Scalable Stream Join Processing with Expensive Predicates: Workload Distribution and Adaptation by Time-Slicing*

Song Wang†
Hewlett-Packard Laboratories
songw@hp.com

Elke Rundensteiner
Worcester Polytechnic Institute
Worcester, MA, USA
rundenst@cs.wpi.edu

ABSTRACT

Multi-way stream joins with expensive join predicates lead to great challenge for real-time (or close to real-time) stream processing. Given the memory- and CPU-intensive nature of such stream join queries, scalable processing on a cluster must be employed. This paper proposes a novel scheme for distributed processing of generic multi-way joins with window constraints, called Pipelined State Partitioning (PSP). We target generic joins with arbitrarily join conditions, which are used in non-trivial stream applications such as image matching and biometric recognizing. The PSP scheme partitions the states into disjoint slices in the time domain, and then distributes the fine-grained states in the cluster, forming a virtual computation ring. Compared to replication-based distribution of non-equi-joins, PSP scheme is superior since: (1) zero state duplication and thus no repeated computations, (2) pipelined processing of every input tuple on multiple nodes to achieve low response time, and (3) cost-based adaptive workload distribution. We have implemented the proposed PSP schemes within the CAPE DSMS. Our experimental study demonstrates the significant performance improvements compared to the state-of-the-art generic distributed stream join algorithms.

1. INTRODUCTION

Recent years have witnessed an increasing interest in data stream management systems (DSMS). Stream applications such as scientific sensor network infrastructures, require processing high-volume streams in a timely manner. The data streams can include text data, multimedia data and other complex objects. Multi-way window-based Join operations (MJ) are commonly used to explore the correlation among multiple stream tuples in scientific and engineering domains [3, 10, 13, 15]. For example, environmental monitoring systems use streams from sensors for possibly complex pattern matching methodologies [3, 15]. Network monitoring systems use deep packet inspection queries to evaluate network traffic flows with content-based analysis methods [13]. These multi-way joins usually have complex and thus expensive joins. We first present three applications below with generic MJs over streams.

Example 1: *In a real-time business intelligence framework, analytic models may be continuously validated against incoming business data to produce short-term predictions. Online evaluation of the effectiveness of analytic models is essential for adaptively choosing the best one from a set of candidates at runtime. One simple metric could be the count of close predictions (which is defined by a user defined function (UDF) that calculates the distance of the predicted value and the real value) within a sliding window. When each vector of parameters, either predicted or real, is treated as a stream tuple, a window-based MJ operator with a UDF, i.e. the join predicate, and a count can be used to calculate the metric. When the vector is long, the number of tuples in the window is large and the UDF is complex, the evaluation of the metric can be very time consuming.*

Example 2: *Multimedia objects, such as camera images of sensitive areas, can be collected in real-time as stream tuples. One common operation for monitoring is calculating the correlation of two or more such image streams to find matching objects. Such correlation can be captured by a similarity matrix with distances between any pair of images. A window-based MJ operator with a UDF, which calculates the difference between two images, can be used to achieve this. When the image is large, the UDF (e.g. pixel based matching algorithm) can be very expensive to evaluate.*

Example 3: *On the Internet, files with similar content yet with distinct descriptions and meta-data are frequent. Some examples are: forwarded identical hot posts/news with different titles and authors, redundant files with different names sent from malicious clients, duplicated copies of hot downloads with different URLs. A publish/subscribe system that wants to detect and then remove these duplications can employ a window-based MJ operator with a UDF, which compares two files, as the join predicate. Since the files could be large and the time interval (i.e. the window) might be large, such MJ operator can be resource intensive.*

These stateful MJs tend to dominate critical resources such as memory and CPU. When facing high-volume in-

*This work was partly supported by NSF under grants IIS 0414567, SGER 0633930 and CRI 0551584.

†Work done when studying in Worcester Polytechnic Institute.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the ACM. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permissions from the publisher, ACM.
EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.
Copyright 2009 ACM 978-1-60558-422-5/09/0003 ...$5.00

put streams, the in-memory processing may at times be beyond the capacity of a single machine [9]. Resorting to a shared-nothing cluster has been recognized as one of the most practical solutions [1], without violating either the result accuracy or the real-time response requirements.

In this paper, we focus on distributed processing of generic MJs with arbitrary join predicates, especially for MJs with expensive join probing cost over large window constraints.

The basic distribution techniques used in database systems can be classified as *pipelined* and *partitioned parallelism* [11]. By streaming the output of one operator into the next operator, the two operators can work in series, termed pipelined parallelism. By partitioning the input data among multiple processors, an operator can be instantiated as many independent instances each working on a part of the data, termed partitioned parallelism. However direct application of these distribution methods is not effective for distributing MJs with arbitrary join conditions.

For pipelined parallelism, the MJ operator must fit into a single machine, which is not always feasible with large window constraints and high volume input streams. Though, we could translate an MJ operator into a join tree composed of a sequence of smaller binary join operators, such method would lose the flexibility of join orderings, which is extremely needed for MJ processing in dynamic environments [22]. Moreover such join tree distribution will scale to at most $k-1$ machines for a k-way MJ operator, while machines available may be much more than k.

Partitioned parallelism only supports equi-joins, since it requires some hash function for disjoint partitioning of tuples. For non-equi-joins, value-based data partitioning cannot be applied without potentially huge data duplication [9], which may abuse memory and cause increased data shipping and processing costs. Moreover, data partitioning [16] assumes that every partition is small enough to be processed by one single machine. This assumption may not always be valid or could rapidly be violated at run-time, especially when processing skewed data.

Our Approach: We propose a novel MJ operator distribution scheme called Pipelined State Partitioning (PSP). PSP separates a macro MJ operator into a series of smaller MJ operators by time-slicing of the states. The sliced MJ operators are connected in a virtual ring architecture. It slices the states into disjoint slices in the time domain, and then distributes these fine-grained state slices among processing nodes in the cluster. Also, due to pipelining, the workload to process every input tuple is distributed to multiple nodes. Thus not only the throughput is increased, but also (which is more important) that the processing time of each input tuple is reduced with more processing power involved. Different from traditional plan-based pipelined parallelism, whose parallelism is bounded by the longest sequence of operators in the query plan, PSP instead enables the optimizer to split the MJ to any number of state-sliced MJ operators.

In this paper, a cost-based optimizer is developed to achieve the optimal state slicing and allocation in terms of query response latency. Runtime adaptive state relocation is also proposed for achieving load balancing and re-optimization in a fluctuating environment.

Compared to existing work on distributed generic MJ processing in [9], the PSP scheme has the following benefits: 1) no state duplication and thus no repeated computations during PSP distribution; 2) pipelined processing in a cluster to achieve not only high throughput but also low response time; 3) arbitrary number of sliced operators at the optimizer's will to achieve optimality with given statistics; and 4) controllable adaptive state partitioning and allocation in the time domain.

Our Contributions:

- We introduce the novel ring-based MJ time-slicing architecture with a proof of correctness.
- The basic PSP model and its extension are discussed in detail.
- The PSP ring is analytically evaluated and tuned based on a cost model.
- The runtime slice allocation and relocation are proposed for adaptive load diffusion.
- The proposed techniques are implemented and evaluated in the *CAPE* DSMS. Results of performance comparison of our proposed techniques with state-of-the-art strategies are reported, confirming the superiority of PSP.

Organization of Paper: The rest of the paper is organized as follows. Sections 2 and 3 define the problem tackled. Section 4 introduces the PSP distribution scheme. Sections 5 and 6 present the cost based analysis and cost based runtime adaptive optimization. Section 7 compares PSP with other generic join distribution schemes. Section 8 reports the experimental results. Section 9 contains related work while Section 10 concludes the paper.

2. PROBLEM DEFINITION

In this paper we tackle the problem of scalable processing of multi-way joins (MJ) having expensive join predicates (usually UDFs) in a share-nothing cluster with a Gigabit network. The objective is to minimize the response time and reduce state memory costs even when faced with high-volume streams with huge windows. We present our approach using time-based windows. Our techniques can be applied to count-based window as well.

We assume that the timestamps of stream tuples are globally ordered [6]. *Sliding windows* [4] define the scope of the otherwise infinite streams for stateful operators. The output of an MJ on data streams $S_1, S_2, ..., S_m$ with window constraints $W_1, W_2, ..., W_m$ and join predicate θ consists of all joined tuples $(s_1, s_2, ..., s_m)$, such that $T - T_{s_i} < W_i$ ($\forall i \in [1, m]$) and $\theta(s_1,\, s_2, ...,\, s_m)$ hold [5]. Here T_{s_i} denotes the timestamp of tuple s_i and T denotes $max(T_{s_i})$,$i \in [1, m]$. The timestamp assigned to the joined tuple is T.

In [5, 22] the non-blocking multi-way symmetric join algorithms with flexible join orderings are introduced. Our proposed approach inherits this flexibility of customized join orderings to assure high performance. Clearly, selection of efficient join orderings is orthogonal to our focus, and any algorithms in [5, 22] could be used for this purpose.

3. STATE-SLICING FOR MULTI-WAY JOINS

Similar to slicing the window states along the time dimension for sharing the computation of aggregates [12] and binary joins [23], we now slice the states for pipelined execution of a multi-way join in the time domain.

Our work is related to the state-slicing concept presented in [23]. In [23], the state-slice concept is used for efficient *sharing* of *binary* join queries with different window lengths in a centralized query engine. The correctness of rewriting a binary sliding window join into a state-slice join chain is guaranteed by the pipelined execution of the input tuples. We propose to explore this property for distributed processing of MJs in a cluster, which is asynchronous in nature.

Applying state-slicing concept for MJ operator faces new challenges since the optimal join orderings must be preserved. Intuitively, by first converting an MJ operator into a binary join tree, the binary state-slice method can be reused in a naive way to process MJs. A binary join tree implies a fixed join ordering for all input stream tuples, which may be sub-optimal compared to the flexible join orderings in a holistic MJ operator [5]. Also, a binary join tree is rather rigid for runtime join ordering re-optimization, when the stream characteristics have changed.

Alternatively, directly applying the binary state-slicing method to MJ operators faces several problems, as explained below with an example. Assume that the MJ to be processed is a 4-way join $A \bowtie B \bowtie C \bowtie D$ and n state-sliced join operators, J_1 to J_n, are connected in a chain structure. The state for each stream in the MJ is partitioned into n parts, denoted as $(A_1, ...A_n)$, $(B_1, ..., B_n)$ and so on. Thus sliced join J_i $(1 \leq i \leq n)$ will hold the sliced states A_i, B_i, C_i and D_i. For one incoming tuple a from stream A, all the sub-join tasks $a \bowtie B_i \bowtie C_j \bowtie D_k$, $(1 \leq i, j, k \leq n)$ must be conducted to generate the complete joined results. Without loss of generalization, consider one sub-join task with $k \leq j \leq i$, then first $a \bowtie D_k$ must be conducted at J_k, since the sliced state D_k is held only at J_k, which is ahead of J_i and J_j in the chain. Similarly, $(a \bowtie D_k) \bowtie C_j$ then is conducted at J_j and so on. That is, the join ordering $(A \rightarrow D \rightarrow C \rightarrow B)$ is determined here by the ordering of the i, j, k in each sub-join task, but not by the plan optimizer. Actually every possible join ordering is used to process certain sub-join tasks, since the order of i, j, k is arbitrary. Such join orderings certainly are not optimal. The benefit of holistic MJ processing is lost and the join performance may be significantly decreased.

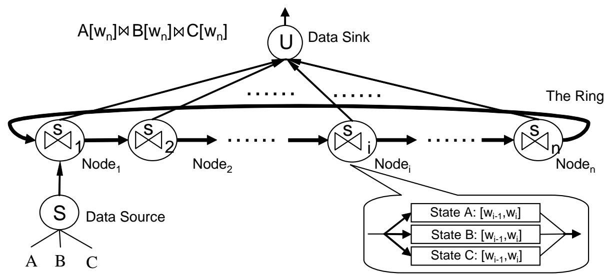


Figure 1: Ring-based Query Plan with Multi-way State-slice Joins.

To inherit the merit of holistic MJ processing with optimal join orderings, we propose a *ring-based query plan* execution framework for multi-way state sliced join processing. Fig. 1 shows the logical ring-based query plan with an example of n (3-way) state sliced joins. Intuitively, selection of the join orderings is independent from the locations of corresponding sliced states for a sub-join task, since the ring structure can bring tuples back to the entry. In the rest of this paper we will use M or m to denote the number of input streams and N or n to denote the number of sliced joins.

PSP is designed to distribute the potentially huge state of the MJ operator to all the processing nodes and consequently render balanced CPU load diffusion. The adaptive workload balance is achieved by dynamic setting the window ranges of the sliced joins at runtime. We discuss the runtime ring plan adaptation in Section 6.

4. THE PSP EXECUTION MODEL

Naturally the time-based operator partitioning provides a novel scheme for distributed processing of expensive join operators. The logical pipelined state partitioning (PSP) model is introduced here while cost-based deployment of the scheme in a cluster is discussed in Section 5.

4.1 State-Slice Ring with Life Control

The logic ring model of PSP in Figure 1 corresponds to a series of multi-way state-sliced join operators connected in a ring structure. Besides stream inputs and output of joined results, each state-sliced join also has special input and output for pipelined propagation of intermediate results.

Definition 1. *A multi-way state-slice join operator J^m on data streams $S_1, S_2, ..., S_m$ is denoted as $S_1[W_1^s, W_1^e] \overset{m}{\bowtie} S_2[W_2^s, W_2^e] \overset{m}{\bowtie} ... \overset{m}{\bowtie} S_m[W_m^s, W_m^e]$, where the superscripts s and e denote the start and end of the window constraints. The joined results of J^m for arrival tuple s_i consist of all tuples $(s_1, s_2, ..., s_m)$, such that $W_j^s \leq T_{s_i} - T_{s_j} < W_j^e$, for all $j \in [1, m]$, $j \neq i$.*

A pipelined state-slice join ring is composed of multiple state-slice join operators on the same data streams. The states of the connected joins have abutting window ranges for each input stream. The slice join having the $W^s = 0$ in the ring is called the *head* of the ring and the one having the largest end window is called the *tail* of the ring. Since the window ranges of sliced joins in the ring are non-overlapping, the whole states are partitioned disjointly.

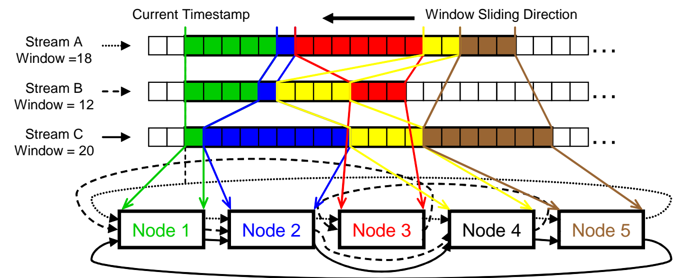


Figure 2: Snapshot of Runtime State Deployment in the Ring-based PSP.

Figure 2 shows a snapshot of the state partitioning and physical deployment for an example 3-way join on stream A, B and C in a 5-node cluster. Each stream has unique window size and the current sliding windows are illustrated with colors. The state of each stream is partitioned disjointly and deployed to nodes 1 to 5. The state deployment can be very flexible: the ring connections and thus the ring lengths could be different from each other.

The PSP execution model includes three closely coupled components to produce complete but not redundant join results, according to the semantics of the multi-way sliding window join. They are: (1) coordinated state maintenance among the sliced join operators in the ring to ensure state consistency; (2) propagation and processing of the intermediate results for correct and complete join results; (3) execution control to avoid infinite looping of tuples in the ring.

The join processing among multiple nodes needs coordination to ensure data consistency. However synchronization in a large cluster is expensive, especially when synchronization needs to be invoked for every input stream tuple in our case. In our PSP model, we make a special design that achieves implicit synchronization based on the FIFO network transmission model. The execution process of each node is solely driven by the tuples in the input network connection. Thus each node can run independently without other synchronization mechanism.

We first discuss the coordination among multiple nodes for processing single input stream tuple. Then we will show how the head node "knows" the end of processing cycle for each current input tuple. Thus the sequential processing of the input tuples can be synchronized and pipelined in the ring. PSP with interleaved processing is discussed later in Section 4.4.

Coordinated State Maintenance.

In [23] two representatives for each input tuple, *build tuple* and *probe tuple*, are used with distinct assigned responsibilities. Build tuples will be inserted into the states of the join operators and will stay there until being purged. The probe tuple instead will be propagated for probing corresponding states. In the ring structure, the build tuples will "move" from the head node towards the tail node steadily. The corresponding probe tuple will be used for state purging and join probing. The following example shows how each node processes state insertion and purging.

Example: Suppose at time t, tuple a_t arrives and then two copies a_t^b (build) and a_t^p (probe) are made. Tuple a_t^b is inserted into the current state (a_t^b, a_j^b, ..., a_{i+1}^b, a_i^b,a_{i-1}^b), ordered decreasingly by the timestamps. Suppose a_i^b and a_{i-1}^b are the only tuples out of the current window range, the state will be (a_t^b, a_j^b, ..., a_{i+1}^b) after purging by a_t^p and the output queue now is (a_t^p,a_i^b,a_{i-1}^b). Consequently, a_{i-1}^b and a_i^b will then be inserted into the state of the next join operator and then a_t^p will purge and probe them in the next node.

PROPERTY 1. *For any node with a state sliced window $[W_i^s, W_i^e]$ on input stream i, before a probing tuple p with timestamp T_p is processed at this node, all build tuples from stream i with timestamp T_i satisfying $W_i^s \le T_p - T_i$ have been inserted into the state of stream i at this node. After purging by tuple p, only tuples with timestamp $W_i^s \le T_p - T_i < W_i^e$ stay in the state.*

The proof of Property 1 is straight-forward since the probing tuple is placed behind all purged tuples in the input queue. Property 1 is guaranteed with the FIFO property of the network connections between processing nodes. The state maintenance at each node is coordinated with every probing tuple. Thus even the state maintenance processes will not happen at the same time in all the nodes, the state can still be consistent in terms of join probing process after finishing of the state purging by the probing tuple.

Coordinated state maintenance achieves maximum loose synchronization in the cluster. That is, the state synchronization is postponed as long as possible until right before the join probing process. Also this coordination involves no extra network messages since the probing tuples have to be propagated for join probing purposes anyway. Compared to brute force broadcasting, the probing tuples are propagated step-by-step along the ring. Extra delay comes from the purging of the state tuples in each node. However since the states are in memory and sorted (or indexed) by timestamps, such cost is very small.

Intermediate Result Propagation and Processing.

Intermediate results are propagated along the ring to probe the next state in the join orderings. We use only one representative, the probing tuple, for the intermediate result tuples since there is no state holding intermediate result.

For an M-way sliding join, the number of types of possible intermediate results is $O(2^M)$. We cannot afford a distinct network connection for each type of intermediate result when M is large. Instead, all intermediate results are transmitted in one connection and the intermediate result schema (piggy-backed) is used to identify the types. Also the schema is used to determine the next state to join with the intermediate result.

An intermediate join result schema is denoted as $(I_1, I_2, ..., I_M)$, where I_i can be a stream "S_i" or null "$-$". Given a set of optimal join orderings, exactly one state exists for each input tuple and intermediate result to probe. We show how the intermediate result is propagated and how the joined result is generated using an example below.

Example: Consider a 4-way join $A \bowtie B \bowtie C \bowtie D$ with the join ordering $C \to B \to A \to D$ for the tuples from stream C. The join result (a_i,b_j,c,d_k), where i, j, and k (without loss of generality, assume $i < j < k$) denotes the serial number of nodes (1 to N) in the ring holding the corresponding state, is formed as follows. Tuple c is propagated to J_j first to probe the B state, and it generates the intermediate result ($-$,b_j,c,$-$). Then the intermediate result is propagated along the ring to join with all the sliced A states. Thus (a_i,b_j,c,$-$) is generated when the intermediate result reaches J_i. Since $i < j$, this propagation passes through the ring head. Then the newly generated intermediate result will be propagated to J_k to finish the join probing. No passing through the ring head is needed since $i < k$.

In the worst case, $(M-1)N$ hops of intermediate propagations are needed for an M-way join evaluation using a sliced join ring of length N. This, potentially causing long response time, motivates the cost-based PSP optimization (see Section 5).

Life Span Control in the Ring.

The purpose of life span control is: (1) dropping the input stream tuples and intermediate result tuples out of the ring at the right time to avoid generating incomplete or redundant join results; and (2) identifying the end of the processing cycle of current input stream tuple at the head node.

Since at any time the states are disjunctively sliced among the processing nodes, then each probing tuple (either input stream tuple or intermediate result) needs to be propagated along the ring exactly one round. To achieve this, every sliced join operator assigns its unique node ID to the intermediate result tuples it generates. When the tuple reached the same operator again after one round propagation along the ring, the tuple is dropped from the system.

Since the processing of the next stream tuple at the head node may cause state shifting along the ring (see 4.4 for interleaved PSP), to ensure the correctness the head node needs to know when the current processing at all nodes is finished. We design a special scheme to synchronize the sequential processing of stream tuples in the cluster.

A special *END* flag is used to mark the last intermediate result tuple with a certain schema generated at the head node. Initially, the END flag is set on the last intermediate result tuple generated by probing with the input stream tuple. A future END flag is set on the last intermediate result tuple generated by probing the previous END tuple. The previous END tuple is dropped according to life span control. Thus at any time, there is one and only one END tuple in the ring. Refer to Figure 3 for detailed steps.

PROPERTY 2. *The END tuple of a certain schema Sch_i is the last intermediate result tuple processed at the head node having the schema Sch_i.*

Proof: Proof by induction.

(1)Base Case: Without loss of generality, assume state of stream S is the first one in the join ordering for input tuple t. The state of S is sliced and distributed in the PSP ring as $S_1, S_2, ..., S_n$. The first END tuple e_1 is the last tuple in the output of $t \bowtie S_1$. At any node i ($1 < i \leq n$) in the ring, tuple t is processed before e_1, since propagation of t is before probing with t at all the nodes. Thus $t \bowtie S_i$ (if any) will appear before e_1 in the input queue of the head node.

(2) Induction: Since e_j is the last one processed by the head node having a certain schema, the next END tuple e_{j+1} will be processed after all other intermediate result tuples of the same schema as e_{j+1}. This is ensured by the FIFO processing sequence along the ring.

■

When the head node sees this END tuple again, it knows all the intermediate results with this schema have been processed by all the nodes. Thus we have:

THEOREM 1. *For each input stream tuple, the processing cycle is ended by the processing of its $(M-1)^{th}$ end tuple at the head node for an M-way join.*

Using Property 2, Theorem 1 can be proved directly.

Broadcasting intermediate results to all the nodes at the same time would be easy but needs synchronization among nodes. Instead our pipelined propagation of the intermediate result guarantees synchronization without extra message processing.

4.2 Execution Algorithm and Time Line

The sliced join execution algorithm is composed of four primitives: insert, cross-purge, propagation and probe, denoted as $in(state)$, $cp(state)$, $pg(op)$ and $pb(state)$ respectively. In an m-way sliced join of streams S^1, S^2, ..., S^m, the execution steps for a newly arriving tuple t in sliced join number op_i are shown in Fig. 3. We define the ID of the intermediate result generated by op_i as the number i.

Fig. 4 illustrates the execution time line in a four node cluster (each node holding one sliced join operator) for a 3-way join operator processing arrived A tuple. The accumulated input queue content is also shown for node M_2 and M_3. Assume the optimal join ordering for A tuples is $A \rightarrow B \rightarrow C$. When tuple a arrives at node M_1 at time 0,

If the tuple t is a build tuple from stream S^j,
1-1. *Insert:* $t.in(S^j_i)$.

If the tuple t is a probe tuple from stream S^j,
2-1. *Purge:* $t.cp(S^l_i)$, $1 \leq l \leq m$. If op_i is the tail op, drop purged tuples; otherwise propagate purged tuples to op_{i+1}.
2-2. *Propagate:* If op_i is the tail op, drop t; otherwise $t.pg(op_{i+1})$.
2-3. *Probe:* $I_i = t.pb(S^l_i)$, S^l_i is the state of the next stream in the given join ordering. I_i is the intermediate result with ID i. For head node, mark the last tuple in the intermediate result as the *END* tuple (If the probing has no output, a Null *END* tuple is generated).
2-4. *Propagate:* Send I_i with $I_i.pg(op_{i+1})$.

If the tuple t is an intermediate result tuple,
3-1. *Propagate:* If $ID \neq i$, $t.pg(op_{i+1})$, otherwise drop t.
3-2. *Probe:* $I_i = t.pb(S^l_i)$, S^l_i is the state of the next stream in the given join ordering. I_i is the intermediate result with ID i. For head node, if tuple t is marked as the *END* tuple and $ID = i$, mark the last output tuple as the new *END* tuple (If the probing has no output, a Null *END* tuple is generated).
3-3. *Propagate:* If I_i is final joined result, send out. Otherwise send I_i with $I_i.pg(op_{i+1})$.

Figure 3: Execution Steps of Sliced Join op_i

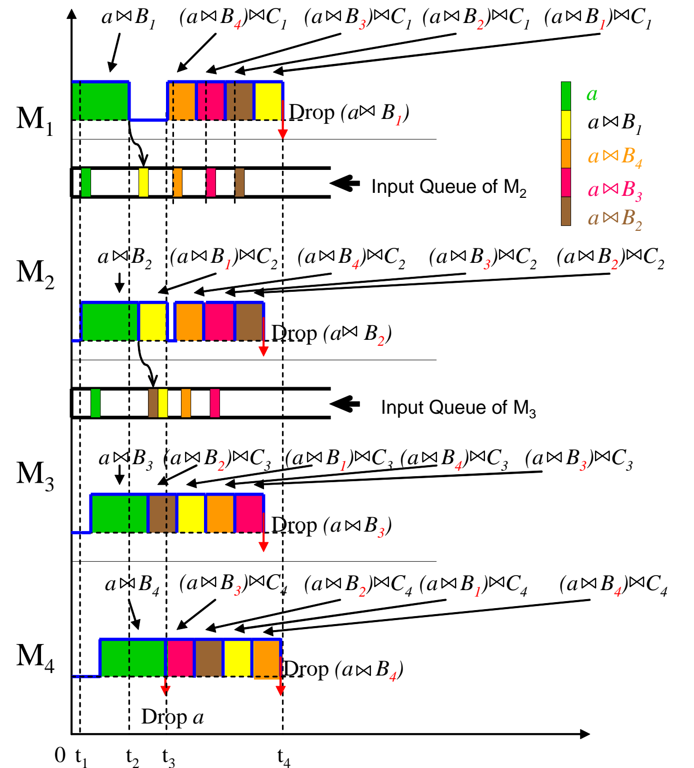


Figure 4: Execution Time Line of the PSP

first a build tuple a^b is made and $a^b.in(S^A_1)$ is called. Then the probe tuple a^p is used to purge and probe state B, i.e. $a^p.cp(S^B_1)$, $(a^p.cp(S^B_1)).pg(M_2)$ and $a^p.pb(S^B_1)$ (done at time t_2). The intermediate result is send to M_2 to join with the states of C and eventually sent back to M_1 to join with S^C_1. M_2 will receive the probe tuple of a at time t_1 and follow the same execution steps as M_1. The intermediate result I_1 will arrive at M_2 following tuple a and will be processed next by M_2. Then same steps are followed by M_3 and M_4. At time

t_4, one period of execution is finished. All the probe tuples and intermediate results are dropped after going through the ring. The next input stream tuple can be processed after t_4.

THEOREM 2. *The union of the join results of the sliced joins in the PSP ring is equivalent to the results of a regular multi-way sliding window join.*

Proof Sketch:
No missing results. From Property 1, the state slices are maintained consistently before any join probing. From Theorem 1, all state slices are probed before the end of processing cycle. The pipelined probing after a full round is guaranteed to cover all the corresponding states need to be probed.
No extra results. Before any join probing, all the windows boundaries of the states are maintained consistently. All the probings are valid.
No redundant results. Property 1 guarantees that the window boundaries of the states are satisfied before probing. Thus the states are maintained disjointly in terms of probing. No redundant probing is conducted.

4.3 Network Cost Estimation of PSP

Compared to the centralized execution of an MJ operators, our PSP may suffer from potentially huge network transmission costs. Here, we will describe how the total network cost is.

For simplicity, we first assume the join selectivities are the same for all join probings, denoted as $S_{\bowtie}$. And the window sizes are the same for all the streams, denoted as W. For an M-way join processed on a PSP ring, the total number of intermediate results with unit stream arrival rate is $S_{\bowtie}W + S_{\bowtie}^2W^2 + ... + S_{\bowtie}^{M-1}W^{M-1} \approx S_{\bowtie}^{M-1}W^{M-1}$ for large $S_{\bowtie}W$.

Having N hops in the PSP scheme, the total network cost for intermediate result transmission is $S_{\bowtie}^{M-1}W^{M-1}N$. In practice, since M is usually small, the network cost may not be huge. Put differently, the size of the join results that have to be transmitted to the end application in any case, is $O(S_{\bowtie}^{M}W^{M})$. That is, the extra network traffic caused by PSP for the intermediate result transmission is $\frac{N}{S_{\bowtie}W}$ times as that for transmitting final join results alone, which is small for large $S_{\bowtie}W$. When the join selectivities are dramatically different from each other, our scheme will employ the optimal join ordering to minimize the size of intermediate results. The above claim still holds.

However, the extra network cost is high when $S_{\bowtie}W \ll N$. There is a major tradeoff in the PSP scheme considering the length of the ring and the extra network cost. The optimal length of the ring is discussed in Section 5.

Also the network transmission time in the PSP scheme will overlap with the concurrently processing time on the nodes (see Fig. 4), which further reduces the impact of network latency.

4.4 PSP with Interleaved Processing

In the PSP scheme shown in Fig. 3, interleaved tuple processing is prohibited to assure consistent maintenance of the states in the basic PSP model. Since the performance of the ring is determined by its slowest processing node, this may cause long idle periods. We thus extend PSP by means of a delayed purge strategy, called PSP-I.

The processing of the next stream input tuple will cause insertion and purging of the states. To avoid state mess-up, every processing node maintains a list of active *StateStart*s and *StateEnd*s pairs. Each pair marks the corresponding states for one of the currently being processed tuples in the system. Instead of purging the states and removing purged tuples before probing, the StateEnds are used to mark the ends of states. The real state purging is postponed until no further probing requires the state anymore.

Although the purged state tuples are not physically deleted from the current node (they are just virtually marked as expired in some sense for a given tuple), they are propagated to the next processing node. During the join probing, only the part of the state within the appropriate StateStarts and StateEnds range relevant to the given tuple is used to join with the incoming tuple. The StateStart and StateEnd pair is expired and removed from the state when the corresponding END tuple is processed, because the later indicates that the tuple has successfully already visited all its join partners. The purge step 2-1 in Fig. 3 is now changed as shown in Fig. 5.

2-1. *Purge:*
2-1-1. *Init:* Init a pair of StateStart and StateEnd.
2-1-2. *Mark:* Mark the states.
2-1-3. *Propagate:* Propagate purged tuples to op_{i+1} (not tail) or drop them (tail).
2-1-4. *Delete:* If the StateEnd has the smallest timestamp in the list, remove tuples older than the StateEnd.

Figure 5: Purge Steps with StateStart and StateEnd in $node_i$, $1 \leq i \leq n$

The interleaved processing of stream tuples induces some duplicated states among neighboring sliced operators due to the postponed deletion. In our implementation, we set a threshold to limit the number of concurrently processed stream tuples in the ring.

5. COST ANALYSIS AND TUNING

In this section, we develop a cost model for PSP and use it to tune the parameters for different performance objectives. Our cost model provides the necessary analytical equations to interrelate the following key parameters of the PSP model: (1) query parameters, including stream arrival rates, window sizes and join selectivities; (2) PSP ring parameter, such as number of nodes in the ring; (3) performance measurements, such as average response latency (average time difference between sending out the joined result and reading in corresponding stream tuple) and output rate.

5.1 Cost Model

For an M-way join operation $S_1 \bowtie S_2 \bowtie ... \bowtie S_M$, parameters for the cost model are given in Table 1.

We assume that the network bandwidth is sufficient here for our workload. More discussion of network bandwidth is postponed to Section 5.3. Then we estimate the sending and receiving latency between processing nodes to be proportional to the number of tuples transmitted. The output rate is estimated with the assumption that all the processing nodes are 100% busy during the execution. That is, the input queues of the head operator are never empty. The output rate under such assumption is the maximum output rate possible.

Table 1: Terms Used in Cost Model

Term	Meaning
λ_i	Arrival Rate of Stream i
W_i	Window Size of the Sliding Window on Stream i
$S_{\bowtie i}$	Join Selectivity for i^{th} probing step
T_j	Time spent to join a pair of tuples
T_p	Time spent to purge one tuple from a state
T_i	Time spent to insert one tuple into a state
T_s	Processing latency to send & receive one tuple
T_n	One hop network transmission latency
N	Number of processing nodes in the ring
M	Number of incoming streams
μ	Service rate of the PSP ring

We first calculate the processing workload L_C for the centralized join processing of one input tuple and the workload L_{PSP} for the processing of the same input tuple in the PSP scheme. The workload indicates the total time needed to process one input tuple. The workload can be calculated by summing up the CPU join time, state maintenance time, network transmission time. We assume an in-memory nested loop join algorithm is employed. We also assume the optimal join ordering for the input tuple from stream S^1 is: $S^1 -> S^2 -> ... -> S^m$ and the processing nodes and the network connections between them are homogeneous.

$$\begin{aligned} L_C &= T_i + T_p + T_j \sum_{2 \le k \le M} \prod_{1 \le i \le k-1} \lambda_i W_i S_{\bowtie i} \\ L_{PSP} &= L_C + T_s N (1 + \sum_{2 \le k \le M-1} \prod_{1 \le i \le k-1} \lambda_i W_i S_{\bowtie i}) \end{aligned} \quad (1)$$

The third item for L_C is the total join probing cost. The second item for L_{PSP} is the total transmission cost for the input tuple and the intermediate results.

For succinctness of the analysis, we simplify the cost model by assuming $\lambda_i = \lambda_j$, $S_{\bowtie i} = S_{\bowtie j}$ and $W_i = W_j$. These assumptions can be relaxed without changing the principles of the cost analysis. Thus:

$$L_{PSP} \approx L_C (1 + \frac{T_s N}{\lambda W T_j}) \quad (2)$$

Every L_{PSP} seconds, PSP processes one input stream tuple. Thus the service rate μ (i.e. the number of tuples processed per second) is given as:

$$\mu = \frac{1}{L_{PSP}} \quad (3)$$

5.2 Cost-based Tuning

Based on the cost model, we perform PSP optimization for the following two important objectives: (1) given a fixed stream arrival rate, maximize the system output rate by tuning the length of the ring; and (2) given a fixed stream arrival rate, minimize the average response latency by tuning the length of the ring. In the following discussion, we assume that we have knowledge of the stream parameters and query parameters (i.e. all terms in Table 1 except μ). All these parameters are straightforward to measure in an actual implementation of PSP.

Maximize Output Rate.

In a homogeneous cluster, all processing nodes have identical CPU power. Assume the output rate for one single processing node with workload L_C is O_S. Then the output rate O_{PSP} in the PSP model with N processing nodes is:

$$O_{PSP} = \frac{O_S N}{1 + \frac{T_s N}{\lambda W T_j}} = \frac{O_S}{\frac{1}{N} + \frac{T_s}{\lambda W T_j}} \quad (4)$$

From Equation 4, as more processing nodes are deployed in the PSP ring, the output rate increases monotonically. That is, using more processing nodes will result in higher output rate.

Minimize Average Response Latency.

To estimate the processing latency of the PSP model, we consider the average latency for join results from one input tuple. We estimate the latency assuming perfect load balancing among all processing nodes. That is there is no bottleneck processing node slowing down the flow along the ring. Such latency is the minimal latency achievable. The latency has mainly two parts: join probing and the network latencies. These two latencies overlap in time during execution. For each processing node, the balanced workload is L_{PSP}/N on average. The join results are generated after total of M rounds of transmission of the intermediate results along the ring. Thus processing latency τ_i for node $i, 1 \le i \le N$ is:

$$\tau_i = (i-1)T_n + \max\{\frac{L_{PSP}}{N}, MNT_n\}$$

Thus the average processing latency τ is:

$$\tau = \frac{N-1}{2} T_n + \max\{\frac{L_C}{N} + \frac{T_s L_C}{\lambda W T_j}, MNT_n\} \quad (5)$$

For clarity, we omit state insertion and deletion delay since they are one time cost for each input tuple. Such delay is independent of the number of join results generated.

From Equations 5, we see the response latency is sensitive to the number of the processing nodes N in the PSP ring. Intuitively, adding more processing nodes increases the CPU power. On the other hand, the longer the length of the ring the higher the network transmission cost. Using standard calculus methodology, we can find exactly the value of N that minimizes the average response latency.

The PSP ring has the minimal processing latency when $N = min\{N_1, N_2\}$, where

$$\frac{L_C}{N_1} + \frac{T_s L_C}{\lambda W T_j} = MN_1 T_n, \ and \ \frac{N_2 - 1}{2} T_n = \frac{L_C}{N_2}$$

The processing latency for each node is decreasing with larger N, while the network latency is increasing. Both facts need to be considered for the optimal ring length with minimal processing latency.

5.3 Network Bandwidth Requirement

Here we estimate the network bandwidth needed for the above cost-based tuning. Ideally the network should be "fast" enough to catch the speed of join processing in each node. The network transmission time is covered by the join probing time when every node never waits for the arrival of new stream tuples. That is, the network transmission time should be shorter than the processing time of one probing tuple. Thus the upper-bound network bandwidth $Band$ can be estimated as below.

$$Band \geq \frac{Tuple_Size}{T_j \min_{1 \leq i \leq M} \{\lambda_i \frac{W_i}{N}\}} \quad (6)$$

When insufficient network bandwidth is available, significant network transmission cost, which is proportional to the number of total intermediate results, will be "visible". In practice, Equation 6 should be kept satisfied.

5.4 Initial State Slicing

When the arrival rates and sliding window sizes are different for each input stream, naturally the optimal ring lengths would be different for individual stream. The problem of achieving global optimal lengths of rings is much harder than the simplified case discussed previously. In fact the search space is exponential since the optimal lengths of the PSP rings are correlated with each other. Thus searching for the optimal initial state slicing is expensive and may not be worthwhile, especially for stream processing in a high dynamic environment. Instead we use following heuristic to achieve a sub-optimal state slicing.

We first sort the streams by $\lambda_i W_i \overline{S_{\bowtie i}}$ in ascending order. Here the $\overline{S_{\bowtie i}}$ denotes the average join selectivity between stream i and other streams in the join graph. Then the optimal lengths of rings are calculated in the order of the sorted list of streams. In the calculation, if the length of the ring for certain stream is unknown, then current length is assigned. For example when calculating the length for stream i and the length for stream j is not available (i.e. stream j is behind stream i in the sorted list), then the ring length for stream j is assigned the same as stream i. The intuition behind this heuristic is that the streams with larger $\lambda_i W_i \overline{S_{\bowtie i}}$ in the sorted list have more impact on the total cost, and should be processed later with more information of other streams.

5.5 Workload Balancing

In Figure 2 we indicate that the deployment of state sliced windows may not be even among all the nodes. Since the PSP is a pipelined execution model, the performance of the PSP ring is determined by the busiest node in the ring. To avoid bottleneck node, workload balance must be achieved for optimal performance.

From Equation 1, the dominant CPU cost for each node is the join probing cost, which is proportional to the total size of the sliced states in the node. To balance the workload of each node, we instead balance the number of state tuples in every node. Since the state slicing boundaries between adjacent join nodes can be performed arbitrarily at optimizer's will, the balanced state distribution can be achieved.

6. ADAPTIVE PSP LOAD DIFFUSION

Adaptive workload diffusion is critical for realistic long running query processing, when stream arrival rates, join selectivities and etc. change at runtime. In PSP, adaptive workload diffusion is achieved by state relocation among the nodes by setting the corresponding window ranges. We tackle two major load re-balancing scenarios: *workload smoothing* among same amount of nodes and *state relocation* with more/less nodes. Both adaptations are rather straightforward and inexpensive to implement.

Workload Smoothing

The runtime stream arrival rates may always fluctuate, while the overall system is not overloaded. In a homogeneous cluster, we initially slice the time-based window ranges evenly among all the nodes, aiming for balanced workload. However fluctuating arrival rates will make the workload on each node unbalanced given fixed window ranges. System performance is slowed-down by the overloaded node in the ring. Here we propose that instead a count-based state slicing can be employed to smooth workload. Each sliced join, except the last one hold by the tail node, has an upper bound of state size and a count-based state purge is employed when the state size grows over the threshold. The upper bound is set periodically according to the given statistics. The correct semantics of the sliding window join is ensured by the tail node since it still uses time based state purging.

Such count-based workload smoothing is effective when the window constraint is large. For small window to be close to the statistic sampling intervals, the statistics may not be precise enough.

State Relocation

Adding/removing of nodes is needed when system is overloading or ring length is not optimal for response time. Two approaches for adaptive optimization are proposed: *passive adjustment* of the window range and *aggressive adjustment* by state relocation.

Passive adjustment aims to relocate the state by setting the window ranges. Consider an example of adding one node to a ring has 3 nodes. We assume the states are sliced equally among the processing nodes N_1 to N_3 (N_4 finally). That is, the states in each processing node will be changed from $W/3$ to $W/4$, with W denoting the window constraint. The state slices in the original processing nodes N_1 to N_3 are step-by-step replaced and shrunk. Finally the new state allocation with one additional processing node is achieved. Similarly, node removal can be conducted. The graceful state adjustment induces no extra migration cost. However a long adjustment latency may occur for large window size.

Aggressive state slice adjustment migrates part of the states along the PSP-D ring. Such state relocation needs to suspend the execution and resume afterward.

To maintain the ring structure, the state slice movement happens only between adjacent processing nodes. Intuitively, a new processing node should be inserted into the ring at the position where the shifted state slice can fill the new node. That is, assume the ring has N nodes originally and another M nodes need to be added into the ring, the $i-$th processing node from the head node need to move $\Delta S_i = \frac{M}{N}i - \lfloor \frac{M}{N}i \rfloor$ state tuples to the next nodes towards the tail node. The new processing node N_j, $1 \leq j \leq M$, needs to be inserted *after* the processing node N_k, such that k is the minimal number with $\frac{M}{N}k > j$. Fig. 6 illustrates the addition of a new processing nodes. Similarly, removal of processing node can be conducted.

The aggressive state relocation involves execution breaks and state migration during the adjustment. Frequent aggressive adjustment should be avoided.

7. DISCUSSION

[9] proposed two *state replication* based distributions for generic MJ operators: aligned tuple routing (ATR) and coordinated tuple routing (CTR). We briefly review these two approaches and compare them with PSP below.

ATR picks one input stream as the master stream and

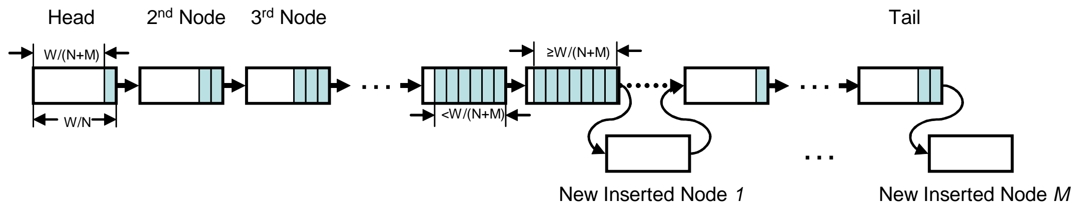


Figure 6: Aggressive State Relocation.

partitions the master stream among the processing nodes. All the other slave input streams are distributed to the processing nodes with some overlaps, to ensure the semantics of the window constraints.

CTR is a multi-hop semantics preserving tuple routing where intermediate join results are transferred among nodes during each hop. A weighted minimum set covering is utilized to identify the optimal routing for each tuple to "find" all correlated states.

Memory: The distribution strategies of both ATR and CTR are based on state (partial) duplication among the processing nodes. Compared to them, our proposed PSP approach does not have any duplicated states at any time.

In ATR the segment length T is an important parameter for the load diffusion. However, the ATR approach works under the condition that the window constraint $W \ll T$. When W is comparable with T, the memory waste and redundant computation can be significant.

In CTR the number of redundant state is determined by the minimum set covering at runtime. CTR faces the following dilemma. The more redundant states, the smaller set covering may exist. Then the incoming tuples will be stored in fewer nodes, which may make future set covering large. More seriously, the states in CTR may converge to one (or a small subset of) node if sometime only one copy of the input tuples is stored in the cluster, since the future set covering will direct all later tuples to that node. Then no distribution is achieved. Unless an optimal insertion algorithm is employed (missing in [9]), which can magically predict future workload diffusion, the CTR is uncontrollable and instable.

Synchronization: ATR results in a set of independent join operators without synchronization. However, CTR does need synchronization among nodes in different hops for maintenance of the states and processing of intermediate results. The synchronization is missed in [9].

CPU: CPU cost comparison is summarized in Fig. 7. Here we list only the main factors affecting CPU cost.

Item	ATR	CTR	PSP
Routing Cost	Low	High	Low
Per Segment Metadata	No	Yes	No
Duplication Removal	No	Yes	No
Load Balancing Granularity	Large	Small	Small
State Management Cost	High	High	Low
Adaptation Cost	Unknown	Unknown	Low
Network Transmission	Low	Middle	High

Figure 7: CPU Consumption Comparison

The CTR employs a complex routing algorithm to determine the optimal routes for each segment of input streams. Such routing cost is per segment cost and may be significant with fine-grained segments. On the contrary ATR and PSP do not require routing by using one hop computation and fixed routing respectively. The routing information and other metadata must be attached to each segment to ensure the correctness in CTR, while no such requirement exists for ATR and PSP-D. Similarly the CTR needs extra work to avoid generating duplicated results while the other two will not generate duplication in the first place. ATR and CTR approaches have duplicated states and thus the state management costs are much higher than PSP.

The disadvantage of PSP is that the network transmission cost may be larger than ATR and CTR, since all input tuples and intermediate results need to be sent along the ring.

8. PERFORMANCE STUDY

8.1 Experimental Settings

Distributed Join Algorithms. We have implemented the proposed PSP in a real distributed DSMS system. Experiments have been conducted to thoroughly test the ability of the proposed solution under various system settings. Our DSMS is implemented in Java. The PSP ring-based query plan is formed first and then deployed in the cluster using the regular pipelined parallelism of the DSMS.

To compare the performance of PSP with other approaches, we also implement the ATR and the CTR proposed in [9]. For ATR, a special stream data diffusion operator is implemented, who is in charge of generating segments for all input streams. The data diffusion operator in our ATR implementation has an important parameter, segment length, for performance tuning. For CTR, the data diffusion operator has a routing table and can calculate routing path for each input stream tuple. To avoid the uncertainty of the minimum set cover algorithm, we add one parameter for our CTR implementation, to enforce the number of copies of the state tuples in the cluster (this number is set to 2 in [9]). We also implemented the synchronization of the multi-hop execution in CTR. That is, no interleaved processing of multiple input segments is allowed to avoid state shifting caused by processing a new segment. To avoid the data diffusion operator to be the bottleneck, it is deployed separately in one node of the cluster without other join operators.

Query Sets. The MJ operator is used to identify similar images from different data sources for movement detection. The join predicate is a UDF which calculates the similarity of two images by comparing their RGB values for each pixel. The evaluation cost of the UDF is proportional to the number of pixels of the images. Our experiments use three different images: *small, middle, large* corresponding to images with 5k, 10k and 20k pixels.

Our pixel based UDF for matching images is just one of many possible image comparison functions. Indeed other

more sophisticated functions exist in the image processing domain. However design of particular functions is beyond the scope of this paper. Instead, expensive join predicates are our focus.

A symmetric nested loop join algorithm is used in the experiments. Each tuple is a byte array having the RGB values of every pixel in an image. The tuple arrival follows the Poisson distribution. The stream input rate is changed by setting the mean inter-arrival time between two tuples.

Evaluation Metrics. We use two measurements in the experimental study. We measure the runtime memory usage in terms of the number of tuples in the states of the joins. We measure the output of the query plan in terms of the average response latency for the join results.

Platform. All experiments are conducted on a cluster having 20 processing nodes and one master node. Each host has two AMD 2.6GHz Dual Core Opteron CPUs and 1GB memory. All the hosts are connected by Gigabit private networks. Each processing node runs an instance of out DSMS query processing engine executing one multi-way sliced join operator. The master node acts as the stream data sender, which runs a stream generator and diffuses generated tuples to the processing node holding the head operator in the ring. The master node also collects the join results from each sliced join as a data sink. Each query processor has a monitor thread collecting the runtime statistics. All the experiments start with empty states in operators.

8.2 Experiment 1: Sensitivity Analysis for PSP

In Section 5, we give an analytical cost model showing the impact of the parameters of the PSP model. In the cost model, the most interesting part is the relationship between the length of the ring and the response time. In this experiment, we show the existence of this relationship by varying the system parameters, including (1) number of ways of joins as: 3-way, 5-way, 7-way and 9-way; (2) join cost as: small, middle and large; (3) join selectivity as: 0.05, 0.1 and 0.5; (4) number of processing nodes: 4-19. Here we use a probabilistic join probing that enables us to control the join selectivities. The sliding window size is set to be 10 seconds for all the streams. The input rate is set to 50 tuples/sec per stream. In all the experiments, the system will run for 600 seconds.

Figures 8(a) and 8(b) show the results for 3-way join with different join selectivities and number of nodes.

The PSP scheme does not have any duplicated states, thus the memory consumptions are pretty stable among all the experiments. The query response latency is sensitive to the join selectivities and the cluster size, which affect the number of intermediate join results and join probing respectively. We observe that the average response time increases for larger join selectivities, since the workload is increased accordingly. When the cluster size increases, the response time will not always decrease. Instead, it may even increase when the size of cluster is too large. In Figure 8(a), the response time increases after having 14 nodes in the ring when join selectivity is 0.5. Also this number is sensitive to the join selectivities since different number of intermediate result will be generated and transmitted along the ring. For smaller join selectivities, we expect a large number of nodes to be optimal in the ring. This is consistent with the cost analysis of the optimal ring length discussed in Section 5.

8.3 Experiment 2: PSP vs. ATR and CTR

The next set of experiments compare the PSP scheme with the ATR and CTR solutions. Note that our implementations of ATR and CTR are on our own code base to assure a comparable platform. We aimed to meticulously recreate those prior methods as described in [9]. Comparison of absolute performance numbers is not intended, given the different code base, language, and hardware. Instead, the trends in the figures are our interest.

Figures 8(c) and 8(d) show the experimental results running the ATR approach. The join selectivity is set to be 0.1. We vary the segment size for ATR from 10 seconds, which is the sliding window size, to 50 seconds. In ATR, corresponding segments of stream tuples are processed at each node. Thus all the workload to process a single input stream tuple is done by a single node only. The result is that the average response time will not decrease when adding more nodes to the system, although the system throughput may increase. The state memory consumption for ATR is increasing steadily with the number of nodes in the system, since more duplicated segments are generated and stored in the states.

Figures 8(e) and 8(f) show the experimental results running the CTR approach. The join selectivity is set to 0.1. We vary the number of copies in the CTR approach from 2 to 9. The number of processing nodes is at least 2 times of the number of copies.

The response time is decreasing when a large number of nodes are used. This result is consistent with the analysis in [9] since the CTR is also a multi-hop join scheme. More nodes in the system will increase the CPU power and decrease the processing time. We also observe that more copies of the states result in larger response time, when the number of nodes is fixed. This is due to the minimal set cover algorithm. When more copies exist, less nodes will participate in processing the input tuple and the response time will increase accordingly. The state memory usage is fairly stable when the number of processing nodes is increased. Also the memory usage will increase with the number of copies used.

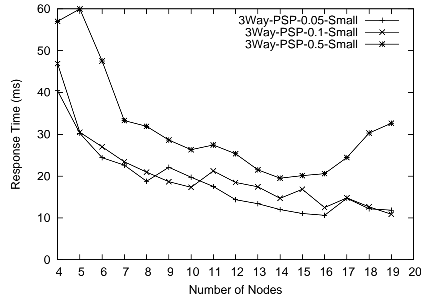
8.4 Experiment 3: Runtime Adaptation of PSP

In these experiments, we compare the response time when runtime adaptation is turn on and off in the PSP model. In the middle of the processing, the arrival rate increases from 50 tuples/sec to 100 tuples/sec. Figures 8(g) and 8(h) compare the performance of PSP under this change. Clearly runtime adaptation can make the system much more stable and robust to the changes.

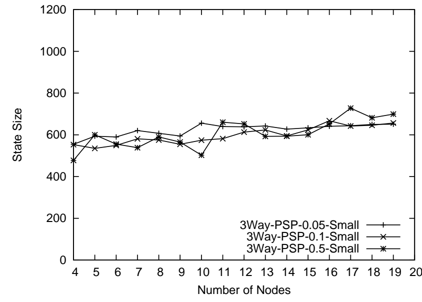
9. RELATED WORK

Parallel and distributed query processing have been the focuses of both academia and industry for a long time [7, 8, 11, 21]. Two main categories of parallelism are employed in the literature: *pipelined parallelism* and *partitioned parallelism*. The proposed PSP scheme belongs to the pipelined parallelism. Superior to the traditional query plan based pipelining, the PSP scheme has the advantage of employing an optimal length pipelining at the optimizer's will.

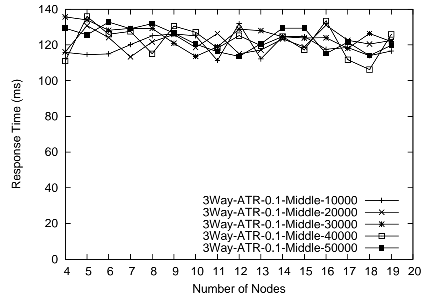
Distributed stream processing has been considered in recent years for distributed Eddies [19], Borealis [1, 2], System S [10], and DCAPE [14]. For distributed processing of stateful stream queries, state partitioning [16] has been proposed. State partitioning has the major limitation of only



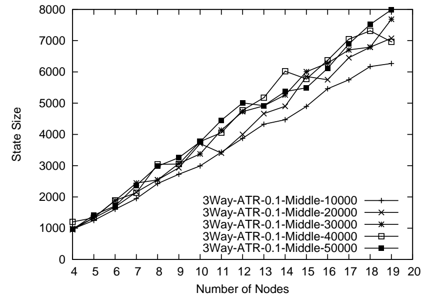
(a) PSP, Response Time



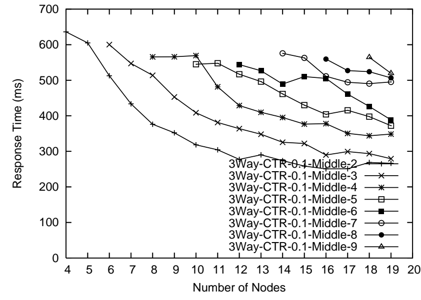
(b) PSP, State Size



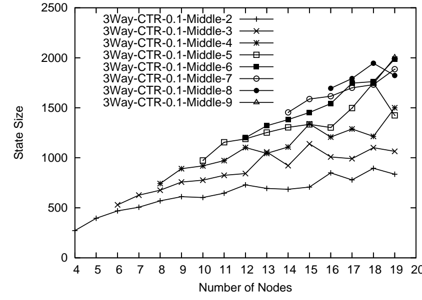
(c) ATR, Response Time



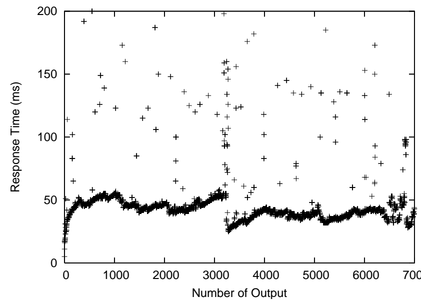
(d) ATR, State Size



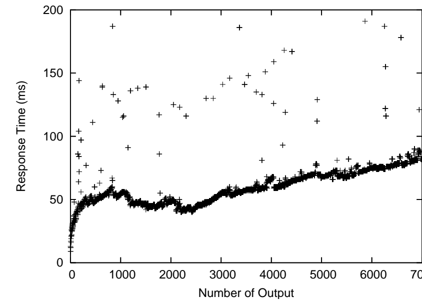
(e) CTR, Response Time



(f) CTR, State Size



(g) PSP with Adaptation



(h) PSP without Adaptation

Figure 8: Experiments Results

supporting equi-joins, while our target instead is generic joins. For generic joins, duplicated data partitions in multiple machines are required for hash-based solution.

For distributed processing of generic joins with arbitrary join predicates, a recent project [9] has proposed two state replication based approaches. As indicated in Section 7, such state duplication may abuse large amounts of memory resources, possibly also causing increased data shipping and processing costs. Section 7 provides a detailed comparison between our PSP scheme and their solutions.

The work in this paper is related to the binary state-slicing concept first presented in [23]. Different from the chain structure used to connect sliced binary join operators, all the sliced MJ operators are connected in a virtual ring architecture. Issues related to the proposed ring based execution are now tackled in this new work.

There are several existing works for finding optimal join orderings for multi-way join operators [5, 22]. Our PSP scheme is clearly orthogonal to this issue. The optimal join orderings identified by such algorithms can thus be directly utilized for processing in our proposed PSP distribution schemes.

Load-shedding [18], approximated query processing [17] and spilling data to disk [20] are alternate solutions for tackling continuous query processing with insufficient resources. Approximated query processing [17] is another general direction for handling such situation. Different from these, we aim to guarantee accurate high-performance processing and thus focus on distributed processing in a cluster. Those works are clearly orthogonal to our work, and can be applied on our solution if the total computation resources of the cluster are found to be insufficient.

10. CONCLUSION

We present a novel scheme PSP for the distributed execution of window-based joins with expensive join predicates. A cost-based analysis of the PSP schemes is conducted considering response time and memory usage. The experimental study demonstrates the significant performance improvements achieved by our solution.

Acknowledgements

We would like to thank our anonymous reviewers for their insightful comments. We are grateful to all the members of the CAPE team and other DSRG members for their support and collaboration.

11. REFERENCES

[1] Y. Ahmad, B. Berg, U. Çetintemel, and et. al. Distributed operation in the borealis stream processing engine. In *SIGMOD*, pages 882–884, 2005.

[2] Y. Ahmad and U. Çetintemel. Network-aware query processing for stream-based applications. In *VLDB*, pages 456–467, 2004.

[3] 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.

[4] B. Babcock, S. Babu, R. Motwani, and J. Widom. Models and issues in data streams. In *PODS*, pages 1–16, June 2002.

[5] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD*, pages 407–418, 2004.

[6] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *ICDE*, pages 118–129, 2005.

[7] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[8] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *Proceedings of ACM SIGMOD*, pages 9–18. ACM Press, 1992.

[9] X. Gu, P. S. Yu, and H. Wang. Adaptive load diffusion for multiway windowed stream joins. In *ICDE*, pages 146–155, 2007.

[10] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *SIGMOD*, pages 431–442, 2006.

[11] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[12] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634, 2006.

[13] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM*, pages 339–350, 2006.

[14] B. Liu, Y. Zhu, M. Jbantova, B. Momberger, and E. A. Rundensteiner. A dynamically adaptive distributed system for processing complex continuous queries. In *VLDB*, pages 1338–1341, 2005.

[15] V. Raghavan, E. A. Rundensteiner, J. P. Woycheese, and A. Mukherji. Firestream: Sensor stream processing for monitoring fire. In *ICDE*, 2007.

[16] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systmes. In *Proceedings of ICDE*, pages 25–36, 2003.

[17] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *VLDB*, pages 324–335, 2004.

[18] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.

[19] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pages 333–344, 2003.

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

[21] P. Valduriez. Parallel database systems: Open problems and new issues. *Distributed and Parallel Databases*, 1(2):137–165, 1993.

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

[23] S. Wang, E. A. Rundensteiner, S. Ganguly, and S. Bhatnagar. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *VLDB*, pages 619–630, 2006.