Strategies for Using Additional Resources in Parallel Hash-based Join Algorithms*

Xi Zhang†, Tahsin Kurc†, Tony Pan†, Umit Catalyurek†,
Sivaramakrishnan Narayanan†, Pete Wyckoff‡, Joel Saltz†

† Dept. of Biomedical Informatics
The Ohio State University
Columbus, OH 43210
{xizhang,kurc,tpan}@bmi.osu.edu
{umit,krishnan,jsaltz}@bmi.osu.edu

‡ Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
pw@osc.edu

Abstract

Hash-based join is a compute- and memory-intensive algorithm. It achieves good performance and scales well to large datasets, if sufficient memory is available to hold the hash table and the distribution of computing load across nodes is balanced. In this paper, we compare three adaptive algorithms that start with a partitioning of the hash table across a group of nodes and expand during the hash table building phase to additional resources, when memory on a node is used up. The split-based algorithm partitions the hash table range assigned to the node, on which memory is full, into two segments and assigns one of the segments to a new node in the system. The replication-based algorithm replicates the hash table range on a new node. The hybrid algorithm combines the first and second strategies in order to address each strategy's short comings. We perform an experimental performance evaluation of these algorithms on a PC cluster. Our results show that among the three algorithms, in most cases the hybrid algorithm either performs close to the better of the two or is the best algorithm.

1 Introduction

This paper is concerned with efficient execution of the equi-join operation, which is one of the most common database operations. Several techniques have been developed for performing join operations efficiently [10, 25]. Among many methods, hash-based join algorithms are considered to be fast and easy to implement. Hash-based joins on large datasets are good candidates for parallelization as they require high computing power and large memory space to maintain intermediate data structures (i.e., the hash table). A number of parallel algorithms have been developed [7, 14, 25, 26, 29, 32]. In these algorithms, the hash table is partitioned into buckets, and the buckets are assigned to processors. Data skew is a challenging problem, in which some of the buckets receive more join elements to be processed, resulting in computational load imbalance. Algorithms have been proposed and evaluated that adaptively partition the hash table and distribute the buckets and computations to achieve computational load balance [4, 8, 12, 13, 17, 18].

Most of these algorithms focus on improving performance assuming a static parallel configuration with in-core buckets. Bucket overflow, in which an in-core bucket becomes full, is an equally important problem. Hash based joins scale well to larger datasets when there is enough memory space to maintain the hash table. In some cases, the size of a join relation may not be known a priori or may not be estimated accurately. Consider a query that selects a subset of two relations using user-defined filters and performs a join operation on the selected elements. In order to estimate the memory requirement, the relations can be sampled and the select operations can be applied on the sampled elements. However, sampling may be expensive, if the user-defined filters are expensive, and may not give accurate estimates. Moreover, in dynamic environments where resources (i.e., the set of nodes in the environment) can be shared by multiple queries submitted to the system, a query may have to start execution using a small number of nodes and dynamically allocate more resources as needed and as they become available.

*This research was supported in part by the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408), #EIA-0121177, #ACI-0203846, #ACI-0130437, #ANI-0330612, #ACI-9982087, Lawrence Livermore National Laboratory under Grant #B517095 (UC Subcontract #10184497), NIH NIBIB BISTI #P20EB000591, Ohio Board of Regents BRTTC #BRTT02-0003.

Algorithm 1 The basic hash-based join algorithm.

```
Allocate HashTable[0...N-1].
for each hash table position $h$ do
    HashTable[$h$] $\leftarrow \{\}$.
for (each element $r$ in R) do
    $h \leftarrow$ HashFunction($r$.join_attribute).
    HashTable[$h$] $\leftarrow$ HashTable[$h$] $\cup \{r\}$.
for (each element $s$ in S) do
    $h \leftarrow$ HashFunction($s$.join_attribute).
    for (each element $r$ in HashTable[$h$]) do
        if ($r$.join_attribute == $s$.join_attribute) then
            Output r and s.
```

In this paper we present and evaluate three algorithms, referred to here as *Expanding Hash-based Join Algorithms*, to avoid bucket overflow. The split-based algorithm partitions the hash table range assigned to the node, on which memory is full, into two segments and assigns one of the segments to a new node in the system. The replication-based algorithm replicates the hash table range on a new node. The third algorithm is a hybrid approach that combines the first and second strategies. Our experimental evaluation of the algorithms on a PC cluster shows that the replication-based algorithm performs better than the split-based algorithm when data distribution is highly skewed and/or the hash table has to be build using the larger of the two relations. Otherwise, the split-based algorithm achieves better performance, as it reduces the communication overhead in the probing phase of the join operation. Among the three algorithms, we observe that the performance of the hybrid algorithm generally is close to the better of the other two algorithms or better than both.

2 Overview: Hash-based Equi-Join Algorithm

In this section we briefly describe the basic hash-based join algorithm. Assume that we have two relations **R** and **S**. The basic hash-based join algorithm consists of two phases; the hash table building phase and the hash table probing phase. One of the relations is chosen for building the hash table, while the other relation is scanned for probing the hash table. The algorithm is shown in Algorithm 1 – we assume that relation **R** is used to build the hash table.

In the hash table building phase, a hash table is allocated in memory. As is seen from the figure, a hash function is applied on the join attribute of each element in relation **R** and the element is inserted into the corresponding hash table position. In the hash table probing phase, the same hash function is applied on the join attribute of each element in relation **S**. The elements that have been stored in the corresponding hash table position during the hash table building phase are searched for matches and matching pair of elements are output.

A simple parallelization of the algorithm partitions the hash table into P buckets, where P is the number of processors in the system. When **R** is processed, a hash function is applied to the join attribute of each element. According to the hash value, the element is sent to the processor to which the corresponding bucket is assigned. In the probing phase, the elements of **S** are partitioned among the processors using the same hash function.

A variety of algorithms have been developed to carry out join operations when the hash table does not fit in memory. The basic out-of-core join algorithm partitions the hash table into N buckets so that each bucket fits in memory. As in the in-core join algorithm, relation **R** is partitioned among the buckets using a hash function. The buckets are written to disk. In the second phase, relation **S** is scanned and partitioned into buckets using the same hash function. These buckets also are stored on disk. In the third phase, the basic in-core hash-based join algorithm is applied to each pair of buckets.

3 Related Work

Efficient execution of join operations is a widely studied topic in database research [3, 6, 7, 8, 9, 16, 20, 21, 23, 24, 25, 26, 29, 31, 32, 33]. In this section we review some of the previous work on parallel join algorithms.

DeWitt *et al.* [8] present four algorithms to improve load balance in joins on parallel machines, when the data is skewed. Their approach is to sample the relations that will be joined to predict the amount of skew in the data and choose the most appropriate algorithm to employ. The predicted amount of skew is also used to determine a mapping of work to processors to achieve load balanced execution. Li *et al.* [18] examine techniques to handle data skew for sort-merge join algorithms. Their goal is to minimize the disk overhead when skew is low and execute the join operations efficiently when skew is high. Lerner and Lifschitz [17] present a survey of various load balancing techniques employed in parallel join algorithms. Bamha and Hains [4, 5] present a data redistribution algorithm for equi-join operations on distributed memory machines. The algorithm uses histograms to compute the frequency of join attribute values in a relation. The relation is redistributed based on the frequency counts such that frequent values are distributed evenly among processors. Imasaki and Dandamudi [13] propose an algorithm that is based on the master-slave paradigm. The master process dynamically sends input data to slave processors and slave processors perform local join operations.

Most of the previous work in parallel join algorithms has focused on static, incremental, and dynamic methods for achieving computational load balance on shared-memory and distributed-memory machines. These algorithms do not address performance issues when the hash table does not fit in memory. Our approach differs from the previous work in that the proposed algorithm attempts to avoid buffer overflow by dynamically allocating additional resources.

The algorithm presented by Amin *et al.* [2] uses additional resources when more memory space is needed. When the available memory space is exceeded, the algorithm allocates a new processor and chooses one of the hash table buckets, assigned to the current set of processors, to partition. The elements of the bucket are redistributed between the new processor and the original processor using a new hash function. The split-based algorithm presented in this paper is based on the algorithm developed by Amin *et al.* The other algorithms differ from this algorithm. In the replication-based algorithm, the hash value range assigned to a hash table bucket, the size of which exceeds the available memory, is replicated on the new processor. This avoids the communication overhead because of redistribution of bucket entries. The hybrid algorithm combines the two algorithms.

The *Distributed Hash Table* (DHT) is one of the core substrate in many peer-to-peer systems to facilitate searching [1, 11, 22, 27, 28, 30]. Decentralization, scalability and availability are key requirements of a peer-to-peer system. DHT proposals, including CAN [27], Chord [30], Pastry [28], and Tapestry [11], aim to address those requirements. In some aspects, DHT implementations are similar to the implementation used in the split-based join algorithm described in this work; hence, they could be employed in the implementation of the split-based join algorithm. However, since the main goal of the DHTs is to provide scalability and availability even beyond thousands of peers, a hash table lookup operation typically requires $O(\log n)$ steps. For the problem described in this work, there are relatively a small number of peers (i.e., processing nodes in the system), peers are tightly coupled, and they are not transient. Hence a much simpler mechanism such as a simple hash-based mapping can be employed to identify the node containing the relevant portion of the hash table in $O(1)$ time.

4 Expanding Hash-based Join Algorithms

In this section , we present three *Expanding Hash-based Join Algorithms* (EHJAs), which are designed to use additional hosts to maintain the hash table in memory. The algorithms are similar to the basic hash-based join algorithm in that they consist of a hash table building phase followed by a hash table probing phase. One of the relations is used to build the hash table. The hash table is probed using the second relation to find joining (matching) elements. The basic idea behind the EHJA is to allocate a new join node and use its resources when a join node runs out of memory during the hash table building phase. In this paper, we assume that the joining elements are either written to disk or forwarded to the client or to the next stage in the query plan. Hence, no bucket overflow is assumed to occur in the probing phase.[1]

The algorithms start with a set of join nodes. The hash table range is partitioned into buckets and each join process is assigned one of the buckets. Each bucket is associated with a disjoint subrange of hash values. In an environment where resources can be shared by other applications, one of the objectives is to minimize execution time without wasting resources. Allocating a large number of nodes would result in high performance since only few nodes, if any, would likely run out of memory during the join processing. However, this also decreases the availability of resources to other applications executing in the environment. In this paper, we experimentally evaluate the impact on performance of the number of initial join nodes. In future work, we plan to examine algorithms for efficient selection of the initial set of join nodes.

4.1 System Architecture

The system architecture consists of three components: *a scheduler*, *data sources*, and *join processes*.

4.1.1 Scheduler

The scheduler is responsible for coordinating the execution of the algorithm. The scheduler maintains a list of *working join nodes* and *potential join nodes*. A working join node is a node to which a portion of the hash table range has been assigned and on which a join process currently runs. Potential join nodes are the nodes in the environment that can be used to run additional join processes.

In the hash table building phase, when a *memory full* message is received from a working join node f, a new join node w is selected from the list of potential join nodes. In our implementation, the node with the largest amount of available memory is selected as the new join node when a working join node is full. The goal of this approach is to minimize the number of additional nodes. A join process on node w is instantiated, the number of data sources and the hash table range are sent to that process. The id of the new join node is sent to node f so that the node can forward to the new join node the data buffers that have been received or pending from data sources but have not been fully processed due to lack of memory space. The id of node w and its hash table range is broadcast to the data sources so

[1]This phase also can be executed using an adaptive algorithm that will expand to additional nodes to avoid memory overflow. The algorithms for the table building phase can be adapted for the probing phase, because operations performed on the elements of the second relation are similar to those performed on the first relation elements.

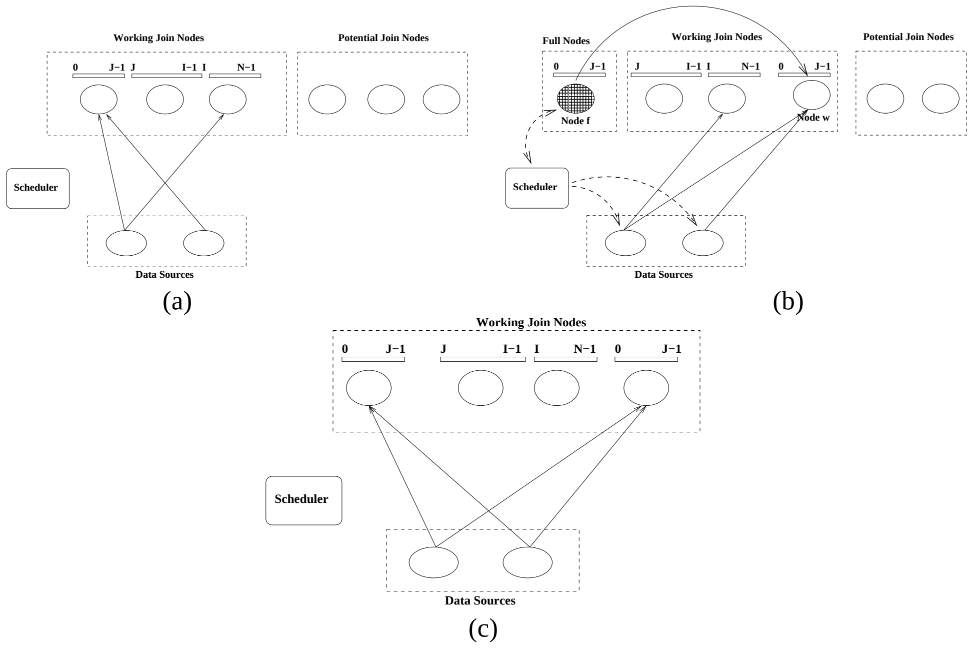

Figure 1. Execution of replication-based EHJA. (a) Initial configuration at the beginning of the hash table building phase. (b) After bucket on one join node is full. (c) The hash table probing phase.

that the data sources update their local list of working join nodes.

The scheduler is also responsible for synchronizing the join processes and data sources between the hash table building and hash table probing phases of the algorithm.

4.1.2 Data Sources

A data source provides the elements of the join relations **R** and **S** to the join processes. Relations **R** and **S** can be distributed over a number of nodes (or sites) in the environment. We should note that the two relations may be generated on the fly (*e.g.,* as a result of a select statement in the query plan) or read from disk on those nodes.

A data source keeps a buffer for each join process in the system. When the elements of relations **R** and **S** are generated or retrieved from disk, they are inserted into the buffers based on their hash values and the partitioning of the hash table among the join processes. When a buffer is full, it is sent to the corresponding join process.

When a data source receives a message from the scheduler process about addition of a new join node, it means one of the working join nodes is full. Depending on the algorithm employed, the full join node f is either moved to the list of join nodes with full hash table buckets (the replication-based and hybrid algorithms) or its hash range is updated (the split-based algorithm), and the new join node w is inserted into the list of working join nodes with its hash range.

When a data source has finished processing all the elements of relation **R**, it waits for a *start probe* message from the scheduler process to proceed to the hash table probing phase. When that message is received, the lists of working and full join nodes are merged. The probing phase is executed with the merged list of join nodes.

4.1.3 Join Processes

A join process is responsible for building and maintaining a portion of the hash table and performing the join operations on the local hash table. A join node is a node on which a join process executes. For simplicity of presentation, we assume that only one join process runs on a join node. Hence, we use join node and join process interchangeably in this paper.

In the hash table building phase, data elements received from data sources are inserted into the hash table based on their hash values. If memory for data elements cannot be allocated, the join process sends a *memory full* message to the scheduler and receives from the scheduler the id of the new join node. Note that during this information exchange, data sources may keep sending data buffers to the join process. Hence, there may be pending messages from data sources. If the replication-based and hybrid algorithms are employed, the join process is responsible for forwarding all the data in these buffers to the new join process. If the split-based algorithm is employed, the join process sends the new join process only the portion of the data that belongs to that process. In the hash table probing phase, the join process receives the data elements of relation **S** and carries out the join operation on these elements.

4.2 The Algorithms

4.2.1 Split-based Algorithm

The split-based algorithm is implemented using the adaptive algorithm proposed in [2] which is based on the linear and dynamic hashing scheme proposed in [19, 15]. The hash table range is initially partitioned among P join nodes; each node stores a single bucket. In the table building phase, pairs of hash functions, h_i and $h_{(i+1)}$ (where i = 0,1,2,...), are used to address the buckets. The function h_i maps a join attribute value K to bucket $K\ mod\ (P * 2^i)$. A pointer, referred to as the split pointer, is used to determine which function should be used for a join attribute element. The split pointer also points to the next bucket to be split, when a bucket overflows. When a bucket, which is addressed by h_i, is split, the hash function to address the two new buckets is set to $h_{(i+1)}$. The elements in the original bucket are mapped to the new buckets by $h_{(i+1)}$. The split operation results in inter-processor communication; the elements that are hashed to the bucket portion assigned to the new join node are sent to that node. The scheduler maintains a barrier split pointer, which follows the split pointer and is incremented only when the scheduler receives a done message from the bucket that will be split. This pointer is necessary to ensure that a bucket is not requested to split while it is being split and that at most two hash functions are active simultaneously. At the end of the table building phase, the scheduler sends the final (i,split pointer) pair to data sources so that they can apply the corresponding hash functions to elements in the probing phase.

A more detailed description of the algorithm can be found in [2]. The main characteristic of the algorithm is that when bucket overflow occurs and new join nodes are allocated, buckets are split among join nodes (working and new join nodes). This operation effectively partitions the hash table range among join nodes. When a split is performed, some of the elements in the overflowed bucket have to be transferred to the new join node. In the probing phase, an element is sent to the join node, which holds the corresponding hash table range. Thus, no extra communication overhead is incurred in the probing phase.

4.2.2 Replication-based Algorithm

The replication-based algorithm starts with a set of working join nodes. The hash table range is partitioned into buckets and each join process is assigned one of the buckets.When a working join node overflows its bucket in the table building phase, the *hash table range* assigned to the bucket is replicated on a new join node. Further elements of relation **R** that map to that range are sent to the new join node. Figure 1 illustrates an example execution. The hash table consists of N elements and its range is initially partitioned among three join nodes. Figure 1(b) shows the configuration after the bucket with hash range $0...J-1$ is full. One of the nodes in the potential join nodes list is added to the set of working join nodes, and the node with the full bucket is moved to the list of full nodes. The arrow between node f and node w in the figure shows the fact that node f sends the buffers that are waiting in the local message queue to node w. Node f stops receiving any more join attribute elements. Data sources are informed of the new join node by the scheduler and direct the remaining elements that are hashed to the same range, to that node.

The hash table probing phase is shown in Figure 1(c). If a hash table range has been replicated during the table building phase, the elements of the probe relation, whose hash values fall into that range have to be sent to all of the join processes that have the same hash table range. As a result, if the algorithm expands to additional nodes during the hash table building phase, the volume of communication in the probing phase may increase.

4.2.3 Hybrid Algorithm

The split-based algorithm results in extra communication in the table building phase, since elements in an overflowed bucket has to be partitioned between two nodes. The replication-based algorithm eliminates this communication overhead. However, it introduces overhead in the probing phase, because an element that is hashed to the hash table range that is replicated has to be broadcast to all the corresponding nodes. The hybrid strategy aims to reduce the communication overheads in both phases by combining the split- and replication-based algorithms. This algorithm introduces an extra step, referred to here as *reshuffling*, between the table building and probing phases.

In the table building phase, the replication-based algorithm is employed. After data sources finish processing the first relation, some of the hash table ranges might have been replicated. In the reshuffling step, all the join nodes are grouped into sets based on the hash table ranges assigned to them. That is, all the nodes with the same hash table range are inserted into the same set. For each set, the reshuffling step repartitions the corresponding hash table range among the nodes in the set. We use a simple greedy heuristic to split the hash table array. The heuristic operates as follows. At the end of the table building phase, each node counts the number of elements at each hash table position. A global sum operation is performed among the nodes that share the same hash table range so that the total number of entries at each hash table position is computed. If there are N nodes in a set, the hash table array is partitioned into N contiguous sub-arrays so that the total number of entries in each array is equal. The hash table entries are redistributed among the nodes. At the end of the reshuffling phase, each join node is assigned a disjoint subset of the hash table range and there remain no replicated hash table ranges. The new partitioning is broadcast by the scheduler to all data sources. In the probing phase, each data source uses the new partitioning to determine to which node they send the second relation tuples. We should note that a tuple is sent to only one node, as in the split-based algorithm.

4.2.4 Performance Analysis

The performance difference between the split-based algorithm and the hybrid algorithm mainly results from the difference in overhead of the split phase in the split-based algorithm and of the redistribution phase in the hybrid algorithm. Assume the bucket size is b bytes, the original number of buckets is n, and the final number of buckets is N at the end of the table building phase. We define the expansion factor $E = \frac{N}{n}$ and T_w as the time to transfer a single byte across the network.

For the split-based algorithm, the number of split operations is equal to $\log E$ and each time the amount of data to be transferred is $\frac{b}{2}$. Thus, the total overhead T_o^{split} is: $T_o^{split} = \log E * \frac{b}{2} * T_w$.

For the hybrid algorithm, the amount of data to be exchanged in the reshuffling phase is $\frac{E-1}{E} * b$. Hence, the overhead T_o^{hybrid} is: $T_o^{hybrid} = \frac{E-1}{E} * b * T_w$.

The above equations suggest that the overhead for the split-based algorithm grows faster than that of the hybrid algorithm as the expansion factor E increases.

5 Experimental Results

In this section, we present a performance evaluation of the Expanding Hash-based Join Algorithms (EHJAs). The experiments were carried out using a Linux cluster, referred to here as OSUMed. OSUMed consists of 24 compute nodes and 1 front-end node. Each node is a Pentium III 933MHz with 512MB of main memory and 300GB local disk space, interconnected with switched 100 Mb/s Ethernet.

Data Generation. In the experiments, we used synthetic relations **R** and **S**, both of which share the same column and row structure. Each element in a relation consists of a 64-bit index (I), a 64-bit join attribute (JA), and n-byte data. The join attribute of relations **R** and **S** were generated using either Uniform or Gaussian distribution. Gaussian distribution was used to model data skew. The random numbers are based on user-specified mean and standard deviation, which are individually set for each relation. The relations were generated on-the-fly on multiple nodes as the join operation progressed. This simulates data streaming from a distributed database or table streams in a multi-join operation. For the experiments in this paper, J_R and J_S were generated with the same mean, sigma, and value range.

Performance Results. The first set of experiments examines the effect of varying the number of initial buckets, while the size of the relations is fixed. In the figures, "Out of Core" (OOC) denotes the non-expanding algorithm, in which only the initial set of join nodes are used. In this algorithm, if the memory space allocated for hash table buckets is exceeded, join is performed out of core. Figure 2 shows that the performance of all four join algorithms improves as the number of initial nodes (initial buckets) is increased, as expected. When 16 nodes are allocated initially, the aggregate memory space is sufficient to hold the hash table in memory. Hence, all the algorithms achieve the same performance. When there are few initial nodes, the three EHJAs outperform the OOC algorithm, since they are able to avoid buffer overflow and achieve better parallelism by recruiting additional nodes as needed. Moreover, the split-based and hybrid-based algorithms achieve better performance than the replication-based algorithm because they introduce less communication overhead. We also observe that the split-based and hybrid algorithms are less sensitive to the number of initial join nodes. Figure 3 shows the table building time for this set of experiments. The amount of extra communication in the hash table building phase is shown in Figure 4 for the three EHJAs. The split-based and hybrid-based algorithms have longer hash table building times than the replication-based algorithm due to the higher communication overhead introduced by the split and reshuffling steps. However, the benefits of better parallelism and less communication overhead in the probing phase outweighs the exta communication overhead in the table building phase in our experimental setup (see Figure 2). In Figure 5 we compare the split and reshuffling times in the split-based and hybrid algorithms, respectively. As is seen from the figure, if the estimation of the initial number of nodes is highly inaccurate, the split-based algorithm results in more overhead than the hybrid algorithm. For the 16-node case in the figure, no overhead is incurred since the hash table fits in aggregate memory across 16 nodes.

The second set of experiments examines the effect of increasing the relation size. The number of initial nodes is fixed at four in these experiments. The size of relations **R** and **S** was varied from 10M (10 Million) to 80M tuples. Figure 6 shows the total execution time, when **R** and **S** are the same size and uniform distribution is used for tuple generation. The split-based and hybrid algorithms scale bet-

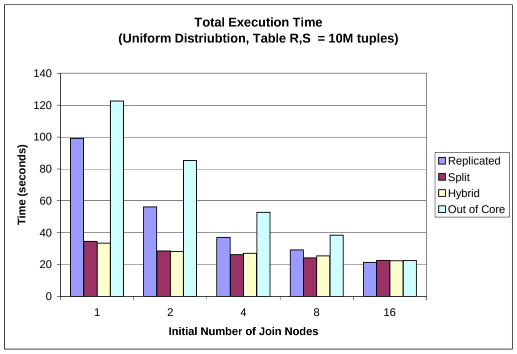

Figure 2. Effect of varying the number of initial working join nodes.

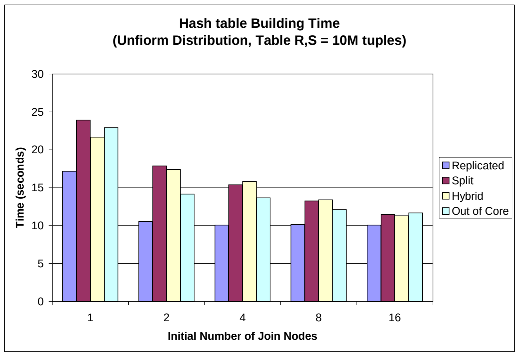

Figure 3. Effect of varying the number of initial working join nodes in the table building phase.

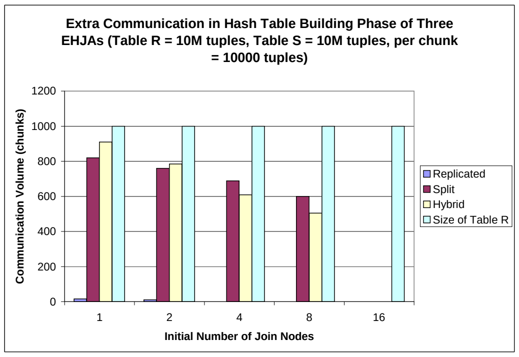

Figure 4. Extra communication volume introduced by different algorithms in the hash table building phase.

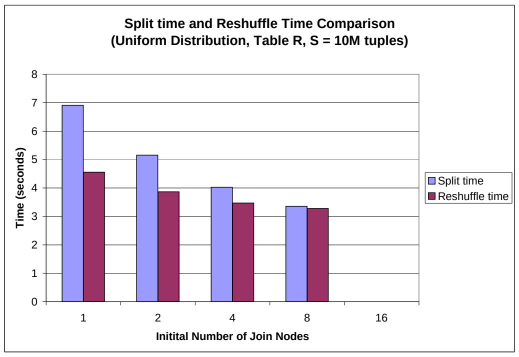

Figure 5. The split time and reshuffle time comparison in the hash table building phase.

ter than the replication-based and OOC algorithms. This is mainly because the algorithms will spread to more nodes as the size of **R** increases. As a result, increasing number of tuples in **S** means the replication-based algorithm will have greater communication overhead in the probing phase. The effect of increasing tuple size is shown in Figure 7. As seen from the figure, the hybrid-based algorithm scales better than the other two, because extra communication for a tuple is done only once in the reshuffling step and no communication overhead is introduced in the probing phase.

The execution times of the four algorithms are shown in Figures 8 and 9, when the larger relation is used in the table building phase. Normally, hash table should be built from the smaller of the two relations in order to reduce the chances of bucket overflow. However, in cases where one relation is generated before or faster than the other relation (e.g., streaming data applications), one may not have a choice but use the larger relation. As is seen from the figures, the replication-based algorithm achieves better performance. In this particular case, the communication overhead in the reshuffling phase is larger than that in the probing phase of the replication-based algorithm. Hence, the hybrid algorithm does not perform as well as the replication-based algorithm.

In the final set of experiments, the effect of data skew is examined. The number of initial nodes is fixed at four, and the sizes of **R** and **S** are set to 10M tuples each. Three sets of **R** and **S** were generated. The first set was generated using uniform distribution, the second using Gaussian with standard deviation of 0.001, and the third using Gaussian with standard deviation of 0.0001, which represents a highly skewed data distribution. The observed behavior is that with higher data skew, larger number of tuples will be hashed to a few join nodes, which creates memory and computation imbalance. The execution time of the algorithms is shown in Figure 10. The results show that all join algorithms adapt well when the skew is not very bad (sigma = 0.001). However, extreme data skew (sigma = 0.0001) results in significant performance degradation, as expected. The performance degradation of the hybrid algorithm is less

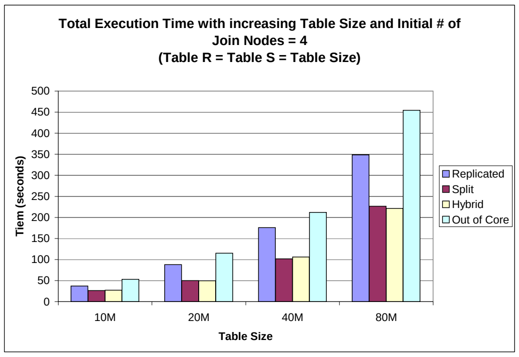

Figure 6. The total execution time of the four algorithms when the size of the relations is varied.

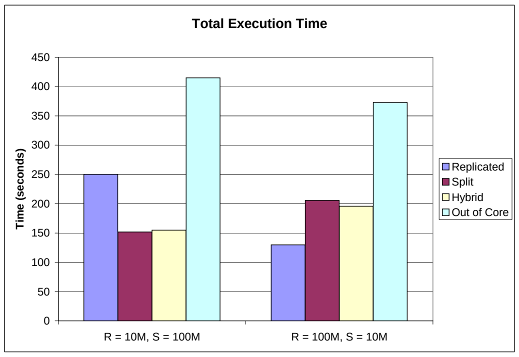

Figure 8. The total execution time of the algorithms when the larger relation is used to build the hash table.

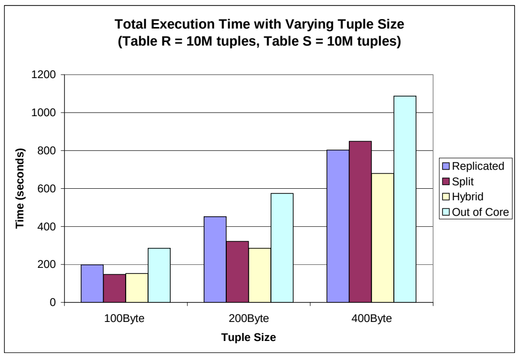

Figure 7. The total execution time of the algorithms when the size of tuples is varied.

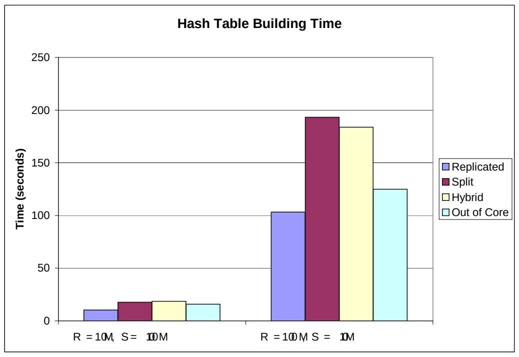

Figure 9. The table building time of the algorithms when the larger relation is used to build the hash table.

than that of the other algorithms; the split-based algorithm has the worst performance among the three algorithms. We attribute this result to the fact that the split-based algorithm is likely to do a lot of splits and communicate the same tuple many times, resulting in higher communication overhead in the hash table building phase as shown in Figure 11. The hybrid algorithm performs the best among the three strategies. This algorithm not only reduces communication overhead, but also does a better job of load balancing because replicated hash table ranges are redistributed in the reshuffling step.

Figures 12 and 13 show the load balance achieved by the EHJAs with uniform and skewed data (sigma = 0.0001) value distribution. As shown in the figures, the split-based and hybrid algorithms both achieve good load balance when data is generated using uniform distribution. However, when data is extremely skewed, the split-based algorithm suffers from load imbalance, whereas the hybrid algorithm still maintains a relatively good load balance. We should note that our main goal in this paper is to avoid bucket overflow. Our results show that the reshuffling step of the hybrid algorithm also helps in alleviating the computational load imbalance due to data skew.

6 Conclusions

In this paper, we compared three adaptive hash-based join algorithms that aim to avoid bucket overflow by allocating additional hosts in the environment. An experimental evaluation of the algorithms was performed on PC clusters. The experimental results show that the replication-based algorithm should be preferred over the split-based algorithm if the distribution of the join attribute values is highly skewed and/or the larger relation has to be used to build the hash table. Otherwise, the split-based algorithm achieves better performance. Among the three algorithms, on the average, the hybrid algorithm generally performs close to the better of the two or is the best algorithm. A common result across all the cases tested in this paper is that

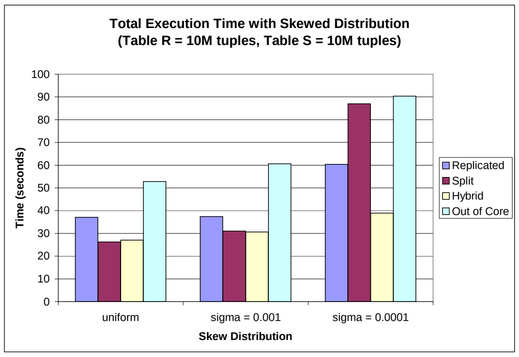

Figure 10. The total execution time of the algorithms when the distribution of join attribute values is skewed.

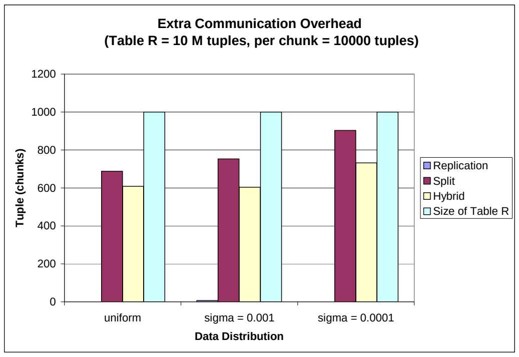

Figure 11. Communication overhead of the algorithms when the distribution of join attribute values is skewed.

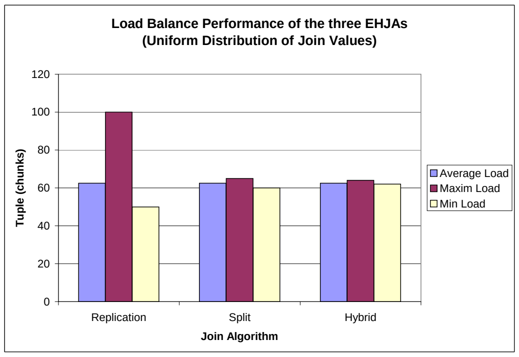

Figure 12. The maximum, minimum, and average load across join nodes with uniform distribution of data values.

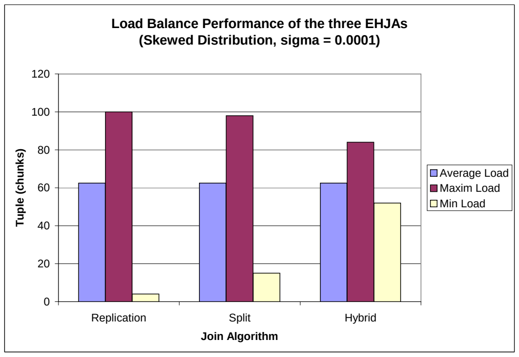

Figure 13. The maximum, minimum, and average load across join nodes with skewed distribution of data values.

the EHJAs achieve good performance by utilizing remotely avaliable memory and computational resources, when large scale equi-join operations are performed with limited memory per node. An EHJA can be an effective alternative to an out-of-core algorithm when the sizes of the join relations cannot be estimated efficiently and the resources in the environment are shared by other queries.

As future work, we plan to investigate the effect of different network configurations and I/O systems on the relative performance of different EHJAs. In this paper, we focused on one-way join operation. We also plan to expand our work to multi-way join operations as well. In a multi-way join operation, performance can be improved if results from joins at intermediate levels are maintained in memory.

References

[1] K. Aberer, P. Cudre-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-grid: a self-organizing structured p2p system. *SIGMOD Rec.*, 32(3):29–33, 2003.

[2] M. B. Amin, D. A. Schneider, and V. Singh. An adaptive, load balancing parallel join algorithm. In *Proceedings of Sixth International Conference on Management of Data (COMAD'94)*, Dec 1994.

[3] M. Bamha and G. Hains. A self-balancing join algorithm for SN machines. In Y. Pan, S. G. Akl, and K. Li, editors, *Proceedings of International Conference on Parallel and Distributed Computing and Systems (PDCS)*, Las Vegas, USA, 1998. IASTED/ACTA Press.

[4] M. Bamha and G. Hains. Frequency-adaptive join for shared nothing machines. *Journal of Parallel and Distributed Computing Practices (PDCP)*, 2(3):333–345, Sep 1999.

[5] M. Bamha and G. Hains. A skew-insensitive algorithm for join and multi-join operation on shared nothing machines. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications*. LNCS 1873. Springer-Verlag, Sep 2002.

[6] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Effi cient processing of spatial joins using r-trees. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of*

Data (SIGMOD93), pages 237–246, Washington, DC, May 1993.

[7] D. DeWitt and R. Gerber. Multiprocessor hash-based join algorithms. In *Proceedings of the 11th International Conference on Very Large Data Bases*, Aug. 1985.

[8] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. Technical Report CS-TR-92-1098, University of Wisconsin-Madison, July 1992.

[9] P. Fotouhi and S. Pramanik. Optimal secondary storage access sequence for performing relational join. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):318–328, Sept. 1989.

[10] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, June 1993.

[11] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed object location in a dynamic network. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 41–52. ACM Press, 2002.

[12] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *17th International Conference on Very Large Data Bases (VLDB'91)*, Barcelona, Catalonia, Spain, Sep 1991. Morgan Kaufmann, ISBN 1-55860-150-3.

[13] K. Imasaki and S. Dandamudi. An adaptive hash join algorithm on a network of workstations. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS'02)*, Fort Lauderdale, CA, USA, Apr 2002.

[14] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, Dec. 2000.

[15] P.-Å. Larson. Dynamic hash tables. *Commun. ACM*, 31(4):446–457, 1988.

[16] C. Lee and Z.-A. Chang. Workload balance and page access scheduling for parallel joins in shared-nothing systems. In *Proceedings of the International Conference on Data Engineering*, pages 411–418, Vienna, Austria, Apr. 1993.

[17] A. Lerner and S. Lifschitz. A study of workload balancing techniques on parallel join algorithms. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, USA, Jul 1998.

[18] W. Li, D. Gao, and R. T. Snodgrass. Skew handling techniques in sort-merge join. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 169–180, Madison, Wisconsin, USA, June 2002. ACM.

[19] W. Litwin. Linear hashing: A new tool for fi le and table addressing. In *Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada, Proceedings*, pages 212–223. IEEE Computer Society, 1980.

[20] H. Lu, K.-L. Tan, and M.-C. Shan. Hash-based join algorithms for multiprocessor computers. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 198–209, Brisbane, Australia, Aug. 1990.

[21] L. Lu and M. Chen. Parallelizing loops with indirect array references or pointers. In *Proceedings of the 4th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1991.

[22] G. S. Manku. Routing networks for distributed hash tables. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 133–142. ACM Press, 2003.

[23] R. Marek and E. Rahm. On the performance of parallel join processing in shared nothing database systems. In *Proceedings of the 5th International Conference on Parallel Architectures and Languages Europe*, pages 622–633, Munich, June 1993. Springer-Verlag.

[24] T. H. Merrett, Y. Kambayashi, and H. Yasuura. Scheduling of page-fetches in join operations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 488–498, Cannes, France, Sept. 1981.

[25] P. Mishra and M. H.Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, Mar. 1992.

[26] M. C. Murphy and D. Rotom. Process scheduling for multiprocessor joins. In *Proceedings of the International Conference on Data Engineering*, pages 140–148, Feb. 1989.

[27] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.

[28] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware'2001*, Germany, November 2001.

[29] D. Schneider and D. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD89)*, pages 110–121, Portland, OR, June 1989.

[30] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[31] T. Urhan and M. J. Franklin. XJoin: Getting fast answers from slow and bursty networks. Technical Report CS-TR-3994, University of Maryland, 1999.

[32] Y. E. I. Viswanath Poosala. Estimation of query-result distribution and its application in parallel-join load balancing. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 448–459, Mumbai, India, Sept. 1996.

[33] A. N. Wilschut, J. Flokstra, and P. M. Apers. Parallel evaluation of multi-join queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD95)*, pages 115–126, San Jose, CA, May 1995.