Dynamic Plan Migration for Snapshot-Equivalent Continuous Queries in Data Stream Systems *

Jürgen Krämer[1], Yin Yang[2], Michael Cammert[1], Bernhard Seeger[1], and Dimitris Papadias[2]

[1] University of Marburg, Germany
[2] Hong Kong University of Science and Technology, China

Abstract. A data stream management system executes a large number of continuous queries in parallel. As stream characteristics and query workload change over time, the plan initially installed for a continuous query may become inefficient. As a consequence, the query optimizer will re-optimize this plan based on the current statistics. The replacement of the running plan with a more efficient but semantically equivalent plan at runtime is called *dynamic plan migration*. In order to have a sound semantic foundation for query optimization, we investigate dynamic plan migration for snapshot-equivalent plans. We develop a general method for dynamic plan migration that treats the old and new plan as snapshot-equivalent black boxes. This enables the query optimizer to apply the conventional transformation rules during re-optimization. As a consequence, our approach supports the dynamic optimization of arbitrary continuous queries expressible in CQL, whereas existing solutions are limited in their scope.

1 Introduction

Dynamic query optimization at runtime is important for a data stream management system (DSMS) because the subscribed queries are long-running and the underlying stream characteristics such as arrival rates and data distributions may vary over time. In addition, the query workload may gradually change. In this case, dynamic query optimization may be useful to enable a DSMS to save system resources by subquery sharing.

There are two major steps in dynamic query optimization. First, the query optimizer needs to identify a plan with optimization potential. For this purpose, a DSMS keeps a plethora of runtime statistics, e. g., on stream rates, and selectivities. In the second step, the query optimizer replaces the currently running, inefficient plan by a semantically equivalent but more efficient new plan. This transition at runtime is called *dynamic plan migration* [1]. Dynamic plan migration is easy as long as query plans only consist of *stateless* operators, such

* This work has been supported by the German Research Foundation (DFG) under grant no. SE 553/4-3.

as selection and projection, and inter-operator queues. In order to perform the migration, it is sufficient to pause the execution of the old plan first, drain out all existing elements in the old plan afterwards, replace the old plan by the new plan, and resume the execution finally. In contrast to *stateless* operators, *stateful* operators like join and aggregation must maintain information derived from previously received elements as state information to produce correct results. Migration strategies for query plans with stateful operators are complex because it is non-trivial to appropriately transfer the state information from an old plan to a new plan.

Besides the essential requirement of correctness, a migration strategy should take the following performance objectives into account. It should (i) not stall query execution for a significant timespan as catching up with processing could cause system overload afterwards, (ii) continuously produce results during migration – the smoother the output rate the better, and (iii) minimize the migration duration and migration overhead in terms of system resources like memory and CPU costs.

To the best of our knowledge, dynamic plan migration has only been addressed in [1] so far. The authors proposed two different migration strategies, *moving states* (MS) and *parallel track* (PT). MS computes the state of the new plan instantly from the state of the old plan at migration start. Afterwards the old plan is discarded and the execution of the new plan is started. In order to apply MS, the query optimizer requires a detailed knowledge about the operator implementations because it needs to access and modify state information. Despite the fact that it may be possible to define those transitions correctly for arbitrary transformation rules involving joins, aggregation, duplicate elimination etc., the implementation will be very complex and inflexible. For that reason, we prefer the second strategy proposed in [1], namely PT, as starting point for our black box migration approach. In contrast to MS, PT runs both plans in parallel for a certain timespan to initialize the new plan gradually with the required state information. Although [1] claims that PT is generally applicable to continuous queries over data streams, we identified problems if stateful operators other than joins occur in a plan, e. g., duplicate elimination and aggregation. In this paper we develop a general migration strategy which overcomes these deficiencies.

The basis of our approach is the temporal semantics for continuous queries over data streams defined in [2]. The algebraic transformation rules known from conventional database systems can be transferred to the stream algebra in [2] because the stream-to-stream operators are *snapshot-reducible* [3, 4] to their counterparts in the extended relational algebra. Snapshot-reducibility is also the reason why the semantics and stream algebra in [2] are in accordance with CQL [5], an extension of SQL for continuous queries. Based on this semantic foundation, our migration strategy requires for correctness that both plans – the old and new – produce snapshot-equivalent results. Note that applying conventional transformation rules guarantees this property for the algebra in [2].

As we generalize PT, we treat the old and new plan as snapshot-equivalent black boxes which can consist of arbitrary operators. The basic idea of our

approach is to define a split time. For all time instants before the split time, results are computed by the old plan, whereas the new plan computes the results for all other time instants. The migration is finished as soon as the timestamps of the elements in all input streams reached the split time.

The contributions of this paper can be summarized as follows:

- We show that PT fails to cope with plans involving stateful operators other than joins.
- We propose our general solution for dynamic plan migration of arbitrary CQL queries, called *GenMig*, prove its correctness, and discuss implementation issues. Besides a straightforward implementation, we suggest two optimizations.
- We analyze the performance of GenMig with regard to the objectives mentioned above, and compare it to PT.

The rest of this paper is organized as follows. Section 2 introduces semantic foundations and briefly summarizes implementation techniques for continuous queries over data streams. In Section 3, we demonstrate that PT fails for plans with other stateful operators than joins. Our general plan migration strategy is presented in Section 4. Section 5 shows the results of our experimental studies. Related work is discussed in Section 6. Finally, Section 7 concludes the paper.

2 Preliminaries

In order to understand the rationale of our approach, it is important to know the underlying semantics and compatible implementations. In analogy to traditional database management systems, we distinguish between a *logical* and a *physical* operator algebra. The logical stream algebra defines the semantics of the operations, whereas the physical algebra provides implementations.

2.1 Semantic Foundations

Time The notion of time is of utmost importance in stream processing. As common in most approaches, we assume each stream element to be equipped with a timestamp, and streams to be ordered by this timestamp attribute. Furthermore, we assume that only a finite number of elements has the same timestamp. Let $\mathbb{T} = (T, \leq)$ be a discrete time domain with a total order $\leq$. We use $\mathbb{T}$ to model the notion of application time, not system time. For the sake of simplicity, let T be the non-negative integers $\{0, 1, 2, 3, \ldots\}$.

Sliding Window Queries Sliding window queries are the most common and important type of continuous queries in DSMS. Without loss of generality, we assume time-based sliding window queries in this paper. A sound semantics for this query type has been established in recent years [2, 5] based on the operations of the well-known extended relational algebra [6, 7].

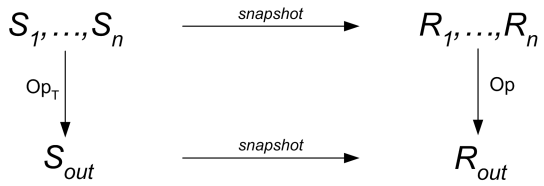

Fig. 1. Snapshot reducibility

The operators in [2] can be classified into two categories: *window* operators and *standard* operators. The window operators model the sliding window semantics. In a logical query plan, a window operator is placed downstream of each source for which a window has been specified in the corresponding query representation, e. g., CQL [5]. The rest of the plan consists of standard operators borrowed from the temporal relational algebra [3]. The standard operators are *snapshot-reducible* to their relational counterparts and are used in the same way.

Definition 1 (Snapshot-Reducibility). *We denote a stream-to-stream operation op_T with inputs $S_1, \ldots, S_n$ as snapshot reducible if for each time instant $t \in T$, the snapshot at t of the results of op_T is equal to the results of applying its relational counterpart op to the snapshots of $S_1, \ldots, S_n$ at time instant t.*

Figure 1 illustrates the temporal concept of snapshot reducibility [3, 4]. A snapshot of a stream at a time instant t can be considered as a relation since it represents all tuples valid at that time instant.

Definition 2 (Snapshot-Equivalence). *Two streams are snapshot-equivalent if for all time instants t, the snapshots at t of both streams are equal.*

We denote two query plans as equivalent if they produce *snapshot-equivalent* results. Note that conventional transformation rules applied to snapshot-reducible operators preserve snapshot-equivalence [3]. Snapshot equivalence is the reason why the common relational transformation rules are still applicable to the stream algebra [2] and make query optimization feasible.

Remark 1. Figure 1 can also be used to motivate that [2] and [5] basically have the same expressiveness. The abstract semantics proposed for STREAM transforms the input streams into relations, uses operations of the relational algebra for processing, and transforms the output relations back into streams. The window specifications are included in the mappings from streams to relations. Hence, the stream-to-stream approach in [2] works completely on the left side of Figure 1, whereas STREAM goes the way round on the right side.

2.2 Interval-based Implementation

There currently exist two major implementation techniques compatible with the semantics introduced above: the interval-based implementation [2, 8], and

the positive-negative tuple approach [5, 9]. We describe the implementation of GenMig for the interval-based approach, but additionally outline how it can be adapted to the positive-negative approach.

The stream-to-stream operator algebra in [2] relies on so-called physical streams.

Definition 3 (Physical Stream). *A physical stream S is a potentially infinite, ordered sequence of elements $(e, [t_S, t_E))$ composed of a tuple e belonging to the schema of S and a half-open time interval $[t_S, t_E)$ where $t_S, t_E \in T$. A physical stream is non-decreasingly ordered by start timestamps.*

The interpretation of a physical stream element $(e, [t_S, t_E))$ is that a tuple e is valid during the time interval $[t_S, t_E)$.

Input Stream Conversion Input streams of many stream applications provide elements with a timestamp attribute, but no time interval. As those streams are usually ordered by timestamps, a physical stream can be generated easily by mapping each incoming element e with its internal timestamp t to $(e, [t, t + 1))$, where $+1$ indicates a time period at finest time granularity.

Window Operator The window operator assigns a validity according to its window size to each element of its input stream. For a *time-based* sliding window, the window size $w \in T$ represents a period in application time. For each element $(e, [t_S, t_S + 1))$, the window operator extends its validity by adding the window size to its end timestamp, i. e., $(e, [t_S, t_S + 1 + w))$. This is intuitive for a time-based sliding window query since a stateful operation downstream of a window operator has to consider an element for additional w time instants. In the general case, where nesting of continuous sliding window queries is permitted, the window operator produces for each incoming element $(e, [t_S, t_E))$ a set of elements by extending the validity of each single time instant by the window size w. Note that a window operator does not affect stateless operations, such as selection and projection.

Query Plans Query plan construction is basically the same as in conventional database systems, except that the query optimizer has to place the window operators in addition. The window operators are placed downstream of the source for which a window has been specified. This placement is performed by the query optimizer when transforming the posed query into the corresponding logical query plan. Thereafter, the query optimizer computes the physical plan by choosing a physical operator for each logical one.

Temporal Expiration Besides the desired semantics, windowing constructs (i) restrict the resource usage and (ii) ensure non-blocking behaviour of stateful operators over infinite streams [10]. For stateful operators like joins, elements in the state expire due to the validity assigned by the window operator. A stateful

operator considers an element $(e, [t_S, t_E))$ in its state as expired if it is guaranteed that it will not be involved in the result production any more. That means, no element in one of its input streams will arrive in the future whose time interval will overlap with $[t_S, t_E)$. According to the total order maintained for each physical stream, this condition holds if the minimum of all start timestamps of the latest incoming element from each input stream is greater than t_E. A stateful operator can delete all expired elements from its state. In those cases where application-time skew between streams and latency in streams becomes an issue, *heartbeats* [11] can be used to explicitly trigger additional expiration steps.

Examples Our examples throughout the paper are based on two stateful operators: join and duplicate elimination. The snapshot-reducible join satisfies the following conditions [2, 8]: (a) for two participating stream elements, the join predicate has to be fulfilled and (b) their time intervals have to intersect. The time interval associated with the join result is set to the intersection of the two participating time intervals. The snapshot-reducible duplicate elimination removes duplicate tuples at each single snapshot. That means, the output must not contain two elements with identical tuples and intersecting time intervals.

2.3 Positive-Negative Implementation

Another common implementation technique for continuous queries is the Positive-Negative (PN) tuple approach, used in STREAM [12] and Nile [9] for instance. The PN implementation is based on streams with elements of the following format: a tuple e, a timestamp $t \in T$, and a sign, $+$ or $-$. A stream is ordered by timestamps. The signs are used to define the validity of elements. The standard operators are modified to handle positive and negative tuples, in particular with regard to temporal expiration. For each incoming stream element with application timestamp t, the window operator sends a positive element with that timestamp, and after $w + 1$ (window size + 1) time units, a negative element is sent with timestamp $t + w + 1$ to signal the expiration. For further details, we refer the reader to [9, 12].

A pair consisting of a positive and its corresponding negative element can be used to express a stream element in the interval-based approach. Namely, $(e, [t_S, t_E))$ can be implemented by sending a positive element $(e, t_S, +)$ and negative element $(e, t_E, -)$. Hence, positive elements refer to start timestamps, whereas negative elements refer to end timestamps. Even at this physical level, the semantic equivalence of both approaches becomes obvious. However, the interval approach does not have the drawback of doubling stream rates due to sending positive and negative tuples.

3 Problems of the Parallel Track Strategy

After outlining PT, we demonstrate that PT produces incorrect results if applied to plans involving stateful operators other than joins. Note that the authors

in [1] claim that their migration strategies are generally applicable. We assume a global time-based window of size w as in [1]. We use the term *box* to refer to the implementation of a plan, i. e., the physical query plan actually executed.

3.1 Parallel Track Strategy

At migration start PT pauses the processing only shortly to plug in the new box. Then, it resumes the old box and runs both boxes in parallel. The results of both plans are merged. Finally, when the states in the old box solely consist of elements that arrived after migration start, the migration is over and the old box can be safely removed. This implies that all elements stored in the state before migration start have been purged due to temporal expiration.

The following aspects guarantee the correctness of the PT strategy according to [1]: (i) Although all elements arriving during migration are processed by both plans, the combined output must not contain duplicates, that means results produced by both plans for the same snapshot. This is achieved by marking the elements with flags, *old* and *new*, which indicate if an element arrived before and after the migration start time, respectively. For combined results, e. g., join results, a *new* flag is assigned if all involved elements had a *new* flag. To guarantee correctness, PT removes all results of the old box that are assigned with a *new* flag as those are additionally produced by the new box. (ii) In order to preserve temporal ordering, the output of the two boxes has to be synchronized. PT simply buffers the output of the new box during migration.

3.2 Problems

The PT strategy as proposed in [1] works well for join reordering but fails if other stateful operators are involved. We illustrate this by a concrete example. Let us consider the migration scenario given in Figure 2 consisting of an equi-join ($\bowtie$) and a duplicate elimination (δ) which is pushed down for optimization purposes. Recall that this is a standard transformation rule which holds in the stream algebra due to our semantic foundations. Figure 2 shows the query plans as well as the inputs and outputs of the operators.

Example 1. We have two input streams A and B delivering the elements listed in the upper right table. The table labelled with $\bowtie$ depicts the correct results of the join inside the old box. The table labelled with δ contains the correct results of the duplicate elimination of the old box. The tables δ'_1, δ'_2, and $\bowtie'$ correspond to the operator results of the new box. Correctness here refers to the snapshot-reducible semantics assumed. Although we use our notation with half-open time intervals to denote the validity of elements, the example solely relies on snapshot-reducible operator semantics and thus is implementation independent. Recall that a time interval just describes a contiguous set of time instants.

All input elements are considered to be valid for 100 time units, which is the global window size. The migration start is at time instant 40. The element $(a, [20, 121))$ from input B, which is marked as *old*, joins with the element

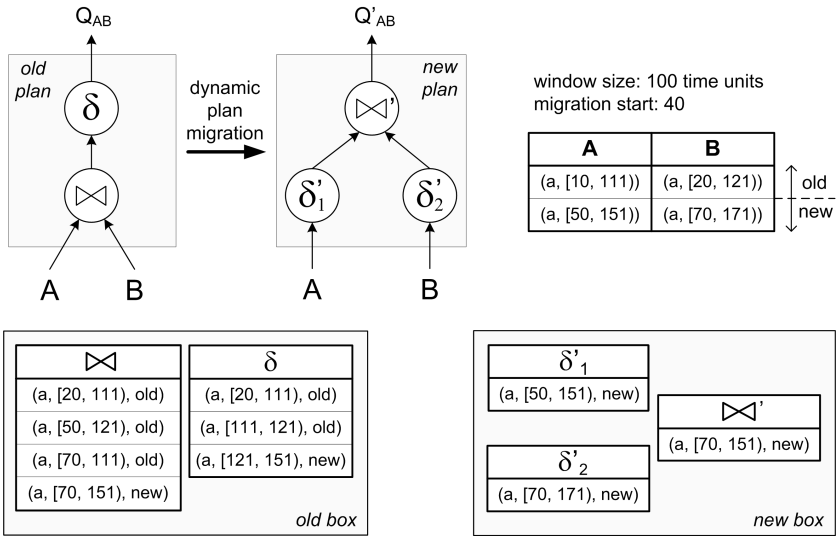

Fig. 2. Plan migration with duplicate elimination

$(a, [50, 151))$ from input A. Hence, the join result $(a, [50, 121))$ is marked as *old*. As $[50, 121)$ overlaps with the time interval of the duplicate elimination's result $(a, [111, 121))$, this is also marked as *old*. Therefore, it definitely belongs to the output of the old box. Unfortunately, $(a, [111, 121))$ has a temporal overlap with $(a, [70, 151))$ which is a result of the new box. Consequently, the complete output contains tuple a for the snapshots 111 to 120 twice. Thus, PT does not produce correct results.

The reason for this problem is that the validity of results of the old box can refer to points in time which are beyond the plan migration start time. These time instants may additionally be addressed by the new box. Since the new box has no information about the old box, duplicates at those time instants may occur in the output, which is the union of both boxes. GenMig overcomes this problem by introducing a split time which is greater than all time instants occurring in the old box.

Note 1. The problem of PT is not restricted to duplicate elimination but arises for other operators as well, e. g., aggregation and difference. Although join reordering is a very important transformation rule which is covered by PT, there exist rules for other stateful operators [3, 13] for which PT will fail in a similar way as shown above.

4 A General Strategy for Plan Migration

In this section we propose *GenMig* which overcomes the problems of PT while maintaining its merits, namely (i) a gradual migration from the old to the new plan, and (ii) generating results during migration.

4.1 Logical View

We first present the basic idea of GenMig from a purely logical and semantic perspective.

GenMig Strategy Given two snapshot-equivalent plans, the query optimizer determines a point in application time denoted as T_{split}. At this time instant the time domain is split into two partitions.

- For all time instants $t < T_{split}$ the results are produced by the old plan.
- For all time instants $t \geq T_{split}$ the results are produced by the new plan.

The union of both plans represents the total results. T_{split} refers to the plan migration end because up from this point in time the new plan produces the output by itself, and thus the old plan can be discarded. The migration duration is the period from migration start to T_{split}.

Correctness Under the assumption that the old and new plan produce a snapshot-equivalent output, the total output of GenMig is complete. The output is computed for every single time instant. Due to snapshot-equivalence it does not matter if the results for a snapshot are produced by either the old or the new plan. The duplicate elimination problem does not arise because the results of the two plans are disjoint in terms of timestamps.

4.2 Physical View

GenMig is very clear and easy at the logical level. However, at the physical level it is impractical to compute the query results for every single snapshot separately.

Interval-based Implementation of GenMig Algorithm 1 shows the implementation of GenMig for the interval-based approach [2, 8]. The input consists of the old currently running box, the corresponding input streams, and the new box without any state information. In contrast to the PT strategy, GenMig does not start with plan migration instantly. It starts monitoring the start timestamps instead.

Remark 2. In contrast to [1] where a single migration start time is used, our approach maintains a migration start time for each input. This has the advantage that GenMig does not require the scheduling to obey the global temporal ordering by start timestamps, which would be in conflict with sophisticated scheduling strategies [14–16] otherwise.

Algorithm 1: GenMig

Input : streams $I_1, \ldots, I_n$, old plan with state information, new plan without state information, global window constraint w
Output : streams $O_1, \ldots, O_m$, new plan with state information

```
1  foreach input stream I_i, i ∈ {1, ..., n} do
2      Start monitoring the start timestamps;
3      Keep the most recent start timestamps of I_i as t_{S_i};
4  Pause the execution of the old plan as soon as t_{S_i} has been set for each input;
5  T_split ← max{t_{S_i} | i ∈ {1, ..., n}} + w + 1 + ε;
6  Insert a Split operator downstream of each source of the old plan;
7  Insert a Coalesce operator at the top of both plans for each output stream;
8  Resume the execution of the old plan and start the execution of the new plan;
9  while min{t_{S_i} | i ∈ {1, ..., n}} < T_split do
10     Continue the execution of both plans;
11 Signal the end of all input streams to the old plan;
12 Stop the execution of the old plan;
13 Pause the execution of the new plan;
14 Remove the old plan, coalesce and split operators;
15 Connect inputs and outputs directly with the new plan;
16 Resume the execution of the new plan;
```

After initialization, i. e., when for each input stream a timestamp t_{S_i} has been determined, the old box is paused. The split time T_{split} is set to the maximum of all t_{S_i} plus the window size w plus 1 plus ε. This setting ensures that T_{split} is greater than any time instant in the old box.

Remark 3. Without loss of generality, we assume ε to be chosen so that T_{split} neither occurs as start nor end timestamp in any input stream. This can for instance be achieved if T_{split} is expressed at a finer time granularity [17] and ε refers to a chronon according to that granularity. From the implementation side, this assumption can be assured easily but is omitted in the algorithms due to clarity reasons.

In addition to setting the split time, two novel operators – split and coalesce – are inserted at the inputs and outputs, respectively. Figure 3 illustrates this placement. For their implementations, see Algorithms 2 and 3. The stateless split operator splits the time interval of an incoming element at T_{split} into two disjoint intervals. The tuple e associated with the first interval is sent to the old box, e associated with the second interval is sent to the new box.

The coalesce operator inverts the effects of splitting. Coalesce merges two equivalent tuples, each of one input, with adjacent time intervals. Contrary to split, coalesce is a stateful operator because it has to maintain data structures, e. g., hash maps, for the detection of equivalent tuples. Furthermore, a heap ordered by start timestamps is required to ensure the ordering property of the output stream.

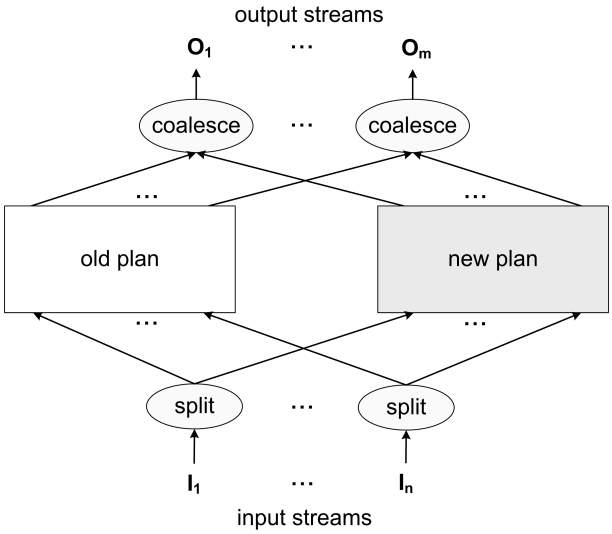

Fig. 3. GenMig strategy

The migration end is defined as the point in application time when the monitored start timestamps of all input streams are equal or greater than T_{split}. Then, the optimizer signals the end of all input streams to the old box in order to drain out intermediate elements. After that, the heap inside the coalesce operator containing the new results can be flushed. Finally, the optimizer shortly interrupts the processing to discard the old box as well as the split and coalesce operators before it continues to execute the new plan stand-alone.

Algorithm 2: Split

Input : stream I, split time T_{split}
Output : streams O_{old}, O_{new}

```
1 foreach new incoming stream element (e, [t_S, t_E)) ∈ I do
2     if t_S < T_split then
3         if t_E ≤ T_split then Append (e, [t_S, t_E)) to O_old;
4         else
5             Append (e, [t_S, T_split)) to O_old;
6             Append (e, [T_split, t_E)) to O_new;
7     else
8         Append (e, [t_S, t_E)) to O_new;
```

Algorithm 3: Coalesce

Input : streams I_0 (from old plan), I_1 (from new plan), hash maps M_0, M_1, heap H, split time T_{split}
Output : stream O

foreach *new incoming stream element* $(e, [t_S, t_E)) \in I_i, i \in \{0, 1\}$ **do**
  $t_{out} \leftarrow 0$;
  if $t_E < T_{split} \vee t_S > T_{split}$ **then**
    Append $(e, [t_S, t_E))$ to H;
  else
    if $i = 0$ **then**
      if $\exists (e', [t'_S, t'_E)) \in M_1$ *where* $e = e'$ **then**
        Append $(e, [t_S, t'_E))$ to H and remove it from M_1;
      else Insert $(e, [t_S, t_E))$ into M_0;
    if $i = 1$ **then**
      if $\exists (e', [t'_S, t'_E)) \in M_0$ *where* $e = e'$ **then**
        Append $(e, [t'_S, t_E))$ to H and remove it from M_0;
        $t_{out} \leftarrow t'_S$;
      else Insert $(e, [t_S, t_E))$ into M_1;
  while $t_{out} \geq H.top.t_S$ **do**
    Append $H.top$ to O and remove it from H;
if *migration finished* **then**
  Flush H and append its elements to O;

4.3 Correctness

Lemma 1. *GenMig produces correct results and preserves the temporal ordering.*

Proof. (sketch)

1. No elements are lost during the insertion and removal of the split and coalesce operators because the processing of the boxes is suspended during these steps.
2. The split operator guarantees that all elements valid at snapshots smaller than T_{split} are transferred to the old box, while the elements with snapshots equal or greater than T_{split} are processed by the new box. This matches with the logical view of GenMig (see Section 4.1). Since no snapshot is lost under this partitioning and each box produces snapshot-equivalent results, the union of both plans contains the entire result.
3. PT uses a marking mechanism to detect duplicates, i. e., results at the same snapshot produced by both plans. We showed in Section 3 that this marking fails for stateful operators other than joins. GenMig overcomes these problems because the split operator ensures that the results of both boxes are disjoint in terms of snapshots due to the choice of T_{split}. As T_{split} is greater

than any snapshot referenced in the old box, the corresponding snapshot-reducible operators in that box will never produce a result with a snapshot equal or larger than T_{split}. Moreover, the new box will never generate results for snapshots smaller than T_{split}. Consequently, GenMig inherently avoids the generation of duplicates as addressed for PT.

4. The temporal ordering is preserved as (i) all operators inside a box produce a correctly ordered output, (ii) the split operators are stateless and do not affect the ordering, and (iii) the coalesce operator explicitly synchronizes the results of the old and the new box according to the start timestamp ordering.
5. The coalesce operator combines the results of both plans. It does not have any semantic effects as coalesce preserves snapshot-equivalence [3]. Because it merges stream elements with identical tuples and adjacent time intervals, it rather serves as an optimization which inverts the negative effects of the split operator on stream rates.
6. Due to the global window constraint of w time units, the maximum interval length in a plan is limited to w time units (see window operator in Section 2). Snapshot-reducibility implies that the time intervals in the output stream of a standard operator can only have shorter time intervals. Hence, setting T_{split} to $max\{t_{S_i} | i \in \{1, \ldots, n\}\} + w + 1 + \varepsilon$ ensures that T_{split} is greater than any time instant occurring in the old box.

□

4.4 Performance Analysis

Given the sufficient-system-resources assumption as in [1], the difference between system and application becomes negligible. We can thus identify durations in application time with those in system time. The migration duration of GenMig is determined by $T_{split} - min\{t_{S_i}\}$ where t_{S_i} denotes the migration start time of input i. For negligible application time skew between streams and negligible latency in streams, the migration duration is approximately w time units due to the choice of T_{split}.

Compared to PT, GenMig has the following advantages:

- For join trees with more than one join, the required time for migration is only w time units instead of $2w$. GenMig requires at most w time units because all elements in the old plan have become outdated at T_{split}. GenMig does not need to wait until all *old* elements were purged from states in the old box as done for PT. Consequently, the allocated memory for the old box can be released earlier.
- GenMig does not require any mechanisms to detect duplicates at the output of the old box. Hence, those costs for duplicate detection can be saved.
- GenMig does not need to buffer the entire results of the new box during migration for ordering purposes. All results produced by the new box during migration can be coalesced and emitted. The size of the heap and hash maps inside the coalesce operator is predominantly determined by the application time skew between the input streams. Heartbeats [11] and sophisticated

scheduling strategies can be used to minimize application time skew and thus the memory allocation of the coalesce operator.
- According to [1], the migration for PT is finished if all *old* elements have been removed from the old box. For join trees with more than one join, this happens after $2w$ time units. Interestingly, the old box only produces output during the first w time units. The other w time units are used to purge all *old* elements from the states. For snapshot-reducible query plans, there will be no output during this timespan. Therefore, PT has the following output rate characteristics. For the first w time units, the output rate corresponds to the output rate of the old box. The next w time units there is no output. At the migration end there is a burst when the buffer on top of the new box is flushed. In contrast, GenMig directly switches from the output rate of the old box to the one of the new box at migration end (T_{split}).

4.5 Optimizations

Optimization 1 - Reference Point Method The reference point method [18, 19] is a common technique for index structures to prevent duplicates in the output. We can use this method as an optimization of GenMig if the following modifications are performed. The split operator has to be modified to send the elements to the old plan without splitting, i. e., with the full intervals. The coalesce operator is removed and replaced by a simple selection and a union. The selection is placed on top of the new box and drops all elements with a start timestamp equal to T_{split}. The union of the old box and the selection generates the final output. We treat the start timestamp of results of the new box as reference point. This reference point is compared with T_{split}. If it is larger than T_{split}, the element is not a duplicate and sent to the output.

Using the reference point method makes coalescing superfluous. Hence, it saves the memory and processing costs spent on the coalesce operator. Both boxes produce their output correctly ordered. As we use the start timestamp as reference point, all results from the old box have a smaller start timestamp than those from the new box. Therefore, it is sufficient to first output the results of the old box and afterwards those of the new box. Under the assumptions of a global temporal scheduling as in [1], no buffer is needed to synchronize the output of both boxes. Note that all additional operators required for GenMig with reference point optimization (split, union, and selection) have constant costs per element.

Optimization 2 - Shortening Migration Duration The migration duration could be shortened if T_{split} is set to the maximum end timestamp inside the old box plus 1 plus ε. This setting still satisfies the correctness condition that T_{split} has to be greater than any time instant occurring in the old box. However, such an optimization requires to monitor the end timestamps in addition. If a DSMS provides this information as some kind of metadata, it could be effectively used to reduce the migration duration and thus gain savings of system resources. This optimization is particularly effective if the plan to be optimized is not close to

window operators. In this case, it is likely that the time intervals are significantly shorter than the window size.

4.6 Positive-Negative Implementation for GenMig

The GenMig algorithm can easily be transferred to the positive-negative tuple approach [9, 5]. Instead of monitoring the start timestamps, the timestamps of the positive elements are monitored. T_{split} is set as proposed in Algorithm 1. The split operator sends all incoming positive and associated negative elements to the new box and additionally to the old box if their timestamps are smaller than T_{split}. Using the timestamp of an element, independent of its sign, as reference point, we accept results from the old box if their timestamps are less than T_{split}, and from the new box if it is greater than T_{split}. Since the results generated by both plans are correctly ordered, it is sufficient to first output the results of the old box and afterwards those from the new box. The migration end is reached if all input streams passed T_{split}.

5 Experimental Evaluation

Although the primary focus of this work is on the semantics, generality, and correctness of GenMig, we conducted a set of experiments that compares GenMig with PT for join reordering. We observed that even in this case, where PT is only applicable, GenMig is at least as efficient as PT. In addition, we validated GenMig for a variety of transformation rules beyond join reordering. However, those experiments only show the correctness of GenMig and point out the potential of dynamic query optimization. As these do not contribute further insights into our approach, we omitted them due to space limitations.

We implemented PT and GenMig in our PIPES framework [20, 21] with Java 5. Our hardware was a PC with an Intel Pentium 4 processor (2.8 GHz) and 1 GB of RAM running Windows XP Professional. To have a fair comparison with the original PT implementation in [1], we executed the plans in a single thread according to the global temporal ordering. Since a comparison with PT is only possible for joins, we executed 4-way nested-loops joins as done in [1].

In our first experiment, we ensured the sufficient-system-resources assumption which means that query execution can keep up with stream rates. Each input stream delivered 5000 random numbers with a rate of 100 elements per second. The random numbers were distributed uniformly between 0 and 500 for streams A and B, and between 0 and 1000 for streams C and D. We performed time-based sliding window equi-joins with a global window size of 10 seconds. The old plan was set to the left-deep join tree $((A \bowtie B) \bowtie C) \bowtie D$ which was rather inefficient due to the huge intermediate result produced by $A \bowtie B$. The goal of the dynamic plan migration was to switch to the more efficient right-deep join tree $A \bowtie (B \bowtie (C \bowtie D))$. The migration started after 20 seconds.

GenMig finished the migration w time units (10 seconds) after migration start as expected. In contrast, PT requires $2w$ time units due to purging all

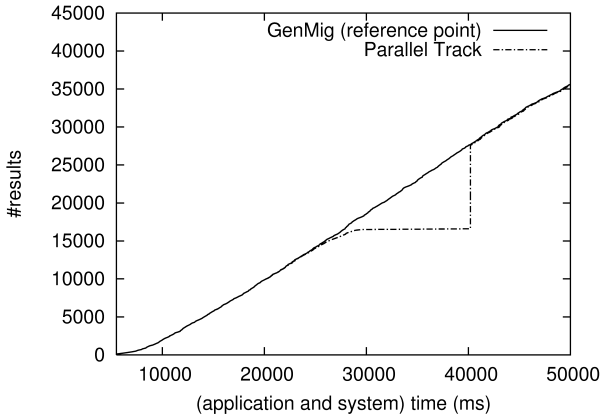

Fig. 4. Characteristics of Parallel Track and GenMig

old elements from the old box. This complies with the analysis in [1]. During migration the output rate of PT decreases because the results of the new box are buffered as shown in Figure 4. For PT, after 30 seconds the output rate is 0 for 10 seconds. During this period the purging of *old* elements took place. At the migration end for PT after 40 seconds (migration start + $2 \cdot$ window size $= 20 + 2 \cdot 10 = 40$), the results of the new box which had been buffered during migration were immediately released. This caused the significant burst in the output rate. Such a burst may lead to a temporary system overload and should be avoided whenever possible. In contrast to PT, GenMig produces results with a smooth output rate during migration.

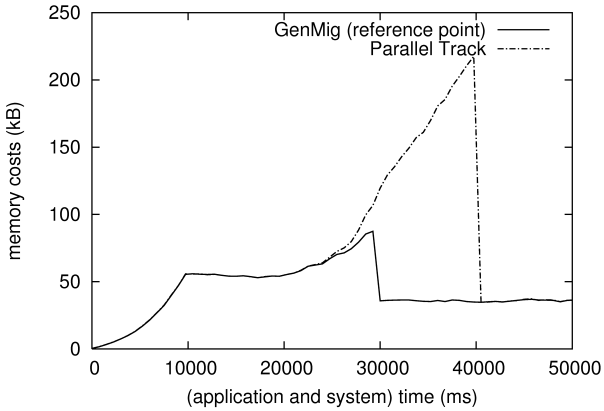

Fig. 5. Memory usage of Parallel Track and GenMig

Figure 5 shows the memory usage of GenMig and PT during the experiment. For sake of comparability, we only measured the memory allocated for the values. We omitted the overhead of timestamps – GenMig requires two timestamps per element (time interval), whereas PT needs up to four. Note that the memory usage can only differ during migration. Figure 5 demonstrates that PT continuously requires more memory than GenMig during this period. Overall, the system has an increased memory usage during plan migration but profits from the reduced memory usage of the new plan afterwards. This temporary increase of memory usage is smaller for GenMig.

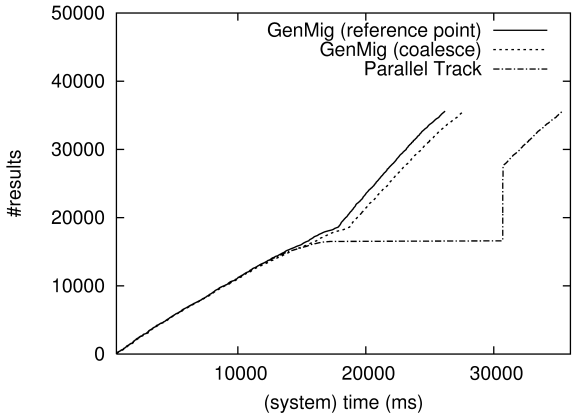

Fig. 6. Performance comparison of Parallel Track, GenMig with coalesce, and GenMig with reference point optimization

Our second experiment was aimed at comparing the total system load of PT, GenMig with coalesce, and GenMig with reference point optimization. We processed the same random numbers associated with the same application timestamps as before. But this time, we processed the input streams as fast as possible. This means, we no longer synchronized application and system time, and the system was saturated. This is a widely accepted approach to determine the efficiency of stream join algorithms [22]. Furthermore, we simulated a more expensive join predicate to emphasize the effects of complex join computations. Figure 6 depicts our performance measurements. At the beginning the slope of the curves, which corresponds to the output rate, is less steep than at the end because the left-deep join plan is not as efficient as the right-deep plan. During migration the slope reaches its minimum as both plans run in parallel. The total runtimes demonstrate that GenMig is superior to PT. Moreover, the reference point optimization improves the coalesce variant of GenMig slightly as it avoids the CPU costs caused by the coalesce operator.

To sum up the experiments, GenMig is not only more general than PT but also more efficient for the case of join reordering where both strategies are applicable.

6 Related Work

In recent years adaptivity issues for query processing and optimization have attracted research attention. The following discussion is limited to work related to stream processing. The interested reader is referred to [23] for a survey beyond streams.

Our work directly refers to the dynamic plan migration strategies proposed in [1] as it generalizes PT towards arbitrary continuous query plans. To the best of our knowledge, the strategies published in [1] are the only methods for dynamic plan migration in DSMS.

There are several papers on different topics of runtime optimizations for DSMS but these do not tackle plan migration issues explicitly. In [24] the problem of executing continuous multiway join queries is addressed for streaming environments with changing data characteristics. GenMig does not aim at optimizing multiway join performance by materializing intermediate join views. In contrast, it is designed more general and treats join reordering as one possible transformation rule. Proactive re-optimization [25] is a novel optimization technique in stream processing in which the optimizer selects the initial query plan with the background information that this plan is likely to be re-optimized in future due to uncertainty in estimates of the underlying statistics. However, the choice of a suitable plan is not the focus of GenMig. It rather describes how to migrate from one plan to another snapshot-equivalent plan.

Eddies [26] perform a very flexible kind of adaptive runtime optimization for continuous queries. Unlike traditional approaches where elements are processed according to a given query plan until the plan is re-optimized, all operators of a query plan are connected with the eddy. For each incoming element, the eddy determines an individual query plan. PT and GenMig are by far not as flexible as eddies, but the per-element routing is expensive and only profitable in highly dynamic environments. Moreover, eddies are limited to queries with selection, projection, and join, whereas GenMig considers more general plans.

Cross-network optimization issues for continuous queries are discussed in [27] for the Borealis stream processing engine, but not at a semantic level with concrete techniques for plan migration as presented in this paper. How GenMig can be adapted to a distributed environment remains as an open issue for future work.

7 Conclusions

In this paper we first identified shortcomings of the parallel track strategy, an existing solution for the dynamic plan migration problem in DSMS. We showed that this strategy fails to cope with plans involving stateful operators other than

joins. We consequently proposed a general approach to dynamic plan migration, called *GenMig*, which enables a DSMS to optimize arbitrary CQL queries at runtime. Our analysis and performance comparison shows that GenMig with its optimizations is at least as efficient as parallel track, while being more general. Due to the underlying semantics, the whole set of conventional transformation rules can be applied for optimization purposes. Moreover, GenMig does not require any specific knowledge about the operator states and implementations because it treats the old and new plans as black boxes which only have to produce snapshot-equivalent results to ensure correctness. Due to its generality and ease of use, GenMig is likely to be integrated into existing and future DSMS as a basic mechanism for the dynamic query optimization of continuous queries.

References

1. Zhu, Y., Rundensteiner, E.A., Heineman, G.T.: Dynamic Plan Migration for Continuous Queries Over Data Streams. In: Proc. of the ACM SIGMOD. (2004) 431–442
2. Krämer, J., Seeger, B.: A Temporal Foundation for Continuous Queries over Data Streams. In: Proc. of the Int. Conf. on Management of Data (COMAD). (2005) 70–82
3. Slivinskas, G., Jensen, C.S., Snodgrass, R.T.: Query Plans for Conventional and Temporal Queries Involving Duplicates and Ordering. In: Proc. of the IEEE Conference on Data Engineering (ICDE). (2000) 547–558
4. Böhlen, M.H., Busatto, R., Jensen, C.S.: Point-Versus Interval-Based Temporal Data Models. In: Proc. of the IEEE Conference on Data Engineering (ICDE). (1998) 192–200
5. Arasu, A., Babu, S., Widom, J.: An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations. In: Proc. of the Int. Conf. on Data Base Programming Languages (DBPL). (2003) 1–19
6. Dayal, U., Goodman, N., Katz, R.H.: An Extended Relational Algebra with Control Over Duplicate Elimination. In: Proc. of the ACM SIGMOD. (1982) 117–123
7. Albert, J.: Algebraic Properties of Bag Data Types. In: Proc. of the Int. Conf. on Very Large Databases (VLDB). (1991) 211–219
8. Krämer, J., Seeger, B.: A Temporal Foundation for Continuous Queries over Data Streams. Technical report, University of Marburg (2004) No. 45.
9. Hammad, M., Aref, W., Franklin, M., Mokbel, M., Elmagarmid, A.: Efficient Execution of Sliding Window Queries over Data Streams. Technical report, Purdue University (2003) No. 35.
10. Golab, L., Özsu, M.T.: Issues in Data Stream Management. SIGMOD Record **32**(2) (2003) 5–14
11. Srivastava, U., Widom, J.: Flexible Time Management in Data Stream Systems. In: Symp. on Principles of Database Systems (PODS). (2004) 263–274
12. Arasu, A., Babu, S., Widom, J.: The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical report, Stanford University (2003) No. 57.
13. Galindo-Legaria, C., Joshi, M.: Orthogonal Optimization of Subqueries and Aggregation. In: Proc. of the ACM SIGMOD. (2001) 571–581

14. Babcock, B., Babu, S., Datar, M., Motwani, R.: Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In: Proc. of the ACM SIGMOD. (2003) 253–264
15. Carney, D., Cetintemel, U., Zdonik, S., Rasin, A., Cerniak, M., Stonebraker, M.: Operator Scheduling in a Data Stream Manager. In: Proc. of the Int. Conf. on Very Large Databases (VLDB). (2003) 838–849
16. Jiang, Q., Chakravarthy, S.: Scheduling Strategies for Processing Continuous Queries over Streams. Lecture Notes in Computer Science **3112** (2004) 16–30
17. Cammert, M., Krämer, J., Seeger, B., Vaupel, S.: An Approach to Adaptive Memory Management in Data Stream Systems. In: Proc. of the IEEE Conference on Data Engineering (ICDE). (2006) 137–139
18. Seeger, B.: Performance Comparison of Segment Access Methods Implemented on Top of the Buddy-Tree. In: Advances in Spatial Databases. Volume 525 of Lecture Notes in Computer Science., Springer (1991) 277–296
19. van den Bercken, J., Seeger, B.: Query Processing Techniques for Multiversion Access Methods. In: Proc. of the Int. Conf. on Very Large Databases (VLDB). (1996) 168–179
20. Krämer, J., Seeger, B.: PIPES - A Public Infrastructure for Processing and Exploring Streams. In: Proc. of the ACM SIGMOD. (2004) 925–926
21. Cammert, M., Heinz, C., Krämer, J., Riemenschneider, T., Schwarzkopf, M., Seeger, B., Zeiss, A.: Stream Processing in Production-to-Business Software. In: Proc. of the IEEE Conference on Data Engineering (ICDE). (2006) 168–169
22. Golab, L., Öszu, M.T.: Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In: Proc. of the Int. Conf. on Very Large Databases (VLDB). (2003) 500–511
23. Babu, S., Bizarro, P.: Adaptive Query Processing in the Looking Glass. In: Proc. of the Conf. on Innovative Data Systems Research (CIDR). (2005) 238–249
24. Babu, S., Munagala, K., Widom, J., Motwani, R.: Adaptive Caching for Continuous Queries. In: Proc. of the IEEE Conference on Data Engineering (ICDE). (2005) 118–129
25. Babu, S., Bizarro, P., DeWitt, D.: Proactive Re-Optimization. In: Proc. of the ACM SIGMOD. (2005) 107–118
26. Deshpande, A., Hellerstein, J.M.: Lifting the Burden of History from Adaptive Query Processing. In: Proc. of the Int. Conf. on Very Large Databases (VLDB). (2004) 948–959
27. Abadi, D.J., Ahmad, Y., et al. M.B.: The Design of the Borealis Stream Processing Engine. In: Proc. of the Conf. on Innovative Data Systems Research (CIDR). (2005) 277–289