is partly to serve as a proof-of-concept, showing that the principles of APEX also work in practice. In addition, the chapter serves as partial proof for all of the four claims that were presented in Section 1.3.

We assume that the reader is familiar with the requirements analysis in Chapter 4, the analysis of related work in Chapter 5, and the description of APEX in Chapter 6 and Chapter 7.

8.1 Introduction

In our analysis of existing disk schedulers in Chapter 5, we found Cello, MARS, and Prism to be the disk schedulers that come closest to our requirements. Ideally, we should implement all three schedulers, and perform comparisons with an implementation of APEX. However, due to limited time, we found it necessary to limit ourselves to one scheduler, in addition to APEX. We chose to implement Cello, since this scheduler appears to be best suited for our environment: MARS lacks support for low-latency service, and Prism does not support interactivity.

We expect APEX, with its batch building principle, to perform well, with respect to disk efficiency. However, providing QoS-support usually has a cost in that respect, and in order to investigate that cost, we have also chosen to compare APEX with a pure performance-oriented scheduling algorithm. For this purpose we chose C-LOOK, since this has proven to be one of the most efficient variants of the SCAN algorithm [106].

When evaluating a disk scheduler, there are several possible approaches, such as using an analytical model, simulating the disk system, or implementing the scheduler on a real system. In our work, we have chosen to use simulations, as this provides us with a very controllable and predictable environment, compared to an implementation on a real system. Still, the environment is relatively realistic, so "real" schedulers can be ported to the simulation environment with a minimum of code modifications. In addition, the simulator enables very detailed measurements, which allows us to investigate the behavior of each scheduler in detail. Using an analytical model is not a feasible solution, since it is very difficult to express the behavior of the different schedulers in such a model.

There are two goals of the simulations: First of all, we use them as proof-of-concept, i.e., we demonstrate that the principles of APEX work in practice. Second, we use the simulations as partial proof of the claims presented in Section 1.3:

- *Claim 1*: We show that APEX can be integrated with a MMDBMS by applying the (simulated) workload of a MMDBMS, and show that APEX can handle this.
- *Claim 2*: The versatility of APEX is demonstrated by applying very different types of workload.
- *Claim 3*: The combination of QoS-support and high bandwidth utilization is demonstrated by comparing APEX with Cello and C-LOOK. By running the same simulations with Cello and APEX, we wish to show that APEX provides better disk efficiency than Cello, given the same level of QoS. We then compare APEX and Cello to C-LOOK, which is used as a reference with respect to pure performance. This gives us an indication of the "cost" of the QoS-support, both in APEX and Cello.
- *Claim 4*: By investigating the behavior of C-LOOK, we get an indication of how this class of schedulers, which is the one used internally in disks, is suited for environments where QoS-support is required.

8.2 Implementation

We first describe the simulation environment, and how the three disk schedulers have been adapted for use in this environment. In addition, we describe the format of the trace-files used as input to the simulations, as well as the format of the output, and how we analyze this information.

8.2.1 DiskSim

We have chosen to use a disk system simulator called DiskSim [30]. This simulation environment is developed at the University of Michigan, and it is intended as a support for research within storage subsystems.

The simulator includes modules for a number of components within the storage subsystem, including disks, controllers, buses, and disk drivers. Since these components are modeled individually, they can be configured and connected in a number of ways, depending on the experiments to be performed.

The disks are modeled particularly detailed, and the disk component has been validated against several commercial disks [12, 105, 106]. According to the authors, DiskSim provides an accuracy that "exceeds any previously reported simulator" [30].

DiskSim can be driven by external disk request traces, or by an internally generated synthetic workload. For our experiments, we rely solely on the use of external disk requests, since this allows full control of the workload.

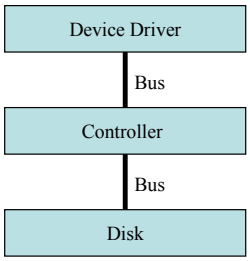

Figure 8-1: Physical organization of the components simulated in DiskSim

Furthermore, DiskSim offers a large number of adjustable parameters. However, included in the DiskSim package is a set of pre-defined configurations of these parameters, based on real components. In our simulations we have used such pre-defined configurations, and it has therefore only been necessary to specify the "physical" setup, i.e., which components that are used, and how they are interconnected. In Figure 8-1, we

show the component organization used in our simulations. The actual setting of each single parameter is shown in Appendix A.

8.2.2 Using Disk Schedulers with DiskSim

We have implemented all three schedulers on top of DiskSim. As illustrated in Figure 8-2, we feed exactly the same workload (i.e., the same trace file) to each of the schedulers, which, in turn, order the requests according to their algorithms. For DiskSim itself, there are simulation parameters that can be adjusted, and finally Cello and APEX provides adjustable simulations parameters such as bandwidth distribution.

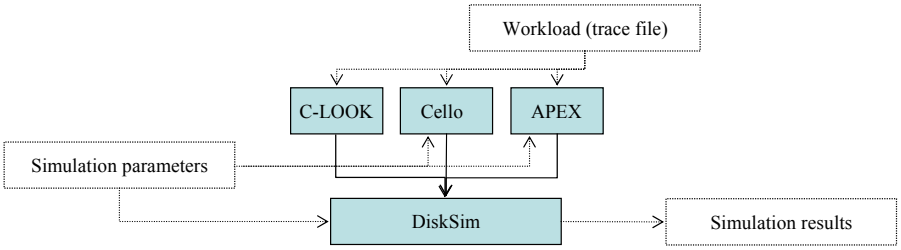

Figure 8-2: Using DiskSim for evaluation of disk schedulers

Common in all three scheduler set-ups is the use of simulated time, held in a global variable. This simulated time is driven by DiskSim, i.e., a new request is added to a queue when the time computed by DiskSim is equal to the arrival time of the request. This works well, because DiskSim updates this simulated time a large number of times for every request that is processed: When a scheduler submits a disk request to DiskSim, it uses the `disksim_interface_request_arrive`-call to submit the request. Before DiskSim returns the call, it updates a global variable with the next point in time at which an event will take place in DiskSim. The simulation in DiskSim is then driven by `disksim_interface_internal_event`-calls. Each call leads to the same global variable being updated with the next point in time for a DiskSim-event to take place.

In summary, each simulation of the serving of a disk request starts with a `disksim_interface_request_arrive`-call, followed by a number of `disksim_interface_internal_event`-calls, typically 500-1000 per disk request. For each call, the simulated time is updated, and this means that we achieve a simulated time with a granularity of less than ten microseconds.

If DiskSim is finished with all queued requests before the arrival time of the next request in the trace, there is a danger that time will stop progressing. However, if such a

situation occurs, we let the scheduler move time forward to the arrival time of this next request.

Since the disk schedulers must relate to a trace file instead of actual disk requests, it has been necessary to make minor changes to the schedulers, compared to a real implementation. Below, we describe the implementation of the three disk schedulers in their simulation version.

Cello

The designers of Cello, P. Shenoy and H. M. Vin, originally implemented their own disk simulator, in order to evaluate disk schedulers. This simulator, incidentally called DiskSim as well, has many features in common with the simulator we use, although it models the disk considerably less detailed and it does not take other components of the storage subsystem into consideration.

The simulation version of Cello, together with the simulator, is publicly available [82], and due to the many similarities between the two simulators, we were able to port Cello with a minimum of modifications. The main difference lies in how the two versions feed in requests from the trace file.

Our implementation uses three threads; one for handling the reading of requests from the trace file and inserting them into the correct queue, one for Cello itself (i.e., picking requests from the class queues and inserting them into the scheduled queue), and one for DiskSim together with the function that picks requests from the scheduled queue. This is slightly different from the real Cello implementation, which uses one thread per class-queue scheduler and one for the class-independent scheduler [85].

C-LOOK

As a reference with respect to performance, we chose to implement C-LOOK, which has proven to be the most efficient variant of the SCAN-algorithms [106]. The differences from traditional SCAN are that the disk arm is allowed to return when it reaches the last cylinder for which there are requests and that the disk arm serves requests only in one direction.

Since C-LOOK is not a QoS-aware scheduler, it is unable to consider the type of the disk requests. Instead, all requests are placed in the same queue, with disk address as the only sort criterion. The queue is split in two: the first part contains all disk requests with block address higher than the current position of the disk head, and sorted on ascending

disk addresses. The second part of the queue contains all disk requests with block addresses smaller than the current head position, and also these requests are sorted on ascending addresses. When all requests in the first part of the queue has been served, i.e., the disk head reaches the outermost position, the second part of the queue becomes the first part, and a new second part is started.

The implementation uses two threads, one for reading requests from the trace file and placing them in the correct position in the queue, and one for DiskSim.

APEX

For APEX, it has been necessary to adapt the queue manager and the bandwidth manager to read from files, instead of receiving requests from the MMDBMS (see Figure 8-3). Thus, at the start of a simulation, the queue specification files are read, and the specified queues are instantiated and assigned bandwidth.

Like in Cello, it has also been necessary to modify the input of requests. In the simulation version, the request distributor actively fetches requests from the trace file each time the simulated time reaches the arrival time of the next request.

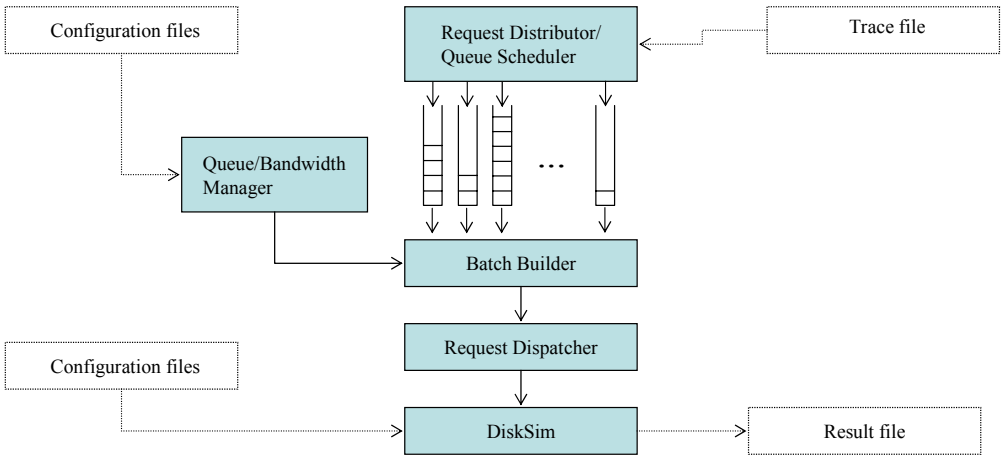

Figure 8-3: Implementation of APEX on top of DiskSim

The `Submit()`-function is implemented according to a single request interface, i.e., only one request can be submitted per system call (see Sub-section 6.1.4). In addition, the final sorting of requests is done in the request dispatcher, instead of in the disk driver, and only one request submitted to DiskSim at a time. This was necessary due to the fact that DiskSim seemed to have problems handling multiple outstanding requests. However, it is not clear whether this was caused by configuration problems or bugs in DiskSim.

On the other hand, Cello requires full control of when and in which order the requests are submitted to the disk driver, making the single-request solution described above a necessity. Thus, the two schedulers are compared under more equal conditions, but APEX loses the potential performance gain provided by disk-internal scheduling.

The APEX implementation uses three threads; one for the request distributor/queue scheduler, one for the batch builder, and one for the request dispatcher, i.e., the `Submit()`-function, together with DiskSim and the queue/bandwidth manager.

8.2.3 Input to the Simulations

The workload for the simulated schedulers is a file containing a list of disk requests. For each simulation, there is only one workload file; thus, if there are more than one type of disk requests (for example, real-time and best-effort requests), these are merged into one file, and sorted on arrival time. Each line in the file constitutes one disk request, and contains the following information:

- *Request flags*: Used to indicate whether the request is a read or write.
- *Device number*: Specifies to which disk the request should be sent.
- *Block number*: The address of the first block to be read. DiskSim uses LBN (logical block number) addressing, thus, the disk appears as a long array of consecutive blocks.
- *Request size*: The number of bytes requested (must be a multiple of the sector size – 512 bytes). In our simulations, the request size is always fixed, such that all requests request the same amount of data, and we use a request size of 65536 bytes (64KB).
- *Queue ID*: The ID of the APEX queue that the request should be put in. As explained in Sub-section 7.3.2, this number is used as an index into the ActiveQueues-array. This parameter is also used in Cello, to identify the correct queue, while it is ignored by C-LOOK.
- *Arrival time*: The time (in seconds) when the request arrives at APEX. As explained in Sub-section 8.2.2, the request distributor compares this value to the simulated time, to know when the request should be inserted into the correct APEX queue. This value is also the sorting criterion in the trace file, such that all requests in the trace are listed in order of ascending arrival time.
- *Deadline*: The deadline (in seconds) for the request, given in absolute time. If a request does not have a deadline, this value is set to zero.

- *Priority*: If the request uses a queue that supports priorities, the priority of the request is given here. Otherwise this value is set to zero.
- *Transaction ID*: This value gives the ID of the transaction that the request belongs to.

The first four parameters constitute a "standard" disk request, and are, together with the arrival time, the parameters required by DiskSim. The remaining parameters are used by APEX (we call these APEX-parameters). Out of the APEX-parameters, Cello uses Queue ID and deadline, and ignores the rest, while C-LOOK ignores all these parameters. In addition, all schedulers use the block number when ordering the requests based on position of the data.

In addition to the workload, APEX also requires information about the queues that should be instantiated. We do this with two configuration files; the first one is used to create the ServiceTable array, and contains a list of queue specifications. For each queue, we specify guarantee level, allocation paradigm, whether it is a candidate queue, etc. The second file specifies which of these queues that should be instantiated, and their queue IDs. In addition, token rate or weight (depending on allocation paradigm), as well as bucket depth are specified for relevant queues.

8.2.4 Output from the Simulations

DiskSim itself provides very detailed statistics about the results of the simulation. However, all this information is aggregated, i.e., it is not possible to identify results for a single transaction, a single queue, or an individual request. In addition, DiskSim only provides information about the handling of requests within DiskSim, while we are also interested in the handling of the requests within the schedulers, which are outside of DiskSim.

To be able to compute statistics at a sufficiently detailed level, we have added a time registration module that, for every disk request, registers when it arrives in the scheduler, when it is submitted to DiskSim, and when DiskSim is finished serving the request. This new module runs outside of DiskSim and it only registers the simulated time, i.e., it just reads the global time variable, and has no possibility of modifying it. Thus, the time registration module has no effect on the simulation results.

Using our module, each request is registered in a result file with the following information:

- *Queue ID*: In a real implementation, this value can be discarded as soon as the request is placed in the correct queue. However, to be able to calculate statistics on a per-queue basis, this value follows the request all the way through the simulator.
- *Block number*: The address of the first sector that was read or written.
- *Arrival time*: The time when the request arrived at the disk scheduler.
- *Start time*: The time when the request was selected for service, i.e., the request was submitted into DiskSim.
- *Finishing time*: The time when DiskSim reported that the request was completed.
- *Service time*: The time needed by DiskSim to serve the request, computed as finishing time minus start time.
- *Response time*: The time needed by the storage subsystem to serve the request, computed as finishing time minus arrival time.
- *Transaction ID*: To be able to compute statistics on a per-transaction basis.

Even though C-LOOK does not utilize any of the APEX-parameters, they are still preserved as the requests pass through the scheduler. Thus, using the result files, we are able to perform a detailed comparison of the behavior of the disk schedulers.

Most of our analysis is performed on a per-queue basis, where we compute average and maximum response times, as well as 95-percentiles. In addition, we measure the throughput of large transfers, by measuring the time from the first to the last disk block being transferred. Finally, by computing average service time, we can say something about the efficiency of the disk schedulers. Lower average service time means that the schedules leads to less non-productive work (i.e., head positioning), which in turn indicates a more efficient disk scheduler.

8.3 Workload

In order to create a realistic workload, we use four different types of workload in our simulations: continuous media playback (corresponding to a multimedia playback query), metadata retrieval query, checkout operation, and low-latency requests. In the following sub-sections, we describe each of these workload types.

8.3.1 Continuous Media Playback

In our simulations, we assume that audio and video are multiplexed when stored on disk. However, due to limitations in our tools for creating traces, it has been necessary to treat the two MMDTs separately, and then join them as a final step.

Video

We assume that the videos stored in the DBS can be coded in different formats, such as MPEG-I, MPEG-2, DVD[9], MPEG-4, etc., based on the preferences of the author and the content of the video. For instance, for a "talking head" type of video, low resolution MPEG-1 may suffice, while a video showing a detailed medical procedure requires much higher resolution. Thus, different videos may have very different bandwidth requirements.

For our experiments, we have used six different videos, encoded in different MPEG formats and in various resolutions (see Table 8-1). "Lecture1" to "Lecture3" are video recordings of classroom lectures held at UniK, while "Doc1" is a documentary on quality assurance in a Norwegian company, made especially for a course in quality assurance held at UniK. "Medical" is an instructional video used by the medical faculty at the University of Oslo, and finally, "Doc2" is a documentary about Cuba, recorded from television. All these videos, except the Cuba documentary, are actually used in electronic distance education today, and, as such, represent a very realistic workload.

Table 8-1: Video traces used in the simulations (the bandwidth requirements in pages/s are based on a page size of 64KB)

Name	Format	GOP	Length (min)	Reso-lution	Rate (fps)	Size (MB)	Avg. bandw. KB/s (pages/s)	Max. bandw. KB/s (pages/s)
Lecture1	MPEG-2	$I(BBP)^4BB$	48	720x576	25	1668.1	582 (9.1)	768 (12)
Lecture2	MPEG-1	$I(BBP)^4BB$	60	320x240	25	348.6	98 (1.53)	192 (3)
Lecture3	MPEG-2	$I(BBP)^4BB$	46	352x288	25	416.1	154 (2.4)	256 (4)
Doc1	MPEG-1	$I(BBP)^4BB$	29	480x360	25	355.8	212 (3.32)	320 (5)
Medical	MPEG-1	$I(BBP)^4BB$	11	720x576	25	221.1	345 (5.39)	448 (7)
Doc2	DVD	$I(BBP)^3BB$	49	720x576	25	1203.7	421 (6.58)	1088 (17)

The videos were first transferred from DV-tape to avi-files, before using TMPGEnc [43] to encode them into the different MPEG formats. Next, traces consisting of long lists

[9] DVD is one particular interpretation of the MPEG-2 standard.

of frame sizes were generated, using a modified version of mpeg2dec [52, 70] . In order to make these traces useable for APEX, we must convert the frame sizes into disk requests, by aligning the frame sizes with the page size. Thus, several small frames may fit within one page, while large frames may require several blocks.

When doing this, we also determine the storage structures of the simulated data. In the simulations, the frames of each video are grouped into one-second chunks, each chunk consisting of 25 frames. To the storage subsystem, such a chunk constitutes an atomic unit with no visible internal structure. For each chunk, we sum up the sizes of the 25 frames, and divide by the page size. If the result for one such chunk is *n* pages, then *n* page requests, *with the same deadline*, are generated.

Audio

We assume that all audio tracks are constant bit-rate, which is in accordance with the MPEG-1 audio standard (for layer 1 and 2, support for variable bit-rate coding is "not mandatory") [71]. In Table 8-2, we list the characteristics of the audio tracks for the different videos. Note that, the video "Medical" does not have a sound track.

Table 8-2: Audio traces used in the simulations

Name	Format	Size MB	Bandwidth KB/s
Lecture1	MPEG-1 Audio layer 2	135.0	48
Lecture2	MPEG-1 Audio layer 2	84.4	24
Lecture3	MPEG-1 Audio layer 2	64.7	24
Doc1	MPEG-1 Audio layer 2	40.8	24
Doc2	MPEG-1 Audio layer 2	133.9	48

As mentioned, we have not been able to provide audio trace data for multiplexed videos, and the size and bandwidth requirements shown in Table 8-2 are therefore based on the standard bit-rates listed in the MPEG-1 audio standard.

Multiplexing Audio and Video

Starting with the original video frame trace, we divide the size of one second of audio with the video frame rate, and, thus, get the "number of audio bytes per video frame", which is constant, since we assume CBR audio. This number then is added to each frame size in the frame trace. Next, we create the disk request traces in the same way as described for video.

Since we use a round-based model, which implies that several frames are requested in each round, this approach provides a relatively good approximation of the disk load.

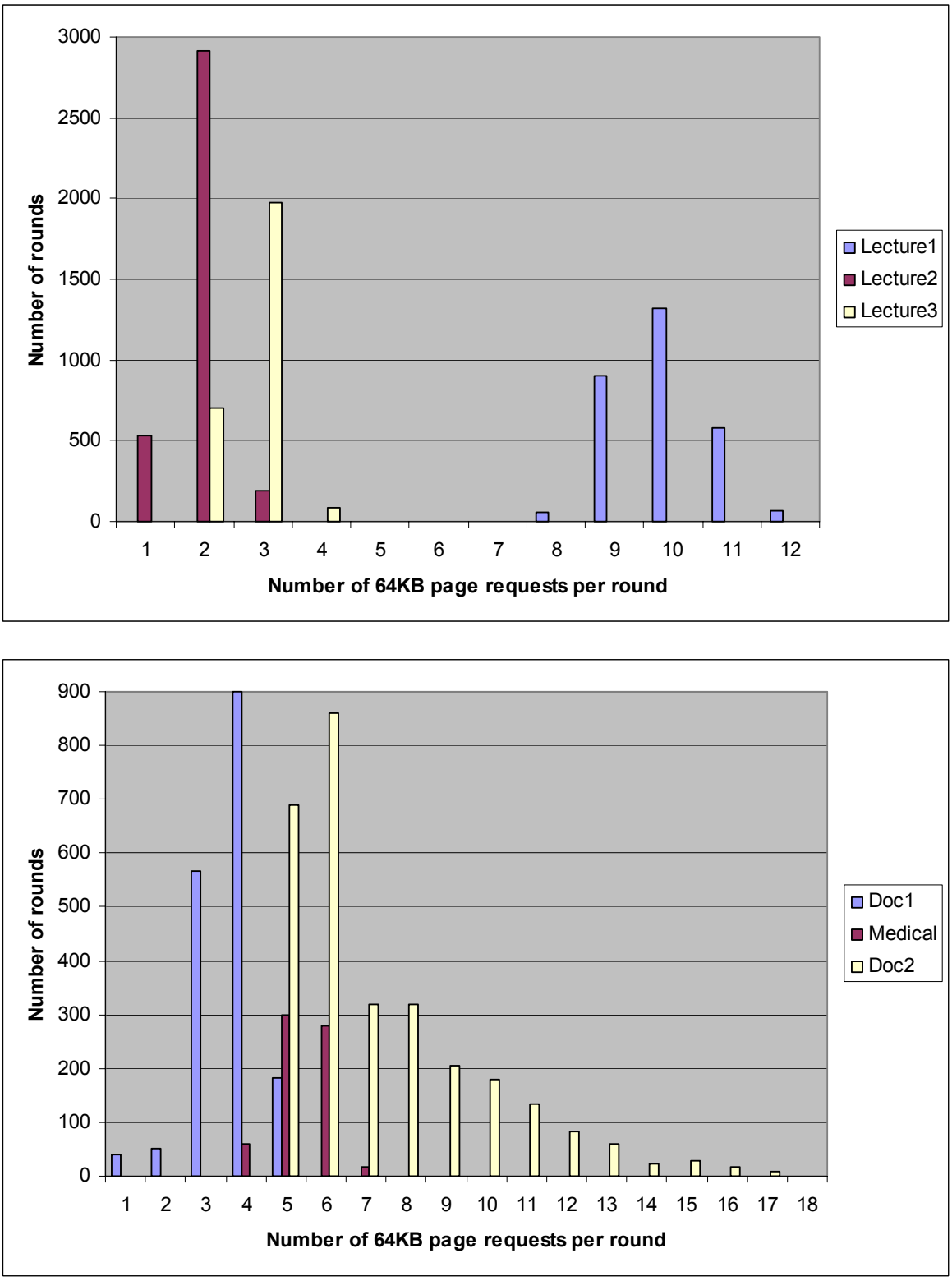

Figure 8-4: Number of 64KB page requests per one-second round for each video

In Figure 8-4, we show the resulting bandwidth distribution for each of the resulting videos with multiplexed audio. To illustrate the variability of the bandwidth, we also show the bandwidth trace for the “Doc2” video, in Figure 8-5.

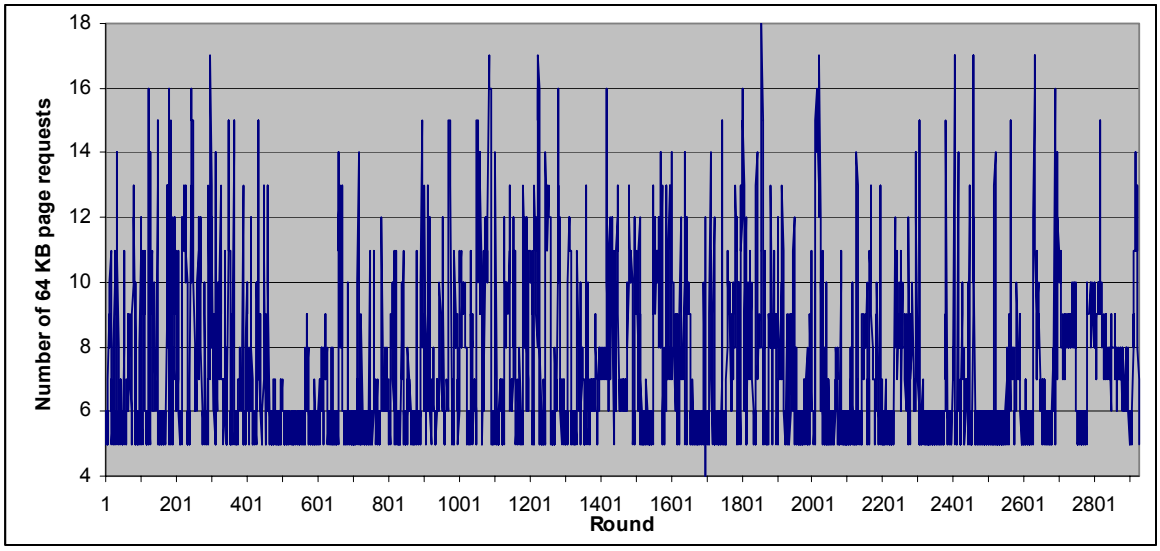

Figure 8-5: Bandwidth trace for the "Doc2" video

8.3.2 Metadata Retrieval Query

This workload type covers metadata retrieval and metadata authoring, and in the remainder, we refer to these as MR-queries. Unfortunately, we have been unable to find adequate traces that show disk request patterns in a DBMS query context. In addition, such patterns will be dependent on the type of DBMS, the storage structures used, the extent to which indexes are used, etc.

Instead, we have created a workload based on measurements performed in [9], where, depending on the query, between 2 and 1011 disk requests were submitted during each query, with an average of 364 requests. These values are based on a page size of 8KB, and we have adjusted them according to the page size we use.

The disk requests constituting a MR-query have an exponentially distributed inter-arrival time, with a mean of 9.7 milliseconds, which is the smallest average inter-arrival time found in [9]. A user-query trace contains a set of such MR-queries, with exponentially distributed inter-arrival times with an average of 10 seconds. In order to vary the workload, we vary the number of such user-query traces. Since each query starts at a random time, this means that two or more queries may run concurrently.

8.3.3 Checkout Operations

This type of traffic is characterized by a large burst of disk requests arriving in a very short time. We assume that the multimedia object being checked out is transferred over the network to a client machine. Thus, the bandwidth of the client's network connection is a limiting factor for the arrival rate of the requests.

In our experiments, we assume that each client has a 100Mb/s network connection, which means that the theoretical upper bound for the page request rate is one request every 5 millisecond, given a page size of 64KB. Thus, in the simulations, the disk requests constituting a checkout operation have an inter-arrival time of 5 milliseconds.

Each checkout operation is for an entire video, thus, depending on the video, between 261.9 and 1823.1 MB is read from disk in such an operation.

8.3.4 Low-Latency Requests

These are requests that require low response times, such as requests for index data in the MMDBMS. The workload is based on the interactive best-effort workload used in the evaluation of Cello [86], and consists of single disk requests with exponentially distributed arrival time. The workload is varied by changing the average inter-arrival time of the request, and we use values from five seconds, down to 0.1 seconds.

8.4 Configuration of Experiments

In addition to the workload, there are several variables that can affect the behavior of the disk schedulers, as we described in Section 6.6. In this section, we show the settings of these parameters in our experiments, as well as the combination of workloads used in the different experiments.

8.4.1 Simulation Parameters

In Section 6.6, we discussed the impact that external factors such as disk block size, data placement, length of round times, and disk performance have on the performance of the disk schedulers. For our simulations, we have chosen one combination of these parameters, which we describe below:

- *Data placement*: In our simulations, we have chosen to use random data placement, in order to set up a worst-case scenario. Thus, when building the trace files, a random disk

block was chosen. If that block was already occupied, the block address was increased by one, and so on, until a free block was found.

- *Disk block size*: We use a disk block size, and, thus, a database page size, of 64 KB. Thus, every disk request sent through the disk schedulers is for 64 KB of data. We have chosen this size, since it is a good trade-off between throughput and latency [39, 63].
- *Round time*: Many multimedia servers use a round time of 0.5 to one second [2, 64, 83, 103], as this represents a good trade-off between disk efficiency, latency, and required buffer space. We have chosen to use a round time of one second in our simulations.
- *Disk performance*: Obviously, the performance of the disk has an impact on the simulation results. However, the disks that DiskSim can simulate must be thoroughly analyzed first, and we are therefore forced to use the disk models that are provided in the DiskSim package. In our simulations we chose the Quantum Atlas 10K, which is one of the fastest disks that have been analyzed for DiskSim. In Table 8-3 we show the characteristics of this disk. With random data placement and a block size of 64 KB, we have measured the average throughput of the disk to be approximately 105 blocks per second. This gives t_{es} = 1/105 = 9.52 ms.

Table 8-3: Characteristics of the Quantum Atlas 10K disk [58]

Size	9100 MB
Interface	Ultra 160 SCSI
Seek time (min / avg / max)	0.8 / 5.0 / 12.0 ms
Rotational latency	3.0 ms
Rotation speed	10000 rpm
Sustained throughput	18 - 26 MB/s

8.4.2 Combinations of Workload

In Section 8.3, we described the four types of workload we have created. For our simulations, we now merge different combinations of these workload types, and use these mixed workloads to evaluate the three disk schedulers.

In Table 8-5 we show the different workload configurations used in our simulations. In addition, we show which workload type we vary, and what we measure. The letters indicating workload type refers to the list in Table 8-4.

Table 8-4: Types of workload

Symbol	Workload
A	Continuous media playback (audio/video)
B	MR-queries
C	Checkout
D	Low-latency

For instance, configuration 2 consists of continuous multimedia data, MR-queries, and checkout operations (A, B, C). We keep the multimedia load and the checkout load constant, and vary the number of MR-query traces. As measurements, we check if there are any deadline violations among requests for multimedia data, we measure the response time of the MR-query requests, and the overall response time and throughput of the checkout requests.

In all the simulations, we assume that the start times for the continuous media playback clients are exponentially distributed, with a mean inter-arrival time of five seconds. In addition, each simulation is run for 600 seconds of simulated time.

Table 8-5: Workload configurations

	Workload types	Workload varied	Measured
1	A, B	B	A (deadline violations) B (response times)
2	A, B, C	B	A (deadline violations) B (response times) C (response time and throughput)
3	A, B, D	D	A (deadline violations B (response times) D (response times)
4	A, B, C, D	A	A (deadline violations) B (response times) C (response time and throughput) D (response times)

8.5 Simulation Results

Each of the workload configurations from Table 8-5 were run on all three schedulers, and in this section, we present the results of these experiments. For the response time measurements, we show the average and maximum value, as well as the 95-percentile for

the measured values, i.e., for each value shown in the graphs, 95% of the actual values measured are smaller than, or equal to, this value.

It is important to note that Cello and APEX are based on different allocation paradigms. Cello, being a proportional-share scheduler, allocates bandwidth in shares (percent) of the total disk bandwidth. APEX, although it supports proportional-share allocation, is basically reservation-based, and reserves bandwidth in pages (i.e., disk blocks) per second. For both schedulers, we first established how much bandwidth that was necessary for the real-time requests, by turning off the work-conservation, and reducing the allocated bandwidth as much as possible without causing deadline violations. This is why there may be slightly different bandwidth allocations for the real-time queues in APEX and Cello.

In the experiments involving more than two types of workload, there are no rules or guidelines for distribution of disk bandwidth between the non-real-time queues. Thus, the distribution used in this paper is just the first of several different alternatives, and more will be tried, as part of future work.

8.5.1 Experiment 1

In this experiment, we use a fixed, real-time workload, corresponding to six playbacks of randomly selected videos. We then vary the MR-query workload by adding more and more traces, each one as described in Sub-section 8.3.2.

We assume that there are 6 video clients, each playing back one of the six videos described above, selected randomly. The start times for the video clients are exponentially distributed, with a mean inter-arrival time of five seconds. The selected videos and their starting times are shown in Table 8-6.

Table 8-6: Start times for video playbacks

	Time	Video
1	0	Lecture1
2	4	Medical
3	5	Doc1
4	14	Lecture1
5	14	Doc2
6	17	Medical

In Cello, we reserved 50% of the bandwidth for the real-time (RT) queue, and 50% for the interactive best-effort (IBE) queue, while for APEX; we reserved 55 pages/s for the real-time queue, and 50 pages/s for the MR-query queue.

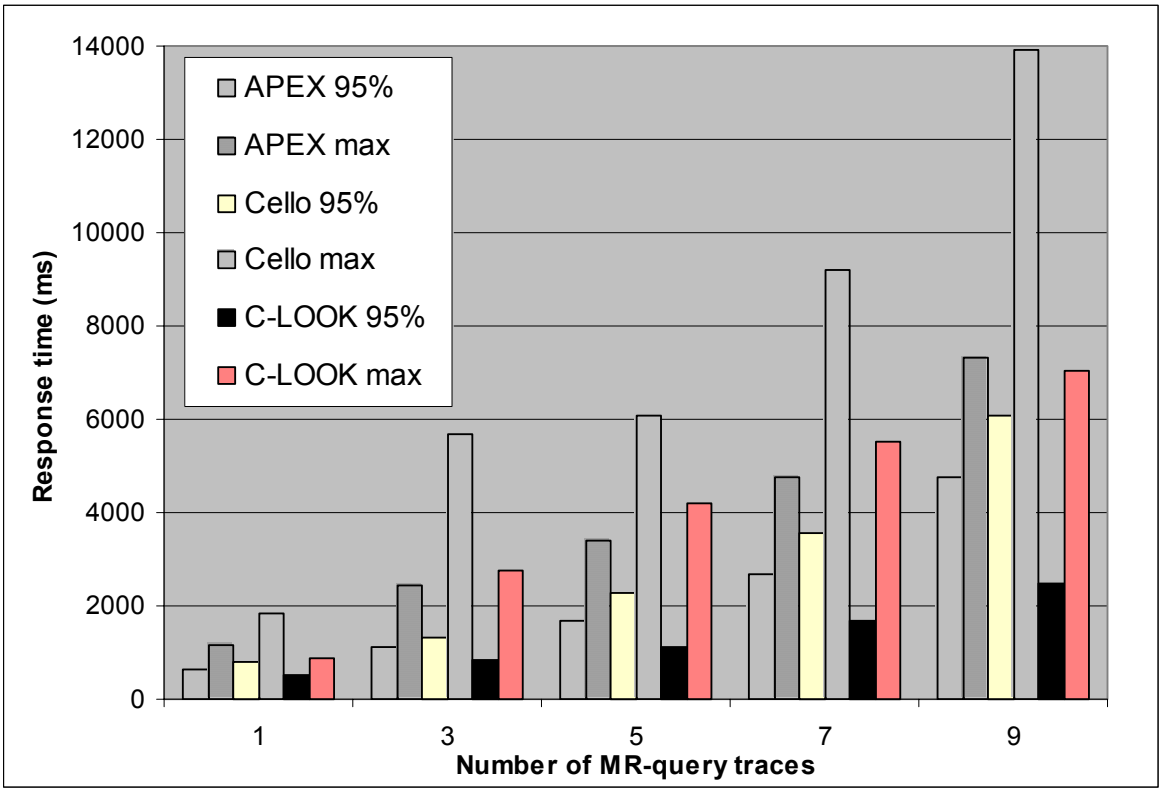

Figure 8-6: Response times for disk requests generated by MR-queries

The result of the simulation is shown in Figure 8-6. We see that for the 95-percentile, C-LOOK performs best, but the maximum response times are considerably higher than APEX. Cello consistently displays the highest response times. As shown in Table 8-7, both APEX and Cello avoid deadline violations, while C-LOOK caused a relatively high number of violations at high load.

Table 8-7: Deadline violations for real-time requests, with nine MR-query traces

	# violations	Average	95%	Maximum
APEX	0	-	-	-
Cello	0	-	-	-
C-LOOK	3266	769 ms	2186 ms	4626 ms

Since we expect the workload to vary extensively, and the bandwidth allocations therefore may be misaligned with the actual needs in short periods, we also investigate how sensitive APEX and Cello are to such misalignment. In this simulation, we use the

same workload, but this time all bandwidth is allocated to the real-time queue in both schedulers. Thus, the MR-query disk requests must rely on work-conservation only.

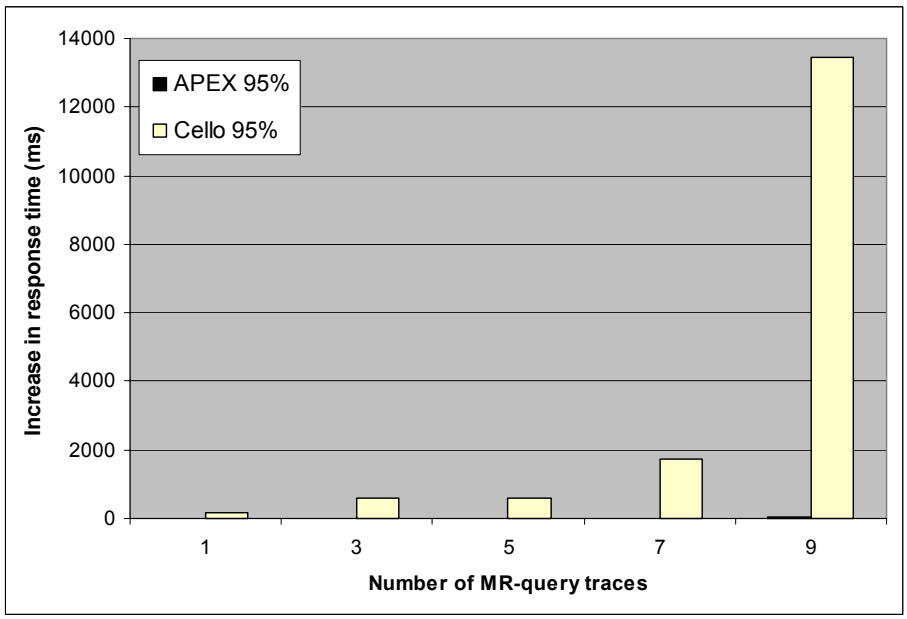

Figure 8-7: Increase in response time for MR-query disk requests with all bandwidth assigned to the real-time queue

From Figure 8-7, we see that APEX performs considerably better than Cello in this experiment. The reason is that, when Cello selects requests during the work-conserving phase, it puts the requests at the end of the scheduled queue, i.e., it does not consider the placement of the requested data on disk. APEX, on the other hand, places all requests in the same batch, regardless of whether the requests were selected during the normal phase or the work-conserving phase, and this batch then is sorted in SCAN.

While this experiment demonstrates the robustness of APEX with respect to *over*-allocation of bandwidth to real-time queues, it is reasonable to assume that in the opposite case, both APEX and Cello will cause deadline violation if the total load is sufficiently high. However, we have not yet performed any such experiments.

8.5.2 Experiment 2

Here we keep the workload from the previous experiment, but in addition, we add a checkout-operation, as described in Sub-section 8.3.3. For Cello, we assign 50% of the bandwidth to the RT queue, 25% to the IBE queue, and 25% to the throughput-intensive best-effort (TIBE) queue (for the checkout operation). The corresponding figures for APEX are 55 page/s, 25 pages/s and 25 pages/s, respectively.

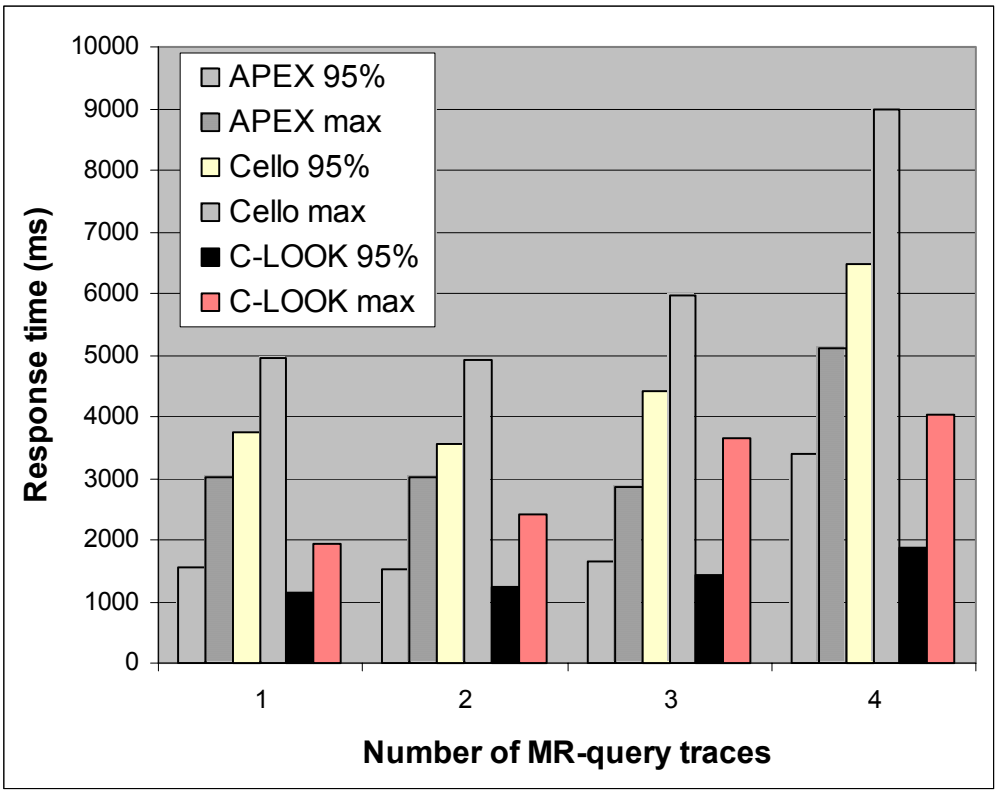

Figure 8-8: Response time for MR-query disk requests

As can be seen in Figure 8-8, APEX performs better than Cello in this experiment. The reason for this is that while APEX groups requests of all types into large batches, Cello treats each request type individually. The result for Cello is three small "SCAN groups" instead of one large, and this reduces the efficiency of the disk.

C-LOOK achieves the lowest response times for MR-query disk requests, but while neither APEX nor Cello violate any deadlines, C-LOOK does: With four MR-query traces, 1246 deadlines were violated by, on average of 540 ms (see Table 8-8).

Table 8-8: Deadline violations for real-time requests, with four MR-query traces

	# violations	Average	95%	Maximum
APEX	0	-	-	-
Cello	0	-	-	-
C-LOOK	1246	540 ms	1422 ms	2779 ms

We also measured the throughput achieved for the checkout-operation, and the results are shown in Figure 8-9. The figure shows the average throughput measured over the entire duration of the operation. As can be seen from the figure, APEX displays the best throughput, followed by Cello, and with C-LOOK last.

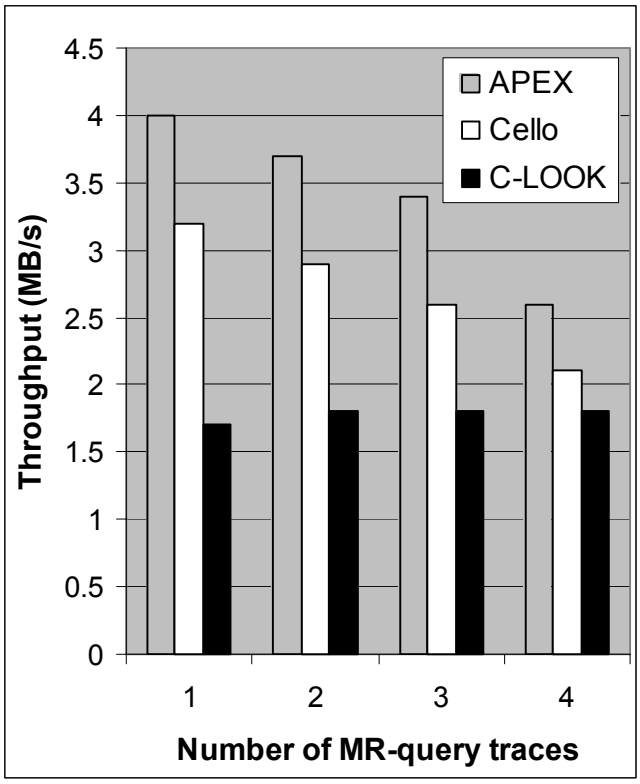

Figure 8-9: Throughput for checkout-operation

8.5.3 Experiment 3

In this experiment, we investigate the ability of the schedulers to offer a low-latency service. APEX offers a special service class for this purpose, while for Cello, we are forced to use the IBE queue, i.e., the same queue that is used for MR-queries.

The real-time workload is the same as in the previous experiments, and the MR-query workload is kept constant, using five traces. We then vary the low-latency workload, by varying the average inter-arrival time of the low-latency requests.

In APEX, we assign 55 pages/s to the real-time queue, 50 pages/s to the MR-query queue, while no bandwidth is assigned to the low-latency queue. In Cello, we assign 50% of the bandwidth to the RT queue, and 50% to the IBE queue.

As can be seen from Figure 8-10, APEX consistently achieves lower response times for the low-latency request, than the other two schedulers, even with an average arrival rate of ten low-latency requests per second. Except for the last simulation (100 ms inter-arrival time), the increase in response times of the MR-query disk requests was less than 4%, for all schedulers. With 100 ms inter-arrival time, the increase was 34%, 19%, and 17%, for APEX, Cello, and C-LOOK respectively.

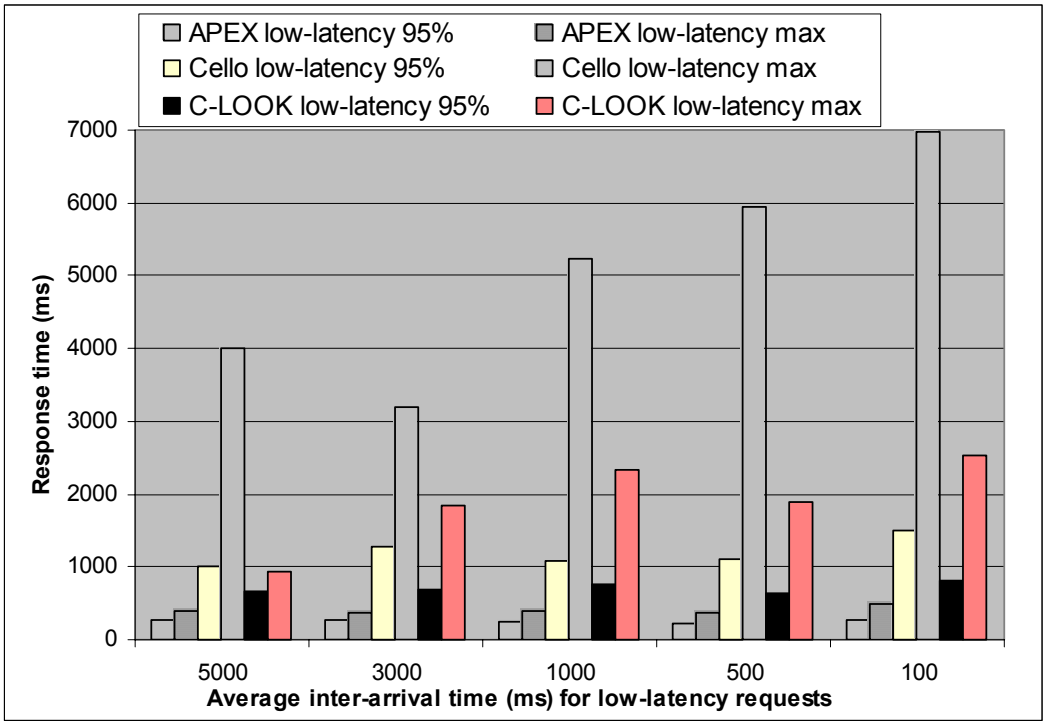

Figure 8-10: Response times for low-latency requests

8.5.4 Experiment 4

In the final experiment, we combine all four workloads, in two different simulations. The first is based on one real-time client, while the second is based on four real-time clients. In both simulations, we use one MR-query trace, and low-latency requests with an average inter-arrival time of one second.

For Cello in the first simulation, we reserve 12% of the bandwidth for the RT queue, 78% to the IBE queue, and 10% to the TIBE queue. The corresponding reservations for APEX are 12 pages/s for the real-time queue, 82 page/s for the MR-query queue, and 11 pages/s for the checkout queue.

Figure 8-11 shows the response times for low-latency and MR-query disk requests, while Table 8-9 shows the throughput for the checkout operation. Neither APEX nor Cello violated any deadlines, and C-LOOK only violated three deadlines, with maximum 221 ms.

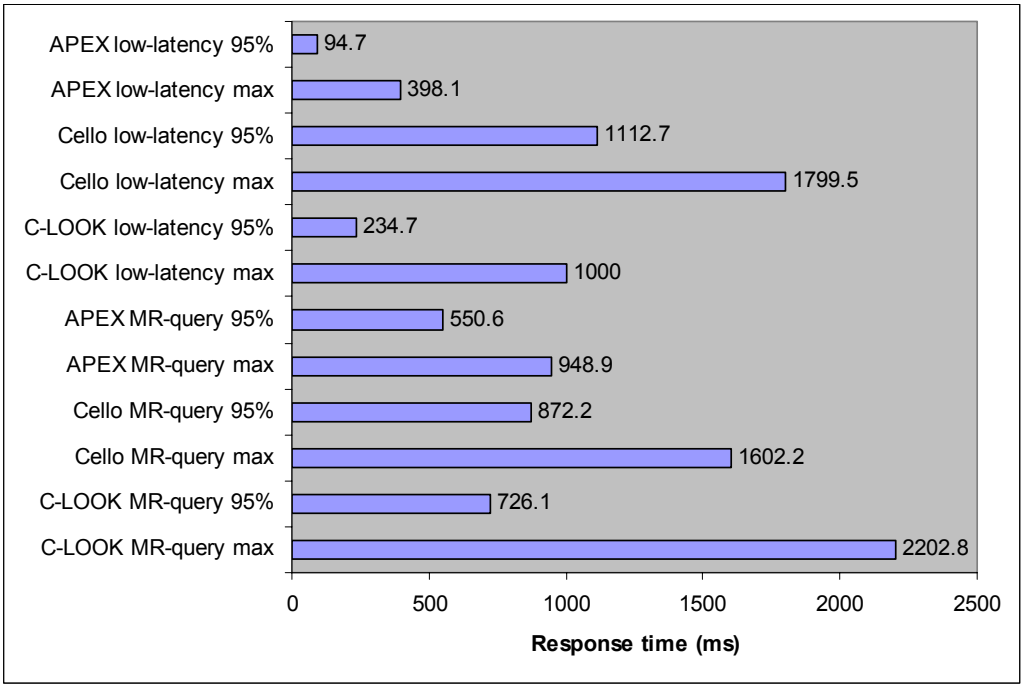

Figure 8-11: Response times for low-latency and MR-query disk requests, with one real-time client

In the second simulation, using four real-time clients, we assign for Cello 30% of the bandwidth to the RT queue, 60% to the IBE queue, and 10% to the TIBE queue. For APEX, we assign 31 pages/s to the real-time queue, 63 pages/s to the MR-query queue, and 11 pages/s to the checkout-queue.

Table 8-9: Throughput for checkout operation (MB/s)

	APEX	Cello	C-LOOK
1 real-time client	5.7	4.4	2.7
4 real-time clients	4.6	3.6	2.5

The results are shown in Figure 8-12, while Table 8-9 shows the throughput for the checkout operation. We see that APEX achieves considerably better response times for low-latency requests than Cello. Also for the MR-query disk requests, APEX performs better than Cello. C-LOOK performs relatively well, but as the maximum values show, the variability of the response times is much higher than for APEX.

Once again, APEX and Cello avoid deadline violation, while C-LOOK violates a modest 53 deadlines, by a maximum of 942 ms.

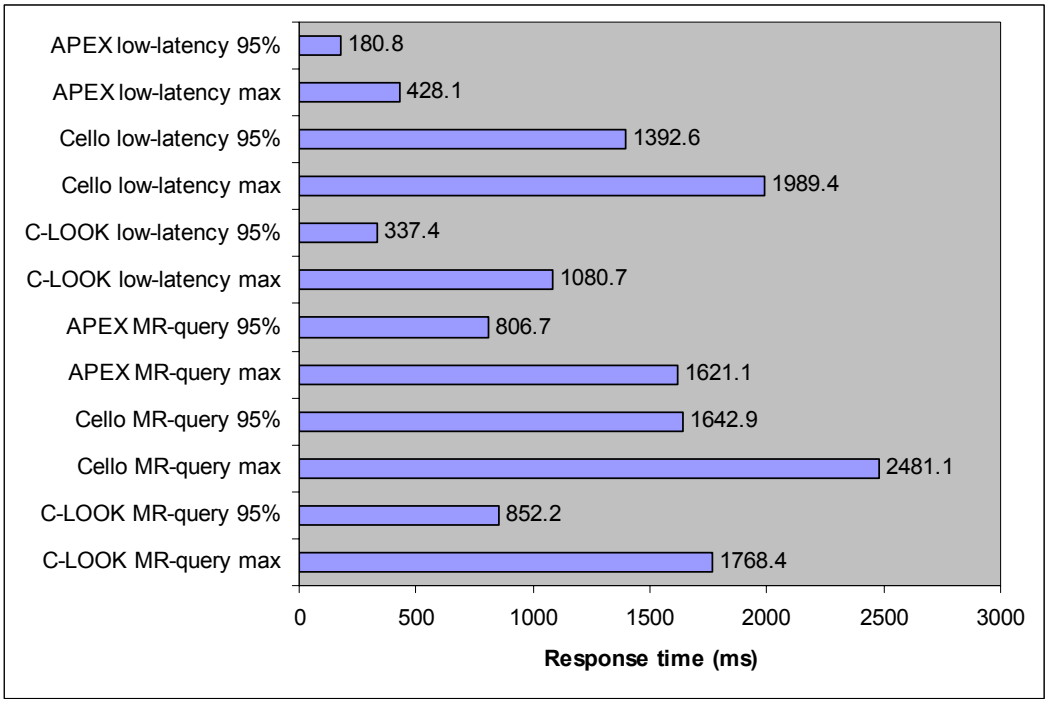

Figure 8-12: Response times for low-latency and MR-query disk requests, with four real-time clients

8.6 Analysis of the Results

As mentioned in Section 8.1, the purposes of the simulations are both to provide proof-of-concept for APEX, and to function as partial proof of the four claims presented in Section 1.3.

From the simulation results, it is clear that the scheduling principles of APEX work in practice, and that the scheduler behaves as expected. During light-load situations, the scheduler works as a FCFS/EDF scheduler, and when the load starts to increase, batches are being formed. We see that the token bucket principle works well, providing each queue with the reserved bandwidth as a minimum, and at the same time limiting the number of requests selected from each queue. In general, the service types are realized as expected. Finally, it is clear that the work-conservation provides very effective redistribution of unused bandwidth.

8.6.1 Proof of Claims

Claim 1: MMDBMS integration

This claim requires that we show that APEX can handle MMDBMS workload efficiently. We consider the workload of experiment 4 to be representative as MMDBMS workload, and if we look at the results from this experiment, it is clear that APEX is able to handle this workload better than Cello and C-LOOK.

Given that metadata retrieval and multimedia authoring queries generate bursts of disk requests, that the presentations contain mixes of (variable bit-rate) multimedia objects, and that the query mix is constantly changing, the overall disk workload is highly and rapidly varying. Thus, being able to apply available bandwidth where it is needed, and doing so without efficiency loss, is very important, and thanks to its very efficient work-conservation, APEX is able to achieve this.

Claim 2: Versatility of APEX

This claim requires us to demonstrate the ability of APEX to handle very diverse workload types efficiently. The workload we have used in our simulations includes four different types. Although this is not an exhaustive selection, the four workload types are very diverse, and have been mixed in different combinations. APEX has proven to handle all of these workload types very well, and this fact supports our claim.

Claim 3: QoS-support and high Bandwidth Utilization

Our simulations show that APEX, compared to Cello, consistently provides both better throughput and lower response times for best-effort services, while at the same time providing the same QoS-level for guaranteed services.

Comparing APEX to C-LOOK, we see that the difference is relatively small. C-LOOK performs slightly better with respect to response times of MR-query disk requests, but tends to have higher variability. For throughput of the checkout-operation and response times of low-latency requests, C-LOOK consistently performs poorer than APEX, and in addition, some of the experiments lead to a large number of deadline violations for real-time disk requests.

From the simulations, we conclude that APEX comes very close to C-LOOK, with respect to bandwidth utilization, i.e., APEX achieves a low amount of non-productive positioning work, and thereby increases performance. At the same time, our scheduler is

able to provide a large set of service types, as well as providing QoS-guarantees similar to Cello. This means that the cost of providing QoS-support in APEX is low. Thus, we consider that the simulations prove the superior combination of QoS-support and high utilization of disk bandwidth offered by APEX (claim 3).

Claim 4: Disk Scheduling is necessary for QoS-support in the storage subsystem

From the simulations we see that C-LOOK, due to its high bandwidth utilization performs well, also with respect to QoS-support; the deadline violations are relatively modest in size. Thus, depending on the latency requirements of the user, and the buffering capabilities in the client hardware, these violations could be masked.

However, in the checkout experiments, we found it necessary to limit the arrival rate of checkout disk requests to C-LOOK, in order to limit the queue length. Otherwise, several hundred requests could be pending, leading to unrealistic working conditions for the disk. During experiments with different maximum values, we discovered that this parameter has a significant impact on the behavior of the scheduler. The results of experiments 2 and 4 are based on a maximum of 25 requests in the queue. If we double this maximum queue length to 50, the checkout throughput increase considerably, but so does the response time for MR-query disk requests, as well as the number of deadline violations.

These results indicate that, although a performance-oriented scheduler like C-LOOK in some cases could meet our QoS-requirements using “brute force”, its behavior varies considerably with varying workload mix, and this unpredictability represents a considerable problem for a system that offers QoS-guarantees.

8.6.2 Observations

The batch building principle seems to work very well. Submitting batches of disk requests provides the disk with good working conditions, since the non-productive disk work (i.e., head positioning) can be amortized over many disk requests, and this is reflected by the simulation results.

However, in our simulations, we found it necessary to limit the maximum batch size. It turned out that during heavy loads, the batches could consist of more than 80 requests, and this proves to be detrimental for the response times of non-real-time requests that are served late in the batch, at the same time as the efficiency gain is relatively limited. By experimenting with different maximum sizes, we find that, with the configuration used in these experiments, limiting the batch size to 50 requests provides a good compromise

between efficiency and response times. In all the experiments, we have therefore used a maximum batch size of 50 requests.

8.6.3 Cello and Over-Provisioning

Our analyses and simulations show the following characteristics of Cello, compared to APEX:

- *Poorer efficiency*: In general, Cello orders disk requests in a less optimal way than APEX, and, therefore, the overall efficiency is reduced.
- *Poor efficiency during work-conservation*: During work-conservation, Cello does not take the position of the requested data into consideration. Thus, efficiency becomes poor during this phase.
- *Higher sensitivity to incorrect bandwidth allocation*: This is strongly related to the work-conservation problem.
- *Lack of low-latency service*: Cello does not provide a low-latency service, apart from trying to serve interactive best-effort requests as soon as possible. Our simulations show that such requests experience considerably longer response times than requests using the low-latency service of APEX.

It is important to investigate whether the issues described above will change by allocating more resources to Cello, i.e., compensating by adding more disks.

First of all, it is obvious that adding disks will not increase the efficiency of Cello, neither in the regular phase, nor in the work-conserving phase. On the contrary, as explained in Section 4.5, increasing the number of disks tends to reduce the overall efficiency, due to an increased amount of non-productive work (i.e., head positioning). Our only option is to use separated storage, such that each disk receives request of only one type. However, as we showed in Section 4.5, this is not an alternative.

In [83], the usage of Cello in a multi-disk context is described. From this description, it is clear that the storage subsystem is realized using one instance of Cello per disk, an approach that is also used for the Prism scheduler [104]. This scheme implies that, the computational overhead of using Cello increases proportionally with the number of disks. Thus, if we increase the number of disks in order to compensate for lacking efficiency, we also increase the computational overhead. In addition, the overhead caused by managing data placement may also increase, as described in Section 4.5.

Consequently, we conclude that adding more disks to a storage subsystem using Cello as disk scheduler cannot increase the efficiency, but it could compensate for the lack of efficiency, by increasing raw bandwidth. The cost of doing so is increased resource (i.e., processor) usage due to more instances of Cello. The lack of a proper low-latency service cannot be alleviated.

8.6.4 Limitations and Open Issues

Though the DiskSim simulation environment provides a very detailed model of the storage subsystem, using simulations still has limitations. First of all, the workload is static, in the sense that it is generated from a trace file. In real life, the arrival of disk request n+1 is often dependent on the finishing time request n. Thus, if block n is delayed, then block n+1 will not be requested until block n has been retrieved. However, with a trace file this "feedback loop" is lost, and block n+1 is requested, even if block n has still not been retrieved from disk.

The consequences of this are difficult to calculate, but it is reasonable to assume that one effect would be that using trace files can lead to a too high arrival rate of requests into the disk scheduler. On the other hand, the static workload is equal for all three schedulers (except for checkout-request for C-LOOK), so at least the results should be comparable.

The use of simulated time also caused some problems for the implementation of the disk schedulers. As mentioned earlier, all three schedulers are multi-threaded, where each thread is responsible for a separate part of the scheduler. In a real-life implementation, these threads will run concurrently, and the shared variables protected by semaphores, or the like. However, the use of simulated time, held by a global variable, made it difficult to avoid race conditions. Instead, we let the threads run round robin, such that there were never more than one active thread. Each time DiskSim updated the simulated time, one "round" of threads were run. This should not have any significant impact on the simulation results, and again, the conditions are equal for all three schedulers.

Unfortunately, limited time has prevented us from performing all the simulations that we would like to do, and this is perhaps the most serious shortcoming. Specifically, we would like to try out other settings of the simulation parameters; other disk layouts, round times, disk block sizes, and disk types. Such simulations would not necessarily provide new information, but they would increase the reliability of the results already presented, by showing that APEX is not dependent on one particular configuration to function.

As explained in Sub-section 8.2.2, we were forced to submit disk requests one by one to DiskSim, instead of submitting whole batches, which is the intention of APEX. This means that, while we still get the advantage of sorting the requests in a batch in SCAN order, we reduce the potential performance gains of the disk-internal scheduling. However, this fact is in the disfavor of APEX, and we expect APEX to perform even better in a real-life implementation. This is supported by results from experiments performed in [101], where disk-internal scheduling achieves up to 30% higher throughput than host-based scheduling.

This last issue is also related to claim 4, stating that disk scheduling is necessary for QoS-support in the storage subsystem. We have used C-LOOK as a substitute for the behavior of a modern disk, which is reasonable, since the internal scheduling of modern disks relies on performance-oriented algorithms like elevator or SPTF (shortest positioning time first). However, our approach means that some of the features of a modern disk are not fully utilized, since only one request is submitted at a time. Thus, in a real-life implementation, one could expect the disk-internal scheduling to support slightly better, but also even more unpredictable.

8.7 Summary

In this chapter, we have presented the simulation environment used in our evaluation of APEX. We have described how the three schedulers have been configured for use in this environment, and we have presented the experiments performed.

In addition, we have presented the results of the simulations, and analyzed these. Finally, we made a critical assessment of the results, and discussed the limitations of using a simulation environment.

With respect to our claims, presented in Section 1.3, we reviewed these in Sub-section 8.6.1, where we showed how this chapter involved all four claims.

In the next chapter, we present our conclusions. We do a final review of the four claims, make a critical assessment of our performance evaluation, and present possible directions for future work.

Chapter 9

Conclusion

In this thesis, we have addressed the challenges of disk scheduling for a MMDBMS. In this chapter, we summarize the contributions of our work. We present a critical review of the four claims made in Section 1.3, and we perform a critical assessment of our results. Finally, we address directions of future research.

9.1 Summary

We have analyzed the requirements of a LoD-system based on a MMDBMS, and used this analysis as a basis for designing a disk scheduling framework that is able to handle the workload of such a system. The key features of our framework are:

- *Dynamic queue management*, meaning that APEX can create and remove queues as needed, and update the bandwidth reservations according to the requirements of the transactions using the queue. Thus, at any given time, only the queues that are needed are actually instantiated, and this contributes to keep the overhead of the scheduler low. In addition, the queue management itself is very simple, requiring little computational resources.
- An *extended token bucket model*, which serves two purposes: the token rate parameter ensures that each queue receives the bandwidth it is entitled to, and the bucket depth parameter prevents queues from starvation, by limiting the per-queue bursts.
- A *batch building principle*, which submits batches of disk requests to the disk driver. In essence, this principle introduces "split-level" scheduling: the higher-level scheduling, performed by APEX, distributes bandwidth and ensures QoS, working with a temporal

granularity based on the needs of the application (i.e., the latency requirements). The lower-level scheduling is performed by the disk-internal scheduler, based on information that is unknown outside of the disk, since it is hidden by the disk itself. Furthermore, submitting many requests at once implies that the unproductive positioning work of the disk can be amortized over multiple requests.

- A *work-conservation feature* that re-distributes unused disk bandwidth without loss of disk efficiency. Thus, during the normal phase, APEX just ensures that each queue gets its rightful share of disk bandwidth, and then during the work-conservation phase, the unused bandwidth is distributed. The result of the two phases is one batch, meaning that no disk efficiency is lost.

We have verified that the scheduling principles work, by implementing our disk scheduling framework in a simulation environment. In addition, we have shown that, despite its advanced functionality, APEX does not represent higher computational complexity than other mixed-media disk schedulers.

9.2 Review of Claims

In Section 1.3, we made four claims about disk scheduling in general and APEX in particular. We now review these claims, based on the work presented in this thesis.

Claim 1

It is possible to design, implement, and integrate a disk scheduler in a MMDBMS, in such a way that it can utilize metadata from the MMDBMS to optimize the scheduling of disk requests.

We started by describing our LoD-scenario, and we analyzed the system architecture with special emphasize on the MMDBMS. Next, we analyzed the requirements of the LoD-system, and showed why existing disk schedulers are insufficient. We also described what information is available from the MMDBMS. Furthermore, we gave a detailed description of the design and implementation of APEX, and showed how our disk scheduling framework is able to use information from the MMDBMS.

To verify this theoretical proof of the claim, we ran simulations, and these confirmed that in the intersection between performance, QoS-support, and multiple service types, APEX performs better than both Cello (representing the class of mixed-media schedulers) and C-LOOK (representing the class of performance-oriented schedulers).

Claim 2

APEX is a highly configurable disk scheduling framework that can be used in a wide variety of contexts.

Since the scenario of our work has been an MMDBMS-based LoD-system, it is natural that we have focused on the requirements of such as system. However, in Section 5.2, we introduced the four scheduler characterization parameters, and used these to describe the requirements to the disk scheduler for other contexts. APEX is designed to support all relevant combinations of these four parameters; thus, it can, in principle, be used in any context that can describe its requirements using these four parameters.

Furthermore, we showed how the modularity of APEX allows it to be tailored for different contexts. The configuration of APEX used in our thesis includes all components, while in other, less dynamic contexts, components that are not needed can be removed.

In addition, the workload used in our simulations is very diverse, and therefore representative for a number of different contexts. As the simulations show, our configuration of APEX handles the entire workload very well, and this confirms the versatility of APEX.

Finally, it should be noted that the batch building principle is common, regardless of the configuration. Thus, in many cases, adding a new configuration is a question of implementing a new queue type. Examples of such types are priorities and request dropping (e.g., dropping requests that will obviously miss their deadlines). In other words, regardless of the type of service offered by the queue, the batch building principle is always used when selecting requests for service, and we have shown that this principle works.

Claim 3

APEX offers a superior combination of QoS-support and high utilization of disk bandwidth.

By comparing existing QoS-aware disk schedulers with the requirements of our LoD-system and the characteristics of modern disks, we pointed out the weaknesses of these schedulers. It is clear that the cost of supporting QoS is relatively high, and the work-conserving facilities tend to reduce disk efficiency.

With its batch principle, APEX provides the disk with very favorable working conditions. The disk is given freedom to organize the requests according to its internal

scheduling algorithm, and the positioning operations can be amortized over several disk requests. In addition, the work-conservation facility that works without efficiency loss ensures a very effective redistribution of unused bandwidth. Finally, the extended token bucket principle allows a very accurate control of the distribution of disk bandwidth.

The simulations we have performed, confirm that these principles work in practice. APEX achieves higher throughput and lower response times than Cello, and when compared with a pure performance-oriented scheduler like C-LOOK, the simulation results indicate that the cost of providing QoS-support in APEX is low.

Claim 4

Disk scheduling is necessary for providing QoS-support in the storage subsystem, but existing QoS-aware disk schedulers do not exploit the capabilities of modern disks.

In Section 4.5, we discussed why integrated storage of the multimedia data is necessary. Such storage requires the ability to differentiate between different service types, and protect the different transactions from each other.

We also described the characteristics of modern disks and controllers. We discussed how the intelligence of these components makes them unpredictable. In addition, the simulations showed that the workload mix has a considerable impact on the behavior of the disk (i.e., the C-LOOK algorithm). Thus, it is difficult to know the order in which the requests will be served, and this problem increases by the usage of integrated storage. Consequently, trying to meet the need for multiple service types with a "brute force" approach is difficult; the lack of isolation between different service types makes it impossible to provide QoS-guarantees without considerable restrictions on all types of traffic to the storage subsystem.

Finally, many QoS-aware disk scheduling algorithms, especially in the class of mixed-media schedulers (including Cello), are dependent on detailed knowledge of the disk behavior. However, the intelligence of modern disks and controllers once again cause problems, since the internal organization and behavior of the disk is hidden. APEX, on the other hand, takes advantage of the capabilities of modern disks, leaving the final ordering of the requests within a batch to the disk itself.

9.3 Critical Assessments

In our experimental evaluation of APEX, we have relied on a simulation environment. Although this environment provides very detailed modeling of disk, controller, and disk driver, there are several factors, both concerning the disk simulator itself and our usage of it, which could affect the evaluation. We discussed these issues in Sub-section 8.6.4, and we refer to this section for details. However, there are also other issues that may affect our results, and we discuss these below.

9.3.1 Workload Quality

Since we do not have a running MMDBMS, it is difficult to verify how representative the workload traces are, and in particular for metadata retrieval queries. The traces we have used are based on measurements performed on real DBMSs, in [9]. However, in this work, relational DBMSs were used, and the measured workload is not necessarily representative for other types of DBMSs. On the other hand, the workload is identical for all three disk schedulers, thus, it does provide a basis for comparison of the disk schedulers.

9.3.2 Workload Differences

As mentioned in Sub-section 8.6.1, it was necessary to limit the maximum queue length in C-LOOK. Otherwise, the queue could contain several hundred requests, which is clearly unrealistic; this scheduler orders request purely on the basis of data placement on disk, and the extremely large SCAN-schedules that were formed, resulted in excessively long response times.

Since this queue length limitation has only been done for C-LOOK, the resulting checkout-workload is not 100% identical for the three schedulers. However, since Cello and APEX consider bandwidth distribution before taking data placement into account, the long queue of checkout-requests has a much smaller impact in these schedulers.

9.3.3 Multiple Disks

We have only performed simulations with a single disk. In a real LoD-system, it is reasonable to expect multiple disks to be used, either in the form of a disk array (e.g., a RAID-system), or as a set of individual disks. Currently, APEX has not been tested in a

multi-disk environment, but as we will come back to in the next section, this is part of our future work.

9.3.4 Simulation Parameters

As we have mentioned earlier, the set of simulations performed so far is relatively limited, and we have focused on MMDBMS workload. By running simulations with other types of workload, as well as other combinations of workload, we would improve the basis of comparison between C-LOOK, Cello, and APEX.

In addition, running simulations with different parameter settings, i.e., disk layout, round time, disk block size, and disk type, would improve the understanding of the behavior of APEX. Furthermore, varying the parameter settings would also provide important information about Cello and C-LOOK. In particular, the use of C-LOOK in combination with extent-based or contiguous data placement is of interest. Since data placement is the only factor that C-LOOK takes into consideration when sorting requests, we expect the workload-dependent behavior revealed in the simulations to be reinforced. Thus, the C-LOOK scheduling algorithm, as well as the SCAN-algorithms in general, could prove to be even less suited for QoS-support using extent-based or contiguous data placement.

9.4 Future Work

In this thesis, we have presented our work on the APEX disk scheduling framework. Through simulations we have shown that the scheduling principles work, and that APEX achieves high disk efficiency under very varying workload conditions. However, there are still a lot of open questions and unsolved tasks. In the following sub-sections, we present what we consider the most important remaining work.

9.4.1 Parameter Settings

With respect to the more theoretical aspects of APEX, there are three issues concerning the configuration of APEX that would benefit from further research:

- *Estimated service time* t_{es}: In our simulation, we have initially set this parameter manually, and then used a simple, automatic fine-adjustment, which compares the estimated finishing time of each batch with the corresponding actual finishing time.

However, more research should be performed in order to find the optimal setting of this parameter. For instance, it is an open question whether this parameter has a fixed, optimal value with a given configuration of the external parameters (i.e., data placement, round time, disk block size, and disk type), or if the workload mix also has an influence, such that t_{es} must be constantly monitored and adjusted. If there is a fixed value, then it would also be of interest to work out the theoretical relationship between t_{es} and the external parameters.

- *Maximum batch size*: As explained in Sub-section 8.6.2, we found that APEX performed better if the maximum batch size was limited. In our simulation, we found the best value for this maximum batch size by using a "trial and error" approach. However, a better theoretically foundation should be worked out, where the maximum size is determined based on data placement, round time, disk block size, and disk type. In addition, there is a possibility that this value is dependent on the workload mix, and therefore should be constantly monitored and adjusted. For instance, if the LoD-system only serves multimedia playback queries, we can allow a very large batch size, but as soon as other queries, such as metadata retrieval queries, are started, the batch size must be reduced. Finally, the size of the disk-internal scheduling queue should also be taken into consideration, since there probably is little performance gain in submitting batches larger than what this queue can hold.
- *Token bucket parameters*: When determining the required bandwidth for the different queues, we turned off the work-conservation, and set the token rate and bucket depth as low as possible without deadlines being violated. However, it is necessary to work out a theoretical relationship between the bandwidth requirement of the multimedia objects stored in the MMDBMS, and the required token rate and bucket depth. For constant bit-rate multimedia objects this is trivial, but for variable bit-rate objects, it is more of a challenge. In addition, the type of guarantee offered also affects the setting of these parameters. For instance, if deterministic real-time guarantee is offered, the token rate must probably be set higher than if a statistical guarantees are offered. Finally, the setting of token rate and bucket depth is done on a per-queue basis. Thus, when several multimedia playback queries share a queue, the setting of these parameters is further complicated.

9.4.2 Multiple Disks

In this thesis, we have only focused on single-disk storage subsystems. However, many servers designed for delivery of multimedia data are based on multiple disks. This is often necessary in order to meet both space and bandwidth requirements.

As explained earlier, both Cello and PRISM are able to handle multiple disks, by running one instance of the scheduler for every disk. This solution is also possible for APEX, and since each instance of APEX in such a case only schedules one disk, there is, in principle, little difference from the single-disk scenario we have used.

However, while Cello and PRISM require detailed knowledge about the behavior of the disk, and therefore cannot handle multiple disks per scheduler, APEX is largely independent of knowledge about disk behavior. Consequently, it is possible that one instance of APEX can handle multiple disks, provided that the disks appear as a single storage unit, such as in a RAID system [69]. The latter condition is necessary, since otherwise, load balancing becomes an issue.

If a single instance of APEX can handle an array of disks, this would imply considerably lower computational overhead, compared to the per-disk approach used by Cello and PRISM. Thus, this issue should definitely be subject to more research.

9.4.3 Round-Less Version of APEX

As mentioned in Sub-section 6.4.2, we originally designed APEX as a pure deadline-based scheduler. We found that this approach was less suited for our scenario than the round-based version, and abandoned this solution. However, for other, non-multimedia, application scenarios, the round-less version of APEX may be of interest.

In Sub-section 5.4.2, we showed that existing real-time disk scheduling algorithms generally provide only one service type, and the round-less schedulers, such as EDF, often suffer from poor disk utilization. Thus, if real-time disk service is required, possibly in combination with other service type requirements, and a round-based approach is not an option, it could be worthwhile doing further research on the round-less version of APEX.

9.4.4 Admission Control

APEX relies on an estimated time for service of a disk request, t_{es}, when computing the batch sizes. This estimated time influences the exploitable capacity of the disk: A larger t_{es} means that fewer requests can be served per time unit; thus, we get a lower throughput.

Furthermore, APEX offers levels of QoS-guarantees that require admission control, and this admission control must also rely on t_{es}. In our work, we have assumed admission control as described in [90], but further investigations of how the admission control can be tailored to the characteristics of APEX should be performed.

9.4.5 Implementation

When evaluating APEX using simulations, we are subject to the limitations of the simulation environment. Thus, the natural step beyond simulations is to implement APEX in a real system. Currently, there is ongoing work on implementing a MMDBMS-based LoD-system in the OMODIS-LoD[10] project [100], and in the longer term, incorporating APEX in this system is a possibility.

In addition, we are considering an implementation of APEX in a Media-on-Demand server being developed within our research group. This would be an implementation in the kernel of the operating system, below the file system and buffer cache. In this version, the queue requests are performed during the `OPEN` system call, where we add parameters for specifying the type of service required, as well as the amount of bandwidth to be reserved. The queue is then associated with the file descriptor used, instead of using explicit queue IDs. Thus, two file descriptors pointing to the same file may provide different service types, if they are associated with different queues in APEX.

9.4.6 "Transaction-Oriented" Service Model

In our work, we have assumed a "service-oriented" model where the service types and levels are realized as a relatively small set of queues. All transactions requiring a particular service then share the APEX-queue that offers this service type and level.

However, APEX is able to support an arbitrary number of queues, and this opens the possibility of a "transaction-oriented" model, where each transaction is assigned a separate

---

[10] The OMODIS LoD Project is funded by the Norwegian Research Council, Distributed IT Systems (DITS) program to UiO/Ifi, 2000-2003.

queue, and the queue is removed when the transaction ends. This approach has not been much discussed in the literature, but one approach that provides something similar, is the "User Safe Disk" (USD) [6].

This solution would imply a somewhat increased queue management overhead, but on the other hand, it would provide a full isolation between the transactions. In a MMDBMS context, where the MMDBMS itself has full control of the transactions, this is perhaps not necessary, but in other application areas, this solution enables APEX to provide a very good protection against clients or applications that do not behave.

It is our view that it could be worthwhile investigating both the "transaction-oriented" model, as well as a hybrid solution, where both models are in use (APEX is able to support this as well). First of all, the utility value of the model should be assessed for different application areas. Then, it is necessary to assess the balance between increased cost, in the form of queue management, and the advantages, in the form of increased isolation.

9.5 Final Remarks

In this thesis, we have presented APEX, a versatile and highly configurable disk scheduling framework. We have shown, by means of simulations, that APEX works as intended, and that it achieves better disk utilization and provides more service types than other disk schedulers.

However, as presented in this thesis, APEX is still very much in an early prototype stage, and as described in Section 9.4, there are a number of possibilities for further research and development. Consequently, we believe that APEX has room for further performance improvement, and that our disk scheduling framework will prove valuable in a number of application areas.

Appendix A

DiskSim Parameters

DiskSim is highly configurable, offering a large number of parameters that can be adjusted. In our simulations, we have chosen to use one of the pre-set configurations included in the DiskSim package.

DiskSim relies on two configuration files, which together specify all the necessary parameters. The first file specifies the configuration of the disk that is used, while the second specifies the interconnection of the different components, i.e., disk, buses, controllers and disk driver. Below, we list the content of the configuration files used in our experiments.

Disk Configuration

```
Disk brand name: QUANTUM_TORNADO
Access time (in msecs):             -2.0
Seek time (in msecs):               -5.0
Single cylinder seek time:          1.24500
Average seek time:                  atlas10k.seek
Full strobe seek time:              10.82800
Add. write settling delay:          0.0000
HPL seek equation values:           0 0 0 0 0 0
First 10 seek times:                1.25 1.16 1.16 1.38 1.36 1.44 1.49 1.64 1.51
1.53
Head switch time:                   0.17600
Rotation speed (in rpms):           10025
Percent error in rpms:              0.0
Number of data surfaces:            6
Number of cylinders:                10022
Blocks per disk:                    17938986
Per-request overhead time:          0.000000
Time scale for overheads:           0.0
Bulk sector transfer time:          0.00000
Hold bus entire read xfer:          0
Hold bus entire write xfer:         0
Allow almost read hits:             0
Allow sneaky full read hits:        0
Allow sneaky partial read hits:     0
Allow sneaky intermediate read hits:0
```

```
Allow read hits on write data:      1
Allow write prebuffering:           0
Preseeking level:                   0
Never disconnect:                   0
Print stats for disk:               0
Avg sectors per cylinder:           1786
Max queue length at disk:           1
Scheduling policy:                  1
Cylinder mapping strategy:          0
Write initiation delay:             0.0
Read initiation delay:              0.0
Sequential stream scheme:           0
Maximum concat size:                0
Overlapping request scheme:         0
Sequential stream diff maximum:     0
Scheduling timeout scheme:          0
Timeout time/weight:                0
Timeout scheduling:                 0
Scheduling priority scheme:         0
Priority scheduling:                0
Number of buffer segments:          10
Maximum number of write segments:   1
Segment size (in blks):             374
Use separate write segment:         0
Low (write) water mark:             0.00
High (read) water mark:             0.00
Set watermark by reqsize:           1
Calc sector by sector:              1
Enable caching in buffer:           0
Buffer continuous read:             0
Minimum read-ahead (blks):          0
Maximum read-ahead (blks):          354
Read-ahead over requested:          0
Read-ahead on idle hit:             0
Read any free blocks:               0
Fast write level:                   0
Immediate buffer read:              0
Immediate buffer write:             0
Combine seq writes:                 1
Stop prefetch in sector:            0
Disconnect write if seek:           0
Write hit stop prefetch:            1
Read directly to buffer:            1
Immed transfer partial hit:         1
Read hit over. after read:          0.00000
Read hit over. after write:         0.00000
Read miss over. after read:         0.00000
Read miss over. after write:        0.00000
Write hit over. after read:         0.00000
Write hit over. after write:        0.00000
Write miss over. after read:        0.00000
Write miss over. after write:       0.00000
Read completion overhead:           0.00000
Write completion overhead:          0.00000
Data preparation overhead:          0.00000
First reselect overhead:            0.00000
Other reselect overhead:            0.00000
Read disconnect afterread:          0.00000
Read disconnect afterwrite:         0.00000
Write disconnect overhead:          0.00000
Extra write disconnect:             0
Extradisc command overhead:         0.00000
Extradisc disconnect overhead:      0.00000
Extradisc inter-disconnect delay:   0.00000
Extradisc 2nd disconnect overhead:  0.00000
Extradisc seek delta:               0.00000
Minimum seek delay:                 0.00000
LBN-to-PBN mapping scheme:          1
```

```
Sparing scheme used:              9
Rangesize for sparing:            1
Number of bands:                  24
Band #1
First cylinder number:            0
Last cylinder number:             432
Blocks per track:                 334
Offset of first block:            0.000000
Empty space at zone front:        0
Skew for track switch:            63.000000
Skew for cylinder switch:         113.000000
Number of spares:                 89
Number of slips:                  0
Number of defects:                0
Band #2
First cylinder number:            433
Last cylinder number:             848
Blocks per track:                 334
Offset of first block:            0.000000
Empty space at zone front:        0
Skew for track switch:            63.000000
Skew for cylinder switch:         113.000000
Number of spares:                 84
Number of slips:                  0
Number of defects:                0
Band #3
First cylinder number:            849
Last cylinder number:             1264
Blocks per track:                 334
Offset of first block:            0.000000
Empty space at zone front:        0
Skew for track switch:            63.000000
Skew for cylinder switch:         113.000000
Number of spares:                 87
Number of slips:                  0
Number of defects:                0
Band #4
First cylinder number:            1265
Last cylinder number:             1688
Blocks per track:                 324
Offset of first block:            0.000000
Empty space at zone front:        0
Skew for track switch:            63.000000
Skew for cylinder switch:         101.000000
Number of spares:                 85
Number of slips:                  0
Number of defects:                0
Band #5
First cylinder number:            1689
Last cylinder number:             2104
Blocks per track:                 324
Offset of first block:            0.000000
Empty space at zone front:        0
Skew for track switch:            63.000000
Skew for cylinder switch:         101.000000
Number of spares:                 84
Number of slips:                  0
Number of defects:                0
Band #6
First cylinder number:            2105
Last cylinder number:             2520
Blocks per track:                 324
Offset of first block:            0.000000
Empty space at zone front:        0
Skew for track switch:            63.000000
Skew for cylinder switch:         101.000000
Number of spares:                 85
Number of slips:                  0
```

```
Number of defects:                0
Band #7
First cylinder number:            2521
Last cylinder number:             2936
Blocks per track:                 324
Offset of first block:            0.000000
Empty space at zone front:        0
Skew for track switch:            62.000000
Skew for cylinder switch:         101.000000
Number of spares:                 84
Number of slips:                  0
Number of defects:                0
Band #8
First cylinder number:            2937
Last cylinder number:             3360
Blocks per track:                 324
Offset of first block:            0.000000
Empty space at zone front:        0
Skew for track switch:            61.000000
Skew for cylinder switch:         98.000000
Number of spares:                 92
Number of slips:                  0
Number of defects:                0
Band #9
First cylinder number:            3361
Last cylinder number:             3776
Blocks per track:                 324
Offset of first block:            0.000000
Empty space at zone front:        0
Skew for track switch:            61.000000
Skew for cylinder switch:         98.000000
Number of spares:                 87
Number of slips:                  0
Number of defects:                0
Band #10
First cylinder number:            3777
Last cylinder number:             4192
Blocks per track:                 324
Offset of first block:            0.000000
Empty space at zone front:        0
Skew for track switch:            61.000000
Skew for cylinder switch:         98.000000
Number of spares:                 85
Number of slips:                  0
Number of defects:                0
Band #11
First cylinder number:            4193
Last cylinder number:             4608
Blocks per track:                 324
Offset of first block:            0.000000
Empty space at zone front:        0
Skew for track switch:            61.000000
Skew for cylinder switch:         98.000000
Number of spares:                 85
Number of slips:                  0
Number of defects:                0
Band #12
First cylinder number:            4609
Last cylinder number:             5032
Blocks per track:                 306
Offset of first block:            0.000000
Empty space at zone front:        0
Skew for track switch:            58.000000
Skew for cylinder switch:         98.000000
Number of spares:                 79
Number of slips:                  0
Number of defects:                0
Band #13
```

```
First cylinder number:              5033
Last cylinder number:               5448
Blocks per track:                   306
Offset of first block:              0.000000
Empty space at zone front:          0
Skew for track switch:              58.000000
Skew for cylinder switch:           98.000000
Number of spares:                   89
Number of slips:                    0
Number of defects:                  0
Band #14
First cylinder number:              5449
Last cylinder number:               5864
Blocks per track:                   302
Offset of first block:              0.000000
Empty space at zone front:          0
Skew for track switch:              57.000000
Skew for cylinder switch:           92.000000
Number of spares:                   88
Number of slips:                    0
Number of defects:                  0
Band #15
First cylinder number:              5865
Last cylinder number:               6280
Blocks per track:                   293
Offset of first block:              0.000000
Empty space at zone front:          0
Skew for track switch:              56.000000
Skew for cylinder switch:           91.000000
Number of spares:                   73
Number of slips:                    0
Number of defects:                  0
Band #16
First cylinder number:              6281
Last cylinder number:               6704
Blocks per track:                   288
Offset of first block:              0.000000
Empty space at zone front:          0
Skew for track switch:              55.000000
Skew for cylinder switch:           91.000000
Number of spares:                   85
Number of slips:                    0
Number of defects:                  0
Band #17
First cylinder number:              6705
Last cylinder number:               7120
Blocks per track:                   288
Offset of first block:              0.000000
Empty space at zone front:          0
Skew for track switch:              55.000000
Skew for cylinder switch:           88.000000
Number of spares:                   72
Number of slips:                    0
Number of defects:                  0
Band #18
First cylinder number:              7121
Last cylinder number:               7536
Blocks per track:                   280
Offset of first block:              0.000000
Empty space at zone front:          0
Skew for track switch:              54.000000
Skew for cylinder switch:           88.000000
Number of spares:                   77
Number of slips:                    0
Number of defects:                  0
Band #19
First cylinder number:              7537
Last cylinder number:               7952
```

```
Blocks per track:                   270
Offset of first block:              0.000000
Empty space at zone front:          0
Skew for track switch:              51.000000
Skew for cylinder switch:           87.000000
Number of spares:                   66
Number of slips:                    0
Number of defects:                  0
Band #20
First cylinder number:              7953
Last cylinder number:               8376
Blocks per track:                   262
Offset of first block:              0.000000
Empty space at zone front:          0
Skew for track switch:              51.000000
Skew for cylinder switch:           83.000000
Number of spares:                   66
Number of slips:                    0
Number of defects:                  0
Band #21
First cylinder number:              8377
Last cylinder number:               8792
Blocks per track:                   255
Offset of first block:              0.000000
Empty space at zone front:          0
Skew for track switch:              48.000000
Skew for cylinder switch:           83.000000
Number of spares:                   66
Number of slips:                    0
Number of defects:                  0
Band #22
First cylinder number:              8793
Last cylinder number:               9208
Blocks per track:                   246
Offset of first block:              0.000000
Empty space at zone front:          0
Skew for track switch:              47.000000
Skew for cylinder switch:           75.000000
Number of spares:                   60
Number of slips:                    0
Number of defects:                  0
Band #23
First cylinder number:              9209
Last cylinder number:               9624
Blocks per track:                   237
Offset of first block:              0.000000
Empty space at zone front:          0
Skew for track switch:              45.000000
Skew for cylinder switch:           75.000000
Number of spares:                   60
Number of slips:                    0
Number of defects:                  0
Band #24
First cylinder number:              9625
Last cylinder number:               10021
Blocks per track:                   229
Offset of first block:              0.000000
Empty space at zone front:          0
Skew for track switch:              44.000000
Skew for cylinder switch:           75.000000
Number of spares:                   716
Number of slips:                    0
Number of defects:                  0
```

```
Byte Order (Endian):               Little
Init Seed:                          42
Real Seed:                          42
Statistic warm-up period:           0.0 seconds
Checkpoint to null every:           0.0 seconds
Stat (dist) definition file:        statdefs
Output file for trace of I/O requests simulated:ioreq

I/O Subsystem Input Parameters
------------------------------

PRINTED I/O SUBSYSTEM STATISTICS

Print driver size stats?            1
Print driver locality stats?        0
Print driver blocking stats?        0
Print driver interference stats?    0
Print driver queue stats?           1
Print driver crit stats?            1
Print driver idle stats?            1
Print driver intarr stats?          1
Print driver streak stats?          1
Print driver stamp stats?           1
Print driver per-device stats?      1
Print bus idle stats?               1
Print bus arbwait stats?            1
Print controller cache stats?       1
Print controller size stats?        1
Print controller locality stats?    1
Print controller blocking stats?    1
Print controller interference stats? 1
Print controller queue stats?       1
Print controller crit stats?        1
Print controller idle stats?        1
Print controller intarr stats?      1
Print controller streak stats?      1
Print controller stamp stats?       1
Print controller per-device stats?  1
Print device queue stats?           1
Print device crit stats?            0
Print device idle stats?            1
Print device intarr stats?          0
Print device size stats?            1
Print device seek stats?            1
Print device latency stats?         1
Print device xfer stats?            1
Print device acctime stats?         1
Print device interfere stats?       0
Print device buffer stats?          1

GENERAL I/O SUBSYSTEM PARAMETERS

I/O Trace Time Scale:               1.0
Number of I/O Mappings:             0

COMPONENT SPECIFICATIONS

Number of device drivers:          1

Device Driver Spec #1
# drivers with Spec:                1
Device driver type:                 1
Constant access time:               0.0
Scheduling policy:                  1
Cylinder mapping strategy:          1
```

```
Write initiation delay:             0.0
Read initiation delay:              0.0
Sequential stream scheme:           0
Maximum concat size:                128
Overlapping request scheme:         0
Sequential stream diff maximum:     0
Scheduling timeout scheme:          0
Timeout time/weight:                6
Timeout scheduling:                 4
Scheduling priority scheme:         0
Priority scheduling:                4
Use queueing in subsystem:          1

Number of buses:                    2

Bus Spec #1
# buses with Spec:                  1
Bus Type:                           2
Arbitration type:                   1
Arbitration time:                   0.0
Read block transfer time:           0.0
Write block transfer time:          0.0
Print stats for bus:                0

Bus Spec #2
# buses with Spec:                  1
Bus Type:                           1
Arbitration type:                   1
Arbitration time:                   0.0
Read block transfer time:           0.0
Write block transfer time:          0.0
Print stats for bus:                1

Number of controllers:              1

Controller Spec #1
# controllers with Spec:            1
Controller Type:                    1
Scale for delays:                   0.0
Bulk sector transfer time:          0.0
Maximum queue length:               1
Print stats for controller:         1

Number of storage devices:          1

Device Spec #1
# devices with Spec:                1
Device type for Spec:               disk
Disk brand name:                    QUANTUM_TORNADO
Disk specification file:            atlas10k.diskspecs

PHYSICAL ORGANIZATION

Driver #1
# of connected buses: 1
Connected buses: 1

Controller #1
# of connected buses: 1
Connected buses: 2

Bus #1
# of utilized slots: 1
Slots: Controllers 1

Bus #2
# of utilized slots: 2
Slots: Controllers 1
```

```
Slots: Devices 1

SYNCHRONIZATION

Number of synchronized sets: 0

LOGICAL ORGANIZATION

# of system-level organizations:    1

Organization #1: Parts Asis Noredun Whole
Number of devices:                  1
Devices:                            1-1
High-level device number:           1
Stripe unit (in sectors):           17938986
Synch writes for safety:            0
Number of copies:                   2
Copy choice on read:                6
RMW vs. reconstruct:                0.5
Parity stripe unit:                 64
Parity rotation type:               1
Time stamp interval:                0.000000
Time stamp start time:              60000.000000
Time stamp stop time:               10000000000.000000
Time stamp file name:               stamps

# of controller-level organizations:0


Process-Flow Input Parameters
-----------------------------

PRINTED PROCESS-FLOW STATISTICS

Print per-process stats?            1
Print per-CPU stats?                1
Print all interrupt stats?          1
Print sleep stats?                  1

GENERAL PROCESS-FLOW PARAMETERS

Number of processors:               1
Process-Flow Time Scale:            1.0

SYNTHETIC I/O TRACE PARAMETERS

Number of generators:               5
Number of I/O requests to generate: 10000
Maximum time of trace generated (in seconds):1000.0
System call/return with each request:0
Think time from call to request:    0.0
Think time from request to return:  0.0

Generator description #1
Generators with description:        5
Storage capacity per device (in blocks):17938986
Number of storage devices:          1
Blocking factor:                    8
Probability of sequential access:   0.0
Probability of local access:        0.0
Probability of read access:         0.66
Probability of time-critical request:1.0
Probability of time-limited request:0.0
Time-limited think times (in milliseconds)
Type of distribution: normal
Mean:                               30.0
Variance:                           100.0
General inter-arrival times (in milliseconds)
```

```
Type of distribution: exponential
Base value:                         0.0
Mean:                               0.0
Sequential inter-arrival times (in milliseconds)
Type of distribution: normal
Mean:                               0.0
Variance:                           0.0
Local inter-arrival times (in milliseconds)
Type of distribution: exponential
Base value:                         0.0
Mean:                               0.0
Local distances (in blocks)
Type of distribution: normal
Mean:                               0.0
Variance:                           40000.0
Sizes (in blocks)
Type of distribution: exponential
Base value:                         0.0
Mean:                               8.0
```

As a final remark, we would like to mention one issue concerning a parameter in DiskSim: for all components (disk, buses, and controllers), there is a parameter called “Bulk sector transfer time”. According to the DiskSim manual, this parameter has the following function [30]:

> *... specifying the time necessary to transfer a single 512-byte block to, from, or through the controller. Transferring one block over the bus takes the maximum of this time, the block transfer time specified for the bus itself, and the block transfer time specified for the component on the other end of the bus transfer.*

The problem with this parameter is that using the values provided by the DiskSim package yields unrealistically low throughput. We have not been able to get any explanation from the authors of DiskSim, and personal communication with Seagate revealed that they had never heard of this parameter. Setting the parameter to zero yielded a disk performance in accordance with what could be expected, based on the disk characteristics. As a consequence, we have chosen to set this value to zero in all our simulations. The result is that the measured disk performance may be somewhat too high, but since this is equal for all schedulers, we do not consider this a problem.

Appendix B

Abbreviations

Prefixes

m	milli (10^{-3})
K	Kilo (2^{10})
M	Mega (2^{20})

Acronyms and Abbreviations

ACRA	Admission Control and Resource reservation Agent
APEX	AdaPtive disk schEduler for miXed-media workloads
APO	Atomic Presentation Object
BM	Buffer Manager
CGM	Computer Generated Multimedia Data
CHS	Cylinder Head Sector
CPO	Composite Presentation Object
CTL	Constant Time Length
DBMS	Database Management System
DBS	Database System
DSM	Decomposition Storage Model
EDF	Earliest Deadline First
FCFS	First Come First Served
fps	frames per second
IBE	Interactive Best-Effort
IDL	Interactive Distance Learning
KB	Kilobyte
LBA	Logical Block Addressing
LBN	Logical Block Number
LDU	Logical Data Unit

LoD	Learning-on-Demand
MB, Mb	Megabyte, Megabit
MMDBMS	Multimedia Database Management System
MMDT	MultiMedia Data Type
MR	Metadata Retrieval
NoD	News-on-Demand
NSM	N-ary Storage Model
OID	Object Identifier
OM	Object Manager
OMODIS	Object-Oriented Modeling and Database Support for Distributed Multimedia Systems
OS	Operating System
PDSM	Physical Data Structures Manager
PM	Presentation Manager
PSM	Physical Storage Manager
PTD_MMDT	Play Time Dependent MMDT
PTI_MMDT	Play Time Independent MMDT
QEP	Query Execution Plan
QID	Queue IDentifier
QM	Query Manager
QoS	Quality of Service
RID	Record Identifier
RT	Real-Time
SCSI	Small Computer Scalable Interface
SM	Storage Manager
SPTF	Shortest Positioning Time First
TA	Time Associator
TIBE	Throughput-Intensive Best-Effort
TM	Transaction Manager
TOOMM	Temporal Object-Oriented MultiMedia data model
VCR	Video Cassette Recorder
VoD	Video-on-Demand
xDSL	Asynchronous/Synchronous Digital Subscriber Line

References

1. Aho, A., V, Hopcroft, J., E, Ullman, J., D, *Data Structures and Algorithms*, Addison-Wesley Series in Computer Science and Information Processing, ed. Harrison, M., A, 1983: Addison-Wesley
2. Anastasiadis, S., V., Sevcik, K., C., Stumm, M., *Server-Based Smoothing of Variable Bit-Rate Streams*, in ACM Multimedia, Ottawa, ON, Canada, October 2001, pp. 147-158
3. Anderson, R., Osawa, Y., Govindan, R., *A File System for Continuous Media,* Transactions on Computer Systems, Vol. 10, No. 4, 1992, pp. 311-337
4. Balkir, N., H, Özoyoglu, G., *Delivering Presentations from Multimedia Servers,* The VLDB Journal, Vol. 7, No. 4, 1998, pp. 294-307
5. Balkir, N., H, Özoyoglu, G., *Multimedia Presentation Servers: Buffer Management and Admission Control*, in International Workshop on Multi-Media Database Management Systems (IW-MMDBMS'98), Dayton, Ohio, USA, August 1998, pp. 154-161
6. Barham, P., *A Fresh Approach to File System Quality of Service*, in NOSSDAV, St. Louis, Missouri, USA, May 1997, pp. 119-128
7. Boll, S., Heinlein, C., Klas, W., Wandel, J., *MPEG-L/MRP: Adaptive Streaming of MPEG Videos for Interactive Internet Applications*, in 6th International Workshop on Multimedia Information Systems (MIS'00), Chicago, USA, October 2000, pp. 104-113
8. Boll, S., Klas, W., Löhr, M., *Integrated Database Services for Multimedia Presentations*, in *Multimedia Information Storage and Management*, Chung, S., M.,, Editor. 1996, Kluwer Academic Publishers
9. Boral, H., DeWitt, D., J, *A Methodology for Database System Performance Evaluation*, in ACM SIGMOD, Boston, MA, USA, June 1984, pp. 176-185
10. Bosch, H., G., P, Mullender, S., J, *Real-time disk scheduling in a mixed-media file system*, Technical Report, INS-R0006, ISSN 1386-3681, 2000
11. Bruno, J., Brustoloni, J., Gabber, E., McSchea, M., Özden, B., A, S., *Disk Scheduling with Quality of Service Guarantees*, in IEEE Conference on Multimedia Computing Systems (ICMCS'99), Florence, Italy, June 1999, pp. 400-405
12. Bucy, J., Ganger, G., *Database of Validated Disk Parameters for DiskSim v2.0*, http://www.pdl.cmu.edu/DiskSim/diskspecs.html, May 2003
13. Buddhikot, M., M, Chen, X., J, Wu, D., Parulkar, G., M, *Enhancements to 4.4BSD UNIX for Efficient Networked Multimedia in Project MARS*, in the IEEE International Conference on Multimedia Computing and Systems (ICMCS'98), Austin, Texas, USA, June/july 1998, pp. 326-337
14. Card, R., Dumas, E., Mével, F., *The Linux Kernel Book*, 1998, Sussex, England
15. Carey, M., J, Jauhari, R., Livny, M., *Priority in DBMS Resource Scheduling*, in 15th VLDB Conference, Amsterdam, The Netherlands1989, pp. 397-410
16. Chang, E., Garcia-Molina, H., *BubbleUp: Low Latency Fast-Scan for Media Servers*, in 5th ACM international conference on Multimedia, Seattle, WA, USA, November 1997, pp. 87-98
17. Chang, E., Zakhor, A., *Cost Analyses for VBR Video Servers,* IEEE Multimedia, Vol. 4, No. 3, 1996, pp. 56-71
18. Chen, M., Kandlur, D., D, Yu, P., S, *Optimization of the Grouped Sweeping Scheduling (GSS) with Heterogeneous Multimedia Streams*, in 1st ACM Multimedia Conference (ACM MM'93), Anaheim, CA, USA, August 1993, pp. 235-241

19. Chen, S., Stankovic, J., A, Kurose, J., . F, Towsley, D., *Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems,* Journal of Real-Time Systems, Vol. 3, No., 1991, pp. 307-336

20. Chung, L., Gray, J., Worthington, B., Horst, R., *Windows 2000 Disk IO Performance*, Technical Report, MS-TR-2000-55, Microsoft, 2000

21. Connolly, T., Begg, C., *Database Systems - A Practical Approach to Design, Implementation, and Management*. Third ed, International Computer Science, ed. McGettrick, A., D, Essex, England, 2000, Addison Wesley

22. Curcio, I., D., D, Puliafito, A., Riccobene, S., Vita, L., *Design and evaluation of a multimedia storage server for mixed traffic,* ACM Multimedia Systems Journal, Vol. 6, No. 6, 1998, pp. 367-381

23. Daigle, S., J, Strosnider, J., K, *Disk Scheduling for Multimedia Data Streams*, in Conference on High-Speed Networking and Multimedia Computing, San Jose, CA, USA, February 1994, pp. 212-223

24. Danthine, A., Bonaventure, O., *From Best Effort to Enhanced QoS*, in International Workshop on Architecture and Protocols for High-Speed Networks, Schloss Dagstuhl, Germany, August/September 1993, pp. 179-201

25. Denning, P., J, *Effects of Scheduling on File Memory Operations*, in AFIPS, April 1967, pp. 9-21

26. Ecklund, D., Goebel, V., Plagemann, T., *The Architecture of an Interactive Multimedia Database Management System*, Technical Report, University of Oslo, 2001

27. Ecklund, D., J*, et al.*, *QoS Management Middleware - A Separable, Reusable Solution* -, in 8th intl. Workshop in Interactive Distributed Multimedia Systems (IDMS01), Lancaster, UK, September 2001, pp. 124-137

28. Ecklund, E., F, *Personal discussions*, Oslo, Norway, 2002

29. Ecklund, E., F, Lund, K., *Based on personal correspondence with Informix*, 2001

30. Ganger, G., R, Worthington, B., L, Patt, Y., N, *The DiskSim Simulation Environment Version 2.0 Reference Manual*, Reference Manual, Carnegie Mellon University, 1999

31. Geist, R., Daniel, S., *A Continuum of Disk Scheduling Algorithms,* ACM Transactions on Computer Systems, Vol. 5, No. 1, 1987, pp. 77-92

32. Gemmel, D., J, *Multimedia network file servers: Multi-channel delay sensitive data retrieval*, in ACM Multimedia '93, Anaheim, CA, USA, August 1993, pp. 243-250

33. Gemmel, D., J, Vin, H., M, Kandlur, D., D, Venkat Rangan, P., Rowe, L., A, *Multimedia Storage Servers: A tutorial,* IEEE Computer, Vol. 28, No. 5, 1995, pp. 40-49

34. GNU, *The GNU C Library - Asynchronous Reads/Writes*, GNU.org, http://www.gnu.org/manual/glibc-2.2.3/html_node/libc_toc.html, March 2002

35. Goebel, V., Eini, I., Lund, K., Plagemann, T., *Design, Implementation, and Application of TOOMM: A Temporal Object-Oriented Multimedia Data Model*, in 8th IFIP 2.6 Working Conference on Data Semantics (DS-8) "Semantic Issues in Multimedia Systems", Rotorua, New Zealand, January 1999, pp. 145-168

36. Goebel, V., Plagemann, T., Berre, A., J, Nygård, M., *OMODIS - Object-Oriented Modeling and Database Support for Distributed Multimedia Systems*, in Norwegian Informatics Conference (NIK'96), Alta, Norway1996, pp. 1-18

37. Golubchik, L., Lui, J., C., S, Muntz, R., *Reducing I/O Demand in Video-On-Demand Storage Servers*, in SIGMETRICS'95/Performance '95, Ottawa, Canada, May 1995, pp. 25-36

38. Gopalan, K., *Real-Time Disk Scheduling Using Deadline Sensitive SCAN*, Technical Report, TR-92, Experimental Computer Systems Labs, Dept. of Computer Science, State University of New York, Stony Brook, 2001

39. Gray, J., Shenoy, P., *Rules of Thumb in Data Engineering*, in 16th IEEE International Conference on Data Engineering (ICDE'2000), San Diego, CA, USA, April 2000, pp. 3-10

40. Guttman, A., *R-trees: A dynamic index structure for spatial searching*, in ACM SIGMOD International Conference on Management of Data, Boston, MA, USA, June 1984, pp. 47-57

41. Halvorsen, P., Goebel, V., Plagemann, T., *Q-L/MRP: A Buffer Management Mechanism for QoS Support in a Multimedia DBMS*, in IEEE International Workshop on Multimedia Database Management Systems (IW-MMDBMS 98), Dayton, Ohio, USA1998, pp. 162-171

42. Haritsa, J., Karthikeyan, M., B, *Disk Scheduling for Multimedia Database Applications*, in International Conference on Management of Data (COMAD'94), Bangalore, India, December 1994

43. Hori, H., *Tsunami MPEG Encoder*, TMPG Enc Net, http://www.tmpegenc.net/, April 2002

44. Härder, T., *Realiserung von operationalen Schnittstellen*, in *Datenbankhandbuch*, Lockemann, P., C., Schmidt, J., W, Editor. 1987, Springer Verlag. pp. 163-335

45. Härder, T., Rahm, E., *Datenbanksysteme: Konzepte und Techniken der Implementierung*, 1999, Springer.

46. Iyer, S., Druschel, P., *Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O*, in 18th ACM Symposium on Operating System Principles (SOSP 2001), Banff, Canada, October 2001, pp. 117-130

47. Jacobson, D., M, Wilkes, J., *Disk Scheduling Algorithms Based on Rotational Position*, Technical Report, HPL-CSP-91-7, Hewlett Packard Labs, 1991

48. Kamath, M., Ramamritham, K., Towsley, D., *Continuous Media Sharing in Multimedia Database Systems*, in 4th International Conference on Database Systems for Advanced Applications (DASFAA'95), Singapore, April 1995, pp. 79-86

49. Kamel, I., Niranjan, T., Ghandeharizedah, S., *A Novel Deadline Driven Disk Scheduling Algorithm for Multi-Priority Multimedia Objects*, in ICDE 2000, San Diego, CA, USA, February 2000, pp. 349-358

50. Kenchamma-Hosekote, D., R, Srivastava, J., *I/O Scheduling for Digital Continuous Media,* ACM Multimedia Systems Journal, Vol. 5, No. 4, 1997, pp. 213-237

51. Krasic, C., Walpole, J., *QoS Scalability for Streamed Media Delivery*, Technical Report, CSE-99-011, Oregon Graduate Institute, 1999

52. libmpeg2, *libmpeg2 - a free MPEG-2 video stream decoder*, http://libmpeg2.sourceforge.net/, April 2002

53. Liu, C., L, Layland, J., W, *Scheduling Algorithms for Multiprogramming in Hard-Real-Time Environment,* Journal of the ACM, Vol. 20, No. 1, 1973, pp. 47-61

54. Lockemann, P., C, *Datenbankimplementierung*, Lecture notes, Lehrstuhl für Systeme der Informationsverwaltung, Universität Karlsruhe, 2000

55. Lumb, C., R, Shcindler, J., Ganger, G., R, *Freeblock Scheduling Outside of Disk Firmware*, in The First USENIX Conference on File And Storage Technologies (FAST'02), Monterey, CA, USA, January 2002

56. Lund, K., Goebel, V., *Adaptive Disk Scheduling in a Multimedia DBMS*, to appear in ACM Multimedia, Berkeley, CA, USA, November 2003

57. Martin, C., Narayanan, P., S, Özden, B., Rastogi, R., Silberschatz, A., *The Fellini Multimedia Storage System,* Journal of Digital Libraries, No., 1997

58. Maxtor, *Atlas 10K SCSI Hard Drives*, Maxtor, http://www.maxtor.com/en/documentation/pdf_specifications/atlas_10k_scsi_specifications.pdf, May 2002

59. McKusick, M., K, Bostic, K., Karels, M., J, Quarterman, J., S, *The Design and Implementation of the 4.4BSD Operating System*. First ed, Addison-Wesley UNIX and Open Systems Series, ed. McKusick, M., K,Quarterman, J., S, 1998, Addison-Wesley

60. Merten, A., G, *Some Quantitative Techniques for File Organization*, Wisconsin, USA, 1970

61. Narashimha Reddy, A., L, Wijayaratne, R., *Providing Deterministic I/O Service for VBR Streams*, Technical Report, TAMU-ECE-9701, Texas A&M University, 1997

62. Narasimha Reddy, A., L, Wyllie, J., *Disk scheduling in a multimedia I/O system*, in 1st ACM International Conference on Multimedia (ACM MM'93), Anaheim, CA, USA, August 1993, pp. 225-233

63. Narasimha Reddy, A., L, Wyllie, J., C, *I/O Issues in a Multimedia System,* IEEE Computer, Vol. 27, No., 1994, pp. 69-74

64. Nerjes, G., Rompogiannakis, Y., Muth, P., Paterakis, M., Triantafillou, P., Weikum, G., *Scheduling Strategies for Mixed Workloads in Multimedia Information Servers*, in Eighth International Workshop on Research Issues in Data Engineering (RIDE'98), Orlando, Florida, USA, February 1998, pp. 121-128

65. Ng, R., T, Yang, J., *An Analysis of Buffer Sharing and Prefetching Techniques for Multimedia Systems,* Multimedia Systems Journal, Vol. 4, No., 1996, pp. 55-69

66. Nievergelt, J., Hinteberger, H., Sevcik, K., D, *The Grid File: An Adaptable, Symmetric Multi-Key File Structure,* ACM Transactions on Database Systems, Vol. 9, No. 1, 1984, pp. 38-71

67. Ohnishi, H., Okada, T., Noguchi, K., *Flow Control Schemes and Delay/Loss Tradeoff in ATM Networks,* IEEE Journal of Selected Areas in Communication, Vol. 6, No. 9, 1988, pp. 1609-1616

68. Pan, H., Ngoh, L., H, Lazar, A., A, *A Buffer-Inventory-Based Dynamic Scheduling Algorithm for Multimedia-On-Demand Servers,* ACM Multimedia Systems Journal, Vol. 6, No. 2, 1998, pp. 125-136

69. Patterson, D., A, Gibson, G., Katz, R., H, *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, in ACM Conference on Management of Data (SIGMOD), Chicago, IL, USA, June 1988, pp. 109-116

70. Presterud, O., *Dataplassering med Støtte for Synkronisering (in Norwegian)*, Master thesis (in progress), UniK, University of Oslo, Oslo, Norway, 2003

71. Puri, A., Schmidt, R., L, Haskell, B., G, *Overview of the MPEG Standards*, in *Multimedia Systems, Standards, and Networks*, Puri, A.,Chen, T., Editors. 2000, Marcel Dekker: New York. p. 87-129

72. Ramakrishnan, K., K*, et al.*, *Operating System Support for a Video-on-Demand File Service,* ACM Multimedia Systems Journal, Vol. 3, No. 2, 1995, pp. 53-65

73. Reisslein, M., Ross, K., Shrestha, S., *Striping for Interactive Video: Is It Worth It?*, in IEEE Int. Conf. on Multimedia Computing and Systems (ICMCS'99), Florence, Italy, June 1999, pp. 635-640

74. Ruemmler, C., Wilkes, J., *An Introduction to Disk Drive Modeling,* IEEE Computer, Vol. 27, No. 3, 1994, pp. 17-29

75. Sahu, S., Shenoy, P., Towsley, D., *Design considerations for integrated proxy servers*, in Int. Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'99), Basking Ridge, NJ, USA, june 1999, pp. 247-250

76. Saparilla, D., Ross, K., W, *Optimal Streaming of Layered Video*, in Joint Conference of the IEEE Computer and Communication Societies (INFOCOM 2000), Tel Aviv, Israel, March 2000, pp. 737-746

77. Scheuermann, P., Weikum, G., Zabback, P., *Data Partitioning and Load Balancing in Parallel Disk Systems*, Technical Report, 209, Department of Computer Science, ETH Zurich, Switzerland, 1994

78. Scheuermann, P., Weikum, G., Zabback, P., *"Disk Cooling" in Parallel Disk Systems,* Data Engineering Bulletin, Vol. 17, No. 3, 1994, pp. 29-40

79. Seagate, *Barracuda ATA IV Family Product Manual*, http://www.seagate.com/support/disc/manuals/ata/snowmasspma100129212.pdf, June 2003

80. Seagate, *Seagate Cheetah X15 Technical Specifications*, Seagate, http://www.seagate.com/cda/products/discsales/enterprise/tech/0,1084,376,00.html, 2000

81. Shenoy, P., *The Case for Reexamining Integrated File System Design*, in Tenth International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'00), Chapel Hill, NC, USA, June 2000, pp. 51-54

82. Shenoy, P., *DiskSim*, http://lass.cs.umass.edu/software/disksim/, May 2002

83. Shenoy, P., J, Goyal, P., Rao, S., S, Vin, H., M, *Symphony: An Integrated Multimedia File System*, in ACM/SPIE Multimedia Computing and Networking 1998 (MMCN'98), San Jose, USA, January 1998, pp. 124-138

84. Shenoy, P., J, Goyal, P., Vin , H., M, *Architectural Considerations for Next Generation File Systems*, in 7th ACM Multimedia Conference (ACM MM'99), Orlando, FL, USA, October 1999, pp. 457-467

85. Shenoy, P., Vin, H., M, *Cello: A Disk Scheduling Framework for Next Generation Operating Systems,* Real-Time Systems Journal: Special Issue on Flexible Scheduling of Real-Time Systems, Vol. 22, No. 1, 2002, pp. 9-47

86. Shenoy, P., Vin, H., M, *Cello: A Disk Scheduling Framework for Next-generation Operating Systems*, in International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS'98), June 1998, pp. 44-55

87. Shreedhar, M., Varghese, G., *Efficient Fair Queuing using Deficit Round Robin,* IEEE/ACM Transactions on Networking, Vol. 4, No. 3, 1996, pp. 375-385

88. Shriver, E., Merchant, A., Wilkes, J., *An Analytic Behavior Model for Disk Drives with Readahead caches and Request Reordering*, in SIGMETRICS, Madison, WI, USA, June 1998 1998, pp. 182-191

89. Sitaram, D., Dan, A., *Multimedia Servers, Applications, Environments, and Design*, San Fransisco, California, 2000, Morgan Kaufmann Publishers

90. Skjelsvik, K., S, *Adgangskontroll og ressursreservasjon for et multimediadatabasesystem (in Norwegian)*, Master thesis, in Institutt for informatikk, University of Oslo, Oslo, Norway, 2002

91. Solomon, D., A, Russinovich, M., E, *Inside Microsoft Windows 2000*. Third ed, Redmond, WA, USA, 2000, Microsoft Press

92. Stoica, I., Abdel-Wahab, H., Jeffay, K., *On the Duality between resource reservation and proportional share resource allocation*, in Multimedia Computing and Networking, San Jose, CA, USA, February 1997, pp. 207-214

93. Tamer Özsu, M., Szafron, D., El-Medani, G., Vittal, C., *An Object-Oriented Multimedia Database System for a News-on-Demand Application,* ACM Multimedia Systems Journal, Vol. 3, No., 1995, pp. 182-203

94. To, T., J, Hamidzadeh, B., *Dynamic real-time scheduling strategies for interactive continuous media servers,* ACM Multimedia Systems Journal, Vol. 7, No. 2, 1999, pp. 91-106

95. Valduriez, P., Khoshafian, S., Copeland, G., *Implementation Techniques of Complex Objects*, in International Conference on Very Large Data Bases (VLDB), Kyoto, Japan1986, pp. 101-110

96. Vin, H., M, Venkat Rangan, P., *Designing a Multi-User HDTV Storage Server,* IEEE Journal on Selected Areas in Communications, Vol. 11, No. 1, 1993, pp. 153-164

97. Vogel, A., Kerherve, B., von Bockmann, G., Gecsei, J., *Distributed Multimedia and QoS: A Survey,* IEEE Multimedia, Vol. 2, No. 2, 1995, pp. 10-19

98. Waldspurger, C., A, Weihl, W., E, *Lottery Scheduling: Flexible Proportional-Share Resource Management*, in USENIX Symposium on Operating Systems Design and Implementation (OSDI), Monterey, CA, USA, November 1994, pp. 1-11

99. Waldspurger, C., A, Weihl, W., E, *Stride Scheduling: Deterministic Proportional-Share Resource Management*, Technical Report, MIT/LCS/TM-528, MIT Laboratory for Computer Science, 1995

100. Wang, C., Ecklund, D. J., Ecklund, E. F., Goebel, V., Plagemann, T., *Design and Implementation of a LoD System for Multimedia Supported Learning for Medical Students*, in World Conference on Educational Multimedia, Hypermedia & Telecommunications (ED-MEDIA 2001), Vol. 3, Tampere, Finland, June 2001, pp. 1983-1988

101. White, B., Ng, W., T, Hillyer, B., K, *Performance Comparison of IDE and SCSI Disks*, Technical Report, Bell Labs, 2001

102. Wijayaratne, R., *Prism: A File Server Architecture for Providing Integrated Services*, Ph.D thesis, Texas A&M University, 2001

103. Wijayaratne, R., Narasimha Reddy, A., L, *Providing QoS guarantees for disk I/O,* Springer Journal on Multimedia Systems, Vol. 8, No., 2000, pp. 57-68

104. Wijayaratne, R., Narasimha Reddy, A., L, *System Support for Providing Integrated Services from Networked Multimedia Storage Servers*, in ACM Multimedia, Ottawa, Ontario, Canada, September/October 2001, pp. 270-279

105. Worthington, B., Ganger, G., Patt, Y., Wilkes, J., *On-Line Extraction of SCSI Disk Drive Parameters*, in ACM SIGMETRICS, May 1995, pp. 146-156

106. Worthington, B., L, Ganger, G., R, Patt, Y., N, *Scheduling Algorithms for Modern Disk Drives*, in ACM Sigmetrics Conference, Nashville, Tennessee, USA, May 1994, pp. 241-251

107. Xiaoye, J., Mohapatra, P., *Efficient admission control algorithms for multimedia servers,* Multimedia Systems Journal, Vol. 7, No., 1999, pp. 294-304

108. Yu, P., S, Chen, M.-S., Kandlur, D., D, *Design and analysis of a grouped sweeping scheme for multimedia storage management*, in Third International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'92), La Jolla, CA, USA, November 1992, pp. 44-55

109. Zabback, P., *I/O-Parallelität in Datenbanksystemen - Entwurf, Implementierung und Evaluation eines Speichersystems für Disk-Arrays*, Doctoral thesis (in German), Report no. 10629, ETH Zürich, Switzerland, 1994

110. Özden, B., Biliris, A., Rastogi, R., Silberschatz, A., *A Low-cost Storage Server for Movie on Demand Databases*, in VLDB, September 1994, pp. 594-605

111. Özsoyoglu, G., Snodgrass, R., T, *Temporal and Real-Time Databases: A Survey,* IEEE Transactions on Knowledge and Data Engineering, Vol. 7, No. 4, 1995, pp. 513-532

112. Özsu, M., T, Szafron, D., El-Medani, G., Vittal, C., *An Object-Oriented Multimedia Database System for a News-on-Demand Application,* ACM Multimedia Systems Journal, Vol. 3, No., 1995, pp. 182-203