Scalable Data Partitioning Techniques for Parallel Sliding Window Processing over Data Streams *

Cagri Balkesen and Nesime Tatbul
Systems Group, ETH Zurich, Switzerland
{cagri.balkesen, tatbul}@inf.ethz.ch

ABSTRACT

This paper proposes new techniques for efficiently parallelizing sliding window processing over data streams on a shared-nothing cluster of commodity hardware. Data streams are first partitioned on the fly via a continuous split stage that takes the query semantics into account in a way that respects the natural chunking (windowing) of the stream by the query. The split does not scale well enough when there is high degree of overlap across the windows. To remedy this problem, we propose two alternative partitioning strategies based on batching and pane-based processing, respectively. Lastly, we provide a continuous merge stage at the end that combines the results on the fly while meeting QoS requirements on ordered delivery. We implemented these techniques as part of the Borealis distributed stream processing system, and conducted experiments that show the scalability of our techniques based on the Linear Road Benchmark.

1. INTRODUCTION

Stream processing has matured into an influential technology over the past decade having a wide range of application domains including sensor networks. Yet, flexibly scaling streaming systems up and down with fluctuating data rates in a cost-effective way continues to be a challenge. With the advances in parallel processing platforms and the emergence of the pay-as-you-go economic model of the new cloud-based infrastructures, there is a more pressing need than ever for flexible distributed and parallel stream processing techniques that can take better advantage of the increased opportunities for elastic scalability.

As in traditional parallel databases, stream processing can be parallelized in two alternative ways: pipelined and partitioned [4]. In pipelined parallelism, continuous queries are partitioned across different nodes (e.g., [11]), whereas in partitioned parallelism data streams are partitioned into different nodes (e.g., [10, 6, 7]). As also pointed out by DeWitt and Gray, pipelined parallelism has limited applicability under certain conditions (for short chains of operators, blocking operators, operators with cost skew) [4]. In the case of streams, there are additional limitations of pipelined parallelism such as the fact that workload can only be divided in operator-granularity and that moving query parts across nodes might require expensive state migration.

In this paper, we investigate partitioned parallelism techniques for data streams as we believe this is key to achieving fine-grained, flexible, and low-overhead scalability for continuous queries. In particular, we address the problem of how to efficiently provide partitioned parallelism for a high-rate input stream which cannot be processed on a single node and cannot be further partitioned using a content-sensitive partitioning technique such as the ones proposed in related work [10, 7]. We present new stream partitioning techniques that take the query semantics into account in a way that respects the natural chunking and ordered processing of the stream by the query. Furthermore, in our approach, partition granularity can be adjusted in order to handle potential overhead due to overlap across the chunks. More specifically, partitioning can be based on windows, batches of windows, or special subwindows called "panes". Finally, we provide a continuous merge stage at the end that combines the results on the fly, while meeting the requirements on ordered delivery.

The rest of this paper is outlined as follows: We present our stream partitioning techniques in Section 2. Then we experimentally compare the scalability of these techniques in Section 3. We briefly summarize the related work in Section 4, and conclude in Section 5 with a discussion of future directions.

2. STREAM PARTITIONING TECHNIQUES

2.1 Overview

Figure 1 shows a high-level overview of our general parallel stream processing framework. We follow the typical split-merge pattern of traditional partitioned data parallelism. Input streams are first fed into a split node that is dedicated to splitting the data into independently processable partitions. Given a degree of parallelism d in the system, the split stage will chop its input into d partitions, each of which will be forwarded to a separate query node. Results from the query nodes will be pushed into a merge node that is dedicated to merging them into a single output stream. The framework also takes QoS requirements of the query and uses them in tuning the behavior of the system. More specifically, QoS is specified in terms of maximum result latency and the maximum degree of disorder in the output stream.

2.2 Basic Sliding Window Partitioning

Most streaming applications need to divide their input into finite chunks of windows before they apply computations on them. Windows are commonly modeled as count-based or time-based with size and slide parameters [5].

*This work has been supported in part by an IBM faculty award.

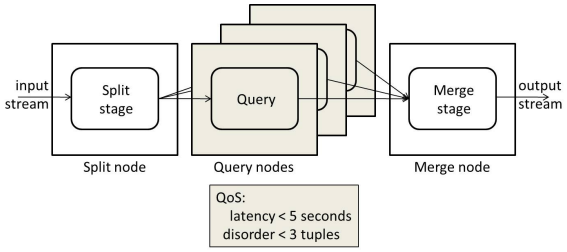


Figure 1: Parallel stream processing framework overview

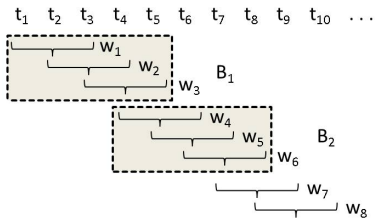


Figure 2: Batch-windows

Any data parallelization scheme on streams should consider how a query chunks its input into windows, since this chunking creates dependencies among stream tuples. Tuples that belong to a common window have a dependency, whereas tuples of different windows are completely independent. As a result, while each window should be processed in a serial fashion, multiple windows can be processed in parallel. This is the key idea behind our basic sliding window partitioning.

In order to capture the window-membership dependency, the split stage of our framework must be window-aware. It should decide which tuples should be in the same partition and should accordingly route them to the appropriate query nodes. We achieve this by utilizing the window specifications and having split stage mark the tuples with window-ids [9]. Tuples with the same window-id must for sure participate in the same data partition.

Split marks the tuples as follows: Each tuple in the system has a unique id (starting from 1 and assigned either by its source or by the system at its arrival). For count-based windows, these tuple-id's are used to determine window boundaries, whereas for time-based windows, the time field must be used instead. Furthermore, each window is also assigned a unique id by the split stage. Each tuple may take part in one or more windows depending on the values of the window size (w) and slide (s). The first and the last windows that a tuple with id i belongs to will have the window-ids $first = \lceil (i - w)/s \rceil + 1$ and $last = \lceil i/s \rceil$, respectively. We also need to mark which tuple is a window-closer so that we can keep track of partition boundaries. For count-based windows, tuple i is a window-closer if $((i \geq w) \wedge ((i - w) \bmod s \equiv 0))$. For time-based windows, this condition is not sufficient as there may be multiple tuples with the same time value. The very last tuple where this condition holds should be marked as a window-closer. To summarize, split marks each tuple with the following metadata: $(first\_window\_id, last\_window\_id, is\_window\_closer)$.

Next, split assigns the tuples with the same window-id to the same data partition, which is then routed to a selected query node. Since our basic window partitioning strategy is content-insensitive, the partitions can be assigned to the nodes simply in a round-robin fashion. For tumbling windows ($s = w$), each tuple belongs to exactly one window. Therefore, input will be perfectly partitioned. However, for sliding windows ($s < w$), windows overlap and therefore, a tuple will have to be replicated in all the partitions where its windows are routed. This not only makes the split stage more expensive, but it also increases the data volume, leading to extra overhead in an already overwhelmed system. In the following two sections, we describe two alternative strategies to overcome this drawback of the basic sliding window partitioning.

2.3 Batch-based Partitioning

Our batch-based partitioning strategy groups a number of consecutive windows into a batch and assigns them to a common partition. This is a coarser-grained partitioning than the basic window-based partitioning. The key idea behind this strategy is to reduce the need for tuple replication to multiple partitions by reducing the overlap across those partitions. Essentially, batching introduces another level of windowing on top of the original query windows; let's call this the "batch-window". Given a query with window size w and slide s, and a batch size of B, then the batch-window would have size $w_b = w + (B - 1) * s$ and slide $s_b = B * s$.

Figure 2 shows an example with $w = 3$, $s = 1$, and $B = 3$. t_i are the stream tuples, w_j are the original query windows, and B_k are the batch-windows. Batch-windows have size $w_b = 3 + (3 - 1) * 1 = 5$ tuples and slide $s_b = 3 * 1 = 3$ tuples.

As in basic partitioning, tuples are marked, but this time with batch-id's in addition to window-id's. Batch-id's are determined using the formulas given in the previous section, using w_b and s_b instead of w and s, respectively. Additionally, tuple i is a batch-window-closer if $(is\_window\_closer(i) \wedge (window\_id \bmod B \equiv 0))$.

This time, split assigns the tuples with the same batch-id to the same data partition and each partition can be assigned to nodes in round-robin. Unlike in the window-based partitioning case, if two windows are in the same batch, then common tuples among them do not need to be replicated to multiple partitions. Only the tuples lying on the batch-window boundaries need to be replicated across partitions. For example, in Figure 2, tuples t_4 and t_5 participate in both batch-window B_1 and B_2. Therefore, they should be replicated to both corresponding partitions. In general for this simple example, at steady state, 2 out of every 3 tuples will have to be replicated to 2 partitions. If window-based partitioning was used instead, each tuple will have to be replicated to 3 partitions. For every 3 tuples, this makes 5 copies in the batch-based approach vs. 9 copies in the window-based approach. The benefit of the batch-based approach would increase further with a larger batch size.

To generalize, in window-based partitioning, each tuple belongs to $\frac{w}{s}$ windows and thus must be replicated to this many partitions. On the other hand, in batch-based partitioning, each tuple belongs to $\frac{w_b}{s_b} = \frac{w+(B-1)*s}{B*s}$ batch-windows and thus must be replicated to this many partitions. It is clear that the larger is the batch size B, the fewer tuples will need to be replicated.

Setting the Batch Size. Choosing the correct batch size is an important decision. As discussed above, a smaller batch size would lead to more tuple replication; on the other hand, a larger one would lead to higher output latency. We will approach this issue from the cost of the aggregate computation, which is the key component contributing to the latency of an output. First, aggregate computation involves evaluating an optional group-by clause, retrieving the hash-table entry for the group, etc. for each received tuple. Let's denote the cost of all these tasks with c_{other}. Second, there is an additional cost originating from window opening and closing tuples. Let's denote it as $c_{open-close}$, which happens B times in each batch. Lastly, each tuple marked by batch-based partitioning updates up

to B windows appearing in its batch; let's denote single update cost with c_{update}. First s tuples in a given batch only update window 1, next s tuples update windows 1 and 2, and so on up to B^{th} s tuples updating windows $1 \dots B$. Finally, remaining $w - s$ tuples update only the B^{th} window. Now, we can model per-tuple cost of aggregate operator as a function of B as follows:

$$Cost(B) = \frac{\frac{B(B+1)}{2} \cdot s + (w - s)}{w + (B-1) \cdot s} \cdot c_{update} + \frac{B}{w + (B-1) \cdot s} \cdot c_{open-close} + c_{other} \quad (1)$$

The input rate at aggregate operator is also a function of batch-size. Total output rate at split divided by parallelization level (d) times the replication factor determines the average rate at an aggregate operator instance. Let's assume R is the input rate at split and denote input rate at each aggregate as a function of B with $Rate(B)$.

$$Rate(B) = \frac{R}{d} \cdot \left(\frac{w + (B-1) \cdot s}{B \cdot s} \right) \quad (2)$$

Given the user-specified maximum latency bound (L_{max}), we can now choose appropriate B at runtime using Equations (1) and (2). Assuming a small headroom ratio H for other overhead, $1 - H$ of processing capacity can be used for operator execution. Accordingly, capacity of aggregate equals $(1 - H)/Cost(B)$ depending on B. Now, the following inequality should hold for fulfilling the maximum target latency L_{max}, where the system is periodically optimized every T time units:

$$\left(Rate(B) - \frac{1 - H}{Cost(B)} \right) * T < L_{max} \quad (3)$$

There is one subtle point. The inequality may not have a solution on positive x-axis. In this case, latency violation will be kept at minimum by minimizing the rational function $f(B) = Rate(B) - \frac{1-H}{Cost(B)}$. By rigorous analysis, we have confirmed that the rational function $f(B)$ attains a local minima for $B > 0$. As a result, an optimal batch-size B can be found either by solving inequality (3) if there is a solution, or by finding the local minima of $f(B)$.

2.4 Pane-based Partitioning

Our pane-based partitioning strategy takes the opposite approach of the batch-based partitioning. Instead of grouping multiple windows into a common partition, windows are divided into sub-windows, which are then assigned to partitions individually. As such, it is a finer-grainer method than both window- and batch-based partitioning. This strategy is based on the panes work of Li et al [8]. In this section, we will first summarize that work, and then describe how we leverage it in solving our problem.

2.4.1 The Panes Technique

The panes technique has been originally proposed to reduce the space and computation cost of sliding window queries by sub-aggregating and sharing computation [8]. The idea is to divide overlapping windows into disjoint panes, over which sub-aggregates can be computed whose results can then be combined into the final aggregate. Figure 3 shows panes p_i over sliding windows w_j. Given a query with window size w and slide s, the stream is separated into panes of size $gcd(w, s)$, each window w_j of size w is composed of $\frac{w}{gcd(w,s)}$ consecutive panes. For example, w_3 is composed of panes p_3 through p_6. Likewise, each pane contributes to $\frac{w}{s}$ windows. For example, p_5 contributes to windows w_2 through w_5.

In pane-based query evaluation, the query is decomposed into two sub-queries: (i) pane-level sub-query (PLQ) that runs over the panes, and (ii) window-level sub-query (WLQ) that runs over the results of the panes. Efficiency is achieved due to two reasons: (i) PLQ is a tumbling-window query (size and slide equals to $gcd(w, s)$), where each tuple is processed only once as it arrives and does not need to be buffered for repeated use, and (ii) WLQ is a sliding-window query (size and slide equals to the original query's, w and s), which does less processing and buffering, since it processes pane results instead of the raw input tuples.

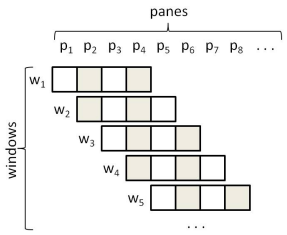


Figure 3: Panes [8]

2.4.2 Ring-based Pane Partitioning

The original panes algorithm has been designed for centralized processing. In our work, we leverage it in a distributed setting. Pane-based processing is a good fit for our problem since it creates non-overlapping sub-windows (i.e., panes) out of overlapping windows, each of which can be efficiently and independently processed in a parallel manner. In this section, we will explain how we adapt the panes idea to parallelizing the processing of sliding window queries.

As in the case of basic partitioning, tuples are marked at the split stage, but this time with pane-id's in addition to window-id's. Pane-id's are determined using the same formulas as in Section 2.2, using w_p instead of w and s_p instead of s, where $w_p = s_p = gcd(w, s)$.

Each pane can potentially be processed on any selected node. At the end of this process, PLQ results are obtained. To obtain the final aggregate result for a given window, a WLQ must additionally aggregate the pane results for that window. This bears a need to communicate pane results among the nodes. Furthermore, a particular node should be responsible for the overall computation and result delivery for a given window. Hence, this node expects to receive pane results of that window from the other nodes. Consequently, if windows and panes were assigned to nodes in an arbitrary fashion, then every node in the system would have to know the locations of all windows and panes so that pane results could be correctly communicated. This would be both disorganized and inefficient.

Instead, we organize nodes in a ring topology and distribute data as follows: As new windows arrive, they are assigned to the next node in the ring in a round-robin fashion. Likewise, pane partitions are also assigned and distributed using a round-robin strategy. This makes it possible that a pane result from the previous window can be sent to the next node in the ring, where ring topology implicitly maps the pane to the next window. Let us explain this mechanism with an example.

Figure 4 illustrates a ring with three query nodes. In this example, the query has $w = 6$ and $s = 4$ tuples. Therefore, each pane has $w_p = s_p = gcd(6, 4) = 2$. Thus, there are 3 panes per window. Every query node receives its assigned pane partitions (P_i) as a stream from the split stage and is responsible for locally computing the PLQ's for those. Additionally, each node is connected to previous and next nodes on the ring via "intermediary streams", which carry the PLQ results (R_i). Each query node is also assigned to manage the overall WLQ computations of certain windows (w_i) based on the PLQ results it receives from its previous neighbor in the ring and its own local sub-window results. For example, w_2 is assigned to $Node_2$, where panes P_4 and P_5 are locally computed.

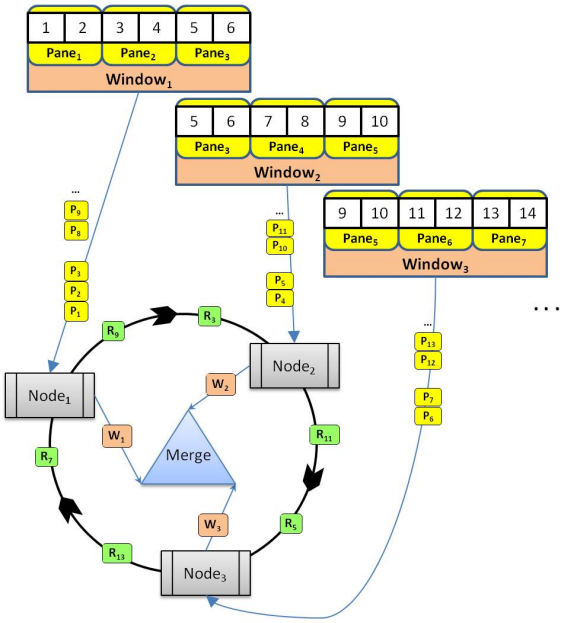


Figure 4: Ring-based pane partitioning and query evaluation

Additionally, $Node_2$ receives the PLQ result R_3 from $Node_1$. Using R_3 and the local results R_4 and R_5, $Node_2$ can produce the WLQ result for w_2. Finally, the WLQ results (w_i) are directly sent to the merge node, where aggregate window results for the query are merged into one output stream.

2.4.3 *Assignment of Windows and Panes to Nodes*

In this section, we will describe how we assign windows and panes to the nodes in the ring. In the following, we will express WLQ-window size (w_w) and slide (s_w) in terms of pane units. For example, in Figure 4, $w_w = 3$ panes and $s_w = 2$ panes.

First, we must ensure that all pane results of a window arrives to a node only from its predecessors. We should assign a set S_i of n consecutive windows to a node i in such a way that the pane results that i receives from node $i-1$ do not overlap with the pane results that i will send to node $i+1$. Node i receives $w_w - s_w$ pane results from node $i-1$, and likewise sends $w_w - s_w$ pane results to node $i+1$. Thus, we should have at least $2 * (w_w - s_w)$ panes contained in set S_i (left-hand side of Figure 5). If S_i contains n consecutive windows, then there are $w_w + (n-1) * s_w$ panes in S_i (right-hand side of Figure 5). Thus, the following should hold:

$$w_w + (n-1) * s_w \geq 2 * (w_w - s_w) \implies n \geq \frac{w_w - s_w}{s_w}$$

Meanwhile, the value of n should be chosen as small as possible for maximizing result sharing across nodes for each round of communication in the ring. Therefore, we choose as $n = \frac{w_w - s_w}{s_w}$. When each node is responsible for n consecutive windows, then each node should receive $w_w - (w_w - s_w) = s_w$ panes from the split node.

Let us illustrate the above on the example of Figure 4. Here, $w_w = 3$ and $s_w = 2$, and each node should be assigned 1 window and 2 panes. This way, each node shares 1 result with its next neighbor, which is different from the 1 result it receives from its previous neighbor. Note that, if n was chosen larger, then some pane results would not be shared across the nodes and there would be more local window computation on the nodes. This would mean less result sharing as well as coarser-granularity parallelism, which would lead to smaller overall query throughput (possibly incurring higher communication costs as well [1]).

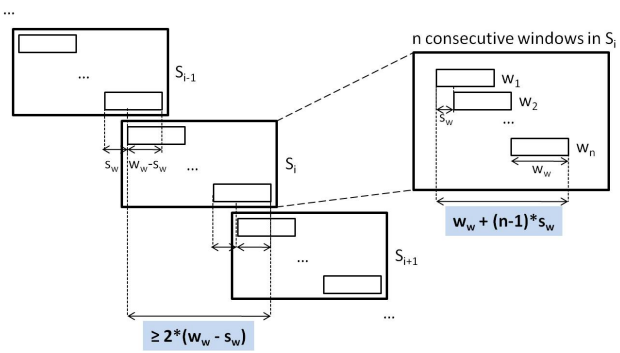


Figure 5: Set of windows for a node

2.5 Result Merging

Results from parallel executions of all partitions are combined via a final merge stage. If the merge emits results simply in the order that they arrive at the merge stage, ordered delivery is not guaranteed. In other words, it may happen that results of a window with window-id i reaches merge after a window with window-id $i+1$, if the latter partition is processed faster. On the other hand, strictly guaranteeing order would require synchronization and result buffering at the merge stage, potentially leading to blocking of result delivery and reduced throughput.

In reality, there are many applications where certain degree of disorder can be tolerated. This flexibility can be exploited in providing a more efficient merge stage. To capture this, we model disorder as a QoS dimension and allow the application to define the maximum degree of disorder that it can tolerate. Then merge ensures that the results are delivered within those limits.

We define the amount of disorder as follows, based on the k-ordering constraint defined in related work [3]: Assume a stream s which has an ordered-arrival constraint on an attribute $s.A$. Then for any tuple s in stream S, all S tuples that arrive at least $k+1$ tuples after s have an A value $\geq s.A$. That is, any two tuples that arrive out of order are within k tuples of each other. $k = 0$ captures the case where there is no disorder.

3. EXPERIMENTAL EVALUATION

In this section, we evaluate our proposed data partitioning techniques for parallel processing of sliding window queries. The goal of this experimental study is to show and compare the scalability of different techniques.

3.1 Experimental Setup

We have implemented our partitioned parallelism techniques as an extension to the Borealis distributed stream processing system [1]. We enriched the query execution framework of Borealis by integrating our techniques for data partitioned evaluation of window-based aggregate queries. More specifically, window-based aggregation query specifications are automatically transformed into a query plan with split, aggregate, and merge operators.

All the experiments were conducted on a shared-nothing cluster of machines, where each machine has an Intel® Xeon® L5520 2.26

[1]We are currently working on a cost model to account for all costs involved, including communication.

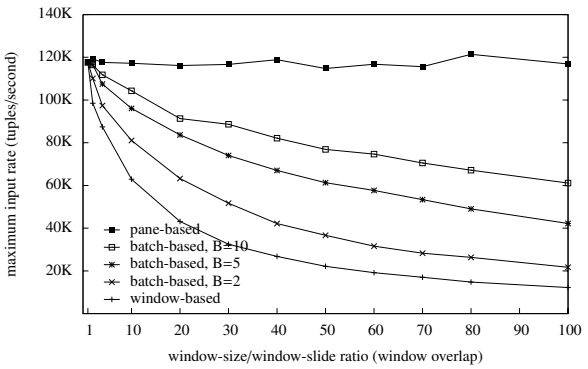


Figure 6: Throughput capacity of the split operator

Ghz Quadcore CPU and 16GB of main memory. The nodes each running Debian Linux are connected by a Gigabit Ethernet.

The workload that we used for our experiments is taken from the Linear Road Benchmark (LRB) [2]. LRB is a traffic simulation system of fictional linear highways where tolls are calculated dynamically depending on the variable conditions such as traffic congestion and accidents. The input data consists of stream of position reports and queries from the simulated vehicles in the highways.

Aggregation queries are used for computing segment statistics in LRB and are some of the most CPU intensive parts of the benchmark. However, window specifications are either tumbling over a minute interval or sliding with limited range and slide. To be able to understand the sensitivity of our techniques to different query properties, we experimented with a set of LRB-like aggregation queries with a wider range of window specifications. Our queries evaluate `count()`, `min()`, `max()`, and `avg()` over position reports similar to the segment statistics in LRB. We change the window specification from experiment to experiment to vary window-size/slide ratio where specified, and keep it fixed otherwise.

3.2 Scalability of the General Framework

In this first set of experiments, we assess the performance of the newly introduced operators in our general split-merge framework and outline their scalability limits. Due to space constraints, we discuss the split that implements our core data partitioning techniques in more detail, and briefly summarize findings about the others.

The Split Operator. Scalability of the split operator is critical, since it enables the parallel query evaluation in the first place as the core of our partitioning techniques are implemented inside the split operator logic. Therefore, efficient partitioning of streams is important for our system's overall scalability.

We used basic sliding window partitioning, batch partitioning, and pane partitioning in order to understand maximum throughput capacity of the split operator for each of them. For batch partitioning, we used 3 different batch-sizes: 2, 5, and 10. We also varied window-size/slide ratio in order to understand how the amount of overlap among consecutive windows affect the cost of split and correspondingly its throughput. In all of the experiments, we gradually increased the input rate until either the split node or one of the aggregate nodes became saturated (i.e., CPU load above 95%).

The result can be seen in Figure 6. First, throughput of pane-based partitioning does not depend on window overlap and stays at a somewhat constant rate. Second, the cost of split operator increases linearly with increasing amount of window overlap in other partitioning techniques, and accordingly the throughput degrades. Basic sliding window partitioning incurs the highest split overhead. The main reason for this is the excessive replication of the tuples for each window that they belong to. Batching technique brings some improvement over basic partitioning, although cost still increases linearly but more slowly with increasing batch-size (B).

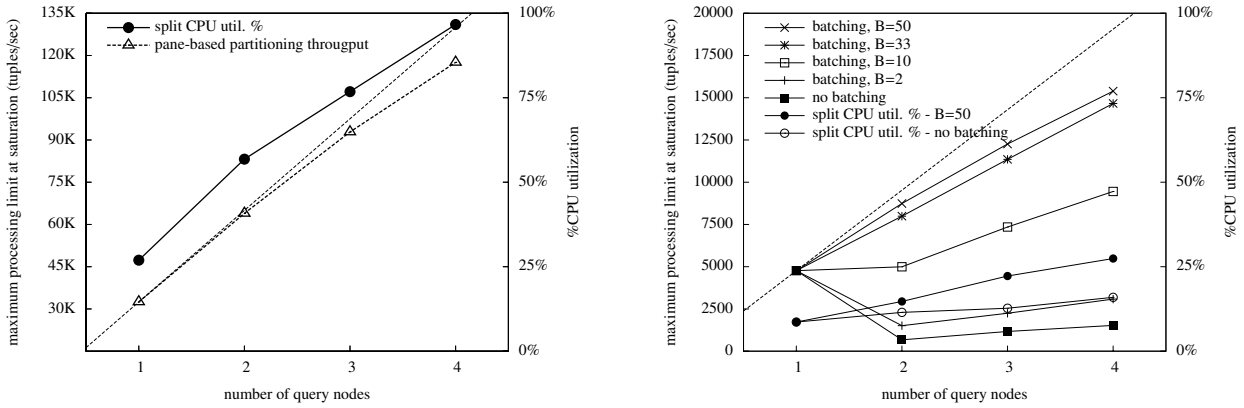
The Merge Operator. We can summarize our experimental findings about the behavior of merge operator as follows. As a general observation, merge is a cheap operation as results arrive almost ordered and input rates are low due to aggregation from the previous stage. In Borealis, tuples are processed one-by-one under low input rates whereas with normal to high input rates there is an opportunity to process a batch of tuples within a single scheduling step. Due to this reason, when the number of input streams of merge and the rates increase, per tuple processing cost of merge gets slightly cheaper. This is directly observed for the unordered delivery case. However, in the case of ordered delivery, execution is more complex and it dominates the cost. Therefore, the merge cost increases by a small margin. Furthermore, latency increases as a direct result of increased buffering for ensuring ordered delivery.

The Aggregate Operator. Our experimental results verify that the cost of the aggregate operator in case of batch-based partitioning is in line with the cost model described in Section 2, and clearly decreases as the batch-size decreases. According to the model, the input rate arriving to an aggregate operator increases with increasing B and the cost of aggregate decreases with decreasing B. As our model suggested, a balance between these two extremes should be found for a given query.

3.3 Scalability of the Partitioning Techniques

In the following set of experiments, we examined the scale-up properties of our partitioning techniques. We varied the number of nodes and kept the window-size/window-slide ratio fixed at 100. In each run, the system was fed with an increasing workload until the query nodes became saturated. Maximum achieved throughput was recorded for each run, and finally the maximum throughput was computed by taking an average over several runs. Figure 7(a) shows the result of our experiment with pane-based partitioning along with the CPU utilization of the split operator at each corresponding run. In pane-based partitioning, per-tuple cost of aggregate stays constant and it enables processing up to 32K tuples/sec per query node instance. As a result, pane-based partitioning scales up close to linear as long as the split operator is capable of feeding the query instances. A slight deviation from linearity starts when the CPU utilization of the split node gets closer to 100%. Despite this slight deviation, pane-based partitioning scales quite well with increasing number of nodes.

In Figure 7(b), the same set of experiments were repeated for basic window and batch partitioning. In basic window partitioning with 1 node, cost of aggregate operator is very high and the aggregate node only handles up to 5K tuples/sec. In this case, tuples are not replicated and rate at aggregate node is same as the actual input rate. However, when we switch to a 2-node case, split replicates each tuple 100 times. Even though the aggregate is capable of handling up to 40K tuples/sec, due to excessive replication, maximum throughput is limited by 800 tuples/sec (800*100/2=40K). As shown in the figure, split node is underutilized at 7% CPU load. In 3-node and 4-node cases, the same arguments are valid and throughput increases at a rate of approximately 400 tuples/sec per added query node. Since the CPU utilization of split node is below 12% with 4 nodes, the system can continue to scale beyond 4 nodes somewhat at a slower rate of 400 tuples/sec per node.



(a) Scale-up of pane-based partitioning

(b) Scale-up of sliding window partitioning

Figure 7: Scalability of the partitioning techniques

Figure 7(b) also shows the scale-up property of batch-based partitioning with different batch size parameters. Batch-based partitioning is obviously a better alternative to basic sliding window partitioning. Especially, batch-based partitioning with increasing batch size scales close to linear. However, the performance increase is slower as batch size gets bigger. For instance from $B = 33$ to $B = 50$, throughput increase is only about 1200 tuples/sec, whereas from $B = 2$ to $B = 10$ it is about 6500 tuples/sec. Evidently, split operator is not a bottleneck for this experiment. Maximum CPU usage in batch-based partitioning with $B = 50$ is 27%.

4. RELATED WORK SUMMARY

In this work, we leveraged the Window-id and Panes works of Li et al in a distributed setting to achieve finer-grained partitioning for sliding windows [8, 9].

Our work mainly relates to the recent previous work in parallel stream processing. There are two main categories of approaches to parallel stream processing: query partitioning and data partitioning. In query partitioning, processing is partitioned onto multiple nodes, which provides inter-operator/inter-query parallelism (e.g., [11, 12]). Load balancing / adaptivity requires moving operators and state across nodes and the granularity of partitioning is only at operator level. In data partitioning, input streams are split into disjoint partitions, where each partition is processed by a replica of the query in a parallel fashion. In one of the earlier works in this area, Flux generalizes the Exchange and RiverDQ approaches of traditional parallel databases to provide online repartitioning of content-sensitive streaming operators such as group-by aggregates [10]. Flux focuses on providing adaptive load balancing and fault tolerance. Ivanova et al instead focused on data partitioning for content-insensitive streaming operators such as windowed aggregates [6]. This work assumes tumbling windows and an order-preserving merge stage at the output. More recently, Johnson et al have proposed a data partitioning strategy for multiple continuous queries with different group-by and join attributes [7].

Our work falls under the data partitioning category and differs from the previous work in the following ways. First, we directly exploit how queries themselves originally partition the data in determining our split strategies. In the case of sliding window queries, this requires us to consider the potential overlap among the windows, which has not been dealt with in any previous work. Second, our focus is not on explicit load balancing. Third, we allow applications to define QoS in terms of latency and ordered delivery.

5. CONCLUSIONS

In this paper, we presented and experimentally compared a set of data partitioning techniques for parallel sliding window processing over data streams. The key idea of our approach is that data streams must be partitioned on the fly in a way that respects the natural windowing and order-sensitivity of continuous queries. We have shown that this can be achieved in several alternative ways, out of which, the pane-based partitioning has proven to have a scale-up behavior that is almost linear. It also incurs less overhead on the split operator and on the aggregate query itself.

We continue to build on this work along several directions. The currently ongoing work includes generalizing our parallelization framework to multiple input sources possibly with rate skew, adding support for adaptivity, and extending the complexity of the query plans (e.g., multiple queries with sharing, holistic aggregates, joins). We are also extending our performance analysis and experimentation framework accordingly. In the longer term, we also plan to tackle fault-tolerance issues.

6. REFERENCES

[1] D. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR Conference*, Asilomar, CA, January 2005.

[2] A. Arasu et al. Linear Road: A Stream Data Management Benchmark. In *VLDB Conference*, Toronto, Canada, August 2004.

[3] S. Babu et al. Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams. *ACM TODS*, 29(3), September 2004.

[4] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6), 1992.

[5] L. Golab and M. T. Özsu. Issues in Data Stream Management. *ACM SIGMOD Record*, 32(2), June 2003.

[6] M. Ivanova and T. Risch. Customizable Parallel Execution of Scientific Stream Queries. In *VLDB Conference*, Trondheim, Norway, August 2005.

[7] T. Johnson et al. Query-aware Partitioning for Monitoring Massive Network Data Streams. In *ACM SIGMOD Conference*, Vancouver, Canada, June 2008.

[8] J. Li et al. No Pane, No Gain: Efficient Evaluation of Sliding Window Aggregates over Data Streams. *ACM SIGMOD Record*, 34(1), March 2005.

[9] J. Li et al. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *ACM SIGMOD Conference*, Baltimore, MD, USA, June 2005.

[10] M. A. Shah. *Flux: A Mechanism for Building Robust, Scalable Dataflows*. PhD thesis, U.C. Berkeley, 2004.

[11] Y. Xing et al. Dynamic Load Distribution in the Borealis Stream Processor. In *IEEE ICDE Conference*, Tokyo, Japan, April 2005.

[12] Y. Xing et al. Providing Resiliency to Load Variations in Distributed Stream Processing. In *VLDB Conference*, Seoul, Korea, September 2006.