A Disk-Based, Adaptive Approach to Memory-Limited Computation of Windowed Stream Joins

Abhirup Chakraborty and Ajit Singh

Dept. of Electrical and Computer Engineering
University of Waterloo, ON, Canada, N2L3G1
`abhirupc@ieee.org, asingh@.uwaterloo.ca`

Abstract. We consider the problem of processing exact results for sliding window joins over data streams with limited memory. Existing approaches either, (a) deal with memory limitations by shedding loads, and therefore can not provide exact or even highly accurate results for sliding window joins over data streams showing time varying rate of data arrivals, or (b) suffer from large IO-overhead due to random disk flushes and disk-to-disk stages with a stream join, making the approaches inefficient to handle sliding window joins. We provide an Adaptive, Hash-partitioned Exact Window Join (AH-EWJ) algorithm incorporating disk storage as an archive. Our algorithm spills window data onto the disk on a periodic basis, and refines the output result by properly retrieving the disk resident data, and maximizes output rate by employing techniques to manage the memory blocks and by continuously adjusting the allocated memory within the stream windows. The problem of managing the window blocks in memory—similar in nature to the caching issue—captures both the temporal and frequency related properties of the stream arrivals. The algorithm adapts memory allocation both at a window level and a partition level. We provide experimental results demonstrating the performance and effectiveness of the proposed algorithm.

1 Introduction

With the advances in technology, various data sources (sensors, RFID Readers, web servers, etc.) generate data as high speed data streams. Traditional DBMSs stores data on a disk and processes streams of queries over the persistent data. Contrary to such a 'store first, query next' model, a data stream processing system should process results for long running, continuous queries incrementally as new data arrive in the system. So, continuous queries over bursty, high volume online data streams need to be processed in online fashion to generate results in real time. The new requirements unique to the data stream processing systems pose new challenges providing the motivation to develop efficient techniques for data stream processing systems. Examples of applications relevant to data stream processing systems include network traffic monitoring, fraud detection, financial monitoring, sensor data processing, etc. Considering the mismatch between the traditional DBMS and the requirements of data stream processing, a number of *Data Stream Management Systems* (DSMS) have emerged [1–3]. In this paper we investigate the problems that arise when processing joins over data streams.

Computing the approximate results based on load shedding [4–6] is not feasible for queries with large states (e.g., join with large window size). It is formally shown in reference [5] that given a limited memory for a sliding window join, no online strategy based on load-shedding can be k-competitive for any k that is independent of the sequence of tuples within the streams. Thus, for a system with QoS-based [1] query output, secondary storage is necessary even to guarantee QoS above a certain limit. Pushing the operator states or stream tuples to the disk has already been adopted in several research works [7–10] that process finite data sets, carries out clean-up phase at the end of the streams. Contrary to this scenario, a sliding window stream join should properly interleave the in-memory execution and disk clean-up phases. Also, Algorithms for processing joins over finite streams [7, 8, 11] are not I/O-efficient and do not consider *disk I/O-amortization* over large number of input tuples. Moreover, these algorithms result in a low memory utilization and a high overhead of eliminating duplicate tuples in join output. Thus exact processing of sliding window stream joins within a memory limited environment is a significant and non-trivial research issue as promulgated in [5, 6, 10].

In this paper, we propose an Adaptive, Hash Partitioned Exact Window Join (AH-EWJ) algorithm that endeavors to smooth the load by spilling a portion of both the window blocks onto the disk. The proposed algorithm amortizes a disk-access over a large number of input tuples, and renders the disk access pattern largely sequential, eliminating small, random disk I/Os; it improves memory utilization by employing *passive removal* of the blocks from the stream window and by dynamically adjusting memory allocation across the windows. To increase the output generation rate, AH-EWJ algorithm employs a generalized framework to manage the memory blocks forgoing any assumption about the models (unlike previous works e.g. [5]) of stream arrival.

2 Related Works

Existing join algorithms on streaming data can be classified into two categories: the first one considers bounded or finite size relations, whereas the second category considers the streams of infinite size.

Join over Bounded Stream. Symmetric Hash Join [12], that extends the traditional hash join algorithm, is the first non-blocking algorithm to support pipelining. The XJoin algorithm [7] rectifies the situation by incorporating disk storage in processing joins: when memory gets filled, the largest hash buckets among A and B is flushed into disk. When any of the sources is blocked, XJoin uses the disk resident buckets in processing join. In reference [9], the authors present multi-way join (MJoin) operators, and claims performance gain while compared with any tree of binary join operators. Progressive-Merge Join (PMJ) algorithm [13] is the non-blocking version of the traditional sort-merge join. The Hash-Merge-Join (HMJ) [8] algorithm combines advantages of XJoin and PMJ. Rate-based progressive join (RPJ) [11] focuses on binary joins, and extends the existing techniques (i.e., those in XJioin, HMJ or MJoin) for progressively joining stream relations. Algorithm proposed in [14] is based on a state manager that switches between in-memory and disk-to-disk phases. In order to maximize overall throughout. RIDER [15] algorithm maximizes output rate and enables the system quickly switch between the in-memory stage and disk-to-disk stage.

These algorithms invokes reactive stages (i.e., phases involving disk-resident data) when the CPU is idle or there lies no more tuple to process. However, in case of the sliding window based joins, the clean up or invalidation is a continuous process and should be interleaved with the stream join processing.

Join over Unbounded Data Stream. Joins over infinite stream assumes application of window operators (time- or tuple-based) to limit the memory requirements of continuous queries, thus unblocking query operators. The work presented in [16] introduces a unit-time-basis cost model to analyze the performance of the sliding window joins over two streams. Reference [6] examines the MAX-subset measure and presents optimal, approximate offline algorithms for sliding window joins. Golab et al. [17] presented and evaluated various algorithms for processing multi-joins over sliding windows residing in main memory. Srivastava et al. [5] propose (age- and frequency-based) models that conserves memory by keeping an incoming tuple in a window up to the point in time until the average rate of output tuples generated using this tuple reaches its maximum value. In [4], the authors propose an adaptive CPU load shedding approach for multi-way windowed stream joins. All of the algorithms shed load and thus does not produce exact join results.

3 Exact Join

The basic join operator considered in this paper is a sliding window equi-join between two streams S_1 and S_2 over a common attribute A, denoted as $S_1[W_1] \bowtie S_2[W_2]$. The output of the join consists of all pairs of tuples $s_1 \in S_1, s_2 \in S_2$ such that $s_1.A = s_2.A$, and $s_1 \in S_1[W_1]$ at time $s_2.t$ (i.e., $s_1.t \in [s_2.t - W_1, s_2.t]$) or $s_2 \in S_2[W_2]$ at time $s_1.t$ (i.e., $s_2.t \in [s_1.t - W_2, s_1.t]$. Here, $s_i.t$ denotes the arrival time of a tuple s_i. The proposed algorithm (AH-EWJ) is based on the hashing methodology. Tuples in the stream windows are mapped to one of the n_{part} partitions, using a hash function $\mathcal{H}$ that generates an integer in the range of $[1, n_{part}]$.

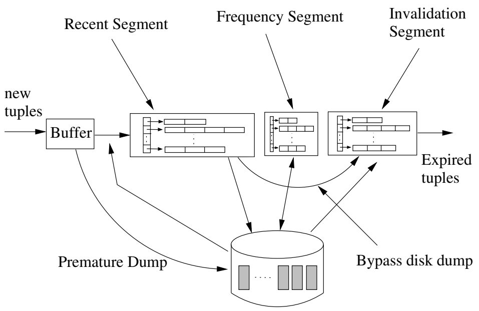

Fig. 1. Organization of incoming tuples within a window

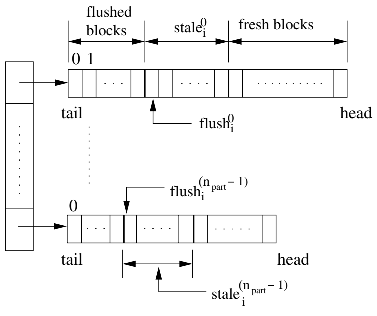

Fig. 2. Data structure of a Recent Segment of a Stream Window

The algorithm consists of four major sub-tasks: maintaining blocks in generative segments, adapting the memory allocation both within and across the stream windows, probing the disk periodically to join the disk resident data with the incoming data, and invalidating the tuples. The algorithm is based on the framework given in Figure 1. The memory portion of each window W_i has three segments in total: invalidation segment (W_i^{inv}), Recent segment (W_i^{rec}), and Frequent segment (W_i^{freq}). Each of the segments is maintained as a hashtable with n_{part} partitions or buckets. We denote the j-th partition of a segment W_i^{seg} as $W_i^{seg}[j]$, where $seg \in \{rec, freq, inv, mem\}$. Due to the lack of space, we omit detailed description of various sub-tasks and refer the readers to [18].

3.1 Memory Stage

The incoming tuples in stream S_i are mapped to the respective partitions using a hash function $\mathcal{H}$ and added to the head block of the partition in the recent segment $W_i^{rec}[p][H]$. Each newly added block within a stream is given a unique number denoted as *block number*. Each block maintains a *productivity value* that records the number of output tuples generated by the tuples in that block. The arriving tuples are joined with the respective partition in the memory segment of the opposite window $W_{\bar{i}}^{mem}[p]$. Blocks (in stream S_i) are added to the recent segment at one end (i.e., head), while the blocks from the other end (i.e., tail) are stored to the disk sequentially during a disk dump. In our join algorithm, we maintain the following constraint:

> Constraint **a**: *At a particular time, the blocks from recent segment of stream S_i should join with all the blocks in the disk portion of window $W_{\bar{i}}$ (i.e., $W_{\bar{i}}^{disk}$) before being dumped to the disk*

Blocks in the recent segment are categorized into three types: flushed blocks, stale blocks and fresh blocks. The fresh blocks are newly arrived blocks that have not yet participated in the disk probing. The stale blocks have already participated in the disk probing, but have not been flushed onto the disk. The flushed blocks have already been dumped onto the disk, and, therefore, can be deallocated to accommodate newly arrived blocks. During the disk probing phase as described in subsection 3.2, blocks retrieved from the disk segment (of the opposite stream) are joined with the *fresh blocks* in the recent segment. After the disk probing, the fresh blocks are marked as stale ones. Blocks from the recent segment are evicted upon the arrival of new tuples.

3.2 Tasks Related to Disk Probing

Frequent Segment Update. The decision about placement or replacement of a block in Frequent segment, that is made periodically, is based on its productivity value. The productivity values are decayed, using a factor α $(0 < \alpha < 1)$, during this update stage. This update stage is carried out during the disk probing. During the update stage, an incoming disk block is brought into the Frequent segment if its productivity exceeds that of a block already in Frequent segment at least by a fraction (ρ). The block having the minimum productivity value among the blocks in the Frequent segment is replaced. To allow efficient decay of the productivity values of the disk resident blocks, we maintain a *productivity table* that stores the productivity values of all the disk resident blocks in memory as a list of tuples <`block number, productivity`>.

Algorithm 1. EVICTRECENTBLOCK(int k)

```
Data: k = stream index, global variables: $W_i; i = 0, 1$
Result: a freed block $b$
1  $i \leftarrow k$;
2  if CANSUPPLY( $\bar{k}$ ) then
3      $i \leftarrow \bar{k}$;
4  if $flush_i = 0$ then
5      if $W_i^{disk} = \phi$ and $free(W_i^{inv}) > 0$ then
6          $p \leftarrow$ partition having the block with the lowest $BN$;
7          remove tailing block $b$ in $W_i^{rec}[p]$;
8          copy $b$ to $W_i^{inv}[p]$ ;
9          decrement $stale_i^p$
10     else
11         $n \leftarrow min(B_{rec}^{min}, \sum_{p=0}^{n_{part}-1} stale_i^p)$;
12         DISKDUMP(i,n) ;
13 if $flush_i \neq 0$ and $free(W_i^{rec}) \leq 0$ then
14     $p \leftarrow \arg\min_{x \mid stale_i^x > 0} (prod_i^x)$;
15     remove tailing block $b$ in $W_i^{rec}[p]$;
16     decrement $flush_i^p$;
17     UPDATEFREQSEGMENT($W_i^{freq}$, b)
18 return $b$
```

Algorithm 2. AH-EWJ

```
   // set to 0; i=0,1; $0 \leq p < n_{part}$
1  initialize variables $premature_i$, $freshN_i$, $UI_i$, $flush_i^p$, $stale_i^p$, $BN_i$;
2  repeat
3      retrieve a tuple $s_i$ from input buffer of $S_i$ in FIFO order ;
4      $p \leftarrow \mathcal{H}(s_i)$ ;
5      $W_i^{rec}[p][H] \leftarrow \{W_i^{rec}[p][H] \cup s_i\}$;
6      if $W_i^{rec}[p][H]$ is full then
7          compute $W_i^{rec}[p][H] \bowtie \{W_{\bar{i}}^{mem}[p]\}$;
8          $W_{rec(i)}[p][H].BN \leftarrow BN_i$ ;
9          increment $BN_i$ ;
10         increment $freshN_i$;
11         if $W_i^{rec}$ is full then
12             $W_i^{rec}[p] \leftarrow$ EVICTRECENTBLOCK(i) ;
13         else
14             add a free block to the head of $W_i^{rec}[p]$ ;
15     if $freshN_i \geq rN_{rec}^{min}$ then
16         $UI_i \leftarrow UI_i + freshN_i \times |b_i^m|$;
17         $freshN_i \leftarrow 0$ ;    // reset $freshN_i$
18         if $UI_i \geq Epoch_i$ then
19             DISKPROBE($W_i^{rec}$, $W_{\bar{i}}^{disk}$, $W_{\bar{i}}^{freq}$, 1);
20             $UI_i \leftarrow 0$;    // reset the count
21         else
22             DISKPROBE($W_i^{rec}$, $W_{\bar{i}}^{disk}$, $W_{\bar{i}}^{freq}$, 0)
23         for $p \leftarrow 0$ to $n_{part} - 1$ do
24             $stale_i^p \leftarrow |W_i^{rec}[p]| - flush_i^p$
25 until Streams ends ;
```

Eliminating Redundant Tuples. A block b_i^p evicted from the Frequent segment of window W_i is already joined with the fresh blocks in the Recent segment of the stream $W_{\bar{i}}$. If the evicted block is not already scanned on the disk, it will be read and be joined with the fresh blocks in the Recent segment of window $W_{\bar{i}}$. To prevent this duplicate processing, we maintain a list E_i containing the block numbers of evicted blocks. If an incoming block from the disk is contained in E_i, we omit processing that block. List E_i is reset to *null* at the end of the update stage.

As described earlier, blocks from the Recent segment are removed on demand. Such passive removals of the blocks might lead to duplicate output generation. Let us consider a block b_i^j in partition $\mathcal{H}(b_j^i)$ of the recent segment W_i^{rec}. The block b_i^j joins with a block $b_{\bar{i}}^k \in W_{\bar{i}}^{rec}[\mathcal{H}(b_j^i)]$. Later b_i^j participates in a disk dump and is stored on the disk. However, due to the passive removal of the blocks from the Recent segment, the same block (b_i^j) remains in memory as a stale block. Now, when a block $b_{\bar{i}}^k$ of the opposite window participates in disk probe, it finds on the disk the block b_i^j already joined in previous step. We use the sequence number of a block, denoted as *block number*, in solving this issue: every incoming block is assigned an unique number from an increasing sequence. Every block b_i^k in partition $\mathcal{H}(b_i^k)$ of the Recent segment W_i^{rec} stores the minimum block number (*minBN*) among the blocks, from the same partition of the Recent segment of the opposite window, that b_i^k joins with. When b_i^k participates in a disk probe, it joins with a block $b_{\bar{i}}^{p} \in W_{\bar{i}}^{rec}[\mathcal{H}(b_k^i)]$ if the block number of the block $b_{\bar{i}}^{p}$ is less than the *minBN* value of the block b_i^k, i.e., $b_{\bar{i}}^{p}.BN < b_i^k.minBN$. As b_i^k is already in W_i^{mem}, any block in $W_{\bar{i}}$ arriving after $b_i^k.minBN$ is already joined with b_i^k. So, during disk probing any block $b_{\bar{i}}^{p} \in W_i^{disk}$ having $b_{\bar{i}}^{p}.BN \geq b_i^k.minBN$ can be omitted.

3.3 Adapting Window and Bucket Sizes

In the join scheme, sizes of the invalidation and the frequent segments are kept constant; the remnant join memory (M_{free}) is allocated between the recent segments. The recent segments adapts their sizes depending on stream arrival rates [19]. We capture the instantaneous arrival rate of a stream using a metric termed as arrival frequency ($C_i; i = 1, 2$), that is maintained using a decaying scheme [18]. Within a recent segment, a partition with the lowest value of the productivity metric ($prod_i^x$) is selected as a viction partition, that yields a taling, flushed block. Algorithm 1 presents the algorithm for evicting a block from a recent segment.

3.4 Join Algorithm

We now present the join algorithm AH-EWJ in Algorithm 2. Within the join algorithm an infinite loop fetches, in each iteration, tuples from an input buffer and joins the fetched tuples with the opposite stream. At each step, the stream having a pending tuple/block with lower timestamp (i.e., the oldest one) is scheduled. A tuple fetched from the buffer is stored in the block at the head of the respective partition within the Recent segment. If the head-block of a partition p becomes full, it is joined with the memory portion of the partition ($W_i^{mem}[p]$). The partition p is allocated a new head block (line 2–2).The variable $freshN_i$ tracks the number of fresh blocks in W_i^{rec}. Whenever a recent segment is filled with at least N_{rec}^{min} fresh blocks, the disk probe phase in invoked (line 2). Frequent-segment-update phase, that occurs at a interval no less than $Epoch_i$, is merged with a disk-probe (described in 3.2). At the end of the disk-probe, parameter $stale_i^p$ for each partition p is changed, converting the fresh blocks within the partition to stale ones (line 2–2).

Table 1. Notations and the system parameters

Notations	Description
S_i, $S_{\bar{i}}$	stream i and opposite to i, respectively
W_i, $W_{\bar{i}}$	Window of stream S_i and $S_{\bar{i}}$, respectively
W_i^{disk}	Disk portion of window W_i
W_i^{rec}	recent segment of window W_i
W_i^{freq}	Frequent segment of window W_i
W_i^{inv}	invalidation segment of window W_i
W_i^{mem}	memory portion of window W_i, $(W_i^{rec} + W_i^{freq} + W_i^{inv})$
$W_i^{seg}[p]$	partition p of $seg \in \{rec, freq, inv\}$
$W_i^{rec}[p][H]$	block at the head of $W_i^{rec}[p]$
b_i^p	block p in window W_i
$b_i^p.minBN$	minimum block number from $W_{\bar{i}}^{disk}$ & $W_{\bar{i}}^{rec}$ that $b_i^p \in W_i^{rec}$ joins with
$b_i^p.prod$	Productivity of block b_i^p

4 Experiments

This section describes our methodologies for evaluating the performance of the AH-EWJ algorithm and presents experimental results demonstrating the effectiveness of the proposed algorithm. We begin with an overview of the experimental setup.

4.1 Simulation Environment

All the experiments are performed on an Intel 3.4 GHz machine with 1GB of memory. We implemented the prototype in Java. The main focus of our experimentation is to observe the effectiveness of the join processing algorithm. The buffer is virtually unbounded and plays no role in the experiments. We generate the time varying data streams using two existing algorithms: PQRS [20] and *b-model* [21]. PQRS algorithm models the spatiotemporal correlation in accessing disk blocks whereas the b-model captures the burstiness in time sequence data [19].

To model the access time for the disk segment, we divide a disk segment (W_i^{disk}) into n_i basic windows $B_{ij}^{win}(j = 1 \ldots n_i)$ [4]. We assume that disk blocks within a basic window are physically contiguous, and the delay in accessing the blocks in a basic window is estimated only by the disk bandwidth (i.e., no seek or latency). However, accessing blocks beyond the boundary of a basic window imparts an extra delay equivalent to the seek time and rotational latency (i.e., an access time). As a base device, we consider IBM 9LZX. We fix the memory page and also the disk block size to 1KB. Each record has a length of 64 bytes. We set the size of the basic window to 1MB. We allocate 70% of memory to the recent segments. The remaining memory is equally distributed among the invalidation and frequent segments. Minimum size of a recent segment (B_{rec}^{min}) is set to the fraction 0.3 of the total memory reserved for the recent segments. We set the minimum delay between successive reactive stages (for RPWJ) as 15sec. The domain of the join attribute A is taken as integers within the range $[0 \ldots 10 \times 10^6]$. In addition to the total pending or buffered tuples, we measure *average production delay*, total CPU time, total disk time and maximum total window size. Unless otherwise stated, the default values used in the experiments are as given in Table 2.

4.2 Experimental Results

In this section, we present a series of experimental results for assessing the effectiveness of the proposed join algorithm. As a baseline algorithm, we use an algorithm termed as RPWJ (Rate-based Progressive Window Join) [19], which extends an existing algorithm RPJ [11]—a variant of XJoin [7]—to process the sliding window joins. The extension is imperative, as RPJ is an algorithm to join only the finite streams. For each set of the experimentations, we run the simulator for 1.6 simulation hours. We start to gather performance data after an startup interval of 12 minutes is elapsed.

Table 2. Default values used in experiments

Parameter	Defaults	Comment
$W_i(i = 1, 2)$	10	Window length(min)
τ	1	slide interval for a Window (min)
λ	1200	Avg. arrival rate(tuples/sec)
α	0.4	Decay parameter for the productivity value
b	0.6	burstiness in traces (captured by *b-model*)
ρ	0.4	block eviction threshold
r	0.9	disk probe threshold
M	20	join memory (MB)
N_{flush}	$0.4M$	flush size for RPWJ
n_{part}	60	hash partitions or buckets

We, now, present the experimental results with varying stream arrival rates. With the increase in the arrival rates, more and more tuples can not be processed within the time limit of the stream window; thus, The percent of i*delayed tuple*s, that can not be

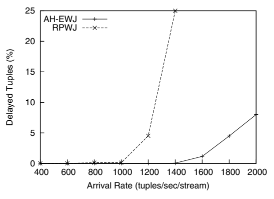

Fig. 3. Percentage of delayed tuples vs stream rates

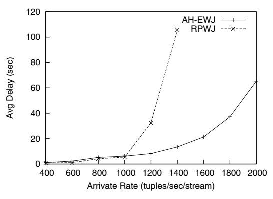

Fig. 4. Average delay vs stream rates

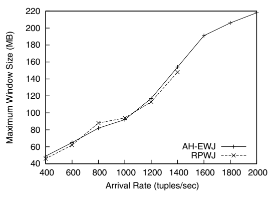

Fig. 5. Maximum window size (MB) vs rates

processed within the time limit of the stream window, increases with the increase in the arrival rates as shown in Figure 3. These delayed tuples get expired at a time later than the usual after they get joined with the respective window. For RPWJ, the percentage of the delayed tuples increases sharply with the increase in arrival rates beyond 1000 tuples/sec; however, in case of AH-EWJ, this percent of delayed tuples remains low even for an arrival rate of 1800 tuples/sec/stream (i.e., 3600 tuples/sec in the system). Here it should be noted that, *only arrival rates do not provide the complete indication about the load applied to the system; the load of the join module is also determined by the window size*.

Figure 4 shows, for both the algorithms, the average delay in producing output tuples with the increase in arrival rates. Figure 5 shows the maximum window size during the system activity. Though the allocated memory per stream window is 20MB, spilling the extra blocks onto the disk does not impart significant increase in average output delay of the AH-EWJ even for arrival rates up to 1600 tuples/sec/stream, at a point where maximum total window size is around 190MB (i.e., 9 times the join memory size). Techniques based on load-shedding would have discarded the extra blocks beyond 20MB losing a significant amount of output tuples that would never have been generated. The RPWJ algorithm becomes saturated for an arrival rate 1400 tuples/sec/stream at a point where the average delay attains a very high value. Hence, for a large window, the proposed technique attains a low average delay of output generation; at the same time, the percentage of the tuples missing the time frame set by the window is very low (0.05% for an arrival rate 1400 tuples/sec/stream). This demonstrates the effectiveness of the AH-EWJ algorithm.

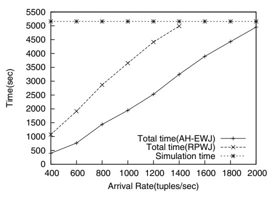

Fig. 6. Total time (disk-IO and CPU time) vs rates

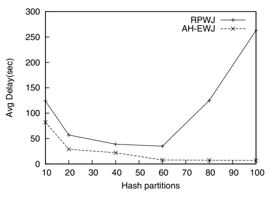

Fig. 7. Avg production delay vs total hash partitions

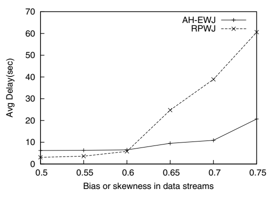

Fig. 8. Avg production delay vs bias in stream arrivals

Figure 6 shows the total system time (disk I/O and CPU time) with varying per stream arrival rates. We notice that with the RPWJ algorithm the system is saturated at the rate of 1400 tuples/sec/stream. On the other hand, with the same arrival rate, the AH-EWJ algorithms achieves around 40% reduction in total execution time.

Figure 7 shows the effect of increasing hash partitions on the performance of the two algorithms. Algorithms based on sort-merge joins can not be applied in processing sliding window joins due to temporal order of the tuples [19]. So, increasing the number of hash partitions is a simple approach to lower the processing overhead. But, the performance of the RPWJ algorithm deteriorates with the increase in the hash partitions; this phenomenon renders the RPWJ unfit for the sliding window joins. On the other hand, AH-EWJ attains lower average delay with an increase in the hash partitions. Figure 8 shows the average delay in generating output tuples with varying bias or burstiness of the data stream. As shown in the figure, with the increase in the bias, AH-EWJ performs significantly better than the RPWJ.

5 Conclusion

In this paper, we address the issue of processing exact, sliding window join between data streams. We provide a framework for processing the exact results of an stream join, and propose an algorithm to process the join in a memory limited environment having burstiness in stream arrivals. Storing the stream windows entirely in memory is infeasible in such an environment. Like any online processing, maximizing output rate is a major design issue in processing exact join. Hence, we propose a generalized framework to keep highly productive blocks in memory and to maintain the blocks in memory during systems activity, forgoing any specific model of stream arrivals (e.g., age based or frequency based model [5]). The algorithm reduces disk dumps by adapting the sizes of both the windows and the partitions based on, respectively, the stream arrival rates and the productivity of blocks within the partitions. The experimental results demonstrate the effectiveness of the algorithm.

References

1. Carney, D.: etintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Monitoring streams – a new class of data management applications. In: Proc. Intl. Conf. on Very Large Databases (VLDB), Hong Kong, China, August 2002, pp. 215–226 (2002)
2. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.A.: TelegraphCQ: Continuous dataflow processing for an uncertain world. In: Proc. Conf. on Innovative Data Systems Research, CIDR (January 2003)
3. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Processing sliding window multi-joins in continuous queries over data streams. In: Proc. ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS), Madison, Wisconsin, USA, June 2002, pp. 1–16 (2002)
4. Gedik, B., Wu, K.-L., Yu, P.S., Liu, L.: A load shedding framework and optimizations for m-way windowed stream joins. In: Proc. Intl. Conf. on Data Engineering (ICDE), Istanbul, Turkey, April 2007, pp. 536–545 (2007)

5. Srivastava, U., Widom, J.: Memory-limited execution of windowed stream joins. In: Proc. Intl. Conf. on Very Large Databases (VLDB), Toronto, Canada, September 2004, pp. 324–335 (2004)
6. Das, A., Gehrke, J., Riedewald, M.: Approximate join processing over data streams. In: Proc. ACM SIGMOD Intl. Conf. on Management of Data, San Diego, USA, June 2003, pp. 40–51 (2003)
7. Urhan, T., Franklin, M.J.: XJoin: A reactively-scheduled pipelined join operator. IEEE Data Engineering Bulletin 23(2), 7–18 (2000)
8. Mokbel, M., Liu, M., Aref, W.: Hash-merge-join: A non-blocking join algorithm for producing fast and early join results. In: Proc. Intl. Conf. on Data Engineering (ICDE), pp. 251–263 (2004)
9. Viglas, S.D., Naughton, J.F., Burger, J.: Maximizing the output rate of multi-way join queries over streaming information sources. In: Proc. Intl. Conf. on Very Large Databases (VLDB), Berlin, Germany, September 2003, pp. 285–296 (2003)
10. Liu, B., Zhu, Y., Rundensteiner, E.A.: Run-time operator state spilling for memory intensive long-running queries. In: Proc. ACM SIGMOD Intl. Conf. on Management of Data, Chicago, Illinois, USA, June 2006, pp. 347–358 (2006)
11. Tao, Y., Yiu, M.L., Papadias, D., Hadjieleftheriou, M., Mamoulis, N.: RPJ: Producing fast join results on streams through rate-based optimization. In: Proc. ACM SIGMOD Intl. Conf. on Management of Data, Baltimore, Maryland, USA, June 2005, pp. 371–382 (2005)
12. Wilschut, A.N., Apers, P.M.G.: Dataflow query execution in a parallel main-memory environment. In: Proc. Intl. Conf. on Parallel and Distributed Information Systems (PDIS), Miami, Florida, USA, December 1991, pp. 68–77 (1991)
13. Dittrich, J.-P., Seeger, B., Taylor, D.S., Widmayer, P.: Progressive merge join: A generic and non-blocking sort-based join algorithm. In: Proc. Intl. Conf. on Very Large Databases (VLDB), Hong kong, China, August 2002, pp. 299–310 (2002)
14. Levandoski, J., Khalefa, M.E., Mokbel, M.F.: Permjoin: An efficient algorithm for producing early results in multi-join query plans. In: Proc. Intl. Conf. on Data Engineering (ICDE), Cancun, Mexico, pp. 1433–1435 (2008)
15. Bornea, M.A., Vassalos, V., Kotidis, Y., Deligiannakis, A.: Double index nested-loop reactive join for result rate optimization. In: Proc. Intl. Conf. on Data Engineering (ICDE), pp. 481–492 (2009)
16. Kang, J., Naughton, J.F., Viglas, S.: Evaluating window joins over unbounded streams. In: Proc. Intl. Conf. on Data Engineering, Bangalore, India, March 2003, pp. 341–352 (2003)
17. Golab, L., Ozsu, T.: Processing sliding window multi-joins in continuous queries over data streams. In: Proc. Intl. Conf. on Very Large Databases (VLDB), Berlin, Germany, September 2003, pp. 500–511 (2003)
18. Chakraborty, A., Singh, A.: A partition-based approach to support streaming updates over persistent data in an active data warehouse. To Appear Proc. IEEE Intl. Symp. on Parallel and Distributed Processing (IPDPS), Rome, Italy, May 2009, pp. 1–11 (2009)
19. Chakraborty, A., Singh, A.: A Disk-based, Adaptive Approach to Memory-Limited Computation of Exact Results for Windowed Stream Joins. Department of Electrical & Computer Engineering, Technical Report UW-ECE #2009-09, University of Waterloo, Canada (2009)
20. Wang, M., Ailamaki, A., Faloutsos, C.: Capturing the spatio-temporal behavior of real traffic data. In: IFIP Intl. Symp. on Computer Performance Modeling, Measurement and Evaluation, Rome, Italy (September 2002)
21. Wang, M., Papadimitriou, S., Madhyastha, T., Faloutsos, C., Change, N.H.: Data mining meets performance evaluation: Fast algorithms for modeling bursty traffic. In: Proc. Intl. Conf. on Data Engineering, February 2002, pp. 507–516 (2002)