Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources

Stratis D. Viglas Jeffrey F. Naughton Josef Burger

University of Wisconsin—Madison
Department of Computer Sciences
1210 W Dayton st.
Madison, WI 53706, USA
{stratis, naughton, bolo}@cs.wisc.edu

Abstract

Recently there has been a growing interest in join query evaluation for scenarios in which inputs arrive at highly variable and unpredictable rates. In such scenarios, the focus shifts from completing the computation as soon as possible to producing a prefix of the output as soon as possible. To handle this shift in focus, most solutions to date rely upon some combination of streaming binary operators and "on-the-fly" execution plan reorganization. In contrast, we consider the alternative of extending existing symmetric binary join operators to handle more than two inputs. Toward this end, we have completed a prototype implementation of a multi-way join operator, which we term the "MJoin" operator, and explored its performance. Our results show that in many instances the MJoin produces outputs sooner than any tree of binary operators. Additionally, since MJoins are completely symmetric with respect to their inputs, they can reduce the need for expensive runtime plan reorganization. This suggests that supporting multi-way joins in a single, symmetric, streaming operator may be a useful addition to systems that support queries over input streams from remote sites.

1 Introduction

Traditionally, multi-way join queries have been evaluated by trees of binary, partially blocking, pipelined join operators. While this has proven effective in the common case of local join inputs, when moving to distributed domains where queries are executed over remote streaming sources, it is worth considering whether or not this traditional approach is sufficient. Prior work has answered that the traditional approach is not sufficient, and has addressed the problem of streaming inputs by retaining binary execution trees but by replacing blocking operators with streaming symmetric operators [11, 14], possibly coupled with dynamic reorganization of the execution tree [7, 12] in response to fluctuating input rates. In this paper we explore the complementary approach of allowing non-binary trees; that is, by generalizing existing streaming binary join algorithms to produce a multi-way streaming join operator, which we call *MJoin*, that works over more than two inputs.

While the MJoin is a simple generalization of symmetric binary join algorithms, to our knowledge such an operator has not been considered in the literature. This is unfortunate because the MJoin has a number of attractive properties in streaming environments. Using a single multi-way join, an arrival from any input source can be used to generate and propagate results in a single step, without having to pass these results through a multi-stage binary execution pipeline. Furthermore, since the operator is completely symmetric with respect to its inputs, there is no need to restructure a query plan in response to changing input arrival rates. However, it was not clear from the outset how these abstract properties would translate into actual performance; it was also not clear exactly how the MJoin operator should handle memory overflow. In this paper we address these issues.

Our main results are that in many cases the MJoin operator produces its output sooner than any tree of binary join operators; that, like the streaming binary join operators on which it is based, MJoin is ideal for in-memory joins but can be extended to provide streaming behavior in the presence of memory overflow; that a technique we term "coordinated flushing" can improve the output rate in the presence of overflow; and finally, that the addition

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

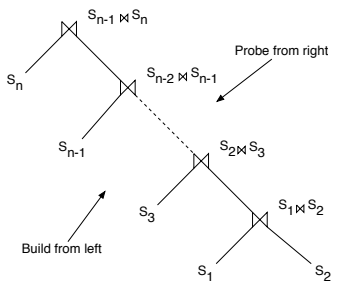


Figure 1: A traditional binary join execution tree

of the MJoin operator introduces an interesting new optimization problem, the problem of deciding how best to partition a large multi-way join into a set of one or more MJoin operators.

To understand the motivation for our work, consider a scenario in which a system runs a multi-way join query over remote data sources arriving as streams. Using traditional query evaluation techniques, the execution plan would be organized as a sequence of binary join operators, as shown in Figure 1. If we assume the joins are implemented with the hash join algorithm, using standard terminology and notation, the system would build hash tables on the left and probe them from the right.

While such an approach has proven effective in traditional centralized systems, in a scenario where inputs are remote the situation is different. To see this, note that in Figure 1, the plan will start producing results only after the hash tables for the left inputs, S_1 and S_3 through S_n, have been built, since with standard hash join operators the build phase of the left input must complete before the probe phase of the right input starts. Worse yet, if the right input of any join blocks, the whole tree blocks. Symmetric binary operators, such as the symmetric hash join [14], were proposed to address this problem by eliminating blocking behavior.

However, even with symmetric nonblocking binary operators, problems may arise. Assuming that all inputs of Figure 1 are remote and that each join is evaluated with a symmetric binary join algorithm, consider the case in which an S_1 arrival joins with $X_{1,2}$ already read S_2 tuples. These $X_{1,2}$ tuples are propagated upstream and, if they contribute to the final result, they have to go through each step of the execution tree until they appear in the output. At each step, the operator at that step handles these tuples, inserting them into one hash table and using them to probe the other. That creates a large number of in-flight tuples, causing additional storage and communication overhead. This overhead can increase the system resources required per output tuple, which in turn can slow the output rate.

A tree of binary operators also introduces secondary, subtler effects, which are inherent in the binary execution tree paradigm. The issue is that if different streaming

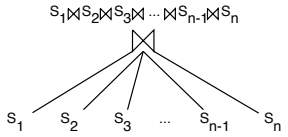


Figure 2: A multiple input join operator

sources deliver their inputs at different rates, the eventual output rate can differ as a function of which tree of binary operators (e.g., deep or bushy, fast inputs high in the tree or at the leaves) the optimizer chooses. This dependence is exacerbated when some or all of the operators in the tree overflow their memory quotas and spool some fraction of their inputs to disk for later processing. Finding the tree that optimizes the output rate in such a scenario is challenging; even worse, if the input rates vary over time, there may be no single tree that is best for the entire duration of the query execution, and the complexity of on-the-fly query plan restructuring becomes necessary [7].

However, it is not clear that the multiway join will be superior in all cases. A particular concern is whether replacing a tree of binary operators by a single multiway operator will cause excessive recomputation of partial results. Another concern has to do with memory overflow — specifically, how can a multiway join flush the many hash tables it builds in a consistent way?

Our goal in this paper is to investigate the opportunities and challenges presented by the introduction of a multi-way streaming join operator. The rest of the paper is organized as follows: Section 2 explains how the previously proposed symmetric binary hash join can be extended to yield the MJoin algorithm, while Section 3 gives our experimental study of the algorithm. Section 4 discusses related work, while Section 5 presents our conclusions and identifies future directions. Finally, for the interested reader, Appendix A presents a rate-based cost model for the proposed operator.

2 Algorithm Description

The basic idea of the MJoin algorithm is simple: generalize the symmetric binary hash join and the XJoin [11] algorithms to work for more than two inputs. However, it turns out that the details are somewhat tricky. The issue is that the algorithm must be ready to accept a new tuple on any input stream at any time; upon such an arrival, it must probe the other hash tables and generate a result as soon as possible; and finally, it must ensure that each result tuple is generated exactly once. These goals are rendered even more complex when some of the inputs overflow the space allocated for their hash tables and tuples must be spooled to disk for later processing.

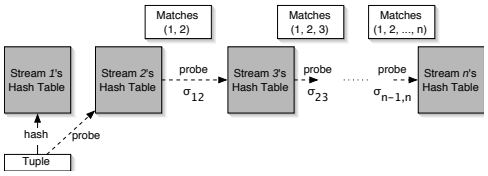


Figure 3: The probing sequence during MJoin

2.1 The Basic Algorithm

Generalizing the symmetric binary hash join to work for more than two inputs is straightforward. The algorithm first creates as many hash tables as there are inputs. When a new tuple arrives at an input, it is inserted into the corresponding hash table and used to probe the remaining hash tables. This generates every possible result tuple that can be produced by joining the new arrival with the memory resident tuples of the other relations. Not all hash tables will be probed for every arrival, as the sequence of probes stops whenever a probe of a hash table finds no matches (since in this case it cannot produce answer tuples.) Figure 3 shows this sequence, where each probe operation is annotated with the probability of its taking place, which is equal to the selectivity of the previous predicate in the sequence (the σ_* factors in the figure. Note that, in general, the σ_* factors may be a function of time.) For instance, for the second probe operation to execute, the first one has to produce matches. The sequence is organized in such a way so that the most selective predicates are evaluated first and it is different for each input. This ensures that the smallest number of temporary tuples is generated.

2.2 Choosing a Probing Sequence

Choosing the correct probing sequence is an important parameter when "setting up" an MJoin operator. To give a brief overview of how this can be easily achieved, consider the case of a query being executed over m inputs. Furthermore, let us assume that the selectivity factors between the joins remain constant throughout query execution. As we mentioned, a simple heuristic of choosing the best probing sequence is to evaluate the least selective join first. In that respect, all we need to do is sort the selectivity factors for each input and that will give us the optimal probing sequence for that particular source. We can sort the $m-1$ factors for the i^{th} input in $(m-1)\log m-1$ steps. Given that we will do this for all m inputs this results in a complexity of $\mathcal{O}(m(m-1)\log m-1)$. We further examine the impact of the probing sequence on the MJoin output rate in Section 3.3.

2.3 Re-computation of Intermediate Results

Because the MJoin operator does not store partial tuples generated by prefixes of probing sequences, it is possible that an MJoin will recompute results that are computed only once in a tree of binary operators. For example, consider a three-way join query between streams S_1, S_2 and S_3. Assume that at some point in time there have been n_1 tuples received from S_1, n_2 from S_2 and n_3 from S_3. Furthermore, assume that a new tuple arrives from S_1 matching with $\sigma_{12}n_2$ tuples from S_2 but with no tuples from S_3. The partial result set of $\sigma_{12}n_2$ result tuples matching with the S_1 arrival are discarded. Then a new tuple arrives from S_3 which matches with the existing partial result set. Clearly, whenever such a situation arises, intermediate results have to be re-computed.

This presents a problematic situation for MJoin, particularly in the presence of inflating join predicates, i.e., predicates for which a joining tuple from one stream joins with multiple tuples from the other streams. In such cases it may be better to materialize the intermediate result set, in other words, to break a single MJoin into smaller MJoins or, in the limit, to break it into a tree of binary joins. This introduces an interesting optimization problem which we consider later in this paper.

In our experiments we did not observe cases in which the recomputation made the MJoin produce results more slowly than a tree of binary operators. This is due to a number of factors: (i) a good choice of a probing sequence will minimize the effect; (ii) probing a hash table for matches is less intensive a CPU operation than inserting a tuple into a hash table; (iii) there are significant memory savings by not storing temporary matches; (iv) queries over streams will most probably contain window predicates, which tends to reduce the number of tuples with which other tuples join, which means that the risk of recomputation is lessened; (v) highly selective join predicates are common (for example, key-foreign key joins), which also lessens the risk of recomputation.

For the interested reader, in the appendix, (A.2) we present a cost model by which one can decide whether a multi-join query should be decomposed into multiple MJoin's given the selectivity factors of the predicates.

2.4 Handling Memory Overflow

We now turn to consider how to handle the case where the inputs may be too large to buffer entirely in memory. Before discussing how to deal with this issue with the MJoin, we review how memory overflow is dealt with in the binary XJoin.

The XJoin, like the Hybrid Hash Join, partitions each of its inputs using some hash function. Each partition has an in-memory portion, and a disk-resident portion. Initially, the disk resident portions are empty. The XJoin has three phases:

1. *The memory-to-memory phase.* If a tuple t arrives on an input stream and there is room in the in-memory portion of the partition to which t hashes, t is inserted into the hash table for its stream and

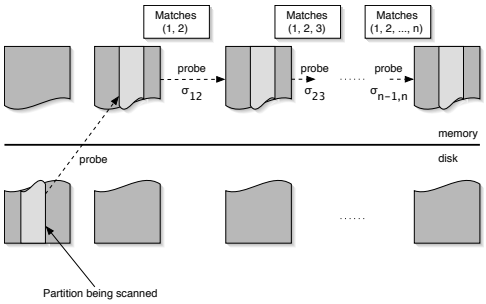


Figure 4: Disk-to-memory operation of MJoin

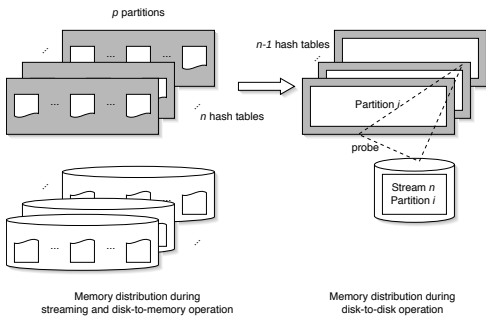


Figure 5: Disk-to-disk memory redistribution and operation of MJoin

probed into the in-memory portion of the corresponding partition of the other stream. If the in-memory portion of the stream to which t hashes is full, then it is flushed to disk (adding to the disk-resident portion of this partition.)

2. *The disk-to-memory phase.* If both of the input streams block, the XJoin takes advantage of this "idle" time to process some of the disk-resident tuples. In more detail, XJoin reads (part of) the disk-resident portion of some partition into memory, then probes it into the in-memory portion of the corresponding partition. In this way the XJoin can continue to generate results while the inputs are blocked. If one of the inputs becomes unblocked, the XJoin reverts to the memory-to-memory phase. The generalization of this phase for MJoin is presented in Figure 4.

3. *The disk-to-disk phase.* After both input streams are exhausted, the XJoin "cleans up" with a disk-to-disk phase, producing all tuples that may have been missed by previous memory-to-memory or disk-to-memory phases. It does this in a way similar to the hybrid hash join: it picks a partition of one of the inputs, builds an in-memory hash table on the (formerly) disk resident portion of this partition, then reads the disk-resident portion of the corresponding partition from the other input, probing the hash table and generating results. Dealing with this third phase in the context of the MJoin and in particular how memory is redistributed when the clean-up phase is executed, is depicted in Figure 5.

The XJoin must take some care to ensure that no result tuple is generated twice; it does so through the use of timestamps.

The same three-phase approach used by XJoin will work for MJoin. However, because of the multi-way nature of MJoin, there are some subtleties with MJoin that do not arise for XJoin. These arise both in the handling of memory overflow and in the use of timestamps to avoid redundantly generating answer tuples. We consider each in turn.

2.5 Memory Overflow in MJoin

Although XJoin's three phase approach to handling memory overflow works for MJoin, with MJoin, in the presence of multiple relations overflowing the output rate may suffer. In fact, maximizing the output rate for MJoin appears to be a very challenging and intriguing problem. It is a function of the input rates of the streams, the join selectivities between the streams, and how MJoin decides to spill the various inputs to disk in the case of memory overflow. To gain some insight into the problem, we consider an important aspect of this problem in a simplified scenario. We explore this problem further in Section 3.5.

We emphasize that our primary goal is to maximize the output rate during the memory-to-memory phase of the MJoin. As with the binary XJoin, in MJoin, the disk-to-memory phase is intended to allow the system to generate outputs while its inputs are blocked, while the disk-to-disk phase is intended to generate any final answers after the inputs have terminated. Interestingly, for the MJoin, how we handle memory overflow determines the output rate of the memory-to-memory phase.

Consider an MJoin $R_1 \bowtie R_2 \ldots \bowtie R_k$, where each of the R_i arrives at the same rate. Suppose that the join selectivities between the streams are all equal to the same constant σ, and that at time t the number of tuples that have arrived on each stream is n_t. Then a new input tuple on any stream can be expected to produce $n_t^{k-1}\sigma^{k-1}$ output tuples.

Now suppose that available memory has been exceeded, and that we have spilled an equal fraction f of all of the streams to disk. Then the expected output of a new arrival will be reduced by a factor f^{k-1}, since only a fraction f of the n_t arrived tuples from each stream is available for probing, and the expected output would be $f^{k-1}n_t^{k-1}\sigma^{k-1}$.

The preceding analysis assumes that all tuples are spilled to disk randomly, without considering the values in their join attributes. It is interesting to ask if the output rate can be increased by smarter ways of spilling to disk. Suppose now that the join is a star join so that the join

predicate is of the form $R_1.A_1 = R_2.A_2 = ... = R_k.A_k$. Then consider the following modification of the basic MJoin. We partition each of the relations into p partitions by hashing on their join attribute. When it comes time to spill some portion of R_i to disk, instead of doing so blindly, we always spill from a coordinated set of partitions. That is, if we spill tuples from partition 3 from R_1 to disk, for all the other streams we also spill tuples from partition 3 to disk, only going on to another partition after all partition 3 tuples have been flushed. We call this approach "coordinated flushing."

Using coordinated flushing, when a new tuple arrives on any input stream, if it falls into an in-memory partition, it is immediately probed in the in-memory partitions of the other streams; if it falls into a disk resident partition, then it is added to an output buffer for that partition and not probed in the other streams'.

With this modification, we have that the expected number of result tuples generated by a new arrival is either zero (if it falls into an on-disk partition, which will occur with probability $1-f$), or $n_t^{k-1}\sigma^{k-1}$ (if it falls into an in-memory partition, which will occur with probability f.) To see the second term, consider the case when we are probing into the in-memory partition of some stream R_i. We have in total n_t tuples that have arrived on that stream; if we probe the current tuple t into any of the on disk partitions, it will not produce any tuples (since they are in different partitions), so the fact that these tuples are on disk does not affect the output rate.

Using this simple analysis, we see that the number of tuples generated in response to a new arrival using coordinated flushing is $fn_t^{k-1}\sigma^{k-1}$ tuples. This is in contrast to the no overflow case, in which a new arrival is expected to generate $n_t^{k-1}\sigma^{k-1}$ tuples; and the random flushing case, in which the arrival is expected to generate $f^{k-1}n_t^{k-1}\sigma^{k-1}$ tuples.

It is interesting to consider the special case of $k = 2$, the binary join for which XJoin was proposed. There we get the initially perplexing result that with random flushing, the expect number of tuples produced due to an arrival would be $fn_t\sigma$; with coordinated flushing, the expected number would be $fn_t\sigma$, exactly the same number. On further thought this makes sense; with random spilling, there is a 100% chance a new tuple will be probed into the other stream's table, where it will find a fraction f of the table in memory; with coordinated flushing, there is an f percent chance it fall into an in-memory partition, in which case it will probe the (same) in-memory partition of the other table, and the fact that other partitions of the other table have been flushed to disk is irrelevant.

Coordinated flushing over all inputs of the join is possible only with "star joins", since the relations being flushed in a coordinated fashion must have the same join attribute. We think this kind of join may be common in streaming applications; for example, often we need to join multiple streams on the time attribute. The general MJoin algorithm, though, is applicable regardless of whether the query under evaluation is a star join or not. The reason for this is that random spilling can always be used throughout the execution of the algorithm.

2.6 Redundant Tuple Detection

As was mentioned in Section 2.4, during the disk-to-memory and disk-to-disk stages there is a possibility that output tuples can be generated multiple times. We call these tuples redundant tuples and in this section we show how to avoid them. There are only two ways in which a tuple and all its matches render a result redundant: (i) if they were present in the memory-resident portions of the hash tables at the same time, or (ii) the tuple was flushed to disk and used to probe the memory-resident hash table portions while its matches were still in memory. Notice, however, the universal qualification of the above clause: a tuple and *all* its matches. Even if a single pair of matches is not a duplicate, the partial join result is a new one and it should be propagated.

The way MJoin eliminates duplicates is based on time-stamps. Each tuple is assigned two time-stamps: one for its arrival into the system and one for its departure from main memory. Additionally, a separate log is kept for each partition of each table, keeping track of when the partition was used for a disk-to-memory probing stage, as well as the latest disk-resident tuple time-stamp for that partition. Deciding whether a candidate result has already been propagated or not is a matter of evaluating two conditions. Assuming a tuple T_i from input i being scanned and a match T_j being tested, then $T_i \bowtie T_j$ has been propagated in the following cases:

1. arrival(T_i) > arrival(T_j) and
   arrival(T_i) < departure(T_j), which means that T_i arrived while T_j was in memory.

2. latest(partition(T_i)) > arrival(T_j) and
   probe(partition(T_i)) > arrival(T_j) and
   probe(partition(T_i)) < departure(T_j), which means that T_j has already been probed by a previous disk-to-memory join of T_i's partition.

This check is performed in a single direction during the second stage, while it is carried out in both directions during the third stage. Moreover, for an overall join result (i.e., $\bowtie_{i=1...n}(T_i)$) to be propagated, the test has to be passed by all possible pairs of tuples. At first glance this may seem as an expensive operation, however, it can be shown that the number of checks is equal to the number used in a binary execution tree employing XJoin as the evaluation algorithm.

2.7 A Concrete Example

To present the algorithm more concretely, we will use the example of Figure 6: a three-way join between S_1, S_2 and S_3. We assume that there is one partition per stream,

while each stream has been allocated a buffer capable of holding two tuples; each tuple is represented by its value in the join attribute. Each tuple is annotated with the interval for which it is/was memory-resident[1], while each disk-resident partition is annotated with the last time it was used to perform a probe of the in-memory hash tables. (A value of "-1" means the partition has never been used to probe.)

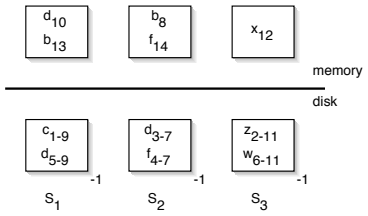


Figure 6: An MJoin scenario

Next, assume two tuples from S_3 arrive; the first one with a value of f and the second one with a value of b, as is the case in Figure 7. The f tuple is given the timestamp 15, inserted into S_3's hash table and used to probe the other hash tables for matches. The execution engine should define the order in which it probes the hash tables, before execution begins. If we assume the probing sequence is $\{S_1, S_2\}$ in our example, no temporary result tuples will be generated. Had the sequence been $\{S_2, S_1\}$, the temporary tuple $\{f_{14}, f_{15}\}$ would have been generated. The second arrival, with a value of b, is given the timestamp 16 and it has the following effects: (a) the tuple is inserted into S_3's hash table, causing the table to overflow, and (b) the value of the tuple's join attribute is used to probe the rest of the hash tables, producing a single result tuple. Figure 8 shows the results of both operations. (After f_{15} has been handled as well.)

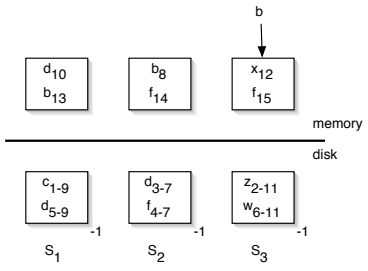


Figure 7: A new arrival from S_3

Next, let us assume that one more tuple has arrived from the third stream with a value of d, obtaining the timestamp 17. At this point, the inputs block, so the algorithm moves on to its disk-to-memory stage, choosing to use S_2's disk-resident portion. It probes S_1's and S_3's memory-resident hash table portions, outputting one result tuple, as shown in Figure 9.

[1]A single number denotes a tuple that has not been flushed to disk yet.

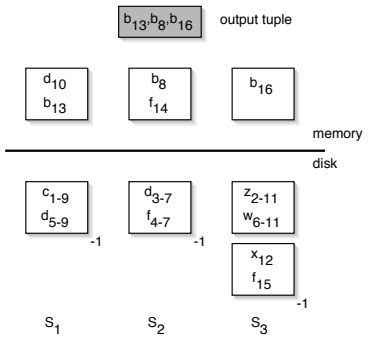


Figure 8: The new arrival from S_3 in Figure 7 results in additional overflow tuples reaching the disk and an output tuple being generated

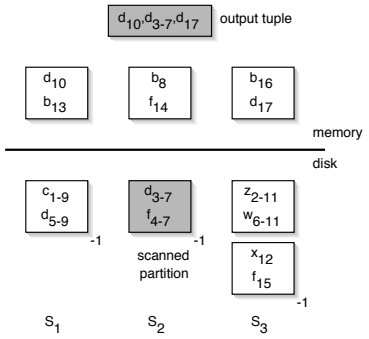


Figure 9: An example of disk-to-memory operation; S_2 is used for probing.

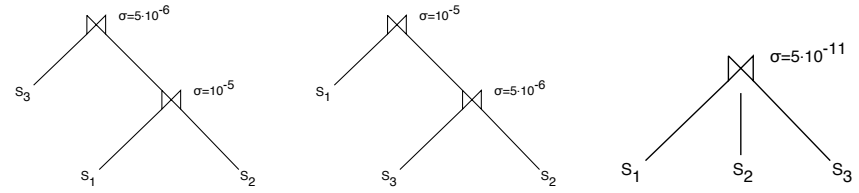
Finally, assume all streams send their *end-of-stream* message after d_{17}'s arrival. This causes the algorithm to revert to its final clean-up stage and perform a three-way disk-to-disk join. Notice that there are two possible result tuples with d values. Only one of them, however, should be propagated to the output, that one being $\{d_{5-9}, d_{3-7}, d_{17}\}$ since the other one, $\{d_{10}, d_{3-7}, d_{17}\}$, has already been generated during the second stage.

3 Experiments

In this section we will present our experimental results from a prototype implementation of MJoin.

3.1 Experimental Setup

Our goal was to measure the performance improvement we would obtain in comparison to other algorithms designed to work over streaming sources. To do so, we developed a stand-alone prototype of the algorithm in Java.



Fluctuating Stream High Fluctuating Stream Low MJoin

Figure 10: The plans used during experimentation with fluctuating input rates

The queries we used were for the most part variants of the Wisconsin Benchmark's *JoinABPrime* query [3], extended to handle multiple sources.

IBM's *jikes* compiler was used for byte-code generation, which was executed using SUN's *HotSpot* virtual machine. All experiments were conducted on a 1GHz Intel Pentium Processor with 1GB of physical memory, running RedHat Linux 7.2. To simulate streaming sources, we assigned an arrival rate to each input and then inserted, between arrivals, random delays following a Poisson distribution with the given arrival rate as its mean[2]. As a rule, we used the slowest stream's inter-arrival rate as the operator's blocking threshold. The joins in Section 3.2 were key-to-key joins so that the choice of probing sequence was not important. For instance, considering three streams R, S and T the where-clause of the query would be:

```
where R.unique1 = S.unique1 and
      S.unique1 = T.unique1
```

We turn to consider inflating joins, in which the probing sequence does matter, in Section 3.3.

3.2 Resilience to Fluctuations in Input Rate

In the experiments of this section we used a three-way-join query between streams S_1, S_2 and S_3 and organized it in the three ways presented in Figure 10. The input cardinalities were 100,000 tuples for streams S_1 and S_2 and 200,000 tuples for stream S_3; we also allocated a memory buffer less than the inputs' sizes. More importantly, we did not keep a constant mean arrival rate for all inputs; in particular, we varied the input rate of S_3 so that it started off fast, slowed down towards the middle of the query and gained speed again in the last third of execution. The objective of this experiment was to verify MJoin's resilience to input rate fluctuations. The results are presented in Figure 11.

As exhibited in the performance results, MJoin had a higher output rate in comparison to the other two plans. An equally interesting point, however, is the switching between performances of the two non-MJoin plans. While the fluctuating stream was fast, the plan that kept it at the top of the execution plan was faster than the one

[2]A Poisson arrival process means that the inter-arrival process follows an exponential distribution with a mean equal to the inverse of the Poisson process's mean.

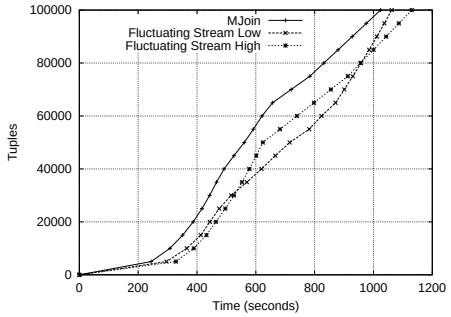


Figure 11: Performance for fluctuating input rate

keeping it at the bottom. Once the stream slowed down, the output rates were reversed, and when the stream returned to its initial rate, the original relative performance again appeared. This validates our intuition that while it is impossible to pick a single tree of binary operators that is always optimal when input rates vary, MJoin is stable and dominates throughout.

3.3 Inflating Joins and the Impact of the Probing Sequence

Our previous experiments dealt with selective joins, more specifically key-to-key joins. In such a scenario, the probing sequence is not significant. In an inflating join scenario, however, as was mentioned in Section 2.1, it is important for the probing sequence to be declared in such a way that the most selective predicate in evaluated first. In that case, the smallest number of temporary results is generated. To show the detrimental effects of the wrong probing sequence, we generated a three-way-join query in the style of the one presented in Section 3.2 but modified it in the following way: S_1 and S_3 both had an input cardinality of 10,000, while S_2 an input cardinality of 15,000. Moreover, instead of joining on the `unique1` attribute of the relation, we joined on one with cardinality 1,000. (For instance, each tuple of S_1 joins with 1,000 tuples of S_2 so their join produces 150,000 results.) Finally, there is no fluctuation in the streams' incoming rates. The mean inter-arrival delay for S_1 and S_2 is set to 20 millisecondss, while the mean inter-arrival

Plan shape	*Hashes*	*Moves*	*Comparisons*
Deep (XJoin)	185,000	185,000	1,500,000
MJoin, correct sequence	35,000	35,000	1,656,727
MJoin, wrong sequence	35,000	35,000	2,004,639

Table 1: Increase in the number of operations due to the wrong probing sequence

delay for S_3 is set to 5 millisecondss.

We employed two MJoin plans and a deep binary join plan. For one of the MJoin plans we had the correct probing sequence (from most selective to least selective predicate) while for the other we declared the sequence in the worst possible way for the MJoin operator, i.e., instead of evaluating the most selective predicate first for all inputs, we evaluated the least selective predicate first. In the query at hand, the two 10,000 tuple streams would first probe the 15,000 tuple stream, producing a larger number of temporary result tuples. The XJoin plan used for comparison evaluated the most selective predicate first.

We instrumented the code in such a way that would allow us to count the number of operations (hashes, moves, comparisons) each operator would perform. The number of extra comparisons between the MJoin employing the correct probing sequence and the MJoin employing the wrong probing sequence is shown in Table 1 while the performance results are presented in Figure 12.

As we see from the performance results, the number of extra comparisons performed due to the wrong probing sequence are almost 25% of the number of expected comparisons. In fact, after some point these extra comparisons make the MJoin perform worse than the XJoin plan. Another important conclusion shown in Table 1 stems from the comparison of operations between the XJoin and the MJoin plans. Although the XJoin plan performs fewer key comparisons, it also performs much more hash and move operations. Clearly, this means that choosing the correct probing sequence for the MJoin operator is crucial.

3.4 Window Joins

When we move from traditional joins over remote sources to consider joins over infinite streams of data, it makes sense to consider window-based joins, i.e., joins that only pair tuples within a bounded time interval of each other. This is because without some sort of window on which tuples can join, in the limit, infinite streams will require infinite memory. To simulate a window-based join scenario, we created a three-way join query, over three relations, each relation containing one million tuples. Moreover, we imposed two window predicates over the query, with each predicate having a horizon of ten thousand tuples, i.e., the predicate would only be evaluated over tuples appearing within 10,000 tuples of each

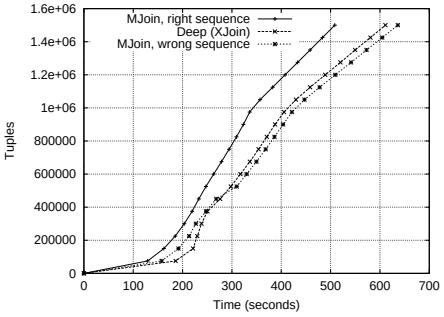


Figure 12: Performance decrease by using the wrong probing sequence

other. The shapes of the three plans we used were similar to the plans of Figure 10; however, each input contained 1,000,000 tuples, streams S_1 and S_2 had a 3 millisecond mean inter-arrival delay, while stream S_3 had a mean inter-arrival delay of 1 millisecond. The experimental results are shown in Figure 13[3]. As in all previous experiments, the MJoin plan is faster. This was expected for one simple reason: by choosing an MJoin evaluation plan for a window of 10,000 tuples, we are able to keep all computation within memory limits, and MJoin has been optimized for in-memory, streaming behavior.

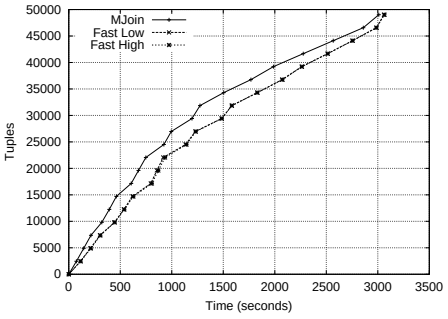


Figure 13: Window join performance

3.5 Coordinated Flushing

In this experiment we wanted to test MJoin's ability to handle memory overflow scenarios, as well as to maximize the output rate by employing the technique of coordinated flushing, introduced in Section 2.5. To do so, we generated a six-way join query. The inputs were divided in two triplets, each triplet materializing the result of a

[3]The "Fluctuating Stream High" plan of Figure 10 corresponds to the "Fast High" plan of Figure 13, while the "Fluctuating Stream Low" plan to the "Fast Low" plan.

Stream	*Tuples*	*Delay*	*Attribute*	*Cardinality*
S_1	10,000	1	`unique1`	10,000
S_2	100,000	5	`tenk6`	10,000
			`unique14`	100,000
S_3	100,000	5	`tenk6`	10,000
S_4	10,000	1	`unique1`	10,000
S_5	100,000	5	`tenk6`	10,000
			`unique14`	100,000
S_6	100,000	5	`tenk5`	10,000

Table 2: Input parameters for the memory overflow scenario. "Delay" is measured in milliseconds.

star-join. The two star-join result sets where then joined on a different attribute. To present it more concretely, assuming S_1 to S_6 are the inputs, the where-clause of the query we ran was the following:

```
where S1.unique1 = S2.tenk6 and
      S1.unique1 = S3.tenk6 and
      S4.unique1 = S5.tenk6 and
      S4.unique1 = S6.tenk6 and
      S2.unique14 = S5.unique14
```

We created three execution plans for the same query. For the first plan, which is shown in Figure 14, we used two MJoins to generate the results of the two star-joins, and then a third MJoin to join these results. The second plan was the same MJoin setup, only employing coordinated flushing to handle memory overflow. The third plan was set up as a binary plan using XJoin. We then streamed the inputs into the system, using the input rates shown in Table 2 (which also shows the input sizes and attribute cardinalities) while the results are shown in Figure 15.

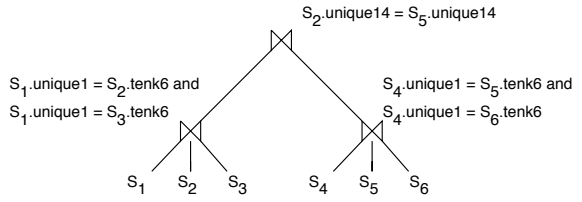


Figure 14: The MJoin used in the memory overflow scenario

The effect of coordinated flushing is evident in the plot of Figure 15 since by keeping more tuples with a higher probability of joining in memory the output rate is maximized. There are certain "jumps" in the output curve for the coordinated MJoin and these jumps appear to be regular. These jumps arise from an artifact of how we have implemented coordinated flushing. What happens is that when an input overflows, one of its partitions is spilled to disk. Since we spill the entire partition to disk, at least temporarily this input has fewer tuples

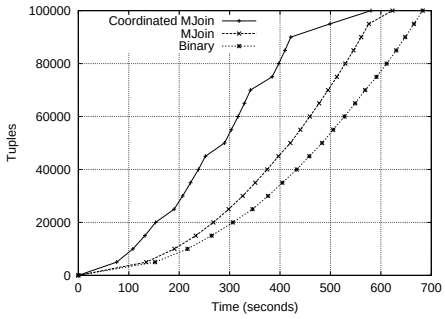


Figure 15: Performance during memory overflow and improvements obtained by employing coordinated flushing

in memory than before, so the output rate drops. Eventually, however, the in-memory partitions grow so that there are again more tuples in memory, and the output rate rises again. The last part of the curve in which the output rate is substantially decreased is due to the fact that a higher percentage of the output has already been generated, due to coordinated flushing, so the operator simply waits for the rest of the output to be produced in its clean-up phase.

3.6 On the Need for Optimization

The next set of experiments we conducted had to do with investigating and proving that even with an operator like MJoin, the need for optimization of join-trees still exists. To do so, we experimented with a six-way join query and five plans for that query. Four of those five plans are shown in Figure 16, where each input is annotated with its size in tuples and its inter-arrival delay. In Figure 16, two binary join plans and two MJoin plans but with smaller (i.e., fewer input streams) MJoin operators are shown. The fifth, not depicted, plan was a single MJoin operator. Execution of these plans, along with the single MJoin plan, yields the performance shown in Figure 17. Though the single MJoin operator is faster in the beginning, its performance degrades over time, while, as time goes by, even the bushy binary plan overtakes it.

Our initial explanation of MJoin's performance degradation had to do with its per-input cost (as this is modeled in Appendix A.) To further follow our intuition, we measured the actual cost in clock ticks of the various parameters appearing in MJoin's cost expression, by accessing the processor's hardware counters. These measurements are presented in Table 3.

We then focused on the denominator of Equation 2, which is the operator's per-time-unit cost. Performing the computation yields that, roughly, the per-time-unit cost is $1.52 \cdot 10^{-3}$ seconds. Looking at the streams' input rates, as shown in Figure 16, it is easy to see that this

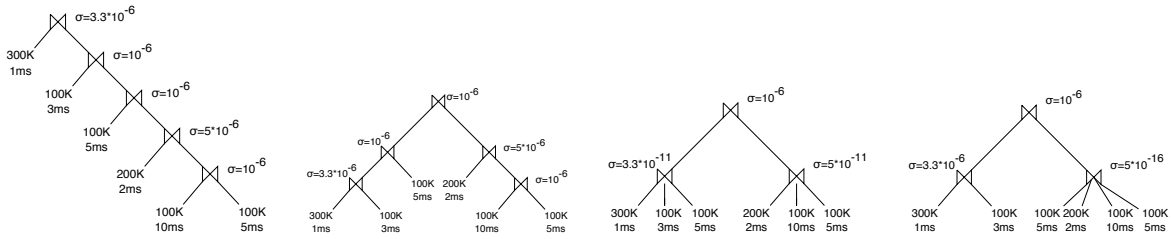


Figure 16: Six-way join execution plans

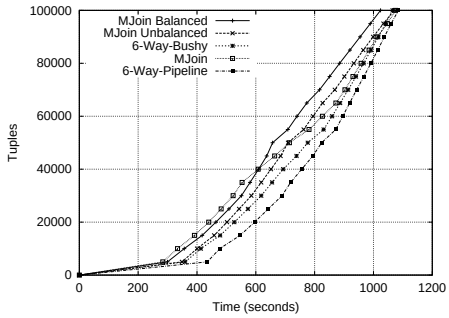


Figure 17: Six-way join performance

Operation	*Cost (clock ticks)*	*Cost (seconds)*
hash	175.008	$1.75 \cdot 10^{-7}$
move	426.518	$4.27 \cdot 10^{-7}$
comp	49.133	$4.91 \cdot 10^{-8}$

Table 3: Cost of various operations as measured by the processor's hardware counters. We were using a 1GHz processor; one clock tick is equal to 10^{-9} seconds.

time is greater than the fastest stream's inter-arrival rate (10^{-3} seconds for the 300,000 tuple stream.) This translates into a backlog of tuples being created for that particular stream; as far as this stream is concerned the CPU is too slow to handle its rate. As time goes by, this backlog starts to dominate the stream's input rate, degrading MJoin's performance. This problem does not occur in the case of the other plans, which use multiple operators, hence are not sensitive to the total number of input streams in the join.

The results show that the plans with multiple small MJoin operators can outperform both the single large MJoin operator and the plans built up with only binary operators. It is interesting to note that the single MJoin operator, while not the best plan overall, dominates in the first stages of the query execution. Clearly, this presents a great opportunity (or challenge, depending upon your perspective!) for query optimization: ideally, the optimizer needs to know how many result tuples it should optimize for, then it needs to choose a plan that distributes the join over the optimal number of MJoin operators of with the right number of inputs. Cost formulas like the ones presented in the appendix can assist the optimizer in this task.

4 Related Work

Join algorithms have been extensively studied in the context of relational database systems, with [5] being the seminal paper on hash-based join algorithms. Also related to our work are Bloom filters [4] and bit-map join indices [10] as ways of efficiently computing joins with the aid of pre-computation. Additionally, and closer related, are the techniques of hash teams [6] and generalized hash teams [8], in which the objective is to minimize the number of performed operations in hash-based evaluation plans by sharing computation and hash table storage space. Again, none of this previous work considered the issue of maximizing the output rate in the presence of varying and unpredictable input rates.

The most relevant remaining work deals with symmetric algorithms and adaptive query execution. The first algorithm to explicitly take into account the streaming nature of its inputs was the Symmetric Hash Join [14]. XJoin [11] extends this work by providing an efficient way to spill overflowing inputs to disk and later join them to produce the final output, while in [7] the authors present a way of adapting symmetric hash join into hybrid hash join whenever inputs become too large to fit in memory. To the best of our knowledge none of the previous work on streaming join algorithms considers the possibility of moving beyond binary operators to multiway join operators.

Evaluating queries over streaming inputs has been studied in the context of query scrambling [12] and adaptive query execution [1, 7]. In the former approach, an execution plan is monitored so that whenever a blocked input is detected the operator(s) using that input are preempted and other, non-blocked, operators are run instead. Adaptive execution frameworks employ similar performance monitoring as their decision strategy but instead of giving precedence to certain operators, they dynamically alter the plan in a way that is believed to over-

come any performance bottlenecks. Our multi-way join operator addresses a similar problem, but without requiring any explicit monitoring or dynamic plan modification. Of course, we do not claim our approach abolishes the need for adaptive execution, since many queries cannot be reduced to a single multi-way join operator; rather, we claim that the introduction of a multi-input join operator reduces the burden placed on an adaptive framework. The MJoin approach is similar in some respects to SteMs [9]. The idea behind a SteM is to push state information from the operator to the source. We differ from that approach in two aspects; a SteM is essentially a first-class operator in the query plan. As such, it introduces additional dataflow logic [1] in the execution plan. Additionally, SteMs try to address the more general problem of sharing computation between sources (SteMs can be shared), which is something we do not deal with in the context of a single algorithm.

5 Conclusions and Future Work

Join algorithms have been extensively explored in the research literature, and at many points during the history of our community one might have concluded that there was nothing more to be discovered about them. However, at least to date, this has always been false — the recent flurry of interest in streaming and/or adaptive join algorithms is one counterexample.

In this paper we continue this investigation into streaming and adaptive join algorithms, but with a new twist: by considering multi-way (beyond binary) symmetric join operators. We have shown that in many cases a multi-way join operator can produce its output in a streaming fashion and at a faster rate than any tree of binary join operators. The introduction of a multi-way join operator has also introduced interesting issues with respect to how to handle memory overflow and how to choose a probing sequence within the join. In future work we plan to explore the optimization problem of how to best to split a very large multi-way join into a set of smaller multi-way joins.

Acknowledgements This work was supported by NSF grant ITR 0086002 and in part by an NCR graduate fellowship.

References

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD Conference*, 2000.

[2] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, 1991.

[3] D. Bitton and C. Turbyfill. A retrospective on the Wisconsin Benchmark. In *Readings in Database Systems*, 1988.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[5] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *Proceedings of SIGMOD'84*, pages 1–8. ACM Press, 1984.

Notation	*Description*
hash	Cost of hashing a key
move	Cost of moving an object in memory
comp	Cost of comparing two keys in memory
r_i	The input rate of the i^{th} stream
σ_k	Selectivity factor of join predicate k

Table 4: Cost variables and notation used for modeling

[6] G. Graefe, R. Bunker, and S. Cooper. Hash joins and hash teams in Microsoft SQL server. In *VLDB Conference*, 1998.

[7] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *SIGMOD Conference*, 1999.

[8] A. Kemper, D. Kossmann, and C. Wiesner. Generalised hash teams for join and group-by. In *VLDB Conference*, 1999.

[9] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, 2002.

[10] P. E. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, 1995.

[11] T. Urhan and M. J. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

[12] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *SIGMOD Conference*, 1998.

[13] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD Conference*, 2002.

[14] A. N. Wilschut and P. M. G. Apers. Pipelining in query execution. In *Conference on Databases, Parallel Architectures and their Applications*, 1991.

A Cost Expressions for MJoin

In this appendix we present a cost model for MJoin operators. Such a cost model is essential if optimizers are to be able to make good decisions about when and how to employ MJoin operators; it is also useful in explaining some of our experimental results in Section 3.6.

A.1 Rate-based Cost Expressions

The purpose of this section is to extract an output rate estimator for MJoin in the fashion of [13]. We will focus on the first stage of the algorithm, since this is the one in which MJoin exhibits streaming behavior. Our cost expressions will make use of the cost variables and notation presented in Table 4.

The output rate of any process is the number of transmitted entities over the time needed to make the transmission [2], i.e.:

$$\text{Output rate} = \frac{\text{Number of outputs transmitted}}{\text{Time needed to make the transmission}} \quad (1)$$

In our approach, we assume the n inputs have rates equal to $r_1, r_2, \ldots, r_n$ tuples/second respectively. As a first step we will concentrate on the numerator in Equation 1 and we will first make a discrete time approximation of

the output rate, before generalizing to continuous time. Over the first second, the operator will receive r_1 tuples from the first stream, r_2 from the second one, and so on. The Cartesian product of these tuples and, hence, the total size of the input that the operator will filter, will then be equal to $C(1) = \prod_{i=1}^{n} r_i$. Assuming k join predicates in the query with each join predicate having a selectivity of f_k the total number of tuples transmitted for arrivals during the first second will be $T(1) = \prod_{j=1}^{k} \sigma_j \cdot \prod_{i=1}^{n} r_i$. During the next second of execution each stream i will have received an additional r_i tuples, a total of $2 \cdot r_i$ for each stream. The size of the Cartesian product is therefore $C(2) = \prod_{i=1}^{n} 2r_i = 2^n \prod_{i=1} r_i$ and the contribution of this input to the output will be $T(2) = \prod_{j=1}^{k} \sigma_j \cdot 2^n \prod_{i=1}^{n} r_i$. From this size, however, we have to discard the inputs handled during the first second of execution since these have been already propagated. After the first second, the contribution of the next second to the output becomes $T(2) = \prod_{j=1}^{k} \sigma_j \cdot 2^n \prod_{i=1}^{n} r_i - T(1) = \ldots = \prod_{j=1}^{k} \sigma_k \cdot \prod_{i=1}^{n} r_i \cdot (2^n - 1)$. By induction, we can prove that the number of transmitted outputs for any time point t will be given by the expression: $T(t) = \prod_{j=1}^{k} \sigma_k \cdot \prod_{i=1}^{n} r_i \cdot (t^n - (t-1)^n - \ldots - 2^n - 1) = \prod_{j=1}^{k} \sigma_k \cdot \prod_{i=1}^{n} r_i \cdot (t^n - \sum_{k=1}^{t-1} k^n)$.

The next step in extracting the operator's output rate is calculating the denominator of Equation 1. For an arrival in any given stream the following operations have to be performed: (i) hash the tuple, (ii) move it into its corresponding hash table, and (iii) probe the rest of the hash tables for matches. Notice, however, that not every tuple probes all hash tables. In a way resembling pipelined execution, it goes to a next hash table only if matches in the previous one exist, as depicted in Figure 3. In total, the cost per arrival will be equal to $hash + move + comp \cdot (1 + \prod_{j=1}^{k-1} \sigma_k)$, where $\prod_{j=1}^{k-1} \sigma_k$ is the cost induced if all probes have to be performed[4]. Since there will be $\sum_{i=1}^{n} r_i$ arrivals for a given second, that makes the time needed to make the transmission equal to $\sum_{i=1}^{n} r_i \cdot (hash + move + comp \cdot (1 + \prod_{j=1}^{k-1} \sigma_k))$. Substituting this last expression and $T(t)$ into Equation 1 yields MJoin's output rate (Equation 2). In Section 3.6 we saw how Equation 2 can identify cases where MJoin's performance might degrade.

$$r_o(t) = \frac{\prod_{j=1}^{k} \sigma_k \cdot \prod_{i=1}^{n} r_i \cdot \left(t^n - \sum_{k=1}^{t-1} k^n\right)}{\sum_{i=1}^{n} r_i \cdot \left(hash + move + comp \cdot \left(1 + \prod_{j=1}^{k-1} \sigma_k\right)\right)} \quad (2)$$

A.2 MJoin Decomposition Cost Expressions

Consider the scenario of a multi-join query over m streaming sources and the problem of deciding whether a single MJoin operator needs to be broken in two smaller MJoin operators, the first one operating over k input streams, while the other operating over l input streams (where $k + l = m$). Suppose that we are dealing with the situation of having n_i tuples for stream i. (This can mean either that the total input size for stream i is equal to n_i or that given the input rate r_i of stream i, after time t we expect n_i arrivals.)

It makes sense to decompose the single MJoin operation if it is cheaper to store the intermediate results from the k and l streams than it is to discard and recompute them. Assuming k_σ predicates between the first k sources and l_σ predicates between the remaining l sources, the the intermediate result size for the k streams is $\prod_{i=1}^{k_\sigma} \sigma_i \cdot \prod_{i=1}^{k} n_i$, while the intermediate result size for the l streams is $\prod_{i=1}^{l_\sigma} \sigma_i \cdot \prod_{i=1}^{l} n_i$. The cost of storing these two results sets is $(\prod_{i=1}^{k_\sigma} \sigma_i \cdot \prod_{i=1}^{k} n_i \prod_{i=1}^{l_\sigma} \sigma_i \cdot \prod_{i=1}^{l} n_i) \cdot (hash + move)$. On the other hand, probing the hash tables of the first k streams for matches will bear a cost of $\prod_{i=1}^{l_\sigma} \sigma_i \cdot \prod_{i=1}^{l} n_i \cdot comp$, that is the number of expected tuples from the l streams times the cost of a comparison. In a similar way, the cost of probing the remaining l streams for matches is equal to $\prod_{i=1}^{k_\sigma} \sigma_i \cdot \prod_{i=1}^{k} n_i \cdot comp$. Overall, the cost of splitting an MJoin of m inputs into two MJoins of k and l inputs will be equal to the quantity $S(k,l)$ given in Equation 3.

$$S(k,l) = \left(\prod_{i=1}^{k_\sigma} \sigma_i \cdot \prod_{i=1}^{k} n_i + \prod_{i=1}^{l_\sigma} \sigma_i \cdot \prod_{i=1}^{l} n_i\right) \cdot (hash + move + comp) \quad (3)$$

The number of times we discard the intermediate result set for each subset of inputs is equal to the number of times a probe from the other subset does not produce any matches. For the first subset of k inputs this number is equal to $\sum_{i=1}^{l_\sigma} (1 - \sigma_i) \cdot \prod_{i=1}^{l} n_i$, that is, the probability of *either one* (hence, the sum) of the l_σ predicates on the remaining l sources *not* producing matches, times the number of expected tuples from the l streams. The cost of discarding the intermediate result will then be equal to $\sum_{i=1}^{l_\sigma} (1 - \sigma_i) \cdot \prod_{i=1}^{l} n_i \cdot comp$ that is, the result size multiplied by the cost of probing (an in-memory comparison.) Using the same reasoning we can compute the cost of discarding the intermediate result for the remaining l inputs is equal to $\sum_{i=1}^{k_\sigma} (1 - \sigma_i) \cdot \prod_{i=1}^{k} n_i \cdot comp$. Overall, the cost of discarding the intermediate result sets is equal to the quantity $D(k,l)$ given in Equation 4.

$$D(k,l) = \left(\sum_{i=1}^{l_\sigma} (1 - \sigma_i) \cdot \prod_{i=1}^{l} n_i + \sum_{i=1}^{k_\sigma} (1 - \sigma_i) \cdot \prod_{i=1}^{k} n_i\right) \cdot comp \quad (4)$$

If $D(k,l) > S(k,l)$ then it is better to decompose the single m-way MJoin into one k-way and one l-way MJoin.

[4]The product's limit is set to $k-1$ instead of k since one probe will always take place.