Dynamic Plan Migration for Continuous Queries Over Data Streams *

Yali Zhu, Elke A. Rundensteiner and George T. Heineman
Department of Computer Science, Worcester Polytechnic Institute,
100 Institute Road, Worcester, MA 01609
{yaliz, rundenst, heineman}@cs.wpi.edu

ABSTRACT

Dynamic plan migration is concerned with the on-the-fly transition from one continuous query plan to a semantically equivalent yet more efficient plan. Migration is important for stream monitoring systems where long-running queries may have to withstand fluctuations in stream workloads and data characteristics. Existing migration methods generally adopt a pause-drain-resume strategy that pauses the processing of new data, purges all old data in the existing plan, until finally the new plan can be plugged into the system. However, these existing strategies do not address the problem of migrating query plans that contain stateful operators, such as joins. We now develop solutions for online plan migration for continuous stateful plans. In particular, in this paper, we propose two alternative strategies, called the moving state strategy and the parallel track strategy, one exploiting reusability and the second employs parallelism to seamlessly migrate between continuous join plans without affecting the results of the query. We develop cost models for both migration strategies to analytically compare them. We embed these migration strategies into the CAPE [7], a prototype system of a stream query engine, and conduct a comparative experimental study to evaluate these two strategies for window-based join plans. Our experimental results illustrate that the two strategies can vary significantly in terms of output rates and intermediate storage spaces given distinct system configurations and stream workloads.

1. INTRODUCTION

Many applications require the monitoring of data streams using standing queries, including sensor networks, stock and medical monitoring systems [2–4, 16, 19]. In those systems, data may stream in from several often distributed network locations, with unpredictable changes in arrival rates and in value distributions. Queries posed over such streaming data are usually long-running and an originally selected query plan may later become sub-optimal or even produce poor performance due to these changes. A stream query engine must cope with such changing characteristics of the streaming environment.

On-the-fly query re-optimization, one critical technology focusing on addressing this problem, has attracted much recent research attention [2,3,5,13,18,21]. Such a solution usually takes two steps. First, the optimizer dynamically selects a new yet equivalent query plan based on system statistics gathered at run time. This is referred to as the dynamic query optimization process. Then the system needs to be migrated from the query plan that it is currently running to the semantically equivalent yet more efficient plan that the optimizer has chosen. We refer to the latter process as *dynamic plan migration*.

A migration strategy must guarantee that it will not alter the results produced by the system during as well as after the plan transition. This correctness implies that results are neither missing nor contain duplicates. Traditionally, a dynamic plan migration strategy [3], even if supposedly dynamic, takes the following steps: 1) pause the execution of the current query plan, 2) drain out all existing tuples in the current query plan, 3) replace the current plan with the new plan, and restart the execution. We refer to this traditional approach as *pause-drain-resume* strategy. The purpose of the draining step is to free the intermediate tuples in the query plan so to prevent any missing output tuples.

The *pause-drain-resume* migration strategy may be adequate to dynamically migrate a query plan that consists of only *stateless* operators, such as select and project. A *stateless* operator does not need to maintain intermediate data nor other auxiliary state information so to be able to generate complete and correct results. So all intermediate tuples in such a stateless query plan exist only in intermediate queues and can be cleaned completely by the drain step during the migration process. On the contrary, a *stateful* operator, such as join, must store all tuples that have been processed thus far from one input stream so to be able to join them with future incoming tuples from the other input stream. For a long-running query as in the case of continuous queries, the number of tuples stored inside a stateful operator, such as a join operator, can potentially be infinite. Several strategies have been proposed to limit the number of intermediate tuples kept in operator states by purging unwanted tuples, including a join with window-based constraints [3, 10, 15, 19] and a join with punctuation-based constraints [8, 20]. In all the above strategies the purge of the

*The research was partly supported by the RDC grant 2003-04 on "On-line Stream Monitoring Systems: Untethered Healthcare, Intrusion Detection, and Beyond."

SIGMOD 2004 June 13-18, 2004, Paris, France.

old tuples inside the state is always driven by the processing of new tuples or new punctuations from input streams.

It is important to note that for a query plan that contains *stateful* operators such as joins, intermediate tuples may exist in both intermediate queues and in operator states. And as noted above, the purge of tuples in the states relies on the processing of new data. However, in the *pause-drain-resume* migration strategy, before embarking on the drain step, the execution of the query plan is paused so that no new tuples beyond the intermediate tuples are being processed until the migration is over. This creates a **deadlock in the migration process**: *the migration is waiting for all old tuples in operator states to be purged from the old plan, while the old tuples in those states are waiting for new tuples to be processed in order to be purged.* This problem has not yet been addressed in the literature, and now is the topic of our current work.

In this paper, we propose two plan migration strategies for continuous queries over streaming data, namely the *moving state strategy* and the *parallel track strategy.* Both strategies deal with the migration of a query plan that contains stateful operators, in particular join operators.

The moving state strategy first pauses the execution of the query plan and drains out tuples inside intermediate queues, similar to the above *pause-drain-resume* approach. However, to avoid loss of any useful data inside states, it then carefully maps and moves over all relevant tuples in the states of the old query plan to their corresponding location in the new query plan. This is shown in Section 4.1.2 to be insufficient and thus we also engage in selectively recomputing intermediate tuples. The execution of the query plan is then resumed with the new plugged-in plan. Since no results are outputted during the migration stage, the output stream may experience a duration of temporary silence.

For applications that desire a smooth and constant output, we design a second migration strategy called the parallel track strategy. This strategy migrates in a more gradual fashion by continuing delivering output tuples even during migration. Instead of moving tuples to the new query plan and discarding the old query, it plugs in the new query plan and starts executing both query plans in parallel. We develop algorithms to eliminate potential duplicates and maintain the appropriate order of output tuples. Once the old plan is found to be "antiquated", it can simply be disconnected and the migration stage is then over.

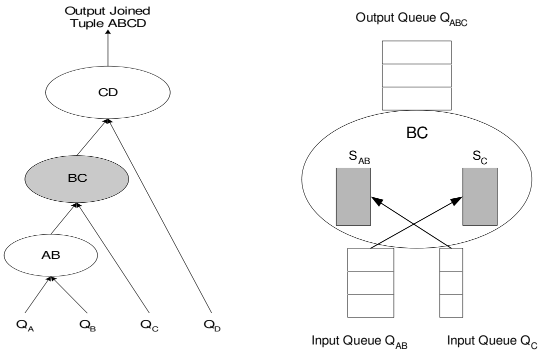
We analyze the performance of these two migration strategies using a cost model. We also have implemented both strategies within the CAPE stream system [7]. This enabled us to conduct an experimental study (not just simulation) comparing these strategies under a variety of stream workloads. Our experimental results show that for smaller window constraints, the moving state strategy migrates more quickly, while for larger window constraints the inverse can be observed. During the migration stage, they exhibit different behaviors in terms of output rates and intermediate storage usage. One strategy may be more suitable than the other depending on the system resource limitations.

The rest of this paper is organized as follows. Section 2 establish the foundations, including time-related properties and window-based state purging algorithms. We give the problem definition in Section 3, with our two migration strategies described in Section 4. We develop a cost-based model and analytically compare the migration strategies in Section 5. Section 6 is devoted to the experimental results, followed by a discussion of related work in Section 7. In Section 8 we draw our conclusions.

2. STATE PURGING ALGORITHMS

2.1 Window Join Operators

We employ a symmetric window-based binary join algorithm [11, 22] as commonly used for join operators in continuous queries [3, 15, 19]. Without loss of generality, we adopt time-based (not tuple-count-based) sliding window constraints. A sample query plan for the query $A \bowtie B \bowtie C \bowtie D$ that consists of three join operators with input streams A, B, C and D is depicted in Figure 1(a). The join operator $B \bowtie C$ in Figure 1(b) has two input queues Q_{AB} and Q_C, two *states* S_{AB} and S_C, one associated with each input queue, and one output queue Q_{ABC}. Each *state* stores the tuples that fall within the current time window from its associated input queue. For each tuple AB from Q_{AB}, the join involves three steps: 1) purge – AB is used to purge tuples in state S_C, 2) join – AB is joined with the tuples left in S_C, and 3) insert – AB is inserted into state S_{AB}. The same process applies similarly to any tuple from Q_C. We call this 3-step process as *purge-join-insert* algorithm.



(a) (b)

Figure 1: Join Operators and Their States

Join states are limited by the sizes of window constraints. A window constraint W_{AB} posed over joining streams A and B indicates that two tuples from streams A and B respectively can be joined only if their timestamps are within W_{AB} from each other. For simplicity in the rest of the paper we assume same (global) time window constraint as in [4] on all pairs of streams, as the example illustrated in Figure 2 for a query $A \bowtie B \bowtie C \bowtie D$, $W_{AB} = W_{BC} = W_{CD} = W_{AD} = W_{AC} = W_{BD}$. In general, the window constraints among join pairs may be different or even unconstrained. However, our migration framework could be easily extended to handle such more relaxed window constraints[1] and due to space limitations this is not described here.

A time-based window requires that each newly arriving tuple has a timestamp. Within each stream entering the leaves of the query plan, the tuples are assumed to be ordered by their timestamps [4, 15, 19]. A tuple has a single timestamp when it first arrives in the stream, referred to as a *singleton tuple.* When two tuples are joined together,

[1]For these situations, the window constraint between any pair is the *shortest path* between the pair in a window constraint relationship graph as the one in Figure 2.

the timestamp for the joined tuple is an array that concatenates the timestamps from both joining tuples, as indicated in Figure 3. Both timestamps are kept because either of them might be used by other join operators in the query plan to purge tuples. We call such a tuple with a combined timestamp a *combined tuple*.

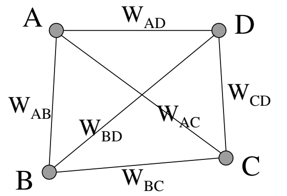


Figure 2: Graph on Window Constraints

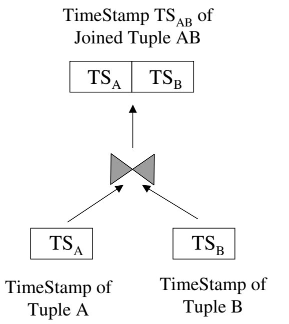


Figure 3: Combined Timestamp

2.2 Purge by Singleton Tuples

The algorithm to purge a state by a singleton tuple is straightforward. For a join operator $A \bowtie B$ with window size W, since tuples from stream A are strictly ordered, a B tuple in state S_B is purged by an A tuple if and only if $(TS_A - TS_B) > W$. A similar algorithm can also be applied when using a singleton tuple to purge a state with combined tuples: If a C tuple is used to purge state S_{AB}, a combined AB tuple is purged by the C tuple if $(TS_C - TS_A) > W$ or $(TS_C - TS_B) > W$.

2.3 Purge by Combined Tuples

Purging a state by a combined tuple is more complex than purging by a singleton tuple, because a combined timestamp has multiple columns and may not be ordered by any of these timestamp columns. By utilizing the same purge algorithm described above, some tuples may be purged by earlier complex tuples even though they may still have the potential to join with future incoming tuples. We thus need a generic and safe purging algorithm for both singlton and complex tuples.

2.4 Timestamp Order

Although a sequence of tuples with combined timestamps is not strictly ordered by any one of its timestamps, some *Timestamp Order* can still be observed.

Lemma 2.1 (Timestamp Order Lemma). *Let t and t' be two tuples in the output queue of a binary window join operator. Both tuples have timestamps of size n, represented as $[TS_1, ..., TS_n]$ and $[TS'_1, ..., TS'_n]$ respectively. If tuple t appears earlier than tuple t' in the queue, then there must exist at least one i ($1 <= i <= n$), such that $TS_i < TS'_i$.*

Proof: We now give a proof by induction on the size of timestamp array n. Suppose that the window join operator has two input queues Q_L and Q_R, two states S_L and S_R, and one output queue Q_{LR}. t and t' are tuples in Q_{LR}.

***Base case*: n = 2.** Let $[TS_1, TS_2]$ and $[TS'_1, TS'_2]$ be the timestamps of tuples t and t' respectively. Tuples with a combined timestamp array of size 2 must be formed by joining two sub-tuples each with a timestamp of size 1. So t is formed by joining t1 with timestamp TS_1 and t2 with timestamp TS_2. And t' is formed by joining t1' with timestamp TS'_1 and t2' with timestamp TS'_2. Without loss of generality, let us assume that t1 and t1' are from Q_L, and t2 and t2' from Q_R. All tuples in Q_L and Q_R are singleton tuples and are strictly ordered by their timestamps.

Since tuple t comes before t' in Q_{LR}, t must have been generated earlier than t'. When sub-tuples t1 and t2 are about to be joined to generate tuple t, two cases are possible: 1) t1 is the *first* tuple in Q_L and t2 is inside S_R, or 2) t1 is inside S_L and t2 is the *first* tuple in Q_R. At this time, sub-tuples t1' and t2' cannot both be in states. Because otherwise they must have been joined already and tuple t' would appear before t in Q_{LR}. So if sub-tuple t1' with timestamp TS'_1 is not yet in S_L, then it is or will arrive in Q_L. In this case we have $TS_1 < TS'_1$ and i = 1. If sub-tuples t2' with timestamp TS'_2 is not in state S_R, then it is or will arrive in Q_R. So $TS_2 < TS'_2$ and i = 2.

From above we conclude that for base case n = 2, there always exists an i such that $TS_i < TS'_i$.

Inductive Hypothesis: Assume that the timestamp order lemma holds for any tuple sequence with size $n <= k$.

Inductive Step: We now show that the timestamp order lemma also holds for sequences with size $n = k + 1$.

The timestamp array for t with size n = k + 1 can be treated as a combination of two sub-tuples t1 and t2 with timestamp arrays as $[TS_1, ..., TS_j]$ and $[TS_{j+1}, ..., TS_{k+1}]$, respectively. Similarly, t' can also be treated as the combination of two sub-tuples t1' and t2' with timestamp array as $[TS'_1, ..., TS'_j]$ and $[TS'_{j+1}, ..., TS'_{k+1}]$, respectively. Since each array is at least of size 1, it must be true that $j <= k$. So both timestamp arrays have a size of at most k.

Using the same reasoning as in the base case, when sub-tuples t1 and t2 are about to be joined to generate tuple t, at least one sub-tuple t1' or t2' does not yet exist in its respective join state. If sub-tuple t1' with timestamp $[TS'_1, ..., TS'_j]$ is not in state S_L, then it is or will arrive in Q_L. Since t1' must come after t1 in Q_L, based on *Induction Hypothesis*, we know that there exists an m ($0 < m <= j$) such that $TS_m < TS'_m$. So in the case i = m. If sub-tuple t2' with timestamp $[TS'_{j+1}, ..., TS'_{k+1}]$ is not in state S_R, then it is or will arrive in Q_R. Since t2' comes after t2 in Q_R, we can again find $i = m$ ($j < m <= k + 1 = n$) such that $TS_m < TS'_m$.

So we conclude that the lemma holds for any tuple sequence in a query plan. □

By utilizing the timestamp order lemma, we now describe the general purge algorithm to safely purge tuples by either a singleton tuple or a combined tuple. We attach a min-max timestamp pair $[TS_{min}, TS_{max}]$ to each tuple, corresponding to the smallest and largest timestamps in its timestamp array. For a singleton tuple, TS_{min} equals TS_{max}.

Lemma 2.2 (Purging Lemma). *Assuming that timestamp order holds for any tuple sequence including queues and states[2] in the query plan, given two tuples t_L (with n timestamps) and t_R (with m timestamps) that have min-max timestamp pairs $[TS_{min_L}, TS_{max_L}]$ and $[TS_{min_R}, TS_{max_R}]$ respectively, if $(TS_{min_L} - TS_{max_R}) > W$, then t_R can be purged from its state by t_L.*

Proof: We need to show that t_R can be safely purged because it can no longer be joined with any tuple that arrives

[2]In the case of hash join, tuples belong to the same hash bucket are assumed to be ordered by their insertion time.

after t_L in that sequence. Because the timestamp order holds for any tuple t'_L arriving after t_L in the same sequence, there exists an i $(0 <= i <= n)$ such that $TS_{i_L} < TS'_{i_L}$. Since TS_{min_L} is the smallest timestamp in the timestamp array of tuple t_L, we know that $TS_{min_L} <= TS_{i_L}$. Thus $TS_{min_L} < TS'_{i_L}$. Now for t_R, given any $j(0 <= j <= m)$ we have $TS_{j_R} <= TS_{max_R}$. Since we know that $(TS_{min_L} - TS_{max_R}) > W$, putting above together, we can get $(TS'_{i_L} - TS_{j_R}) > W$. Since the global window constraint is assumed in any join pair, for any tuple t'_L that comes after tuple t_L in the same sequence, it is outside the W window frame from tuple t_R. So we conclude it is safe to purge t_R. □

The above general purging lemma works for both singleton and combined tuples. To our best knowledge, our timestamp order lemma and purging algorithm are the first algorithms to explicitly deal with the purging of a combined tuple with multiple timestamps. In the case of singleton tuples, our purging algorithm is essentially the same to the commonly used purge algorithm [3, 15, 16, 19]. In some literatures [4, 6, 19], a combined tuple bears only one timestamp. This timestamp can be either the combined tuple's output time from an operator [19], or the minimal timestamp [4] or the maximal timestamp [6] of the combined tuple. In these cases the purging of combined tuples is simplified to be the same as purging of singleton tuples.

3. PROBLEM DEFINITION

We use the term *box* to refer to the plan or sub-plan selected for migration. Each *box* consists of a set of operators that together represent a valid connected query sub-plan. It can be as large as the complete plan or as small as one operator. Each box can have several *box root* operators each associated with a *box output queue*, and several *box leaf* operators each associated with a *box input queue*. *Box intermediate queues* connect operators inside a box. A queue inside our query plan can have multiple operators as its producers that append new tuples to the end of the queue, and multiple operators as consumers that fetch tuples from the top of the queue. Such a shared queue stores one cursor for each consumer that points to the position of the tuple that this consumer would fetch next.

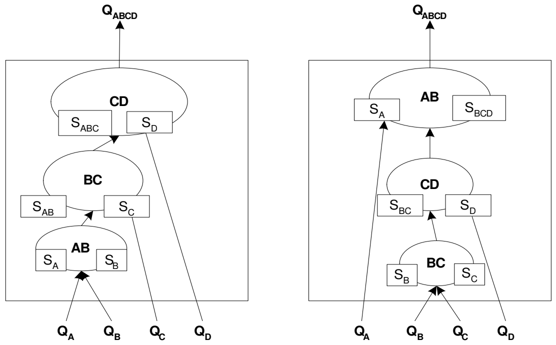


Figure 4: Two Exchangeable Query Boxes

The migration problem can then be defined as the process of transferring an old box containing the old query plan to a new box containing the new query plan. The old and new query plans must be equivalent to each other, indicating that the old and new boxes have the same sets of box input and output queues, as shown in Figure 4.

For a migration strategy to be valid, several requirements need to be satisfied. First, it is crucial that the migration strategy will not alter the correctness of the results of the query plan. The same set of results, without any tuples missing and without duplicates, should be generated for a query plan independent from whether or not the migration has been applied during its execution.

A more subtle problem is the *order* of the results. The dynamic migration may change the structure of the query plan, for example, switch the order of join operators, and hence the order of the output tuples may also be affected. In particular, the change of tuples' order inside a box intermediate queue may affect the processing of other upstream operators in the query plan. For instance, as described in Section 2.4, a join operator may rely on a certain timestamp order to purge tuples in the states. If the tuples' order is disturbed by a migration strategy, the consequences may be that some tuples may be purged from the states even if they may still be useful to future tuples. Our general purge algorithm is based on the timestamp order property described in Lemma 2.1. By preserving this property, although the *exact* order of the tuples may have been changed by the dynamic migration, the purge of the join operators can still be executed correctly. In view of this, it is crucial to maintain the timestamp order property. We thus aim to design our migration strategies to guarantee this property.

4. MIGRATION STRATEGIES

We denote the time period of each online plan migration process as *migration stage*, with the migration start time as T_{M_start} and the migration end time as T_{M_end}. During the migration stage, we refer to the states in the old box as *old states*, and states in the new box as *new states*. All tuples existing in the old box at T_{M_start} are called *old tuples*, and any tuple entering old and new boxes after that time point are called *new tuples*. That is, it is not the system time that determines a tuple's old or new status, but rather the location of the tuple at T_{M_start}. If a tuple enters the old box any time during the migration stage, although it has arrived in the system or has been generated before T_{M_start}, it is still treated as a new tuple by the old box. A combined tuple that has any of its sub-tuple marked as *old* is referred to as an *old* tuple, since it still has some contents that exist in the old box at T_{M_start}. A combined tuple is considered a *new* tuple only if all its sub-tuples are *new*.

4.1 Moving State Strategy

The basic idea of the *moving state strategy* is to safely move old tuples in old states directly into the states in the new box without losing any useful data. In this section, we detail the necessary steps of the moving state strategy, including *state matching*, *state moving* and *state recomputing*.

4.1.1 State Matching and Moving

State matching determines the pairs of states, one in the old and one in the new box, between which tuples can be safely moved. Two states can move tuples in between them if and only if they contain tuples with the same schema. In our query plans, a tuple's schema is defined by all its column IDs. We define a state's ID as the same to its tuple's schema, and all tuples in one state have the same schema. If two states have the same state ID, we say that those two states are *matching states*. In Figure 5, states (S_A, S_B, S_C, S_D) exist in both boxes and are matching states. States (S_{BC}, S_{BCD})

appear in the new box only, and states (S_{AB}, S_{ABC}) appear in the old box only. These are thus unmatched states.

After the *state matching*, we can then take the *state moving* step to move tuples between all pairs of matching states. A naive state moving method is that for all matching states, we directly move the tuples from the old state to its matching new state. This method, although correct, is a waste of both time and storage. An improved method is to share a state using the queue sharing technique described in Section 3. We create a new cursor for each matching new state that points to the first tuple in its matching old state, indicating that all tuples in the old state are now shared by both matching states. The cursors for the old matching states are then dereferenced to complete this state moving process.

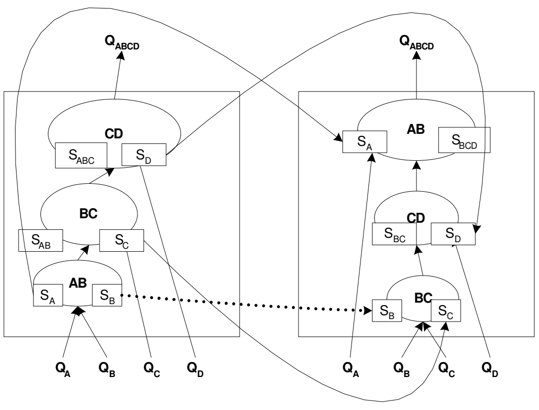


Figure 5: Moving State Strategy

4.1.2 State Recomputing

Two questions remain regarding the unmatched states in both old and new boxes: 1) Can we leave the unmatched states in the new box empty? 2) Can we throw away the old tuples inside the unmatched states in the old box?

To answer the first question, we need to determine whether or not the complete set of results can be generated if the unmatched states in the new box are left empty. We again use the migration example shown in Figure 5, with the old box on the left and the new box on the right. Each ABCD tuple in the output queue Q_{ABCD} can be treated as a combination of four *sub-tuples* A, B, C and D, originally from Q_A, Q_B, Q_C, and Q_D respectively. We divide all the possible outcomes of tuple ABCD based on the *old/new* status of their sub-tuples. Figure 6 shows a list of all 16 possible cases with their case #.

We now show that by leaving the unmatched states in the new box empty, tuples in some of the 16 cases may be lost. Figure 7 depicts the status of the new box right after the state matching and moving steps. We show each tuple inside the states and input queues by its sub-tuples' *old/new* status. The two unmatched states S_{BC} and S_{BCD}, both empty, are shaded grey.

Assume that now we discard the old box and start executing the new query plan with the unmatched states being empty. In the join operator $B \bowtie C$ in Figure 7, only *new* B tuples can be joined with *old* or *new* C tuples in S_C.[3] Also, only *new* C tuples can be joined with *old* or *new* B tuples in S_B. Hence only combined BC tuple with its two sub-tuples' old/new status as (new, old), (old, new) or (new, new) can be generated by the join operator $B \bowtie C$ and later be inserted into state S_{BC}. The combination (old, old) would never be generated and inserted into S_{BC}. This means that among the 16 cases in Figure 6, cases #1, #2, #5 and #9 cannot be generated by the query plan after migration, because those cases all require that both sub-tuples B and C are *old*. The same kind of problem occurs when leaving the other unmatched state S_{BCD} empty.

By leaving unmatched states in the new box empty, we lose the *all-old* combinations of sub-tuples in these states. This leads to the loss of some result tuples as shown in the example above. So before restarting the execution of the query plan, some computations need to be undertaken first for the unmatched states in the new box in order to gain back those *all-old* combinations. We refer to this step as *state recomputing*. We have designed a recursive algorithm shown in Algorithm 1 to compute the unmatched states in the new box. It is designed for binary join operators to keep it simple, but could easily be modified to suit multiple-input join operators as well.

Case #	*A*	*B*	*C*	*D*
1	Old	Old	Old	Old
2	Old	Old	Old	New
3	Old	Old	New	Old
4	Old	New	Old	Old
5	New	Old	Old	Old
6	Old	Old	New	New
7	Old	New	Old	New
8	New	New	Old	Old
9	New	Old	Old	New
10	Old	New	New	Old
11	New	Old	New	Old
12	Old	New	New	New
13	New	Old	New	New
14	New	New	Old	New
15	New	New	New	Old
16	New	New	New	New

Figure 6: Possible Old/New Status for Tuples in Output Queue ABCD

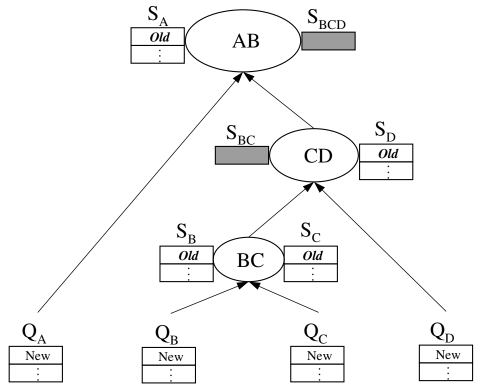


Figure 7: Empty Unmatched States in the New Box

4.1.3 Safe State Discarding

Now we need to address the question if it is safe to discard the old tuples inside those unmatched states in the old box. As for the example in Figure 5, we have to determine if we can discard the old tuples in states S_{AB} and S_{ABC} inside the old box on the left. To answer this question we need to

[3]In Figure 7 S_C only contains *old* tuples. However, each *new* C tuple inserted into S_C may have been joined with B tuples, and after a while the state S_C may contain both *old* and *new* tuples.

know if any of those *old* tuples in the unmatched old states may have the potential to join with any *new* tuples.

Taking the unmatched old state S_{AB} in the old box as an example, clearly it can be discarded if the following condition holds: All sub-tuples A and B of the AB tuples in S_{AB} also exist in states S_A and S_B respectively. This is because the states S_A and S_B are already shared by the new states in the new box. This way no data would be lost by discarding the unmatched old state S_{AB}. However, we can show that the above condition cannot be guaranteed. For example, inside the join operator $A \bowtie B$ in Figure 5, some tuples A and B in S_A or S_B may have already been purged by newer tuples from the input queue Q_B and Q_A. Before these tuples are being purged from S_A and S_B, they may have already joined with other B and A tuples and the joined AB tuples may have already been inserted into S_{AB}. Hence *not* all sub-tuples A and B in S_{AB} are necessarily present in S_A and S_B. After the state matching, moving and recomputing state, if we discard the unmatched old state S_{AB}, some tuples in state S_{AB} that may still be able to join with a new tuple C may then be lost. Then the results of the query plan may be incomplete.

Algorithm 1 Recomputing unmatched states

```
During matching state step, mark a state as "matched" if
it has a matching state.
To start, set current_op = new box root operator
recompute_states(current_op)
while current_op has more state do
   get the next state S_i of current_op;
   get the child operator child_op that has its output queue
   associated with S_i;
   if child_op is not new box operator then
      continue;
   end if
   if S_i is unmatched then
      get child_op_l_state;
      get child_op_r_state;
      if either state of child_op is unmatched then
         recompute_states(child_op);
      end if
      S_i = window_join(child_op_l_state, child_op_r_state);
      mark S_i as "matched";
   end if
   recompute_states(child_op);
end while
```

We can still discard the unmatched old states if the following condition is met: The tuples in an unmatched old state that contain sub-tuples that do not exist in any matched states are impossible to join with any future incoming tuples. We again use the example in Figure 5 to illustrate how this condition can be met. It is clear that at T_{M_start}, the query plan in the old box has only processed tuples with timestamp up to T_{M_start}. In the unmatched old state S_{AB}, all the sub-tuples A and B that do not exist in S_A and S_B must have timestamps earlier than $(T_{M\_start} - W)$ (otherwise they cannot have been purged from S_A and S_B). So if we can guarantee that all new C tuples from Q_C have a timestamp larger or equal to T_{M_start}, then they are not able to join with any tuples in S_{AB} that contain sub-tuples A and B that do not exist in S_A and S_B.

Caused by either a scheduling algorithm or the availability of system resources, tuples may not be processed immediately after they arrive and may rather be accumulated in the input queues. So at T_{M_start}, tuples in Q_C may have timestamps earlier than T_{M_start}. If we want to safely discard any unmatched states in the old box, one practical method is to finish processing all the accumulated tuples in the old box's input queues that have arrived before T_{M_start}. This works fine if all the old box input queues are stream input queues, which means they are input queues to the leaf operators. However, if the old box contain only a sub-tree of the complete query tree and the box input queues are not the stream input queues of the whole query plan, we need to identify all the queues (from box input queues down to the stream input queues) that may have some contribution in terms of forwarding tuples to the old box. The accumulated tuples that arrive before T_{M_start} in the involved stream queues then need to be processed and pushed up the query tree until reaching the output queue of the old box.[4][5] The method cleanAccumulatedTuples() completes this task. Its psuedo-code is omitted here for space reasons.

4.1.4 Overall Moving State Algorithm

Putting all the pieces together, we now show the complete algorithm for our moving state strategy in Algorithm 2.

Algorithm 2 Moving State Migration

```
cleanAccumulatedTuples();
connect input and output queues of old and new boxes;
match_states(old_box, new_box);
move_states(old_box, new_box);
recompute_states(root_op_of_new_box);
disconnect old box from current query plan;
start executing query plan with new box;
```

Once the moving state migration starts, after cleanAccumulatedTuples(), no new results are produced until the steps of matching, moving and recomputing states are finished. The length of this output silence is closely related to the amount of tuples that need to be moved or recomputed during the migration stage. This duration of output silence may be less desirable for applications that are in favor of a more steady output rate. To solve this problem, we design the second migration strategy, the parallel track strategy, to continuously deliver outputs even during the migration stage.

4.2 Parallel Track Strategy

The basic idea for the *parallel track migration strategy* is that at the migration start time, the input queues and output queue are connected and shared between the old box and the new box, using the *queue sharing* technique depicted in Section 3. Both boxes are then being executed in parallel, while waiting for all old tuples in the old box to be gradually

[4] We have further optimized this step by finding the largest timestamp of the first tuple (or T_{M_start}, whichever is smaller) in all the involved stream input queues, and push up all accumulated tuples in these queues that have timestamps no later than this largest timestamp.

[5] We have also developed another method to gain back all the sub-tuples in the unmatched states that do not exist in the matched states and insert them back to the matched states. Due to space limits, detailed discussion is omitted.

purged. During this process, new outputs are still being continually produced by the query plan.

When the old box contains only *new* tuples, it is safe to discard the old box. This is because all old tuples have finished their duty in terms of contributing to the generation of output results from the old box. Since we have been executing the new box in parallel with the old box when the migration first starts, all the new tuples now in the old box exist in the new box as well.

4.2.1 Correctness of the Results

Correctness of the results involves two aspects: the outputs are complete and do not contain duplicates. We use the example in Figure 4 to show that by going through parallel track migration to transfer the query plan from the left to the right, all 16 possible sub-tuple combinations of any output tuple ABCD, as listed in Figure 6, can still be obtained. In our parallel track strategy, both old and new boxes are running in parallel until all the tuples with *old* status are purged from the old box. By this time, the output tuples that contain any old sub-tuple, as in the cases #1-#15, have already been generated by the old box, either before (case #1) or during the migration stage (cases #2-#15). Since the new box starts its execution right after T_{M_start}, its states are initially all empty, and all the *new* tuples fed into the old box are also being processed by the new box. All output tuples from the new box will have all their four sub-tuples marked as *new*, reflecting case #16 in Figure 6. Thus all 16 cases are covered by either the old box or the new box.

4.2.2 Duplicate Elimination

We must also ensure that no duplicate tuples are being generated. If we use the parallel track strategy described above, although the old box will cover all 15 cases consisting of at least one *old* sub-tuple, it may also generate the all-new sub-tuple combination belonging to case #16 in Figure 6, duplicate to the output results from the new box.

To solve this duplication problem, a naive approach would be to discard from the old box *any* tuples with all-new sub-tuples. However, this method is too aggressive and will lose some must-have tuples. For example in the join operator $B \bowtie C$, we cannot discard any combined tuple AB from input queue Q_{AB} with both sub-tuples A and B marked as *new*, because this AB tuple may still be able to join with an *old* C tuple in state S_C, and generate output tuples that belong to either case #8 or case #14 in Figure 6. Even if the AB tuple ends up joining with a *new* C tuple, the joined tuple ABC, with all its sub-tuples marked as *new*, may still join with an *old* D tuple in state S_D. So the final joined tuple ABCD belongs to case #15, which can only be generated by the old box.

Thus the root join operator of the old box is the only safe place to eliminate duplicates. This is done by preventing a *new* tuple from joining with another *new* tuple. Hence if two tuples that are about to join are both *new*, we simply skip the join step in the regular purge-join-insert symmetric join algorithm. The purge and insert steps are however still undertaken as usual.

4.2.3 Timestamp Order Preservation

As described at the end of Section 3, the timestamp order must be preserved to ensure that the correct results are being generated. During the parallel migration stage, both the old and the new box share the same output queue into which both will insert output tuples. Keeping the timestamp order of the tuples in the output queue requires that both the old and the new box coordinate with each other to output tuples in the proper order.

Two characteristics of our parallel migration strategy assist in developing a valid method for preserving timestamp order. First, since each box is executed as a valid sub-query plan, the timestamp order among the output tuples from each box is preserved. Secondly, any output tuple from the old box will be guaranteed to have at least one sub-tuple being *old*, and all output tuples from the new box will have all sub-tuples as *new*. This means that any tuple generated by the new box will have at least one of its timestamps to be later than any tuple generated by the old box.

Taking advantage of those two characteristics, we develop an easy yet effective method to preserve the timestamp order in the parallel track strategy. During the migration stage while both boxes are executing, we only output tuples generated by the old box into the shared output queue. Any output tuples generated by the new box are instead held in a temporary buffer. When the old box is removed, all output tuples held in the temporary buffer are then inserted into the output queue all at once.

4.2.4 Overall Parallel Track Algorithm

As described above, although join operators in both boxes are executed in parallel during the migration stage, besides the regular join operation, they may have other tasks to finish: The old box root operator needs to avoid joining two *new* tuples to prevent duplicate results, and the new box root operator needs to hold any results during the migration stage in a temporary buffer to preserve the timestamp order. We use the $W\_Join()$ method for the regular purge-join-insert symmetric window join algorithm described in Section 1. The methods used by the operators in the old box and the new box are referred to as $W\_Old\_Join()$ and $W\_New\_Join()$ respectively.

Algorithm 3 Parallel Track Strategy

```
Pause execution of old box at T_M_start;
Connect input and output queues of old and new boxes;
Start a separate thread to run Monitor_Old_Box();
while No signal from thread Monitor_Old_Box() do
  Old operators run W_Join_Old();
  New operators run W_Join_New();
end while
Disconnect old box from current query plan;
Operators in new box resume running W_Join();
```

To determine when to finish the migration, each operator has an IF_FINISHED flag initialized to be false. During the migration stage, each operator in the old box checks at intervals to see if all *old* tuples have been purged from its states. Once this is the case the operator sets its IF_FINISHED flag to be true. The system also runs a light-weighted monitor method called $Monitor\_Old\_Box()$ in a separate thread to check at intervals all the IF_FINISHED flags of the operators in the old box. If an all-true scan is detected, it sends a signal to the main thread to tell it to finish the migration by disconnecting the old box from the current query plan. The complete algorithm for the parallel track migration strategy is shown in Algorithm 3.

Table 1: Terms Used in Cost Model

Term	Meaning
N	Number of operators in the old box
M	Number of operators in the new box
T_m	Time spent for each string comparison
T_c	Time spent to create a new cursor
λ_A	Average tuple input rate from Q_A
λ_B	Average tuple input rate from Q_B
σ_{AB}	Reduction factor of join operator $A \bowtie B$
W	Global time window constraint
T_j	Time spent to join a pair of tuples
T_s	Time spent to insert one tuple into a state
$\|S_A\|$	Number of tuples in state S_A
$\|S_B\|$	Number of tuples in state S_B

5. COST ANALYSIS

In this section, we describe cost models for estimating the migration length and the system processing time required by each migration strategy.

5.1 Analysis of Moving State Strategy

To estimate how long it takes to finish a moving state migration, we need to add up the time spent on each migration step, including clean accumulated tuples, state matching and moving, and state recomputing. The cost model utilizes the binary nested-loop join algorithm with global window constraint for simplicity, but it can easily be extended to cover other join algorithms. We also assume that the system has enough computing power and memory resources to keep up with the query processing without much delay given the incoming data load.

Given the sufficient-system-resources assumption, new tuples are generally being processed immediately without being accumulated in the input queues. So the time spent on the cleanAccumulatedTuple() method is likely to be small compared to other migration steps and is thus not counted in the model. The time spent on state matching and moving is related to the total number of states in both boxes. State matching is basically a string matching between two lists of state IDs. Moving a state is creating a new cursor to a state so to enable its sharing between two matching states. Thus its costs are minimal.

A list of terms and their meanings used in our model are listed in Table 1. The time spent on state matching T_{match} and state moving T_{move} can be calculated as below. Here we use the minimum of N and M to estimate the number of matching state pairs.

$T_{match} = 4NMT_m$ and $T_{move} = 2min(N, M)T_c$

In order to estimate the time spent on the state recomputing step, we develop a general model to estimate the time to recompute a single state. This model can then be applied to each state that needs to be recomputed to get the total recomputation time. Assume we have a join operator $A \bowtie B$ with two input queues Q_A and Q_B, two states S_A and S_B, and one output queue Q_{AB}. Without loss of generality, the tuple A and B each can be either a singleton or a combined tuple. Suppose that the state S_{AB} needs to be recomputed. This is done by joining tuples from S_A and S_B using the purge-join-insert symmetric join algorithm but skipping the purge step. The time spent on this recomputing process can be formulated as: $T_{S_{AB}} = T_j|S_A||S_B| + T_s|S_A||S_B|\sigma_{AB}$.

Given the time window W and input rates from inputs A and B, the state sizes of S_A and S_B, represented as $|S_A|$ and $|S_B|$, can be estimated as: $|S_A| = \lambda_A W$, and $|S_B| = \lambda_B W$.

Putting the above formulae together, we get the time for recomputing S_{AB} from S_A and S_B as:

$$\begin{aligned} T_{S_{AB}} &= T_j\lambda_A\lambda_B W^2 + T_s\lambda_A\lambda_B W^2\sigma_{AB} \\ &= \lambda_A\lambda_B W^2(T_j + T_s\sigma_{AB}) \end{aligned} \quad (1)$$

If another unmatched state above S_{AB} needs to be recomputed, according to Equation 1, the output rate λ_{AB} is then required. This can be estimated using Equation 2.

$$\lambda_{AB} = \lambda_A|S_B|\sigma_{AB} + \lambda_B|S_A|\sigma_{AB} = 2\lambda_A\lambda_B W\sigma_{AB} \quad (2)$$

If we denote T_S as the total time spent on recomputing all unmatched states in the new box, the total migration length of the moving state strategy T_{MS} can be estimated using the following model:

$$T_{MS} = T_{match} + T_{move} + T_S \quad (3)$$

5.2 Analysis of Parallel Track Strategy

We denote T_{PT} as the length of the migration stage for the parallel migration strategy. For this strategy, all old tuples (tuples with at least one old sub-tuple) need to be purged from the old box in order to finish the migration stage. Suppose that h ($h >= 1$) is the height of the query tree inside the old box. We analyze the time spent on the parallel track migration stage in two cases:

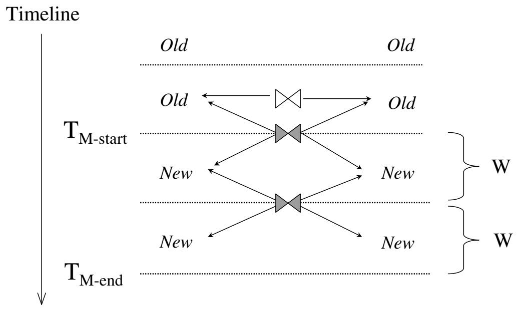


Figure 8: 2W to purge all old tuples

1) h = 1. In this case the query tree has only one level of join operators. For a join operator inside the old box to purge all old tuples from one of its two states, the join operator must process new tuples from another input that arrive in the next W time units. Given that the system has enough computing power, $T_{PT} = W$.

2) $h > 1$. This means that in the old box there is at least one join operator which is above another join operator. Figure 8 depicts the old and new tuples along a timeline. The migration start and end time is marked beside the timeline. From the figure we can see that when the migration begins, W time window's new tuples from the box input queues are needed to purge old tuples inside the states of box leaf operators. However, as these new tuples are used to purge old tuples, they may also join with some of the old tuples and the results are being inserted into the state of the join operators above the box leaf operators. Because the joined tuples contain an *old* sub-tuple, they are treated as *old* tuples and need to be purged as well. In order to do so, the old box needs to process another W time window's new tuples to completely purge these *old* tuples from the old box. So in this case, $T_{PT} = 2W$.

Other even older tuples may exist in the old box when the migration first starts, represented by the first line of "old" in Figure 8. These tuples will be purged by the first W new tuples after migration starts, and will not be able to join with any of the new tuples.

As a summary, given sufficient system processing power, T_{PM} has a linear relationship with the global window size W. It can be formulated as:

$$T_{PT} = \begin{cases} W & if \quad h = 1 \\ 2W & if \quad h > 1 \end{cases} \tag{4}$$

Equation 4 shows that in order to complete a parallel track migration, both old and new boxes need to process at most 2W worth of new tuples. However, this is valid only when the system has enough processing power so that a tuple arrives in the system can be processed immediately. If the system processing power is not sufficient, the actual migration length may be longer than 2W. We now give the cost model to estimate the cost of system processing time during the parallel track migration. As in the cost analysis for the moving state strategy, we have developed general formulae to estimate the processing time spent on any join operator (let us denote it as $A \bowtie B$) in the old box and any join operator (let us denote it as $B \bowtie C$) in the new box.

We first compute T_{AB}, the total cost of processing tuples in 2W timeframe in operator $A \bowtie B$ inside the *old box*. It is easy to see that for each new tuple A, the average number of tuples B that will be purged from state B is $\lambda_b \frac{1}{\lambda_a}$, and vice versa. The same method in Equation 2 can be applied to compute the tuple output rate from operator $A \bowtie B$.

$$\begin{aligned} T_{AB} &= Cost\ of\ Purge + Cost\ of\ Insert + Cost\ of\ Join \\ &= 2W[T_s(\frac{\lambda_a}{\lambda_b}\lambda_b + \frac{\lambda_b}{\lambda_a}\lambda_a + \lambda_a + \lambda_b) + T_j(\lambda_a|S_B| + \lambda_b|S_A|)] \\ &= 2W[2T_j\lambda_a\lambda_b W + 2T_s(\lambda_a + \lambda_b)] \end{aligned} \tag{5}$$

One major difference between operators inside the old and the new boxes is that the states of operators inside the new box all start empty. The sizes of the states keep on increasing with no tuples being purged until the W*th* time unit, after which tuples begin to be purged and the state size on average is limited by the window size W. This leads to different methods of computing processing cost and tuple output rate for a join operator inside the *new box*. These are described in Equations 6 and 7.

$$\begin{aligned} T_{BC} &= Cost\ for\ the\ first\ W + Cost\ for\ the\ second\ W \\ &= W[T_s(\lambda_b + \lambda_c) + T_j(\lambda_a \int_0^W \lambda_b t dt + \lambda_b \int_0^W \lambda_a t dt)] \\ &+ W[2T_j\lambda_a\lambda_b W + 2T_s(\lambda_a + \lambda_b)] \end{aligned} \tag{6}$$

$$\lambda_{BC} = \begin{cases} \int_0^t 2\lambda_b\lambda_c\sigma_{bc} t dt & if \quad t \le W \\ 2\lambda_b\lambda_c\sigma_{bc} W & if \quad t > W \end{cases} \tag{7}$$

6. EXPERIMENTAL RESULTS

6.1 Experimental Setup

We embed our migration strategies into the CAPE system [7] and conduct various experiments to compare their performance. We use the query in Figure 4 as the foreground query on which the migration is performed to transfer the plan from the left to the right. System parameters such as stream input rates, operator reduction factors and global time window are varied to reflect the changes of workload and data characteristics. The query engine also simultaneously executes multiple background queries with their system parameters kept stable.

Our stream data generator generates tuples with arrival patterns modeled as the Poisson process. The mean inter-arrival delay between two consecutive tuples is exponentially distributed in order to model the Poisson arrival pattern. In each experiment, the stream generator continuously generates streams for 50,000ms. All query plans are being executed for a time period much longer than the global window in order to pass the warm-up phase. A migration strategy is then activated by the change of system parameters for the foreground query plan.

All implementation is done in Java. The experiments were run on a machine running windows 2000 with Pentium-III processor at 500MHz and 384M of main memory.

6.2 Length of Migration Stage

In this section, we analyze the experimental results related to the measured length of the migration stage and compare them with the estimation models described in Section 5.

Both old and new query plans in Figure 4 have a height h = 3. According to the Equation 4 in Section 5.2, the total length of the migration stage of the parallel track strategy should be $T_{PT} = 2W$.

Given the same query plans, by applying the Equations 1, 2 and 3 from Section 5.1, we can estimate the length of the migration stage for the moving state strategy as:

$$\begin{aligned} T_{MS} &= \lambda_B\lambda_C W^2(T_j + T_s\sigma_{BC}) \\ &+ 2\lambda_B\lambda_C\lambda_D W^3(T_j\sigma_{BC} + T_s\sigma_{BC}\sigma_{BCD}) \end{aligned} \tag{8}$$

From the above results, we see that T_{PT} grows linearly with W. However, T_{MS} is controlled by several parameters, including input rates from Q_B, Q_C and Q_D, reduction factors σ_{BC} and σ_{BCD}, and the global time window W, with which it has a polynomial relationship.

The above estimations are based on the assumption that the system has enough processing power to handle incoming tuples without much delay. We judge the availability of system processing power in our experimental setup by comparing the total system running time vs. the stream generator running time. In our experiment, the stream generator in each experiment runs for 50,000 ms, generating stream tuples according to the given mean inter-arrival time. The system stops executing the query plan when there are no more tuples to process. If the system finishes at about 50,000ms as well, it implies that the system has enough processing power to keep up with the given parameter configurations.

To verify these estimations on the length of the migration stage, we run three sets of experiments:

1) Set 1: Only W increases linearly, while all other parameters are kept constant.

2) Set 2: I_B, the tuple inter-arrival time of stream B, is decreased (indicating that input rate λ_B is increased) while keeping all other parameters the same.

3) Set 3: W is increased linearly while other parameters are kept the same. The difference from set 1 and 3 is that set 3 has higher configurations with respect to input rates and operators' reduction factors.

Table 2: Parameter Configurations

Parameters	Section 6.2			Section 6.3	
	set1	set2	set3	set1	set2
W(ms)	vary	1000	vary	1000	2000
$I_A(ms)$	100	50	100	100	50
$I_B(ms)$	100	vary	12	100	50
$I_C(ms)$	100	50	12	100	50
$I_D(ms)$	100	50	12	100	50
σ_{AB}	0.1	0.1	0.1	0.1	0.2
σ_{BC}	0.05	0.05	0.1	0.02	0.05
σ_{CD}	0.02	0.02	0.1	0.02	0.05

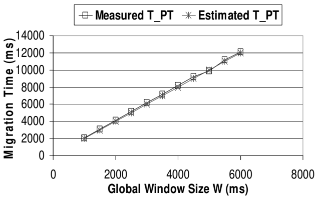


Figure 9: T_{PT} vs. W

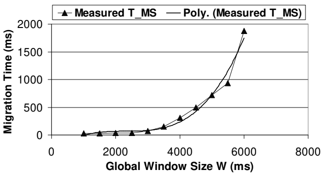


Figure 10: T_{MS} vs. W

Figures 9 and 10 depict the results of the first experimental set. Figure 9 illustrates that T_{PT} has a linear relationship with W and is statistically equivalent to 2W, as is suggested by the Equation 4 in the case of $h > 1$. The increasing curve of T_{MS}, marked as "Measured T_{MS} in Figure 10, indicates a close to polynomial relationship with W. A polynomial trendline marked as "Poly.(Measured T_{MS})" is depicted as well.

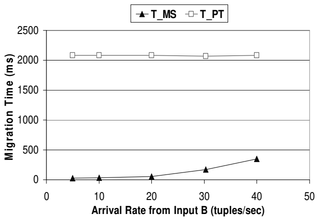


Figure 11: T_{MS} and T_{PT} vs. λ_B

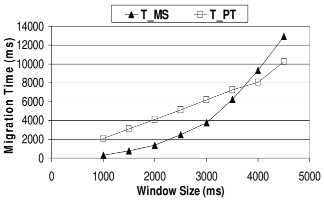


Figure 12: Comparison of T_{MS} and T_{PT} vs. W

Figure 11 displays the results of set 2 when increasing the input rate λ_B. It shows that the increase of λ_B has almost no effect on T_{PT}, which is fairly stable. However, when λ_B increases, T_{MS} increases as well.

The results of the experimental set 3 are depicted in Figure 12. At small window constraint sizes, the moving state strategy migrates faster because the state sizes limited by the window constraint are small. As the window size increases, the parallel track migration time increases linearly, and stays at about 2W. Since the total migration time for the moving state strategy has a polynomial relationship with the window size, the gap between the two lengths of migration stages is getting smaller. After a certain window size, the parallel track strategy surpasses the moving state and becomes the faster one of these two strategies.

6.3 Effects on Minimizing Intermediate Data

A common goal for a query optimizer is to minimize a query plan's intermediate data. This is usually achieved by pushing the operators with the smallest reduction factors down the query plan tree. In this section, we study the performance of both migration strategies working with an optimizer that has such an optimization goal.

We have conducted two sets of experiments with the parameters' configurations shown in Table 2. Parameters in set 1 are set to be low to create the situation of sufficient system computing resources, while set 2 configures parameters to their high values to model the scenario that the system computational power is not sufficient to process the old query plan. Hence, a large delay for processing new tuples is expected for the second set. In all the experiments, we start migrating the old plan to the new plan after the old plan has been executed for 10,000ms.

The results of the first experimental set with a low configuration are shown in Figures 13 and 14. Each graph depicts the results for four different cases: 1) the moving state strategy (MS), 2) the parallel track strategy (PT), 3) the new query plan only (New), and 4) the old query plan only (Old).

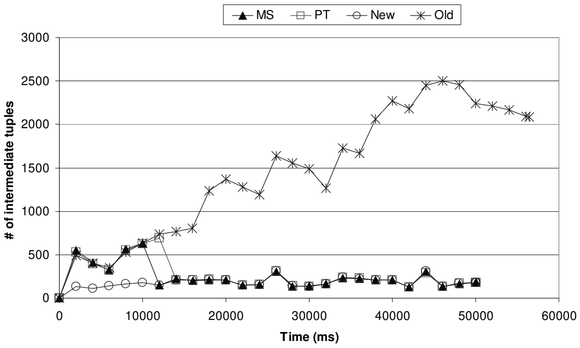


Figure 13: Intermediate Tuple Counts - Low Config

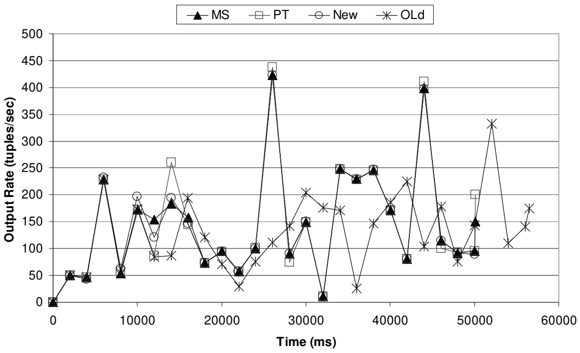


Figure 14: Output Rate - Low Config

Figure 13 shows the intermediate tuple counts for the above four cases. The new plan has a much smaller intermediate tuple count than the old plan throughout the experiment. At the first 10,000ms, the three lines overlap each other indicating that they have the same performance. However, starting from around 10,000ms, two plans are migrating to the new plan each using one of the migration strategies. When given sufficient system processing power, which usually indicates a small window size, the moving state strategy starts to have the same intermediate tuple count as the new plan case earlier than the parallel strategy. This is because it usually migrates to the new plan faster given a smaller window size. Both plans going through two different migration strategies eventually have the same intermediate tuple count as the one running the new plan only.

Figure 14 depicts the four cases with respect to their output rates. No strong advantage can be observed for either migration strategy. This may be due to the fact that the migration stage under a low configuration is usually short. So even the parallel track strategy keeps on producing new tuples during the migration stage, the plan using the moving state strategy is able to migrate to the new plan faster so the output silence is short enough to be neglected.

The situation changes for the second experimental set with a high configuration. Figures 15 and 16 show the results of all four cases. Since the system has insufficient processing power to keep up with the old query plan, the new query plan as well as the query plan with migration both out-perform the old query plan dramatically in all experimental results.

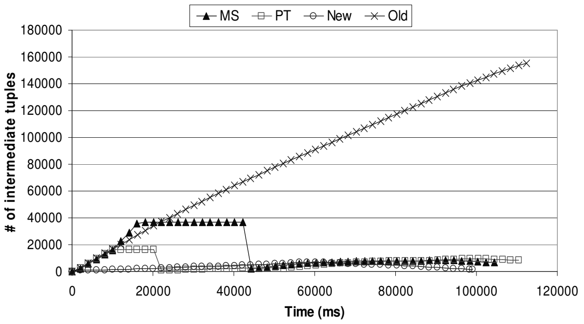


Figure 15: Intermediate Tuple Counts - High Config

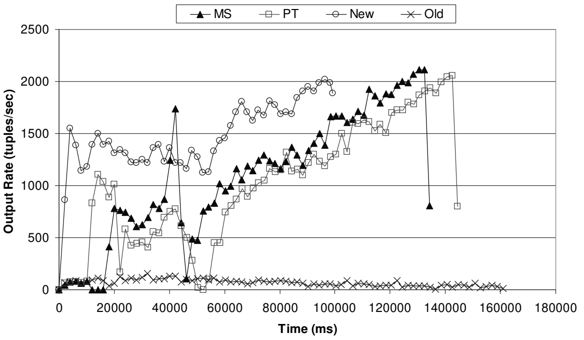


Figure 16: Output Rate - High Config

The parallel strategy has a smaller intermediate tuple count as shown in Figure 15 and a higher output rate at the initial stage of migration in Figure 16. This is because the state sizes are much larger and thus the migration time is much longer than what we have seen in the case of a low configuration. During the state recomputing of the moving state strategy, tuples in all states cannot be disposed until the migration is over. There is a noticeable output silence between 10,000ms and 20,000ms in Figure 16 for the moving state strategy.

On the other hand, the parallel track strategy starts executing both the old and the new plans immediately, so intermediate tuples are being consumed (purged), and some output tuples are being generated while the migration is ongoing. Figures 15 and 16 show that although the total time for the migration stage is still smaller in the case of the moving state strategy, as it ends the overall execution earlier, during the migration stage, the parallel strategy has a better output rate and a smaller intermediate tuple count.

7. RELATED WORK

Although there is a renewed and more pressing need for dynamic query plan optimization and migration for continuous queries in streaming environments, on-the-fly query plan re-optimization has been explored for static databases [1,9,12,14].

[14] utilizes a run-time statistics collector and reconfigures only the *unprocessed* portion of the running query plan to improve performance. This solution is not very practical for stream processing, because all operators in a long running query plan may have been executing by the time the migration is needed. The dynamic optimization for static databases proposed in [1] only applies to *scan* operators and thus is limited in its usage.

[9,12] describe a query plan competing model to dynamically change the running query plan to another plan. The approach requires that before the query starts, several plans have been chosen and will be executed in parallel. After a while the plan that has the best performance thus far will then be running alone with all other plans being discarded. Although this approach shares some ideas with our parallel track migration strategy, it is technically difficult or almost impossible to come up with the candidates for query plans before continuous queries start running. Furthermore, this dynamic plan migration or re-configuration can be applied only once, and is thus too limiting for a long running query.

The research in [3] proposes to utilize the pause-drain-resume strategy for dynamic plan migration. We now put forth that this strategy has not explicitly addressed how to handle the case of query plans containing stateful operators such as window joins with intermediate states. [18] targets the dynamic plan migration in the context of long-running queries in a distributed database system. The proposed migration strategy cannot be undertaken whenever an optimizer has selected a new query plan, but rather it needs to wait until all involved operators entering their own suspendable point. This extra wait is undesirable in a volatile streaming environment since the new plan may be sub-optimal again before the migration can even start.

Several dynamic query re-optimization by changing the structure of the query plan have been proposed in [5, 21]. Most such optimization strategies alter the order of operators inside the query plan to achieve a better performance. However, these works do not address how to migrate from one plan to another plan at run time, once the optimizer has picked a better plan for the system. This however is the exact problem we are addressing in this paper.

[16] introduces adaptive query plan execution by routing tuples among operators inside a query plan. This novel adaption method is different from the generally adopted query plan re-optimization and migration method, in which tuples follow the same assumed optimal processing path until the structure of the plan is re-optimized. Eddy's always-adapting solution makes it suitable for a highly dynamic environment. However, the flexibility of Eddy comes at times at the price of a per-tuple based overhead since extra information needs to be carried or computed to make routing decisions. Furthermore, the eddy approach has the inherent problem of having to recompute all delta intermediate results in the case of multiple joins. This can cost large amounts of processing time given high stream rates and join selectivities. For an changing environment that is not highly dynamic, the re-optimization and migration method may

have better performance given its batch processing nature.

Existing research has also shown how to migrate parts of a query plan to other processors (machines) according to current system statistics [17]. In this case the structure of the query plan itself remains unchanged. This is a different problem from the plan migration problem discussed in this paper. Our plan migration targets the situation that the structure of the query plan has changed, yet the execution of the query plan remains on the same processor.

8. CONCLUSIONS

In this paper, we have described two dynamic query plan migration strategies, namely the moving state strategy and the parallel track strategy. Both support migration of continuous query plans that contain stateful operators, such as joins. Each of our migration strategies has been designed to guarantee that the correct results of the query plan are not being altered, and the timestamp order of the tuples in any queue is preserved. We also present a model to estimate the length of the migration stage for each migration strategy. Our experimental results confirm our analytical model.

Our migration strategies are designed to be generic to work with any dynamic plan optimization algorithm. We have tested the performance of our migration strategies with the common plan optimization technique of minimizing intermediate results by pushing down operators with high reduction factors. The experimental results show that given sufficient system resources, the moving state strategy usually finishes the migration stage quicker and has a better performance in terms of intermediate results. However, if the system has insufficient processing power to keep up with the old query plan, the parallel track strategy, which can continuously output results even during the migration stage, is observed to have less intermediate results, and a better output rate during the migration stage.

9. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments. We are grateful to Luping Ding, Timothy Sutherland, Hong Su, Brad Pielech, Jinhui Jian and Nishant Mehta for their enormous efforts put into the CAPE [7] system.

10. REFERENCES

[1] G. Antoshenkov. Dynamic optimization of index scan restricted by booleans. In *Proceedings of the IEEE Conference on Data Engineering*, pages 430–440, 1996.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of PODS*, pages 1–16, 2002.

[3] D. Carney, U. Cetintemel, M. Cherniack, and et al. Monitoring streams - a new class of data management applications. In *Proceedings of VLDB Conference*, pages 215–226, 2002.

[4] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *Proceedings of VLDB Conference*, pages 203–214, 2002.

[5] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *Proceedings of International Conference on Data Engineering*, pages 345–356, 2002.

[6] M. Dalar, B. Babcock, S. Babu, and R. Motwani. Chain: Operator scheduling for memory minimization in stream systems. In *Proceedings of ACM-SIGMOD*, pages 253–264, 2003.

[7] DatabaSe Research Group(DSRG), Worcester Polytechnic Institute. Cape: Continuous adaptive processing engine, http://davis.wpi.edu/dsrg/CAPE.

[8] L. Ding, N. Mehta, E. A. Rundensteiner, and G. T. Heineman. Joining punctuated streams. In *EDBT Conference*, pages 587–604, March 2004.

[9] G. Graefe and R. Cole. Optimization of dynamic query evaluation plans. In *Proceedings of ACM-SIGMOD Conference*, pages 150–160, 1994.

[10] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *Proceedings of VLDB Conference*, pages 297–308, 2003.

[11] P. J. Hass and J. M. Hellerstein. Ripple joins for online aggregation. In *Proceedings of ACM-SIGMOD Conference*, pages 287–298, 1999.

[12] Y. Ioannidis, R. T. Ng, K. Shim, and T. Sellis. Parametric query optimization. In *Proceedings of 18th VLDB Conference*, pages 103–114, 1992.

[13] Z. G. Ives, A. Y. Halevy, and D. S. Weld. An xml query engine for network-bound data. In *VLDB Journal*, pages 11(4): 380–402, 2002.

[14] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of ACM-SIGMOD Conference*, pages 106–117, 1998.

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

[16] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of ACM-SIGMOD*, pages 49–60, 2002.

[17] K. W. Ng, Z. Wang, R. R. Muntz, and S. Nittel. Dynamic query re-optimization. In *Proceedings of International Conference on Scientific and Statistical Databases*, pages 264–273, July 1999.

[18] K. W. Ng, Z. Wang, R. R. Muntz, and E. C. Shek. On reconfiguring query execution plans in distributed object-relational dbms. In *Proceedings of International Conference on Parallel and Distributed Systems*, pages 59–66, 1998.

[19] R. Notwani, J. Widom, A. Arasu, and et al. Query processing, appromixation, and resource management in a data stream management system. In *Proceedings of CIDR Conference*, pages 1–16, January 2002.

[20] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. In *IEEE Transactions on Kowledge and Data Enginerring*, pages 15(3):555–568, May/June 2003.

[21] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of ACM-SIGMOD*, pages 37–48, 2002.

[22] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.