XJoin and the Benefits of Free Work

Justin Forrester Jonathan Ledlie
{jforrest, ledlie}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706, U.S.A.

Abstract

We report here on our experience implementing a new join algorithm called XJoin. XJoin improves the performance of the standard Hybrid Hash Join algorithm by allowing the joining of tuples that had been flushed to disk during otherwise idle periods. It is fairly straightforward to implement from the descriptions in the literature, with the exception of a few important neglected details. Our performance study shows the XJoin algorithm performs quite well when given tuple inputs that are bursty and/or slow.

1 Introduction

The XJoin algorithm seeks to efficiently join two remote sources in limited memory situations by performing "free" background work when both sources become blocked [1]. When allotted a size of memory that is independent of the relation sizes, it performs matches like a Symmetric Hash Join until memory is full [2]. When this occurs, the largest memory partition is flushed to disk, enabling more input tuples to stream in and be matched. While tuples are available from either input source, this process continues. When both inputs become blocked, however, the algorithm aims to join the tuples that were on disk when their memory partition was probed. It is these matches which are considered "free" because they occur when no input is coming in from the network. Instead of saving this clean up work until the end, XJoin is able to continuously stream output until it has exhausted all of this free work, by which time the network inputs will have presumably resumed. Our results section shows clearly that XJoin is successful in its goals, and that a significant number of output tuples are generated even when both input streams are blocked.

XJoin compares favorably with other hash join algorithms in several respects. Symmetric hash join requires enough memory to contain both relations; by definition it cannot send tuples to disk because they are immediately inserted into one hash table and then used to probe the other. Hybrid Hash Join (HHJ) solves this problem by sending partitions to disk, but it only sorts through them when both inputs have completed sending all their tuples. Also, it does not use all of its memory all of the time and can end the input stage with little of its memory in use. Like HHJ, XJoin sends partitions to disk when its memory capacity has been reached, but XJoin is able to join tuples from disk during periods of delay instead of waiting until all tuples have been received. Neither Symmetric nor Hybrid perform work when the sources become blocked. The main drawback to XJoin is that if these free joins result in few matches on a machine that is already loaded, CPU time is wasted doing many repetitive comparisons.

2 Algorithm Details

XJoin works in three stages. The first and second stages run while there are still tuples coming from either source, and the third stage is a cleanup executed after all the tuples have been received. The first stage hashes tuples into partitions and then probes the complementary memory partition for a match. If the memory allocated to the join has been exhausted, tuples are flushed to disk to make room for more incoming tuples. If both sources become blocked, the first stage yields to the second. This stage chooses a disk partition, reads the tuples it contains into memory and probes the corresponding memory partition of the other relation. The tuples in this disk partition cannot be discarded at this point because they may still join with inputs that have not yet arrived. The second and third stages avoid producing spurious duplicates though keeping timestamps of when the second stage was run for a particular disk partition. Thus, if a tuple in a disk partition is repeatedly run against the same tuples in a memory partition, timestamps show that the two have already matched and the match is dropped. The third stage is a clean up stage. For each set of partitions, it loads all of one into memory and then streams the corresponding disk and memory partitions by it. Once all the partitions have been processed, the join is complete.

3 Project Overview

After reading about the general operation and advantages of the XJoin algorithm, we decided to implement it for our 764 course project. To build our implementation of XJoin, we went through several steps, not the least of which was discovering the nuances of the algorithm buried in the lines of the technical report we used as our basis. We chose to implement the algorithm as a stand-alone application, as opposed to within the confines of an existing DBMS. We made this decision for a simple reason: our primary goal in pursuing this project was to learn and evaluate the XJoin algorithm, not to learn how to add join algorithms to existing databases. The overhead required to get a real DBMS up and running and figure out how to modify it did not seem worth the added realism. We simply wanted to learn and to evaluate the algorithm, and we were able to accomplish this without incurring the logistical overhead of a real DBMS.

We implemented the XJoin algorithm using C++. Our main objects and their simplified uses are as follows:

- XJoin – controls the joining of tuples, flushing of partitions to disk and subsequent reading back, through first, second, and third stage functions.
- MemPartition – contains hash table tuples that are in memory.
- DiskPartition – represents hash table tuples that have been flushed to disk because of memory constraints.
- WITuple – our basic tuple structure, with contents being a simplified version of a Wisconsin Benchmark tuple (a unique_1 integer and a pad of 284 bytes). [3]

To simulate the conditions of a real DBMS, the XJoin algorithm runs in one process, while two other processes produce the tuples to be joined and deliver them to the XJoin process. The tuple producers communicate the tuples to the XJoin process via standard TCP/IP sockets. We ran both the XJoin process and the tuple producing processes on the same machine and used local loopback connections. We structured the tuple producers in such a way that we could vary both the length of delays and the number of tuples sent between each delay. We structured the XJoin object so that it would execute using an amount of memory that we specified. As such, we were able to exercise a great deal of control over how the algorithm executes, both in terms of memory and tuple input rate.

4 Lessons Learned

Urhan and Franklin do a fairly good job explaining the basic algorithm in their XJoin paper. Using the paper as a guide, we were able to finish the bulk of the algorithm easily. However, the paper glosses over two key points that we were forced to discover the hard way. The first of these sticking points was in the final join of tuples in the third stage. During this stage, all the partitions undergo final processing. For each partition pair, the smaller partition gets read fully into memory, and the corresponding disk partition of the other relation is read from disk and used to probe the memory partition. We implemented this but overlooked a fairly big issue: what if there were tuples in the probee partition that were still in the memory? Following the paper's lead, we simply assumed that all we had to do was to read in the flushed partition and probe the corresponding memory partition. However, we overlooked the tuples of the prober partition that were still in memory and had not been flushed to disk. There was a possibility that they may have never been joined with some tuples from the probee disk partition. Hence, our algorithm occasionally did not join all the tuples. This problem was very subtle and difficult to debug because the code ran fine, but occasionally did not produce the correct answer. The paper was of little help, but we were finally able to think through the problem and discover the solution. Once we identified the problem, the solution was the addition of a fairly simple check to see if there were tuples from the prober partition left in memory. If it was not empty, we first probed the smaller partition with the tuples from the prober partition that were still in memory, then with the prober's tuples from disk.

The second difficulty for us was again in the third stage. Unlike the previous bug that caused us to join too few tuples during the third stage, this bug caused us to create duplicate output tuples. The problem had to do with the processing of timestamps during the third stage. Each tuple is assigned two timestamps in the XJoin algorithm: one when it arrives and is used to probe the other relation in the first stage, and one when it is flushed to disk due to memory constraints. Each time the second stage is activated, we record both the time and what tuples are being used to probe the corresponding memory partition. Using these timestamps, we can detect when duplicate output tuples are being generated and prevent them from being sent up the query tree. During the third stage, the tuples are brought in from disk partitions and used to probe the memory partitions of the other relation. When two tuples satisfy the join predicate, we check to see if they were matched in a previous run of the second stage by examining the activation records of the second stage from the disk partition. If we see that this tuple has been brought in previously and used to probe the memory partition before, this match is a duplicate. The paper describes this situation fairly well, and we were able to code a solution readily. However, what the paper does not mention is the fact that you must also check the second stage activation records of the complementary disk partition. We overlooked this important aspect and consequently had duplicate matches occasionally. Both bugs occurred only periodically due to unfortunate timings in flushing to disk and running the second stage and were difficult to diagnose.

5 Experimental Setup and Results

All experiments we performed were executed on Sun Sparc-20 workstations running Solaris 2.7. Each machine contained a single Sparc processor running at 300 MHz.

The first experiment that we performed was intended to show that the use of the second stage does in fact lead to an increase in the rate of production of output tuples. We experimented with different memory sizes to show the varying efficacy of the second stage. For this experiment, we held the parameters of the source fixed at 10,000 tuples/burst with 10-second delays between bursts. Preliminary results indicated that these values would show clearly the effect of the second stage in the XJoin algorithm.

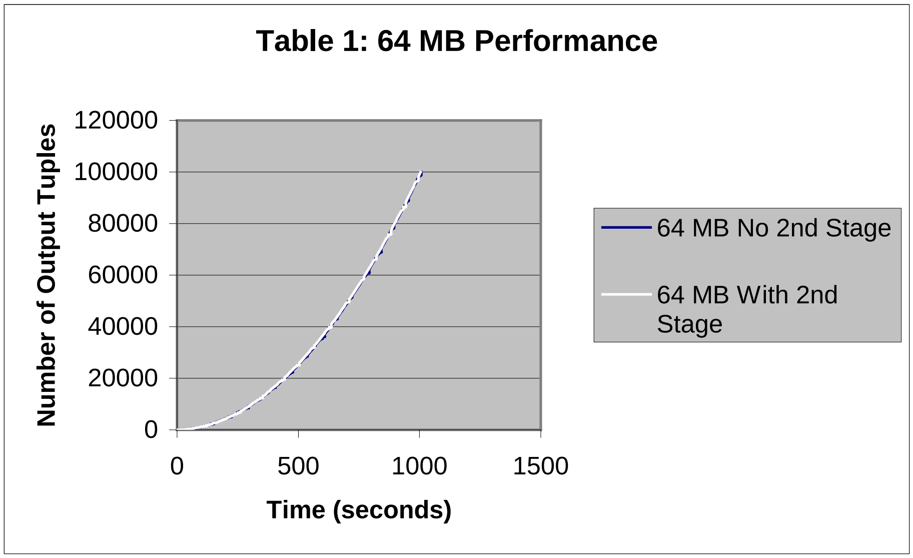
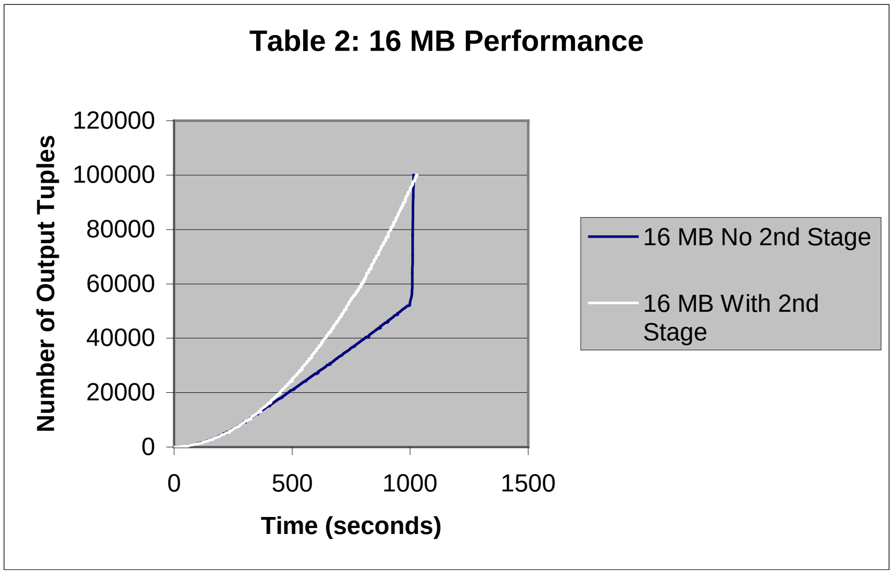


Table 1 shows the relative performance of the XJoin algorithm with 64 MB of RAM available. We see that there is no discernable difference between the algorithm when using the second stage versus not. This is due to the size of our input relations. Each input relation was a total of 28.8 MB in size. Hence, both relations could be fit completely in memory, thereby negating any effect of the second stage because no tuples were ever flushed to disk.



In Table 2, we begin to see the impact of the second stage as the memory available to the algorithm was squeezed down to 16 MB. In this case, the relations would not fit in memory, and partitions were forced to disk. We see the algorithm without the second stage produce at a fairly constant rate, and then jump dramatically when it transitions to the third stage. In contrast, the use of the second stage allowed the

XJoin algorithm to continue to produce a high number of tuples even in the face of delayed inputs. This would allow operators higher in the query evaluation tree to take advantage of the earlier arrival of tuples.

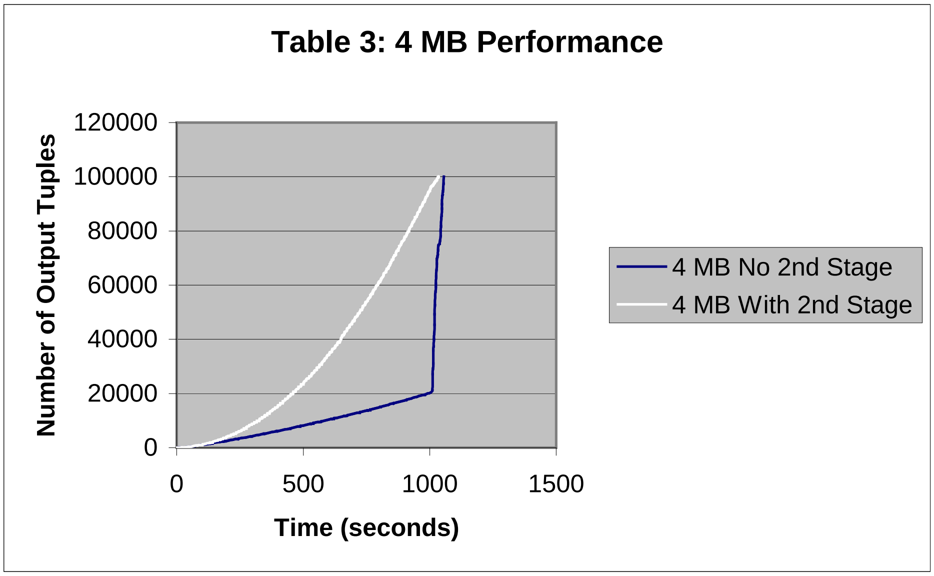


Table 3 shows the trend continues when the memory is further constrained to only 4 MB. Here we see quite clearly that the use of the second stage allows a near steady stream of tuples to be produced, while not using the second stage causes nearly all the tuples to be joined at the very end, in the third stage. It is here where we see the clearest case for the use of XJoin. In this 4 MB case, the difference that the second stage makes is quite dramatic. Certainly, the XJoin algorithm with its second stage would be a great asset under such memory constrained and slow input conditions.

Our second experiment shows how the burst size and delay lengths impact the performance of the XJoin algorithm. We held the memory size constant at 4 MB, and allowed the second stage to run. Table 4 displays the results of the experiment. We see that burst length seems to have the largest impact on performance. When relatively large bursts are used, 10000 tuples in this case, the outputs are generated very rapidly. The most interesting observation in Table 4 is what happens when the burst size is small and the delay is long. We see that the performance in terms of tuples joined is not degraded by too much due to the slow and bursty input characteristics. In fact, we see in general that the performance degrades much more gracefully when using the second stage due to the fact that it can do work and join tuples while it is waiting for inputs. In addition, we observe that when given lots of tuples with short delays, the XJoin algorithm performs admirably. This is important because we do not want the added complexity of the algorithm to impact the output rate when the tuple sources are producing tuples at a fast rate. Presumably this would be the common case, and our results show that XJoin is not sacrificing any performance in exchange for being able to do work during periods of delay.

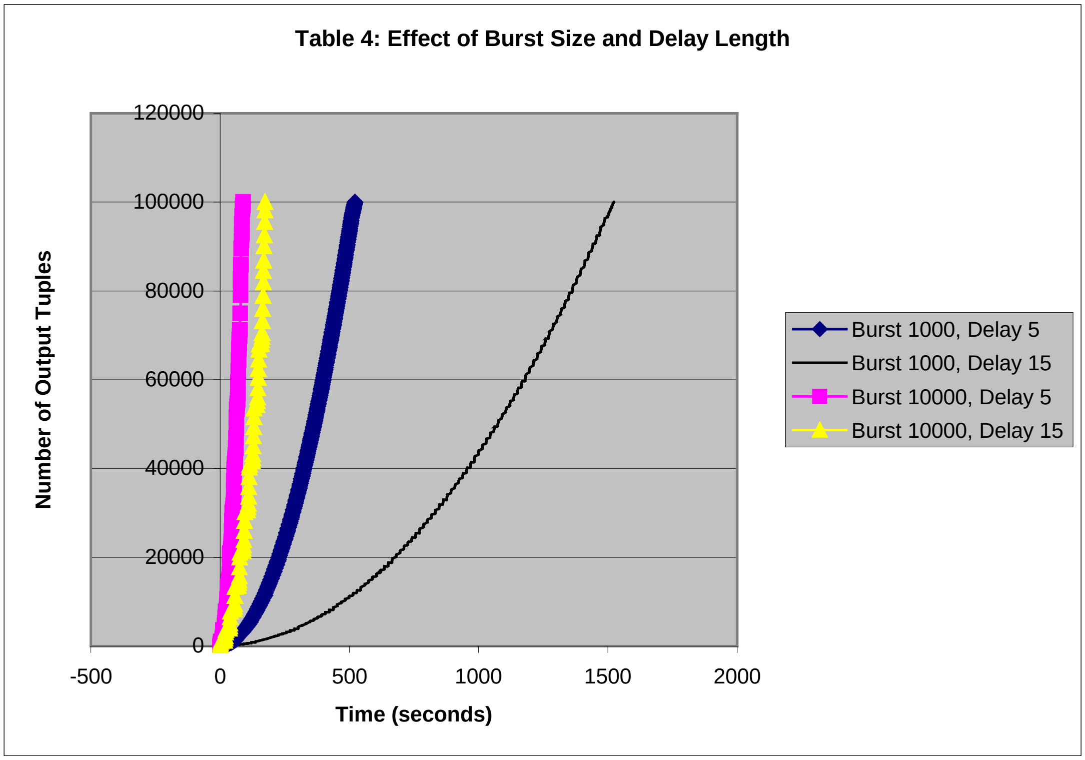


Table 4: Effect of Burst Size and Delay Length

6 Conclusions

Our goals in undertaking this project were twofold: understanding and implementing the XJoin algorithm, and evaluating its effectiveness given varying input and memory constraints. We certainly accomplished both goals. Our implementation of the XJoin algorithm allowed us to gain intimate insight into the algorithm and its possible uses in real world DBMS's. We were able to gain further experience in the implementation of database join algorithms, and discover aspects of the XJoin algorithm not mentioned in the paper detailing it. Our experimental results show that the XJoin algorithm is quite effective in situations of low memory with bursty and/or delayed tuple sources. Moreover, this effectiveness is essentially free as long as the inputs are delayed, so the algorithm does not sacrifice system resources for the increased performance. All in all, it was a successful project in which we both learned a great deal.

References

[1] T. Urhan, M. Franklin. "XJoin: Getting Fast Answers From Slow And Bursty Networks". University of Maryland Computer Science Department Technical Report. CS-TR-3994, UMIACS-TR-99-13. February 1999.

[2] 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.

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