Active Storage for Database Applications

by

Tina Miriam John

B.Tech., University of Kerala, 2003

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

at the

University of Connecticut

2008

APPROVAL PAGE

Master of Science Thesis

Active Storage for Database Applications

Presented by
Tina Miriam John, B.Tech.

Major Advisor _____
John A. Chandy

Associate Advisor _____
Mohammad H Tehranipoor

Associate Advisor _____
Chun-Hsi Huang

University of Connecticut
2008

Acknowledgements

I would like to thank and acknowledge my advisor, *Dr. John A. Chandy* for his constant faith in me throughout the long, challenging, yet exciting work on this thesis. I am grateful to his patience, openness to ideas, discussions and ease to work with. I definitely couldn't have done this without his help and guidance. I am thankful to *Janardhan* and *Anu*, my co-workers at SNSL, for their cooperation and also for sharing their ideas with me.

Last and most importantly, I would like to thank my parents, and my dear husband *Junu* for their constant support and encouragement in all my endeavors. I am forever grateful to their love and understanding that has made this thesis happen.

Contents

List of Tables

Table

List of Figures

Figure

Abstract

The increasing performance and decreasing cost of processors and memory are causing system intelligence to move from the CPU to peripherals such as disk drives. Storage system designers are using this trend toward excessive computation capability to perform more complex processing and optimizations directly inside the storage devices. Taking advantage of the extra processing power on individual disk drives to run application-level code has been called active storage in previous research. This idea of moving portions of an applications processing to run directly at disk drives can dramatically reduce data traffic and take advantage of the parallel storage already present in large systems today.

This thesis contributes to active storage in two areas. We first demonstrate active storage on an iSCSI OSD standards-based object oriented framework. This framework can support a variety of data-intensive applications including database applications. We then move on to exploring parallel databases and integrating active storage in such systems. In particular, this work concentrates on finding out optimal data partitioning for an active storage based database system.

To my family...

Chapter 1

Introduction

1.1 Motivation

The cost and silicon real estate needed for any particular computational need is continually dropping. At some point, additional processing power can be had at negligible cost. The question then becomes simply where to place this computation power in a system to support the widest range of tasks efficiently. The contention of this work is that processing power is already moving into peripheral devices and that applications can achieve significant performance gains by taking advantage of this trend. Specifically, this work focuses on how data-intensive applications can directly exploit the processing power of the controllers in individual commodity disk drives to improve both individual application performance and system scalability.

The same trends in chip technology that are driving microprocessors toward ever larger gate counts drive disk manufacturers to reduce cost and chip count in their devices while simultaneously increasing the total amount of local processing power available on each device. One use for this increasing computation power on disk controllers is to enrich their existing interface. For example, recent advances in network-attached storage are integrating storage devices into general-purpose networks and offloading a range of highlevel functions directly to the devices. This eliminates servers as a bottleneck for data transfers between disks and clients and promises significant improved

scalability through higher-level interfaces.

At the same time, as systems get faster and cheaper, people compute on larger and larger data sets. A large server system today will easily have a hundred disk drives attached to it. This large number of drives is necessary either to provide sufficient capacity or sufficient aggregate throughput for the target application. Taking this trend and extrapolating to future drive capabilities gives a promising picture for on-drive processing. A pessimistic value for the on-drive processing already in todays commodity SCSI disk controllers is 400 MHz, with perhaps 100 MB/s of sustained bandwidth in sequential access. This means that a system with one hundred disks has 40 GHz of aggregate processing power and 10 GB/s of aggregate bandwidth at the disks. Typical multiprocessor systems can not achieve these aggregate values in a single node, but computing clusters can be constructed to achieve the same capability. The advantage of the active disk cluster over a compute node cluster, is the ability of the disk to manage the data directly on disk in terms of layout and scheduling. Data processing at the disk can also reduce the amount of data transferred, thereby reducing communication costs.

In addition, as storage is connected to a large collection of hosts by taking advantage of network-attachment and storage area networks, the interconnection network will rapidly become a principle bottleneck in large-scale applications. If data can be processed directly by the devices at the "edges"of the network, then the amount of data that must be transferred across this bottleneck can be significantly reduced. This is the principle idea which fuels an Active Storage System.

In this thesis, we first explore Object Based storage devices, which store and manage data containers called objects. The NASD project from CMU [GVM00] [GNA+96] was the initial introduction to the concept of object based storage. NASD pursues the motion that aggregation of storage devices is best managed through a central policy server (possibly a cluster of servers), while most commands and data transfers

move directly between device and client, bypassing the server. Upon approval by the central server, clients can access (directly and in parallel) all devices containing data of interest. We then integrate the idea of such object based storage devices with Active Storage Systems.

Next, we delve into the world of intensive data computation applications , such as enterprise database applications. Database management systems (DBMS) provide important support for commercial applications. Currently there is a rapidly growing trend among businesses to analyze their increasing volumes of transaction data for various types of trends, including sales and purchasing, inventory, and budget.The emergence of these applications has fueled the need for parallel DBMS software from commercial vendors. This thesis, in particular,focuses on how data partitioning in a DBMS affects the performance. Based on our studies, we suggest an intelligent data partitioning and application training system that improves performance in an Active Storage based Database Application.

1.2 Active Disks

Active Disks are hailed to be the next-generation of disk drives that provide an environment for executing application code directly at individual drives, thus providing the basis for an Active Storage System. By partitioning processing across hosts and storage devices, it is possible to exploit the cycles available at storage, reduce the load on the interconnection network, and perform more efficient scheduling. For example, an application that applies selective filters to the stored data and only ships summary information across the network, or that makes scheduling decisions based on local information at the individual drives can make more effective use of network and host resources. This promises both improved individual application performance and more scalable systems. The basic processing in an Active Disk system compared to a traditional host-based system is illustrated in Figure 1.1. Instead of running

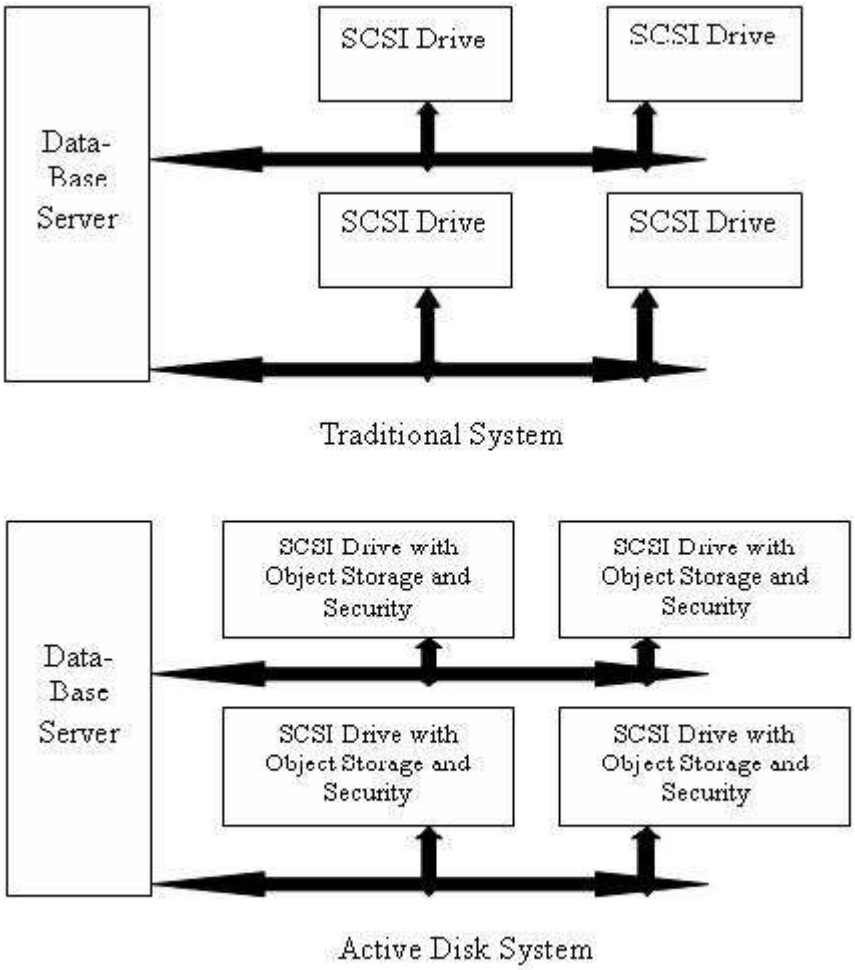

Figure 1.1: Architecture of an Active Disk system vs. a traditional server

all application processing at the host, and forcing all the raw data to move from the disk drives, through the storage interconnect, and to the host before processing, Active Disk applications execute on both the host and the disks. The core portion of the applications data-intensive processing is extracted and executed in parallel across all the disks in the system. The diagram in Figure 1.1 shows the Active Disks as an extension of NASD (Network-Attached Secure Disks), which means they are network-attached, present an object- instead of a block-level interface, and contain provisions for a robust security system enforced directly at the drives. These three characteristics are not strictly necessary for Active Disks, and one could definitely imagine providing execution functionality in more traditional SCSI disks.

A proposal to perform data-intensive operations using processing elements directly attached to disk drives may seem familiar to anyone who has worked in the database area as the database machines that dominated database research in the 70s and 80s. The concept of Active Disks is close to the ideas pursued in the work on database machines and many of the lessons from that work are applicable today. Since that time, technology trends in several areas have shifted the picture considerably, providing an opportunity to revisit the arguments explored in these projects.

1.2.1 Applications and Algorithms

There are a number of data-intensive database, data mining, and multimedia applications, which with a few changes will allow for them to efficiently leverage an Active Disk system. Listed below are a few of them.

1.2.1.1 Scans

The most compelling applications for Active Disks are the "embarrassingly parallel"scan operations that can be easily split among a set of drives and that perform highly selective processing before the data is placed on the interconnection network.

The first data mining application examined are database queries that wish to find records in a database that are most similar to a particular, desirable record. For the Active Disk system, each disk contains an integral number of records and the comparisons are performed directly at the drives. The host sends the target record to each of the disks which determine the k closest records in their portions of the database. These lists are returned to the server which combines them to determine the overall k closest records. Because the application reduces the records in a database of arbitrary size to a constant-sized list of k records, the selectivity is arbitrarily large.

The second data mining application is the Apriori algorithm that computes association rules in sales transactions [RR95]. The computation is done in several passes, first determining the items that occur most often in the transactions (the 1-itemsets) and then using this information to generate pairs of items that occur often (2-itemsets) and larger groupings (k-itemsets). For the Active Disks system, the counting portion of each phase is performed directly at the drives. The central server produces the list of candidate k-itemsets and provides this list to each of the disks. Each disk counts its portion of the transactions locally, and returns these counts to the server. The server then combines these counts and produces a list of candidate (k+1)-itemsets.

1.2.1.2 Sorting

There are many reasons for sorting a set of records, and sort is a popular system benchmark. Sorting is most often done in the context of database systems, where it is usually combined with another operation, as discussed in the next sections. First, the basics of sorting in an Active Disk system.

There are several ways to partition a sorting algorithm on Active Disks, depending on the available functionality at the disks. The primary resource constraints for sorting are the disk bandwidth (each item of data has to be read and written at least once), network bandwidth (how often the data must move between the source disks,

the host, and the destination disks on the interconnect), and memory size (which determines how large individual runs in a multiple-pass sort will be). As we will see, the main determinant of performance for large data sets turns out to be the interconnect bottleneck.

The most common method for out-of-core sorting is Merge Sort [Knu79] [vN63] which performs a sequence of in-memory sorts on small subsets of the data, followed by a series of merge phases that combine the sorted subsets until the entire dataset is in order.In an Active Disk algorithm, we save two of these transfers by having each drive perform the sorting of its own data locally, and performing only the final merge step at the host. Instead of providing the raw data to the host, the drives provide already sorted runs that must simply be merged by the host.

A further extension that uses the ability to compute at the disks is to perform the sort using only the keys and introduce a "shuffling"phase once the keys are in sorted order. In practice, most in-core sorting is done on keys only, rather than entire records, since keys are usually small relative to whole records. If we apply this in an Active Disk system, we can save several network transfers and one write of the entire data to disk by doing run formation using only the keys, instead of the full records.

1.3 Parallel Databases

Large-scale parallel processing technology has made giant strides in the past decade, and there is no doubt that it has established place for itself. At this time, however, almost all of the applications harnessing this technology are scientific or engineering applications. The lack of commercial applications for these parallel processors may in part be due to the perceived robustness and usability of these systems. When compared with mainframes, large-scale parallel processing systems do not put much emphasis on availability and reliability and do not provide much software support for system management and application development. However, the

current generation of massively parallel processor systems, such as the IBM Scalable POWERparallel Systems, are much more robust and easier to use. In the past, a number of research prototypes, including Gamma [DSD+90], Bubba [BAC+90], and XPRS [WM94], have tried to understand the issues in parallel databases. These and other projects addressed important issues such as parallel algorithms for execution of important database operations, query optimization techniques,data placement,and database performance. The DB2 Parallel Edition (DB2 PE) [BFG+95], developed by IBM, is another innovative parallel database software program that provides both scalability and capacity, along with better query processing performance.

1.3.1 Architecture Approaches

Parallel database systems can be built on top of different hardware architectures, the main three as shown in Figure 1.2.

1.3.1.1 Shared Memory

In shared-memory systems, multiple processors share a common central memory. With this approach, communication among processors is through shared memory, thus there is little message overhead. In addition, the software required to provide parallel database processing is considerably less complex. Consequently, many commercial parallel database systems available today are based on the shared-memory architecture.

1.3.1.2 Shared Disk

In shared-disk systems, multiple processors,each with its local memory, share a pool of disks. Shared-disk systems avoid the central memory access bottleneck, but introduce the difficult problem of connecting all processors to all disks. This can be especially difficult inth e caseof a large number of processors and disks. In addition,

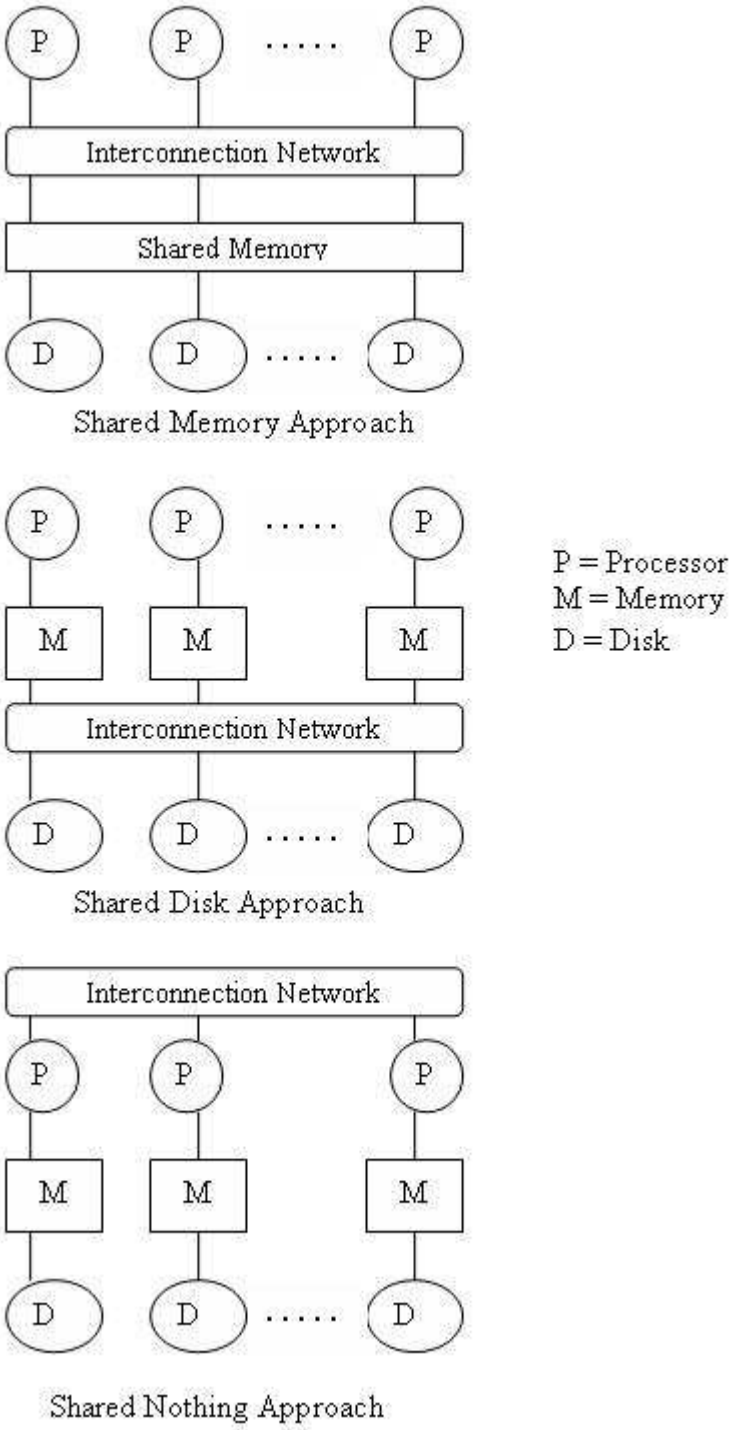

Figure 1.2: Parallel Database Approaches

shared disks present the most challenging task of transaction management because of the need to coordinate global locking activities (without the help of shared memory) antdo synchronize log writing among all processors.

1.3.1.3 Shared Nothing

With the shared-nothing approach, each processor has its own memory as well as local disks. Except for the communication network, no other resources are shared among processors. Shared nothing does not have the memory access bottleneck problem, nor does it have the problem of interconnecting a large number of processors and disks. The major complexity in supporting the shared-nothing approach is the requirement of breaking a Structured Query Language (SQL) request into multiple subrequests sent to different nodes in the system and merging the results generated by multiple nodes.Researchers and developers havaer gued that the shared-nothing approachis the most cost-effective alternative and the most promising approach for high-performance parallel database systems

1.4 Outline

This thesis has been organized into 4 chapters. The first chapter gives the introduction and lays the background for this work. It also briefly discusses about the origin of Active Disks and the significance of parallel databases. Chapter 2 elaborates on the framework that was developed to support active storage on object based devices. Chapter 2 also provides our results of active computation done on our framework. Chapter 3 discusses our methodology to build an efficient Active Storage based parallel database system. Finally, in Chapter 4, we conclude with our ideas for future work.

Chapter 2

Active Storage with Object Based Devices

2.1 Introduction

The database machines proposed and developed in the 70s and 80s are the main relatives of Active Storage systems. They had special processing components that handled the execution of portions of the function of a database system. These devices were custom-designed to implement a particular algorithm and assumed a particular set of queries and a particular data layout.

Previous work on active disks was brought to life by the very database machines described above. They provide an environment for executing application code directly at individual drives [Rie99]. By partitioning processing across hosts and storage devices, it is possible to exploit the cycles available at storage, reduce the load on the interconnection network, and perform more efficient scheduling. For example, an application that applies selective filters to the stored data and only ships summary information across the network, or that makes scheduling decisions based on local information at the individual drives can make more effective use of network and host resources. This promises both improved individual application performance and more scalable systems.

In this chapter we demonstrate an object oriented framework for supporting applications that benefit from an Active Storage system, such as the Active Disks men-

tioned previously. The framework can support a variety of data-intensive applications including database operations. In the following sections, we describe the basic approach that was followed to come up with the framework. Then we go on to describe the implementation details of the sample application developed, to test out the framework. Finally we describe our experimental setup and compare the performance of an existing single server system to a prototype system emulating Object Oriented Active Disks.

2.2 Related Work

The basic idea of executing functions in processing elements directly attached to individual disks was explored extensively in the context of database machines such as CASSM [YL75], RAP [OSS75], and numerous others [DH81]. These machines didn't catch much public interest due to the limited performance of disks at the time and the complexity of building and programming special-purpose hardware that could only handle limited functions. Instead, database research has developed large-scale, shared-nothing database servers with readily available processing elements. It has recently been suggested that the logical extension is to perform all database processing inside programmable, "smart" system peripherals [Gra97].

Prior work on Active Disks follows from network-attached secure disks (NASD), in which computational power at the storage level for parallel and network file system functions are exploited, as well as traditional storage optimizations [GNA+97, GNA+98]. The NASD work experimented on several types of applications that can benefit from Active Disks - including filters, multimedia, batching, and storage management - and enumerated the challenges to providing an execution environment on commodity disk drives [Rie99, RGF98].

Acharya, et al. used the idea of Active Disks on a set of similar applications, including database select, external sort, data cubes, and image processing, using an

extended-firmware model for next-generation SCSI disks [AUS98]. Similarly, a group at Berkeley has independently estimated the benefit of Active (Intelligent in their terminology) Disks for improving the performance of large SMP systems running scan, hash join, and sort operations in a database context [KPH98].

Our work is most similar to recent work has examined the use of active disk principles in the Lustre parallel file system [PNF07]. Their work uses a "filter" in the object storage target stack that processes streams of data from defined active objects. It differs from our work in that we allow the client to define an object-oriented class structure for an object and uses the OSD framework as an asynchronous RPC mechanism to access object methods. Thus, the execution is not dependent on operating on streams of data, but can operate on the entire object if necessary.

2.3 Object-based Storage Devices

The Object-Based Storage Devices (OSD) specification [ANS08] has introduced a new set of device-type specific commands into the SCSI standards family. The specification defines the OSD model and their required commands and command behavior. Figure 2.1 illustrates the basic model of OSD in comparison to a traditional block-based device model for a file system. The traditional functionality of file systems is divided mainly so that the increased intelligence that is available in such storage devices can be taken advantage of. Object-based Storage devices have the capability of managing their storage capacity and exposes a file-like storage interface,objects, to their hosts. These objects behave exactly like files. They can be created and destroyed and can grow and shrink their size during their lifetimes. As in a file system, commands are available that can be used to read or write any consecutive stream of the bytes making up a storage object. The important difference here is that, in addition to storing data in the form of objects, the OSD storage management component also maintains other information about the objects in the form of attributes,

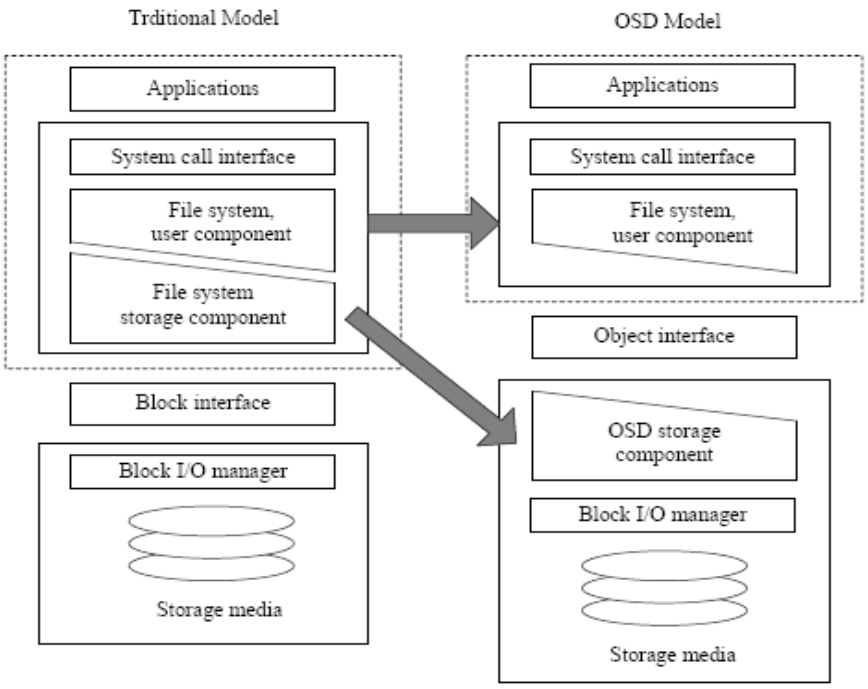

Figure 2.1: Comparison of traditional and OSD storage models [DHH+06]

e.g. size, usage quotas, security credentials and associated user name.

2.3.1 OSD Objects

Storage objects that are used to store regular data are called user objects in the OSD specification. Three other kinds of objects are also defined within the specification, namely, root objects, partition objects and collection objects. There is one root object for every OSD logical unit [ANS04]. The root object forms the top most module of an OSD logical unit, similar in concept to a partition table for a logical unit of block devices. User objects are put into partitions that are represented by partition objects. The maximum number of partitions permissible within a logical unit is given by a quota defined for the logical unit in the root object. Every user object belongs to one and only one partition. Collections are represented by collection objects. Each collection object belongs to one and only one partition and may contain zero or more user objects belonging to the same partition. Unlike user objects, the other three kinds of objects do not contain a read/write data area. All relationships between objects are represented by object attributes discussed in the next section.

The unique combination of the Partition ID and the User Object ID identifies each OSD object as illustrated in Table 2.1. The ranges not specified in the table are reserved.

Partition_ID	User_Object_ID	Object type
0	0	root object
2^{20} to $2^{64}-1$	0	partition object
2^{20} to $2^{64}-1$	2^{20} to $2^{64}-1$	collection/user object

Table 2.1: Object identification numbers

2.3.2 Object Attributes and Attribute Pages

Object attributes are used to associate meta data with any OSD object, i.e., root, partition, collection or user. Attributes are organized in pages for identification and reference. Attribute pages associated with an object is uniquely identified by their attribute page numbers ranging from 0 to 2^{32} - 1. This page number space is divided into several segments so that page numbers in one segment can only be associated with certain type of object. For instance, the first segment from 0x0 to 0x2FFFFFFF can only be associated with user objects.

Attributes within an attribute page have similar sources or uses. Each of them has an attribute number between 0x0 and 0xFFFFFFFE that is unique within the attribute page. The last attribute number, i.e., 0xFFFFFFFF is used to represent all attributes within the page when retrieving attributes.

The OSD specification defines a set of standard attribute pages and attributes that can be found in [BMLX03]. Certain range of attribute pages and attribute numbers are reserved for other standards, manufacturer specific or vendor specific ones. By this way, new attributes can be defined to allow OSD to perform specific management functions. In [LDR05], a new attribute page containing QoS related attributes is defined to enable OSD to enforce QoS.

2.3.3 Commands

The OSD commands execution follows a request-response model as defined in SCSI Architecture Model (SAM-3) [ANS04]. This model can be represented as a procedure call as following: Service response = Execute Command(IN(I_T_L_x Nexus, CDB, Task Attribute, [Data-In Buffer Size], [Data- Out Buffer], [Data-Out Buffer Size], [Command Reference Number), OUT([Data-In Buffer], [Sense Data], [Sense Data Length], Status))

The meaning of all inputs and outputs are defined in SAM-3 [ANS04]. The OSD

specification additionally also defines the contents and formats of CDB, Data-Out Buffer, Data-Out Buffer Size, Data-in Buffer, Data-in Buffer Size and sense Data.

The OSD commands use the variable length CDB format defined in SPC-3 but has a fixed length of 200 bytes. Each OSD command has an opcode 0x7F in CDB to differentiate it from commands of other command sets. In the same CDB, a two-byte service action field specifies one of the twenty-three OSD service requests defined in the OSD specification. Some of the CDB fields are specific to service actions and others are common for all commands. Every CDB has a Partition ID and a User Object ID, the combination of which uniquely identifies the requested object in a logical unit. Any OSD command may retrieve attributes and any OSD command may store attributes. Twenty-eight bytes in CDB are used to define the attributes to be set and retrieved. Two other common fields in CDB are capability and security parameters that will be explained later.

Both Data-In Buffer and Data-Out Buffer contains multiple segments, including command data segments, parameter data segments, set/get attribute segments and integrity check value segments. Each segment is identified by the offset of its first byte from the first byte of the buffer. Such offsets are referenced in CDB to indicate where to get data and where to store data.

If the return status of an OSD command is CHECK CONDITION, sense data are also returned to report errors generated in OSD logical units. The sense data contain information that allows initiators to identify the OSD object in which the reported error was detected. If possible, a specific byte or range of bytes within a user object is identified as being associated with an error. Any applicable errors can be reported by include the appropriate sense key and additional sense code to identify the condition. The OSD specification chooses descriptor format sense data to report all errors so several sense data descriptors can be returned together.

2.3.4 Security Model

Figure 2.3 shows the OSD security model consisting of four components [ANS08, FNN+05]: (a) Application Client, (b) Security Manager, (c) PolicyManager, and (d) Object Based Storage Device (OBSD). Whenever an application client performs an OSD operation, it first contacts the security manager in order to get a capability including the operation permission and capability key to generate an integrity check value with OSD Command Description Block (CDB). When the security manager receives the capability request from the application client, it contacts the policy manager to get a capability including the permission to perform the operation. After obtaining the capability, the security manager creates a capability key with a key shared between the security manager and OBSD and makes the credential consisting of the capability and capability key, which is returned to the application client. Now the application client copies the capability included in the credential to the capability section of the CDB and generates an integrity check value for the CDB with the received capability key. The CDB with the digested hash value called the request integrity check value is sent to the OBSD. When the OBSD receives the CDB, it checks the validity of the CDB with the request integrity check value. The shared secret between the security manager and OBSD for the authentication of the CDB is maintained by SET KEY and SET MASTER KEY commands [ANS08].

2.3.5 Reference Implementation

The reference OSD implementation used in this work consists of client components and server components shown in Figure 2.2 as grayed blocks. The client components include three kernel modules - the osd file system (osdfs), the scsi object device driver (so) and the iSCSI initiator host driver. The osd file system is a simple file system using object devices instead of block devices as its storage. The so driver is a SCSI upper-level driver and it exports an object device interface to applications

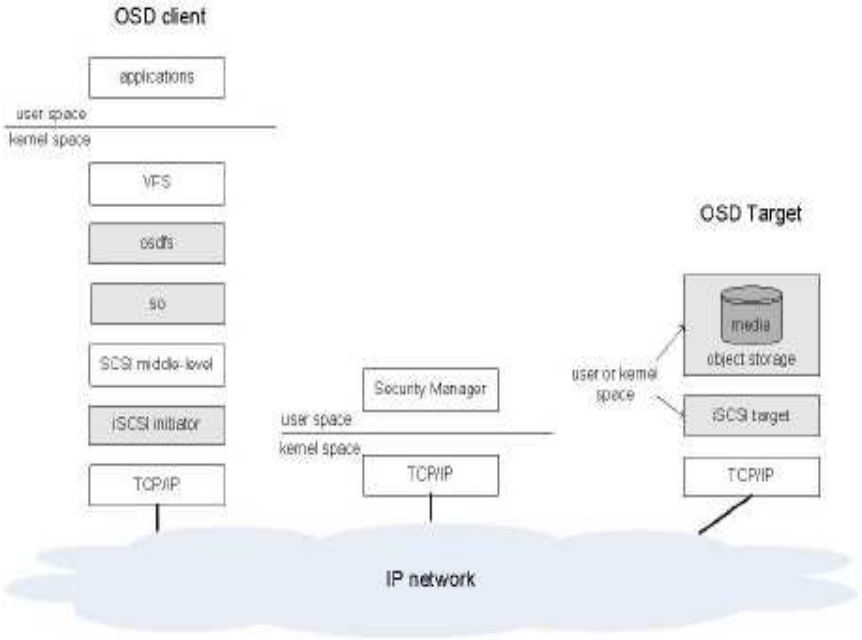

Figure 2.2: Overview of reference implementation [DHH+06]

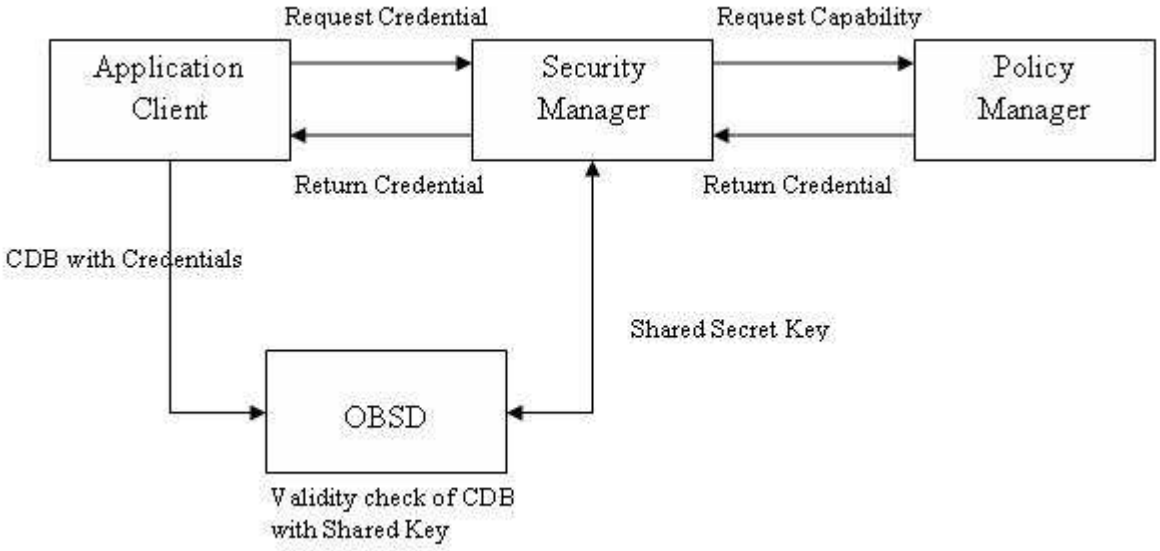

Figure 2.3: OSD Security Model

like osdfs. The iSCSI initiator driver is a SCSI low-level driver providing iSCSI transport to access remote iSCSI targets over IP networks. The server components include the iSCSI target server and the object storage server. The iSCSI target driver implements the target side of the iSCSI transport protocol. The object target server module manages the physical storage media and processes SCSI object commands. The functions and internal architectures of these components are elaborated elseware.

2.4 Basic Approach

To capitalize on the parallelism in an active storage system, we designed and developed a Record Storage Application, the details of which are provided in the next section. The application was modeled to closely resemble a database system. The objective of this application development was to exemplify a framework that supports the parallel execution of any kind of application, and to analyze the performance gains of object oriented Active Disk computation over traditional single server systems.

2.4.1 Design Methodology

The basic design principles of developing an application to run in an Active Disk environment are to

- Exploit parallelism in the data-intensive portion of the data processing
- Separate the code to be executed at the disks as much as possible, from the code that controls the execution flow,so as to form self-contained and manageable units, and
- Take advantage of variations in available resources during execution, by using primitives.

These three principles allow for a great deal of performance and flexibility while placing and executing the application code. The Active Disk application code is

divided such that the core piece of code will be made to run at each of the drives, while the high-level synchronization, control, or merging code is made to run at the host. This forms a client-server parallel programming model as illustrated in Figure 2.4. Initialization of input parameters takes place at the host and is sent across to all of the disks. Each of the disks operates on its own local data and produces its portion of the final result. The host collects the results from all the disks and merges them in an appropriate manner to form the final result. Since there is a portion of code that runs at the host and processes data from the disks, it is always possible to correct any incomplete or inaccurate computation at the drive, as long as the drive functions always act conservatively. When the drive is in doubt whether to include a particular record into its result set, it is expected to add the record anyway, so that the final filtering done at the host will be able to make a better judgment about it.

One of the major objectives of this work was to be able to facilitate the transfer of program files, related to an object type under consideration, to a OSD target which has no prior knowledge about that object type. For instance, the target could be ignorant of the methods that operate on a pdf or text file object. This capability allows any kind of function to be invoked by the client on any target. For this reason, we chose to implement our application in Java, thus allowing the methods pertaining to each object type to be encapsulated within its own class. In order for the OSD target to execute the methods, this Java code must first be downloaded to the target and then executed. This class file transfer is achieved by setting an attribute on the object where the attribute data is the JAR file associated with the class.

The class file, associated with the object as an attribute, defines the methods for that object. A client can execute methods on that object by using the OSD communication channel. The OSD T10 standard recently added the capability to specify the presence of a CDB continuation segment in the Data- Out buffer by including the CDB CONTINUATION LENGTH field to the basic OSD CDB format. This

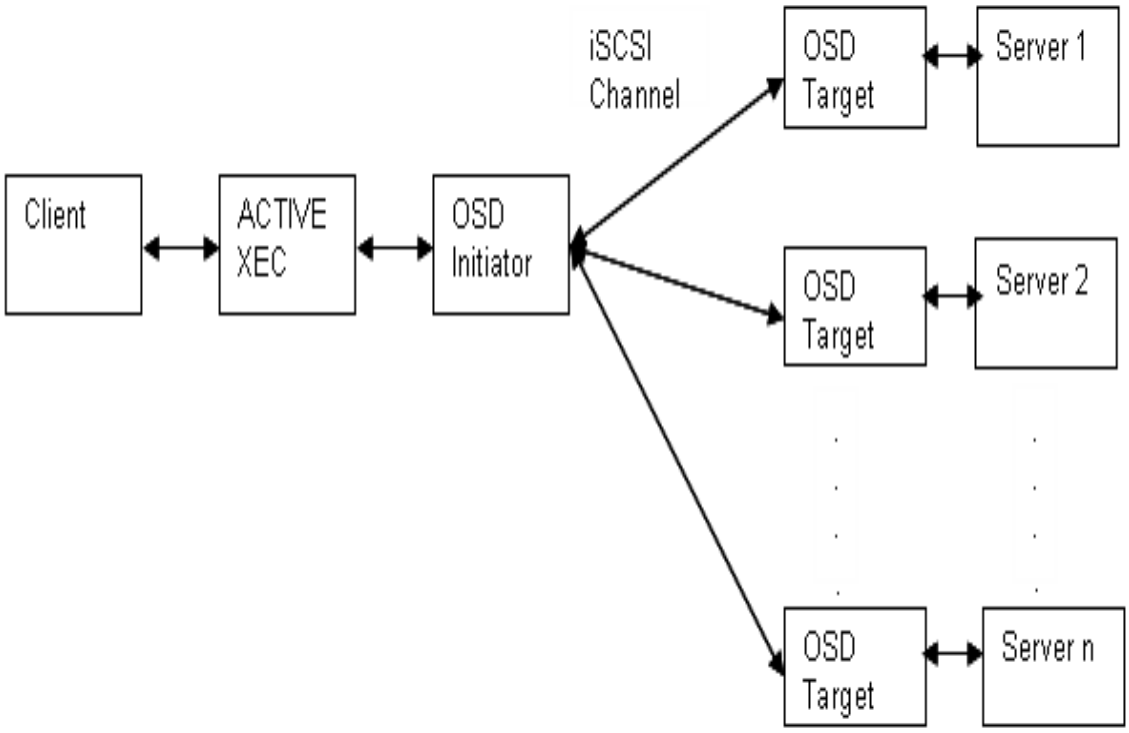

Figure 2.4: Basic structure of Active Disk computation via OSD

field specifies the number of bytes in the CDB continuation segment of the Data-Out buffer. The CDB CONTINUATION DESCRIPTOR TYPE field in the continuation segment specifies the format of the following CDB descriptor type specific data. A value of 0x0001h for the descriptor type supports scatter/gather features. The CDB continuation segment is followed by the command data segment, if any, in the Data-Out buffer. For our implementation we added a new CDB descriptor type, 0x0011h, to indicate that the continuation segment contains class file attribute data. The command data segment of the Data-Out buffer carries the method invocation associated with the command - i.e. the method name and parameters.

A java server runs on each OSD target, which takes in the requests coming from the client and first determines the type of object that the client request is meant for. Then the corresponding objects methods are executed. In the case of a class file transfer, the java server first downloads the transferred class file to an appropriate location and then uses the newly downloaded class to invoke the objects methods.

2.4.2 Sample APIs Developed

Two main functionalities that have been implemented in the application are the record sorting and matching record search functions, which are expected to benefit the most from such active disk computations. Sorting is a popular system benchmark and is most often done in the context of database systems, where it is usually combined with another operation. We use at the client a variant of the Merge Sort algorithm, which is the most common out-of-core sorting methodology. Merge Sort is based on the assumptions that: 1) it is less time consuming and less complex(requiring less cycles) to sort a small list than a large list , and 2) fewer operations are required to construct a sorted list from two sorted lists than from two unsorted lists. The algorithm performs a sequence of recursive in-memory sorts on small subsets of the data set, followed by a series of merge phases that combine the sorted subsets until

the entire dataset is in order.

The matching record search function was implemented to emulate scan operations in databases. When such searchs require a high degree of selectivity from a large dataset, active storage computation can reduce the transfer of data from the servers to the client , thereby decreasing network conjestion. Once the search results are received from every server, then the job of the client is as trivial as to just append them all together, thus forming the final result.

2.4.3 Approach and Implementation

This section provides an outline of the structure of the application described above, that execute on Active Disks, including the design philosophy, the structure of the on-drive code and the host code. The data path of the commands is as shown in Figure 2.4.

Our implementation is based on the iSCSI OSD reference implementation developed by the University of Minnesota [DHH+06]. The reference implementation is built using Linux kernel modules and user-level libraries as shown in Figure 2.2. The kernel modules are used to implement a primitive OSD file system and the user-level code provides the communication channel and the implementation of the OSD target semantics. Since our framework does not require the use of the OSD file system, our implementation is entirely user based.

2.4.3.1 Client Design

The client side of our application was designed to have APIs for multiple operations on an object , that can exploit the parallelism offered by such a framework. Continuing with the above discussed examples of record sorting and searching, the splitting of the application code is done as follows. Each target sorts the data set it contains for the particular database object, in parallel, and transfers the sorted

result back to the client. The client receives the sorted data sets from each of the targets and merges them together to produce the final sorted result set. This scheme of computation totally eliminates the need to transfer all the data initially to the client for the sorting phase, as it is done directly at the target. In the case of record search, each target searches for records matching the given search criteria and sends back just the resulting records back to the client. The client, just as in the case of sorting, collects the partial result records from all the OSD targets and combines them together to produce the final search result. Since filtering of the data occurs at each of the targets, the data that gets transferred across the network is just a fraction of the initial data set, thereby reducing network bottleneck. The set of client side APIs developed for testing this framework for database objects is as shown in Figure 2.5. APIs for creation and manipulation of the objects are also provided, so as form a complete system. Similar client classes can be developed for any type of object, with APIs for the methods of interest.

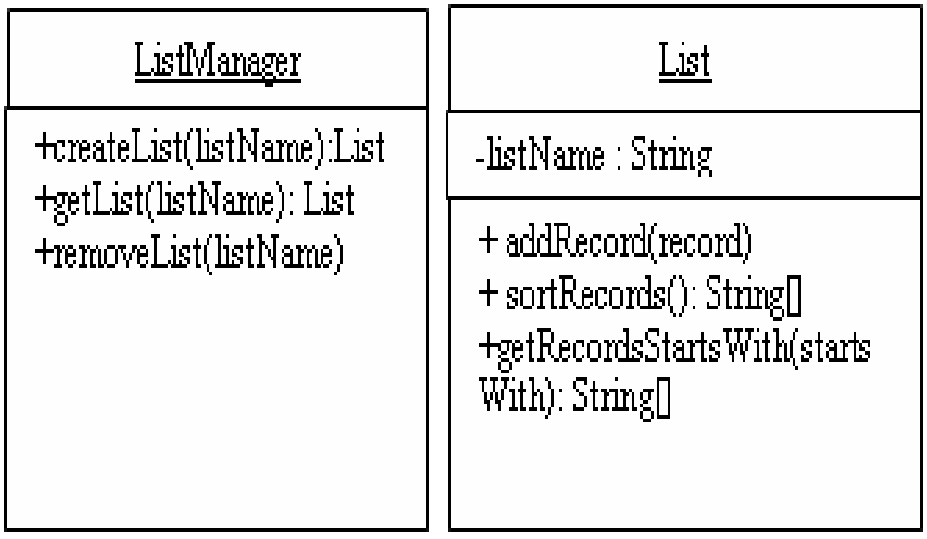

Figure 2.5: Client side class diagrams

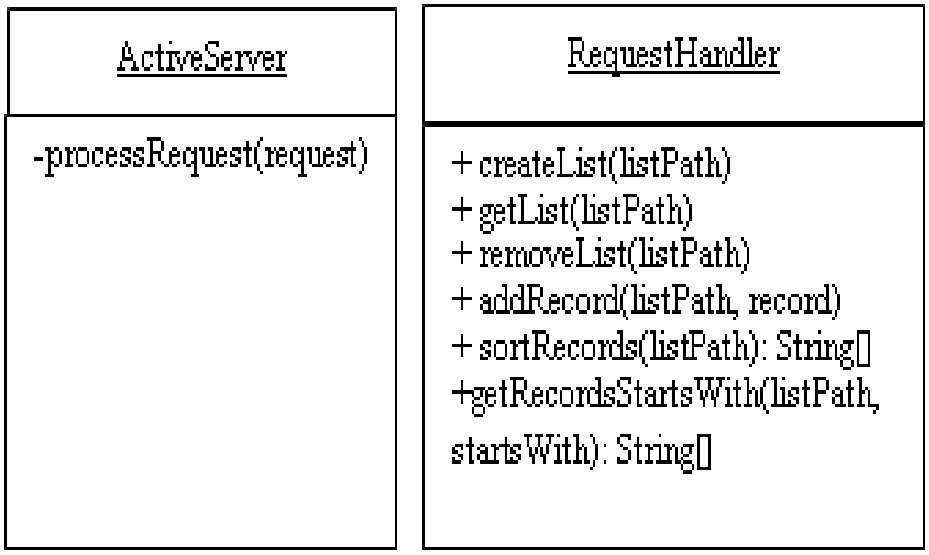

Figure 2.6: OSD Target Server side class diagrams

2.4.3.2 OSD Target Server Design

The target server is designed to have a dedicated process, constantly listening to requests being sent from the client. In order to ensure the parallelism inherent with Active Disk applications, the client spawns a thread for each server, while sending requests. The individual server threads, forwards the requests from the client to their respective servers, and waits to receive the response back from them. The main mentionable classes here, in the database object example, are the ActiveServer and the RequestHandler as shown in Figure 2.6. The ActiveServer class receives the client requests and calls the appropriate function in the RequestHandler class. which was downloaded from the client. The RequestHandler class is the one that contains the methods for the object as described in the previous section.

In order to extend this framework to support multiple object types, a separate RequestHandler class has to be generated for each object type. The appropriate RequestHandler class is then called by the ActiveServer class depending on the object type.This is made possible by utilizing the property of reflection in Java language. This extensible architecture makes it very easy to add new objects and their functions to the framework. And as described above, the class file transfer capability allows the newly created RequestHandler classes of the objects to be transferred to the targets, along with the OSD commands. Thus extending the framework is made to become a trivial task.

The protocol used here for the client server communication is iSCSI. Configuration files have also been made use of, for specifying server parameters such as server names, port numbers, and directory paths for the object creation and so on.

2.4.3.3 OSD Design Changes

The OSD implementation used had to be tailored to fit the requirements of the application, in quite a few areas. The main idea adopted here was to relate the API of

the object to an OSD command, such that issuing the OSD command from the OSD initiator to an OSD target, would be equivalent to sending a request from the client to a server via a TCP/IP socket connection. we chose to add a new OSD EXECUTE METHOD command for these APIs, in order to keep the method execution code self contained.

The interface between the Java application program client and the OSD client is implemented as data pipes. The same holds for the interface between the OSD target and the Java server program. When the application client issues a request, the ACTIVEXEC module receives the request and packages it into an appropriate EXECUTE METHOD command CDB. Application level parameters, if any, are placed as object attribute values in the CDB, and so is the class file data. Once the CDB reaches the OSD target, it is decapsulated and the parameters are passed to the java server, which takes up the execution of the requested method.

2.5 Results

The experimental setup we used to test out our application is a 16 node Linux cluster. One node was set up as the client machine and up to 8 nodes were used as OSD targets. Several test runs were conducted, with varying number of targets and varying data set sizes. Some of our experimental results are tabulated as below.

Table 2.2 show the results for a data set of 1 million records. The total sort time is the time required for the client to issue sort execute methods to the targets and then retrieve the data and do a merge sort locally. The average sort time is the average time required on each target node to do the read from disk and then sort. The results show that the targets sort times decrease linearly. The total sort time shows improvement but is not linear with respect to the number of target nodes. The reason for this is that the Java virtual machine begins to run against its heap size limit and as a result the merge time takes longer than expected.

We also measured the effect of using iSCSI as an RPC transport mechanism rather than a purely TCP/IP socket based implementation. We constructed a socket-based implementation that was hard coded to perform application-specific RPCs. Through experiments, we found that the iSCSI OSD implementation is comparable in run-time to the socketbased implementation with no more than 10performance for the iSCSI implementation.

No. of OSD Targets	Total Sort time (ms)	Avg.Target Sort time (ms)
1	19931	16689
2	12645	8716
4	8761	4227
8	7437	2351

Table 2.2: Results for 1M Records

2.6 Summary

In this chapter, we have demonstrated an object-oriented active storage framework built upon the iSCSI OSD standard. Many data-intensive applications have small CPU and memory requirements and are attractive for execution across Active Disks. Implementing this in the form of objects allows for better portability, reusability of components, extensibility and other such advantages of object oriented programming. A further improvement to this framework is to extend it to support any type of object. Currently the application supports only List objects. Another idea to pursue is to explore alternate methods for parallel programming, so as to reduce or even eliminate the overhead introduced by multi-threading, for small data sizes.

Chapter 3

Active Storage for Database Applications

3.1 Introduction

Parallel query processing [GD92] has evolved from being a research idea (e.g., Gamma, XPRS [DSD+90] [WM94]) to being a standard feature provided by most DBMS vendors (e.g., Tandem, Teradata, Oracle, Informix XPS, and DB2). The parallelism in these systems is highly scalable, with vendors reporting good speedup with even 1000s of parallel nodes [GD92].

Traditionally, parallel query systems have been classified as shared nothing (SN), shared memory (SMP), and shared disk (SD) [Sto86]. A common characteristic of all these three types is that they rely on dedicated processors, pre-configured and pre-assigned for the parallel query task.

For SMP systems (and SMP nodes in SD/SN systems), this coupling is done in hardware. In SD systems, the compute nodes are often connected by specialized interconnects to a shared storage. SN systems are more loosely coupled due to a cluster-architecture. But the compute nodes still need to be dedicated for the parallel query task because the data is prepartitioned across the compute nodes (typically by hash or range of a join column value). In general, both SN and SD systems rely on partitioning as the primary means of parallelizing expensive query operations such as hash joins.

Active storage based database systems can be considered to be a shared nothing parallel query system, with each disk having its own dedicated processor. Since no resources are shared in a shared-nothing system, we make use of the idea of function shipping [BFG+95], wherein database operations are performed directly where the data resides. This is indeed the whole idea of Active Disks, where aplication processing is being pushed from the top tiers to as close to the actual data as possible. This minimizes network traffic by filtering out unimportant data and achieving good parallelism. A major task in a shared-nothing implementation is to split the incoming SQL request into many subtasks; these subtasks are then executed on different processors (if required, interprocess and interprocessor communication is used for data exchanges). Typically, a coordinator serves as the application interface, receiving the SQL request and associated host variables and returning the answers to the application.

In this chapter we propose an intelligent utility that gives suggestions to the DBA on optimal data partitioning rules for a parallel database application, based on the queries that come into the system. A parallel database uses multiple computation nodes to accelerate data processing. Using multiple computation nodes often requires that the data itself be split or partitioned across the different nodes, such that each node is responsible for a subset or copy of the data. Our partitioning utility provides a method to automatically partition or replicate data according to the requirements of the database application. It also allows parallel databases to be used with smart storage devices that can execute queries locally to the data. The organization of this chapter is as follows. First, we describe specific database related cases where a DBMS can benefit from an Active Storage System. Then we give the details of such a parallel DBMS architecture and the components comprising it. Finally we will describe the data partitioning utility, which helps to ease the transition of an enterprise DBMS to an Active Storage DBMS.

3.2 Potential Benefits

There are three main operations of importance in any relational database system,namely,select, aggregation and joins. In this section , we describe how these three operations can benefit from an Active Storage System.

3.2.1 Select

The select operation is an obvious candidate for an Active Disk function. The where clause in a SQL query can be performed directly at the drives, returning only the matching records to the host. The query is parsed by the host and the select condition is provided to all the drives. The drives then search all the records in a particular table in parallel and return only the records that match the search condition to the host, who combines the received records to form the final query result. A sample select query used in the TPC-D [Cou] benchmark is as given below:

*select * from lineitem where l_shipdate* $\geq$ '$1998-09-02$'

3.2.2 Aggregation

The purpose of sorting in database systems is almost always as an input step to another operation, either a join, an aggregation, or as the final step in a projection (duplicate elimination). The biggest benefit of performing sorting at Active Disks comes when it is combined with one of these other steps directly at the disks. Aggregation combines a set of records to compute a final sum, average, or count of groups of records with particular key values. If this summing or counting can be done at the disks as records are sorted on the group by columns then the network traffic can be greatly reduced by returning only the sums or counts from the individual disks for final aggregation at the host. A sample aggregation operation is illustrated by the query:

select sum(l_qty), sum(l_price(1-l_disc)) group by l_disc.*

3.2.3 Joins

A join combines data from two (or more) relations, so it is more complex than the select or project, which operate on only a single relation at a time.The purpose of a join is to combine two input relations, R and S on a single join attribute. If the value of the attribute for a particular record in R matches any record in S, then the combined record is output. Highly selective joins will benefit significantly from the reduction in data transfer by operating directly at the drive level. When the relations R and S are split properly accross all the disks, the joins can be performed in parallel at each , with the final merging of results at the host. More details on join processing appear in the subsequent sections.

3.3 Architecture Overview

The basic structure of a parallel DBMS is as shown in Figure 3.5. It consists of a single client/frontend, which maintains a two way communication channel with multiple database servers for query processing. When the client sends an SQL request, it is forwarded in parallel to all the servers. The servers execute on their local data and return the results to the client , where they are combined.

3.3.1 Partitioning Techniques

Once it is decided that the records of a table be split across multiple nodes, a suitable partitioning technique need to be applied on each table to assign each table row to a table partition. The common techniques in use today are listed below.

3.3.1.1 Round-Robin

In this scheme, each row that is added to a table , is sent to one of the available nodes sequentially in circular or round-robin fashion. This scheme ensures equal distribution of table rows across all nodes and hence a good technique to apply if load balancing is of high priority. The main drawback is that the data is randomly distributed and hence related data will not be placed together. This can be troublesome especially while processing joins.

3.3.1.2 Range

Range partitioning is a relatively simple scheme wherein table rows are split across multiple nodes based on the value of a specified table attribute. Each node will hold the rows of the table that fall within the attribute value range assigned to that node. In this case, similar data end up being together. But the data need not be equally distributed.

3.3.1.3 Hash

Hash partitoning is similar to range partitioning, except that here the table row is sent to a particular node based on the hash value of the specified table attribute. By chosing an efficient hash function, both equal distribution of data and co-location of related rows can be acheived.

The strategy chosen for the query execution mainly depends on which of the above mentioned partitioning techniques are used, as described below.

3.3.2 Query Execution Strategies

Query execution can be viewed as a flow of data along a plan path, with sends and receives being used for intertask communication. A query optimizer typically chooses :

- The optimal join order.
- The optimal way to access each table and compute each join.
- The nodes on which operations needs to be performed.
- Decide between system resources and response time as the metric for determining cost of a plan.

Consider the simple query from the TPC-C [Cou]benchmark, given below:

select S_NAME, S_ADDRESS from SUPPLIERS where S_REGION = 'ASIA''

In such queries, the execution plan is pretty straightforward. Regardless of how the SUPPLIER table data is split across the multiple nodes, each node has to return only those tuples with S_REGION = 'ASIA'Ėven in the case of an aggregation operation, such as a COUNT(*), each node need to return only its local count of tuples to the coordinator. The coordinator then sums up all the counts to return the result to the application. This is the plan depicted in Figure 3.1.

But when it comes to a join, an execution strategy decision actually has to be made. Consider a basic join query, taken again from the TPC-C benchmark.

select CUSTNAME from CUSTOMERS, ORDERS where O_CUSTKEY = C_CUSTKEY and O_ORDERDATE $\geq$ '$02-02-94$'

The query selects the names of all customers who placed orders after a certain date. It requires that the ORDERS and CUSTOMERS tables be joined on their CUSTKEY attribute. This join operation can be performed by a variety of different strategies in a parallel database environment.

3.3.2.1 Collocated Join

Let the partition keys of the ORDERS and CUSTOMERS tables be CUSTKEY and let them be in the same nodegroup. Then, the records of both tables having a particular CUSTKEY value will reside on the same node. For example, CUSTKEY

value of 10000 may be mapped to node 100 but is the same for both tables. Thus, the join operation can be performed on the local partitions of the two tables. No data exchange is required, and the entire operation can be done in one node process that scans the two tables and joins them and then ships the result to the coordinator. Figure 3.2 shows the task structure for this join.

3.3.2.2 Directed Join

Let the partition key for CUSTOMERS be CUSTKEY and ORDERS be ORDERKEY. Here, we cannot perform a collocated join operation since records of the ORDERS table with a particular CUSTKEY value could reside on all nodes. One efficient way to match the CUSTKEYs of ORDERS and CUSTOMERS is to hash the selected ORDERS rows using its CUSTKEY attribute and direct the rows to the appropriate CUSTOMERS nodes. This strategy localizes the cost of the join to partitions at each node and at the same time tries to minimize the data transfer. Figure 3.3 represents this.

3.3.2.3 Broadcast Join

Consider the following query between the CUSTOMERS and SUPPLIERS table.

select CUSTNAME, SUPPNAME, C_NATION from CUSTOMERS, SUPPLIERS where C_NATION = S_NATION.

The query tries to find customers and suppliers in the same region. Let the partitioning key for CUSTOMERS be CUSTKEY and that of SUPPLIERS be SUPPKEY. Note that C_NATION and S_NATION could have been the respective partition keys of the two tables; however, CUSTKEY and SUPPKEY are used more often in queries and are more likely candidates. Given this, the optimizer cannot try to localize the join operation on the C_NATION and S_NATION attributes. Hence, a strategy of broadcasting the selected rows of either table to all the nodes of the other tables is

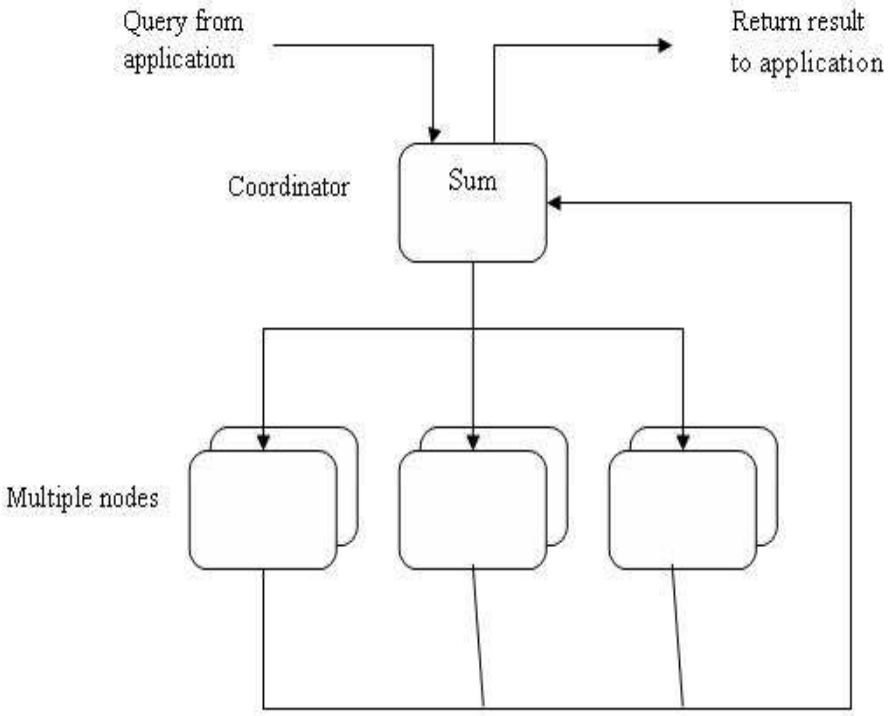

Figure 3.1: Simple Query

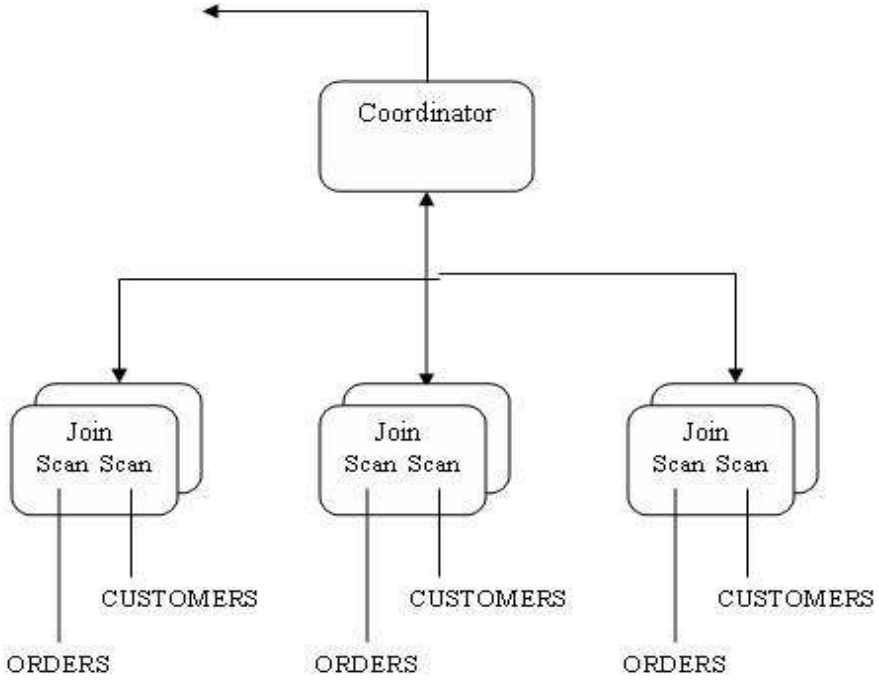

Figure 3.2: Collocated Joins

considered. The broadcast essentially causes one ttaob blee materialized fully at each node containing a partition of the other table. Now, a join at all nodes will produce the complete result of the query.

3.3.2.4 Repartitioned Join

In this strategy, the optimizer decides to explicitly repartition both tables on their joining attributes in order to localize and minimize the join effort. In the example query previously described for broadcast joins, the optimizer will repartition the CUSTOMERS table on C_NATION and the SUPPLIERS table on S_NATION on some common set of nodes. The repartitioned tables can then be joined in a collocated join fashion at each node, as shown in Figure 3.4

From the various query processing strategies described above, it is clear that a central controller/coordinator is a necessity for seamless communication between the client and the servers. It is the job of the coordinator to decide the routing of the queries to the appropriate servers and combining of the results coming from multiple nodes. Also, collocated join processing turns out to be the best candidate strategy for query processing. Only for collocated joins is the need for transfer of tuples between intermediate nodes completely eliminated, thus forming an efficient Active Storage parallel database system.

3.3.3 Middleware Details

The main component of interest here is the middleware which sits between the client and the servers. The middleware should have the capability to talk the database's backend and frontend protocol, and maintains a connection between them. Therefore, a database application (frontend) thinks that the middleware is the actual DB server, and the server (backend) sees the middleware as one of its clients. Because the middleware is transparent to both the server and the client, an existing database

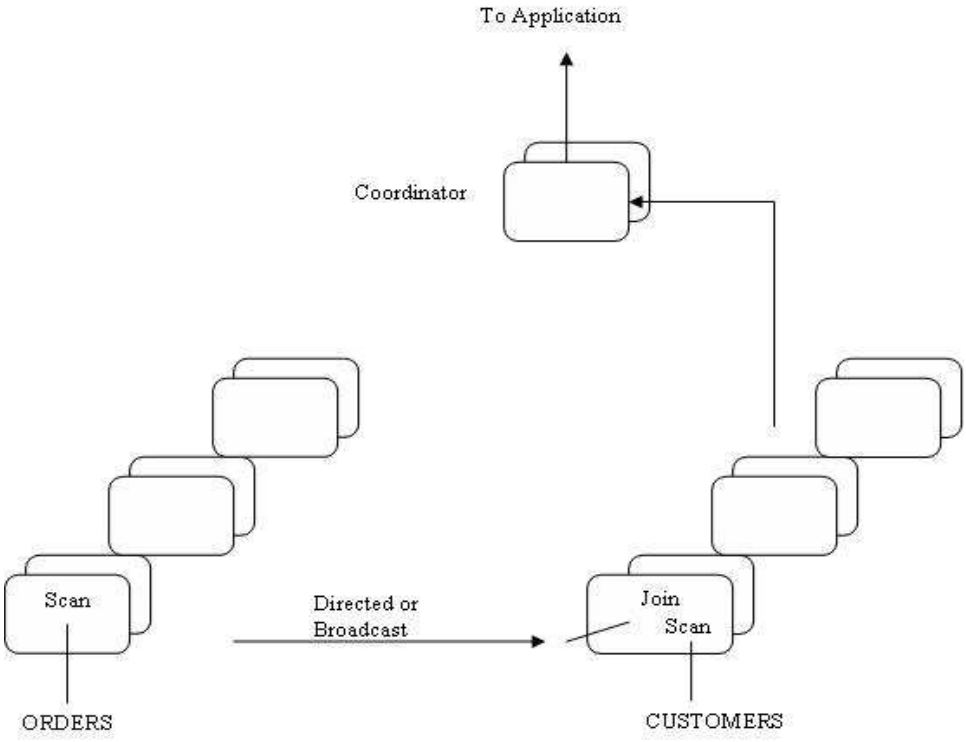

Figure 3.3: Directed and Broadcast Join

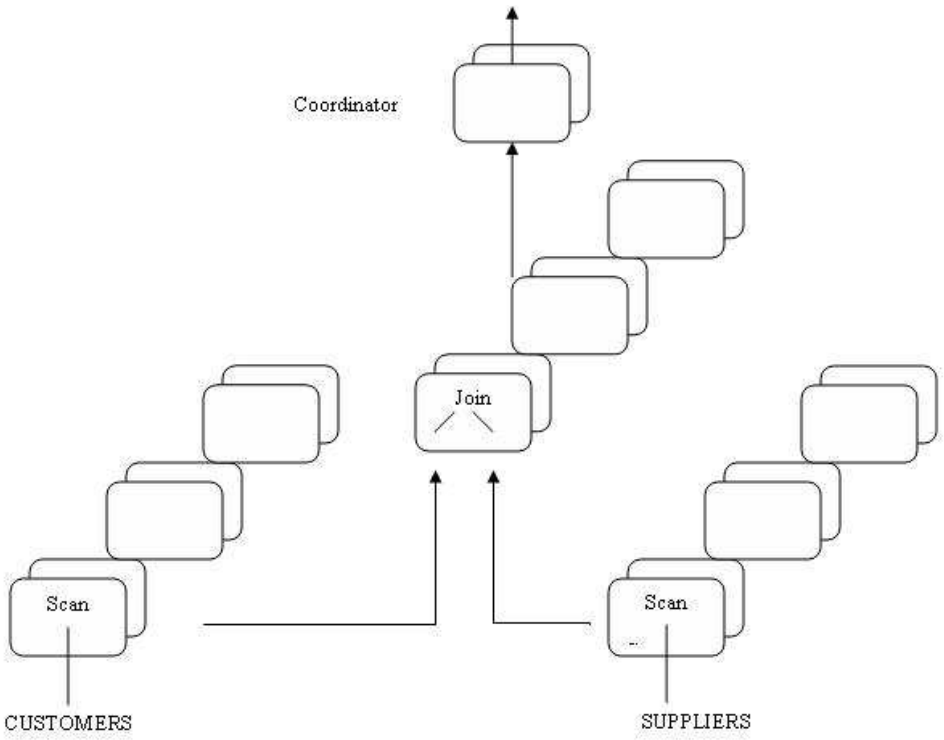

Figure 3.4: Repartitioned Join

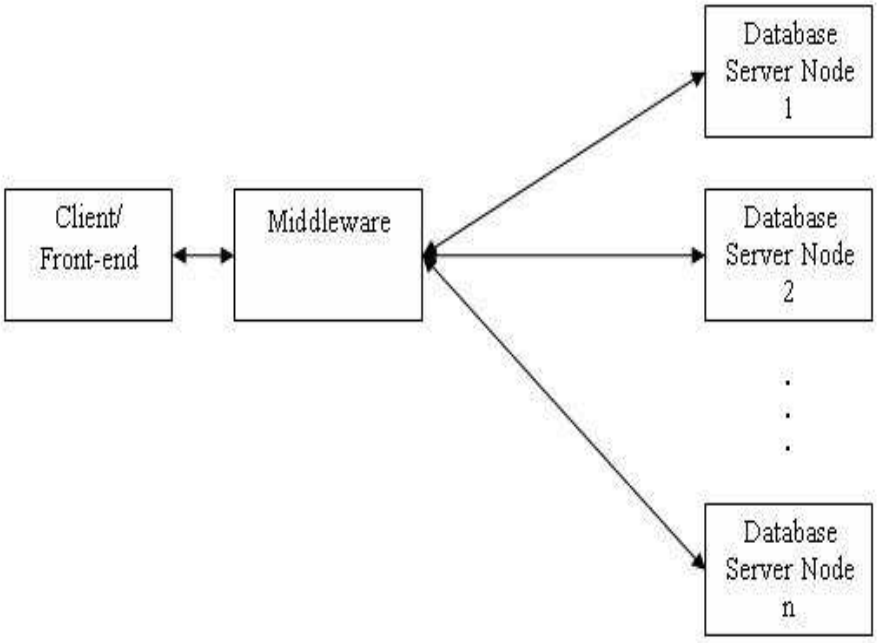

Figure 3.5: Parallel Database Architecture

application can be used with the middleware almost without any change to its sources.

The middleware that is stuck between the client and the servers should cater to a few important features, as described below.

3.3.3.1 Replication

Replication enables the same data to be copied to multiple database nodes. Tables can be registered within the middleware, for replication beforehand. If a replication rule has been set for a partcular table, then during table creation time, the table is created across all the nodes attached to the middleware. And each record that is added to the replicated table is in effect inserted into the replicated tables in each node. Using the replication functionality enables creating a realtime backup on two or more physical disks, so that the service can continue without stopping servers in case of a disk failure.

3.3.3.2 Load Balancing

If a database is replicated, executing a SELECT query on any server should return the same result. This paves way for exploiting the replication feature described above ,inorder to reduce the load on each server by distributing incoming SELECT queries among multiple servers.This can greatly improve the system's overall throughput as multiple queries are being processed in parallel. At best, performance improves proportionally to the number of backend servers available. Load balance works best in a situation where there are a lot of users executing many queries at the same time.

3.3.3.3 Parallel Querying

Just as in the replication of tables,in the parallel query mode also, the tables are registered beforehand to be distributed across multiple nodes. Such registered

tables are created on all the nodes. But unlike as in replication, the table records are not inserted into all nodes. Instead, data partitioning rule functions can be set up for each table, which takes a value for the partitioning key column, and returns the corresponding DB node ID. When tuples are added to a table, it is added to the DB node whose ID is returned by the partitioning function for that table. Hence each tuple in the original table is going to reside in only one of the multiple nodes available. When a query comes in, the middleware routes it to all the servers, thus enablinbg the query be executed on all the servers concurrently to reduce the overall execution time. The partial results obtained from query execution on each of the servers, are collected by the middleware, where it is combined to form the final result, that is returned to the client. It should also be possible to use replication in conjunction with paralle querying.

Any generic middleware that provides for these functionalities can be used as the coordinator for a parallel database system. Load balancing might not be required as a feature in all cases, esecially if the parallel query data distribution functions are efficient enough to ensure an equal distribution of data among all the nodes.

3.4 Partition Optimizer

Traditionally, the DBMS relies on partitioning to get parallelism in I/O, and to reduce network traffic. The task here is to break up the select-project-join fragments, with aggregation and group by(SPJAG) over tables in the shared storage into SPJAGs over work-units that can be done on the coprocessors.

3.4.1 Partitioning Rules

Work allocation in parallel query plans is most popularly done through Exchange [G.G90] operators. Both base tables and intermediate results are split according to the join column as part of query execution, and each node gets to operate on a table

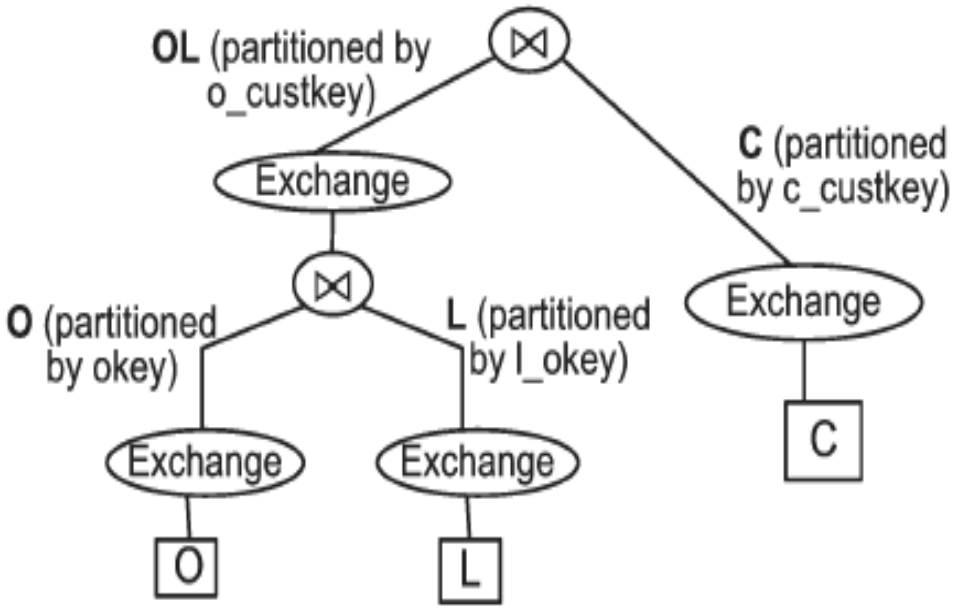

Figure 3.6: Exchange-based plan for OxLxC [G.G90]

subset with particular hash value (or range) of the join column. Figure 3.6 shows an example plan. The problem with this work allocation approach as listed below:

- First, since tuples are shipped between plans, Exchange is very vulnerable to load spikes and failures at even a single compute node.
- Shipping large sets of tuples between nodes also increase network bottleneck.

To avoid these limitation, we partition data not at the operator level but rather at the join fragment level and send the entire SPJAG query to each node, to be run independently. The results from each node are then combined at the coordinator/client node to produce the final result set of the query. In our architecture, the middleware functions as the coordinator. But there are a few details to consider before being able to send the entire query to each node.

Single table queries, whether they be selects, aggregates, group by etc, will work flawlessly with any kind of partitioning scheme for the tables, with the SN network architecture. This is so since there is no dependency for the query with any other table and hence it does not matter how the tables other than the table of concern is partitioned. The only consideration here is to partition the table on any one table column equally across all the nodes, inorder to maximize parallel efficiency of the system. But when it comes to two or multi table queries, such as joins, the choice of the partitioning table column becomes a very cruicial decision. Since we are dealing with static sets of table data in each of the nodes, the partitions should be such that all the table tuples required to process a multi-table query, for eg. a join query, at any node should lie within that node itself. Such a partition reduces, if not completely eliminates transfer of immediate results between nodes. Hence the need to find the optimal set of partitioning key columns for each table in the database.

The main idea behind our intelligent data partitioning utility is that , in any real world enterprise database application, the queries that have to be processed by an

application will belong to a certain finite set of queries. This finite set of queries may be known either at the onset of the application or will gradually grow and reach a steady state when the application stabilizes. Initially, the database tables are partitioned on a key column, chosen intuitively by the DBA. This initial choice of the key column can be made based on the expected probability of join queries, the size of the tables, and the general purpose of the application. In the case of transitioning an existing application to a parallel DB system,, there will be no initial partitioning. The next step is to insert a query logger into the middleware, which will log all queries that come into the DBMS, on a daily basis. The logged queries can then be used as the input data for the partitioning utility. This helps in modeling the real-time, real-world usage of the application.

The partitioning utility parses each of the logged queries and creates a mapping between the different tables and their columns on which joins are being performed. It also keeps a count of the number of times a join on each table column is being invoked. The utility then selects the column , on which the maximum number of joins occur, for each table. Finally, it outputs the selected key columns for each table, as the optimal partitions. It might be necessary to analyse the queries logged on more than a single day, depending on the usage of the system, inorder for the partitioning utility to reach its decision and train the system more accurately.

Another piece of information that the utility uses to suggest an optimal partition, is the size of the database tables, if provided. If it is seen that a particular table is small in size, but accessed frequently in a lot of queries, then it makes sense to replicate that table across all nodes. The utility will then suggest that table as a candidate for replication on all the nodes, instead of partitioning it across the nodes.

Ultimately, the decision whether to repartition/replicate the tables depends on the DataBase Administrator(DBA). The partitioning utility's main purpose is to help the DBA make a better decision, by analyzing the realtime usage of the system.

3.5 Results

3.5.1 Database Specification

We make use of the open source PostgreSQL database,version 8.3.3, [Pos] for our implementations and analysis. In database jargon, PostgreSQL uses a client/server model. A PostgreSQL session consists of the following cooperating processes (programs):

- A server process, which manages the database files, accepts connections to the database from client applications, and performs actions on the database on behalf of the clients.
- The users client (frontend) application that wants to perform database operations. Client applications can be very diverse in nature: a client could be a text-oriented tool, a graphical application, a web server that accesses the database to display web pages, or a specialized database maintenance tool.

As is typical of client/server applications, the client and the server can be on different hosts. In that case they communicate over a TCP/IP network connection.

3.5.2 Middleware Specification

PostgreSQL recently came up with a middleware called pgpool, that works between PostgreSQL servers and a PostgreSQL database client. It provides all the features that we requires for our purposes and then some. Some of the execution modes provided by pgpool as as listed below.

- Limiting exceeding connections,ie, queueing the excess inorder to reduce resource consumption and maintain system performance.
- Connection pooling, ie, reusing connections to the servers to reduce connection overhead and improve system throughput.

- Replication.
- Load Balance.
- Parallel Query.

Pgpool has a main configuration file , in which the required modes of operation can be set. The interdependence of the various modes are as shown in Table 3.1. The System DB is a central database that maintains partition and replication rules, in the format of a normal database table. Tables to be partitioned are registered in the dist_def table of System DB. The general format of a partitioning rule is as below.

CREATE TABLE dist_def(
dbname TEXT, – database name
schema_name TEXT, – schema name
table_name TEXT, – table name
col_name TEXT NOT NULL CHECK (col_name = ANY (col_list)), – partitioning key column name
col_list TEXT[] NOT NULL, – names of table attributes
type_list TEXT[] NOT NULL, – types of table attributes
dist_def_func TEXT NOT NULL, – name of the partitioning rule function
PRIMARY KEY (dbname,schema_name,table_name));

Likewise, tables to be replicated are registered with the replicate_def table, the format of which is as given below.

CREATE TABLE replicate_def(
dbname TEXT, –database name
schema_name TEXT, –schema name
table_name TEXT, –table name
col_list TEXT[] NOT NULL, – names of table attributes

type_list TEXT[] NOT NULL, – types of table attributes

PRIMARY KEY (dbname,schema_name,table_name));

As with the client and the servers, pgpool and the System DB can reside on a separate nodes. For our experiments, we set Parallel Query mode to true and both Replication

Function/Mode	Connection Pool Mode	Replication Mode	Parallel Query Mode
Connection Pool	O	O	O
Replication	X	O	X
Load Balance	X	O	X
Parallel Query	X	X	O
No. of Servers	1 or more	2 or more	2 or more
System DB required?	no	no	yes

Table 3.1: Configuration modes for pgpool

and Load Balancing modes to false. If tables need to be replicated, set Replication mode to true, add the table data and then restart pgpool with the above said mode settings.

3.5.3 Testbench Schema

The testbench that we use for our setup is the OSDL's Database Test Suite (OSDL-DBT2) [Sui], inspired from the TPC-C benchmark [Cou]. The benchmark is centered around the principal activities (transactions) of an order-entry environment. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. While the benchmark portrays the activity of a wholesale supplier, it is not limited to the activity of any particular business segment, but, rather represents any industry that must manage, sell, or distribute a product or service.

The benchmark consists of eight tables , the schema of which is as shown in the tables 3.2 to 3.9. We ran the benchmark for about 6 hours, with a warehouse scale factor of 10 and a single connection. The cardinalities of the database tables,

Column Name	Column Type
w_id	INTEGER
w_name	VARCHAR(10)
w_street_1	VARCHAR(20)
w_street_2	VARCHAR(20)
w_city	VARCHAR(20)
w_state	char(2)
w_zip	char(9)
w_tax	REAL
w_ytd	NUMERIC

Table 3.2: Warehouse

Column Name	Column Type
d_id	INTEGER
d_w_id	INTEGER
d_name	VARCHAR(10)
d_street_1	VARCHAR(20)
d_street_2	VARCHAR(20)
d_city	VARCHAR(20)
d_state	char(2)
d_zip	char(9)
d_tax	REAL
d_ytd	NUMERIC
d_next_o_id	INTEGER

Table 3.3: District

Column Name	Column Type
c_id	INTEGER
c_d_id	INTEGER
c_w_id	INTEGER
c_first	VARCHAR(16)
c_middle	char(2)
c_last	VARCHAR(16)
c_street_1	VARCHAR(20)
c_street_2	VARCHAR(20)
c_city	VARCHAR(20)
c_state	char(2)
c_zip	char(9)
c_phone	char(16)
c_since	TIMESTAMP
c_credit	char(2)
c_credit_lim	NUMERIC
c_discount	REAL
c_balance	NUMERIC
c_ytd_payment	NUMERIC
c_payment_cnt	REAL
c_delivery_cnt	REAL

Table 3.4: Customer

Column Name	Column Type
no_o_id	INTEGER
no_d_id	INTEGER
no_w_id	INTEGER

Table 3.5: New Order

Column Name	Column Type
o_id	INTEGER
o_d_id	INTEGER
o_w_id	INTEGER
o_c_id	INTEGER
o_entry_d	TIMESTAMP
o_carrier_id	INTEGER
o_ol_cnt	INTEGER
o_all_local	REAL

Table 3.6: Orders

Column Name	Column Type
ol_o_id	INTEGER
ol_d_id	INTEGER
ol_w_id	INTEGER
ol_number	INTEGER
ol_i_id	INTEGER
ol_supply_w_id	INTEGER
ol_delivery_d	TIMESTAMP
ol_quantity	REAL
ol_amount	REAL
ol_dist_info	VARCHAR(24)

Table 3.7: Order Line

Column Name	Column Type
i_id	INTEGER
i_im_id	INTEGER
i_name	VARCHAR(24)
i_price	REAL
i_data	VARCHAR(50)

Table 3.8: Item

Column Name	Column Type
s_i_id	INTEGER
s_w_id	INTEGER
s_quantity	VARCHAR(24)
s_dist_01	VARCHAR(24)
s_dist_02	VARCHAR(24)
s_dist_03	VARCHAR(24)
s_dist_04	VARCHAR(24)
s_dist_05	VARCHAR(24)
s_dist_06	VARCHAR(24)
s_dist_07	VARCHAR(24)
s_dist_08	VARCHAR(24)
s_dist_09	VARCHAR(24)
s_dist_10	VARCHAR(24)
s_ytd	NUMERIC
s_order_cnt	REAL
s_remote_cnt	REAL
s_data	VARCHAR(50)

Table 3.9: Stock

generated with the available benchmark scripts is as shown in Table 3.10. The

Table Name	Cardinality
Warehouse	10
District	100
Customer	300000
Item	10000
Orders	300000
Stock	1000000
New_order	90000
Order_line	2848020

Table 3.10: Database Table Cardinalities

SQL queries thus generated, and logged with the query logger inserted into pgpool, were used as the data for our data partitioning utility. The partitioning suggestions returned by our utility is as summarized in Table 3.11. The partition suggestions for the three DB tables as listed, are based on the real-time join queries that are generated by the DBT2 benchmark. A suitable distribution function can now be set up for the suggested tables, based on their suggested columns. Since the warehouse table is small in size and frquently accessed, it is suggested to be replicated across all nodes. It has to be noted here that even though the District table is comparitively small in size, since it is suggested to be partitioned, it will not be suggested for replication too. The tables that are not mentioned in the utility result can be hash partitioned on any one of their columns as the location of such records will not affect system performance.

3.6 Summary

In this chapter, we proposed and implemented a method for intelligent data partitioning for parallel databases.Current parallel databases partition data based on a database administrators knowledge of the data structure and is partitioned statically by the database administrator. With large databases, it may be difficult for

a database administrator to pick an optimal partition that delivers the best performance. The novelty of our method is the use of query logging to analyze previous queries and thus, create an optimal partition. Since it learns from past queries, it can generate more efficient partitions than an administrator generated partition. An avenue for further research in this area is to verify the partition algorithm on different data sets and storage architectures. This thesis verifies the utility only on the open source DBT2 benchmark. Also, these algorithms will need to be expanded to handle multi-table joins and investigate query dependent layout strategies.

Partitions	
Table Name	Column Name
stock	s_i_id
order_line	ol_w_id
district	d_w_id

Replications
warehouse

Table 3.11: Partition Result

Chapter 4

Conclusions

In this thesis, we explored Active Disks and the integration of Active Storage Systems with real world applications. In particular, we investigated the Object-based Storage Device (OSD) standard which provides a major step towards enabling explicit application awareness in storage systems behind a standard, fully interoperable interface. OSD provides the right interface for an active storage system, which prompted us to develop an object oriented framework for such application aware storage systems. Our framework answers the question of how to enable downloading of application code to storage nodes, thus allowing for explicit communication between the application and the storage system. It also proves that application execution time improves significantly, though not linearly, through active commputation.

This thesis also studied Active Storage for database applications, which is the ideal candidate for utilizing the OSD framework described above. The database-aware storage plans described in [RSI07] lays the ground work for such a system. Our particular contribution was to develop an intelligent data partitioning method that maximizes the efficiency and accuracy of a parallel database application built on the premises of an OSD Active Storage system. This utility helps to remove the inaccuracy inherent with the intuitive decision taken by the DBA on how to partition data in such a parallel DB system. This is particularly important when an existing

enterprise application is being migrated from a single server system to a multi server parallel database system. In such a scenario, the right partitioning of the table data is crucial inorder for the accurate operation of the system.

Bibliography

[ANS04] ANSI. **Information Technology - SCSI Architecture Model - 3 (SAM-3)**, September 2004.

[ANS08] ANSI. **Information Technology - SCSI Object Based Storage Device Commands -2 (OSD-2)**, January 2008.

[AUS98] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In **Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems**, 1998.

[BAC+90] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. **IEEE Transactions on Knowledge and Data Engineering**, 2(1), 1990.

[BFG+95] Chaitanya Baru, Gilles Fecteau, A Goyal, H Hsiao, A Jhingran, S. Padmanabhan, G. P. Copeland, and W. G. Wilson. Db2 parallel edition. **IBM Systems**, 1995.

[BMLX03] Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Lan Xue. Efficient metadata management in large distributed storage systems. In **Proceedings of the IEEE/NASA Goddard Symposium on Mass Storage Systems and Technologies**, pages 290–298, April 2003.

[BS02] Eitan Bachmat and Jiri Schindler. Analysis of methods for scheduling low priority disk drive tasks. In **Proceedings of the ACM SIGMOD International Conference on Management of Data**, pages 55–65, 2002.

[Cou] Transaction Processing Performance Council. http://www.tpc.org.

[DH81] D.J. DeWitt and P. Hawthorn. A performance evaluation of database machine architectures. In **Proceedings of International Conference on Very Large Data Bases (VLDB)**, September 1981.

[DHH+06] D. Du, D. He, C. Hong, J. Jeong, V. Kher, and Y. Kim. Experiences in building an object-based storage system based on the osd t-10 standard. Technical Report DTC-13-006, Digital Technology Center, University of Minnesota, 2006.

[DSD+90] D.J. Dewitt, S.Ghandeharizadeh, D.A.Schneider, A.Bricker, Hsaio H.I, and R.Rasmussen. The gamma database machine project. In **IEEE Transactions on Knowledge and Data Engineering**, March 1990.

[FNN+05] Michael Factor, David Nagle, Dalit Naor, Eric Reidel, and Julian Satran. The osd security protocol. In **Proceedings of the International IEEE Security in Storage Workshop**, December 2005.

[GD92] Jim Gray and David.J. DeWitt. Parallel database systems: The future of high performance database systems. **Communications of the ACM**, 35(6), February 1992.

[G.G90] G.Graefe. Encapsulation of parallelism in the volcano query processing system. In **ACM SIGMOD Record**, June 1990.

[GNA+96] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, and David Rochberg. A Case for Network-Attached Secure Disks. Technical Report CMU-CS-96-142, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, September 1996.

[GNA+97] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. In **Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems**, June 1997.

[GNA+98] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Harding, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In **Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems**, October 1998.

[Gra97] J. Gray. What happens when processing, storage, and bandwidth are free and infinite? In **Proceedings of IOPADS**, November 1997. Keynote Address.

[GVM00] Garth A. Gibson and Rodney Van Meter. Network attached storage architecture. **Communications of the ACM**, 43(11):37–45, November 2000.

[Knu79] Donald E Knuth. The art of computer programming. 3, 1979.

[KPH98] K. Keeton, D. A. Patterson, and J. M. Hellerstein. The intelligent disk (IDISK): A revolutionary approach to database computing infrastructure. Technical report, University of California at Berkeley, May 1998. White Paper.

[LDR05] Yingping Lu, David Du, and Tom Ruwart. Qos provisioning framework for an osd-based storage system. In **Proceedings of the IEEE/NASA Goddard Symposium on Mass Storage Systems and Technologies**, April 2005.

[MSC+86] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. Rosenthal, and F. Donelson Smith. Andrew: A distributed personal computing environment. **Communications of the ACM**, 29(3):184–201, March 1986.

[OSS75] E. A. Ozharahan, S. A. Schuster, and K. C. Smith. RAP: Associative processor for database management. In **Proceedings of AFIPS Conference**, 1975.

[PNF07] Juan Piernas, Jarek Nieplocha, and Evan J. Felix. Evaluation of active storage strategies for the lustre parallel file system. In **Proceedings of Supercomputing**, November 2007.

[Pos] PostgreSQL. http://www.postgresql.org/.

[RCT94] T. Ts'o R. Card and S. Tweedie. Design and implementation of the Second Extended File System. In **Dutch International Symposium on Linux**, 1994.

[RGF98] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In **Proceedings of International Conference on Very Large Data Bases (VLDB)**, August 1998.

[Rie99] Erik Riedel. **Active Disks - Remote Execution for Network-Attached Storage**. PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University, 1999. Tech. Report no. CMU-CS-99-177.

[RR95] R.Agrawal and R.Srikant. Fast algorithms for mining association rules. In **Proceedings of International Conference on Very Large Data Bases (VLDB)**, September 1995.

[RSI07] Aravindan Raghuveer, Steven W. Schlosser, and Sami Iren. Enabling database-aware storage with osd. In **Proceedings of the IEEE/NASA Goddard Symposium on Mass Storage Systems and Technologies**, 2007.

[Sto86] Michael Stonebraker. The case for shared nothing. **Database Engineering**, 9:4–9, 1986.

[Sui] Database Test Suite. http://osdldbt.sourceforge.net/.

[SW96] Stefan Savage and John Wilkes. AFRAID – A frequently redundant array of independent disks. In **Proceedings of the USENIX Technical Conference**, pages 27–39, January 1996.

[vN63] John von Neumann. Collected works. 6, 1963.

[WM94] W.Hong and M.Stonebraker. Optimization of parallel query execution plans in xprs. In **Readings in database systems (2nd ed.)**, June 1994.

[YL75] S. Y. W. Yu and G. J. Lipvski. CASSM: A cellular system for very large data bases. In **Proceedings of International Conference on Very Large Data Bases (VLDB)**, 1975.