CS-TR-3994, UMIACS-TR-99-13

XJoin: Getting Fast Answers From Slow and Bursty Networks

Tolga Urhan

Institute for Advanced Computer Studies
Computer Science Department
University of Maryland
College Park, MD 20742
urhan@cs.umd.edu

Michael J. Franklin

Institute for Advanced Computer Studies
Computer Science Department
University of Maryland
College Park, MD 20742
franklin@cs.umd.edu

Abstract

The combination of increasingly ubiquitous Internet connectivity and advances in heterogeneous and semi-structured databases has the potential to enable database-style querying over data from sources distributed around the world. Traditional query processing techniques, however, fail to deliver acceptable performance in such a scenario for two main reasons: First, they optimize for delivery of the entire query result, while on-line users would typically benefit from receiving initial results as quickly as possible. Second, slow or bursty delivery of data from remote sources can stall query execution, making the already inadequate batch-like behavior even worse. Both of these problems can be addressed using fully pipelined query execution. The symmetric hash join operator supports such pipelining, but it requires all base data and intermediate results to be memory-resident, which is unacceptable for complex queries over large datasets. In this paper we present a multi-threaded extension of the symmetric hash join, called XJoin, that can execute effectively with far less memory. By reactively scheduling background processing, XJoin hides intermittent delays in data arrival to produce more tuples earlier. XJoin includes a very efficient, on-the-fly algorithm for preventing duplicates from being created by its independently running threads. We have implemented the XJoin operator and added it to the PREDATOR Object-Relational DBMS. Using this implementation along with traces obtained by monitoring Internet data delivery, we show that XJoin is an effective solution for providing fast query responses to users even in the presence of slow and bursty remote sources.

1 Introduction

1.1 Wide-Area Query Processing

The explosive growth of the Internet and the World Wide Web has made tremendous amounts of data available on-line. Currently, searching for information in this huge database is usually done using navigational methods, such as following links, or by submitting a few terms to a search engine. Such limited querying capability arises in part, due to the lack of structure and semantics in most web data sources. Fortunately, however, this situation is beginning to change due to emerging standards such as XML, as well as to the development of technology for wrapping sources to provide relational-style interfaces. As a result, it is becoming possible to pose more sophisticated, declarative queries over data sources that are widely distributed across the Internet.

Beyond the issues of structure and semantics, however, there remain significant technical obstacles to building responsive, usable query processing systems for wide-area environments. A key performance issue that arises in such environments is *response-time unpredictability*. Data access over wide-area networks involves a large number of remote data sources, intermediate sites, and communications links, all of which are vulnerable to overloading, congestion, and failures. Such problems can cause significant and unpredictable *delays* in the access of information from remote sources. These delays, in turn, cause traditional distributed query processing strategies to break down, resulting in unresponsive and hence, unusable systems.

In previous work [AFTU96] we identified three classes of delays that can affect the responsiveness of query processing: 1) *initial delay*, in which there is a longer than expected wait until the first tuple arrives from a remote source; 2) *slow delivery*, in which data arrive at a fairly constant but slower than expected rate; and 3) *bursty arrival*, in which data arrive in a fluctuating manner, with bursts of data followed by long periods of no arrivals. With traditional query processing techniques, query execution can become blocked even if only one of the accessed data sources experiences such delays.

We developed Query Scrambling to address this problem and showed how it can be used to hide initial delays [UFA98] and bursty arrivals [AFT98]. Query Scrambling is a *reactive* approach to query execution; it reacts to data delivery problems by on-the-fly rescheduling of query operators and restructuring of the query execution plan. Query Scrambling is aimed at improving the response time for the *entire* query, and may actually slow down the return of some initial results in order to minimize the time required to produce the remaining portion of a query answer once all necessary data has been obtained from all of the remote sources.

In this paper, we explore a complementary approach, based on a non-blocking join operator we call XJoin. XJoin extends the symmetric hash join (SHJ) [WA90, HS93] to use secondary storage, which allows it to be used with large inputs and to run concurrently with other query operators in a bushy query plan. Simply extending SHJ to use secondary storage, however, is insufficient for tolerating significant delays in receiving data from remote sources. For this reason, a key component of XJoin is a *reactively scheduled* background process, which opportunistically utilizes delays to produce more tuples earlier. We show that by using XJoins it is possible to produce query execution plans that can better cope with data delivery problems and that can deliver initial results orders of magnitude faster than traditional techniques, with in many cases, little or no degradation in the time required to deliver the entire result.

1.2 Solution Overview

The XJoin approach is based on two fundamental principles:

1. *It is optimized for producing results incrementally as they become available.* When used in a fully pipelined query plan, answer tuples can be returned to the user as soon as they are produced. The early delivery of initial answers can provide tremendous improvements in the responsiveness of the system. Furthermore, in many situations, users require only a small subset of the total query answer [CK97], so returning initial results quickly is the key to system usability.

2. *It allows progress to be made even when one or more sources experience delays.* There are two reasons for this. First, by using less memory XJoin allows for bushier query plans than are possible with other pipelined join methods. Thus, some parts of a query plan can continue while others are stalled waiting for input. Second, by employing background processing on previously received input, an XJoin operator can run and produce results even when both of its inputs become blocked.

The symmetric hash join, on which XJoin is based, was aimed at addressing similar issues. As originally proposed, however, symmetric hash join requires that hash tables for both of its inputs be kept in main memory until all of the tuples have been received from both of its inputs. As a result, symmetric hash join cannot be used for joins with large inputs, and the ability to run multiple joins (e.g., in a bushy query plan) is severely limited. XJoin avoids these problems by allowing tuples from one or both of the inputs to be temporarily spooled to secondary storage. In a sense, XJoin provides for symmetric hash join, the same flexibility that the hybrid hash join provides for the classic hash join [Sha86]. Not surprisingly, similarly to hybrid hash join it is based on partitioning.

The main challenges in developing XJoin include the following:

- Managing the flow of tuples between memory and secondary storage.
- Controlling the background processing that is initiated when inputs are delayed.
- Ensuring that the full answer is ultimately produced (i.e., no answers should be lost).
- Ensuring that no duplicate tuples are inadvertently produced.

In this paper, we describe the design and performance of XJoin, focusing on how it addresses the above issues. In order to evaluate its efficiency and its ability to deal with delays we have implemented it in the context of the PREDATOR Object-Relational DBMS [SP97]. Using network traces recorded by transferring data from various remote sites in the Internet along with workloads derived from the Wisconsin benchmark we show that XJoin is indeed an effective solution for providing fast query responses to users in the presence of slow and bursty remote sources.

The work described in this paper is related to other recent projects on improving the responsiveness of query processing, including techniques for returning initial answers more quickly [BM96, CK97] and those for returning continually improving answers to long running queries [VL93, HHW97]. Our work differs from this other research due to (among other reasons) the focus on coping with unpredictable delays arising from wide-area remote data access. As mentioned previously, the XJoin approach is complementary to our earlier work on Query Scrambling for unpredictable delays in distributed query processing [AFTU96, AFT98, UFA98], as well as to other dynamic approaches such as [ONK+97, TRV96]. A recent paper [IFFL+99] describes the Tukwilla system that contains an operator similar to XJoin, which can adapt to limited memory, but that differs from XJoin in several ways. Most importantly that operator does not include the reactively scheduled background processing of XJoin (called the "second stage"), which is a key to its performance. This is and other related work is discussed in more detail in Section 5.

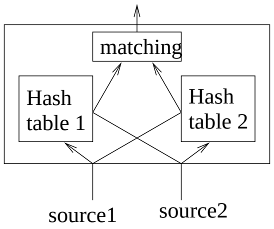

Figure 1: Symmetric Hash Join [WA90,HS93].

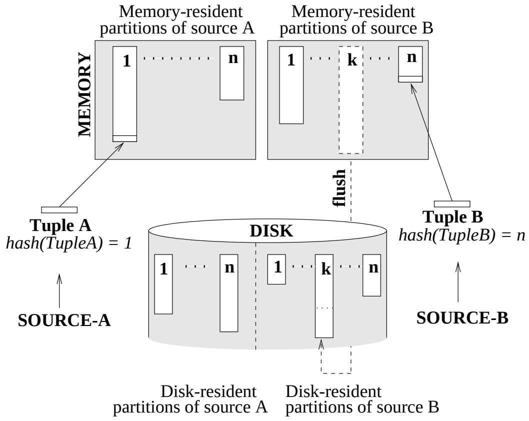

Figure 2: Handling the partitions.

2 XJoin

XJoin is based on the Symmetric Hash Join (SHJ) [WA90, HS93], which was originally designed to allow a high degree of pipelining in parallel database systems. Unlike traditional hash joins, which build a hash table on one input and then probe it with tuples from the other input, SHJ (shown in Figure 1) builds two hash tables, one for each source. When a tuple arrives on one of the inputs, it is first inserted into the hash table for that input, and then immediately used to probe the hash table of the other input. Thus, even if one source becomes temporarily blocked, the operator can still produce results.

It may not be immediately apparent that the algorithm is correct, i.e. that 1) it produces all the tuples in the result, and 2) it produces no extra duplicate result tuples. The correctness of symmetric hash join is due to the fact that probing, which produces the output tuples, is performed for *all* matching pairs once and only once, namely when the later-arriving tuple of the pair is processed. As we will see, preserving these correctness properties in XJoin requires some additional attention, due to its more asynchronous nature.

XJoin extends the symmetric hash join to use less memory by allowing parts of the hash tables to be moved to secondary storage. It does this by partitioning its inputs, similar to the way that hybrid hash join solves the memory problems of classic hash join.

2.1 Partitioning

XJoin splits both of its inputs into a number of partitions based on a hash function.[1] Each partition is composed of a *memory-resident* portion and a *disk-resident* portion. The memory-portion contains the tail (i.e., recently arrived tuples) of the partition, and the disk-resident portion contains the rest. The memory-resident portions are maintained as *hash tables* as in symmetric hash join. Each memory-resident portion (for *both* sources) has at least one block of memory reserved for it at all times. The remaining memory (if any) is divided evenly between the two sources and is used to allow the memory-resident portions to grow as tuples arrive from the sources.

When XJoin receives a tuple from one of its sources, it inserts the tuple into its corresponding partition, which is found by applying a hash function to its join attribute (Figure 2). When the memory becomes full,

[1] The number of partitions is determined by using the formula $\sqrt{F \times ||R||}$ where F is the "fudge" factor, and $||R||$ is the number of pages in the smaller input [Sha86].

the tuples of the partition with the largest memory-resident portion are written to the disk. This grows the size of the disk-resident portion (e.g., partition k of source B in Figure 2). The memory-resident portion of that partition is then reduced to a single (initially, empty) block, and the remaining free blocks are made available for use by any of the partitions as new tuples arrive. The flushing process is repeated whenever the memory becomes full.

In the remainder of the paper we use the following notion: P_{iA} denotes the ith partition of source A. The memory and disk-resident portions of P_{iA} are referred to as MP_{iA} and DP_{iA} respectively. Thus, we always have $MP_{iA} \cup DP_{iA} = P_{iA}$ and $MP_{iA} \cap DP_{iA} = \emptyset$. The two input sources will be referred to as A and B.

2.2 The Three Stages of XJoin

XJoin proceeds in three stages, each of which is performed by a separate thread. The first stage joins tuples in the memory resident portions of the partitions, acting similarly to the standard symmetric hash join. The second stage joins tuples from disk with tuples that have not yet been flushed to disk. The third stage is a clean-up stage, which performs any necessary matching to catch any results missed by the first two stages. The first and second stages run in an interleaved fashion — the second stage takes over when the first becomes blocked due to a lack of input. These stages are terminated after all input has been received, at which point the third stage is initiated.

If care is not taken, spurious duplicate result tuples can be produced by the interaction of the three stages. In order to prevent duplicates XJoin employs a fast, on-the-fly duplicate detection mechanism, which will be presented in Section 2.3. We now describe the three stages in more detail.

First Stage

The first stage works similarly to the original symmetric hash join. The main difference is that in XJoin, the tuples are organized in *partitions* based on their join attribute values. Figure 3 shows an example of the first stage when a tuple is received from $Source_A$ (respectively $Source_B$) that hashes to partition i (resp. partition j). If there is room for the tuple in memory, then the tuple is stored in its partition and used to probe the memory-resident portion of the corresponding partition for the other source. Any matches that occur are output as results. If, however, memory is full, then one of the memory-resident portions is chosen as a victim and flushed to disk as described in Section 2.1. Join processing then continues as usual. The first stage runs as long as at least one of its inputs is producing tuples. When the first stage times-out on *both* of its inputs (e.g., due to some unexpected delays), it blocks and the second stage is allowed to run. The first stage terminates when it has received all of the tuples from both of its inputs.

Second Stage

The second stage (shown in Figure 4) is activated whenever the first stage blocks. It picks a disk-resident portion of one of the partitions[2], say DP_{iA} and uses its tuples to probe the memory-resident portion of the corresponding partition of the other source (i.e., MP_{iB}, in this case). Any matches found are output (subject

[2] The partition is chosen using optimizer-generated estimates of the output cardinality and the cost of performing the stage using the partition.

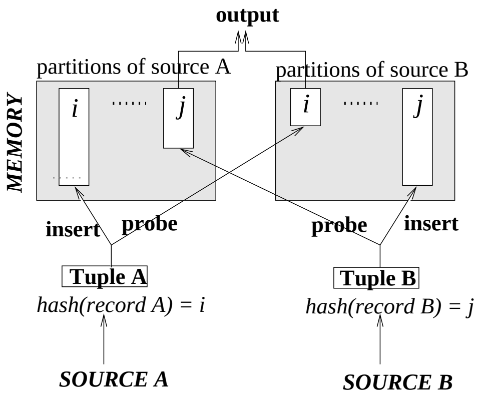

Figure 3: Stage 1 - Memory-to-Memory joins

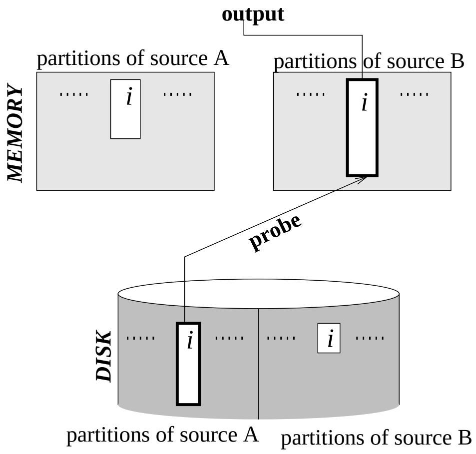

Figure 4: Stage 2 - Disk-to-Memory joins

to duplicate detection as described in Section 2.3) as result tuples. After a disk-resident portion has been completely processed, the operator checks to see if either of the join inputs are ready to begin producing tuples again, if so, then the second stage halts and the first stage is resumed. If not, then a different disk-resident portion is chosen and the second stage is continued.

In contrast to the first stage, this stage does not depend on the availability of tuples from the sources. It deals with the tuples that have already been stored locally (either in the memory or on the disk) and can produce output even when both relations are delayed. The second stage may use the same partition multiple times, as the partition grows over the course of the join execution. Once the second stage begins processing a disk-resident portion of a partition, it runs until that portion is completely processed. Only after the partition has been processed does the operator check to see if either of the input sources has become unblocked. If so, then the first stage is resumed, otherwise, the second stage continues by choosing another partition to process. This coarse-grained approach simplifies the implementation of the second stage, but it also raises some performance risks. The second stage incurs overhead in the hope of generating result tuples. This overhead is essentially “free”, as long as the inputs of the XJoin are delayed, as no progress could be made in that situation anyway. If, however, one or both of the inputs becomes unblocked, the additional costs of the second stage could delay the delivery of results. This aspect of the second stage is investigated in the experiments described in Section 4.

Third Stage

The third stage executes after all tuples have been received from both inputs. It is a clean-up stage that makes sure that all the tuples that should be in the result set are ultimately produced. This step is necessary because the first and second stages may only partially compute the result. The first stage fails to join tuples that were not in the memory at the same time. If two matching tuples are received far apart in time, one of them might have already been flushed to disk by the time the other tuple arrived. Such pairs of tuples cannot be joined by the first stage. The second stage similarly fails to join two tuples if one of them is not in the memory when the other is brought from the disk and used to probe the memory-resident tuples.

The third stage joins all the partitions (both the memory-resident and disk-resident portions) of the two sources. It distinguishes between the smaller and the larger of the input relations while operating. Assume source A produces the smaller input. The third stage first brings all the tuples of DP_{1A} into the memory and creates a hash table for them. Of course, this process might require some memory-resident portions of other partitions to be flushed to disk. The hash table is then probed with the tuples from both MP_{1B} and DP_{1B}. This process is repeated for all the remaining partitions. The operation of the third stage is similar to the processing done by hybrid hash join after it has partitioned its inputs. The third stage typically will perform less work, however, because it produces only those result tuples that haven't already been produced by the first and second stages, which can result in significant savings in CPU usage.

2.3 Handling Duplicates in XJoin

As stated in the previous section, the multiple stages of XJoin may produce spurious duplicate tuples because they can potentially perform overlapping work. Duplicates can be created in both the second and third stages. The first stage does not generate spurious duplicates for the same reason that duplicates are not created by the original symmetric hash join, namely, that it matches pairs of memory resident tuples only at the instant that the later of the two tuples arrives from its source.

The creation of duplicates in the second and third stages and the mechanism used to avoid these problems are described in the following two sections. The duplicate prevention mechanisms rely on timestamp values that are maintained by the XJoin operator. Timestamps are implemented using a counter whose value is incremented every time a new tuple is received from a source or flushed to the disk.

XJoin augments the structure of tuples to contain two timestamps that are set during the first stage. One timestamp, called the arrival timestamp (ATS) is assigned when the tuple is first received from its source. The second timestamp, called the departure timestamp (DTS) is assigned when the tuple is flushed from memory to make room for additional input as described previously. The ATS and DTS together describe the time interval during which a tuple was in the memory-resident portion of its partition. The ATS and DTS of a tuple are never changed once they are assigned.

2.3.1 Detecting duplicates in the second and third stages

As stated previously, duplicates cannot be generated in the first stage, but both the second and third stages do have the potential of creating duplicates. Tuples that are matched during these stages could have been matched by the first stage, or by a previous run of the second stage. Checking for the matches from first stage is easy. For a pair of tuples to have been matched by the first stage they both must have been in the memory at the same time, thus they must have overlapping ATS and DTS ranges. Any such pair of tuples are not considered for joining by the second or third stages. Figure 5 shows two cases. The tuples in Figure 5(a) have overlapping ranges (in fact $Tuple_A$ was in the memory the entire time that $Tuple_B$ was in memory) so these tuples should not be joined again by the later stages. In contrast, the tuples in Figure 5(b) were never in the memory at the same time during the first stage so they could not have been joined during the first stage.

The ATS and DTS can be used to avoid duplicates of results produced by the first stage, but they do not solve the potential for duplicates created by multiple runs of the second stage. Recall that the second stage uses tuples from one source that have been evicted from memory to probe the memory-resident portion of the corresponding partition of the other source. If a disk-resident tuple is used by several different runs of the second stage, then spurious duplicates could be created. This latter problem is solved by using timestamps to record the times that the second stage has used each disk-resident portion.

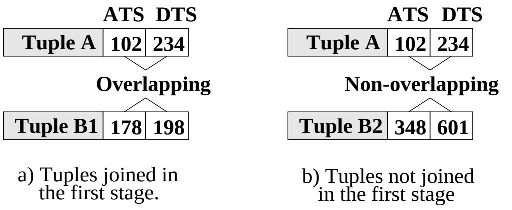

Figure 5: Detecting tuples joined in 1st stage

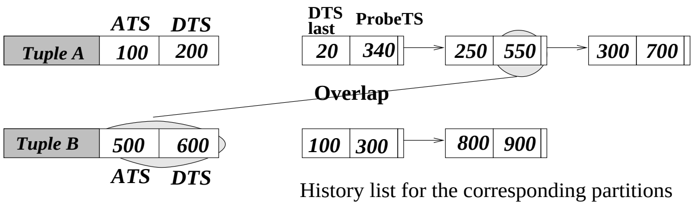

Figure 6: Detecting tuples joined in 2nd stage

These additional timestamps are recorded in a linked list associated with each disk-resident portion. The linked list contains an entry for each time the second stage has processed that portion. The entries in the list are of the form $\{DTS_{last},\ ProbeTS\}$ where DTS_{last} is the DTS value of the last tuple of the disk-resident portion that was used to probe the memory-resident tuples, and $ProbeTS$ is the timestamp value at the time that the second stage was executed[3]. These entries can be used to infer that all tuples of disk-resident portion having DTS values up to (and including) DTS_{last} were used by the second stage at time $ProbeTS$.

Checking for tuples matched in the second phase is then performed as follows: For two matching tuples $Tuple_A$ and $Tuple_B$, belonging to partitions DP_{iA} and MP_{iB} respectively, we first scan the linked list constructed for DP_{iA} to find out if and when $Tuple_A$ was used to probe the memory-resident tuples. Then we check whether $Tuple_B$ was in the memory during one of these probings, by using its ATS and DTS values. If so we deduce that $Tuple_A$ was already matched against $Tuple_B$ so these two tuples are not matched again. Note that this same approach is used to avoid generating spurious duplicates in the third stage as well, by performing the check symetrically (i.e., in both directions).

Figure 6 shows an example of the use of the linked lists. From the top list, it can be deduced that $Tuple_A$ was matched against memory-resident tuples twice, at times 550 and 700. By examining the ATS and DTS of $Tuple_B$, it is learned that $Tuple_B$ was in the memory from time 500 to 600. Thus, $Tuple_B$ must have already been matched (by the second stage) with $Tuple_A$ at time 550 so they should not be matched again.

2.4 Further optimizations

2.4.1 Adding a cache

The previous sections described the basic XJoin algorithm and the techniques it uses to avoid generating spurious duplicate result tuples. In this section we discuss an important optimization that can be applied to the second stage. Recall that the second stage picks a disk-resident portion of a partition, say DP_{iA}, and joins it with MP_{iB}. In the basic algorithm the tuples of DP_{iA} are discarded after they are used. If instead, some tuples from DP_{iA} could be retained (i.e., cached) in memory, then a subsequent run of the second stage joining DP_{iB}

[3] Note that the timestamp value remains unchanged during an execution of the second stage since no tuples can be added to or evicted from memory while it is executing.

and MP_{iA} could also use the cached tuples from DP_{iA}.

Figure 7 shows two consecutive runs of the second stage with the caching optimization enabled. In the first run the second stage algorithm reads DP_{iA} from the disk and uses it to probe MP_{iB}. Some of its tuples are also stored in the cache. In the next run of the second stage DP_{iB} is read from the disk and joined with MP_{iA} and with the tuples of DP_{iA} that have been stored in the cache.

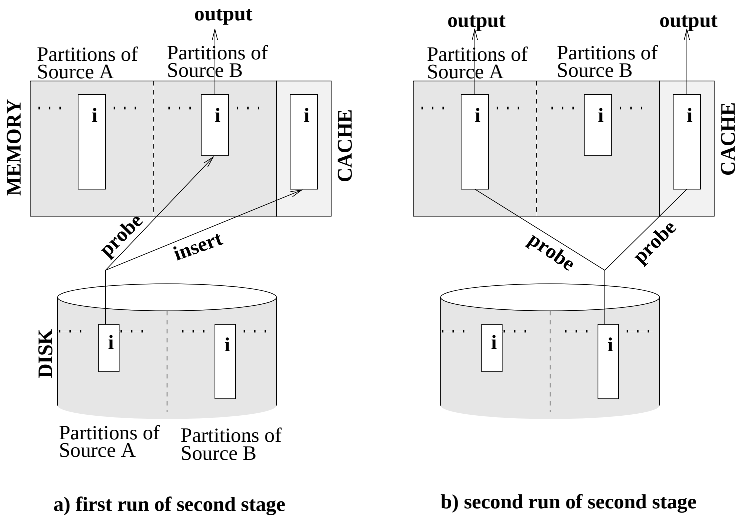

Figure 7: Two consecutive runs of the second stage with caching.

This optimization has the potential to produce a significant number of additional result tuples at the expense of some extra space allocated to the cache.[4] This optimization, however, introduces a third potential point of duplicate creation. In the following, we describe a very simple and fast algorithm for avoiding such duplicates.

The algorithm is based on the observation that the join between the cached tuples from a disk-resident portion DP_{iA} and the tuples from DP_{iB} can be thought of as a join between two *intervals*. One interval summarizes which tuples of DP_{iA} were in the cache, and the other interval summarizes which tuples of DP_{iB} were used to probe the cache. Such intervals can be represented by storing only two timestamp values for each partition: The DTS of the last tuple from DP_{iA} that was able to get into the cache, and the DTS of the last tuple from DP_{iB} that was used to probe the cache. We call these two values DTS_{cache} and DTS_{probe}.

If we picture the DTSs of the tuples from these two partitions being laid out as the axes of a matrix, the DTS_{cache} and DTS_{probe} define a rectangular region on this matrix which is justified to the origin (Figure 8). In the second and third stages, any matching pair of records whose DTS values are found to fall into this rectangle are not joined again. What is nice about keeping this history as a rectangular region is that when two partitions are joined the same way again the new rectangular region will completely overlap the previous region. This is because at least as many tuples as were cached in the previous run will be able to get into the cache. Also at least as many tuples of the probing partition as were used in the previous run will be used again this time. Thus the rectangular region grows monotonically larger, overlapping the previous one. This property makes it safe to keep only one interval per partition of each source.

[4] In our implementation we use 10% of the memory allocated to XJoin for the cache. If the cache is not large enough to hold all the tuples of a disk-resident portion only its *leading* tuples are stored.

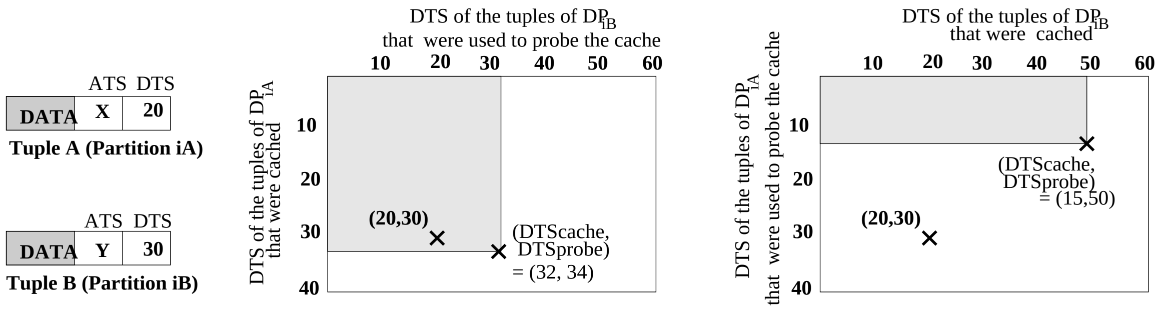

Figure 8: Detecting the disk-to-cache joins.

Figure 8 shows the case where two tuples from partitions DP_{iA} and DP_{iB}, with DTS values of 20 and 30 respectively, are about to be joined. Two rectangular regions have already been constructed: The first rectangle summarizes the tuples joined when DP_{iA} was cached and probed with DP_{iB}, and the second rectangle shows when DP_{iB} was cached and probed with DP_{iA}. The first rectangle denotes that tuples of DP_{iA} having DTS values up to 32 were cached and probed with tuples of DP_{iB} having DTS values up to 34. These values are 15 and 50 respectively for the second rectangle.

When two tuples, $Tuple_A$ and $Tuple_B$, are about to be matched we first check (using the first rectangle), if $Tuple_A$ was cached and $Tuple_B$ was used to probe it. The second rectangle is then used in the same manner to check if $Tuple_B$ was cached and was probed by $Tuple_A$. In the example of Figure 8, the point represented by the DTSs of these two tuples,i.e. (20,30), is found to fall within the first rectangle, thus, we deduce that $Tuple_A$ was cached and $Tuple_B$ was used to probe it, so these tuples are not joined again.

2.4.2 Controlling the second stage

Recall that the main idea behind the second stage is to perform additional processing *during* delays in order to produce more tuples. As will be seen in Section 4, this reactive operation has the potential to dramatically improve the speed with which results are provided to the user. As described in Section 2.2, however, the overhead incured by the second stage is hidden only when both inputs to the XJoin experience delays. Furthermore, due to the complexities of scheduling and duplicate detection, the operation of the second stage is fairly coarse grained; once processing begins on a partition, that processing is run to completion before the first stage can be resumed. As a result, there is a tradeoff between the agressiveness with which the second stage is run, and the benefits to be obtained by using it.

To address this tradeoff, our implementation includes a mechanism that can be used to restrict the second stage to processing only those partitions that are likely to yield a significant number of result tuples. This *activation threshold* is specified as a percentage of the total number of result tuples expected to be produced from a partition during the course of the entire join. For example, if the join of a partition is expected to contribute 1000 tuples to the result, a threshold value of 0.01 will allow the second stage to process the partition as long as it is expected to produce 10 or more tuples. Thus, a lower activation threshold results in a more agressive use of the second stage. The calculations of the expected numbers of tuples are performed using statistics as well as the linked lists which store the history of the execution of the second stage for the partitions.

3 Experimental Environment

In order to study the performance of XJoin we have implemented it in an extended version of PREDATOR [SP97], an Object-Relational DBMS that uses SHORE as its underlying storage manager. In previous work We extended PREDATOR to support Query Scrambling [UFA98]. Most of the relational engine has been modified to support both a data-driven and a control-driven execution model, and to process arbitrary bushy plans. In addition to adding the XJoin operator itself, we extended the query optimizer to account for the operator and to provide some of the statistics and calculations required by XJoin.

Because performing experiments directly on the Internet would not provide repeatable results, we instead modeled the behavior of the network using trace data that could be easily replayed. These traces provide detailed information about the characteristics of data transfer as perceived by the receiver. It includes connection delays, when and how much data was received, the silent periods between the data transfers, etc. In order to obtain these traces we performed experiments on the Internet by fetching large files (such as images, large texts etc) from 15 randomly chosen sites. The sources were sampled every hour for two days, and statistics were collected. We have seen that the sources exhibited widely different degrees of burstiness and we saw average transfer rates that ranged between 5 KBytes/sec and 180 KBytes/sec with most values occurring between the 20 KBytes/sec and 120 KBytes/sec.

From the arrival patterns we collected, we chose two as representatives of the behavior of a bursty and a fast source (figures 9, and 10). The arrival patterns in these figures show the quantity of data is received at the query site. Each trace is bucketized by dividing the time axis into 100 buckets and aggregating the amount of data transferred within each bucket. Each impulse gives the magnitude of the data transfer (in bytes) during the corresponding time bucket. The bursty and fast patterns have average transfer rates of 23.5 KBytes/sec. and 129.6 KBytes/sec. respectively. Since the first pattern has a slow overall transfer rate we refer to it as the "slow" arrival pattern.

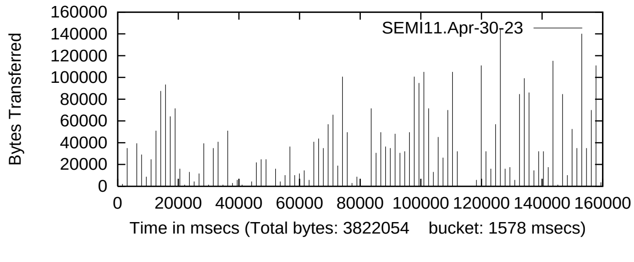

Figure 9: Bursty arrival. Avg. Rate 23.5 KBytes/sec.

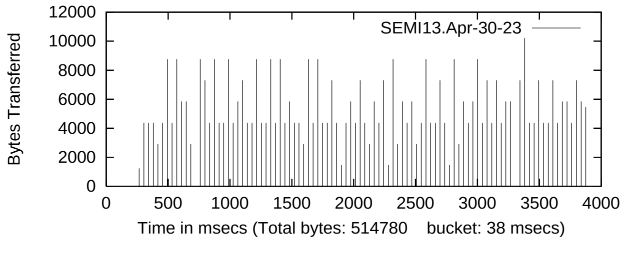

Figure 10: Fast arrival. Avg. Rate 129.6 KBytes/sec.

The arrival behavior of a remote relation is modeled by using one of the network traces in the following manner. First the network trace is read and stored in memory (network traces are usually small). The tuples of the relation are then generated on-the-fly in the PREDATOR *scan* operator, but rather than sending tuples of the relation directly to the parent (i.e., join) operator, the *scan* introduces artificial delays corresponding to the arrival pattern in the chosen network trace.

In the experiments we use a database containing up to six 100,000 tuple Wisconsin benchmark relations [BDT83]. Each tuple is 288 bytes for a total of 28.8 MB per relation. For some of the experiments we project these tuples down to 86 bytes or 8.6 MB total. Each relation is populated according to the Wisconsin benchmark specifications, using different random seeds. All of the experiments described here use a unique, unclustered integer attribute (unique1) as the join attribute so the result cardinality of all of the queries is also 100,000 tuples.

We ran the experiments on a Sun Ultra 5 Workstation running Solaris 2.6, with 128 MBytes of real memory, and approximately 4 GBytes of disk space. Each disk and memory page is 8KBytes. In all the experiments the SHORE (i.e., storage manager) buffer size was set at 800 Kbytes; the amount of main memory allocated to each XJoin operator is 3 MBytes, unless noted otherwise.

4 Results

In this section we investigate the performance of XJoin and compare it to that of Hybrid Hash Join (HHJ).[5] We examine three variations of XJoin, in order to separate out the contributions of the major components of the algorithm. The first variation, labeled *XJoin-No2nd*, does not use the second stage at all. Thus, it simply runs the first stage (flushing tuples to disk as necessary) until all input tuples have been received, at which point the third stage is initiated. The second variation, labeled *XJoin-NoCache* uses second stage, but without the caching optimization described in Section 2.4.1. The third variation, labeled *XJoin*, is the full implementation of XJoin as described in Section 2 (i.e., including the second stage with caching).

We have examined two variations of HHJ: one in which tuples from remote sources are fetched only when they are required by the operator, and one in which tuples from *all* of the remote sources are fetched in parallel, and those that are not needed when they arrive are temporarily spooled to the local disk. Reading the tuples in the background reduces the amount of time that the operator must spend waiting for the tuples to arrive (especially for slow sources) so the latter approach was found to perform better in all of our experiments. Thus, the results we report for HHJ in this section are those for the version that reads all inputs in parallel. This improvement to HHJ levels the playing field since XJoin also inherently accesses its inputs in parallel.

4.1 Experiment 1 - Basic performance of XJoin

In the first set of experiments, we examine the basic performance of XJoin and HHJ using a simple two-way join query and four different delay scenarios. The join in this query is the 1-to-1 join of 100,000-tuple Wisconsin relations described in the previous section. The delay scenarios are constructed by applying the four combinations of the slow and fast arrival patterns shown in Figures 9, and 10 to the two input relations. In this experiment, the join operators are allocated 3 MBytes of memory, and the input relations contain 28.8 MBytes each. Note for this first set of experiments, the *activation threshold* for the second stage is set to 0.01, which is a fairly aggressive setting. This threshold is the focus of the second set of experiments, described in Section 4.2.

Figures 11 thru 14 show the cumulative response times for the four algorithms (the three variations of XJoin

[5] We also compared the performance of XJoin to the basic Symmetric Hash Join (SHJ). In the case where there is sufficient memory to run SHJ, XJoin was found to perform nearly identically to SHJ. When there is less memory (as is the case in the experiments reported here) SHJ was found to thrash, at which point it became impractical to use. Thus, we do not report results for SHJ here.

plus HHJ) as result tuples are produced for the four combinations of network traces. The x-axis shows a count of the result tuples produced (from one to 100,000) and the y-axis shows the time at which that result tuple was produced. As can be seen in the figures, in all cases the three variants of XJoin produce the first answers several orders of magnitude faster than HHJ, thereby providing far superior interactive performance. Perhaps even more surprisingly, the XJoin variations are competitive even in terms of the completion time in most of the cases. Among the XJoin variants, for cases with at least one slow source, the full XJoin algorithm performs best, and the variant without the second stage performs worst the entire execution of the query. In fact, in these cases, XJoin-No2nd provides little if any advantage over HHJ after the delivery of the first 10% of the answer. These results demonstrate the importance of the reactive background processing of the second stage for coping with delays and bursty arrival. In the case where both sources are fast, this ordering of the XJoin variants holds for the delivery of the initial results, but inverts for the later results. In fact, for this last case, HHJ performs somewhat better than XJoin and XJoin-NoCache for the delivery of the second half of the result. This latter result demonstrates the pitfalls of overly aggressive use of the second stage in cases where tuple delivery is relatively fast. These results are described in more detail in the following sections.

4.1.1 Slow Network

Figure 11 shows the case when both sources are slow. HHJ starts producing tuples very late, since it waits for the entire build relation to arrive before it produces any tuples. It then produces the remaining tuples fairly quickly, since it has already prefetched most of the probe relation while processing the build relation. In contrast to HHJ, all of the XJoin variants produce all the tuples faster than HHJ, with several orders of magnitude improvement in getting the initial results.

A comparison of the XJoin variants reveals the benefits of the second stage and of using a cache. XJoin-No2nd, which does not use the second stage, performs comparably to the other two XJoin variants for the first few results, but its performance quickly degrades. The first stage stays active until the 850th second, i.e. until both sources arrive completely. During this time it produces only about 9000 tuples due to the limited memory. Recall that the first stage can only join tuples while they are memory resident. In this case, only about 5% of the tuples can be memory resident at a given time. The third stage takes over after all the input has arrived and produces the remaining 91000 of the tuples in 45 seconds. Thus, although XJoin-No2nd still performs better than HHJ it produces the majority of the tuples only after a large latency.

The effect of enabling the second stage is shown by the XJoin-NoCache curve. In this case the second stage is interleaved with the first stage and both stages are executed in this fashion until the 851th second. In the same time it takes XJoin-No2nd to produce only 9000 tuples XJoin-NoCache produces about 77000 tuples. Therefore the second stage considerably boosts (by a factor of about 8.5) the performance of XJoin in this case. Recall that the advantage of the second stage is that it joins memory-resident tuples with disk-resident tuples while waiting for additional input to arrive. Without the second stage, these joins would be delayed until all input was received. Comparing the XJoin and XJoin-NoCache lines, it can be seen that adding a cache to the second stage provides further improvement in this case by allowing additional tuples to be joined during delays.

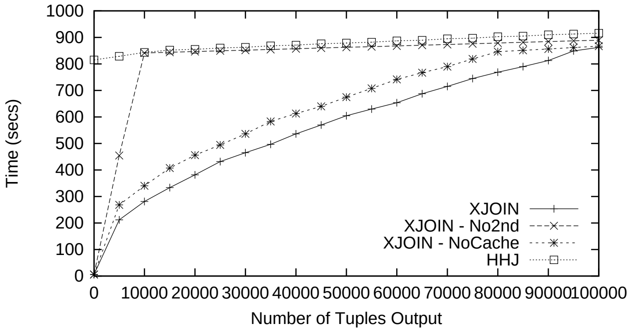

Figure 11: Slow build, Slow probe

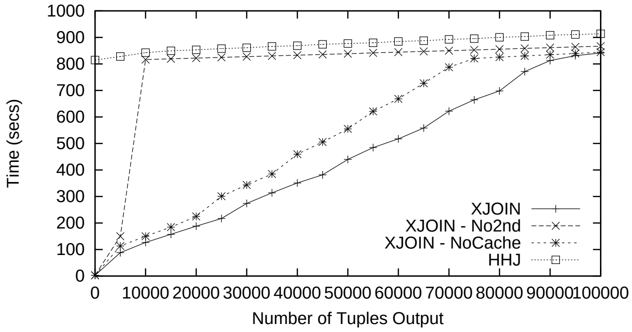

Figure 12: Slow build, Fast probe

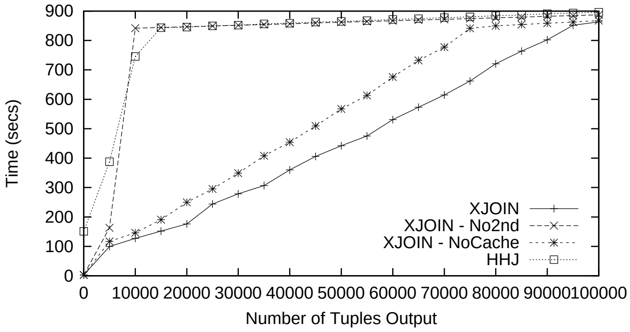

Figure 13: Fast build, Slow probe

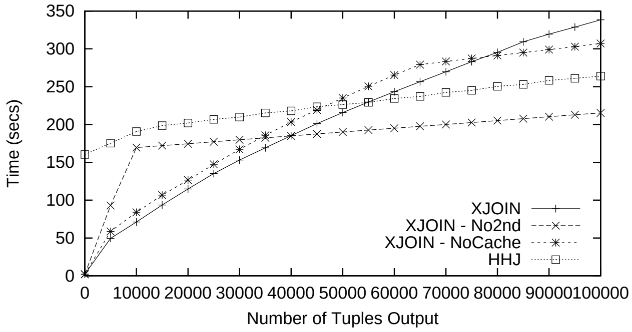

Figure 14: Fast build, Fast probe

In this case, for example, the first 5,000 tuples are produced 29% faster when a cache is used.

Interestingly, the performance benefits of XJoin carry over even to the last tuple delivered in this case, indicating that the third stage of XJoin is as fast or faster than the later stages of HHJ once all input tuples have arrived. The reasons for this are twofold. First, recall that HHJ spools tuples of the probe relation to disk until they are needed. Thus, the probe relation tuples must be partitioned before they can be used. If this overhead were removed (which, conceivably, it could be), HHJ would produce its last tuple slightly faster, namely at about the time that XJoin-No2nd completes, but still later than the other XJoin variants. The more fundamental difference is that due to the tuples produced by the second stage, XJoin-NoCache and XJoin both produce far fewer tuples in the third stage. The creation of result tuples incurs CPU overhead; incurring this overhead in the second stage allows it to be overlapped with the arrival delays, while in contrast, incurring the overhead after all tuples have arrived (as is done by XJoin-No2nd and HHJ) simply adds to the completion time of the query.

This experiment showed the following results for a slow or bursty environment: (a) XJoin can provide major improvements in the delivery of initial answers to users, (b) the use of reactive background processing to exploit delays (as embodied in the second stage) is crucial for attaining good performance beyond just the initial tuples, and (c) the use of a cache can further improve performance. We now turn to the other three delay scenarios studied in this experiment.

4.1.2 Mixed Network

Figures 12 and 13 show the results when one of the sources is fast and the other is slow. As in the slow/slow case, the XJoin variants all perform as well or better than HHJ, and the XJoin variants that use the second stage perform significantly better. The XJoin variants exhibit the same behavior in both the slow/fast and fast/slow cases since they do not distinguish between the build and probe relation. Compared to the slow/slow case, the XJoin variants that include the second stage perform better, as they are able to exploit the faster arrival of either of the inputs. XJoin-No2nd produces its initial tuples slightly earlier than in the slow/slow case, but it still incurs significant blocking, which causes it to lag far behind the other XJoins for most of the execution. The performance of HHJ for the initial results is dominated the by speed of the build relation, thus it has much better initial performance for the fast/slow case than for the slow/fast case. Still, in either situation, HHJ incurs significant slowdown compared to XJoin-NoCache and the full XJoin.

4.1.3 Fast Network

XJoin is intended to solve the problem of bursty and slow arrivals and the previous three cases have shown that XJoin effectively deals with these arrival problems. Now we examine how XJoin performs when the network is relatively fast. In this case (shown in Figure 14) all three XJoin variants still provide substantial benefits in the delivery of the initial results. Users of HHJ would have to wait over 2.5 minutes before seeing their first results, whereas XJoin would give the first result after only about 2 seconds. As the query progresses, however, the benefits of XJoin diminish in this case. In fact, HHJ is able to deliver the second half of the result faster than XJoin-NoCache and the full XJoin here. Furthermore, XJoin-No2nd, which performed the worst of the 3 variants in the previous cases, loses here initially, but ultimately delivers the last 60% of the result faster than the variants that use the second stage. These results demonstrate the tradeoffs of the background processing done by the second stage that were identified in Section 2.2. Namely, that the overhead incurred by the second stage pays off only when it is used to offset delays. Recall that the fairly coarse-grained nature of the second stage causes it to commit to some amount of work each time it is executed. In this experiment, the second stage was run in a very aggressive manner (i.e, *activation threshold* = 0.01). This aggressiveness is beneficial in the presence of delays, but as shown in this case, can hurt in their absence. Thus, in the next set of experiments we investigate a way of controlling the aggressiveness of the second stage.

4.2 Experiment 2 – Controlling the Second Stage

The previous set of experiments showed that the second stage of XJoin is crucial for a good interactive performance when dealing with slow and bursty data sources, but that with more reliable sources, it could adversely affect the overall query response time. One way to reduce this overhead is to employ the second stage less aggressively, i.e. less often. Recall that the ability of the second stage to operate can be manipulated by varying the *activation threshold* presented in Section 2.4.2. A higher threshold value makes it less likely for the second stage to be executed.

Looking more closely at the results of the previous experiment, it can be seen that the benefits of the second stage are highest during the early parts of the query execution and that they diminish as the execution progresses. This behavior arises because as the disk-resident portions of the partitions grow, the coarseness of the second stage increases (recall that a disk-resident portion is the unit of granularity at which the second stage executes). This increases the risk of executing the second stage. Because of this behavior we have developed a version of XJoin, called XJoin-Dyn, that is more aggressive in the early stages of the query and becomes less aggressive as more of the result is produced. XJoin-Dyn starts with a low threshold value (0.01), and linearly increases it to a much larger value (0.20) as the percentage of result tuples produced grows.

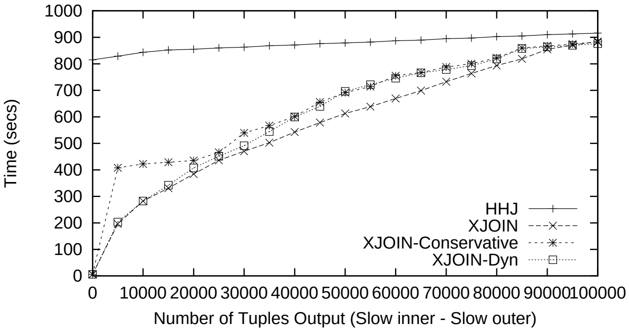

Figure 15: 2 Slow relations

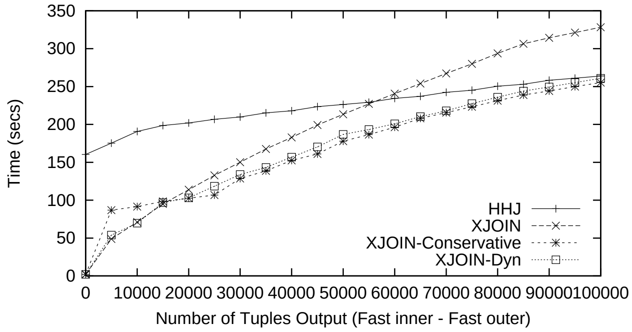

Figure 16: 2 Fast relations

The effectiveness of XJoin-Dyn is examined in Figures 15 and 16, which show the performance for the query used in the previous experiments with the slow/slow and fast/fast network settings respectively. In addition to XJoin-Dyn, the figures show the HHJ and XJoin (i.e., threshold = 0.01) algorithms, as well a conservative version of XJoin with the threshold fixed at 0.20 (labeled XJoin-Conservative in the figures). As seen in the previous experiments, when both sources are slow (Figure 15) the more aggressive XJoin performs best. The less aggressive setting (XJoin-Conservative) provides poor initial performance, but provides reasonable (although still worse than XJoin) performance as the query progresses. XJoin-Dyn initially follows XJoin but then becomes more like XJoin-Conservative as it becomes increasingly conservative, that is, it performs slightly worse than the aggressive setting.

With two fast sources, (Figure 16) as seen in the previous section, the aggressive setting (XJoin) provides good performance initially, but quickly falls behind, eventually losing even to HHJ. In this case XJoin-Dyn is able to combine the best aspects of both conservative and aggressive behavior. Its initial aggressiveness leads to good interactive performance, while its gradual conversion to a more conservative behavior reduces the overhead. Thus, in this case, XJoin-Dyn is able to provide excellent interactive performance initially, while performing better than HHJ throughout the entire query. Because of its good behavior for fast networks and small costs for slow networks, we adopt this dynamic version of XJoin for the remaining experiments, and simply refer to it as “XJoin”.

4.3 Experiment 3 – The effect of memory size

Recall that a prime motivation for designing XJoin was the huge memory requirements of the symmetric hash join. The memory requirements of XJoin are reduced by allowing the staging of data to disk. This staging, however adds overhead, both in terms of disk I/O and in terms of duplicate detection. In order to explore the effect of memory size on XJoin, we have performed experiments varying the size of the memory. For these experiments, we used the single join query similar to the one used in the previous sections, but with the size of the input relations projected down to 8.6 MBytes (rather than the 28.8 MBytes used earlier). We use three different memory allocations: 3MB, 10MB, 20MB. 3MB represents the limited memory cases where neither of the inputs fit into the memory, while the other two represent cases where one input, and both inputs fit into the memory respectively.

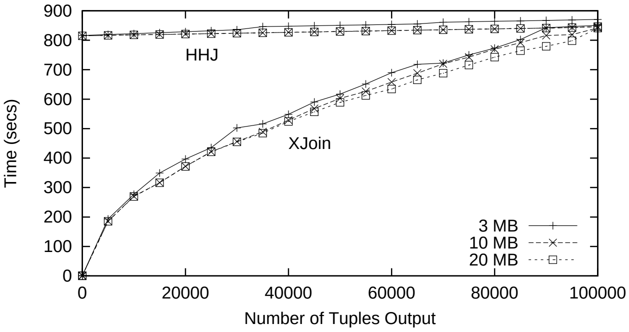

Figure 17: Slow network, Varying memory

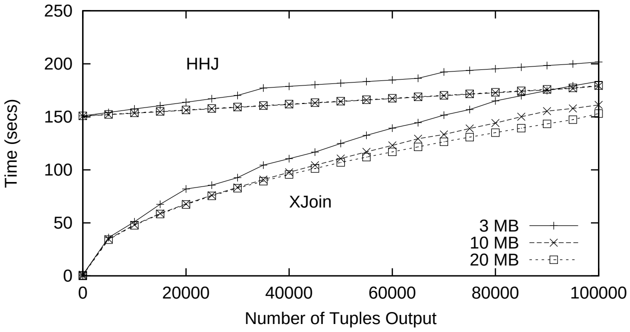

Figure 18: Fast network, Varying memory

Figures 17 and 18 show the result delivery times of XJoin and HHJ for the three memory sizes in the slow/slow and fast/fast cases respectively. In all the cases XJoin performs better than HHJ, with orders of magnitude improvement in the interactive region. XJoin also wins in terms of the completion time in all the cases. Note that when both relations fit in memory (e.g., 20 MB), the basic Symmetric Hash Join (SHJ) is a viable alternative. In this case however, its performance (not shown) is virtually identical to that of XJoin for all delay scenarios.

4.4 Experiment 4 – Stress testing XJoin

Finally, we briefly report on one additional set of experiments that investigates the impact of query complexity on XJoin. We varied the number of relations in the query from 2 to 6 (i.e., 1 to 5 joins). Again, all joins were on the unique1 attribute, so the cardinality of all the results was 100,000 tuples. Up to 6 different relations were used, each one was projected down to 8.6 MBytes. The available memory was scaled with the queries by allocating 3 MBytes to each join operator. We have repeated the experiment for two network settings: Table 19 contains the production times of various tuples for XJoin and HHJ when all the sources are slow. Each row corresponds to a different degree of complexity in terms of the number of relations joined. Table 20 compares the performance of both operators in a similar manner, but for an extreme case; i.e. when none of the inputs is problematic.

In general we see that in all the cases for both network settings, XJoin continues to deliver very good performance in delivering the initial portions of the query results. When the network is slow (table 19) XJoin performs better than HHJ in all the cases including the query completion time. Thus, XJoin is effective in coping with slow and bursty sources even for fairly complex queries.

Table 20 shows the performance of both algorithms in the unlikely case where none of the remote sources are problematic. Here too, XJoin performs better than HHJ in delivering the initial results (e.g., up to the first 20,000 tuples) for all of the cases. For example, even with 6 relations, XJoin produces the first tuple over 6 times faster than HHJ. With increasing query complexity and the lack of delays, however, the delivery of the later tuples by XJoin begins to lag behind that of HHJ. This behavior arises because the second stages of individual operators are controlled on a per-operator basis rather than globally, and thus, the second stage of one operator can interfere with the first and third stage processing of other operators. While this issue could be addressed by a more global scheduling mechanism, even in the extreme case here (with 6 fast relations) the degradation in the delivery of the last tuple is only 35%.

5 Related work

Carey and Kossmann [CK97] worked on returning the first N tuples of the result of queries including an ORDER BY clause. Their method tries to minimize the time it takes to produce first N tuples, although the user may not get any result until all N tuples are computed. Their algorithm does not attempt to compute more than N tuples and does not extend to the bursty and slow environments since joins would still block when a relation shows bursty nature.

Bayardo and Miranker [BM96] worked on returning the first result of a query. In doing so they have modified the well known nested loops join algorithm to detect and minimize the wasted work in a long pipeline of nested loops join operators. The limitation of their work is twofold. First nested loops join is an inefficient join for large inputs when there are no indices on the join attributes of the relations. Thus nested loops does not scale well if the user wishes to get more than one result fast. Second, the output characteristics of the nested loops join is affected by the behavior of the inner relation. If the inner relation is slow or bursty the output of the join will also be slow or bursty.

The online aggregation work presented in [HHW97] allows the output of a query (in their case an aggregate query) to be presented to the user as it improves in real time. This requires joins that are non-blocking, i.e. that supply their output early and in a steady fashion. In [Hel97], Hellerstein discusses different versions of nested loops join algorithms with different traversal patterns. One could change the traversal pattern dynamically if one of the input sources block. However the query execution is still likely to suffer from the inefficient nature of the nested loops join when the inputs are large.

The Approximate query processor [VL93] first computes a semantically approximate answer to a query. This answer is then refined over time. However their work requires that the base relations be partitioned on some of selection attributes for efficient processing. This may not be true on a widely distributed heterogeneous system.

Rels	**First tuple**		**5,000**		**20,000**		**50,000**		**Last Tuple**	
	xjoin	HHJ	xjoin	HHJ	xjoin	HHJ	xjoin	HHJ	xjoin	HHJ
2	5	823	195	826	452	836	668	836	860	878
3	17	862	340	873	561	901	763	930	865	949
4	46	916	482	938	684	979	786	992	907	1018
5	85	353	563	965	709	1019	864	1036	981	1066
6	79	992	400	1075	631	1120	860	1144	952	1174

Figure 19: Tuple Production rates of XJoin and HHJ in secs. Slow Network

Rels	First tuple		5%		20%		50%		Last Tuple	
	xjoin	HHJ	xjoin	HHJ	xjoin	HHJ	xjoin	HHJ	xjoin	HHJ
2	1	150	36	153	87	163	127	181	178	201
3	4	184	90	195	157	223	233	252	296	270
4	17	285	175	307	282	348	378	362	470	387
5	35	353	274	417	406	469	538	487	635	516
6	75	476	381	559	598	605	803	629	892	660

Figure 20: Tuple Production rates of XJoin and HHJ in secs. Fast Network

Also their method is not directly extensible to the bursty environments.

Our previous work on Query Scrambling [UFA98] tries to react to the changes in the network by dynamically restructuring the query plan on the fly. This approach is aimed primarily at improving the overall response time of the entire query, rather than improving the response of the first few tuples. As such, we have developed XJoin as a complementary approach to Query Scrambling.

Finally, as mentioned in the introduction, we have just recently become aware of a related project that has been developed independently from and concurrently with the XJoin work. This is the Tukwilla system being developed at Washington [IFFL+99]. This system includes techniques based on Query Scrambling, but also includes a pipelined join operator based on Symmetric Hash Join. The join operator used in Tukwilla adjusts to limited memory by flushing tuples to secondary storage in various ways, and then completes the join using techniques similar to Hybrid Hash Join. This approach is similar to the first stage and third stages of XJoin. As demonstrated in the preceding sections, a key component of XJoin's ability to provide fast answers in the presence of slow and bursty sources is the background processing that is triggered in response to both inputs becoming temporarily delayed. This "second stage", is a source of additional complexity and tradeoffs in the algorithm (e.g., duplicate elimination, scheduling, degree of aggressiveness, etc.), but is also crucial for good performance in an unpredictable environment. Thus, our experiments lead us to believe that the XJoin operator is a better solution in dealing with bursty and slow sources.

6 Conclusion and Future Work

In this paper, we addressed the problem of getting fast query answers in an unpredictable communications environment, such as the Internet. We presented a multi-staged join algorithm, called XJoin, that extends the symmetric hash join. XJoin has similar performance when memory is plentiful, but can operate efficiently with far less memory, making it much more practical for this environment.

The XJoin lowers its memory requirement by partitioning its inputs and achieves a high level of pipelining by running its stages in an interleaved fashion. We have also presented a fast, on-the-fly duplicate elimination algorithm to eliminate potential duplicate tuples that could be created due to the overlapping nature of the stages. An important feature of XJoin is its ability to react to delays and take advantage of silent periods to produce more tuples faster. The algorithm includes a dynamic mechanism by which its aggressiveness in exploiting delays is adjusted during the execution of a query.

We have implemented XJoin in the PREDATOR Object-Relational DBMS, and compared its performance with that of hybrid hash join using real network traces. We performed at detailed experimental study, which investigated the performance of XJoin in the presence of different data delivery rates, memory sizes, and query complexity. In all the cases studied, XJoin had much better (often by several orders of magnitude) interactive performance (i.e., in terms of producing the initial portions of the result) than hybrid hash join, and in most cases it performed better than hybrid hash join for the entire query, delivering even the final result tuple as fast or faster.

In terms of future work, we plan to tie our current work with our previous work on Query Scrambling to provide a unified set of techniques for dealing with the problem of unpredictable data delivery in wide-area networks. We also plan to work on delivering more "interesting" portions of a result (such as some subset of columns) faster in wide-area environments. Such query behavior is desirable when the semantics of the application are such that some identifiable portions of the data are substantially more important than others.

References

[AFTU96] L. Amsaleg, M. J. .Franklin, A. Tomasic, and T. Urhan. Scrambling Query Plans to Cope With Unexpected Delays. *PDIS Conf.*, Miami, USA, 1996.

[AFT98] L. Amsaleg, M. J. .Franklin, and A. Tomasic. Dynamic Query Operator Scheduling for Wide-Area Remote Access. *Journal of Distributed and Parallel Databases*, Vol. 6, No. 3, July 1998.

[BDT83] D. Bitton, D. J. Dewitt, C. Turbyfill. Benchmarking Database Systems, a Systematic Approach. *VLDB Conf.*, Florence, Italy, 1983.

[BM96] R. Bayardo, and D. Miranker. Processing Queries for the First Few Answers. *Proc. 3rd CIKM Conf.*, Rockville, MD, 1996.

[CK97] M. J. Carey, and D. Kossman. On Saying "Enough Already!" in SQL. *ACM SIGMOD Conf.*, Tucson, AZ, 1997.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2), 1993.

[GRVB98] J. Gruser, L. Raschid, M. E. Vidal, L. Bright Wrapper Generation for Web Accessible Data Sources. Int. Conf. Cooperative Information Systems, New York City, NY, 1998.

[HHW97] J. M. Hellerstein, P. J. Hass, and H. J. Wang. Online Aggregation. *ACM SIGMOD Conf.*, Tucson, AZ, 1997.

[Hel97] J. M. Hellerstein. Online Processing Redux. *Data Engineering Bulletin*, 20(3), 1997.

[HS93] W. Hong, M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. *Distributed and Parallel Databases*, 1(1):9-32, 1993.

[IFFL+99] Z. Ives, D. Florescu, M. Friedman, A. Levy, D. S. Weld. An Adaptive Query Execution System for Data Integration. *To appear in ACM SIGMOD Conf.*, Philadelphia, PA, 1999.

[ONK+97] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, A. Dogac. Dynamic Query optimization in Multidatabases. *Data Engineering Bulletin*, Vol. 20, No. 3, September, 1997.

[Sha86] L. D. Shapiro. Join Processing in Database Systems with Large Main Memories. ACM Transactions on Database Systems, 11(3), 1986.

[SP97] P. Seshadri, M. Paskin PREDATOR: An OR-DBMS with Enhanced Data Types. *ACM SIGMOD Conf.*, Tucson, Arizona, 1997.

[TRV96] A. Tomasic, L. Raschid, and P. Valduriez. Scaling Heterogeneous Databases and the Design of DISCO. *ICDCS Conf.*, Hong Kong, 1996.

[UFA98] T. Urhan, M. J. .Franklin, and L. Amsaleg. Cost Based Query Scrambling for Initial Delays. *ACM SIGMOD Conf.*, Seattle, WA, 1998.

[VL93] S. V. Vrbsky, and J. W. S. Liu. Approximate, A Query Processor that Produces Monotonically Improving Approximate Answers. *IEEE Transactions od Knowledge and Data Engineering*, Vol.5, No.6, December 1993.

[WA90] A. N. Wilschut, and P. M. G. Apers. Pipelining in Query Execution. *Conf. on Databases, Parallel Architectures, and their Applications*, Miami, 1991.