HybMig: A Hybrid Approach to Dynamic Plan Migration for Continuous Queries

Yin Yang
Department of Computer Science
Hong Kong University of Science and Technology
Clearwater Bay, Hong Kong
yini@cs.ust.hk

Jürgen Krämer
Department of Mathematics and Computer Science
University of Marburg
Marburg, Hessen, Germany
kraemerj@mathematik.uni-marburg.de

Dimitris Papadias
Department of Computer Science
Hong Kong University of Science and Technology
Clearwater Bay, Hong Kong
dimitris@cs.ust.hk

Bernhard Seeger
Department of Mathematics and Computer Science
University of Marburg
Marburg, Hessen, Germany
seeger@mathematik.uni-marburg.de

Abstract

In data stream environments, the initial plan of a long-running query may gradually become inefficient due to changes of the data characteristics. In this case, the query optimizer will generate a more efficient plan based on the current statistics. The on-line transition from the old to the new plan is called *dynamic plan migration*. In addition to correctness, an effective technique for dynamic plan migration should achieve the following objectives: (i) minimize the memory and CPU overhead of the migration, (ii) reduce the duration of the transition, and (iii) maintain a steady output rate. The only known solutions for this problem are the *moving states* (MS) and *parallel track* (PT) strategies, which have some serious shortcomings related to the above objectives. Motivated by these shortcomings, we first propose HybMig, which combines the merits of MS and PT, and outperforms both on every aspect. As a second step we extend PT, MS and HybMig to the general problem of migration, where both the new and the old plans are treated as black boxes.

To appear in IEEE TKDE.

Index Items: H.2.4.h Query processing

Contact Author:
Dimitris Papadias
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
Office: ++852-23586971 http://www.cs.ust.hk/~dimitris/
Fax: ++852-23581477 E-mail: dimitris@cs.ust.hk

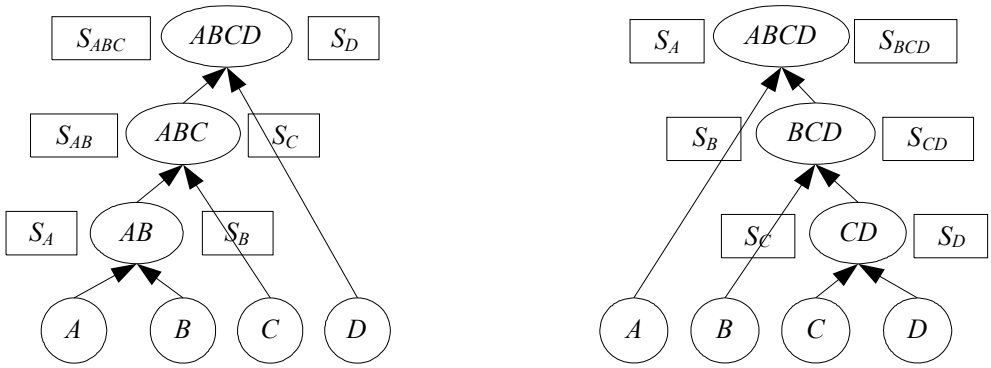
1. Introduction

Long-running, continuous queries are typical in data stream management systems. During the execution of such a query, the major factors affecting the efficiency, such as the selectivity and the stream rate, may change in an unexpected manner [AH00]. Consequently, the running query plan, which is computed based on past statistics, may become inefficient. When this situation occurs, the query processor needs to adopt a semantically equivalent but more efficient new plan, optimized using the current statistics. The change of query plans must be performed online, without affecting the correctness of the output. This transition is referred to as *dynamic plan migration* [ZRH04].

The problem of dynamic plan migration is challenging when the running query plan involves stateful operators. An operator is *stateful*, if it contains internal states that are computed based on previously received tuples. Otherwise, it is *stateless*. For example, a selection that filters the input stream according to a static condition is usually implemented as a stateless operator. In contrast, more complicated operators like joins and aggregations must maintain internal states in order to generate the correct answer. Since these internal states contain information that is vital to the correctness of the output, simply discarding a running query plan with stateful operators causes information loss.

To illustrate this problem, we use an example of join reordering with symmetric, sliding-window, binary joins [HH99, GO03]. Figure 1.1 shows two semantically equivalent plans joining four input streams A, B, C and D. The old plan, i.e., the one that is currently running, is a left-deep plan, while the new plan (after migration) is a right-deep plan. The internal states are shown as rectangular boxes on both sides of each join operator (denoted by an oval). In the old plan $P_{(((AB)C)D)}$, the topmost operator $ABCD$ has two internal states S_{ABC} and S_D, storing tuples

from *ABC* and *D*, respectively[1]. These tuples have been processed, but cannot be discarded since they may generate future results. For instance, although a tuple in S_{ABC} has already been joined with every entry of S_D, it may still be matched with a subsequent arrival from input *D*.



old plan: $P_{(((AB)C)D)}$ new plan: $P_{(A(B(CD)))}$

Figure 1.1 Two semantically equivalent plans

The only known solutions for dynamic plan migration are the *moving states* (MS) and *parallel track* (PT) strategies [ZRH04]. MS temporarily suspends the output stream and generates the operator states in the new plan. Continuing the example, MS first moves S_A, S_B, S_C and S_D from the old plan to the new one. Then, it computes S_{CD} and S_{BCD} by joining tuples in S_B, S_C and S_D. Finally, it discards the old plan and produces results using only the new one. These state computations can be very expensive. Furthermore, since the query results during the migration are delayed, MS may cause a violation of the responsiveness requirements of the query [ACG+04].

PT runs both plans in parallel and combines their results. During the transition, PT outputs results produced by both plans after eliminating duplicates. As discussed in Section 2, the output rate of PT gradually decreases to zero during migration. Furthermore, the application of PT

[1] We use juxtaposition to denote the join operation and parenthesis to specify the join order, e.g., $P_{(((AB)C)D)}$ denotes the left-deep join plan. Join predicates and inter-operator queues are omitted for brevity. We assume that a predicate is checked at the lowest join operator. Section 2.1 provides further explanation on the contents of the internal states and the join algorithm.

imposes some rather restrictive semantics for temporal ordering. Finally, PT and MS were proposed for join re-ordering and their extension to plans with arbitrary operators is not clear.

Motivated by these problems, we first propose HybMig, a hybrid approach that combines and extends the advantages of MS and PT without sharing their shortcomings. HybMig generates steady output rate during the transition, and has lower CPU cost and shorter migration duration than both MS and PT. Furthermore, it is compatible with *both* the temporal ordering definition in [ZRH04] and a more popular *max*-ordering. Then, we extend HybMig (as well as, PT and MS) to the general problem of dynamic plan migration, treating both plans as black boxes. The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 describes the algorithmic framework of HybMig. Section 4 adapts all methods to general plans with arbitrary operators. Section 5 contains a comprehensive set of experiments and Section 6 concludes the paper.

2. Related work

Section 2.1 discusses some basic concepts in the literature of data streams, and Section 2.2 presents MS and PT. Section 2.3 overviews alternative approaches to adaptive query optimization.

2.1 Preliminaries

Sliding windows are the most common technique to deal with infinite inputs. According to this methodology, each incoming tuple t_i is associated with a timestamp $t_i.ts$. Two tuples can be joined if the difference of their timestamps does not exceed a pre-defined window, which may be different for each pair of streams. Hereafter, for simplicity we assume the existence of a *global window* size w. Formally, two input tuples t_i and t_j with timestamps $t_i.ts$ and $t_j.ts$ can join only if $|t_i.ts - t_j.ts| \leq w$. In our discussion, we follow the *global ordering assumption*, that is, for any two

input tuples t_i and t_j, t_i arrives before t_j if and only if $t_i.ts \leq t_j.ts$.

Join processing involves three steps: *purge-probe-insert*. Consider for instance, the operator *AB* in the left plan of Figure 1.1. An incoming tuple t_i from input stream *A* first *purges* tuples of S_B, whose timestamp is earlier than $t_i.ts$-w (such tuples cannot be joined with any subsequent arrival from *A*, due to the global ordering assumption). Then, it *probes* S_B and joins with its tuples. Finally, t_i is *inserted* into S_A. Once a join result is generated, it must also be assigned a timestamp, since it may constitute an input for a subsequent operator. Let t be a result produced by joining input tuples $t_1, \ldots, t_m$ (also called *components* of t). There are several alternatives for assigning $t.ts$, given the timestamps $t_1.ts, \ldots, t_m.ts$. A popular one is $t.ts = \max_{i=1}^{m} t_i.ts$ [ABW]. For example, in Figure 1.1, let t be an output tuple of the operator *AB*, produced by joining t_1 (from *A*) and t_2 (from *B*). Then, $t.ts$ is the later timestamp between $t_1.ts$ and $t_2.ts$. Assuming that $t_1.ts$ and $t_2.ts$ represent the arrival time of the two tuples, then $t.ts$ can be interpreted as the earliest time that t can be created.

Furthermore, it is essential for the correctness of the *purge-probe-insert* framework that output tuples be produced in the order of their timestamps. Consider, on the contrary, that another tuple t' with $t'.ts > t.ts$ in the result of *AB* is generated before t. The purging step triggered by t' removes from S_C all tuples whose timestamp is before $t'.ts$-w. Thus, when t is produced later, its matching S_C tuples in the interval [$t.ts$-w, $t'.ts$-w) have already been expunged, leading to incomplete results. In order to avoid this problem, every operator must adhere to the following *max ordering requirement*: for any two output tuples t and t' with component timestamps $\{t_1.ts, \ldots, t_m.ts\}$ and $\{t'_1.ts, \ldots, t'_m.ts\}$, if t is produced before t', then $\max_{i=1}^{m} t_i.ts \leq \max_{i=1}^{m} t'_i.ts$.

2.2 Plan Migration Methods

Moving states (MS) consists of three main steps: *state matching*, *state moving*, and *state re-computation*. The first step identifies matching states, i.e., pairs of states, in the old and the new plan respectively, whose tuples have the same schema. In Figure 1.1, S_A, S_B, S_C, S_D exist in both plans and are matching states. S_{AB} and S_{ABC} exist only in the old plan, whereas S_{CD} and S_{BCD} exist only in the new one. These correspond to unmatched states. State moving transfers the matching states of the old to the new plan. For instance, after the second step, S_A, S_B, S_C, S_D in the new plan of Figure 1.1, contain the tuples that existed in the corresponding state of the old plan immediately before migration. Note that the unmatched states (S_{BC}, S_{BCD}) of the new plan are still empty. The final step fills these states in a bottom-up manner. Specifically, MS first constructs S_{CD} by joining S_C and S_D. Then, it builds S_{BCD} by joining S_B and S_{CD}. The unmatched states S_{AB} and S_{ABC} in the old plan are then discarded and the migration process terminates.

The cost of MS is dominated by the re-computation step with complexity $O(w^h)$, where w is the length of the sliding window and h is the height of the operator tree [ZRH04]. As shown in the experiments of [ZRH04], MS is expensive for several stream settings. Furthermore, during its execution the output stream is suspended, potentially causing a bottleneck for the entire system.

Whereas MS exploits re-usability, *parallel track* (PT) utilizes parallelism by executing both the old plan and the new plan simultaneously and combining their results. Figure 2.1 illustrates the application of PT to the example of Figure 1.1. After the initialization of the new plan all its states are empty (i.e., there is no state moving and re-computation as in MS). The input streams A, B, C, D are connected to both plans, which run in parallel. Let *new* be the tuples that arrive after the start of the migration and *old* be the ones that already exist in S_A, S_B, S_C, S_D of the old plan. In order to avoid duplicates, results containing only *new* tuples are eliminated at the

topmost operator (*ABCD*) of the old plan, because they are generated by the new plan. Note that such duplicates cannot be eliminated before *ABCD* because a result of *ABC* that includes only *new* tuples may still be joined with an *old* D tuple. In general, the new plan produces only output where all tuples are *new*. Every other combination of *old* and *new* tuples is generated by the old plan. The migration terminates when the old plan contains only *new* tuples.

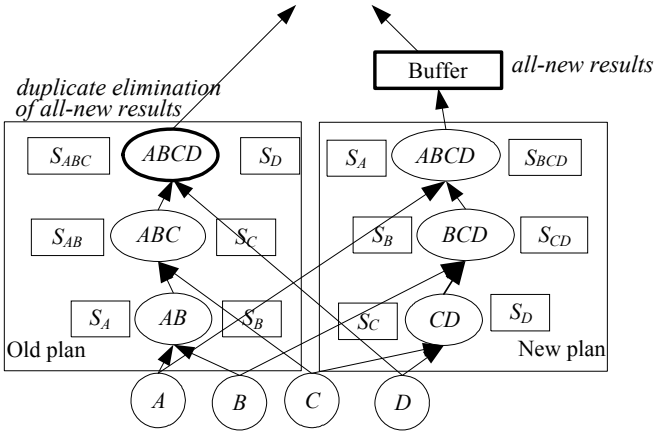


Figure 2.1 Example of PT

The parallel execution cannot guarantee the correct order of the output tuples, since results can be generated by the new or the old plan in any order. To overcome this complication, PT uses the following *PT ordering requirement*: for any two output tuples t and t' with component timestamps $\{t_1.ts, \ldots, t_m.ts\}$ and $\{t'_1.ts, \ldots, t'_m.ts\}$, if t is produced before t', then there must exist i ($1 \leq i \leq m$) such that $t_i.ts \leq t'_i.ts$. This requirement is implemented by directing the output of the new plan to a buffer. The contents of the buffer are released after the termination of the migration. Thus, every tuple t generated by the previous plan precedes any other t' produced by the new one. Since t contains at least an *old* component t_i, the condition $t_i.ts \leq t'_i.ts$ is always satisfied.

Note that this definition of temporal ordering is not compatible with the *max ordering requirement* discussed in Section 2.1. Assume, for instance, that t has component timestamps $t_1.ts=1$, $t_4.ts=4$, where t_1 is *old* and t_4 is *new*. Similarly, the components of t' are $t_2.ts=2$, $t_3.ts=3$,

where both t_2 and t_3 are *new*. According to *max ordering*, t should be generated after t' because it has a larger maximum timestamp ($t.ts$=4, $t'.ts$=3). PT, however, can only produce t before t' since it contains an *old* tuple. A straightforward extension of PT to capture other temporal ordering definitions, i.e. to buffer all results for a sufficiently long period and sort them before output, incurs significant performance penalty.

Another problem of PT refers to the output rate. According to [ZRH04], the migration duration for h>1 is $2w$, where w is the window size and h is the height of the operator tree. As discussed in Section 3.5, under the *max ordering* and *global ordering* assumptions, a tighter bound for the duration is w. Throughout migration, the only reported tuples come from the old plan since the new plan is buffered. These tuples have at least one *old* component and gradually expire, so that the output rate constantly decreases during the transition.

2.3 Alternative Approaches

Besides dynamic plan migration, there are other approaches to adaptive query processing that render plan migration either trivial, e.g., [VNB03, BMWM05] or unnecessary e.g., [AH00, MSHR02]. *M-Join* [VNB03] only stores past received tuples from the source streams. Each source maintains an ordering of other streams according to which the join is performed; plan migration simply means changing these orderings without any additional cost. The inherent drawback of this approach is that no intermediate results are stored and work must be repeated. In the old plan of our running example, an incoming tuple t_D from source stream D joins directly with intermediate ABC tuples stored in S_{ABC}. If *M-Join* were used, t_D would join with A tuples in S_A, then with B tuples in S_B, and finally with C tuples in S_C. In general, although *M-Join* requires less memory and simpler plan migration algorithm, its CPU cost can be prohibitively expensive as shown in the experiments of [BMWM05], especially when both the stream rates and the join

selectivity are high.

Adaptive caching [BMWM05] is based on *M-Join*, and stores the intermediate results in the form of caches. Migration can be performed by simply discarding all previous caches and gradually filling the new ones. When the transition starts, the system performance suddenly drops to that of the *M-Join*, and gradually improves. This method is only discussed for hash joins and its application to other join algorithms (e.g., nested loop joins), or other stateful operators, is not straightforward.

Eddies [AH00, MSHR02] incorporate a different perspective to adaptive query processing. Queries are processed without fixed plans. Instead, an optimized routing for each tuple is computed individually, so that the system is very adaptive to changes in data characteristics. The cost of this flexibility, however, is high computation overhead since optimization is performed at the tuple level. Moreover, similar to *M-Join*, since *Eddies* do not store intermediate results, they are inefficient for high stream rates and selective operators. An adaptive approach for the Eddies architecture [DH04] alleviates this problem by storing intermediate results, and performing explicit plan migrations using a method similar to MS.

Finally, plan migration is only one of several interesting problems related to query adaptivity in stream management systems. *Query scrambling* [UFA98] proactively joins received tuples to prepare for future changes in stream characteristics (while plan migration is performed after the change). *Operator scheduling* [BBD+04] adjusts resource allocation to different operators in order to reduce main memory consumption. These approaches can be combined with plan migration to enhance performance.

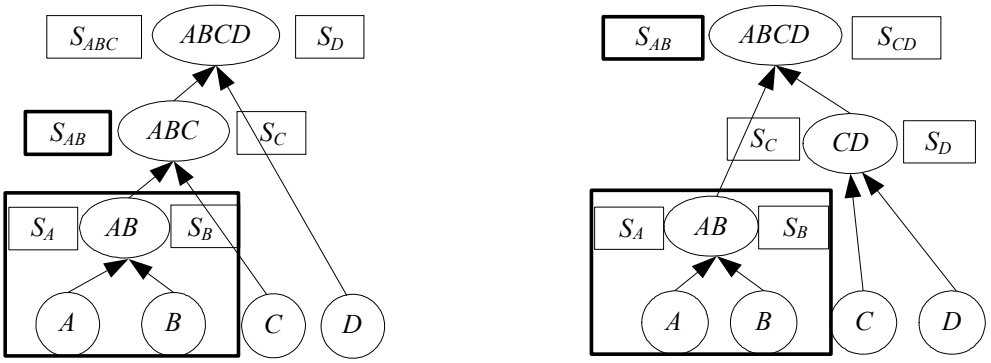
3. HybMig

HybMig integrates the re-usability aspects of MS with the parallel execution paradigm of PT.

Section 3.1 discusses a pre-processing step that performs sub-query sharing. Section 3.2 presents the main algorithmic module of HybMig. Section 3.3 explains the mechanism for preserving temporal ordering. Section 3.4 optimizes HybMig through a background state computation. Section 3.5 summarizes the properties of HybMig.

3.1 Sub-query Sharing

Sub-query sharing [KFHJ04] is commonly used in stream processing systems to eliminate redundant work. Figure 3.1 illustrates the application of sub-query sharing to dynamic plan migration. Note that the old (left-deep) and new (bushy) plans share operator *AB*. Clearly the state S_{AB} in the new plan does not need to be computed by joining tuples in S_A and S_B, but can be moved directly using a step similar to *state moving* in MS.



old plan: $P_{(((AB)C)D)}$ new plan: $P_{((AB)(CD))}$

Figure 3.1 A sharing opportunity

The application of this optimization to PT is not trivial. In order to provide a solution, observe that when the old and the new plans run in parallel, the common *sub-query AB* is computed twice. Figure 3.2 shows the HybMig solution for this plan migration task: the operator *AB* is shared between the old and the new plans, thus the redundant work is avoided. Equivalently, *AB* can be treated as a single source. If $E = AB$, the old plan can be expressed as $P_{((EC)D)}$ and the new one as $P_{(E(CD))}$.

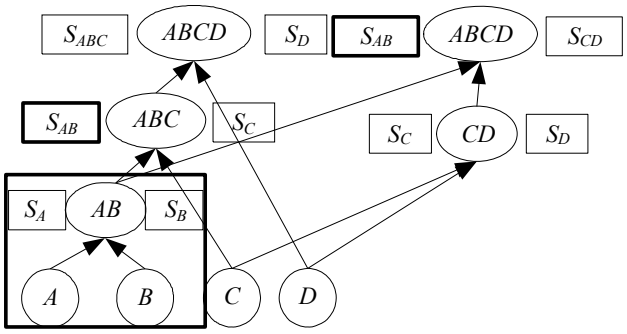


Figure 3.2 Migrating from $P_{(((AB)C)D)}$ to $P_{((AB)(CD))}$

In general, when the old and the new plans contain common sub-queries, we call the migration *reducible*. Another example of reducible task is the migration from $P_{(((AB)C)D)}$ to $P_{((A(BC))D)}$. The difference between the two plans is the join ordering for obtaining *ABC*; the last join operator which joins *ABC* with *D* can be shared between the two plans. HybMig treats reducible migration tasks with the *decomposition* module of Figure 3.3. Once the task is found to be reducible, it is decomposed into two smaller tasks (two pairs of new/old plans). The decomposition is done recursively until the migration task is non-reducible. A trivial migration task whose new and old plans are the same is simply discarded.

```
Decomposition (MigrationTask T, Operator o)
// Input= T: the current migration task
// o: the root operator of the current sub-tree in the old plan of T
1. For each input stream i of o
2.    If i is the output stream of another operator o'
3.       If there exists an operator n' in the new plan with semantically equivalent output as o'
4.          Treat o' and n' as source streams in T. Discard T if its old and new plans are identical
5.          Create a new migration task T' migrating the sub-tree in the old plan rooted at o' to the sub-tree
               in the new plan rooted at n'. Discard T' if its old and new plans are identical
6.          Decomposition(T', o');
7.       Else
8.          Decomposition(T, o');
```

Figure 3.3 The decomposition module of HybMig

In the example in Figure 3.1, the original migration task is decomposed into two smaller tasks: T_1: $P_{(AB)}$ *to* $P_{(AB)}$ and T_2: $P_{((EC)D)}$ *to* $P_{(E(CD))}$, where $E = (AB)$. T_1 is trivial and discarded. Decomposition constitutes a pre-processing step, which generates several non-reducible

migration tasks. For the following discussion, we assume that decomposition has already been applied and the only matching states are those whose schema contains a single source stream.

3.2 Shifting Workload

In Section 2.2, we defined as *new* the tuples that arrive after the start of the migration and *old* as the ones that arrived before. A join result may contain any possible combination of *old* and *new* tuples. Table 3.1 illustrates these combinations for the output of an operator *ABCD*. The first case (all components are *old*) has been produced before the start of the migration. The remaining ones are generated during or after the migration. For PT, the old plan produces tuples of cases 2-15, and the new one generates only tuples of case 16. Intuitively, especially at the beginning of migration, most of the work is performed by the old plan. Furthermore, since, as discussed in Section 2.2, duplicate elimination in the old plan occurs at the topmost operator, several of the intermediate results (containing only *new* components) may be redundant. The goal of HybMig is to shift the computation to the new plan as quickly as possible, and to eliminate duplicate results at the lowest operator.

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

Table 3.1 Old/new status for the four components

Assuming again the migration task of Figure 1.1, HybMig performs the state matching/moving step of MS, i.e., it moves S_A, S_B, S_C, S_D of the old plan to their corresponding operators in the new plan. Since the two plans run in parallel, we do *not* remove these states from the operators in

the old plan. Instead, they are *shared* between the two plans. In Figure 3.4, the four states S_A, S_B, S_C, S_D in the new plan are dashed to indicate that they are shared with the old plan. The dashed states are only *conceptual*, i.e., for a pair of matching states in the old and the new plan there exists only one list of tuples in the system since their contents are exactly the same during the entire migration. For instance, when a *new* tuple A arrives, there is a single insertion in S_A, which is reflected in both plans. Since matching states are shared and contain no duplication, the memory overhead of the migration is minimal. This step is called *state sharing*.

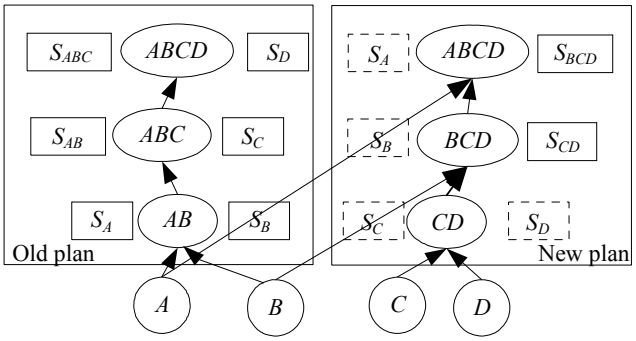


Figure 3.4 State sharing in HybMig

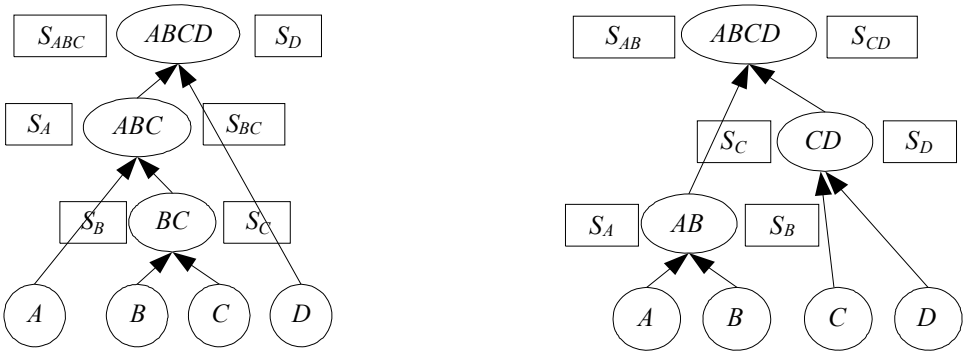
After state sharing, the only empty states in the new plan are S_{CD} and S_{BCD}, which are unmatched. Thus, the new plan can immediately produce results whose C and D components are not simultaneously *old*. More precisely, the plan can generate tuples of cases 2, 3, 4, 6, 7, 8, 10, 11, 12, 14, 15 and 16 in Table 3.1. The remaining cases (5, 9, and 13) must be produced by the old plan. Note that the new plan generates 12 out of the 15 cases (excluding case 1 that exists before migration) as compared with only 1 case using PT. This means that HybMig shifts the workload earlier (than PT) to the new plan, leading to better performance since the new plan is supposed to be much more efficient. This is verified in our experimental evaluation, where HybMig outperforms PT immediately after the beginning of the migration period.

It now remains to clarify how the output of the old plan is restricted to the cases 5, 9 and 13, not generated by the new one. The common property of these three cases is that both *C* and *D* components are *old*. Therefore, when a *new C* or *D* arrives, it should not be joined in the old plan. This implies that the input sources *C* and *D* can be *disconnected* from the old plan, as illustrated in Figure 3.4. Furthermore, a *new A* or *B* tuple should be joined only with *old C* and *D*. This is achieved by modifying the join predicates in the top two operators *ABC* and *ABCD*. Consider, for instance, a new tuple *AB* at the operator *ABC*. Since S_C is shared (i.e., common) in the two plans, it contains both *new* and *old C* tuples. In order to ensure that the *AB* tuple is joined only with *old C*, we change the join predicate of *ABC* to: *original predicate* AND (*C is old*). The second condition is satisfied if the timestamp of *C* precedes the starting time of migration ts_{start}.

HybMig_RD (QueryPlan P_{old}*, QueryPlan* P_{new}*)*
1. Suspend query processing
2. Connect all input sources of P_{old} to P_{new}, combine the output of P_{old} to P_{new} as the new output stream
3. Share all matching states between P_{old} and P_{new}
4. Identify the two source streams I_1 and I_2 that are joined first in P_{new}
5. Disconnect I_1 and I_2 from P_{old}
6. P_{old}*.join_predicate* = P_{old}*.join_predicate* AND (*I_1 is old*) AND (*I_2 is old*)
7. Resume query processing

Figure 3.5 HybMig for right-deep new plan

The case that the new plan is a bushy join is more complicated. Consider the migration task shown in Figure 3.6. After sharing the matching states, the new plan is not capable of producing two kinds of output tuples: (i) those that have *old A* and *B* (Case 2, 3 and 4), because the state S_{AB} is empty and (ii) those that have *old C* and *D* (Case 5, 9 and 13), because S_{CD} is empty. These combinations have to be generated by the old plan. In general, if the new plan is bushy, we cannot simply disconnect a source stream from the old plan because a result (of the old plan) may have a new tuple from any stream. In our example, there is always a case among the ones that cannot be produced by the new plan (2, 3, 4, 5, 9, 13) that has *new* in column *A*, *B*, *C* or *D*.



old plan: $P_{(((AB)C)D)}$ new plan: $P_{((AB)(CD))}$

Figure 3.6 Migration task to bushy plan

Instead of disconnecting inputs, HybMig prunes intermediate results using a *characteristics table* that summarizes the properties of the tuples that must be generated by the old plan. Table 3.2 illustrates these properties for the migration task of Figure 3.6. The "*" symbol means that the corresponding component can be either *new* or *old*. For example, a *new* tuple from *C* in the old plan, must join with an *old B* in the first join operator *BC*, and an *old A* in the second operator *ABC*. This is achieved by modifying the join predicates to *original predicate* AND ((*A and B are old*) OR (*C and D are old*)).

Case ID	*A*	*B*	*C*	*D*
C_1	*Old*	*Old*	*	*
C_2	*	*	*Old*	*Old*

Table 3.2 Characteristics table for Figure 3.6

Figure 3.7 illustrates the general HybMig algorithm, which also handles the case that the new plan is right-deep. In particular, let I_1 and I_2 be the inputs of the lowest operator in the right-deep join. The characteristics table contains a single entry <*,...,*, *old*, *old*>, meaning that the old plan can only generate tuples involving *old* I_1 and I_2. Thus, a *new* tuple I_1 or I_2 is immediately pruned, which in effect disconnects these two streams from the old plan. In terms of implementation, we can use tuple lineage for efficient pruning of intermediate results. Specifically, a bitmap is appended to each intermediate tuple indicating which entries in the characteristics table are

violated. Whenever we join with new tuples in the operator states, we check all currently satisfied entries and update the bitmap for the result tuple. If all entries are violated, the intermediate tuple is discarded.

HybMig (QueryPlan P_{old}, QueryPlan P_{new})
1-3. // the first three steps are the same as *HybMig_RD* shown in Figure 3.5
4. Condition $C_{characteristics}$ = false;
5. For each pair of source streams I_1 and I_2 that are directly joined in P_{new} in a lowest-level operator
6. $C_{characteristics} = C_{characteristics}$ *OR* (I_1 *and* I_2 *are old)*
7. P_{old}. *join_predicate* = P_{old}. *join_predicate* AND $C_{characteristics}$
8. Resume query processing

Figure 3.7 The general HybMig algorithm

3.3 Preservation of Temporal Ordering

Recall from Section 2.2 that PT adopts the requirement that if an output tuple t is produced before t', then there must exist a pair of components such that $t_i.ts \leq t'_i.ts$. In order to satisfy this requirement, it buffers the output of the new plan until the end of the migration period, increasing the memory consumption and decreasing the output rate. Moreover, the *PT ordering requirement* is not compatible with the commonly used *max ordering requirement* which states that if t is produced before t', then $max_{i=1}^{m}\ t_i.ts \leq max_{i=1}^{m}\ t'_i.ts$. HybMig can adhere to *both* temporal ordering requirements, by adopting two mechanisms that incur minimum overhead.

The first mechanism is shown graphically in Figure 3.8a. When a new tuple t arrives from a source stream, HybMig first feeds it to the old plan (Step 1) and produces results (Step 2), which are directly inserted into the output queue. After that, t gets processed in the new plan (Step 3), possibly generating more results (Step 4). If another tuple t' arrives during the processing of t, it waits until all four steps are finished, after which the Step 1 of t' starts. Note that t' is not necessarily from the same source stream as t. This processing order takes precedence over scheduling mechanisms in a DSMS, e.g., [BBD+04].

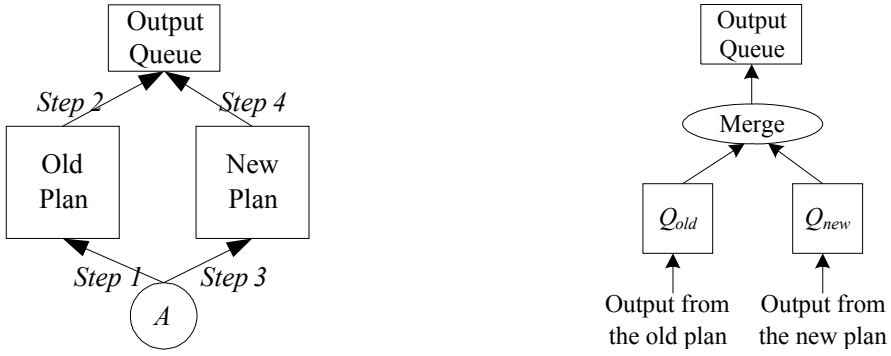


Figure 3.8 Order preserving mechanisms of HybMig

Proof of correctness: We first show that the above implementation of HybMig always complies with the *max ordering requirement*. Based on the global ordering assumption, the timestamp $t_i.ts$ of an incoming tuple t_i is larger than (or equal to) that of all the tuples stored in the operator states. This means that during the processing of t_i, the largest component timestamp of all the generated results is $t_i.ts$. Now assume two output tuples t and t' such that t is generated before t'. Let t_i and t'_i be the latest components of t and t', respectively. In order to satisfy the *max ordering requirement* it must hold that $t_i.ts \leq t'_i.ts$. Since t appears before t' in the output queue, t_i must arrive before t'_i because HybMig completes generating all outputs for one incoming tuple before starting another. Thus, due to the global ordering assumption, $t_i.ts \leq t'_i.ts$.

Next we show that HybMig satisfies the *PT ordering requirement*, given that the old and the new plan comply with it. Assume again output tuples t (with latest component t_i) and t' (with latest component t'_i) such that t is generated before t'. Our goal is to show that there is a pair of components such that $t_k.ts \leq t'_k.ts$. If t and t' are generated by the same plan, then they satisfy PT ordering by default. Similarly, if t_i and t'_i are different, then $t_i.ts \leq t'_i.ts$, as shown in the proof for *max ordering*. Thus, it remains to analyze the case that (i) t and t' are generated by different plans and (ii) $t_i = t'_i$. Since t_i is the latest component of t and t', they were both generated during the

processing of t_i. HybMig completes the processing of t_i in the old plan before feeding it to the new one. Thus, t must be generated by the old plan and t' by the new one. Therefore, t must satisfy at least one case in the characteristic table, meaning that at least one component t_k is *old*. The counterpart t'_k of this component in t' is *new*, implying that $t_k.ts \leq t'_k.ts$. □

In some systems involving distributed operators, it may not be possible to process tuples in the order imposed by the four steps of Figure 3.8a. In these cases, we allow parallelism by using two FIFO queues Q_{old} and Q_{new} to synchronize the outputs, as shown in Figure 3.8b. Specifically, the tuples generated from the old and the new plans are appended to Q_{old} and Q_{new}, respectively. A special *merge* operator ensures temporal ordering by combining the tuples from the two queues so that their order is exactly the same as that of the serialized version. In particular, the operator extracts all tuples from Q_{old} with the same latest component t_i (i.e., generated by the processing of t_i). Next, it switches to Q_{new} reading tuples (also generated by the processing of t_i) until the latest component changes. Then, it goes back to Q_{old} and repeats the same process in a round-robin fashion. This solution allows a high degree of parallelism, at the cost of additional CPU/memory overhead for maintaining the queues.

3.4 Background State Computation

Background state computation (BSC) is an optimization of HybMig that reduces the migration duration for systems involving long windows and relatively low arrival rates. Such a system may have additional capacity during the migration, which is utilized by BSC to compute the operator states of the new plan in the background. The migration terminates when BSC completes all operator states in the new plan. Similar to MS, the computation of the states is performed in a bottom-up manner. The major difference is that MS must complete *all* operator states before returning any result, whereas *every* state computation of BSC has an immediate effect on the

main thread of HybMig.

Consider the example in Figure 1.1 and its HybMig solution in Figure 3.4. BSC first computes the state S_{CD} by joining the *old* tuples in S_C and S_D. Then, it merges the computed S_{CD} (denoted as S_{CD}^{BSC} in the following) with the currently running S_{CD} (denoted as S_{CD}^{Main}), which contains only *new* components, to obtain the complete operator state. This merging is done by prepending S_{CD}^{BSC} to S_{CD}^{Main}. For example, when $S_{CD}^{\text{BSC}} = (t_1, t_2, t_3)$ and $S_{CD}^{\text{Main}} = (t_4, t_5)$, the merged state will be $(t_1, t_2, t_3, t_4, t_5)$. When the operator states are implemented as linked lists, the merging simply means linking two lists together, which takes constant time. In addition, this merging step preserves both *max* and *PT* temporal ordering, since S_{CD}^{BSC} contains only *old* tuples while S_{CD}^{Main} contains only *new* tuples.

The main thread then uses the complete S_{CD}, meaning that the only incomplete state in the new plan is S_{BCD}. Accordingly, the only kind of output tuples that cannot be generated by the new plan are those whose *B*, *C*, *D* components are simultaneously *old* (Case 9). Therefore, we can immediately disconnect source stream *B* from the old plan. This accelerates HybMig because more workload (Case 5 and Case 13) is shifted to the new plan.

After the computation of S_{CD}, the background thread computes S_{BCD}, which is subsequently merged with the currently running S_{BCD}. The old plan is discarded and the migration is over. In general, if the new plan is a right-deep join, the computation of each state (by BSC) causes the disconnection of one input from the old plan. The stream to be disconnected can be determined by examining the schema difference between the last and next computed state. In our example, after the termination of S_{CD} and before the computation of S_{BCD}, stream *B* should be disconnected.

If the new plan is bushy, BSC always chooses one of the lowest states to compute, breaking ties arbitrarily. In the example of Figure 3.6, the two states S_{AB} and S_{CD} to be computed are on the same level, so BSC can start with either one, say, S_{AB}. After computing S_{AB}, the entry <*old, old, *, **> in the characteristic table can be removed, and correspondingly the workload of producing three cases (2, 3, 4) of output tuples is shifted to the new plan. Since the characteristics table now contains a single entry <**, *, old, old*>, the effect of removing <*old, old, *, **> is the same as that of disconnecting streams *C* and *D* from the old plan. Note that the speed-up is higher than that of right-deep plans, where a state computation disconnects only one input. The migration terminates after the completion of the last state S_{CD}. Figure 3.9 shows the general BSC algorithm handling all types of plans.

BSC (QueryPlan P_{new})
// running in a background thread
1. Repeat
2. Identify the lowest incomplete state *S* in P_{new}, breaking ties arbitrarily. Suppose the two child states of *S* are S_1 , S_2
3. *BSCJoin*(S_1, S_2, *S'*) *// S' is the join result of old S_1, S_2 tuples*
4. $P_{new}.S = S' + P_{new}.S$ *// S is then complete*
5. Notify *HybMig* to remove the disjunctive item corresponding to *S* from $C_{characteristics}$
6. If the sibling state of *S* is *complete in* P_{new}
7. Let S_p be the parent state *of S*
8. *C* = (all source streams in S_p are *old*)
9. Notify *HybMig* that $C_{characteristics} = C_{characteristics}$ OR *C*
10. Until all states in P_{new} are completed
11. Notify *HybMig* that the migration is complete

Figure 3.9 *Background state computation*

As a further optimization, BSC uses a slightly different join algorithm from that of MS. A key observation is that old tuples gradually expire when BSC is executed in the background thread. In order not to produce a result tuple that expires before the computation is completed, we join the tuples in *reverse temporal order* using the symmetric join algorithm. The computation terminates when all remaining tuples expire. This join algorithm is described in Figure 3.10.

```
BSCJoin (OperatorState S1, OperatorState S2, OperatorState Sout)
// Input= S1, S2: the two operator states to be joined
// Output=Sout: the state obtained by joining tuples in S1 and S2
1.  Tuple ti = S1.last, tj = S2.last;  // last: the tuple with latest timestamp
2.  While ti ≠ NULL Or tj ≠ NULL
3.      If tj = NULL Or (i ≠ NULL And ti.ts >tj.ts)
4.          If ti.ts + w < now   // meaning ti is expired
5.              Return
6.          For tk = S2.last DownTo tj Step tk = tk.previous_tuple
7.              tjoin = ti JOIN tj
8.              Prepend tjoin to Sout if it satisfies the join predicate
9.          ti = ti.previous_tuple  // ti is assigned NULL if it has no previous tuple
10.     Else
11.         Symmetric to the above case
```

Figure 3.10 The join algorithm used in BSC

The correctness of BSC lies in two facts. First, BSC does not cause any missing or duplicate results. This is because at any time instance, the main thread HybMig properly divides the workload between the two plans, i.e. a result tuple will be produced either by the old plan or by the new plan. Second, the inclusion of BSC in HybMig does not violate its temporal ordering constraint, independently of the temporal ordering (*max* or *PT*) used. This is because both *BSCJoin* and the state merging procedure preserve temporal ordering.

3.5 Discussion

Assuming that the system has sufficient processing capacity (at least equal to the amortized processing cost per tuple), the migration duration of HybMig is w. After w, the old plan cannot produce any results and thus is safely removed. This can be proven by contradiction. Suppose that the old plan generates a result tuple t after $ts_{start} + w$, where ts_{start} is the beginning of migration. According to the division of workload between the two plans, t must contain at least one *old* component t_j with $t_j.ts < ts_{start}$. Meanwhile, the latest component of t must be a *new* tuple t_i with $t_i.ts > ts_{start} + w$. Then, $t_i.ts - t_j.ts > w$, contradicting the global window constraint. Note that this result is independent of the temporal ordering (*max* or *PT*) and applies to both HybMig and PT. When BSC is invoked, the migration duration of HybMig depends on the termination

time of the background thread. If BSC completes all states in the new plan at $t_{bsc\_finish} < t_{start} + w$, the migration duration is $t_{bsc\_finish} - t_{start}$; otherwise, it remains w.

In summary, HybMig integrates the advantages of PT and MS: (i) for low stream rates, BSC utilizes the idle system resource to compute states, thus behaving as an optimized[2] version of MS, without suspending the output stream; (ii) for high stream rates (where background computation is not possible), it behaves more like optimized PT, with improved re-usability aspects, higher output rates and better-founded temporal ordering semantics.

4. Extending to Arbitrary Plans

In practice, a plan may involve other stateful operators beyond joins. Such an example is the *user-defined aggregation operator* (UDA), used in systems like Aurora [ACC+03]. Since UDA is Turing complete [LWZ04], its internal states can be very complex. On the other hand, MS, PT, as well as HybMig, focus on migration tasks involving only join re-orderings. In this section we generalize the three techniques to handle arbitrary plans treated as black boxes. Section 4.1 presents the *generalized moving states* (GMS), Section 4.2 the *generalized parallel track* (GPT), and Section 4.3 the *generalized hybrid migration* (GHM) approach. Section 4.4 concludes with an analytical comparison of the three techniques.

4.1 Generalized Moving States

GMS first suspends the output stream, extracts *old* tuples[3], i.e. those that arrived between $ts_{start} - w$ and ts_{start}, and feeds them to the new plan in the order of timestamps. Recall, that MS would re-compute operator states. In GMS this is not possible, since plans constitute black boxes. Instead, GMS (i) suspends the input streams, (ii) *runs* the new plan to process the old tuples and (iii)

[2] Recall that unlike MS, in BSC each state computation has an immediate effect on the output rate.
[3] In order to do this the system either provides an interface to extract old tuples from operator states as in MS, or proactively keeps a copy of tuples from source streams as in [BBD05].

ignores all outputs produced. When this process terminates, the old plan is discarded and migration is complete. The input streams are connected to the new plan, and query processing is resumed. The correctness of GMS lies in the fact that the operator states in the new plan are properly built before it starts to generate outputs.

4.2 Generalized Parallel Track

The most challenging part for adapting PT to the general migration problem is how to combine the results of the two plans so that duplicates are avoided and temporal ordering is preserved. The answer of GPT to this question is surprisingly simple: it does not combine the results of the two plans during the migration. Instead, GPT directly outputs all results from the old plan and connects the output of the new one to a *null sink* as shown in Figure 4.1. The duration of migration using GPT is *w*, after which both the old plan and the null sink are discarded. Query processing resumes with the new plan.

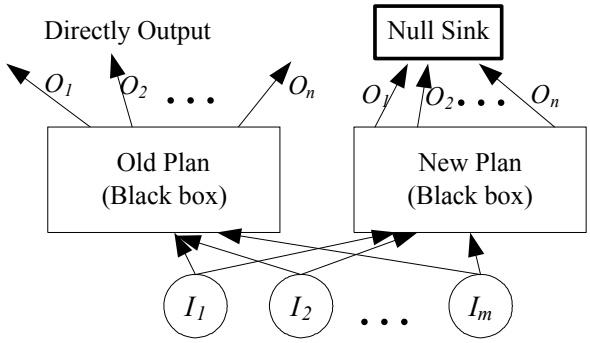


Figure 4.1 The Generalized PT solution

Proof of correctness: The outputs of GPT are correct, complete and properly ordered both *during* and *after* the migration. During migration, the new plan P_{new} has no impact on the output; the old plan P_{old} does not undergo any modifications and thus continues to generate correct outputs. After the migration, P_{old} is dropped and P_{new} continues to generate outputs. Thus, it suffices to prove that P_{new} generates exactly the same result as P_{old} if P_{old} were still running. Suppose that there is a new plan P'_{new} identical to P_{new} that started at the same time as P_{old}. Since the new and

the old plans are semantically equivalent, the outputs of P'_{new} and P_{old} are indistinguishable.

Next we show that P_{new} has exactly the same operator states as P'_{new}, and, therefore, generates identical results. A key observation is that after w, all tuples that arrived before the start of migration have expired and do not affect the operator states of P'_{new}. The only tuples that influence the states of P'_{new} are those that arrive during the migration. Since P_{new} receives all these tuples, its operator states are identical to those of P'_{new}. Therefore, after the migration, the output of P_{new} is exactly the same as P'_{new}, which is identical to that of P_{old}. □

Note that GPT, when applied to join reordering, overcomes the decreasing output rate and temporal ordering incompatibility problems of PT. In particular, the output rate is the same as if only the old plan were running during the entire migration period. Furthermore, since the output of GPT is identical to that of P_{old}, the ordering of results is correct under *any* definition of temporal ordering.

4.3 Generalized Hybrid Migration

GHM combines the general frameworks of GPT and GMS to simultaneously meet responsiveness goals and minimize the migration duration. As shown in Figure 4.2, GHM is similar to GPT, with three notable differences. First, the inputs are not directly connected to the new plan. Instead, each feeds to a FIFO queue, initialized with *old* tuples extracted from the old plan (i.e., similar to GMS). The second difference between GHM and GPT is that during migration, the old plan is given a higher priority in order to achieve high output rate. This is performed through a process similar to *background state computation*, i.e., the new plan is executed as a background thread. Finally, the duration of migration may be shorter than w. In particular, the migration task terminates when all the queues are empty, signaling that the two plans are consuming the same inputs. Then the old plan, the *null sink*, and the input queues are

discarded. The inputs are linked directly to the new plan, and the outputs of the new plan are connected to their corresponding output queues. Compared to GPT, GHM is more proactive since it exploits spare system capacity to accelerate the migration.

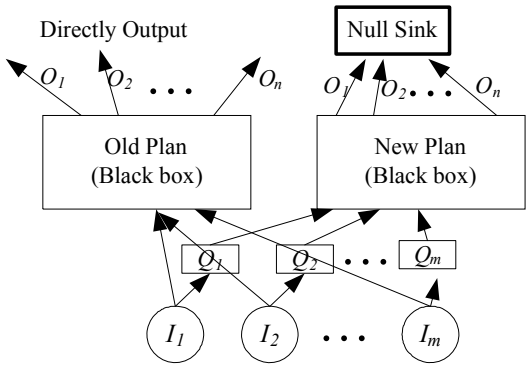


Figure 4.2 The Generalized Hybrid Migration solution

Proof of correctness: Similar to GPT, during migration, the old plan generates correct results. The non-trivial part of the proof is to show that after migration the new plan P_{new} has the correct states. Suppose ts_{finish} is the timestamp that the migration terminates. Clearly, ts_{finish} is later than the timestamp ts_{start} when the migration starts, thus ts_{finish}-$w \geq ts_{start}$-w. At ts_{finish}, the new plan has processed all tuples from ts_{start}-w to ts_{start} extracted from the old plan, and all incoming tuples from ts_{start} to ts_{finish}. Therefore, the new plan has processed all tuples in the last w time units. As in the case of GPT, the new plan has the correct operator states and continues to produce correct outputs after the migration. □

4.4 Analysis

Consider that during migration the average stream rate for each stream is τ, the total number of streams is n and the average time required by the old (new) plan to process an input tuple is denoted as T_{old} (T_{new}). For GMS, the migration duration is equal to the total CPU time spent on processing the *old* tuples. Assume that the average stream rate between $ts_{start} - w$ and ts_{start} is τ', and the average CPU time consumed (by the new plan) for processing one *old* tuple is T'_{new}. Note

that the new plan is supposed to be better at processing *new* tuples than *old* ones, i.e., $T'_{new} \geq T_{new}$. Then, the migration duration/CPU time of GMS is

$$Duration_{GMS} = CPU_{GMS} = T'_{new} \times \text{number of } old \text{ tuples} = T'_{new} \times (\tau' w \times n) = \tau' n w T'_{new}$$

During the migration period, the system is kept at its peak load, and thus incoming tuples are buffered in the input queues waiting to be processed. The number of these tuples can be estimated by $\tau n \times Duration_{GMS} = \tau n(\tau' n w T'_{new})$. After the migration, the system starts to process these buffered input tuples and produce results. Consequently, there is a sharp spike in the output rate. At each time unit during this spike, $1/T_{new}$ input tuples are processed and at the same time τn new input tuples flow in. Therefore, the length of this spike can be estimated by

$$SpikeLen_{GMS} = \tau n(\tau' n w T'_{new}) \Big/ 1/T_{new} - \tau n = \tau \tau' n^2 w T'_{new} T_{new} \Big/ 1 - \tau n T_{new}$$

GPT executes both plans in parallel during its migration duration: $Duration_{GPT} = w$. Considering that τn input tuples arrive at each time unit, overall $\tau n \times w$ tuples need to be processed by both plans. Thus, its CPU cost is

$$CPU_{GPT} = \tau n w (T_{old} + T_{new})$$

The old plan produces all outputs. Considering that the old plan has a higher priority, one can assume that the new plan starts processing a tuple after the old plan completes it. Thus, given that the system has sufficient processing capacity, the average delay of an output tuple is T_{old}.

Finally in the case of GHM, the migration period can be divided into two phases with duration d_1 and d_2 respectively. The first phase ends when the new plan has processed all *old* tuples. Similar to GMS, the new plan (running as a background thread) needs to process $\tau' n w$ *old* tuples, each costs T_{new} time. Meanwhile, the old plan (main thread) has to process $\tau n d_1$ new

arrivals during d_1. Considering that the system is at its peak load, we have

$$d_1 = \tau' n w T'_{new} + \tau n d_1 T_{old}$$

$$\text{Therefore, } d_1 = \tau' n w T'_{new} \Big/ 1 - \tau n T_{old}$$

The second phase ends when the migration is over, i. e. when all input queues are empty. Similar to the first phase, the length d_2 is equal to the total CPU time of the two threads during the second phase:

$$d_2 = \tau n (d_1 + d_2) T_{new} + \tau n d_2 T_{old}$$

$$\text{Therefore, } d_2 = \tau n d_1 T_{new} \Big/ 1 - \tau n T_{new} - \tau n T_{old} = \tau \tau' n^2 w T_{new} T'_{new} \Big/ (1 - \tau n T_{old})(1 - \tau n T_{new} - \tau n T_{old})$$

Thus, the total migration duration (which equals total CPU time similar to GMS) of GHM is

$$Duration_{GHM} = d_1 + d_2 = \tau n d_1 T_{new} (1 - \tau n T_{old}) + \tau \tau' n^2 w T_{new} T'_{new} \Big/ (1 - \tau n T_{old})(1 - \tau n T_{new} - \tau n T_{old})$$

Because all results are produced by the old plan (main thread), the average output delay is the same as for GPT, i.e., T_{old}.

5. Experiments

We have implemented the proposed algorithms, as well as MS and PT, in C++ using the PIPES data stream management system [KS04]. For our experiments, we use the same methodology as [ZRH04]. Specifically, we pick two different query execution plans, P_{old} (before the migration) and P_{new} (after). Every experiment consists of two phases: a *warm-up* and a *migration* phase. During the warm-up, P_{old} is executed for a sufficiently long time (5 times the window length in all experiments), with the stream characteristics set in favor of the old plan. Then, we alter the stream properties, so that P_{new} becomes more efficient, and start the migration phase. All joins

are implemented using the nested loop algorithm.

The experiments investigate the output rate, memory consumption and the CPU cost (during the migration phase), as a function of the following parameters: (i) size of the sliding window, (ii) stream rate, and (iii) number of participating streams. The ranges of these parameters[4] are summarized in Table 5.1, with the default values shown in bold. All the experiments are executed on a workstation with two Pentium IV 3.0GHz CPUs and 2Gbytes main memory. We use the *serialized* mechanism (see Figure 3.8) to preserve tuple ordering in HybMig, but the *parallel* mechanism leads to similar conclusions. Section 5.1 presents a comparison of HybMig with existing methods, Section 5.2 focuses on the effects of *sub-query sharing* and *background state computation* on HybMig, and Section 5.3 discusses the generalized algorithms.

Parameter	***Range & Default***
Window size w (in minutes)	1, 2, **3**, 4, 5
Stream rate τ (in tuples/second)	0.4, 0.7, **1**, 1.3, 1.6
Number of streams n	4, 5, **6**, 7

Table 5.1 Parameters under investigation

5.1 Comparison of HybMig, MS and PT

In the first set of experiments, the query to be evaluated is a clique-join of all streams, i.e., there is a join condition between each pair of streams. P_{old} is always an extreme left-deep plan, while the new plan can be right-deep (denoted as P_{new}^{r}) or bushy (P_{new}^{b}). During the warm-up phase, we feed the first stream (A) with rare values that are difficult to join with other streams. We use sel_h to denote this high selectivity (of joining A with another stream) and sel_l for the selectivity of joining any other pair of streams. In all experiments, we set sel_l=0.05 and sel_h=0.0025. Since joining A with other streams leads to small output size, an extreme left-deep join is the best plan (P_{old}) during this phase.

[4] For simplicity we assume that all streams have the same arrival rate. Experiments with different rates for different streams yield similar results and are omitted for brevity.

When the migration phase begins, we change the stream values such that the last stream (e.g., F when joining 6 streams), instead of the first one, is fed with rare values (leading to sel_h). Accordingly, a right-deep join (P_{new}^{r}) that processes F first is preferable during (and after) the migration phase. A bushy plan is constructed by modifying the corresponding right-deep plan so that the $(n\text{-}4)^{th}$ and the $(n\text{-}3)^{th}$ streams are joined first. Since bushy plans also join the last stream first, they are more efficient than left-deep plans. Table 5.2 illustrates all plans used in this set of experiments. Note that these migration tasks do not have sharing opportunities. Furthermore, we do not apply BSC since we want to demonstrate the superiority of HybMig without any optimizations. Sub-query sharing and BSC are evaluated independently in Section 5.2.

n	P_{old}	P_{new}^{r}	P_{new}^{b}
4	$P_{(((AB)C)D)}$	$P_{(A(B(CD)))}$	$P_{((AB)(CD))}$
5	$P_{((((AB)C)D)E)}$	$P_{(A(B(C(DE))))}$	$P_{(A((BC)(DE)))}$
6	$P_{(((((AB)C)D)E)F)}$	$P_{(A(B(C(D(EF)))))}$	$P_{(A(B((CD)(EF))))}$
7	$P_{((((((AB)C)D)E)F)G)}$	$P_{(A(B(C(D(E(FG))))))}$	$P_{(A(B(C((DE)(FG)))))}$

Table 5.2 All query execution plans

Figure 5.1 compares the output rate of MS, PT, HybMig as a function of time (migration starts at time 0), using the default settings, P_{old} and P_{new}^{r}. The output rate of MS is zero during the first 50 seconds of the migration phase because MS re-computes the states in the new plan. As shown in the next experiment, during this time the system is completely saturated. After that, MS processes delayed input tuples and the output rate soars to 42.8 tuples per second. The output rate of PT constantly decreases, reaching zero at w (180 seconds). This happens because eventually the old tuples expire, while the output of the new plan is buffered. The burst after the end of the migration occurs when PT releases the contents of the buffer. The spikes of the output rate in both MS and PT may lead to system overload and should be avoided. On the other hand, HybMig has a steady output rate. In particular, no output tuple is delayed more than 1 second. The results for bushy plans are similar and omitted.

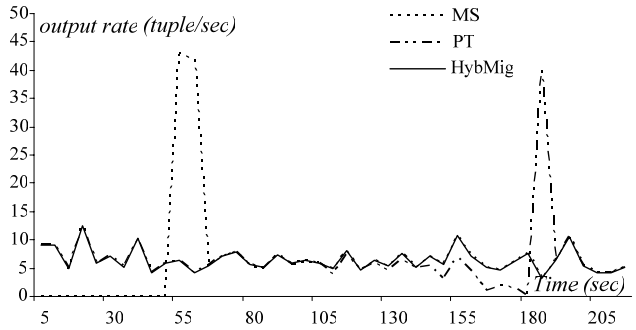
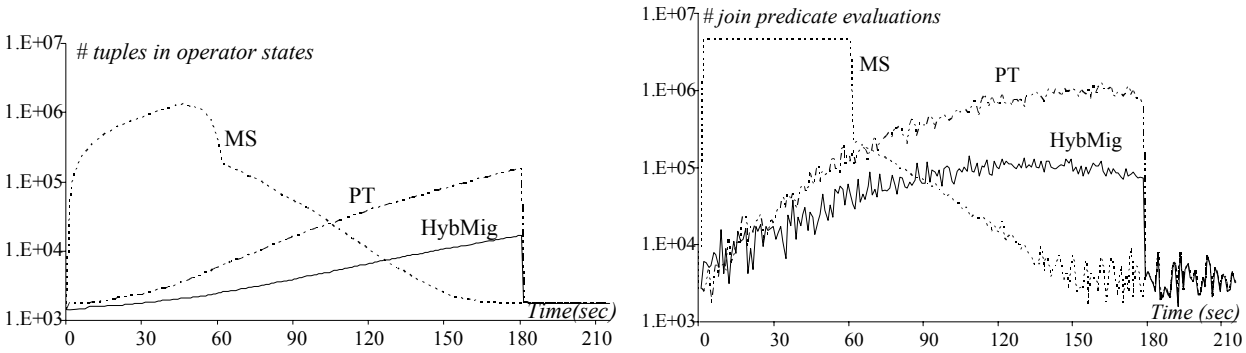


Figure 5.1 Output Rate/Time (w=3,τ=1,n=6,P_{old} to P_{new}^{r})

Next we compare the memory consumption (Figure 5.2a) and CPU cost (Figure 5.2b) over 1.2w time (216 seconds) after the migration starts. For each second, which is the finest time unit, the space consumption is measured as the total number of tuples in all operator states at the end of the second. The CPU cost is the total number of times that the join predicate is evaluated during that second. We use these measures because they are platform-independent and can be calculated accurately. Both memory and CPU axes are in logarithmic scale.



a. Memory consumption b. CPU cost

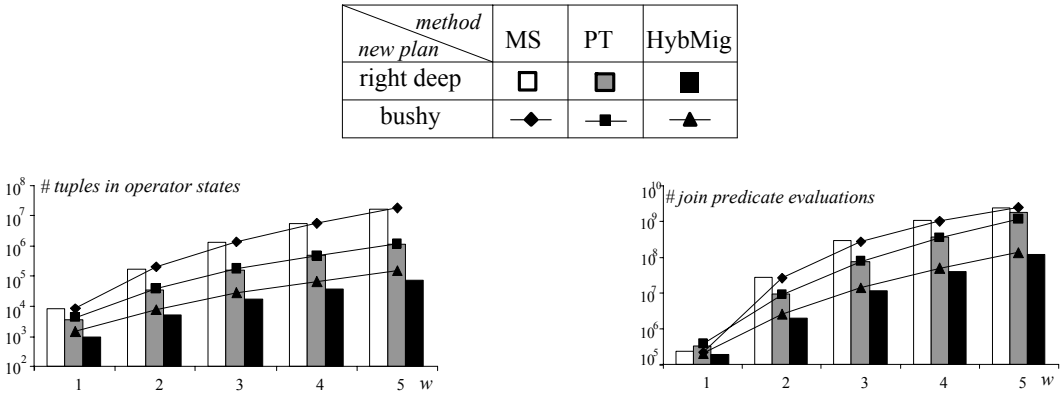
Figure 5.2 Overhead/Time (w=3,τ=1,n=6,P_{old} to P_{new}^{r})

According to Figure 5.2b, for MS the system is initially saturated due to the state re-computation operations. This leads to high space consumption as the operator states are filled and the new tuples wait to be processed. After that both memory and CPU costs drop until w, at which point all old tuples expire. Note that the memory consumption starts decreasing earlier (at 49 sec) than the CPU cost (at 60 sec). This is because after state re-computation terminates at the 49th second,

the system has to process delayed inputs, during which the system is still saturated but memory cost drops quickly. This period corresponds to the spike in the output rate of MS shown in Figure 5.1. On the other hand, the memory and CPU overhead of PT constantly increases until *w*, reflecting the inefficiency of the old plan. HybMig has a more balanced behavior and lower overall cost than both algorithms. Furthermore, it outperforms PT at every time-stamp. The sudden decrease of the overhead for HybMig (and PT) at *w* is caused by the elimination of the old plan, which signals the end of migration.

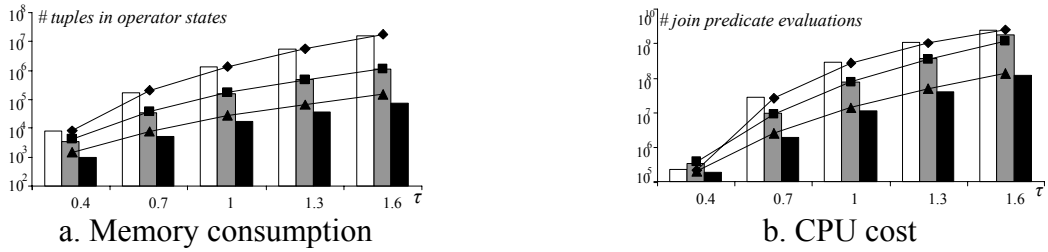
We now investigate the effect of the window size (minutes), average stream rate (tuples/second) and number of participating streams. Specifically, we set two parameters to their default values, and vary the third one in the range shown in Table 5.1. Figures 5.3 - 5.5 present the memory and CPU overhead for right deep plan P_{new}^{r} (columns) and bushy plan P_{new}^{b} (lines). In each experiment, we measure peak memory and total CPU cost during the migration phase. All vertical axes are in logarithmic scale. HybMig clearly outperforms MS and PT by a wide margin under all settings. These results, combined with the constant output rate (Figure 5.1) and stable behavior (Figure 5.2) confirm the superiority of HybMig over the existing approaches.

Note that in Figure 5.4 we limit the maximum stream rate to 1.6 tuples per second, because (i) the CPU overhead of MS and PT increases fast with the rate, and the experiments (for MS and PT) take very long to terminate; (ii) when the stream rate exceeds 2.3, MS causes memory overflow. Furthermore, the rates remain constant throughout the lifespan of the streams. We follow this approach since we change the stream characteristics (and the plan of choice) by altering the data distribution (i.e., the join selectivity as discussed in the beginning of the Section). The important issue is that the new plan fits the new stream characteristics better than the old one, independently on why the change occurred (e.g., due to distribution or stream rate).



a. Memory consumption b. CPU cost

Figure 5.3 Overhead/Window w (τ=1,n=6)



a. Memory consumption b. CPU cost

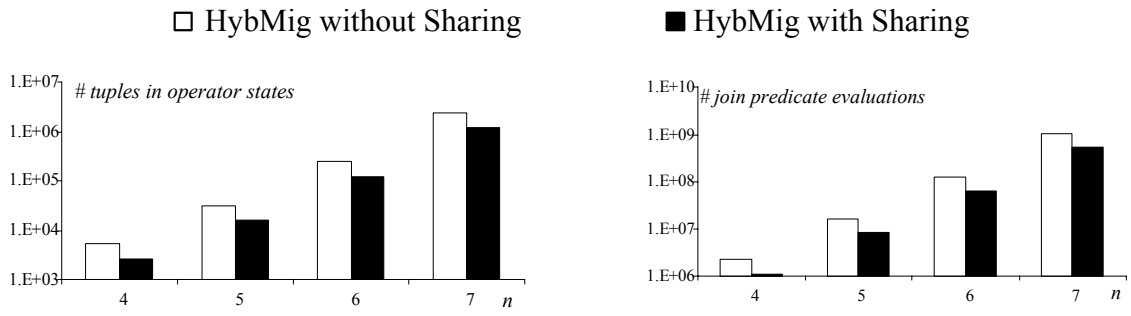
Figure 5.4 Overhead/Stream rate τ (w=3, n=6)

a. Memory consumption b. CPU cost

Figure 5.5 Overhead/Number of streams n (w=3, τ=1)

5.2 Effects of Optimizations

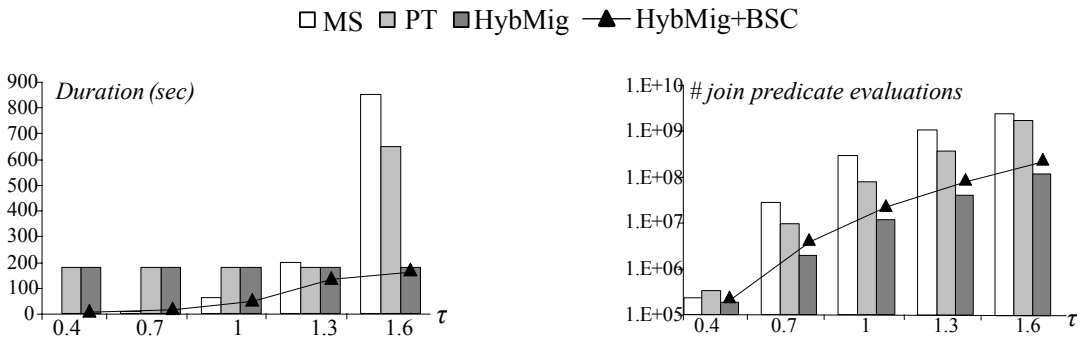
In order to evaluate sub-query sharing, we use a set of migration tasks with sharing opportunities: (i) P_{old} is the same as in Section 5.1, while (ii) P_{new} is constructed by changing P_{old} to a bushy plan that joins the last two streams first. For example, when n=6, $P_{old} = P_{(((((AB)C)D)E)F)}$ and $P_{new}=P_{((((AB)C)D)(EF))}$. Figure 5.6 compares two versions of HybMig, with and without sharing, as a function of number of participating streams n (the remaining parameters w and τ have their default values). Sharing improves performance up to a factor of 2.



a. Memory consumption b. CPU cost

Figure 5.6 Effect of sub-query sharing/ n (w=3, τ =1)

Recall that *background state computation* uses idle CPU cycles to reduce the duration of migration when the system is under-utilized (i.e., low arrival rates). Figure 5.7a illustrates the migration time as a function of the stream rates. The diagram also includes MS and PT, since they were not evaluated on this aspect in Section 5.1. MS has the shortest duration for τ up to 1 tuple/sec, and the longest for larger values. This is because for low rates, the number of tuples in operator states is small and the unmatched states of the new plan can be efficiently computed (signaling the end of migration). On the other hand, for $\tau > 1$, the computation of intermediate states becomes very expensive and the migration duration may exceed w several times (about 5 for τ =1.6). The duration for PT is always w (=180 seconds) except when the system is overloaded (e.g., τ =1.6), because in this case the total CPU cost exceeds the window length.



a. Duration b. CPU cost

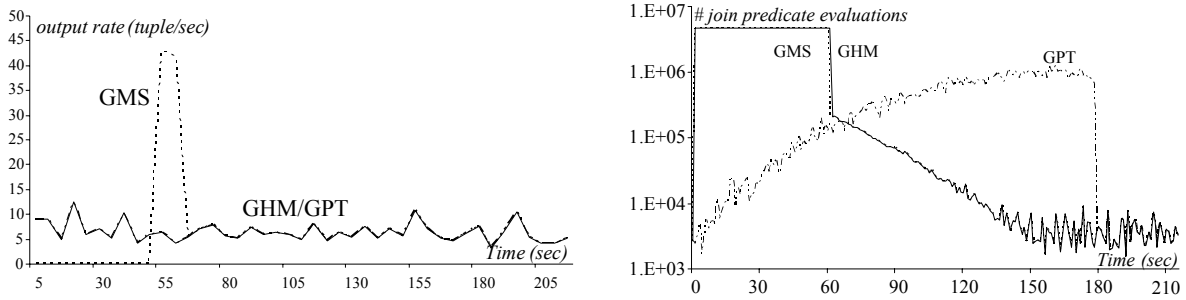
Figure 5.7 Effect of BSC/ τ (w=3, n=6,P_{old} to P_{new}^{r})

In HybMig (without BSC) the migration requires 180 seconds in all cases because, due to the

lower CPU cost, the system can handle the tested arrival rates. The application of BSC reduces significantly the migration time for low arrival rates (i.e., the duration is negligible for τ =0.4 and τ =0.7). As the rates increase, the effect of BSC decreases since the higher load does not permit extensive background computations. Figure 5.7b shows the CPU overhead versus τ. Even with the application of BSC, HybMig has a lower cost than MS and PT.

5.3 Evaluation of the Generalized Methods

In the experiments for the generalized methods we use a migration task similar to the default task in Section 5.1 (w=3, τ=1, n=6, P_{old} to P_{new}^{r}), except that (i) we add a sliding-window *median* operator on top of the topmost join in each plan and (ii) every plan is encapsulated into a single black box UDA. Since the median operation is not distributable, we cannot combine the results from the two plans. Meanwhile, the operator states are not accessible because of the encapsulation of the UDAs, thus state re-computation is impossible. Therefore, MS, PT and HybMig are inapplicable. Instead, Figures 5.8 compare their generalized versions GMS, GPT and GHM in terms of the output rate and CPU overhead.



a. Output Rate/Time b. CPU Overhead/Time

Figure 5.8 Evaluation of GMS, GPT and GHM (w=3,τ=1,n=6)

The rates of GHM and GPT are similar, whereas GMS suspends the output stream during the migration. Comparing Figures 5.8a and 5.1, note that (as discussed in Section 4.2) GPT does not suffer from the bursty behavior of PT. Regarding the CPU overhead, GHM and GMS are similar

since they both re-compute intermediate states in the new plan. The sudden drop in the CPU costs signals the end of the migration, which in GHM and GMS is much earlier that GPT.

6. Conclusion and Future Work

This paper investigates dynamic plan migration in data stream management systems. The existing techniques MS and PT: (i) incur high memory/CPU overhead, (ii) lead to bursty output rates and (iii) in the case of PT, have limited applicability to a specific temporal ordering requirement. Motivated by these problems, we propose HybMig, a novel technique, which outperforms the previous approaches on all aspects. In addition to join re-ordering, we study dynamic migration for plans involving arbitrary operators. We treat these plans as black boxes and propose three techniques, GMS, GPT and GHM, motivated by MS, PT and HybMig, respectively.

This work opens several directions for future work. So far we have focused on the case that exact results are required during migration. The first interesting problem is how to obtain a good approximation of the output during migration, in the presence of insufficient system resources. Furthermore, we intend to investigate the problem of plan migration in relational databases when a query needs to access multiple portions of the database that exhibit different properties, as suggested in [BBD05].

References

[ABW] Arasu, A., Babu, S., Widom, J. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal,* to appear.

[ACC+03] Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S. B. Aurora: a New Model and Architecture for Data Stream Management. *VLDB J.* 12(2): 120-139, 2003.

[ACG+04] Arasu, A., Cherniack, M., Galvez, E. F., Maier, D., Maskey, A., Ryvkina, E., Stonebraker, M., Tibbetts, R. Linear Road: A Stream Data Management Benchmark. *VLDB*, 2004.

[AH00] Avnur, R., Hellerstein, J. M. Eddies: Continuously Adaptive Query Processing. *SIGMOD*, 2000.

[BBD05] Babu, S., Bizarro, P., DeWitt, D. Proactive Re-Optimization. *SIGMOD*, 2005.

[BBD+04] Babcock, B., Babu, S., Datar, M., Motwani, R., Thomas, D. Operator Scheduling in Data Stream Systems. *VLDB Journal,* 13(4): 333-353, 2004.

[BMWM05] Babu, S., Munagala, K., Widom, J., Motwani, R. Adaptive Caching for Continuous Queries. *ICDE*, 2005.

[DH04] Deshpande, A., Hellerstein, J. M. Lifting the Burden of History from Adaptive Query Processing. *VLDB*, 2004.

[GO03] Golab, L., Öszu, M. T. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. *VLDB*, 2003.

[HH99] Hass, P. J., Hellerstein, J. M. Ripple Joins for Online Aggregation. *SIGMOD*, 1999.

[KFHJ04] Krishnamurthy, S., Franklin, M. J., Hellerstein, J. M., Jacobson, G. The Case for Precision Sharing. *VLDB*, 2004.

[KNV03] Kang, J., Naughton, J. F., Viglas, S. Evaluating Window Joins over Unbounded Streams. *ICDE*, 2003.

[KS04] Krämer, J., Seeger, S. PIPES – a Public Infrastructure for Processing and Exploring Streams. *SIGMOD*, 2004.

[LWZ04] Law, Y., Wang, H., Zaniolo, C. Query Languages and Data Models for Database Sequences and Data Streams. *VLDB*, 2004.

[MSHR02] Madden, S., Shah, M., Hellerstein, J. M., Raman, V. Continuously Adaptive Continuous Queries over Streams. *SIGMOD*, 2002.

[UFA98] Urhan, T., Franklin, M. J., Amsaleg, L. Cost Based Query Scrambling for Initial Delays. *SIGMOD*, 1998.

[VNB03] Viglas, S., Naughton, J. F., Burger, J. Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. *VLDB*, 2003.

[ZRH04] Zhu, Y., Rundensteiner, E. A., Heineman, G. T. Dynamic Plan Migration for Continuous Queries Over Data Streams. *SIGMOD*, 2004.