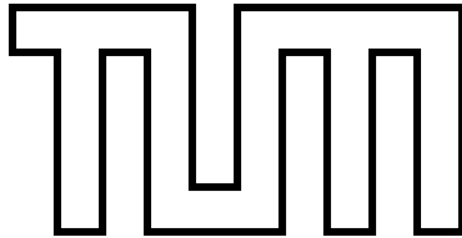

Technische Universität München

Fakultät für Informatik

Lehrstuhl für Rechnertechnik und Rechnerorganisation

Diplomarbeit

Treatment of Arbitrary-Size Data in Autonomous Disks

Michael Borgwardt

Themensteller: Prof. Dr. Arndt Bode

Betreuer: PD. Dr. Thomas Ludwig
Prof. Haruo Yokota

Abgabetermin: 15. Oktober 2001

Erklärung

Ich versichere, daß ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Oktober 2001

____
(Unterschrift des Kandidaten)

Zusammenfassung

Die vorliegende Diplomarbeit beschäftigt sich mit einer Erweiterung eines bestehenden Projekts, den *Autonomous Disks*. Dabei handelt es sich um "intelligente" Festplatten die, über ein Netzwerk verbunden, einen Cluster bilden, der selbstständig und automatisch hochenwickelte Funktionen zur Datenspeicherung zur Verfügung stellt, die transparente Ausfallsicherheit und Lastausgleich beinhalten. Eine zentrale Rolle spielt dabei der *Fat-Btree*, eine in hohem Maße skalierbare parallele Indexstruktur.

Die Implementation des Konzepts war jedoch vor Beginn dieses Projekts nicht in der Lage, Datenobjekte ("Ströme" genannt) zu verarbeiten, die größer als eine disk page waren. Es ging nun darum, erstens diese Beschränkung zu überwinden um beliebig große Ströme speichern zu können, und zweitens das System um die Fähigkeit zu erweitern, einzelne Ströme über mehrere Platten zu verteilen, um dann beim Auslesen die Übertragungsraten der einzelnen Platten zu kombinieren, sogenanntes *declustering*. Dabei sollten selbstverständlich die anderen Fähigkeiten der Autonomous Disks in vollem Umfang erhalten bleiben.

Diese Ziele wurden erreicht; zudem waren auf dem Weg dorthin einige Hilfsfunktionen zu implementieren, die von sich aus auch nützlich sind: zuerst mußte ein komplettes Allokationssubsystem für die Verwaltung des Plattenplatzes erstellt werden, dann wurde das Transportprotokoll um die Fähigkeit erweitert, große Ströme in kleineren Teilen zu übertragen und auf jeden beliebigen Punkt innerhalb eines Stroms zugreifen zu können, ohne den gesamten Strom übertragen zu müssen.

Schließlich wurde auch das Declustering implementiert, und zwar auf eine allgemeine Art und Weise die es erlaubt, zwischen mehreren verschiedenen Verteilungsstrategien zu wählen und neue Strategien einfach hinzuzufügen. Experimentelle Ergebnisse bestätigen dass dies gelang, und die erwarteten Leistungszunahmen auch wirklich erzielt werden können.

Acknowledgments

This diploma thesis would not have been possible without the contribution and support from the following people and organizations:

- Professor Haruo Yokota from the Tokyo Institute of Technology, who provided the foundation on which the practical part of this project was built and helped me during my work with vital advice and criticism.
- The Japanese education ministery and the Tokyo Institute of Technology, who made it possible for me to stay in Japan and work on this project.
- Dr. Thomas Ludwig, who provided advice and support for the writing of this thesis.

Contents

1 Introduction 11

2 Prior Environment 13
2.1 The Autonomous Disks Concept . . . 13
2.1.1 Clusters . . . 13
2.1.2 Internal Structure of an Autonomous Disk . . . 13
2.1.3 ECA Rules . . . 15
2.1.4 Interface . . . 15
2.1.5 Features . . . 17
2.1.6 Comparison to Other Concepts . . . 18
2.1.7 Prototype . . . 19
2.2 The Fat-Btree Parallel Index . . . 19
2.2.1 Data Distribution . . . 19
2.2.2 Index Structure . . . 20
2.2.3 Read Performance . . . 20
2.2.4 Write Performance . . . 20
2.2.5 Comparison to Other Parallel Indexes . . . 21

3 Goals 23
3.1 Treatment of Arbitrary-Size Data . . . 23
3.2 Allocation and Deallocation . . . 23
3.3 Declustering . . . 25
3.4 Consistency with Autonomous Disks Properties . . . 26

4 Related Work 27
4.1 Allocation . . . 27
4.1.1 Allocation Strategies . . . 27
4.1.2 Traditional File Systems . . . 29
4.1.3 Buddy Systems . . . 30
4.1.4 XFS . . . 31
4.2 File Systems . . . 31
4.2.1 BPFS . . . 32
4.2.2 Stacked Filesystems . . . 34
4.2.3 Galley . . . 36

4.3 Declustering . . . 37
4.3.1 One-Dimensional Data . . . 37
4.3.2 Two-Dimensional Data . . . 40
4.3.3 Higher Dimensional Data . . . 43
4.3.4 Similarity Graphs . . . 45

5 Design **47**
5.1 Allocation and Deallocation . . . 47
5.1.1 Main Data Structures . . . 47
5.1.2 Internal Allocation . . . 48
5.1.3 Deferred Coalescing . . . 49
5.1.4 Clustering Allocation . . . 50
5.2 Arbitrary-Length Streams, Random Access . . . 50
5.3 Management of Declustered Streams . . . 51
5.4 Management of Substreams . . . 52
5.4.1 Placement of Substreams . . . 52
5.4.2 Finding Substreams . . . 53
5.4.3 Flexibility . . . 53
5.5 Alternatives . . . 53
5.5.1 Allocation Mechanisms . . . 53
5.5.2 Storage Outside the Index . . . 54
5.5.3 StreamManager . . . 54

6 Implementation and Interfaces **55**
6.1 Allocation . . . 55
6.1.1 Package Structure . . . 55
6.1.2 Hardware Interface . . . 55
6.1.3 System Interface . . . 56
6.2 Random Access Within Streams . . . 57
6.2.1 Downwards-compatible Interface . . . 57
6.2.2 Division of Requests . . . 58
6.3 Declustering Functionality . . . 58
6.4 Intergration of Declustering Methods . . . 59
6.4.1 Name Generation . . . 59
6.4.2 Interface . . . 59

7 Declustering Methods **61**
7.1 Uniform Pseudo-Random Distribution (UPRD) . . . 61
7.2 Round-Robin Distribution (RRD) . . . 62
7.3 Multi-dimensional data . . . 63

8 Performance **65**
8.1 Transfer Rate . . . 65
8.2 Jitter . . . 65

8.3 Response Time 67
8.4 Influence of Substream Size 68

9 Conclusion **71**

10 Future Work **73**
10.1 Integration 73
10.2 More Declustering Methods 73
10.3 Automated Declustering 73
10.4 Improved Streaming 74

1 Introduction

This document describes a project which aimed at adding the ability to efficiently handle large data items to an existing project called “Autonomous Disks”. Autonomous Disks form a data repository consisting of a cluster of “intelligent” high-function disks that are attached to a network. They cooperate to offer parallel indexing, fault tolerance and skew handling to clients in a completely transparent fashion.

However, data items were previously restricted to one disk page in size. Removing this restriction required a number of modifications and extensions of the system, first of all a fully featured allocation and deallocation subsystem. Furthermore, *declustering*, the distribution of single data items over several disks, was also a desired functionality. The effectiveness of this technique depends greatly on how exactly the parts are distributed, and so it is beneficial to have a choice of different declustering methods.

The structure of this paper is as follows: section 2 describes the prior environment of the Autonomous Disks, their design and capabilities, as well as those of the Fat-Btree, the index structure used by the Autonomous Disks. Section 3 explains the motivation and various goals of this project. In section 4, a survey of related work is provided, showing similar efforts that influenced my work, and others that were of interest but not directly applicable.

Then, in section 5, the structural design of my contribution is shown, while section 6 gives insight into the actual implementation and the internal and external interfaces. Section 7 provides information about the concrete declustering methods implemented (or designed) in the project. The performance of the implementation is shown and analyzed in section 8. Section 9 concludes the description of the project, and section 10 suggests possible future work.

2 Prior Environment

The project described here did not require or allow a completely independant new design; the work was based on an existing design, the Autonomous Disks described in [1] (of which the Fat-Btree [2] is a vital part), and an existing implementation of that design. In order to provide a proper background for the rest of the paper, this chapter describes that environment.

2.1 The Autonomous Disks Concept

Autonomous Disks [1] are high-function disks that realize indexed shared-nothing parallel data storage, offering functionality that borrows from both file system and database roots, and featuring fault tolerance as well as the handling of data distribution and access skews, transparent to the client systems.

2.1.1 Clusters

The Autonomous Disks act as a cluster of cooperating nodes without any permanent "master" or "coordinator" node[1] and coordinate the above-mentioned tasks among each other, so that client systems don't need to be aware of the number, state and location of individual disks at all; the client can simply address the cluster as a single abstract entity. In practice, each disk accepts all possible requests and redirects them as necessary. Furthermore, dynamic reconfiguration of the cluster is possible, either deliberately (by adding disks or increasing data redundancy) or when a disk fails.

2.1.2 Internal Structure of an Autonomous Disk

Figure 2.2 shows the logical internal structure of an Autonomous Disk; each box corresponds to a Java object on the runtime level. The EIS commands (see section 2.1.4) that make up the communication between clients and disks as well as among disks are received by a `CommunicationManager`, which also dispatches outgoing commands. They are passed through FIFO Queues to either a `RuleManager` or a `RuleManagerLog`, depending on their type. The latter is used only on log disks (see section 2.1.5 and figure 2.1).

[1] However, it may be necessary for one of the disks to temporarily assume such a role for certain tasks to be completed

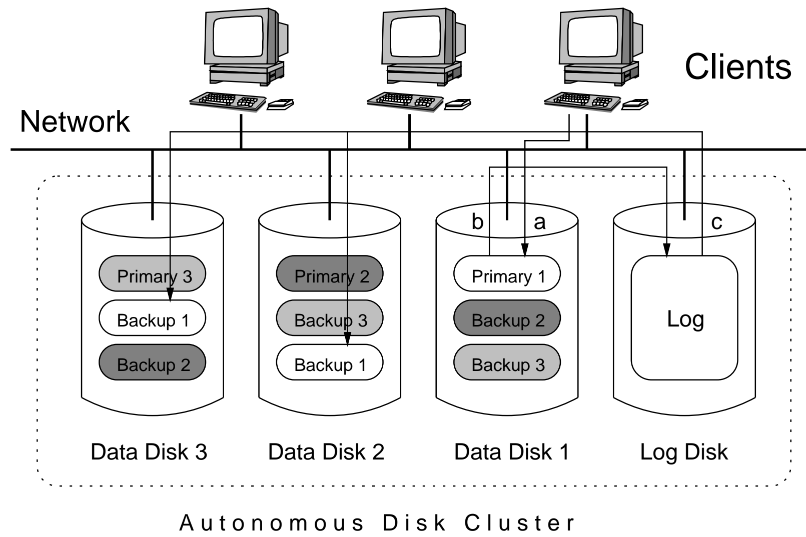

Figure 2.1: Autonomous Disk Cluster with double data redundancy and data path for an update operation. a: original update request — b: synchronous logging — c: asynchronous submission to backup

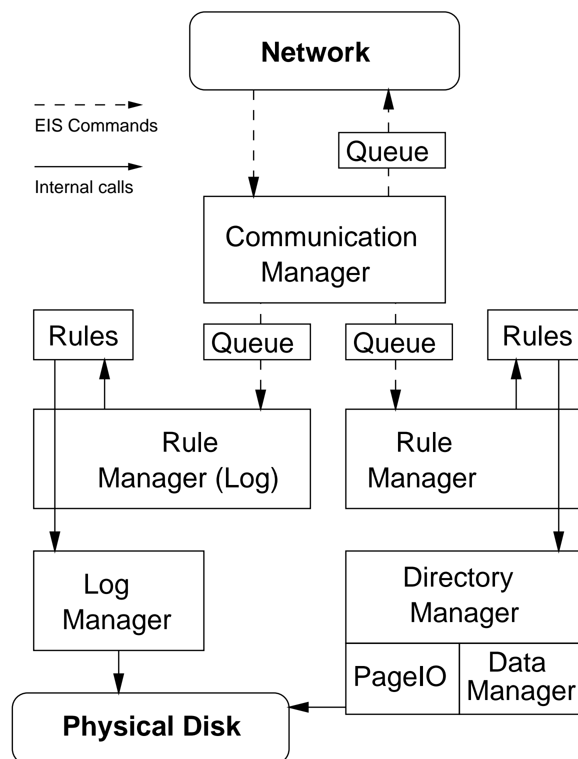

Figure 2.2: Internal logical and object structure of an Autonomous Disk

Both types of rule managers create instances of ECA Rules (see section 2.1.3) based on the type of and data in the EIS commands, and execute them. The rules are given the rule manager object as a context and access the rest of the system by calling methods provided by the the rule manager classes.

These methods[2] in turn do part of their work by using the `LogManager` class (only on log disks) or the `DirectoryManager` (access to the main index), `DataManager` (high-level read and insert methods that combine the actual data and index manipulations) and `PageIO` (low-level disk access) classes. The latter three are tightly interconnected and call each other's methods to some degree.

2.1.3 ECA Rules

All operations in the Autonomous Disks are controlled by rules following the event-condition-action paradigm described in [4]. This means that there are certain *events* defined that, when they occur, cause the system to start processing all rules which declare that event as their trigger. Each rule also defines *conditions*, and stops being processed if they are not fulfilled. If they are, then the *action* defined by the rule is executed. New rules can be added to adjust or extend the functionality of the system. An example for an ECA rule:

Event: Arrival of an `insert` command
Condition: StreamID is not present on this disk according to index
Action: Forward command to the disk that contains the StreamID

Generally, the events triggering the rules in the Autonomous Disks are arrivals of some sort of message over the network from a client or another disk.

2.1.4 Interface

Data items are abstracted as untyped data streams, and the Autonomous Disks are intended to provide general storage services for all kinds of data, such as XML documents, video data or sensor readings. Similar to OSD proposals [3] of an extended SCSI command set, such object-oriented treatment of data represents a more advanced interface than the block-oriented commands normally used to control disks.

EIS Commands

In the Autonomous Disks, communication between clients and disks as well as among disks happens through *external interface stream* (EIS) commands. These all have the same standardized format. Here is a excerpt[3] from the source code, showing the essential member variables of the `EISCommand` class:

[2] which constitute the *internal stream commands*; see section 2.1.4
[3] slightly modified for aesthetic purposes

```
int type;
SiteID SourceID;
String StreamID;
byte[] Stream;
```

`type` defines the type of command (such as insert, retrieve, result); its interpretation is fixed by constant class members of the `EISCommand` class, the names of which are used instead of numeric values in the code.

`SiteID` specifies the identity of the node the command was issued by, though this can be indirect. More concretely, it is the node to which results will be directed. Usually, `SiteID` will represent an IP number.

`StreamID` is the name of the Stream which is affected by the command, as used in the main index; in some commands, this may be unnecessary and thus have a `null` value. In my work, this fied was extended to carry additional information for declustering, see section 6.2.1.

`Stream` is the actual data being transferred. Again, this can be `null` in some commands.

As explained in section 2.1.3, EIS commands trigger the execution of rules, so the abstract client interface actually consists of both the EIS commands, the installed rules, and the possibility to add new rules or modify old ones.

IS Commands

The rules are given the fields of the triggering EIS command as data and an execution context (see section 2.1.2) which provides a number of *internal stream* (IS) commands that the rules can use to do check their conditions and execute their actions. Thus, the IS commands provide the interface of the Autonomous Disks system towards the (possibly user-provided or -modified) rules. Originally, the following IS commands were implemented:

send: Send an EIS command to another disk or a client.

traverse: Scan the index[4] for a stream ID.

insert_local: Insert a stream into the index[4].

read_local: Read data from the physical disk using the index[4].

catchup: Used on log disks to start comitting the asynchronous backups (see section 2.1.5).

[4] There are actually two almost identical commands; one that operates on the primary copy and another that uses the backup

waitAck: Wait for an acknowledgement message after sending an EIS command.

Two more commands, **compose** and **decompose** (which are needed for declustering), were mentioned in [1] but not implemented. The implementation of these commands (see section 6.3) was at the core of the practical part of this diploma thesis.

2.1.5 Features

Fault Tolerance

Fault tolerance is implemented by asynchronous backups: each update operation is first synchronously written to a dedicated log disk which then periodically submits the operations to one or more backup copies of the database. These copies reside on the same disks as the primary working copy, but interleaved in a way that guarantees that the entire database can still be accessed even when one[5] disk fails (see figure 2.1).

The disadvantage of this approach is that the synchronous write to the log disk adds twice the network latency to the response time of an insert request. However, in a multithreaded implementation, this time could be used to service other requests.

Theoretically, it would be possible to have a disk perform both logging and storage, but a dedicated log disk has the advantage of operating sequentially, and is therefore able to log operations from several normal disks. Experiments have shown a single log disk to become the bottleneck only when logging more than five disks; beyond that point, it becomes necessary to add more log disks. However, this depends on the nature of the operations being done; should the normal disks somehow also be able to operate sequentially, the log disk would become a bandwidth bottleneck earlier.

Transparent skew handling

Skew handling is provided by the Fat-Btree parallel index described in section 2.2, which serves as the main index for the streams stored in the Autonomous Disks. Disks should collect usage information, both in terms of space and access load, and pass it around[6]. Based on this information, access and data distribution skews can be reduced by moving the index ranges that the individual disks cover, and move data accordingly.

Unfortunately, since the ranges must be continuous in order to enable the index to operate efficiently (see section 2.2.1), there will sometimes be an imperfect tradeoff between the reduction of data distribution and access skews:

- If an index range contains a small amount of data that is accessed very frequently, it will have to occupy a disk exclusively in order to avoid an access bottleneck, even though that could leave the disk mostly empty. This effect will be reduced by caching.

[5] or more, depending on the number of copies

[6] A circular token-passing scheme is envisioned for this. The information doesn't need to be completely up-to-date since it is only used for a possibly inaccurate prediction of future access frequency anyway.

- If a range contains a very large amount of data that is never accessed, it would also need to have an entire disk dedicated to it for space reasons, wasting that disk's ability to perform actual work (except for doing the partial index lookups for some requests and forwarding them to the right disk).

2.1.6 Comparison to Other Concepts

There are a number of networked storage concepts which the Autonomous Disks can be compared to:

SAN

Storage Area Networks provide centralized storage space which is accessed over a network. However, this network is dedicated completely to that task and separate from the general communications network. Additionally, the client interface usually is at a very low level, e.g. SCSI commands. Obviously, there is little resemblance between SANs and the Autonomous Disks, save the fact that both allow the storage to be removed from individual machines and managed and reconfigured centrally.

Databases

Database servers such as the Oracle or IBM DB2 products, or more recently developed relational database systems, offer far more powerful functionality than the Autonomous Disks. However, features such as fault tolerance and skew handling, as well as declustering, are usually add-ons that need to be purchased separately and require additional configuration and administration overhead. Generally, the broad functionality offered by databases also comes at the cost of very high complexity in usage and administration. Autonomous Disks, on the other hand, offer some of the core functionality and some additional features at comparable or even increased (due to the Fat-Btree) performance while needing very little administration.

NAS

Network Attached Storage is a rather broad category which generally means storage servers that are accessed through high-level interfaces (usually on a filesystem level) across a general-purpose network. Examples are Sun's NFS and the SMB protocol used by Microsoft's "file sharing" services. Here, too, fault tolerance, skew handling and declustering are usually expensive and complicated add-ons. Still, this category accomodates the Autonomous Disks best.

All in all, The Autonomous Disks are probably best described as a networked hybrid of a database and a parallel filesystem that offers a useful subset of the normal features and adds some important features in an easy-to-use way.

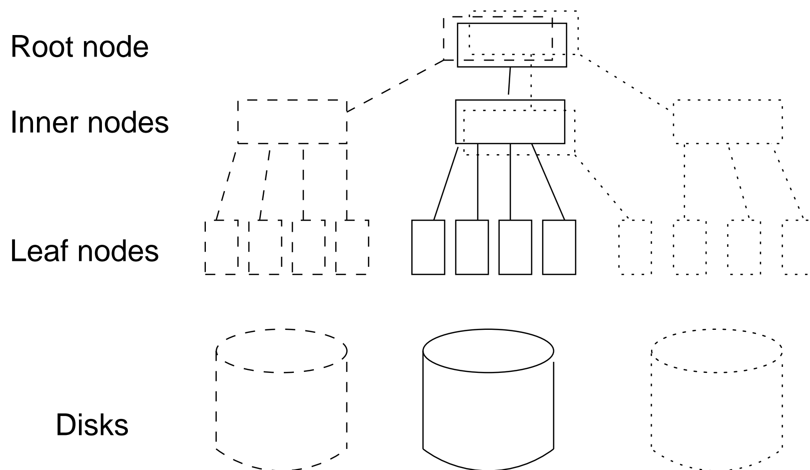

Figure 2.3: Structure of the Fat-Btree index; line styles show which nodes are placed on which disk, including duplicates

2.1.7 Prototype

There is a working, partial prototype implementation using the Java programming language[7] and advanced commodity hardware: Most of the development and some of the performance testing took place on PCs with 700 Mhz Intel Celeron processors equipped with 256MB RAM and connected through switched 100 MBit Ethernet. Some of the benchmarks could be run on a configuration with faster CPUs and connected through Gigabit Ethernet. Data migration and dynamic reconfiguration, as well as the Fat-Btree index are not yet fully implemented. The work described in this paper consisted of an extension of this prototype.

2.2 The Fat-Btree Parallel Index

A central role in the Autonomous Disks is played by the Fat-Btree [2], which is a B+ tree based, parallel directory structure that avoids bottlenecks for both reading and writing.

2.2.1 Data Distribution

Data items are distributed across the processing elements in a simple range partitioning scheme, but the B-tree index allows migration of data items to adjust for data distribution and access skews as mentioned in section 2.1.5. The range partitioning is a necessary result of the index structure (see section 2.2.2); were data items scattered over disks freely and the Fat-Btree rules still applied, all internal nodes of the tree would have to be copied on most disks, resulting in an index structure (and performance) much like the Copy-whole-Btree (see section 2.2.5).

[7] IBM Java2 SDK 1.3 on Linux

Additionally, this keeps the number of request forwards to a minimum as described in section 2.2.3. In particular, a request that arrives at the disk that actually holds the data item in question will never need to be forwarded.

2.2.2 Index Structure

The index structure of the Fat-Btree is displayed in figure 2.3. Basically, the policy is to have each processing element (PE) keep a copy of all tree nodes that must be traversed in order to reach *any* of the data items stored on that PE. Effectively,

- Leaf nodes will usually not be duplicated.
- The root node is duplicated on all PEs.
- Nodes in inbetween levels are duplicated on some PEs, the more the closer to the root they are.

This policy has the advantage of allowing both read and write accesses with high performance, unlike the alternative designs (see section 2.2.5), which perform well in only one of these and poor in the other.

2.2.3 Read Performance

The Fat-Btree offers good read performance, as index lookup work can be balanced between all PEs. When an index traversal arrives at a tree node which is not on the current PE, it always means that the data item affected by the request is also not on the current PE. The request will be forwarded to one of the disks which hold a copy of the node. The average number of forwards (Which of course directly affects response time in a negative way) will be low, but increasing (very slowly) with the number of disks and size of the tree.

2.2.4 Write Performance

Write performance is also good. Of course, having copies of index nodes means that there will be some synchronization overhead when one of them is changed as the result of an insert or delete operation, reducing performance. However, the impact of this is much less than might be expected, because the number of copies decreases exponentially towards the lower (i.e. close to the leaf nodes) levels of the tree, while the probability of an insert or delete request resulting in changes in a node decreases exponentially in the *opposite* direction in a B-tree.

In other words: the nodes that are most likely to have many copies (which would result in the most overhead when changed) are also exactly the nodes which will change most infrequently, while the nodes which are most often changed, the leaf nodes, are those with the lowest probability that there is a copy at all. In the end, write performance is nearly as good as if there were no synchronization overhead, because due to the nature of B-trees, the vast majority of write operations affect only one leaf node.

2.2.5 Comparison to Other Parallel Indexes

There are two widely used ways to build a parallel system with an index, both of which have severe restrictions in their performance; restrictions that the Fat-Btree overcomes.

Single-Index-Node:

In this system, the index is kept on a single PE, while the data is split across all the others. The advantage is that there are no duplicates, so also no synchronization overhead: write performance is good. Additionally, there will always be exactly one forward, because the index lookup is always completed on the index PE. However, the index PE can very quickly become a bottleneck for both reading and writing, which means that this approach is not usable for large databases. The Fat-Btree, on the other hand, exhibits no bottlenecks.

Copy-Whole Index:

Here, the entire index is duplicated on all PEs. Obviously, this means that lookup work can be perfectly balanced between all PEs, and that there will be at most one request forward, sometimes none. The problem is that each operation that changes the index requires maximal synchronization overhead, since all PEs are involved. Thus, the design is not suitable for any database with frequent changes. Again, compare the Fat-Btree, which has sometimes synchronization overhead, but infrequently and usually small.

3 Goals

The project had at its beginning the single, simple goal of enabling the Autonomous Disks to efficiently handle large data items. However, this required some preliminary work and a number of separate modifications, and eventually a set of somewhat independant goals formed, which are described in this chapter.

3.1 Treatment of Arbitrary-Size Data

Obviously, a generalized data repository should be able to handle data items of arbitrary size, as applications that manipulate massive amounts of data, such as multimedia streams, large XML documents, sensor data from manufacturing processes, stock market data or data transmitted by astronomical observation satellites, become more common or begin to employ more sophisticated, database-like storage systems for their superior data access and manipulation capabilities.

This demand was also honored in the SQL99 extension of the SQL language, which introduced the so-called *BLOB*, or "binary large object" data types into an environment which had previously been concerned only with the storage and manipulation of relatively small data items.

Thus, the Autonomous Disks should also be able to handle streams of arbitrary size. At the beginning of the project, the prototype implementation (see section 2.1.7) was not capable of storing streams larger than one disk page. Of course, clients could theoretically break up their data into disk-page sized chunks and submit these with generated names, and one of the parallel efforts did this, but it is obviously an unsatisfying solution because it makes the Autonomous Disks harder to use. Also, since clients have no knowledge of the internal structure of an Autonomous Disks cluster, they cannot perform optimal declustering (see section 3.3).

3.2 Allocation and Deallocation

Another feature that the prototype lacked was the ability to allocate and deallocate space freely. Space was allocated at the end of a file, one page at a time, and deallocation was not supported at all. In order to improve the overall capability, I decided to first implement a full allocation subsystem instead of just extending the existing system to also allocate larger extents.

An allocation subsystem must mainly perform two tasks:

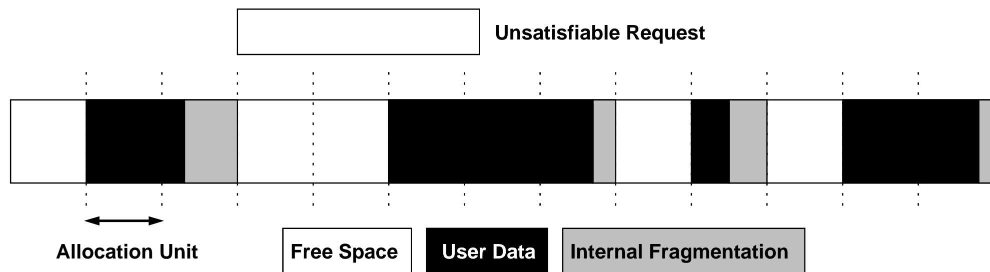

Figure 3.1: Partially allocated storage. A request for more than two allocation units cannot be satisfied, though enough free space is available: external fragmentation

- Accept requests for the *allocation* of a given amount of space, find for each request a large enough amount of free space to satisfy it, mark the space as used and make it available to the originator of the request.
- Accept *deallocation* requests for space that is not anymore needed by an application and add it to the free space pool so that it can be reused.

Additionlly, there are two important constraints on *how* allocators should perform these tasks:

Speed There should be little delay, i.e. the allocator should not need to perform many CPU- or IO-intensive operations.

Efficiency The allocator should waste as little space as possible.

The latter requirement is the more problematic one, as it has been shown in [8] that it can not be satisfied for all cases: For any given allocation mechanism, there is always some sort of allocation pattern that will result in high *fragmentation*.

Fragmentation is the term generally used for space "wasted" by allocators. There are two distinct types of fragmentation, and some allocators use a tradeoff between the two types. See figure 3.1 for an explanatory image.

Internal Fragmentation is space that is allocated but not used, due to constraints on allocation sizes or object alignment.

External Fragmentation is space that is free, but cannot be used to satisfy a request, usually because the free space is divided into several separate chunks, while the request needs one continuous chunk. Another source of external fragmentation are alignment restrictions in some allocator mechanisms[1] that prevent consecutive free space from being allocated as a whole in some cases.

I wanted to provide the ability to allocate continuous extents of disk space as required, as well as deallocate unneeded space and automatically coalesce adjacent free extents,

[1] Namely in buddy systems, see section 4.1.3

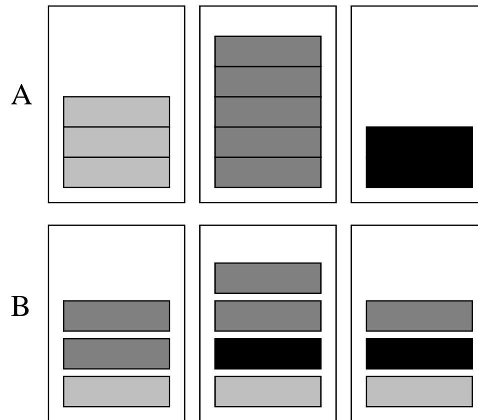

Figure 3.2: A: monolithic streams make skew handling impossible – B: declustered streams enable skew handling

and do it in a fashion that would result in little overhead and low fragmentation. Later, I decided to add the ability to allocate space "near" a given block address, a technique that might be used to reduce seek times by keeping related data structures close together.

3.3 Declustering

As streams become larger, it is more and more desirable to distribute (or "decluster") them across several disks, since that makes it possible to combine the transfer rates of the disks to achieve a higher total throughput and lower response time. Furthermore, skew handling is much easier when there are many small streams instead of few large ones, because skew handling is done by migrating streams across disks, and the streams' size therefore acts as a "grain" on skew handling and limits its applicability, as shown in figure 3.2. In fact, declustering already represents a simple method of avoiding, or at least reducing data distribution and access skews.

Finally, since the implemented allocation subsystem (see section 5.1) avoids the indirection-related speed penalty of indexed allocation, it has to allocate space requests continuously. If the available free space is fragmented (which cannot be avoided in some situations), a large allocation request may fail even though enough space is available. However, several smaller requests are much more likely to be satisfied even when fragmentation is severe. Of course, this phenomenon is basically the same as indexed storage, imposing a similar speed penalty, but yielding the additional benefits mentioned above.

Data declustering across disks has been the subject of many previous research efforts, such as [17] [14] [24], and different methods with various requirements and strengths have been developed. Obviously, it is desirable to allow the choice between several declustering methods since some methods may be especially suited for a particular kind of environment or data. Ideally, it should be possible to easily add new declustering methods.

3.4 Consistency with Autonomous Disks Properties

Of ourse, the implementation of the above goals should also be done in a way that interferes as little as possible with the rest of the system, retains the other features of the Autonomous disks and creates as little additional complexity as possible.

This sounds obvious, but as we will see, it's not easy to do and therefore important to keep in mind as a major separate goal.

4 Related Work

Any serious scientific work must take into consideration previous research efforts on the same or similar subjects, to avoid repeating work that has already been done, especially in cases where other, superior techniques have been found. Therefore, this chapter provides a survey of related efforts.

4.1 Allocation

As described in section 5.1, the first step in my modifications of the Autonomous Disks design was the addition of a subsystem for disk space allocation and deallocation. There are a number of approaches to and theoretical considerations of this classical problem, and I took several of those into account.

4.1.1 Allocation Strategies

In [6], Wilson et al. provided a survey of dynamic storage allocation techniques to avoid fragmentation. In [7], two of the same authors elaborated upon their conclusions by benchmarking the performance of a number of allocation techniques for allocation traces obtained from various real-world applications. Their work concentrates on allocation of semiconductor memory, not harddisks, but I believe that their conclusions are still applicable to some degree.

In [6], the authors argue that there are three distinct aspects of solutions to the allocation problem:

Strategy: a set of overall long-term decisions which aim to provide good allocation that exploits regularities in program behaviour. A major part of a strategy may be the definition of what exactly constitutes "good" allocation.

Policy: actual rules that specify which of the sufficiently large free blocks should be used to satisfy a given request

Mechanism: the algorithms, data structures and other details of a real implementation of a given policy.

According to the authors, too much attention has been paid to detailed mechanisms, too little to policies and virtually none to strategies. They also argue that the randomized synthetic traces upon which most previous allocator benchmarks had been based are

not realistic, because real world applications exhibit phase behaviour in both request sizes and object lifetimes, and there is often a correllation between the two. The lack of these characteristics in randomized traces can affect fragmentation both positively and negatively.

However, in [7], the authors found that a number of well-known allocation techniques[1] produced extremely low fragmentation when processing their traces. They concluded that the phase behaviour of real-world applications favors these allocators, while randomized traces penalize them unnecessarily, causing programmers to avoid using them, especially since some of them are (falsely) thought to be slow.

In regard to the question of allocation strategies, the authors concede (not openly though) that it is hard to find a policy that implements a given desired strategy, and to understand exactly what strategy a given policy implements. They focus mostly on policies, but claim that the well-performing allocators mostly do so because they all implement two successful strategies: placing objects allocated at the same time together because they tend to be deallocated at the same time, and allocating recently deallocated space preferentially in order to give other free space time to be coalesced.

When estimating the applicability of these results to the allocation of harddisk space, there are several issues to consider. First of all, the experiments in [7] used only short-time runs of programs; after the end of the process, all address space allocated to it is freed, and so fragmentation is a considerably smaller problem than with harddisks, where the lifetime of an area with allocation activity, usually an entire filesystem, is much longer - often years[2].

On the other hand, allocation and deallocation occurs much more slowly on disks, and there is no reason that the fragmentation situation would become dramatically worse over time when the experiments showed very little fragmentation during a limited-time program run that already displayed some phase behaviour.

A more important difference is the parallel usage of disk space by many applications, even many users. In [6] and [7], a basic assumption was that an allocator needs to deal with allocation and deallocation requests of only one application, but this is hardly ever true with harddisks. Thus, the experimental results cannot be expected to apply completely.

However, it seems extremely unlikely that parallel usage would result in any kind of malicious regularities - at worst, the allocation patters should be similar to the random traces that Wilson et al. deemed unfit for allocation benchmarks. More likely, some of the beneficial phase behaviour that led to the very low fragmentation observed in [7] would be preserved and the fragmentation be somewhere between that and the one observed with random traces.

Since studies cited in [6] showed the same allocation techniques that performed so

[1] Namely, certain variants of "best fit" and "first fit" as well as a size-class-based allocator developed by Doug Lea.

[2] At first glace, the fragmentation problem may also seem to apply to the entirety of physical RAM while the computer is running, but due to the virtual memory architectures of modern operating systems, it is not an issue

well in [7] to perform at least adequately[3] for random traces, it seems warranted to use one of those techniques and expect fragmentation to be tolerable. My implementation (see section 5.1) thus adopted "address-ordered best fit", a policy that showed the second-lowest fragmentation in [7] and has quicker implementations than the even better "address-ordered first fit" policy. The actual implementation is highly efficient and scalable, inspired by the XFS file system (see section 4.1.4).

Another aspect of allocation that was analzed in [6], and which deserves brief mention is the concept of *deferred coalescing*. This manifests as a simple list of fixed size in which recently freed extents are stored, with no attempt of coalescing them with adjacent free space. When an allocation request is served, this list is searched sequentially before the normal meachanism is applied. The reason to do this is that after an extent is freed, it will often be immediately reused to serve an allocation request, in which case the coalescing would have been unnecessary work. The technique affects both speed and fragmentation, but the effects are not well researched. It was adopted for the Autonomous Disks' allocation subsystem, not because of speed gains but because a design necessity explained in section 5.1.2.

4.1.2 Traditional File Systems

Traditionally, space management in file systems has been done through bitmaps, where the state of each block — used or unused — is stored as one bit in the bitmap. This is simple to implement, but has the obvious disadvantages that finding enough free space to satisfy a request requires a sequential scan through the bitmap, and the necessary work for allocating or deallocating a block also increases linearly with its size. However, there are some factors that make this approach viable reagrdless:

- File systems are normally used to store many comparatively small files, so allocation requests are usually easy to satisfy.
- Most file systems use some kind of "indexed allocation", in which a file can consist of a (possibly large) number of non-continuous chunks, so that large allocation requests can be satisfied by using several smaller free extents, avoiding exhausive searches.
- The allocation unit (the block size) is comparatively large, making the bitmap and thus the constant factor of the linear cost relatively small.

For these reasons, most file systems choose the bitmap approach despite its speed penalty. Examples are traditional Unix file systems like the BSD FFS or Linux ext2, Windows NT's NTFS [9], the Macintosh HFS [10] and the BeOS file system [11].

Since the Autonomous Disks are a hybrid of file system and database (see section 2.1.6), I wanted to avoid the speed penalty of indexed allocation for most data

[3] Under strong assumptions, especially of randomness, they follow the "fifty percent rule" developed by Knuth in [5], which says that the number of unused (including unusable due to fragmentation) blocks tends to be half as much as the number of blocks in use.

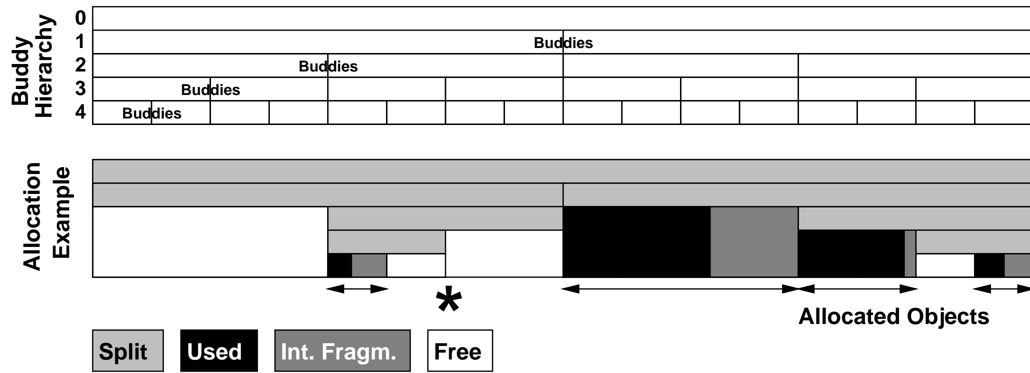

Figure 4.1: Binary buddy system, free and allocated. *: Free blocks cannot be allocated together due to alignment

items. This also made the bitmap approach infeasible, so I needed to find a more scalable solution.

4.1.3 Buddy Systems

One group of storage management methods that avoid sequential scanning are *buddy systems*, as described in [5]. In a buddy system, space is divided into fixed-size blocks that can be hierarchically subdivided. Each block has a *buddy*, with which it can be combined to form a block of the next higher hierarchy level. Blocks are allocated as a whole, so requests have to be rounded up to the next higher block size. If no block of that size is available, a block of the next higher size is split and one of the two buddies used. The most common case is the *binary buddy system*, in which block sizes are powers of two. See figure 4.1 for an explanatory image.

The advantage of this system is that finding a free block of the appropriate size can be done very easily and quickly by keeping a lists of the free blocks in each size class. Finding a block's buddy is a matter of address calculation, and so coalescing is also simple and speedy if usage information is kept in a data structure, such as a hierachical bitmap, that performs these operations quickly.

Unfortunately, the system also has a severe disadvantage: the size classes result in a very high internal fragmentation (25% on average for the binary buddy system), and alignment restrictions (see figure 4.1 for an example) add to external fragmentation. Consequently, buddy systems exhibited prohibitively high fragmentation in [7]. For this reason, the technique is not very popular and it is usually not considered at all for harddisk allocation, possibly also because the well-known implementation method described in [5] is not appropriate for use on harddisks.

However, Koch described in [12], how a binary buddy system was successfully used for disk space allocation on the Dartmouth Time-Sharing System. To my knowledge, this is the only buddy-system-based file system implemented and described in literature. The key to its success was again the (extent-based) indexed allocation, which made it

possible to drastically reduce fragmentation by allocating a file on a small set[4] of non-continuous extents instead of a single one. For example, a file of 19K+1 bytes in size would be allocated two extents of 16K and 4K, instead of one extent of 32K, as would be necessary in a "pure" binary buddy system, thus reducing internal fragmentation from 40.6% to only 5%. This optimization was performed mostly by a periodically running utility that reallocated files with high internal fragmentation, since allocation has to be done when necessary during *write* operations, and cannot assume knowledge of the file size after the final *close* operation.

This unique approach seems surprisingly effective and somewhat unfairly ignored by file system designers, though the need for the reallocation utility limits its usability to systems in which file creation, growing and shrinking are relatively infrequent, and that have off-peak times during which the relatively resource-hungry utility can run without impeding productive operation.

In regard to the Autonomous Disks, however, it is not applicable, due to the lack of indexed allocation. Additionally, Autonomous Disks might well be used in database-like environments where the overhead resulting from the reallocation utility would not be tolerable.

4.1.4 XFS

In [13], Sweeney describes the architecture of XFS, the default file system used in Silicon Graphics' IRIX operating system since 1996. It is not really a parallel file system because it does not directly employ or support any kind of parallelism, but it *is* designed not to obstruct parallel operation when the underlying storage is some sort of disk array.

More interesting from my viewpoint was the highly scalable design of its internal data structures, including those used in the management of free space:

XFS uses B-trees to efficiently implement a "best fit" allocation policy. Free space is managed in *extents* of consecutive free blocks. The extents are kept in *two* B-trees, one indexed on extent size, the other on the extents' starting address. The first is used to find the smallest extent that can satisfy an allocation request[5], while the address-indexed tree is used to coalesce free extents and to find free space "near" a given address[6].

The efficiency and scalability of this approach, even when not using indexed allocation, made it ideal for the allocation subsystem in the Autonomous Disks, and I therefore adopted this design in my implementation described in section 5.1.

4.2 File Systems

As mentioned in section 2.1.6, the Autonomous Disks are basically something intermediate between a parallel file system and a (very simple) parallel database with the added capabilities of skew handling and fault tolerance. Even though the file system aspect was

[4] Only around 1.3 on average

[5] this constitutes the "best fit" allocation policy.

[6] This allows related data to be kept close together, which reduces seek times.

not considered much in [1], it is in my opinion the dominant one, therefore I considered other research efforts on (especially parallel) file systems prominently.

4.2.1 BPFS

In [28], Russell describes the architecture of BPFS, a Basic Parallel File System. This work was not considered when designing my architecture, but in retrospect, my declustering subsystem ended up being remarkably similar to the BPFS architecture.

BPFS is designed as a modular, distributed parallel filesystem for use on clusters of workstations. Its core is an architecture and a set of protocols for communication between the components. The implementation described in [28] is relatively low-level, providing no filesystem structure and using an underlying native filesystem to provide the storage and indexing. The BPFS functionality is divided into four main components:

Clients collect the data for a user or application. They are expected to be realized as libraries that encapsule the BPFS functionality and provide an API that may be similar or identical to existing ones[7]. The number of client processes on each node depends on the implementation.

Servers control the actual data, and each server process is solely responsible for all access to its assigned storage; this makes sense because the physical disk forces a sequential execution of requests anyway.

Managers handle the metadata of files, the declustering, creation and deletion. One manager process is solely responsible for all metadata access to a set of files. It needs some storage in which to keep the metadata.

Agents act as proxies for client requests and perform caching as well as hiding the declustering from the client. There is one agent process for each open file and it performs all operations on that file.

These components are distributed over three types of "logical nodes"; there can be multiple instances of each type, distributed among a network of (possibly) heterogenous physical nodes. A physical node can contain instances of more than one type of logical node. The three types of logical nodes are:

Client Nodes consist of a user application process that accesses BPFS via a client component.

Server Nodes consist of a server component, a large amount of storage which it controls, and a number of agent components.

Manager Nodes consist of a manager component and a (relatively small) amount of storage for metadata.

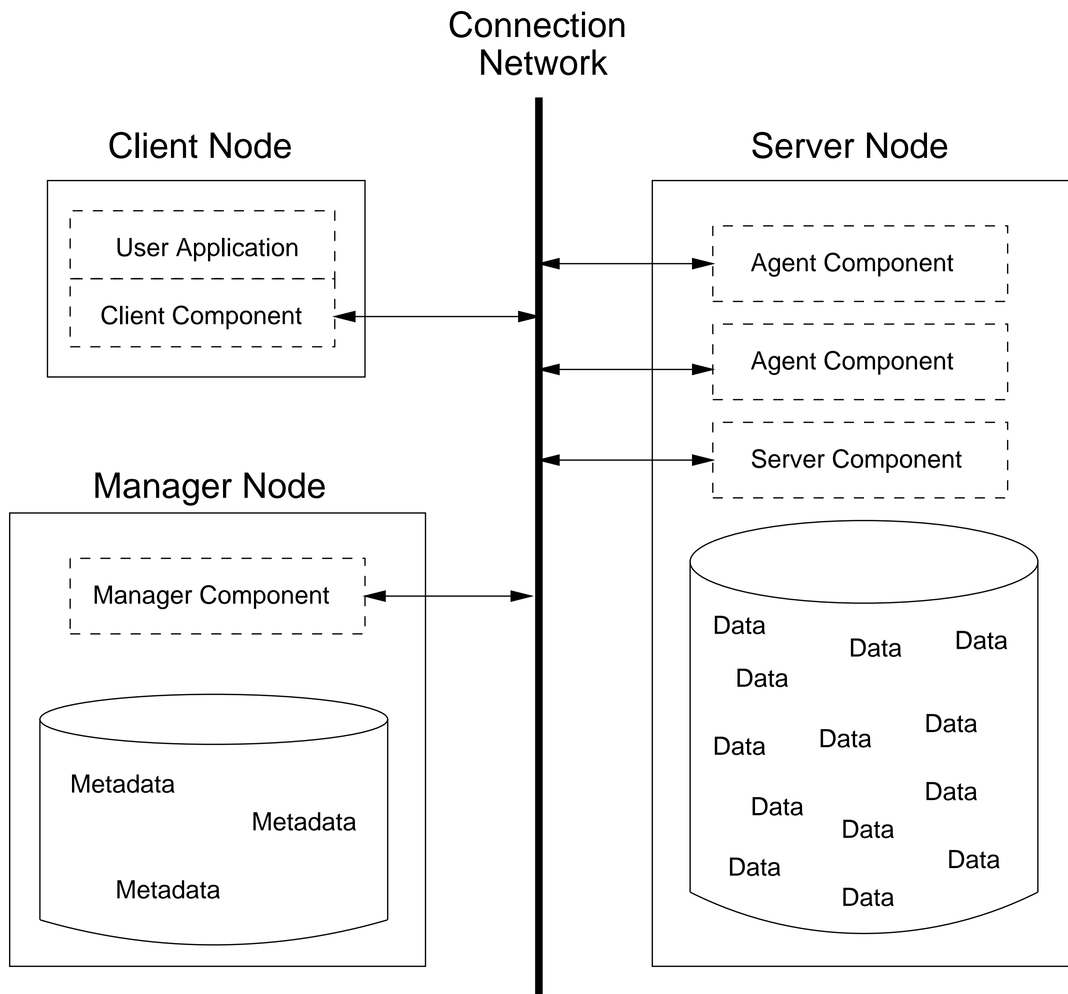

Figure 4.2: The architecture of BPFS

Declustering is performed by mapping locations within a file on names of (and offsets within) "data components" which are distributed across the disks. The mapping function can be supplied by the client and the mapping data is stored as file metadata in the file's manager component.

It's easy to see that this is basically the same approach taken in my approach of declustering for the Autonomous Disks. There are, however, also some differences:

- The BPFS handling of mapping functions appears more flexible, especially the fact that a client is described as being free to supply a completely new mapping fuction. However, it is not described how this function is stored and computed by the manager. This feature might add considerable complexity to the implementation.
- My design doesn't have the division into separate manager, server and agent processes; each Autonomous disk performs all of these tasks. Especially notable is the lack of a separate agent component; the autonomous Disks design is stateless and does not require the opening of files. This avoids resource bottlenecks[8] and setup delays, though having a separate agent process might be beneficial for large operations and especially for stream-oriented applications that require a constant data rate (see section 10.4) and prevent superstream index entries from becoming bottlenecks (see section 5.3).
- The BPFS design does not address issues such as fault tolerance[9], indexing and load balancing, while in the Autonomous Disks design, these features are seamlessly provided by the rest of the system. It is conceivable that a different BPFS implementation might provide these on a separate layer, but the result could be less efficient because the overall design does not take them into account.

  This is probably a very general tradeoff between modularity and performance: clearly separated modules faciliate design and maintenance on a high level and allow parts of the system to be replaced separately from the rest, but if the APIs between the modules are too narrow, they might not allow some forms of interaction that might be beneficial, especially for performance. On the other hand, if they are too big, they become difficult to use and the probability increases that there are hidden dependencies that effectively destroy the modularity.

Still, the similarity between the two approaches is strong and can be seen as an indication that the design is fundamentally sound.

4.2.2 Stacked Filesystems

In [29], Heidemann developed what he calls the "stackable" design of file systems. This basically means a layered approach in which each layer accepts any kind of operation, and

[7] one of the implemented client libraries mimics the standard C IO library

[8] the agents are analogous to the file descriptors in traditional Unix file systems

[9] a severe drawback, since the probability of a hardware failure exponentially approaches one as the number of involved components increases

	res_path	create	unlink	open	read	write	alloc	free	rd_block	wr_block
Read-Only	○	×	×	•	○	×	○	○	○	○
Directory	•	•	•	○	○	○	○	○	○	○
Inode/File	○	•	•	•	•	•	×	×	○	○
Allocation	○	○	○	○	○	○	•	•	×	×
Cache	○	○	○	○	○	○	○	○	•	•
Block	○	○	○	○	○	○	○	○	•	•
Default	×	×	×	×	×	×	×	×	×	×

×: reject ○: unknown/pass-through •: implement/modify.

Figure 4.3: A stacked file system and a part of its operations vector, showing how operations are handled through the stack. Note that an implemented operation in one layer may result in this or other operations of the layer below being called.

either executes it (possibly calling one or more operations of the layer beneath), rejects it (creating an error) or, if it isn't concerned with the operation (the most common case), passes it through unchanged.

Each layer should provide a separate, small part of the filesystem functionality. Examples are encryption layers, cache layers or network transparency layers. The functionality might also be extremely small, such as compatibility adjustments or making a file system read-only. This is called a "featherweight layer" by Heidemann and given special attention. Additionally, he develops a cache coherence framework to make caching on different layers possible while keeping data consistent.

The reason for the development of stackable design is stated as the need to make file systems easier to design and implement, and (more importantly) allow binary compatibility between file system modules from different parties (i.e. software vendors) without requiring access to the source code.

Heidemann himself mentions that there are strong parallels between his stacking techniques and object-oriented design[10]. Each layer might be implemented as a class that has an object of the next lower layer as an instance member, and featherweight layering is nearly equivalent to the creation of a subclass that overrides one or a few methods.

However, the central feature of the stacking concept (though it is not stressed in the paper) is the dynamic configuration of a stack of layers that were written independantly from each other, in which a global *operations vector* is created that consists of function pointers to the implementations of all operations offered by the stack. This would be very inelegant or difficult to emulate in Java, requiring either the use of "generic" methods that take e.g. an array of byte arrays as their arguments and parsing those, or some sort of on-the-fly rewriting of classes by a customized `ClassLoader` to make them contain pass-through methods for operations offered by lower layers that they don't modify.

Since the Autonomous Disks prototype implementation (see section 2.1) on which

[10] or languages, such as our implmentation language Java

my work was based did not use a layer approach for its design, and since the object-orientation of the implementation language already provides benefits (in regard to design ease) of the stacking technique while the others (binary compatibility) are of questionable relevance, especially for a research prototype system, I chose not to try applying the stacking technique to my design.

4.2.3 Galley

in [30], Nieuwejaar and Kotz present Galley, a parallel file system that is intended mainly for use on massively parallel supercomputers. They cite results showing that the actual access patterns of scientific applications developed for such systems are fundamentally different from those in uniprocessor or vector supercomputers and therefore not adequately handled by traditional parallel file systems. In particular, such applications tend to make large numbers of regular, but small and non-consecutive file accesses, which traditional file systems don't parallelize effectively.

The main improvement in Galley is therefore the introduction of a powerful (and very complex) interface that application programs can use to specify their access patterns, in particular *strided* patterns, which are very common in scientific applications. A strided access pattern is one that consists of multiple accesses of identical size and with identical intervals between them. Galley allows the application to specify both simple and nested strided access patterns, giving Galley much more information than other file systems which it can use to execute requests in parallel, coalesce them if possible, and perform intelligent disk scheduling.

Declustering is provided by splitting files into a fixed number[11] of *subfiles*, where each disk contains at most one subfile of any given file. Additionally, each subfile can contain one or more (completely separate) *forks* of data. Subfiles and forks must be explicitly addressed by the application (or a library it uses). This gives the developer maximum flexibility for the development of IO-optimal parallel algorithms.

Galley obviously provides very useful functionality that could result in large performance gains in its intended target environment: scientific applications that do "number crunching" on massively parallel systems. However, this environment differs a lot from that which Autonomous Disks are intended for. Applications programmers who want to make use of Galley's performance improvements need significant knowledge of the system details and must write their applications specifically for Galley, while a key feature of Autonomous Disks is easy, transparent use of their features. Additionally, Galley is designed to cater optimally to a single parallel application that uses a few massive data structures, and it might not perform very well in a multiuser environment with many different data items, for which Autonomous Disks are intended.

For these reasons, and because Galley does not address issues such as skew handling and fault tolerance, which are key features of Autonomous Disks, Galley is largely incompatible with my project and was not considered as a source of ideas during its design and implementation.

[11] to be chosen when creating a file

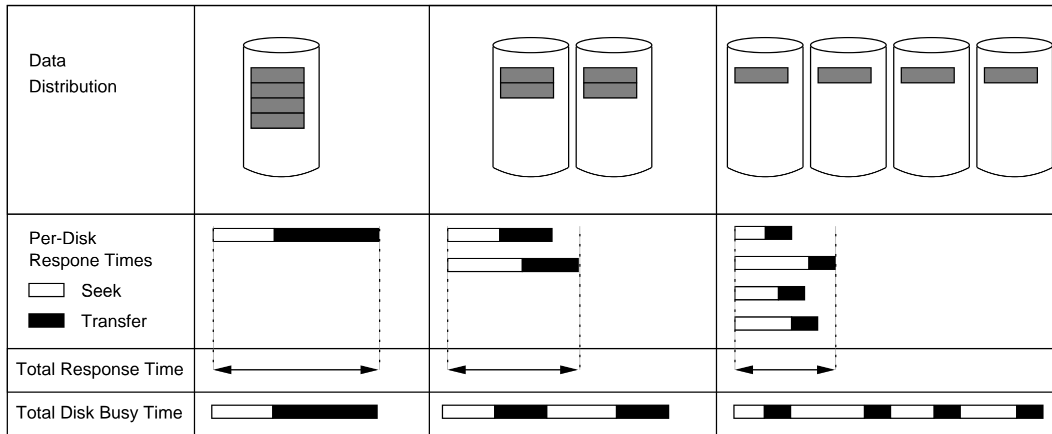

Figure 4.4: Influence of striping on response time and disk busy time

4.3 Declustering

Enabling the Autonomous Disks to provide declustering was a central part of my work, therefore other research efforts in this field were of course also relevant. Papers on declustering usually describe only certain strategies of distributing the data, not ways to manage the distributed data; these tend to be too integrated into the overall system design to be generalizable and are therefore found in papers that describe such designs (like the one in section 4.2.1).

However, even though I only implemented two unsophisticated declustering strategies that are not based on any particular paper, taking into account available declustering strategies was important, since looking at them provides requirements that a declustering framework such as the one described in sections 5.3 and 5.4 has to meet in order to support a variety of declustering strategies.

4.3.1 One-Dimensional Data

Stripe Size

In [15], Scheuermann, Weikum and Zabback present a model for data partitioning across disks that takes into account response time, throughput and load balancing. I did not consider the load balancing aspect, because the Autonomous Disks already provide load balancing. However, their work clearly shows that the *stripe size*, the unit into which data is divided when declustering it, is an important tuning parameter. There are two main considerations that form a tradeoff:

- The stripe size should be small enough so that most requests involve all disks, to benefit from the aggregated transfer rate of the disks - the main advantage of declustering. Thus, the average request time should be taken into account.

- On the other hand, a smaller stripe size leads to increased overall disk busy time per request, due to disk seeks. This reduces the throughput when there are multiple requests being served at the same time.

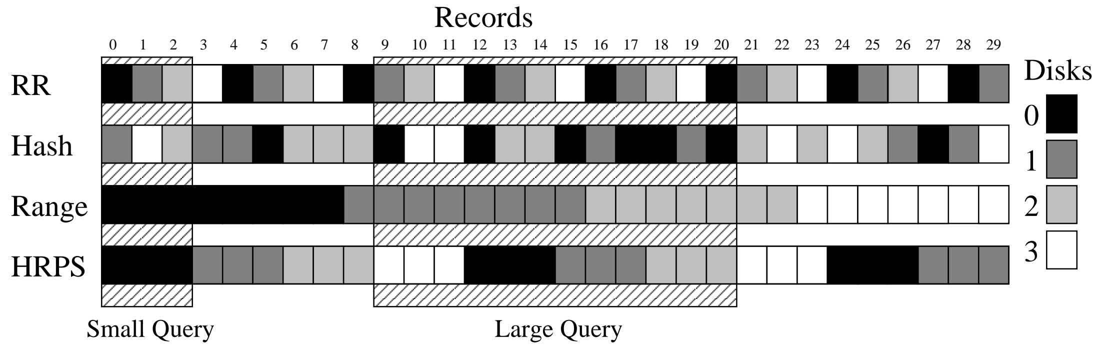

Figure 4.5: Declustering of record-based data: Round-Robin, Hash, Range and Hybrid-Range declustering

This tradeoff is illustrated in figure 4.4. The importance of the right stripe size was taken into account when designing the Autonomous Disks declustering in such a way that the stripe size for each stream can be chosen separately, as described in section 5.4.

Hybrid-Range Partitioning

In [14], Ghandeharizadeh and DeWitt proposed their Hybrid-Range Paritioning Strategy for declustering in multiprocessor databases. They observed that for record-based data, hash-based and round-robin declustering has the result that even small range queries have to be processed on all PEs, incurring a relatively large parallelism overhead. On the other hand, simple range partitioning has the disadvantage of confining even very large range queries to just one or two PEs, so that no intra-query parallelism is used and the response time is long.

They therefore proposed a declustering strategy in which the data is split into relatively small fragments, each containing records with a small range of the index attribute. These fragments are then distributed across all the disks using a round-robin scheme. The result (see figute 4.5) is that small range queries are confined to one or a few PEs, minimizing overhead, while larger ones use all disks to improve the response time. The size of the fragments can be optimized, taking into consideration the average resource requirements and size of queries to achieve the distribution that matches the workload best.

The Autonomous Disks (or rather, the Fat-Btree) use a range partitioning with flexible ranges for the main index attribute, as described in section 2.2. However, individual streams may have an internal database-like structure, such as the files used by Berkeley DB style databases. Even non-database streams may frequently experience "range queries", i.e. accesses to a certain limited range within the stream (for example displaying a short excerpt of a video stream).

Therefore, it is beneficial to use a hybrid-range partitioning for the declustering of individual streams in the Autonomous Disks, and the approach taken by me (see section 5.3) is quite similar. Of course, record-based hash or round-robin declustering is not possible at all with generalized, unstructured data streams, so I chose hybrid-

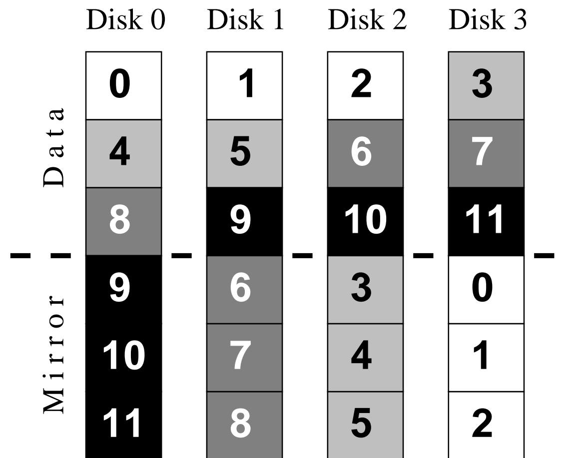

Figure 4.6: The RAID-x Orthogonal Striping and Mirroring scheme

range declustering over a simple range partitioning scheme. In addition to the possible performance-related advantages, this was also chosen because it allows the skew handling to be more fine-grained (see section 3.3 and figure 3.2).

Raid-x

In [16], Hwang et al. presented RAID-x, a new distributed RAID architecture. The distinguishing features of their system are:

- A block distribution method for achieving both fault tolerance and high throughput for reading and writing called Orthogonal Striping and Mirroring (OSM). Data blocks are striped across all disks, but backup copies are grouped together and written sequentially, as shown in figure 4.6.
- Cooperative Disk Drivers (CCD), an architecture for establishing a transparent single I/O space across the nodes of the parallel RAID cluster.
- A disk configuration in which each node of the cluster has several disks, forming a two-dimensional disk array that allows both parallel and pipelined access.

It is not clear how much the integration of these features is necessary, or whether they could be used independently from each other. The result are aggregated read transfer rates comparable to those of a RAID-5 configuration, and just like RAID-5, RAID-x can tolerate a single disk failure. However, because there is no parity calculation, write performance is much better than RAID-5. The drawback is that RAID-x requires 50% data redundancy, but tolerates only a single disk failure.

In regard to the Autonomous Disks, the OSM distribution might be beneficial, but [16] does not compare its performance to RAID-0/1 (combined mirroring and striping), which is a common configuration and very similar to the way the Autonomous Disks declustering operates. In general, the RAID-x design is intended for block-level access

and it would not be easy to apply it to an object-oriented architecture, especially with regard to the skew-handling-induced data migration of the Autonomous Disks.

The Cooperative Disk Drivers, on the other hand, are quite similar to the way the Autonomous Disks operate: a serverless cluster of nodes forming a single data repository towards clients. However, since they provide only block-level access and neither skew handling nor dynamic reconfiguration, their job is much simpler.

4.3.2 Two-Dimensional Data

In the 1990ies, several research efforts produced a continuous evolution of declustering schemes that achieve good response times for range queries on (mainly) two-dimensional data, such as the retrieval of all available information on an area of a map selected by a user. The problem here is to distribute the information across the disks in such a manner that — as far as possible — each possible range query results in all disks being used equally.

Unfortunately, Abdel-Ghaffar and El Abbadi showed in [18] that such a *strictly optimal* scheme exists only under very narrow conditions[12], namely if either M, the number of disks, is 1,2,3 or 5, or the number of tiles in all but one of the dimensions is only 1 or 2. However, the aforementioned evolution has yielded methods that come closer and closer to a strictly optimal declustering.

Disk Modulo (figure 4.7)

The first such scheme was the *Disk Modulo* scheme introduced in [19] and extended in [20], in which the data is divided into tiles and tile (x, y) is assigned to disk number $(x + y) \bmod M$. This is applicable for any M, but the performance is lower than all the other methods.

Field-XOR (figure 4.10)

Another method is the *Fieldwise Exclusive-or* scheme, in which tiles are assigned to disk number $(x \otimes y) \bmod M$ (the binary representations of the tile numbers are combined through a bitwise exclusive-or operation). This achieves better performance, but it requires M to be a power of 2 and elaborate correction techniques if the number of tiles is smaller than M in one of the dimensions.

[12] However, this limitation is true only for the chosen partition of the index space into rectangular tiles, a choice that seems obvious, but has severe drawbacks, as discussed in section 4.3.3

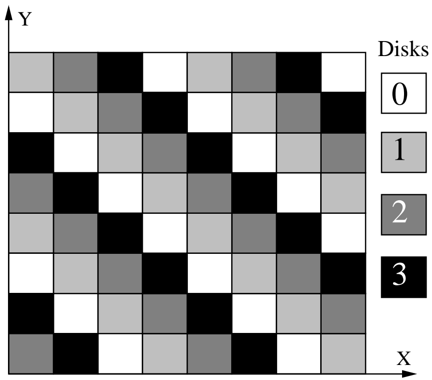

Figure 4.7: Disk Modulo declustering

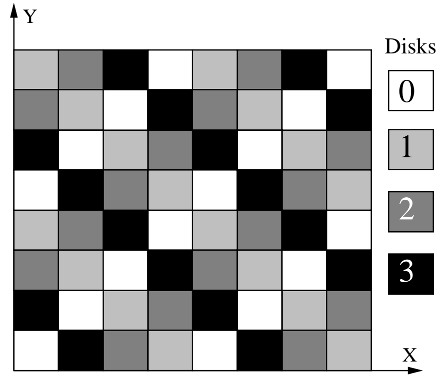

Figure 4.10: Field-XOR declustering

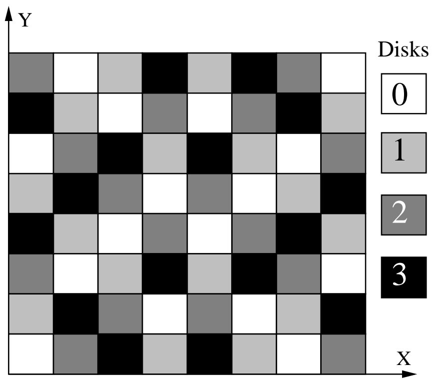

Figure 4.8: ECC declustering

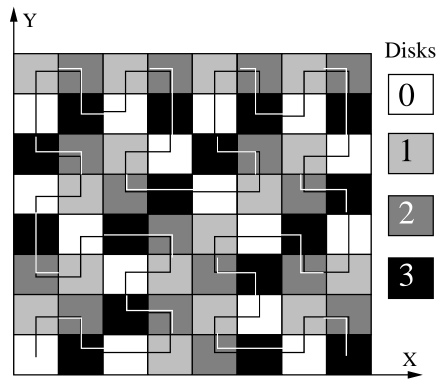

Figure 4.11: HCAM declustering. The convoluted line is the space-filling Hilbert curve.

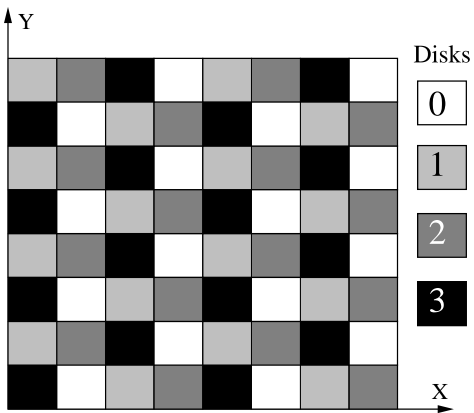

Figure 4.9: Cyclic Declustering (skip value: 2)

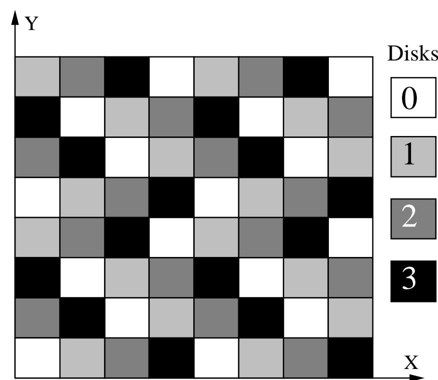

Figure 4.12: Golden Ratio Sequences declustering.

Method	Tiles per dimension	Number of Disks
Disk Modulo	Any	Any
Field-XOR	Complications if < # disks	Power of 2
Error-Correcting Codes	Power of 2	Power of 2
Hilbert Curve	Any	Any
Cyclic Declustering	Any	Any
Golden Ratio Sequences	Any	Any

Figure 4.13: Limitations imposed by the various two-dimensional declustering methods

Error-Correcting Codes (figure 4.8)

Both of these methods are outperformed significantly (as shown in [23]) by the *ECC* scheme presented in [22], which uses error-correcting gray codes. The binary representations of the tile coordinates are grouped on the disks in such a way that those grouped on each disk form a gray code, with large Hamming distances between them[13], which should result in good declustering. The disadvantage of this method is that it needs both M and the number of tiles in both dimensions to be a power of 2.

Space-Filling Curves (figure 4.11)

HCAM, a declustering method that performs equal or better than ECC (but only for square or near-square queries) while not putting any restrictions on M or the number of tiles, was introduced in [23]. In HCAM, the tiles are linearized along a space-filling Hilbert curve, and assigned to the disks in a round-robin fashion along this linearization.

Cyclic Declustering (figure 4.9)

Cyclic Declustering, a generalization of the Disk Modulo method, was presented in [24]. It places tile (x, y) on disk $(x + H \cdot y) \bmod M$, where H is a *skip value* that has to be chosen by some algorithm or heuristic to yield good results. Three such methods were presented: the first (RPHM) based on a simple algorithm to choose a skip value that is relatively prime to M. This performs better than all previous methods for most queries, but it strongly favors even values of M. A second algorithm is GFIB, which finds a better H relatively prime to M based on Fibonacci numbers. The performance is even better in nearly all cases. A final method relies on an exhaustive search of all possible values and simulation of their performance to find the value of H with the best result, but due to the computational overhead, this quickly becomes infeasible for larger values of M and larger maps. All of these methods put no restrictions on M and the number of tiles.

[13] i.e. they differ in many bit positions

Golden Ratio Sequences (figure 4.12)

The so far best declustering scheme for two-dimensional data was introduced in [25], involving somewhat nonintuitive but not really complex[14] computations based on Golden Ratio sequences. It outperforms even the (practically infeasible) exhaustive-search cyclic declustering in most cases, and imposes no restrictions. Moreover, it stands out as the only declustering method whose performance could be mathematically analyzed. The analysis suggests that the GRS scheme has a worst case response time within a factor of 3 of the (often not existing) strictly optimal scheme for any query, and within a factor of 1.5 for large queries, while the average performance is expected to differ from the optimal by only a factor of 1.14.

4.3.3 Higher Dimensional Data

All of the two-dimensional declustering schemes mentioned in section 4.3.2 can be generalized for a higher number of dimensions. However, as the number of dimensions increases, a silent assumption that all of these schemens make becomes untrue: the assumption that dividing the index space into square tiles yields good performance for declustering. There are several interrelated problems:

- The number of tiles increases exponentially with the number of dimensions this way. This means that it may become impractical to divide the space into more than two or three tiles in each dimension [15] because the large number of tiles incurs an increasing overhead in data management.
- Additionally, the number of tiles adjacent to any given tile also increases (though only linearly), which makes it more difficult or impossible to find an optimal declustering scheme for small queries. The non-existance of an optimal scheme mentioned in section 4.3.2 is proven only for rectangular tiling.
- Another problem is data distribution: the nature of the Euclidic distance metric has the effect that in higher-dimensional spaces, most data will be situated close to the surface of the index hypercube [16]. Eqidistant tiling is not well fit to deal with such a distribution.
- Even worse, not only the overall number of tiles increases exponentially with the dimension, but also the number of tiles adjacent to an intersection, which will have to be loaded for any query covering that intersction. In the worst case, a high dimensionality may require using only two tiles per dimension, with the result that even very small queries have a high probability of encompassing the single central intersection and thus requring *all* tiles to be loaded: for all hypercubes with a base

[14]and computationally feasible, even for large M and large maps

[15]Three tiles per dimension result in almost 60,000 overall tiles in 10 dimensions

[16]A 5-dimensional hypercube with a base length of 0.5 contains less than 4% of the volume of a unit hypercube

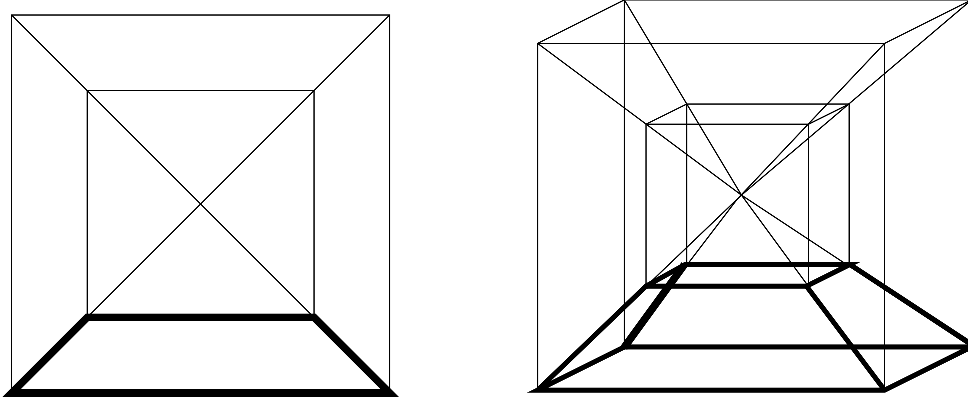

Figure 4.14: Hypertrapezoid pertitioning scheme for 2 and 3 dimensions, one hypertrapezoid tile highlighted

length over 0.5 (which may still be very small in terms of relative volume), this is actually a certainty.

After establishing in this way that equidistant rectangular tiling is not suitable for high-dimensional data, Ferhatosmanoglu et al. suggested in [27] an alternative partitioning scheme[17]. They divide the space not into rectangular tiles, but into concentric hypercubes, with a possible subdivision into hypertrapezoids by intersecting the concentric hypercubes with hyperpyramids formed by the center of the index space as the pyramids' tip and its surfaces as their bases. This solves all the problems mentioned above:

- Both the total number of tiles and the number of tiles adjacent to an intersection increase linearly instead of exponentially with the number of dimensions: a hypercube of dimension n has $2n$ surfaces, and there is one hyperpyramid for each surface.
- The number of hypertrapzoids into which each hyperpyramid is divided can be chosen freely and increases the total number of tiles by only a constant, not exponential, factor.
- The concentric hypercubes (and thus the hypertrapezoids) can be spaced non-equidistant to achieve good data distribution: the outermost ones should be very close together, since they will contain the most data.
- While the hypertrapezoid partitioning does not alleviate the problem of the increase of adjacent tiles making optimal declustering difficult, pure concentric-hypercube based paritioning makes it trivial to find an optimal declustering scheme[18].

[17] their work is heavily based on an earlier paper [26] by Berchtold et al.

[18] For example, a simple round-robin distribution advancing from the innermost hypercube outwards

However, the hypertrapezoid partitioning also has at least one disadvantage, which is not mentioned in [27]: even small queries close to a corner of the index space would tend to require almost half of the total data to be loaded.

Anyway, this method, as well as the two-dimensional declustering methods of section 4.3.2, are not directly applicable to the Autonomous Disks, since they use only a one-dimensional index. However, it would be feasible to offer limited support for an internal two- or multidimensional structure of streams, and offer special declustering methods for them. This was not implemented, but in section 7.3, a way to integrate this capability into the current code with small effort is outlined.

Since Autonomous Disks are intended as a general data repository, the ability to effciently handle multi-dimensional data is a valuable addition to the feature set, and the relative ease with which this declustering method could be added ahows the flexibility of the declustering framework.

4.3.4 Similarity Graphs

In [17], Liu and Shekhar present a very generalized method for finding a good declustering layout for a particular set of data and queries. Their idea is based on viewing the data items as nodes of a graph; the graph's edges represent the likelihood of two nodes being accessed together and are weighted accordingly. A heuristic method is used to partition the graph over the available disks in such a way that data items that are frequently accessed together are placed on different disks as much as possible, so that they can be retrieved in parallel. An incremental placement scheme for the gradual accumulation of data items is also suggested.

The obvious problem is that building the similarity graph in the first place requires a lot of knowledge about access probabilities that may not be available at all or may be costly to gather or maintain. An example for a case where this is not a problem is the range query oriented two-dimensional data partitioning discussed in section 4.3.2. Assuming that all queries are equiprobable or that their probability distribution is known, the edge weights can be computed relatively easily. However, in general this question must be solved separately for any actual application of the method, so in reality it simply replaces the task of finding a good declustering scheme with that of finding meaningful probabilities of data items being accessed together. The advantage is that the latter task seems to be more straightforward to solve in most cases.

Another disadvantage is that graph partitioning is a costly (in terms of CPU usage) operation, though this is partially alleviated by modern, powerful CPUs and the incremental placement method that uses only a subset of the graph to find a local optimum placement.

In case of the Autonomous Disks, I chose not to implement this method because the disks themselves do not generally have the means to provide the edge weights, and clients should be able to use the Autonomous Disks as easily as possible. Additionally, the similarity graphs partitioning does not yield a function-based partitioning, so the location of each data item has to be recorded as metadata, resulting in a possibly large amount of metadata, something I deliberately wanted to avoid (see section 5.4).

5 Design

Design changes and extensions that allowed the treatment of large data items, as well as some supporting functionality, were the core contribution of my project. In this chapter, the changes and considerations that motivated them are explained.

5.1 Allocation and Deallocation

The first extensive programming work done in this project was the implementation of a fully featured *allocation subsystem* to manage the disk space on each Autonomous Disk. Figure 5.1 describes the "paths" of space in the resulting system.

5.1.1 Main Data Structures

To attain the goals described in section 3.2 (fast allocation, low fragmentation), the "address-ordered best fit" allocation policy which performed so well in the fragmentation benchmarks mentioned in section 4.1.1 was chosen. For an efficient implementation, the approach used in XFS (see section 4.1.4) was adopted.

The main data structures are two B+ trees[1], both of which contain all extents of continuous free space.

- One tree is indexed on the size of the extents, with the starting address as secondary key (to make each entry unique). This is used directly to implement the "best fit" policy, as it easily allows to find a free extent that satisfies a request.

- The second tree is indexed on the starting address. It is used for coalescing free extents with their neighbours if possible, and also for clustering allocation (see section 5.1.4)

Generally, an allocation request is served by looking through the size-indexed tree for the smallest free extent that is large enough to satisfy the request, and removing it from the trees. If the extent does not fit the request exactly (i.e. it is larger than necessary), it is split and the excess space is reinserted into the trees as a new free extent.

[1] Unrelated to the Fat-Btree, the Autonomous Disk's main index, which has a much less efficient implementation, partially because it is distributed across several disks.

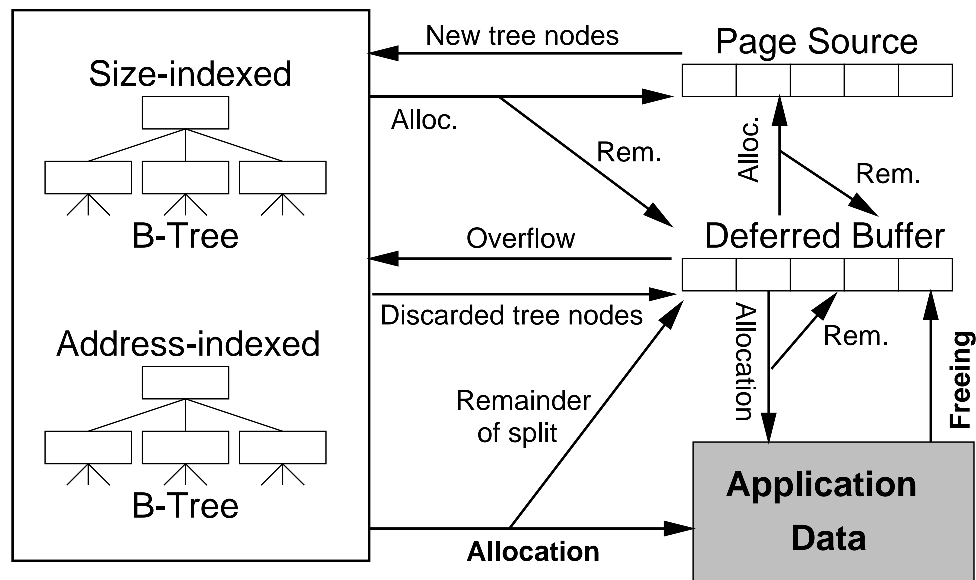

Figure 5.1: "Space paths" in the allocation subsystem

When space is freed, the address-indexed is used to find out whether there are free extents directly adjacent. If one is found, it is removed from the trees and the two or three[2] extents are coalesced into one, which is then inserted into the trees.

Thanks to the B-trees, this is a highly scalable implementation that makes it possible to find the best fit even on a very large and highly fragmented disk with hundreds of thousands of free extents. However, there are some additional complications.

5.1.2 Internal Allocation

The algorithm outlined above fails to take into account one important factor: the B-trees in which the free extents are stored also take up space and must be stored on the disk. Since their size is highly variable, they cannot be put in preallocated space. Instead, the tree nodes must be dynamically allocated and deallocated just like the application data.

This is a classical "hen-egg problem": the tree nodes *cannot* be allocated in the same way as application data, because when a tree node is allocated or deallocated, it means that the tree is in the process of being modified, therefore possibly in an inconsistent state, and thus cannot be used for another allocation or deallocation. This is solved by using two additional data structures: a page source and a page sink.

Page Source:

In abstract, an object which yields free pages for use as tree nodes when desired. It is implemented as a small buffer that holds enough pages (or their addresses, to be

[2]if there were free extents on both sides of the one being freed

exact) to satisfy the need for new tree pages during a single deallocation operation[3]. The maximum number n of pages that might be needed can be estimated as follows:

$$n = 2 \left\lceil log_b \left(\frac{p}{2} \right) \right\rceil$$

where p is the number of pages on the disk, divided by 2 to get the number of free extents in the worst case of maximal fragmentation. We take the bth logarithm of this, b being the minimal branching degree of the trees (determined by the node size and the fact that a node is half full in the worst case), to get the maximal height of a tree. This number is rounded up to give the maximal number of new pages necessary when a node split in a tree of maximal size propagates up to the root, and multiplied by 2 to account for the presence of two trees. To give a practical example: on a 100GB disk using 4KB pages, p is about 26,000,000 and b is 127. In this case, n is 8.

When the reserve becomes too small, it is refilled. In order to reduce the overhead, the buffer actually holds up to $2n$ pages, and is refilled by requesting n consecutive new pages only when it's half-full. Since the actually required n is quite low for normal cases (as seen above), it might be advisable to use a larger number to reduce overhead further, decrease fragmentation and increase locality by keeping the tree nodes in somewhat larger clusters.

Page Sink:

Does the opposite job of the page source by accepting pages from discarded tree nodes and inserting them into the trees not immediately but later, when the allocation or coalescing request[4] is finished.

Page sink and page source could theoretically be the same object, one which reuses discarded node pages, but it still would have to use the real allocation subsystem for both allocation and deallocation since the trees' overall size might grow or shrink drastically in some situations.

5.1.3 Deferred Coalescing

Deferred Coalescing is a technique mentioned in section 4.1.1 that seeks to speed up the allocation process by keeping a "deferred buffer" with fixed maximum size, containing recently freed extents in memory without coalescing them. The list is searched before resorting to the normal allocation mechanism.

Since the "page sink" described in section 5.1.2 already implements this by necessity for pages used by the allocation subsystem's B-trees, it was an easy decision to extend it to do general deferred coalescing. Thus, all freed extents are put into this "deferred buffer" first, which acts as a FIFO queue, inserting the oldest extents in it into the trees when its designated size is exceeded.

[3] The trees will grow only during deallocation, when a new free extent is inserted

[4] Nodes will be discarded during allocation *and* coalescing, when extents are deleted from the trees - in the latter case only temporarily.

This size of the buffer directly determines how effective the speed gains of deferred coalescing will be; if it's very small, there will be no effect, but if it's too large, the sequential search might also have a negative effect, and it can increase fragmentation if it keeps too much free space uncoalesced.

Another question is what to do if a search of the buffer yields a free extent larger than the requirement of the allocation request. One possibility would be to use this extent immediately and not look through the trees. Obviously, this is the fastest option. However, it would mean that the allocation policy would partially turn into "LIFO first fit"[5], a policy that exhibited an intolerably high fragmentation in [7]. Therefore I instead chose to look through both the deferred buffer and the size-ordered B-tree (except when the buffer yields an exact fit) in order to implement a "best fit" policy.

5.1.4 Clustering Allocation

In [13], the capability of the B-tree based allocation mechanism to provide "clustering", i.e. to use the address-indexed B-tree to find free space near a given address is stressed. The benefit is that data that is likely to be accessed together can be kept close together on disk to reduce seek times. This capability was not really necessary for the Autonomous Disks and is unlikely to be of much use except maybe for the Fat-Btree itself, but adding it to the allocation subsystem was a relatively simple task, since the data structures support it inherently.

5.2 Arbitrary-Length Streams, Random Access

The prototype had to be modified to enable it to handle streams of any size, which was achieved by adding a length field to the index entries and allocating the necessary number of disk pages.

But this was not sufficient. For some of the possible applications, especially multimedia data, stream sizes could easily reach into the gigabyte range. As described in section 2.1.4, the prototype implementation relies on transferring each stream as a whole before starting to process it, and during processing, it actually needs to be copied several times. For large streams, this would require an impossibly high amount of main memory. Additionally, all other requests would have to wait for the large stream to be processed, resulting in intolerably high response times.

To solve this problem, it is necessary to transfer large streams in smaller pieces, small enough to comfortably process several of them in main memory and cause no undue delays for other requests. This could have been achieved with a sequential model, in which the pieces of a stream are always transfered in order, starting at the stream's beginning, and this option was considered and partially implemented, as mentioned in section 5.5.3. However, at the price of an only slightly more complex interface, it was possible to allow random access to any point within a stream, without needing to transfer the entire stream.

[5] This means using the first free extent that is large enough, looking at recently freed extents first

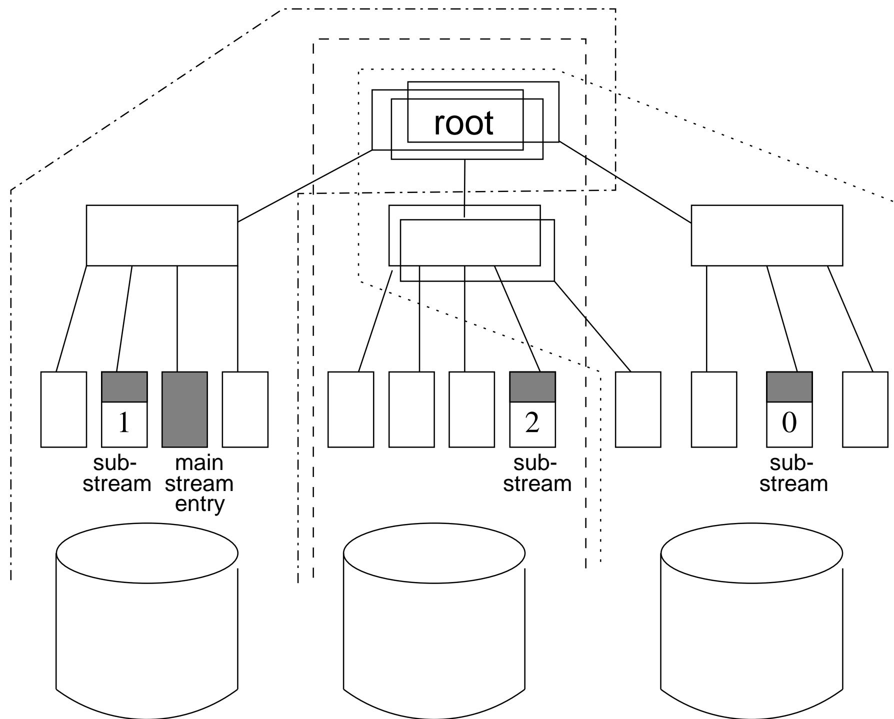

Figure 5.2: Stream and substreams in the Fat-Btree index

This is a very desirable capability. It might be argued that it is unnecessary because if the stream has an internal stucture that results in partial accesses, the parts should be stored separately in the index. However, the argument does not hold, because the structure and access patterns may not lend themselved to indexing very well. As an example, consider a video stream. Users will sometimes want to skip to a particular point and watch only a short scene. The preferred scenes will vary between users, so they cannot be used in an indexing scheme.

An additional advantage of the random access capability was that it allows stateless handling of declustered streams, as described in the following sections.

5.3 Management of Declustered Streams

An important question is how to store and access declustered streams in general. If the solution is to be a part of the Autonomous Disks at all, the only logical solution is to treat accesses to a declustered stream exactly like those to a normal stream, through a normal entry in the index. This is the only method that retains transparency and the Autonomous Disk features, especially the scalability of the Fat-Btree index. Unfortunately, it means that that the superstream entry can be somewhat of a bottleneck, although it requires no data-related disk accesses.

What distinguishes declustered streams from non-declustered ones, is the "type" field in the index entry (other types are non-declustered streams and index nodes). Each

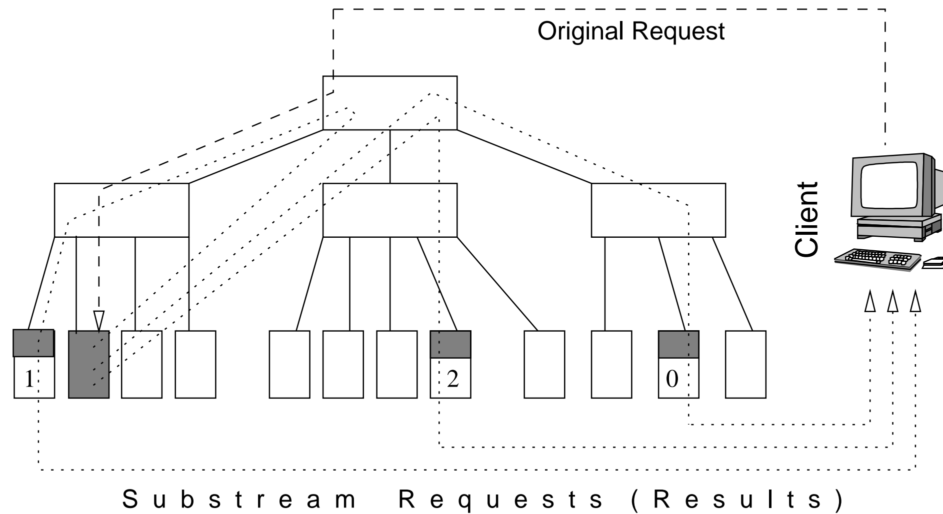

Figure 5.3: Access to a declustered stream

declustering method described in section 7 has its own type number. Furthermore, the entry must store the information necessary to locate the substreams so that accesses to parts of a declustered stream can be redirected to the appropriate substreams as described in figures 5.2 and 5.3. See section 6.3 for details.

5.4 Management of Substreams

5.4.1 Placement of Substreams

The classical way to manage substreams would be to store them separately from the main index and keep a record of their disk location. However, this would run counter to the main feature of the Autonomous Disks: the transparent on-line load balancing through the Fat-Btree index. It would either disable skew handling almost completely for the declustered streams, or require an additional skew handling mechanism. Of course, the declustering itself already prevents a data distribution skew[6], but it cannot prevent and access skew if there are "hot spots" inside a stream that are more frequently accessed than the rest of the stream. Thus, the absence of skew handling is not acceptable, and developing a separate skew handling mechanism for declustered streams would complicate the design unnecessarily.

Additionally, disk location based declustering is not feasible, because the asynchronous update operations used by autonomous disks cannot guarantee that the same disk blocks will be allocated for the backup copy. Thus, the primary substream location record could not be used to find the backup copies of these substreams, which would defeat the fault tolerance of Autonomous Disks.

Therefore, the substreams of a declustered stream are placed in the Fat-Btree index

[6]or not, depending on method, see figure 7.1

just like normal streams. The placement of the substreams is then dependant solely on the choice of name for them, so a declustering method is nothing more than a method of generating names for substreams, which is a simple and straightforward task.

5.4.2 Finding Substreams

Another question is: how to re-generate the substream names for each access? The obvious way would be to store them as a list with the superstream, in a form of indexed allocation similar to the inodes used in many current Unix filesystems. This would obviously be the most flexible way, and could even allow the relocation of individual substreams for skew handling. However, it would again complicate the design, and since the number of substreams is potentially huge, performance could also suffer.

For these reasons, I instead chose to implement methods that allow the substream names to be re-generated each time they are needed, and which require only a small amount of persistent storage for meta-information that makes this consistent. A generalized interface makes it very easy to add new declustering methods, as long as they work within the mentioned constraints[7].

5.4.3 Flexibility

Additionally, the size of substreams can be chosen individually for each declustered stream. This size (often referred to as "stripe size") is an important tuning parameter for declustering, and its optimal value depends both on the average request size for that particular stream and the system load, as described in section 4.3.1, so it is beneficial not to use a fixed value.

Many declustering methods are developed to be efficient for a particular kind of data (such as those described in section 4.3.2). Thus, it is desirable to allow different declustering methods to coexist in an environment, so that clients can choose the one that works best with their data. This is one of the core features of the system presented here. How it works in detail is explained in section 6.4.

5.5 Alternatives

This section presents some alternative design ideas that were considered at one point or another in the project, but discarded for various reasons.

5.5.1 Allocation Mechanisms

When planning the allocation subsystem, several ideas were considered before settling on the B-tree based system. The alternatives were a buddy system (see section 4.1.3), which would have offered simple and fast allocation and deallocation, or a system based on a

[7] placement through name generation based on stream name and offset, only a small amount of persistently stored metadata

(possibly hierarchical) bitmap, which would have made coalescing of free extents trivial. A hybrid of these was also considered. However, all of these methods had unacceptable disadvantages: Buddy systems show very high fragmentation, bitmap-based systems require sequential scans, and a hybrid would have been very complex or even impossible to implement properly.

5.5.2 Storage Outside the Index

When designing the allocation subsystem, I planned to store substreams of declustered streams outside the Fat-Btree index, or possibly in a separate index. The performance penalty for having to go through the main index twice[8] for each access seemed too large. However, further considerations exposed many weaknesses in this concept, mainly the fact that it could not retain all the features of the Autonomous Disks (see section 5.4.1 for an explanation). For this reason, I decided to use the main index for the substreams, and the benchmarks in section 8.1 show that performance is not really a problem.

5.5.3 StreamManager

A preliminary concept, which was also implemented partially, required the "opening" of streams to allow subsequent accesses to bypass the Fat-Btree index, providing leightweight in-memory access information for open streams[9] through a class called `StreamManager`. However, it was abandoned because it would have meant a more complex communication protocol between the Autonomous Disks and clients, and because the index nodes for a repeatedly accessed stream are very likely to be cached in memory, alleviating the performance impact of traversing the index every time.

[8] Once to find the the superstream entry, a second time for the substream
[9] similar to the file descriptors used in many operating systems

6 Implementation and Interfaces

In this chapter, details of how the design decisions described in the previous chapter were implemented, and the interfaces between those parts and the rest of the system are described. Hopefully it will answer any questions the previous chapter left open, and help future developers of the system.

6.1 Allocation

6.1.1 Package Structure

The allocation subsystem is structured into two Java packages, both subpackages of `jp.ac.titech.cs.de.AutoDisk`, following the widely used package naming convention based on internet domains. The packages are:

`Basic` Contains supporting classes that do not directly provide allocation. The B-tree and hardware interfaces are found here.

`Allocation` These classes provide the actual allocation and deallocation functionality.

6.1.2 Hardware Interface

In order to provide a general allocation subsystem, there has to be a clearly defined interface to the underlying hardware, using functionality that any storage hardware should be able to provide. It is defined in the Java interface `PageDevice`, which assumes that the hardware is block-oriented (i.e. the unit of access is a fixed-size disk block or page) and that blocks are addressed linearly. The interface requires the following methods:

- `int pageSize()`
  Returns the size of the access unit / page.

- `void close()`
  Releases all resources associated with the device (e.g. cache memory).

- `long pageNumber()`
  Returns the amount of storage that the device offers.

- `byte[] read(long address, byte[] data, int offset)`
  Copies one page of data from the device into a an array in main memory, beginning at a specified offset in the array.

- `void write(long address, byte[] data, int offset)`
  Copies one page of data from a specified offset inside an in-memory array onto the device.

The allocation subsystem, and therefore the Autonomous Disks can run on top of any device that can be accessed in this way. For the prototype, an implementing class based on the class `RandomAccessFile` in the standard package `java.io` was used.

6.1.3 System Interface

The unit in which space is allocated and deallocated is an *extent*. An extent is a data structure that simply describes the starting address and the size (in disk pages) of the space. The `Extent` class holds these as (`long`) member variables.

The interface through which the Autonomous Disks[1] access the allocation subsystem is defined in the Java interface `Allocatable`. It extends `PageDevice` and requires the following additional methods:

- `void setRoot(int index, long address)`
  To be useful, storage must be organized in some sort of index. This method allows to set the "entry points" or root addresses of one or more indexes on the device. The information is stored in a master disk page which also contains meta information about the allocation subsystem.

- `long getRoot(int index)`
  This method is used to retrieve any of the index root addresses.

- `long freePages()`
  Returns the number of free (unused) disk pages on the device.

- `Extent allocate(long size)`
  Attempts to allocate the given number of disk pages and returns an extent that describes the allocated space. The method fails if there are not enough contiguous free pages left.

- `void free(Extent extent)`
  Returns the given extent to the free space pool.

- `Extent allocateNear(long base, long size)`
  As described in section 5.1.4, this method tries to allocate space as close as possible to the given address, using the leaf node chaining of the address-ordered B+ tree (see section 5.1.1).

[1] Or any other system that wants to use the allocation subsystem

- `void setNearSteps(int steps)`
  This sets the number of entries in the tree through which the above method looks before giving up.

The interface is implemented in the class `AllocatableDevice`, using the data structures described in section 5.1.1. Using it required only minor modifications to the `DataManager` and `PageIO` classes (see section 2.1.2).

6.2 Random Access Within Streams

6.2.1 Downwards-compatible Interface

In order to allow large streams to be stored in the Autonomous Disks, it was necessary to transfer them in smaller units (see section 5.2). This in turn required an extension of the interface used in the EIS commands (see section 2.1.3), adding "offset" and (for read requests) "length" fields. However, these fields are not of interest to some parts of the system (such as the logging facilities), and in order to avoid bloated method signatures[2] and retain downwards compatibility for existing code, I chose to make them part of the `StreamID` field.

The field is of type `String`, and there are a number of characters which may be part of a string that are not properly usable for a StreamID, most prominently newline characters. Thus, I defined the `StreamID` to contain several fields (optional except for the first), divided by newline characters:

1. The actual StreamID
2. The offset within the stream where the read or write request or reply starts.
3. The length field, which has two possible interpretations:
   - For read requests, it specifies the amount of data to be read; this can be very large, since it can be served by responses that each carry only parts of the data (with corresponding offsets).
   - In case of a write request, the amount of data is determined by the accompanying data field, while the length field specifies the *total* length of the stream. This is necessary so that when the first write request for a new stream arrives, the disk can allocate enough space to hold the entire stream.
   - For replies to read requests, the length field is unnecessary and can be left out.

In this way, previously written client code, which transfers only entire streams, still works unchanged, and additional fields for declustering can be added easily (see section 6.3).

[2] Especially since the declustering funtionality requires even more fields.

6.2.2 Division of Requests

For read requests, the Autonomous Disks automatically process the request by reading and transferring smaller parts. The maximum size of these parts, the "maximum direct transfer unit", is currently a global parameter set in a configuration file, but it could also be automatically adjusted, perhaps according to the available main memory.

For write requests, it is the responsibility of the client to divide the request into sufficiently small parts, because the EIS commands are transferred as a whole. The client also has to process partial replies correctly; this is a non-trivial task since they can arrive out of order in case of a declustered stream where several disks deliver data in parallel.

6.3 Declustering Functionality

Providing the declustering functionality required several separate modifications and additions to the system. Each of them is not very complex, but they all have to cooperate and handle all kinds of special cases. These are the noteworthy changes:

- As mentioned in section 5.3, addition of new values for the type field in the Fat-Btree index entry, one (integer) value for each declustering method, and corresponding entries in the configuration file that specify the class which implements that declustering method (see next section).

- Implementation of the `decompose` IS command (see section 2.1.4) which redirects write requests for a declustered stream to the appropriate substream or substreams, according to the chosen declustering method.

- Implementation of the complementary `compose` IS command which redirects read requests for a declustered stream to one or more substreams. When doing this, it has to append additional "substitution" StreamID, offset and length subfields to the `StreamID` (see section 6.2.1) field in the read requests to substreams. These are necessary so that the replies to the requests (which go directly to the client) contain the correct ID, offset and length for the superstream instead of the substream.

- Modification of the ECA rule that handles write requests (see section 2.1.3) to recognize either an optional subfield of the `StreamID` field that determines the declustering method to be used for newly created streams, or a corresponding type value in the index entry for existing declustered streams, and to use the `decompose` command for such streams.

- Modification of the ECA rule that handles read requests to recognize declustered types in the index entry and use the `compose` command for such streams. Furthermore, it has to recognize and use the "substitution" StreamID, offset and length fields issued by the `compose` command.

- Modification of the logging mechanism (see section 2.1.5) so that it records *only* the first write request to a newly created declustered stream (which causes it to be created) without any data. The data writing will be logged when the substream writes are processed, and if the superstream writes were also logged, the data would be logged and backed up twice.

6.4 Intergration of Declustering Methods

At the core of the declustering framework, the environment for using various declustering methods had to be provided and the interface between those methods and the rest of the system defined.

6.4.1 Name Generation

As explained in section 5.4.1, substream placement is done through name generation. The most important part of a substream name is a short *placement prefix*, which determines where in the index (and thus, on which disk) the substream will be placed.

The prefix is followed by a sequence number and the superstream name to ensure that each substream name is unique and there are no "collisions".

Basically, there are two types of declustering methods. They differ in the amount of persistently stored information they require:

- Some methods may need no information at all, except for that found in the superstream index entry. Although they still need to store the substream size, it can be fit into the address field of the index entry, which is not necessary since there is no data directly associated with the entry. I call such declustering methods *nodeless*. They have the advantage that an access might require one less disk seek compared to a method that needs to access state information.

- In contrast, other methods may need state information to be stored persistently. These methods will have some space associated with the superstream index entry and use the information there to create the placement prefixes. The substream size also has to be stored there, since the address field is used to locate the state information.

The state information to be used for newly created streams may change with time, e.g. to reflect different disk ranges as a result of load balancing, but already created streams always have to use the same state information so that they always generate the same placement prefixes. For an example, see the RRD declustering method in section 7.2.

6.4.2 Interface

As stated in section 3.3, one goal of the project was to allow various different declustering methods to coexist and new methods to be added easily. To provide a clearly defined interface, the following classes are used:

- The abstract class `DistributionMethod`. All declustering methods must be subclasses of this. It defines only one method, `String getPrefix(String name, long offset)`, which provides the placement prefix for a substream of the given stream that begins at the given offset. Nodeless methods can directly extend this class.

- The abstract class `NodeDistributionMethod`, a subclass of the above which is used for declustering methods that need persistently stored state information. Its prefix-generating method takes the state as an additional argument, and it offers two more methods, one that generates the "current" state (used when creating a declusterd stream) and one that extracts the stripe size from that state information.

Adding a new declustering method to the system thus requires only an implementation that extends one of these classes, and an entry in the configuration file so that the system can find it. Then, it can be chosen as the declustering method for new streams (see section 6.3).

7 Declustering Methods

The declustering functionality described in sections 5.3 and 6.3 is the most important result of my project. However, it only provides a framework to be used by actual declustering methods. Those that I implemented or envisioned are described in this chapter.

7.1 Uniform Pseudo-Random Distribution (UPRD)

This method distributes the substreams uniformly across the index name space by creating pseudo-random placement prefixes. The pseudo-random function is seeded with a value based on the stream name and the offset of the substream within the stream, so that it acts as a hash function; this distribution method is nodeless (see section 6.4.1) and thus could be faster.

However, it has a severe drawback: it copes very badly with data distribution skews. In the worst case (as shown in figure 7.1), all of the previously present data could be concentrated in a very small portion of the index name space, so that only one or two disks would be designated for the entire rest of the name space, and all substreams of a UPRD-declustered stream would be placed on those one or two disks, which defeats the performance goal of declustering. Of course, the skew can be removed by the load-balancing mechanism inherent to the Autonomous Disks (see section 2.1.5), but this is unnecessary work. On the other hand, if a significant portion of the streams are declustered using the UPRD method, a large distribution skew is not likely to develop in the first place.

Another weakness of the UPRD method is that when there is only a small number

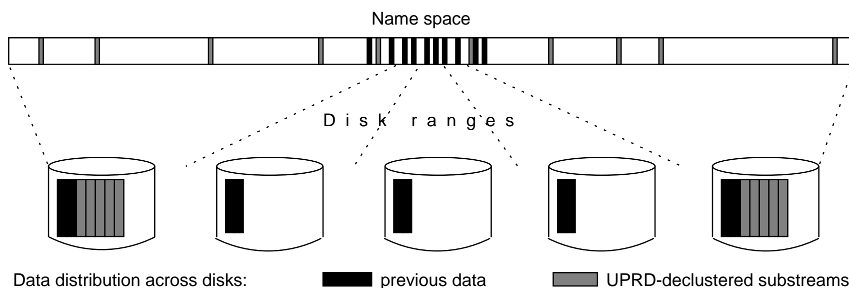

Figure 7.1: UPRD and data distribution skew (worst case)

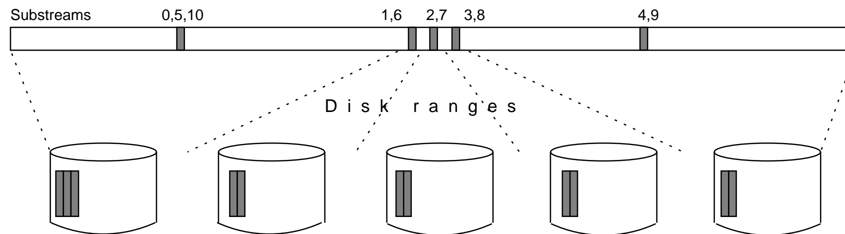

Figure 7.2: Substream placement using round-robin distribution

of disks in the cluster, substreams that are adjacent in the superstream will often be placed on the same disk, which reduces the performance benefit of declustering, but this problem decreases in reverse proportion to the number of disks.

7.2 Round-Robin Distribution (RRD)

Round-robin declustering is perhaps the most simple and obvious way of distributing a file across several disks. To achieve a round-robin distribution based on name generation, knowledge of the range of namespace that each disk spans is necessary, but that knowledge is easy to acquire, either by keeping a record of lexicographically maximum and minimum inserts on each disk, or by periodically querying the index. The information then has to be shared among all the disks along with the load and data distribution information that must already be shared to coordinate the skew handling described in the Fat-Btree concept [2]. Usually, this is done by passing around a token that contains the information for all disks, and each disk updates the information concerning itself before passing on the token. The range information is therefore not necessarily always up-to-date, but this is not a problem as long as the same information is always used for each stream. Since skew handling was not yet implemented in the prototype, I could simply use the fixed range information derived from the configuration file.

Based on the range information, a placement prefix in the center of each disk's range is chosen. In order to be able to recreate the substream names even after the disks' name ranges have changed (due to skew handling-induced migration of data), the placement prefix for each disk must be persistently stored in a disk page allocated for the superstream. Since the sequence numbers can easily be computed, only one prefix for each disk must be stored, which results in a small space requirement that is independant of stream length.

Of course, such migration can also affect the declustered substreams themselves, and some of them that were previously placed on different disks may then be placed on the same disk. This could reduce the performance gain of declustering, but should not be seen as a fundamental flaw, for the following reason: migration is done only to improve the global load situation, and an improved global load balancing is more important than having optimal declustering for individual streams. Generally, it is not possible to predict future access skews, therefore any distribution method will suffer decreased performance when an access skew occurs.

However, the migration of declustered substreams could be delayed by using the median of the disk ranges for the placement prefix instead of the center. The performance decrease could be partially alleviated by placing subsequent streams not on adjacent disks, but instead skipping one or more disk in each step of the round-robin distribution, so that they are less likely to end up on the same disk.

In the benchmarks in section 8, RRD performed better than UPRD in nearly all tests, so it should be used preferentially, unless there is a specific reason to believe that UPRD would perform better for a particular application.

7.3 Multi-dimensional data

Note: these are theoretical suggestions, the described declustering scheme was not implemented.

In sections 4.3.2 and 4.3.3, declustering methods for two and more dimensions are surveyed. These cannot be used directly for the Autonomous Disks, since the main Fat-Btree index (see section 2.2) is not a multidimensional index.

However, as explained in section 6.2, offsets within streams are transmitted as variable-length strings. This would make it possible to offer limited support for multidimensional structure *within* a stream and use the above-mentioned multidimensional declustering methods for such streams in the following way:

1. The multidimensional space is divided into tiles, rectangular or hyperpyramid (see section 4.3.3).

2. The data in each tile is aggregated and the tiles distributed across the disks using some sort of mapping (see sections 4.3.2) and placement prefixes as with the RRD.

3. In queries, the offset and length fields (see section 6.2 for the current situation) are subdivided to contain multidimensional values so that multidimensional instead of onedimensional ranges can be specified.

Unfortunatley, there are some severe drawbacks: First, identifying the tiles covered by a query is a potentially complex task that would have to be performed in the ECA retrieve rule in the current design (see section 2.1.2 for the internal structure of the Autonomous Disks), which is not a satisfying solution, since the rule is already rather complex.

The most logical solution would be to make the dividing and issuing of substream requests a part of the "Distribution Method" module, which would require making the interface between the Autonomous Disks and the distribution methods (described in section 6.4) far more complex.

But even then, offering transparent *write* access to such a stream would be extremely difficult if not impossible, because there is no logical way to divide the data that arrives with a ranged multidimensional write among the tiles that the range covers. The underlying problem is that, while a multidimensional *location* can be specified, the data itself

is one-dimensional and unstructured, and solving it would require a much more complex system. Most likely, some sort of multidimensional index structure would be necessary[1].

Unfortunatley, Berchtold et al. suggest in [26] that with increasing dimensionality, index structures in general become less efficient and eventually inferior to sequential scans, so there may not be a good solution to this problem at all.

To conclude: multidimensional range queries can be supported efficiently using declustering, but transparency is attainable only for reading, while insertion, modification or addition of data is probably not possible without detailed knowledge of the declustering scheme and the structure of the Autonomous Disk cluster.

[1] This would also be the only way to deal well with data distribution skews, which the scheme described so far does not

8 Performance

Theoretical work is only half the scientist's job at most. Its results always have to be tested under realistic conditions to verify their applicability and expose mistakes and factors that were not taken into accout. This chapter provides some experimental results on the performance of the declustering framework.

8.1 Transfer Rate

The performance of the two declustering methods with a growing number of disks is seen in figure 8.1. A stream of 90 MB size was declustered using a substream size (40 KB) equal to the maximal transfer unit and then read back in its entirety, measuring the time until the entire stream was retrieved. The results of 40 such operations were averaged.

Surprisingly, there is no performance increase with more than two disks, even though enough network bandwidth would be available in the Gigabit Ethernet setup. Instead, the bottleneck actually lies with the client CPU, namely the Java networking code, which was simply unable to receive data quickly enough, even though it was discarded immediately after reception.

So in order to see whether declustering gives performance gains, the amount of data had to be reduced artificially. This was done by modifying the data retrieval code in the Autonomous Disks to let it pause one millisecond before the delivery of each reply EIS command to the client.

The results seen in figure 8.2 are, of course, much lower transfer rates, but they exhibit an almost perfect linear increase with the number of disks, which is exactly the desired and expected benefit of declustering.

The RRD persistently shows a slightly higher performance, probably a result of the fact that it never places adjacent substreams on the same disk.

8.2 Jitter

This benchmark used the same configuration as in section 8.1; in fact, the measurements were performed along with the transfer rate measurements on the same operations. However, here the time difference between the arrival of subsequent substreams at the client was measured — the substreams can obviously arrive out of order when they come from different Autonomous Disks.

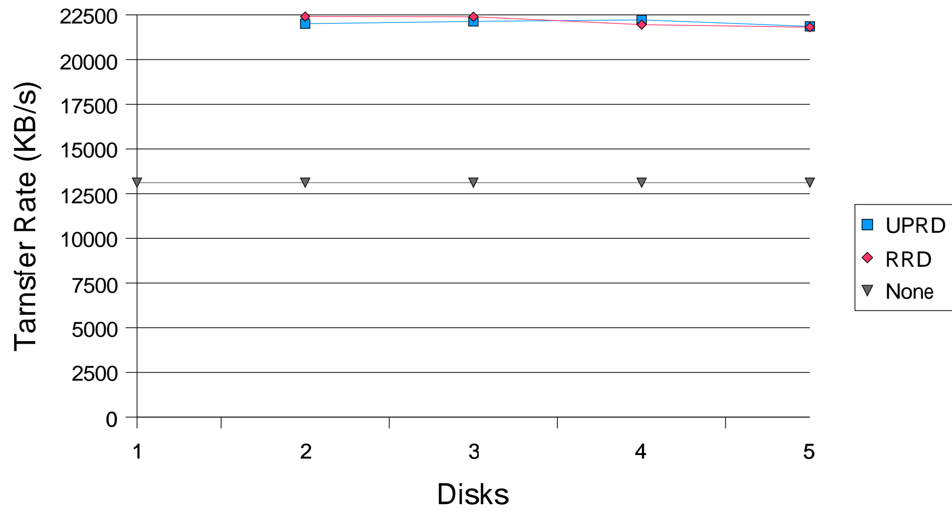

Figure 8.1: Transfer rate bottlenecked by client CPU

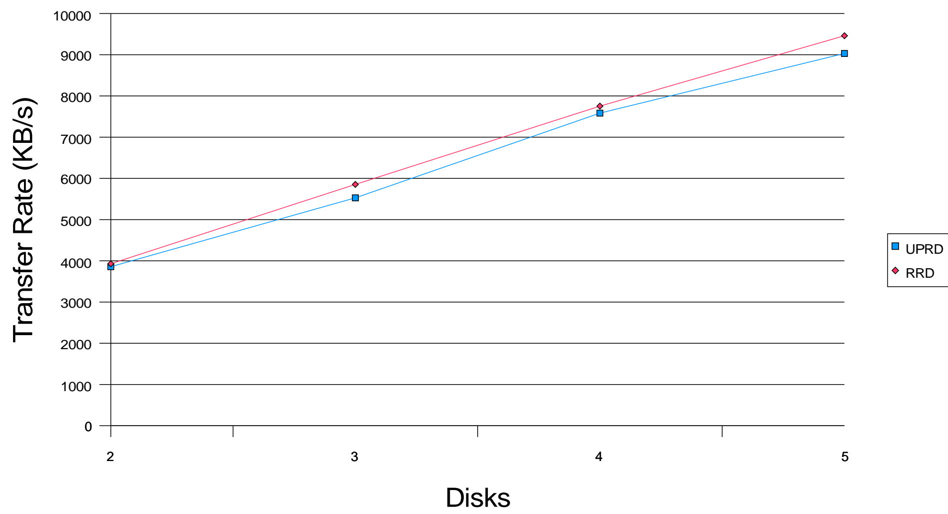

Figure 8.2: Transfer rate with artificial slowdown

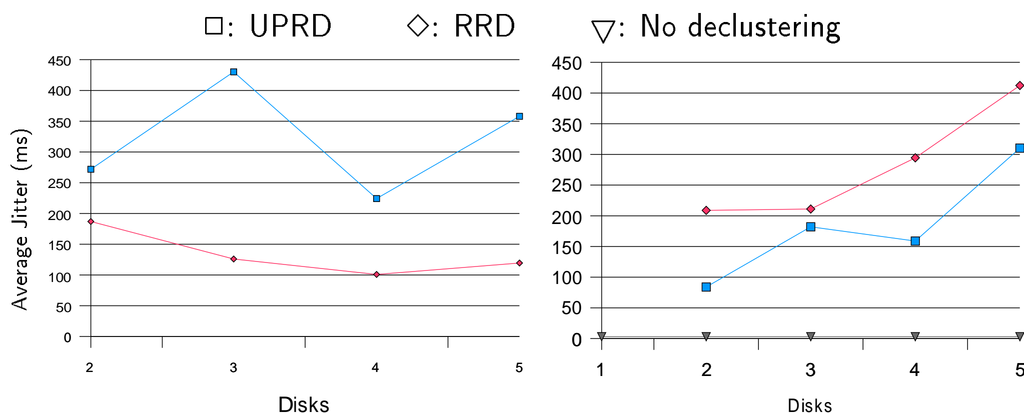

Figure 8.3: Average jitter with and without artificial slowdown

This time difference is commonly known as "jitter", and it is problematic for applications, such as video playback, that require a constant data rate. When data does not arrive continuously, it must be buffered at the client, which can be expensive or increase response time for user commands.

The problem is that the Autonomous Disks do not provide facilities to ensure that replies are delivered at a specific time; the FIFO queues used for scheduling (see section 2.1.2) and especially the lack of synchronization makes jitter unpredictable and hard to control.

The experimental results displayed in figure 8.3 reflect this: While an undeclustered stream could be delivered with extremely low jitter, the RRD shows higher jitter than the UPRD without the artificial slowdown, but the slowdown reverses the situation and also removes the increase of jitter with the number of disks. Furthermore, the UPRD seems to favor even disk numbers for some unexplainable reason, and a slightly different slowdown method led to in vastly different results. Under a high load, results are likely to look different, too.

Generally, this situation is not acceptable for applications that require both the performance gains of declustering seen in section 8.1 and low jitter. To improve it, I suggest the introduction of deadline-based scheduling and tight clock coupling — see section 10.4.

8.3 Response Time

This benchmark was intended to test the influence of the size of the Fat-Btree index on the performance of declustered streams. It was performed on a 4-disk cluster, and the index was first filled with an increasing number of dummy entries. Then a 90 MB stream was inserted once using the UPRD and once with the RRD. Finally, for each of these two streams, 100 reads of 100 byte size were performed at random offsets within the stream, measuring the time until the response was received and taking the average over the 100 requests.

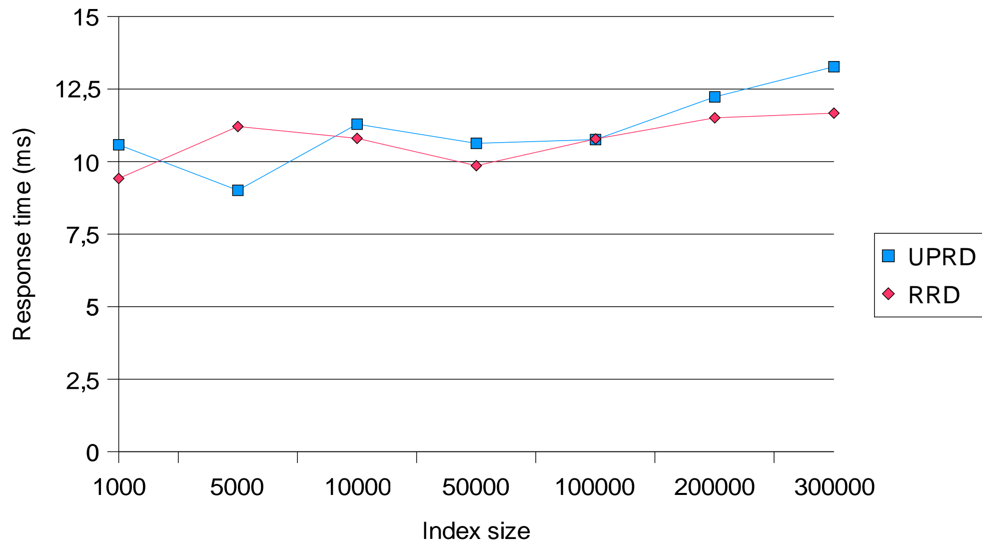

Figure 8.4: Influence of index size on response time

In the result, only a tiny increase in response time is visible, even for very large indexes. The reason for this is probably that the indexes were still small enough to fit into the main memory disk cache — the machines had a large amount of RAM and the OS did not allow switching off the caching. Still, the index construction already took a long time, so larger indexes were not used.

There does not appear to be much difference between the two declustering methods here. RRD performed slightly better, even though the UPRD theoretically requires one less disk seek, but this factor, too, was eliminated by the cache, especially since only one stream was accessed, so the state data of the RRD always consisted of the same block. For a large number of declustered streams, the impact of disk seeks on response time can be expected to be twice that of non-declustered streams.

8.4 Influence of Substream Size

As explained in section 4.3.1, the substream size is a tuning parameter that influences performance in various ways. The results of benchmarks on the prototype system are shown in figure 8.5.

Most interesting is the linear increase of the transfer rate with increasing substream size. This is the expected result of the fewer disk seeks that are necessary. However, the fact that the performance levels out at about 80KB substream size is not a sign that a further increase is useless: it actually results from a network bottleneck at the client, because the benchmark was carried out on the configuration that uses 100 MBit Ethernet (see section 2.1.7).

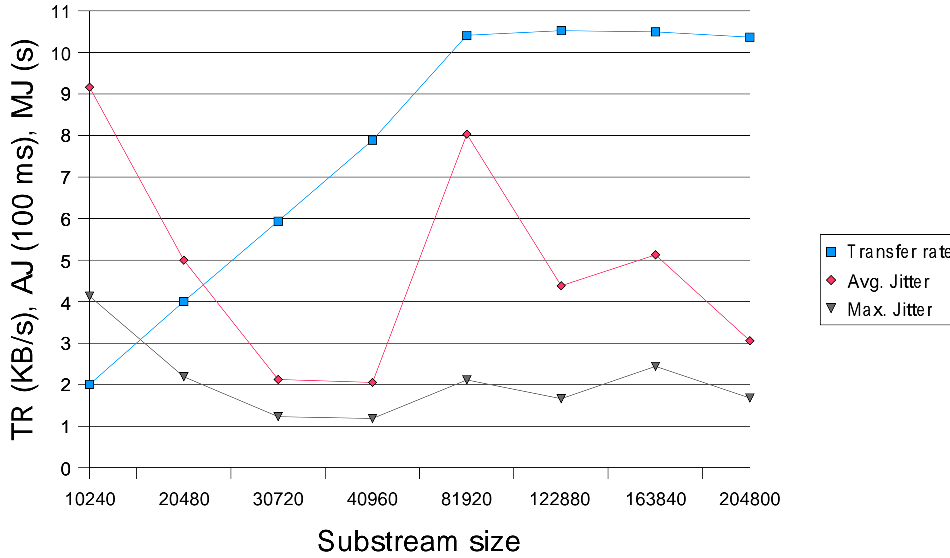

Figure 8.5: Influence of substream size (UPRD declustering over 4 disks)

Analytically, the transfer rate r can be expected to behave as in the following function:

$$r = n\frac{s}{t_s + \frac{s}{r_{max}}}$$

where n is the number of disks working in parallel, t_s the average seek time of the disks, r_{max} their maximal transfer rate (during access) and s the substream size. Using a t_s of 14 ms and an r_{max} of 13 MB/s (which are realistic values for current disks) yields the graph seen in figure 8.6. This agrees with the experimental results and suggests that the transfer rate improves little beyond about 500KB. Whether that is an acceptable value depends on the considerations explained in section 4.3.1.

The jitter graphs in figure 8.5 behave somewhat strangely, showing a clear drop at first, but then a sharp increase at 80KB, followed by a slow decrease. This might be a result of the network transfer limit (see section 5.2) of 40KB: until that point, substreams are transferred as a whole. Beyond it, they are split over several EIS commands. Anyway, this should not be given too much attention, since the jitter is highly volatile due to the total lack of facilities aimed to avoid it,as mentioned in section 8.2.

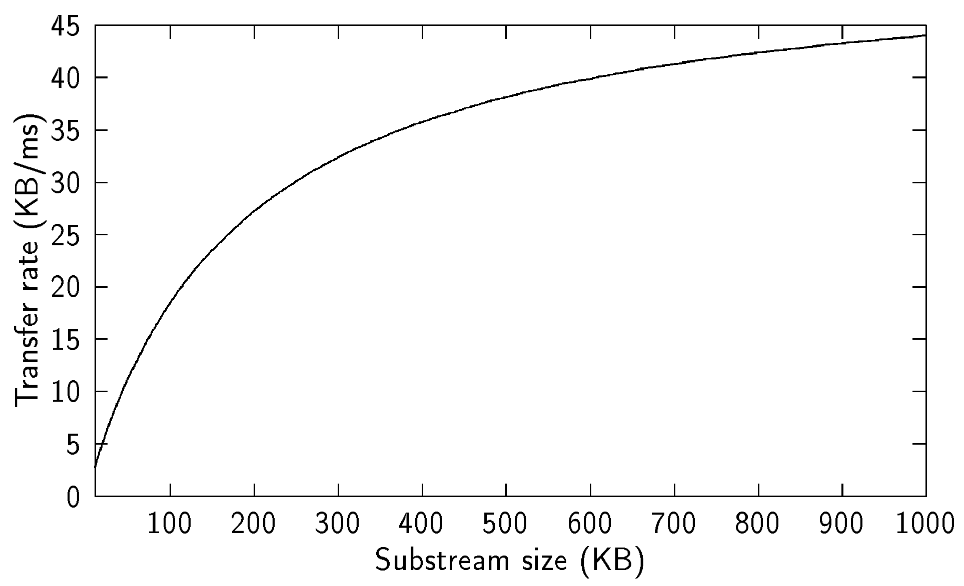

Figure 8.6: Predicted influence of substream size on transfer rate

9 Conclusion

This project consisted of the common task of adding features to a previously existing system. The Autonomous Disks were extended to support arbitrarily large data items. All modifications had to be done in a way that fully retains the previous features (scalable parallel index, transparent fault tolerance and load balancing).

During the course of the project, several new features were added to the system, some of them necessary to support the main functionality:

- A fully functional allocation subsystem based on an efficient allocation policy and a scalable implementation. It serves space requests quickly and manages freed space in a way that avoids fragmentation. This subsystem could also easily be used for other projects.
- An extension of the transport protocol that allows random access to any offset within a stream and the transfer of large streams in many small pieces.
- A framework for declustering that allows streams to be distributed across several disks for fast retrieval and easier load balancing. Different declustering schemes can coexist.
- Two generalized declustering methods were implemented to prove the usability of the framework.

Of course, the completed system had to be tested not only for correctness but also for performance. The results were quite satisfying, showing that all major goals were realized: The benefit of declustering is exactly as expected, resulting in a nearly linear increase of retrieval speed as the number of disks increases. Even for very large indexes, the negative effect on response time was very small, though mainly as a result of the large amount of memory available for caching. Substream size was shown to have a large impact on the attainable transfer rate, proving it was important to make it variable.

One problem that manifested was *jitter*, the time difference between the delivery of adjacent substreams. The reason for this problem is the lack of timed delivery mechanisms in the Autonomous Disks. Without these, declustering is not appropriate for applications that require low jitter (without declustering, jitter was quite low).

Broadly speaking, all major goals of the project were realized, and so I was able to complete it successfully.

10 Future Work

Computer systems are hardly ever completely finished; there are always things left to improve or features to add. Here is a short list of suggestions how to improve and extend my work.

10.1 Integration

The additional features implemented in this project must of course be integrated with other parallel efforts to implement features the prototype is missing, such as a full implementation of the Fat-Btree, skew handling and dynamic reconfiguration of the cluster. For the allocation subsystem, this task is trivial, because it is a separate part that is accessed through a narrow interface. However, the partial transfer and declustering features are likely to overlap considerably with the code areas of the parallel efforts, which makes integration a complex task.

10.2 More Declustering Methods

To make full use of the declustering framework, more declustering methods should be available. First, the design suggestions in section 7.3 for multidimensional data could be implemented.

10.3 Automated Declustering

Since transparent use of the features is one of the most important aspects of the Autonomous Disks, it would be highly desirable to automatize the declustering, and choose a declustering method and substream size (see section 5.4) automatically, according to the type of data. The type might be recognized through "magic numbers" within the file or supplied by the client OS, while the substream size can be chosen according to the load situation[1] and statistics about the average request size for the given type of data or the client node. However, stream size would probably be considered before all of these factors — declustering makes only sense for streams that are considerably larger than a single substream.

[1] this information must be available anyway to implement skew handling

Should the load situation change or the assumptions about the average request size prove false, the stream might be re-declustered with a different size; this is also possible for streams where a suboptimal substream size was chosen by the client. Of course, this requires usage statistics to be kept for individual streams, which might be too much overhead to be practical. A possibility would be to keep such statistics only about the most frequently accessed streams; of course, some statistics are necessary to identify these in the first place.

10.4 Improved Streaming

In section 8.2, I observed that the current system is not very fit for "streaming" data which needs to be delivered to the client at a constant rate. One reason for this is the first-come-first-serve request scheduling of the Autonomous Disks, which makes it impossible to prioritize or delay requests. Switching to a deadline-based scheduler would be a big improvement in this regard and might also have other benefits. Of course, it necessitates the internal clocks of the disks and clients to be coupled tightly, possibly by using NTP, the Network Time Protocol.

Additionally, some sort of interface might be needed to let clients specify the rate at which they need the data. To support this, it might be beneficial to introduce some sort of "file opening" command, similar to the Stream Manager that was decided against in the design phase (see section 5.5.3) and especially the Agent component of BPFS (see section 4.2.1). The additional complexity this would result in must be weighted against how necessary the ability to supply data at a constant rate is, and whether it is not sufficient to simply have the client issue requests at a certain rate.

Bibliography

[1] Haruo Yokota. Autonomous Disks for Advanced Database Applications. In *Proceedings of the 1999 International Symposium on Database Applications in Non-Traditional Environments DANTE '99*, pages 435–442, 1999 (http://www.computer.org/Proceedings/dante/0496/04960435abs.htm)

[2] Haruo Yokota, Yasuhiko Kanemasa, Jun Miyazaki. Fat-Btree: An Update-Conscious Parallel Directory Structure. In *Proceedings of the 15th International Conference on Data Engineering*, pages 448–457, 1999 (http://computer.org/proceedings/icde/0071/00710448abs.htm)

[3] Storage Networking Industry Association, OSD Workgroup http://www.snia.org/English/Work_Groups/OSD/

[4] Angelika Kotz, Klaus Dittrich, Jutta Müller. Supporting Semantic Rules by a Generalized Event/Trigger Mechanism. In *Proceedings of the International Conference on Extending Data Base Technology*, pages 76–91, 1988

[5] Donald Knuth, *The Art of Computer Programming* Vol. 1, Addison-Wesley, Reading, Massachussetts, 1969

[6] Paul R. Wilson, Mark S. Johnstone, Michael Neely and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In Proceedings of the 1995 International Workshop on Memory Management, Springer Verlag LNCS, 1995 (http://http.cs.berkeley.edu/~fateman/264/papers/wilson-allocsrv.ps)

[7] Mark S. Johnstone, Paul R. Wilson. The Memory Fragmentation Problem: Solved? In *ACM SIG-PLAN Notices*, volume 34 No. 3, pages 26–36, 1999 (ftp://ftp.dcs.gla.ac.uk/pub/drastic/gc/wilson.ps)

[8] M.R. Garey, R.L. Graham and J.D. Ullman. Worst-case analysis of memory allocation algorithms. In *Fourth Annual ACM Symposium on the Theory of Computing*, 1972

[9] Helen Custer. *Inside the Windows NT File System*. Microsoft Press, Redmond, 1994

[10] *Inside Macintosh: Files* Apple Computer, Cupertino

[11] Dominic Giampaolo. *Practical File System Design With The Be File System*. Morgan Kaufmann Publishers, San Francisco, 1999

[12] Philip D. Koch. Disk File Allocation Based on the Buddy System. In *ACM Transactions on Computer Systems*, Vol. 5, No. 4 352–370, November 1987

[13] Adam Sweeney. Scalability in the XFS File System. In *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, 1996 (http://www.cs.columbia.edu/~nieh/teaching/e6118_s00/papers/sweeney_xfs96.pdf)

[14] Shahram Ghandeharizadeh, David J. DeWitt. Hybrid-Range partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines. In *Proceedings of the 16th VLDB Conference*, pages 481–492, Brisbane, 1990

[15] Peter Scheuermann, Gerhard Weikum, Peter Zabback. Data partitioning and load balancing in parallel disk systems. In *The VLDB Journal (1998) 7*, pages 48–66

[16] Kai Hwang, Hai Jin, Ryo Ho. RAID-x: A New Distributed Disk Array for I/O-Centric Cluster Computing. In *Proceedings of 9th IEEE International Symposium on High Performance Distributed Computing (HPDC-9)* pages 279–286, Pittsburgh, 2000 (http://andy.usc.edu/papers/HPDC601.pdf)

[17] Duen-Ren Liu, Shashi Shekhar. Partitioning Similarity Graphs: A Framework for Declustering Problems. In *Information Systems, 1996, volume 21,6* pages 475–496 (ftp://ftp.cs.umn.edu/dept/users/shekhar/declusterTR94-18.ps)

[18] K. Abdel-Ghaffar, A. El Abbadi. Optimal Allocation of Two-Dimensional Data. In *International Conference on Database Theory*, pages 409–418, Delphi, 1997 (http://www.cs.ucsb.edu/research/trcs/docs/1996-25.ps)

[19] H.C. Du and J.S. Sobolewski. Disk allocation for cartesian product files on multiple disk systems. In *ACM transactions on Database Systems*, pages 82–101, 1982 (http://www.acm.org/pubs/citations/journals/tods/1982-7-1/p82-du/)

[20] J. Li, J. Srivastava and D. Rotem. CMD: a multidimensional declustering method for parallel database systems. In *Proceedings of the 18th Conference on Very Large Data Bases*, pages 3–14, Vancouver, 1992 (ftp://ftp.cs.umn.edu/dept/ users/padma/declustering/vldb92.ps.Z)

[21] M.H. Kim and S. Pramanik. Optimal file distribution for partial match retrieval. In *Proceedings of the ACM International Conference on Management of Data*, 1988 (http://www.acm.org/pubs/citations/proceedings/mod/50202/p173-kim/)

[22] C. Faloutsos and D. Metaxa. Declustering using error correcting codes. In *ACM Proceedings of the Symposium on Principles of Database Systems*, pages 253–258, Philadelphia, 1989 (http://www.acm.org/pubs/citations/proceedings /pods/73721/p253-faloutsos/)

[23] Christos Faloutsos, Pravin Bhagwwat. Declustering Using Fractals. in *PDIS 2 Conference*, pages 18–25, San Diego, 1993 (ftp://olympos.cs.umd.edu/pub/TechReports/pdis93.ps.Z)

[24] Sunil Prabhakar et al. Cyclic Declustering of Two-Dimensional Data. In *14th International Conference on Data Engineering* pages 94–101, Orlando, Florida, Feb 1998 (http://www.cs.purdue.edu/homes/sunil/pub/icde98.pdf)

[25] Randeep Bhatia, Rakesh K. Sinha and Chung-Min Chen. Declustering using Golden Ratio Sequences. In *Proceedings of the 16th International Conference on Data Engineering*, San Diego, 2000
(http://www.computer.org/proceedings/icde/0506/05060271abs.htm)

[26] S. Berchtold, C. Böhm, H.-P. Kriegel. The pyramid-technique: towards breaking the curse of dimensionality. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 142–153, Seattle, June 1998
(http://www.acm.org/pubs/citations/proceedings/mod/276304/p142-berchtold/)

[27] H. Ferhatosmanoglu, D. Agrawal, and A. El Abbadi. Concentric hyperspaces and disk allocation for fast parallel range searching. In *Proceedings of the International Conference on Data Engineering*, Sydney, Australia, March 1999.
(http://www.cs.ucsb.edu/research/trcs/docs/1999-03.ps)

[28] Robert D. Russell. The Architecture of BPFS: A Basic Parallel File System. In *Proceedings of the 2nd IASTED International Conference on Parallel and Distributed Computing and Networks*, pages 214–220, Brisbane, 1998
(http://www.cs.unh.edu/~rdr/pdcn98.ps)

[29] John S. Heidemann. Stackable Design of File Systems. PhD thesis, University of California, Los Angeles, 1995
(http://www.isi.edu/~johnh/PAPERS/Heidemann95e.html)

[30] Nils Nieuwejaar, David Kotz. The Galley Parallel File System. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, 1996 (http://www.cs.dartmouth.edu/~dfk/nils/papers/galley.html)