Paper title:	Maximizing the Output Rate of Multi-Join Queries over Streaming Information Sources
Paper ID #:	325
List of authors:	Stratis D. Viglas, Jeffrey F. Naughton, Josef Burger
Contact author:	Stratis D. Viglas
Contact author's address:	Department of Computer Sciences, University of Wisconsin Madison, 1210 W Dayton st, Madison, WI, 53706, USA
Contact author's e-mail:	stratis@cs.wisc.edu
Topic area:	Core Database Technology
Category:	Research
Relevant topics:	Optimisation and Performance, Databases and database services in new context - Internet and the WWW

Maximizing the Output Rate of Multi-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 53706, WI
e-mail: {stratis,naughton,bolo}@cs.wisc.edu

Abstract

Recently there has been a growing focus in the research community on join query evaluation for scenarios in which input characteristics may not be entirely known and inputs enter the system at highly variable and unpredictable rates. The proposed solutions to date rely upon some combination of streaming binary operators and "on-the-fly" query plan reorganization to deal with this unpredictability. In this paper, we consider a different approach, and propose a multi-input streaming join algorithm we call MJoin. We show through experiments with a prototype implementation that in many instances the MJoin produces outputs sooner than any tree of binary operators, and that it adapts well to changing input parameters without query plan modification. This suggests that the MJoin operator may be a useful addition to systems that evaluate queries containing joins over streaming inputs.

1 Introduction

Join algorithms have been in the focus of the database research community for a long time. With the advent of the Internet, however, the assumptions underlying previous work have to be revisited. Though execution time minimization continues to be a key factor, two new aspects of the problem have come into focus: (i) the unpredictable streaming nature of the input sources, along with the lack of any information regarding their size and data distributions, and, (ii) the goal of producing results at the highest possible rate (rather than completing the computation as soon as possible). This paper addresses both of these issues by proposing a multi-input join algorithm that collapses multiple binary joins into a single multi-way join operation. Using this 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 all its inputs, there is no need to restructure a query plan in response to changing data rates in the inputs. The new multi-way join operator does require more memory than traditional binary hash-based join algorithms over disk-resident data. However, its memory requirements are lower than those of previously proposed streaming binary join algorithms.

Our operator is optimized for in-memory performance, so it is ideal if the inputs fit in memory, or if the join has accompanying "window" predicates that can be used to bound the memory required for each input. It is, however, designed in such a way that is able to flush overflowing inputs to disk and later process them, either whenever streaming inputs block, or whenever all streams finish. We have implemented a prototype of our algorithm and evaluated its performance for both the main memory and overflow scenarios.

As we will see, the addition of such an algorithm to a system does not obviate the need for an optimizer. In fact, it introduces a new and interesting optimization problem, as there are cases in which a pipelined tree of several smaller (fewer input stream) MJoin operators performs better than a single larger MJoin operator. Thus, ideally, an optimizer must decide the number of MJoin operators to use, and allocate input streams to these operators. However, for smaller joins (e.g., less than five input streams in our experiments) plans with a single MJoin operator are dominant.

2 Motivation

Consider the case in which a process receives data from input streams, wishing to group tuples from all the streams by joining them on some common attribute. This type of join query naturally arises in queries over stream data; for example, the join attribute could be *time* if we want to group sensor readings that occurred at the same time; it could be *traffic rate* if we are tracking highway congestion, or, it could be *pressure* or *temperature* if we want to track weather fronts;

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment

Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

and so forth. Using traditional database techniques, the execution plan would be organized as a sequence of binary join operators as shown in Figure 1. Each pair in the chain is joined by a binary join operator, which then feeds its output to a subsequent join operator, until all inputs are exhausted. If a hash join algorithm were chosen the system would build hash tables from the left and probe them from the right, while the optimiser would choose the inputs' order, according to their sizes and/or data distribution statistics.

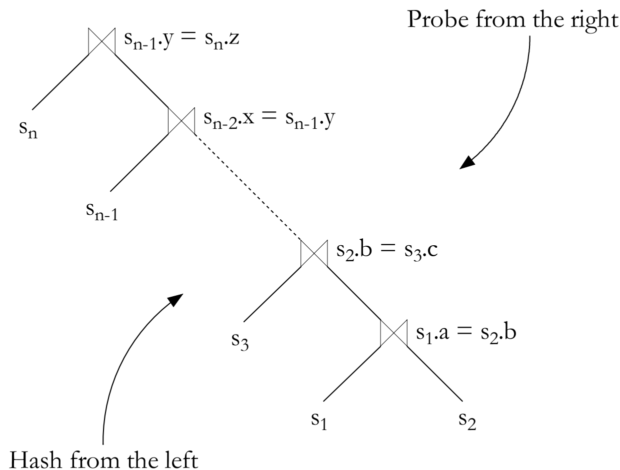

Figure 1: A traditional binary join execution tree

In a scenario where inputs are network streams, rather than disk-resident local files, however, the situation is different. In Figure 1, if any of the streams s_1 or s_3 through s_n is unbounded, the plan will never produce any results, because with standard blocking operators the build phase of the left inputs must complete before the probe phase of the right input starts. Symmetric binary operators, such as the symmetric hash join, address this problem, because they have the potential to produce an output whenever there is an arrival on either of their input streams.

Even with symmetric binary operators, problems may arise. Assuming that all inputs of Figure 1 are streams and 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 existing s_2 tuples. These $X_{1,2}$ tuples are propagated up-stream 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 them, inserting into one hash table and probing the other. That creates a large number of intermediate result tuples travelling through the system, causing substantial additional storage and communication overhead. This overhead can increase the system resources required per output tuple, which in turn can slow the effective rate of the output.

A tree of binary operators also introduces secondary, subtler effects. The issue is that if different input streams 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 optimiser 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 optimises 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, and the complexity of on-the-fly query plan restructuring becomes necessary.

To alleviate this problem, we propose the use of a multi-way symmetric join operator, which we call the MJoin, as depicted in Figure 2. In this scenario, the same joining s_1 arrival could be used to probe all s_2, …, s_n hash tables and propagate its contribution to the result, rendering unnecessary the need for intermediate storage and additional communication. By being able to generate results in a single step, this multi-way join maximizes the output *rate* of the plan in terms of outputs produced per unit time, and eliminates the difficult decision of which binary tree of symmetric operators to use. In the rest of this paper we will present such an operator, which we call *MJoin.*

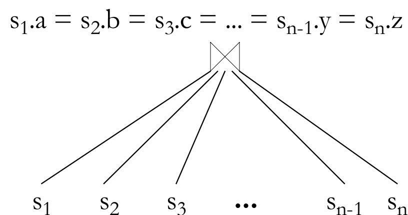

Figure 2: A multiple input join operator

The rest of the paper is organized as follows: Section 3 presents the related literature in the area, while Section 4 presents the basic functionality of the algorithm. Section 5 presents a cost model of the proposed operator, in terms of output rate and computational cost, while Section 6 deals with our experimental study of the algorithm. Finally, Section 7 presents our conclusions and identifies our future directions.

3 Related Work

Optimising for the first tuple of the result was the objective of the Britton-Lee optimiser [Sch83], while a similar notion of optimising for a specific subset of the result, namely the *Top/Bottom-N* results, was investigated in [CK97[1]] and [CK97[2]]. These ideas are similar to ours in the sense that they have as their goal generating a specific subset of the result as soon as possible. We differ in that this previous work did not consider the impact of streaming and unpredictable input data.

Join algorithms have been extensively studied in the context of relational database systems, with [Sha86] being the seminal paper that classified and evaluated hash-based join implementations. Join queries over distributed data have been studied in [ML86] and in parallel databases [DG+90]. The problem in distributed databases' context is to efficiently decide *where* to ship relations in order to perform the join, while, at the same time, perform a significant amount of work locally [Bab79], [BG+81],

[BC+81], [B70], [DG85]. Communication, synchronization and resource allocation is mainly the issue in parallel databases [DG+90], while [DN+92] presents an efficient way of dealing with data skew. Also related to our work are *hash teams* [GB+98] and *generalized hash teams* [KK+99], 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.

Our previous work on *rate-based optimisation* [VN02] addresses the issue of optimising queries over streaming information sources. The main idea of that paper is to switch from a cardinality-based cost model to a rate-based cost model since, in a streaming environment, the objective is to identify plans that, given the input rates of the streams, will produce the desired number of results the fastest. Unlike this paper, that paper considered only the optimisation problem in the context of previously proposed evaluation algorithms, and did not propose any new operators.

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* [WA91]. XJoin [UF00] 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 [IF+99] 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 multi-way join operators.

Streaming operation has been studied in the context of *query scrambling* [UFA98] and *adaptive query execution* [AH00], [IF+99]. In the former approach, an execution plan is monitored so that whenever a blocked input is detected the operator(s) using that input are pre-empted 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 overcome any performance bottlenecks. Our multi-way join operator addresses a similar problem, but without requiring any explicit monitoring or dynamic plan modification. Furthermore, as we show in Section 6.2, in some instances our new multi-way join operator makes more efficient use of resources than any tree of symmetric binary join operators. 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 minimizes the burden placed on an adaptive framework.

Finally, work has been done in the context of continuous queries over data streams. Two possible directions have been identified: the first aims at characterising the behaviour of these queries with respect to their memory requirements [AB+01], [BJ01]. The second aims at identifying and maintaining *stream statistics* for *sliding window* queries [DG+02]. Our work is orthogonal to these studies, since our algorithm is another place in which the same issues of memory management and statistics' maintenance arise.

4 Algorithm Description

The basic idea of the MJoin algorithm is very simple: generalize the symmetric binary hash join algorithm 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 appears 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.

The "predecessor" of our proposal, XJoin, addresses some of these issues, but in a smaller context since it only deals with two input streams at a time. Building on the principles of XJoin, we employ a similar three-step join strategy:

- As long as there are input arrivals, the algorithm performs in a *memory-to-memory* fashion, ensuring each input generates the largest possible partial join result it can.
- Once the inputs are blocked a *disk-to-memory* join operation initiates, joining portions of the algorithm's on-disk state to its current in-memory state.
- Once all inputs have seen their end, a last *disk-to-disk* operation takes over, generating the complete result.

In this paper, we focus on joins of the form discussed in Section 2 – that is, equijoins over an attribute common to all of the input streams. The memory-to-memory phase of our MJoin operator trivially extends to handle more general joins. However, the disk-to-memory and disk-to-disk phases are problematic if more than two input streams overflow to disk. (The problem is simultaneously partitioning multiple input streams in a consistent way when the overflowing relations have join conditions on multiple attributes.) While extending the MJoin operator to handle such cases provides interesting material for future work, we think that the MJoin operator as proposed in this paper is useful, because (a) as discussed in Section 2, joining multiple streams on a common attribute is a natural class of query, and (b) we expect many streaming joins in practice will be "window-joins" in which the window predicates ensure that the computation can be expected to remain memory resident.

In Section 4.2 we will present the three stages in more detail. Before doing so, however, we will focus on the key data structures, and how the algorithm employs them.

4.1 Data Structures

MJoin maintains a number of *hash tables* equal to its number of input streams. Each hash table is partitioned in an equal number of partitions, each having the ability to flush to disk whenever it becomes too large to fit in memory. Whenever a new tuple arrives, it is hashed into the corresponding stream's table in two steps: first, to identify which partition it belongs to and, second, into a slot of that particular partition. In addition to the in-memory state, there is a related on-disk state, referring to the portions of each hash table that have been flushed to disk due to memory overflow.

Figure 3 presents the operator's state, zooming in to one hash table. It shows the state when a newly arrived tuple is hashed into the corresponding hash table. A portion of each partition has already been flushed to disk.

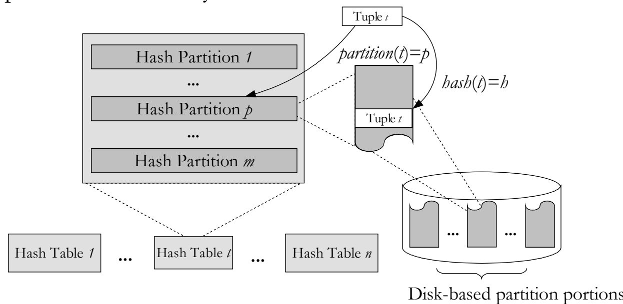

Figure 3: State of the MJoin operator

4.2 The Three Stages of MJoin

Like its predecessor the XJoin, the MJoin operates in three stages: an *in-memory* hash/probe operation that is active as long as the input streams have arriving tuples; whenever the inputs are blocked, a *disk-to-memory* thread takes over to match disk-resident data with in-memory portions of the input, while after all input has been read a *disk-to-disk* stage produces the final output.

4.2.1 In-Memory Operation

The first stage of MJoin is the one that deals with newly arrived tuples. Each new arrival triggers two operations:

1. A hashing of the new tuple into the corresponding stream's hash table.
2. A probing sequence of the other streams' hash tables for matches.

Not *all* hash tables will be probed by a single arrival, however. In a way resembling pipelined execution, a temporary result tuple is generated after each probe operation and goes on to probe the next hash table only if matches in the previous one exist[1]. Figure 4 depicts this sequence, where each probe operation is annotated with the probability of its taking place, which is equal to the previous in the sequence predicate's selectivity (the f_* factors in the Figure). For instance, for the second probe operation to execute, the first one has to produce matches.

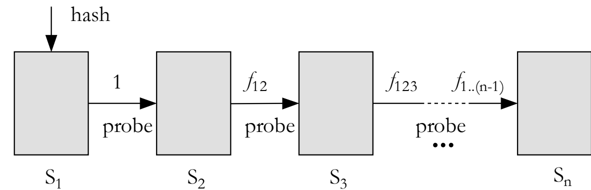

Figure 4: The probing sequence during MJoin.

Once all matches have been produced, and assuming all join predicates are satisfied, the Cartesian product between the new tuple and its matches is propagated to all subsequent operators. Figure 5 depicts the first stage's operation in which, for the sake of exposition and to avoid cluttering the figure, we present the Cartesian product's generation as a separate step.

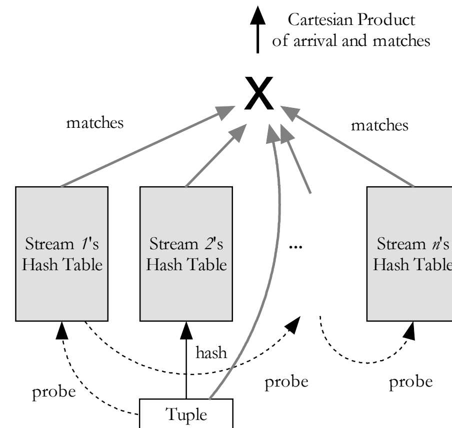

Figure 5: In-memory operation of MJoin

4.2.2 Disk-to-Memory Operation

Whenever MJoin's input blocks a *disk-to-memory* thread takes over, joining a disk-resident portion of one of the streams' hash table partitions with the in-memory portions of the other streams' hash tables. A salient problem in this stage is making sure no already propagated partial join results are re-created, which we will defer until Section 4.2.4. Figure 6 presents the second stage, again making the same simplifying assumption of Cartesian product generation that Figure 5 makes.

4.2.3 Disk-to-Disk Operation

Once an end-of-stream message from all inputs has been received, the last stage of MJoin, which joins all disk resident portions of the hash tables, takes over. This is achieved by performing the equivalent of the second phase of a multi-input hybrid hash join. Assuming there are *n* hash tables and *p* partitions in each hash table, memory is redistributed in *n-1* hash tables so that the smallest partition fits in memory (if it does not, multiple passes for that partition must be employed). A scan of the largest partition is then initiated, probing the hash tables, producing

[1] In fact, this entails the need for join ordering when optimising the query; each input needs to know the sequence in which it will probe the rest of the inputs' hash tables.

matches, detecting duplicates and outputting join tuples. This procedure is carried out for all *p* partitions. Figure 7 presents the redistribution of memory during the third stage and a sketch of the probing process.

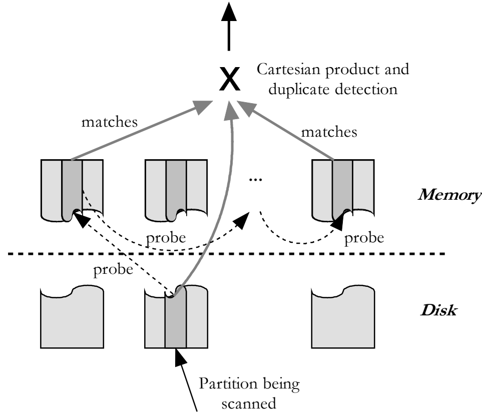

Figure 6: Disk-to-memory operation of MJoin

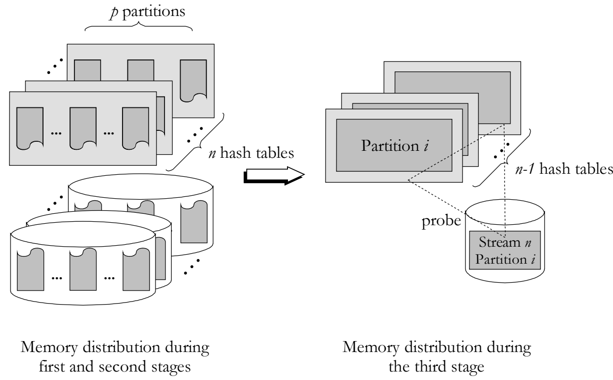

Figure 7: Disk-to-Disk memory redistribution and operation of MJoin

4.2.4 Duplicate Detection

MJoin uses a duplicate detection mechanism so that no duplicates are ever produced during the second and third stages. There are only two ways in which a tuple and *all* its matches render a result a duplicate:

1. If they were present in the memory-resident portions of the hash tables at the same time.
2. 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 breaches a condition, the partial join result is a new one and should be propagated.

The way MJoin eliminates duplicates is based on *time-stamps*. Each tuple is assigned two time-stamps: an arrival into the system and a departure from main memory. Additionally, a separate log is kept for each partition of each table, keeping track of when a partition was used for the second 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 then 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. T_i.*arrival*() > T_j.*arrival*() and T_i.*arrival*() < T_j.*departure*(), which means that T_i arrived while T_j was in memory.
2. *latest*(*partition*(T_i)) > T_j.*arrival*() and *probe*(*partition*(T_i)) > T_j.*arrival*() and *probe* (*partition*(T_i)) < T_j.*departure* () which entails that T_j has already been probed by a previous disk-to-memory join of T_i's partition.

This test 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., $\bowtie_{i=1\ldots n}(T_i)$) to be propagated, the test has to be passed by all possible *pairs* of constituent tuples. At first glance this may seem as an expensive operation, however, after careful consideration it becomes obvious that the number of checks is equal to the number used in a binary execution tree employing XJoin as the evaluation algorithm.

4.3 Implementation Abstractions

The way the algorithm has been implemented leaves a number of parameters to the programmer for definition and tuning. In particular, these abstractions are:

- *Partition picking policy during the second stage:* A number of possibilities exist here; one could pick the one with the most tuples, or the one with the smallest partition, or something in the middle.
- *Blocking strategy:* The input can be considered blocked whenever all inputs block, one of them blocks, or a given number of them blocks.
- *Aggressiveness of second stage:* A small timeout after which the input is considered blocked denotes an aggressive strategy. So does the ability to execute the second stage as an additional *thread* executing concurrently with the other threads, allowing the in-memory join process of the operator to handle any new arrivals, instead of waiting for a disk-to-memory stage to finish before the operator moves on to handle all (buffered) new arrivals.

5 Cost Expressions for MJoin

In this section we present a cost model for MJoin operators. Such a cost model is essential if optimisers 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 6.7. However, this section is not essential to the understanding of the bulk of this paper, and the reader who wishes to avoid getting bogged down in its details can safely skip it.

The purpose of this section is to extract specific cost expressions for MJoin in terms of a *rate-based* cost model. Our cost expressions will make use of the cost variables and notation in Table 1.

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 ith stream
f_k	Selectivity *f* of join predicate *k*

Table 1: Cost variables and notation used for modelling

5.1 Rate-based Cost Expressions

When dealing with streaming sources it is useful to consider a rate-based optimization framework (instead of a cardinality-based framework). In such a framework the decision basis is the predicted *output rate* of an operator when there is some prior knowledge of its input rates. The purpose of this section is to extract an output rate estimator for MJoin. Again, we will focus on the first stage of the algorithm, since this is the one in which MJoin exhibits streaming behavior.

The output rate of any process is the number of transmitted entities over the time needed to make the transmission [BG91], i.e.:

$$\textit{Output rate} = \frac{\textit{Number of ouputs transmitted}}{\textit{Time needed to make the transmission}} \qquad \textbf{Equation 1}$$

In our approach, we assume the *n* inputs have rates equal to r_1, r_2, …, 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 $T(1)=\prod_{j=1}^{k} f_j \cdot \prod_{i=1}^{n} r_i$. 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 $C(2)=\prod_{i=1}^{n} 2r_i = 2^n \prod_{i=1}^{n} r_i$ and the contribution of this input to the output will be $T(2)= \prod_{j=1}^{k} f_k \cdot 2^n \prod_{i=1}^{n} r_i$. From this size, however, we have to discard the inputs handled during the first second of execution since these have been already propagated. The contribution of the second second to the output becomes

$$T(2) = \prod_{j-1}^{k} f_k \cdot 2^n \prod_{i=1}^{n} r_i - T(1) = \ldots = \prod_{j-1}^{k} f_k \cdot \prod_{i=1}^{n} r_i \cdot \left(2^n - 1\right)$$

By induction, we can prove that the number of transmitted outputs for any time point *t* will be given by the following expression:

$$T(t) = \prod_{j=1}^{k} f_k \cdot \prod_{i=1}^{n} r_i \cdot \left(t^n - (t-1)^n - \ldots - 2^n - 1\right) = \prod_{j=1}^{k} f_k \cdot \prod_{i=1}^{n} r_i \cdot \left(t^n - \sum_{t=1}^{n-1} t^n\right)$$

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 Figure 4 depicts

In total, the cost per arrival will be equal to *hash* + *move* + *comp*$\cdot(1+\prod_{j=1}^{k-1} f_k)$, where $\prod_{j=1}^{k-1} f_k$ is the cost induced if *all* probes have to be performed[2]. 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*$\cdot(1+\prod_{j=1}^{k-1} f_k)$)). Substituting this last expression and *T*(*t*) into Equation 1 yields MJoin's output rate (Equation 2), which, as was the case in [VN02], is time-dependent. In Section 6.7 we will see how Equation 2's output prediction rate can in fact identify cases where MJoin's performance might degrade.

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

6 Experiments

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

6.1 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. The queries we used were variants of the Wisconsin Benchmark's [BT88] *JoinABPrime* query, extended to handle multiple sources. All joins were performed on the `unique1` attribute of the relations ensuring the size of the result set was equal to the size of the smallest participating relation.

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

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
```

IBM's *jikes* compiler was used for byte-code generation, which was executed using SUN's *HotSpot* virtual machine. All presented 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 [BG91] with the given arrival rate as its mean. As a rule, we used the slowest streams inter-arrival rate as the operator's blocking threshold[3].

6.2 In-Memory Performance

The first set of experiments we performed deals with MJoin's performance when data fit entirely in memory. These experiments highlight the fact that MJoin performs fewer CPU-bound operations (hashes, moves, comparisons) than a pipelined XJoin plan over the same inputs. We used a simple three-way join query that we modeled in five possible ways. Figure 8 depicts the plans while Table 2 presents the streams' parameters. In the first plan, all sources have the same arrival rate; in the top-right plan one of the sources is rendered considerably faster than the rest and it is put at the top of the execution plan, while in the bottom-left plan the fast source is kept at the bottom of the plan. For comparison, and regardless whether streams have the same or different input rates, we employ the same MJoin plan, the one on the bottom-right of Figure 8.

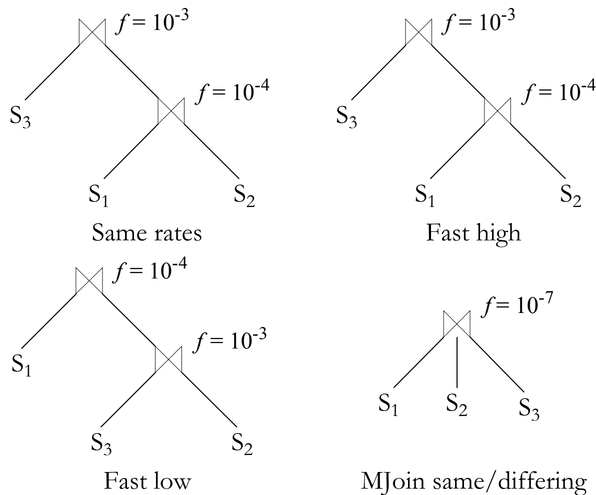

Figure 8: Plans used for in-memory experiments

We executed the five plans, keeping track of when each tuple appeared in the output, which is essentially an indication of a plan's output rate. The results are plotted in Figure 9 and Figure 10. To verify our claim that MJoin's superiority stems from performing fewer operations, we instrumented the code to count each operation as it took place. The results were in accordance with our intuition and the algorithm's cost model and are presented in Table 3. We see that although in total MJoin performs more comparisons (a side-effect from its symmetric nature) it also performs 10,000 fewer *hash* and *move* operations.

Stream	*Size (tuples)*	*Inter-arrival delay (msec)*
S1	10,000	20 (5 for same rates)
S2	15,000	20 (5 for same rates)
S3	10,000	5

Table 2: Parameters for in-memory experiments

Plan shape	*Hashes*	*Moves*	*Comparisons*
Deep	45,000	45,000	45,000
MJoin	35,000	35,000	50,189

Table 3: Number of operations during in-memory experiments

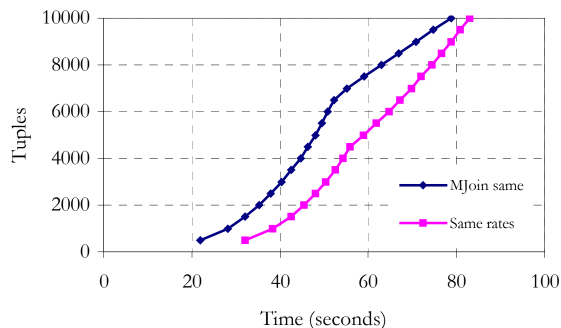

Figure 9: In-memory performance when input rates are the same

6.3 Scaling the Input Size

To investigate MJoin's performance as input sizes grow, we scaled the input sizes some (but not considerably – for that see Section 6.5) and allocated a memory buffer less than the inputs' sizes so that parts of the hash-tables would be flushed. Table 4 shows the parameters we used, while Figure 11 shows the results. Again, MJoin was significantly better than the other two plans, between which, the one keeping the fastest stream at the top of the execution plan exhibits a better performance in terms of output production rate. Seeing that placement of a fast (or slow) stream has a significant impact on plan performance, we decided to experiment with streams of varying rates, i.e., not having a steady mean arrival rate.

[3] A Poisson arrival process entails that the inter-arrival process follows an exponential distribution with a mean equal to the reverse of the Poisson process's mean.

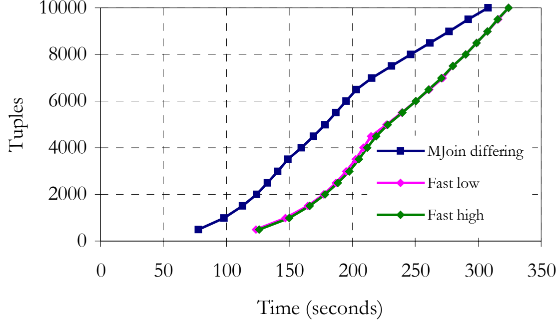

Figure 10: In-memory performance for varying input rates

Stream	*Size (tuples)*	*Inter-arrival delay (msec)*
S1	100,000	10
S2	100,000	5
S3	200,000	1

Table 4: Parameters for scaled input sizes

6.4 Resilience to Fluctuations

In the experiments of this section, we used the same plans as in the previous section, but 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. Figure 12 presents the results.

As expected from the findings of the previous section, 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 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.

6.5 Window Joins

When dealing with join evaluation over streaming sources, 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 joins. In such a scenario, the inputs' hash tables are invalidated whenever the window expires. 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-based predicates over the query, with each predicate having a horizon of ten thousand tuples, i.e., the inputs' hash tables were to be invalidated whenever ten thousand tuples from a stream were read. The three plans we used were similar to the *Fast High*, *Fast Low* and *MJoin* plans of Figure 8 while the inputs' parameters are presented in Table 5. Figure 13 depicts the experimental results.

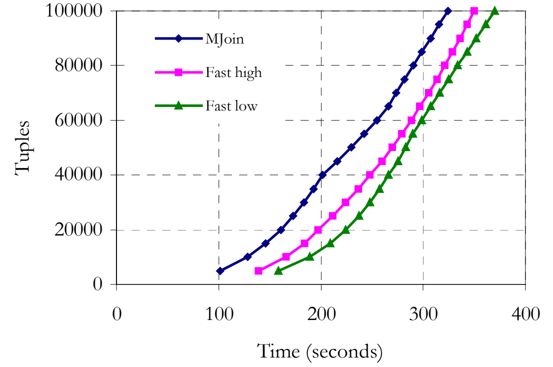

Figure 11: Scaled input size performance

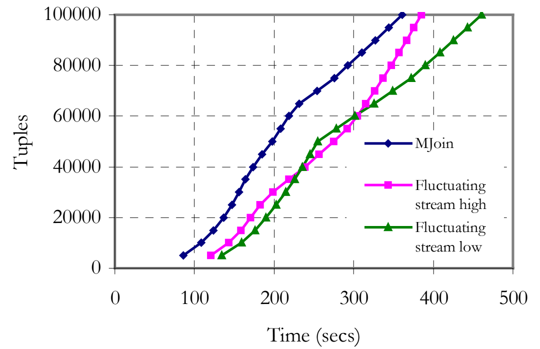

Figure 12: Performance for fluctuating input rate

As in all previous experiments, in the case of window-based predicates the MJoin plan exhibits better performance. 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 optimised for in-memory, streaming behaviour.

Stream	*Size (tuples)*	*Inter-arrival delay (msec)*
S1	1,000,000	3
S2	1,000,000	3
S3	1,000,000	1

Table 5: Stream parameters for window joins

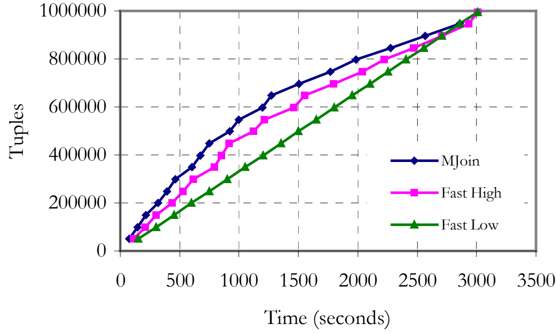

Figure 13: Window join performance

6.6 Scaling the Number of Joins

The next set of experiments deals with MJoin's ability to scale with respect to the number of joins in the execution plan. To test this, we generated five way join queries and we organized their plans using (i) a deep pipelined plan, (ii) a bushy pipelined plan (both of the above employing binary XJoin operators), or (iii) a plan having all join operations handled by a single MJoin operator. Figure 14 presents the plans and the sources' input parameters, while Figure 15 depicts the experimental results.

Again, MJoin performed better in terms of output rate and was able to generate the final result sooner than the other two plans. The point of interest in this experiment, however, is that as time progressed MJoin's performance advantage over the other plans degraded, until by the end the bushy plan had almost caught up. The next section explains this phenomenon, which indicates that MJoin is not always the algorithm of choice, and suggests that an optimiser is needed to determine when the MJoin should be used.

6.7 On the Need for Optimisation

The final set of experiments we conducted has to do with investigating and proving that even with an operator like MJoin, the need for optimisation of join trees still exists. What triggered this part of our research, were the experimental results of Section 6.6, where we saw MJoin's performance starting to degrade. Suspecting that the problem was MJoin's inherent complexity on a per-input basis (as the cost model of Section 5.1 shows) we added an additional input and one more join predicate to the query, thus having to perform a six-way join. Again, we generated three arbitrary plans, one organized as a pipelined plan, one organized as a bushy plan, and finally a single-operator six-way plan using MJoin. Figure 16 presents the two non-MJoin plans, where each input is annotated with its size in tuples and its inter-arrival delay. Execution of these plans, along with the single MJoin plan, yields the performance observed in Figure 17. Though MJoin behaves better in the initial execution stages, the bushy plan overtakes it as time goes by.

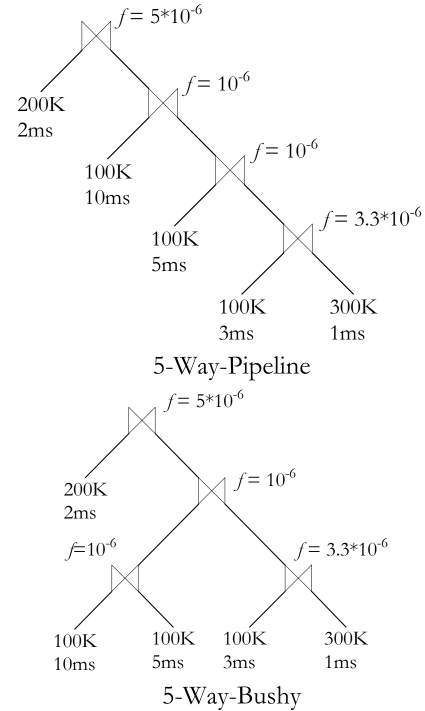

Figure 14: Plans used when scaling the number of joins

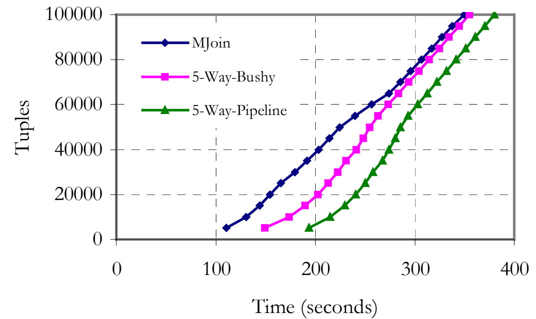

Figure 15: Experimental results after scaling the number of joins

Our initial explanation of MJoin's performance degradation had to do with its per-input cost, as this is modelled in Section 5.1's cost model. 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. Table 6 presents these measurements.

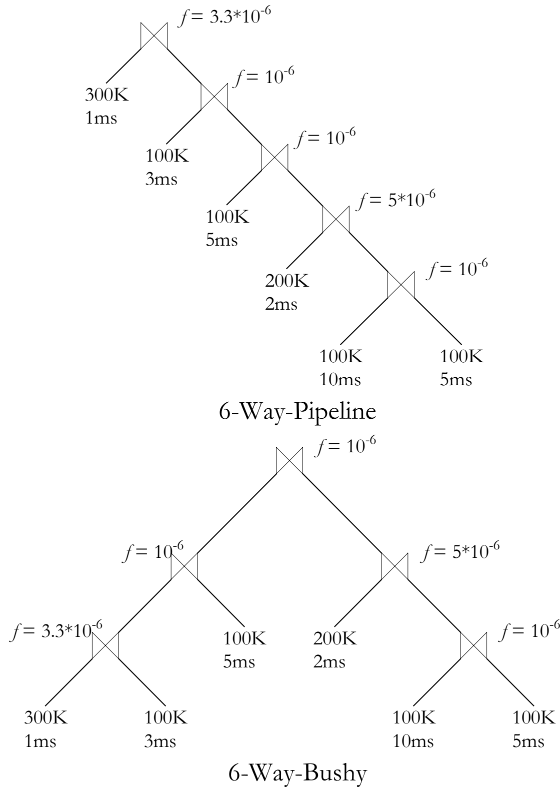

Figure 16: Six-way join execution plans

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 inFigure 16, it is easy to see that this time is greater than the fastest stream's inter-arrival rate (10^{-3} seconds for the three-hundred tuple stream). This translates into a backlog of tuples being created for that particular stream; as far as the 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 bushy plan, which uses multiple binary operators hence is not sensitive to the total number of input streams in the join.

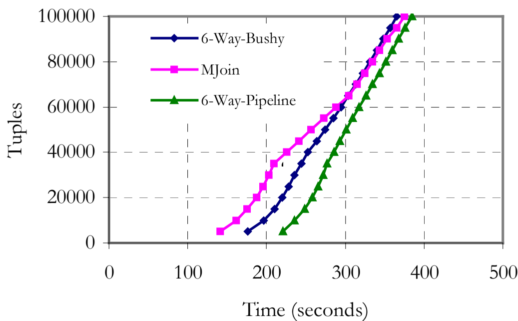

Figure 17: Six-way join performance

Operation	Average cost (clock ticks)	Average cost (seconds)[4]
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 6: Cost of various operations as measured by the processor's hardware counters

Since our earlier experiments that MJoin is superior for smaller numbers of joins, we experimented with plans that use two smaller MJoin operators instead of one large one. Figure 18 shows the plans we tested, while Figure 19 presents the measured performance of these plans, along with that of the bushy plan and the single MJoin plan.

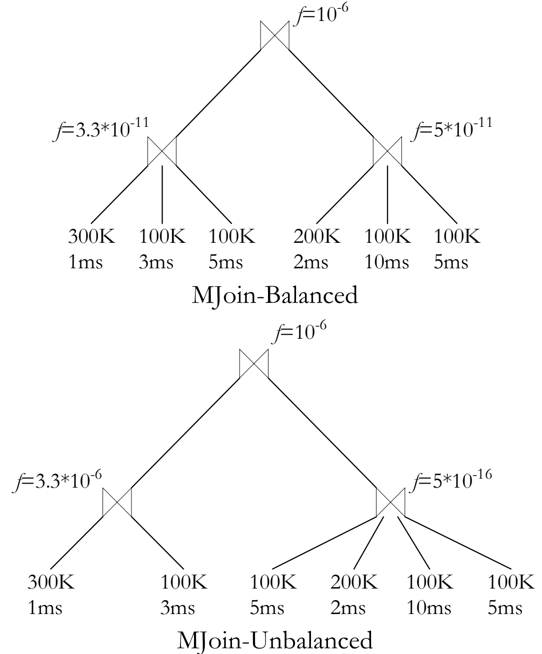

Figure 18: The two extra six-way join plans

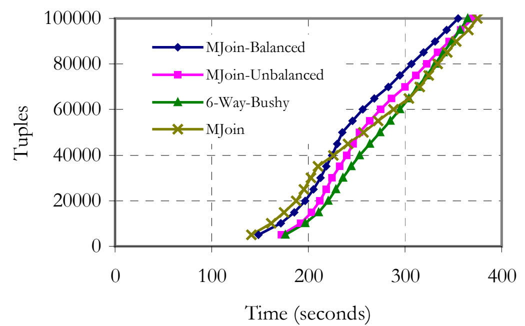

Figure 19: Revised six-way join plan performance

Figure 19 shows that the plans with multiple small MJoin operators can outperform both the single large

[4] We were using a 1GHz processor; one clock tick is equal to 10^{-9} seconds.

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 optimisation: ideally, the optimiser needs to know how many result tuples it should optimise 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 Section 5 can assist the optimiser in this task.

7 Conclusions and Future Work

In the previous sections we proposed and evaluated a new join algorithm that addresses issues that arise with multi-way join queries over streaming inputs. In particular, our claim is that by using a multi-input symmetric join operator instead of a pipelined execution plan of symmetric binary operators, we obtain better performance. To support our claim, we validated our intuition by conducting a series of experiments over multi-way join queries using the Wisconsin Benchmark data set. In future work we plan to extend the MJoin operator to handle more classes of join queries and investigate the optimisation problems it introduces.

Bibliography – References

[AB+01] A. Arasu, B. Babcock, S. Babu, J. McAlister and J. Widom, *Characterizing Memory Requirements for Queries over Continuous Data Streams*, Stanford Technical Report, November 2001, http://dbpubs.stanford.edu/pub/2001-49.

[AH00] R. Avnur and J. M. Hellerstein, *Eddies: Continuously Adaptive Query Processing*, Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas, USA, May 2000, pp. 261-272.

[B70] B. H. Bloom, *Space/Time Trade-offs in Hash Coding with Allowable Errors*, CACM, 1970, (13) 7:422-426.

[BT88] Bitton, D. and C. Turbyfill, *A Retrospective on the Wisconsin Benchmark*, in Readings in Database Systems, edited by Michael Stonebraker, Morgan Kaufman, 1988.

[Bab79] E. Babb, *Implementing a Relational Database by Means of Specialized Hardware*, TODS, 1979, (4) 1:1-29.

[BC+81] P. A. Bernstein and D.-M. W. Chiu, *Using Semi-Joins to Solve Relational Queries*, JACM, 1981, (28) 1:25-40.

[BG+81] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve and J. B. Rothnie Jr., *Query Processing in a System for Distributed Databases (SDD-1)*, TODS, 1981, (6) 4:602-625.

[BG91] D. Bertsekas and R. Gallager. *Data Networks*, Prentice Hall, 2nd edition, 1991.

[BJ01] S. Babu, and J. Widom, *Continuous Queries over Data Streams*, SIGMOD Record, Sept. 2001.

[CD+00] J. Chen, D. J. DeWitt, F. Tian and Y. Wang. *Niagara-CQ: A Scalable Continuous Query System for Internet Databases*, Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas, USA, May 2000, pp. 379-390.

[CK97[1]] M. J. Carey and D. Kossmann, *Processing Top N and Bottom N Queries*, Data Engineering Bulletin, 1997, (20) 3:12-19.

[CK97[2]] M. J. Carey and D. Kossmann, *On Saying "Enough Already!" in SQL*, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, pp. 219-230.

[DG+02] M. Datar, A. Gionis, P. Indyk and R. Motwani, *Maintaining Stream Statistics over Sliding Windows*, 2002 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002).

[DG+90] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H-I Hsiao and R. Rasmussen. *The Gamma Database Machine Project.* TKDE, 1990, (2) 1:44-62.

[DG85] D. J. DeWitt and R. H. Gerber, *Multiprocessor Hash-Based Join Algorithms*. Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden, pp 151-164.

[DN+92] D. J. DeWitt, J. F. Naughton, D. A. Schneider and S. Seshadri. *Practical Skew Handling in Parallel Joins*, 18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, pp 27-40.

[GB+98] G. Graefe, R. Bunker and S. Cooper, *Hash Joins and Hash Teams in Microsoft SQL Server*, Proceedings of the 24th VLDB Conference, New York, USA, 1998, pp. 86-97

[IF+99] Z. G. Ives, D. Florescu, M. Friedman, A. Levy and D. S. Weld. *An Adaptive Query Execution System for Data Integration*, Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, Philadelphia, Pennsylvania, June 1999.

[KK+99] A. Kemper, D. Kossmann and C. Wiesner, *Generalized Hash Teams or Join and Group-by*, Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

[ML86] L. F. Mackert and G. M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries, VLDB'86 Proceedings of the 12th International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, pp 149-159.

[Sch83] G. Schumacher. *GEI's Experience with Britton-Lee's IDM*, IWDM, 1983, pp. 233-241.

[Sha86] L. D. Shapiro. *Join Processing in Database Systems with Large Main Memories*, TODS, 1986, (11) 3:239-264.

[UF00] T. Urhan and M. J. Franklin. *XJoin: A Reactively-Scheduled Pipelined Join Operator*, IEEE Data Engineering Bulletin, June 2000, (23) 2:27-33.

[UFA98] T. Urhan, M. J. Franklin and L. Amsaleg. *Cost Based Query Scrambling for Initial Delays*, Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, Seattle, Washington, USA, June 1998, pp. 130-141.

[VN02] S. Viglas and J. F. Naughton. *Rate-based Optimization for Streaming Information Sources*, to appear in SIGMOD 2002.

[WA91] A. N. Wilschut and P. M. G. Apers. *Pipelining in Query Execution*, Conference on Databases, Parallel Architectures and their Applications, Miami, 1991.