Distrib Parallel Databases (2006) 20:199–224
DOI 10.1007/s10619-006-0238-5

GRACE-based joins on active storage devices

Vassilis Stoumpos · Alex Delis

Published online: 25 September 2006

Abstract Contemporary long-term storage devices feature powerful embedded processors and sizeable memory buffers. *Active Storage Devices* (ASD) is the hard disk technology that makes use of these significant resources to not only manage the disk operation but also to execute custom application code on large amounts of data. While prior research has shown that ASDs perform exceedingly well with filter-type algorithms, the evaluation of binary-relational operators has been limited. In this paper, we analyze and evaluate inter-operator parallelism of GRACE-based join algorithms that function atop ASDs. We derive accurate cost expressions for existing algorithms and expose performance bottlenecks; upon these findings we propose *Active Hash Join*, a new algorithm that exploits all system resources. Through experimentation, we confirm that existing algorithms are best suited for systems with either small or large numbers of ASDs. However, we find that the "adaptive" nature of Active Hash Join yields enhanced parallelism in all cases, especially when the aggregate ASD resources are comparable to the main CPU and main memory.

Keywords Active storage devices · Join processing on disk architectures · Adaptive hash-join · Intelligent disks · Evaluation of algorithms

1 Introduction

Powerful processors and sizable memory chips are embedded in modern hard disk drives as the means to improve arm scheduling and cache management. The *Active Storage Devices*

Recommended by: Ahmed Elmagarmid

Work partially supported by the University of Athens Research Foundation.

V. Stoumpos · A. Delis (✉)
Department of Informatics and Telecommunications, University of Athens, 15771 Athens, Greece
e-mail: ad@di.uoa.gr

V. Stoumpos
e-mail: stoumpos@di.uoa.gr

(ASD) architecture uses on-disk resources to not only offer enhanced traditional I/O services but also to execute "imported" application pertinent code and manage data [1, 18]. In an ASD architecture a sizeable fraction of application-level code that operates on data can be shifted from the main CPU and be placed to the storage devices. ASDs use the surplus of processing power of their embedded processors and their available extra buffer space to execute application-level code [1].

We use the term ASD to unify previous hardware architecture proposals like Active-Disks [18], Intelligent-Disks [11], Network Attached Storage [8], and Smart Disk Cluster [15] into a higher-level view of the storage subsystem. From this perspective, we don't propose a new hardware architecture, but acknowledge the fact that it is both feasible and advantageous to build ASDs: devices that, besides long-term storage, offer an embedded CPU and embedded memory buffers. Regardless of the differences between hardware architectures, applications are allowed access to ASD-embedded resources, similarly to what holds for the main system resources. In the following, we discern between *disk* and *host* resources, i.e. resources available either embedded in ASDs or on the main system. For simplicity we assume that an array of ASDs is available and that each constituent ASD is attached to the main system bus.

We follow the *Disklet* [1] programming paradigm that is general enough to adapt to all hardware architectures discussed. A Disklet is a small piece of data processing code, that is initially installed on the ASD and afterwards is triggered whenever data is read from or written to the disks; it can also be viewed as a data filter. Disklets read data from multiple input streams and propagate processing results to multiple output streams as well. In addition, the Disklet paradigm is flexible enough to allow chaining of Disklets in the same ASD to implement complex algorithms. The operation of disklets always supersedes regular I/O operation. Since the effective bandwidth of individual disk units is mainly restricted by the seek and rotation times, ASDs impose no further constraints because pages are effectively retrieved/stored and processed, if necessary, in pipelined fashion.

The rationale behind ASDs is to "migrate" processing closer to data [10]. This paradigm is common nowadays and extends to a wide range of applications from intelligent computer peripherals that offer ease of use, to sophisticated graphics cards that offload the host CPU from geometric and visual manipulations [6]. In ASDs, "migrating" processing closer to data improves algorithm performance for to two reasons: the *inherent parallelization* in an ASD array and the *reduced bus bandwidth needs*. Although less powerful than the host CPU, the embedded processors in an array of ASDs may operate on data in parallel. Furthermore, given enough storage devices, the aggregate processing power and memory on ASDs exceed the host CPU speed and memory size. Hence, the attained parallelism is expected to rise as more disks are integrated in the array. In terms of bus utilization, only the result of data processing needs to be transfered through the system bus, for input data is processed on disks. Therefore, "pushing" processing to disks greatly reduces the needs in bus bandwidth, since for most data processing algorithms the size of the result is orders of magnitude less than the input.

It has been shown that ASDs demonstrate promising performance for various applications such as image processing, data mining, multimedia applications, data warehousing, and decision support systems [1, 3, 11, 18]. These applications rely on filter-type algorithms, which are naturally suited for ASDs. In the context of database systems, ASDs have also been proven beneficial, especially in the case of filter-type queries, like table scan [12, 17]. In addition, complete database query plan trees, with multiple join and sort operators, are possible to decompose into smaller parts that may be executed effectively on ASDs [15, 23]. However, inter-operator parallelism in binary operators such as joins has received limited

attention. Previous research has mostly focused either on the specific hardware architecture choices to support ASDs, or on the execution of complex, commercial load queries [11, 15, 18, 23]. In this paper, we propose, analyze and evaluate equi-join operators in the context of ASDs, focusing on inter-operator parallelism.

Equi-join is arguably the most useful relational operator, but there is no straightforward implementation of it atop ASDs. In contrast to filter-type algorithms, that merely pass over the input data once, equi-join algorithms scan their input multiple times and flush to disk potentially voluminous intermediate data [22], in addition to the join result. Furthermore, ASD-equipped systems constitute a multi-computing environment: a parallel processing system composed of heterogeneous processors interconnected with a common bus. Subsequently, only parallel or distributed algorithms are suited for ASDs; based on the techniques used to reorganize the data prior to actual processing these algorithms fall under either the *symmetric partitioning* or *fragment and replicate* techniques [9]. In this paper, we consider only GRACE-based joins as they offer opportunities for parallelism and are the key reference of symmetric partitioning techniques. Our main motivation is to better understand the behavior of join algorithms in the context of ASDs and not to consider every possible join algorithm and produce a winner.

Processing migration to ASDs is not always beneficial though. To better capture the trade-offs involved, we use the processing model proposed in [18] to capture all data processing as a pipeline consisting of the following stages: *data reading* from the disks, *processing on the disk*, *transfer to the host* intermediate data through the system bus, so that subsequent *processing on the host* CPU occurs; then result data is *transfered to the disks*, over the system bus again, for the final *writing* to the disks. These distinct pipelined stages should ideally last the same so that no bottlenecks are formed. System and application parameters affect how long each stage lasts. In practice, the number of employed disks, the bus bandwidth, the input to output size ratio, the processing load, and other parameters, may introduce different bottlenecks. Of course, join algorithms, which operate in multiple separate steps, cannot be modelled by a single pipeline; we have to consider one pipeline for every algorithmic step. Depending on the pipeline bottleneck and the load distribution, ASDs resemble, in terms of operation, either a distributed or a parallel system. This dual nature of ASD architectures poses challenges for efficient data processing and more specifically for database operations. These challenges include effective load sharing, interconnect bandwidth usage, and sustenance of high throughput.

Proposed symmetric partitioning equi-join algorithms fall in two categories: either place the full processing load on the ASDs, or become oblivious of ASDs altogether. Modeling the behavior of these algorithms showed that in some steps resources are underutilized, so we propose a new equi-join algorithm, termed *Active Hash Join*, that fully harnesses all resources available. Towards this goal, Active Hash Join considers the diversity of the processing resources and balances load accordingly so that no processing bottlenecks are formed. Therefore, we conduct detailed simulation experiments that enable us to examine the performance features of the presented algorithms and empirically evaluate their trade-offs. The embedded nature of ASDs renders experimentation with prototypes very challenging. Thus, we resort to simulating their operation in par with prior efforts [1, 17]. In our simulated environment, given that a large number of disk devices is available, we ascertain that join techniques which function exclusively at the host are outperformed by their ASD-based counterparts. If however, the aggregate resources available on all disks are comparable to those in the host in terms of both processing power and memory size, we establish that Active Hash Join's distribution of processing load to all constituent resources offers much enhanced parallelism.

Database machines [5] share a number of similarities with ASDs as they involve multiple independent disks for high I/O throughput and parallel processing. However, database machines used specialized hardware that did not trade-off well cost with attained performance [2]. Modern technology has lifted these limitations by making constituent system components more powerful and affordable: all hard disk devices today feature standardized, mass-produced CPUs and memory chips that allow for highly sophisticated interfaces, as Fiber Channel Arbitrated Loop (FC-AL) etc. [21]. Hence, there is room for performance improvement by adding more processing capacity on disks and making database management systems aware of it. ASD-enabled systems involve multiple independent disks, similar to RAIDs [4] to achieve high I/O rates. But, contrary to ASDs, RAIDs target high availability via redundancy and ease of data management and administration. Thus, filling a RAID with ASDs transforms it into a parallel data processing substrate, opening new alternatives to data management. As arrays of disks are now very common in high-end database management systems and embedded processors are widely spread, it is possible to lift the performance bottleneck of the storage subsystem. It is thus, our belief that modern database management systems should be ASD-aware in order to achieve maximum performance.

In the following, we present the model on which we form cost expressions in Section 2. Section 3 describes our GRACE-based equi-join algorithms and respective load balancing strategies. In Section 4, we analytically estimate the performance of all algorithms and in Section 5, we evaluate them using our simulation engine and present our main experimental results. Conclusions and future work are found in Section 6.

2 Processing model for ASDs

2.1 Model description

Computer systems that feature ASDs allow for effective *pipelined* execution of data processing algorithms. The execution pipeline consists of six distinct stages: *read*, *disk processing*, *shipping to host*, *host processing*, *shipping to disks*, and *write*. These stages, which are illustrated in Fig. 1, capture a typical data processing operation. Initially, data is read from the disk platters to on-disk buffers and processed by disk-embedded CPUs. Then data is shipped from disks to host memory buffers, via the system bus, so that further processing is carried out by the host CPU. Finally, data is shipped over the system bus to on-disk buffers and data is written to disk platters. As pipeline operation is the basis of our performance model for ASDs, filter-type algorithms can be easily fashioned after it. On the other hand, more complex algorithms, that scan input data multiple times or write intermediate data to disks, are best viewed as a series of separate steps, where each step is modelled with a different pipeline. This is the case for equi-joins over large relations: temporary or intermediate data is

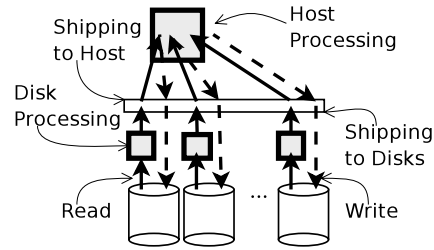


Fig. 1 Stages of a pipeline operation on ASDs

stored on disk [22]; in addition to the join output that needs to be stored on the disks to support cursors on the result or long operator pipelines in the query execution engine. Therefore, join algorithms are divided in distinct steps, each of which is modelled with a single pipeline that reads, process, and writes data in a certain fashion. We consider all pipelines independently; the overall algorithm performance is a combination of how all pipelines operate.

Given a certain pipeline, that an algorithm step is fashioned after, the processing model of Fig. 1 yields interesting indications about which of the 6 pipeline stages is less efficient. Ideally, all pipeline stages should last the same; longer lasting stages are performance bottlenecks. To this effect, response time expressions may expose potential bottleneck stages. Throughput rate expressions are also useful, but do not reveal the pipeline bottleneck, for the bottleneck does not necessarily operate at the lowest throughput rate. For example, the pipeline bottleneck may produce sizeable amounts of output, but another pipeline stage may produce such a small output that its throughput rate is less than the corresponding rate of the bottleneck. In order to reach sound conclusions, when we consider the throughput rate of each pipeline stage, we also need to take into account the utilization of system components and the size of data transferred between stages. Of course, all these measures are interrelated but we consider them all, because they offer a multitude of ways to approach and analyze the operation of a pipeline.

Clearly, model expressions are valid for a single pipeline and do not hold for the entire execution of a multi-step algorithm. Although this may seem as a drawback of the model, it is in fact a useful tool, as it reveals certain deficient parts of algorithm execution that otherwise would be "hidden". In the same spirit, estimating overall algorithm performance by combining the characteristics of all pipelines is in most cases straightforward. For example, the overall algorithm response time is the sum of the respective response times of all pipelines. In this Section we present the performance model in the context of a single pipeline or, in other words, in the case of a single-step algorithm. We exploit our findings for the multi-step join algorithms in Section 3.

2.2 Model performance expressions

All model expressions are formed with the help of system and application parameters, presented in Tables 1 and 2 respectively. Starting from the time spent on every pipeline stage, we need to take into account that N_{in} bytes of input are read from the disks and only N_{out} bytes of the result are written back. The disk and host CPUs process the data executing w_{disk} and w_{host} instructions per byte of input. As a result of disk processing, the size of data processed by the host processor is N_{mid}, which may not be equal to the pipeline input size.

Table 1 System parameters

Measure	Description
S_{host}	Host CPU speed in instructions per second
S_{disk}	Disk embedded CPU speed in instructions per second
M_{host}	Host memory size in bytes
M_{disk}	Disk embedded memory size in bytes
B_{bus}	Disk bus bandwidth in bytes per second
B_{read}	Disk read bandwidth in bytes per second
B_{write}	Disk write bandwidth in bytes per second
$\alpha = \frac{B_{\text{write}}}{B_{\text{read}}}$	Disk bandwidth ratio of write over read. $\alpha \approx 0.85$
d	Number of disk drives

Table 2 Application parameters

Measure	Description
N_{in}	The size of the pipeline input in bytes
N_{out}	The size of the pipeline output in bytes
N_{mid}	The size of the disk processing stage in bytes
w_{host}	Host CPU instructions per byte of input
w_{disk}	Disk embedded CPU instructions per byte of input

In general, the pipeline stages are not completely independent; read and write stages operate on the same disks, while data shipment stages between disks and host operate on the system bus. Whenever stages of the same pipeline operate on a common resource, the resource is time-shared. Therefore, the time required for these stages to complete is the sum of the time each stage lasts if it made exclusive use of the resource. The total time needed for the pipeline to complete is the maximum of the time needed to read and write data ($T_{\text{read}} + T_{\text{write}}$), the time needed to transfer data between disks and host ($T_{\text{up}} + T_{\text{down}}$), the time needed to process the data on disk (T_{disk}), and the time needed to process data on host (T_{host}). The overall response time expression follows:

$$T = \max\left\{\frac{N_{\text{in}}}{dB_{\text{read}}} + \frac{N_{\text{out}}}{dB_{\text{write}}}, \frac{N_{\text{mid}} + N_{\text{out}}}{B_{\text{bus}}}, \frac{N_{\text{in}} w_{\text{disk}}}{dS_{\text{disk}}}, \frac{N_{\text{mid}} w_{\text{host}}}{S_{\text{host}}}\right\} \tag{1}$$

For every stage, the throughput rate is calculated as the amount of data produced at that stage over the amount of time the stage lasted. The throughput rate expressions follow:

$$R_{\text{read}} = \frac{N_{\text{in}}}{T_{\text{read}} + T_{\text{write}}} = \frac{\alpha N_{\text{in}}}{N_{\text{out}} + \alpha N_{\text{in}}} dB_{\text{read}} \tag{2}$$

$$R_{\text{write}} = \frac{N_{\text{out}}}{T_{\text{read}} + T_{\text{write}}} = \frac{\alpha N_{\text{out}}}{N_{\text{out}} + \alpha N_{\text{in}}} dB_{\text{read}} \tag{3}$$

$$R_{\text{up}} = \frac{N_{\text{mid}}}{T_{\text{up}} + T_{\text{down}}} = \frac{N_{\text{mid}}}{N_{\text{mid}} + N_{\text{out}}} B_{\text{bus}} \tag{4}$$

$$R_{\text{down}} = \frac{N_{\text{out}}}{T_{\text{up}} + T_{\text{down}}} = \frac{N_{\text{out}}}{N_{\text{mid}} + N_{\text{out}}} B_{\text{bus}} \tag{5}$$

$$R_{\text{disk}} = \frac{N_{\text{mid}}}{T_{\text{disk}}} = \frac{N_{\text{mid}}}{N_{\text{in}}} \frac{dS_{\text{disk}}}{w_{\text{disk}}} \tag{6}$$

$$R_{\text{host}} = \frac{N_{\text{out}}}{T_{\text{host}}} = \frac{N_{\text{out}}}{N_{\text{mid}}} \frac{S_{\text{host}}}{w_{\text{host}}} \tag{7}$$

The throughput rates presented here are valid under the assumption that each pipeline stage lasts as long as it is necessary. In practice, all stages need to synchronize, so they last as long as the bottleneck stage, which is defined as the longer lasting stage. Consequently, if a pipeline stage becomes a bottleneck, the throughput rate of other stages drops as they are forced to last longer. Note that as not all stages produce the same volume of output we cannot infer the bottleneck stage by the lowest output rate. We derive the overall pipeline throughput rate from $R = \frac{N_{\text{out}}}{T}$, where T is the overall pipeline response time from Eq. (1). Finally, utilization estimates for the different components in an ASD configuration are produced as the fraction of the time the component would need to carry out its load if it had exclusive use of all

resources, over the time the pipeline lasted in practice. From (1), using the respective time estimations, the disk and bus bandwidth utilization estimations are $U_{rw} = (T_{read} + T_{write})/T$ and $U_{bus} = (T_{up} + T_{down})/T$ respectively. Similarly, the utilization of the disk processor is $U_{disk} = T_{host}/T$ and $U_{host} = T_{disk}/T$ is host processor utilization.

The expressions presented here capture the basic characteristics of pipeline execution, regardless of the algorithm used. These expressions also hold for systems equipped with traditional disks, where no processing and, thus, no data reduction on disks occur. Therefore, for systems that rely on traditional disks, we replace $w_{disk} = 0$ and $N_{mid} = N_{in}$ in the model expressions. We note that when traditional disks are used, the disk processing rate becomes infinitely large,[1] which denotes that the disk processing stage is a mere pass-through, imposing no delays at all.

2.3 A sample application

To better understand the model expressions, we consider a sample data filtering application and various alternatives to run it on an ASD architecture. Assume two commutative filtering operators: a *heavyweight* and a *lightweight*. The heavyweight filtering comes at a cost of 5 instructions per byte of input and filters out all but one byte for every 100 bytes of input. On the other hand, the lightweight operator filters out half of the input and requires 1 instruction per byte of input. The sample application filters its input through both filters; an example of such an application is a table-scan query with two selection predicates, each with different evaluation cost. In this scenario, ASD architectures offer two types of processing resources to execute the two operators, opening in total four alternatives: place both operators on host, place both operators on disks, place the heavyweight operator on host while leaving the lightweight on disks, and vice versa. In the first alternative, the host CPU caries out all filtering, so we name it *host-only*. In host-only, the lightweight operator precedes the heavyweight to reduce total number of executed instructions. For N bytes of input, filtering with the lightweight operator and then with the heavyweight incurs $N \cdot 1 + (N/2) \cdot 5 = N \cdot 3.5$ instructions. If operators were exchanged, a total of $N \cdot 5 + (N/100) \cdot 1 = N \cdot 5.01$ instructions would be required. A similar approach is followed by the *disk-only* alternative, with the exception that the disk-embedded CPUs do all the filtering. Finally, *heavy-on-disks* and the *heavy-on-host* alternatives respectively place the heavyweight operator on disks and on the host. Note that the heavy-on-disk alternative incurs $N \cdot 5.01$ instructions, while the other three alternatives require $N \cdot 3.5$ instructions. To complete the sample application scenario, we assume a system equipped with a 200 MB/s bus, that features a host CPU running at 1 GHz and d disk-embedded CPUs running at 50 MHz. The size of the input is 1 GB.

We compare all four alternatives with a varying number of disks, ranging from 1 to 256, to examine system performance as a function of the number of employed disks. We first substitute in Eq. (1) all parameters, except from the number of disks, according to each processing alternative. The plots of the analytical model expressions for all four cases are shown in Fig. 2 with the number of disks placed on the x-axis, in logarithmic scale, and the response time on the y-axis, in logarithmic scale too. As shown in Fig. 2(a), in the host-only case the read and write stages are the longer lasting, for as much as 20 disks. For more than 20 disks, the bottleneck shifts to the system bus, but it could have also been the host CPU for a different example. In the heavy-on-host alternative, presented in Fig. 2(c), we

[1] Replacing w_{disk} with 0 in Eq. (6) yields ∞.

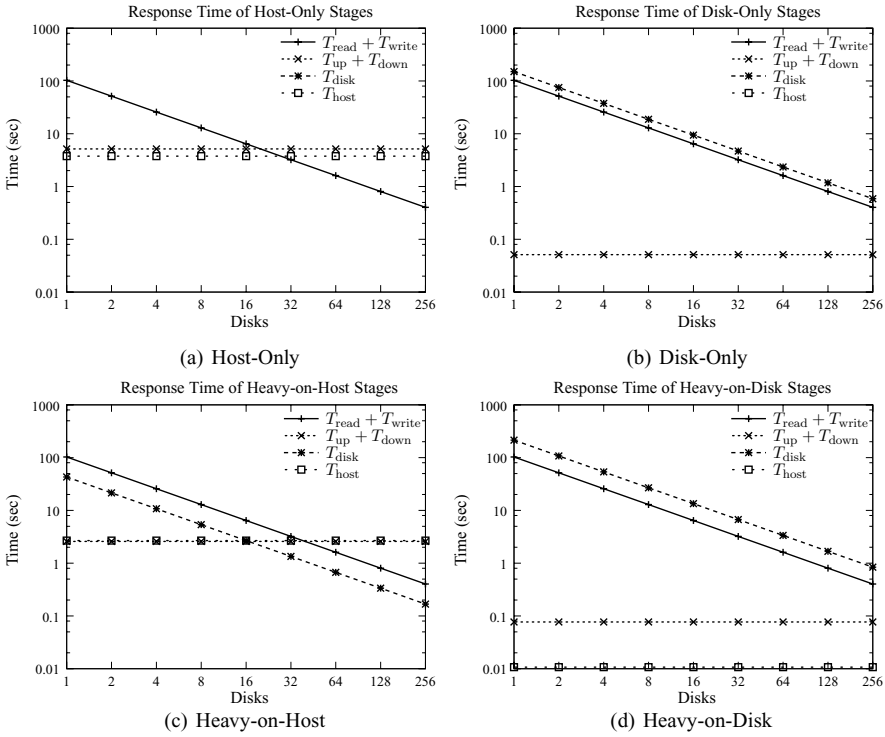


Fig. 2 Analytical estimation of pipeline stages response time as the number of disks in the system increases, for the four different load balancing alternatives in the sample application. The scale is logarithmic in both axes

see the same pattern; the difference here is that host processing stage and the stages that ship data over the system bus are faster, since part of the processing is carried out on the disks. Therefore, for more than 20 disks the overall response time is shorter compared to the host-only case. Continuing, with Figs. 2(b) and (d), we see that in the disk-only and the heavy-on-disk cases the disk CPUs are saturated, because they are significantly slower than the host CPU. In all cases though, the disk processing stage scales well as the number of disks increases; the bus is saturated for more than 256 disks, outside the plot range. Therefore, the overall pipeline response time becomes less as more disks become available, eventually outperforming host-only and heavy-on-host alternatives.

To complete our view of the system performance, we also evaluate the analytical throughput rate expressions of all pipeline stages for all four alternatives. We substitute in Eqs. (2)–(7) the values of the system parameters for the sample application. We also substitute application parameters according to the four load balancing arrangements and list the results in Table 3 in MB/s. Each column stands for the throughput rate of an individual pipeline stage. Regardless of the load balancing in effect, the amount of data read and written remains the same, thus read and write throughput rates are identical. It is also important to note that the throughput rate of the disk processing stage R_{disk} rises as more ASDs are available. Thus, load balancing arrangements that place the larger portion of the load on ASDs, such as disk-only and heavy-on-disk, offer better scalability. In these two cases, the disk-embedded

Table 3 Throughput rates for the 4 load balancing arrangements. Throughput rates are in MB/sec, with d being the number of employed disks

#	Load balancing	w_{disk}	w_{host}	R_{read}	R_{write}	R_{up}	R_{down}	R_{disk}	R_{host}
(1)	Host-only	0.0	3.5	$19.9d$	$0.1d$	199.0	1.0	$+\infty$	1.36
(2)	Disk-only	3.5	0.0	$19.9d$	$0.1d$	100.0	100.0	$0.07d$	$+\infty$
(3)	Heavy-on-host	1.0	5.0	$19.9d$	$0.1d$	198.0	2.0	$23.84d$	1.91
(4)	Heavy-on-disk	5.0	1.0	$19.9d$	$0.1d$	133.3	66.7	$0.10d$	476.84

CPUs are slow and become bottlenecks. Hence, it is only with massive parallelism that we achieve high performance in these cases.

In general, the feasible load balancing arrangements heavily depend on the application at hand. This has to do with how easy it is to split the execution of the algorithm used to address the application. In addition, the number of employed disks changes the system characteristics drastically, so proper load balancing schemes must also take into account the number of employed disks. Thus, we start by establishing a set of feasible balancing arrangements for the algorithm at hand. Then, we select the optimal load balancing arrangement for a certain range of disks in the system. With the above application, we demonstrated that it is not straightforward to execute algorithms on ASDs. We established the set of four feasible balancing arrangements considering the execution of two operators on two processing nodes. We demonstrated that no arrangement is optimal in the general case. Instead, we can only guarantee that an alternative is preferred over the others only for a certain range of disks. It is clear that in an ASD system balancing the processing load of any application is a function of numerous parameters, with the number disks being the most important.

3 Join algorithms on active storage devices

We consider equi-joins $R \bowtie S$ of two relations R and S and always assume the outer relation R to be smaller one. The number of blocks and the size of a tuple in relation R are $|R|$, and t_R correspondingly. In the following, we model each join algorithm step as a pipeline. This process poses two problems: first, it is not easy to break the incurred processing into isolated parts as it is the case with database `SELECT` for example [1]. Second, joins potentially yield substantial data writing that affects overall system performance [22]. We consider GRACE-based, or *symmetric partitioning* techniques [9], that require two phases: initially, both relations are *partitioned* by hashing on the value of the join attributes. During the second phase, the relation partitions that end-up on the same processing node are joined independently from other partitions. At the end, the union of all partial results forms the result set.

ASDs constitute a hybrid between parallel and distributed systems. The number of employed disks and the amount of data shipped over the system bus are key in deciding whether a specific ASD configuration should be treated as a distributed or a parallel system. Whenever the bus bandwidth is over-utilized, ASD systems are similar to distributed systems, where the interconnect network bandwidth is the vital resource. This is the case when *sizeable* data transfers occur or when a *large* number of disks is used to offer high aggregate disk bandwidth. On the other hand, whenever a *small* number of disks is used or *shorter* data transfers occur, ASD systems bare greater resemblance to parallel systems where multiple processing nodes are interconnected with a high speed bus. In this context, we seek

the combination of employed disks and bus bandwidth consumption that allows for high efficiency. In the following, we discuss the operation of join algorithms so that, in Section 4, we can estimate the amount of data they send through the system bus. In Section 5, we evaluate algorithm performance under different system configurations, mainly by varying the number of employed disks. Consequently, we identify combinations of algorithm and range of disks that are highly efficient.

3.1 GRACE-based join algorithms

With the exception of already sorted relations, it has been shown that hash-based join algorithms outperform others in both single and multiprocessor systems [20, 22]. The *Simple Hash Join* (SH) algorithm [14] illustrates the basic operation of all hash-based join algorithms: a hash table of the outer relation is built, so that the inner relation tuples probe the hash table for matches in an efficient way. The *GRACE Hash Join* (GH) algorithm [14] uses a hash function on the join attribute to partition both relations in *buckets* small enough so that SH is carried out on each pair of buckets without hash overflows that are expensive. Thus, GH involves a *bucket forming* phase that creates the relation buckets and then a series of join phases each computing the join of each segment, that in union yield the join result. A combination of Simple Hash Join and GRACE Hash Join, named *Hybrid Hash Join* (HH) [22], concurrently runs the bucket forming phase with the first join phase, where the join result of the first bucket pair is produced. Overlapping the two phases enhances the performance of HH compared to other algorithms [20], but complicates the cost estimation expressions making the performance bottlenecks harder to identify. In order to describe in more detail the behavior of an ASD system, we do not consider the Hybrid Hash Join algorithm, but rather opt for it as a straightforward improvement.

In the frame of symmetric partitioning techniques, we have three options when it comes to selecting the processing nodes for join evaluation: take into account only the host resources, only the disk resources, or both. Although each option gives a "different" algorithm, all algorithms follow the steps presented in Fig. 3. In the beginning, two *redistribution steps* form the relation buckets; we name these two steps the *redistribution phase*. Depending on the number of created buckets, multiple *build* steps are required along with the matching *probe steps*. The build step reads an outer relation bucket and the probe step reads the corresponding inner relation bucket; we name these two steps together *join phase*. Before we consider each option, we note the following two aspects:

1. *Overflow-free join*. In the bucket forming phase of all algorithms we constrain the bucket size so that only overflow-free joins occur, in the GRACE spirit. Thus, depending on the available memory at the processing node, the bucket sizes form accordingly.
2. *Virtual buckets*. Typically, the host CPU processes data at a much higher rate than a single disk can read. To overcome this deficiency, the host CPU always operates on *virtual buckets*, that is buckets stripped across all disks [13]. When a virtual bucket is read, all disks contribute with their respective portions and collectively exhibit high read rate.

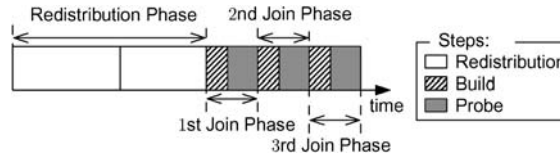


Fig. 3 The steps GRACE-based join algorithms consist of

Algorithm 1 GRACE Hash Join — GH

```
1: k_GH = ⌈F|R|/|M_host|⌉ {*number of iterations*}
2: for all i = 1 to k_GH do
3:     create R_i and S_i virtual buckets
4: create Result virtual bucket
5: for all t ∈ R do {*form outer buckets*}
6:     write t to R_i, for i = hash(t, k_GH) + 1
7: for all t ∈ S do {*form inner buckets*}
8:     write t to S_i, for i = hash(t, k_GH) + 1
9: for i = 1 to k_GH do
10:    create empty host memory hash table H
11:    for all t ∈ R_i do {*build hash table*}
12:        hash t in H
13:    for all t ∈ S_i do {*probe hash table*}
14:        probe H with t for matches
15:        write matches ⋈ t to Result
```

3.2 The GRACE hash join algorithm—GH

Ignoring the processing capabilities of ASDs, GRACE Hash Join (GH) places all processing load on the host CPU, using the host's buffer space. Algorithm 1 provides an outline for GH. Two preliminary *redistribution* steps partition inner and outer relation tuples into k_{GH} buckets, using the same hash function on the join attributes. The hash function *hash*(t, k_{GH}) hashes tuple t to $[0 \ldots k_{\text{GH}} - 1]$. Then, a *build* step follows to build an in-memory hash table with the tuples in an outer relation bucket. In the succeeding *probe* step, tuples from the corresponding inner relation bucket are used to probe the hash table for matches and output join results. More build and probe steps follow until all k_{GH} buckets are processed. To lessen the I/O bottleneck, all relation buckets and the join result are stored as virtual buckets. In this regard, all disks contribute to attain collectively high read and write rate. The number of iterations (build/probe steps) required is determined by the amount of free host memory. Relying only on $|M_{\text{host}}|$ blocks forces the number of iterations k_{GH} to be more or equal to $F|R|/|M_{\text{host}}|$, where F is an incremental factor to deal with non perfect hashing of tuples in the hash table [22]. Using only the host processor, allows for a single relation bucket pair to be processed in every join iteration.

3.3 The Disk Hash Join algorithm—DH

Disk Hash Join (DH) [17] relies solely on disk processing power and aims at high parallelism. The rationale behind DH is that for a large number of disks the available on-disk resources are far more than their host counterpart. DH follows the common algorithmic structure presented in Fig. 3 and its outline is listed in Algorithm 2. The redistribution phase of DH stores $d \times k_{\text{DH}}$ buckets of R evenly on all d disks, along with the matching buckets from S. Thus, after hashing tuple t in $[0 \ldots d \cdot k_{\text{DH}} - 1]$, i and j are set appropriately so that t ends up in the bucket on disk j that will be used in the i-th join iteration. In this bucket forming scenario, all buckets are stored exclusively on the disk that they will be processed on, contrary to what happens in GH where buckets are stripped. Therefore, the build and probe steps are executed efficiently, without copying tuples through the system bus. Indeed, the i-th build step creates one hash table on every disk j, for $j \in [1 \ldots d]$, using tuples from buckets $S_{i,j}$. Then, the i-th probe step follows that reads the matching $S_{i,j}$ buckets and probes all d on-disk hash tables in parallel. The system bus is used only to propagate the join result to the host memory and

Algorithm 2 Disk Hash Join — DH

```
1: k_DH = ⌈F|R|/|dM_disk|⌉ {*number of iterations*}
2: for all i = 1 to k_DH do
3:    create R_{i,j} and S_{i,j} buckets on all disks j
4: create Result virtual bucket
5: for all t ∈ R do {*form outer buckets*}
6:    let i = hash(t, d · k_DH)divd + 1 and j = hash(t, d · k_DH)modd + 1
7:    write t to R_{i,j}
8: for all t ∈ S do {*form inner buckets*}
9:    let i = hash(t, d · k_DH)divd + 1 and j = hash(t, d · k_DH)modd + 1
10:   write t to S_{i,j}
11: for i = 1 to k_DH do
12:   create empty disk hash tables H_j on all disks j
13:   for all disks j concurrently do {*build disk hash tables*}
14:      for all t_j ∈ R_{i,j} do
15:         hash t_j in H_j
16:   for all disks j concurrently do {*probe disk hash tables*}
17:      for all t_j ∈ S_{i,j} do
18:         probe H_j with t_j for matches_j
19:         write matches_j ⋈ t_j to Result
```

flush the result relation, so the host CPU and memory are minimally utilized. The bucket size is limited by the on-disk memory and, hence, must not exceed M_{disk}. Since d bucket pairs are processed in every join phase, one on each disk, at least $F|R|/(d|M_{\text{disk}}|)$ iterations are needed.

3.4 The active hash join algorithm—AH

Here, we propose Active Hash Join (AH), that exploits both host and disk resources and follows the common skeleton we present in Fig. 3. In order to harness all system resources, AH creates two types of buckets during the redistribution phase: virtual buckets, similar to those used in GH, that are to be processed on the host CPU, and single-disk buckets, similar to those used in DH, that are to be processed by the disk-embedded CPUs. This approach allows for the build and probe steps to operate as if GH and DH where executed concurrently. We illustrate the data flow in AH in Fig. 4, where solid lines indicate input to join processing stages and dashed lines indicate join results flow. The amount of processing carried out by the host and disk processors is proportional to the amount of data they process. In the general case, ρ part of the input is processed on the disks and $(1-\rho)$ on the host; we use the

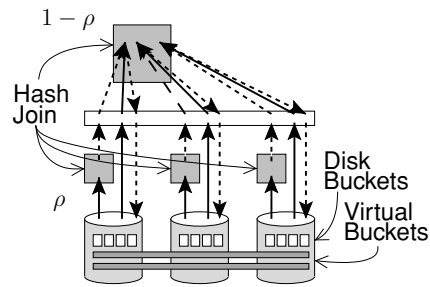


Fig. 4 Data flow in active hash join

term *load balancing factor* for ρ. The selection of the optimal values for ρ is related to the configuration of the system and defer its discussion to Section 4.

Algorithm 3 lists an outline of our proposed AH method, for a load balancing factor ρ. In AH, the two redistribution steps use a common hash function to hash each tuple in $[0 \ldots 1]$. Tuples that fall in $[0 \ldots \rho)$ will be processed on the disks, so they end up in one of $d \cdot k_{\text{AH}}$ buckets, in a DH fashion. Similarly, tuples that fall in $[\rho \ldots 1]$ will be processed on the host, so they are partitioned in k_{AH} GH-type buckets. In total, $(d+1) \cdot k_{\text{AH}}$ disjoint buckets from each relation are created after the redistribution phase. In every join iteration, d hash tables are built on disks and one in host memory; then all $d+1$ hash tables are probed for matches in parallel. In more detail, the i-th build step uses tuples from $R_{i,j}$ to populate the hash table on disk j, and tuples from R_i to populate the host hash table. Similarly, the i-th probe step uses tuples from $S_{i,j}$ to probe the hash table on disk j, and tuples from S_i to probe the host hash table. Hash tables built on ASDs must not exceed M_{disk} bytes, the maximum buffer space on disks, and the host memory hash table must be less than M_{host}. As far as on-disk processing is concerned, only ρ part of the $|R|$ outer relation blocks are used to build the disk-resident hash tables, so at least $k_{\text{disk}} = \rho F|R|/(d|M_{\text{disk}}|)$ steps are required. Similarly, we use the remaining $(1-\rho)$ part of the outer relation blocks $|R|$ to populate the host hash table, so $k_{\text{host}} = (1-\rho)F|R|/|M_{\text{host}}|$ steps are necessary from the viewpoint of host processing. In order to avoid hash-overflows we safely set the number of iterations k_{AH} above $\max\{k_{\text{disk}}, k_{\text{host}}\}$. With this setting, we split the processing load according to ρ and, at the same time, avoid hash overflows. In every join iteration, $\rho|R|/k_{\text{AH}}$ blocks are used to

Algorithm 3 Active Hash Join—AH

```
1: k_disk = ⌈ρF|R|/d|M_disk|⌉
2: k_host = ⌈(1 − ρ)F|R|/|M_host|⌉
3: k_AH = max{k_disk, k_host} {*number of iterations*}
4: for all i = 1 to k_AH do
5:     create R_{i,j}, S_{i,j} buckets on all disks j
6:     create R_i, S_i virtual buckets
7: create Result virtual bucket
8: for all t ∈ R do {*form outer buckets*}
9:     if hash(t, 1) < ρ then
10:        write t to R_{i,j}, as DH would
11:    else
12:        write t to R_i, as GH would
13: for all t ∈ S do {*form inner buckets*}
14:    if hash(t, 1) < ρ then
15:        write t to S_{i,j}, as DH would
16:    else
17:        write t to S_i, as GH would
18: for i = 1 to k_AH do
19:    create empty host memory hash table H
20:    create empty disk hash tables H_j on all disks j
21:    for all  all disks j and host concurrently do {*build d + 1 hash tables*}
22:        for all t ∈ R_i and t_j ∈ R_{i,j} do
23:            hash t in H, as GH would
24:            hash t_j in H_j, as DH would
25:    for all all disks j and host concurrently do {*probe d + 1 hash tables*}
26:        for all t ∈ S_i and t_j ∈ S_{i,j} do
27:            probe H with t, as GH would
28:            probe H_j with t_j, as DH would
```

populate all d disk-resident hash tables; the remaining $(1-\rho)|R|/k_{\mathrm{AH}}$ blocks are used to populate the host memory hash table.

4 Analysis of algorithms

In the following, we analyze the performance characteristics of all join algorithms discussed. We consider one step at a time and point to the parameters that govern the performance of the pipelines. We start with the redistribution step, which is common to all algorithms, and continue with the build and probe steps for every algorithm in particular. Finally, we discuss optimality criteria that help designate the load balancing factor ρ in AH, based on system characteristics.

Throughout this discussion, we will refer to Tables 4 and 5, that list the application parameters for different algorithmic steps. For every algorithmic step, that is modelled as a pipeline, we present the input size N_{in} and the output size N_{out} in Table 4, along with the amount of data N_{mid} copied from the disks to the host. Table 4 contains the aggregate amount of block transfers for all steps to facilitate comparisons between steps; for example the k_{GH} build steps of GH read in total $|R|$ blocks. Similarly, Table 5 lists the number of instructions executed for every byte of input for all algorithmic steps. Table 5 has one column for the processing carried out on the host CPU and one column for the processing carried out on d disk CPUs.

Table 4 Blocks transferred between pipeline stages

#	Alg. step	N_{in}	N_{mid}	N_{out}
(1)	Redist. R	$\|R\|$	$\|R\|$	$\|R\|$
(2)	Redist. S	$\|S\|$	$\|S\|$	$\|S\|$
(3)	GH Build	$\|R\|$	$\|R\|$	0
(4)	GH Probe	$\|S\|$	$\|S\|$	$\|R \bowtie S\|$
(5)	DH Build	$\|R\|$	0	0
(6)	DH Probe	$\|S\|$	$\|R \bowtie S\|$	$\|R \bowtie S\|$
(7)	AH Build	$\|R\|$	$(1-\rho)\|R\|$	0
(8)	AH Probe	$\|S\|$	$(1-\rho)\|S\| + \rho\|R \bowtie S\|$	$\|R \bowtie S\|$

Table 5 Number of instructions per byte of input

#	Alg. Step	Host CPU	d Disk CPUs
(1)	Redist. R	w_{part}	0
(2)	Redist. S	w_{part}	0
(3)	GH Build	w_{build}	0
(4)	GH Probe	w_{probe}	0
(5)	DH Build	0	w_{build}
(6)	DH Probe	0	w_{probe}
(7)	AH Build	w_{build}	ρw_{build}
(8)	AH Probe	$\frac{(1-\rho)\|S\|}{N_{\mathrm{mid}}} w_{\mathrm{probe}}$	ρw_{probe}

4.1 Redistribution steps

Regardless of redistribution policy, the processing cost incurred in the redistribution step is minimal: the hash value for every tuple is computed and the tuple is copied to the appropriate write buffer. The application parameters of the redistribution pipelines, are listed in rows 1 and 2 in Tables 4 and 5. The redistribution of either join relation incurs no data reduction, so the input size matches the output size, as the first two rows in Table 4 indicate. As far as processing load is concerned, we denote the number instructions executed per byte of input with w_{part}, which is a function of the tuple size and the join attributes size. As listed in Table 5, during the redistribution step, the host CPU carries out all processing and the disk-embedded CPUs remain idle. To expose pipeline bottlenecks, we replace the application parameters of the redistribution step in Eq. (1). Since the processing cost is low, data transfers dominate the pipeline. In particular, when few disks are employed, the read and write stages are the bottlenecks; for more disks, the aggregate disk bandwidth increases and the stages that ship data over the system bus become the bottleneck.

4.2 Grace hash join steps

GH comprises two redistribution steps and k_{GH} build and probe steps. Tables 4 and 5 list the application parameters for the build and probe pipelines of GH, in rows 3 and 4 respectively. In the build step, w_{build} instructions are executed per byte of input, that compute the hash value of each tuple and add it to a hash table. In the probe step, w_{probe} instructions are executed per byte of input, that compute the hash value of each input tuple, probe a hash table for matches, and compute the join result. The factors w_{build} and w_{probe}, which are significantly larger than w_{part}, change for different tuple size, join attributes, and join selectivity. In rows 3 and 4 of Table 5, we see that the host CPU executes all processing in the build and probe steps, and the disk-embedded CPUs have no load. GH requires k_{GH} iterations, so each pipeline operates approximately on $1/k_{\text{GH}}$ portion of the input relations. The build pipeline produces no output; only the probe pipeline outputs approximately $|R \bowtie S|/k_{\text{GH}}$ bytes. To facilitate comparisons between algorithms, we list the total amount of data transfers for all iterations needed by each algorithm in Table 4.

Based on Table 4, we note that the total output of all build and probe steps of GH is $|R \bowtie S|$, from rows 3 and 4, while the output of the two redistribution steps, from rows 1 and 2, is $|R| + |S|$, which is significantly larger. On the other hand, considering the same rows in Table 5, the processing load placed on the host CPU is w_{build} and w_{probe}, which is much greater from the redistribution load w_{part}. Depending on the actual application parameter values, we expect, based on Eq. (1), that for a large number of disks either the system bus or the host CPU will be fully utilized and will form bottlenecks.

4.3 Disk hash join steps

DH depends on k_{DH} build and probe steps, with the respective application parameters listed in rows 5 and 6 in Tables 4 and 5. The build and probe steps of DH differ from their GH counterparts in that the load is placed solely on disk-embedded CPUs. In this regard, we see in Table 5 that the load on the host CPU for GH, in rows 3 and 4, is identical to the load on the disk CPUs for DH, in rows 5 and 6. Furthermore, data shipment over the system bus is minimal in DH, because during the build step no data are sent to the host and during the probe step only $|R \bowtie S|$ blocks of the join result are transfered. Table 4 lists the total number of blocks transfered for all k_{DH} iterations of DH.

Comparing the application parameters of DH with those of GH, it is clear that the stress on the system bus is completely lifted. Unfortunately, this comes at the cost of placing all processing load on the disks, which are relatively less powerful than the host CPU. In this regard, we assume that the system parameters are such that the disk processing stage fails to process tuples at the full disk read rate; here, Eq. (1) indicates that the disk processing stage is the bottleneck. Although disk processing is the bottleneck, given enough disks the aggregate on-disk processing power exceeds the host CPU speed and DH outperforms GH. In this direction, we see that k_{DH} decreases for an increasing number of disks, hence k_{DH} can be either greater or less than the constant k_{GH}. Subsequently, in DH, increasing the number of employed disks lessens the join iterations, yielding much more parallelism.

4.4 Active hash join steps

In AH, ρ portion of the join evaluation occurs at the disks and $(1-\rho)$ on the host. The application parameters of the build and probe steps of AH, shown in the last two rows of Tables 4 and 5 respectively, are greatly affected by ρ. As far as the build step stage is concerned, only $\rho|R|$ part of the input $|R|$ is being processed, for the other $(1-\rho)|R|$ part is merely forwarded to the host. Similarly, during the probe step $(1-\rho)|S|$ part of the input is forwarded to the host, along with the part of the join result $\rho|R \bowtie S|$ computed on disks. In the same spirit, the load placed on the disk CPUs is lessened by factor ρ, compared to the DH case, as the last two rows indicate in Table 5. On the other hand, during the build step the host CPU executes w_{build} instructions per byte of input, as it processes all $(1-\rho)|R|$ blocks of input to build the host hash table. During the probe step, the host CPU uses only $(1-\rho)|S|$ blocks out of the total N_{mid} size of its input, thus the instructions incurred per byte of input are $\frac{(1-\rho)|S|}{N_{\mathrm{mid}}} w_{\mathrm{probe}}$. If join selectivity is low, that is $|R \bowtie S| \ll |S|$, we approximate $N_{\mathrm{mid}} = \rho|R \bowtie S| + (1-\rho)|S| \approx (1-\rho)|S|$, so the processing load in the probe step of AH is approximately w_{probe}.

Compared to GH and DH, AH makes use of all resources and also has the ability to gauge the stress placed on each resource. Adjusting the load distribution factor ρ lets AH behave similar to GH for $\rho \to 0$ or DH for $\rho \to 1$. We conjecture that AH outperforms GH and DH, because it features high levels of parallelism. Indeed, in AH (a) all available processing power in the system is used (b) larger hash tables are maintained because all available memory is used, and (c) extra tuning of load distribution allows for sophisticated bucket manipulation.

Ideally, both disk and host processing stages should last the same, thus we adjust ρ such that the following equations hold.

$$T_{\mathrm{disk}} = T_{\mathrm{host}} \longrightarrow \frac{\rho|S|w_{\mathrm{probe}}}{dS_{\mathrm{disk}}} = \frac{(1-\rho)|S|w_{\mathrm{probe}}}{S_{\mathrm{host}}} \longrightarrow \frac{\rho}{1-\rho} = \frac{dS_{\mathrm{disk}}}{S_{\mathrm{host}}} \tag{8}$$

In other words, disk and host resources should assume processing load proportional to their respective processing power. This load balancing scheme is easy to implement as it depends only on system parameters, but it should not be enforced at the cost of introducing hash table overflows or saturation of the system bus bandwidth. Thus, to avoid overflows we must respect the following inequality:

$$k_{\mathrm{AH}} \geq \left\lceil \max\left\{ \frac{\rho F|R|}{d|M_{\mathrm{disk}}|}, \frac{(1-\rho)F|R|}{|M_{\mathrm{host}}|} \right\} \right\rceil \tag{9}$$

which is illustrated in lines 1–3 in Algorithm 3. Avoiding bus bandwidth saturation is more complicated though. We do not want the pipeline stages that ship data over the system bus to be the bottleneck, so we derive from Eq. (1) that either $T_{\text{read}} + T_{\text{write}} \geq T_{\text{up}} + T_{\text{down}}$ or $T_{\text{disk}} = T_{\text{host}} \geq T_{\text{up}} + T_{\text{down}}$ must hold. Assuming that $|R \bowtie S| \ll |S|$, so that $N_{\text{mid}} \approx (1 - \rho)|S|$, and using the application parameters for AH from Tables 4 and 5 we rewrite the inequalities as follows:

$$T_{\text{read}} + T_{\text{write}} \geq T_{\text{up}} + T_{\text{down}} \longrightarrow B_{\text{bus}} \geq (1 - \rho) d B_{\text{read}} \tag{10}$$

$$T_{\text{disk}} = T_{\text{host}} \geq T_{\text{up}} + T_{\text{down}} \longrightarrow B_{\text{bus}} \geq \frac{S_{\text{host}}}{w_{\text{probe}}} \tag{11}$$

Inequality (10) demands that the $(1 - \rho)$ portion of the aggregate disk read rate, that is used for reading the data the host CPU operates on, does not exceed the bus bandwidth. In general this inequality holds, because as the number of disks d increases more processing is placed on the array of ASDs and ρ factor rises. Inequality (11) poses a complementary demand: the host CPU must process data faster than the system bus can transfer it.

Thus, for a specific system setting, Eq. (8) yields the optimum value for ρ which does not lead to bucket overflows due to inequality (9). If ρ is such that it leads to bus saturation, we can always resort to greater values to avoid this situation. The latter would certainly alter the optimality of the load balancing scheme. Fortunately, for a large range of pragmatic system parameter values that we evaluated, balancing load proportional to the respective processing power on host and disk CPUs does not lead to bus saturation.

5 Experimental results

We developed an extensive simulation engine that offers evaluation of the relative performance of joins on ASD architectures and assessment of the involved trade-offs. All join algorithms implemented operate on generated data, and actually materialize the join results. In the following, we give an overview of our simulation engine and continue with different simulation runs.

5.1 Simulation engine

Our discrete event simulator is built on the CSIM library [16] and assumes all system components the system consists of: disks, processors, memory modules, and buses. All system components are configurable to operate according to the system parameters listed in Table 1. Especially in the case of disks and buses, data are handled at a rate that is lower than the nominal by a bandwidth limitation factor, which models the latency of the actual devices. The bandwidth reduction factor is a random variable that follows the exponential distribution exponential distribution with mean set at 20% on average for disks and a 5% on average for buses. Thus, the nominal read rate of a disk is B_{read}, but in the simulator the read rate is varying over time and is on average $0.8B_{\text{read}}$. Our simulator is built to model join operator execution at the application level as there is very little to gain from very detailed disk and bus simulations [7, 19]. Especially for our test runs, input relations and buckets are read sequentially imposing minimal arm movement on the disks.

In our simulator, algorithmic steps are built using *actions*, which are interconnected via shared *structures*. Actions are interconnected so that they model the flow of data described

in the ASD model of Fig. 1. The creation and usage of a structures is translated in system resource usage. For example, a hash table is a structure that uses buffer space either on the host or on a disk. Another structure is a relation that uses disk space and bandwidth when asked to store or retrieve relation tuples. Overall, the structures available in the simulation engine are hash tables, relations, and various types of buffers. *Actions* are the dynamic entities in our simulation engine: they operate on structures and communicate among themselves. The pipeline processing terminates when all *actions* stop. An example of an *action* is a disk data reader, which models a stream of read requests to a disk. While a disk reader is active, the appropriate disk and memory entities are updated to track resource usage. Another example of an *action* is a hash table builder, that builds a hash table with tuples tuples from its input. During the execution of a hash table builder the incurred operations are logged in the appropriate CPU and memory entities. Our simulation engine provides reader and writer *actions*, hash table build and probe *actions*, and various data flow *actions* such as tuple copiers or tuple routers.

A typical algorithm step involves multiple data readers that input data to disk-resident buffers. Using these buffers, a set of *actions* that use disk resources process the tuples and output the results in different buffers. The on-disk processing output is copied from disks to to host-resident buffers, by means of copier *actions*. These *actions* use the system bus entity to log bandwidth usage. On the host, more *actions* operate on the data and place the results in appropriate buffers. All *actions* that operate in the host rely solely on the host CPU and memory. Finally, the contents of the result buffers in the host memory have to be stored in disk relations. A set of copier *actions* reads data from the host buffers, over the system bus, to on-disk buffers and a set of writer *actions* stores them in the appropriate relation. Note that all these *actions* operate concurrently, in the sense that while tuples are processed on the host, a subsequent block might as well be read from a disk and a host buffer may be flushed on another disk.

The join algorithms built for the simulation actually evaluate the join result in multiple steps. The simulation engine also accounts for for several implementation details that we omitted for brevity in the presentation of Section 4. For instance, our engine automatically reserves I/O buffers to accommodate the requisite I/O opertions.

5.2 Baseline experiment

The baseline simulation run considers a 2 GHz processor system with 100 MB of memory buffers allocated on the host for join processing. ASDs are equipped with a 150 MHz processor and 10 MB of memory. All disks read and write data at 20 MB/sec and 14 MB/sec respectively. High-end servers attach large number of disks to different disk controllers, so as the number of disks increases we assume more bus bandwidth is available. Specifically, the bus bandwidth is 100 MB/sec for every 8 disks, or 70 MB/sec if less than 8 disks are used. In the baseline run, we evaluate the join of a 200 MB relation (2,3 million tuples) with a 400 MB relation (3,5 million tuples) that produces a 4 MB relation (32,000 tuples). In our setup the hash table fudge factor F[2] is set to 1.2, making the entire outer relation hash table 240 MB in size. A hash table of this size cannot fit entirely in host memory, requiring more than one iteration from GH. Other algorithms may require less iterations, provided they use on-disk resources. For brevity, we designate ρ to be set to 0.5 as this value provides

[2] Refer to Section 3.2 for more on factor F.

an overall good compromise. In the following, we consider the baseline simulation results separately for each algorithmic step.

5.2.1 The redistribution phase

The redistribution phase involves a full read and a full write of both relation data. In contrast, all join phases together involve only one full scan of each relation and writing the result, which is significantly smaller. Thus, it comes as no surprise that the redistribution phase consumes as much as 60% of the total query time, for all algorithms we consider. Our experiments confirmed that a good portion of time in join evaluation is spent on the redistribution of relations. As expected, the bus is the main pipeline bottleneck, for it offers limited bandwidth compared to the collectively high read rate from all disks. Since the redistribution phases of GH, DH, and AH only differ in the way they partition the input relations, there is no performance difference among them.

5.2.2 The join phase

The number of iterations required for join evaluation depends on the memory used by every algorithm. In the baseline experiment, relying solely on 100 MB of host memory requires 3 iterations, while for more than 24 disks the aggregate on-disk memory grows to 240 MB which can hold the entire outer relation hash table. Unfortunately, no matter how many disks are used and whether they can maintain the whole outer relation hash table, DH cannot avoid the redistribution, because tuples need to be retrieved from matching buckets. The same holds for AH, but only for the portion of data that is processed on the disks; a custom redistribution phase could compute the join of $(1 - \rho)$ portion of the input on the host and concurrently redistribute the remaining ρ portion. Nevertheless, achieving a small number of join iterations assumes more data is being processed concurrently, in other words raises the level of parallelism. As the number of disks grows, more memory is available, more processing is carried out in a single step and the join evaluation time is shorter. For the baseline case, DH needs a single join iteration when more than 24 disks are available; AH needs one iteration for more than 16 disks. In contrast, GH requires 3 join iterations, no matter how many disks are used.

In Fig. 5 we present the total time spent on all probe steps for an increasing number of available disks in the baseline experiment, with the x-axis in logarithmic scale. The plot of the build pipelines execution time is similar and we omit it for brevity. We roughly identify three *regions* in the plot of the baseline experiment: the *host region* for 1–4 disks, the *disk-and-host region* for 8–16 disks, and the *disk region* for more than 32 disks. To facilitate the discussion on each region, we also present in Fig. 6 the component utilization for each algorithm with the number of disks placed in logarithmic scale on the x-axis.

In the host region, where only a few disks are available, the on-disk resources are not enough to make a noticeable difference over using them or not, thus GH dominates. For as little as 4 disks, the collective read rate is around 80 MB/s, which does not saturate the bus. As Fig. 6(a) indicates for GH, disks read blocks at their full read rate, while the host CPU and system bus are less than 50% utilized. In the same region, for DH, as Fig. 6(b) illustrates, the on-disk CPUs are too slow to keep-up with the rate data are read from the disks. Therefore, the on-disk processing capacity is soon exhausted and and yields the longer-lasting pipeline stage. Similarly, AH suffers due to $\rho = 0.5$ and so placing half of the load on disk resources. Properly adjusting ρ to values closer to 0 solves this problem for AH.

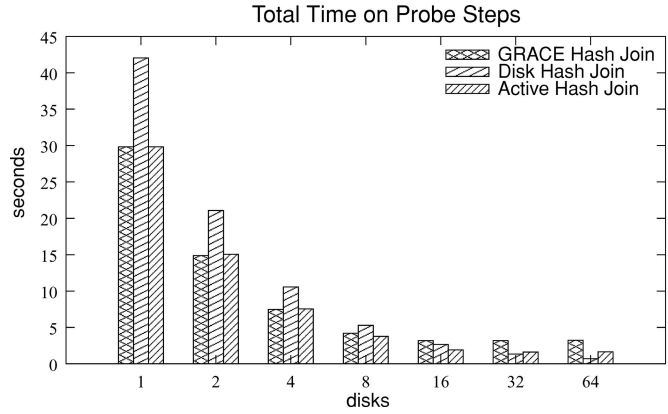


Fig. 5 Response time of all probe steps of GH, DH, and AH

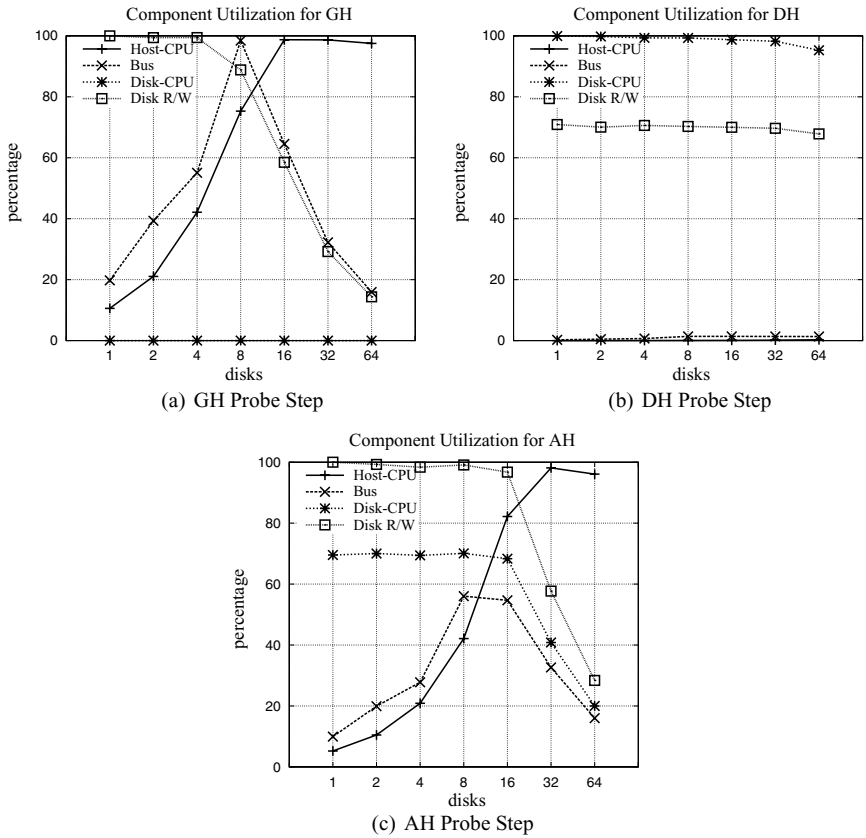


Fig. 6 Component utilization during the first probe step of algorithms GH, DH, and AH

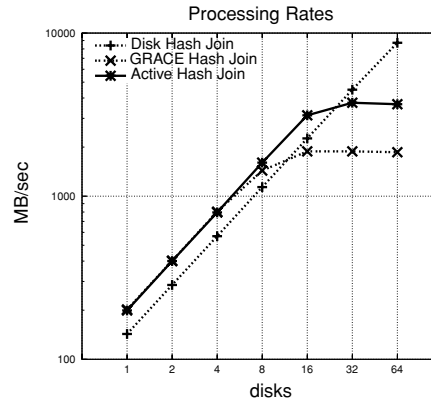
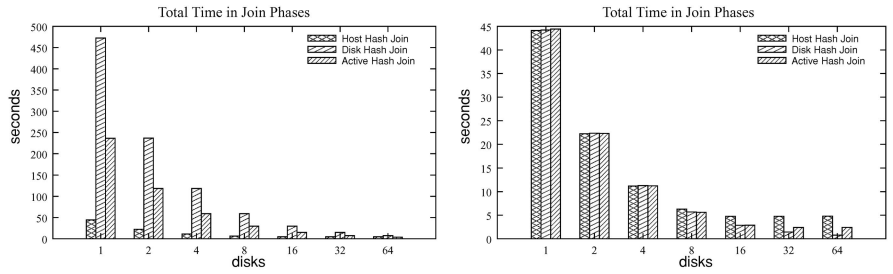


Fig. 7 Processing rates under GH, DH, and AH

In the disk region, where numerous disks are available, the on-disk aggregate resources outnumber those on the host and DH dominates this region. Interestingly, disk-embedded processors are too slow and fail to process data at the full disk read rate as presented in Fig. 6(b) for DH, where disk CPU utilization is close to 100%. However, for 32 or more disks, massive parallelism pays off for DH, as the whole outer relation hash table may well fit in the aggregate on-disk memory completing join evaluation in a single iteration, i.e. with just a build and a probe step. In addition, it turns out that relying on the host processing power does not scale well with the number of disks. In Fig. 7 we present the processing throughput rate versus the number of disks for all processing stages, with both axes set in logarithmic scale. Although AH outperforms GH, both algorithms reach a maximum throughput for more than 32 disks; in contrast, DH processing rate is proportional to d. Indeed, as Figs. 6(a) and (c) show for GH and AH respectively, the host CPU is the processing bottleneck in the disk region so we are better off using it for data gathering, rather than processing. Of course, AH overcomes this deficiency by properly adjusting the load balancing factor ρ to values closer to 1.

In the disk-and-host region, where a moderate number of disks is available, the performance of GH and DH is roughly equal. This is due to the processing resources on disks and host being of comparable magnitude. However, it is only AH that boosts performance by putting to use all system resources. Figure 7 verifies AH's lead in the disk-and-host region. The high processing rate is attributed to the two parallel processing stages, one running on disks and the other on the host, that share the total load. In addition, for this range of disks, AH uses all memory buffers available and computes the join result in one iteration, while DH requires 2 iterations and GH 3. In the disk-and-host region, GH fails to use the disk embedded CPUs and saturates the system bus, while DH fails to use the host CPU and under-utilize the system bus (Figs. 6(a) and (b)). AH overcomes both issues, by exploiting unused bus bandwidth to distribute $(1 - \rho)$ portion of the processing to the host, without introducing a bottleneck (Fig. 6(c)).

We established a 99% confidence interval for the baseline simulation for less than 20 runs with different random number streams. Furthermore, the simulation runs revealed the pipeline bottlenecks we expected. Finally, we verified the configuration of the load balancing factor, as AH outperformed other algorithms when the system parameters where such that $\rho \approx 0.5$.



(a) Low-speed disk CPU - 50 MHz. (b) High-speed disk CPU - 400 MHz.

Fig. 8 Time spent on join processing for slow (a) and fast (b) disk-embedded CPUs

5.3 System parameters

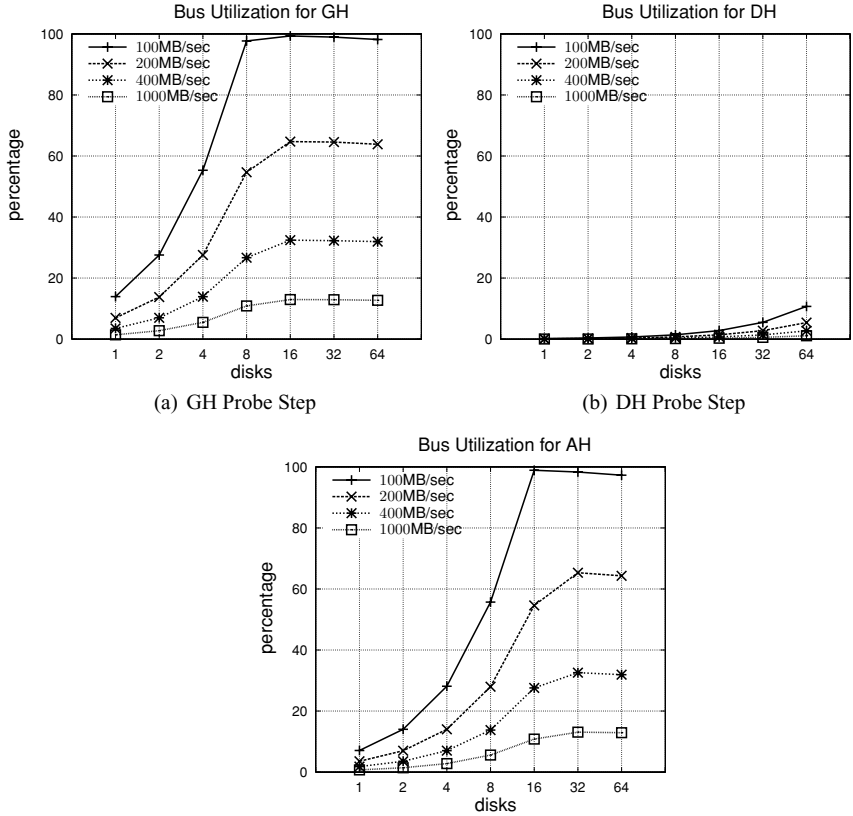
Altering the baseline experiment, we used different simulation settings to assess algorithm performance for various bus bandwidth sizes and different disk processing speeds. With these experiments we had the opportunity to confirm the role of processing power and memory size ratios between disks and host as the primary factors determining the region boundaries. In addition, model expressions offered sound causes for all algorithm deficiencies we encountered, making algorithm setup and tuning a straightforward process.

5.3.1 Disk processor speed

In order to investigate the role of the CPU speed ratio between disks and the host we kept all system parameters equal to the baseline case and varied the disk embedded processor speed. We assumed a wide range of disk processor speeds, starting from the slowest 50 MHz CPU, presented in Fig. 8(a), and stepping up to the fastest 400 MHz CPU, presented in Fig. 8(b). When low-speed disk embedded CPUs are used, disk processing pays off only for a large number of disks, while high-speed disk embedded CPUs favor processing on the disks. In this context we found that the boundaries of the regions observed in the baseline case changed as the disk processing speed changed: faster disk embedded processors contracted all regions but the disk region, slower disk embedded processors widen them. For example, in Fig. 8(a), the host region ranges between 1 and 32 disks, because the disk-embedded CPUs are slow, making DH/AH need more time to compute the join result than GH, which does not rely on ASDs. In contrast, in Fig. 8(b), where the ASDs have unusually high processing power, DH dominates when more than 8 devices are put to use. We successfully traced moving region boundaries to adjust load distribution factor ρ. We also found that the disk embedded memory size affected the region boundaries in a similar way.

5.3.2 Bus bandwidth

To examine the impact of the available bus bandwidth, we modified our baseline experimental setup so that the bus bandwidth was 100 MB/sec, 200 MB/sec, 400 MB/sec and 1000 MB/sec, regardless of the number of disks in the system. These settings represent different systems



(a) GH Probe Step (b) DH Probe Step

(c) AH Probe Step

Fig. 9 Bus utilization for different bus bandwidth for all algorithms during the first join phase

from low to high bus bandwidth, in contrast to the baseline setup where 8 disks resided on a single 100 MB/sec controller, modelling a high-throughput system. We found that the three regions we observed in the baseline case remain to a large extend unaffected from variations in the bus bandwidth. Lowering the bus bandwidth had a great impact on algorithm performance. Of course, low bus bandwidth damaged the redistribution pipelines mostly. As far as the build and probe pipelines are concerned, GH was most affected the most and DH the least; as expected, AH performance was a compromise between the performance of GH and DH. Figure 9(a) depicts bus utilization under GH, where the 100 MB/sec. bus is saturated for more than 8 disks. In contrast, DH that computes the join result on disks, utilizes less than 10% of the bus bandwidth, as presented in Fig. 9(b). AH saturates the slow 100 MB/sec. bus too, but only for more than 16 disks, as shown in Fig. 9(c). A high bus bandwidth on the other hand, places a heavy load on processors, which sooner or later cause bottlenecks. Notice, for example the bus utilization for GH in Fig. 9(a). As more disks are added to the system, increasing the aggregated read rate, one would expect the bus utilization to rise too. In practice though, the bus utilization is constant for more than 16 disks, because the host CPU cannot process data at these rates. Properly adjusting ρ in AH, with the help of our optimality criteria, overcomes these bottlenecks.

5.4 Join parameters

5.4.1 R and S relation sizes

We set the outer R and inner S relation size ratio to 1, 3, and 4 in order to examine the impact of input size to the pipeline operation. As the input size grew, the join evaluation took more time to complete, but apart from response time differences, the characteristics of pipeline execution didn't change. The input size of pipeline stages hardly affects model expressions such as resource utilization or throughput rate; in contrast the fraction of output over the input size does. Our experiments showed that all pipeline stages operated under the same throughput rates and all resources where utilized equally for varying input size.

5.4.2 Data schemata

Using the baseline setup, we fed GH, DH, and AH with joins of different selectivity. The model selectivity (in model terms $\frac{|R \bowtie S|}{|S|}$) of the baseline case is 0.01. We also experimented with low join selectivity, equal 0.001, and foreign key join selectivity, equal to 1.5. As expected, the small difference in the output size between the lower selectivity and the baseline cases hardly affected disk and bus bandwidth needs. On the other hand, the foreign key join constitutes the bus the primary bottleneck. The performance degradation is such that all join phases lasted as long as the redistribution phase. Selectivity above 1 drastically changes region boundaries as found in the baseline case; one can say that the host region exchanges positions with the disk region. It is clear that applications that produce more output than their input, as a foreign key join does, are not suited for ASD-based systems.

6 Conclusions and future work

Active Storage Devices offer an alternative to contemporary database systems for enhanced efficiency, by means of massive parallelism and reduced bus bandwidth consumption. In this paper, we propose Active Hash Join, a new equi-join operator algorithm that contrary to competing proposals harnesses resources from both the system host and constituent disks in an ASD configuration. We show in theory, and verify through detailed simulation, that our Active Hash Join approach offers high inter-operator parallelism, especially when the processing power between the disks and the host is comparable. Apart from Active Hash Join, we also consider two other GRACE-based join algorithms, namely GRACE Hash Join, that is oblivious to ASDs, and Disk Hash Join, which exclusively relies on ASD resources. We model algorithm execution with the intention to identify algorithm deficiencies or pipeline execution bottlenecks. As a result, we ascertain that GRACE Hash Join and Disk Hash Join are biased towards either the host or the disk resources, respectively. This is an important deficiency of GRACE-based join algorithms we consider, which are memory sensitive. In contrast, Active Hash Join has the ability to not only use all system resources, but also fine-tune the load balance between resources in an optimal way, based on system characteristics.

We argue that technology trends lead towards ASD systems for two reasons: first, arrays of disks are common nowadays to facilitate management of data; second, disk drives feature commodity chips to support sophisticated drive operations. As technology advances in these directions, it is only natural for database systems to become ASD-aware and overcome

shortcomings in I/O performance. In this context, query planning becomes an even more challenging problem, because there are more candidate join algorithms for a join operator. Inter-operator parallelism can reduce the overall time in well known left-deep query trees, but also opens new alternatives in exploring "bushy" query trees. Thus, in the future we plan to look into multi-query optimization in ASD environments, borrowing successful paradigms from distributed and parallel systems. Another research direction, in call for minimizing the cost of the data redistribution phase, is the application of ρ-based load balancing in join operators that use fragment and replicate techniques atop ASDs.

Acknowledgments We are grateful to the reviewers for comments that improved our presentation.

References

1. A. Acharya, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," in Procs. of the 8th Int. Conf. on ASPLOS, 1998, pp. 81–91.
2. H. Boral and D.J. DeWitt, "Database machines: An idea whose time has passed? A critique of the future of database machines," in Procs. of the 3rd International Workshop on Database Machines, 1983, pp. 166–187.
3. G. Chen, M.T. Kandemir, and A. Nadgir, "Compiler-based code partitioning for intelligent embedded disk processing," in Languages and Compilers for Parallel Computing (LCPC), 2003, pp. 451–465.
4. P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson, "RAID: High-performance, reliable secondary storage," ACM Computing Surveys, vol. 26, no. 2, pp. 145–186, 1994.
5. D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen, "The gamma database machine project," IEEE TKDE, vol. 2, no. 1, pp. 44–62, 1990.
6. A. Fournier and D. Fussell, "On the power of the frame buffer," ACM Transactions on Graphics, vol. 7, no. 2, pp. 103–128, 1988.
7. G. Ganger, B. Worthington, and Y. Patt, "The disksim simulation environment version 1.0 reference manual," Technical Report CSE-TR-358-98, Department of Computer Science and Engineering, University of Michigan, Ann Arbor, MI Feb 1998.
8. G.A. Gibson, D.F. Nagle, K. Amiri, J. Butler, F.W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka, "A cost-effective, high-bandwidth storage architecture," in ASPLOS-VIII: Proc. of the 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 1998, pp. 92–103.
9. G. Graefe, "Query evaluation techniques for large databases," ACM Computing Surveys, vol. 25, no. 2, pp. 73–170, 1993.
10. J. Gray, "Put everything in the storage device," Talk at the NASD Workshop on Storage Embedded Computing, 1998.
11. K. Keeton, D.A. Patterson, and J.M. Hellerstein, "A case for intelligent disks (IDISKs)," SIGMOD Record, vol. 27, no. 3, pp. 42–52, 1998.
12. K. Keeton, D.A. Patterson, and J.M. Hellerstein, "The intelligent disk (IDISK): A revolutionary approach to database computing infrastucture," Unpublished White paper, 1998.
13. M. Kitsuregawa and Y. Ogawa, "Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (SDC)," in Procs. of the 16th VLDB Int. Conf., Aug. 1990, pp. 210–221.
14. M. Kitsuregawa, H. Tanaka, and T. Moto-oka, "Application of hash to database machine and its architecture," New Generation Computing, vol. 1, no. 1, pp. 63–74, 1983.
15. G. Memik, M.T. Kandemir, and A. Choudhary, "Design and evaluation of a smart disk cluster for DSS commercial workloads," Journal of Parallel and Distributed Computing (JPDC), vol. 61, no. 11, pp. 1633–1664, 2001.
16. Mesquite Software, "CSIM 18 Simulation Engine," http://www.mesquite.com/, 2006.
17. E. Riedel, C. Faloutsos, and D. Nagle, "Active disk architecture for databases," Technical Report CMU-CS-00-145, Carnegie Mellon University, April 2000.
18. E. Riedel, G.A. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia," in Procs. of 24th VLDB Int. Conf., Aug. 1998, pp. 62–73.
19. C. Ruemmler and J. Wilkes, "An introduction to disk drive modeling," IEEE Computer, vol. 27, no. 3, pp. 17–28, 1994.

20. D.A. Schneider and D.J. DeWitt, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," in Procs. of the 1989 ACM SIGMOD Int. Conf., May/June 1989, pp. 110–121.
21. Seagate Technologies, "Cheetah hard drives family overview," http://www.seagate.com/products/enterprise/cheetah.html, 2005.
22. L.D. Shapiro, "Join processing in database systems with large main memories," ACM Transactions on Database Systems, vol. 11, no. 3, pp. 239–264, 1986.
23. M. Uysal, J. Saltz, and A. Acharya, "Evaluation of active disks for decision support databases," in Proceedings of the 6th IEEE International Symposium on High-Performance Computer Architecture, Toulouse, France, 2000.