Active Disk Architecture for Databases

Erik Riedel[1], Christos Faloutsos, David Nagle

April 2000
CMU-CS-00-145

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

{riedel,christos,nagle}@cs.cmu.edu

Abstract

Today's commodity disk drives, the basic unit of storage for computer systems large and small, are actually small computers, with a processor, memory and a network connection, in addition to the spinning magnetic material that stores the data. Large collections of data are becoming larger, and people are beginning to analyze, rather than simply store-and-forget, these masses of data. At the same time, advances in I/O performance have lagged the rapid development of commodity processor and memory technology. This paper describes the use of Active Disks to take advantage of the processing power on individual disk drives to run a carefully chosen portion of a relational database system. Moving a portion of the database processing to execute directly at the disk drives improves performance by: 1) dramatically reducing data traffic; and 2) exploiting the parallelism in large storage systems. It provides a new point of leverage to overcome the I/O bottleneck. This paper discusses how to map all the basic database operations - select, project, and join - onto an Active Disk system. The changes required are small and the performance gains are dramatic. A prototype based on the Postgres database system demonstrates a factor of 2x performance improvement on a small system using a portion of the TPC-D decision support benchmark, with the promise of larger improvements in more realistically-sized systems.

1. now with Hewlett-Packard Labs, riedel@hpl.hp.com

This research was sponsored by DARPA/ITO through ARPA Order D306, and issued by Indian Head Division, NSWC under contract N00174-96-0002. Partial funding was provided by the National Science Foundation under grants IRI-9625428, DMS-9873442, IIS-9817496, and IIS-9910606. Additional funding was provided by donations from NEC and Intel. We are indebted to generous contributions from the member companies of the Parallel Data Consortium. At the time of this writing, these companies include Hewlett-Packard Laboratories, LSI Logic, Data General, EMC, Compaq, Intel, 3Com, Quantum, IBM, Seagate Technology, Hitachi, Infineon, Novell, and Wind River Systems. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any supporting organization or the U.S. Government.

ACM Computing Reviews Keywords: B.4.2 Input/output devices, H.2.8 Database applications, C.3.0 Special-purpose and application-based systems, B.4 Input/Output and Data Communications.

1. Introduction

An Active Disk is a novel storage architecture that leverages the computation power available in commodity disk drives to perform application-level processing. Instead of simply processing low-level storage protocols such as SCSI, these disk drives are able to execute application-level code, making significantly better use of storage's aggregate processing power and the interconnection network between storage devices and servers [Acharya98, Keeton98, Riedel98]. Applications can take advantage of the parallelism in large storage systems and the reduction in bandwidth possible if data is operated on at the edges of the network, before it is placed onto the expensive, shared storage interconnect.

The motivation for Active Disks is illustrated in Table 1, which shows two commercial database systems and compares the aggregate computation power in the host and at the disks. If each disk drive contains a processor with a 100 MHz RISC core, which modern drive chips are beginning to offer [Cirrus98, Siemens98], the aggregate computation power across storage is nearly an order of magnitude more than the host itself. This is true even in large SMP systems, where the number of disks is scaled up to balance the demands of the processors. In addition, the aggregate bandwidth available at the disks usually far exceeds the ability of the I/O sub-system to feed the processors. Using the disks' processing power to offload a portion of the host work is a promising way to take advantage of these trends, given the inherent physical and cost limitations to building very large SMP systems and very high bandwidth I/O channels. A system with Active Disks can achieve much higher application-level throughput than a system with traditional disks by offloading a significant portion of the host's work to the disks and reducing interconnect traffic.

Previous work [Acharya98, Keeton98, Riedel98] has shown that similar systems can achieve performance increases across a range of applications such as data mining, image processing, and some database functions. In this paper, we focus on how Active Disks can support a real database system. We begin by explaining the modifications to a relational database system necessary to take advantage of Active Disks, and then describe our prototype implementation using the PostgreSQL database system. We show that all the basic database operations can be mapped onto Active Disks with an appropriate choice of low-level primitives. Finally, we use a portion of the TPC-D

System	Use	Host Processing	Disks	On-Disk Processing	Disk Advantage	System Bus	Storage Throughput	Mismatch Factor
AlphaServer 1000/500	TPC-C, OLTP	500 MHz	61	6,100 MHz	12.2 x	266 MB/s	610 MB/s	2.3 x
1 x 500 MHz			266 GB			64-bit PCI		
AlphaServer 8400	TPC-D 300, DSS	7,344 MHz	521	52,100 MHz	7.1 x	532 MB/s	5,210 MB/s	9.8 x
12 x 612 MHz SMP			2.2TB			2 x 64-bit PCI		
Proliant 8000	TPC-H 100, DSS	4,400 MHz	152	15,200 MHz	3.5 x	266 MB/s	1,520 MB/s	5.7 x
8 x 550 MHz SMP			1.2 TB			64-bit PCI		

Table 1: Comparison of computing power vs. storage power in three commercial database servers. If we estimate that next-generation disk drives will have 100 MHz of available processing power, large database systems will contain much more processing power on their combined disks than at the server processors. In addition, even assuming only a conservative 10 MB/s per disk for sequential access, the I/O sub-systems of these servers cannot deliver the aggregate bandwidth of this number of disk drives. All three systems are audited TPC benchmark systems with published configurations and performance results [TPC97, TPC98, TPC00] and are representative of the disk to processor ratios in these types of systems. The AlphaServer 1000 systems most closely matches the prototype "host" system used in this paper, while the 8400 and Proliant represent larger-scale systems for decision support.

decision support benchmark to demonstrate improvements between 10% and a factor of 2x in our small prototype system - with the promise of larger improvements in more realistically-sized systems such as the ones in Table 1.

	Compaq SCSI disk drive	Quantum Snap Server
capacity	9.1 GB	10.0 GB
disk speed	10,000 rpm	5,400 rpm
average seek	5.6 ms	9.5 ms
processor	25 MHz RISC core	133 MHz Pentium
memory	4 MB	64 MB
weight	0.6 kg	1.6 kg
volume	370 cm^3	2100 cm^3
interface	40 MB/s SCSI-3	100 Mb/s 100Base-TX
cost	$449	$499

Table 2: Comparison of SCSI disk and network storage device. The table shows the characteristics of the disk drives in the Proliant TPC-H system shown in Table 1 compared with a Snap Server 1000 from Quantum. The Snap Server contains a disk drive with a second processor and memory board to provide the network-attached storage functionality. This is still below the level of integration and cost reduction that should be possible with the drive control chips currently being developed, but the direction in performance and price comparison is promising [Compaq00, Quantum99].

While disk drives using these 100 MHz and faster control chips and many megabytes of memory are not yet available, the integration trends in the industry are promising. Table 2 compares the performance and cost of the high-end SCSI disk drive used in the Proliant system from Table 1 with a network-attached storage device available today that already contains this level of processing power.

The rest of this paper is organized as follows. Section 2 briefly discusses previous work on database machines and the trends that have changed the landscape since the time of that research. Sections 3 and 4 describe the mapping of the core database functions - select, project, and join - onto Active Disks, and the modifications to the PostgreSQL database system to support Active Disks. Sections 5 describes our experimental setup and compares the performance of a prototype system using Active Disks against a database server with traditional disks. Section 6 discusses related work. Finally, Section 7 concludes and briefly discusses areas of future work.

2. Background

The basic idea of executing database functions in processing elements directly attached to individual storage devices was explored extensively in the context of database machines such as CASSM [Su79], RAP [Ozkarahan75], and numerous others [DeWitt81]. These machines implemented a variety of algorithms directly in hardware and showed dramatic speedups on certain query streams. There were several difficulties, including basic technology limitations, that prevented these systems from becoming widely available, but the landscape has changed sufficiently since the time of this research to merit a re-evaluation of this work.

The main counter-arguments to the database machines of the 70s and 80s were summarized by Boral and DeWitt. Specifically, that 1) a single general-purpose host processor was sufficient to execute select at the full data rate of a single disk, so no additional hardware was necessary, 2) special-purpose hardware increased the design time and cost of the machine, and 3) for a significant fraction of database operations, such as sorts and joins, simple select filters in hardware did not provide significant benefits [DeWitt81, Boral83].

Changes in storage technology in the intervening years have allowed disk performance to catch up with the power of host processors. The performance of single disks has increased about thirty-fold and continues to grow at 20% per year. However, the biggest change since that time is the widespread use of disk arrays that use a large number of disks in parallel. This allows aggregate storage bandwidth to meet, and exceed, host processing rates simply by adding disks, pushing the bottleneck back into the CPU and I/O interconnects.

At the same time, the rapid improvements in silicon technology allow general-purpose processing cores [Turley96] to replace what would have been special-purpose silicon in the older database machines. This, along with the growing popularity of a common "mobile" programming language in Java [Gosling96, Levin99], overcomes the limitation of developing special-purpose microcode useful only for very specific custom hardware. The basic challenge to Active Disk code is ensuring that parallel code exists for executing the basic parts of an algorithm, so that these can be moved to the disks. In the context of database systems, such parallel algorithms have been studied in numerous projects since the days of the database machines [DeWitt92].

We will demonstrate that by taking advantage of these technology changes, all of the core database functions, not just select, can be efficiently implemented using a relatively small amount of code executed directly at the storage devices - and that the performance benefits are compelling given the technology that will be available in the near future. We discuss select, project (with the closely-related aggregation), and join and describe the architecture and algorithms to efficiently execute any SQL query against a table stored across a set of Active Disks.

3. Database Operations & Algorithms

This section outlines the basic operations in a relational database system and describes how Active Disks can be used for each of them. The most effective algorithms for Active Disks are those that can operate in parallel across a large number of disks, are highly selective to reduce network traffic, have low communication among disks, and low memory requirements. The intent is not to replace general-purpose processors or large-memory multi-processors, but to take advantage of a pre-packaged combination of processing power, memory, and bandwidth to accelerate I/O-intensive processing.

3.1. Scan - Select

The `select` operation, as shown in Figure 1, is an obvious candidate for an Active Disk function. The `where` clause in a SQL query can be performed in parallel at the drives, each operating on its local data and returning only the matching records to the host. This is the operation that was implemented by many of the early database machines [Ozkarahan75, Smith79, Su79, Martin94]. The query is parsed by the host and the select condition provided to all the drives. The drives then search all the records in their portion of a table in parallel and return only matching records. If the condition is highly selective, this greatly reduces the amount of data that must traverse the interconnect, as records that will not be part of the result must never leave the disks.

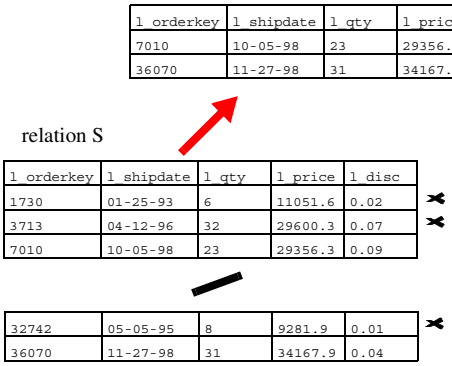


Figure 1: Illustration of a select operation in a database system. A single relation is searched for records that match the given value for the l_shipdate attribute. There are two means of data reduction in a select. The first is the selection of matching records (5% in this example) and the second is the elimination of unneeded columns from the relation (over 75% of the data in this example).

```
select l_qty, l_price from lineitem
where l_shipdate >= '1998-09-02'
```

This operation requires very little state at the drives, as only the search condition and the page currently being operated on must be stored in drive memory.

3.2. Aggregation - Project

The basis for projection and aggregation is the elimination or combination of duplicates, which can be done in two ways: a) sorting or b) hashing. Sorting is one of the basic functions within a database system, and a number of algorithms have been proposed for mapping sort to an Active Disk-style system [Keeton98]. However, sorting is not one of the most effective Active Disk applications due to the large amount of communication required among nodes for a full sort. This problem is simplified in the context of the database system because sorting is almost always used 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 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. The disks return only the sums or counts from the individual disks for final aggregation at the host. Similarly, if duplicate elimination can be done while sorting locally at the disks, then the duplicate records must not be needlessly transferred across the network. An example aggregation operation is shown in Figure 2 which illustrates a query to determine the total number of items returned by customers, and the total revenue lost due to these returns, summarized by reason for the return.

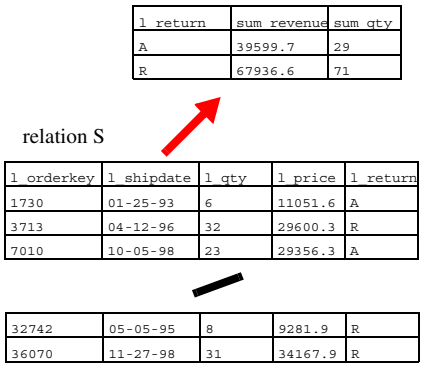


Figure 2: Illustration of an aggregation operation in a database system. A single relation is processed and values in the requested columns are summed together. Other operations include min, max, count, and average. Records with the same value for the l_return column are combined into a single sum.

```
select sum(l_quantity),
sum(l_price*(1-l_discount))
from lineitem group by l_return
```

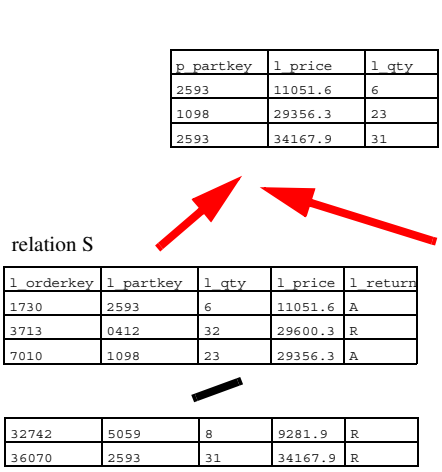


Figure 3: Illustration of a join operation in a database system. Two relations are processed and combined based on a join attribute.

An alternative method for doing aggregation is via hashing [Graefe95], since sorted order isn't strictly required to aggregate groups of records. It is only necessary to combine records with the same key values, not completely sort the relation.

We have chosen to use sorting with a replacement selection algorithm as the on-drive primitive for projection and aggregation. Each drive operates on its local data and combines the records into a local sum, then all the local results are returned to the host for final aggregation. We choose replacement selection because it has good adaptive behavior in the presence of changing memory conditions, allows straightforward implementation of record aggregation and only requires memory proportional to the output of the aggregation or projection, not the input relation [Nyberg94]. A solution using hashing would have similar benefits given the appropriate choice of hash functions to match the skew within a particular set of data.

3.3. Joins

Selective joins will benefit significantly from a reduction in data transfer by operating directly at the drives, and from the offloading of the host processor in doing table lookups. The difficulty with join is that it involves two relations, which may require both significant communication among disks, and large amounts of memory at the individual disks.

3.3.1. Join Algorithms

There are a number of algorithms for performing joins, depending on the absolute size, the relative size, and the existing sort order of the relations being joined. The purpose of a join is to combine two 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. The relation R is, by definition, the smaller of the two relations. It is possible to perform *n-way* joins among a larger number of relations, and these are done as a series of 2-way joins. The choice of join order greatly affects the overall performance and is a major focus of query optimization research [Selinger79]. A two-way join is shown in Figure 2 for a simplified portion of Query 5 from TPC-D.

Join is defined as the cartesian product of two relations followed by a selection on the *join attribute* shared by the two relations. There are a number of algorithms for executing a join, and we will briefly describe them to motivate the one we choose for Active Disks.

Nested-Loops is the most straightforward algorithm, but is efficient only for small R. The name itself explains the basic algorithm, which proceeds as two nested loops, choosing a block of records from S and looping through R looking for matches, then choosing the next block from S and repeating the process. The advantage of this algorithm is that it requires very little memory, essentially only buffering for the blocks of tuples currently being compared.

Merge-Join takes advantage of the fact that the input relations are already sorted on the join attribute and performs the join by simply merging the two lists of records. It does not require repeated passes across R as in Nested-Loops because the records are known to be sorted, so the join can proceed to match records in order, as in a Merge Sort. It also have the memory advantage of Nested-Loops because only the tuples currently being examined need to be in memory. When only one of the relations is sorted, the optimizer must decide whether it is less expensive to sort the second relation and perform a Merge-Join, or revert to Hash-Join as if both were unsorted.

Hash-Join [Kitsuregawa83] uses a hash table of R and has been shown to be the best algorithm choice except in the case of already sorted relations [Schneider89, Schneider90]. Hash-Join builds a hash table of R in memory and then reads S sequentially, hashing the join key for each record, probing the table for a match with R, and outputting a joined record when a match is found. This is more effective than Nested-Loops as each of the relation needs to be read only once. The amount of memory required will be proportional to the size of R, with some overhead for the hash table structures.

Hybrid Hash-Join is an extension of Hash-Join [DeWitt84] that can be used when the inner relation is larger than the amount of memory available. Whereas Hash-Join requires sufficient memory for the entire hash table of R, the Hybrid algorithm can adapt to changing memory conditions. It can use additional memory pages that become available during its processing or give up memory pages when they are needed elsewhere, at the cost of additional disk accesses to store intermediate results and manage additional passes over the relations [Zeller90, Pang93].

3.3.2. Semi-Join

Another way to address the problem of an R that is larger than the available memory is by performing a *semi-join* [Bernstein81]. In a semi-join, that amount of memory required is reduced by not retaining an entire copy of R at each disk, but only the join keys necessary for determining whether a particular record from S should be included in the result. This allows us to achieve the full selectivity savings of the join, without requiring memory for all of R.

The join is performed in two phases. In the first phase, the keys of R are used to filter S and extract only the records that match the keys of R (producing S', which contains only records that will match records from R). The records from S' are then used to probe the R hash table as in hash-join, but with each record guaranteed to find a match in R. The algorithm requires much less memory than a full join in the first phase, because only the keys of R must be kept in memory.

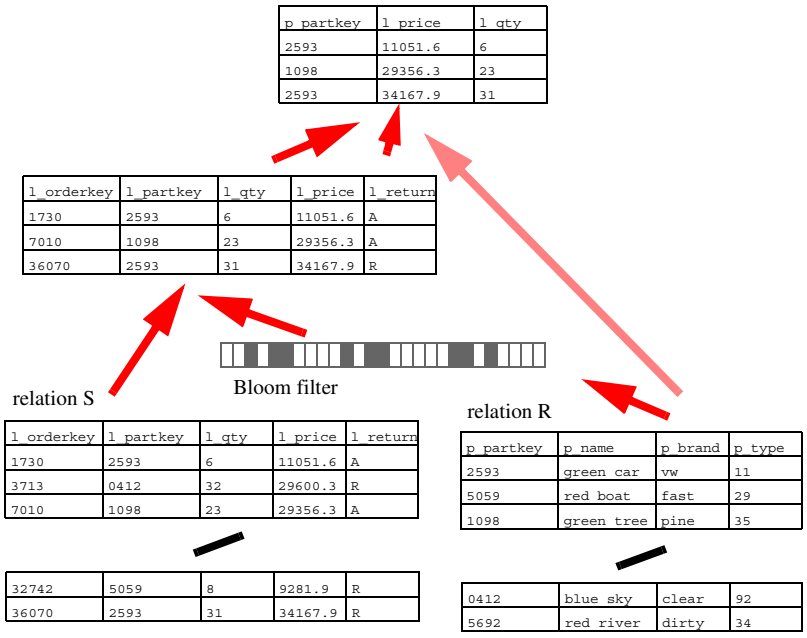


Figure 4: Illustration of the Bloom Join algorithm. Keys from the inner relation (R) are used to form a hash-based bit-vector that is broadcast to all the disks. The disks use this bit vector to filter records from the outer relation (S) before records are returned to the host. The filter will return some number of "false positives" since the bit vector cannot represent all the keys exactly (multiple keys may hash to the same bit position), but it will always return all the necessary records from S. This provides most of the selectivity benefit of a highly selective join, while requiring only constant memory at the drives - the size of the Bloom filter can be chosen based on the memory available, rather than requiring memory proportional to the size of the relations in a particular query.

3.3.3. Bloom Join

If even the keys of R necessary for a semi-join exceed the available memory capacity, a Bloom Join [Mackert86] can be used in place of the semi-join. This algorithm uses a hash-based bit vector built up from the join keys of R to eliminate tuples from the outer relation before they are sent to the host, as illustrated in Figure 4. The goal, as with semi-join, is to exclude tuples from S that will not find a match in R and therefore will not contribute to the final result.

Instead of storing all the distinct values of the join attribute from R, we create a bit vector b[*1*...*n*], initially set to all '0's. For each value of the join attribute in R, we hash it to a value in the range *1* to *n* and set the corresponding bit to '1'. We then use this bit vector when processing tuples from S. If we apply the same hash function to the join attribute in S, then any tuple for which the bit is set to '0' can be excluded from the result, since it will not match any tuples from R. This will allow some number of "false positives" from S to be sent back to the host, but it will give us the selectivity benefits of semi-join while using only constant memory. The memory required for a Bloom join at the drives is independent of the size of the relations and can get significant selectivity benefits with only a small amount of memory, as shown in Table 3 for a number of the TPC-D queries. Since this algorithm uses constant memory and achieves much of the selectivity benefit of semi-join when megabyte-size filters are used, we choose this as the preferred method for joins on Active Disks.

query	join	size of Bloom filter					
		128 bits	1 kilobyte	8 kilobytes	64 kilobytes	1 megabyte	ideal
Q3	1.1	1.0	1.8	3.0	3.0	3.0	4.8
Q5	4.1	1.1	4.5	4.5	4.5	4.5	4.5
Q9	1.1	1.0	9.1	9.1	9.1	9.1	20.0
Q10	2.1			3.0	4.8	4.8	12.5

Table 3: Sizes and selectivities of joins using Bloom filters of varying size. Note that these measurements are based on a particular choice of execution plan, the sizes required for the different joins would be different if the join order were changed.

4. Database Structure

This section describes the modifications required for a database system to support Active Disks. We have chosen to modify PostgreSQL version 6.5, an open-source database system based on work originally done at Berkeley, for use with our prototype Active Disks. We use this system due to the ready availability of the code, its use in past published research, and the difficulty of collaborating with a commercial vendor, but we feel the nature of the changes should be applicable to other systems as well.

The common structure for the processing done at the Active Disks is to push down qualification, aggregation, and semijoin operations into what would be a parallel table scan in a modern database system. The Active Disk operators act as filters on the data as it is read from the disks, reducing the amount of data returned and offloading repetitive per-record computations from the host. Only filtered records are returned to the host for final processing and output.

The downside of doing this type of filtering at the drives is that full records are not returned to the host for later re-use. In the case of small relations, this decrease in cache efficiency could well outweigh the benefits of Active Disk processing. This problem can be addressed by having the query optimizer choose plans where small relations (for which caching will be beneficial) are returned to the host in their entirety, while large relations (which would normally bypass the buffer pool anyway) are processed at the disks and not cached at the host. This also avoids the cache coherence problems that arise if Active Disks are allowed to process pages that may be dirty in the host's buffer pool.

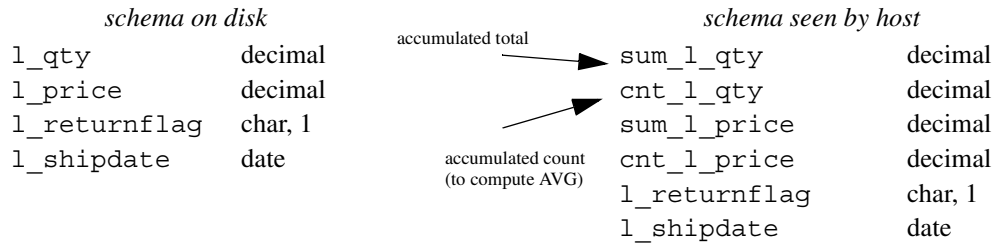
There are some additional locking and concurrency issues that must be addressed in the presence of updates. We feel these are important considerations in the design of an Active Disk database system, but for the prototype described here, we assume that relations being processed at the disks are marked uncacheable at the hosts, and that cursor stability [Gray92] is sufficient for the decision support queries we evaluated.

4.1. Host Modifications

The code running at the host is a version of the PostgreSQL system with two sets of modifications, as detailed in Table 4. Changes were made in the storage layer to provide striping and use a network storage interface for disk access, rather than a traditional filesystem, and in the *scan* function to provide a way to ship filter conditions to the drives and start Active Disk processing.

A "bypass" point is added to each of the execute nodes where parameters for scans, aggregations, and semijoins are passed to the disks. For scans and semi-joins, the work at the drives simply reduces the number of records that are processed at the host. For aggregation, the schema as

seen by the host is changed to accommodate the computation already done at the drives, as illustrated below:



for a portion of the schema used in Query 1.

4.2. Query Optimization

The query optimizer must be modified to take into account the presence of a particular number of Active Disks, as well as their memory sizes and processing rates. The PostgreSQL optimizer already maintains information about the costs of each of the basic execute nodes, including all the possible join algorithms, selections, aggregations, and index scans. It also maintains a set of statistics for each table in the database, tracking the data type, minimum and maximum values, and an estimate of the disbursion of the values currently in the database for that attribute. Knowledge of the data types, the range of values, and the disbursion allows the optimizer to estimate the size of outputs and intermediate results at each stage of the query, and estimate the resources required and relative cost of different possible execution plans, including whether the use of Active Disk processing is appropriate for a particular stage of the query.

4.3. Active Disk Code

The PostgreSQL code required at the drives is also shown in Table 4. The on-drive code must understand the layout of pages on disk, the schemas of the database, and tuple layout formats. The

module	**original**		**modified host** (new & changed)		**Active Disk**	
	Files	Code	Files	Code	Files	Code
access	72	26,385	-	-	1	838
bootstrap	2	1,259	-	-	-	-
catalog	43	13,584	-	-	-	-
commands	34	11,635	-	-	-	-
executor	49	17,401	9	938	4	3,574
parser	31	9,477	-	-	-	-
lib	35	7,794	-	-	-	-
nodes	24	13,092	-	-	6	4,130
optimizer	72	19,187	2	620	-	-
port	5	514	-	-	-	-
regex	12	4,665	-	-	-	-
rewrite	13	5,462	-	-	-	-
storage	50	17,088	1	273	-	-
tcop	11	4,054	-	-	-	-
utils/adt	40	31,526	-	-	2	315
utils/fmgr	4	2,417	-	-	1	281
utils	81	19,908	-	-	1	47
Total	578	205,448	12	1,831	15	9,185
					New	1,257

Table 4: Code changes required to adapt PostgreSQL to Active Disks. The table shows the major modules of PostgreSQL, the changes required to the code at the host, and the amount of code that runs at the drives for each module. Note that only the ADT functions necessary to support the TPC-D queries are included in the drive code - to support the range of "basic" database type, another 5,000 lines of code would be added have to be added to the on-disk code.

query	tbls	lineitem		order		quals		joins		semijoin	aggregation		time (s)
Q1	1	scan	2.3	-	-	1x cond	1.1	-	-	-	8x SUM/AVG/COUNT, 6x arith	6104.0	4,357.1
Q3	3	index	-	scan	3.8	1x cond	2.1	HJ, NL	5.7	1.9	1x SUM, 2x arith	81.0	754.8
Q4	2	index	-	scan	2.9	2x cond	30.5	-	-	-	1x COUNT	41.0	625.4
Q5	6	scan	3.6	scan	3.8	-	-	4x HJ, NL	490.7	736.0	1x SUM, 2x arith	9.0	1,988.2
Q6	1	scan	3.2	-	-	5x cond	68.6	-	-	-	1x SUM, 1x arith	73.0	63.1
Q9	6	scan	2.8	scan	4.5	-	-	6x HJ	13.8	24.9	1x SUM, 4x arith	75.0	2710.8
Q10	4	scan	3.2	scan	4.5	1x cond	4.6	3x HJ	1.0	1.0	1x SUM, 2x arith	1088.0	810.1
Q13	2	index	-	scan	2.9	1x cond	1250.0	NL	-	-	1x SUM	1.0	5.9
Q17	2	scan	3.6	-	-	1x cond	-	HJ	1051.8	255.5	2x SUM/AVG	5.8	166.0
total											all 17 queries		15,264.0

Table 5: Details of selected TPC-D queries. The queries listed cover all query types and represent over 75% of the execution time of the 17 read-only queries in the benchmark. The access method for the two largest tables in the database is given, as well as the selectivity savings of performing the scan, qualification, join, and aggregation steps at the disks. The selectivity savings are not cumulative, so the overall savings for Q1 is over 15,000 if the scan, qualification, and aggregation are *all* done at the disks. The second column of join selectivity gives the savings if only a semi-join with a 1-megabyte Bloom filter is used at the drives. The final column gives the performance of each query on the Digital 8400 system listed in Table 1 [TPC98]. Savings and query plans were determined by PostgreSQL running on the validation data set, but the query plan details are consistent with previous studies, including those of commercial database systems [Barroso98, Tamaru99].

code also includes operators for dealing with basic calculations on database fields for all the data types that the drive supports. This includes comparison operators for scan and sorting and arithmetic operators for aggregation. In addition, the code contains the three core primitives adapted for use within the constraints of the drive environment. The preceding section discussed the algorithms that are used to implement each of the database operations. Select is performed in the straightforward manner; aggregation, projection and sorting are performed using a single replacement selection sort primitive that allows combining of records when performing aggregation or duplicate elimination; and join is performed by forming a fixed-size Bloom filter at the host, using this to perform a semi-join at each of the drives, and completing the final join at the host. These three primitives are sufficient to implement all the data-intensive portions of the query processing.

4.4. Benchmark Evaluation

Table 6 shows the details of a number of the queries from the TPC-D benchmark and shows the selectivity savings possible if the various processing steps are performed at the disks. Simply using the disks for the scan primitive to eliminate table columns that are not needed in the query reduces data by a factor of three on average. Including the qualification conditions increases the savings to an aggregate of between 2.5 times and over 100 times reduction, depending on the query. Processing large joins gives double-digit gains and near 500 times for Q5. Semijoin, even with fixed-size Bloom filters, gives reductions similar to doing the full joins, while using only constant memory. The reduction for a semijoin may be less than for a join due to the false positives possible with Bloom filters. The semijoin reduction may be more due to the expansion of the cartesian product in the full join. Finally, it is clear that including aggregation at the drives gives the largest benefits, although in many of the queries, the absolute savings at this point are only several dozen pages.

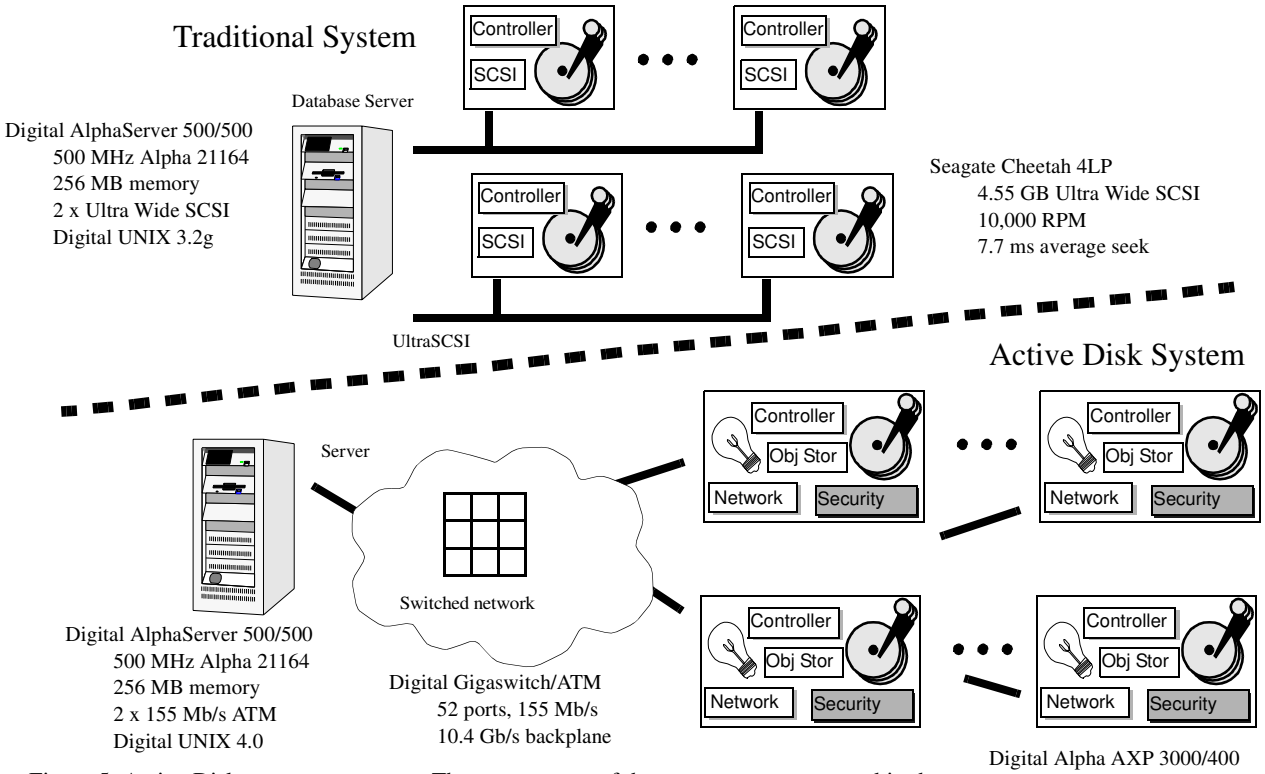


Figure 5: Active Disk prototype systems. The components of the two systems compared in the prototype numbers to follow. The diagram shows a traditional server system with directly-attached SCSI disks. The lower picture shows an Active Disk system using network-attached disks.

5. Experiments

The testbed used for all of our experiments consists of ten prototype Active Disks, each one a six-year-old Digital Alpha 3000/400 (133 MHz, 64 MB, Digital UNIX 3.2g) with two 2.0 GB Seagate ST52160 Medalist disks. A single Digital AlphaStation 500/500 (500 MHz, 256 MB, Digital UNIX 3.2g) with four 4.5 GB Seagate Cheetah disks on two Ultra-Wide SCSI busses is used for the server case. All these machines are connected by an Ethernet switch and a 155 Mb/s OC-3 ATM switch. This setup is illustrated in Figure 5 showing the details of both systems.

All of our experiments compare the performance of the single server machine with directly-attached SCSI disks against the same machine with network-attached Active Disks, each of which is a workstation with two directly-attached SCSI disks[1].

1. the need for two physical disks on each single Active Disk is an artifact of using old workstations not explicitly designed for this purpose. The 3000/400 contains two narrow SCSI busses, with a maximum bandwidth of 5 MB/s each. The Seagate Medalist disks are capable of 7 MB/s each, but the use of the narrow SCSI busses limits sequential throughput to a total of 6.5 MB/s when two are used in combination. For all the queries presented here, the use of two disks instead of a single, faster disk does not impact the system performance and will be pessimistic to the Active Disk case, since the server disks are each capable of a much higher 11 MB/s.

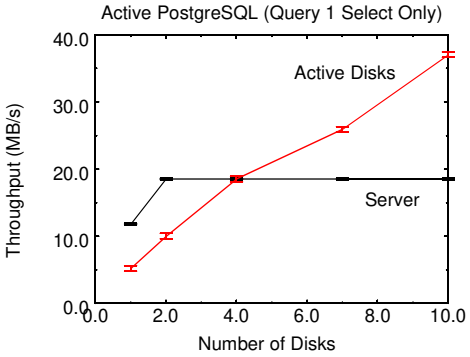


Figure 6: Performance of PostgreSQL select. The PostgreSQL select operation shows linear scaling with number of disks up to 37 MB/s with 10 disks, while the server system bottlenecks at 18 MB/s.

5.1. Database - Select (Query 6)

Figure 6 compares the performance of a database server with traditional disks against a server with an equivalent number of Active Disks for a simple select query. The query is:

```
select * from lineitem where l_shipdate > '1998-09-02'
```

using tables and test data from the TPC-D benchmark. The records in the database cover dates from 1992 through the end of 1998, so this query returns about 4% of the total records in the `lineitem` table. This query performs the qualification at the disks and returns a record to the host only if the condition matches. This is very similar to the processing of Q6 in TPC-D. We used a modified Q1 because it does not benefit from the declustering (on `l_shipdate`) that most TPC-D benchmark systems do precisely to speed up Q6. The implications of physical design for Active Disk systems is further discussed in Section 5.5.

The server performs better than the Active Disk system for small numbers of disks, since each individual disk is much less powerful than the 500 MHz host. Once the aggregate compute power of the disks passes that of the host, the Active Disk system continues to scale while the server performance remains flat, no matter how much aggregate disk bandwidth is available. Notice that the performance increase in the Active Disk system is somewhat less than linear. This is due to the sequential overhead of performing the query - primarily the startup overhead of initiating the query and beginning the Active Disk processing. This overhead is amortized over the entire size of the table processed. For the experiments in the chart, the table is only 125 MB in size, so the overhead is significant and noticeable in the results. A real TPC-D system sized for a 300 GB benchmark, would have a `lineitem` table of over 100 GB and this startup overhead would be negligible.

5.2. Database - Aggregation (Query 1)

Figure 7 compares the performance of the database server against a server with Active Disks for a simple aggregation query. The query being performed is:

```
select l_returnflag, l_linestatus,
sum(l_qty), sum(l_price*(1-l_disc))
```

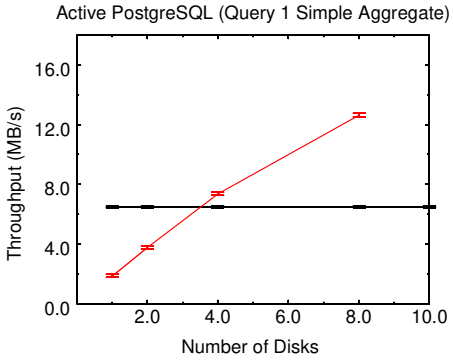


Figure 7: Performance of PostgreSQL aggregation. The PostgreSQL aggregation shows linear scaling with the number of Active Disks and reaches 13 MB/s with eight disks, while the server bottlenecks on the CPU at 6.5 MB/s.

```
from lineitem where l_shipdate <= '1998-09-02'
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus
```

which is a simplified version of Q1. This query summarizes about 95% of the records in the `lineitem` table. The prototype performs both the qualification and aggregation at the disks and returns per-disk summaries that are then combined into a single set of results at the host.

The computation required for aggregation is significantly more than for the select. The same comparison as in the select is performed to identify records that match the qualification condition. Matching records are then inserted into a sorted heap using replacement selection, combined based on the `group by` keys and aggregated into the sums and averages specified by the query.

5.3. Database - Join (Query 9)

Figure 8 compares the performance of a database server against a server with Active Disks for a simple two-way join. The query being performed is:

```
select sum(l_quantity), count(*) from part, lineitem
where p_partkey = l_partkey and p_name like '%green%'
group by n_name, t_year order by n_name, t_year desc
```

which is a simplified version of Q9. The records in the database cover 1,000 unique items, so this query matches about 10% of the unique part numbers in the database. The prototype performs a

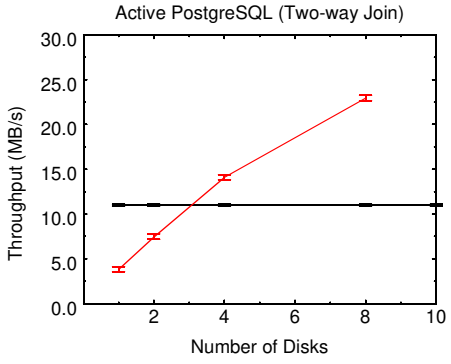


Figure 8:'Performance of a 2-way PostgreSQL join. The PostgreSQL join scales nearly linearly to 24 MB/s with Active Disks, and is limited to 11 MB/s in the server system.

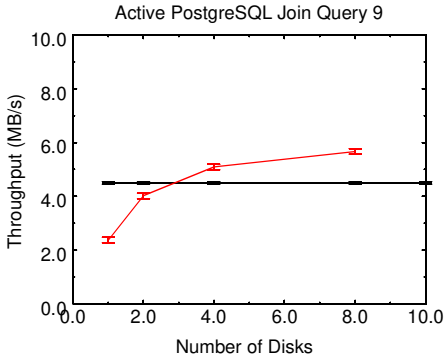


Figure 9: Performance of a 5-way PostgreSQL join. This query has a significantly higher serial fraction than the previous applications, so the scaling with Active Disks drops off relatively early. The performance improvement is about 11% with eight disks.

semijoin at the disks and returns a record to the host only if the join key matches the Bloom filter created from the `part` table.

The results in Figure 9 show the performance of a more complex join, executing the full 5-way join given by Q9. The query being performed is:

```
select n_name, t_year,
sum(l_price*(1-l_disc)-ps_supplycost*l_quantity)
from part, supplier, lineitem, partsupp, order, nation, time
where s_suppkey = l_suppkey
and ps_suppkey = l_suppkey and ps_partkey = l_partkey
and p_partkey = l_partkey
and o_orderkey = l_orderkey
and t_alpha = o_orderdate
and s_nationkey = n_nationkey
and p_name like '%green%'
group by n_name, t_year order by n_name, t_year desc
```

again using tables and data from the validation data set.

This query has a much higher serial fraction than the two-way join, and shows the performance limitation in the Active Disk much sooner than the simple join. The serial fraction of this entire query is more than 80%, so the maximum speedup possible with Active Disks is a factor of 1.25x, even with perfect parallel scaling. The prototype results show an 11% improvement in performance with eight disks.

5.4. Database - Summary

Table 6 summarizes the results of the last several sections and compares the performance of the server system and the Active Disk prototype on several of the most expensive queries from the TPC-D benchmark. We see that the scan-intensive applications, including the 2-way join, show linear scalability, while the more complex joins have significantly higher serial overheads, but still show significant speedups with Active Disk processing. These results are with a small prototype system of only eight disks, which is much smaller than the system that are built in practice for this type of workload. We would expect proportionally higher performance for a system with

query	type	input (MB)	output (KB)	disks	host (s)	tput (MB/s)	Active Disk (s)	tput (MB/s)			parallel fraction
Q1	aggr	494	0.2	8	76.0	6.5	38.0	12.6	100%	cpu-bound	0.99
Q5	join (6)	494	0.1	8	219.0	2.2	186.5	2.6	17%	cpu-bound	0.28
Q6	select	494	5057	8	27.2	18.8	17.0	29.0	60%	disk-bound	0.99
Q9s	join (2)	494	0.5	8	44.5	11.1	21.6	22.9	108%	cpu-bound	0.99
Q9	join (6)	494	0.5	8	95.0	5.2	85.4	5.8	11%	cpu-bound	0.19

Table 6: Summary of TPC-D results using PostgreSQL. A portion of the TPC-D benchmark running on the PostgreSQL database system modified to use Active Disks.

60 disks, such as the one shown in Table 1, as well as for the multi-processor systems that have higher host processing rates, but also considerably more disks.

5.5. Physical Design

The choice of database layout across the disks in a large storage system is critical to the performance of the queries, and much care is taken by designers of TPC benchmark systems to organize data on disks in a way that is advantageous to the largest number of queries. According to the benchmark rules, the data layout cannot take direct advantage of knowledge of what queries will be executed, as the intent is to model *ad-hoc* queries that are not known beforehand. In practice, database implementors must choose one attribute for declustering the `lineitem` table, and `l_shipdate` is the most advantageous. This means, however, that ad-hoc queries that happened to use the `l_receiptdate` or `l_commitdate` will perform poorly. The use of Active Disks simplifies this problem by providing much greater resources for such parallel scans. The same declustering can also be performed in an Active Disk system, speeding queries based on `l_shipdate`, but queries on other columns will perform much better than they would in a traditional system, easing the sensitivity to the one-time choice of physical layout.

In order to make most effective use of the Active Disk processing power, it will be beneficial to spread particular tables across the largest number of disks possible, which adds an additional consideration to physical design in the presence of Active Disks.

Our prototype has not considered the use of indices in processing Active Disk queries. When an index scan is chosen by the optimizer, Active Disk processing is not used. There may be some benefits from also performing these indexed scans in parallel, or maintaining parallel indices across the disks, but we have not yet explored this direction.

5.6. Computation Requirements

The application characteristics of each of the database operations for the prototype platform are shown in Table 7. We see that the select is the least expensive, using less than four instructions per byte of data processed. It also uses very little memory since only enough memory to evaluate one page of tuples at a time is required.

The computation required for aggregation is significantly more than for the select. The same comparison as in the select is performed to identify records that match the qualification condition. Matching records are then sorted and combined using the `group by` keys and aggregated into

application	computation (instr/byte)	computation (cycles/byte)	throughput (MB/s)	memory (KB)	selectivity (factor)	code (KB)
database select	3.8	6.5	19.5	88	52.0	20.5 (13.3)
database aggregation	15.0	31.1		120	31.9	26.7 (18.4)
database join	3.4	6.2	20.0	88	4.3	19.8 (14.4)

Table 7: Costs of the database operations. Computation requirement, memory required, and the selectivity factor in the network. The computation requirement is shown in both instructions per byte and cycles per byte. The last column also gives the total size of the code executed at the drives (and the total size of the code that is executed more than once).

the sums and averages specified by the query. Each disk returns the aggregation values for its portion of the relation, and these results are then combined at the host.

The join must hash each of the join attribute values and apply the Bloom filter to determine whether a particular record will contribute to the result, returning only matching records. The records from all the disks are combined at the host and joined using a complete hash table.

5.7. "Beta" Prototype

Table 8 compares the prototype discussed so far based on 6-year-old workstations with a new engineering prototype developed for us that has a form factor and processing capability much closer to what we would expect first-generation Active Disks to have. We see that the basic processing rates are very comparable for the two devices. The "beta" device has a significantly faster processor and disk, giving it an advantage over the workstation prototype, but does have a significantly higher cycle/byte cost for the basic computations. These "beta" devices have only recently been made available to us, and we are continuing to experiment with them to further characterize the performance of the PostgreSQL code in this embedded system version of Active Disks.

	"alpha" prototype	"beta" prototype
processor	133 MHz Alpha	200 MHz StrongARM
memory	64 MB	32 MB
storage	4.2 GB	13 GB
disk bandwidth	6.5 MB/s	11 MB/s
network bandwidth	10.5 MB/s	9.5 MB/s
operating system	Digital UNIX	Linux
select	5.5 MB/s	6.8 MB/s
(cycles/byte)	6.5	12.6
aggregation	1.8 MB/s	1.6 MB/s
(cycles/byte)	31.1	52.0
join	4.3 MB/s	7.3 MB/s
(cycles/byte)	6.2	11.0

Table 8: Comparison of our "alpha" and "beta" prototypes. The table shows the characteristics of two generations of Active Disk prototypes. The "alpha" prototype is a Digital AXP 3000/400 workstation with two Seagate Medalist disk drives, as described earlier in the paper. The "beta" prototype is an engineering prototype using an embedded processor and a single Quantum Viking disk drive. We see that the processing rates for select and join are higher in the "beta" prototype due to the higher disk bandwidth, while the aggregation is slightly slower due to the higher cycle/byte cost of the computation, which is only partially overcome by the faster processor.

6. 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 [Su79], RAP [Ozkarahan75], and numerous others [DeWitt81]. These machines fell out of favor 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. Extensive work on parallel database

systems has explored all of the basic database algorithms discussed and their mapping to parallel systems that serve as the basis for choosing the proper algorithms that fit the characteristics of Active Disk systems. An excellent survey of many years of this work can be found in [DeWitt92].

Previous work on Active Disk systems has explored the benefits of mapping applications from data mining, image processing, sorting, and data cubes onto an architecture with computation at storage nodes [Acharya98, Riedel98]. Work on Intelligent Disks explored the use of processing on storage devices to offload portions of an SMP database system using an analytic model for speedup [Keeton98]. Simulations of the use of Active Disks for decision support have shown that such systems can outperform traditional SMP servers, and provide performance similar to clusters of workstations [Uysal00]. The work discussed here extends this work and provides measurements from a running prototype implementation to validate that the benefits promised in these papers are realizable in a realistic database system with relatively small modifications.

Work in the SmartSTOR project at Berkeley and IBM explored the use of a similar architecture for decision support queries and concluded that the processor available on an individual disk drive would be insufficient for handling the computation rates required, and that an additional processor board supporting several disks should be used instead [Hsu99]. This study also concluded that the increased use of pre-computed aggregates would reduce the need for parallel scans of the data, as many queries would be answered directly from these aggregates. This trend away from the intended ad-hoc nature of the TPC-D benchmark has led the Transaction Processing Performance Council to obsolete TPC-D and create two new benchmarks: TPC-R for reporting and TPC-H for ad-hoc querying, where the first allows the use of pre-aggregates, and the second does not [TPC99]. The benefits discussed in this paper will continue to be more applicable to queries of the ad-hoc nature, as embodied by TPC-H.

Implementation work at the University of Tokyo has demonstrated performance results similar to those presented here by mapping a database system onto the nodes of a parallel cluster of commodity PCs. They also introduce a transposed file system organization for the columns in their database that allows them to obtain the scan benefits discussed here without on-disk support for selectively returning columns of individual records to the host [Tamura99].

Work at Digital, Compaq, and Rice University considered the memory and processor performance of large decision support systems, among other workloads [Barroso98, Ranganathan98]. The authors conclude that I/O performance is no longer the primary determinant of large-system performance, as improvements in disk arrays and software structures allow sufficient disk bandwidth to be added to a system until the processors are the bottleneck. Our work on Active Disks takes advantage of these same trends toward systems with large numbers of individual disks to provide more efficient processing in these systems. The intent of Active Disks is not to replace large-memory multi-processors, but to boost the performance of these systems by taking advantage of resources that already exist inside the I/O sub-system and are not being taken advantage of due to limited interfaces.

7. Conclusions and Future Work

We have shown that all the basic database functions can be efficiently mapped onto an Active Disk system. The changes necessary to adapt an existing relational database system for use with Active Disks required modifying about 2% of the code at the host, and running about 10% of the total database system code at the drives. We have demonstrated a speedup of up to 2x using a prototype system with eight Active Disks compared to a server system with traditional disks. We expect these improvements to scale well to the systems with dozens and hundreds of disks that are commonly used today.

We have also provided preliminary results for an engineering prototype that is much closer to the form factor and processing capability we would expect commercially available Active Disks to have. Initial results are promising that the results obtained in our "alpha" prototype will transfer to the "beta" prototype, and we are actively developing a testbed including fifty of the "beta" devices for larger scalability studies.

The intent of this work is to provide a proof of concept implementation to illustrate the benefits possible with Active Disks. There are a number of issues in concurrency control and fault tolerance that we have not discussed here, including how to most efficiently handle updates in this highly distributed system. These items, as well as further exploration of the appropriate programming models for Active Disk functions are active parts of our future work.

8. Acknowledgements

The authors wish to thank David Rochberg for help with the "beta" prototype under extreme deadline pressure, Khalil Amiri and Howard Gobioff for many valuable discussions, and all the other members of the Parallel Data Lab for their invaluable support in so many ways. We also thank our industry partners from the Parallel Data Consortium and the participants in the various industry meetings that inspired and supported this work.

9. References

[Acharya98] Acharya, A., Uysal, M. and Saltz, J. "Active Disks" *ASPLOS*, October 1998.

[Barroso98] Barroso, L.A., Gharachorloo, K. and Bugnion, E. "Memory System Characterization of Commercial Workloads" *ISCA*, June 1998.

[Bernstein81] Bernstein, P.A. and Goodman, N. "Power of Natural Semijoins" *SIAM Journal on Computing* 10 (4), 1981.

[Boral83] Boral, H. and DeWitt, D.J. "Database Machines: An Idea Whose Time Has Passed?" *International Workshop on Database Machines*, September 1983.

[Cirrus98] Cirrus Logic, Inc. "New Open-Processor Platform Enables Cost-Effective, System-on-a-chip Solutions for Hard Disk Drives" *www.cirrus.com/3ci*, June 1998.

[Compaq00] Compaq Corporation "QuickSpecs: Compaq SCSI Hard Drive Option Kits (Servers)" *Data Sheet*, January 2000.

[DeWitt81] DeWitt, D.J. and Hawthorn, P.B. "A Performance Evaluation of Database Machine Architectures" *VLDB*, September 1981.

[DeWitt92] DeWitt, D.J. and Gray, J. "Parallel Database Systems: The Future of High Performance Database Processing" *Communications of the ACM* 36 (6), June 1992.

[Graefe95] Graefe, G. and Cole, R.L. "Fast Algorithms for Universal Quantification in Large Databases" *ACM Transactions on Database Systems* 20 (2), June 1995.

[Gray92] Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, September 1992.

[Hsu99] Hsu, W.W., Smith, A.J. and Young, H.C. "Projecting the Performance of Decision Support Workloads on Systems with Smart Storage (SmartSTOR)" *Technical Report CSD-99-1057*, University of California - Berkeley, August 1999.

[Keeton98] Keeton, K., Patterson, D.A. and Hellerstein, J.M. "A Case for Intelligent Disks (IDISKs)" *SIGMOD Record* 27 (3), August 1998.

[Kitsuregawa83] Kitsuregawa, M., Tanaka, H. and Moto-Oka, T. "Application of Hash To Data Base Machine and Its Architecture" *New Generation Computing* 1, 1983.

[Levin99] Levin, R. "Java Technology Comes of Age" *News Feature, java.sun.com*, May 1999.

[Mackert86] Mackert, L.F. and Lohman, G.M. "R* Optimizer Validation and Performance Evaluation for Distributed Queries" *VLDB*, 1986.

[Martin94] Martin, M.W. "The ICL Search Accelerator, SCAFS: Functionality and Benefits" *ICL Systems Journal* 9 (2), November 1994.

[Nyberg94] Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J. and Lomet, D. "AlphaSort: A RISC Machine Sort" *SIGMOD*, May 1994.

[Ozkarahan75] Ozkarahan, E.A., Schuster, S.A. and Smith, K.C. "RAP - An Associative Processor for Data Base Management" *Proceedings of AFIPS NCC* 44, 1975.

[Pang93] Pang, H., Carey, M.J. and Livny, M. "Partially Preemptible Hash Joins" *SIGMOD*, May 1993.

[Quantum99] Quantum Corporation "Snap Server 1000 is the Industry's First Sub $500 Server" *News Release*, November 1999.

[Ranganathan98] Ranganathan, P., Gharachorloo, K., Adve, S.V. and Barroso, L.A. "Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors" *ASPLOS*, October 1998.

[Riedel98] Riedel, E., Gibson, G. and Faloutsos, C. "Active Storage For Large-Scale Data Mining and Multimedia" *VLDB*, August 1998.

[Schneider89] Schneider, D.A. and DeWitt, D.J. "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment" *SIGMOD*, 1989.

[Schneider90] Schneider, D.A. and DeWitt, D.J. "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines" *VLDB*, 1990.

[Selinger79] Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A. and Price, T.G. "Access Path Selection in a Relational Database Management System" *SIGMOD*, May 1979.

[Siemens98] Siemens Microelectronics, Inc. "Siemens Announced Availability of TriCore-1 For New Embedded System Designs" *News Release*, March 1998.

[Smith79] Smith, D.C.P. and Smith, J.M. "Relational DataBase Machines" *IEEE Computer*, March 1979.

[Tamura99] Tamura, T., Oguchi, M. and Kitsuregawa, M. "High Performance Parallel Query Processing on a 100 Node ATM Connected PC Cluster" *IEICE Transactions on Information and Systems* E82-D (1), January 1999.

[TPC97] TPC-C Rev. 3.3 Rating for a Digital AlphaServer 1000A 5/500, Transaction Processing Performance Council, *www.tpc.org,* April 1997.

[TPC98] TPC-D Rev. 1.3.1 Rating for a Digital AlphaServer 8400 5/625 12 CPUs using Oracle8, Transaction Processing Performance Council, *www.tpc.org,* May 1998.

[TPC98a] Transaction Processing Performance Council, "TPC Benchmark D (Decision Support) Standard Specification 1.3.1", *www.tpc.org*, February 1998.

[TPC99] Transaction Processing Performance Council, "TPC Benchmark H (Decision Support) Standard Specification 1.2.1", *www.tpc.org*, June 1999.

[TPC00] TPC-H Rev. 1.2.1 Rating for a Compaq Proliant 8000 with Microsoft SQL Server 2000, Transaction Processing Performance Council, *www.tpc.org*, April 2000.

[Turley96] Turley, J. "ARM Grabs Embedded Speed Lead" *Microprocessor Reports* 2 (10), February 1996.

[Uysal00] Uysal, M., Acharya, A. and Saltz, J. "Evaluation of Active Disks for Decision Support Databases" *International Symposium on High Performance Computer Architecture*, January 2000.

[Zeller90] Zeller, H. and Gray, J. "An Adaptive Hash Join Algorithm for Multiuser Environments" *VLDB*, 1990.