Optimizing State-Intensive Non-Blocking Queries Using Run-time Adaptation

Bin Liu, Mariana Jbantova, Elke A. Rundensteiner
Worcester Polytechnic Institute, Worcester, MA 01609

E-mail: binliu@alum.wpi.edu, (jbantova | rundenst)@cs.wpi.edu

Abstract

Main memory is a critical resource when processing non-blocking queries with state intensive operators that require real-time responses. While partitioned parallel processing can alleviate the stringent memory demands in some cases, in general even in a distributed system main memory remains bounded. In this work, we thus investigate the integration of two run-time adaptation techniques, namely, state spill to disk and state relocation to an alternate machine, to handle this memory shortage problem. We analyze the tradeoffs regarding key factors affecting these two run-time operator state adaptation techniques in a modern compute-cluster environment. Two strategies, lazy-disk and active-disk, are then proposed that integrate both state spill and state relocation adaptations with different emphasis on local versus global decision making. Extensive experiments of the proposed query processing system conducted on a compute-cluster (not merely a simulation) confirm the effectiveness of these strategies.

1. Introduction

Characteristics of Non-Blocking Pipelined Queries. Non-blocking query processing with data being pushed asynchronously from various data sources into the system and producing query results in real time as data comes through has become the focus of recent research. Efficient processing of such non-blocking queries is the key to the success of many applications including remote sensor monitoring and online transaction processing [2, 4].

Current research of non-blocking query processing often assumes that query operators have fairly small-sized operator states, e.g., small-window joins or even stateless operators such as select and project [2, 4, 11]. Query operators with potentially huge operator states, such as multi-joins, have not yet been carefully studied in this context. However, such operators are common in data integration and data warehousing environments. For instance, a real-time data integration system such as the emerging electronic brokerage systems could help financial analysts and brokers in making timely decisions regarding foreign and domestic ventures. Here, stock prices, volumes, exchange rates, ask and bid offers and external reviews are continuously sent to the integration server during working hours, say 9AM-4PM. The server is required to integrate these input streams and output the results to various decision support systems as well as other integrated applications as shown in Figure 1. This way, analysts and brokers make decisions in real time based on the most up-to-date information.

Such financial data integration systems often have to connect

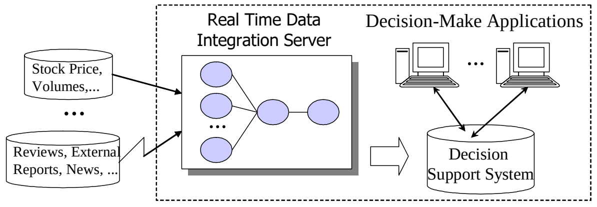
Real Time Data Integration Server
Decision-Make Applications
Stock Price, Volumes,...
Reviews, External Reports, News, ...
Decision Support System

Figure 1. A Real-time Data Integration System

streams from numerous market participants which may include hundreds of banks, international corporations and other financial institutions, matching order information based on market participants' ids and financial instrument types. Thus such systems must effectively handle multi-join integration queries. Moreover, the larger the number of different types of information streams such a system integrates, the more competitive the system often becomes, since it can serve the needs of its users better.

Let us assume a financial data integration system built to meet the needs of a financial consultant to a big corporation which has many international ventures requiring often payments in foreign currency. Since exchange rates of currencies fluctuate often based on how a country's economy is doing, such a corporation may need to reduce the transaction exposure of its international ventures by deciding whether to participate in the futures market and hedge the exchange rate at which it will have to buy the currency it needs for a future payment earlier or to wait until the payment is due and buy the currency at the then-spot rates. To advise such a corporation a financial consultant might want to know current exchange rates for the currency under question from different banks and possibly to read reports on the country's economy. The financial consultant might also be interested in finding out which brokers sell the currency at the lowest price. An example of a multi-join query answering one of the above mentioned inquiries is:

```
QUERY 1:
  SELECT brokerName, min(price)
    FROM bank1,bank2,bank3
   WHERE bank1.offerCurrency=bank2.offerCurrency
         AND bank2.offerCurrency=bank3.offerCurrency
         AND bank1.offer=bank2.offer
         AND bank2.offer=bank3.offer AND
         bank1.timestamp$>=$bank2.timestamp+window
         AND bank1.timestamp>=bank3.timestamp+window
GROUP BY brokerName
```

In the above example assume the data integration server is connected to the networks of other banks and a company that is constantly providing financial reports on countries' economies and currency rates in the form of continuous streams. The stringent requirement of generating near real-time results demands efficient main memory based query processing. This is particularly critical for data integration type queries that are state intensive in nature such as the multi-join integration queries mentioned above. In this work, we thus focus on adapting operator states to address this run-time memory shortage for queries with state intensive operators, such as multi-join queries [26]. These queries are common in data integration related applications as shown in Figure 1. As we need accurate query results and thus cannot afford to lose financial data, we cannot resort to techniques such as load shedding or approximations [23]. As motivated in Figure 1, we assume the query is long running but finite. However, the techniques we study in this work could also be applied to cases with infinite data streams as long as operators have finite window sizes, a common situation in continuous query processing environments.

State-Level Adaptations. One solution to address the memory shortage, as discussed in XJoin [25], Hash-Merge Join [17], and [15], is to temporarily push memory resident states into disks. This approach delays the processing of certain states (the disk resident states) until a later time when more resources would be available. The processing of the disk resident states is referred as *state cleanup*. Cleanup generates any missed results. We refer to this pushing and cleaning process as *state spill* adaptation. Given a monotonic increase of the operator states during the run-time phase due to long-running queries and requiring full and accurate final results, there may be no opportunity to perform the state cleanup during the run-time phase. Thus, these disk resident states would typically be processed after the run-time phase finishes [1].

An alternate solution is to distribute state intensive operators to multiple machines with each machine processing a partition of the input data. This is referred to as *partitioned parallel processing* [10, 14, 18, 20]. Then, we would move state partitions across machines when only a subset of machines is overloaded while others may exhibit available memory resources. For simplicity, we call this type of adaptation *state relocation*. As advantage, the adapted states remain in main memory and thus active once the relocation is completed. However, this type of adaptation will not always solve the overall memory shortage problem since even the aggregated main memory of multiple machines remains limited.

Contributions. While previous work studied the state relocation and state spill adaptations separately [17, 20, 25], we now investigate both state level adaptations in an integrated manner. Clearly, such a comprehensive solution is needed since state spill may not be efficient due to the access of slow secondary storage, while state relocation alone often cannot fully resolve the memory shortage problem. We analyze the tradeoffs regarding the factors to be considered when adapting states of *multi-input* operators using either of these two techniques in a practical cluster environment.

The main contributions of this work are:

- A comprehensive integrated solution is designed to maximize the run-time query throughput in environments where the memory of the distributed system is not sufficient for the query processing. In particular, we propose two integration strategies, namely, *lazy-disk* and *active-disk*, to apply both spill and relocation adaptations.

- The integration of these two techniques requires 1) a protocol to coordinate decision making at both local and global control agents, and 2) an assessment of the tradeoffs to know when to favor one technique over the other. A new partition group level productivity metric is designed to select the best partitions to be used by either adaptation process.

- The integrated adaptation strategies have been implemented in the framework of a distributed software architecture with a coordinator as a global adaptation controller agent in charge of monitoring and coordinating the work of each query processor and its local adaptation controller.

- Extensive experimental evaluation to compare these proposed strategies in the new context of continuous data stream processing has been conducted on a modern PC-compute cluster – that is, the observations are not just derived based on simulation-based studies. The experimental results confirm the effectiveness of the proposed strategies.

[1]We assume applications here for which out-of-order delivery of output tuples is acceptable.

The rest of the paper is organized as follows. Section 2 overviews the basics of our approach. Sections 3 and 4 analyze the state spill and relocation solutions, respectively. Section 5 presents the integrated adaptation strategies, along with their experimental evaluation. Sections 6 and 7 discuss related work and conclusion.

2. Basics of Our Approach

Partitioning State Intensive Operators. We review partitioned processing for multi-input state intensive operators. Throughout this work, we use a symmetric multiple-way hash join operator [26] as a representative example of state intensive operators due to joins being one of the most common class of queries. Other state intensive operators can also be addressed in a similar manner as long as their functionality can be distributed to multiple machines with each machine only processing non-overlapping partitions. As discussed in [10, 20], a *split* operator is inserted in front of each input stream of such a partitioned operator. This split operator partitions an input stream and sends the appropriate partitions to each machine that houses an instance of this partitioned operator. For simplicity, we will henceforth refer to each instantiation of the operator that runs in a particular machine as an *instance* of the partitioned operator.

For example, assume we process a three-way join query ($A \bowtie B \bowtie C$) as shown in Figure 2 (a). The join is defined as A.A_1 = B.B_1 = C.C_1 where A, B, and C denote the three input streams and A_1, B_1, and C_1 are the join columns. As shown in Figure 2(b), the query is partitioned and run on two machines. The $Split_A$ operator partitions the input stream A based on the column A_1, while the $Split_B$ operator partitions the input stream B based on B_1, and so on [2] A *union* operator, if needed for appropriate result merging, can be inserted into the output streams of all instances of the partitioned operator to combine the results into one stream for further processing. Stateless operators such as split and union are evenly distributed among all available machines during such a partitioned processing as they consume very limited memory and thus tend not to be the bottleneck in terms of resource consumption.

[2]To keep the discussion focussed, we assume here that each partitioned join operator has all its join conditions defined on the same domain for each input. This is trivially true for binary operators and typically also assumed for m-way continuous join opeators by the stream literature. Trees of such operators, each with its own join columns, can be naturally supported [15]. Further, strategies

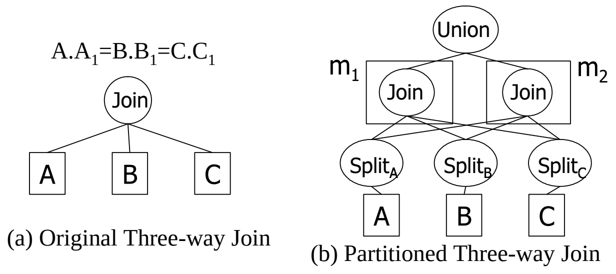

Figure 2. Example of Partitioned Processing

Adaptation without Re-Hashing. To effectively adapt partitions without ever having to rehash any of the existing partitions at run time, each split operator divides each input stream into a much larger number of partitions than the number of available machines. We identify each partition by a unique *partition ID*, i.e., 1, 2, . . ., n (with n the number of distinct partitions). For example, we might work with 500 partitions over 10 machines. Adaptation such as spilling or relocating is now done at the granularity of these 500 partitions. This method has first been applied in the early data skew handling literature [7] as well as in the recent partitioned stream processing work Flux [20].

Partition-Group Granularity for Adaptation. We organize operator states based on input partitions as done in [15]. Single input query operators, such as the ones tackled in Flux [20], have to adapt partitions from one input stream only. However, the multi-input operators we focus on have in their states partitions from different input streams with the same partition ID. Thus, multiple ways of organizing partitions are possible.

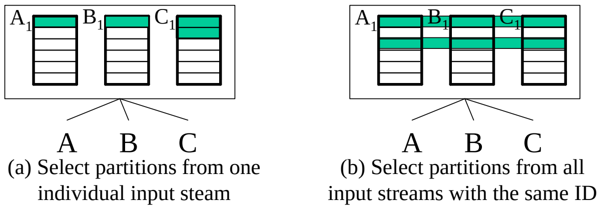
A$_1$
B$_1$
C$_1$
A
B
C
(a) Select partitions from one individual input steam
A$_1$
B$_1$
C$_1$
A
B
C
(b) Select partitions from all input streams with the same ID

Figure 3. Composing Partition Groups

As proposed in XJoin [25], we could choose partitions from one input at a time and adapt them independently (Figure 3(a)). However, this strategy has two potential drawbacks in partitioned processing of multi-way join queries. (1) It increases the complexity in the cleanup process. This is because if partitions have been pushed to disk, this requires us to keep track of the timestamps of when each of these partitions was pushed, and the timestamps of each tuple in order to avoid duplicates in the cleanup process. For example, tuples from partition A_1 have been pushed into disk at time t during execution. As done in previous work [15], we use A_1^1 to denote this part of partition A_1. Then all the tuples from B_1 and C_1 with a timestamp greater than t have to join with the A_1^1 in the cleanup process. Given A_1, B_1, and C_1 could be pushed into the disk more than one time, the cleanup needs to be carefully synchronized with the timestamps of the input tuples and the timestamps of the partitions being pushed. (2) Worse yet, if state relocation were to move partitions from individual inputs to different machines, this then would force us to process tuples for that partition with across machine joins. For example, if we have partition A_1 in machine M_1, while partitions B_1 and C_1 are in machine M_2, then a newly incoming tuple that belongs to A_1 has to access both machines M_1 and M_2 to produce the join result. Instead, if we were to put all three partitions A_1, B_1, and C_1 in the same machine, we could access one machine only to produce this join result. Thus this would tend to be more efficient.

for handling single n-ary join with different join columns can be devised.

Accordingly, we group the partitions with the same partition ID across all input streams of an operator together as the smallest unit to be adapted, as illustrated in Figure 3(b). This avoids the expensive processing of queries across multiple machines. It also greatly simplifies the cleanup process as discussed above because neither timestamps nor other metadata has to be kept at the operators. For simplicity, we henceforth call all partitions with the same partition ID even when from different inputs one *partition group*, or in short, partition if the context is clear. As unlike [15] where the focus is on dependencies at the partition level across operators in a pipeline, here we focus on distributed query processing, thus the idea of ”across-state-partition” even makes more sense as it keeps the join local to a machine and thus much cheaper compared to alternate partition choices. While the techniques proposed in this work are equally applicable to the granularities of partitions and partition groups, without loss of generality we henceforth will assume the later granularity for simplicity reasons.

Partition Group Productivity Metrics. The usage of partition groups as adaptation unit helps to simplify statistics collection by reducing it to the granularity of each partition group — compared to either full operator-level (too coarse-grained) or tuple-level (too fine-grained and thus not practical).

Here, we propose a new metric to be employed by the policies of both adaptation techniques, called *partition group productivity*. For each partition group, we record its current size, represented by P_{size}. We also record how many tuples have been generated from this partition group, denoted by P_{output}. We define the *productivity* of each partition group as P_{output}/P_{size}. Given a similar size P_{size}, a small P_{output}/P_{size} value indicates that only few output results have been generated so far.

The productivity value of each partition group reflects the input data that has been processed so far. They are updated when new data gets processed. As commonly assumed in databases for lack of any knowledge about the future, we also assume here that the values we observed so far would be indicative of the trends of behavior of the partition groups in the near future. Clearly, alternate ways of computing the productivity value exist. For example, we can maintain snapshots of historical values and assign higher weights to more recent values using an amortized weight function to compute a tuned partition productivity value, depending on the perceived stability of the operator's behavior. Alternative cost models could be easily plugged into our system in the future if it turns out to be necessary, as the particular policies of run-time state adaptation are independent of cost model details .

Distributed Software Architecture: Local and Global Adaptation Controllers. We analyze our run-time adaptation strategies based on a distributed system with the following architecture (Figure 4): A dedicated *global coordinator* (GC) is in charge of a set of *query engines* (QE) running on different machines. The global coordinator distributes the query and connects operators that are distributed into different query engines. It collects and analyzes running statistics of each processor. It also makes *coarse-grained adaptation decisions* such as how many states to relocate from one processor to the other but not which partition groups. A query engine takes care of executing the portions of the continuous query plans assigned to it. Each query engine reports statistics to the global coordinator. A *local adaptation controller* at each engine is responsible for choosing partition groups to adapt (to be spilled or relocated). It also engages in the protocol to relocate operator states from or to a query engine on another machine.

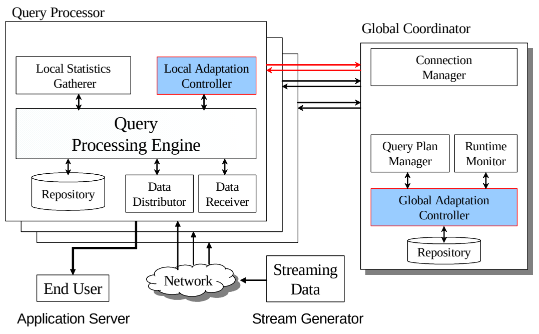

Figure 4. System Architecture

In this architecture, the state relocation decision is made by the global coordinator based on the collected system statistics, while the detailed decision of choosing which partitions to spill/relocate is handled by the local adaptation controller in each processor. As discussed in [21], the global coordinator is very scalable and not likely to become a bottleneck since it collects only light-weight statistics from the processors. Unlike Flux [20], which puts all the adaptation and partitioning functionalities into each individual Flux query operator, our architecture naturally facilitates tiered adaptation decision making. Below we exploit this property to design effective adaptation strategies.

3. State Spill Adaptation

The integrated adaptation strategies proposed in this paper are based on two independent query processing optimization techniques, namely *state spill* and *state relocation*. Thus, before presenting our integrated solution, we first study each individual technique in our targeted cluster environment.

State spill refers to the process of pushing memory resident states into disks temporarily when memory overflow happens. Given a monotonic increase of memory usage during run-time, these spilled states will be kept in disk (inactive) until the memory overflow has been addressed. State spill thus requires a second *cleanup disk phase* during which disk resident states need to be brought back to memory for further processing - so that missing results are produced while preventing duplicates. This *state cleanup process* can be performed at any time when memory becomes available. Note that multiple partition groups may exist given one partition ID. This is because once a partition group has been pushed into disk, new tuples with the same partition ID may continue to accumulate to form a new partition group in main memory. If needed this partition group could be pushed into disk again.

The tasks that need to be performed in the cleanup can be described as follows: (1) Organize the disk resident partition groups based on their partition ID. (2) Merge partition groups with the same partition ID and generate missing results. (3) If a main memory resident partition group with the same ID exists, then merge this memory resident part with all disk resident ones. We have observed that incremental view maintenance algorithms [13] can be applied to merge partition groups and produce missing results. The details of the cleanup process are omitted for space reasons.

Within our framework, the state spill process is local to a query engine. Here we utilize a throughput-oriented state spill strategy that aims for a high run-time output rate – though other metrics would be possible. That is, we aim to generate as many output results as possible given part of the memory resident states are to be pushed into disks (temporarily inactive). As our experiments confirmed, a high overall throughput run-time phase also reduces the efforts in the cleanup process as more work would have been already completed.

To achieve maximal throughput, the state spill policy must decide which partition groups to push when memory overflows. Different flush policies have been discussed in the literature. For example, XJoin [25] flushes the largest partition. Our spill policy uses the metrics as defined in Section 2 to rank partition groups by their productivities and then selects partitions that are less productive to be pushed in each spill process. The intuition is that the partitions left in main memory are more likely to produce more results than the ones that have been pushed into disks.

3.1 Experimental Setup and Environment

Experimental Environment. The system described in Section 2 used in our experiments is deployed on a 10-machine cluster. Each machine in the cluster has dual 2.4Ghz Xeon CPUs with 2G main memory. These machines are connected via a private *gigabit* ethernet. Due to the security settings of the cluster, in our experiments we cannot stream data from outside the cluster. However, this does not impact the validity of the results of our experiments. We dedicate three machines to run the *global coordinator*, *stream generator*, and *application server* respectively. The stream generator continuously generates stream input tuples for queries to process, while the application server processes the output results. Other machines are deployed as processors as necessary for the given experiment.

Data Characteristics of Long-running Queries. Similar to [7, 22, 23, 25], we use synthesized data in our experiments to be able to control various factors of the input streams. We define *join multiplicative factor* as the average number of tuples with the same join value per stream for a given time period. This join multiplicative factor is closely related to the number of join results. For example, we assume each input stream of a three-way join has 5 tuples with a join column value 1 after processing 2000 tuples. Thus, a total of $5 \times 5 \times 5 = 125$ tuples will be generated with a join column value of 1. Clearly, after processing another 2000 tuples, each stream would have seen 10 such tuples with join value 1 in total. Thus, the total output generated at that point would be $10 \times 10 \times 10 = 1000$. As we intuitively see, the output rates (as well as states of stateful operators) monotonically increase during such long-running execution. That is, the join multiplicative factor increases as more input tuples get processed. We thus define the term *join multiplicative factor increase rate* (r), or simply *join rate*, to describe that the join multiplicative factor increases r after processing every k tuples. Here, k is referred as *tuple range* of the input stream. The join rate of partitions can also be defined similarly. Note that given a uniform distribution of join values, the join rate of each partition is the same as the join rate of the whole input stream. This no longer holds for non-uniform distributions.

3.2 State Spill Evaluation

We first investigate the sensitivity of how much state is to be pushed in each spill process (Figures 5 and 6). We run the three-way join query, described in Section 3.1, on one single machine. The input rate is set to 30 ms per input stream. The tuple range of each input is set to 30K. The join rate is set to 3. The state spill is triggered whenever the memory usage of the machine is over 200MB. A k%-push means that k% of the main memory states are chosen to be pushed to disk. We vary k from 10 to 100 in this experiment. We randomly choose partition groups from the operator state for this experiment since we investigate the impact of which amount of state is to be pushed in each adaptation. As a comparison, we also provide the throughput of the query when it is fully processed in main memory (labeled as 'All-Mem').

Seen from Figure 5, the more states are being pushed into the disk each time, the smaller the overall throughput. This is as expected since the states being pushed are no longer active.

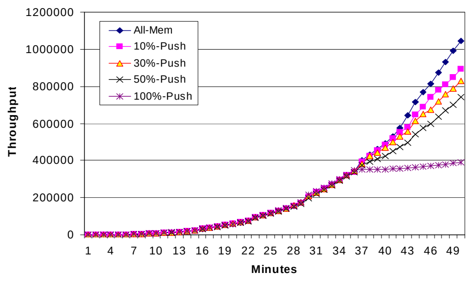

Figure 5. Varying k%. Impact on Throughput.

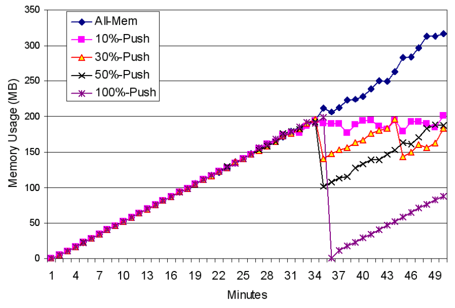

Figure 6. Varying k%. Impact on Memory Usage.

Figure 6 shows the corresponding memory usage changes. We can see that the main memory utilization can be effectively controlled to avoid system crash due to memory overflow. We also see that the more states (a higher percentage) we push in each adaptation, the fewer times we need to trigger the state-spill process. In Figure 6, each zag in the line represents one adaptation process.

Without loss of generality, we observe that some intermediate value of push volumes in the range of 10% to 50% is most favorable given our experimental environment. It balances the main memory usage (number of adaptations) and the impact on overall throughput. In subsequent experiments, when we want to hold characteristics in our experimental runs steady to allow us to focus on the effect of particular parameters, we choose such a middle range value, say 30% as default, unless otherwise stated.

Next, we study the effectiveness of productivity as metric for deciding which partition groups to push to disk. Figure 7 shows the impact of choosing different partition groups on the overall run-time throughput. Here, the input stream has 1/3 of the partitions with an average join rate of 4, 1/3 with an average join rate of 2, while the rest have a join rate of 1. The 'push-less-productive' corresponds to the case of pushing partition groups with the smallest P_{output}/P_{size} value first. The 'push-more-productive' denotes the pushing of partition groups with the largest P_{output}/P_{size} value first. Seen from Figure 7, the push-less-productive strategy has a much higher run-time throughput. This is because leaving partition groups with high productivity in main memory is more likely to generate more output results as input tuples come through. In Figure 7 we find that after 40 minutes of query execution, the push-less-productive partitions strategy performs about 70% better in terms of output rate. Our experiments also confirm that this policy helps to reduce the cleanup efforts as more work has been accomplished before the cleanup starts. In the above experiment, the push-less-productive strategy uses $26,879$ ms to generate $194,308$ tuples during the cleanup, while the push-more-productive one generates $992,893$ tuples in around $359,396$ ms.

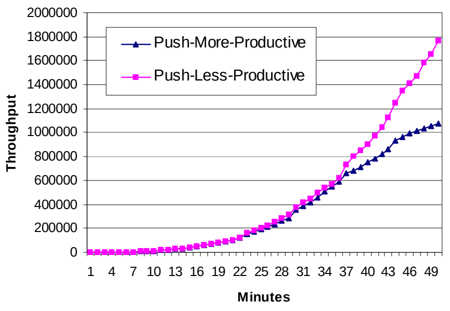

Figure 7. Throughput-Oriented Spill Evaluation

4. State Relocation Adaptation

Uneven workload may arise among machines in a distributed environment. Thus, while one machine runs out of memory, another machine may still have ample memory at its disposal for holding additional states. Unlike state spill, having operator states stay in main memory (even at an alternate machine in the cluster) would positively affect the overall query processing. Thus, state relocation, which moves operator states across machines, may then become a preferred choice to maximally utilize all resources.

State relocation requires knowledge from multiple machines to make an adaptation decision. In our system we have a global coordinator (GC), which monitors the performance of all query processing engines (QE). Thus it has global knowledge of the overall system performance. In the distributed system used in our experiments, the global controller coordinates the execution of the different query optimization strategies, and is thus responsible for the state relocation process. Various schemes of relocation among a set of machines have been studied in the literature. Here we proceed with a simple model, namely a pair-wised state relocation scheme. Other models could fairly easy be incorporated into our framework. We refer to the machine with the maximally used memory (M_{max}) as the *sender*, and to the machine with the least memory used (M_{least}) as the *receiver*. In each adaptation, the global coordinator moves $(M_{max} - M_{least})/2$ amount of states from the sender to the receiver if it is observed that M_{least}/M_{max} reaches a threshold θ_r, i.e., $M_{least}/M_{max} < \theta_r$. Thus, ideally, both machines will have about $(M_{max} + M_{least})/2$ memory usage after the adaptation. Note that the actual partition groups to be moved are decided by the local adaptation controller of the machine with the most used memory. Given such tiered decision architecture, the global coordinator only requires to collect very light-weight running statistics, such as main memory usage. This helps to increase the scalability of the global coordinator, thus reducing the possibility of it becoming a bottleneck in the query processing. Previous work [21] has proven the cost of communication in the context of our fast network to be very low.

4.1 Moving States Across Machines

No operator states should be missing or corrupted in the relocation process. To achieve that, we design an 8-step protocol to coordinate the run-time operator state movement. The overall interactions between the global coordinator (GC) and each individual query engine (QE) (for those query engines involved in the adaptation process) are described by the sequence diagram illustrated in Figure 8. During state relocation, the global coordinator controls the overall adaptation process, while the local adaptation controller in each query engine is responsible for receiving moving requests from the global coordinator and performing corresponding actions. The state relocation process is triggered by the global coordinator whenever an overloaded query engine is detected. Note that since a split operator in charge of redirecting incoming tuples to the correct instances of the partitioned stateful operators sits in front of each partitioned operator, all tuples, belonging to the partition groups affected by the current adaptation process, which arrive during a state relocation process are temporarily buffered at the query engine on which the corresponding split operator sits. Later when the adaptation process is over all buffered tuples are redirected to the stateful operators based on the new partition group mapping. Due to space limits more detailed discussion of state relocation protocols is omitted.

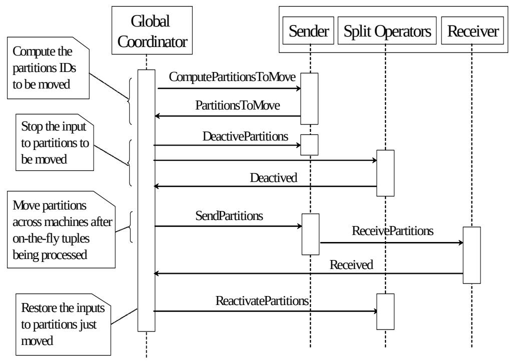
Global Coordinator
Sender
Split Operators
Receiver
Compute the partitions IDs to be moved
Stop the input to partitions to be moved
Move partitions across machines after on-the-fly tuples being processed
Restore the inputs to partitions just moved
ComputePartitionsToMove
PartitionsToMove
DeactivePartitions
Deactived
SendPartitions
ReceivePartitions
Received
ReactivatePartitions

Figure 8. State Relocation Protocols Diagram

4.2 Evaluation of State Relocation

We study the following two parameters in evaluating the state relocation: (1) threshold θ_r , and (2) minimal time-span between two consecutive relocations τ_m. The global coordinator triggers state relocation if and only if when M_{least} / $M_{max} < \theta_r$ and the time elapsed since the last relocation is greater than τ_m.

Figure 9 explores the threshold-related aspect of the above questions. The query as described in Section 3.1 is run in two machines. Each machine processes about half of all partitions. The maximal memory of each machine is set large enough to have the query completely run in main memory. We use a worst case situation in terms of input stream fluctuations having each machine alternatively change its demand of main memory. For example, partitions assigned to machine 1 get 10 times more tuples than those of machine 2 for the first five minutes. After that, machine 2 gets 10 times more tuples than machine 1 for the next 10 minutes, and so on. Thus the main memory usage of these two machines alternates dramatically every 10 minutes. Given this setup, the state relocation may keep on moving states constantly back and forth among two machines, i.e., the danger of thrashing by wasting time on moving states may arise. We now study the stability of our method in such a dynamic situation.

Figure 9 shows the impact of choosing the threshold θ_r with τ_m being set to 45 seconds. We vary θ_r from 50% to 90%. A high percentage indicates that a larger number of adaptations is triggered with each adaptation only moving a small amount of states. Seen from Figure 9, the throughput when choosing different θ_r is almost the same. All of them experience throughput similar to that of pure main memory processing with no adaptations (All-mem). In Figure 9, a total of 24 relocations have been conducted when θ_r is set to 90%, while only 2 adaptations when θ_r equals 50%.

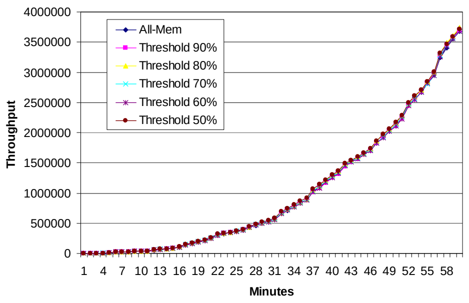

Figure 9. Varying Thres. θ_r

Figure 9 and other experiments we have conducted regarding the minimal time-span between two consecutive relocations confirm that the cost of our pair-wised state relocation is low in the context of our test environment, a modern small-range cluster of machines. Thus we conclude that potentially we could perform such state relocations frequently without impacting the overall performance. The state relocation cost is expected to be higher if the underlying network is slow and unreliable.

Figure 10 shows the change of memory usage with θ_t = 90% and τ_m = 45 seconds. The 'no-relocation-M1' and 'no-relocation-M2' show the memory usage respectively of machines M_1 and M_2 without state relocations. As can be seen, the memory consumption alternatively changes due to our input data pattern. 'with-relocation-M1' and 'with-relocation-M2' indicate the memory usage after the state relocations. We can see that the main memory usage remains largely balanced due to the relocation. Applying state relocation maximizes the opportunity for full memory based processing. It thus has the potential to result in a higher overall throughput since the cost of state relocation is not expensive as shown by Figure 9.

Figure 11 illustrates the benefits of state relocation. The query is run over three machines. We change the initial distribution of partitions to make one machine process 60% of all partitions, while the other two have 20% of partitions respectively. We set θ_r = 80% and τ_m = 45 seconds. In this setup, state spill is triggered when the main memory usage of the machine is over 200MB.

Seen from Figure 11, the throughput of the 'no-relocation' case drops after running for 40 minutes. This is because main memory of the machine having 60% of the partitions overflows and starts

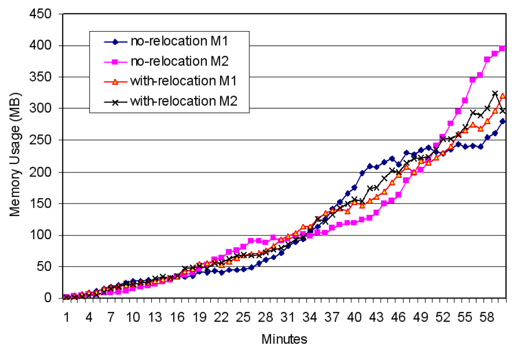

Figure 10. Memory Usage

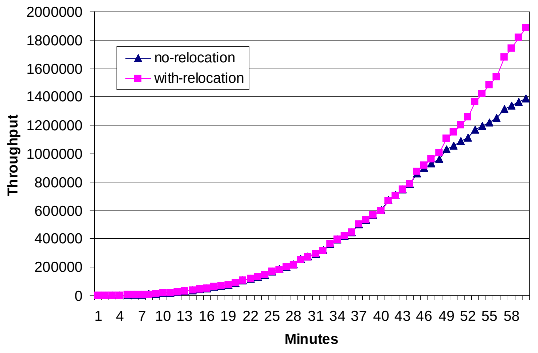

Figure 11. Relocation vs. Spill

pushing states into disks. On the other hand, the 'with-relocation' adapts these states to other machines having all states kept in main memory. Thus, it generates output continuously at a maximal rate in the run-time phase instead of waiting until the clean up stage.

5. Integrating State Spill and Relocation Adaptations

When aggregated memory of all machines is not sufficient for the query processing, state spills cannot be avoided any longer simply by relocating states across machines. This is because some machines or even all machines may suffer from memory overflow. In this section, we thus propose two solutions to combine both state spill and relocation into integrated strategies aimed at maximizing run-time throughput in such memory-constrained environments. Each strategy description is accompanied by pseudocode for more clarity. Variables used in the integrated strategies' algorithms are explained in Tables 1 and 2.

5.1 Lazy-Disk Control Strategy

In the lazy-disk strategy, state spill is postponed until there is no main memory in the cluster that can hold the states from overloaded machines. That is, when the global coordinator observes that $M_{least}/M_{max} < \theta_r$, the state relocation adaptation is chosen. This aims to have as many states as possible kept in main memory. While the local adaptation controller observes that the memory usage of the machine is going to overflow, it then triggers the state spill on that particular machine. Algorithm 1 describes the sequence of steps performed by the GC and all QEs affected by our first proposed solution, called *lazy-disk* approach.

The lazy disk approach focuses on the main memory usage, that is, both types of adaptations are purely driven by main memory usage. We push the less productive partitions (with small P_{output}/P_{size} values) to disk in the state spill process, while we choose the productive partitions (with large P_{output}/P_{size} values) to move in the state relocation adaptation. With productive partitions likely to be kept in main memory, this strategy aims to produce high throughput in the run-time phase. This strategy design is driven by our experimental observations in Section 3.1, namely, to keep as many states as possible in main memory.

Variable name	**Description**
ss_	marks state spill mode or events
sr_	marks state relocation mode or events
threshold^{sr} threshold^{mem} threshold^{prod}	tunable thresholds determining when states are to be relocated among QEs or spilled at a potentially locally overloaded QE
max_product	estimated maximum QE productivity per time period
min_product	estimated minimum QE productivity per time period
sr_timer	determines the frequency of statistics evaluation sent to to the GC by all QEs. The statistics are used to handle uneven load distribution situations
s_timer	determines how often memory at a QE is measured to detect and prevent memory overflow problems by triggering a local state spilling process
lb_timer	load balancing timer determining how often the GC should evaluate the statistics sent to it by the QEs.

Table 1. Variables for Lazy- and Active-Disk Strategies

QE Modes of Operation	**Description**
State-relocation mode (sr_mode)	Indicates that GC has triggered the state-relocation protocol which is carried out by the GC and all affected QEs.
State-spill mode (ss_mode)	Indicates that a QE is in the process of spilling states to disk to free memory. It is triggered in different ways based on the selected adaptation strategy.
Normal mode (normal_mode)	Normal query plan execution. No memory overflow problems detected, and thus no adaptation is currently attempted.

Table 2. Execution Modes of a Query Engine

5.2 Evaluation of Lazy-Disk Control

Similar to the experiment shown in Figure 11, a lazy-disk adaptation approach has the potential to fully utilize all available main memory in the cluster. Figure 12 shows the performance of the lazy-disk approach in a memory constrained environment. The query, refer to Section 3.1, is deployed on three machines. We set a skewed initial distribution with one machine being assigned 2/3 of all partitions, while other two machines share evenly the rest 1/3 of partitions. In this setup, if we do not apply state relocation, then only one machine gets overloaded. We call this 'no-relocation' approach. Using the lazy-disk approach, all three machines will eventually get overloaded due to giving priority to state relocation. Only once the memory across all machines is exhausted will this strategy trigger the state spill processes. Seen from Figure 12, the lazy-disk approach has a higher overall throughput than the 'no-relocation' since the lazy-disk approach makes full use of available main memory in the cluster during the query processing.

Even if the query workload is extremely heavy, i.e., each machine in the cluster does not have sufficient memory to process the partitions assigned to them, a lazy-disk approach still has benefits. To illustrate, we again deploy the query into three machines and have one machine get more partitions than the others. We run the query for 6 hours, so that each machine has a large amount of states beyond the available main memory. We again compare the performance of lazy-disk and no-relocation. In this experiment,

Algorithm 1 Lazy-Disk Strategy

```
   EVENTS AT GC
1: $\underline{sr\_timer\_expired}$:
2:     $sr\_timer.reset()$
3:     $process\_stats()$
4:     $calculate\_cluster\_load()$
5:     $max\_load :=$ get max load among all query engines ($QE_i$)
6:     $min\_load :=$ get min load among all query engines ($QE_i$)
7:     if min_load/max_load $< threshold^{sr}$ then
8:       QE_receiver:=$getQE(min\_load)$
9:       QE_sender:=$getQE(max\_load)$
10:      relocate_amount:=$computeAmount(max\_load, min\_load)$
11:      trigger $start\_sr(relocate\_amount)$ event at GC
   EVENTS AT QE
12: $\underline{cptv}$:
13:    if $mode = normal\_mode$ then
14:      $mode := sr\_mode$
15:      $parts\_move\_list = computePartsToMove(parts\_to\_move)$
16:      $cptv_t rig_r := true$
17:      trigger $ptv(parts\_to\_move)$ event at GC
18:    else (must be in state spill mode)
19:      wait until $mode \neq ss\_mode$
20:      $mode := sr\_mode$
21:      $parts\_to\_move = computePartsToMove(relocate\_amount)$
22:      $cptv\_trig\_r := true$
23:      trigger $ptv(parts\_to\_move)$ event at GC
24: $\underline{ss\_timer\_expired}$:
25:    $ss\_timer.reset()$
26:    if $QE\_memory > threshold^{mem}$ then
27:      if $mode = normal\_mode$ then
28:        $mode := ss\_mode$
29:        $spill\_amount = computeSpillAmount()$
30:        $stateSpill(spill\_amount)$
31:        $mode = normal\_mode$
32:      else (don't spill now, wait until next timer expires)
```

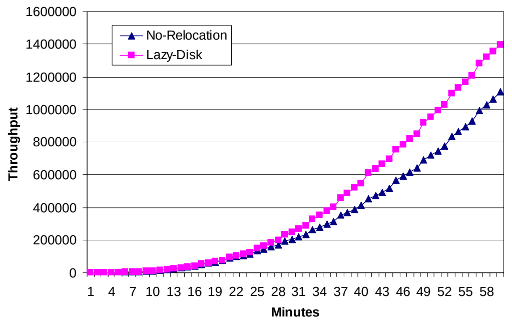

Figure 12. Lazy-Disk vs. No Relocation

the overall results generated by the two approaches are similar as they have similar amount of states being pushed to disk. However, the clean up stage of these two approaches are dramatically different. The no-relocation approach takes more than 1600 seconds to produce 2,023,781 tuples in the clean up stage. This is because most work is done by one machine. While the lazy-disk approach only takes less than 400 seconds to clean up, since work is already evenly distributed among all three machines before cleanup starts.

5.3 Active-Disk Control Strategy

State spill in the lazy-disk approach is a *local decision*. This means the decision is made by the query processor as the memory overflow happens at a local machine. However, the productivity of partitions among machines might not be the same. For example, the least productive partition in one machine, the candidate to be pushed to disk there, may still be much more productive than many other partitions in another machine. Thus, *if we raise the state spill decision to the global coordinator at a global level*, we could instead spill the globally least productive partitions among all machines. This should free more aggregated main memory space across the cluster for the globally most productive partitions.

Corresponding to this idea, we now design the *active-disk* approach which actively controls state spill adaptations at the global (instead of at the local) level. The global coordinator monitors both the main memory usage and the average productivity rates of machines in the cluster. Here, the *average productivity rate* (R) of one machine is defined as the total number of tuples that have been generated by this machine during the sampling time divided by the number of partition groups in the machine. Algorithm 2 describes the pseudocode for the distributed protocol that takes place at the global coordinator and the query engines during an active-disk adaptation process.

Algorithm 2 Active-Disk Strategy

```
   EVENTS AT GC
1: $\underline{lb\_timer\_expired}$:
2:   $lb\_timer.reset()$
3:   $process\_stats()$
4:   $calculate\_cluster\_load()$
5:   $max\_load :=$ get max load among all query engines ($QE_i$)
6:   $min\_load :=$ get min load among all query engines ($QE_i$)
7:   if min_load/max_load $< threshold^{sr}$ then
8:   QE_receiver:=$getQE(min\_load)$
9:     QE_sender:=$getQE(max\_load)$
10:     relocate_amount:=$computeAmount(max\_load, min\_load)$
11:     trigger $start\_sr(relocate\_amount)$ event at GC
12:   else
13:     $max\_product :=$ get max productivity among all query engines ($QE_i$)
14:     $min\_product :=$ get min productivity among all query engines ($QE_i$)
15:     if $max\_productivity$ / $min\_product > threshold^{prod}$
16:       QE_sender:=$getQE(min\_product)$
17:       $spill\_amount$:=computeAmountToSpill();
18:       trigger $start\_ss(spill\_amount)$ event at QE_sender
   EVENTS AT QE
19: $\underline{cptv}$:
20:   $mod := sr\_mode$
21:   $parts\_move\_list := computePartsToMove(parts\_to\_move)$
22:   $cptv\_trig\_r := true$
23:   trigger $ptv(parts\_to\_move)$ event at GC
24: $\underline{start\_ss}$:
25:   $mode := sr\_mode$
26:   $stateSpill(spill\_amount)$
27:   $mode := normal\_mode$
```

As in the lazy-disk approach, if $M_{least}/M_{max} < \theta_r$ we run out of local main memory, then the state relocation is triggered again aiming to have all data in memory whenever possible. However, if $M_{least}/M_{max} \geq \theta_r$, then we compare the average productivity rate of each machine. If one machine has a much lower average productivity rate, for example, $R_{max}/R_{min} > \lambda$, we force the partitions of the lower average productivity rate machine to be pushed into disks. This would leave main memory space for the highly productive partitions in other machines to be relocated into these machines. This would help the overall performance since high productive partitions remain in main memory. The global coordinator does not select the globally least productive partitions to be pushed to disk as this would require the collection of more statistics, increasing network costs and reducing scalability.

However, pushing more states than necessary could be counterproductive, resulting in a decrease of the overall performance as well. In the active-disk strategy, we set the maximal amount of states being pushed by the global coordinator to be less than $M_{query} - M_{cluster}$, where M_{query} denotes the estimation of the overall memory consumption for the query, while $M_{cluster}$ is the overall available memory of the cluster. This aims to assure that data that fits into memory is left there.

5.4 Evaluation of Active-Disk Control

We postulate that the active-disk approach could further improve run-time query throughput if the global coordinator observes major differences of productivity among machines. Figure 13 shows one comparison of lazy- versus active-disk approach. In this experiment, we set the tuple range of the input stream to 30K. We set the partitions assigned to machine m_1 to have a high average join rate of 4, while partitions in the other two machines have a low average join rate of 1. The lazy-disk approach does nothing at the global coordinator level if the memory usage among the three machines is running out roughly equally at all machines. While the active-disk approach forces lower productive partitions to be pushed into disks since the average productivity of partitions in machine m_1 is much larger than that of the other two, but only if extra memory is needed. Note that in both approaches, each machine triggers the state spill process as its memory usage reaches its threshold (60 MB). Here, the state relocation threshold θ_r is set to 0.8, while the minimal time span of two relocations τ_m is set to 45 seconds. The productivity threshold λ that triggers a 'force state spill adaptation' is set to 2.

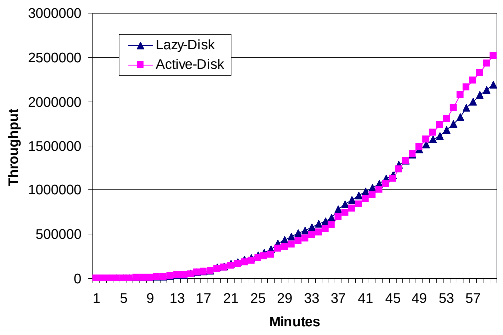

Figure 13. Lazy-Disk vs. Active-Disk, 1

Seen from Figure 13, the active disk strategy experiences a slight drop in the throughput after it starts pushing partitions into disks. Gradually, however, it outperforms the lazy-disk strategy since more high productive partitions remain in main memory.

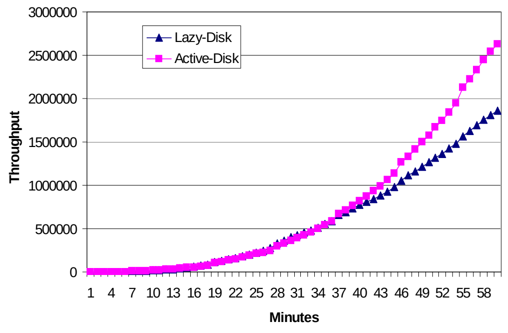

Figure 14. Lazy-Disk vs. Active-Disk, 2

As the difference of average productivity of different machines increases, then the active-disk approach can further improve the run-time query throughput compared to the lazy-disk approach. We set partitions assigned to machine m_1 (with a join rate 4) to have a small tuple range (15K), while set the partitions assigned to the other two machines (with a join rate 1) to have a large tuple range (45K). This setup further differentiates the average productivity values of machines. Having a smaller tuple range indicates a larger join factor value given the same number of input tuples (See Section 3.1). It thus further increases the number of output tuples. As expected, the active-disk approach has a major throughput improvement compared with that of the lazy-disk approach (see Figure 14). In these experiments skewed data was used.

However, as we discussed earlier, we need to control the total amount of states being pushed by the global coordinator. This is because too many pushes, more pushes beyond what is necessary, could decrease the performance. In general, in our framework we do set some maximum amount of states to be pushed. In the case of our experiments we had utilized 100 MB.

6. Related Work

Continuous query processing [1,4,5,16] is closely related work in that it applies a push-based non-blocking processing model. A variety of techniques have been investigated to address the scalability concerns of continuous query processing, including load shedding [23], operator-state purging [8] and adaptive scheduling and processing [16]. In this work, we instead focus on adapting operator states to handle the memory shortage problem.

State spill has also been investigated in a central environment. The focus of [22] is on the evaluation of different flushing strategies for a partitioned hash-join operator. Both XJoin [25] and Hash-Merge Join [17] essentially incorporate data management into their respective join algorithms. They adapt memory resident states from individual input streams to disks when memory overflow happens. As discussed in Section 3, this strategy does not work well for multi-input operators, especially in a partitioned parallel processing environment. Moreover, these strategies are designed to work in a central environment. In a distributed environment where both state spill and state relocation are necessary, new challenges arise, such as how to integrate both adaptation methods into one strategy. Our recent work [15] on state spill only, also in central context, instead focuses on the interdependencies of a query plan composed of a pipeline of stateful operators.

Distributed continuous query processing over a shared nothing architectures has recently received attention [1, 20]. In existing systems such as Aurora* and Borealis [27], and also in our prior work [21], operators are assumed to be small enough to fit completely within one single machine. Thus, the adaptation in such systems [27] focuses on balancing the load by moving complete query operators across machines. The basic unit to be adapted in these systems is at the granularity of a complete operator. While in this work, we instead investigate methods of adaptation at the granularity of operator state partitions.

Flux [20] is among the first to discuss partitioned parallel processing and its adaptations in the continuous query context. It makes use of the exchange architecture proposed by Volcano [10] by inserting split operators into the query plan to achieve partitioned processing for stateful query operators, focussing on single-input aggregate operators. [20] does not explicitly discuss coordination techniques between state spilling and state reallocation as we do in our work. Moreover, Flux focuses on single input query operators. Issues for complex stateful operators like multi-joins, such as how to organize states from different input steams, have not been addressed. As discussed in Section 5.2, our proposed active-disk solution makes proactive state spill decisions across multiple machines. This helps to improve the overall run-time throughput, as our experiments confirm. Unlike [20] based on simulation only and short 60-second experiments, our work is based on one-hour experiments conducted on a real software system.

Distributed Eddies [24] addresses run-time optimization of query plans in a distributed environment. However, [24] use a completely different tuple-level optimization approach by routing individual tuples through the different operators in different orders. They address neither partition-level state spilling nor partition-level relocation of operators' states to different machines.

Parallel and distributed query processing has been the focus of both academia and industry for a long time [3, 9, 12]. Partitioned parallel processing for complex operators such as joins has also been studied both by others [6, 19] as well as by our prior work [14]. Correspondingly, data skew handling techniques [7] have been proposed. All these provide general background for the work presented in this paper. However, they are typically studied under a traditional processing model assuming static queries. Unique properties such as push-based processing (requires non-blocking processing), little statistics about input streams at query definition time (requires adaptation at run time) and long running or even infinite streams (high demand on system resources) differentiate this work from traditional distributed processing.

7. Conclusion

In this work, we have studied the mechanisms and policies of state level adaptations that aim to overcome the run-time main memory shortage problem for long-running non-blocking queries in a distributed environment. We have proposed two strategies that integrate disk-based (state spill) and distributed (state relocation) adaptation techniques in main memory constrained environments. Note that such integration has not been carefully studied in the literature, yet it is necessary in modern cluster environments since the main memory of even a distributed system remains limited. Extensive experiments have been conducted with a working software system installed on a modern PC-compute cluster confirming the effectiveness of our proposed strategies.

8. References

[1] D. Abadi, Y. Ahmad, and et. al. The design of the Borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[2] D. J. Abadi, D. Carney, and et al. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[3] J. F. Annita N. Wilchut and P. Apers. Parallel evaluation of multi-join queries. In *SIGMOD*, pages 58–67, 1995.

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

[5] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: a scalable continuous query system for internet databases. In *ACM SIGMOD*, pages 379–390, 2000.

[6] M.-S. Chen, M.-L. Lo, P. S. Yu, and H. C. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In *VLDB*, pages 15–26, 1992.

[7] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, pages 27–40, 1992.

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

[9] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *ACM SIGMOD*, pages 9–18. ACM Press, 1992.

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

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

[12] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.

[13] W. J. Labio, R. Yerneni, and H. García-Molina. Shrinking the warehouse updated window. In *ACM SIGMOD*, pages 383–395, June 1999.

[14] B. Liu and E. A. Rundensteiner. Revisiting parallel multi-join query processing via hashing. In *VLDB*, pages 829–840, 2005.

[15] B. Liu, Y. Zhu, and E. A. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *SIGMOD*, pages 347–358, 2006.

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

[17] M. Mokbel, M. Lu, and W. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *ICDE*, page 251, 2004.

[18] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *ACM SIGMOD*, pages 110–121, 1989.

[19] D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *VLDB*, pages 469–480, 1990.

[20] M. A. Shah, J. M. Hellerstein, and et. al. Flux: An adaptive partitioning operator for continuous query systmes. In *ICDE*, pages 25–36, 2003.

[21] T. Sutherland, B. Liu, M. Jbantova, and E. Rundensteiner. D-CAPE: Distributed and self-tuned continuous query processing. In *CIKM*, pages 217–218, 2005.

[22] Y. Tao, M. Yiu, and D. Paradias. RPJ: Producing fast join results on streams through rate-based optimization. In *SIGMOD*, pages 371–382, 2005.

[23] N. Tatbul, U. Cetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.

[24] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pages 333–344, 2003.

[25] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

[26] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003.

[27] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the Borealis stream processor. In *ICDE*, pages 791–802, 2005.