Hash-Merge Join: A Non-blocking Join Algorithm for Producing Fast and Early Join Results[*]

Mohamed F. Mokbel Ming Lu Walid G. Aref

Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398

{mokbel,mlu,aref}@cs.purdue.edu

Abstract

This paper introduces the hash-merge join algorithm (HMJ, for short); a new non-blocking join algorithm that deals with data items from remote sources via unpredictable, slow, or bursty network traffic. The HMJ algorithm is designed with two goals in mind: (1) Minimize the time to produce the first few results, and (2) Produce join results even if the two sources of the join operator occasionally get blocked. The HMJ algorithm has two phases: The hashing phase and the merging phase. The hashing phase employs an in-memory hash-based join algorithm that produces join results as quickly as data arrives. The merging phase is responsible for producing join results if the two sources are blocked. Both phases of the HMJ algorithm are connected via a flushing policy that flushes in-memory parts into disk storage once the memory is exhausted. Experimental results show that HMJ combines the advantages of two state-of-the-art non-blocking join algorithms (XJoin and Progressive Merge Join) while avoiding their shortcomings.

1. Introduction

Traditional join algorithms [9, 16, 19] assume that all input data is available beforehand. This assumption is not valid for web-based applications. A web query retrieves data items from remote sources via a network connection. Network traffic may be unpredictable, slow, or bursty, which may result in blocking input data [1, 20]. The blocking behavior of network traffic makes the traditional join algorithms unsuitable for pipelined query plans [18]. In addition, traditional join algorithms optimize the query execution for the production of the entire join result. However a typical internet user may be interested only in the first few results [3, 4].

With the goals of avoiding the blocking behavior of remote data sources and producing join results as quickly as possible, a family of non-blocking join algorithms are developed (e.g., see [7, 14, 10, 13, 15, 21, 23, 24]). Non-blocking join algorithms have the ability to produce join results even if one or both sources are blocked. Thus, a fully pipelined query plan can still function properly even with blocking sources. In addition, non-blocking join algorithms are optimized to produce the first few results as quickly as possible. Thus, it is suitable for the case when the users are interested only in getting the first few answers. In addition to web-based applications, non-blocking join algorithms are useful in data integration [13], parallel databases [6], online aggregation [10, 12], providing approximate answers [17, 22], spatial databases [15], and adaptive query processing [2, 11].

In this paper, we propose the *Hash-merge* join algorithm (HMJ, for short); a novel non-blocking join algorithm that deals with unpredictable and slow network traffic. The *Hash-merge* join algorithm is designed with two goals in mind: (1) Minimizing the time to produce the first few results. (2) Providing the ability to produce join results even if the two sources of the join operator are blocked.

The *Hash-merge* join algorithm has two phases: The hashing and merging phases. The hashing phase employs an in-memory hash-based join algorithm that produces join results as quickly as data arrives. Once the memory gets filled, certain parts of the memory are flushed into disk storage to free memory space for the newly incoming tuples. If one of the sources is blocked for any reason, e.g., due to slow or bursty network traffic, the hashing phase can still produce join results from the unblocked source. If the two input sources are blocked, the HMJ algorithm starts its merging phase. In the merging phase, previously flushed parts in disk are joined together using a sort-merge-like join algorithm. Thus, the HMJ algorithm can produce join results even if the two sources are blocked. Once the block-

[*]This work was supported in part by the National Science Foundation under Grants IIS-0093116, EIA-9972883, IIS-9974255, IIS-0209120, and EIA-9983249.

ing of any of the two sources is resolved, the HMJ algorithm switches back to the hashing phase. The HMJ algorithm switches back and forth between the two phases until all data items are received from remote sources. Then, the whole memory is flushed into disk storage and the merging phase takes place to produce the final part of the join result.

The HMJ algorithm combines the advantages of two state-of-the-art non-blocking join algorithms, XJoin [20, 21] and Progressive Merge Join (PMJ) [7, 8] while avoiding their shortcomings. XJoin stores incoming tuples in memory while employing an in-memory hash-based join algorithm to produce fast join results. When memory gets filled, the largest hash bucket is flushed into disk. Although XJoin produces fast results, its I/O complexity is high and hence a large total time to produce the entire join result. On the other side, PMJ partitions the memory into only two partitions, one for each source. Once the memory gets filled, each partition is sorted and is joined with the other partition, and then is flushed to disk. Thus, in PMJ, no join results are produced until the memory gets filled. This results in a higher initial delay than that of XJoin for producing the first results. However, PMJ employs a sort-merge-like join algorithm to join disk-resident data. Thus, PMJ performs less I/O's and hence less overall time than XJoin for producing the entire results of a join.

Similar to XJoin, the proposed *Hash-merge* join algorithm employs an in-memory hash-based join algorithm to produce fast and early results. To avoid the drawbacks of XJoin, HMJ employs a new flushing policy, termed the *Adaptive Flushing* policy that aims to synchronously flush two hash buckets, one from each source, into disk storage. By using the *Adaptive Flushing* policy, HMJ can use a refined version of the sort-merge-like join algorithm as in PMJ. Thus, HMJ results in less I/Os and overall time than XJoin for producing the total result. In summary, the contributions of this paper are as follows:

1. We propose the *Hash-merge* join algorithm; a new non-blocking join algorithm that is applicable in environments where data arrives from remote sources via unpredictable network connections (Section 3).

2. We propose a synchronized flushing policy, termed the *Adaptive Flushing* policy that can be used in conjunction with any hash-based non-blocking join algorithm. The *Adaptive Flushing* policy is adaptable to the fluctuations of data arrival rates. The main goal of the *Adaptive Flushing* policy is to always keep the memory balanced between the two remote sources, even if one of the sources has a higher arrival rate than the other. As we will see in Section 4, and in the performance section, keeping the memory balanced makes the join algorithm more responsive to producing fast results once a new data item is received.

3. We prove the correctness of HMJ by proving the following: (a) Completeness, i.e., all join results will be produced by HMJ. (b) Uniqueness, i.e., HMJ produces duplicate-free results (Section 5).

4. We provide experimental evidence that HMJ (with the *Adaptive Flushing* policy) outperforms XJoin and PMJ (Section 6).

The rest of the paper is organized as follows: Section 2 highlights related work for non-blocking join algorithms. Sections 3 introduces the HMJ algorithm. The *Adaptive Flushing* policy is introduced in Section 4. The proof of correctness of the HMJ algorithm is given in Section 5. A study of the performance of HMJ and a discussion of the results are presented in Section 6. Finally, Section 7 concludes the paper.

2. Related Work

Hash-based join algorithms. The non-blocking symmetric hash join [23, 24] extends the traditional hash join algorithm to support pipelining. Two in-memory hash tables with m buckets are maintained for sources A and B. Once a new tuple t, with a hash value $h(t)$, is received from source A, t is used to probe bucket $h(t$ of source B. Then, t is stored in bucket $h(t)$ of the hash table of source A. The symmetric hash join algorithm requires that the two relations fit in memory. The XJoin algorithm [20, 21] extends the symmetric hash join to be applied for disk-resident data. XJoin starts by joining tuples in memory, similar to the symmetric hash join. When memory gets filled, the largest hash bucket among all A and B buckets is flushed into disk. When both sources are blocked, XJoin performs join using the buckets previously flushed into disk. The double Pipelined Hash Join (DPHJ) [13] is another extension of the symmetric hash join algorithm. DPHJ has two stages. The first stage is similar to the in-memory join in the symmetric hash join and XJoin. In the second stage, pairs that are not joined together in the first phase are marked and are joined in disk. DPHJ is suitable for moderate size data, but does not scale well for large data sizes.

Sort-based join algorithms. The progressive-merge join (PMJ) algorithm [7, 8] is the non-blocking version of the traditional sort-merge join. The main idea of PMJ is to read as much data as can fit in memory. Then, in-memory data is sorted and is joined together, and then is flushed into disk. When all data is received, PMJ joins the disk-resident data using a refinement version of the sort-merge join that allows producing join results while merging.

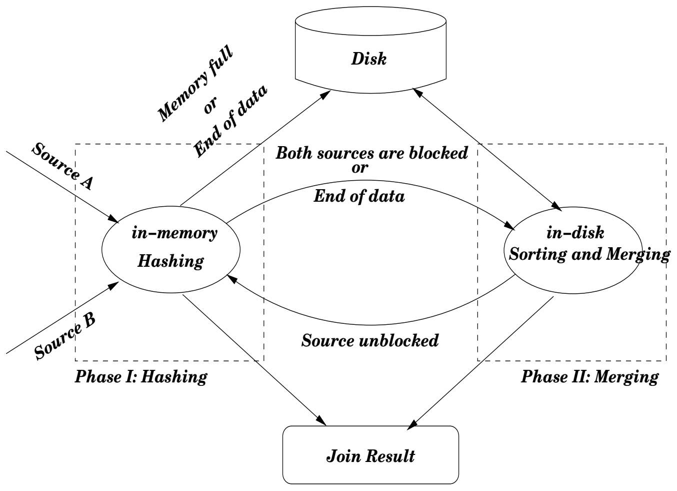


Figure 1. The Hash-merge join algorithm.

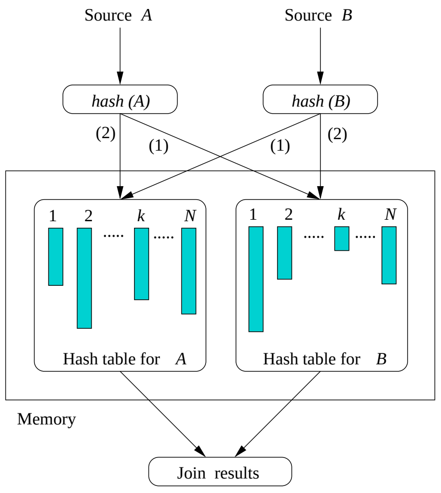


Figure 2. Sketch of the hashing phase.

Nested-loop-based join algorithms. The family of ripple joins [14, 10] generalize block nested-loop join and hash join. Ripple joins automatically adjust their behavior to provide precise confidence interval for online aggregation. The scalability of ripple joins is discussed in [14]. However, ripple joins are geared towards online aggregation, thus the quality of the results (obtained from statistical measures) controls the join processing.

3. Hash-Merge Join Algorithm

The *Hash-merge* join algorithm (HMJ, for short) has two phases: The hashing and the merging phases. Figure 1 provides a state diagram of HMJ. HMJ starts with the hashing phase where input tuples from two remote sources A and B are received. Incoming tuples are stored in in-memory hash buckets based on their hash values. In-memory tuples are joined together during the hashing phase and are added to the output result. Once the memory gets filled, certain parts of memory are flushed into disk. If both sources are blocked for any reason, e.g., due to a slow network, then HMJ transfers control to the merging phase. In the merging phase, tuples that are previously flushed to disk are joined together. Thus, HMJ has the ability to produce join results even if both sources are blocked. If the blocking of any source is resolved, HMJ returns to the hashing phase. HMJ alternates between the hashing and merging phases till the whole data is processed. Then, the merging phase takes control to output the final part of the result.

3.1. Phase I: Hashing

Figure 2 sketches the hashing phase of HMJ. Two in-memory hash tables, each with N buckets, are maintained for sources A and B. Buckets are allowed to have different sizes. As a consequence, the memory is not necessarily divided evenly between sources A and B. In the following, we use the symbols A_k and B_k to denote the kth bucket in the hash table for sources A and B, respectively.

The pseudo code of the hashing phase is given in Figure 3. Once a new tuple t with hash value $h(t)$ is received from source A (respectively, source B), we check if there is enough memory to accommodate t (Step 1 in Figure 3). If there is enough memory space, t is used to probe the hash table of source B (respectively, source A). Thus, t is joined with all tuples in bucket $B_{h(t)}$ (respectively $A_{h(t)}$) (Step 3 in Figure 3). Then, the tuple t is stored in bucket $A_{h(t)}$ (respectively, $B_{h(t)}$) (Step 4 in Figure 3). However, if memory is exhausted, we need to free some part of memory to accommodate t and other incoming tuples. A certain flushing policy (see Section 4) is used to free part of the memory. The main idea is to choose two buckets A_k and B_k with the same hash value k. Then, A_k and B_k are sorted internally in memory, and are synchronously flushed into disk.

If one of the sources, say source A, is blocked for any reason (e.g., a slow or bursty network connection), the hashing phase can still produce join results. Tuples from the unblocked source B can still be received and are used to probe the in-memory hash table of A to produce join results. HMJ transfers the control from the hashing phase to the merging phase only if: (1) The two sources are blocked, or (2) All data is processed. In the former case, the *Hash-merge* join algorithm returns to the hashing phase when the blocking behavior of any of the sources is resolved.

The idea of the hashing phase is similar to that of the symmetric hash join [23, 24] and to the first stage of both XJoin [21] and the Double Pipelined Hash Join (DPHJ) [13]. However, there are two major differences:

```
**Procedure** HashingPhase(tuple $t$, source $A$ ($B$))
**Begin**
  1. If there is no enough memory to accommodate $t$
      (a) The flushing policy chooses two buckets $A_k$ and $B_k$ as victims.
      (b) Sort buckets $A_k$ and $B_k$ in memory.
      (c) Flush buckets $A_k$ and $B_k$ into disk.
  2. Compute the hash value $h(t)$ of tuple $t$.
  3. Join tuple $t$ with all tuples in bucket $B_{h(t)}$ ($A_{h(t)}$).
  4. Store tuple $t$ in bucket $A_{h(t)}$ ($B_{h(t)}$).
**End.**
```

Figure 3. Pseudo code of the hashing phase

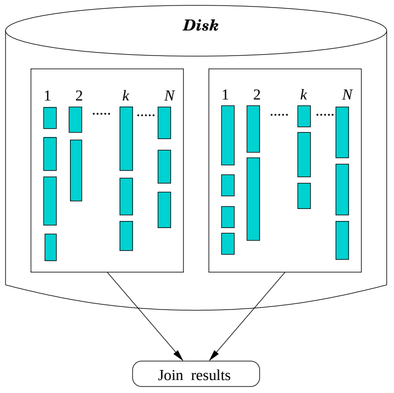


Figure 4. The layout of the disk at the start of the merging phase.

(1) In HMJ, selecting the victim partitions to be flushed into disk is handled in a different way than in the cases of XJoin and DPHJ. Basically, HMJ selects two victim partitions (with the same hash value), one from each source. On the other hand, both XJoin and DPHJ choose only one partition from one source to be flushed. Notice that in the symmetric hash join, there is no such victim partitions, where it is assumed that all data items can fit in memory. (2) In HMJ, flushed partitions need to be sorted in memory before flushing.

3.2. Phase II: Merging

The merging phase of the *Hash-merge* join algorithm deals with in-disk hash buckets that are previously flushed into disk during the hashing phase. Figure 4 gives a snapshot of the disk storage upon the start of the merging phase. For each hash bucket with hash value h, there are m_h blocks for sources A and B. The m_h blocks indicate that this bucket has been chosen as a victim m_h times in the hashing phase. For example, in Figure 4, bucket 1 has four blocks while bucket 2 has only two blocks.

The pseudo code of the merging phase is given in Figure 5. The main idea of the merging phase is to apply a refinement version of the traditional sort-merge join algorithm for each individual bucket. Thus, the sort-merge algorithm is applied N times (Step 1 in Figure 5). We introduce a parameter f to tune the performance of the merging phase. f indicates the *fan in* of the sort-merge algorithm, i.e., the number of partitions to be merged in each step of the merging phase. Thus, to merge all blocks in each bucket, we need $Log_f A_{mi}$ passes for each bucket, where A_{mi} is the number of blocks in bucket A_i (Step 2 in Figure 5). For each pass, we use the sort-merge join with two refinements: (1) Join results are produced during the merging. Thus, the blocking behavior of separating the sorting and merging steps is avoided (Step 3a in Figure 5). (2) To avoid producing duplicate results, we do not produce the tuples that result from blocks that are of the same number (Step 3b in Figure 5). These tuples are already produced either in the hashing phase or in an earlier merging pass. Notice that such two blocks have been flushed into disk concurrently, after being completely joined with each other in memory. The duplicate-free results are continuously sent to the output stream for further processing (e.g., a pipelined query plan) (Step 3c in Figure 5).

Figure 6 gives an example of the merging phase in the non-blocking HMJ algorithm. One bucket from source A (respectively, B) has two blocks A_{b1} and A_{b2} (respectively, B_{b1} and B_{b2}). The pairs of (A_{b1},B_{b1}) and (A_{b2},B_{b2}) are already joined together either in the hashing phase or in an early merging pass where the tuples (4,4) and (6,6) are produced. In the merging phase, only the pair of blocks (A_{b1},B_{b2}) and (A_{b2},B_{b1}) need to be joined. Duplicate avoidance is employed by checking whether the produced tuples come from buckets with the same number or not.

The merging phase of HMJ is similar to the merging phase of the progressive merge join algorithm (PMJ) [7, 8] in the sense that both algorithms employ a refinement version of the traditional sort-merge join algorithm. However, two differences can be distinguished: (1) HMJ applies the sort-merge join algorithm N times for the N hash buckets, while in PMJ, the sort-merge join algorithm is applied only once, where there is only one bucket per data source. (2) HMJ transfers control back and forth between the hashing and merging phases, while in PMJ, the merging phase starts after the data is finished and is processed in memory.

```
Procedure MergingPhase()
Input:
  • $A$ and $B$: Two disk partitions, each with $N$ hash buckets,
    correspond to sources $A$ and $B$ (e.g., see Figure 4). $A_i$ ($B_i$)
    is the $i$th bucket of source $A$ ($B$) with $m_i$ blocks. $A_{ik}$ ($B_{ik}$)
    denotes the $k$th block of the $i$th bucket for source $A$($B$).
  • $f$: The fan in; the number of blocks to be merged each time.
Begin
  1. For i =1 to N
  2. Do $Log_f A_{mi}$ times
  3. For k =1 to $A_{mi}$/f step f
     (a) Sort and merge the blocks $A_{ik}, A_{i(k+1)}, \cdots, A_{i(k+f-1)}$ with the blocks
         $B_{ik}, B_{i(k+1)}, \cdots, B_{i(k+f-1)}$ using a modified version of the
         traditional sort-merge join algorithm that can produce join results (x,y) while sorting.
     (b) If a join result (x,y) comes from two similar blocks, i.e., $x \in A_{ij}, y \in B_{ij}$,
         then ignore the tuple (x,y). Otherwise, add the tuple (x,y) to the result set S.
     (c) Send S to the output stream.
End.
```

Figure 5. Pseudo code of the merging phase.

3.3. Number of Hash Buckets

The choice of the number of hash buckets in HMJ results in a trade-off between the efficiency of the hashing and merging phases. The hashing phase favors a large number of hash buckets (i.e., many small-sized buckets) for two reasons: (1) A newly arriving tuple will be tested for the join condition with a limited number of tuples. (2) Flushed partitions will have small sizes, thus memory is almost always full, which results in more join results during the hashing phase. On the other hand, the merging phase favors a small number of hashing buckets (i.e., few large-sized buckets) for two reasons: (1) Having large sized buckets results in less number of in-disk buckets, hence, less number of times to apply the merging among in-disk buckets (Step 3 in Figure 5). (2) Flushing large size buckets results in almost full disk pages. Thus, the utilization of disk pages is increased.

To resolve this issue, we use a large number of hashing buckets during the hashing phase. However, when flushing, we combine each p consecutive buckets together. Thus, if the number of hashing buckets in the hashing phase is h, then the number of hash buckets in disk would be h/p. The flushing policy chooses one corresponding pair (i.e., ones with the same hash value from each source) of the h/p buckets as the victim buckets.

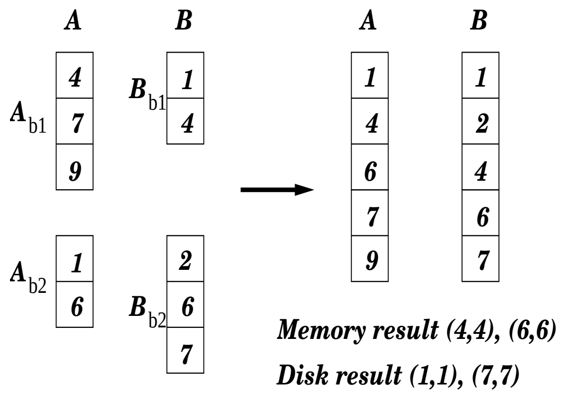


Figure 6. Example of the merging phase.

4. Flushing Memory Partitions

Flushing in-memory buckets into disk plays an important role in the efficiency of HMJ. In this section, we discuss some flushing policies that can fit in HMJ. Based on the naive policies, we distinguish three requirements that need to be satisfied by an efficient flushing policy. Then, we develop the *Adaptive Flushing* policy that produces the best results for HMJ. To illustrate our ideas, we use the example given in Figure 7. A memory of size 100 is divided into ten hash buckets; five for each source. A flushing policy needs to choose two victim buckets; one from each source, with the same hash value. To speed up the process of selecting victim buckets, we maintain an in-memory summary table that keeps track of the number of tuples in each bucket pair for both sources, along with the total number of tuples.

Flush All Policy. In this policy, we combine all the in-memory buckets into only one bucket. Then, the whole memory is flushed into disk. *Flush All* policy is used in the progressive merge join algorithm [7], where there is only one bucket for each source. The main motivation for the *Flush All* policy is: (1) Flushing the whole memory results in less I/O where pages are completely full. (2) Hash buckets are organized in disk in large blocks. Large blocks result in a more efficient merging phase. However, the *Flush All* policy results in major drawbacks: (1) After flushing, the whole memory is freed. Free memory results in a significant delay in producing join results in the hashing phase. Consider the case that a new tuple arrives while the memory is only 10% full. The new tuple has little chance to be joined with any other in-memory tuple. (2) Combining all tuples in only one bucket results in joining unnecessary tuples in disk. For example, a tuple with hash value h_1 will be joined with tuples with hash value h_2.

Flush Smallest Policy. The *Flush Smallest* policy selects victim bucket pairs with smallest total size. For example, in Figure 7, the *Flush Smallest* policy chooses the fourth

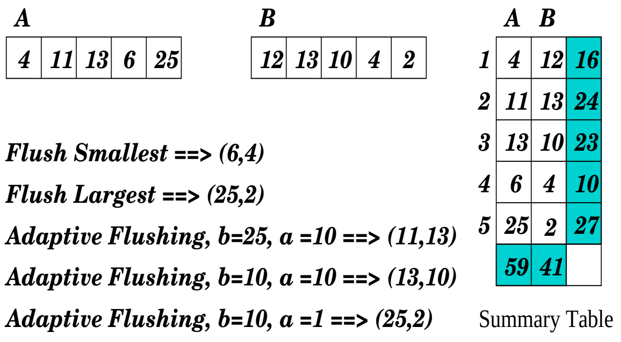


Figure 7. Example of flushing policies.

bucket pair, where it has the smallest total (10) among all in-memory bucket pairs. The *Flush Smallest* policy is biased towards the hashing phase. The main idea is to always keep the memory almost full. Then, a newly arriving tuple has a high chance to be joined with other in-memory tuples. However, the performance deteriorates in the merging phase since most disk-based blocks are of small sizes. In addition, the hashing phase results in excessive I/Os due to the continuous flushing of small partitions.

Flush Largest Policy. The *Flush Largest* policy selects victim bucket pairs with largest total size. For example, in Figure 7, the *Flush Largest* policy chooses the fifth bucket pair, where it has the largest total (27) among all in-memory bucket pairs. The *Flush Largest* policy is biased towards the merging phase. The main idea is to always have in-disk large blocks. Large blocks result in an efficient merging phase. At the same time, the *Flush Largest* policy does not free all the memory. Thus, join results can still be produced from the hashing phase. However, the *Flushing Largest* policy has the following drawbacks: (1) Selecting the largest sum bucket pair may result in flushing small buckets. For example, in Figure 7, a bucket with size two from source B is flushed. (2) If the memory is not balanced between the two sources, i.e., source A has 80% of the memory, while source B has only 20%, selecting the largest bucket pair may result in increasing the memory skewing.

Based on the above discussion, we identify three main requirements that need to be considered when designing a flushing policy for the HMJ.

1. **Supporting the hashing phase.** The flushing policy should always keep enough in-memory tuples such that newly incoming tuples can produce in-memory join results.

2. **Supporting the merging phase.** The flushing policy should avoid flushing small partitions that deteriorate the performance of the merging phase.

3. **Keeping balanced memory.** The flushing policy should try to keep the memory balanced between sources A and B. To illustrate the importance of having a balanced memory, assume the case where 90% of the memory is allocated for source A while only 10% is allocated for source B. A newly arrived tuple from A has little chance to find matching tuples from B. Thus, the performance of the hashing phase is reduced. In addition, flushed buckets from B tend to have small sizes. Thus, the performance of the merging phase deteriorates.

In the rest of this section, we propose the *Adaptive Flushing* policy; a flushing policy that works along with HMJ and fulfills the above three requirements.

4.1. Adaptive Flushing Policy

The main idea of the *Adaptive Flushing* policy is to make the flushing adaptable to the changes in the blocking behavior of both sources. For example, if source A blocks, then, the memory may have more tuples from B than A. The *Adaptive Flushing* policy aims to balance the memory to have similar number of tuples from each source. To tune the adaptability of the *Adaptive Flushing* policy towards memory balancing, we use the parameter b. If $|A|$ and $|B|$ are the ratios of tuples from A and B, respectively, to all memory tuples, then we say that the memory is balanced only if $absolute(|A| - |B|) < b$. To avoid flushing small buckets, the *Adaptive Flushing* policy uses the parameter a to indicate the smallest acceptable size for a certain bucket to be flushed. Figure 8 gives the pseudo code of the *Adaptive Flushing* policy.

Initially, the *Adaptive Flushing* policy has a search space S that contains all the possible bucket pairs (A_k, B_k). If the memory is balanced (Step 1 in Figure 8), the search space S is limited to the bucket pairs whose sizes are greater than the acceptable bucket size a. If there is no bucket pair that satisfies the smallest bucket size threshold, the search S is kept to the whole set of bucket pairs. Furthermore, the search space S is limited (if possible) to those bucket pairs that will not affect memory balancing upon flushing. Finally, the victim bucket pair is the pair with largest total size from the limited search space S.

In the case of unbalanced memory (Step 2 in Figure 8), the search space S is limited to those bucket pairs that reduce the memory unbalancing. For example, if the memory has more tuples from A than from B, then the victim bucket pair should have more A tuples than B tuples. Furthermore, if possible, the search space S is limited to those bucket pairs of size larger than a. Finally, the victim bucket pair is the one with largest total size from the limited search space S.

For example, consider applying the *Adaptive Flushing* policy with $b = 25$ and $a = 10$ to the memory layout in Figure 7. The difference in memory ratio is $59\% - 41\% =$

```
Procedure Adaptive Flush Policy()
  • Input: S: The set of all bucket pairs $(A_k, B_k)$, $a$: The acceptable partition size, $b$: Balancing threshold.
  • Output: Two victim partiitons $A_h$ and $B_h$.
Begin
  1. If $absolute(|A| - |B|) < b$        //Memory is balanced
       • S' = Set of pairs $(A_k, B_k)$, where $|A_k| \geq a$, $|B_k| \geq a$.
       • If S' $\neq \phi$, then S = S'
       • S' = Set of pairs $(A_k, B_k) \in S$, such that removing $(A_k, B_k)$ will not affect the memory balancing.
       • If S' $\neq \phi$, then S = S'
       • Return $(A_h, B_h) \in S$, where $|A_h|+|B_h|$ is maximum.
  2. If $|A| \geq |B|$
       • S = Set of pairs $(A_k, B_k)$, where $|A_k| \geq |B_k|$.
     else
       • S = Set of pairs $(A_k, B_k)$, where $|B_k| \geq |A_k|$.
  3. S' = Set of pairs $(A_k, B_k) \in S$, where $|A_k| \geq a$, $|B_k| \geq a$.
  4. If S' $\neq \phi$, then S = S'
  5. Return $(A_h, B_h) \in S$, where $|A_h| + |B_h|$ is maximum.
End.
```

Figure 8. The Adaptive Flushing Policy.

$18\% < 25\%$. Thus, the memory is considered balanced. Then, the search space is limited to the second (11,13) and third (13,10) bucket pairs where all buckets are of size ≥ 10. Since both bucket pairs do not affect the memory balancing with respect to b, we choose the bucket pair with largest total size (11,13). If the balanced factor is set to $b = 10$, while keeping $a = 10$, then the memory is considered unbalanced. The search space is limited to those bucket pairs with $|A_k| \geq |B_k|$, i.e., the third (13,10) and fifth (25,2) pairs. With the acceptable threshold $a = 10$, the search space is limited to only the third bucket pair (13,10). Notice that the idea of having the parameter a is to avoid selecting small buckets. Thus, if we set $a = 1$, while keeping $b = 10$, then the *Adaptive Flushing* policy would select the bucket pair (25,2).

5. Correctness of the Hash-merge Join Algorithm

In this section, we give a proof of correctness of the non-blocking *Hash-merge* join algorithm (HMJ). The correctness proof is divided into two parts: First, we prove that HMJ is complete i.e., all result tuples are produced. Second, we prove that HMJ is a duplicate-free join algorithm, i.e., output tuples are produced exactly once.

Theorem 1 *For any two sources A and B, HMJ produces all output results of $A \bowtie B$.*

Proof: Assume that $\exists(r, s) : r \in A, s \in B$, and (r, s) satisfies the join condition. However, the tuple (r, s) is not reported by HMJ. Since (r, s) satisfies the join condition, then $r \in A_h$ and $s \in B_h$. Assume that $r \in A_{hk}, s \in B_{hm}$, which means that r and s are in the kth and mth blocks of the hash buckets with value h, respectively. Then, there are exactly two possible cases:

Case 1: $k = m$. In this case, the flushing policy guarantees that the blocks A_{hk} and B_{hm} were in memory at the same time. If the data item r arrives before s, then r will be stored in bucket A_{hk} (Step 4 in Figure 3) without joining with s. Later when s arrives, it will probe the block A_{hk} (Step 3 in Figure 3), and join with r. Notice that we guarantee that r is still in memory at the arrival of s. Otherwise the condition $k = m$ is violated. The same proof is applicable when s arrives before r. Thus, the tuple (r, s) cannot be missed in case of $k = m$.

Case 2: $k \neq m$. In this case, one of the blocks A_{hk} or B_{hm} is flushed to disk before the other one is created. Thus, A_{hk} and B_{hm} are disk-based blocks. In the merging phase, all disk-based blocks are joined together using a refinement version of the traditional sort-merge join algorithm (Step 3a in Figure 5). Thus, the tuple (r, s) cannot be missed in the merging phase.

From Cases 1 and 2, we conclude that the assumption that (r, s) is not reported by the *Hash-merge* join algorithm is not possible. Thus, HMJ produces all output results.

$\square$

Theorem 2 *For any two sources A and B, HMJ produces all output tuples of $A \bowtie B$ exactly once.*

Proof: Assume that $\exists(r, s) : r \in A, s \in B$, and (r, s) satisfies the join condition. Assume that HMJ reports the tuple (r, s) twice. We denote such two instances as $(r, s)_1$ and $(r, s)_2$. Thus, we identify the following three cases:

Case 1: $(r, s)_1$ and $(r, s)_2$ are both produced in the hashing phase. Assume that the data item r arrives after s. Once the tuple r arrives, r probes the hash bucket of s and outputs the result $(r, s)_1$. Then, during the hashing phase, only newly incoming tuples are used to probe the hash buckets of r and s. Thus, the tuple (r, s) cannot produced again in the hashing phase.

Case 2: $(r, s)_1$ and $(r, s)_2$ are produced in the merging phase. Once the tuple $(r, s)_1$ is reported in the merging phase (Step 3a in Figure 5), then r and s are merged into bigger blocks with similar block numbers b. Step 3b in Figure 5 avoids the reporting of output tuples that comes from similar block numbers. Thus, the tuple $(r, s)_2$ cannot be produced in the merging phase.

Case 3: One of the tuples, say $(r,s)_1$, is produced in the hashing phase, while the other one is produced in the merging phase. Since $(r,s)_1$ is reported in the hashing phase, then we guarantee that the blocks that contain r and s are flushed into the disk at the same time. Thus, these blocks have the same block number. Similar to the proof of Case 2, blocks with similar numbers do not report any join results. Thus, the tuple $(r,s)_2$ cannot be produced by the merging phase.

From the above three cases, we conclude that the assumption that the tuple (r,s) is reported twice is not valid. $\square$

6. Experimental Results

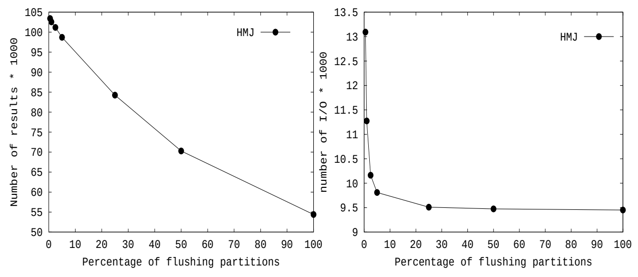
In this section, we give experimental evidence that HMJ is superior to other non-blocking join algorithms (e.g., XJoin [21] and the Progressive Merge Join (PMJ) [7]). The experiments in this section are divided into three categories:

- **Flushing policy.** The objective of this set of experiments is to study the effect of different flushing policies on the performance of HMJ.
- **Fast and reliable networks.** This set of experiments compares the performance of HMJ to both XJoin and PMJ in the case of fast and reliable networks, i.e., the sources are never blocked.
- **Slow and bursty networks.** This set of experiments compares the performance of HMJ to both XJoin and PMJ in the case of slow and bursty networks (i.e., the sources are subject to blocking).

All the experiments in this section are conducted on Intel Pentium IV CPU 1.4GHz with 256MB RAM running Linux 2.4.4. HMJ, XJoin, and PMJ are implemented using GNU C++. Unless mentioned otherwise, the data set involved in the join operation contains 1,000,000 tuples. Join keys are uniformity distributed in a range of 2,000,000 values. The memory size is set to accommodate 10% of the input data. For most of the experiments, we measure the time and I/O needed to produce the kth output tuple. For large output results, a typical user would be interested in only the first few results. Thus, the distribution of data and the join selectivity does not affect the experimental results since we are not interested in the whole result. Instead, we are interested only in the first few results.

6.1. Flushing Policy

In this set of experiments, we study two aspects related to the flushing policy implementation inside HMJ. First, we study the impact of the number of hashing buckets used in the hashing phase. Second, we study the impact of different flushing policies. The results are shown for the experiments performed in the case of fast and reliable networks. However, similar results are obtained when applying the same experiments to slow and bursty networks.



(a) Result (b) I/O

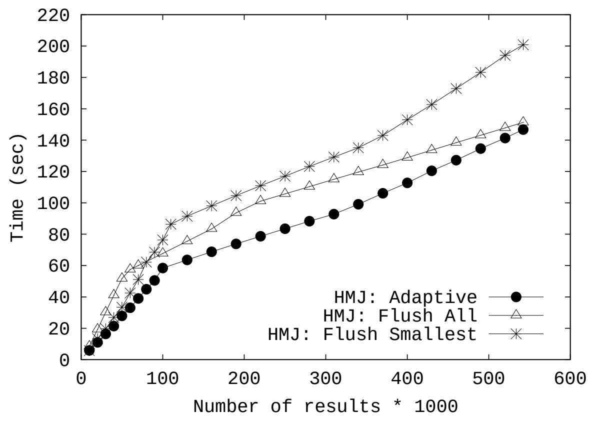
Figure 9. The impact of flushing size.

6.1.1 Number of Hash Buckets

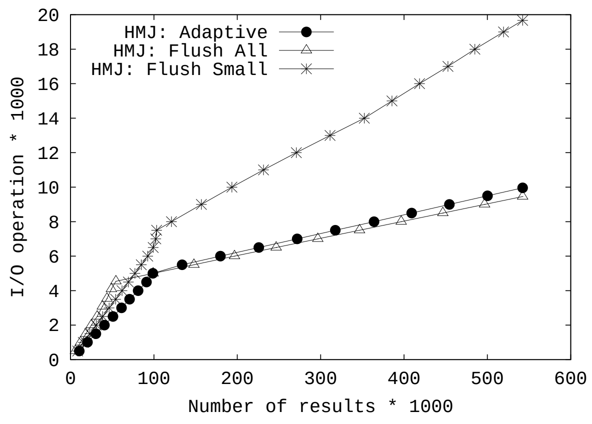
In these experiments, we aim to specify the best value for the parameter p; the percentage of the number of flushed buckets to the total number of hash buckets. (refer to Section 3.3). Experiments are performed using the *Adaptive Flushing* policy. Similar performance is achieved when using other flushing policies. Figure 9a gives the effect of varying p from 1% to 100% on the number of produced results from the hashing phase. As discussed in Section 4, as more in-memory buckets are flushed, there is a less chance of producing join results from the hashing phase, basically, because new incoming tuples have less chance to be joined with existing in-memory tuples. However, as given by Figure 9b, flushing more memory results in much less I/O overhead. As a compromise, setting p to about 5% achieves a trade-off between producing in-memory results in the hashing phase and less I/O in the merging phase. For the following experiments, we set p to 5%.

6.1.2 Different Flushing Policies

The experiment in Figure 10 is designed to test the impact of different flushing policies on the performance of HMJ. Mainly, we compare the *Flush All*, *Flush Smallest*, and *Adaptive Flushing* policies with respect to the time and I/O needed to produce the kth result. For the *Adaptive Flushing* policy, we set the acceptable bucket size a to be the average bucket size (i.e., M/h), where M is the memory size, and h is the number of hash buckets. The balancing factor b is set to be $M/5$. These values for a and b give the best performance for the *Adaptive Flushing* policy. Notice that



(a) Time



(b) I/O

Figure 10. Performance of different flushing policies.

we do not include the *Flush Largest* policy in our comparison, since the *Flush Largest* policy is a special case of the *Adaptive Flushing* policy by setting $a = 0, b = M$.

In Figure 10, we notice that all policies result in a plotting with almost two segments. The segment with higher slope indicates the join results that are produced in the hashing phase. The second segment with lower slope indicates the join results produced in the merging phase. For example, the *Adaptive Flushing* policy produces 100K tuples in the hashing phase. Figure 10a gives the time required to produce the kth result of HMJ. The *Adaptive Flushing* policy always outperforms the other policies. The *Flush All* policy produces less results during the hashing phase due to the fact that newly incoming tuples have less chance to be joined with in-memory tuples. Figure 10b gives the number of I/O's required to produce the kth result. For early join results (e.g., up to 100K results), the *Adaptive Flushing* policy has the best performance. However, during the merging phase, the *Flush All* policy slightly outperforms the *Adaptive Flushing* policy due to the fact that having large size buckets on disk would reduce the number of I/O's. For the rest of experiments, we use HMJ with the *Adaptive Flushing* policy.

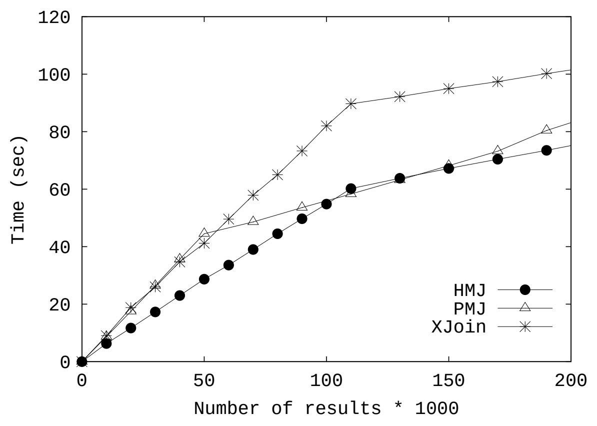
6.2. Fast and Reliable Networks

In this section, we consider input data from distributed sources with similar arrival rates via a fast and reliable network. Thus, there is no blocking behavior. In the experiments, we join two sources with 1M data items for each. The output result is around 550K tuples. However, we focus only in the first 200K results. Figure 11a gives the time required to produce the kth tuple. HMJ consistently outperforms XJoin and PMJ for up to 200K results. Both HMJ and XJoin produce 100K results during the hashing phases, while PMJ produces 50K results in the first phase (sorting phase). However, it takes around 90 seconds from XJoin to finish the hashing phase, while the hashing phase in HMJ takes around 60 seconds. The main reason for the efficiency of the hashing phase in HMJ is due to the flushing policy, where the flushed buckets are smartly chosen to keep room for having more in-memory join results.

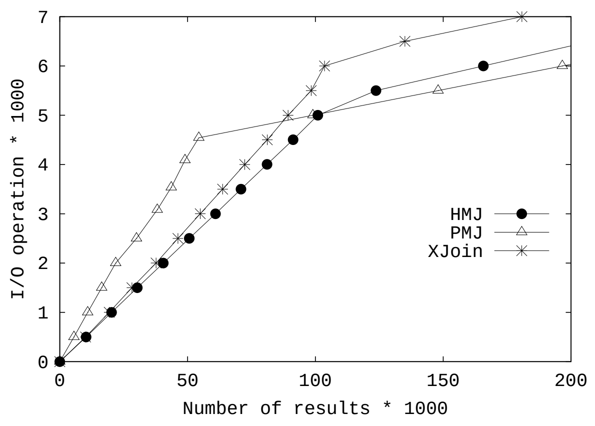
Figure 11b gives the number of I/Os required to produce the kth output tuple. For up to 100K, both HMJ and XJoin have less I/O than PMJ. This behavior is mainly because both HMJ and XJoin flush small buckets in their hashing phase rather than flushing the whole memory as in PMJ. Once the sorting phase of PMJ is done, large buckets are organized on disk. Thus, the number of I/Os in the merging phase of PMJ is less than the number of I/Os in the merging phase of HMJ. If we consider only producing early results up to 100K, then HMJ is clearly superior in terms of both time and I/O.

Figure 12 considers the case when the input data arrives from two sources with different arrival rates. The arrival rate of input data from source A is five times the arrival rate of data from source B. The results in the merging phase almost have the same behavior as in Figure 11. However, in the hashing phase, both HMJ and XJoin are more stable to the variations in arrival rates than PMJ. Also, unlike Figure 11, the hashing phase of HMJ is finished before the hashing phase of XJoin. The main reason is that using the *Adaptive Flushing* policy in HMJ always keeps the memory balanced even if the data arrival is not.

Figure 13 gives the time required to produce the first 1000 results. In this experiment, we vary the memory size



(a) Time



(b) I/O

Figure 11. Fast and Reliable Networks.

from 2% to 50% of the input data. We plot only HMJ against PMJ. XJoin has performance similar to that of HMJ as both algorithms rely on the original symmetric hashing join for producing the first few results. For very small memory sizes (less than 5%), PMJ needs to flush the whole memory more than once to get the first 1000 results. With the increase in memory size, the number of flushes decreases, thus better time is achieved. However, with the increase in memory, PMJ does not produce any results till the memory is exhausted. Thus, the time to fill the memory is increased with the increase of memory size. For HMJ, increasing the memory size does not affect the performance. In-memory join results are produced without the need to fill the memory.

6.3. Slow and Bursty Networks

In this section, we consider the case of slow and bursty networks. We assume that data arrives from the two sources A and B with Pareto distribution; a distribution that is widely used in case of slow and bursty networks [5]. A data source is considered to be blocked if no tuple arrives within a certain time threshold T. Figure 14a gives the time for producing the kth result. Both HMJ and PMJ have a step-like performance due to the switching between the first phase (i.e., the hashing phase in HMJ and the sorting phase in PMJ) and the merging phase. XJoin does not have such behavior because XJoin operates on three stages. The first stage is the hashing, while the second stage is joining between memory partitions with disk partitions. The third stage is a cleaning stage (starts after 200K tuple) that joins in-disk partitions. The third stage of XJoin takes control when the input data is finished.

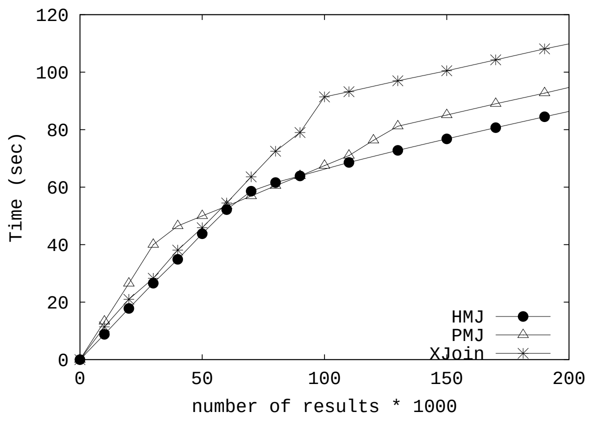
The overall performance of HMJ outperforms XJoin and PMJ. The main reason is the efficiency of the hashing phase in HMJ. This can be noticed from the slope of each segment in Figure 14a. Comparing PMJ with HMJ, the slope of segments that correspond to the first phase is lower for HMJ indicating that more results are produced. On the other side, for the segments that represent the merging phase, the slope of PMJ segments is lower than those of HMJ.

Although XJoin has the same hashing phase as that of HMJ, HMJ outperforms XJoin during the hashing phase. The main reason is that the *Adaptive Flushing* policy employed by HMJ keeps the memory balanced and makes use of the properties of the hashing phase. On the other side, the flushing policy employed by XJoin (flush the largest bucket from only one source) results in an unbalanced memory. Thus, the hashing phase of XJoin may not produce many results as in HMJ.

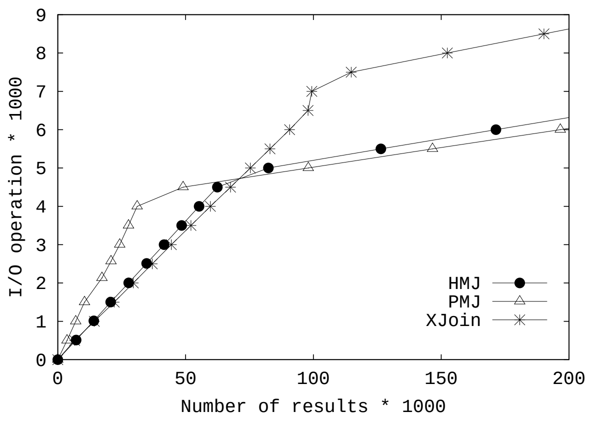
Figure 14b gives the number of I/Os needed to produce the kth result. HMJ has similar I/O performance as that of PMJ for the first 200K results. The main reason for the I/O performance of PMJ is that PMJ flushes large buckets into disk. XJoin clearly has the worst performance of I/O, mainly becuase of flushing small memory blocks into disk. Generally, when the interest is only in the earlier results, HMJ is better than XJoin and PMJ in both time and I/O performance.

7. Conclusion

This paper proposes the *Hash-merge* join algorithm (HMJ, for short); a non-blocking join algorithm for producing fast and early join results. HMJ works on environments where the data are coming from different sources via a slow



(a) Time



(b) I/O

Figure 12. Different arriving rates in fast and reliable networks.

and bursty network. HMJ can produce join results even if one or both sources are blocked. HMJ has two phases: The first phase (the hashing phase) is responsible for producing fast and early join results by employing a hash-based in-memory join algorithm. The second phase (the merging phase) is responsible for producing join results when the two input sources are blocked by employing a refinement version of the traditional in-disk sort-merge join algorithm. An elegant flushing policy (termed the *Adaptive Flushing* policy) is employed in HMJ to link both the hashing and merging phases. The *Adaptive Flushing* policy is responsible for flushing memory partitions into disk. The correctness of HMJ with respect to completeness (i.e., all output tuples are produced) and uniqueness (i.e., no duplicate results are produced) is proved. Comprehensive experimental results show that the performance of HMJ outperforms two state-of-the-art non-blocking join algorithms, XJoin [20, 21] and the progressive merge join (PMJ) [7, 8], in terms of the I/O and time needed to produce early join results.

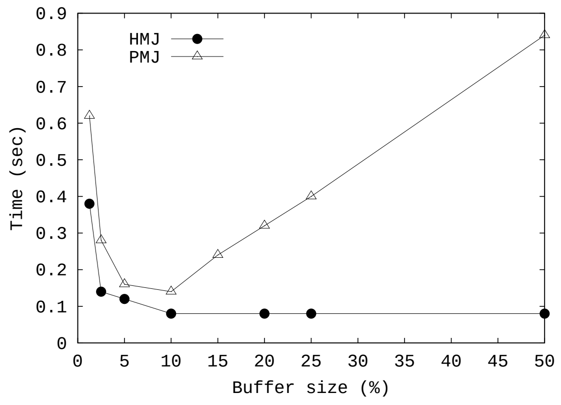


Figure 13. Producing the first 1000 results.

References

[1] L. Amsaleg, M. J. Franklin, A. Tomasic, and T. Urhan. Scrambling Query Plans to Cope With Unexpected Delays. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, PDIS*, pages 208–219, Miami, FL, Dec. 1996.

[2] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 261–272, Dallas, TX, May 2000.

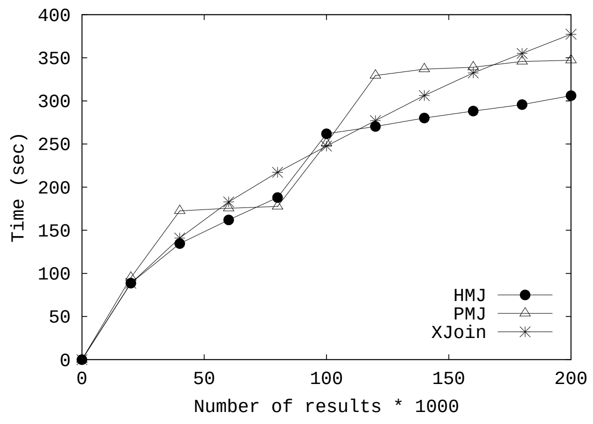
[3] R. J. Bayardo and D. P. Miranker. Processing Queries for First Few Answers. In *Proceedings of the International Conference on Information and Knowledge Managemen, CIKM*, pages 45–52, Rockville, MD, Nov. 1996.

[4] M. J. Carey and D. Kossmann. On Saying "Enough Already!" in SQL. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 219–230, Tucson, AR, May 1997.

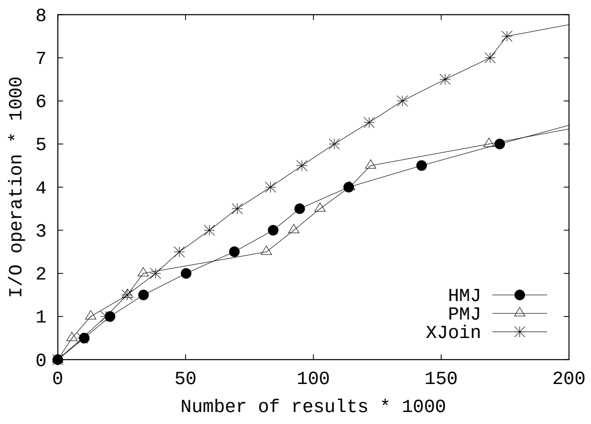
[5] M. E. Crovella, M. S. Taqqu, and A. Bestavros. *Heavy-tailed probability distributions in the world wide web*, chapter A practical guide to heavy tails: statistical techniques and applications, pages 3–26. Chapman Hall, 1998.

[6] D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM, CACM*, 35(6):85–98, 1992.

[7] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer. Progressive Merge Join: A Generic and Non-blocking Sort-based Join Algorithm. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 299–310, Hong Kong, Aug. 2002.



(a) Time



(b) I/O

Figure 14. Slow and Bursty Networks.

[8] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer. On Producing Join Results Early. In *Proceedings of the ACM Symposium on Principles of Database Systems, PODS*, pages 134–142, San Diego, CA, June 2003.

[9] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[10] P. J. Haas and J. M. Hellerstein. Ripple Joins for Online Aggregation. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 287–298, Philadelphia, PA, June 1999.

[11] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.

[12] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 171–182, Tucson, AR, May 1997.

[13] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An Adaptive Query Execution System for Data Integration. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 299–310, Philadelphia, PA, June 1999.

[14] G. Luo, C. J. Ellmann, P. J. Haas, and J. F. Naughton. A Scalable Hash Ripple Join Algorithm. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 252–262, Madison, WI, 2002.

[15] G. Luo, J. F. Naughton, and C. Ellmann. A Non-blocking Parallel Spatial Join Algorithm. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 697–705, San Jose, CA, Feb. 2002.

[16] P. Mishra and M. H. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113, 1992.

[17] A. Motro. Using Integrity Constraints to Provide Intensional Answers to Relational Queries. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 237–246, Amsterdam, The Netherlands, Aug. 1989.

[18] D. A. Schneider and D. J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 110–121, Portland, OR, May 1989.

[19] L. D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems , TODS*, 11(3):239–264, 1986.

[20] T. Urhan and M. J. Franklin. XJoin: Getting Fast Answers From Slow and Burst Networks. Technical Report CS-TR-3994, UMIACS-TR-99-13, Computer Science Department, University of Maryland, Feb. 1999.

[21] T. Urhan and M. J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.

[22] S. V. Vrbsky and J. W.-S. Liu. APPROXIMATE - A Query Processor that Produces Monotonically Improving Approximate Answers. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 5(6):1056–1068, 1993.

[23] A. N. Wilschut and P. M. G. Apers. Pipelining in Query Execution. In *Databases, Parallel, Architectures, and their applications*, Miami, FL, 1990.

[24] A. N. Wilschut and P. M. G. Apers. Datafbw Query Execution in a Parallel Main-Memory Environment. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems, PDIS*, pages 68–77, Miami, Florida, Dec. 1991.