Structure and Performance of Decision Support Algorithms on Active Disks

Mustafa Uysal
Dept. of Computer Science
University of Maryland
College Park

Anurag Acharya
Dept. of Computer Science
University of California
Santa Barbara

Joel Saltz
Dept. of Computer Science
University of Maryland
College Park

Abstract

Growth and usage trends for large decision support databases indicate that there is a need for architectures that scale the processing power as the dataset grows. These trends indicate that the processing demand for large decision support databases is growing faster than the improvement in performance of commodity processors. To meet this need, several researchers have recently proposed Active Disk/IDISK architectures which integrate substantial processing power and memory into disk units. In this paper, we examine the utility of Active Disks for decision support databases. We try to answer the following questions. First, is it possible to restructure algorithms for common decision support tasks to utilize Active Disks? Second, how does the performance of Active Disks compare with that of traditional servers for these tasks? Finally, how would Active Disks be integrated into the software architecture of decision support databases?

1 Introduction

Growth and usage trends for large decision support databases indicate that there is a need for architectures that scale the processing power as the dataset grows. These trends indicate that the processing demand for large decision support databases is growing faster than the improvement in performance of commodity processors. Results from the 1997 and 1998 Winter Very Large Database surveys document the growth of decision support databases [52, 53]. For example, the Sears Roebuck and Co decision support database grew from 1.3 TB in 1997 to 4.6 TB in 1998. The usage trends indicate that there is a change in user expectations regarding large databases – from primarily archival storage to frequent reprocessing in their entirety. Patterson et al [37] quote an observation by Greg Papadopolous - while processors are doubling performance every 18 months, customers are doubling data storage every nine-to-twelve months and would like to "mine" this data overnight to shape their business practices [36].

To meet this need, several researchers have recently proposed Active Disk/IDISK architectures which integrate substantial processing power and memory into disk units [2, 22, 27, 42]. These architectures allow application-specific code to be downloaded and executed on the data that is being read from (written to) disk. To utilize Active Disks, an application is partitioned between a host-resident component and a disk-resident component. The key idea is to offload bulk of the processing to the disk-resident processors and to use the host processor primarily for coordination, scheduling and combination of results from individual disks. Active Disks present a promising architectural direction for two reasons. First, since the number of processors scales with the number of disks, active-disk architectures are better equipped to keep up with the processing requirements for rapidly growing datasets. Second, since the processing components are integrated with the drives, the processing power will evolve as the disk drives evolve. This is similar to the evolution of disk caches – as the drives get faster, the disk caches become larger.

In this paper, we examine the utility of Active Disks for decision support databases. We try to answer the following questions. First, is it possible to restructure algorithms for common decision support tasks to utilize Active Disks? To be able to take advantage of Active Disks, it should be possible to partition these algorithms such that most of the processing can be offloaded to the disk-resident processors. Second, how does the performance of Active Disks compare with that of traditional servers for these algorithms? Finally, how would Active Disks be integrated into the software architecture of decision support databases?

We address these questions in three ways. First, we present Active Disk algorithms for a suite of eight common decision support tasks: select, aggregation, group-by, the *datacube* operation [24], external sort, project-join queries, datamining association rules from retail transaction data and materialized views. We have derived these algorithms from well-known efficient algorithms in the literature [4, 7, 14, 21, 40, 54]. Second, we compare the performance of Active Disks with that of shared memory multiprocessor servers for these tasks. Shared memory multiprocessors are widely used for relational databases (Strenstrom et al [48] estimate that in 2000, 40% of such machines will be sold for handling relational databases). Finally, we show how the stream-based programming model that has been proposed for Active Disks [2] meshes well with the software architecture of relational databases.

2 Background: Active Disks

In this section, we provide a brief introduction to Active Disks. Active Disks integrate significant processing power and memory into a disk drive and allow application-specific code to be downloaded and executed on the data that is being read from (written to) disk. To utilize Active Disks, an application is partitioned between a host-resident component and a disk-resident component. The key idea is to offload bulk of the processing to the disk-resident processors and to use the host processor primarily for coordination, scheduling and combination of results from individual disks. Figure 1 presents a schematic for Active Disk architectures.

Acharya et al [2] propose a stream-based programming model for disk-resident code (a *disklet*) and its

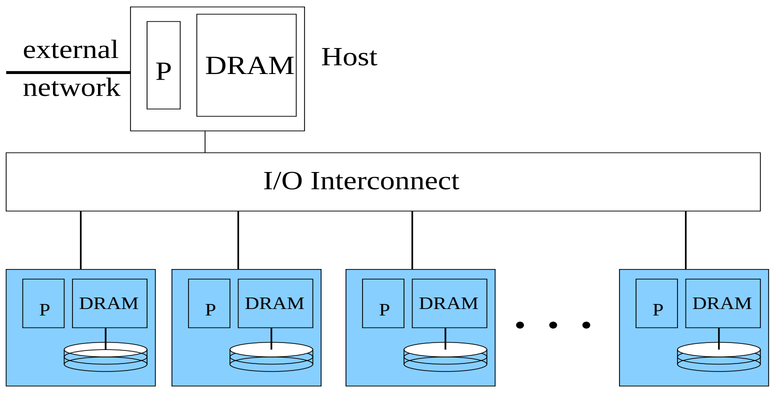


Figure 1: Schematic of Active Disk architectures. The I/O interconnect is Fibre Channel (one or more arbitrated loops – multiple loops are connected via switches). Note that individual disks can communicate directly with each other as well as with the front-end host. All interaction with external clients is handled by the front-end host.

interaction with its peers on other disks as well as on the front-end host. Disklets take streams as inputs and generate streams as outputs. Each disklet must have at least one input stream and at least one output stream. Files (and ranges in files) are represented as streams. Streams are accessed using a standard interface which delivers the data in buffers whose size is known a priori. Each disklet has an initialization function which is run when the disklet is installed and a processing function (`read/write`) which is run as data is read/written. It may, optionally, contain long-term scratch space (which is allocated on its behalf before it is installed and is automatically reclaimed after it exits), a set of parameters that can be used to customize its behavior, and a finalization function which is run when the disklet terminates (either by consuming the data on all its input streams or by calling `exit`).

A disklet is not allowed to initiate I/O operations on its own. All I/O operations are initiated by the host-resident program and are checked for validity by the host-resident file-system. This has two advantages. First, disklets cannot corrupt the file-system. Second, the operating-system layer on the disk need not provide file-system functionality. While a disklet is not allowed to initiate I/O operations, it is allowed to *skip* subranges in an input stream by notifying the operating-system layer on the disk. The skipped subranges are not delivered to the disklet. This allows disklets to safely implement algorithms in which future I/Os depend on data from previous I/Os. For example, an algorithm that uses a disk-resident index to decide which chunks of data are to be read can be implemented by a disklet with two input streams – one corresponding to the index file and the other corresponding to the data file. It uses the data delivered on the index stream to decide which parts of the data stream are to be read and which are to be skipped. An important feature of this model allows multiple streams to be merged. Buffers delivered over the streams being merged are interleaved in arbitrary order and are delivered as coming from a single stream to the destination disklet. This allows disklets with a large number of communicating peers to avoid needless polling on individual streams.

A disklet cannot allocate or free memory. All memory management is done by the operating-system layer on the disk. Furthermore, all memory accesses by a disklet must be within a sandbox defined by the buffers for its input stream(s) and the long-term scratch space (if any). The disklet binary is analyzed at download-time (as in software fault-isolation [50]); disklets that *may* violate memory-safety are rejected. The stream-based programming model simplifies the analysis as it defines a natural sandbox for disklets.

Communication between a disklet and its environment is restricted to its input and output streams. The sources and sinks for these streams are specified by the host-resident program as a part of the installation of the disklet. A disklet is not allowed to determine (or change) where its input stream comes from or where its output stream goes to. This has two advantages. First, a disklet does not handle buffering and scheduling for its communication, the operating-system layer does. This reduces the complexity of disklets. Second, in a heterogeneous environment with both Active Disks and conventional disks, this allows disklets that process data from conventional disks to be transparently executed on the front-end host (or on dedicated computation nodes).

Active Disks require a thin layer of operating system support (the *DiskOS*) at the disk. The DiskOS provides three services – memory management, stream communication and disklet scheduling. The stream-based model simplifies memory management as all memory is allocated in contiguous blocks whose size is known a priori and the lifetime of all blocks is known. The stream-based model also simplifies the communication support required as all stream buffers are allocated and managed by the DiskOS. Depending on the amount of memory available, it can allocate multiple buffers and overlap data movement and computation. The stream-based model also simplifies scheduling for disklets. A disklet is ready to run whenever there is new data available on one or more of its input streams.

3 Algorithms

In this section, we present Active Disk algorithms for eight common decision support tasks: select, aggregation, group-by, the *datacube* operation [24], external sort, project-join queries, datamining association rules from retail transaction data and maintaining materialized views. In addition, we also describe shared memory multiprocessor (SMP) algorithms for these tasks. We use the SMP algorithms to help compare the performance of Active Disks with that of SMPs. For each task, we started with a well-known efficient algorithm from the literature and adapted it for each architecture and the corresponding programming model.

For Active Disks, we adapted the algorithms to use the stream-based programming model. Note that, overlapping computation, communication, and I/O is handled by the DiskOS (the disk-resident OS layer) by using multiple buffers per stream. For SMPs, we adapted the algorithms to use one-way block-transfers (`shmemput/shmemget`) and remote queues [13] for moving data between processors. Given the volume of data being transferred and the one-way nature of the data movement, block-transfers and remote queues are suitable for these tasks. We striped each file over all disks using a 64 KB chunk size per disk. To take advantage of the aggressive I/O subsystem, each processor issues up to four 256 KB asynchronous requests (each request transferring 64 KB from four disks). Note that for `sort` and `join`, which shuffle their entire dataset and write it back to disk, we partitioned the disks into separate read and write groups (as in NOW-sort [7]). Since all processors can address all disks, we did not a priori partition the input datasets to processors. Instead, we maintained two shared queues (read/write) of fixed-size blocks in the order they appear on disk. When idle, each processor locks the queue and grabs the next block off the queue. This technique reduces the seek costs at the disks as the overall sequence of requests roughly follows the order in which data has been laid out on disk. A priori partitioning of the dataset would result in a potentially long seek for every request.

SQL `select` and `aggregate`: these are simple one-pass algorithms – `select` filters tuples from a relation based on a user-specified predicate and `aggregate` computes a single aggregate value for all tuples in a relation.[1] The Active Disk algorithm performs the filtering/aggregation locally and forwards the results to the front-end host. The front-end concatenates/aggregates data from different disks. In the SMP algorithm, each processor dynamically selects 256 KB chunks from the input relation and directly writes the results to the destination buffer using block-transfer. Both `select` and `aggregate` perform little computation/byte.

SQL `group-by`: The `group-by` operation computes a one-dimensional vector of aggregates indexed by a list of attributes [32]. It partitions a relation into disjoint sets of tuples based on the value(s) of index attribute(s) and computes an aggregate value for each set of tuples. Graefe [21] shows that hashing-based techniques outperform sort-based and nested-loop-based techniques for implementing the group-by operation. Accordingly, we used the hashing-based algorithm from [21] as the starting point for our algorithms. The Active Disk algorithm performs the `group-by` in two steps. In the first step, each disk performs local `group-by`s as long as the number of aggregates being computed fits in its memory. When it runs out of space at a disk, it ships the partial results to the front-end and reinitializes its memory. The front-end accumulates the partial results. In the SMP algorithm, each processor computes a local version of the `group-by`; results from all processors are merged at the end. Note that, in our experiments, `group-by` generates significantly larger results than `select`. It also performs more computation/byte as it needs to maintain a hash-table of aggregates.

Datacube: the `datacube` is the most general form of aggregation for relational databases. It computes multi-dimensional aggregates that are indexed by values of multiple aggregates [24]. In effect, a datacube computes group-bys for all possible combinations of a list of attributes. Several efficient methods for computing a datacube are presented in [4]. We use one of these algorithms, called *PipeHash*, as the starting point for our algorithms. PipeHash represents the datacube as a lattice of related group-bys. A directed edge connects group-by i to group-by j if j can be generated from i and has exactly one less attribute. Each edge has an

[1]Using one of the five SQL aggregation operations: `min`, `max`, `sum`, `avg` and `count`.

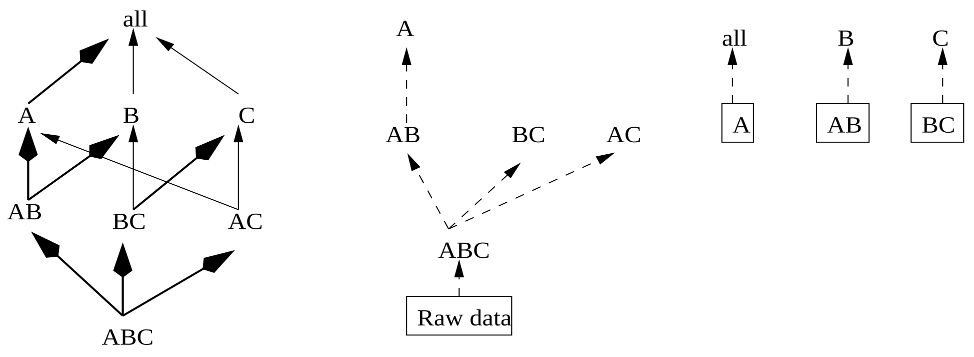


Figure 2: Search lattice and pipelines for the PipeHash algorithm. The bold lines in the search lattice indicate the minimum spanning tree computed by the algorithm using estimated sizes of individual group-bys. The right hand side shows four pipelines. In each pipeline, the data placed in a box at the bottom is read from disk.

associated weight which reflects the estimated size of the group-by. PipeHash determines the set of group-bys to perform by computing a minimum spanning tree over the lattice (see Figure 2 for an example). It schedules the group-bys as a sequence of pipelines; all the group-bys in a pipeline are computed as a part of a single scan of disk-resident data. The results of each pipeline are stored back on disk and are used as input for following pipelines (see Figure 2 for examples of pipelines). For individual group-bys, PipeHash uses a hashing-based technique [21]. The Active Disk algorithm partitions the available memory at each disk in proportion to the estimated size of the group-bys being performed in the pipeline (Figure 3). For the final combination operation for every group-by, it partitions the range of values of the attribute being grouped over to all the disks; each disk is responsible for combining results from all peers for that range of values. Each disk performs local `group-by`s as long as the number of aggregates being computed fits in its memory. When it runs out of space, it partitions the partial results and ships each partition to the disk that is responsible for the corresponding range of values. The SMP algorithm performs the group-bys in a batched manner – similar to that for `group-by`. After all the results for a group-by have been accumulated, the result is written to disk. Note that since `datacube` performs multiple group-bys in a single scan, it performs more computation per byte read than `group-by`. Also, since it computes a multi-dimensional aggregate, it generates and communicates significantly more data.

External sort: we used the two-pass parallel `NOW-sort` [7] as the starting point for our sort algorithms. NOW-sort is based on a long history of external sorting research in the database community (e.g. [3] and [34]) and currently holds the record for the fastest external sort (the Indy MinuteSort record [23]). The Active Disk algorithm uses two disklets for the first phase, the `partitioner` and the `sorter` (Figure 4(a)). The partitioner uses its scratch-space to form as many buckets as the number of disks. It examines each record and appends it to the bucket corresponding to its destination disk. When one of these buffers fills, it is forwarded to the `sorter` disklet on the destination disk. The `sorter` sorts each buffer using a partial radix sort and writes it to disk by sending it to the output stream. In the second phase, each sorted partition

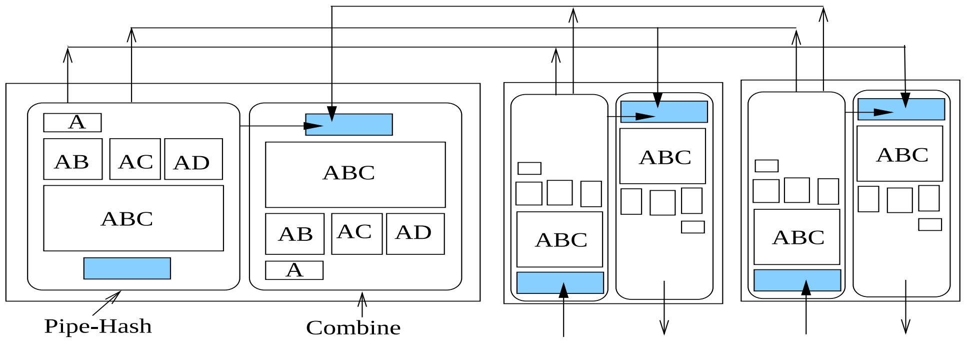


Figure 3: Schematic of disklet organization for `datacube`. There are two disklets running on each Active Disk, `pipe-hash` and `combine`. Shaded boxes indicate inputs to each disklet (i.e. stream buffers). Each `group-by` is allocated a hash table in the scratch space whose size is proportional to the estimated size of the `group-by`. After the pipeline completes execution (or runs out of memory), `pipe-hash` partitions the local `group-by` based on the keys and sends the partitions to `combine` disklets on destination disks. After all contributions are combined, `group-bys` are written to disk. Note that all the streams destined to a `combine` disklet are merged.

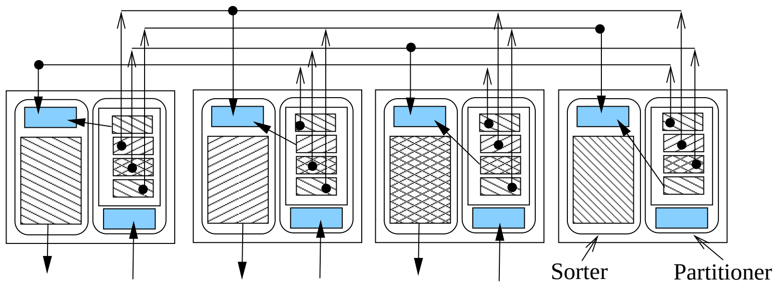
created in the first phase is mapped to a different stream; these streams are attached to a `merger` disklet as its inputs (Figure 4(b)). The `merger` selects the lowest key from *all* of its input streams and copies it to its output stream (which is mapped to the output file). Note that, no data is sent to host in this phase; merging is done locally at each Active Disk.

The SMP algorithm is multi-threaded. The first phase uses two threads on every processor - a *reader-thread* to read data and move tuple pointers to buckets and a *writer-thread* to sort each bucket with partial-radix sort[2] and write the bucket. The second phase uses three threads per processor to merge the sorted partitions created in the first phase. A *reader* reads one block from each sorted partition into one of four[3] sets of merge buffers; a *merger* selects the lowest-valued key from the current block of each partition and copies it to one of four write buffers; a *writer* writes buffers to disk.

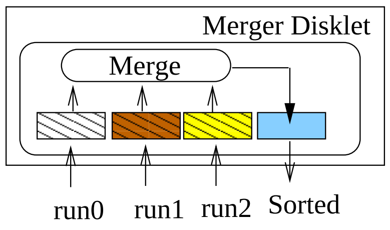
The Active Disk algorithm is fully pipelined in that it overlaps reading data, sending data to peers and sorting and writing data. The SMP algorithm overlaps just two operations; reading and writing operations are performed synchronously (Arpaci-Dusseau et al [7] recommend that for less than four disks, all operations should be overlapped whereas for more than four disks, only the first two should be overlapped). The first pass of these algorithms repartitions their entire input on disks. The second pass for the Active Disk algorithm is localized; each disk operates on its own partition. The SMP algorithm uses a disjoint set of disks for reading and writing in the first phase – it divides the total number of disks into two groups for this purpose. Note that the first pass of `sort` is communication-intensive and requires all-to-all communication. Since it repartitions its entire dataset, `sort` performs significantly more communication than `datacube`.

Project-Join query: we used sort-merge join algorithms for this task. A sort-merge join partially sorts

[2]Making two passes over the keys with a radix size of 11-bits [3] plus a cleanup.
[3]Two buffers in the original algorithm.



(a) First pass



(b) Second pass

Figure 4: Organization of the disklets for external sort. The `partitioner` maintains a bucket for each destination disk in its scratch space. When a bucket fills, it is forwarded to the corresponding `sorter` disklet. `Sorter` combines the tuples from all disks in its scratch space - when this space is filled, tuples are sorted and written to disk to create a *run*. All the streams destined to a `sorter` disklet are merged. Figure 4(b) presents the structure of a *merger* disklet that is used to merge multiple runs on a disk into a sorted file. Note that the second pass is local to each Active Disk.

each of the relations being joined and performs a join by stepping through the partially sorted relations using a pair of loops. We based our join algorithms on the two-pass NOW-sort. The first two passes of these algorithms are similar, in structure, to the first pass of a two-pass sort: the first pass repartitions and creates sorted *runs* for the first relation; the second pass does the same for the second relation. The third pass of these algorithms is similar to the second pass of a two-pass sort: it maintains a heap for the heads of the sorted runs for each relations and performs the join by picking elements from the two heaps. The first two passes of these algorithms have large communication and I/O requirements. The third pass for the Active Disk algorithm is localized as each disk operates on its own partition. The SMP algorithm uses a disjoint set of disks for reading and writing in the first two passes – it partitions the total number of disks into two groups for this purpose. Note that the first pass of `join` is communication-intensive and requires all-to-all communication. Since both `sort` and `join` repartition their entire dataset, their communication requirements are similar.

Datamining: we focus on determining frequent itemsets for mining association rules in retail transaction data [5]. We used the `eclat` algorithm [54] as the starting point for our algorithms. It is a multi-pass algorithm with the first two passes same as the *count distribution* algorithm proposed by Agrawal et al [6]. After the first two passes, it clusters the candidate itemsets into equivalence classes and uses these classes to filter, transpose and repartition the input data sets. The third pass is localized and does not require any communication. It is I/O-optimized as each processor is able to perform all its remaining computation with a single scan of its partition. Unlike external sort and sort-merge join, this algorithm repartitions only a fraction of its input dataset (the exact fraction depends on the parameters the algorithm is run with). The *eclat* algorithm was originally described for shared memory multiprocessors. We adapted it for Active Disks by reverting to *count distribution* in the first two passes. The original SMP algorithm performed fine-

```
CREATE VIEW foo AS
SELECT store.manager, sale.sale_id, item.item_id, item.item_name
FROM   store, sale, item, line
WHERE  store.store_id = sale.sale_id and
       sale.sale_id = line.sale_id and
       line.item_id = item.item_id and
       store.state = ''CA'' and
       sale.year = 1998
```

Figure 5: Example of a select-project-join view (from [40]).

grained updates; we modified it to batch the updates to the counters associated with itemsets. The original algorithm built a large triangular array of counters in its second pass. We noticed that a large fraction of the elements were zero in all our experiments and optimized it for memory consumption by using a sparse array. Note that `dmine` needs to communicate only the counters in the first two passes and a significantly reduced version of its input data in the third pass. Its communication requirements, therefore, are smaller than that of `sort`, `join` and `datacube`.

Maintaining Materialized Views: a view is a derived relation defined in terms of base relations stored in the database. Materialized views are pre-computed views that are used to speedup queries to decision support databases – in effect, materialized views are cached versions of views. Incremental view maintenance keeps cached views consistent with their base relations as the base relations are modified. Mumick et al [14, 40] have proposed efficient algorithms for maintaining select-project-join views (see Figure 5 for an example of such views). We borrowed the ideas of deferred maintenance and self-maintainable views from their algorithms. We first describe our core algorithm for parallel maintenance of select-project-join views. Then, we describe its Active Disk and SMP versions.

Each relation is partitioned over all disks using its join-attribute as the partitioning attribute. For the view defined in Figure 5, the relations `store`, `sale`, `line` and `item` are partitioned on the attributes `store_id`, `sale_id`, `item_id` and `item_id` respectively. An auxiliary view V_i is created for every base relation R_i (as in [40]) by applying the appropriate select and project operations. The auxiliary view corresponding to a relation is partitioned the same way as the relation itself and is kept sorted. Updates to the base relations are allowed to proceed immediately. This allows base relations to be consistent at all times. Updates to auxiliary views are deferred by adding the tuples to (unsorted) deltas. Updates to both auxiliary and primary views are propagated at *refresh* time. The algorithm propagates the updates one join at a time. We illustrate the structure of the algorithm by describing its operation for a sequence of two joins ($R_1 \bowtie R_2 \bowtie R_3$). The inputs for this join are: V_1, V_2 and V_3, the sorted auxiliary views for R_1, R_2 and R_3 respectively; and ΔV_1, ΔV_2, and ΔV_3, the corresponding (unsorted) deltas.

1. Create sorted runs for ΔV_1

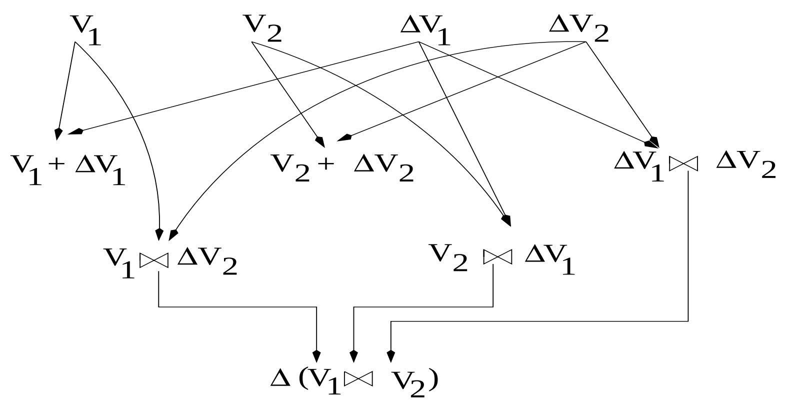


Figure 6: Merge operation in the materialized view algorithm. This operation executes five pipelines concurrently to produce $V_1 + \Delta V_1$, $V_2 + \Delta V_2$, $V_1 \bowtie \Delta V_2$, $V_2 \bowtie \Delta V_1$ and $\Delta V_1 \bowtie \Delta V_2$. $\Delta(V_1 \bowtie V_2)$ is obtained by taking the union of the output of last three pipelines. Note that, since pipelines are inter-dependent, the order in which the pipelines get executed is data-dependent.

2. Create sorted runs for ΔV_2
3. Merge V_1, V_2, sorted runs of ΔV_1, and sorted runs of ΔV_2 to generate: (a) updated and sorted versions of V_1 and V_2, and (b) $\Delta(V_1 \bowtie V_2)$. Recall that input versions of V_1 and V_2 are assumed to be already sorted. Details of this merge operation are shown in Figure 6.
4. Partition the tuples in $\Delta(V_1 \bowtie V_2)$, based on their join-attribute, into buffers destined for every disk. Send each buffer to its destination disk.
5. Receive tuples from $\Delta(V_1 \bowtie V_2)$ sent by other disks. Create sorted runs before writing to disk. *Note that steps 3, 4, and 5, that is, the merging, partitioning, and receiving operations are overlapped.*
6. Create sorted runs for ΔV_3
7. Merge $V_1 \bowtie V_2$, V_3, sorted runs of $\Delta(V_1 \bowtie V_2)$, and sorted runs of ΔV_3 to generate: (a) the updated $V_1 \bowtie V_2$ and V_3, and (b) $\Delta(V_1 \bowtie V_2 \bowtie V_3)$.

The Active Disk algorithm performs all operations in disk. The algorithm operates in a sequence of phases. Each phase performs one join and consists of a localized part, in which sorted runs are created and the merge operation is performed, and a distribution part in which the data for the next join is repartitioned. The deltas for the subsequent join are sorted into runs as they arrive at the destination disk. The SMP algorithm is similar.

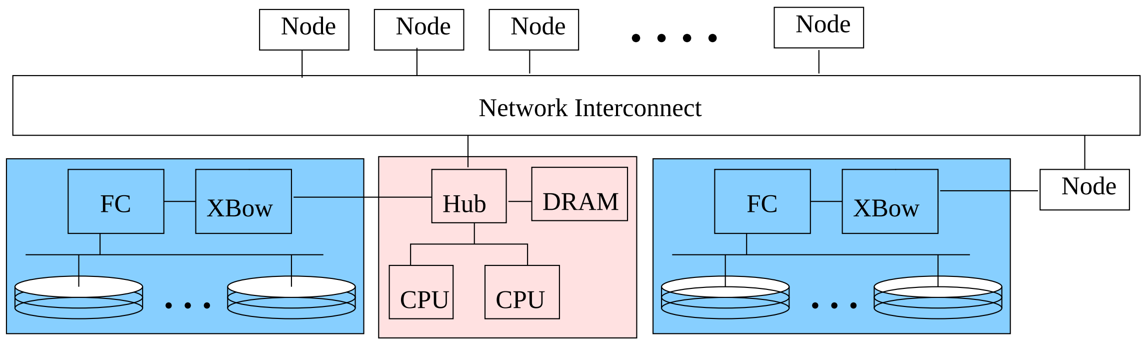


Figure 7: Schematic of the shared memory architectures assumed in the experiments.

4 Evaluation

4.1 Configurations

For comparison between architectures, we configured each of them with identical disks and used configurations with equal number of disks (and processors). For the rest of the components, we follow the configuration guidelines suggested by experts. For each architecture, we defined configurations with 16, 32, 64 and 128 disks (and processors). To understand the impact of scaling individual components and to identify the bottleneck resources for individual algorithms, we performed additional experiments by selectively scaling individual components.

For all configurations, we assumed disks similar to the Seagate 39102FC from the Cheetah 9LP disk family [45]. These disks have a spindle speed of 10,025 rpm, a formatted media transfer rate of 14.5-21.3 MB/s, an average seek time of 5.4 ms/6.2 ms (read/write) and a maximum seek time of 12.2 ms/13.2 ms (read/write). They support Ultra2 SCSI and dual-ported Fibre Channel interfaces.

Active Disks: For the Active Disk configurations, we assumed that: (1) a Cyrix 6x86 200MX processor (200 MHz) and 32 MB of 10ns SDRAM were integrated in the disk units; (2) all the disks were connected by a dual-loop Fibre-channel interface with a bandwidth of 200 MB/s (100 MB/s per loop); (3) the disks can directly address and communicate with each other using a SCSI-like interface; and (4) communications with clients are handled by a front-end host with a 450 MHz Pentium II and 1 GB RAM.

To identify the bottleneck resources for individual algorithms, we studied alternative configurations that individually scaled: (1) the aggregate bandwidth of the serial interconnect to 400 MB/s; and (2) the memory integrated into the disk unit to 64 MB and 128 MB. No software changes were associated with scaling the bandwidth of the serial interconnect. To take advantage of the additional memory available in the 64 MB/disk and 128 MB/disk configurations, the number of buffers allocated per stream by the DiskOS was increased from two to four and eight respectively. This allowed these configurations to tolerate longer communication and I/O latencies.

Shared memory multiprocessors (SMPs): For the SMP configurations, we followed the guidelines for configuring decision support servers (as quoted in [27]): (1) put as many processors in a box as possible to amortize the cost of enclosures and interconnects; (2) put as much memory as possible into the box to avoid going to disk as much as possible; and (3) attach as many disks as needed for capacity and stripe data over multiple disks to quickly load information into memory. We assumed an SMP configuration similar to the SGI Origin 2000: (1) two-processor boards (with 250 MHz processors) that directly share 128 MB memory; (2) a low-latency, high-bandwidth interconnect between these boards (1μs latency and 780 MB/s bandwidth); (3) a high-performance block-transfer engine (521 MB/s sustained bandwidth [28]); (4) a high-bandwidth I/O subsystem (two I/O nodes with a total of 1.4 GB/s bandwidth), similar to XIO, that connects to the network interconnect; and (5) a dual-loop Fibre Channel I/O interconnect (200 MB/s) for all disks. Figure 7 illustrates the SMP configurations. Note that the amount of memory is scaled with the number of processors – a 64-processor configuration having 4 GB and a 128-processor configuration having 8 GB.

We assumed that these machines ran a standard full-function operating system like IRIX and provided the `lio_listio` asynchronous I/O interface and user-controllable disk striping for individual files. Further, we assumed that these machines provided a *remote queue* abstraction (as suggested by Brewer et al [13]). To study the impact of variation in the I/O interconnect, we studied alternative configurations that scaled the bandwidth of the serial I/O interconnect to 400 MB/s.

4.2 Simulator

To conduct these experiments, we developed a simulator called *Howsim* which simulates all three architectures. *Howsim* contains detailed models for disks, networks and the associated libraries and device drivers and relatively coarse-grain models of processors and I/O interconnects.

For modeling the behavior of disk drives, controllers and device drivers, *Howsim* uses the *Disksim* simulator developed by Ganger et al [19]. *Disksim* has a detailed disk model that supports zoned disks, spare regions, segmented caches, defect management, prefetch algorithms, bus delays and control overheads. *Disksim* has been validated against several disk drives using the published disk specifications and SCSI logic analyzers; it achieves high accuracy - the worst case *demerit figure* [43] for *Disksim* is only 2.0% of the corresponding average response time [19]. For modeling I/O interconnects, *Howsim* uses a simple queue-based model that has parameters for startup latency, transfer speed and the capacity of the interconnect.

For modeling the behavior of user processes, *Howsim* uses a trace of processing times and I/O requests for individual tasks. It models variation in processor speed by scaling these processing times. To acquire the traces of processing time for user-level tasks, we implemented each algorithm on a DEC Alpha 2100 4/275 workstation with 256 MB of memory. We ran each algorithm with the same dataset and I/O request sizes as used in our experiments. For algorithms that use the amount of memory available as an explicit parameter

(Sort, Join, Datacube and Materialized Views), we generated traces for multiple memory sizes – to allow us to simulate architectures with different amounts of memory.

For modeling operating system behavior on hosts, *Howsim* uses parameters that represent the time taken for individual operations of interest: `read/write` system calls, context switch time, the time to queue an I/O request in the device-driver and the time to service an I/O interrupt. We obtained the first two using `lmbench` [31] on a 300MHz Pentium II running Linux (10μs for read/write calls, 103μs for context-switch). We charged a fixed cost of 16μs to queue an I/O request in the device-driver.

For Active Disks, *Howsim* models a preliminary implementation of *DiskOS* which provides support for scheduling disklets as well as for managing memory, I/O and stream communication. It uses a modified version of *DiskSim* that is driven by the disk operating system layer. Disklets are written in C and interact with *Howsim* using a stream-based API [2]. `Howsim` has additional parameters for the DiskOS. For this study, we assumed the system call and context switch costs on the DiskOS to be 1 μs. In addition, another 1 μs is charged to initiate a disk request from DiskOS and to service an interrupt from the disk mechanism. Given that disklets execute within the same protection domain as the DiskOS, we believe that these costs are reasonable.

For SMPs, *Howsim* models two-processor boards connected by a low latency, high bandwidth interconnect. For communication, it models one-way block-transfers, `shmemget/shmemput`, as available on the SGI Origin. Block transfers are suitable for the algorithms under consideration as they move large volumes of data in relatively large chunks. For synchronization on SMPs, *Howsim* provides spin-locks, remote queues [13] and global barriers. We used the at-memory fetch-and-op primitive as provided by SGI Origin for spin-locks (which cost around 3μs [26]). *Howsim* models a high-bandwidth I/O subsystem similar to the XIO subsystem available in the Origin 2000. The disk model is driven by a striping library written on top of the raw-disk access library.

4.3 Datasets

We used 16 GB datasets for all the tasks except `join` for which we used a 32 GB dataset and `mview` for which we used a 15 GB dataset. In this section, we describe the structure of the datasets for the different tasks.

Select, Aggregate, Group-by: for these tasks, we used a dataset with about 268 million tuples, each tuple being 64 bytes. For `select`, we test a single 4-byte field with a selectivity of 1%. For `aggr` and `group-by`, we used the `sum` operation on a 4-byte field. For `group-by`, we used a 4-byte field with 13.5 million distinct values as the grouping attribute.

Datacube: for `dcube`, we used a dataset with 536 million tuples. Each tuple had eight 4-byte attributes. We used four attributes as group-by attributes and the remaining four as aggregation attributes with `sum` as the aggregation function. The number of distinct values for each of the group-by attributes were 5.36

million, 536,000, 53,600 and 5,360. We created this dataset by scaling one of the datasets used in the paper that described the *PipeHash* algorithm [4].

Sort: for `sort`, we used a dataset with 100-byte tuples and 10-byte uniformly distributed keys. The total number of tuples was about 170 million. We created this dataset based on the standard sort benchmark described in [23].

Project-Join: for `join`, we used a dataset with 64-byte tuples and 4-byte uniformly distributed keys. The projection operation extracted eight 4-byte fields from each tuple. Each relation was 16 GB and contained 268 million tuples. The output for `join` was about 108 MB.

Datamining: for `dmine`, we used a dataset with 300 million transactions. The total number of items was 1 million and the average length of the transactions was 4 items. We generated this dataset using the Quest datamining dataset generator which we obtained from IBM Almaden [39]. For generating the frequent itemsets, we used a *minimum support* parameter of 0.001 (0.1%).

Materialized views: for `mview`, we used a dataset with three 4 GB derived relations (134 million tuples each). The size of deltas for each relation was assumed to be 1 GB each (33.5 million tuples each). We assumed 32-byte tuples for the derived relations with 4-byte fields. Further, each join operation produced a delta view of 1 GB.

5 Results

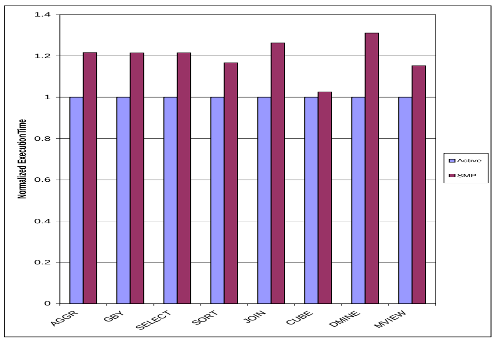
Figure 8 compares the performance of all eight tasks on comparable configurations of both architectures. The results for each task on configurations of a particular size (16/32/64/128) are normalized with respect to the performance of the same task on the Active Disk configuration of the same size. We make three observations:

- For the same I/O interconnect, Active Disk configurations perform better than the corresponding SMP configurations. We note that this is true for all tasks and all configurations examined in our experiments. This includes tasks with small amounts of computation per byte of I/O (e.g., `aggregate`, `select`, and `group-by`), as well as tasks with substantially larger amounts of computation per byte of I/O (e.g., `sort`, `dcube`, `dmine`). Note that this is in spite of the fact that the Active Disks contain a 200 MHz CPU whereas the SMP configurations contain a 250 MHz CPU.

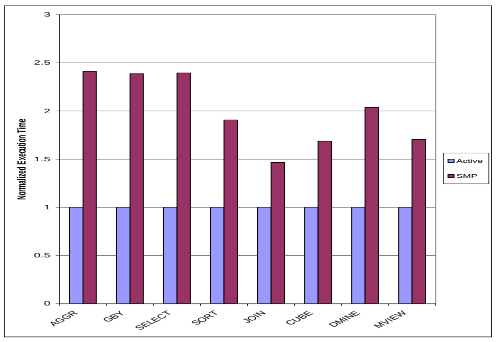
- The performance advantage of Active Disks increases with configuration size. For 16-disk configurations, the decision support tasks we examined run up to 1.3 times faster on Active Disks; for 128-disk configurations, these tasks run between 3 and 9.5 times faster on Active Disks. The largest performance differences (8.5-9.5 fold on 128-disk configurations) are for tasks that allow large data reductions on

Active Disks, such as `aggregation` and `select`. However, even tasks that repartition all or part of their input dataset, such as `sort`, `join`, `dmine`, `mview`, are significantly faster (4-6 fold on 128-disk configurations) on Active Disks. Note that, the performance of `group-by` on Active Disks does not scale beyond 32-disk configurations. This is due to the serialization of the result collection at the front-end host.

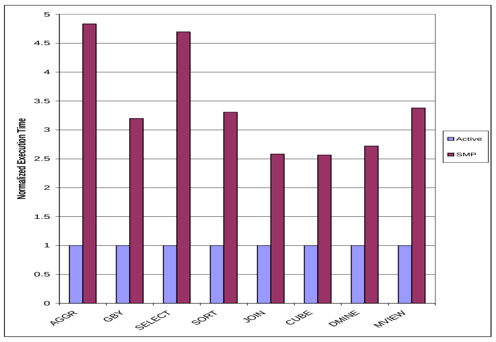
- For tasks that repartition all or part of their input datasets, the performance advantage is primarily due to the local nature of subsequent processing. In SMP configurations, data accessed in all phases passes over the I/O interconnect, whereas in Active Disk configurations, no data passes over the I/O interconnect after repartitioning.



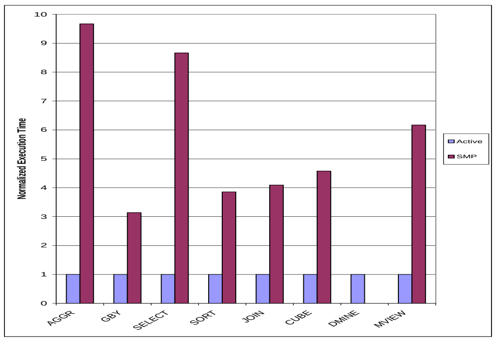
(a) 16 disk configurations



(b) 32 disk configurations



(c) 64 disk configurations



(d) 128 disk configurations

Figure 8: Performance of all eight tasks on comparable configurations of both architectures. The results for each task on configurations of a particular size (16/32/64/128) are normalized with respect to the performance of the same task on the Active Disk configuration of the same size. Note that the range of the y-axis is *different* for every graph. *The bar for* `dmine` *on the 128-node SMP configuration is missing as we could not complete the simulation in time.*

Figure 9 shows how the performance of each task scales for both architectures. The performance of most tasks saturates relatively early on SMP configurations. Tasks whose performance is dominated by I/O, such as `select`, `aggregate` and `group-by`, achieve no benefit from larger configurations. Task performance scales better on Active Disks. The performance of `group-by` on Active Disks saturates earlier than comparable tasks such as `select`. This is because for large configurations, the front-end host becomes a bottleneck for combining the results from individual disks. Note that `join` seems to achieve better than perfect scaling for both architectures. We suspect that this is due to cache capacity effects during the merge phase. A similar result was reported by Arpaci-Dusseau et al [7] for NOW-sort with more than 17 runs per node (our experiments had between 2 and 80 runs per node).

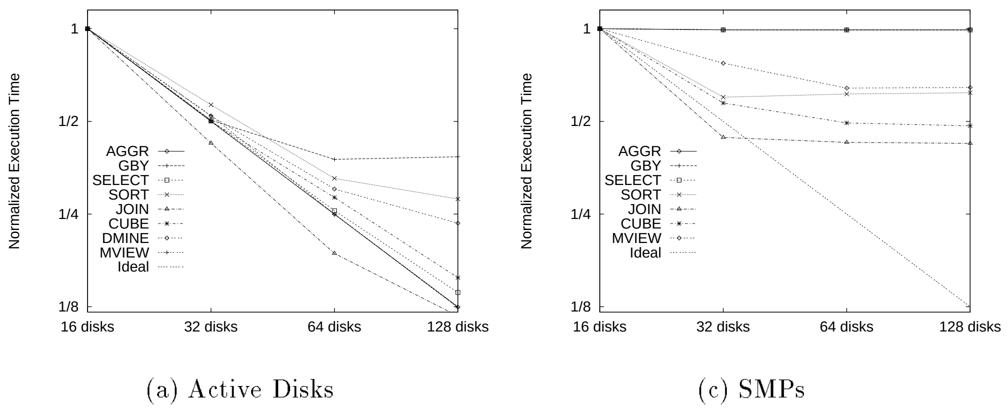
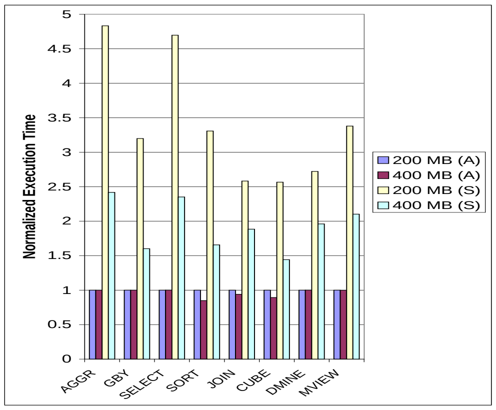
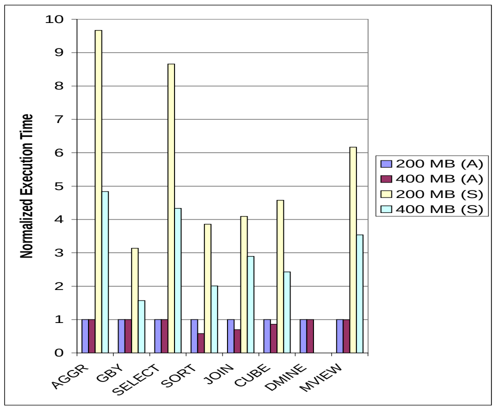


Figure 9: Variation in performance of all eight tasks with configuration size for both architectures. The results for each task on a given architecture are normalized with respect to the performance of the same task on the 16-node configuration of the same architecture. The `ideal` curve is for reference and indicates perfect scaling. Note that `join` seems to achieve better than perfect scaling for all architectures. We suspect that this is due to cache capacity effects during the merge phase.

Figure 10 examines the impact of varying the I/O interconnect for both architectures. We note that doubling the I/O interconnect bandwidth has a large impact on the performance of SMP configurations for all tasks. This indicates that the I/O interconnect is a major bottleneck for these tasks on SMP configurations. For the Active Disk configurations, only `sort`, `join` and `dcube` achieve some, albeit small, performance improvement. This indicates that even for 64-disk and 128-disk configurations, the I/O interconnect does not become a bottleneck for Active Disks. We would like to point out that Active Disk configurations with a 200 MB/s I/O interconnect outperform SMP configurations with a 400 MB/s I/O interconnect (up to 1.5 to 4.8 times faster for these tasks on 128-disk configurations).



(a) 64-disk configurations



(b) 128-disk configurations

Figure 10: Impact of varying the I/O interconnect bandwidth for both architectures. The results for each task on a given configuration are normalized with respect to the performance of the same task on the Active Disk configuration of the same size. In the legend, **A** stands for Active Disks and **S** stands for SMP.

6 Discussion

In this section, we discuss two issues. First, we discuss the integration of Active Disks into the software architecture of decision support databases. Second, we give an indication of the cost of comparable configurations for Active Disks and shared memory multiprocessors.

Relational databases have long used demand-driven dataflow based architectures. According to Graefe [21], *"this model of operator implementation and scheduling resembles very closely those used in relational systems, e.g., System R (and later SQL/DS and DB2), Ingres, Informix, and Oracle as well as in experimental systems, e.g., the E programming language used in EXODUS [41], Genesis [12], and Starburst [25]. Operators implemented in this model are called iterators, streams, synchronous pipelines, row-sources, or similar names in the lingo of commercial systems"*. According to Barclay et. al. [11], *"relational databases are ideally suited to dataflow approach"* and that *"the database community has adopted a dataflow approach to describe and implement parallel algorithms"*.

The stream-based programming model proposed for Active Disks closely resembles the operator/iterator based model used by relational databases. Corresponding to the `open`, `next`, `close` operations for an iterator, a stream in this model has `openStream`, `getNextBuffer`, `closeStream` operations. Disklets attached to a stream perform the operations corresponding to the iterator function. The scratch space for a disklet corresponds to the *state record* for an iterator. According to Graefe [21], *"the type of state records is different for each iterator as it contains iterator-specific arguments and local variables (state), while the iterator is suspended, e.g., currently not active between invocations of the operator's* `next` *procedure"*. This is exactly the purpose of the scratch space in disklets. Given this close correspondence between the models, we expect that with a stream-based programming model, Active Disks could be integrated into the software architecture of decision support databases with relative ease.

It remains a question, however, whether database vendors would choose to integrate Active Disks into their software. Keeton et. al. [27] indicate that initial reactions to active disks have been mixed: some industrial database experts suggesting that Active Disks can succeed only if they require minimal change to existing database software. We believe that this initial reluctance can be overcome for two reasons. First, with the stream-based programming model, changes required to integrate Active Disks into existing database software are not likely to be extensive. Second, in other situations requiring modifications to existing software, database vendors have not been inflexible. Examples include the migration of most database servers to SMPs and the recent incorporation of Java virtual machines in some databases. To cite examples of the latter, Oracle's *Aurora* project [33] plans to integrate a Java virtual machine in the database engine such that Java code can call SQL and SQL can call Java code (*Aurora* is scheduled to ship with Oracle 8.1). IBM's DB2 UDB5 allows extension of the database server using Java user-defined functions and stored procedures [15]. User-defined functions may be used in an SQL expression to compute a complex function of several values in a given row. They can also be used in the FROM clause in a query for on-the-fly creation of tables.

To provide an indication of price/performance ratio for the architectures compared in this study, we estimated the prices of both architectures for 64 node configurations. We estimated the price of the SMP configuration using the SGI Origin 2000. The Avalon project at Los Alamos Labs quotes the list price of a 64-processor SGI Origin 2000 with 250MHz processors and 8 GB memory to be about $1.8 million [9]. Estimating the cost of 4 GB of memory to be (a generous) $300,000, we estimate the cost of the configuration studied in this paper to be about $1.5 million. We estimated the price of the Active Disk configuration using Seagate ST39102 as the disks ($670) with Cyrix 6x86 200MHz as the embedded processor ($45), 32 MB SDRAM as the embedded RAM ($35), $100 for the Fibre Channel interconnect and $150 as the *premium* for being a high-end component. We assumed the Dell Poweredge 4300 as the front-end host with a LP3000 Fibre Channel host bus adaptor from the Emulex Corporation.[4] With these components, the price for a 64 disk configuration adds up to about $74,000 ($1000 for each Active Disk, $600 for the Fibre Channel adaptor and $9000 for the front-end).

7 Related work

The idea of embedding a programmable processor in a disk is not new. The I/O processors in the IBM 360 allowed users to download *channel programs* that were able to make I/O requests on behalf of the host programs [38]. One of the ISAM implementations on the IBM 360 used channel programs to traverse disk-resident linked lists. Database machines [46] in the late 1970s proposed various levels of processor integration

[4] *http://www.emulex.com*

into disk: processor per track [30, 35, 47, 49], processor per head [10, 29], and “off-the-disk” designs that used a shared disk cache with multiple processors and multiple disks [16, 44]. There are several differences between the database machines of late 1970's and today's Active Disks. Unlike the database machine processors, the processors proposed for Active Disks are general-purpose commodity components and are likely to evolve as the disks evolve. Unlike the low-bandwidth interconnects used in database machines, today's serial I/O interconnects such as Fibre Channel provide high bandwidth for commodity disk drives. Unlike the limited repertoire of operations performed by database machines, Active Disks take advantage of algorithmic research for shared-nothing architectures to provide efficient implementations of a wide variety of operations. Furthermore, Active Disks can be used to perform operations for non-relational data such as image processing [2, 42] and file-system and security-related processing [18, 51].

Riedel et al [42] have also evaluated Active Disk architectures for databases. They show that Active Disks are able to provide scalable performance for nearest neighbor search in multimedia databases, for frequent itemset determination for datamining association rules in retail transaction data and for edge-detection algorithms. Their results indicate that these algorithms can achieve significant gains from the use of Active Disks. Based on an analysis of several technological trends, Keeton et al [27] propose an architecture (IDISK) in which a processor-in-memory chip (IRAM [37]) is integrated into the disk unit and the disk units are connected by a crossbar. They argue that IDISK architectures offer several potential price and performance advantages over traditional server architectures for decision support.

Given the volume of data processed and the cost of fetching data from disk, optimizing I/O-intensive algorithms is often a matter of setting up efficient pipelines where each stage performs some processing on the data being read from disk and passes it on to the next stage [1, 7]. As a result, dataflow-based models have been proposed by several researchers. Barclay et al [11] proposed a dataflow-based technique for parallelizing the loading of a large database. Similar techniques are used by the Gamma [17] and Volcano [20] parallel databases. Recently, Arpaci-Dusseau et al [8] have proposed a dataflow-based programming model for scheduling I/O-intensive tasks on clusters.

8 Conclusions

There are four main conclusions of our study. First, for the same I/O interconnect, disks, and number of processors, Active Disk configurations perform better than the corresponding SMP configurations for a wide variety of decision support tasks. We note that this is true for all tasks and all configurations examined in our study. This includes tasks with small amounts of computation per byte of I/O as well as tasks with substantially larger amounts of computation per byte of I/O. This is in spite of the fact that the Active Disks contain a 200 MHz CPU whereas the SMP configurations contain a 250 MHz CPU. We would like to point out that Active Disks achieve this performance at a small fraction of the cost of shared memory multiprocessor servers. The performance of the SMP configurations was limited by I/O interconnect bandwidth, particularly for large configurations. The I/O interconnect sits in between all the processors and all the disks; the data for many tasks passes over it multiple times. Our results indicate that, Active Disk configurations with a 200 MB/s I/O interconnect outperform SMP configurations with a 400 MB/s I/O interconnect (up to 1.5 to 4.8 times faster on 128-disk configurations).

Second, the performance advantage of Active Disks increases with configuration size. For 16-disk configurations, the tasks we examined in this study run up to 1.3 times faster on Active Disks; for 128-disk configurations, these tasks run between 3 and 9.5 times faster on Active Disks. The largest performance differences (8.5-9.5 fold on 128-disk configurations) are for tasks that allow large data reductions on Active Disks. Even tasks that repartition all or part of their input dataset, are significantly faster (4-6 fold on 128-disk configurations) on Active Disks.

Third, we note that adding more memory to Active Disks provides limited advantage. Our results indicate that increasing the disk-memory to 128 MB provides less than 8% performance for 16 and 32 disk configurations, less than 11% for 64 disk configurations and less than 21% for 128 disk configurations. This is not surprising given the streaming nature of decision support tasks.

Finally, we note that there is a close correspondence between the stream-based programming model proposed for Active Disks and the operator/iterator model used by most relational database systems. Given this similarity, we expect that with a stream-based programming model, Active Disks could be integrated into the software architecture of decision support databases with relative ease.

Acknowledgments

We would like to thank Greg Ganger for the *DiskSim* simulator. We would like to thank Mohammed Zaki for his implementation of the *eclat* algorithm which we used as the codebase for our implementation. We would like to thank Quest datamining group at IBM Almaden for their dataset generator. We would like to thank Ioana Stanoi and Amr El Abbadi for discussions regarding maintaining materialized views. We would like to thank Jim Humphries and the other recipients of `para-staff@umiacs.umd.edu` for their continued support.

References

[1] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz, and A. Sussman. Tuning the performance of I/O-intensive parallel applications. In *Proceedings of the Fourth ACM Workshop on I/O in Parallel and Distributed Systems*, May 1996.

[2] A. Acharya, M. Uysal, and J. Saltz. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of ASPLOS VIII*, pages 81–91, Oct 1998.

[3] R. Agarwal. A super scalar sort algorithm for RISC processors. In *Proceedings of 1996 ACM SIGMOD International Conference on Management of Data*, pages 240–6, 1996.

[4] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proceedings of the 22nd International Conference on Very Large Databases*, pages 506–21, 1996.

[5] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD Conference on Management of Data*, pages 207–16, 1993.

[6] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):962–9, 1996.

[7] A. Arpaci-Dusseau, R. Arpaci-Dusseau, D. Culler, J. Hellerstein, and D. Patterson. High-performance sorting on networks of workstations. In *Proceedings of SIGMOD'97*, 1997.

[8] R. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. Culler, J. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with Rivers: Making the Fast Case Common. Submitted for publication, 1998.

[9] The Avalon FAQ[5], 1998.

[10] J. Banerjee, R. Baum, and D. Hsiao. Concepts and capabilities of a database computer. *ACM Trans. on Database Systems*, 3(4), Dec 1978.

[11] T. Barclay, R. Barnes, J. Gray, and P. Sundaresan. Loading databases using dataflow parallelism. *SIGMOD Record*, 23(4):72–83, 1994.

[12] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twitchell, and T. Wise. GENESIS: An extensible database management system. *IEEE Transactions on Software Engineering*, 14(11), 1988.

[13] E. Brewer, F. Chong, L. Liu, S. Sharma, and J. Kubiatowicz. Remote queues: Exposing message queues for optimization and atomicity. In *Proc. of the 7th SPAA*, pages 42–53, 1995.

[14] L. Colby, T. Griffin, L. Libkin, I. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proc. of SIGMOD'97*, pages 469–80, 1997.

[15] IBM DB2 Java Enablement. http://www.software.ibm.com/data/db2/java/index.html, 1998.

[16] D. DeWitt. DIRECT - a multiprocessor organization for supporting relational database management systems. *IEEE Trans. on Computers*, 28(6):395–406, Jun 1979.

[5] http://cnls.lanl.gov/avalon/FAQ.html

[17] D. DeWitt, S. Ghandeharizadeh, and D. Schneider. A performance analysis of the Gamma database machine. *SIGMOD Record*, 17(3):350–60, 1988.

[18] G. Gibson et al. File server scaling with network-attached secure disks. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '97)*, 1997.

[19] G. Ganger, B. Worthington, and Y. Patt. The DiskSim Simulation Environment Version 1.0 Reference Manual[6]. Technical Report CSE-TR-358-98, Dept of Electrical Engineering and Computer Science, Feb 1998.

[20] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. *SIGMOD Record*, 19(2):102–11, 1990.

[21] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, Jun 1993.

[22] J. Gray. Put EVERYTHING in the Storage Device. Talk at NASD workshop on storage embedded computing[7], June 1998.

[23] J. Gray. The Sort Benchmark Home Page. Available at *http://research.microsoft.com/research/barc/-SortBenchmark/*, 1998.

[24] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proceedings of the 12th International Conference on Data Engineering*, pages 152–9, New Orleans, February 1996.

[25] L. Haas, W. Chang, G. Lohman, J. McPherson, P. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst midfight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[26] D. Jiang and J. Singh. A methodology and an evaluation of the SGI Origin 2000. In *Proc. of the Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 171–81, Madison, WI, June 1998.

[27] K. Keeton, D. Patterson, and J. Hellerstein. The Case for Intelligent Disks (IDISKS). *SIGMOD Record*, 27(3), 1998.

[28] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *In Proc. of Intl. Symposium on Computer Architecture*, pages 241–51, Denver, CO, June 1997.

[6] Available at http://www.ece.cmu.edu/ ganger/disksim/disksim1.0.tar.gz
[7] *http://www.nsic.org/nasd/1998-jun/gray.pdf*

[29] H. Leilich, G. Stiege, and H. Zeidler. A search processor for database management systems. In *Proc. of VLDB'78*, 1978.

[30] S. Lin, D. Smith, and J. Smith. The design of a rotating associative memory for relational database applications. *ACM Trans. on Database Systems*, 1(1):53–75, Mar 1976.

[31] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *In Proc. of 1996 USENIX Technical Conference*, Jan 1996.

[32] J. Melton and A. Simon. *Understanding the New SQL: A Complete Guide.* Morgan Kaufman, 1993.

[33] S. Meyer. Oracle's Aurora Java Virtual Machine. In *Proc. of OOPSLA-98*, page 181, 1998. In the panel on "The New Crop of Java Virtual Machines".

[34] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: a RISC machine sort. In *Proceedings of 1994 ACM SIGMOD International Conference on Management of Data*, Minniapolis, May 1994.

[35] E. Ozkarahan, S. Schuster, and K. Sevcik. Performance evaluation of a relational associative processor. *ACM Trans. on Database Systems*, 2(2), Jun 1977.

[36] G. Papadopolous. The future of computing. Unpublished talk at NOW Workshop, July 1997.

[37] D. Patterson et al. Intelligent RAM (IRAM): the Industrial Setting, Applications, and Architectures. In *Proceedings of the International Conference on Computer Design*, 1997.

[38] D. Patterson and J. Hennessey. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, 2nd edition, 1996.

[39] IBM Quest Data Mining Project. The Quest retail transaction data generator[8], 1996.

[40] D. Quass, A. Gupta, I. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. of PDIS'96*, 1996.

[41] J. Richardson and M. Carey. Programming constructs for database system implementation in EXODUS. In *Proc. of SIGMOD'87*, 1987.

[42] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large scale data mining and multimedia applications. In *Proceedings of 24th Conference on Very Large Databases*, 1998. To appear.

[43] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, March 1994.

[8] Available at *http://www.almaden.ibm.com/cs/quest/syndata.html.*

[44] S. Schuster, H. Nguyen, E. Ozkarahan, and K. Smith. RAP.2 - an associative processor for databases and its applications. *IEEE Trans. on Computers*, 28(6), 1979.

[45] Seagate Technology Inc. *The Cheetah 9LP Family: ST39102 Product Manual*, July 1998. Publication number 83329240 Rev B.

[46] D. Slotnick. Logic per track devices. *Advances in Computers*, 10:291–6, 1970.

[47] D. Slotnick. Logic per track devices. *ACM Trans. on Database Systems*, 1(3), Sep 1976.

[48] P. Strenstrom, E. Hagersten, D. Lilja, M. Martonosi, and M. Venugopal. Trends in shared memory multiprocessing. *IEEE Computer*, 1997.

[49] S. Su and G. Lipovski. CASSM: a cellular system for very large databases. In *Proc. of VLDB'75*, pages 456–72, 1975.

[50] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 203–16, 1993.

[51] R. Wang. A file system for intelligent disks[9]. Talk at NASD/NSIC Meeting, June 1998.

[52] R. Winter and K. Auerbach. Giants walk the earth: the 1997 VLDB survey. *Database Programming and Design*, 10(9), Sep 1997.

[53] R. Winter and K. Auerbach. The big time: the 1998 VLDB survey. *Database Programming and Design*, 11(8), Aug 1998.

[54] M. Zaki, S. Parthasarathy, and W. Li. A localized algorithm for parallel association mining. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1997.

[9]http://www.nsic.org/nasd/1998-jun/wang.pdf