Nick Kisel & Kevin Moy
11 May 2020

Pipelining via PMJ and XJoin

One of the most costly operations in a query is a join, in both runtime and IO cost. Ideally, one would want to implement *pipelining* with these join operators to reduce these costs. Pipelined operators in a query plan output records from one operator to another without having to write them to disk. Unfortunately, standard join algorithms such as Sort-Merge Join and Grace Hash Join are both pipeline breakers, in that we must wait to process the entirety of the input before we can create an output. In this paper, we analyze two enhanced versions of these algorithms, Progressive Merge Join (1) and XJoin (2) respectively, which allow us to retain the pipelining property during joining operations.

The sort-merge join algorithm is a notorious pipeline breaker in that it requires sorting two relations before a single record can be returned, inconvenient for any application that further processes the data or decides whether to abort the transaction after seeing the first few records. The bottleneck to the output of the first record is the size of the largest relation to be joined: its full size must be read and written for every sorting pass.

Progressive Merge Join does away with the long delay before outputting the first record by sorting both relations - referred to as *R* & *S* onwards - at the same time, and merging them during this sort. This "sort" phase is accomplished by loading as much of both relations (the sizes of the runs do not have to be equal or even similar) into memory as possible, concurrently sorting the runs of each relation, and joining all matching records of the respective runs (1). In essence, a sort-merge join has been completed on the run-sized portions of both relations, and a few corresponding records can be delivered to the next join operator. Note that while the sequence of records output from merging each group of sorted runs is sorted, by no means are all records produced by PMJ in sorted order. Once the records of any two sorted runs have been matched, the runs are stored on disk for further matching later on; when this process exhausts all records of both relations, PMJ moves into the “merge” phase.

The merge phase is similar to that of the original SMJ (external merge sort), but using the same strategy of merging sorted runs of R & S simultaneously. Following the philosophy of producing matches as quickly as possible, the matching records of the two merged runs are joined together. Notice at this point that some records have already been output by certain combinations of records of R & S, and could be output again; for this reason PMJ maintains a record of the runs of R & S it has already joined, checking against this record before writing new tuples to the next operator (1). Thereby the two relations are each merged back into their sorted orders, allowing all matching records between them to be joined together.

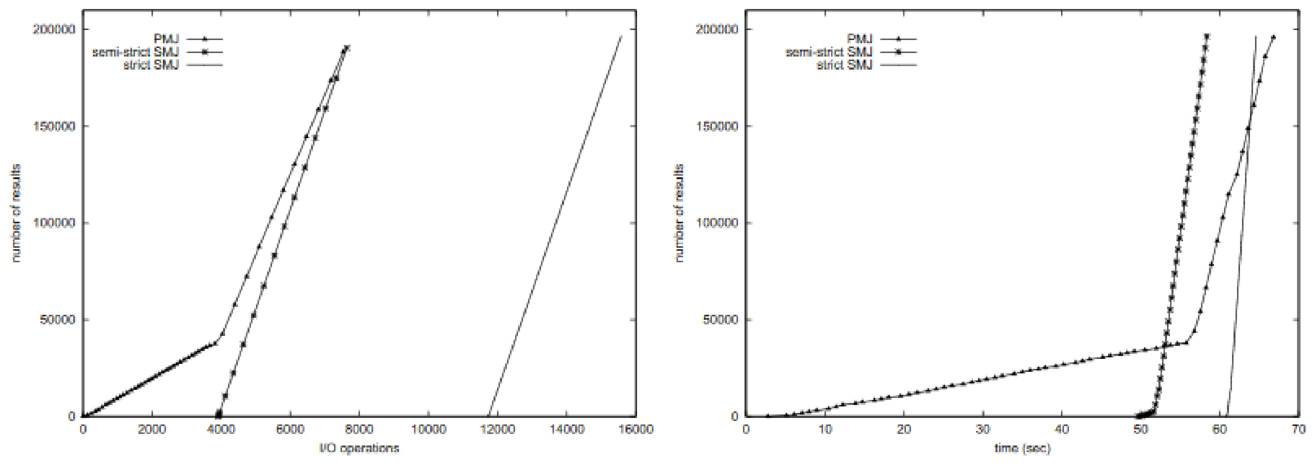
The result of the PMJ algorithm is that small runs of individually sorted output can be output over the course of the entire join process, with the tradeoff that the overall output does not come in sorted order as in SMJ.

Formalizing the memory requirements and constraints of each section is simple, since most components require the cost of doing two sort-merges simultaneously:

For the merging portion, we need at least 6 pages of memory: we need to merge at least 2 sorted runs of R together at a time, so we need two input buffers, and we need a place to write our merging output, so 1 output buffer is required for R, a resultant 3 pages for that relation; an equivalent 3 pages is needed to do the same for S at the same time. Therefore, given B buffer pages, this can be scaled up to 2 pages dedicated to output buffers and B-2 pages dedicated to input buffers, where R & S share the B-2 buffer pages according to their sizes. Once R & S have been merged, writing the output records needs two input buffers - one for each table - and one output buffer page for the resulting joined records. Therefore, the minimum memory cost of sorting is doubled, but the cost of merging remains the same.

However, because the B memory pages need to be shared between the two tables, the number of sorting passes taken by PMJ is strictly greater than that of SMJ. Both relations will have less memory pages at their disposal for sorting, with at least one table having half or less of the memory pages. Asymptotically, the runtime does not change; in practice, however, every single table in a query could require an increased number of sorting passes, meaning that PMJ takes significantly longer to finish, stalling the result of the transaction and outweighing the benefit of pipelining.

According to comparisons of PMJ and SMJ on tables of 2 million uniformly distributed integer records, each of size 8 MB (further examination at http://www.jcmit.net/memoryprice.htm shows that 64 & 128 MB memory was just becoming affordable at the time of test in 2002), the overall I/O cost of PMJ and SMJ is relatively similar, while the overall time taken for PMJ is about 20% slower (1). In regards to its pipelining efficacy, PMJ steadily delivers about 20% of the resulting tuples before the SMJ has output a single tuple.



(a) I/O (b) time

Figure 6: Equi-join UNI1 ⋈ UNI2: Number of result tuples as a function of I/O and time

Yet another join that remedies pipeline breaking is symmetric hash join. Symmetric hash join is a simple, fully in-memory algorithm for joining two tables on equality predicates which can output as soon as two matching records are streamed into the hash tables. For two tables R & S, it uses the same hash function to build hash tables for each relation as records from the relations come in. Whenever a record from R is streamed in, it is added to R's hash table, and the record is checked against all records in the corresponding partition of S's hash table for matches. The same is symmetrically done for S, so that all resulting records are output when the second matching tuple is streamed in. However, symmetric hash join is constrained by the ability to hold all records of both tables in memory.

On the other hand, Grace Hash Join fixes this inability to work with tables that do not fit in memory by partitioning tables until each partition of the smaller table fits in memory. Then, each partition goes through the naive hash join algorithm of building and probing corresponding partitions of two tables. Unfortunately, records cannot be emitted while partitioning tables in GHJ, and it is therefore a pipeline breaker.

Xjoin combines the strengths of SHJ and GHJ without pipeline-breaking when working with two tables that cannot fit in memory by pushing parts of the hash tables to disk; specifically, for each partition of the hash table, the *tail* consisting of the most recently arrived tuples to that partition remains in memory (that way, it acts like an output buffer that can be flushed to disk and replaced by a fresh, empty page for new tuples which arrive later), while the rest of the pages of that partition lie on disk.

Xjoin has three "stages" run on three separate threads: a memory-to-memory stage, a memory-to-disk stage, and a disk-to-disk stage (2). In the first stage, a tuple streamed in is compared to the *tail* of the opposite relation's hash table, with partition pages being flushed to disk as they fill. Evidently, any tuple streamed in will not be matched with those already flushed to disk.

Whenever the first stage is blocked, meaning either no new tuples are coming in from the network or all tuples have been consumed, the second stage begins (2). This disk-to-memory stage compares previously partitioned tuples materialized on disk with the in-memory tuples of the opposite table. Notice that some combinations of tuples addressed in this stage may already have been compared during earlier iterations of the first stage; there is an on-the-fly duplicate detection algorithm to remedy this (discussed after description of the third stage).

The third stage consists of building and probing, as in Grace Hash Join: it combines the smaller table's memory and disk portions of a partition, creating an in-memory hash table of that partition. Then, the records of the other relation's partition are used to probe this hash table for output, meaning every possible match is output by this stage.

Evidently, the process could produce a valid output without tuples by just waiting until this third stage to match all the tuples together, but this prevents the constant throughput generated by the first two stages and desired by pipelining. To prevent duplicate output in the second and third stages of Xjoin, all received tuples are marked with an arrival timestamp (ATS) marking when they were inserted into a hash table and a departure timestamp (DTS) marking when they were moved to disk (2). During the second stage, any compared tuples with overlapping (ATS, DTS) ranges are not output, as they must have already been compared in the first stage. This same check is also completed in the third stage; in addition, any time the second stage compares two tuples, it logs the last tuple of the corresponding partition that accessed that tuple and the timestamp of their comparison such that the third stage skips comparison of two tuples that are logged as having been compared.

Naturally, the question arises of which join algorithm is “better” to implement in general. Obviously, the logic of Progressive Merge Join and Grace Hash Join are completely different; PMJ utilizes the sorting and merging logic of Sort-Merge Join, while XJoin utilizes the hash-table building of Grace Hash Join. Much of their optimality is incredibly system-dependent, but as neither table produces fully sorted output, most comparison falls in regards to their general performance on joining standard input relations.

In terms of general cost, according to a study that ran IO and time cost tests of PMJ and XJoin on various networks, Progressive Merge Join appears slightly superior in regards to *both* runtime and IO cost. On larger relations, Progressive Merge Join incurs less IOs than XJoin (3).

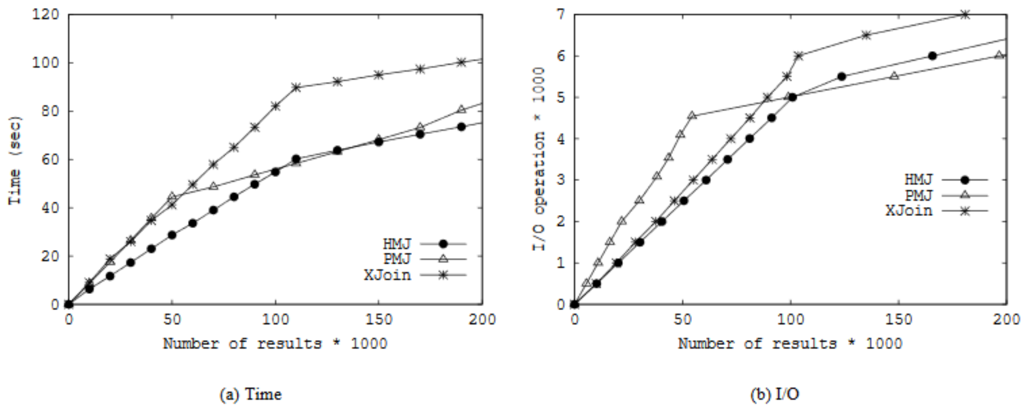


Figure 11. Fast and Reliable Networks.

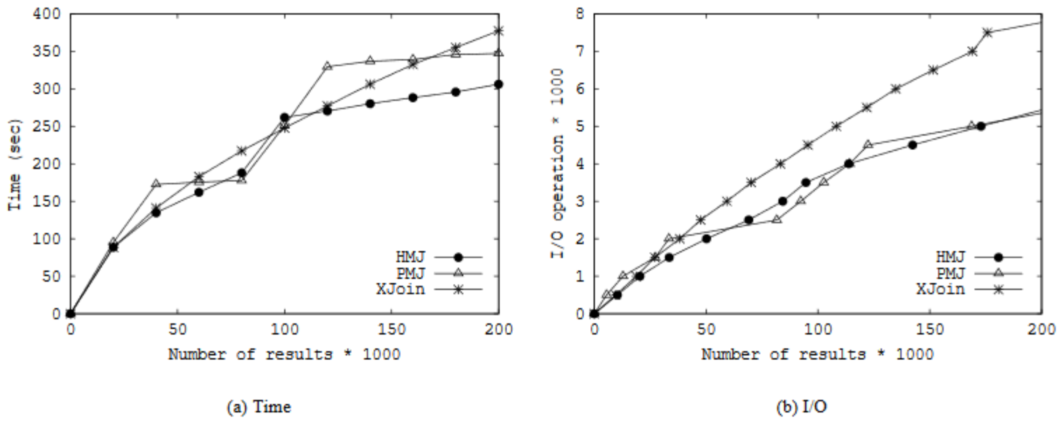


Figure 14. Slow and Bursty Networks.

This may be because as input table sizes get very large, the partitions that XJoin creates have to be read many more times, to the point of creating cross joins between pages of partitions. As noted by System R's third heuristic, cross joins are significant IO incurrence sinks, contributing to XJoin's lower performance.

The benefit of Xjoin, however, lies in its consistency regardless of network conditions. Because of its aforementioned three-stage design, Xjoin can work on generating tuples despite that no new records are being streamed in. In the data from (3), this manifests in its relatively constant release of records over time (slope), contrasted with the relative blocking of PMJ during unavailability of new records. In fact, despite that XJoin's IO cost was significantly higher for the slow network join, it completed in roughly the same amount of time as PMJ while delivering tuples more consistently.

Joins are generally some of the most costly operations of a query, and the inability to operate in a pipeline will cause significant setbacks in performance. Enhanced join algorithms such as XJoin and Progressive Merge Join solve these "blocking" problems and add their own benefits as well. While these two algorithms have adapted existing sorting and hashing techniques for pipelining, room for improvement via more recent or future research is always in demand. For example, analysis of more nuanced join algorithms such as Hash-Merge Join could prove to combine the strengths of Progressive Merge Join and XJoin. Additionally, we could analyze the behavior of these two joins in special cases, such as performance on joins containing many input tables as well as different join plans other than left-deep.

References

(1) https://dl.acm.org/doi/pdf/10.5555/1287369.1287396?download=true

Paper: Dittrich, Jens-Peter, et al. "Progressive merge join: A generic and non-blocking sort-based join algorithm." *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Morgan Kaufmann, 2002.

(2) https://drum.lib.umd.edu/bitstream/handle/1903/997/CS-TR-3994.pdf?sequence=2

Urhan, Tolga, and Michael J. Franklin. *XJoin: Getting fast answers from slow and bursty networks*. 1999.

(3) https://www-users.cs.umn.edu/~mokbel/papers/hashmj-icde04.pdf

M. F. Mokbel, M. Lu and W. G. Aref, "Hash-merge join: a non-blocking join algorithm for producing fast and early join results," *Proceedings. 20th International Conference on Data Engineering*, Boston, MA, USA, 2004, pp. 251-262.