XJoin: A Reactively-Scheduled Pipelined Join Operator*

Tolga Urhan
University of Maryland, College Park
urhan@cs.umd.edu

Michael J. Franklin
University of California, Berkeley
franklin@cs.berkeley.edu

Abstract

Wide-area distribution raises significant performance problems for traditional query processing techniques as data access becomes less predictable due to link congestion, load imbalances, and temporary outages. Pipelined query execution is a promising approach to coping with unpredictability in such environments as it allows scheduling to adjust to the arrival properties of the data. We have developed a non-blocking join operator, called XJoin, which has a small memory footprint, allowing many such operators to be active in parallel. XJoin is optimized to produce initial results quickly and can hide intermittent delays in data arrival by reactively scheduling background processing. 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 Wide-Area Query Processing

The explosive growth of the Internet and the World Wide Web has made tremendous amounts of data available on-line. Emerging standards such as XML, combined with wrapper technologies address semantic challenges by providing relational-style interfaces to remote data. 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 traditional query processing techniques, query execution can become blocked even if only one of the accessed data sources experiences such delays.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*This work was partially supported by the NSF under grant IRI-94-09575, by the Office of Naval Research under contract number N66001-97-C8539 (DARPA order number F475), by a Siemens Faculty Development Award, and by an IBM Partnership Award.

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 using a non-blocking join operator we call XJoin. XJoin 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 observed by the users.
2. *It allows progress to be made even when one or more sources experience delays.* There are two reasons for this. First, XJoin requires less memory, which allows for bushier plans. Thus, some parts of a query plan can continue while others are stalled waiting for input. Second, by employing background processing on *previously received* tuples from both of its inputs, an XJoin operator can produce results even when both inputs are stalled simultaneously.

XJoin is based on the Symmetric Hash Join (SHJ) [WA91, HS93] which was originally designed to allow a high degree of pipelining in parallel database systems. As originally proposed, however, SHJ requires that hash tables for both of its inputs be kept in main memory during most of the query execution. As a result, SHJ 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 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 in spirit to the way that hybrid hash join solves the memory problems of classic hash join.

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 XJoin 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.

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.

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. The Tukwila system [IFFL^{+}99] incorporates an extension of SHJ called Double Pipelined Hash Join (DPHJ) that can work with limited memory. DPHJ differs from XJoin in several details such as the way in which tuples are flushed to secondary storage. More importantly, as originally specified, DPHJ does not include reactively-scheduled background processing for coping with delayed sources. Both DPHJ and XJoin can be thought of as types of Ripple Joins [HH99] which are a class of pipelined join operators that allow the order of data delivery to be adjusted dynamically.

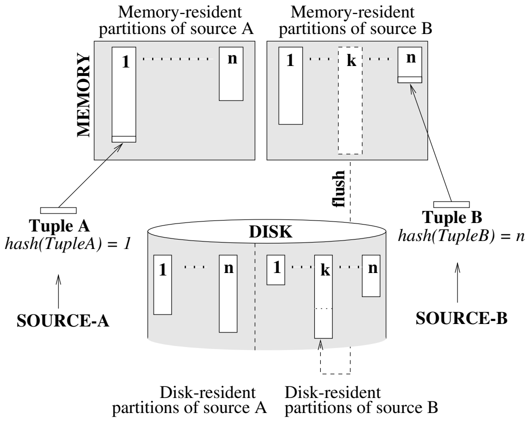

Figure 1: Handling the partitions.

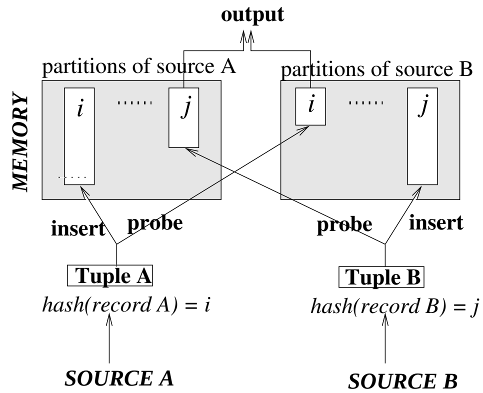

Figure 2: Stage 1 - Memory-to-Memory joins

2 The Design of XJoin

In this section we only give a brief overview of the mechanisms used by XJoin. A more detailed description of XJoin is given in [UF99].

2.1 The Three Stages of XJoin

XJoin proceeds in three stages, each of which is performed by a separate thread. The first stage joins memory-resident tuples, acting similarly to the standard symmetric hash join. The second stage joins tuples that have been flushed to disk due to memory constraints. The third stage is a clean-up stage, which performs any necessary matching to produce 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.

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 (Figure 1). In general each partition can consist of two portions: a memory-resident portion, which stores the most recently arrived tuples for that partition, and a disk-resident portion, which contains tuples of the partition that have been flushed to disk due to memory constraints. When an input tuple arrives from a source, if there is memory available for the tuple then it is simply placed in its partition and used to probe the memory-resident portion of the corresponding partition for the other source (Figure 2). If, however, memory is full, then one of the partitions is chosen as a victim and its memory-resident tuples flushed to disk (i.e., appended to its disk-resident portion). Join processing then continues as usual. The first stage runs as long as at least one of its inputs is producing tuples. If the first stage ever 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 is activated whenever the first stage blocks. It first chooses a partition from one source using optimizer-generated estimates of the output cardinality and the cost of performing the stage using the partition. [1] It then uses the tuples from the disk-resident portion of that partition to probe the memory-resident portion of the corresponding partition of the other source. Any matches found are output (subject to duplicate detection as described in Section 2.2) as result tuples. After a disk-resident portion has been completely processed, the operator checks to see if either of the join inputs have resumed producing tuples. If so, then the second stage halts and the first stage is resumed, otherwise a different disk-resident portion is chosen and the second stage is

[1] Note that the same partition can be used multiple times, as the partition grows over the course of the join execution.

continued. As an additional optimization, tuples brought into memory during one iteration of the Second Stage can be probed with disk-resident tuples from the corresponding partition of the other source in the subsequent iteration.

It is important to note that XJoin follows the Query Scrambling philosophy of hiding delays by performing other work. In particular, the second stage incurs processing and I/O overhead in the hope of generating result tuples. *This work is essentially free as long as the inputs of the XJoin are delayed*, as no progress could be made in that situation anyway. This is where the benefit of the second stage comes in. The risk is that when one or both of the inputs become unblocked it is not noticed until after the current disk-resident partition has been fully processed. In this case, the overhead of the second stage is no longer completely hidden.

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.

2.2 Handling Duplicates in XJoin

The multiple stages of XJoin may produce spurious duplicate tuples because they can perform overlapping work. Duplicates can be created in both the second and third stages. To address this problem XJoin uses a duplicate prevention mechanism based on timestamps.

XJoin augments the structure of each tuple with two persistent timestamps: an Arrival TimeStamp (ATS), which is assigned when the tuple is first received from its source and a Departure TimeStamp (DTS), which is assigned when the tuple is flushed from memory. The ATS and DTS together describe the time interval during which a tuple was in the memory-resident portion of its partition.

These timestamps are used to check whether two tuples have previously been matched by the first stage or second stage. If so these tuples are not matched again. 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 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.

The ATS and DTS are not enough to detect tuples matched in the second stage. In order to solve this problem XJoin maintains a linked list for each partition processed by the second stage. 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. [2] 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 $ProbeTS$.

When two tuples, T_A and T_B, are later matched we first check when T_A was used to probe memory-resident tuples using the linked list maintained for the partition it belongs. If T_B was memory-resident during this time we do not join these two tuples again. The same check is performed for the symmetrical case to determine if T_A was memory resident when T_B was used to probe memory-resident tuples.

2.3 Controlling the second stage

Recall that the overhead incurred by the second stage is hidden only when both inputs to the XJoin experience delays. As a result, there is a tradeoff between the aggressiveness 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

[2] 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.

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 aggressive use of the second stage.

In our implementation of XJoin we dynamically change the value of *activation threshold*, starting with an aggressive value (0.01) and gradually make it more conservative (up to 0.20) as more output tuples are produced. This has the effect of emphasizing the interactive performance at the beginning of the execution and overall performance (i.e., a more traditional criterion) towards the end of the execution.

3 Experimental Results

We have implemented XJoin in an extended version of PREDATOR [SP97], an Object-Relational DBMS, and performed detailed experiments to investigate performance issues associated with various aspects of XJoin. Due to space limitations, however, we only present a portion of the results here. Detailed results can be found in [UF99].

3.1 Experimental Environment

In the experiments we modeled the behavior of the network using trace data that was obtained by fetching large files from 15 randomly chosen sites. From these arrival patterns, we chose two as representatives of the behavior of a bursty and a fast source (figures 3, and 4). The arrival patterns in these figures show the quantity of data received at the query site. We refer to the bursty pattern also as “slow” arrival pattern due to its low transfer rate.

The database used in the experiments contained up to six 100,000 tuple Wisconsin benchmark relations [BDT83]. Each input tuple is 86 bytes after projections have been applied. Join attributes used are one-to-one, producing 100,000 result tuples. We ran the experiments on a Sun Ultra 5 Workstation with 128 MBytes of memory. In the experiments the XJoin operator is given 3 MBytes of memory.

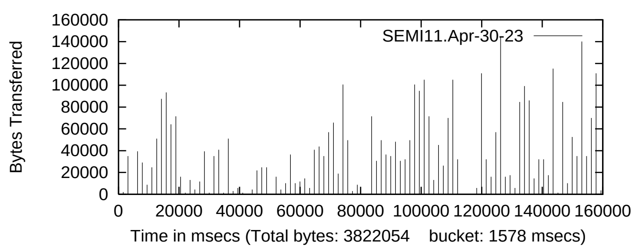

Figure 3: Bursty arrival. Avg. Rate 23.5 KBytes/sec.

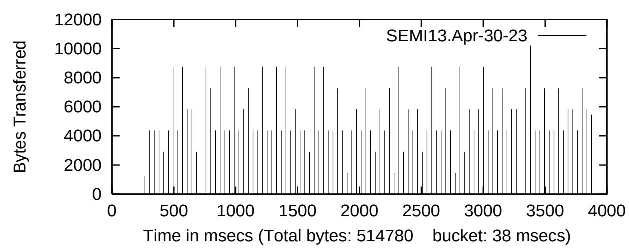

Figure 4: Fast arrival. Avg. Rate 129.6 KBytes/sec.

3.2 Results

We compared the performance of XJoin to that of Hybrid Hash Join (HHJ). In order to separate out the contributions of the major components of the algorithm we also examined two other XJoin variants. The first variant, labeled *XJoin-No2nd*, does not use the second stage at all. The second variant, labeled *XJoin-Aggr* is an aggressive version of XJoin which uses an aggressively set *activation threshold* (i.e., 0.01) . We also tried to improve the responsiveness of HHJ by allowing base tuples to be fetched in parallel in the background. This parallelism allows HHJ to overlap delays from one input with the processing of the other.

Figures 5 and 6 show the cumulative response times for the four algorithms for the bursty and fast arrival cases respectively. The x-axis shows a count of the result tuples produced and the y-axis shows the time at which that result tuple was produced. In both cases XJoin and its variants produce the first answers several orders of

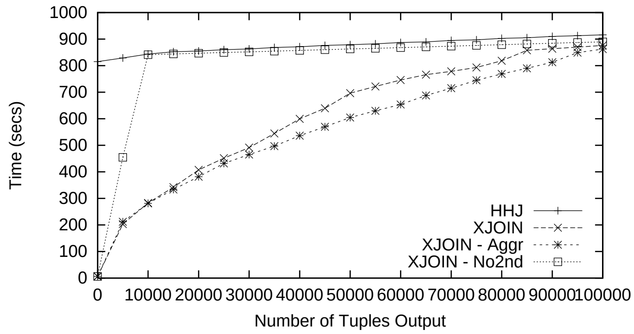

Figure 5: Slow arrival

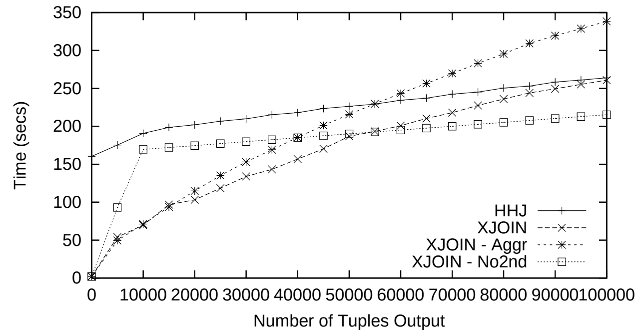

Figure 6: Fast arrival

magnitude faster than HHJ, thereby providing far superior interactive performance. XJoin also outperforms HHJ in terms of the time to return the entire result.

A comparison between the XJoin and XJoin-No2nd highlights the importance of the second stage in improving the responsiveness of the system. XJoin-No2nd, although performing competitively for the very initial results, fails to maintain this performance for majority of the results. A comparison between XJoin and XJoin-Aggr is perhaps more interesting as it demonstrates the tradeoff of executing the second stage. In the slow network case XJoin-Aggr performs slightly better than XJoin in the middle range. This is because there is enough delay to hide the extra work introduced by XJoin-Aggr. However this improvement is at the expense of poor response in the fast network case (Figure 6). In the absence of enough delay to overlap the overhead of second stage XJoin-Aggr falls behind XJoin.

Other results, not included in this paper, have also showed the superiority of XJoin in delivering the initial portion of the result under variety of conditions. Experiments measuring the effect of memory size have shown that XJoin has robust performance even with very limited memory. Further experiments stress tested XJoin by running queries involving up to 5 join operators (up to 6 inputs). In all the cases XJoin was able to outperform HHJ in delivering the initial portion of the result with only minor degradation in delivering the last tuple.

4 Conclusion and Future Work

In this paper, we described the design of XJoin, a reactively scheduled pipelined join operator capable of providing responsive query processing when accessing data from widely-distributed sources. XJoin incorporates the Query Scrambling philosophy of hiding unexpected problems in data arrival by performing other (non-scheduled) useful work. The smaller footprint is obtained through the use of partitioning. The delay-hiding feature is implemented through the use of a reactively-scheduled "second stage", which aims to produce result tuples during periods of delayed or slow input by joining tuples of one input that have been spooled to secondary storage with the memory-resident tuples of the other input.

In terms of future work, we plan to investigate the scheduling issues in complex query plans with multiple XJoin operators. Currently XJoin operators are scheduled in a round-robin fashion. Rate at which initial portion of the result delivered can be improved by scheduling more productive operators (i.e., low cost operators that contribute more to the result) frequently. 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.

In the larger context, XJoin represents one piece of technology that can help extend database systems to the wide-area environment. In fact, there are a spectrum of techniques for making query processing more adaptive, ranging from delayed-binding, to adaptive re-optimization and beyond. One interesting recent development is the "Continuous Query Optimization" (CQO) developed by Avnur and Hellerstien [AH00], which foregoes traditional optimization for an adaptive queue-based scheduler that in effect learns an efficient query plan during the query execution. We plan to investigate the integration of XJoin with such mechanisms as part of the Telegraph project at Berkeley.

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.

[AH00] R. Avnur, J. Hellerstien. Continous Query Optimization. *ACM SIGMOD Conf.*, Dallas, TX, 2000.

[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.

[HH99] P. J. Haas, J. M. Hellerstein. Ripple Joins for Online Aggregation. *ACM SIGMOD Conf.*, Philadelipha, PA, 1999.

[HHW97] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. *ACM SIGMOD Conf.*, Tucson, AZ, 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. *ACM SIGMOD Conf.*, Philadelphia, PA, 1999.

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

[UF99] T. Urhan, M. J. .Franklin. XJoin: Getting Fast Answers from Slow and Bursty Networks. *University of Maryland Technical Report, CS-TR-3994.*, February, 1999.

[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.

[WA91] A. N. Wilschut, and P. M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. *1st Int'l Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, 1991.