Worcester Polytechnic Institute
Digital WPI

Computer Science Faculty Publications | Department of Computer Science

3-11-2005

Adapting State-Intensive Non-Blocking Queries over Distributed Environments

Bin Liu
Worcester Polytechnic Institute, binliu@cs.wpi.edu

Elke A. Rundensteiner
Worcester Polytechnic Institute, rundenst@cs.wpi.edu

Follow this and additional works at: https://digitalcommons.wpi.edu/computerscience-pubs

 Part of the Computer Sciences Commons

Suggested Citation

Liu, Bin , Rundensteiner, Elke A. (2005). Adapting State-Intensive Non-Blocking Queries over Distributed Environments. .
Retrieved from: https://digitalcommons.wpi.edu/computerscience-pubs/79

This Other is brought to you for free and open access by the Department of Computer Science at Digital WPI. It has been accepted for inclusion in Computer Science Faculty Publications by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.

WPI-CS-TR-05-04 Mar 2005

Adapting State-Intensive Non-Blocking Queries over Distributed Environments

by

Bin Liu
Elke A. Rundensteiner

Computer Science Technical Report Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Adapting State-Intensive Non-Blocking Queries over Distributed Environments

Bin Liu and Elke A. Rundensteiner
Department of Computer Science, Worcester Polytechnic Institute
100 Institute Road, Worcester, MA 01609–2280
{binliu|rundenst}@cs.wpi.edu

March 11, 2005

Abstract

Main memory is a critical resource in push-based non-blocking query processing, especially for queries with stateful operators. Works in the literature apply *partitioned parallel processing* to alleviate the stringent memory demands. However, main memory of a distributed system remains limited. Thus, there is a demand for efficient main memory usage even for partitioned parallel queries. In this work, we first investigate two adaptations, namely, *disk-based adaptation* and *distributed adaptation*, that adapt operator states when memory overflow happens for complex multi-input operators. We analyze the tradeoffs regarding the factors and polices to be used when adapting operator states to overcome memory overflow. Two approaches, namely, *lazy-disk* and *active-disk* adaptations, are proposed to integrate the disk-based and distributed adaptations when the aggregated main memory of a distributed system is not sufficient for the query processing. Both appproaches aim to maximize the overall throughput. Extensive experiments have been conducted on a working system. These experiments reveal various aspects of partitioned parallel processing and their adaptation strategies.

1 Introduction

1.1 Motivation

Non-blocking pipelined query processing with data being pushed asynchronously to the server and producing query results as data comes through, e.g., continuous query processing over streaming environments [2, 3], has been the growing research focus in recent years. Unlike static queries in a traditional database system, this processes data that is continuously arriving and outputs query results in a real-time fashion. Efficient processing of such non-blocking query is the key to the success of many applications including remote sensor monitoring [18] and network traffic management [25].

Current research of non-blocking pipelined query processing usually assumes that query operators only have small-sized operator states, i.e., small-window joins, or no state operators such as select and project [2, 14, 3, 4]. However, query operators with potentially huge operator states, such as multi-joins, have not been deeply studied in the non-blocking query processing context. Such query operators are rather common in a data integration or data warehousing environment. For example, a real time data integration system as shown in Figure 1 is critical for financial analysts to make timely decisions. Here, stock prices, volumes and external reviews are continuously sent to the integration server during the working hours (i.e., 9AM-4PM). The server is required to process these input streams as fast as it can and output the data to the decision support system. This way, analysts are able to analyze and make decisions based on up-to-dated information. Two factors are important for the integration server: (1) The ability to produce as many results as possible if not all of them when data comes through. Thus the decision-maker applications could have more information available instantly during working hours. (2) The capability to process all these data in an efficient manner to minimize the overall processing time. This would benefit other data warehouse applications that rely on the complete data. In this context, the overall processing time contains two parts, a specified *normal-run* time (i.e., 2 hours) and a *post-run* time (i.e., cleaning up disk resident states if operator states have been pushed into disks during normal-run time when memory overflow happens).

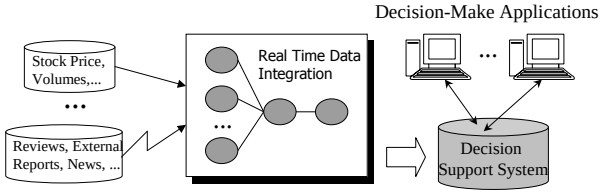


Figure 1: A Real-time Data Integration System

The stringent requirement of generating near real time results demands efficient main memory based query processing. This is because any delay in the query processing will cause the input data to accumulate in the system. This may accelerate slow down of the overall processing. However, the main memory is a limited resource. Moreover, given a long running query with a high rate of input streams, the system can quickly use up all available main memory.

This is a particularly critical challenge for data integration type queries that are complex and stateful in nature, i.e., multi-joins [1]. Thus, there is demand for efficiently using main memory during the query processing.

Previous work, mainly in the continuous query processing context, has studied techniques such as load shedding [2] and operator-state purging [11] to cope with the shortage of resources. However, these techniques are not an option if we need accurate query results. Moreover, a system may still experience resource shortage even after applying the above techniques if a certain QoS requirement has to be met.

In this work, we focus on adapting operator states to address main memory shortage during the query processing. We focus in particularly on queries having state-intensive operators, i.e., multiple-way joins [28]. These queries are common in data integration related applications as shown by Figure 1. We assume that we need accurate query results. Thus we cannot resort to techniques such as load shedding and state-purging in the processing. This implies that the stateful operator states are monotonically increasing during the process. 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.

1.2 Partitioned Processing and Adaptations

One natural solution to address the problem of main memory shortage, as discussed in XJoin [27] and Hash-Merge Join [20], is to choose memory resident states and push them into local disks when memory overflow happens. As can be seen, this type of approach is designed to delay the processing of certain states. Given a continuously high input rate of data streams (thus a monotonically increase of the operator states), it is not likely to process these disk-resident states during the execution. Thus, it requires a *clean up stage* to process these disk resident states and generate missing results. Here, we refer this process as *disk-based adaptation*.

Another solution to address the memory shortage is to distribute large stateful query operators into multiple machines and process them in parallel. This is referred as a *partitioned*

[1]Note that there are stateless operators such as select, project, or operator states are not monotonically increasing such as operators have a fixed window size (a common condition in continuous query processing). However, we are more interested in large stateful operators in this work.

parallel processing [22, 13, 24]. In such a distributed environment, when only a subset of machines get overloaded, we can choose states from the overloaded machine and send them over to a less loaded machine. For simplicity, we call this type of adaptation *distributed adaptation*. The potential advantage of this distributed adaptation is that the adapted states remain active in the main memory once the adaptation is completed. However, this type of adaptation may not solve the memory shortage problem since the aggregated main memory of multiple machines is still limited.

1.3 Contributions

Literature in continuous query processing thus far studied these two types of adaptations separately [27, 20, 24]. In this work, we argue that both adaptations need to be considered in an integrated manner for a realistic distributed processing. This is because a disk-based adaptation may not be that efficient, while a distributed adaptation alone may not fully resolve the problem of memory shortage. We analyze the tradeoffs regarding the factors and policies to be considered when adapting states of multi-input operators to overcome the main memory shortage. Two adaptation approaches aim to maximize the overall *throughput*, namely, *lazy-disk* and *active-disk*, are proposed that integrate both disk-based and distributed adaptations when the aggregated memory of a distributed system is still not sufficient for the query processing. Extensive experimental evaluations have been conducted on a working software system confirming the effectiveness of our proposed adaptation approaches by comparing it to both the disk-based and distributed adaptations in isolation.

2 Related Work

Continuous query processing [2, 3, 19, 6, 5, 29] is closely related to our work in that it applies a push based non-blocking processing model. Continuous query processing also faces scalability concerns due to high rates of inputs and possibly infinite data streams. A lot of techniques with different research focuses have been investigated to address this problem, such as load shedding [2], operator-state purging [11] and adaptive scheduling and processing [4, 19]. While in this work, we focus on adapting the memory usage for complex stateful query operators

in a distributed environment with possible huge volumes of states. This issue has not been carefully addressed in the continuous query processing literature yet.

Distributed continuous query processing over a shared nothing architecture, i.e., a computing cluster, has been investigated in the literature to address the resource shortage and scalability concerns [1, 26, 24]. In existing systems such as Aurora* [8] and Borealis [1], operators are assumed to be small enough to fit completely within one single machine. Thus, their main focus is how to distribute the query plan over multiple machines while treating each operator as one atomic unit. The adaptation in such systems [30] mainly focuses on balancing the load. The basic unit to be adapted in the system is always at the granularity of one complete operator. D-Cape [26] is another system that distributes and adapts continuous queries at an operator-level. In this work, we instead investigate methods of adapting operator states to optimize the main memory usage.

Flux [24] the first work in the literature to discuss the partitioned parallel processing and the distributed adaptation in a continuous query processing context. It makes use of the exchange architecture that was proposed by Volcano [13] by inserting split operators into the query plan to achieve partitioned processing for large stateful query operators. However, Flux does not address that how disk-based adaptation could be seamlessly incorporated into the system. As we motivated earlier, a pure distributed adaptation may not fully solve the memory shortage problem. Moreover, it only discusses single input query operators. Given complex stateful query operators such as multiple-way join, more issues such as how to organize states from different input steams need to be addressed even for a pure distributed adaptation.

Disk-based adaptation for non-blocking query operators has also been investigated in the literature. As discussed above, both XJoin [27] and Hash-Merge Join [20] adapt memory resident states from individual input streams to disks when memory overflow happens. As we will discuss later, this strategy does not work well for multiple input query operators, especially in a partitioned parallel environment. Moreover, these strategies are designed to work in a central environment. In an environment where both disk-based and distributed adaptations are necessary, again, issues such as how to integrate them need to be considered.

Parallel and distributed query processing has been the focus of both academia and industry for a long time [9, 12, 15]. Partitioned parallel processing, especially for complex operators such

as joins, has also been studied in [23, 7, 17]. Correspondingly, data skew handling techniques [10] have been proposed. All these works provide the necessary background for the distributed continuous query processing and its forms of adaptations. However, they are typically studied under a traditional database processing model assuming static queries. Unique properties such as push-based processing (requires a non-blocking pipelined processing), little statistics about input data streams at query definition time (requires adaptation at run time) and long running or even infinite data streams (high demand on the system resources) differentiate this work from traditional distributed and parallel query processing.

To the best of our knowledge, this is the first paper aiming to incorporate two types of adaptations at the operator-state level for complex non-blocking queries having multiple inputs operators, i.e, multi-way joins. As we motivated in Section 1.1, both adaptations are critical in cases when the aggregated main memory of the distributed system is still not sufficient for the query processing task.

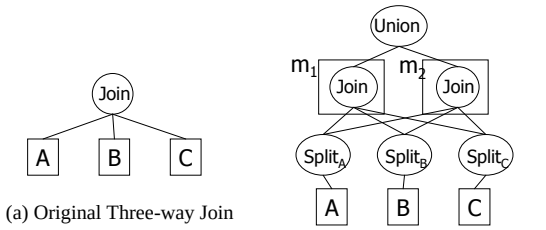
3 Partitioned Query Processing

We assume that the stateful operators in the non-blocking query plan are so large that they cannot fit in main memory of one single machine. As a result, partitioned parallelism [13, 24] that aims to partition and run one single operator in multiple machines will be applied to these high-volume state operators. Throughout this work, we use a symmetric multiple-way hash join query operator [28] as a representative example of such a large stateful operator. Other complex multiple input operators can also be addressed in a similar manner as long as they can be distributed to multiple machines with each machine only processing partitions of inputs.

3.1 Partitioning Large Stateful Operators

The first question need to be addressed is how to achieve such a partitioned parallel processing for complex multi-input operators. As discussed in [13, 24], we insert a *split* operator in front of each input stream of the operator to be partitioned. This split operator partitions each input stream and sends partitions to each machine. For simplicity, we will henceforth refer to the operator that is running in a particular machine as an *instance* of the partitioned operator.

For example, we assume to 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. Here A, B, and C denote the input streams as well as join relations of the query. A_1, B_1, and C_1 are join columns of relations A, B and C respectively. As shown in Figure 2(b), to partition the query plan on two machines, we insert one split operator for each input stream. 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) Original Three-way Join

(b) Partitioned Three-way Join

Figure 2: Example of Partitioned Processing

Thus, each operator instance only processes non-overlapping partitions of the input stream (a portion of the whole input data). A *union* operator, if needed for result merging, is inserted into the query plan after the partitioned operator. That is, it is inserted into the output streams of all instances of the partitioned operator to combine the results for further processing.

Such partitioned processing has advantages over centralized processing. Clearly, more resources are available to the query processing given multiple machines are involved in the computation.

3.2 Initial Distributions and Connections

An initial distribution, referred as distributing query operators (plans) to multiple machines, has to be established given no available statistics information about the operators and input streams before the execution. Here, we design an initial distribution algorithm that is simply based on the operator type. Basically, we have the complex query operator (the operator

[2]Note that for multiple-joins that defined on different columns, input streams cannot be split based on the same partitioning function. In that case, more data structures would be needed to keep the operator states. The discussion of this is out of the scope of the paper. We assume the partitioned processing and corresponding split operators have been figured out.

with possible large volume of states) assigned to all available machines. We then distribute the rest of query operators, such as split and union, using a Round-Robin algorithm to all these machines. We have each machine had a full copy of the original query plan. All the query operators in each machine are deactivated by default initially, and we only activate the operators that are assigned to the machine in its query plan copy. The query plan then will be connected together based on the distribution algorithm.

For example, Figure 3 illustrates the distribution of the three-way join query over 4 machines. As discussed above, all 4 machines have a full copy of the query plan. Here, the dark operator represents that the operator is activated in the machine. Thus, each machine will have one join operator and one light states operator assigned as shown in Figure 3 (b).

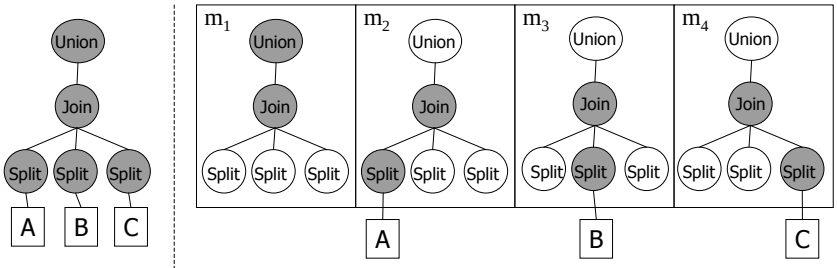


Figure 3: Partitioned Query Plan Distribution

The query operators that assigned to different machines need to be connected together based on the query plan. Basically, if two operators that have producer-consumer relationship are activated in the same machine, we then use the memory-based queue to connect them together. If these two operators are assigned to different machines, we then create a Socket connection to connect them. For example, the connection of the initial distribution as described in Figure 3 (b) can be shown by Figure 4. Here, the solid lines represent the connections between operators, while the dashed lines are no longer connected. Note that the output connections of the split operator are created dynamically based on the number of machines are assigned to the down stream operator, the 3-way join operator in this case. If we change the distribution, the connections also have to be changed correspondingly.

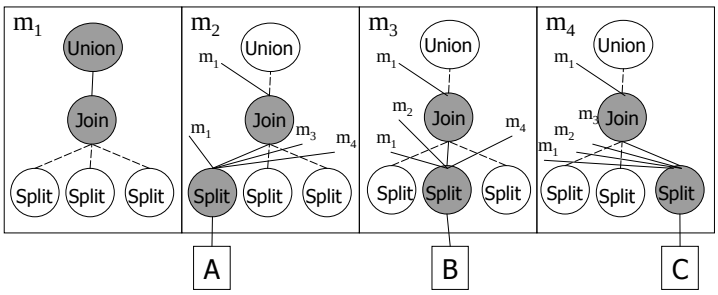


Figure 4: Partitioned Query Plan Connection

3.3 Experimental Environment Description

The experiments are conducted based on the D-Cape system developed at WPI. The overall system architecture is described in Figure 5. We have a dedicated *distribution manager* in charge of a set of *query processors*. The distribution manager distributes the query plan and connects operators together. It collects and analyzes running statistics of each query processor. It makes *global adaptation decisions* such as moving states from one query processor to another. The core part of each query processor has a Cape continuous query engine [21] installed. The Cape engine takes care of executing the continuous query plans (operators) assigned to it. Each query processor also has modules to collect running statistics, receive and distribute tuples, and make *local adaptation decisions* such as choosing operator states to be adapted.

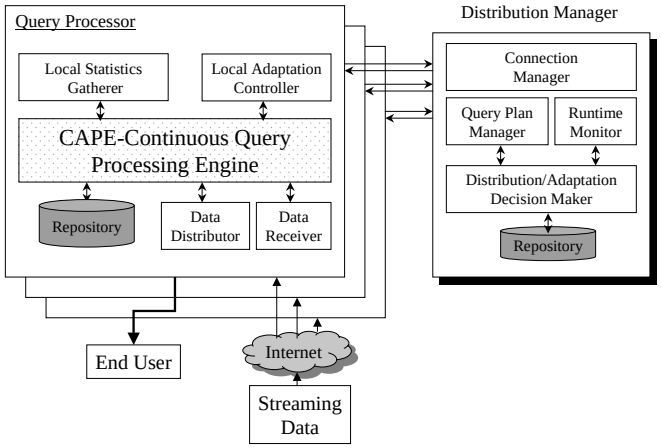


Figure 5: D-Cape System Architecture

The system is deployed on a 10-machine cluster. Each machine in the cluster has dual 2.4Hz Xeon CPUs with 2G memory. These machines are connected via a private gigabit ethernet.

We choose three of them to run the distribution manager, stream generator, and application server respectively. The stream generator continuously generates stream input tuples for query to process, while the application server processes the output results of the query plan. All the other machines are deployed as query processors as necessary for the given experiment.

3.4 Experimental Data Sets and Queries

We use a three-way symmetric hash join query plan as described in Figure 2(a) to report our experimental results in the following sections [3]. The join is defined on the first column of each input streams. We partition each input stream into 300 partitions based on its join column value. We use synthesized data in our experimental studies to be able to control various factors of the input streams. In this work, we use the following two factors to control the generation of input streams. We use term *tuple range* to indicate the possible number of tuples with distinct join values of an input stream. We define *range join ratio* as the number of result tuples that are expected to be produced given one input tuple within one range of tuples. For example, assuming the tuple range is 30K, while the range join ratio is 2. Then, within the first 30K tuples being processed, a tuple received is expected to produce 4 result tuples. While in the next 30K tuples, it is expected to produce 16 result tuples. Note that this range join ratio equally applies to the partitions of input stream.

3.5 Partitioned Parallel Performance

The performance of such partitioned parallel processing needs to be conducted first. The performance analysis is divided into two parts. One, how it performs given a 'small' query plan that could be processed by a single machine? Two, how the performance changes given the query plan is beyond the processing capability of one single machine?

Figure 6 shows the performance changes of partitioned parallel processing for a 'small' query plan. In this experiment, stream generator sends one tuple per 20ms on average for each input. The tuple range is set to 50K, while the join ratio is set to 1. The query plan runs over 40 minutes. The data points displayed in Figure 6 are the average of three runs.

[3] As we will discuss shortly, the adaptation policies we developed are based on main memory usage and the output rate, they are insensitive to the query plan itself. We have worked with other query plans and they show similar results.

Here, we set memory of the query processor large enough to run the query plan completely in main memory. We then partition query plan and run it on multiple machines. The results are shown in Figure 6 (the i in M_i indicates the number of machines). The X-axis represents the minutes that have been run, while the Y-axis denotes the overall throughput of the query plan correspondingly. From Figure 6, we can see that there is negligible effects on the overall throughput when we assign more machines to this query plan. This result seems contradictory to the traditional distributed and parallel processing at first sight since more resources usually result in improvement in the overall performance. However, this is because our main focus in this work is on memory usage due to the operator states. We assume that the query processor has enough CPU processing capability to keep up with the data input rates. That is, no input data will be accumulated in the query processor input queue. Given that, adding extra memory beyond the maximally needed would not help the performance.

This experiment shows that the above partitioned parallel processing only adds neglibile overhead to the continuous query processing. However, this partitioned processing for small query plans over a large number of machines is not necessary. This is because the split operator has to partition and send input data to all the machines. This eventually incurs overhead in the query processing. Results in Figure 6 also shows that the throughput of 'M5' has a slight drop compared with the throughput of 'M1'.

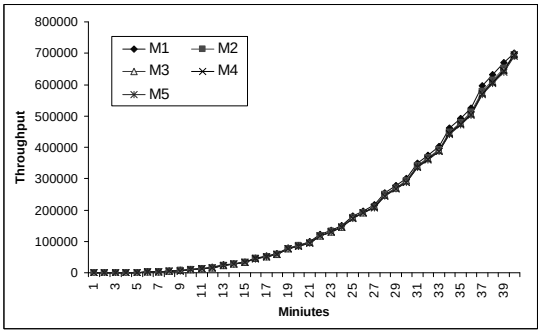


Figure 6: Partitioning Small Query Plan

Figure 7 shows how the partitioned parallel processing affects the overall performance for 'large' query plans. We change the data stream generator so that it generates one tuple per 10ms on average. Moreover, we also set a join ratio of 2 to have the query plan generate more output tuples. We set the maximal main memory of each query processor to 200MB. From

Figure 7, we see the query only runs for about 25 minutes given one single machine. Then it stops due to the out of main memory exception. If we deploy this query plan on two machines, it runs about 35 minutes before main memory overflows. However, as we discussed above, once we have enough resources to run the query, adding more machines will not improve the overall throughput. In this experiment, we see that 'M_4' and 'M_5' have almost the same throughput over the 40 minute run.

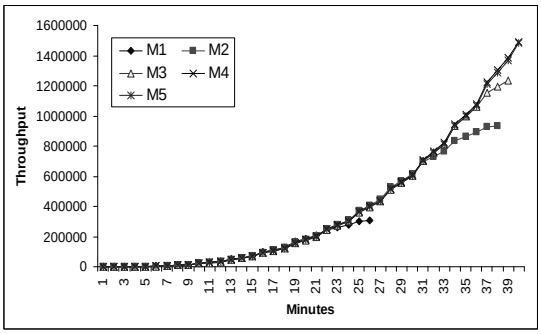


Figure 7: Partitioning Large Query Plan

4 Disk-Based Adaptation

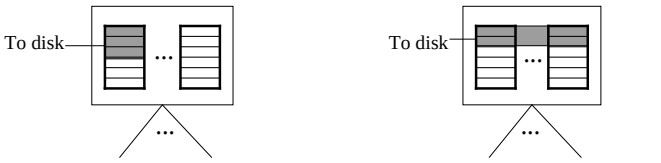
4.1 State Partitions and Partition Groups

Disk-based adaptation selects memory resident states and pushes them into disks when memory overflow happens. Given a monotonic increase of memory usage during the execution (normal-run time) (this could especially occur for the long running integration queries as motivated in Section 1), these pushed states will be kept in disk (inactive) until memory overflow has been addressed, i.e., at the end of the query processing.

To facilitate the run time adaptation, we divide the input streams into a much larger number of partitions than the number of machines, e.g., 500 number of partitions over 10 machines. This enables us to choose the appropriate partitions to adapt at run time, and to avoid repartitioning during the adaptation. This method has first been applied in early data skew handling literature such as [10] as well as recent partitioned continuous query processing work Flux [24]. Each input partition is identified by a unique partition ID. We also organize operator states based on the input partitions. That is, all the operator states produced from tuples of a

particular partition will be identified by the ID of the input partition. For simplicity, we also use the term partitions to refer the corresponding operator states if the context is clear.

For a single input query operator, as tackled in Flux [24], no other options are available except selecting state partitions to adapt. These states are from the same input stream. However, for a multiple-input operator, there are partitions from different inputs in the operator states with the same partition ID. Thus, multiple ways of organizing partitions are possible [4]. First, we could choose partitions from one input at a time as shown in Figure 8(a). This strategy has two drawbacks in a partitioned processing environment. (1) It may increase the complexity in the clean up stage, for example, it would be complex to find the corresponding partitions in the other inputs. Things will get worse if the partitions with the same partition ID have been pushed more than one time. This requires us to keep timestamps when eliminating the duplicates in the clean up stage. (2) The partitioned parallel processing performs better when partitions with the same partition ID from different inputs are being processed locally in the same machine. If we adapt partitions from individual inputs to other machines, this amounts to query processing over remote machines. The second option, illustrated in Figure 8(b), is to group the partitions with the same partition ID as one whole unit of adaptation granuality. For simplicity, we call n partitions with the same ID from n different inputs one *partition group*. Thus, one partition group will be the smallest unit to be adapted. This strategy greatly simplifies the clean up stage as we will discuss it shortly. It also avoids the expensive processing of queries over across multiple machines. Without loss of generality, we may use term partition to refer partition group if the context is clear.



(a) Select states from one input only (b) Select all states with the same ID

Figure 8: Choosing Partitions

[4]Note that we discuss the possible smallest unit when choosing partitions in the adaptation here. The strategy of selecting partition units to adapt will be described later.

4.2 Clean Up Disk Resident Partitions

Disk resident states have to be brought back to main memory to be combined with the memory resident part of states to produce missing results. We call this process the *clean up stage*. Note that this clean up stage can be performed once main memory overflow has been addressed during the execution. In the clean up stage, we should produce all missing results due to these disk resident states while prevent duplicates.

As discussed above, we use a partition group as the unit when choosing operator states to put into disks. For a partitioned multiple-way hash join operator, the output will only be generated from the partition group. Given that, the tasks that need to be performed in the clean up stage can be described as follows: (1) Organize the disk resident partition groups based on their partition ID. 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 would accumulate new partitions with the same partition ID in main memory. Later, partitions with the same ID could be pushed into the disk again. (2) Merge partition groups with the same partition ID and generate missing results. Since main memory resident partition group with the same ID may exist, we thus need to merge this memory resident part with all these disk resident ones.

The merge of partition groups with the same ID can be described formally as follows. We use a subscript to indicate the partition ID, while we use the superscript to identify the partition groups with the same partition ID that have been pushed at different times. For example, let us assume the query is defined as $A \bowtie B \bowtie C$. There are k times that the partition group with partition ID i have been pushed, represented as (A_i^1, B_i^1, C_i^1), (A_i^2, B_i^2, C_i^2), ..., (A_i^k, B_i^k, C_i^k) respectively. Here, (A_i^t, B_i^t, C_i^t), $1 \leq t \leq k$ denotes the t-th time that the partition group with ID i has been pushed into the disk. Note that the results generated from each partition group have been been produced. In this case, all the results such as $A_i^1 \bowtie B_i^1 \bowtie C_i^1$, $A_i^2 \bowtie B_i^2 \bowtie C_i^2$, ..., $A_i^k \bowtie B_i^k \bowtie C_i^k$ have been generated. For ease of description, we denote these partition groups by $P_1, P_2, \ldots, P_k$, while we denote the results generated from each partition group as V_1, V_2, ..., V_k.

Merging two partition groups with the same partition ID results in a combined partition

group with the operator states from both partition groups. For example, the merge of P_1 and P_2 results in a new partition group P_{12} now having the operator states $A_i^1 \cup A_i^2, B_i^1 \cup B_i^2, C_i^1 \cup C_i^2$. Note that the final output V' should have $(A_i^1 \cup A_i^2) \bowtie (B_i^1 \cup B_i^2) \bowtie (C_i^1 \cup C_i^2)$. We must generate the missing part $\Delta V = V' - V_1 - V_2$ in the clean up process for these two partition groups.

The incremental view maintenance algorithm as described in [16] can be applied to compute the missing results. In this example, if we assume A_i^1, B_i^1, C_i^1 as the base state, while treat A_i^2, B_i^2, C_i^2 as the incremental changes. Thus, the part $V' - V_1$ can be computed by Equation 1. By further removing V_2, we generate exactly the missing results of P_1 and P_2. A more general case given n way joins can also be computed in a similar way.

$$\begin{aligned} V' - V_1 &= A_i^2 \bowtie B_i^1 \bowtie C_i^1 \\ &\cup \ (A_i^1 \cup A_i^2) \bowtie B_i^2 \bowtie C_i^1 \\ &\cup \ (A_i^1 \cup A_i^2) \bowtie (B_i^1 \cup B_i^2) \bowtie C_i^2 \end{aligned} \tag{1}$$

This process can be applied iteratively until all partition groups with the same partition ID have been merged. As a final step, we need to combine these disk resident partitions with the corresponding memory resident parts. The merge process is the same as we just discussed.

4.3 Throughput Oriented Disk Adaptation

A strategy needs to be devised for which partition groups to push into disks. Different partitions to be pushed may have different impact on the overall performance. We propose a throughput-oriented disk-based adaptation strategy that aims to keep the throughput of the query as high as possible. That is, we want to generate as many output results as possible even part of memory resident states are pushed into disks (no longer active). Ideally, given a high overall throughput in the normal-run time, it also reduces the efforts in the clean up stage as more work would have been completed.

The intuition behind our throughput-oriented disk-based adaptation can be summarized below. (1) To identify the partitions that are least being used, and push them into disks. (2) If all partitions are frequently being used, choose partitions that have a low output rate as candidates to be pushed. Thus, memory space is left for active and high output rate partitions

that could potentially generate more output results.

Our design of composing partition groups helps to achieve the above thoughput-oriented strategy. In each partition group, we record the last time when a input tuple being inserted into the hash table, denoted by T_u. We record the current size of each partition group, represented by P_s. We also record how many tuples have been generated from this partition group, denoted by P_o. We use term *inactive time*, defined as the current time T_c minus T_u, to indicate the activity of the partition group. If the inactive time is larger than a threshold θ_t, we then choose the partition group as the candicates to be pushed. This is similar to the commonly used LRU strategy. We define the *productivity* of each partition group as P_s/P_o. Given a similar size P_s, a large P_s/P_o value indicates that only few output results have been generated so far. We thus prefer to choose the partitions with large P_s/P_o value to adapt into the disk [5]. Thus, the partitions left in the main memory are likely to produce more results than the ones have been pushed into disks.

4.4 Experimental Evaluation

In the first experiment, we aim to answer how much state is best to be pushed into the disk when main memory gets overloaded. We run the query plan on one single machine in the cluster for over 50 minutes. The input tuples are set to arrive at a rate of every 30ms on average from each stream. The tuple range is set to 30K, and the join ratio of each partition is set to 3. In this experiment, the disk-based adaptation is triggered when the memory usage of the machine is over 200MB. If not specified explicitly, we set the inactive time threshold θ_t equal to half of the total execution time in our experiments.

Figure 9 depicts the overall throughput with different percentages of states being pushed when overload happens. A 0%-Push denotes that all states are kept in main memory, 10%-Push means to choose 10% of all main memory states and put them into disks, and so on. We randomally choose partitions from the operator's states. Seen from Figure 9, the more states

[5]Note that both the $T_c - T_u$ and P_s/P_o values of each partition group only reflect the input data that has been processed so far. It keeps on changing when new data gets processed. In this work, we assume that the value we observed so far could indicate the trends of behavior of the partition group. Given a more dynamic environment that the properties of input data keeps on changing, we would apply other techniques to predict the productivity of each partition group. The discussion of these prediction techniques are beyond the scope of this paper.

are being pushed into the disk, the smaller the overall throughput. This is as expected since the states being pushed are no longer active.

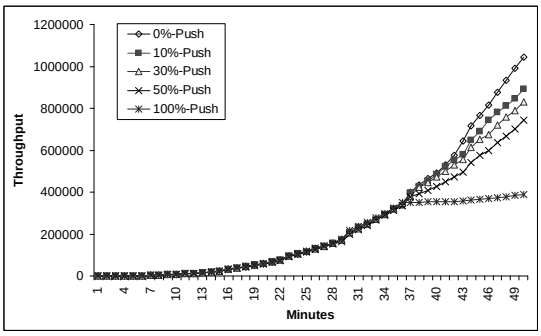


Figure 9: Percentage of States being Pushed

Figure 10 shows the corresponding memory changes when pushing different percentages of states. Here, the memory is projected based on the memory usage of all operators and queues in the machine. From Figure 10, we see that the main memory usage can be effectively controlled by the adaptation. We also see that the more states we push each time, the fewer we need to trigger the disk-based adaptation.

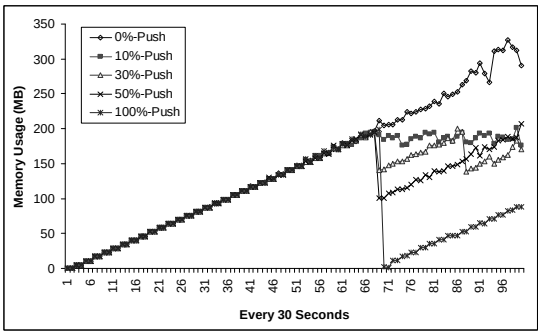


Figure 10: Memory Usage vs. Percentage Pushed

If not specified explicitly, we will use the 30%-push for the disk-based adaptation in the following experiments. This is because it balances the main memory usage (number of pushes) and the overall throughput.

However, given a fixed amount of states (percentage) to be pushed, the question still remains which partition groups to choose to be pushed. Figures 11 and 12 show the impact of choosing different partition groups on the overall performance. In Figure 11, we set 1/3 of the partitions to have an average join ratio of 4, 1/3 of partitions to have a join ratio of 2, while the rest of

the partitions have a join ratio of 1. We compare our throughput-oriented strategy to the other end of the spectrum. Here, the 30%-Low-OutputRate line represents the pushing of partition groups with large P_s/P_o (less productive partitions) values. As comparison, the 30%-High-OutputRate denotes the pushing of partitions with smaller P_s/P_o values instead. Seen from Figure 11, the 30%-Low-OutputRate strategy has a much higher throughput. This is because leaving the partition groups with high output rate in main memory is more likely to generate more output results as input tuples come through.

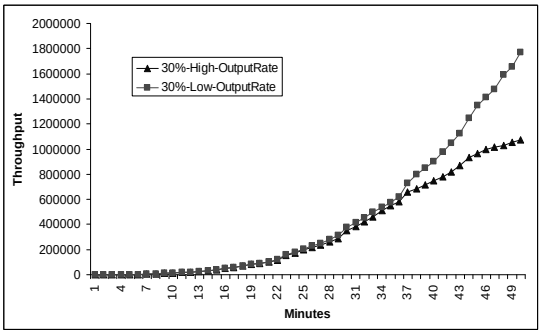


Figure 11: Various Join Ratio Paritition s

A high overall throughput in the execution (normal-run) time also helps to reduce the clean up stage efforts. This is because more work have been done before the clean up stage. In the above experiment, the 30%-Low-OutputRate strategy uses 26,879 ms to generate 194,308 tuples in the clean up stage, while the 30%-High-OutputRate one generates 992,893 tuples using around 359,396 ms.

Even in cases when the join ratios of different partitions are similar, the throughput-oriented strategy may still help. Figure 12 depicts the overall throughput when we set the join ratio of all partitions to 3 (by a uniform distribution of join column values). As can be seen, the 30%-Low-OutputRate strategy still slightly outperforms the 30%-High-OutputRate.

5 Distributed Adaptation

5.1 Adapting States Across Machines

Uneven workload may arise among machines in a distributed processing environment. Thus, while one machine runs out of memory, another machine in the system may still have memory

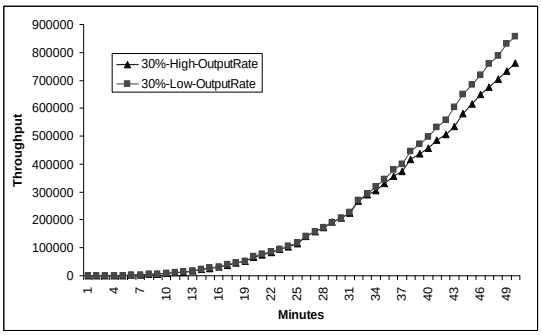


Figure 12: Even Join Ratio Partitions

space for more operator states. Having operator states stay in main memory would not affect the overall query processing. Thus, such distributed adaptation is a good choice when we have sufficient memory.

We first provide further support for our decision to use partition groups as the basic unit, especially in the distributed adaptation. For example, we assume that a three-way join $A \bowtie B \bowtie C$ is deployed on 2 machines. Each input stream is partitioned into 500 partitions, with each machine processing half of them. For example, machine 1 processes partitions with ID from 1 to 250, while machine 2 processes partitions with ID from 251 to 500. We now decide to move partition with ID 1 from machine 1 to machine 2 at run time. If only state A_1 is moved to machine 2 while B_1 and C_1 remains in machine 1. Then all the following inputs that belong to partition 1 have to be processed using a distributed join process in two machines. In this case, a tuple from stream A (belongs to A_1) will first probe machine 1 (B_1 and C_1) to produce join results. After that, the tuple is sent to machine 2 for hashing into A_1. As can be seen, such across-machine processing can be very expensive. Instead, if we move the partition group (A_1, B_1, C_1) from machine 1 to machine 2, then all the subsequent tuples with partition ID 1 would be directly sent to machine 2. Thus, all the processing related to joining tuples in partition 1 is simply moved from machine 1 to machine 2.

Distributed adaptation is a global process since it requires knowledge from other machines to make an adaptation decision. As illustrated in Figure 5, the distribution manager in our system gathers running statistics of each machine including main memory usage. Once the distribution manager observes an accelerated uneven load (i.e., memory usage) among the machines, it initiates the distributed adaptation process.

The distributed adaptation starts when the distribution manager finds the difference between the maximal amount of memory used (referred as M_m) and the least amount of memory used (referred as M_l) reaches a certain threshold (θ_d). For example, $M_l/M_m < 0.8$. We then move $(M_m - M_l)/2$ amount of states from the most used machine to the least used one. For simplicity, we call the machine with the most used memory the *sender*, and the machine with the least memory usage the *receiver*.

The distributed adaptation protocol (moving operator states between the receiver and the sender at run time) is composed of 4 sets of actions with a total of 8 steps, as illustrated in Figure 13.

The first set is to compute what partitions to move, as depicted by Figure 13(1). It has two steps. In **step 1**, the distribution manager sends a *ComputePartitionsToMove* message to the sender. This message is used to tell the sender that the amount of operator states that will be moved out. Then, in **step 2**, the sender calls the local adaptation manager to find the actual partitions that will be moved. After that, the IDs of the partition groups to be moved are returned to the distribution manager via a *PartitionsToMove* message.

Once the partitions to be moved have been computed, we need to stop sending inputs to these partitions and make sure all on-the-fly tuples have been processed before moving the operator states. These steps are performed in the second set of actions, deactivating partition inputs, as illustrated by Figure 13(2). That is, in **step 3**, after receiving the *PartitionsToMove* message from the sender, the distribution manager sends a *DeactivatePartitions* message to the sender and the machines where split operators are activated with the partition IDs to be moved. After receiving the message *DeactivatedPartitions*, the sender is tiggered to start checking whether the flag *End-Of-Moved-Partitions* from split operators have been received. While the machine on which split operator(s) is running notifies the split operator to perform the following two tasks. (1) The split operator stops sending the tuples with IDs that will be moved. Then it sends out a special tuple (*End-of-Moved-Partitions*) to the sender to indicate that no more tuples will arrive belonging to the partitions that are to be moved. (2) The split operator creates a temporary space for storing the following inputs that are to be moved. After that, the split operator sends the *Deactivated* message back to the distribution manager in **step 4**. Note that once the sender receives the *End-of-Moved-Partitions* tuples from all

split operators, it denotes that all on the fly tuples have been processed.

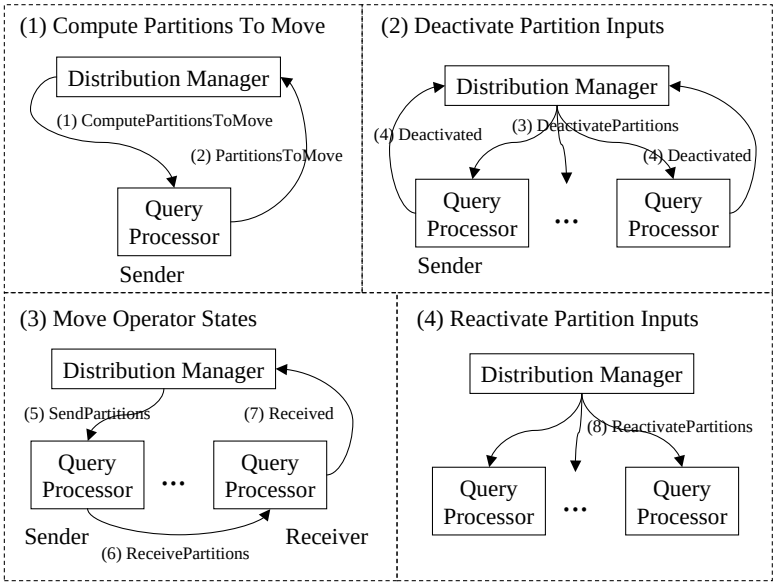


Figure 13: Distributed Adaptation Protocols

In **step 5**, the distribution manager sends a *SendPartitions* message to the sender after it receives *Deactivated* messages from all machines with split operators. In **step 6**, the sender prepares to move partitions after receiving the *SendPartitions* message. The sender has to wait until the *End-of-Moved-Partitions* messages from all split operators have been received to make sure all on-the-fly tuples have been processed. Then, it sends a *ReceivePartitions* message to the receiver with the actual partitions. While in **step 7**, the receiver gets the partition groups when it receives the *ReceivePartitions* message. The receiver returns a *Received* message back to the distribution manager after that. In **step 8**, the distribution manager sends the *ReactivatePartitions* message to machines where the split operators are running. It is used to notify the split operator to resume the normal process. The split operator will now send the tuples with partition IDs that just moved to the receiver machine. Note that the split operator will first process the data in its temporary space, if not empty.

5.2 Performance of Distributed Adaptation

The question now arises for what are the costs for distributed adaptation. Here, we investigate the following two questions: (1) when we could start distributed adaptation, i.e., the difference

between maximally used memory (M_m) and the minimal used memory (M_l). (2) How often we could run the distributed adaptation, i.e., the minimal time-span between two distributed adaptations without negatively affect the overall performance?

Figures 14 and 15 aim to answer the above questions. In this setup, the query is partitioned into two machines. Each machine processes about half of all partitions. We set up the input streams to have each machine alternatively change its rate of memory usage. For example, partitions assigned to machine 1 get 10 times more tuples than that of machine 2 for the first five minutes. After that, the machine 2 will get 10 times more tuples than that of the machine 1 for the next 10 minutes, and so on. Thus the main memory usage of these two machines changes alternatively. In this extreme environment, the distributed adaptation may keep on moving states among two machines. We set the main memory of each machine large enough to have the query completely run in main memory even without any adaptation.

In Figure 14, a k%-difference strategy means to start distributed adaptation whenever $M_l/M_m <$ k%. A high percentage indicates that a larger number of adaptations are triggered with each adaptation only moving a small amount of states. The minimal time-span of two adaptations in this experiment is set to 45 seconds. Seen from Figure 14, the overall throughput of these different strategies are almost the same. And all of them are also close to the pure main memory processing without any adaptations. This implies that the cost of distributed adaptation is low. It thus has the potential to perform such distributed adaptations frequently without impacting the overall performance. In this 60 minutes running, the 90% strategy conducts a total of 24 adaptations. While a 50% strategy only has 2 times. This is as expected.

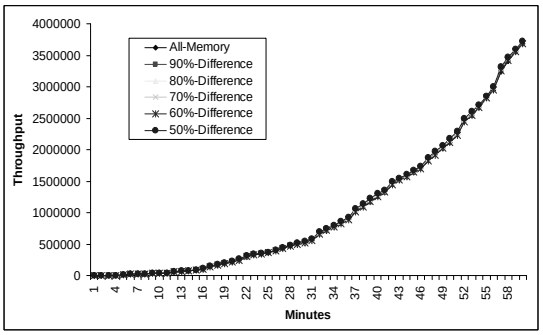


Figure 14: Percentage to Start Adaptation

Figure 15 shows the impact of changing the minimal time-span between two adaptations. In this setup, we use the 90%-difference strategy which has a large number of adaptations. We change the minimal time-span from 15 seconds, 30 seconds, to 45 seconds. From Figure 15, we again see that the overall throughput also does not change too much. In this run, the 'Every 15s' line performs 31 adaptations, the 'Every-30s' line has 27 adaptations, while the 'Every-45s' line has 24. This also confirms that the cost of distributed adaptation in our environment is rather low. From Figures 14 and 15, we can see that such distributed adaptation will not incur significant overhead on the query processing even if the adaptation may not be necessary.

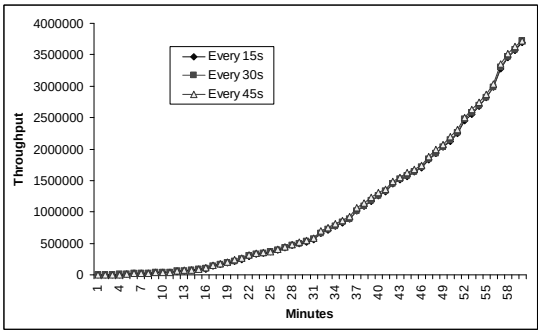


Figure 15: Impact of Minimal Time-span

Figure 16 shows the corresponding memory usage changes for the '90%-Difference' and 'Every-15s' strategy. The 'No-Distributed-Adapt-M1' and 'No-Distributed-Adapt-M2' show the memory usage if no distributed adaptations are applied. As can be seen, the memory consumption alternatively changes due to our input data pattern. Lines 'Distributed-Adapt-15s-M1' and 'Distributed-Adapt-15s-M2' show the memory usage after adaptation. We can see the main memory usage is balanced by the distributed adaptation.

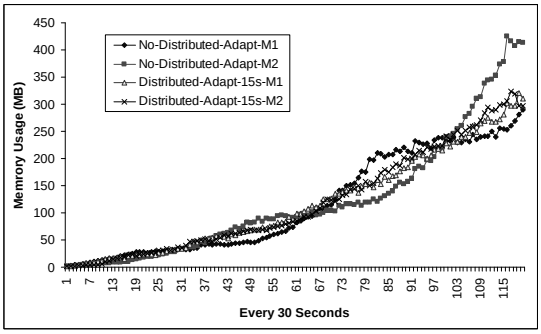


Figure 16: Memory Usage Changes

Applying distributed adaptation has the potential to achieve full main memory based processing if the aggregated main memory is sufficient for the query processing. It thus results in a much high performance. Figure 17 illustrates the experimental results of the benefits of distributed adaptations. The query is run over three machines. We change the initial distribution of partitions to make one machine have more partitions than the other two. We apply the '80%-difference' and 'Every 45s' strategy in the distributed adaptation.

Seen from Figure 17, the overall throughput of the 'No-Distributed-Adaptation' strategy drops after running for 40 minutes. This is because the main memory of the machine having most of the partitions overflows and starts pushing states into disks. While the 'Distributed-Adaptation' strategy adapts these states to other two machines, then all these states are still kept in main memory. Thus, it generates output continuously instead of waiting until the clean up stage.

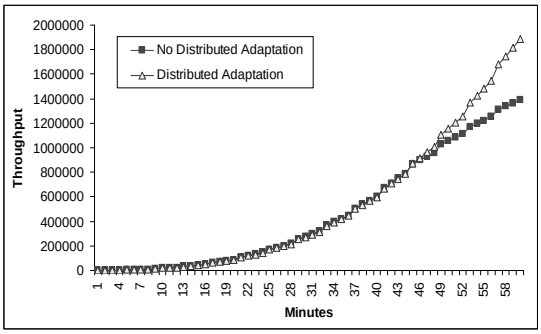


Figure 17: Distributed vs. Disk Adaptation

6 Integrating Disk-based and Distributed Adaptations

In case of the aggregated memory of all machines are still not sufficient for the query processing, then the disk-based adaptation cannot be avoided even by adapting states across machines. This is because some machines (or even all machines) in the cluster get memory overflow.

In this extreme yet practical situation, we propose two strategies to combine both disk and distributed adaptations aiming for maximal overall throughput in the query execution. The first solution, called *lazy-disk* approach, is to postpone the disk based adaptation until there is no main memory in the cluster that can hold the states from the overloaded machine. That

is, when memory usage of one individual machine is beyond the given threshold, then the disk-based adaptation is triggered on that particular machine. While the distribution manager observes that the difference between the maximal used memory and the least used memory reaches certain threshold, then the distributed adaptation is started to balance the memory usage among machines. Thus, it could have more states kept in main memory. Note that two restrictions have been added when mixing these two types of adaptations: (1) These two adaptations are not concurrent, which means only one adaptation will be processed at a time. (2) The distributed adaptation will not choose partitions that have been pushed into disks to adapt. This is to avoid unnecessary overhead and complexity in the clean up stage.

The interactions in this lazy-disk approach can be described by the sudo code sketched in Figure 18. That is, the adaptation decisions are made separately. The distribution manager starts distributed adaptation when the $M_l/M_m < \theta_d$. While the disk-based adaptation is triggered for each individual machine if $M_u/M_a > \theta_m$. Here, M_u denotes the main memory has been used so far, while M_a represents the available memory of the machine. The *nodiskAdaptation* is set by the sender's machine in the distributed adaptation process to make sure no disk-based adaptation will be processed in that machine if its local adapter is computing the partitions to be moved across machines. The *indiskAdaptation* is set when the disk-based adaptation starts. It will force the sender's machine in a distributed adaptation to wait until the disk-based adaptation finishes.

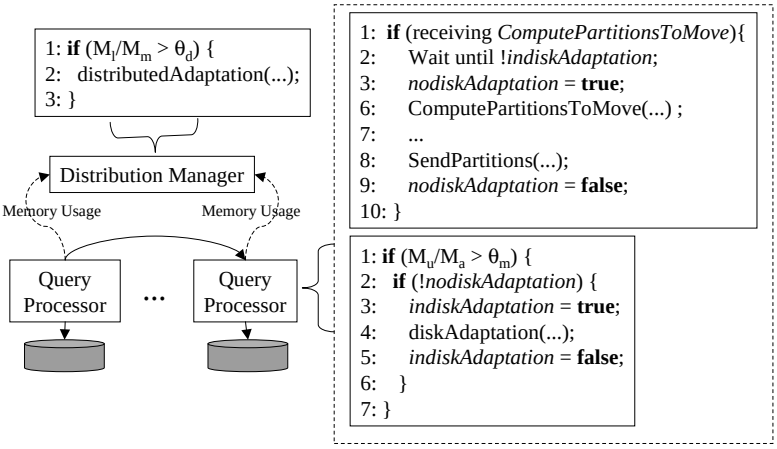


Figure 18: Lazy-Disk Adaptation Approach

As can be seen, this lazy disk approach focuses on the main memory usage only since the

adaptation is driven purely by main memory usage. Note that in this strategy, we push the less productive partitions (larger P_s/P_o values) to disk in the disk-based adaptation. While in the distributed adaptation, we always choose the productive partitions (smaller P_s/P_o values) to move. Thus, more results are likely to be generated in the normal-run time.

The disk-based adaptation in the above approach is a local decision. This means the decision is made by the query processor as the memory overflow happens. However, the productivity of partitions among machines might not be the same. For example, the least productive partition in one machine, as the candidate to be pushed into disks, may still be much more productive than the partitions in another machine. Thus, if we move the disk-based adaptation decision to a global level (to the distribution manager), it would help to choose globally the least productive partitions among all machines to be pushed into disks. This should free more main memory space for the higher output rate partitions.

We now propose an *active-disk* approach which actively performs the disk-based adaptation. As illustrated in Figure 19, we monitor both the main memory usage and the average output rate of machines in the cluster. If the difference between the maximal memory usage and the least memory usage reaches a certain threshold θ_m, then the distributed adaptation is triggered as discussed above. If the memory usage across the machines in the cluster is balanced, then we compare the average output rate of each machine. If one machine has a much lower output rate, for example, $Max_{Rate}/Min_{Rate} > \theta_f$, we then force the partitions of the lower output rate machine to be pushed into disks. Here, the average output rate of one machine is defined as the number of tuples have been generated during the sampling time (the time intervals between two statistics checkings) divided by the number of partition groups in the machine. Given this, we would leave main memory space for the partitions in other machines that having higher output rates to be adapted into these machines. This would help the overall performance since higher output rate partitions are remained in main memory.

However, pushing too many states into disks than necessary could decrease the overall performance as well. In the active-disk strategy, we set the maximal amount of states being pushed by the distribution manager to be less than $M_q - M_c$, where M_q denotes the estimation of the overall main memory consumption of the query, while M_c is the overall main memory of the cluster.

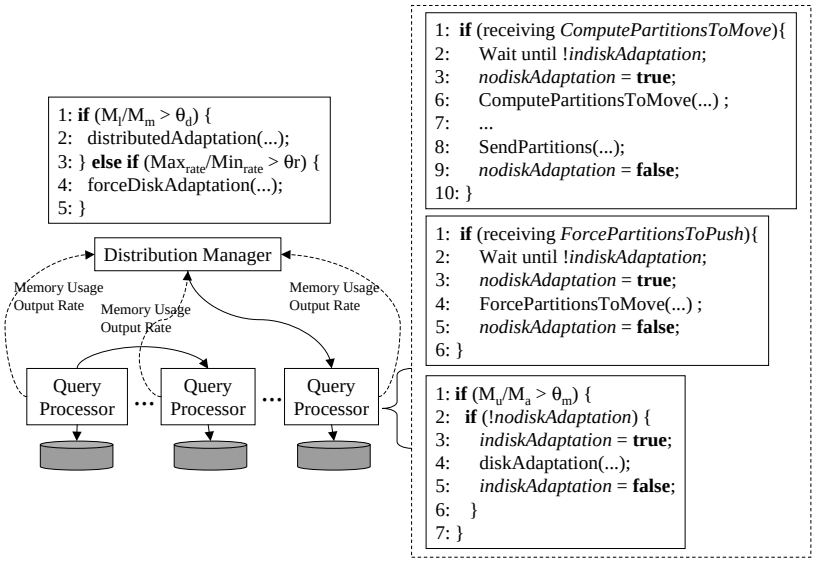


Figure 19: Active-Disk Adaptation Approach

6.1 Experimental Studies of Lazy-Disk and Active-Disk Approaches

A lazy-disk adaptation approach could fully make use of all main memory in the cluster. As we have shown in Section 5.2, the distributed adaptation only cause negilible overhead on the actual query processing, it thus results in a high throughput in the query processing. Figure 20 shows the performance of the lazy-disk approach when main memory of all machines is not enough for the query processing. The query is deployed on three machines. We set a skewed initial distribution that one machine has been assigned 2/3 of all partitions, while another two machines get 1/3 of the partitions. In this setup, if we do not apply distributed adaptation, then only one machine gets overloaded. While the other two machines can process its partitions fully in main memory. While using the lazy-disk approach, all three machines will eventually get overloaded and trigger disk-based adaptation.

From Figure 20, we see that the lazy-disk approach has a much higher overall throughput than the 'No-Distributed-Adaptation'. This is because the lazy-disk approach can fully make use of all available main memory in the cluster during the query processing.

Even in cases that the main memory of the cluster is extremely low, i.e., all machines cannot process the partitions assigned to them fully in main memory, a lazy-disk approach still has benefits. To illustrate, we again deploy the query into three machines and have one machine

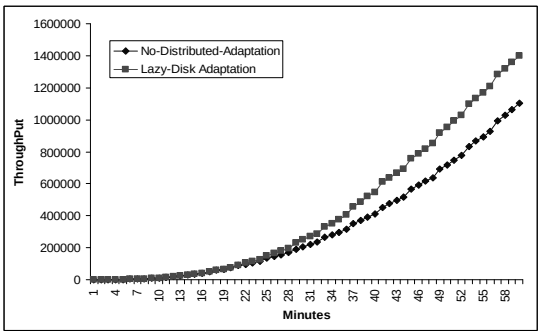


Figure 20: Lazy-Disk vs. No Distributed

get more partitions that 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 compare the performance of the lazy-disk approach with the No-Distributed-Adaptation. In the experiment, the overall results generated in this 6 hour run by these two approaches are similar. However, the clean up stage of these two approaches are dramatically different. The no-distributed approach takes more than 1600 seconds to produce 2,023,781 tuples in the clean up stage. This is because most of work is done by one machine. While the lazy-disk approach only takes less than 400 seconds to clean up. This is because the work is evenly distributed among all three machines.

An active disk adaptation strategy could further help the overall throughput of the query if the distribution manager observes the difference of the output rate of machines while the memory usage is balanced. Figure 21 shows one comparison of these two approaches. In this setup, we set the partitions assigned to machine 1 having a high join ratio of 4, while other two machines have a low join ratio of 1. The lazy-disk approach does nothing if the memory usage among three machines are balanced. While the active-disk forces low productive partitions to be pushed into disks since the output rate of machine 1 is much larger than that of the other two. Seen from Figure 21, the active disk strategy experiences a drop in the throughput after it starts pushing partitions into disk. However, in the long run, the active-disk strategy outperforms the lazy-disk, since more high productive partitions are remained in main memory.

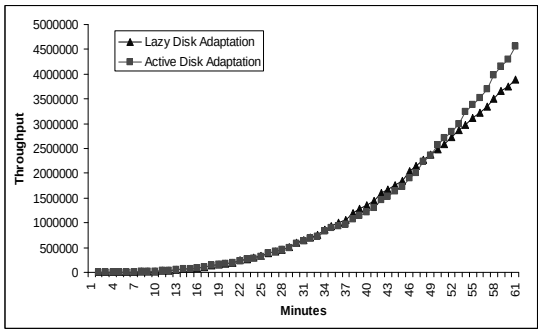


Figure 21: Lazy-Disk vs. Active-Disk

7 Conclusion

In this work, we have extensively studied the tradeoffs and policies of adapting operator states of complex non-blocking multi-input operators to overcome main memory overflow. All experiments are conducted on a working software system. We have proposed two adaptation approaches that integrate the disk-based adaptation and the distributed adaptation. Such integration has not been carefully studied in the literature, yet, it is necessary in practical environment since the main memory of a distributed system remains limited.

References

[1] D. Abadi, Y. Ahmad, and et. al. The Design of the Borealis Stream Processing Engine. Technical report, Brown University, Dept. of Computer Science, July 2004.

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

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

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

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

[7] 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 *Proceedings of VLDB*, pages 15–26, 1992.

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

[9] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

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

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

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

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

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

[15] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.

[16] W. J. Labio, R. Yerneni, and H. García-Molina. Shrinking the Warehouse Updated Window. In *Proceedings of SIGMOD*, pages 383–395, June 1999.

[17] B. Liu and E. A. Rundensteiner. Revisiting Parallel Multi-Join Query Processing via Hashing . Technical Report WPI-CS-TR-05-05, Worcester Polytechnic Institute, February 2005.

[18] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, pages 555–566, 2002.

[19] 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.

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

[21] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. Continuous Query Engine with Heterogeneous-Grained Adaptivity. In *Proceedings of VLDB Demo Session*, pages 1353–1356, 2004.

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

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

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

[25] B. Shivnath, S. Lakshminarayan, and W. Jennifer. A data stream management system for network traffic management. In *Proceedings of Workshop on Network-Related Data Management (NRDM 2001)*, 2001.

[26] T. Sutherland and E. Rundensteiner. D-CAPE: A Self-Tuning Continuous Query Plan Distribution Architecture. Technical report, Worcester Polytechnic Institute, Dept. of Computer Science, July 2004.

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

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

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

[30] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *Proceedings of ICDE*, page to appear, 2005.