Run-Time Operator State Spilling for Memory Intensive Long-Running Queries [*]

Bin Liu, Yali Zhu, and Elke A. Rundensteiner
Department of Computer Science, Worcester Polytechnic Institute
Worcester, Massachusetts, USA
{binliu, yaliz, rundenst}@cs.wpi.edu

ABSTRACT

Main memory is a critical resource when processing long-running queries over data streams with state intensive operators. In this work, we investigate state spill strategies that handle run-time memory shortage when processing such complex queries by selectively pushing operator states into disks. Unlike previous solutions which all focus on one single operator only, we instead target queries with multiple state intensive operators. We observe an interdependency among multiple operators in the query plan when spilling operator states. We illustrate that existing strategies, which do not take account of this interdependency, become largely ineffective in this query context. Clearly, a consolidated plan level spill strategy must be devised to address this problem. Several data spill strategies are proposed in this paper to maximize the run-time query throughput in memory constrained environments. The *bottom-up* state spill strategy is an operator-level strategy that treats all data in one operator state equally. More sophisticated partition-level data spill strategies are then proposed to take different characteristics of the input data into account, including the *local output*, the *global output* and the *global output with penalty* strategies. All proposed state spill strategies have been implemented in the D-CAPE query system. The experimental results confirm the effectiveness of our proposed strategies. In particular, the *global output* strategy and the *global output with penalty* strategy have shown favorable results as compared to the other two more localized strategies.

1. INTRODUCTION

Processing long-running queries over real-time data has gained great attention in recent years [2, 3, 6, 14]. Unlike static queries in a traditional database system, such query evaluates streaming data that is continuously arriving and produces query results in a real time fashion. The stringent requirement of generating real time results demands efficient main memory based query processing. Therefore long-running queries, especially complex queries with multiple potentially very large operator states such as multi-joins [21], can be extremely *memory intensive* during their execution.

[*]This work was partly supported by the National Science Foundation under grants IIS-0414567.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.
Copyright 2006 ACM 1-59593-256-9/06/0006 ...$5.00.

Memory intensive queries with multiple stateful operators are for instance common in data integration or in data warehousing environments. For example, a real-time data integration system helps financial analysts in making timely decisions. At run time, stock prices, volumes and external reviews are continuously sent to the integration server. The integration server must join these input streams as fast as possible to produce early output results to the decision support system. This ensures that financial analysts can analyze and make instantaneous decisions based on the most up to date information.

When a query system does not have enough resources to keep up with the query workload at runtime, techniques such as load shedding [19] can be applied to discard some workload from the system. However, in many cases, long-running queries may need to produce *complete* result sets, even though the query system may not have sufficient resources for the query workload at runtime. As an example, decision support applications rely on complete results to eventually apply complex and long-ranging historic data analysis, i.e., quantitive analysis. Thus, techniques such as load shedding [19] are not applicable for such applications.

One viable solution to address the problem of run-time main memory shortage while satisfying the needs of complete query results is to push memory resident states temporarily into disks when memory overflow occurs. Such solutions have been discussed in XJoin [20], Hash-Merge Join [15] and MJoin [21]. These solutions aim to ensure a high runtime output rate as well as the completeness of query results for a query that contains a single operator. The processing of the disk resident states, referred to as *state cleanup*, is delayed until a later time when more resources become available. We refer to this pushing and cleaning process as *state spill* adaptation.

However, the state spill strategies in the current literature are all designed for queries with one single stateful operator only [15, 20, 21]. We now point out that for a query with multiple state intensive operators, data spilling from one operator can affect other operators in the same pipeline. Such interdependency among operators in the same dataflow pipeline must be considered if the goal of the runtime data spilling is to ensure high output rate of the whole query plan. This poses new challenges on the state spill techniques, which the existing strategies, such as XJoin [20] and Hash-Merge Join [15], cannot cope with.

As an example of the problem considered, Figure 1 shows two stateful operators OP_i and OP_j with the output of OP_i directly feeding into OP_j. If we apply the existing state spill strategies on both operators separately, the interdependency between the two operators can cause problems not solved by these strategies. First, the data spill strategies would aim to maximize the output rate of OP_i when spilling states from OP_i. However, this could in fact backfire since it would in turn increase the main memory consumption of OP_j. Secondly, the states spilled in OP_i may have the potential to have made a high contribution to the output of OP_j. However, since they are spilled in OP_i, this may produce the opposite of the intended effect, that is, it may reduce instead of increase the output rate of OP_j. This contradicts the goal of the data spill strategies applied on OP_j.

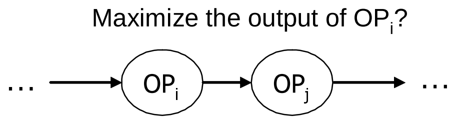


Figure 1: A Chain of Stateful Operators

In this work, we propose effective runtime data spill strategies for queries with multiple inter-dependent state intensive operators. The main research question addressed in this work is how to choose which part of the operator states of a query to spill at run-time to avoid memory overflow while maximizing the overall query throughput. Another important question addressed is how to efficiently clean-up disk-resident data to guarantee completeness of query results. We focus on applications that need accurate query results. Thus, all input tuples have to be processed either in real time during the execution stage or later during the state clean-up phase.

Several data spill strategies are proposed in this paper. We first discuss the *bottom-up* state spill strategy, which is a operator-level strategy that treats all data in one operator state equally. We then propose more sophisticated partition-level data spill strategies that take different characteristics of the input data into account, including a localized strategy called *local output*, and two global throughput-oriented state spilling strategies, named *global output* and *global output with penalty*. All proposed data spill strategies aim to select appropriate portions of the operator states to spill in order to maximize the run-time query throughput. We also propose efficient clean-up algorithms to generate the complete query results from the disk-resident data. Furthermore, we show how to extend the proposed data spill strategies to apply them in a parallel processing environment.

For long-running queries with high stream input rates and thus a monotonic increase of operator states, the state cleanup process may be performed only after the run-time execution phase finishes. In this paper we focus on this case. For queries with window constraints and bursty input streams, the in-memory execution and the disk clean-up may need to be interleaved at runtime. New issues in this scenario include timing of spill, timing of clean-up, and selection of data to clean-up. We plan to address these issues in our future work.

The proposed state spill strategies and clean-up algorithms have all been implemented in the D-CAPE continuous query system [13]. The experimental results confirm the effectiveness of our proposed strategies. In particular, the *global output* strategy and the *global output with penalty* strategy have shown more favorable results as compared to the other two more localized strategies.

The remainder of the paper is organized as follows. Section 2 discusses basic concepts that are necessary for later sections. Section 3 defines the problem of throughput-oriented data spilling we are addressing in this paper. The global throughput-oriented state spilling strategies are presented and analyzed in Section 4. Section 5 discusses the clean-up algorithms. In Section 6, we show how to apply the data spilling strategies in a parallel processing envionment. Performance evaluations are presented in Section 7. Section 8 discusses related work and we conclude in Section 9.

2. PRELIMINARIES

2.1 State Partitions and Partition Groups

Operators in a continuous long-running queries are required to be non-blocking. Thus many operators need *states*. For example, a join operator needs states to store tuples that have been processed so far so to join them with future incoming tuples from the other streams. In case of high stream arrival rates and long-running time, the states in an operator can become huge. Spilling one of these large states in its entirety to disk at times of memory overflow can be rather inefficient, and possibly even not necessary. In many cases, we need the flexibility to choose to spill part of a state or choose to spill data from several states to disk to temporarily reduce the query workload in terms of memory.

To facilitate this flexibility in run time adaptation, we can divide each input stream into a large number of partitions. This enables us to effectively spill some partitions in a state without affecting other partitions in the same state or partitions in other operator states. This method has first been found to be effective in the early data skew handling literature, such as [9], as well as in recent work on partitioned continuous query processing, such as Flux [18].

By using the above stream partitioning method, we can organize operator states based on the input partitions. Each input partition is identified by a unique partition ID. Thus each tuple within an operator state belongs to exactly one of these input partitions and would be associated with that particular partition ID. For simplicity, we also use the term partition to refer the corresponding operator state partition.

The input streams should be partitioned such that each query result can be generated from tuples within the same partition, i.e., with the same partition ID [1]. In this way, we can simply choose appropriate partitions to spill at run time, while avoiding repartitioning during this adaptation process. Figure 2 depicts the stream partitioning for a join query ($A \bowtie B \bowtie C$). The join is defined as A.A_1 = B.B_1 = C.C_1 where A, B, and C denote input streams (join relations) and A_1, B_1, and C_1 are the corresponding join columns. Here, the $Split_A$ operator partitions the stream A based on the

[1]For m-way joins (m > 2) [21] with join conditions defined on different columns, more data structures are required to support this partitioned m-way join processing. The discussion of this is out of the scope of the paper since we focus on the aspect of run-time state adaptation in this work.

value of column A_1, while the $Split_B$ operator partitions the stream B based on B_1, and so on. As we can see, in order to generate a final query result, tuples from stream A with partition ID 1 only need to join with tuples with the same partition ID from streams B and C.

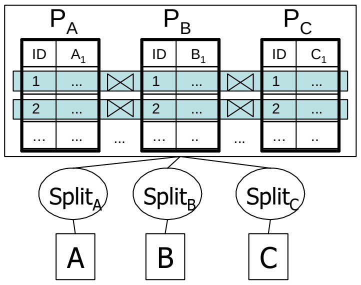
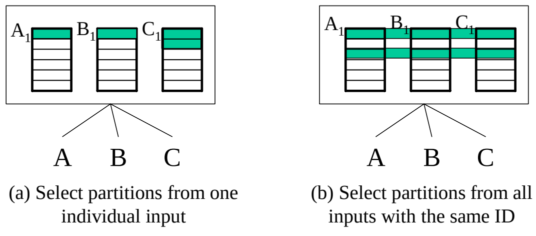


Figure 2: Example of Partitioned Inputs

When spilling operator states, we could choose partitions from each input separately, as shown in Figure 3(a). Using this strategy requires us to keep track of the timestamps of when each of these partitions was spilled to disk, and the timestamps of each tuple in order to avoid duplicates or missing results in the cleanup process. For example, partition A_1 has been spilled to the disk at time t. We use A_1^1 to denote this part of the partition A_1. All the tuples from B_1 and C_1 with a timestamp greater than t have to eventually join with the A_1^1 in the cleanup process. Since A_1, B_1, and C_1 could be spilled 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 spilled.

An alternative strategy is to use a *partition group* as the smallest unit of adaptation. As illustrated in Figure 3(b), a partition group contains partitions with the same partition ID from all inputs. During our research, we found that using the granularity of a partition group can simplify the cleanup process (described in Section 4). Therefore, in our work we choose to use the notion of a partition group as the smallest unit to spill to disk. From now on, we use the term partition to refer to a partition group if the context is clear. Since a query plan can contain multiple joins, the partition groups here are defined for each individual operator in the plan. Different operators may generate a tuple's partition ID based on different columns of that tuple. This arises when the join predicates are non-transitive. Therefore a tuple may hold different partition IDs in different operators.



A_1
B_1
C_1
A B C
(a) Select partitions from one individual input
(b) Select partitions from all inputs with the same ID

Figure 3: Composing Partition Groups

As an additional bonus, the approach of partitioning input streams (operator states) naturally facilitates efficient partitioned parallel query processing [10, 18]. That is, we can send non-overlapping partitions to multiple machines and have the query processed in parallel. The query processing can then proceed respectively on each machine. This will be further discussed in Section 5.

2.2 Calculating State Size

Serving as the basis for the following sections, we now describe how to calculate the *operator state size* and the *state size of the query tree.* The operator state size can be estimated based on the average size of each tuple and the total number of tuples in the operator. The total state size of the query tree is equal to the sum of all the operator state sizes. For example, the state size of $Join_1$ (see Figure 4) can be estimated by $S_1 = u_a{*}s_a{+}u_b{*}s_b{+}u_c{*}s_c$. Here, s_a, s_b, and s_c denote the number of tuples in $Join_1$ from input stream A, B and C respectively, and u_a, u_b, and u_c represent the average sizes of input tuples from the corresponding input streams.

In Figure 4, I_1 and I_2 denote the intermediate results from $Join_1$ and $Join_2$ respectively. Note that the average tuple size of I_1 can be represented by $u_a + u_b + u_c$, while the average tuple size of I_2 can be denoted by $u_a + u_b + u_c + u_d$ if no projection is applied in the query plan. This simple model can be naturally extended to situations when projections do exist.

The size of operator states to be spilled during the spill process can be computed in a similar manner. For example, assume d_a tuples from A, d_b tuples from B, and d_c tuples from C are to be spilled. Then, the spilled state size can be represented by $D_1 = u_a * d_a + u_b * d_b + u_c * d_c$.

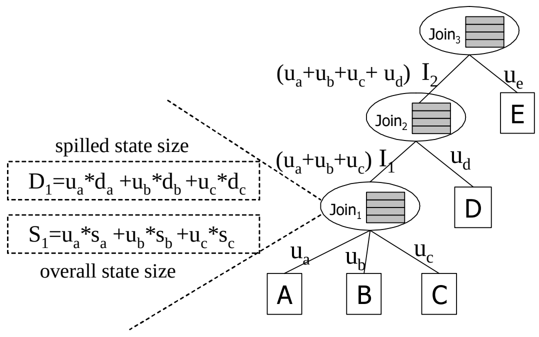


Figure 4: Unit Size of Each Stateful Operator

Thus, the total percentage of states spilled for the query tree can be computed by the sum of state sizes being spilled divided by the total state size. For the query tree depicted in Figure 4, it is denoted by $(D_1 + D_2 + D_3)/(S_1 + S_2 + S_3)$. Here S_i represents the total state size of operator $Join_i$, while D_i denotes the operator states being spilled from $Join_i$ $(1 \leq i \leq 3)$.

3. THROUGHPUT-ORIENTED STATE SPILL STRATEGIES

As discussed in Section 1, our goal is to keep the runtime throughput of the query plan as high as possible while at the same time preventing the system from memory overflow by applying runtime data spilling when necessary. Given multiple stateful operators in a query tree, partitions from all operators can be considered as potential candidates to be pushed when main memory overflows. We now discuss various strategies to choose partition groups to spill from multiple stateful operators.

State spill strategies have been investigated in the literature [15, 20, 21] to choose partitions from one *single* stateful operator to spill to disk with the least effect on the overall throughput. However, as discussed in Section 1, the existing strategies are not sufficient to apply on a query tree with multiple stateful operators, because they do not consider the interdependencies among a chain of stateful operators in a dataflow pipeline. As we will illustrate below, a direct extension of the existing strategies for one single operator does not perform well when applied to multiple stateful operators.

The decision of finding partitions to spill can be done at the *operator-level* or at the *partition-level*. Selecting partitions at the *operator-level* means that we first choose which operators to spill partitions from and then start to spill partitions from this operator until the desired amount of data is pushed to disk. If the size of the chosen operator state is smaller than the desired spill amount, we would choose the next operator to spill partitions from. In other words, by using the operator-level state spill, all partitions inside one operator state are treated uniformly and have equal chances of being spilled to disk. The state spilling can also be done at the *partition-level*, which treats each partition as an individual unit and globally choose which partitions to spill without considering which operators these partitions belong to. In this section, we present various state spill strategies at both the *operator-level* and the *partition-level*.

We first investigate the impact of pushing operator states to disk in a chain of operators. Figure 5 illustrates an example of an operator chain. Each operator in the chain represents a state intensive operator in a query tree. Note that it does not have to be a single input operator as depicted in the figure. s_i represents the corresponding selectivities of operator OP_i $(1 \le i \le n)$.

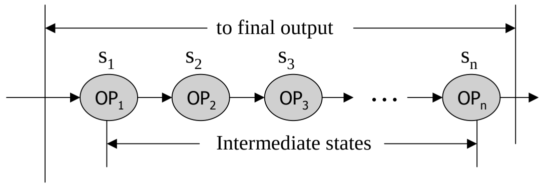


Figure 5: An Operator Chain

For such an operator chain, Equation 1 estimates the possible number of output tuples from OP_n given a set of input tuples t to OP_1.

$$u = \prod_{i=1}^{n} s_i \times t \tag{1}$$

The total number of tuples that will be stored somewhere within this chain due to these t input tuples, which also corresponds to the increase in the operator state size, can be computed as follows[2]:

[2]We assume that all input tuples to stateful join operators have to be stored in operator states. In principle, other stateful operators can be addressed in a similar manner.

$$I = \sum_{i=1}^{n} [(\prod_{j=1}^{i-1} s_j \times t)] \tag{2}$$

More precisely, OP_1 stores t tuples, OP_2 stores $t * s_1$ tuples, OP_3 stores $t * s_1 * s_2$ tuples, and so on. Thus, if we spill t tuples at OP_1, then all the corresponding intermediate results generated due to the existence of these t tuples and would have been stored in OP_2, OP_3, ..., OP_n now would not exist any more. Note that spilling any of these intermediate results would have the same overall effect on the final output, i.e., spilling the $t * s_1$ tuples at OP_2 would decrease the same amount of the final output as spilling t tuples at operator OP_1, as estimated by the Equation 1.

3.1 Operator-Level State Spill

3.1.1 Bottom-up Pushing Strategy

Inspired by the above analysis, we now propose a naive strategy, referred to as *bottom-up pushing*, to spill operator states of a query tree with multiple stateful operators at the operator-level. This strategy always chooses operator states from the bottom operator(s) in the query tree until enough space has been saved in the memory. For example, in Figure 5, the bottom operator is OP_1. Partition groups from bottom operators are chosen randomly and have equal chances to be chosen.

Intuitively, if partition groups from the bottom operator are chosen to be pushed to disk, less intermediate results would be stored in the query tree, compared to pushing states in the other operators. Thus, the bottom-up pushing strategy has the potential to lead to a smaller number of state spill processes, because less states (intermediate results) are expected to be accumulated in the query tree.

However, having a smaller number of state spill processes does not naturally result in a high overall throughput. This is because (1) the states being pushed in the bottom operator may contribute to a high output rate in its down stream operators, and (2) the cost of each state spill process may not be high, thus having a large number of state spill processes may not incur significant overhead on the query processing.

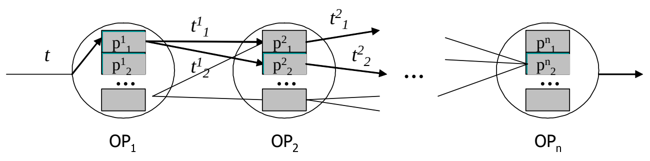


Figure 6: A Chain of Partitioned Operators

Moreover, the output of a particular partition of the bottom operator is likely to be sent into *multiple different* partitions of the down stream operator(s). For example, as illustrated in Figure 6, assume the t input tuples to OP_1 are partitioned into partition group P_1^1. Here the superscript represents the operator ID, while the subscript denotes the partition ID. After the processing in OP_1, t_1^1 result tuples are outputted and partitioned into P_1^2 of OP_2, while t_2^1 tuples are partitioned into P_2^2 of OP_2. The partitions P_1^2 and P_2^2 of OP_2 may have very different selectivities. For example, the output t_2^2 may be much larger than t_1^2 while the size of these two partitions may be similar. Thus, it may be worthwhile to keep P_1^1 in OP_1 even though certain states (in P_1^2 of OP_2) will be accumulated at the same time.

3.1.2 Discussions On Operator-Level State Spill

As we can see, the relationship between partitions among adjacent operators is a many-to-many relationship. Pushing partition groups at any operator other than the root operators may affect multiple partition groups at its down stream operators. However, an operator-level strategy, such as the presented bottom-up strategy, does not have a clear connection between the partition pushing and its effects on the overall throughput.

Another general drawback of the operator-level spilling is that it treats all partitions in the same state as having the same characteristics and the same effects on query performance when consider data spilling. However, different partitions may have different effects on the memory consumption and the query throughput after the data spilling. For example, some tuples have data values that appear more often in the stream, so they may have higher chances to joins with other tuples and produce more results. Thus we may need to make decisions on where to spill data on a finer granularity.

3.2 Partition-Level State Spill

To design a better state spilling strategy, we propose to globally select partition groups in the query tree as candidates to push. Figure 7 illustrates the basic idea of this approach. Instead of pushing partitions from particular operator(s) only, we conceptually view partitions from different operators at the same level. That is, we choose partitions globally at the query level based on certain cost statistics collected about each partition.

The basic statistics we collect for each partition group are P_{output} and P_{size}. P_{output} indicates the total number of tuples that have been output from the partition group, and P_{size} refers to the operator state size of the partition group. These two values together can be utilized to identify the *productivity* of the partition group. We now describe three different strategies for how to collect P_{output} and P_{size} values of each partition group, and how partition groups can be chosen based on these values with the most positive impact on the run time throughput.

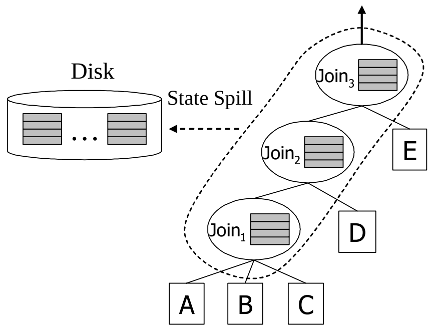


Figure 7: Globally Choose Partition Groups

3.2.1 Local Output Strategy

The first proposed partition-level state spill strategy, referred to as *local output*, updates P_{output} and P_{size} values of each partition group locally at each operator. The P_{size} of each partition group is updated whenever the input tuples are inserted into the partition group. While P_{output} value is updated whenever output tuples are generated from the operator.

Figure 8 illustrates this localized approach. When t tuples input to $Join_1$, we update P_{size} of the corresponding partition groups in $Join_1$. When t_1 tuples are generated from $Join_1$, then P_{output} value of the corresponding partition groups in $Join_1$ and the P_{size} value of related paritition groups in $Join_2$ are updated. Similarly, if we get t_2 from $Join_2$, then P_{output} of corresponding partition groups in $Join_2$ and P_{size} in $Join_3$ are updated.

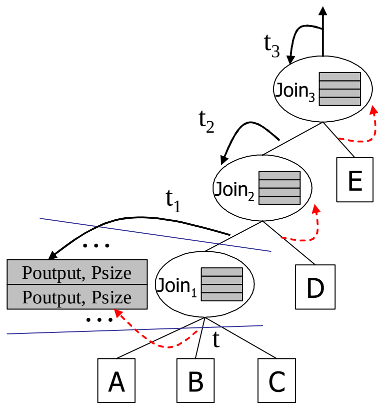


Figure 8: A Localized Statistics Approach

Different from the previous operator-level state spill, when selecting partitions to spill, this strategy chooses from the set of all partitions across all operators in the query plan based on their *productivity values* (P_{output}/P_{size}). Hence this is a partition-level state spill strategy. We push the partition group with the smallest productivity value among all partition groups in the query plan.

However, this approach does not provide a global productivity view of the partition groups. For example, if we keep partition groups of $Join_1$ with high productivity values in main memory, this in turn would contribute to generating more output tuples to be input to $Join_2$. All these tuples will be stored in $Join_2$ and hence will increase the main memory consumption of $Join_2$. This may cause the main memory to be filled up quickly. However, these intermediate results may not necessarily help the overall throughput since these results may be dropped by its down-stream operators.

3.2.2 Global Output Strategy

In order to maximize the run-time throughput after pushing states into disks, we need to have a global view of partition groups that reflects how each partition group contributes to the final output. That is, the productivity value of each partition group needs to be defined in terms of the whole query tree.

This requires the P_{output} value of each partition group to represent the number of final output tuples generated from the query. The productivity value, P_{output}/P_{size}, now indicates how 'good' the partition group is in terms of contributing to the final output of the query. Thus, if we keep the partition groups with high global productivity value in main memory, the overall throughput of the query tree is likely to be high compared with the previously described pushing strategies. Note that the key difference of this global output approach from the local output approach is its new way of computing the P_{output} value.

We have designed a *tracing algorithm* that computes the P_{output} value of each partition group. The basic idea is that whenever output tuples are generated from the query tree, we figure out the lineage of each output tuple. That is, we trace back to the corresponding partition groups from different operators that have contributed to this output. The tracing of the partition groups that contribute to an output tuple can be computed by applying the corresponding split operators. This is feasible since we can apply the split functions on the output tuple along the query tree to identify all the partition groups that the output tuple belongs to. Such tracing requires that the output tuple contains at least all join columns of the join operators in the query tree.

The main idea of the tracing algorithm is depicted in Figure 9. When k tuples are generated from $Join_3$, we directly update the P_{output} values of partition groups in $Join_3$ that produce these outputs. To find out the partition groups in the $Join_2$ that contribute to the outputs, we apply the partition function of $Split_2$ on each output tuple. Since multiple partition groups in the $Join_2$ may contribute to one partition group in $Join_3$, we need to trace for each partition group that is found in $Join_2$. Similarly, we apply the partition function of $Split_1$ to find the corresponding partition groups in operator $Join_1$. Note that we do not have to trace and update P_{output} for each output tuple. We only update the value with a random sample of the output tuples.

The pseudocode for the tracing algorithm for a chain of operators is given in Algorithm 1. Here, we assume that each stateful operator in the query tree keeps reference to its immediate upstream stateful operator and reference to its immediate upstream split operator. Upstream operator of an operator *op* here is defined as the operators that feed their output tuples as inputs to the operator *op*. Note that for a query tree, multiple immediate upstream stateful operators may exist for one operator. We can then similarly update the tracing algorithm to use a breadth-first or depth-first traversal algorithms of the query plan tree to update the P_{output} values of the corresponding partitions.

Algorithm 1 updateStatistics(tpSet)

```
/*Tracing and updating the $P_{output}$ values for a given set of
output tuples tpSet.*/
1:  op ← root operator of the query tree;
2:  prv_op_ref ← op.getUpStreamOperatorReference();
3:  prv_split_ref ← op.getUpStreamSplitReferences();
4:  while ((prv_op_ref ≠ null) && (prv_split_ref ≠ null)) do
5:      for each tuple $tp \in tpSet$ do
6:          $cPID$ ← Compute partitionID of tp in prv_op_ref;
7:          Update $P_{output}$ of partition group with ID $cPID$;
8:      end for
9:      prv_op_ref ← prv_op_ref.getUpStreamOperatorReference();
10:     prv_split_ref ← prv_split_ref.getUpStreamSplitReference();
11: end while
```

Given the above tracing, the P_{output} value of each partition group indicates the total number of outputs that have been generated that have this partition group involved in. The update of P_{size} value is the same as we have discussed in the local output approach. Thus, P_{output}/P_{size} indicates the *global productivity* of the partition group. By pushing partition groups with a lower global productivity, the overall run-time throughput would be expected to be better than the localized approach as well as the bottom-up approach.

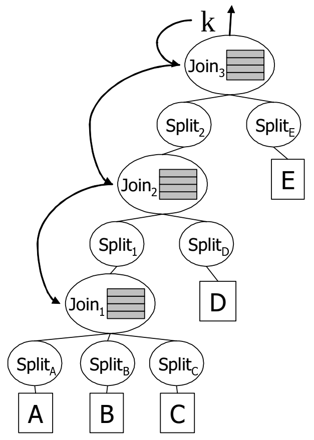
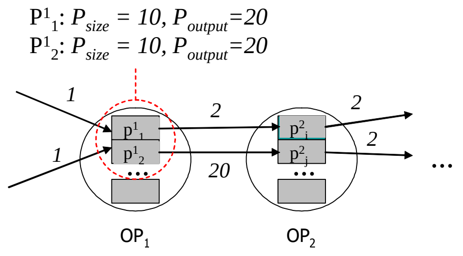


Figure 9: Tracing the Output Tuples

3.2.3 Global Output with Penalty Strategy

In the above approaches, the size of the partition group P_{size} reflects the main memory usage of the current partition group. However, as previously pointed out, the operators in a query tree are not independent. That is, output tuples of an up stream operator have to be stored in the down stream stateful operators. This indirectly affects the P_{size} of the corresponding partition groups in the down stream operator.



P^1_1: P_{size} = 10, P_{output}=20
P^1_2: P_{size} = 10, P_{output}=20
1
1
2
20
2
2
p^1_1
p^1_2
p^2_i
p^2_j
...
OP1
OP2

Figure 10: Impact of the Intermediate Results

For example, as shown in Figure 10, both partition groups P^1_1 and P^1_2 of OP_1 have the same P_{size} and P_{output} values. Thus, these two partitions have the same productivity value in the global output approach. However, P^1_1 produces 2 tuples on average that are output to the OP_2 given one input tuple, while P^1_2 generates 20 tuples on average given one input tuple. All intermediate results have to be stored in the down stream stateful operators. Thus, pushing P^1_2 instead of P^1_1 can help to reduce the memory that will be needed to store possible intermediate results in downstream operators.

To capture this effect, we define an intermediate result factor in each partition group, denoted by P_{inter}. This factor indicates the possible intermediate results that will be stored in its down stream operators in the query tree. In this strategy, the productivity value of each partition group is defined as $P_{output}/(P_{size} + P_{inter})$.

This intermediate result factor can be computed similarly as the tracing of the final output. That is, whenever an intermediate result is generated, we update the P_{inter} values of the corresponding partition groups in all the upstream operators. Figure 11 illustrates an example of how tracing algorithm can be utilized to update P_{inter}. In this example, one input tuple to OP_1 eventually generates 2 output tuples from OP_4. The number in the square box represents the number of intermediate results being generated.

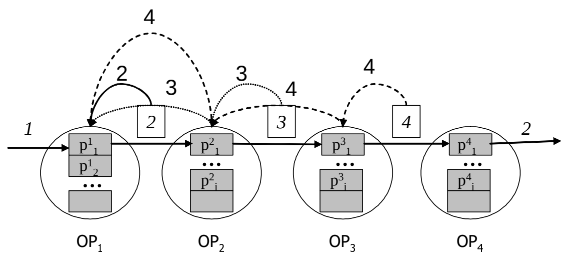


Figure 11: Tracing and Updating P_{inter} Values

4. CLEAN UP DISK RESIDENT PARTITIONS

4.1 Clean Up of One Stateful Operator

When memory becomes available, disk resident states have to be brought back to main memory to produce missing results. This *state cleanup process* can be performed at any time when memory becomes available during the execution. If no new resources are being devoted to the computation, then this cleanup process may likely occur at the end of the run-time phase. In the cleanup, we must produce all missing results due to spilling data to disk while preventing duplicates. Note that multiple partition groups may exist in disk for one partition ID. This is because once a partition group has been pushed into disk, new tuples with the same partition ID may again accumulate and thus a new partition group forms in main memory. Later, as needed, this partition group could be pushed into the 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 the disk resident ones.

Figure 12 illustrates an example of the partition groups before and after the cleanup process. Here, the example query is defined as $A \bowtie B \bowtie C$. We use a subscript to indicate the partition ID, while we use a superscript to distinguish between the partition groups with the same partition ID that have been pushed at different times. The collection of superscripts such as $1 \sim r$ represents the merge of partition groups that respectively had been pushed at times $1, 2, \ldots, r$.

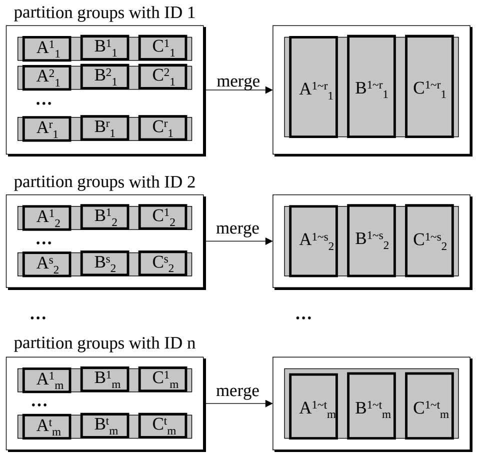


Figure 12: Example of Cleanup Process

The merge of partition groups with the same ID can be described as follows. For example, assume that a partition group with partition ID i has been pushed k times to disk, 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^j, B_i^j, C_i^j), $1 \leq j \leq k$ denotes the j-th time that the partition group with ID i has been pushed into the disk. For ease of description, we denote these partition groups by $P_i^1, P_i^2, \ldots, P_i^k$ respectively.

Due to our usage of the idea of spilling at the granularity of complete partition groups (see Section 2.1), the results generated between all the members of each partition group have already been produced during the previous run-time execution phase. In other words, all the results such as $A_i^1 \bowtie B_i^1 \bowtie C_i^1$, $A_i^2 \bowtie B_i^2 \bowtie C_i^2$, ..., $A_i^k \bowtie B_i^k \bowtie C_i^k$ are guaranteed to have been previously generated. For simplicity, we denote these results as V_i^1, V_i^2, ..., V_i^k. These partition groups can thus be considered to be *self-contained* partition groups given the fact that all the results have been generated from the operator states that are included in the partition group.

Merging two partition groups with the same partition ID results in a combined partition group that then contains union of the operator states from both partition groups. For example, the merge of P_i^1 and P_i^2 results in a new partition group $P_i^{1,2}$ now containing 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 output $V_i^{1,2}$ from partition group $P_i^{1,2}$ should be $(A_i^1 \cup A_i^2) \bowtie (B_i^1 \cup B_i^2) \bowtie (C_i^1 \cup C_i^2)$. Clearly, a subset of these output tuples have already been generated, namely, V_i^1 and V_i^2. Thus now we must generate the missing part in the merging process for these two partition groups in order to make the resulting partition group $P_i^{1,2}$ self-contained. This missing part is $\Delta V_i^{1,2} = V_i^{1,2} - V_i^1 - V_i^2$.

Here, we observe that the problem of merging partition groups and producing missing results is similar to the problem of the incremental batch view maintenance [11,16]. We thus now describe the algorithm for incremental batch view maintenance and then show how to map our problem to the view maintenance problem so to apply existing solutions from the literature to our problem [11,16].

Assume a materialized view V is defined as an n-way join upon n distributed data sources. It is denoted by $R_1 \bowtie R_2 \ldots \bowtie R_n$. There are n source deltas (ΔR_i, $1 \leq i \leq n$)

that need to be maintained. As was mentioned earlier, each ΔR_i denotes the changes (the collection of insert and delete tuples) on R_i at a logical level. An actual maintenance query will be issued separately, that is, one for insert tuples and one for delete tuples.

Given the above notations, the batch view maintenance process is depicted in Equation 3.

$$\begin{array}{rcl} \Delta V & = & \Delta R_1 \bowtie R_2 \bowtie R_3 \ldots \bowtie R_n \\ & + & R_1' \bowtie \Delta R_2 \bowtie R_3 \ldots \bowtie R_n \\ & + & \ldots \\ & + & R_1' \bowtie R_2' \bowtie R_3' \ldots \bowtie \Delta R_n \end{array} \tag{3}$$

Here R_i refers to the original data source state without any changes from ΔR_i incorporated in it yet, while R_i' represents the state after the ΔR_i has been incorporated, i.e., it reflects $R_i + \Delta R_i$ ('+' denotes the union operation). The discussion of the correctness of this batch view maintenance itself can be found in [11, 16].

Intuitively, we can treat one partition group as the base state, while the other as the incremental changes. Thus, the maintenance equation described in Equation 3 can be naturally applied to merge partitions and recompute missing results.

Lemma 4.1. *A combined partition group $P_i^{r,s}$ generated by merging partition groups P_i^r and P_i^s using the incremental batch view maintenance algorithm as listed in Equation 3 is self-contained if P_i^r and P_i^s were both self-contained before the merge.*

PROOF. Without loss of generality, we treat partition group P_i^r as the base state, while P_i^s as the incremental change to P_i^r. Incremental batch view maintenance equation as described in Equation 3 produces the following two results: (1) the partition group $P_i^{r,s}$ having both states of P_i^r and P_i^s, and (2) the incremental changes to the base result V_i^r by $\Delta V_i^{r,s} = V_i^{r,s}$ - V_i^r. Since two partition groups P_i^r and P_i^s already have results V_i^r and V_i^s generated, the missing result of combining P_i^r and P_i^s can be generated by $\Delta V_i^{r,s}$ - V_i^s. As can be seen, $P_i^{r,s}$ is self-contained since it has generated exactly the output results $V_i^{r,s} = (\Delta V_i^{r,s}$ - $V_i^s) + (V_i^r + V_i^s)$. ☐

As an example, let us assume A_i^1, B_i^1 and C_i^1 are the base states, while A_i^2, B_i^2 and C_i^2 are the incremental changes. Then, by evaluating the view maintenance equation in Equation 4, we get the combined partition group $P_i^{1,2}$ and the delta change $\Delta V_i^{1,2} = V_i^{1,2} - V_i^1$. By further removing V_i^2 from $\Delta V_i^{1,2}$, we generate exactly the missing results by combining P_i^1 and P_i^2.

$$\begin{array}{rcl} V_i^{1,2} - V_i^1 & = & A_i^2 \bowtie B_i^1 \bowtie C_i^1 \\ & \cup & (A_i^1 \cup A_i^2) \bowtie B_i^2 \bowtie C_i^1 \\ & \cup & (A_i^1 \cup A_i^2) \bowtie (B_i^1 \cup B_i^2) \bowtie C_i^2 \end{array} \tag{4}$$

Lemma 4.2. *Given a collection of self-contained partition groups $\{P_i^1, P_i^2, \ldots, P_i^m\}$, a self-contained partition group $P_i^{1\sim m}$ can be constructed using the above given incremental view maintenance algorithm repeatedly in $m-1$ steps.*

PROOF. A straightforward iterative process can be applied to combine such a collection of m partition groups. The first combination merges two partition groups, while the remaining m-2 partition groups are combined one at a time. Thus the combination ends after m-1 steps. Given each combination results in a self-contained partition group based on Lemma 4.1, the final partition group is self-contained. ☐

Based on Lemmas 4.1 and 4.2, we can see that the cleanup process (merging partition groups with the same partition ID) successfully produces exactly all missing results and no duplicates. Note that memory resident partition groups can be combined with the disk resident parts in exactly the same manner as discussed above. As can be seen, the cleanup process does not rely on any timestamps. We thus do not have to keep track of any timestamps during the state spill process.

4.2 Clean Up of Multiple Stateful Operators

Given a query tree with multiple stateful operators, when operator states from any of the stateful operators have been pushed into the disk during run-time, the final cleanup stage to completely remove all persistent data should not be performed in a random order. This is because the operator has to incorporate the missing results generated from the cleanup process of any of its up stream operators. That is, the cleanup process of join operators has to conform to the partial order as defined in the query tree.

Figure 13 illustrates a 5-join query tree ((A ⋈ B ⋈ C) ⋈ D) ⋈ E with three join operators $Join_1$, $Join_2$, and $Join_3$. Assume we have operator states pushed into the disk from all three operators. The corresponding join results from these disk resident states are denoted by ΔI_1, ΔI_2, and ΔI_3. From Figure 13, we can see that the cleanup results of $Join_1$ (ΔI_1) have to be joined with the complete operator states related to stream D to produce the complete cleanup results for $Join_2$. Here, the complete stream state D includes states from the disk resident part ΔI_2 and the corresponding main memory operator state. The cleanup result of $Join_2$, (ΔI_2 + $\Delta I_1 \bowtie D$), has to join with the complete stream state E in $Join_3$ to produce the missing results.

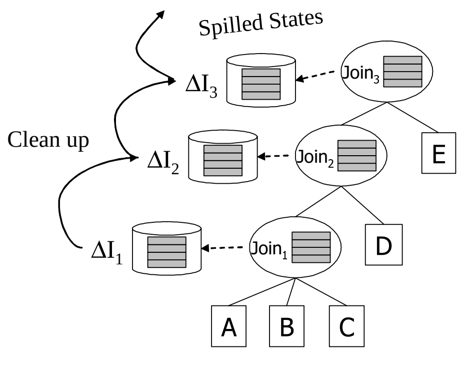


Figure 13: Clean Up the Operator Tree

Given this constraint, we design a *synchronized cleanup process* to combine disk resident states and produce all missing results. We start the cleanup from the bottom operators(s) which are the furthest from the root operator, i.e., from all the leaves. The cleanup process for operators with the same distance from the root can be processed concurrently. Once an up stream operator completes its cleanup process, it notifies its down stream operator using a control message interleaved in the data stream to signal that no more intermediate tuples will be sent to its down stream operators hereafter. This message then triggers the cleanup process of the down stream operator. Once the cleanup process of an operator is completed, the operator will no longer be scheduled by the query engine until the full cleanup is accomplished.

This synchronized cleanup process is illustrated in Figure 13. The cleanup process starts from $Join_1$. The generated missing results ΔI_1 are sent to the down stream operators. $Join_1$ then generates a special control tuple '*End-of-Cleanup*' to indicate the end of its cleanup. The down stream stateful operator $Join_2$ starts its cleanup after receiving the control tuple. All the other non-stateful operators, such as split operators, simply pass the '*End-of-Cleanup*' tuple through to their down stream operator(s). This process continues until all cleanup processes have been processed.

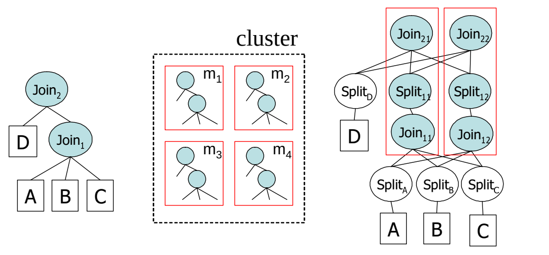
Note that in principle it is possible to start the cleanup process of all stateful operators at the same time. However, this may require a large amount of main memory space since each cleanup process will bring disk resident states into the memory. On the other hand, the operator states of the down stream operators cannot be released in any case until its up stream operators finish their cleanup and compute the missing results. While for the synchronized method, we instead bring these disk resident states into memory sequentially one operator at a time. Furthermore, we can safely discard them once the cleanup process of this operator completes.

5. APPLYING TO PARTITIONED PARALLEL QUERY PROCESSING

A query system that processes long-running queries over data streams can easily run out of resources when processing large volume of input stream data. Parallel query processing over a shared nothing architecture, i.e., a cluster of machines, has been recognized as a scalable method to solve this problem [1, 8, 18]. Parallel query processing can be especially useful for queries with multiple state intensive operators that are resource demanding in nature. This is exactly the type of queries we are focusing on in this work. However, the overall resources of even a distributed system may still be limiting. A parallel processing system may still need to temporarily spill state partitions to disk to react to overall resource shortage immediately. In this section, we illustrate that our proposed state spill strategies natually can be extended to also work for such partitioned parallel query processing environment. This observation broadens the applicability of our proposed spill techniques.

The approach of partitioning input streams (operator states) discussed in Section 2.1 is still applicable in the context of parallel query processing. In fact, it helps to achieve a partitioned parallel query processing [7, 12, 17]. We can simply spread the stream partitions across different machines with each machine only processing a portion of all inputs.

Figure 14 depicts an example of processing a query plan with two joins in a parallel processing environment. First, stateful operators must be distributed across available machines. In this work, we choose to allocate all stateful operators in the query tree to all the machines in the cluster, as shown in Figure 14(b). Thus, each machine will have exactly the same number of stateful operators defined in the query tree activated. Each machine processes a portion of all input streams of the stateful operators. The partitioned stateful operators can be connected by split operators as shown in Figure 14(c). One split operator is inserted after each instance of the stateful operator. The output of the operator instance is directly partitioned by the split operator and then shipped to the next appropriate down stream operators. Note that other approaches exist for both allocating stateful operators across multiple machines and connecting such partitioned query plans. However, the main focus of the work here is to adapt operator states to address the problem of run time main memory shortage. The exploration of other partitioned parallel processing approaches as well as their performance are beyond the scope of this paper.



(a) Original Query (b) Allocating Multiple Stateful Operators (c) Composing Partitioned Query Plan

Figure 14: Partitioned Parallel Processing

The throughput-oriented state spill strategies discussed in Section 3 naturally apply to the partitioned parallel processing environments. This is because the statistics we collect are based on main memory usage and operator states only.

However, given partitioned parallel processing, when applying the global output or the global output with penalty state spill strategy, the P_{output} value must be traced and then correctly updated across multiple machines. For example, as shown in Figure 15, the query plan is deployed in two machines. If k tuples are generated by $Join_3$, we directly update the P_{output} values of partition groups in $Join_3$ that have produced these outputs. To find out the partition groups in $Join_2$ that contribute to the outputs, we then apply the partition function of $Split_2$ on each output tuple. Note that given partitioned parallel processing, partition groups from different machines may contribute to the same partition group of the down stream operator. Thus, the tracing and updating of P_{output} values may involve multiple machines. In this work, we design an *UpdatePartitionStatistics* message to notify other machines of the update of P_{inter} and P_{output} values. Since each split operator knows exactly the mappings between the partition groups and the machines, it is feasible to only send the message to the machines that have the partition groups to be updated.

The revised updateStatistics algorithm is sketched in Algorithm 2. We classify partition group IDs by applying the current split function into *localIDs* and *remoteIDs* depending on whether the ID is mapped to the current machine. Then for the partition groups with *localIDs*, we update either P_{inter} or P_{output} based on whether the current *tpSet* is a set of intermediate results. While for the *remoteIDs*, we compose *UpdatePartitionStatistics* messages with appropriate information and then send the messages to the machine that holds the partition groups with their IDs in *remoteIDs*.

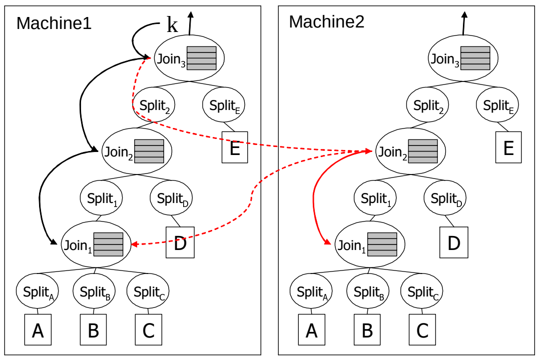


Figure 15: Tracing the Number of Output

Algorithm 2 updateStatisticsRev(tpSet,intermediate)

```
/*Tracing and updating the $P_{output}/P_{inter}$ values for a given set of output tuples tpSet. intermediate is a boolean indicating whether tpSet is the intermediate results of the query tree*/
1: op ← root operator of the query plan;
2: prv_op_ref ← op.getUpStreamOperatorReference();
3: prv_split_ref ← this.getUpStreamSplitReference();
4: while ((prv_op_ref $\neq$ null) && (prv_split_ref $\neq$ null)) do
5:    for each $tp \in tpSet$ do
6:       $cPID$ ← Compute partitionID of $tp$ in prv_op_ref;
7:       Classify $cPID$ into $localIDs/remoteIDs$;
8:    end for
9:    if (intermediate) then
10:      Update $P_{inter}$ of $localIDs$;
11:   else
12:      Update $P_{output}$ of $localIDs$;
13:   end if
14:   Compose & send UpdatePartitionStatistics msg(s) for $remoteIDs$;
15:   prv_split_ref ← prv_split_ref.getUpStreamSplitReference();
16:   prv_op_ref ← prv_op_ref.getUpStreamOperatorReference();
17: end while
```

6. PERFORMANCE STUDIES

6.1 Experimental Setup

All state spilling strategies discussed in this paper have been implemented in the D-CAPE system, a prototype continuous query system [13]. We use a five-join query tree illustrated in Figure 15 to report our experimental results. The query is defined on 5 input streams denoted as A, B, C, D, and E with each input stream having two columns. Here $Join_1$ is defined on the first column of each input stream A, B, and C. $Join_2$ is defined on the first join column of input D and the second join column of input C, while $Join_3$ is defined on the first column of input E and the second column of input D. The average tuple interarrival time is set to be 50 ms for each input stream. All joins utilize the symmetric hash-join algorithm [22].

We deploy the query on two machines with each processing about half of all input partitions. Each machine has dual 2.4Hz Xeon CPUs with 2G main memory. All input streams are partitioned into 300 partitions. We set the memory threshold (θ_m) for state spilling to be 60 MB for each machine. This means the system starts spilling states to disk when the memory usage of the system is over 60 MB.

We vary two factors, namely the *tuple range* and the *range join ratio*, when generating input streams. We specify that a data value V appears R times for every K input tuples. Here K is defined as the *tuple range* and R the *range join ratio* for V. Different values (partitions) in each join operator can have different range join ratios. The average of these ratios is defined as the *average join ratio* for that operator.

6.2 Experimental Evaluation

Figure 16 compares the run-time phase throughput of different state spilling strategies. Here we set the average join ratio of $Join_1$ to 3, while the average join ratio of $Join_2$ and $Join_3$ is 1. In Figure 16, the X-axis represents time, while the Y-axis denotes the overall run time throughput.

From Figure 16, we can see that both the *local output* approach and the *bottom-up* approach perform much worse than the *global output* and the *global output with penalty* approaches. This is as expected because the *local output* and the *bottom-up* approaches do not consider the productivity of partition groups at the global level. From Figure 16, we also see that the *global output with penalty* approach performs even better than the *global output* approach. This is because the global output with penalty approach is able to efficiently use the main memory resource by considering both the partition group size as well as the possible intermediate results that have to be stored in the query tree.

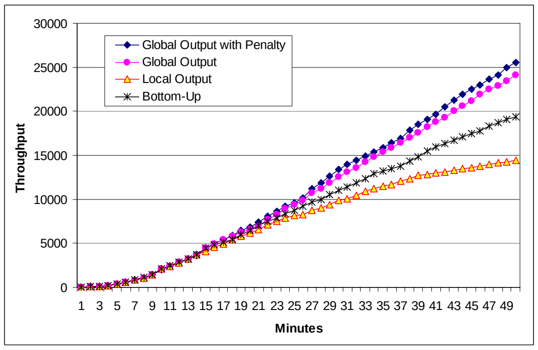


Figure 16: Comparing Run-time Throughput with Join Ratio 3,1,1

Figures 17 and 18 show the corresponding memory usage when applying different spilling strategies. Figure 17 shows the memory usage of the *global output* approach and *global output with penalty* approach. Note that each 'zig' in the lines indicates one state spill process. From Figure 17, we can see that the *global output* approach has a total of 13 state spill processes in the 50 minutes running, while the *global output with penalty* approach only spills for 10 times. Again, this is expected since the *global output with penalty* approach considers both the size of the partition group and the overall memory impact on the query tree.

As discussed in Section 3, having a smaller number of state spill processes does not imply a high overall run time throughput. In Figure 18, the *bottom-up* approach only has 7 times of adaptations. However, the run time throughput of the *bottom-up* approach is much less than the *global output*

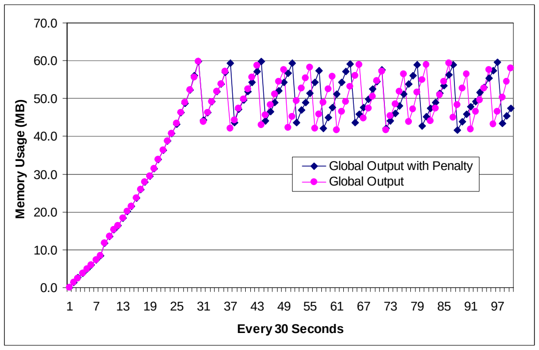


Figure 17: Memory Usage: Global Output vs. Global Output with Penalty

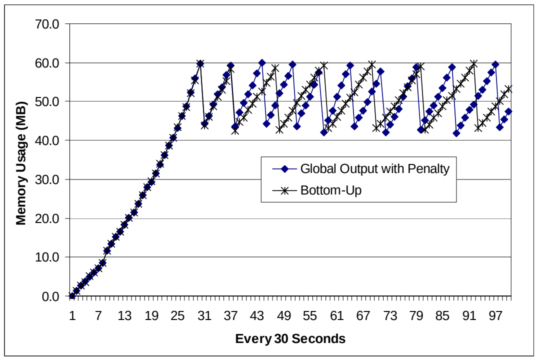


Figure 18: Memory Usage: Global Output with Penalty vs. Bottom-up

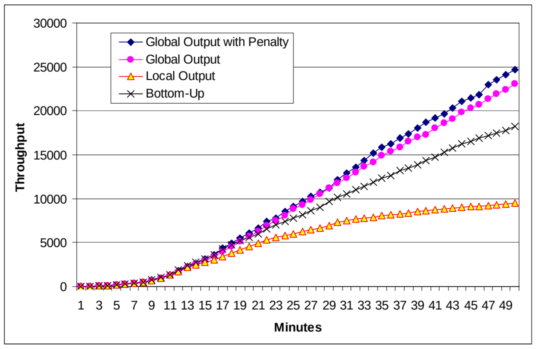


Figure 19: Comparing Run Time Throughput with Join Ratio 1,3,3

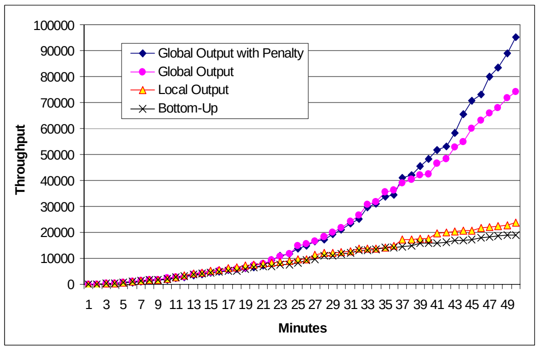


Figure 20: Comparing Run Time Throughput For Join Ratio 3,2,3

with penalty approach as shown in Figure 16. This is because having high productive partition groups in main memory helps to keep high overall throughput.

Figures 19 and 20 show the run time phase throughput when we run queries with different join ratios. In Figure 19, we set the join ratio of the three joins to be 1, 3 and 3. In Figure 20, we set the join ratio of the three joins to be 3, 2 and 3. From both figures, we see similar trends as we have observed in Figure 16: The *global output with penalty* approach always outperforms other state spill strategies, and both the *bottom-up* and the *local output* approaches are much worse than the two global approaches.

The main memory usage of these two experiments also shows a similar pattern as illustrated in Figures 17 and 18. That is, the *global output with penalty* approach requires less adaptations compared with the *global output* approach. While the *bottom-up* approach requires even less number of adaptations than the *global output with penalty* approach. As can be seen, less number of adaptations does not imply a high run-time throughput.

The cleanup time depends on where the operator states are pushed in the query tree. As we discussed in Section 4, the lower the level from which the partition groups are pushed, the higher the clean up cost. This is because the cleanup process needs to be sequentialized according to the partial order defined in the query tree. Therefore, the cleanup time of these approaches varies depending on the queries and the settings. In the experiment illustrated in Figure 16, the total cleanup time of the *global output with penalty* approach takes *495,741* ms, while the cleanup time of the *global output* approach takes *305,997* ms. For the same experiment shown in Figure 19, the cleanup time of the *global output with penalty* approach takes *278,234* ms, while the *global output* approach takes *362,752* ms. In all the above experiments, the *bottom-up* approach takes much longer to clean up disk resident states since this strategy tends to push partitions at the bottom operators.

In summary, in all of our experiments (including others not reported here due to space reasons), the two global output approaches consistently outperform the alternative approaches in the overall runtime throughput while typically also outperforming in the final cleanup costs.

7. RELATED WORK

State spill adaptation for one single operator has been investigated in the literature. As discussed in this paper, existing state spill solutions for stateful operators, including XJoin [20], Hash-Merge Join [15] and MJoin [21], are all designed to work with a single stateful operator. Such local approaches, as we have shown in our work, are not adequate for a query with multiple stateful operators that can be interdependent on each other. This is an important problem that has yet to be addressed in the current literature, and it is now the focus of this paper.

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 the scalability concerns [1, 8, 18]. In existing systems such as Aurora* [8] and Borealis [1], the main focus is to distribute the query plan across multiple machines and to balance the workload by moving complete query operators across machines.

Flux [18] discusses the partitioned parallel processing and the distributed adaptation in a continuous query processing 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 large stateful operators. However, Flux mainly focuses on adapting operator states across machines. Moreover, it focuses on one single operator only. In our work, we instead investigate methods of adapting operators with large states by state spilling at the level of state partitions.

Query processing over data streams [2, 3, 5, 6, 14] in general has gained growing research attentions in recent years. Such query processing faces scalability concerns due to high rates of inputs and possibly infinite data streams. Many techniques have been investigated to address this problem. For example, load shedding techniques [2, 19] aim to drop input tuples to handle the run time resource shortage while having the query results within certain predefined QoS requirements. However, this technique is not suitable for systems that require accurate query results, which are the systems our work focuses on. Adaptive scheduling and processing [4, 14] techniques have also been proposed. They focus on adapting the order of operators or tuples being processed. In this work, we instead focus on adapting the memory usage for multiple stateful operators with possibly huge volumes of states. The issue we tackle in this paper has not been explicitly studied in the context of stream processing yet.

8. CONCLUSION

In this work, we have studied the mechanisms and policies of spilling operator states of complex long-running queries with multiple state intensive operators to overcome run-time memory overflow. Such queries are rather common in a data integration context since the integration queries are complex and stateful in nature. Multiple throughput oriented state spill strategies are proposed. All these adaptation strategies have been implemented in the D-CAPE system [13]. Extensive experiments have been conducted and the results confirm the effectiveness of our proposed solutions.

9. REFERENCES

[1] D. Abadi, Y. Ahmad, and et. al. The design of the borealis stream processing engine. In *Proceedings 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] 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 *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 *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 *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 *CIDR Conference*, 2003.

[9] 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.

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

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

[12] B. Liu and E. A. Rundensteiner. Revisiting Parallel Multi-Join Query Processing via Hashing . In *Proceedings of VLDB*, pages 829–840, 2005.

[13] B. Liu, Y. Zhu, and et. al. A Dynamically Adaptive Distributed System for Processing Complex Continuous Queries. In *VLDB Demo*, pages 1338–1341, 2005.

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

[15] 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.

[16] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay. How To Roll a Join: Asynchronous Incremental View Maintenance. In *SIGMOD*, pages 129–140, 2000.

[17] 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.

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

[19] 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.

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

[21] 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.

[22] 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.