Fragment and Replicate Algorithms for Non-Equi-Join Evaluation on Smart Disks

Vassilis Stoumpos
University of Athens,
15784, Athens, Greece
Email: stoumpos@di.uoa.gr

Alex Delis
University of Athens,
15784, Athens, Greece
Email: ad@di.uoa.gr

***Abstract*—The predicates in a non-equi-join can be anything but equality relations. Non-equi-join predicates can be as simple as an inequality expression between two join relation fields, or as complex as a user-defined function that carries out arbitrary complex comparisons. The nature of non-equi-join calls for predicate evaluation over all possible combinations of tuples in a two-way join. In this paper, we consider the family of fragment and replicate join algorithms that facilitates non-equi-join evaluation and adapt it in a *Smart Disk* environment. We use Smart Disk as an umbrella term for a variety of different storage devices featuring an embedded processor that may offload data processing from the main CPU. Our approach partially replicates one of the join relations in order to harness all processing capacity in the system. However, partial replication introduces problems with synchronizing concurrent algorithmic steps, load balancing, and selection among different join evaluation alternatives. We use a processing model to avoid performance pitfalls and autonomously select algorithm parameters. Through experimentation we find our proposed algorithms to utilize all system resources and, thus, yield better performance.**

***Index Terms*—database join, non-equi-joins, smart disks, active disks, fragment and replicate parallelism, array of disks**

I. Introduction

Join is arguably the most commonly used operator in database systems. The join operator is invoked with a set of *predicates* on two *relations*. The join predicates determine whether two tuples, one from each join relation, produce a match. The type of database join, equi-join or non-equi-join, is determined by the nature of the predicates involved. The presence of an equality join predicate, where a field from one join relation must be equal with a field from the other relation makes an equi-join. In general, an equi-join is efficiently evaluated with hash-based techniques and we do not consider it in this work [1]. Instead, we consider non-equi-joins, which is a wider class of join predicates that produce a join match if, for example, the two relation fields are not equal, or one is greater than the other. More complicated non-equi-join predicates follow some user-defined distance metric and ask for field values to be within some distance in this metric. In general, any user-defined function can serve as a join predicate, and therefore, no assumption can be made about two particular field values being a join match. To this end, the strong requirement of answering a non-equi-join query is that the join predicates have to be evaluated on all tuple combinations from the two relations. This leads to high volumes of data being retrieved from disks into memory buffers and processed. Due to limited memory space, multiple iterations over the same data is the common scenario.

Our goal is to efficiently evaluate non-equi-join over an array of *Smart Disks*. We use *Smart Disks* as a general term to describe the wide range of architectures where storage devices sport data processing capabilities [2]–[5]. The outline of a smart disk is presented in Figure 1. The interface controller in a smart disk manages two types of requests: traditional I/O requests and *smart* I/O requests. As presented in Figure 1, traditional I/O requests are serviced by copying blocks between buffers and disk platters. In contrast, smart requests extend the parameters of traditional I/O requests, so that specific data processing occurs as part of the request locally, inside the device. To this end, smart disks feature embedded processors and memory buffers in order to carry out the requested data processing. As an example, assume the smart disk interface controller receives a smart read request. In response, the smart disk reads the respective blocks from the disk platters into the device buffers, similar to what a traditional read request would call for. However, the blocks are not forwarded to the caller until they are processed with the instructions received as part of the smart request [2]. Then, the data processing output is forwarded to the caller to complete the request. Likewise, a smart write request asks for processing the write blocks first, and then, writing the outcome on the disk platters.

Smart disks follow the general architectural paradigm of "migrating" processing closer to data [3], [6]. Arguably, there are applications where pushing data processing on a single

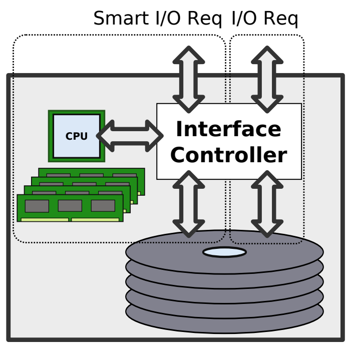

Fig. 1. A Smart Disk.

smart disk improves the overall performance. However, the potential for performance gain is greater in high-end data servers that are build atop arrays of smart disks. The rationale behind arranging smart disks in an array is twofold: diminish the size of data transfers over the disk interconnect and use the array as a massively parallel computing substrate. In the rest of this paper, we assume an array of smart disks is available and distinguish between *disk* and *host* resources, i.e. resources available either embedded in the disk array or on the main system. For simplicity, we assume that all array disks are attached on a common interconnect network, although in practice a hierarchy of bus channels is used.

Interestingly, the evaluation of non-equi-join algorithms on a smart disks array, utilizing the processing capacity of both host and disk resources is a problem with many operational parameters. On one hand, the specific system configuration has to be taken into account, so that, for example, when a high-speed system bus is available more aggressive parallelization is pursued. On the other hand, the characteristics of the specific join evaluation are also important. For example, "migrating" processing closer to data might not be beneficial, subject to the input to output size ratio [3]. If this is the case, the traditional approach of processing data away from where data is stored should be pursued. As a third example, an expensive user-defined join predicate that incurs high processing load, makes I/O load a secondary issue and calls for processing closer to data. Consequently, we incorporate all operational parameters in a processing model in order to estimate the cost of the possible alternatives. By means of this model, our approach autonomously selects the least expensive alternative, and thus, is well-suited for a database system that adapts to workload changes in a constantly-evolving system.

In Section II, we present our approach for non-equi-join evaluation on an array of smart disks that distributes the load between disk and host resources. We start with describing the details of partially replicating one relation, and continue with the description of possible alternatives for join evaluation. The details of load balancing and the cost estimates for different execution alternatives are considered in Section III. In Section IV we describe our experimentation setup and illustrate how our approach autonomously adapts to different hardware setups. We give an overview of related work in Section V and conclude with future work in Section VI.

II. Smart Disk ρ-FRJ

A. Initial Data Layout

We assume the two join relations R and S are block stripped on a smart disk array. In this sense, without assuming any particular tuple order, the first block from a relation is stored on the first disk, the second block on the second disk, etc. in a round-robin fashion. We note with R_i the part of relation R on disk i, so that $\cup_i R_i = R$ and $\cup_i S_i = S$. The blocks of a relation are evenly distributed to disks, so we can safely assume $|R_i| = |R|/d$, where $|R|$ is the number of blocks for relation R and d is the number of disks.

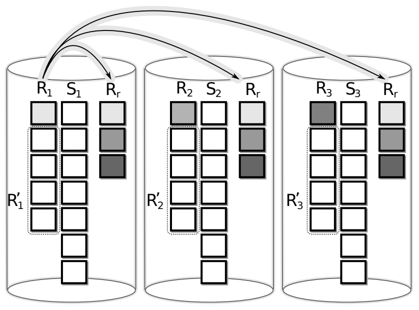

Fig. 2. Block placement after ρ replication on 3-disk array.

B. Traditional, non Smart Disk Approach

The traditional approach to the $R \bowtie S$ evaluation, which is smart disk oblivious, calls for Nested Loops Join (NLJ). In NLJ, blocks from the smaller relation, assume R, are *read* from all disks concurrently into main memory. If $|M|$ blocks are available in main memory, then if $|R| < |M|$ the relation is read in one step; otherwise, $k = \lceil |R|/|M| \rceil$ steps are required. When a read step completes, a *match* step follows. In the match step the entire S is read concurrently, to increase the overall read rate, and one I/O buffer is reserved in main memory per disk. . When a block from S is available in an I/O buffer, all combinations with the read blocks from R are considered and the respective join output is produced. After computation, the I/O buffer is cleared for the next block to be read in. In the general case, the total number of I/O blocks read is $|R| + k|S|$, where $k = \lceil |R|/|M| \rceil$.

C. Replication in ρ-FRJ

Our approach aims at using the processing capacity in smart disks in combination with the host resources. We view all smart disks as peer processing nodes, while the host represents a significantly more powerful node in the "network". The family of Fragment and Replicate Join (FRJ) algorithms is suitable for evaluating non-equi-join in distributed or parallel systems [7]. The skeleton of a FRJ algorithm that evaluates $R \bowtie S$ is as follows: the smaller relation is *replicated* to all processing nodes; then, each node computes the join between the local replica and the local *fragment* of the non-replicated relation. Consider the initial data placement, where disk i holds the fragments R_i and S_i of the join relations. If we chose to replicate R, then the entire relation should be copied over across all disks, since every disk serves as a processing node. In contrast, the host does not participate in replication, since the host has access to all disks at the same time. We respond to this asymmetry in resources and connectivity with our proposed ρ-FRJ method, that depends on the load balancing factor ρ. In the following, we describe how the load balancing factor alters the replication process. We present the alternatives to actually evaluate the join result with a given replication in the next section.

The load balancing factor ρ denotes the portion of input, and therefore processing load, that will be handled by the disk array, while the remaining $(1 - \rho)$ portion is managed by the host. To this end, ρ-FRJ creates on every disk a *partial replica* R_r of the first $\rho|R|$ blocks of R. Therefore, after replication, disk i carries the fragments S_i and R_i, and also the partial replica R_r of R. We note with R'_i the part of R_i that was not replicated, and we note with $R_f = \cup_i R'_i$ the part of R that was not replicated. In this sense, $R_i \cap R_r \neq \emptyset$ and $R'_i \cap R_r = \emptyset$. In Figure 2 we present the block layout after replication for a 3-disk example case. In the example we select R for replication and set $\rho = 0.2$, so the first block of all R_i is replicated resulting in a 3-block partial replica R_r on every disk.

D. Execution Alternatives in ρ-FRJ

After the ρ-based replication completes, join evaluation can start. In ρ-FRJ the original $R \bowtie S$ is split into several smaller joins of different sizes, respecting the asymmetry in the processing capacity of nodes. ρ-FRJ processes these smaller joins independently and concatenates the respective results to form the final $R \bowtie S$ outcome. More specifically, every smart disk i computes $R_r \bowtie S_i$ independently. At the same time, the host computes $R_f \bowtie S$, where R_f is the portion of R that was not replicated. The host and disk joins are evaluated with NLJ, using the host and disk resources respectively. Thus, ρ-FRJ follows the same execution pattern as NLJ, where k iterations of read and match steps are carried out. However, in ρ-FRJ multiple joins are involved, so the data flow is different from the one in NLJ. In Figure 3 we present the steps taken by ρ-FRJ. First, the partial replication step is carried out and then k iterations follow. In every iteration, the disk and host joins concurrently *read* blocks into disk and host memory buffers respectively. A read step is followed by a *match step* where the disk and host joins lookup in the disk and host buffers respectively for matches. In the following we discuss the synchronization issues that arise from concurrently executing disk and host joins and present alternatives to evaluate these joins. We defer the estimation of the number of iterations k until the next section.

In order to evaluate $R_r \bowtie S_i$ on all disks requires the entire $S = \cup_i S_i$ to be read. Therefore, ρ-FRJ *synchronizes* the on-disk joins with the host join, so that the blocks from relation S are read only once. To achieve this performance gain, ρ-FRJ forces all $d+1$ joins to take read, and respectively match, steps the same time. However, the number of steps required by NLJ to evaluate the host join might be different from the number of steps for the disk joins, depending on the available memory. In this case, ρ-FRJ forces all joins to perform the same number of iterations. Finally, to complete join synchronization, all joins must use the same relation when reading blocks into memory, and respectively, when looking for matches in memory. This means that after the decision for the relation to replicate has been made, ρ-FRJ has to choose between two alternatives.

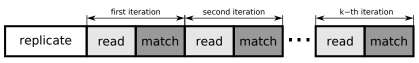

Fig. 3. Steps in ρ-FRJ.

(a) Concurrently read blocks from R_r and R_f in the read steps and then, at match steps, read all S blocks.
(b) Read blocks from S in the read steps and then, at match steps, concurrently read blocks from R_r and R_f.

In Figure 4, we present the data flow in our running 3-disk example for scenario (a), where the replicated relation is used in read steps. Here, the main memory can hold up to nine blocks, while the disk memory has room for three blocks. In main memory three of the blocks are reserved for I/O buffers, one for each disk, while one block is reserved for I/O buffers in disk-embedded memory. Disk i cannot hold the entire R_r to evaluate $R_r \bowtie S_i$ and similarly, there is no room on the host for all R'_i, in order to evaluate $R_f \bowtie S$. In this example, the on-disk join requires two read steps to complete join evaluation and so does the host join. At the first read step, two blocks from R_r are read into the smart disk buffers, while six blocks from R' are concurrently read from all disks into the main memory buffers. Then, at the match step, the entire S is read concurrently on all disks to compute matches (dashed lines). A block from S_i is first read in the I/O reserved buffer in disk i and the respective output for $R_r \bowtie S_i$ is produced. Then, the same block is forwarded to the appropriate I/O reserved buffer in the host to participate in $R_f \bowtie S$. Note that all on-disk buffers of R_r have the same content, and also that the entire S is read one block at a time and contributes to different joins in a pipelined fashion.

The alternative scenario (b) calls for using the fragmented relation in the read steps and is presented in Figure 5. Again, there is not enough room for the entire S to be read in neither the disk nor host memory. In this case, 4 iterations are necessary in order to evaluate the join. In every iteration, 2 blocks from S_i are read into smart disk i buffers, and at the same time, these 2 blocks are also forwarded to fill the main memory buffers. Then, smart disk i reads, one block at a time, the entire R_r and the entire R'_i concurrently into the appropriately reserved I/O buffers on disk and host (dashed lines). The blocks from R_r contribute in the join evaluation of $S_i \bowtie R_r$ on disk i, while the blocks from R_f contribute in the join evaluation of $S \bowtie R_f$ on the host. Note that the main memory buffer contains the same tuples as the union of buffers on all disks.

III. Processing Model

A. Processing Pipeline

We model the algorithmic execution in smart disk environments in terms of a *processing pipeline* [3], [8]. This pipeline comprises six *stages*: *reading* data from disks, *on-disk processing*, *copy to host* over the interconnect network, *host processing*, *copy to disks*, and *writing* back to disks. The pipeline stages operate in parallel and the overall pipeline speed is determined by the slower stage, termed the pipeline *bottleneck*. Therefore, in order to fully evaluate system performance we have to determine the amount of data conveyed between stages and the rate at which each stage consumes data. Note however, that one cannot model the entire join

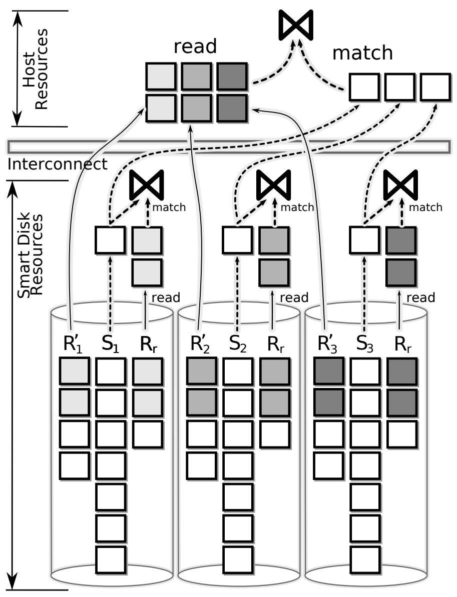

Fig. 4. Read replica R_r and R_f in memory. Use S to produce matches.

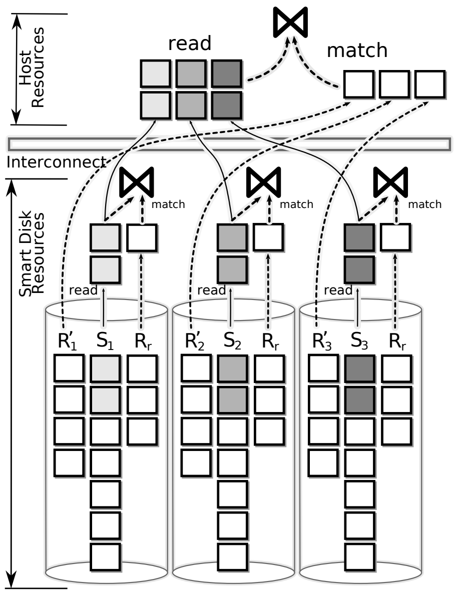

Fig. 5. Read S in memory. Use replica R_r and R_f to produce matches.

evaluation as a single pipeline, but only as a series of pipelines. In the ρ-FRJ case, we model each step as a different pipeline, as illustrated in Figure 3. In this sense, ρ-FRJ starts with a single replication pipeline. Then, a pair of read and a match pipelines follows. The read pipeline reads blocks from disks into memory buffers, and when the read pipeline completes, the match pipeline follows and produces matches. As presented in Figure 3, ρ-FRJ execution might involve more than one pair of read and match pipelines. In the following, we offer a detailed analysis of the trade-offs involved in the pipelines for replication, read, and match steps. The goal is to form pipelines in such a way so that bottlenecks are eliminated, or in other words, to make all pipeline stages last equally long.

The input to the replication pipeline is the first ρ fraction of blocks from the replicated relation. Each disk reads it's own share of blocks and broadcasts it on the interconnect network. At the same time, disks create a replica file and append to it all broadcast blocks they receive, including their own broadcast. If R is the replicated relation and ρ is the load balancing factor, then $\rho|R|$ blocks are read and broadcast on the interconnect, while $d\rho|R|$ blocks are written to disks. Thus, in the replication pipeline we expect the write stage to be the bottleneck because it receives d times larger input than all other stages, and has the lowest consumption rate. Especially in the replication pipeline, the pipeline bottleneck cannot be affected by changing the value of ρ.

B. Read and Match Pipelines

Consider from Section II-D, the two *execution alternatives* available to ρ-FRJ. After replication is complete, ρ-FRJ can use R_r and R_f as input to either the read pipeline (Figure 4) or the match pipeline (Figure 5). In both the read and match pipelines the block data flow resulting from having R_r and R_f as input is the same. The blocks from R_f are managed by the host, while the R_r blocks, which are read in parallel, are handled on-disk. In contrast, if S is the input to either a read or a match pipeline, the data flow has the S blocks first being processed on-disk and then being forwarded and further processed on host. In light of this observation, we consider the performance of a read or match pipeline, when either S or the pair of R_r and R_f are used as input.

When S is used as input to either read or a match pipeline, the read stage reads $|S|$ blocks and the smart disk array processes the same $|S|$ blocks. Then, the same number of blocks is transferred through the interconnect and processed on the host. The build pipeline produces no output, while the outcome of the match pipeline is $|R_r \bowtie S| + |R_f \bowtie S| = |R \bowtie S|$. Typically, in smart disk environments the output size is orders of magnitude less than the input size [3], so the write stage is not expected to slow down the pipeline. Except from the write stage, all stages have the same input size, so the bottleneck is the stage with the slower consumption rate. The processing load during the match pipeline is higher than during read pipeline, but still not high enough to cause a bottleneck. In practice, even expensive predicates, like distance between tuples in some metric or string matching, only require a few CPU instructions, so the processing stages exhibit high throughput rates. Therefore, the bottlenecks can only be formed in two places: either the read stage or the copy-to-host stage. In either case, we cannot change the bottleneck by affecting the load balancing factor ρ when S is used as input to either a read or a match pipeline.

We now consider the case where the read and match pipelines have R_r and R_f as their input. The important change here is that the total number of blocks read $d|R_r| + |R_f|$ is higher than $|R|$, because the same replicas R_r are read locally in all disks. In this sense, ρ-FRJ is expected to read more blocks than NLJ. However, it is not the total number of I/Os that determines performance, but the rate at which data are read. Recall that typically the system bus cannot sustain the aggregate read rate of a large number of disks. Consider NLJ where $|R|$ blocks are read from d disks and transferred through the system bus. The NLJ performance is limited by the copy-to-disk stage reading $|R|$ blocks at the system bus speed. Similarly, in ρ-FRJ the copy-to-host stage transfers $|R_f| = (1-\rho)|R| < |R|$ blocks, which means that the copy-to-host stage completes faster compared to NLJ. Clearly, as $\rho \rightarrow 1$ and $|R_f|$ becomes smaller, the load placed on the interconnect network will reduce. However, as $\rho \rightarrow 1$ the total number of read blocks increases rendering the read stage a bottleneck. In this context, we have the opportunity to adjust the value of ρ so that the performance bottleneck is eliminated.

ρ-FRJ increases ρ up to the point where the read stage lasts as long as the copy-to-host stage, and hence, is not the bottleneck. This means that while the system bus is busy transferring data at full speed, the underutilized disks read replica blocks in parallel. We use B_{read} and B_{bus} to denote the maximum disk read rate and the maximum interconnect network bandwidth.

$$\frac{d|R_r| + |R_f|}{dB_{read}} \leq \frac{|R_f|}{B_{bus}} \rightarrow \frac{d\rho + (1-\rho)}{(1-\rho)} \leq \frac{dB_{read}}{B_{bus}}$$
$$\rightarrow \rho \leq \max\left\{0, \frac{dB_{read} - B_{bus}}{dB_{read} + (d-1)B_{bus}}\right\} \qquad (1)$$

Although (1) guards against saturating the read stage, ρ-FRJ uses a complementary mechanism to synchronize the disk and host joins. This mechanism does not directly depend on the data flow in the pipeline so we devote the next section for presenting it.

C. Join Synchronization

As discussed in Section II-D, the reason for join synchronization is the performance gain from reading the fragmented relation in one pass. In practice, join synchronization requires all joins to use the same number of steps and execute them in tandem. As discussed in Section II-B, NLJ always uses the smaller relation in the read step in order to reduce the total number of I/Os. In ρ-FRJ, where NLJ is used to evaluate disk and host joins, the same rule applies, with the exception that the number of I/O requests is not the only factor to consider. Instead, the performance of the pipeline bottleneck is the main criterion. To this end, first we consider the two execution alternatives and provide the respective analytic formulae for the number of iterations necessary. Second, we present the expected number of blocks fed into pipeline stages and produce cost expressions based on the performance of pipeline bottlenecks.

When the fragmented relation S is used in the read steps, the number of iterations k_S is the maximum of iterations NLJ requires for the disk and host joins.

$$k_S = \max\left\{\left\lceil\frac{|S|}{d|M_d|}\right\rceil, \left\lceil\frac{|S|}{|M_h|}\right\rceil\right\} \qquad (2)$$

Similarly, when R_r and R_f are used in the read steps, disk i has to maintain $|R_r|$ blocks, while the host maintains $|R_f|$. Thus, ρ-FRJ uses the maximum number of iterations for the two cases according to NLJ.

$$k_R = \max\left\{\left\lceil\frac{\rho|R|}{|M_d|}\right\rceil, \left\lceil\frac{(1-\rho)|R|}{|M_h|}\right\rceil\right\} \qquad (3)$$

In order to enhance performance, ρ-FRJ adjusts ρ in (3) so that the number of iterations is reduced.

$$\left\lceil\frac{\rho|R|}{|M_d|}\right\rceil = \left\lceil\frac{(1-\rho)|R|}{|M_h|}\right\rceil \quad \longrightarrow \quad \rho \approx \frac{|M_d|}{|M_d| + |M_h|} \quad (4)$$

In light of this, ρ-FRJ has to combine (1) and (4); the first equation guards against the read stage becoming the bottleneck, while the second minimizes the number of iterations.

Note that so far we have not made the assumption that $|R| < |S|$. In fact, as the following cost expressions indicate, it is possible to decrease cost by replicating the larger relation. ρ-FRJ first considers either R or S for replication. Then, a second decision has to be made for which relation will be used in the read steps, making a total of four alternatives. The cost of the four alternatives is estimated before ρ-FRJ starts execution, so that the least expensive alternative is followed.

$$\begin{aligned}C = \min\{ \\ & C_{\text{repl}}(R,\rho_R) + C_{\text{read}}(R_r,R_f) + k_R C_{\text{match}}(S/d,S), \\ & C_{\text{repl}}(R,\rho_R) + C_{\text{read}}(S/d,S) + k_S C_{\text{match}}(R_r,R_f), \\ & C_{\text{repl}}(S,\rho_S) + C_{\text{read}}(S_r,S_f) + k_S C_{\text{match}}(R/d,R), \\ & C_{\text{repl}}(S,\rho_S) + C_{\text{read}}(R/d,R) + k_R C_{\text{match}}(S_r,S_f)\} \quad (5)\end{aligned}$$

The values of k_R and k_S are taken from (3) and (2). The cost of replication is governed by the write stages bottleneck: $C_{\text{repl}}(X,q) = \frac{q|X|}{B_{write}}$. The cost of reading relations D and H respectively to disk and host memory buffers is determined by the bottleneck in the copy-to-host stage: $C_{\text{read}}(D,H) = \frac{|H|}{B_{bus}}$. Similarly the cost for the match pipeline is $C_{\text{match}}(D,H) = \frac{|H|}{B_{bus}}$. Considering (1) and (4) we use a range of ρ values for (5), so we note with ρ_R and ρ_S the best values for replicating R and S respectively.

IV. Experimental Results

We conduct detailed simulation experiments that enable us to examine ρ-FRJ features and empirically evaluate the trade-offs involved. The nature of smart disks renders experimentation with prototypes a challenging problem on its own. In light of this, we resort to simulating all components of a smart disk architecture on par with prior efforts [2], [8], [9]. We developed our smart disk system simulator atop the CSIM discrete event simulation library. We simulate all smart disk system components: disks, processors, memory modules, and buses. In order to simulate entire smart disk systems we appropriately configure and combine components together. In our experiments, system components are configured to operate according to the parameters presented in Tables I and II. The implementation of join algorithms make use of the simulated system resources according to their needs. So at runtime, join algorithms use various simulated resources and simulator statistics are updated accordingly. Join algorithms operate on generated relation data and fully evaluate and store to simulated disks the join outcome.

TABLE I
SMART DISK PARAMETERS

Parameter	Value
Read Rate	80 MB/sec
Write Rate	60 MB/sec
CPU Speed	400 MHz
Memory	100 MB

TABLE II
SYSTEM PARAMETERS

Parameter	Value
CPU Speed	2 GHz
Memory	500 MB
System Bus	300 MB/sec
Disks on Bus	8

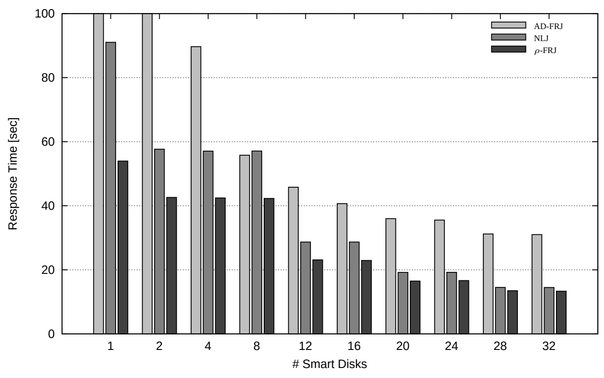

Fig. 6. Join evaluation time for different number of disks.

A. Baseline Experiment

In the baseline run, we evaluate the join of a 1 GB relation ($3, 5$ million tuples) with a 5 GB relation ($20, 1$ million tuples) that produces a 223 MB relation ($394, 000$ tuples). The join predicate accepts tuples that are within a distance threshold, in the euclidean metric space. In the following we consider three different algorithms. The first is our proposed ρ-FRJ that considers all execution alternatives and uses the load balancing factor ρ according to our description in the previous section. The second algorithm is NLJ that does not use the smart disk processing resources. Although we implemented NLJ as a separate algorithm, its behavior is identical to ρ-FRJ for $\rho = 0$. We actually compared the behavior of NLJ and ρ-FRJ with $\rho = 0$ as a sanity check for our testbed and verified that their results matched. The third algorithm we consider is *Active Disk FRJ* (AD-FRJ) [9]. The AD-FRJ algorithm fully replicates the smaller relation across disks, so its behavior is identical to ρ-FRJ for $\rho = 1$. As with NLJ, we implemented AD-FRJ separately and compared it's behavior with ρ-FRJ for $\rho = 1$, only to found they completely match. In all algorithm implementations, the necessary read and write buffers were allocated in disk and host memory as described in Section II. After I/O buffer allocation was complete, the remaining memory was available to the algorithms.

In Figure 6 we present the join evaluation time of the three algorithms we consider, in systems with a varying number of attached disks, ranging from 1 to 32 disks. Note that for every 8 disks, we use a separate 300 Mb/sec disk bus. In all cases, AD-FRJ lasts longer than the other algorithms. The main problem with AD-FRJ is that the cost of a full replication overshadows any benefit from fast join evaluation. In addition, even when only a few smart disks are employed AD-FRJ places the entire join evaluation load on them leaving the host processor underutilized. NLJ places all load on the host which delivers better performance than AD-FRJ. Still, the performance of NLJ is limited by the interconnect network bandwidth, which is over 98% utilized. Note that the NLJ response time is the same when the number of disks is 20 or 24 (3 disk buses), as well as when the disks are 28 and 32 (4 disk buses). In these two cases, although the number of disks increases the interconnect is saturated and becomes the pipeline bottleneck in the read and match pipelines.

In ρ-FRJ, the interconnect network is also the bottleneck in the read and match pipelines. However, ρ-FRJ exploits this fact and sends more I/O requests on the disks which are underutilized. During the NLJ read pipeline, disk utilization is less than 20%, when more than 8 disks are used, while ρ-FRJ utilizes 60% of disk throughput rate. In this context, ρ-FRJ performance on the host is comparable to the NLJ performance. However, ρ-FRJ uses the smart disk buffers, and thus, requires less iterations for join evaluation. In this setup, NLJ requires 3 steps, while ρ-FRJ requires 2 due to the extra smart disk embedded buffers. On average, NLJ lasts 25% more compared to ρ-FRJ.

B. Time Spent in Steps

In Figures 7 and 8 we present the total time spent in the replication, read, and match steps when, respectively, 8 and 16 disks are employed; in Table III we list in the corresponding component utilization. We see in both figures that although the replication cost of AD-FRJ is prohibitively high, the read and match steps are faster in AD-FRJ compared with the other two algorithms. The read and match steps in ρ-FRJ exhibit almost 100% disk utilization, which means that as more disks are added the steps will require less time to complete. However, putting more disks in the system will not decrease the replication cost, as AD-FRJ replicates $d|R|$ blocks.

In contrast, NLJ has no replication and places all processing on the host, which means voluminous data has to go through the interconnect. Therefore, the bus becomes the bottleneck as it is over 90% utilized. This is the window of opportunity for ρ-FRJ: while NLJ stalls the disks waiting for the bus transfers (disk utilization $<$ 13%), ρ-FRJ places more read requests on disks (disk utilization $>$ 30%), and thus, allows for more data to be processed at no extra cost. Note in Figures 7 and 8 that the read steps in ρ-FRJ and NLJ last the same, although ρ-FRJ reads more data. As more data is read during the read step, the less iterations are necessary for join evaluation. In this particular case, where ρ-FRJ requires 2 iterations, while NLJ requires 3, the match steps in ρ-FRJ are 50% faster than the match steps of NLJ. Overall, NLJ is 35% slower than ρ-FRJ in the 8-disk case and 25% slower in the 16-disk case.

As more disks are put to the system the interconnect bandwidth increases, so the match steps last shorter and the performance gain is limited. However, the available host memory to algorithms decreases as more disks are added, because the number of I/O buffers increases. Constantly adding more disks will eventually lead NLJ to more iterations that will rapidly drop its performance.

TABLE III
COMPONENT UTILIZATION FOR 8 AND 16 DISKS

		8-disks				16-disks			
Algorithm	Step	Disk	Disk-CPU	Bus	CPU	Disk	Disk-CPU	Bus	CPU
NLJ	repl	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
NLJ	read	11.64%	0.00%	94.42%	8.02%	11.84%	0.00%	92.05%	15.69%
NLJ	match	12.79%	0.00%	99.79%	14.65%	12.85%	0.00%	99.55%	29.22%
AD-FRJ	repl	99.62%	1.68%	16.36%	0.00%	99.55%	1.46%	8.25%	0.00%
AD-FRJ	read	79.48%	22.77%	0.00%	0.00%	78.55%	22.22%	0.00%	0.00%
AD-FRJ	match	75.70%	38.64%	0.00%	0.00%	75.92%	38.16%	0.00%	0.00%
ρ-FRJ	repl	98.80%	1.66%	16.20%	0.00%	98.57%	1.45%	8.17%	0.00%
ρ-FRJ	read	30.18%	5.07%	97.14%	8.27%	47.75%	9.84%	94.48%	16.09%
ρ-FRJ	match	13.32%	6.78%	99.81%	15.13%	13.56%	6.76%	99.60%	30.18%

C. Load Factor ρ

In Figure 9 we present the execution time for ρ-FRJ in the 8-disk and 16-disk system, with a fixed load balancing factor ρ and a fixed execution alternative. More specifically, we vary the ρ value from 0 to 1 and force ρ-FRJ to replicate the smaller relation and use the replica during the read step. This is the execution alternative that ρ-FRJ chooses when self-adjusting the ρ value. Note that the x-axis in Figure 9 is in logarithmic scale. We see that for both the 8-disk and the 16-disk systems a load factor less than 0.003 does not reduce the join evaluation time. However, for ρ between 0.05 and 0.2 the evaluation time drops by 30%. Then, for greater values of ρ the performance significantly deteriorates. Note that NLJ performance is on the leftmost part of the graph, for $\rho = 0$, while the AD-FRJ performance on the rightmost, for $\rho = 1$. Note that, the AD-FRJ performance in Figure 9 is much worse compared to Figures 7 and 8. This is due to ρ-FRJ using a fixed execution alternative in this experiment. Should all four execution alternatives were enabled, ρ-FRJ would select to replicate the smaller relation, but then use it in the match step, which is the case in Figures 7 and 8.

This indicates that, equations (1) and (4) are guiding ρ-FRJ to the right direction. First, ρ-FRJ restricts ρ, according to (1), to be $\rho < 0.124$ for 8 disks, and $\rho < 0.169$ for 16 disks. In this value range for ρ, blocks are read into disk buffers "for free", as long as the system bus is saturated. Second, ρ-FRJ adjusts ρ according to (4) so that the number of iterations is minimum. For the 8-disk system $0.030 < \rho < 0.192$, while for the 16-disk system $0.062 < \rho < 0.192$. We conclude that the two equations serve as good heuristics to guide ρ-FRJ to autonomously adjust the join evaluation load. Note however, that the underlying model is flexible enough to cover a much wider case of system and application parameters.

V. RELATED WORK

The Smart Disk architecture favors filter-type algorithms, therefore applications like data mining, multimedia applications, data warehousing, large string database search, and decision support systems are well suited for Smart Disks [2], [3], [10], [11]. A different line of research uses smart disks to transparently reorganize the disk data structures used by a file system or a database system, based on information about how these systems use their data [12], [13]. More radical approaches push to disks the file system implementation [14], security and user authentication [15], or convert disks from block stores into object stores [16]. In addition, Smart Disks improve the performance of database systems, especially in the case of filter-type operators, like table scan, sorted batch construction in merge sort, and top-k queries [5], [9], [17], [18]. However, binary matching operators, like join, pose different problems due to the large number of comparisons between the inputs. Subject to the available buffer space, joins are completely different from filter-type algorithms in that joins scan their input multiple times and flush to disk potentially voluminous intermediate data [1].

Parallel join algorithms fall in two categories according to the technique for data reorganization: *symmetric partitioning* and *fragment and replicate* [7]. The symmetric partitioning approach on a smart disk array hashes input relations in bucket pairs that are processed independently on smart disks [9]. A more general approach considers both disk and host resources in join evaluation [8]. Our algorithm is different in that it is capable to evaluate any join predicate, not only equality predicates. Nevertheless, ρ-FRJ borrows the ρ load balancing scheme [8] in order to achieve high component utilization. Fragment and replicate techniques are the only choice in cases of non-equi-join evaluation. AD-FRJ that falls in this category replicates the smaller relation across all disks, so that independent on-disk joins can follow [9]. In addition, a variation of AD-FRJ employs bit-vector filters to avoid replication of tuples that are guaranteed to have no match, but this variation only applies to equi-join predicates. Our proposed ρ-FRJ, which falls in this category is a generalized version of AD-FRJ that performs partial replication and deals with synchronization issues, alternative execution plans and load distribution.

We use the term *Smart Disks* as an umbrella term for different proposals in the literature. *Active Disks* [2], [3] and *Intelligent DISKs* [4], constitute one realization of the Smart Disk architecture, where hard disk drives come equipped with embedded general-purpose programmable processors and extra memory dedicated to data processing. Another Smart Disk architecture example is *Network-Attached Secure Disks* [15], and *Clusters of Smart Disks* [5]. Here, disks are attached to some device that exports a network interface. This network interface is a mediator that features sophisticated protocols that, either directly or not, forward requests to disks. Clearly, the traditional storage device interface of read/write requests is

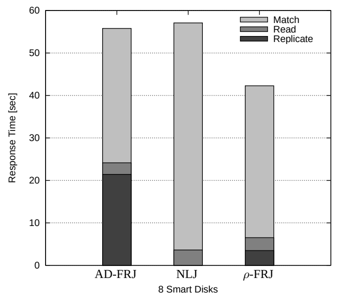

Fig. 7. Time of steps for 8 disks.

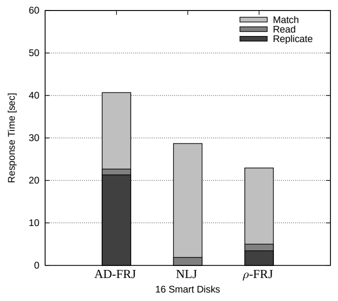

Fig. 8. Time of steps for 16 disks.

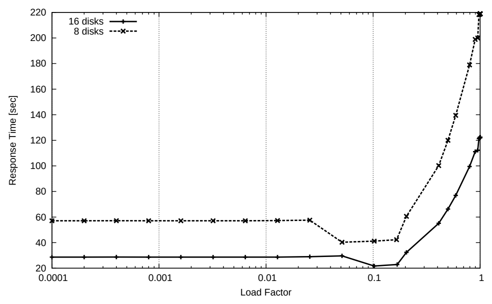

Fig. 9. Total time for different load balancing factor ρ.

inadequate to support application-pertinent on-disk processing. One interface extension models processing as a chain of filters, or *disklets* [2] that are installed on disks. In this context, applications are asked to specify the input source, the chain of disklets for processing, and the output target. A second extension assumes devices store objects (as opposed to blocks) with methods that the caller can invoke [19]. This extension is more radical as it expects applications to change their viewpoint from storing data to storing objects, with types, methods, and potentially inheritance properties from other objects. In this environment, applications start data processing by means of a method call, that might lead to other method calls, forming a complex invocation graph.

VI. Conclusions

We use fragment and replicate techniques to evaluate non-equi-joins in parallel on Smart Disks arrays. Non-equi-joins cover a wide range of user-defined join predicates, for which the database management system cannot perform any optimization. In this work for example, we consider a join predicate that matches tuples within a distance threshold in the euclidean metric space. This type of query cannot be evaluated by hashing tuples or partitioning in ranges of values. In this context, our proposed ρ-FRJ performs a partial replication of the first ρ fraction of one of the join relations. Partial replication enables ρ-FRJ to harness disk and host resources, and hence, evaluate join faster than NLJ, that uses only host resources, and AD-FRJ, that uses only disk resources.

However, partial replication introduces challenging problems. The processing load has to be balanced between disks and host without forming performance bottlenecks. In addition, disk and host joins have to be synchronized, so that the non-replicated relation is read in one pass. This opens four different execution alternatives to ρ-FRJ, with no alternative being the best choice in all cases. To this end, ρ-FRJ follows a detailed processing model that adjusts ρ without forming bottlenecks, evaluate the cost of alternatives, and equally place load on disks and host.

References

[1] L. Shapiro, “Join Processing in Database Systems with Large Main Memories,” *ACM TODS*, vol. 11, no. 3, pp. 239–264, 1986.

[2] 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.

[3] E. Riedel, G. A. Gibson, and C. Faloutsos, “Active Storage for Large-Scale Data Mining and Multimedia,” in *Procs. of 24th VLDB Int. Conf.*, August 1998, pp. 62–73.

[4] K. Keeton, D. Patterson, and J. Hellerstein, “A Case for Intelligent Disks (IDISKs),” *SIGMOD Record*, vol. 27, no. 3, pp. 42–52, 1998.

[5] G. Memik, M. T. Kandemir, and A. Choudhary, “Design and evaluation of smart disk architecture for dss commercial workloads,” *International Conference on Parallel Processing*, p. 335, 2000.

[6] J. Gray, “Put Everything in the Storage Device,” Talk at the NASD Workshop on Storage Embedded Computing, 1998.

[7] G. Graefe, “Query Evaluation Techniques for Large Databases,” *ACM Computing Surveys*, vol. 25, no. 2, pp. 73–170, 1993.

[8] V. Stoumpos and A. Delis, “Grace-based joins on active storage devices,” *Distributed and Parallel Databases*, vol. 20, no. 3, pp. 199–224, November 2006.

[9] E. Riedel, C. Faloutsos, and D. Nagle, “Active Disk Architecture for Databases,” Carnegie Mellon University, Tech. Rep. CMU-CS-00-145, April 2000.

[10] W. Hsu, A. Smith, and H. Young, “Projecting the performance of decision support workloads on systems with smart storage (smartstor),” *Proceedings of the Seventh International Conference on Parallel and Distributed Systems (ICPADS’00)*, p. 417, 2000.

[11] R. Chamberlain, R. Cytron, M. Franklin, and R. Indeck, “The Mercury system: Exploiting truly fast hardware for data search,” *Int. Workshop on Storage Network Architecture and Parallel I/Os*, pp. 65–72, 2003.

[12] M. Sivathanu, V. Prabhakaran, F. Popovici, T. Denehy, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, “Semantically-smart disk systems,” in *Proceedings of the FAST ’03 Conference on File and Storage Technologies*. USENIX, March 31 - April 2 2003.

[13] M. Sivathanu, L. Bairavasundaram, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, “Database-aware semantically-smart storage,” in *Proceedings of the FAST ’05 Conference on File and Storage Technologies*. USENIX, December 13-16 2005.

[14] H. Lim, V. Kapoor, C. Wighe, and H. David, “Active disk file system: A distributed, scalable file system,” *Proceedings of the Eighteenth IEEE Symposium on Mass Storage Systems and Technologies*, 2001.

[15] G. Gibson, D. Nagle, K. Amiri, J. Butler, F. 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 8-th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1998, pp. 92–103.

[16] M. Mesnier, G. Ganger, E. Riedel, and C. Mellon, “Object-based storage,” *Communications Magazine, IEEE*, vol. 41, no. 8, pp. 84–90, 2003.

[17] S. Chiu, W. Liao, A. Choudhary, and M. Kandemir, “Processor-embedded distributed smart disks for I/O-intensive workloads: architectures, performance models and evaluation,” *Journal of Parallel and Distributed Computing*, vol. 64, no. 3, pp. 427–446, 2004.

[18] M. Uysal, A. Acharya, and J. Saltz, “Evaluation of active disks for decision support databases,” *6th Int. Symp. High-Performance Computer Architecture*, pp. 337–348, 2000.

[19] G. Gibson, D. Nagle, W. II, N. Lanza, P. Mazaitis, M. Unangst, and J. Zelenka, “Nasd scalable storage systems,” *Proceedings of the USENIX99 Extreme Linux Workshop*, 1999.