JISC: Adaptive Stream Processing Using Just-In-Time State Completion*

Ahmed M. Aly[1], Walid G. Aref[1], Mourad Ouzzani[2], Hosam M. Mahmoud[3]
[1]Purdue University, West Lafayette, IN
[2]Qatar Computing Research Institute, Doha, Qatar
[3]The George Washington University, Washington, D.C.
{aaly, aref}@cs.purdue.edu, mouzzani@qf.org.qa, hosam@gwu.edu

ABSTRACT

The continuous and dynamic nature of data streams may lead a query execution plan (QEP) of a long-running continuous query to become suboptimal during execution, and hence will need to be altered. The ability to perform an efficient and flawless transition to an equivalent, yet optimal QEP is essential for a data stream query processor. Such transition is challenging for plans with stateful binary operators, such as joins, where the states of the QEP have to be maintained *during* query transition without compromising the correctness of the query output. This paper presents Just-In-Time State Completion (JISC); a new technique for query plan migration. JISC does not cause any halt to the query execution, and thus allows the query to maintain steady output. JISC is applicable to pipelined as well as eddy-based query evaluation frameworks. Probabilistic analysis of the cost and experimental studies show that JISC increases the execution throughput during the plan migration stage by up to an order of magnitude compared to existing solutions.

1. INTRODUCTION

The evolution of sensor capabilities has led to countless applications and systems that require *continuous* monitoring of data streams. Examples include sensor networks, spatio-temporal databases, and medical monitoring systems.

To support such continuous queries, data stream management systems (DSMSs) have to consider settings in which queries are increasingly complex, statistics are not always available, and data is being streamed from distributed sources [1, 2] with changes in arrival rates and value distributions. These settings render the traditional *optimize-then-execute* techniques inapplicable [1] since the query execution plan of a long-running continuous query may become suboptimal at any time during execution. This calls for new techniques that can alter the QEP of a continuous query *during* its execution, i.e., *optimize-at-runtime*.

Optimize-at-runtime plan transitions are particularly challenging for QEPs with stateful operators such as joins. The challenge resides in how to perform the transition while maintaining the execution state and without compromising the correctness of the query output. Existing techniques for QEP adaptation rely on either (1) avoiding operator states altogether, e.g., [3], or (2) eagerly computing the execution state of the new plan at the time of plan transition, e.g., the Moving State Strategy [4]. In the former approach, avoiding operator states during normal execution allows the transition to any new plan with minimum overhead. However, since there are no intermediate states at any operator, intermediate results have to be recomputed for each input tuple even if there is no plan transition, which is expensive, especially in the case of multiple joins. In the latter approach, when a plan transition is initiated, the query execution is halted and then the states of the operators in the new plan that are not in the old plan are computed *all at once*. Computing these states is usually expensive and causes high output latency until the execution is resumed with the new plan. This output latency is not suitable for applications that require steady query output, e.g., safety critical monitoring and real-time applications. This approach is also not suitable for fast data streams as the input buffers may get overflowed while the execution is halted until the plan transition stage is complete.

Maintaining the states of the QEP after a plan transition without halting the query execution, i.e., maintaining steady query output, is challenging. In [4], the Parallel Track Strategy addresses this challenge by running two plans in parallel (the old and new plans). New tuples are processed in both plans as long as the old plan has the old state entries. When the old state entries in the old plan are replaced by new ones, the old plan is discarded. Although the Parallel Track Strategy has minimal output latency, its execution throughput drops by 50% during plan migration; every tuple is processed by both the old and new plans, i.e., it is processed twice, until the old plan is discarded. Furthermore, since two QEPs are running and each QEP produces its own output, duplicate elimination is required on top of the two QEPs to merge their outputs. Other techniques based on the Parallel Track strategy have been proposed in the literature (see [5, 6]). Unfortunately, these techniques inherit the low execution throughput of the Parallel Track Strategy during plan migration.

This paper introduces Just-In-Time State Completion (JISC); a new technique for plan adaptation of continuous queries over data streams. Instead of fully computing the missing states upon plan transition or running two plans in parallel, JISC *completes* the states of the new plan *on-demand* during execution, i.e., performs lazy state migration. In highly dynamic environments where the queries' operators and streams have fluctuating selectivities and arrival rates, overlapped plan transitions can happen frequently. JISC's lazy migration strategy enables avoiding performance thrashing in such scenarios; as we show in the Related Work

*This work was supported by National Science Foundation Grants IIS 0916614, IIS 1117766, and IIS 0964639 and by an NPRP grant from the Qatar National Research Fund.

10.5441/002/edbt.2014.08

Section, this is an unsolved issue in all other existing techniques.

While ensuring the correctness of the query output, JISC does not require the query execution to halt, i.e., JISC has minimal output latency, and hence maintains steady query output. During normal operation (before or after a plan transition), JISC incurs almost no overhead over the regular execution of the query. We should note that JISC is applicable to pipelined as well as eddy-based query evaluation frameworks.

The contributions of the paper are summarized as follows:

- We study the existing approaches for query plan adaptation in DSMSs and highlight their drawbacks.
- We introduce JISC and show how it can be applied to left-deep as well as bushy QEPs.
- We show how JISC can be applied to an eddy-based execution framework, e.g., STAIRs, and demonstrate how JISC can enhance its performance.
- We study the performance of JISC analytically and prove a sharp concentration law that governs its performance.
- We study the performance of JISC experimentally by comparing it to the state of the art.

The rest of the paper is organized as follows. Section 2 presents the query execution model and defines the problem. Section 3 discusses the related work. Section 4 introduces JISC and explains its details. Sections 5 and 6 study the performance of JISC both analytically and experimentally. Section 7 concludes the paper.

2. PRELIMINARIES

This paper focusses on the efficiency of query execution during a plan transition. However, we do not address the actual conditions that trigger a plan transition; which is an orthogonal issue. We refer the reader to the literature , e.g., [7, 8, 9, 10]), for more details on this issue. In the rest of this section, we present the execution model and define the problem.

2.1 Execution Model

We assume that a query is compiled into a binary tree-structured plan of pipelined operators, where each operator is aware of its parent, as well as its left and right operators. Although they can be addressed in a similar manner, n-ary operators, e.g., MJoin [11, 1] are not discussed in this paper. In the case of unary operators, only a parent and a child operators exist. In the case of leaf operators, e.g., stream-scan, only a parent operator exists. We assume that all operators are push-based (as in [12]), i.e., each operator sends its output tuples directly to the next (parent) operator in the pipeline, and there is an input queue for each operator to buffer the tuples. Tuple load shedding may occur when tuples overflow the input buffers. However, this issue is orthogonal to the issues addressed in this paper.

To simplify the presentation of the paper, we first study the problem of plan transition with respect to join operators. In Section 4.7 we show how the techniques presented in the paper can be applied to other operators, e.g., set-difference.

For equi-joins, we employ *symmetric hash join* [13, 1]. When a tuple appears at either input of the operator, it is inserted into the hash table of that stream, then the opposite hash table is probed. If a match is found, the corresponding join-tuple is added to the join-state and is passed to the next operator. Figure 1 gives a left-deep plan employing symmetric hash join for the query $(R \bowtie S) \bowtie T$.

Since hash joins are applicable only to equi-joins, we use a nested-loops join for general theta joins. If a query requires equi-joins between a set of streams and general theta-joins between another set, we form a hybrid plan, with a mix of nested-loops and hash joins [7].

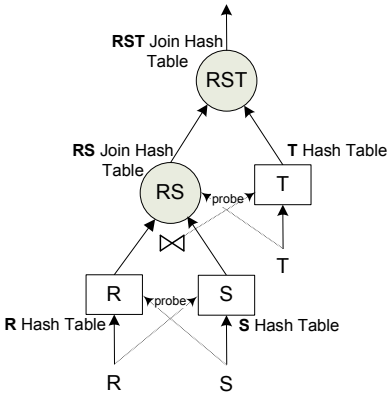

Figure 1: A left-deep plan with two symmetric hash joins.

Sliding Windows

Because of the endless nature of data streams, *sliding windows* are usually specified in the query to have a bound on its focus of interest. For sliding windows, query execution remains essentially the same except that when the window slides, tuples that become outside the window have to be deleted from the join operator data structures [1]. For correctness, a removed tuple from a state has to be traced throughout the whole execution pipeline in a bottom-up fashion, removing (e.g., as in the case of a join) or possibly adding (e.g., as in the case of set difference) all the corresponding state entries at all the upper operators. For more details on how sliding windows are maintained, the reader is referred to [14, 15, 16].

2.2 Problem Definition

Query plan adaptation is the process of dynamically transferring the query execution from an old (suboptimal) plan to a new (optimal) plan. It has to guarantee that exactly the same output is produced for the same input, with or without a transition. For long-running continuous queries, once the current query execution plan is detected to be suboptimal, transition to a new optimized plan should take place during execution, i.e., *optimize-at-runtime*.

In this paper, we address a key challenge in optimizing plan transitions at runtime, namely transferring the execution to the new plan *without halting* it, i.e., keeping a steady query output, while incurring minimum execution overhead.

When a plan transition takes place, binary operators, e.g., joins, need special handling as they embed states. Otherwise, the correctness of the execution is compromised. Consider a query that joins the streams R, S, T, and U. Assume that a transition from the plan $((R \bowtie S) \bowtie T) \bowtie U$ to the plan $((S \bowtie T) \bowtie R) \bowtie U$ takes place (See Figure 2). The join-state ST does not exist in the old plan, but exists in the new plan. The join-state ST is needed to correctly execute the query. However, upon transition, state ST is empty. Unless the join-state ST is computed for the new plan, there is a risk of losing output tuples as well as producing incorrect output tuples. We illustrate these risks by the following three scenarios:

1. Consider the scenario in which a tuple, say r, from R should join with tuples, say s, t, and u, from S, T, and U, respectively. Before plan transition, only s, t, and u have arrived and have been processed through the old plan. Right after plan transition, r is received. The join operator $(S \bowtie T) \bowtie R$ probes State ST checking if a joined pair from S and T that can join with r exists. Since State ST is empty, nothing is found. The output corresponding to the quadruple (r, s, t, u) is missed.

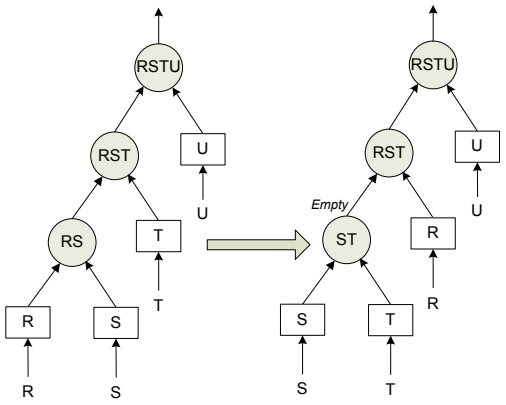

Figure 2: Transition from an old QEP $((R \bowtie S) \bowtie T) \bowtie U$ to a new QEP $((S \bowtie T) \bowtie R) \bowtie U$. The join-state ST is empty in the new QEP right after the transition because the join $S \bowtie T$ does not exist in the old plan.

2. Consider the scenario in which the quadruple, say (r, s, t, u), is already part of the answer before plan transition. Then, right after plan transition, the window of S slides such that s is outside the window, so the quadruple (r, s, t, u) should no longer be part of the query answer. The join operator $S \bowtie T$ probes State ST to see if a joined pair related to s exists. Since State ST is empty, nothing is found and no further action is propagated up the pipeline. The output related to that window slide is missed. The quadruple (r, s, t, u) is still part of the answer of the query (State $RSTU$) although it should not.
3. Consider the scenario in which a tuple, say r, from R that should join with tuples, say s, t, and u, from S, T and U, respectively. Before plan transition, only r, s, and t have arrived, and have been processed through the old plan. Right after plan transition, the window of S slides such that s is outside the window. Thus, the quadruple (r, s, t, u) should not be produced as output, even if u arrives. The join operator $S \bowtie T$ probes its State ST to see if a joined pair related to s exists. Since State ST is empty, nothing is found and no further action is propagated up the pipeline. Thus, State RST will have an invalid state entry related to the r, s, and t. When u arrives, it probes State RST and finds the invalid state entry for (r, s, t). As a result, a wrong output corresponding to the quadruple (r, s, t, u) is produced.

We clearly see from the above scenarios how plan transition is challenging for QEPs with binary operators. Unless careful migration of states is performed, losing output tuples or producing incorrect output tuples can happen.

3. RELATED WORK

In this section, we discuss state-of-the-art techniques for query plan adaptation of continuous queries in DSMSs.

3.1 Binary-State Avoidance

CACQ [3] introduces a mechanism for continuous query execution that supports plan transition. In CACQ, there are no intermediate (binary) states since pipelining of operators is simply avoided. Instead, all query operators are connected through an eddy operator [17] that acts as a router for the tuple flow through the QEP. In other words, each tuple can have its own plan. Whenever the routing decision changes, the eddy operator is notified, and simply routes tuples according to the new plan. CACQ splits a binary join into two unary operators called SteMs (State Modules) [18]. Thus, a join tree is broken down into multiple SteMs (one per input stream). Upon reception of a tuple from any stream, that tuple is joined across all the SteMs of all the other streams until it emerges as output or else disqualifies.

Since the eddy is the *next* operator of every SteM, every tuple passes through the eddy as many times as it qualifies a join. This leads to a drop in the execution throughput by nearly half of its value compared to a normal pipelined mode. Moreover, CACQ introduces a per-tuple overhead as each tuple maintains a bit-vector that represents the progress of the tuple throughout the execution, i.e., which operators have been processed and which are remaining.

While CACQ consumes no computation at the time of plan transition, the use of SteMs without storing any intermediate results is inefficient during normal operation (i.e., when no plan transition takes place). More specifically, consider the QEP $((R \bowtie S) \bowtie T) \bowtie U$ in Figure 2a. Assume that a tuple from R joins with a tuple from S and a tuple from T, but the joining tuple from U has not arrived yet. Since CACQ has no join-state, the arrival of a new tuple from U will trigger three joins with the SteMs of R, S, and T. However, if the join-state RST is kept as in Figure 2a, *only* one join will be needed with the join-state RST. This difference becomes significant when the number of joins in the QEP is large.

3.2 Moving State Strategy

To avoid the large number of joins in CACQ, the *Moving State* Strategy [4] has been proposed. Given a normal QEP, once a plan transition is triggered, the execution *halts* altogether. Any missing states in the new plan are computed and then normal execution proceeds with the new plan.

Similarly, in [19], within the eddy framework, the STAIR operator uses intermediate join states as in the Moving State Strategy. A STAIR operator encapsulates the join-state that typically would be stored inside the join operators. In particular, each join operator is replaced by two STAIRs that interact with the eddy directly. These two STAIRs are called duals of each other. Executing queries using STAIRs is similar to that of symmetric join operators. Instead of routing a tuple to a join operator, the eddy performs an insert into one STAIR, and a probe into its dual (i.e., opposite).

Both the Moving State Strategy and STAIRs employ a *greedy* state migration policy that *eagerly* migrates state once routing decisions change, ignoring the cost of state migration. Both strategies share the following drawbacks:

- Both strategies require the execution to stop until the states in the new plan are computed, and hence cannot maintain steady query output.
- Since both strategies require the execution to stop, overflow in the main-memory buffers of the input data streams is likely to occur during plan migration, especially for streams with high arrival rates.
- In highly dynamic environments Both strategies can result in suboptimal performance for two reasons: (1) the query engine might thrash by performing too many migrations in response to fluctuating selectivities, and (2) a large state migration can happen at the end of query execution, with no subsequent payoff, e.g., if the migrated states end up being not used by any of the upcoming tuples.

Our proposed technique, JISC, avoids these drawbacks by maintaining steady query output and computing the missing states on demand, i.e., performing lazy migration.

3.3 Parallel Track Strategy

The *Parallel Track* Strategy [4] achieves query plan adaptation without halting the query execution. Once a plan transition is triggered, the old and new execution plans run simultaneously, i.e., all new tuples are processed through both plans. *Duplicate elimination* is performed at the root operator combining both plans [4]. The old plan is discarded when it contains only new entries in its states. Other techniques, namely [5, 6], combine the Moving State and Parallel Track strategies. While these techniques achieve some gains compared to the standard Parallel Track strategy, they inherit its drawbacks, which we highlight below:

- Since new tuples get processed twice during the plan transition stage, the execution throughput drops by 50%. This drop may continue for a long time (until all old entries are replaced by new ones). This is especially true for execution plans that involve large numbers of operators with large numbers of state entries.
- The need for duplicate elimination increases the execution overhead.
- The cost for detecting when to discard the old plan is high. Each operator in the old plan periodically checks if all the old tuples have been purged from its state. This check is repeated until the old plan is discarded, and hence introduces significant overhead.
- In highly dynamic environments where overlapped plan transitions can occur, multiple (more than two) QEPs can be simultaneously executing, and hence the performance of the query execution can severely degrade. In addition, the cost of duplicate elimination will multiply.

Our proposed technique, JISC, employs a single QEP for both the old and new plans, and hence avoids the above drawbacks.

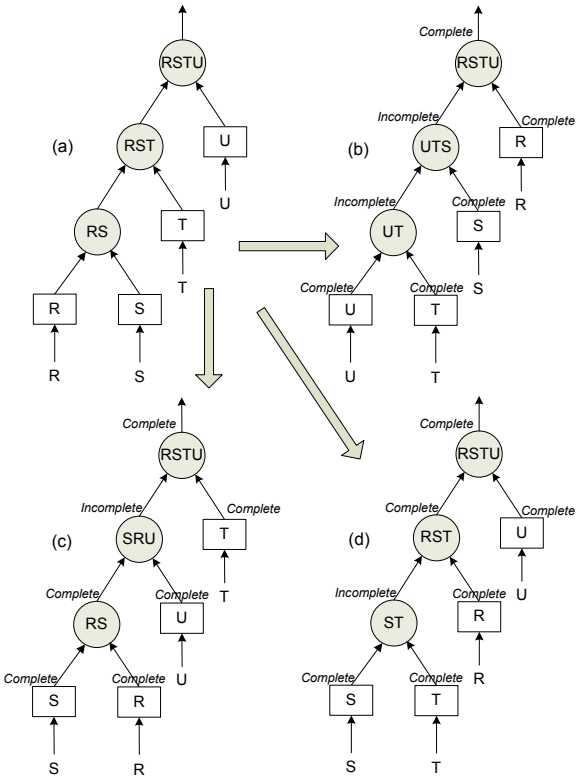

Figure 3: Complete vs. Incomplete States. (a) Old plan. (b), (c), and (d) Some possible new plans. In (d) for example, State RST is complete because it exists in the old plan, while State ST is incomplete because it does not exist in the old plan.

4. JUST-IN-TIME STATE COMPLETION

We introduce Just-In-Time State Completion (JISC); a new technique for plan adaptation of continuous queries over data streams. JISC is applicable to both pipelined and eddy-based execution frameworks. Upon plan transition, JISC does not cause any halts to the execution in order to migrate the states from the old to the new plan, and thus maintains a *steady* query output. Instead of fully computing the missing states upon plan transition or running two execution plans in parallel, JISC *incrementally* completes the states *during* execution on an as-needed basis. States are completed on-demand, i.e., whenever a binary operator in the new plan requires a state entry that is missing, only that state entry is computed, hence the name Just-In-Time State Completion.

When a plan transition takes place, JISC classifies the states of the new plan into *incomplete* and *complete* according to the following definition. Figure 3 illustrates some query plan transitions and the corresponding incomplete and complete states.

DEFINITION 1. During a plan transition, a state in the new plan is ***complete*** if it exists in the old plan, otherwise, it is ***incomplete***.

JISC tries to use work that has been already performed in the old plan. On the one hand, a state in the old plan that exists in the new plan is copied to the new plan and marked as complete, e.g., State RST in Figure 3a is copied to the plan in Figure 3d. On the other hand, a state in the old plan that does not exist in the new plan is discarded, e.g. State RS in Figure 3a is discarded in Figure 3d.

JISC is progressive in nature, i.e., as tuples are processed, the entries of an incomplete state are incrementally completed. Once all the entries of an incomplete state are completed, that state is marked *complete* (refer to Section 4.3 for details).

We illustrate how states are completed incrementally. For illustration, we use symmetric hash join with a join attribute, say ID. Assume that a join at node, say j, has two child operators, say j_L and j_R. Assume further that j_L is incomplete. When a tuple, say t, reaches j coming from j_R, the join at j probes the opposite state (i.e., the hash table of j_L). Assume that the joining tuple is not found in j_L's state. Since j_L is incomplete, only the entries in j_L that correspond to the tuples that join with t need to be completed. State completion of the entries corresponding to ID in j_L starts at the *highest* operator with a *complete* state that lies in the underlying subtree below j_L. Whenever any of the probed states is incomplete, state completion for the designated ID can be triggered recursively. State completion for ID's entry propagates upwards and stops at the operator at which it was initiated, i.e., at j_L.

For example, consider the QEP in Figure 3b, a new tuple, say r, from Stream R causes a probe to the incomplete State UTS, which triggers a state completion. This state completion starts from the state of Stream S (the highest complete state under the subtree of UTS). If r joins with S, State UT is probed. Since UT is incomplete, state completion will be invoked recursively to start from the state of U and continues upwards (completing states UT and UTS) until it stops at $UT \bowtie S$. At this moment, it is guaranteed that the entries corresponding to r are complete in State UTS. Hence, the join can proceed normally. As another example, consider the QEP in Figure 3c, a tuple, say t, from Stream T causes a probe to the incomplete state SRU, which initiates state com-

Procedure 1 JISC Join

***Terms*:** $tuple$: Tuple to be joined. $oppstOp$: Opposite operator. $highestComp$: Highest operator with complete state in the subtree of $oppstOp$.

```
1:  if (oppstOp.state.contains(tuple.joinAttr)) then
2:      joinState.add(joinTuples)
3:      parent.process(joinTuples)
4:  else
5:      if (tuple.isFresh and oppstOp.state.isIncomplete) then
6:          completeState(tuple, highestComp, oppstOp)
7:          if (oppstOp.state.contains(tuple.joinAttr)) then
8:              joinState.add(joinTuples)
9:              parent.process(joinTuples)
10:         end if
11:     end if
12: end if
```

Procedure 2 CompleteStateBT (recursive state completion for bushy trees)

***Terms*:** $tuple$: Tuple whose corresponding entries will be completed. $oppstOp$: Opposite operator. $highestComp$: Highest operator with complete state in the subtree of $oppositeOp$.

```
1:  if (oppositeOp.state.contains(tuple.joinAttr)) then
2:      joinState.add(joinTuples)
3:      parent.process(joinTuples)
4:  else
5:      if (oppositeOp.state.isIncomplete) then
6:          completeStateBT(tuple, highestComp, oppositeOp)
7:          if (oppositeOp.state.contains(tuple.joinAttr)) then
8:              joinState.add(joinTuples)
9:              parent.process(joinTuples)
10:         end if
11:     end if
12: end if
```

Procedure 3 CompleteStateLDT (state completion for left-deep trees)

***Terms*:** $tuple$: Tuple whose corresponding entries will be completed. $startOp$: Operator at which state completion should start. $stopOp$: Operator at which state completion should stop.

```
1:  currentOp ← startOp.parent
2:  while (true) do
3:      if (leftOp.state.contains(tuple.joinAttr) and
        rightOp.state.contains(tuple.joinAttr)) then
4:          currentOp.joinState.add(joinTuples)
5:          tuple ← joinTuples
6:      else
7:          return
8:      end if
9:      if (currentOp = stopOp) then
10:         return
11:     else
12:         currentOp ← currentOp.parent
13:     end if
14: end while
```

pletion. This state completion starts from the state of Stream U that probes State RS and finds it complete. State completion will continue upwards (completing only State SRU), then stops at the operator $(S \bowtie R) \bowtie U$.

The above procedure for recursive state completion is general and applies to the left-deep as well as bushy query evaluation pipelines (QEPs). A left-deep QEP has an additional property that can be utilized to further simplify state completion. Mainly, the states of all the inner streams, i.e., the right branches are complete. Only the left branches (the outers of the joins) can be incomplete. Therefore, for state completion, we can pick the highest node from among the left branches in the left deep QEP that is complete and move upwards. Notice that we can always find a complete state in any branch since the states of the leaf nodes are always complete. Therefore, we can eliminate the recursion steps for left deep QEPs. Refer to Figure 3b for illustration. When a new tuple from Stream R arrives and causes a probe to the incomplete State UTS, we can start directly from State U since it is the highest state in the left branch of the left deep tree that is complete. Then, we move upwards while probing the states of the opposite operators and completing the states until reaching State UTS.

Procedure 1 gives pseudo-code for JISC when applied for symmetric hash join. The flag `isFresh` in Line 5 helps avoid rechecking for the state of an incomplete tuple multiple times, e.g., when that tuple is not part of the join result. Section 4.4 explains this issue further. Procedures 2 and 3 give pseudo-code for state completion for bushy and left-deep QEPs, respectively. JISC invokes either procedure according to the type of QEP involved.

4.1 Safe Plan Transition

In JISC, a state in the old plan that also exists in the new plan is copied to the new plan. However, a state in the old plan that does not exist in the new plan is discarded. The process of discarding states is critical to the correctness of query execution, hence, the question: *"When is it safe to discard the states of the old plan?"*.

As mentioned in Section 2, we assume that each operator in the QEP has an input queue that buffers the tuples. If the state of an operator is discarded while there are tuples at its input queue, correctness of the execution of the whole query is compromised. Consider for example the QEP in Figure 3a. The join operator $(R \bowtie S) \bowtie T$ (with State RST) has an input queue, say q, that contains tuples with the join schema of $R \bowtie S$, that are supposed to join with Streams T and U. As discussed in Section 2, States RST, T, and U are essential for the correct processing of the tuples in q. If a plan transition to the QEP in Figure 3b is to take place, State RST will be discarded. Moreover, according to the new join order, States T and U will be replaced with States S and R, respectively. This means that the tuples in q will not have the states necessary for correct processing.

To solve the above problem, JISC does not switch the execution to the new plan and discard any states until all the tuples in the input queues are completely processed through the old plan. Once a plan transition is decided, the input queues to all the query operators are cleared, i.e., all the tuples that are received before a plan transition is decided are processed through the old plan, and are pushed up the QEP until they reach the output. Afterwards, the old plan is discarded, and processing with the new plan takes place. Then, tuples that are received after the plan transition are processed through the new plan. This also means that each input tuple is processed only once, either in the old plan or in the new plan, which guarantees that JISC is duplicate-free.

Note that the buffer-clearing phase described above does not cause any output delays since during that phase tuples that are received before the plan transition are processed through the old plan. Immediately after the buffer-clearing phase, tuples that are received after the plan transition are processed through the new plan. Thus, the output latency is always minimal. It should also be noted that the buffer-clearing phase is also necessary for the correctness of the Moving State Strategy [4]. In other words, both JISC and the Moving State Strategy share this phase once a plan transition is decided. The main difference between the two strategies is that after the buffer-clearing phase, the Moving State Strategy computes *all* the states of the new plan in a greedy manner, leading to significant output latency. On the other hand, JISC employs a lazy approach by computing the states of the new plan only on-demand, leading to minimal output latency.

4.2 Handling Sliding Windows

As discussed in Section 2.1, for sliding windows, tuples that are outside the window have to be deleted from the states inside the operators. In a regular execution pipeline, i.e., where there is no plan adaptation, the removal of the state entries corresponding to a tuple, say t, starts at the hash table of the stream of t, then goes up the pipeline to the next join state. If a join-state entry corresponding to t is found, it is removed, and then the removal propagates to the next operator, and so on. This removal process continues until no join-state entry corresponding to t is found, i.e., no match, or when the root operator is reached. In JISC, we follow a similar procedure for clearing the states related to a window slide, except that if the join-state is incomplete, the removal process continues up the pipeline, regardless of finding a match in the join-state. This is illustrated in the example to follow.

Recall the third scenario presented in Section 3 for the plan transition in Figure 2 in which a tuple, say r, from Stream R should join with tuples, say s, t, and u, from Streams S, T and U, respectively. Before plan transition, only r, s, and t have arrived, and have been processed through the old plan. Right after plan transition, the window of Stream S slides such that s is outside the query window. As a result, the quadruple (r, s, t, u) should not be produced as output, even if u arrives. The join operator $S \bowtie T$ probes its join-state ST to see if a joined pair from S and T related to s exists. Since the join-state ST is empty, nothing is found. However, the removal action is propagated up the pipeline because ST is an incomplete state. Hence, the join-state RST will remove the state entry related to r, s, and t. Now, when u is received, it probes State RST and does not find any (invalid) state entries that match, and no wrong output is produced.

4.3 State Completion Detection

One important issue is how to efficiently detect that an incomplete state has become complete and mark it as such. A similar issue arises in the Parallel Track Strategy [4] to detect when it is safe to discard an old plan. The Parallel Track Strategy forces each operator in the old plan to periodically check if all old tuples have been purged from its state. This check is applicable to JISC as well, but is costly, especially for execution plans that involve a large number of operators with possibly a large number of state entries.

We introduce a more efficient solution by keeping an integer counter at each binary operator with an incomplete state. When a transition takes place, the counter's value is initialized according to one of the following cases:

- Case 1: If both the left and right operators have complete states, e.g., State RS in Figure 3c, the counter is initialized to be the *smaller* of the number of distinct values of the join attribute inside the left or the right operators' states.
- Case 2: If either the left or right operator has an incomplete state, e.g., State SRU in Figure 3c, the counter is initialized to be the number of distinct values of the join attribute inside the complete state, e.g., number of distinct values in State U in Figure 3c.
- Case 3: If both the left and right operators have incomplete states, e.g., as what can happen in a bushy plan, the counter's value is meaningless.

For *left-deep* QEPs, the right operator is a stream-scan operator that always has a complete state, so, either Case 1 or Case 2 holds. Whenever the missing join state entries in an incomplete state that correspond to a join attribute value in the left or the right value are computed, the counter is decremented accordingly. When the counter's value reaches zero, the state is declared complete.

For *bushy* QEPs, JISC initializes the counter if either Case 1 or Case 2 holds. For Case 3, JISC detects that a state is complete whenever the states of both its right and left operators *get completed*. By *get completed*, we mean that the state was incomplete right after a plan transition and was incrementally completed through the state completion procedures. Leaf joins apply the *counting* technique (as in Cases 1 and 2) to detect state completion. Notice that, in a leaf join, one of the children is necessarily a scan operator which has a complete state. When a leaf join-state gets completed, it notifies its parent. The parent checks if both of its children have complete states, and if this is the case it marks its state as complete and recursively notifies its parent.

4.4 Avoiding Repeated Computations

Consider the plan transition in Figure 2b. Assume that a tuple, say r_1, is received from Stream R that has the join attribute value v. The join operator at the root probes State UTS to find a matching tuple for r_1. Since UTS is incomplete, state completion is triggered and the corresponding state entries for v are computed at States UT and UTS. Assume that after this step, UTS is still incomplete, i.e., not all state entries have been computed. Assume further that another tuple, say r_2, is received from Stream R, and has the same join attribute value v. Since UTS is still incomplete, the corresponding state entries for v would be computed again at States UT and UTS. However, the computations for completing States UT and UTS are exactly the same for both r_1 and r_2 because both tuples have the same join attribute value. Hence, unless additional measures are taken, repeated computations would occur for r_2.

To avoid these repeated computations, JISC *attempts* to compute the state entries that correspond to a certain join attribute value *at most once*. An attempt to compute the state entries that correspond to a certain join attribute value, say v, happens the first time a tuple having the join attribute value v is received after the transition takes place. Once this tuple is processed, the status of further tuples received later having the same attribute value v is switched from *fresh* to *attempted*. Thus, JISC classifies the received tuples based on their join attribute values according to the following definition:

DEFINITION 2. A tuple is said to be ***fresh*** if no other tuple having its join attribute value is received after plan transition. Otherwise, the tuple is said to be ***attempted***.

The process of completing the state for a tuple having the attribute value v will happen at most once, regardless of the states being complete or not. The check for the per-tuple flag *isFresh* (Line 5 in Procedure 1) prevents repeated computations for tuples that have corresponding complete state entries at all states, even if some states are incomplete.

The information according to which the tuples are classified into *fresh* or *attempted* is available at the state (hash table) of the stream. In addition, JISC records the timestamp of the most recent plan transition. Once a tuple is received from a stream, the hash table of that stream is probed ($O(1)$ CPU time) to check if another tuple having the same join attribute has been received after the most recent plan transition.

As discussed in Section 4.2, for sliding windows, a state-clearing tuple related to a window slide propagates up the pipeline and if an incomplete join-state is reached, the tuple continues to propagate regardless of finding a match in the join-state. As an optimization, we apply this state-clearing technique only for fresh tuples. An attempted tuple is guaranteed to have complete state entries at all the operators and thus is allowed to propagate up the pipeline only if a match is found in the join-state.

4.5 Overlapped Plan Transitions

JISC is a lazy plan migration strategy, i.e., JISC completes the states only as needed. This means that a new plan transition might occur while the effect of a previous plan transition is not cleared, i.e., not all states have been completed. If Definition 1 is to be directly applied in case of overlapped plan transitions, erroneous query output can occur.

Consider the scenario in Figure 4, in which a plan transition happens (from Plan (a) to Plan (b)) and State ST in Plan (b) is declared incomplete according to Definition 1. During the processing of the new incoming tuples, a new plan transition (from Plan (b) to Plan (c)) takes place before all the incomplete states get completed. If Definition 1 is to be applied again, State ST in plan (c) will be declared complete although it is not. This can lead to incorrect output tuples as discussed in Section 2.2.

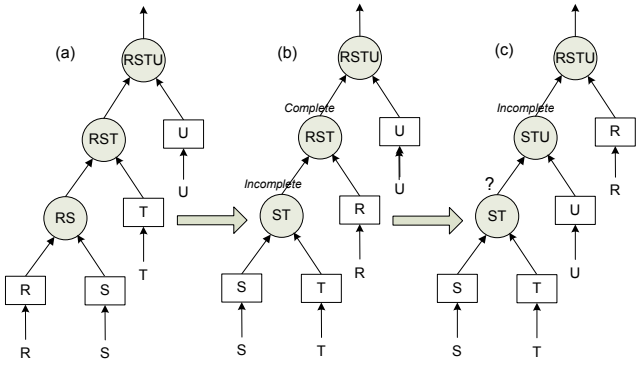

Figure 4: Transition from plan (b) to plan (c) takes place while State ST is still incomplete. If Definition 1 is to be directly applied, State ST in plan (c) will be declared complete although it is not.

To solve this problem, JISC declares a state in the new plan as complete only when it is also complete in the old plan. If a state in the new plan exists in the old plan, but is incomplete in the old plan, then it will remain incomplete in the new plan. Thus, according to the scenario in Figure 4, State ST in plan (c) will be declared incomplete and hence will avoid the incorrect output tuple scenarios.

4.6 Applying JISC to STAIRs

A close look at STAIRs [19] reveals that it is actually the same as the Moving State Strategy when applied to eddies. When a plan transition takes place, STAIRs eagerly computes the states of the new plan by performing the Promote and Demote operations on *all* state entries. However, the cost of the Promote and Demote operations can be amortized across the whole execution by performing these operations in an on-demand basis. Thus, JISC can be applied to the eddy framework (i.e., STAIRs) as follows. When a plan transition happens, Demote operations are performed to all the states in the new plan that do not exist in the new plan, i.e., these states are discarded. The states of the STAIR operators in the new plan are classified into complete and incomplete according to Definition 1.

When a tuple, say t, probes a STAIR operator with an incomplete state, t is routed to the highest STAIR with a complete state in the underlying logical subtree in the logical plan. This is equivalent to the Promote operation of the state entry corresponding to t. State completion detection of a STAIR follows a similar procedure as in a regular execution pipeline. A STAIR's state is declared complete when all its state entries are promoted.

4.7 JISC with Other Operators

By definition, a unary operator's state, e.g., U in Figure 3b is always complete. Aggregators, e.g., groupby's, are unary operators which have no issues during a plan migration. For instance, if a count is maintained on top of the QEPs of Figure 2, it will not be affected by a plan transition; according to Definition 1, the root of the QEP always has a complete state. Thus, the main issue during a plan transition is to carefully handle the states of binary operators. So far, we have introduced JISC for join operators only. Below, we briefly explain how JISC can be applied to other binary operators.

In general, all binary operators share the same operational mechanism as they rely on the existence of the states of the left and right operators as well as their own states. The only difference is in the semantics of each operation. Unlike join operators in which tuples are always added to the join state, a tuple received at a set-difference operator may be *removed* from the set-difference state. For instance, consider the set-difference operator $R - S$ that retrieves the tuples in Stream R that do not exist in Stream S. On the one hand, a received tuple from Stream S is then used to probe the set-difference state. If a match is found, then the matched tuple is *removed* from the set-difference state. On the other hand, a received tuple from Stream R is used to probe the state of S. If *no* match is found, then the tuple is added to the set-difference state.

Applying JISC to a QEP with set-difference operators is straightforward; the same conditions for safe plan transition and state completion detection apply. However, according to the logic of the set-difference operator, if a tuple received from an inner stream probes an incomplete state, it is forwarded *up* the pipeline (instead of down as in joins) until it hits the first complete state. This way, it gets cleared from the operator states in a way similar to handling sliding windows as in Section 4.2.

For illustration, consider a left-deep QEP $(((A-B)-C)-D)$ that migrates to $(((A-D)-B)-C)$. Upon plan transition, according to Definition 1, States AD and ADB are incomplete while State $ADBC$ is complete. A received tuple from D probes the set-difference state AD which is incomplete, so the tuple gets forwarded *up* the QEP until it reaches the first complete state, i.e., $ADBC$ in this case. Similarly, a received tuple from B probes State ADB which is incomplete, so the tuple gets forwarded up the QEP until it reaches State $ADBC$. Notice that tuples received from the outer stream, i.e., A probe State D which is complete because it is a unary operator.

5. ANALYSIS OF JISC

Although JISC aims at computing the missing states of an operator, JISC does not add *any* memory overhead, except for the counter that is used for state completion detection, which is a simple integer per operator. JISC does not allocate any additional memory to complete the states of an operator; it just keeps adding the missing states until an operator eventually functions as if in a regular QEP. Thus, we need to analyze only the execution time of JISC.

5.1 Benefits of Lazy Migration

5.1.1 Steady Query Output

The overall execution time of the Moving State Strategy [4] is close to that of JISC. The difference between the two techniques is in the latency of the output. JISC has minimal output latency because it employs a lazy migration strategy, and hence maintains steady query output. In contrast, the Moving State Strategy employs an eager migration strategy by computing all the missing (incomplete) states of the new plan all at once. If all the streams have the same sliding window sizes, say w, the state recomputation

step of the Moving State Strategy takes $O(w^h)$ time units, where h is the height of the QEP tree (see [4] for more details). Similar numbers can be deduced for STAIRs if the Promote and Demote operations are eagerly performed upon plan transition. For both techniques, during the plan transition stage, the execution is suspended, leading to the drawbacks highlighted in Section 3.2.

5.1.2 Thrashing Avoidance

In addition to maintaining steady query output, the on-demand strategy of state computation employed by JISC leads to several advantages over the state of the art plan migration techniques. Consider a dynamic scenario in which the selectivities of the operators in the QEP keep fluctuating, leading to multiple successive plan transition decisions. In this case:

- The Parallel Track Strategy will keep running multiple (more than two) QEPs, and hence the performance of query execution can severely degrade. In addition, the cost of duplicate elimination will multiply.
- The Moving State Strategy will perform many migrations. In each migration, all the states of the new QEP are eagerly computed without necessarily being used. Thrashing will occur. A huge effort will be consumed in these computations with no subsequent payoff.

JISC avoids the above problems by computing the missing states on demand, i.e., performing lazy migration (refer to Section 4.5).

5.2 Throughput During a Transition Stage

Consider the plan transition in Figure 5 in which there is a change in the order of the joins with streams R and S. The upper and lower subtrees remain the same. According to Definition 1, both subtrees have complete states at all their operators, and only one incomplete state exists at the join operator of S.

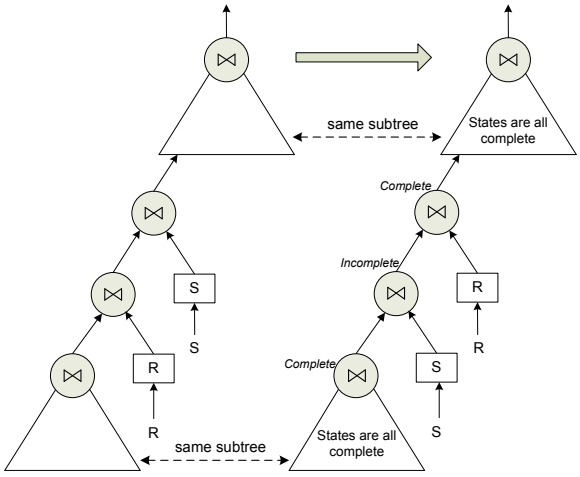

Figure 5: A transition to a plan that has two unchanged subtrees with complete states at all operators.

Since CACQ does not store any state, all intermediate results (join states) do not exist and have to be recomputed for every input tuple. For the Parallel Track Strategy, the execution will proceed with both the old and new plans as long as there are old entries in the old plan. However, since the subtree is unchanged, many redundancies occur in computing the states of the unchanged subtrees. JISC avoids this redundancy by detecting that all the states in the unchanged subtrees are complete, and hence require no additional work. This, in turn, implies a potentially open-ended gain in the performance of JISC compared to CACQ and the Parallel Track Strategy, as the complete subtrees can have an arbitrarily large number of operators and arbitrarily large window sizes.

It is clear that JISC achieves good performance gains when the plan transition results in less number of incomplete states. The question that we address now is: *"What is the average number of complete states that can result after a plan transition?"* Consider the plan transition in Figure 6. According to Definition 1, the middle subtree will have incomplete states, while the upper and lower subtrees will have complete states. Observe that the number of incomplete states depends on the position of the streams that exchanged positions in the QEP, e.g., streams R and S in Figure 6.

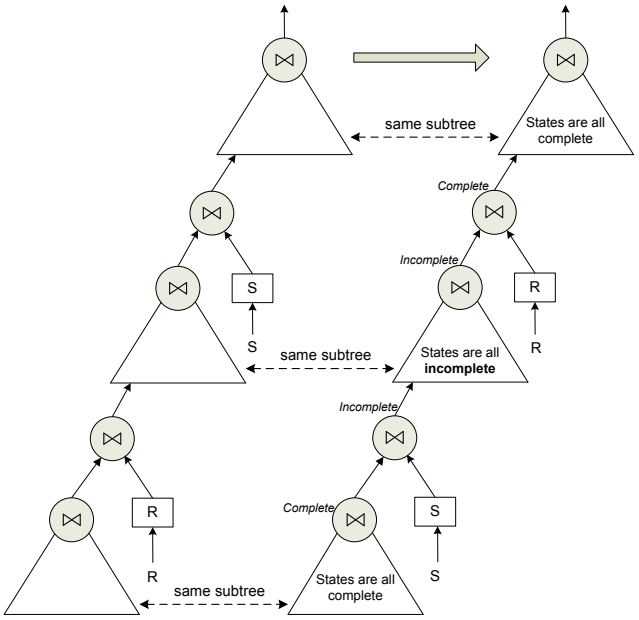

Figure 6: An arrangement of complete/incomplete states right after a typical plan transition.

In a data streaming environment, operator selectivities are estimated before query execution. Query execution starts with a QEP that has the operators ordered based on these estimates. In particular, the QEP is set to have the most selective operators (joins) at the bottom of the plan, i.e., joins in the new plan are somehow inversely sorted according to their estimated selectivities. Since these estimates can be far from reality, during execution, the query optimizer decides to switch the order of some joins based on the runtime feedback.

Without loss of generality, we consider pairwise join exchanges, i.e., only two streams exchange positions in the QEP, e.g., the joins of streams R and S in Figure 6. A set of join exchanges of any size can be mapped to a set of pairwise join exchanges. Assume a left-deep QEP with n operators, joining $n + 1$ streams. We label the streams in a bottom-up fashion with the labels $\{1, 2, 3, \ldots, n\}$, with the streams of the leaf join having the same label, $\{1\}$, as we are assuming symmetric hash join. We also label each internal node (representing an operator), with the label of its right child (stream), and think of the labels as "positions" up the chain.

Our key insight in the remaining of the analysis is that the positions of the streams to be swapped are usually close. In other words, the probability that a stream at the top of the plan is swapped with a stream at the bottom of the plan is usually very low. The reason is that the initial setup of the QEP is based on selectivity estimates

that are usually not too far from the reality. Even if there is some inaccuracy in these estimates, this inaccuracy may affect the order of the operators that are not too far from each others, otherwise there must be fatal error in the estimates. Even if there are fatal errors in the estimates, or if the initial positions are chosen at random, after a warm-up sequence of plan transitions, the QEP will be stable and the positions to be swapped during further execution will be close.

Assume that the positions of the streams that exchange positions are chosen at random, say I and J, where $I < J$. Here "random" refers to a triangular probability distribution, that diminishes proportionately according to the difference $J - I$. We choose this distribution to coincide with the insight discussed above. In other words, for a pair (i, j), with $i < j$,

$$\mathbf{Prob}(I = i, J = j) = \frac{1}{j - i} \times \alpha_n, \tag{1}$$

where the proportionality factor must be

$$\alpha_n = \frac{1}{n \cdot H_n - n}, \tag{2}$$

where $H_n = \sum_{r=1}^{n} 1/r$ is the n^{th} harmonic number [20].

The number of incomplete states is $J - I$ (refer to Figure 6, where positions I and J represent the positions of streams R and S in the new plan, respectively). Thus the number of complete states, say C_n is

$$C_n = n - (J - I). \tag{3}$$

PROPOSITION 1.

$$\mathbf{E}[C_n] = \frac{2nH_n - 3n + 1}{2H_n - 2},$$

$$\mathbf{Var}[C_n] = \frac{2n^2 H_n - 5n^2 + 6n - 2H_n - 1}{12(H_n - 1)^2}.$$

Proof.

$$\begin{aligned}
\mathbf{E}[C_n] &= n - \sum_{1 \le i < j \le n} (j - i)\, \mathbf{Prob}(I = i, J = j) \\
&= n - \sum_{1 \le i < j \le n} (j - i)\, \frac{\alpha_n}{j - i} \\
&= n - \alpha_n \binom{n}{2}.
\end{aligned}$$

The mean formula follows as given after simplification. The variance is

$$\begin{aligned}
\mathbf{Var}[C_n] &= \mathbf{Var}[J - I] \\
&= \sum_{1 \le i < j \le n} (j - i)^2\, \mathbf{Prob}(I = i, J = j) \\
&\quad - \big(\mathbf{E}[J - I]\big)^2 \\
&= \sum_{1 \le i < j \le n} (j - i)\, \alpha_n - \alpha_n^2 \binom{n}{2}^2 \\
&= \frac{n^2 - 1}{6H_n - 6} - \alpha_n^2 \binom{n}{2}^2.
\end{aligned}$$

The variance in the form given follows after some cancellations. □

PROPOSITION 2. *As* $n \to \infty$,

$$\mathbf{E}[C_n] = n - \frac{n}{2 \ln n} + O\Big(\frac{1}{\ln n}\Big),$$

$$\mathbf{Var}[C_n] = \frac{n^2}{6 \ln n} + O\Big(\frac{n^2}{\ln^2 n}\Big).$$

Proof. This asymptotic approximation ensues after using the well known asymptotic relation $H_n = \ln n + \gamma + O(1/n)$, where γ is Euler's constant $0.577215665\ldots$. □

The variance is not too large, which gives us a concentration law.

PROPOSITION 3.

$$\frac{C_n}{n} \xrightarrow{\mathcal{P}} 1$$

Proof. Let $\varepsilon > 0$ be fixed. By Chebyshev's inequality [21], we write

$$\mathbf{Prob}\Big\{\big|C_n - \mathbf{E}[C_n]\big| > \varepsilon\Big\} \le \frac{\mathbf{Var}[C_n]}{\varepsilon^2}.$$

Replace ε by $\varepsilon\, \mathbf{E}[C_n]$, and write the inequality as

$$\mathbf{Prob}\Big\{\Big|\frac{C_n}{\mathbf{E}[C_n]} - 1\Big| > \varepsilon\Big\} \le \frac{\mathbf{Var}[C_n]}{(\varepsilon\, \mathbf{E}[C_n])^2} = O\Big(\frac{1}{\ln n}\Big) \to 0.$$

Hence $\frac{C_n}{\mathbf{E}[C_n]} - 1$ converges in probability to 0 (that is, $\frac{C_n}{\mathbf{E}[C_n]} \xrightarrow{\mathcal{P}} 1$). We also know that $\frac{\mathbf{E}[C_n]}{n} \to 1$. Multiplying the latter two convergence relations, we get the result; see [21] for this type of manipulation. □

The result shows that there is high probability for C_n, the number of complete states, to stay near n. That is, when we move to an alternative plan via JISC, most of the states are complete, and only little work is needed to complete a few (a sublinear number of) incomplete states. Thus, JISC is robust.

6. EXPERIMENTAL STUDY

In this section, we study the performance of JISC compared to CACQ [3], STAIRs [19], as well as the Moving State and the Parallel Track Strategies [4]. Since JISC adds *no* memory overhead (cf. Section 5), except for the counters used for state completion detection, our main performance measure is the execution time.

We implemented the Eddies, SteMs, and CACQ techniques as in [22, 17, 18, 23, 3, 19]. We also implemented the Moving State and Parallel Track Strategies as in [4]. All implementations are in Java. Experiments were conducted on a machine running Windows 7 with Intel Core2 Duo CPU at 2.1 GHz and 4 GB of main memory.

We generate QEPs with different numbers of joins. We uniformly generate the data and uniformly distribute it across the different streams. For instance, after generating 10 million tuples for a QEP with 10 streams, each stream's share would be around 1 million tuples. Unless stated otherwise, the window size corresponding to each stream is 10,000 tuples.

6.1 Performance during a Transition Stage

In the following experiments, we measure the execution time of the different plan migration strategies during the migration stage. We force a plan transition while executing the queries after processing 10 million tuples. To have a consistent comparison among the strategies, we process tuples until the old plan of the Parallel Track Strategy is discarded, i.e., the migration stage ends. Then, we process the same tuples using both JISC and CACQ. We measure the execution time each strategy takes to process these tuples.

Figure 7 gives the performance of the three strategies during the plan migration stage for different queries with different numbers of joins. In this experiment, we address the best case for JISC, in which the new plan has only one incomplete state as illustrated in Figure 5. JISC has the best performance, especially for plans with large number of joins, in which the speedup reaches an order of magnitude compared to the Parallel Track Strategy.

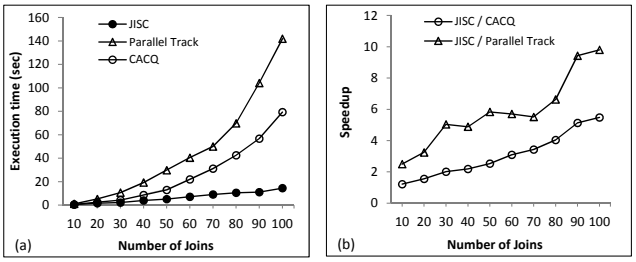

Figure 7: Performance during the plan migration stage (best case). (a) Running time. (b) speedup.

Figure 8 gives the result of a similar experiment for JISC's worst case scenario, i.e., when the transition results in all the states of the new plan being incomplete. Notice that the speedup of JISC decreases compared to that of Figure 7 due to the overhead of state completion. The performance of both CACQ and the Parallel Track Strategy is the same in Figures 8 and 7 because both strategies do not differentiate between a complete and an incomplete state.

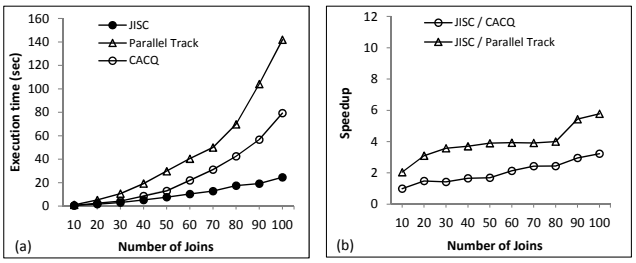

Figure 8: Performance during the plan migration stage (worst case). (a) Running time. (b) speedup.

6.2 Overhead During Normal Operation

The following experiment illustrates how JISC performs during normal operation of a query (before or after a plan transition), i.e., when all the states are complete. We generate a plan with 20 joins. We execute 10 million tuples on JISC as well as a plan with pure symmetric hash joins. This is equivalent to comparing JISC with the Parallel Track Strategy, since, during normal operation, only one plan will be running in the Parallel Track Strategy. We also compare JISC to CACQ by processing the same tuples.

Figure 9 gives the performance of the three strategies during normal operation. JISC introduces minimal overhead to the symmetric hash join plan. Moreover, JISC is nearly twice as fast as CACQ because, in the latter, each tuple gets processed by the eddy operator as many times as for the join operators.

6.3 Latency

In this experiment, we measure the time it takes from the moment a plan transition is triggered until the first output tuple is produced, i.e., the output latency. In JISC, the output latency is negligible compared to that of the Moving State strategy. In the latter, execution completely stalls until all the states of the new plan are computed, and hence the latency is very high especially for QEPs with nested-loops joins. Figure 10 ((a) for hash join and (b) for nested-loops join) gives the output latency of both JISC and the Moving State Strategy at different window sizes in log-scale for a QEP of 20 joins. QEPs with higher numbers of joins were experimented and produce similar performance figures. Furthermore, similar performance figures can be obtained for the output latency of STAIRs as it applies a greedy state migration policy.

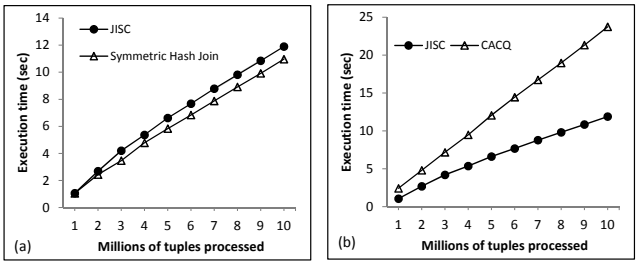

Figure 9: Performance during normal operation. (a) JISC vs. Symmetric Hash Join (Parallel Track Strategy). (b) JISC vs. CACQ.

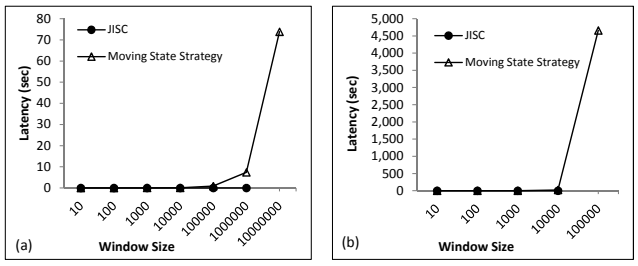

Figure 10: Output latency due to a plan transition. (a) QEP of hash joins. (b) QEP of nested-loop joins.

It is clear from Figure 10 that JISC has minimal output latency. The output latency for the Moving State Strategy is relatively low for QEPs of hash joins, i.e., it is in the order of seconds even for large window sizes. However, for QEPs of nested-loops joins (i.e., for general theta joins), the output latency is very high. For example, in Figure 10, for window sizes of 100,000 tuples, the latency is about 4600 seconds (76 minutes). If the figure is extrapolated for window sizes of 1,000,000 tuples, the latency will be in the order of hours and days. Due to this latency, the Moving State Strategy (and similarly for STAIRs) is not suitable for applications that require frequent plan transitions (applications involving streams with fluctuating selectivities and stream rates).

6.4 Frequency of Plan Transition

In the following experiments, we show the effect of varying the frequency of plan transition on the performance of JISC. We generate a QEP with 20 joins. We force plan transitions at different frequencies, i.e., we force a plan transition every 1, 2, . . ., 10 million tuples. We generate 20 million tuples so that a plan transition occurs at least twice for all frequencies. Figures 11 and 12 give the execution times when the plan transitions result in incomplete states at all operators (worst case) and in only one incomplete state just below the root operator (best case), respectively.

Figures 11 and 12 illustrate that JISC achieves higher throughput than both CACQ and the Parallel Track Strategy at any plan transition frequency. Since CACQ operates the same way irrespective

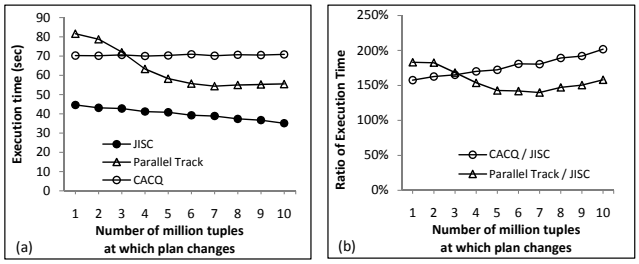

Figure 11: Performance at various plan transition frequencies. States of new plan are all incomplete (worst case).

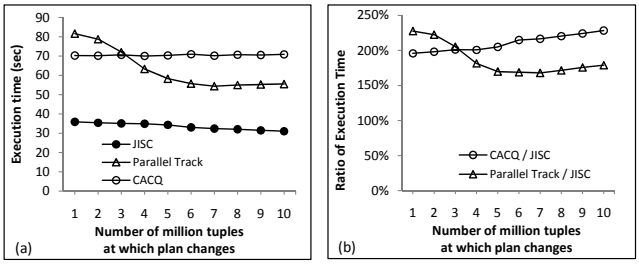

Figure 12: Performance at various plan transition frequencies. New plan has only one incomplete state (best case).

of plan transition, it is not affected by the plan transition frequency. The parallel Track Strategy achieves the worst performance when the rate of plan transition is high (every 1 or 2 million tuples). The reason is that the old plan is not discarded along with the needed duplicate elimination component on top of both the old and new plans. Notice that JISC achieves lower throughput at high plan transition rates because the incomplete states never get completed, and require additional processing. From Figures 11 and 12, it is clear that CACQ and the parallel Track Strategy have almost the same performance. However, the slope of the the curve for JISC's execution time is lower in Figure 12 due to the existence of many complete states that do not require overhead.

7. CONCLUSIONS

JISC is a new technique for plan adaptation of continuous queries over data streams. It is applicable to both pipelined and eddy-based QEPs. During plan transition, JISC maintains steady query output in contrast to existing plan adaptation techniques that can exhibit significant output latencies. Maintaining steady query output is essential for applications that require continuous monitoring or real time response.

JISC employs a lazy state migration strategy that computes the missing states in the new QEP on-demand in a single QEP. In highly dynamic scenarios where the queries and streams have fluctuating selectivities and arrival rates, overlapped plan transitions are imminent. The lazy migration strategy enables JISC to avoid performance thrashing, which is a vital issue in all other existing techniques.

A key property of JISC is that during plan transition, it detects the unchanged parts of the QEP and avoids recomputing their states. Our probabilistic analysis demonstrates that the number of operators with unchanged states in the new QEP is usually close to the total number of operators. For that reason, JISC outperforms existing techniques that also maintain steady query output, e.g., the Parallel Track Strategy, leading to performance gains during a plan migration stage that can reach up to an order of magnitude.

8. REFERENCES

[1] A. Deshpande, Z. G. Ives, and V. Raman, "Adaptive query processing," *Foundations and Trends in Databases*, 2007.

[2] G. Cormode and M. N. Garofalakis, "Streaming in a connected world: querying and tracking distributed data streams," in *SIGMOD Conference*, 2007.

[3] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman, "Continuously adaptive continuous queries over streams," in *SIGMOD Conference*, 2002.

[4] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman, "Dynamic plan migration for continuous queries over data streams," in *SIGMOD Conference*, 2004.

[5] J. Krämer, Y. Yang, M. Cammert, B. Seeger, and D. Papadias, "Dynamic plan migration for snapshot-equivalent continuous queries in data stream systems," in *EDBT Workshops*, 2006, pp. 497–516.

[6] Y. Yang, J. Krämer, D. Papadias, and B. Seeger, "Hybmig: A hybrid approach to dynamic plan migration for continuous queries," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 3, pp. 398–411, 2007.

[7] L. Golab and M. T. Özsu, "Processing sliding window multi-joins in continuous queries over data streams," in *VLDB*, 2003.

[8] L. Ding, K. Works, and E. A. Rundensteiner, "Semantic stream query optimization exploiting dynamic metadata," in *ICDE*, 2011.

[9] J. Chen, D. J. DeWitt, and J. F. Naughton, "Design and evaluation of alternative selection placement strategies in optimizing continuous queries," in *ICDE*, 2002.

[10] S. Viglas and J. F. Naughton, "Rate-based query optimization for streaming information sources," in *SIGMOD Conference*, 2002.

[11] S. Viglas, J. F. Naughton, and J. Burger, "Maximizing the output rate of multi-way join queries over streaming information sources," in *VLDB*, 2003.

[12] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik, "Aurora: a new model and architecture for data stream management," *VLDB J.*, 2003.

[13] A. N. Wilschut and P. M. G. Apers, "Dataflow query execution in a parallel main-memory environment," *Distributed and Parallel Databases*, 1993.

[14] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid, "Scheduling for shared window joins over data streams," in *VLDB*, 2003.

[15] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid, "Incremental evaluation of sliding-window queries over data streams," *IEEE Trans. Knowl. Data Eng.*, 2007.

[16] T. M. Ghanem, A. K. Elmagarmid, P.-Å. Larson, and W. G. Aref, "Supporting views in data stream management systems," *ACM Trans. Database Syst.*, 2010.

[17] R. Avnur and J. M. Hellerstein, "Eddies: Continuously adaptive query processing," in *SIGMOD Conference*, 2000.

[18] V. Raman, A. Deshpande, and J. M. Hellerstein, "Using state modules for adaptive query processing," in *ICDE*, 2003.

[19] A. Deshpande and J. M. Hellerstein, "Lifting the burden of history from adaptive query processing," in *VLDB*, 2004.

[20] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Addison-Wesley Professional, Mar. 1994.

[21] A. Karr, *Probability*. Springer, New York, 1993.
[22] A. Deshpande, "An initial study of overheads of eddies," *SIGMOD Record*, vol. 33, pp. 44–49, March.
[23] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah, "Telegraphcq: Continuous dataflow processing for an uncertain world," in *CIDR*, 2003.

APPENDIX

Correctness of JISC

For JISC to be valid, it has to guarantee that the output corresponding to an input tuple is the same whether the plan changes or not. In this section, we provide a proof of correctness for JISC.

Before we proceed with the proof, we note that in JISC, the position at which a tuple enters the new plan relative to the position at which it should have entered the old plan does not affect the way the tuple is processed. What really affects the processing of a tuple are the following two factors: (1) whether the tuple probes a complete or an incomplete state (according to Definition 1), (2) whether the tuple is a *fresh* or an *attempted* tuple (according to Definition 2). We illustrate this fact by the following scenarios.

Consider the plan-transition in Figures 3a (old plan) and 3b (new plan). Since all the intermediate states are incomplete, any *fresh* tuple that enters the new plan at any height (except the root operator) requires state completion. For example, consider a *fresh* tuple from Stream S that enters the new plan (Figure 3b) at Height 2, i.e., the second join in the pipeline from the bottom. That same tuple would have entered the old plan (Figure 3a) at Height 1, i.e., the lowest join in the pipeline from the bottom. In this case, state completion for that tuple is required since it probes an incomplete state in the new plan. However, consider the plan-transition in Figures 3a (old plan) and 3c (new plan). A tuple from Stream U enters the old plan at Height 3. When a *fresh* tuple from Stream R enters the new plan at Height 2, which is lower, no state completion is required, since it probes a complete state.

We prove that JISC is ***Complete***, i.e., does not miss any valid output tuple, ***Closed***, i.e., does not produce any invalid output tuple, and ***Duplicate-free***, i.e., every valid output tuple is produced exactly once. The proofs assume that all the binary operators in a query execution pipeline are joins that are executing Procedure 1. Refer to Definitions 1 and 2 for the terms *complete*, *incomplete*, *fresh*, and *attempted*.

THEOREM 1. After a query plan-transition takes place, for any input tuple, JISC does not miss any output that would have been produced if the transition does not take place.

PROOF. Assume a tuple, say t, is received, and would result in the output o_{miss} if the current execution plan does not change. Assume that a plan-transition takes place and the output o_{miss} is missed (i.e., is not produced as output). At the point t enters the pipeline, the state of the opposite operator is probed (Line 1). Execution proceeds with one of the following cases:

- Case 1: If the opposite state is complete, the state entries corresponding to t will be found, and joined tuple(s) will pass to the next operators until the output o_{miss} is produced.
- Case 2: If the state of the opposite operator is incomplete and t is *fresh*, the state entries corresponding to t will not be found and will be completed (Line 6) and then t will be able to join (Lines 7 and 8). The joined tuple(s) will pass to the next operators until the output o_{miss} is produced. Moreover, all the states of the intermediate joins will have complete entries corresponding to the join attribute value of t, so that the next received tuples having the same join attribute value (i.e., attempted tuples) will require no further state completion overhead.
- Case 3: If the state of the opposite operator is incomplete and t is *attempted*, the state entries corresponding to t will be found, even if the state is incomplete. Since t is *attempted*, it is guaranteed to have complete state entries at all the intermediate joins after the first tuple having its join attribute value (i.e., *fresh* tuple) is processed as in Case 2. Thus, the joined tuple(s) will pass to the next operators until the output o_{miss} is produced.

From Cases 1, 2, and 3, we conclude that the assumption that o_{miss} is missed is invalid. Thus, JISC does not miss any valid output tuple, i.e., JISC is **complete**. □

THEOREM 2. After a query plan-transition takes place, for any input tuple, JISC does not produce any output tuple that would not result if the transition does not take place.

PROOF. Assume an update tuple, say t, is received, and does not result in any output tuples if the current execution plan does not change. Assume that a plan-transition takes place and the output o_{wrong} is produced.

Since there should be no output tuple produced if the transition does not take place, at some point (join operator) in the pipeline, t should not join. At that point, t is checked with the state of the opposite operator and nothing is found. Execution proceeds with one of the following cases:

- Case 1: If the state of the opposite operator is complete, then execution stops, and o_{wrong} is not produced.
- Case 2: If the state of the opposite operator is incomplete and t is *attempted*, then execution stops (since the corresponding state entries are guaranteed to exist as discussed in the above proof), and o_{wrong} is not produced.
- Case 3: If the state of the opposite operator is incomplete and t is *fresh*, state completion is applied (Line 6), but nothing is added to the state of the opposite operator. This is because Procedures 2 or 3 add state entries to the join-state only when the join succeeds (Lines 1 and 7 in Procedure 2, or Line 3 in Procedure 3). Thus, t will not join (Line 7 in Procedure 1 returns *false*), and o_{wrong} is not produced.

From Cases 1, 2, and 3, we conclude that the assumption that the output o_{wrong} is produced is invalid. Thus, JISC does not produce any invalid output tuple, i.e., JISC is **closed**. □

In JISC, once a plan-transition is decided, the input queues to all the query operators have to be cleared, i.e., all the tuples that are received before a plan-transition is decided have to be processed through the old plan, and pushed up the QEP until they reach the output. This way, a tuple is allowed to be processed by an operator at most once. Afterwards, the old plan is discarded, and the new plan takes place. Then, tuples that are received after the plan-transition are processed through the new plan. This means that every input tuple is processed only once, either in the old plan, or in the new plan. Thus, the tuple's corresponding output should appear at most once, i.e., with no duplication. This argument leads to the conclusion that JISC is **duplicate-free**, as the following Theorem states.

THEOREM 3. Before or after a query plan-transition takes place, JISC does not produce any output more than once.