# Adapting Partitioned Continuous Query Processing in Distributed Systems

Yali Zhu and Elke A. Rundensteiner
Worcester Polytechnic Institute, Worcester, Massachusetts, USA
{yaliz, rundenst}@cs.wpi.edu

## Abstract

*Partitioned query processing is an effective method to process continuous queries with large stateful operators in a distributed systems. This method typically partitions input data into non-overlapping portions, with each query plan instance installed on a separate machine processing only one portion of the data. Dynamic redistribution of load among machines is then employed to handle fluctuating stream characteristics. However, existing load redistribution solutions have made the implicit assumption that no local query optimization is conducted at runtime on any of the participating machines, i.e., all local query plan instances are static and thus remain identical. This is restrictive for dynamic stream systems, where data partitions may experience significant fluctuations in selectivities or arrival rates over time - thus warranting local plan reoptimization. This raises the new problem that the heterogeneity of plan shapes among different machines must be tackled when doing load redistribution. To address this, we propose two new load balancing strategies along with corresponding protocols, that can balance the workload across a set of machines while seamlessly handling the complexity caused by local plan changes. The PTLB strategy is plan-agnostic, requiring no detailed knowledge of the underlying query plan. The MSLB strategy is plan-aware, that is, it rebalances the load by comparing the plan shape differences on the participating machines. All proposed techniques have been implemented in the DCAPE continuous query system. Our experiments demonstrate that the application of both query optimization and load balancing results in superior performance compared to applying either of the adaptation techniques alone — as has been the state-of-the-art in the current literature. Our evaluation compares the relative applicability and efficiency of the two proposed techniques PTLB and MSLB.*

## 1. Introduction

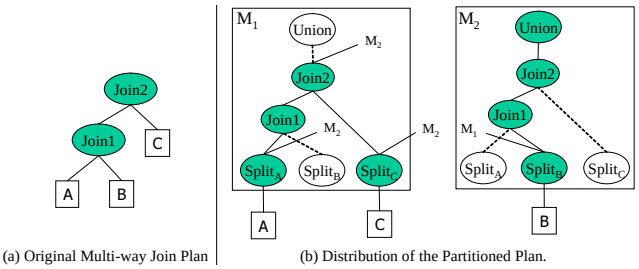
Continuous queries have become popular in recent years due to demands of numerous applications, including online auctions, financial analysis, sensor monitoring systems, etc [2, 4, 5, 15, 17]. A continuous query engine takes in real-time streaming data and sends out results in a continuous fashion. High stream input rates and cost-intensive query operations may cause a continuous query system to run out of resources. Distributed continuous query processing over a shared nothing architecture, i.e., a cluster of machines, is a prevalent method for solving this scalability problem [1, 7, 8, 13, 16].

Distributing the query workload across multiple machines can greatly improve the system performance due to the availability of aggregated resources, including both CPU and memory. However, uneven workload among machines may occur over time due to (1) the lack of initial cost information at the time when first distributing the queries, and (2) the potentially fluctuating nature of the incoming data streams even if the statistics could be measured at runtime. This imbalance of workloads on different machines may impair the benefits of distributed processing. Thus, *dynamic load balancing*, which deals with the problem of re-distributing workload across machines in the cluster, has emerged as a crucial technology for distributed continuous query systems [1, 8, 13, 16].

### 1.1 Partitioned Query Processing

Partitioned parallelism [10] is a common method for processing query operators with large states in a distributed system. Instances of each query operator will be installed on multiple processors, with the input data being partitioned among these operator instances. The outputs from all operator instances are unioned to form the final output stream. Such partitioned parallelism, which have been routinely applied for traditional query processing [9,10], has been shown to be also affective for continuous queries [16].

For example, the continuous query plan with two joins in Figure 1(a) can be assigned to two machines as in Figure 1(b). Each machine runs instances of both join operators. To partition the data, we add three *split operators* and a *union operator* to the query plan. The *split operators* operate as routers: They apply partition mapping functions, such as value-based mapping, to divide the streams of input tuples into partitions and direct these partitions to the corresponding machines. The darker shading indicates that the operator is active on that machine.




(a) Original Multi-way Join Plan (b) Distribution of the Partitioned Plan.

**Figure 1. Distribution of Partitioned Plan**

By using the partitioned parallelism, we now have the choice of moving only some partitions of an operator state to another machine at runtime – without having to move complete operators each time [1]. This enables fine-grained runtime adaptation.

## 1.2 Limitations of Existing Strategies

The load balancing strategies currently proposed in the literature for partitioned continuous queries make the implicit assumption that the partitioned query plans on different machines remain identical [13, 16]. They have not considered the situation that the local query optimizer restructures the shape of the query plan residing on its machine. Given this strong restriction, no existing work on partitioned continuous query processing thus far has considered integrating the load balancing with query optimization. Consequently, the effects of query optimization and its impact on load rebalancing strategies remain an open issue to date.

This clearly is a major limitation, as runtime query optimization has been shown to be critical for streaming systems [3, 5, 15, 20]. That is, some data partitions may experience characteristics rather distinct from those of other partitions over time [14]. Let us consider a partition containing IBM stock quotes. This partition may experience a high selectivity, if some major shifts in the market raise interest in the performance of those stocks compared to others. Hence, the local optimizer then would determine the locally optimal plan based on observed data statistics of its data portion. This raises the new problem that the heterogeneity of plan shapes among different machines must be tackled when doing load redistribution.

Further load balancing strategies just move workload from one machine to another, while the total resource consumption in the system as a whole is not decreased. On the other hand, plan optimization may be able to decrease the resource consumption on each machine, therefore decreasing the overall resource consumption in the distributed system. For example, a plan optimization may dynamically switch two join operators in a plan in the face of changing statistics. It is well known that such optimization may drastically reduce the intermediate results, leading to less CPU and memory costs on this machine as well as less overall resources required to process this query in the distributed system.

## 1.3 New Research Problems

Local query optimization however complicates load rebalancing strategies. Traditionally, load rebalancing algorithms assume that the shape of the query plan stays the same throughout the query execution. Therefore balancing loads among machines can be simply achieved by moving some load (partitions) from overloaded machines to under-loaded machines. However, this is no longer valid if local query optimization has been applied. Since each machine can apply its own local optimization separately from other machines, at any given time, the shapes of the query plan on different machines can be distinct from one another.

To illustrate the problem, we use the query example from Figure 1. As depicted in Figure 2, each join operator instance has two states, with each state containing several partitions (without loss of generality, here we assume value-based partitioning is applied in the split operator). Each partitioned state contains a set of partitions with different partition IDs. From Figure 2, we can see that M1 processed partitions with IDs 1, 2, 3, and M2 processes partitions with IDs 4, 5.

After machine M2 applies the local plan optimization, the two joins on M2 are switched. Now the query shapes on the two machines are distinct from each other. A new partitioned state $P_{BC}$ has been created on M2 as the result of this plan optimization. If at this time the load balancing algorithm decided to move all partitions with ID 2 from M1 to M2, the partition 2 belonging to the

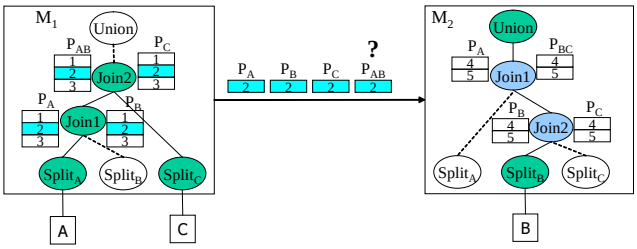



**Figure 2. Problem with Simple Partition Moving During Load Rebalancing**

partitioned state $P_{AB}$ cannot be put into any join state on M2, because it does not have any matching state on that machine. Simply discarding this unmatched partition would cause loss of data. This problem of integrating load rebalancing with query optimization remains an unaddressed problem to date. Clearly, strategies need to be devised that can support the heterogeneity of plan shapes on difference machines during load rebalancing. This is now the focus of our work.

## 1.4 Our Research Outline

In this paper, we solve the new problems of integrating query optimization with the partition-level runtime load balancing for continuous queries. Our first research goal is to study the effects of adding plan optimization to distributed continuous query processing. Our goal here is not to propose new methods for plan optimization (rewriting of query plans) nor for load balancing (deciding which partitions to move when and to which other machines) — rather we adopt existing techniques for these well-studied tasks from the literature [1, 16]. Instead, we focus on the evaluation of the effectiveness of plan optimization versus load balancing in isolation in terms of the relative performance gains achievable as well as scopes of applicability. More importantly, we study performance gains achievable by their integrated forms. This evaluation is conducted through experimental studies in an actual stream query system running on a compute cluster. This is the first result on this topic in the literature.

As our second research goal, we propose to design, implement and evaluate advanced strategies that can conduct load rebalancing while taking the heterogeneity of query plan shapes on difference machines into account. Our focus here is on the protocols, their synchronization and correctness. We design two new load balancing strategies, namely PTLB and MSLB, and their corresponding protocols that can balance workload while seamlessly handling the complexity caused by local plan heterogeneity. The PTLB strategy is a general load balancing strategy that requires no detailed knowledge of the underlying query plans, such as types of operators and shapes of query plans. We then propose the more plan-aware MSLB strategy, which rebalances the workload by comparing the detailed shapes of the query plans among different machines. For simplicity of exposition, our techniques are explained for query plans with their stateful operators being joins, such as typical symmetric window-based joins [4, 12, 17, 19], given their importance and prevelance in stream queries. Plans with other stateful operators, such as duplicate elimination and groupby, which could be handled in a similar vein, are left as future work.

We have implemented the proposed strategies in a continuous query system called D-CAPE [13]. One key contribution of our work is then this experimental study assessing the proposed

methods in a real stream system. That is, we have experimentally evaluated the effects of query optimization, load rebalancing, and their integration for partitioned continuous query processing on an actual cluster of PCs. Our experiments show that the combination of query optimization and load balancing has superior performance compared to applying either of the two adaptation techniques alone (as done in the current literature). A comparative study assessing the relative applicability and efficiency of the two proposed techniques PTLB and MSLB is also conducted. The MSLB is shown to be more efficient than the PTLB in many situations, while the PTLB is shown to win under certain conditions.

For the remainder of this paper, we discuss related work in Section 2. The two proposed load balancing strategies and their protocols in a distributed system are described in Sections 3 and 4, respectively. Section 5 shows our experimental evaluations. We conclude our work in Section 6.

## 2. Related Work

Existing distributed stream systems [1,7,8] use one operator as the basic unit for load balancing. This assumes that each operator is small enough to fit on one machine. Partitioned parallelism is a general query plan distribution strategy [9,10]. Flux [16] is the first to apply partition-level load redistribution to continuous queries. Flux focused on single-input operators, namely, group-by. They assume that all query instances have the same query shapes. Our research instead proposes load balancing strategies to deal with the heterogeneity of plan shapes with stateful join operators among different machines.

Continuous query optimization has been studied in recent years [2,6,11,18]. [18] proposes a rate-based algorithm to optimize continuous multiple joins to achieve high output rate. [3] proposes heuristics-based join ordering algorithms for mjoin that consider dependent join selectivities. [15] introduces the Eddy approach of adaptively executing a query by routing tuples among operators. Eddy's always-adapting solution makes it suitable for a highly dynamic environment. These solutions all focus on optimizing continuous queries based on statistics collected at runtime.

Our own earlier work on dynamic plan migration [20] is the first to deal with the problem of safely transferring the currently running plan to the new plan generated by the optimizer. This earlier work in part has inspired our solution now proposed for the distributed scenario. However, the two problems are significantly different. The former focuses on migrating the states of a currently running query plans in a central environment, that is, one machine. Here instead, we are addressing the distributed scenario where the plans to "balance load" between are residing on distinct machines. One, this now requires carefully synchronized coordinations both within and also among the participating machines to assure correctness. Two, we now focus on relocating only individual partitions of states while leaving others behind – that is, we are relocating partial state between query plans. This is in contrast to the centralized migration, where the complete operators (along with their full state) are simply migrated into other operators or other positions within the query plan.

## 3. Plan-Agnostic Load Balancing Strategy

### 3.1 Basic Idea : Duplicated Processing

Here, we assume that the load balancer has selected the partitions to be moved from one query plan to a second query plan by any standard techniques, such as in [16]. We call the machine where the first partition resides before the relocation the *sender machine* and the second machine the *receiver machine*. The basic idea for this plan-agnostic strategy is now for the corresponding split operator to duplicate any newly arriving data belonging to these to-be-moved partitions, so that they can be send to both the sender and the receiver machines concurrently. Both machines then process this same portion of data in parallel. This effort must continue until all tuples of the to-be-moved partitions that were residing in the sender query plan at the time the relocation started have been purged out of their respective operator states on the sender machine due to the arrival of the newer tuples. The purging itself proceeds as usual according to the operators's window semantics [4, 12, 17]. Here we say a tuple is *old* if it exists in any partition before the load balance starts. A tuple is *new* if it arrives after load balance has started. Clearly, the receiver machine would not be containing any tuples of the to-be-moved partitions at the relocation-start-time. Thus all its tuples will be *n*ew.

When the to-be-moved partitions on the sender machine contain only *new* tuples, it can be shown that it is safe to discard the old partitions from sender. This is because all old partitions have finished their duty in terms of contributing to the generation of output results from the sender machine. Since we have been feeding the same data belonging to these to-be-moved partitions to the receiver machine in parallel when the load balancing first starts, all the new tuples belonging to these partitions now in the sender machine exist in the receiver machine as well. So if the old partitions are discarded from the sender machine at this time, no useful data will be lost.

We must ensure that no duplicate tuples are being generated. If we use the parallel track strategy described above, although the sender machine will generate all output tuples from the to-be-moved partitions that consist of at least one *old* sub-tuple, it may also generate the all-new sub-tuple combination, duplicate to the output results from the receiver machine.

To solve this duplication problem, the root join operator of the sender machine can prevent a *new* tuple from joining with another *new* tuple. Hence if the join predicate is evaluated on two tuples that are both *new*, we simply skip the join step in the regular purge-join-insert symmetric join algorithm. The purge and insert steps are however still undertaken as usual.

For this strategy, all old tuples (tuples with at least one old sub-tuple) need to be purged from the to-be-moved partitions. Suppose that *h* ($h >= 1$) is the height of the query tree on the sender machine. We analyze the time spent on the parallel track strategy, denoted henceforth as $T_{PT}$, in two cases:

1) h = 1. In this case the query tree has only one level of join operators. For a join operator on the sender to purge all old tuples in the to-be-moved partitions from one of its two states, the join operator must process new tuples from another input that arrive in the next W time units. Therefore relocation time $T_{PT} = W$.

2) $h > 1$. This means that on the sender there is at least one join operator which is above another join operator. When the load balance begins, W time window's new tuples from the input queues are needed to purge old tuples inside the to-be-moved partitions of leaf operators on the sender machine. However, as these new tuples are used to purge old tuples, they may also join with some of the old tuples and the results are being inserted into the state of the join operators above the leaf operators. Because the

joined tuples contain an *old* sub-tuple, they are treated as *old* tuples and need to be purged as well. In order to do so, the sender machine needs to process another W time window's new tuples to completely purge these *old* tuples from the old partitions. So in this case, relocation time $T_{PT} = 2W$.

In summary, the lower bound of time spent on finishing the parallel track process is $2W$ for a query with more than one join operator, given that $W$ is the window size of the query. The lower bound is W if the query contains only one join operator.

## 3.2 Distributed PTLB Protocols

In this section, we describe the distributed Parallel Track Load Balancing (PTLB) protocol we have designed to apply the basic idea described above to solve the problem of load balancing among machines with heterogeneous plans in a distributed environment. The distributed protocol is critical because we need careful coordination among sender machine (the one sends the partitions), receiver machine (the one receives the partitions), and the distribution manager (the one that makes the load rebalance decision) in order to guarantee correct load rebalancing. This is to ensure that no on-the-fly data is missing, duplicated or corrupted.

We have designed a 5-step PTLB communication protocol to achieve the PTLB once our system has made the decision to apply load rebalancing. This decision making model is called distribution manager (DM). Each step contains a message passing between DM and one of the query processors. The query example in Figure 2 is used here to illustrate the execution of the protocol.

Steps 1 and 2 of the PTLB protocol involve communications between the distribution manager and the sender machine to calculate the partitions that need to be moved from the sender to the receive. These steps are depicted in Figure 3. When the DM makes the decision to invoke load balancing, it has already calculated three variables used in the load balancing process: the sender machine which has the highest memory consumption (denoted by $M_{max}$), the receiver machine which has the least memory consumption (denoted by $M_{least}$), and the amount of partitions in terms of memory the sender should send to the receiver. Therefore, in the first step of load balancing, the DM sends a request *computePartitionsToMove* to the sender (assumed to be M1 in Figure 3), with the amount of partitions that need to be moved. Upon receiving such a request, the sender machine selects the partitions whose total memory consumed is close to the amount of memory that is to be moved. In step 2, the sender then sends the IDs of the selected partitions, denoted as *partitionsToMove* in Figure 3, back to the DM.

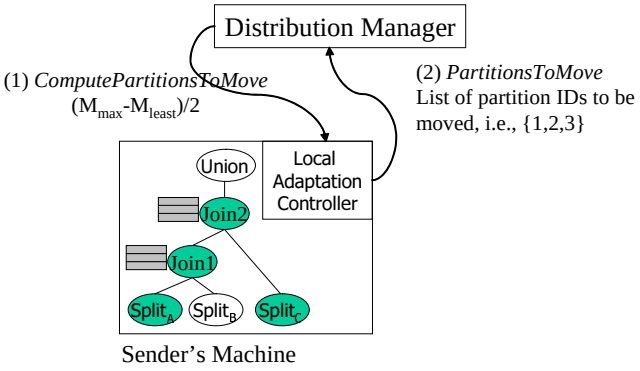



**Figure 3. PTLB: Compute Partitions to Move.**

Each partition ID represents all the partitions on the sender machine with that same partition ID. In fact, each partitioned state can have a partition with the selected ID. Therefore each partition ID indicates one partition from each state. Our mechanism is to choose all the partitions in all the states with the same partition ID as a whole unit to move. This avoids joins across multiple processors. For example, as shown in Figure 2, we denote the partition with ID 2 in partitioned state $P_A$ as partition $A_2$. If we only move partition $A_2$ from $M_1$ to $M_2$, then after the load balancing, the newly coming tuples to partition $A_2$, which is now located on $M_2$ would have to probe and join partition $B_2$, which is still located on $M_1$. Therefore in our load balancing process, the unit to move between two machines is not a single partition, but is a *partition group* that contains all the partitions with the same ID on the sender machine.

Steps 3 and 4 exploit parallel processing principles. In Step 3, the DM sends a *DuplicatePartitions* to the sender machine as well as all the machines with active split operators. Upon receiving the message, an active split operator will add entries to the existing partition mapping table, which map each of the to-be-moved partitions to the receiver machine. This allows the split operator to hereforth forward tuples that belong to these selected partitions to both the sender and the receiver machines. Whenever a tuple is forwarded to a sender machine, the split operator sets a flag on the tuple as *new*. This indicates that this tuple is also being sent to the receiver machine. The flag on all other tuples, including the tuples being sent to the receiver machine in parallel, are by default set to be *old*.

Upon receiving the *DuplicationPartitions*, the join operators on the sender machine process as follows in order to avoid producing duplicate results from the sender and the receiver.

- For all join operators except the root join operator on the sender machine, a *new* tuple is being treated the same as an *old* tuple. When a joined tuple is outputted from a join operator, the joined tuple is set to be new *only when* all its sub-tuples are *new* as well. Otherwise, the tuple is still set to be *old*.
- At the root join operator, when two tuples are to be joined, if both tuples are marked as new, they are *not* joined together. Instead, the tuples are just used to purge partitions and are then inserted to the corresponding partitions. This is because the new-to-new joins are to be done on the receiver machine.
- The sender machine sends an *AllOldPurged* message back to the DM when all old tuples have been purged from all the partitions that belong to the set of to-be-moved partitions.

As the last step of the PTLB protocol, Step 5 the DM sends a *DeletePartitions* message to the sender machine and all machines with active split operators. Each active split operator will then remove the entries that map the to-be-moved partition IDs to the sender machine. This allows the split operator to forward new tuples belonging to these partitions to the receiver machine only. The split operator then puts an *EndOfPartitions* flag to all the output queues connecting to the sender machine. When a join operator has received the *EndOfPartitions* flag from all its input queues, it can delete the to-be-moved partitions from its states. The join operator also forwards an *EndOfPartitions* flag to its parent. When the root join operator has received all the *EndOfPartitions* flags from its input queues, the PTLB process is considered to be over.

Algorithms 1 and 2 sketch the high level interactions between the distribution manager and the processors on each machine during the runtime PT load balancing process. Algorithm 1 describes the basic operations of the distribution manager. Similarly, Algorithm 2 describes the steps performed on a participating processor during the PTLB process. Here, the *send* and *wait* are primitive operators designed to send or wait for messages across machines.

**Algorithm 1** PT-State-Rebalance:Manager(sender, receiver, amt)

*/*It controls load balance process by sending control messages to participating machines and waiting for corresponding responses.*/*

1: **send** *ComputePartitionsToMove*(amt) msg to *sender*;
2: **wait** until get *PartitionsToMove* msg;
3: **send** *DuplicatePartitions* to sender & machines with active split operator(s);
4: **wait** until get *AllOldPurged* msg from the sender machine;
5: **send** *DeletePartitions* msg to sender & machines with active split operator(s);

**Algorithm 2** PT-State-Rebalance:Processor()

*/* To receive messages, perform corresponding actions, and return message(s) to the distribution manager.*/*

1: **while** (keepGoing) **do**
2: wait for control message of PTLB protocol;
3: **switch**(protocol)
4: *ComputePartitionsToMove*: /**compute partitions to be moved**/
5: compute partitions to move;
6: send *PartitionsToMove* msg to Distribution Manager;
7: *DuplicatePartitions*: /**send new tuples to both machines**/
8: split operators start sending new tuples to both machines;
9: root join operator waits for old tuples to be purged;
10: root join operator sends *AllOldPurged* msg to DM;
11: *DeletePartitions*: /**Delete to-be-moved partitions**/
12: split operators stop sending tuples in the given partitions to the sender machine;
13: join operators on sender remove given partitions;
14: **end while**

In summary, the PTLB is a general strategy that does not need to care about the detailed properties about the plan itself, such as the types of the operators and the shapes of the plans. This simplifies the process of load balancing, especially when the plan shapes can be different between the sender and the receiver. It also has the advantage of not having to stop the query execution in the to-be-moved partitions at any point of time. It thus does not have to deal with on-the-fly tuples. However, this simplicity comes with a price of both CPU and memory overhead, which will prevail as long as the balancing process is ongoing. The whole process can take as long as 2W timeframe to finish. This is undesirable for continuous queries with large windows, which are in fact the ones that most likely need to be executed in a distributed system in the first place. It also incurs the extra overhead of having to store the same set of tuples for these to-be-moved partitions on both the sender machine and the receiver machine. To overcome these shortcomings, we design the second runtime load balancing protocol, the moving state protocol, which is described in the next section.

# 4. Load Balancing With Plan-Aware Strategy

## 4.1 Basic Idea: Moving Partitions

The basic idea of the *moving state strategy* is to safely move to-be-moved partitions on the sender machine directly into the states on the receiver machine without losing any useful data. In this section, we describe the necessary steps of the moving state strategy, including *state matching*, *state moving* and *state recomputing*.

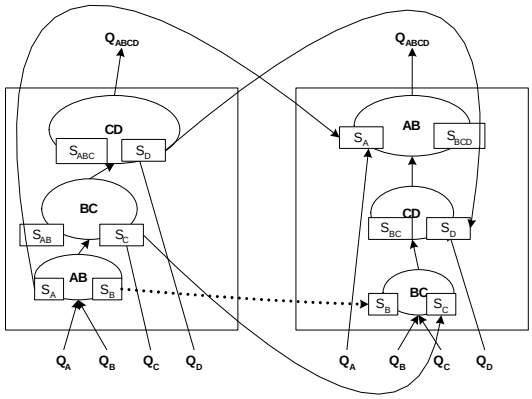



**Figure 4. Example of Moving States From Old Plan (left) to New Plan (right).**

*State matching* determines the pairs of states, one in the sender and one in the receiver machine, between which tuples can be safely directly moved. Two states can move tuples in between them if and only if they contain tuples with the same schema. In our query plans, a tuple's schema is defined by all its column IDs. We define a state's ID as the same to its tuple's schema, and all tuples in one state have the same schema. If two states have the same state ID, we say that those two states are *matching states*. In Figure 4, states $(S_A, S_B, S_C, S_D)$ exist in both boxes and are matching states. States $(S_{BC}, S_{BCD})$ appear in the new box only (the box on the right), and states $(S_{AB}, S_{ABC})$ appear in the old box only (the box on the left). These are thus unmatched states.

After the *state matching*, we can then take the *state moving* step to move tuples between all pairs of matching states. The details of how these move is achieved is describe in Section 4.2.

Assume that now the chosen partitions have indeed been moved safely, then the partitions on the receiver machine start to be executed with the unmatched states being empty. In the join operator $B \bowtie C$ in Figure 5, only *new* B tuples can be joined with *old* or *new* C tuples in $S_C$.[1] Also, only *new* C tuples can be joined with *old* or *new* B tuples in $S_B$. Hence only combined BC tuple with its two sub-tuples' old/new status as (new, old), (old, new) or (new, new) can be generated. The combination (old, old) would never be generated.

Therefore, before restarting the execution of the query plan on the receiver machine, we need to gain back those *all-old* combinations via *state recomputing*. This can be done by recursively recomputing the unmatched states from bottom to top.

## 4.2 Distributed MSLB Protocols

[1]In Figure 5 $S_C$ only contains *old* tuples. However, each *new* C tuple inserted into $S_C$ may have been joined with B tuples, and after a while the state $S_C$ may contain both *old* and *new* tuples.

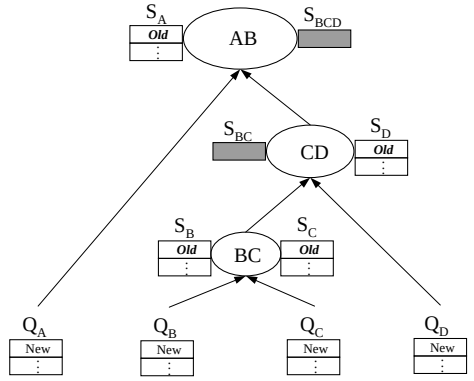



**Figure 5. Empty Unmatched Partitions**

The key of the distributed moving state load balance (MSLB) strategy is that now we need to carefully synchronize the participating machines, including the distribution manager (DM), the sender and the receiver in order to achieve the load balancing without resulting in any loss, duplicate or incorrect query results. Hence we have developed an 8-step communication protocol to achieve the MS load rebalancing. Each step consists of one or more message exchanges between distribution manager (DM) and one of the query processors.

Steps 1 and 2 in the MSLB protocol correspond to communication between the distribution manager and the sender machine. Basically the DM requests the sender machine to calculate the IDs of the partition groups that needs to be moved to the receiver machine. Steps 3 and 4 denote exchanges between the DM and processors to deactivate to-be-moved partitions before they are really moved between machines. These steps are necessary because the processing of the to-be-moved partitions needs to be stopped before they can be safely moved to another machine. In Step 3, the DM sends a *deactivatePartitions* message, along with the to-be-moved partition IDs calculated in Steps 1 and 2, to sender machine and all machines with active split operators. In this example, both machines have active split operators and thus both will receive such message from the DM.

On machines with active split operators, after receiving the *deactivatePartitions* message, an active split operator will take the following three actions in that order: 1) First, it removes the to-be-moved partition IDs from its partition mapping table, so that newly arriving tuples belonging to these partitions will no longer be forwarded to the sender machine. 2) Because after the first action, any new tuple belonging to these partitions won't be forwarded to any machine, the split operator needs to create buffers to temporarily hold these new tuples. 3) Lastly, the split operator inserts an *EndOfPartitionInputFlag* into each output queue that connects to the sender machine. After all active split operators on a machine has taken these three actions, the machine sends a *Deactivated* message back to the DM as Step 4.

On the sender machine, after receiving the *deactivatePartitions* message, the sender machine sets each join operator on that machine to count the number of *EndOfPartitionInputFlag* this operator has received. When a join operator has received the same number of the *EndOfPartitionInputFlags* as its input queues, it forwards this flag to its parent operator. When the root join operator has received such flags from all its input queues, this means that all operators on the sender machine have finished processing all tuples that belong to the to-be-moved partitions. This means that no more tuples that belong to these partitions will come. The sender machine then sends a *Deactivated* message back to the DM as Step 4.

Steps 3 and 4 not only deactivate to-be-moved partitions, they also allow the operators on the sender machine to finish processing all on-the-fly tuples in these input queues that belong to these to-be-moved partitions. This clean-up stage is necessary, because if the partitions were to be moved right away without the clean-up, the on-the-fly tuples won't be able to join with these already moved partitions on the sender machine. We thus may miss some of the query results due to load balancing process.

The actual partition movement is achieved in Steps 5, 6 and 7, as depicted in Figure 6. The DM first waits for the *Deactivated* message from all the involved machines. After that, as Step 5, the DM sends a *SendPartitions* message to the sender machine. Upon receiving such a message, as Step 6 the sender machine packs all the partition groups with the selected IDs and sends them to the receiver machine using a *ReceivePartitions* message.

After receiving the *ReceivePartitions* message from the sender, along with all the partition groups, the receiver machine then conducts the following process:

- First, the receiver machine extracts all the partitions from the received partition groups.
- It then applies the *state partition matching* step, as described in Section 4.1, to match each single partition's schema with the existing states on the sender machine. If a match is found, the single partition is then inserted to the state that has the same schema. At this point, this single partition should have a partition ID different from any existing partitions in that state. If a single partition cannot be matched with any state, this single partition is then discarded by the receiver. Using the example in Figure 2, the moved partition group contains four single partitions $P_{A2}$, $P_{B2}$, $P_{C2}$ and $P_{AB2}$. The first three single partitions will be inserted into states $P_A$, $P_B$ and $P_C$ on machine M2 respectively, while the single partition $P_{AB2}$ is discarded since it does not match any states on machine M2.
- After the partition matching step, all the states that do not have a matching partition inserted will require a partition recomputation to regain the partitions that have the moved partition IDs. This can be done by recursively recomputing these single partitions in a bottom up fashion. Again using the example in Figure 2, the state $P_{BC}$ does not have any matching partition. So the partition $P_{BC2}$ that should have been moved from the sender machine would now be recomputed by joining the moved single partitions $P_{B2}$ and $P_{C2}$. Note that we only need to join partitions with the same ID as the to-be-generated partition.

After the partitions are being moved and recomputed, as Step 7 the receiver machine sends a *Received* message back to the DM. This partition moving procedure is general, that is, it would also work when local plan optimization had not been invoked in the system, meaning the shape of query plans stay unchanged. In that case, all partitions transfered between two machines will be matching partitions on the receiver machine. Therefore no partition recomputation is necessary.

As the last step (Step 8) of the MSLB protocol, the DM sends a *ReactivatePartitions* message to all machines with active split

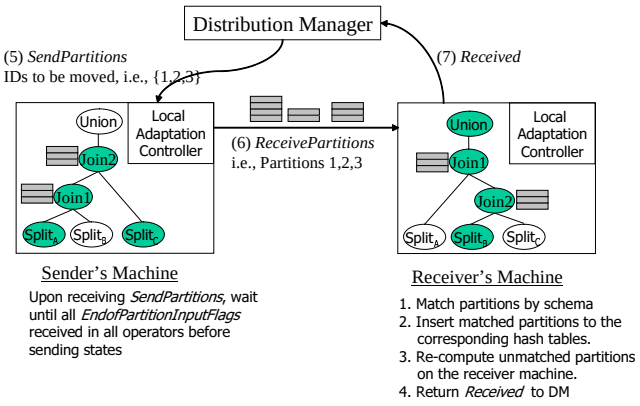



**Figure 6. MSLB: Move & Recomp. Partitions.**

operators. Upon receiving such a message, the split operator will start forwarding new tuples belonging to the moved-partitions to the receiver machine. The tuples in the temporary buffers will also be forwarded to the receiver machine all at once, after which the temporary buffers are removed from the machine. The process of MS load balancing is then finished.

Algorithms 3 and 4 sketch the high level interactions between the distribution manager and the query processors on each machine during the distributed MSLB process. Algorithm 3 describes the basic operations of the distribution manager.

**Algorithm 3** MS-State-Rebalance:Manager(sender, receiver, amt)

*/*It controls load balance process by sending moving protocols to local machines and waiting for corresponding responses.*/*

1: **send** *ComputePartitionsToMove*(amt) msg to *sender*;
2: **wait** until get *PartitionsToMove* msg;
3: **send** *DeactivatePartitions* to sender & machines with split operator(s);
4: **wait** until get all *Deactivated* msgs;
5: **send** *SendPartitions* msg to *sender*;
6: **wait** until get *Received* msg;
7: **send** *ReactivatePartitions* msg to machines with split operator(s);

Similarly, Algorithm 4 describes the steps performed on a participating processor during the state relocation process. The algorithm waits for control messages in the MSLB protocol. It performs corresponding actions based on the messages it has received.

**Discussion** In summary, the MS load rebalance strategy selects partitions to move and then directly moves these partitions from the sender machine to the receiver. Different from the PTLB strategy, it needs to have the knowledge of the detailed information about the query plan. However, it directly moves partitions from the sender and the receiver without delay, therefore it can release the burden of the sender right away, which is supposed to be the over-loaded machine of the two. It also does not incur the extra overhead of having to send new tuples to both the sender and the receiver, as in the PTLB strategy.

# 5. Experimental Evaluation

**Algorithm 4** MS-State-Rebalance:Processor()

*/* To receive messages, perform corresponding actions, and return message(s) to the distribution manager.*/*

1: **while** (keepGoing) **do**
2: wait for control messages of the MSLB protocol;
3: **switch**(received protocol)
4: *ComputePartitionsToMove*:
5: compute partitions to move;
6: send *PartitionsToMove* msg to Distribution Manager;
7: *DeactivatePartitions*:
8: deactivate partition inputs;
9: send *Deactivated* msg to Distribution Manager;
10: *SendPartitions*: /**send out partitions**/
11: wait on-the-fly tuples being processed;
12: send partitions via *ReceivePartitions* msg to receiver;
13: *ReceivePartitions*: /**receive, insert and recompute partitions**/
14: extract single partitions from partition groups received;
15: insert matching single partitions to corresponding states;
16: recompute single partitions in unmatched states;
17: send *Received* msg to Distribution Manager;
18: *ReactivatePartitions*: /**resume & redirect inputs for moved partitions* */
19: reactivate moved partitions;
20: redirect moved partitions' input;
21: **end while**

Our experimental evaluation focuses on two studies. First, we show the benefits of adding local plan optimization in the distributed continuous query processing along with the load balancing adaptation. Second, we compare the performances of the two proposed load balancing strategies.

## 5.1 Experimental Setup

We have implemented the dynamic query optimization and the two proposed load balancing strategies in a distributed continuous query processing system called D-CAPE [13]. The D-CAPE system consists of a distribution manager, a stream generator and arbitrary number of query engines. Each machine runs a query engine. The distribution manager collects statistics from each query engine and initiates global load balancing among machines. The stream generator generates tuples with arrival patterns modeled as the widely adopted Poisson process. System parameters such as stream input rates and global time windows are varied to reflect the changes in workload and data characteristics.

All experiments are run on a 10-node clusters. Each node has dual 2.4Hz Xeon CPUs with 2G main memory. We use the query in Figure 1 as the experiment query. The join operators have instances installed on all machines. Split and union operators are added to the plan accordingly. We devote one machine each to run the distribution manager, the stream generator and the end application that receives query results. The remaining nodes can be utilized to execute the query plan.

## 5.2 Benefits of Local Query Optimization

Our first goal for experimental evaluation is to show that local query optimization does boost the performance of partitioned CQ processing. To show the added benefits of local optimization, we compare the query performances in the following four settings:

- *No-Adapt*: In this setting, the same query plan is executed from the beginning to the end. Neither local optimization nor load balancing is applied to the query execution.
- *LM-only*: Only Local Machine query optimization (LM) is applied as the form of adaptation during query execution.
- *PTLB-only* or *MSLB only*: Only PTLB or MSLB is applied as the adaptation method during query execution.
- *LM-PTLB* or *LM-MSLB*: Both query optimization and load balancing are applied during query execution.

In this set of experiments, each of the three stream inputs (streams A, B and C) is partitioned into 100 partitions. The initial input rates are all set to be 100 tuples/sec. The initial plan joins streams A with B then C. At the 30th second, the input rates of B and C are both changed to 5 tuples/sec. This motivates the switch of the two join operators to get a more efficient plan by dynamic plan optimization. The partition functions in the split operators are initially set so that one machine in the system gets 50% of the total workload, while the rest of the workload is divided evenly among all machines. This indicates that load balancing is necessary to obtain a good query performance.

We show the results of applying LM with PTLB in Figures 7 and 8, which compare the performances of the four settings described above in terms of query throughput and total tuples in system, respectively, when PTLB is applied as the load balancing strategy. Here the term "total tuples" accounts for all tuples across all machines, not just tuples on one machine. This shows the system performance as a whole.

The performance comparisons in term of throughput (accumulated) is shown in Figure 7. It is clear that the execution with neither forms of adaptation performs the worse. When applying the *PTLB-only*, the performance improves about 100% because the workloads are more balanced on all machines. The execution with only local plan optimization (*LM-only*) but no load balancing also generates about twice the number of tuples generated by *No-Adapt*. This shows that local plan optimization, as a runtime adaptation technique, can be as powerful as the widely used load balancing. Lastly, the execution with both forms of adaptions (*LM-MSLB*) has the best performance, producing about 330% more tuples than the *No-Adapt*. This illustrates that combining the two techniques can lead to a better performance than applying either adaptation alone.

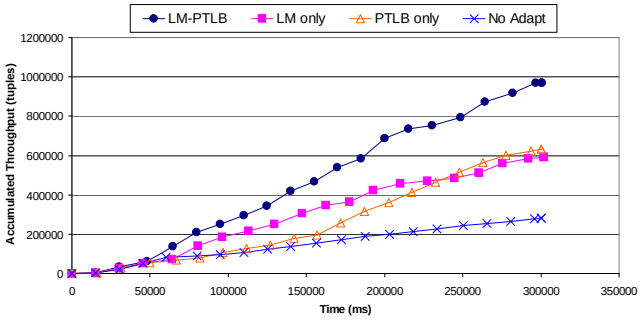



**Figure 7. Accumulated Throughput Comparisons (PTLB).**

The total number of tuples in the system is a good indicator for how well the query performs. A build-up of tuples in the system indicates that the query engine is not able to keep up with the current workload. Figure 8 depicts the comparison of total system tuples among the four settings. The two settings with local plan optimization (*LM-only* and *LM-PTLB*) have much lower system tuple build-up than the other two settings (*PTLB-only* and *No-Adapt*). This is because both settings can apply query optimization as soon as the changes in stream input rates are first detected. The reduction of system tuples happens for *PTLB-only* as well but is much more behind the above two settings because it needs to deal with a much higher tuple build-up. The *No-Adapt* has about the same highest build-up as *PTLB-only* and number of tuples slowly drops as a result of slower stream input rates. But this drop may happen too slow and thus too late for a system with limited amount of memory. Both the *PTLB-only* and *NO-Adapt* have higher likelihood of causing system overflow than the other two settings. This set of results shows that load balancing itself may sometimes have very limited impact on lowering the total memory cost of the system. Plan optimization, even if local, can be much more critical when it comes to releasing the burden of memory in the system.

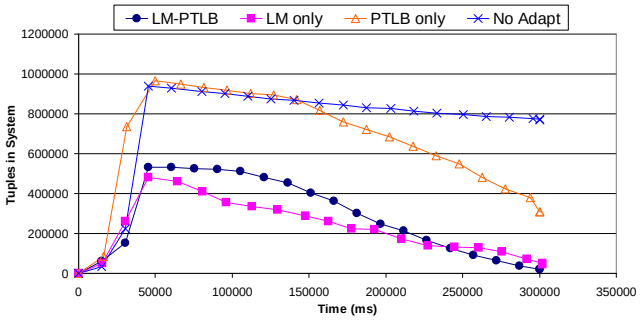



**Figure 8. Total Tuples Comparisons (PTLB).**

When MSLB is applied as the load balancing strategy, we observe very similar patterns in both performance charts as compared to the corresponding performance charts depicted in Figures 7 and 8, respectively. Further discussions on this set of results are omitted here due to space limits.

In summary, our experiments have shown that applying query optimization can be very effective when processing distributed continuous query plans. Furthermore, we have made the following three observations based on our tested scenarios: 1) local query optimization can be as effective as load balancing in by improving partitioned continuous query performance in distributed systems. 2) query optimization can decrease the total system resource consumptions while load balancing only balances the workload but does not decrease it. 3) Combining both adaptation techniques can significantly improve query performances beyond what would be achievable by only applying each adaptation individually.

## 5.3 Comparing PTLB and MSLB

In this evaluation, we compare the runtime performances of the two proposed load balancing techniques, namely PTLB and MSLB. We vary window sizes and stream rates, in order to compare the two strategies in a range of parameter settings, from low, medium to high. The stream rates are set to be one of the three values: 30, 40 or 50 tuples/sec, while the window sizes are set to be one of the four settings: 15, 30, 45, 60 second. Therefore we have 3 x 4 = 12 different experimental settings. During our experiments, we run each setting for at least 5 times, and get the average

of the total throughput as the throughput of that setting. All the other environment setup is the same as in the previous section.

For each setting, we run the experiment with no adaptation to serve as base performance. We then run the experiment by applying either PTLB or MSLB to adapt the query plan. The average throughputs of PTLB runs and MSLB runs are then divided by the base average throughput to get the scaled *throughput ratio*. The throughput ratio for the run with no adaptation is 1 since it is divided by itself. The larger the throughput ratio is, the better the query performs as compared to the run without adaptation.

Figures 9, 10 and 11 depict the results of the 12 settings with different combinations of window sizes and stream rates. Each figure compares the throughput ratio of the base case, the PTLB run and the MSLB run. Figure 9 shows the results of the 4 settings in which the stream rates are set to be 30 tuples/sec. We can see that as the size of the window grows, the difference of average throughput ratios between the base case and either PTLB run and the MSLB run are getting larger.

The difference between PTLB and MSLB also changes from insignificant, when window size is small, to about 25% difference, with the MSLB gaining the edge. This is because, as the window size grows, the total time to finish PTLB also becomes larger ( estimated as 2W). This means the over-loaded machine will continue to be overloaded because it needs to purge out all the old tuples. This slow relief can have a negative impact on the overall system performance. In comparison, MSLB releases the overloaded machine right away by moving tuples to another machine. Even if some states are unmatched and need to be recomputed, this work will be done at the receiver side, which is expected to be the underloaded machine. Therefore the impact of such recomputation to the overall query performance would be rather light.

We can observe similar but more dramatic trends in Figure 10, where the stream rates are all set to be 40 tuples/sec. Since the stream rate is higher than in the previous set of results, the lead of the PTLB and MSLB versus the base case is much larger even when the window size is small.

In Figure 11, when the stream rate is set to the relatively high 50 tuples/sec, the trend is a bit different. First, when the window size is small, the different between PTLB or MSLB and the base case is very large. On average, the PTLB produces about 90% more tuples while the MSLB produces about 100% more tuples than the base case. However, as the window size grows larger, this difference is not further enlarged. Instead, the gap between the base and the PTLB is getting narrower. This is because as both stream rates and window sizes are high, the PTLB starts to take a long time and consume large amounts of system resources in order to purge all old tuples on the already overloaded sender machine. Therefore it becomes less efficient. On the other hand, the MSLB is becoming more efficient in comparison to the PTLB, demonstrating that MSLB is a better choice when parameters have high values.

Figure 12 compares the average total time taken by the two load balancing strategies in the 4 experiment settings when the arrival rates are set to be 40 tuples/sec. Similar results are also observed for the other 8 settings but are omitted here due to space limit. As have been estimated, the PTLB always takes approximately 2W time to finish, while the MSLB usually takes much shorter time to complete the whole process.

So far our experimental results have shown that the MSLB strategy is winning. However, given certain combinations, the PTLB can perform better than the MSLB as well. This is when

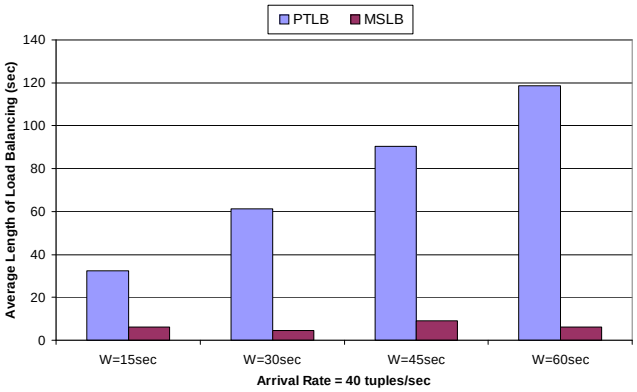



**Figure 12. Average Lengths of Load Balance ($\lambda$ = 40).**

the cost for state moving and recomputation is high (large state size) while the cost for processing new tuples is relatively low (low stream rates). Such situation will happen when the stream statistics changes shortly before the load balancing process.

We set up an experiment to reflect this situation. For the three input streams, A, B and C, the initial input rates are 100 tuples/sec. At 30th second, the input rates for B and C slow down to 5 tuples/sec. This triggers a local query optimization on the machine with the highest workload. The load balancing process is then invoked. The result of this experimental setup is shown Figure 13. As we can see, the PTLB starts to have better performance after the load balancing process is triggered. This is because such stream changes benefit PTLB as it lowers the cost of purging old tuples. However, since the state size has already grown very large at this point, the cost of moving state tuples and recomputing unmatched states can be high. So in this case PTLB is winning.

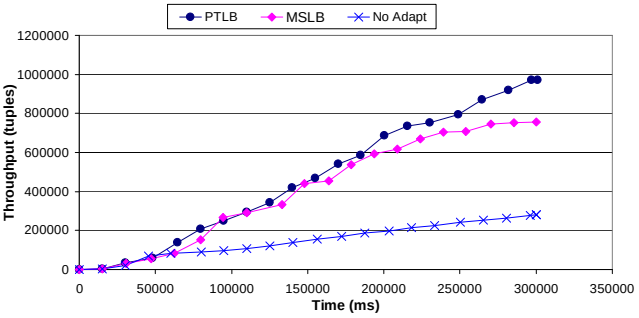



**Figure 13. PTLB-better-than-MSLB Case.**

**Discussion.** In summary, we have demonstrated that MSLB has better performance than PTLB because the former utilizes the underloaded machine more while the latter continues using the already overloaded machine to purge old tuples. However, under certain circumstances, the cost of state purging can be smaller than the cost of state moving and state recomputing. This may occur when the data statistics change towards the direction that decreases the cost of PTLB. In this case applying PTLB can be more efficient than applying MSLB.

## 6. Conclusions

Existing load balancing solutions have made the simplifying assumption that query plan instances on all machines are static,

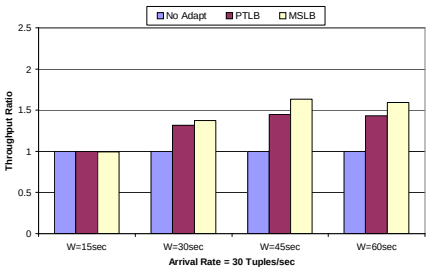



**Figure 9. Throughput comparisons ($\lambda$ = 30).**

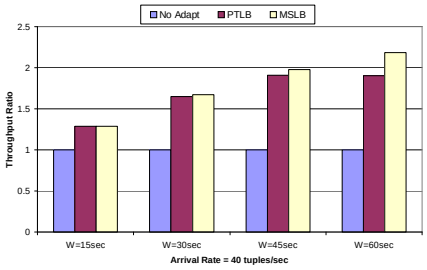



**Figure 10. Throughput comparisons ($\lambda$ = 40).**

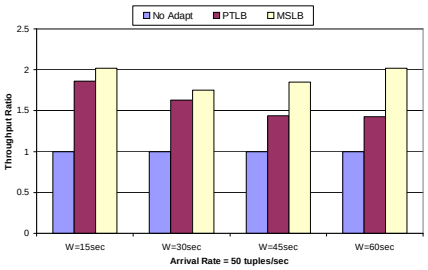



**Figure 11. Throughput comparisons ($\lambda$ = 50).**

i.e., no query optimization is conducted at runtime. This is clearly unrealistic for dynamic stream systems. In this paper, we point out that adding plan optimization to distributed continuous query processing is beneficial but doing so also creates new problems in dynamic load balancing. The new problem is the heterogeneity of query plan shapes among machines as a result of applying local query optimization, which has yet to be dealt with by current state-of-the-art load balancing strategies. We therefore propose two new load balancing strategies, namely the PTLB and the MSLB strategies, along with their corresponding protocols, that can balance the workload while seamlessly handling the complexity caused by local plan changes in the system. The PTLB strategy is a general load balancing strategy that requires no knowledge of the underlying query plan optimization. The MSLB strategy, on the other hand, rebalances the workload by comparing the detailed shapes of the query plans among different machines. Both strategies have been implemented in our prototype continuous query system. Our experiments show that the combination of query optimization and load balancing exhibits significantly superior performances than applying each adaptation technique alone. The MSLB is shown to be more efficient than the PTLB in most experiments.

Our future work will investigate other stateful operators, such as groupby and duplicate elimination. Also, other integration issues, such as relocation decisions based on local plan choices may be studied.

## 7. References


[1] D. J. Abadi, Y. Ahmad, M. Balazinska, and et. al. The design of the borealis stream processing engine. In *CIDR*, 2005.

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

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

[4] D. Carney, U. Cetintemel, M. Cherniack, and et. al. Monitoring streams: A new class of data management applications. In *VLDB*, pages 215–226, 2002.

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

[6] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *ICDE*, pages 345–356, 2002.

[7] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemal, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

[8] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, pages 948–959, 2004.

[9] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *ACM SIGMOD*, pages 102–111, 1990.

[10] W. Hasan. Optimizing response time of relational queries by exploiting parallel execution. In *Ph.D Thesis, Stanford University*, 1995.

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

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

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

[14] B. Liu, Y. Zhu, and E. A. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *ACM SIGMOD*, 2006.

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

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

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

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

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

[20] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *ACM SIGMOD*, pages 431–442, Paris, France, June 2004.