Driver input selection for main-memory multi-way joins

Emmanouil Valsomatzis[*]
Department of Computer Science
Aalborg University, Denmark
evalsoma@cs.aau.dk

Anastasios Gounaris
Dept. of Informatics
Aristotle University of Thessaloniki, Greece
gounaria@csd.auth.gr

ABSTRACT

Stream query processing has a particularly broad range of applications from sensor data processing and internet traffic analysis to runtime monitoring of stock market and server logs, and scientific simulations. This work focuses on multi-way join queries over streamed data, which are processed with the help of a n-ary join. More specifically, we propose a novel main-memory variant of the influential *MJoin* operator proposed in [35], which processes input data in batches with a view to improving the CPU efficiency, and explicitly controls the order of execution within each batch without being restricted by the time of input arrival, as current state-of-the-art solutions do. To this end, we also propose policies for selecting the execution order, and we show that our approach can yield important performance benefits.

1. INTRODUCTION

In the last decade, the database research community has focused its attention on query processing over continuous and possibly unbounded input streams rather than on stored data sets. A data stream is a real-time, continuous, ordered (either explicitly by timestamp or implicitly by arrival time) sequence of items. Stream query processing has a particularly broad range of applications from sensor data processing and internet traffic analysis to runtime monitoring of stock market and server logs, and scientific simulations (e.g., [1, 6, 7, 9, 12, 13, 25, 30, 33, 38]). Example queries include moving averages of recent stock prices and finding correlations between the prices of several stocks [14].

In general, querying data streams involves running a query over a period of time, which can be of infinite or finite length, and generating new answers as new items arrive. These types of queries are known in the literature as continuous, standing, or persistent queries. They also share a common requirement not to employ blocking operators, i.e., operators that must consume the entire input before any results are

produced, thus resulting in fully pipelined query plans that comprise non-blocking operator implementations (e.g., [15, 21]).

This work focuses on multi-way join queries over streamed data. Several applications which involve decision support, data intensive solutions and advanced data management over complex objects, such as graphics, artificial intelligence, and geometric modeling, usually have to specify their desired results in terms of multi-way join queries [31]. All such applications are also encountered in streaming scenarios as well. For example, in network packet monitoring, the network administrator may want to monitor the traffic of data packets passing though different routers with the objective of finding packets with the same destination IP address; this task can be formed as a multi-way join in a straightforward manner [4]. As another example, in building monitoring using sensor networks, one may want to keep track of the temperature, humidity, and light intensity measured by sensors in a room. The sensor readings of each measurement type are sent to their respective sinks as a stream. The monitoring task in each room can be specified as a distributed stream join query that joins on the same room id from three sensor-reading streams [33]. In [2, 3], multi-way join queries are used to detect and track the propagation of various phenomena, that strike a sensor field. Other applications of multi-way joins in sensor networks are object tracking, surveillance, and environmental monitoring [17, 32, 29].

That kind of queries are often characterized by unpredictable data arrival rates and stand to benefit from advanced join algorithms that are capable of tolerating initial delays, volatile arrival order and bursty arrival rates. We can classify the corresponding query plans in two broad categories: (i) traditional left (or right) deep plans that comprise a series of binary pipelined joins; and (ii) plans in which all the joins are encapsulated in a single n-ary join operator. The most prominent representative of n-ary joins is *MJoin*, which was initially proposed in [35] and is a generalization of symmetric binary join algorithms.

The *MJoin* algorithm first creates a hash table on each join attribute of every input stream. When a new tuple arrives from any input, then this input plays the role of the *driver* relation: the new tuple is inserted into the corresponding hash table and then is used to probe the remaining hash tables of the other relations (also termed as *driven* relations) in order to produce results. There are several options for choosing the best probing sequence, which may also account for evolving selectivities (e.g., [11]). In their traditional implementation, *MJoins* process input tuples for any source as

---

[*]Research conducted at the Aristotle University of Thessaloniki.

they arrive, i.e., they do not control the order of processing of input tuples, although different tuples may follow different execution paths.

A common problem stemming from the fact that arrival rates are volatile and potential bursty is that the execution of *MJoins* may lead to significantly large idle times when all sources are temporarily blocked; this problem is evident even in variants that employ a reactive phase, where data that have previously been flushed to disk are joined during the periods where no inputs are available [5, 8, 34]. In this work, we deal with the problem of such idle times in settings where all data can fit into main memory, which is rather common in window joins. Our main rationale is to group tuples in batches and activate the join algorithm periodically. The benefits of such an approach are twofold. Firstly, it improves the CPU efficiency, since the CPU is fully utilized when activated and totally idle otherwise; a similar approach has been shown to improve energy efficiency in data centers [22]. Secondly, since input tuples are stored in incoming buffers, the algorithm is not forced to process them in strict timestamp order to ensure correctness. For example, a more recent tuple may be processed before an earlier tuple from a different input stream without sacrificing result correctness; in this work we show that such an approach leads to improved performance in terms of running time and capability of producing early results and of consuming the input as fast as possible.

In summary, the contribution of this paper is as follows. We propose a variant of the main-memory *MJoin* algorithm, which (i) is activated periodically and processes tuples in batches, and (ii) does not process tuples in strict timestamp order in each batch. Further, we propose specific policies for specifying the order of execution, which result in adaptive variants for *MJoin* that aim to optimize the result production rate and the input consumption rate within each batch through dynamic choices of the driver input. Our thorough experiments prove the proposal's efficiency. For bursty arrival rates and stream inputs with different selectivities, we observed reductions in the running time up to 50%. Note that although we consider finite streams, so that they can fit into the main memory, our technique can be easily applied to scenarios that include windowed multi-way joins over infinite streams, where the aggregate size of windows can fit into the main memory. Furthermore, our proposal can be applied to any case where a multi-way join over streaming inputs is employed under the condition that the rate of data arrival is lower than the processing rate of the join, as typically happens, and the user can tolerate the small delays due to batching and the fact that the processor is not continuously running. In addition, our proposal is applicable to scenarios where a single processor *has to* be shared across multiple multi-way join applications, e.g., in algorithmic trading.

The remainder of the paper is structured as follows. The next section discusses the related work. In Section 3 we explain our approach and we present adaptive policies for choosing the driver input. Section 4 deals with the evaluation, and we conclude in Section 5.

2. RELATED WORK

Non-blocking join algorithms are useful in a wide range of web-based applications, such as data integration [19], online aggregation [16, 18], provision of approximate answers [27, 20, 36], spatial databases [26], and adaptive query processing
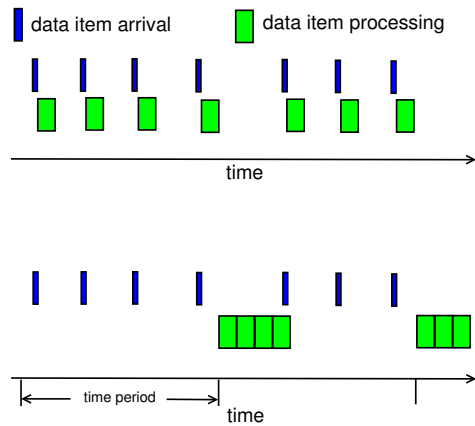

Figure 1: The traditional (top) and our novel (bottom) high level approach to processing tuples in main-memory MJoins.

[11]. Most of the research effort in this area has been put on disk-based variants that have the ability to produce join results even if one or both sources are blocked, and to hide idle times by employing a reactive phase, where data that have been previously flushed to disk are joined during the periods where no inputs are available [5, 8, 23, 34]. All these proposals are orthogonal to ours, which refers to main-memory execution only.

The capability to control and adapt the driver tables in multi-way query plans can be exploited for a variety of purposes as to react to stalls, to an unexpectedly large driver table, to changing user requirements and to switch drivers based on improved selectivity estimation [11]. Choosing the driver inputs has been considered an important topic in adaptive query processing (e.g., [24]), but to date, has not been examined in the context of main-memory *MJoins*, which process tuples simply in the order they arrive [35].

Reordering the tuples has been also proposed in [28] to permute the data items at the source so to allow aggregate queries to report back more interesting results earlier. In our techniques, we essentially reorder tuples as well but for a different purpose, namely to improve performance. Finally, processing and taking decisions at a batch level is not new in adaptive query processing. For example, in [10], the adaptive decisions are revised after a batch of tuples has been processed so that the cost of making routing decisions can be effectively amortized.

3. ADAPTIVE CHOICE OF THE DRIVER INPUT IN MJOINS

3.1 High-level approach

In this work, we develop a novel flavor of main-memory *MJoins* [35] that is capable of temporarily storing input tuples in limited-sized incoming buffers and activate the join processing phase periodically. In this way, the processing takes place in batches. More specifically, we consider a centralized multi-way join over M remote finite data streams denoted as $S_1, S_2, ..., S_M$. The data, upon their arrival, are concentrated in a unit that consists of M temporary storage buffers in timestamp order (arrival times). The incoming tuples are stored in those buffers until the join process kicks

off. We assume that the length (in time units) of each time period is fixed. According to this approach, during each period, the join process is mostly idle while tuples from any input arrive. When the join process is activated, at the end of each period, the CPU is fully utilized, and processes all the tuples that have arrived up to the activation time point; the corresponding dataset is referred to as a data batch. This can lead to better CPU efficiency and thus energy savings. Moreover, at this point, data from each data source may have become available, thus the algorithm gains control on the order of execution of the inputs because it is not restricted to process the buffered tuples in strict timestamp order. The high-level approach is depicted in Figure 1, where the traditional and the proposed processing in *MJoins* are shown at the top and the bottom part, respectively.

After a tuple from a specific input has been selected for processing, we assume that the probing sequence is defined using known techniques, such as those described in [11], since it is important for the probing sequence to be declared in such a way that the most selective predicate is evaluated first. In that case, the smallest number of temporary results is generated. With the probing sequence having been defined for each input, the main choice to be made by the algorithm is the selection of the appropriate input to serve as the driver one. To this end, we focus on the selection of the driver input, testing different policies and checking their impact on the rate of consuming the incoming tuples and producing early results. This implies that, within each batch, tuples may not be processed in timestamp order. For example, a tuple with a later timestamp may be processed earlier than a more recent tuple from another input.

Our policies are inherently adaptive in that the resulting processing plans are continuously revised across different batches to reflect changes in arrival rate characteristics. Moreover, the probing sequence techniques in [11] can also account for evolving selectivities as well. In this work, we assume that the selectivities are known throughout the multiway join execution (either through offline statistics or online monitoring). For presentation simplicity reasons, we will assume that there is a single join attribute per input. The notation used for describing our policies is summarized in Table 1.

3.2 Policies for Driver Selection

In this section, we elaborate on the policies for driver selection. These policies are applied during the processing of each batch separately. The first one, termed as *Timestamp-based*, is the one employed in the original proposal of *MJoins* in [35], whereas the next three ones aim at optimizing aspects such as the result output rate and the input consumption rate. We assume that, when an input tuple is selected, then it completes its processing (i.e., probes all hash tables until it is dropped or produced in the output) before the next tuple is processed.

Timestamp-based: this policy simply processes tuples in timestamp order and is presented in Figure 2. This means that the driver input stream may vary arbitrarily during the batch processing based on the relative order of the tuple arrival for each input stream. No effort is made to optimize any property. In Fig. 2, the method $process\_next\_tuple(buffer_j^i)$ denotes the processing of the tuple with the smallest timestamp from S_j in the i-th batch We also employ another method called $input\_stream\_ts()$ that returns the input iden-

S_j	the j-th finite data stream, j $\in [1, M]$
$|S_j|$	the size of S_j
$|Mjoin|$	the size of the complete result set produced by the Mjoin operator
σ_{S_j}	selectivity of stream $S_j = \frac{|Mjoin|}{|S_j|}$
$\sigma_{j \bowtie l}$	selectivity of the join operator $S_j \bowtie S_l$, which is equal to the ratio of the size of $S_j \bowtie S_l$ to the size of their cartesian product, $j, l \in [1, M]$
σ_{Mjoin}	the ratio of the size of the complete join to the product of the sizes of all streams, $\frac{|Mjoin|}{|S_1| \times |S_2| \times \ldots \times |S_j|}$
K	the total number of batches, $\lceil \frac{last\_timestamp - first\_timestamp}{time\_period} \rceil$
$batch_i$	the set of all tuples in the i-th batch, $i \in [1, K]$
$buffer_j^i$	the set of tuples from stream S_j in $batch_i$
ht_j	the hash table built for S_j
$outBuffer_j^i$	the number of tuples produced when $buffer_j^i$ is processed
$outRate_j^i$	the rate of producing output tuples output when $buffer_j^i$ is processed, $\frac{outBuffer_j^i}{|buffer_j^i|}$

Table 1: Definitions of terms used in techniques

```
1: for i = 1 to K do
2:    while batch_i NOT EMPTY do
3:        j ← input_stream_ts()
4:        process_next_tuple (buffer_j^i)
5:    end while
6: end for
```

Figure 2: The Timestamp-based policy.

tifier of the stream, the buffer of which contains the tuple with the smallest timestamp.

Consumption rate: the rational of this policy is to maximize the consumption rate in the sense that it aims to reduce the size of each buffer in the batch that is being processed as fast as possible. This is performed with a view to freeing up memory earlier in the execution. To achieve this, we select the driver inputs according to their selectivity. More specifically, as the selectivities of each join considered to be known, we sort all the plans in an ascending order of selectivities σ_{S_j}. Then, we first choose to process all the tuples from the most selective input, then from the second most selective, and so on. Apart from the different criterion used, a main difference compared to the timestamp-based policy is that the complete buffer of an input stream is processed before the first tuple of another buffer is processed, i.e., the processing of buffers is not interleaved. The pseudocode for this policy is in Figure 3.

Initial output size: our third decision policy aims at maximizing the production of early results. Given the selectivities for each join, we can estimate the size of the result size for each input tuple by multiplying the selectivity with the size of the relevant hash table that the input tuple probes. Given also that we know the number of tuples from each buffer in each batch, we can estimate the total number of output tuples that are expected to be produced for each buffer. An estimate of $outBuffer_j^i$ is provided with the help

```
1: for i = 1 to K do
2:    while batch_i NOT EMPTY do
3:       order streams by ascending order of σ_{S_j}
4:       j ← next stream in the ordered list
5:       while buffer_j^i NOT EMPTY do
6:          process_next_tuple (buffer_j^i)
7:       end while
8:    end while
9: end for
```

Figure 3: The Consumption rate policy.

```
1: for i = 1 to K do
2:    while batch_i NOT EMPTY do
3:       order streams by descending order of outBuffer_j^i
4:       j ← next stream in the ordered list
5:       while buffer_j^i NOT EMPTY do
6:          process_next_tuple (buffer_j^i)
7:       end while
8:    end while
9: end for
```

Figure 4: The Initial output size policy.

of the following equation:

$$outBuffer_j^i = |buffer_j^i| * \sigma_{Mjoin} * \prod_{l=1}^{M-\{j\}} |ht_l| \qquad (1)$$

Thus we employ a greedy algorithm that, in each step, selects the input stream with the highest value of $outBuffer_j^i$, as shown in Figure 4.

Output rate: Our final policy extends the previous one by taking into account also the number of incoming tuples required to produced the output dataset. In this way, this policy focuses on the output production rate, whereas the previous one focuses on the output size. The difference is in Line 3 of the algorithm in Figure 4, where the ordering is by $outRate_j^i$ instead of $outBuffer_j^i$; the $outRate_j^i$ is defined in Table 1.

4. EVALUATION

4.1 Experimental setup

We examine a 3-way join over a common attribute of three different artificial finite data streams, namely $S1$, $S2$ and $S3$. We use six different datasets, which differ in the input stream arrival patterns. Although the streams in all six datasets do not have the same arrival rate and may also be of different size, we ensure that they finish approximately simultaneously.

The combination of the different selectivities and the arrival rate patterns aim to cover a broad range of illustrative scenarios. In all our experiments each time unit corresponds to a nanosecond. In the first dataset, the data arrival rates are as follows: $S1$ and $S2$ tuples arrive at a uniform average rate of 1 tuple per 10 time units. However, the rate of $S3$ varies during the execution of the query. The arrival rate changes from 1 tuple per 5 time units to 1 per 15 time units approximately each quarter of the input. In the second dataset, the tuples of $S1$ arrive at a uniform average rate of 1 tuple per 10 time units, and the arrival rates of both $S2$

	Dataset I - IV	Dataset V-VI
$\sigma_{S1 \bowtie S2}$	$5 * 10^{-9}$	10^{-5}
$\sigma_{S1 \bowtie S3}$	10^{-6}	10^{-5}
$\sigma_{S2 \bowtie S3}$	$5 * 10^{-6}$	10^{-5}
$\sigma_{(S1 \bowtie S2) \bowtie S3}$	10^{-3}	10^{-4}
$\sigma_{(S1 \bowtie S3) \bowtie S2}$	$5 * 10^{-6}$	10^{-4}
$\sigma_{(S2 \bowtie S3) \bowtie S1}$	10^{-6}	10^{-4}
σ_{S_1}	$5 * 10^{-12}$	10^2
σ_{S_2}	$5 * 10^{-12}$	10
σ_{S_3}	$5 * 10^{-12}$	1

Table 2: Selectivities for the datasets used in the experiments.

Stream	Dataset I - IV	Dataset V-VI
S1	10^6	10^4
S2	10^6	10^5
S3	10^6	10^6

Table 3: Streams' cardinalities.

and $S3$ vary during the execution of the query. More specifically, the arrival rate of $S2$ alternates between 1 tuple per 15 times units to 1 per 5 times units approximately each quarter of the input. The arrival rate of $S3$ varies in an inverse manner: it changes from 1 tuple per 5 time units to 1 per 15 time units every almost one fourth of the execution. In the third dataset, we follow the same fluctuation for streams $S1$ and $S2$ as in the second dataset. However the arrival rate of stream $S3$ follows a sinusoidal distribution.

The next three datasets employ a b-model of 20%-80% distribution introduced in [37] and used to simulate bursty network behaviour. In the fourth dataset, the data arrival rates are as follows: $S1$ and $S2$ tuples arrive at a uniform average rate of 1 tuple per 10 time units, whereas $S3$ follows a 20%-80% b-model distribution. Datasets V and VI follow a similar data arrival pattern (the b-model distribution in Dataset V is 80%-20% though) but the selectivities of the streams vary, as explained below. The arrival rates for all datasets are presented in Figure 5.

Since our policies are orthogonal to issues related to adaptations to changing selectivities, and in order to allow easier analysis of the evaluation results, we consider scenarios where the selectivities are uniform across the complete datasets as shown in Table 2. The total size of each stream for each dataset is shown in Table 3.

Apart from the varying arrival rates and the different selectivities considered, we also investigate the impact of different time periods. As such, we consider two types of data batches: in the former, the data items in a data batch belong to a period of 100K time units, and in the latter, the period is 1M time units. All experiments were repeated 10 times, and here, we present the average values. The experiments were conducted on a 2GHz Intel core i7 processor with four cores, L2 Cache of 256 KB, L3 Cache of 6 MB and physical memory of 4 GB (2 of 2 GB of 1333MHz DDR3). In such a setting (i.e., for those arrival rates, data sizes and resources available), the complete execution of the multi-way join can fit into the main memory and the processing rate is significantly higher than the arrival rate, which allows us to activate the CPU only periodically without sacrificing the
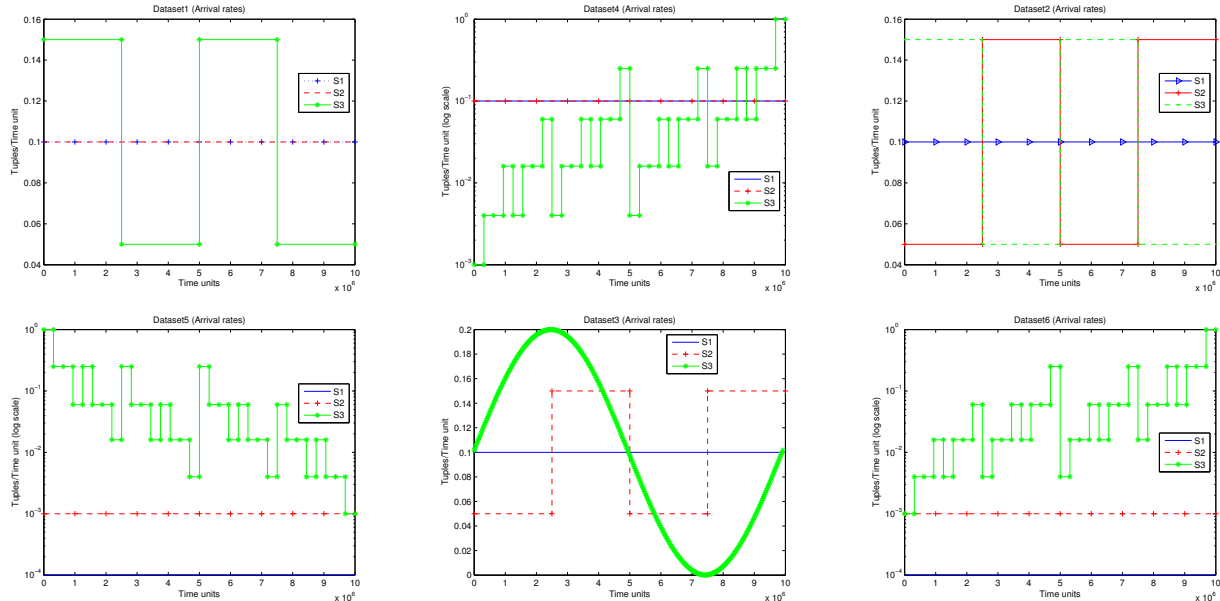
Figure 5: Arrival rates for the datasets used in the experiments.

performance in terms of elapsed time between the arrival of the first tuple and the finish of the processing of the last tuple.

4.2 Experiments

In the experiments, we investigate two aspects, namely the total running time, and the capability to produce results in each batch as early as possible. The total running time is the time spent on processing, where the CPU is utilized and is measured with the help of the `nanoTime()` Java method. The figures presented in this section correspond to the average behaviour across the last 80% of the batches. We disregard the first 20% of the batches in order to allow the hash tables of the *MJoin* to be filled in with a considerable amount of data. When the behaviour of each policy differs significantly throughout the complete execution, we examine the behaviour of the first and the second 40% of the batches separately. Since we present the average behaviour based on the behaviour across several batches, all values are normalized based on the behaviour of the *timestamp-based* policy, i.e., running time 1 corresponds to the running time the *timestamp-based* policy needs to complete the processing of a batch. Additionally, for comparison purposes, we also investigate the behaviour of a *round-robin* driver input selection policy. According to that policy, we select the oldest remaining tuple from each buffer in a batch in a round robin manner.

The average behaviour of all policies regarding the result production for Datasets I-III in a single batch are shown in Figure 6. We do not show the *consumption rate* policy because it is based on the selectivities; however, in these scenarios the selectivities are equal and thus the policy degrades to a round-robin one. From the figure, one can observe that the *timestamp-based* and the *round robin* policies act similarly and finish almost at the same time in all the scenarios (the deviation is less than 0.2%, which is considered negligible). The *initial output size* and the *output rate* policies also exhibit similar behaviour and they are more efficient than

the other approaches: firstly, they manage to produce the 60% of the results in less than the 30% of the time needed the timestamp-based policy to process a batch. Secondly, their total running time is approximately 20% lower than the *timestamp-based* policy.

We now turn our attention to cases with bursty arrival patterns, as covered by Datasets IV-VI. The results are summarized in Figures 7 and 8. For Dataset IV (1st column in the figures), we observe a similar behaviour of the policies for both time period sizes. The *timestamp-based* and the *round robin* policies still behave similarly and finish almost at the same time. In the first half of the join process (Figure 7) the *round robin policy* is slightly faster. The *initial output size* policy and the *output rate* have also similar behaviour and, on average, manage to reduce the total running time by approximately 25% for batches corresponding to a period of 0.1M time units and by 20% for batches with period of 1M time units. The improvements in the running time are higher in the initial part of the execution, where the data distribution is sparse.

For Datasets V and VI, where the stream selectivities differ, we also evaluate the *consumption rate* policy. For Dataset V (2nd column in Figures 7 and 8), the *initial output size* and *output rate* policies produce 80% of the result tuples in the first 20%-30% of the processing time, which is much faster than any other policy. In addition, they both manage to reduce the total running time, whereas the *timestamp-based* policy is inferior also to the other two policies with regards to the total running time. The behaviour of the *consumption rate* policy is interesting: although it produces only a very small portion of the final results up to 70% of the total time, it reduces the total running time similarly to the *output rate* policy. The highest speed-up is achieved by *initial output size* for batch period of 1M time units, where the reduction is about 50% (i.e., half the time of the *timestamp-based* method)). The same policy is also the one that incurs the lowest total running time for Dataset VI. For Dataset VI, the consumption rate reduces the total running
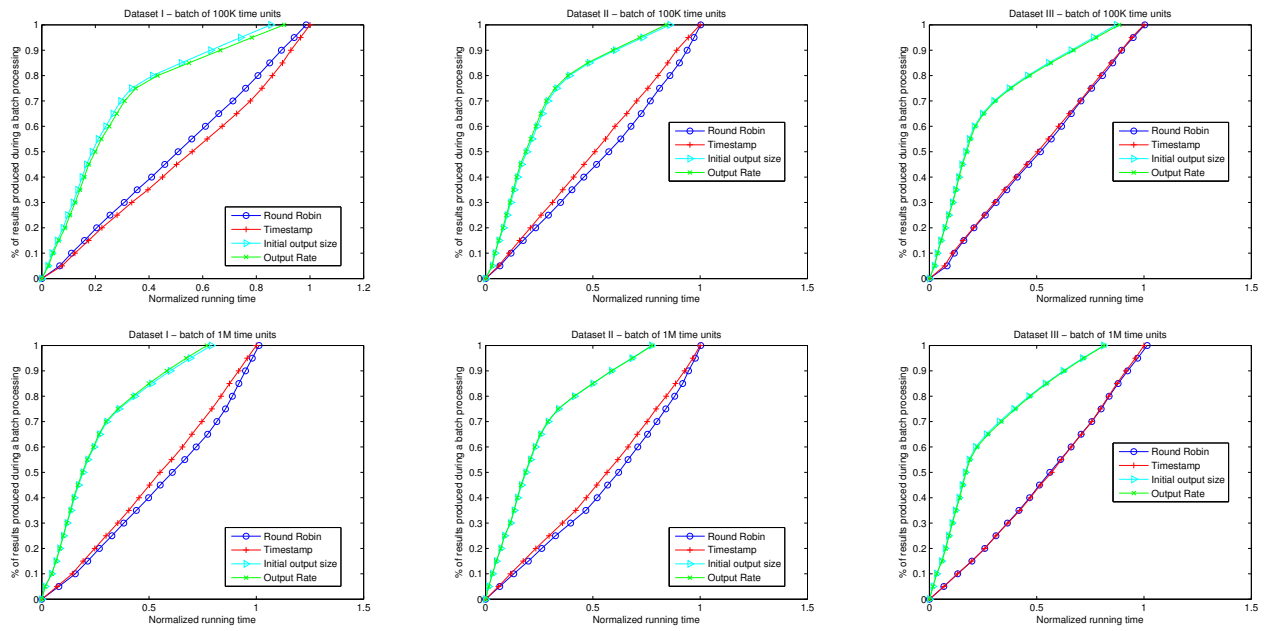
Figure 6: Total running time and production rate for Datasets I-III. The batch time period is 0.1M time units (top) and 1M time units (bottom).
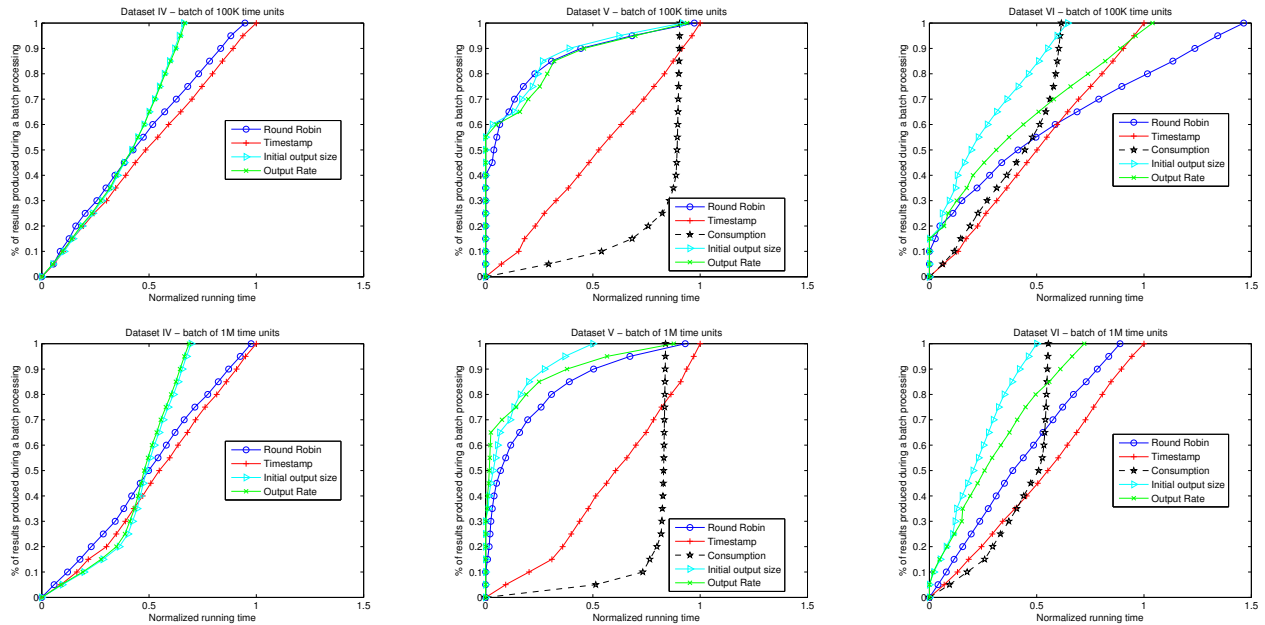

Figure 7: Total running time and production rate for the 1st half of the batches of Datasets IV-VI. The batch time period is 0.1M time units (top) and 1M time units (bottom).

time similarly for the first half, but is much slower in producing early results. In general, also for this dataset, the *timestamp-based* policy is inferior to all three policies proposed in terms of running time and early result production. Tables 4 and 5 summarize the results of this experiment.

Discussion. The experimental evaluation supports our intuition that it is not beneficial to process input tuples in timestamp order. Actually, the *timestamp-based* method is dominated by our proposals in all three dimensions inves-

tigated: aggregate running time, early result production, and early input consumption. Interestingly, even a simple *round-robin* technique may outperform the *timestamp-based* method for bursty arrival rates and large batch sizes. The inferior performance of the *timestamp-based* method is due to both (i) the higher number of probes, and (ii) the more frequent changes in driver selection that incur some overhead. Among the three proposed techniques, there is no technique that is consistently better in all our experiments. However,
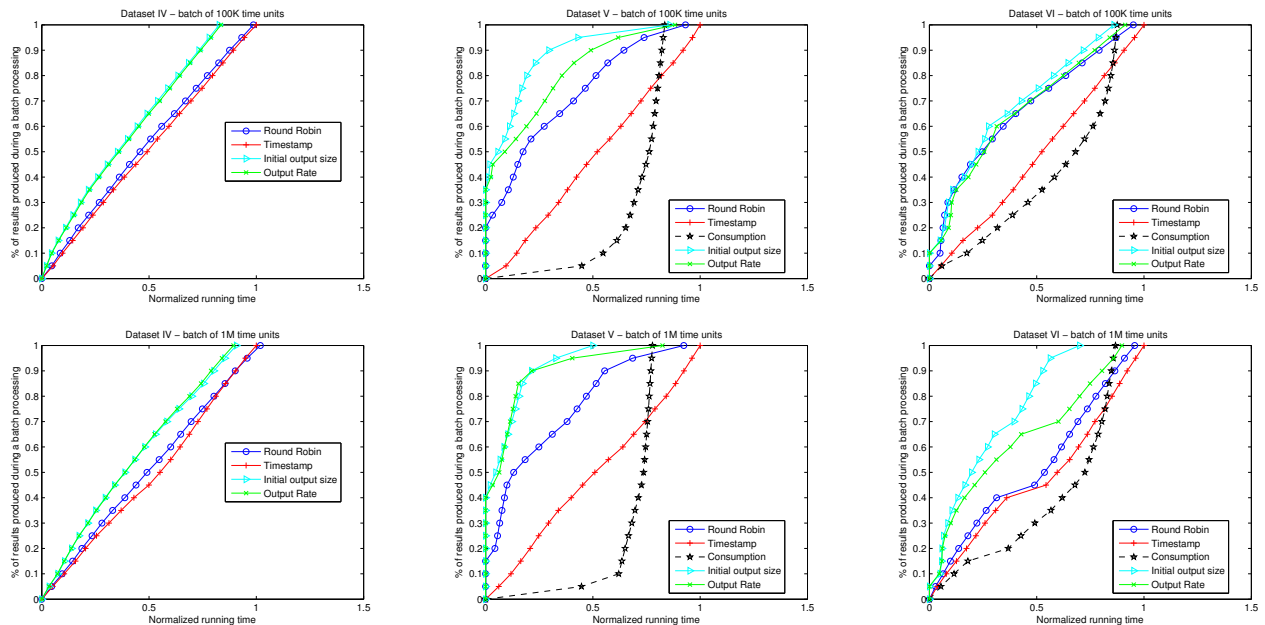
Figure 8: Total running time and production rate for the 2nd half of the batches of Datasets IV-VI. The batch time period is 0.1M time units (top) and 1M time units (bottom).

	Round-robin		Timestamp-based		Consumption Rate		Initial Output size		Output Rate	
Dataset	batch 0.1M	batch 1M	batch 0.1M	batch 1M	batch 0.1M	batch 1M	batch 0.1M	batch 1M	batch 0.1M	batch 1M
Dataset I	0.985	1.012	1	1	N/A	N/A	0.854	0.791	0.903	0.769
Dataset II	1.002	1.002	1	1	N/A	N/A	0.859	0.773	0.840	0.777
Dataset III	1.003	1.014	1	1	N/A	N/A	0.871	0.811	0.884	0.815
Dataset IV	0.966	0.997	1	1	N/A	N/A	0.744	0.801	0.755	0.789
Dataset V	0.948	0.927	1	1	0.863	0.808	0.882	0.5	0.906	0.851
Dataset VI	1.128	0.926	1	1	0.785	0.728	0.785	0.611	0.956	0.819
average	1.005	0.979	1	1	0.824	0.768	0.832	0.714	0.874	0.803

Table 4: Normalized running time of join results.

	Round-robin		Timestamp-based		Consumption Rate		Initial Output size		Output Rate	
Dataset	batch 0.1M	batch 1M	batch 0.1M	batch 1M	batch 0.1M	batch 1M	batch 0.1M	batch 1M	batch 0.1M	batch 1M
Dataset I	0.5	0.4	0.45	0.45	N/A	N/A	0.85	0.85	0.85	0.85
Dataset II	0.45	0.4	0.5	0.45	N/A	N/A	0.85	0.85	0.85	0.85
Dataset III	0.5	0.45	0.5	0.45	N/A	N/A	0.8	0.8	0.8	0.8
Dataset IV	0.55	0.5	0.5	0.45	N/A	N/A	0.65	0.6	0.6	0.6
Dataset V	0.85	0.9	0.5	0.45	0.1	0.05	0.95	1	0.9	0.95
Dataset VI	0.65	0.55	0.5	0.45	0.4	0. 35	0.8	0.95	0.7	0. 7
average	0.58	0.53	0.49	0.45	0.25	0.2	0.82	0.84	0.78	0.79

Table 5: Fraction of join results produced at 0.5 of normalized time units of execution.

Initial-output size is the most efficient, mainly because it is the fastest, especially in scenarios with bursty arrival rates and non uniform selectivities across the streams; in those scenarios we observed reductions in the running time by up to 50%. Due to its low running times, it manages to consume the input buffers earlier than even the *consumption rate* policy.

5. CONCLUSIONS

In this work, we proposed a novel variant of main-memory *MJoins*, which does not process input tuples eagerly but stores them in temporary buffers and periodically activates the CPU to process all the accumulated data. Since *MJoins* have multiple inputs, the choice of the driver input in our approach becomes flexible. Apart from processing tuples in timestamp order as in [35], we propose policies to perform driver input selection, and we show that our policies can outperform timestamp-based methods in terms of the aggregate time that the CPU needs to process the join. Moreover, the proposed driver selection policies are more efficient in producing early results and consuming the input buffers as early as possible. Among our proposals, the best performing one

is shown to be the one that decides on the driver input selection based on the expected size of results to be produced. In realistic scenarios, where the data arrival rates are bursty the stream selectivities differ, the reduction in running time can reach 50%.

This work can be extended in several ways. Firstly, an interesting direction for future work is to conduct theoretical analysis of the relative performance of the proposed policies in order to identify the conditions under which a policy becomes the most efficient one. Secondly, we aim to further examine the impact of more inputs in the multi-way join; in our evaluation, we have evaluated only 3-way joins. Finally, we plan to integrate our methodology to adaptive disk-based *MJoin* variants.

6. REFERENCES

[1] G. Abdulla, T. Critchlow, and W. Arrighi. Simulation data as data streams. *SIGMOD Record*, 33(1):89–94, 2004.

[2] M. H. Ali, W. G. Aref, R. Bose, A. K. Elmagarmid, A. Helal, I. Kamel, and M. F. Mokbel. Nile-pdt: A phenomenon detection and tracking framework for data stream management systems. In *VLDB*, pages 1295–1298, 2005.

[3] M. H. Ali, M. F. Mokbel, W. G. Aref, and I. Kamel. Detection and tracking of discrete phenomena in sensor-network databases. In *SSDBM*, pages 163–172, 2005.

[4] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 407–418, 2004.

[5] M. Bornea, V. Vassalos, Y. Kotidis, and A. Deligiannakis. Adaptive join operators for result rate optimization on streaming inputs. *IEEE Trans. Knowl. Data Eng.*, 22(8):1110–1125, 2010.

[6] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD Conference*, pages 379–390, 2000.

[8] S. Chen, P. Gibbons, and S. Nath. Pr-join: a non-blocking join achieving higher early result rate with statistical guarantees. In *SIGMOD Conference*, pages 147–158, 2010.

[9] C. Cortes, K. Fisher, D. Pregibon, and A. Rogers. Hancock: a language for extracting signatures from data streams. In *KDD*, pages 9–17, 2000.

[10] A. Deshpande. An initial study of overheads of eddies. *SIGMOD Record*, 33(1):44–49, 2004.

[11] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[12] J. Gehrke and S. Madden. Query processing in sensor networks. *IEEE Pervasive Computing*, 3(1):46–55, 2004.

[13] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. Quicksand: Quick summary and analysis of network data. Technical Report 2001-43, DIMACS, 2001.

[14] L. Golab and L. Golab. *Sliding Window Query Processing over Data Streams*. PhD thesis, University of Waterloo, 2006.

[15] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, 2003.

[16] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *Proc. of ACM SIGMOD*, pages 287–298, 1999.

[17] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid.

[18] Stream window join: Tracking moving objects in sensor-network databases. pages 75–84, 2003.

[18] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. pages 171–182, 1997.

[19] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *Proc. of ACM SIGMOD*, pages 299–310, 1999.

[20] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the dbo engine. *ACM Trans. Database Syst.*, 33(4), 2008.

[21] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.

[22] W. Lang and J. Patel. Energy management for mapreduce clusters. *Proc. VLDB Endow.*, pages 129–139, 2010.

[23] J. J. Levandoski, M. E. Khalefa, and M. F. Mokbel. Permjoin: An efficient algorithm for producing early results in multi-join query plans. pages 1433–1435, 2008.

[24] Q. Li, M. Shao, V. Markl, K. Beyer, L. Colby, and G. Lohman. Adaptively reordering joins during query execution. In *ICDE*, pages 26–35, 2007.

[25] B. Liu, Y. Zhu, and E. A. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *SIGMOD Conference*, pages 347–358, 2006.

[26] G. Luo, J. F. Naughton, and C. J. Ellmann. A non-blocking parallel spatial join algorithm. In *ICDE*, pages 697–705, 2002.

[27] A. Motro. Using integrity constraints to provide intensional answers to relational queries. In *Proc. of VLDB' 89*, pages 237–246, 1989.

[28] V. Raman, B. Raman, and J. M. Hellerstein. Online dynamic reordering for interactive data processing. In *In VLDB*, 1999.

[29] S. Srinivasan, H. Latchman, J. Shea, T. Wong, and J. McNair. Airborne traffic surveillance systems: video surveillance of highway traffic. In *Proc. of the ACM 2nd Int. Workshop on Video surveillance & sensor networks, VSSN*, pages 131–135, 2004.

[30] M. Sullivan and A. Heybey. Tribeca: a system for managing large databases of network traffic. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '98, pages 2–2, 1998.

[31] M. syan Chen, M. Lo, P. S. Yu, and H. C. Young. Applying segmented right-deep trees to pipelining multiple hash joins. *IEEE Transactions on Knowledge and Data Engineering*, 7:656–668, 1995.

[32] R. Szewczyk, E. Osterweil, J. Polastre, M. Hamilton, A. Mainwaring, and D. Estrin. Habitat monitoring with sensor networks. *Communications of the ACM*, 47:34–40, 2004.

[33] T. M. Tran and B. S. Lee. Distributed adaptive windowed stream join processing. *IJDST*, 2(2):59–81, 2011.

[34] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.

[35] S. Viglas, J. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003.

[36] S. V. Vrbsky and J. W.-S. Liu. Approximate - a query processor that produces monotonically improving approximate answers. *IEEE Trans. Knowl. Data Eng.*, 5(6):1056–1068, 1993.

[37] M. Wang, N. H. Chan, S. Papadimitriou, C. Faloutsos, and T. M. Madhyastha. Data mining meets performance evaluation: Fast algorithms for modeling bursty traffic. In *ICDE*, pages 507–516, 2002.

[38] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *VLDB*, pages 358–369, 2002.