Chapter 1

CAPE: A CONSTRAINT-AWARE ADAPTIVE STREAM PROCESSING ENGINE *

Elke A. Rundensteiner, Luping Ding, Yali Zhu, Timothy Sutherland and Bradford Pielech
{rundenst, lisading, yaliz, tims, winners}@cs.wpi.edu

1. Introduction

1.1 Challenges in Streaming Data Processing

The growth of electronic commerce and the widespread use of sensor networks has created the demand for online processing and monitoring applications [17, 23, 26]. In these applications, data is no longer statically stored. Instead, it becomes available in the form of continuous streams. Furthermore, users often ask long-running queries and expect the results to be delivered incrementally in real time. Traditional query execution techniques, which assume finite persistent datasets and aim for producing a one-time query result, become largely inapplicable in this new stream paradigm due to the following reasons:

- The data streams are potentially infinite. Thus the existence of blocking operators in the query plan, such as group-by, may block query execution indefinitely because they need to see all input data before producing a result. Moreover, stateful operators such as join may require infinite storage resources to maintain all historical data for producing exact results.

- Data streams are continuously generated at query execution time. Meta knowledge about streaming data, such as data arrival patterns or data statistics, is largely unavailable at the initial query optimization phase. Therefore the initial processing decisions taken before query execution commences, including the query plan structure, operator execution algorithm and operator scheduling strategy, may not be optimal.

- Stream environments are usually highly dynamic. For example, the data arrival rates may fluctuate dramatically. Moreover, as other queries are registered into or removed from the system, the computing resources available for processing may vary greatly. Hence an optimal query plan may become sub-optimal as it proceeds, requiring run-time query plan restructuring and in some cases even across-machine plan redistribution.

It is apparent that novel strategies must be found to tackle the evaluation of continuous queries in such highly dynamic stream environments. In particular, this raises the need to offer adaptive services at all levels of query processing. The challenge is to cope with the variations in both stream environment and system resources, while still guaranteeing the precision and the timeliness of the query result. This is exactly the challenge that the stream processing system introduced in this chapter, named CAPE (for **C**onstraint-Aware **A**daptive Stream **P**rocessing **E**ngine) [21], tackles.

1.2 State-of-the-Art Stream Processing Systems

Many existing stream processing systems have begun to investigate various aspects of adaptive query execution. STREAM [20] for instance applies runtime modification of memory allocation and supports memory-minimizing operator scheduling policies such as Chain [3]. Aurora [1] supports flexible scheduling of operators via its Train scheduling technique [8]. It also employs the load shedding when an overload is detected. In [7], they point towards ideas for developing a distributed version of the Aurora and Medusa systems, including fault tolerance, distribution and load balancing. TelegraphCQ [6] provides a very fine-grained adaptivity by routing each tuple individually through its network of operators. While offering maximal flexibility, this comes with the overhead of having to manage the query path taken on an individual tuple basis and of having to recompute intermediate results.

These systems also consider the constraint-exploiting query optimization, in particular, they all incorporate various forms of sliding window semantics to bound the state of stateful operators. In addition, the STREAM system also exploits static k-constraints to reduce the resource requirements [4]. However, none of these systems considers punctuations which can be used to model both static and dynamic constraints in the stream context. Further optimization opportunities enabled by the interactions between different types of constraints are also not found in these systems.

1.3 CAPE: Adaptivity and Constraint Exploitation

In this chapter, we will describe CAPE, a **C**onstraint-Aware **A**daptive Stream **P**rocessing **E**ngine [21], that we have developed to effectively evaluate continuous queries in highly dynamic stream environments. CAPE adopts a novel architecture that offers highly adaptive services at all levels of query processing, including reactive operator execution, adaptive operator scheduling, runtime query plan restructuring and across-machine plan redistribution. In addition, unlike other systems that under resource limitation duress load shedding and thus affect the accuracy of the query result [1], CAPE instead focuses on maximally serving precise results by incorporating optimizations enabled by a variety of constraints. For instance, the CAPE operators are designed to exploit dynamic constraints such as punctuations [11, 26] in combination with time-based constraints such as sliding windows [5, 15, 16, 20] to shrink the runtime state and to produce early partial results.

This chapter now describes four core services in CAPE that are constraint-exploiting and highly adaptive in nature:

- The *constraint-exploiting reactive query operators* exploit constraints to reduce resource requirements and to improve the response time. These operators employ an adaptive execution logic to react to the varying stream environment.

- The *introspective execution scheduling framework* adaptively selects one algorithm from a pool of scheduling algorithms that helps the query to best meet the optimization objectives.

- The *online query plan reoptimization and migration* restructures the query plan at runtime to continuously converge to the best possible route that input data goes through.

- The *adaptive query plan distribution framework* balances the query processing workload among a cluster of machines so to maximally exploit available CPU and memory resources.

We introduce the CAPE system in Section 2. The design of the CAPE query operators is described in Section 3. Sections 4, 5 and 6 present our solutions for operator scheduling, online plan reoptimization and migration, and plan distribution respectively. Finally, we conclude this chapter in Section 7.

2. CAPE System Overview

CAPE embeds novel adaptation techniques for tuning different levels of query evaluation, ranging from intra-operator execution, operator scheduling, query plan structuring, to plan distribution. Each level of adaptation is able to yield maximally optimized performance in certain situations by working on their own. However, none of them is able to handle all kinds of variations that may occur in a stream environment. In addition, the improper use of all levels of adaptation may cause either optimization counteraction or oscillating re-optimizations, which should both be avoided. Hence an important task is to coordinate different levels of adaptations, guiding them to function properly on their own and also to cooperate with each other in a well-regulated manner. CAPE not only incorporates novel adaptation strategies for all aspects of continuous query evaluation, but more importantly, it employs a well-designed mechanism for coordinating different levels of adaptation.

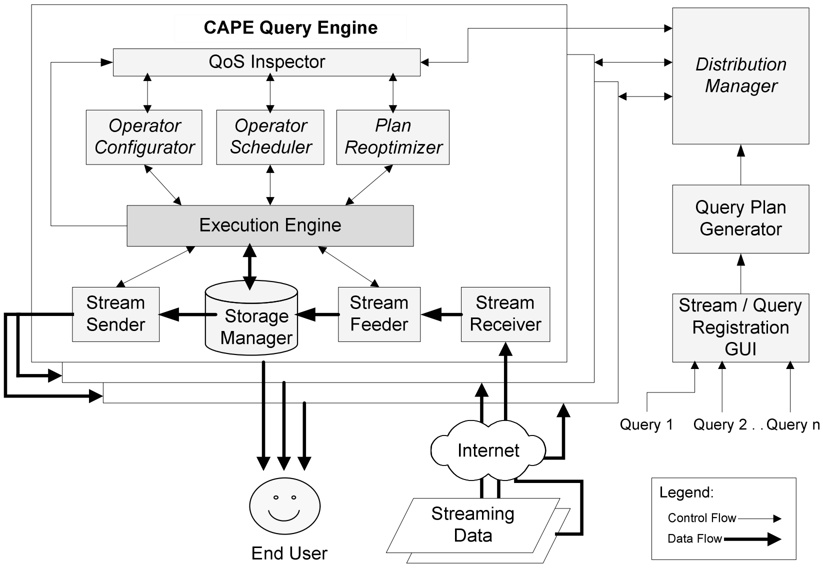


Figure 1.1. CAPE System Architecture.

In the system architecture depicted in Figure 1.1, the key adaptive components are Operator Configurator, Operator Scheduler, Plan Reoptimizer and Distribution Manager. Once the Execution Engine starts executing the query plan, the QoS (Quality of Service) Inspector will regularly collect statistics from the Execution Engine at each sampling point. All the above four adaptive components then use these statistics along with QoS specifications to determine if they need to adjust their behavior.

To synchronize adaptations at all levels, we have designed a ***heterogeneous-grained adaptation schema***. Since these adaptations deal with dissimilar runtime situations and have different overheads, they are invoked in CAPE under different frequencies and conditions. The current adaptation components in CAPE and the granularities of adaption are shown in Figure 1.2, with the adaption interval increasing as we go from the inner to the outer layers of the onion shape.

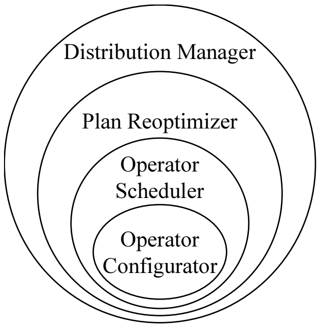


Figure 1.2. Heterogeneous-grained Adaptation Schema

The intra-operator adaptation incurs the lowest overhead so that it functions within an operator's execution time slot. Our event-driven intra-operator scheduling mechanism enables the operators, especially the stateful and blocking ones, to adjust their execution at runtime through the Operator Configurator. The Operator Scheduler is able to adjust the operator processing order after a run of a single operator or a group of operators, called a scheduling unit. After a scheduling unit finishes its work, the scheduler will check the QoS metrics for the operators and decide which operator to run next or even switch to a better scheduling strategy. This is a novel feature unique to our system. The Plan Reoptimizer will wait for the completion of several scheduling units and then check the QoS metrics for the entire query plan residing on its local machine to decide whether to restructure the plan. The Distribution Manager, which potentially incorporates the highest costs in comparison with other adaptive components due to across-machine data transfers, is invoked the least frequently, i.e., it is assigned the longest decision making interval. If a particular machine is detected to be overloaded, the Distribution Manager will redistribute one or multiple query plans among the given cluster of machines. While the Plan Reorganizer migrate the old plan to a new plan structure, the Distribution Manager instead migrates a query plan from one machine to another machine.

3. Constraint-Exploiting Reactive Query Operators

As described in Section 1.1, uncertainties may exist in many aspects of a streaming environment, including the data arrival rate, the resource availability, etc. The operators in CAPE are designed to react to such variations by adapting their behavior appropriately [11]. Moreover, these operators exploit various constraints to optimize their execution without sacrificing the precision of the query result [12].

In this section we use the design of a join operator as an example to illustrate the optimization principles inherent in the CAPE operator design. We highlight in particular two features unique to CAPE: the adaptive operator execution logic and the exploitation of punctuations.

For clarity of presentation, we use a join over two streams S_1<A, B_1> and S_2<A, B_2> with the join condition S_1.A = S_2.A. The schema of the join result is <A, B_1, B_2>. We assume that each tuple or punctuation has a timestamp field *TS* that records its arrival time. We also assume that tuples and punctuations in both streams have a global ordering on their timestamp.

3.1 Issues with Stream Join Algorithm

Pipelined join operators have been proposed for delivering early partial results in processing streaming data [14, 19, 27, 29]. These operators build one state per input stream to hold the already-processed data.. As a tuple comes in on one input stream, it is used to *probe* the state of the other input stream. If a match is found, a result is produced. Finally the tuple is *inserted* into the state of its own input stream. In summary, this join algorithm completes the process of each tuple by following the *probe-insert* sequence.

Some issues may arise with this algorithm. As tuples continuously accumulate in the join states, the join may run out of memory. To prevent data loss, part of the state needs to be flushed to disk. This may cause many expensive I/O operations when we try to join new tuples with those tuples on disk. As more data is paged to disk, the join execution will be slowed down significantly. In addition, the join state may potentially consumes infinite storage.

3.2 Constraint-Exploiting Join Algorithm

In most cases it is not necessary to maintain all the historical data in the states. Constraints such as sliding windows [5, 15, 16, 20] or punctuations [11, 26] can be utilized by the join to detect and discard no-longer-needed data from the states. This way the join state can be shrunk in a timely manner, thereby reducing and even eliminating the need of paging data to disk.

We first consider the sliding window, a time-range constraint. Assume that in the join predicate, two time-based windows W_1 and W_2 are specified on streams S_1 and S_2 respectively. A new tuple from S_1 can only be joined with tuples from S_2 that arrived within the last W_2 time units. So can new tuples from S_2. Hence the join only needs to keep tuples that have not yet expired from the window. Any new tuple from one stream can be used to remove expired tuples from the other stream. Accordingly, the probe-insert execution logic should be extended to add a third operation that *invalidates* tuples based on the sliding window constraints.

Punctuations are value-based constraints embedded inside a data stream. A punctuation in a stream is expressed as an ordered set of patterns, with each pattern corresponding to one attribute of the tuple from this stream. The punctuation semantics define that no tuple that arrives *after* a punctuation will match this punctuation. As the join operator receives a punctuation from one stream, it can then purge all already-processed tuples from the other stream that match this punctuation because these tuples no longer contribute to any future join results. In response, a new operation, namely *purge*, that discards tuples according to punctuations, needs to be added into the join execution logic.

Below we use an example query in an online auction application to briefly illustrate how these constraints are used to optimize the evaluation of this query. As shown in Figure 1.3, in this auction system, the items that are open for auction, the bids placed on the items and the registered users are recorded in the *Item*, *Bid* and *Person* streams respectively. When the open duration for a certain item expires, the auction system can insert a punctuation into the Bid stream to signal the end of bids for that item, e.g., the punctuation $<$1082, $*$, $*$, $*>$ on item 1082 in the figure. Figure 1.3 also shows a stream query in CQL language [2] and a corresponding query plan. For each person registered with the auction system, this query asks for the count of distinct categories of all the items this person has bid on within 12 hours of his registration time.

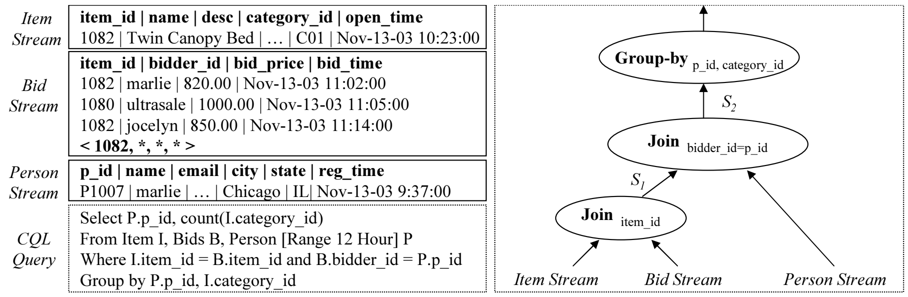


Figure 1.3. Example Query in Online Auction System.

In the first join (Item $\bowtie$ Bid) in the query plan, when a punctuation is received from the Bid stream, the tuple with the matching *item_id* from the Item stream can be purged because it will no longer join with any future Bid tuples. The second join ($S_1 \bowtie$ Person) applies a 12-hour window on Person. Tuples from stream S_1 can be used to invalidate expired tuples from the Person stream.

Besides utilizing punctuations to optimize their own execution, the operators can also propagate punctuations to help other operators. In the above example, when a Person tuple moves out of the window, no more join results will be produced for this person. A punctuation can then be propagated to trigger the group-by operator to emit a result for this person. This way the group-by operator is able to produce real-time results instead of being blocked indefinitely.

3.3 Optimizations Enabled by Combined Constraints

Either punctuation or sliding window can help shrink the join state. We now show that when they are present simultaneously, further optimizations can be enabled. Such optimizations are not achievable if only one constraint type occurs. We first present a theorem as the foundation of these optimizations.

THEOREM 1.1 *Assume t_i is announced by a punctuation to be the last tuple ever in stream S_i that has value a_k for join attribute A. Once t_i expires from the sliding window, no more join results with A=a_k will be generated thereafter.*

Based on this theorem, we derive a *tuple dropping invariant* for dropping new tuples that won't contribute to the join result. This further reduces the join workload without compromising the precision of the join result.

DEFINITION 1.2 **(Tuple Dropping Invariant.)** *Let t_i be the last tuple from stream S_i that ever contains value a_k for the join attribute A and let latestTS be the timestamp of the latest tuple being processed thus far. Drop tuple t_j from stream S_j ($j \neq i$) if t_i.TS < latestTS - W_i and t_j.A = t_i.A and t_j.TS $\geq$ latestTS.*

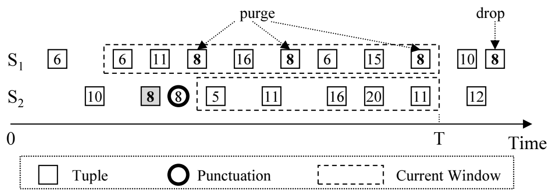


Figure 1.4. Dropping Tuples Based on Constraints.

Figure 1.4 shows an example of applying the tuple dropping invariant. The last tuple with join value 8 in stream S_2 expires from the window at time T. Hence, the tuple dropping invariant is satisfied. In the figure, four tuples in stream S_1 are shown to have join value 8. Three of them arrived before time T so that they have joined with the matching tuples from S_1 and have been purged by the purge operation. Another tuple with this join value is arriving after time T. This tuple can then be dropped with no need to be processed based on the tuple dropping invariant.

Now we consider how the combined constraints assist the punctuation propagation in the join operator. Assume the join receives a punctuation $<a_1, *>$ from S_1, which declares that no more tuples from S_1 will have value a_1 for attribute A. The join, however, may not be able to immediately output a punctuation $<a_1, *, *>$ because tuples with A=a_1 are still potentially coming from S_2. This may render future results with A=a_1. Only when the join state contains no tuple with A=a_1 from stream S_1, we can safely output this punctuation.

We observe that without sliding window, we can only propagate punctuations in a very restrictive case, i.e., punctuations are specified on the join attribute. When the punctuation on a certain join value has been received from both streams, we know that all results with this join value have been produced. The join is then able to propagate this punctuation.

In the presence of both punctuations and sliding windows, a more efficient propagation strategy can be achieved in the *invalidation* operation of the join algorithm. As we invalidate expired tuples from the window, we also invalidate punctuations. When a punctuation from one stream moves out of the window, all tuples from this stream that match this punctuation must have all expired from the window. Therefore the propagation condition is satisfied and this punctuation becomes propagable. Also note that the punctuations propagated by this strategy are not necessary to be on the join attribute.

3.4 Adaptive Component-Based Execution Logic

As described in Section 3.2, the join algorithm may involve numerous tasks: (1) *memory join*, which probes in-memory join state using a new tuple and produces results for any matches, (2) *state relocation*, that moves part of the in-memory state to disk when running out of memory, (3) *disk join*, that retrieves data from disk into memory for join processing, (4) *purge*, that purges no-longer-useful data from the state according to punctuations, and (5) *invalidation*, that removes expired tuples from the state based on the sliding window.

The frequencies of executing each of these tasks may be rather different due to performance considerations. Memory join is executed as long as new tuples are ready to be processed. This guarantees the join result to be delivered as soon as possible. State relocation is applied only when the memory limit is reached. This way the I/O operations are reduced to a minimum. Disk join also involves I/O operations. Hence it is scheduled only when the memory join cannot proceed due to the delays in data delivery. The purge incurs overhead in searching for tuples that satisfy the purge invariant. Depending on how frequently the punctuations arrive, we may choose to run the purge task after receiving a certain number of punctuations (*purge threshold*) or when the memory usage reaches the limit. Similarly, the invalidation task also incurs overhead in searching for expired tuples. We may decide to conduct this task after processing a certain number of new tuples (*invalidation threshold*).

Due to the dynamic nature of the streaming environment, the threshold associated with these tasks may vary over time. For example, as other queries enter and leave the system, the memory limit of a operator may be decreased and increased accordingly. The traditional join algorithm that follows a fixed sequence of operations is inappropriate for achieving such fine-tuned execution logic. In response, we have devised a component-based adaptive join algorithm to cope with the dynamic streaming environment. For this algorithm, we design five components for accomplishing the five tasks listed above. We also employ an *event-driven* framework to adaptively schedule these components according to certain changes that may affect the performance of the operator.

As shown in Figure 1.5, the memory join is scheduled as long as new tuples are ready to be processed. Meanwhile, an *event generator* monitors a variety of runtime parameters that serve as the component triggering conditions. These parameters include the memory usage of the join state, the number of punctuations that arrived since the last purge, etc. When a parameter reaches the corresponding threshold, e.g., when the memory usage reaches the *memory limit*, a corresponding event will be invoked. Then the memory join is suspended and the registered listener to the invoked event, i.e., one of the components, will be scheduled to run. After the listener finishes its work, the memory join will be resumed. We have defined the following events for the join operator:

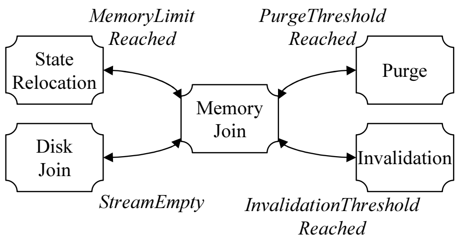


Figure 1.5. Adaptive Component-Based Join Execution Logic.

1 *StreamEmpty* signals that both input streams currently contain no tuple.

2 *PurgeThresholdReached* signals that the number of unprocessed punctuations has reached the *purge threshold*.

3 *MemoryLimitReached* signals the memory used by the join state has reached the *memory limit*.

4 *InvalidationThresholdReached* signals that the number of newly-processed tuples since the last invalidation has reached the *invalidation threshold*.

The join operator maintains an *event-listener registry*. Each entry in the registry lists the event being generated, additional conditions to be checked and the listener (component) which will be executed to handle the event. The

Table 1.1. Example Event-Listener Registry.

Events	*Conditions*	*Listeners*
StreamEmpty	Activation threshold: 70%.	Disk Join
PurgeThresholdReached	None.	Purge
MemoryLimitReached	There exist unprocessed punctuations.	Purge
MemoryLimitReached	There are no unprocessed punctuations.	State Relocation

registry is initiated at the static query optimization phase and can be updated at runtime. The thresholds used for invoking the events are specified in the *event generator*. They can be changed at runtime. Table 1.1 shows an example registry for a join with no sliding window applied. Hence, no entry in the registry corresponds to the invalidation component.

3.5 Summary of Performance Evaluation

From our experimental study on the join operator in CAPE [11, 12], we have obtained the following observations:

- By only exploiting punctuations, in the best case the join state consumes nearly constant memory. The shrinkage in state also helps improve the tuple output rate of the join operator because the probe operation can now be done more efficiently.

- In terms of the sliding window join, if the window contains a large amount of tuples, by in addition exploiting punctuations, the memory consumption of the join state is further reduced and the tuple output rate increases accordingly due to the tuple dropping.

- The adaptive execution logic enables the join operator to continue outputting results even when the data delivery experiences temporary delay. In addition, the appropriate purge threshold and invalidation threshold settings help the join operator to achieve a good balancing between the memory overhead and the tuple output rate.

4. Adaptive Execution Scheduling

Rather than randomly select operators to execute or leave such execution ordering up to the underlying operating system, stream processing systems aim to have fine-grained control over the query execution process. In response, scheduling algorithms are being designed that decide on the order in which the operators are executed. They target specific optimization goals, such as increasing the output rate or reducing the memory usage.

4.1 State-of-the-Art Operator Scheduling

Current stream processing systems initially employed traditional scheduling algorithms borrowed from the realm of operating systems [9, 30], such as Round-Robin and FIFO. More recently, customized algorithms designed specifically for continuous query evaluation have been proposed, including Chain [3] in STREAM and Train [8] in Aurora. The Chain scheduling strategy is designed with the goal of minimizing intermediate queue sizes, thereby minimizing the memory overhead. However, it is not targeting at meeting other Quality of Service (QoS) requirements. The Train scheduling algorithms, four in total, are variations each tuned for a particular QoS criterion.

We have experimentally compared these popular algorithms under a variety of stream workloads within the CAPE testbed. This experimental study [24] reveals that each of these algorithms is good at improving the system performance in one specific manner, e.g., Chain for reducing memory usage and FIFO for increasing the result output rate. Thus, even though many scheduling algorithms exist in the literature, there is no one algorithm that a system can utilize to satisfy the diversity of system requirements common to stream systems. In other words, it is difficult to design a scheduling algorithm that always functions effectively even when experiencing a wide variety of changing conditions, including changing QoS requirements, the addition of new queries or runtime query plan reoptimization (See Section 5).

The existing stream systems usually select one scheduling algorithm at the beginning of the query execution and then stick with it. This overlooks the fact that as the stream environment experiences changes, the initially optimal scheduling algorithm may become sub-optimal over time. One possible solution to this dilemma may be to put a human administrator in charge of the decision on selecting scheduling algorithms. However, it is often impossible for an administrator to know a priori which scheduling algorithm to pick, or more challenging even which algorithm to turn on or off at runtime based on the behavior of the system. It is exactly this issue of automating the scheduling algorithm selection that we address in CAPE.

4.2 The ASSA Framework

We have designed a framework for the *Adaptive Selection of Scheduling Algorithms* (ASSA for short) [24]. The ASSA architecture is depicted in Figure 1.6. ASSA is equipped with a library of scheduling algorithms, ranging from well established ones like Round Robin and FIFO to recently proposed ones like Chain and Train. As new scheduling algorithms are developed, they can easily be plugged into this library.

In a nutshell, having several algorithms at its avail with each targeting different QoS requirements, the *Strategy Selector* dynamically selects one scheduling

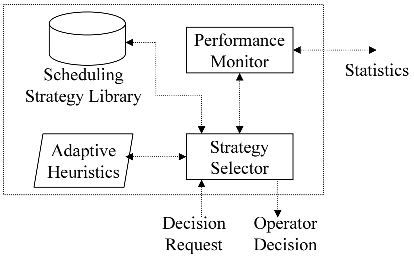


Figure 1.6. Architecture of ASSA Scheduler.

algorithm from the library for scheduling the execution of the operators in a plan. For this, the *Performance Monitor* must establish some measures by which the algorithms can be compared and ranked in terms of their expected effectiveness. ASSA learns about the impact of each algorithm on the query system by observing how well each algorithm has done thus far during execution. This learned knowledge is encoded into a score for each algorithm.

The *Strategy Selector* then utilizes those learned scores to guide the selection of the candidate algorithm. It applies a lightweight *Adaptive Heuristic*, also called Roulette Wheel heuristic [18], that selects the next candidate algorithm to use for scheduling based on its perceived potential to fulfill the current set of QoS requirements. ASSA then simply asks this selected algorithm to pick the next operator to execute. Lastly this decision is then reported to the Execution Engine which carries out the control of the actual execution flow.

4.3 The ASSA Strategy: Metrics, Scoring and Selection

We now describe the specified QoS requirements that the ASSA selector utilizes to assess the effectiveness of a scheduler. We also discuss the fitness score assigned to each scheduler to capture how well it performed relative to the other algorithms.

Quality of Service Requirements. Our system allows for the system administrator to specify the desired execution behavior as a composition of several metrics. A QoS requirement consists of three components: the statistic, quantifier, and weight. The statistic corresponds to the metric that is to be controlled. Performance metrics considered include throughput (the number of result tuples produced), memory requirements, and freshness of results (the amount of time a tuple stays in the system). The quantifier, either *maximize* or *minimize*, specifies what the administrator wants to do with this preference. The weight is the relative importance of each requirement, with the sum of all weights equal to 1. We combine all of the QoS requirements into a single set called a QoS specification. This specification is our indicator of how we want the system to perform overall. Table 1.2 shows an example QoS specification. Here, the administrator has specified that the system should give highest priority to minimizing the queue size and next highest to maximizing the output rate.

Table 1.2. An example QoS specification

Statistic	Quantifier	Weight
Input Queue Size	minimize	0.75
Output Rate	maximize	0.25

QoS requirements guide the adaptive execution by encoding a goal that the system should pursue. Without these preferences, the system will not have any criteria by which to determine how well or poorly a scheduler is performing. The requirements specify the desired behavior in relative terms, such as maximize the output rate or minimize the queue size(s) and their relative importance. Absolute requirements are too dependent on data arrival patterns and in fact in many cases are simply not achievable.

Scoring the Scheduling Algorithms. During execution, the Execution Engine will update the statistics that are related to the QoS requirements. Once updated, the system needs to decide how well the previous scheduler, S_{old}, has performed, and compare this performance to that of the other scheduling algorithms. To accomplish this, a function is developed to quantify how well an algorithm is performing for a particular QoS metric. First, the system calculates the mean and the spread of the values of each of the statistics specified in the service preferences for each category. Next, using the statistics from S_{old}, the relative mean of each of the statistics is calculated and then normalized.

The scoring function weighs the individual QoS metrics for relative importance (by multiplying by its corresponding weight w_i) and then normalizes the collected statistics for those metrics such that one algorithm can be ranked against another. We compute an algorithm's overall score, *scheduler_score*, by combining the relative performance for all of the QoS metrics into one QoS specification. The score assigned to an algorithm is not based solely on the previous time that it was used, but rather it is an exponentially smoothed average value over time. By comparing S_{old}'s *scheduler_score* with the scores for the other algorithms, the adapter is in a position to select the next most promising scheduling candidate.

Guidelines for Adaptation. Several guidelines are considered when using the scores to determine the next scheduling algorithm. Initially, all scheduling algorithms should be given a chance to "prove" themselves. Otherwise the decision would be biased against the algorithms that have not yet run. Therefore, at the beginning of execution, we allow some degree of exploration on the part of the adapter. Second, not switching algorithms periodically during execution (i.e., greedily choosing the next algorithm to run) could result in a poor performing algorithm being run more often than a potentially better performing one. Hence, we periodically explore alternate algorithms. Third, switching algorithms too frequently could cause one algorithm to impact the next and skew the latter's results. For example, using Chain could cause a glut of tuples in the input queues of the lower priority operators. If a batch-tuple strategy were to be run next, its throughput would initially be artificially inflated because of the way Chain operated on the tuples. More generally, when a new algorithm is chosen, it should be used for enough time such that its behavior is not significantly over-shadowed by the previous algorithm. For this, we empirically set delay thresholds before reassessing the potential of a switch to be undertaken.

Adaptive Selection Process. After each algorithm is given a score, the system needs to decide if the current scheduling algorithm performed well enough that it should be used again or if better performance may be achieved by changing algorithms. Considering Guideline 1 above, initially running each algorithm in a round robin fashion is the fairest way to start adaptive scheduling.

Once each algorithm has had a chance to run, there are various heuristics that could be applied to determine if it would be beneficial to change the scheduling algorithm. In an effort to consider all scheduling algorithms while still probabilistically choosing the best fit we adopted the Roulette Wheel strategy. This strategy assigns each algorithm a slice of a circular "roulette wheel" with the size of the slice being proportional to the individual's score. Then the wheel is spun once and the algorithm under the wheel's marker is selected to run next. This strategy was chosen because it is lightweight and does not cause significant overhead. In spite of its simplicity, this strategy is shown to significantly outperform single scheduling strategies (See Section 4.4). While this strategy may initially choose poor scheduling algorithms, over time it should fairly choose a more fit algorithm. The strategy also allows for a fair amount of exploration and thus it prevents one algorithm from dominating.

4.4 Summary of Performance Evaluation

An extensive experimental study on performance of ASSA can be found in [24]. We now briefly summarize the overall observations from this study:

- For the special case of a QoS specification consisting of only one single metric, ASSA indeed picks the one most optimal algorithm from all available algorithms in the library.

- For a complex QoS specification combining multiple requirements, ASSA also significantly improves performance over the run of any individual algorithm by working with some combination of algorithms.
- ASSA is able to react to QoS requirements even as they are changed at runtime by the system administrator.
- The overhead for the adaptation itself, i.e., the score calculation and the switching among algorithms, is shown to be negligible.
- ASSA is shown to be general, i.e., new scheduling solutions developed in the future can be plugged into the library of ASSA at any time.

5. Run-time Plan Optimization and Migration

Query plan optimization is critical for improving query performance. In a stream processing system, data is not present at the time when a query starts but is streaming in as time goes by. The long-running continuous queries have to withstand fluctuations in stream workload and data characteristics. Therefore, compared to static query processing system, a stream processing system has a much more pressing need to re-optimize the continuous query plans at run-time. A run-time plan optimization procedure takes three steps:

- ***Step 1:*** The optimizer decides *when* to invoke the optimization procedure. Too frequent optimization creates extra burden on the system resources, and too infrequent optimization may skip good optimization opportunities and hurt the system performance as well. The timing of the optimization is critical and needs to be carefully tuned. We present the solution in CAPE for this issue in Section 5.1.
- ***Step 2:*** The optimizer constructs a new query plan that is semantically equivalent to the currently running plan yet more efficient in terms of system resource consumption or performance. This is done by applying heuristics and rewriting rules to the old query plan based on gathered system statistics. We will discuss the optimization heuristics in CAPE in Section 5.2.
- ***Step 3:*** The optimizer migrates the old running query plan to the new plan that it has chosen. We refer to this process as *dynamic plan migration*. A novel feature of the run-time optimizer in CAPE, not yet offered by other stream engines, is that it can efficiently at run-time migrate a stream query plan even if it contains stateful operators. This dynamic plan migration step is the critical service that enables optimization to occur at runtime for stream query processing. We will discuss dynamic plan migration in Sections 5.3 and 5.4.

5.1 Timing of Plan Re-optimization

In the CAPE system, the plan optimization procedure can be invoked in two modes: the *periodic mode* and the *event-driven mode*.

In the periodic mode, the optimizer is invoked at a pre-specified optimization interval. As mentioned in Section 2, we adopt a heterogeneous-grained adaptation framework, in which each adaptation technique is assigned an adaptation interval based on its overhead and its perceived potential gain. Since the cost of the plan re-structuring is usually between the costs of the operator scheduling (which is very low effort) and the across-machine plan re-distribution (which is a more involved effort), so is its adaptation interval. The optimization interval can also be tuned dynamically based on certain changes in streaming data arrival rates or data distributions.

The CAPE system also identifies types of events that represent critical optimization opportunities that are unique to a stream processing system, as detailed in Section 5.2. Whenever one of the events occurs, it will trigger the optimizer running in the periodic mode to switch to the event-driven mode. The optimizer then immediately reacts to the triggering event by taking the corresponding actions typically in the form of applying customized heuristics. Once the optimization has completed, the optimizer returns back to its default mode, i.e., the periodic mode.

5.2 Optimization Opportunities and Heuristics

Many commonly used heuristics and rewriting rules in static database are also applicable for continuous query optimization. In CAPE, an optimizer in the periodic mode for example applies the following heuristics:

- The optimizer pushes down the select and project operators to minimize the amount of data traveling through the query plan, unless sharing of partial query plans dictates a delay of such early filtering.

- The optimizer merges two operators into one operator whenever possible, such as merging two select operators or merging a group-by operator with an aggregate operator, to reduce scan of data via shared data access and to avoid context switching.

- The optimizer switches two operators based on their selectivities and processing overhead. If Sel_i and Cost_i represent the selectivity and the processing cost of an operator op_i, then operators op_i and op_j with op_i consuming data produced by op_j can be switched if $(1 - Sel_i)/Cost_i > (1 - Sel_j)/Cost_j$.

We have also identified several new optimization opportunities that are unique to the stream processing system and its dynamic environment. We have incorporated these stream-specific optimization heuristics into CAPE as well.

Register/De-register Continuous Queries. A stream processing system often needs to execute numerous continuous queries at the same time. Sharing among multiple queries can save a large amount of system resources. In addition, queries may be registered into or de-registered from the system at any time. The above features can affect the decision of the optimizer. As an example, assume the system currently has one query plan with one select and one join operator, and after a while another query is registered which contains the same join as the first query but no select. In this case, the optimizer can pull up the select operator so the two queries can share the results from the join operator. Later if the second query is de-registered from the system, the optimizer may need to push the select down again. So the events of query registering/de-registering create new optimization opportunities that CAPE utilizes to trigger heuristics for query-sharing optimizations.

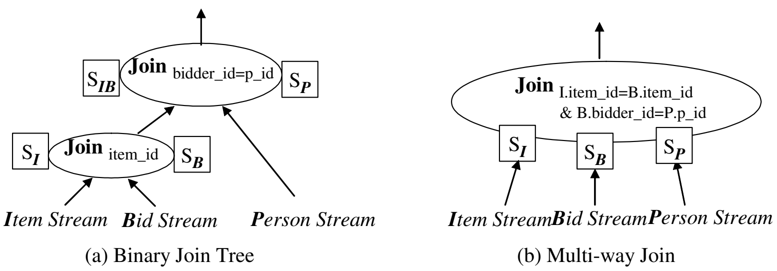


Figure 1.7. A Binary Join Tree and A Multi-way Join Operator.

Multi-Join Queries. Choosing the optimal order of multiple join operators has always been a critical step in query optimization. There are two popular methods to process a continuous query with multiple joins: a binary join tree as in traditional (static) databases, and a single *multi-way join* operator [13, 28]. For the two joins $Join_{I.item\_id=B.item\_id\ \&\ B.bidder\_id=P.p\_id}$. in the query defined in Figure 1.3, Figures 1.7 (a) and (b) depict a query plan composed of several binary joins and out of one multi-way join operator respectively. A binary join tree stores all intermediate results in its intermediate states, so no computation will be done twice. On the other hand, a multi-way join operator does not save any intermediate results, so all useful intermediate results need to be recomputed. A binary join tree saves CPU time by sacrificing memory, while a multi-way join sits on the opposite end of the spectrum. Dynamically switching between these two achieves different balancing between CPU and memory resources.

The CAPE system monitors the usage of CPU and memory while processing multiple joins. When the ratio of CPU to memory is greater or smaller than some pre-defined threshold, the optimizer enters the event-driven mode and switches between these two methods accordingly.
Punctuation-Driven Plan Optimization. The characteristics of punctuations available to the query also affect the plan structuring decision making. Considering the example query shown in Figure 1.3, the two join operators in the query plan are commutative, hence rendering two semantically equivalent plans. In the plan shown in Figure 1.3, some join results of the first join may not be able to join with any Person tuple in the second join because they don't satisfy the sliding window constraint applied to the Person stream. If we choose to do (Bid⋈Person) first, the sliding window constraint will drop expired tuples early so to avoid unnecessary work in the later join. However, in this plan, both join operators need to propagate punctuations on the Person.p_id attribute to help the group-by operator. This incurs more propagation overhead than the first plan in which only the second join needs to propagate punctuations. The optimizer in CAPE will choose the plan with less cost by considering these factors related to punctuation-propagation costs and punctuation-driven unblocking.

5.3 New Issues for Dynamic Plan Migration

Dynamic plan migration is the key service that enables plan optimization to proceed at runtime for stream processing. It is a unique feature offered by the CAPE system. Existing migration methods instead adopt a *pause-drain-resume* strategy that pauses the processing of new data, drains all old data from the intermediate queues in the existing plan, until finally the new plan can be plugged into the system.

The *pause-drain-resume* migration strategy is adequate for dynamically migrating a query plan that consists of only *stateless* operators (such as select and project), in which intermediate tuples only exist in the intermediate queues. On the contrary, a *stateful* operator, such as join, must store all tuples that have been processed thus far to a data structure called a *state* so to be able to join them with future incoming tuples. Several strategies have been proposed to purge unwanted tuples from the operator states, including window-based constraints [5, 15, 16, 20] and punctuation-based constraints [11, 26] (See Section 3). In all of these strategies the purge of the old tuples inside the state is driven by the processing of new tuples or new punctuations from input streams.

For a query plan that contains *stateful* operators, the draining step of the pause-drain-resume step can only drop the tuples from the intermediate queues, not the tuples in the operator states. Those could only be purged by processing new data. Hence there is a dilemma. CAPE offers a unique solution for stateful runtime plan migration. In particular, the we have developed two alternate migration strategies, namely *Moving State Strategy* and *Parallel Track Strategy*. These strategies are now introduced below.

5.4 Migration Strategies in CAPE

Below, the term *box* is used to refer to the plan or sub-plan selected for migration. The migration problem can then be defined as the process of transferring an old box containing the old query plan to a new box containing the new plan. The old and new query plans must be equivalent to each other, including identical sets of box input and output queues, as shown in Figure 1.8.

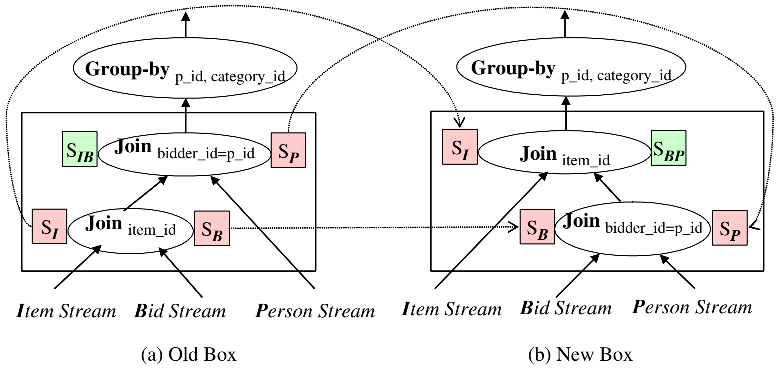


Figure 1.8. Two Exchangeable Boxes

Moving State Strategy. The moving state strategy first pauses the execution of the query plan and drains out tuples inside intermediate queues, similar to the above *pause-drain-resume* approach. To avoid loss of any useful data inside states, it then takes a three-step approach to safely move old tuples in old states directly into the states in the new box. These steps are *state matching*, *state moving* and *state recomputing*.

State matching determines the pairs of states, one in the old and one in the new box, between which tuples can be safely moved. If two states have the same state ID, which are defined as the same as its tuples' schema, we say that those two states are *matching states*. In Figure 1.8, states (S_I, S_B, S_P) exist in both boxes and are matching states. During the step of *state moving*, tuples are moved between all pairs of matching states. This is accomplished by creating a new cursor for each matching new state that points to its matching old state, such that all tuples in the old state are shared by both matching states. The cursors for the old matching states are then deleted. In the *state recomputing* step, the unmatched states in the new box are computed recursively from the leaves upward to the root of the query plan tree. Since the two boxes have the same input queues, the states at the bottom of the new box always have a matching state in the old box. Using the example shown in Figure 1.8, we have identified an unmatched state S_{BP} in the new box. We can recompute S_{BP} by joining the tuples in S_B and S_P.

Once the moving state migration starts, no new results are produced by the targeted migration box inside the larger query plan until the migration process is finished. Of course, the remainder of the query plan continues its processing. This way the output stream may experience a duration of temporary silence. For applications that desire a smooth and constant output, CAPE offers a second migration strategy called the parallel track strategy. This alternate strategy can still deliver output tuples even during migration.

Parallel Track Strategy. The basic idea for the *parallel track migration strategy* is that at the migration start time, the input queues and output queue are connected and shared between the old box and the new box. Both boxes are then being executed in parallel until all old tuples in the old box have been purged. During this process, new outputs are still being continually produced by the query plan. When the old box contains only *new* tuples, it is safe to discard the old box. Because the new box has been executed in parallel with the old box from the time the migration first starts, all the new tuples now in the old box exist in the new box as well. So if the old box is discarded at this time, no useful data will be lost.

A valid migration strategy must ensure that no duplicate tuples are being generated. Since the new box only processes *new* tuples fed into the old box at the same time, all output tuples from the new box will have only *new* sub-tuples. However, the old box may also generate the all-new tuple case, which may duplicate some results from the new box. To prevent this potential duplication, the root operator of the old box needs to avoid joining tuples if all of them are *new* tuples. In this way, the old box will not generate the all-new tuples.

Cost of Migration Strategies Detailed cost models to compute the performance overhead as well as migration duration periods have been developed [31]. This enables the optimizer in CAPE to compute the cost of these two strategies based on gathered system statistics, and then dynamically choose the strategy that has the lowest overhead at the time of migration.

The two migration strategies have been embedded into the CAPE system. While extensive experimental studies comparing them can be found in [31], a few observations are summarized here:

- Given sufficient system resources, the moving state strategy tends to finish the migration stage quicker than parallel track.
- However, if the system has insufficient processing power to keep up with the old query plan, the parallel track strategy, which can continuously output results even during the migration stage, is observed to have a better output rate during the migration stage.

- Overall, the costs of both migration strategies are affected by several parameters, including the stream arrival rates, operator selectivities and sizes of the window constraints.

6. Self-Adjusting Plan Distribution across Machines

While most current stream processing systems (STREAM [20], TelegraphCQ [6], and Aurora [1]) initially have employed their engine on a single processor, such an architecture is bound to face insurmountable resource limitations for most real stream applications. A distributed stream architecture is needed to cope with the high workload of registered queries and volumes of streaming data while serving real-time results [7, 22]. Below we describe our approach towards achieving a highly scalable framework for distributed stream processing, called Distributed CAPE (D-CAPE in short) [25].

6.1 Distributed Stream Processing Architecture

D-CAPE adopts a Shared-Nothing architecture shown to be favorable for pipelined parallelism [10]. As depicted in Figure 1.9, D-CAPE is composed of a set of query processors (in our case CAPE engines) and one or more distribution managers. We separate the task of distribution decision making ("control") from the task of query processing to achieve maximal scalability. This allows all query engines to dedicate 100% of their resources to the query processing. The distribution decision making is encapsulated into a separate module called the Distribution Manager. A Distribution Manager typically resides on a machine different from those used as query processors, though this is not mandatory.

D-CAPE is designed to efficiently distribute query plans and continuously monitor the performance of each query processor with minimal communication between the controller and query processors. At runtime, during times of heavy load or if it is determined by D-CAPE that reallocation will boost the performance of the system, query operators are seamlessly reallocated to different query processors. By multi-tiering distribution managers, we can exploit clusters of machines in different locations to process different workloads.

As illustrated in Figure 1.9, the Distribution Manager is composed of four core components and three repositories. The *Runtime Monitor* listens for periodic statistics reported by each query processor, and records them into the *Cost Model Table*. These run-time statistics form the basis for determining the *workload* of processors and for deciding operator reallocation. The *Connection Manager* is responsible for physically sending a sequence of appropriate connection messages according to our connection protocol to establish operators to machines. The *Query Plan Manager* manages the query plans registered by the user in the system. The *Distribution Decision Maker* is responsible for deciding

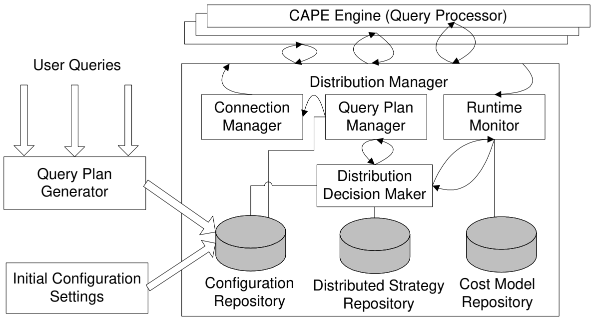


Figure 1.9. Distribution Manager Architecture

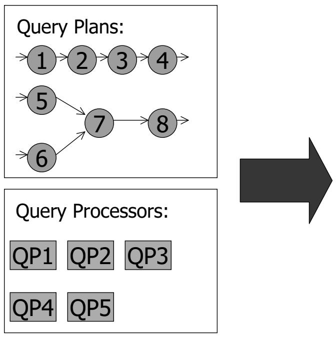
how to distribute the query plans. There are two phases to this decision. First, an initial distribution is created at startup using static information about query plans and machine configurations. Second, at run-time query operators are re-allocated to other query processors depending on how well the query processors are perceived to be performing by the Decision Maker.

The Distribution Manager is designed to be light-weight. Only incremental changes of the set of query plans are sent to the query processors to reduce the amount of time the Distribution Manager spends communicating with each processor at run-time. Our empirical evaluation of the Distribution Manager shows that the CPU is rarely used, primarily, only when calculating new distribution ([25]). Furthermore, the network traffic the DM creates is minimal. In short, this design of D-CAPE is shown to be highly scalable.

6.2 Strategies for Query Operator Distribution

Distribution is defined as the physical layout of query operators across a set of query processors. The initial distribution of a query plan based only on static information at query startup time is shown to directly influence the query processing performance. The initial distribution depends only on two pieces of information: the queries to be processed and the machines that have the potential to do the work. The Distribution Decision Maker accepts both the description of the query processors and query plans as inputs and returns a table known as a *Distribution Table* (Figure 1.10). This table captures the assignment of each query plan operator to the query processor it will be executing on.

The methodology behind how the table is created depends on the Distribution Pattern utilized by the Decision Maker. This allows us the flexibility to easily plug in any new Distribution Pattern into the system. Two distribution algorithms that were initially incorporated into D-CAPE are:



Distribution Table

Operator	QP
Operator 1	QP 1
Operator 2	QP 1
Operator 3	QP 2
Operator 4	QP 3
Operator 5	QP 4
Operator 6	QP 4
Operator 7	QP 4
Operator 8	QP 5

Figure 1.10. Distribution Table

- **Round Robin Distribution.** It iteratively takes the next query operator and places it on the still most "available" query processor, i.e., the one with the fewest number of assigned operators. This ensures fairness in the sense that each processor must handle an equal number of operators, i.e., an equivalent workload.

- **Grouping Distribution.** It takes each query plan and creates sub-plans for each query by maximally grouping neighboring operators together so to construct connected subgraphs. This aims to minimize network connections since adjacent operators with joint “pipes” are for the most part kept on the same processor. Then it divides these connected subgraphs among the available query processors.

After a distribution has been recorded into the distribution table, then the Connection Manager distributes the query plan among the query processors. Once the Connection Manager has completed the initial setup, query execution can begin on the cluster of query processors, now no longer requiring any interaction from the Distribution Manager.

6.3 Static Distribution Evaluation

Our work is one of the first to report experimental assessments on a working distributed stream processing system. For details, the readers are referred to [25]. Below we list a summary of our findings:

- D-CAPE effectively parallelizes the execution of queries, improving performance even for small query plans and for lightly loaded machines, without ever decreasing performance beyond the central solution.

- The total throughput is improved when using more query processors over that when using less processors. This is because we can assign a larger CPU time slice to each operator.
- The larger the query plans in terms of number and type of operators, the higher a percentage of performance improvement is achievable when applying distribution (on the order of several 100%). In many cases while the centralized CAPE fails due to resource exhaustion or a lack of processing power, the distribution solution continues to prevail.
- The Grouping Distribution generally outperforms the Round Robin by several fold. In part, this can be attributed to the Grouping Distribution being "connection-aware", i.e., due to it minimizing the total amount of data sent across the network and the number of connections.

6.4 Self-Adaptive Redistribution Strategies

When we first distribute a query plan, we only know static information such as shape and size of the query plan, the number of input streams, and data about the layout of the processing cluster. Dynamic properties such as state size, selectivity, and input data rates are typically not known until execution. Worse yet, these run-time properties tend to change over time during execution.

Due to such fluctuating conditions, D-CAPE is equipped with the capability to monitor in a non-obtrusive manner its own query performance, to self-reflect and then effectively redistribute query operators at run-time across the cluster of query processors. We will allow for redistribution among *any* of the query processors, not just adjacent ones, in our computing cluster.

Towards this end, we require a measure about the relative *query processor workload* that is easily observable at runtime. One such measure we work with is the rate at which tuples are emitted out of each processor onto the network. This dynamically collected measure is utilized by the on-line redistribution policy in D-CAPE for deciding *if*, *when* and *how* to redistribute.

Algorithm 1 Overall Steps for Redistribution.

1: $costTable \leftarrow costModel.getTable()$
2: $maxCost \leftarrow costTable.getMaxCost()$
3: $minCost \leftarrow costTable.getMinCost()$
4: **if** $max - min > redistributionPercent$ **then**
5:   **while** $!valid(newDistribution)$ **do**
6:     $newDistribution \leftarrow RedistributionPolicy.redistribute()$
7:   **end while**
8:   $differenceTable \leftarrow newDistribution - currentTable$
9:   $connectNewDistribution(differenceTable)$
10:   $currentTable \leftarrow newDistribution$
11: **end if**

While new policies can be easily plugged into D-CAPE framework, one of the redistribution policies we found to be effective in D-CAPE is called the *degradation redistribution policy*. This policy alleviates load on machines that have shown a degradation in cost since the last time operators were allocated to the machine. If the cost has degraded beyond a certain threshold, we aim to stop this degradation by moving the 'most costly' operators to other query processors. This policy gives higher preference to those operators that will remove a network connection from the overall distribution of operators — driven by our empirical observation of the direct impact of higher connection loads on the resulting system performance.

In general, any of the redistribution policies in D-CAPE, including the degradation policy above, employs the steps detailed in Algorithm 1 for realizing the desired re-distribution. These steps use our our handshake protocol between the Distribution Manager and the designed for moving query operators between processors The cost of moving an operator has been shown to be negligible in our system due to this carefully designed connection protocol. Intuitively, since we create the connections for the data to flow *before* we start sending the data, we are able to "flip a switch" and in the eyes of the query processor, turn off one operator and turn it on on another machine instantaneously.

6.5 Run-Time Redistribution Evaluation

Our results confirm that dynamic redistribution is a viable and even necessary option for handling the performance degradation observed at runtime. Our results illustrate that redistribution can tune the execution if the initial distribution is found to be bad or if it turns bad over time. While a detailed experimental study can be found in [25], key experimental observations are shown below.

- The overhead for redistributing an operator or even a complete sub-plan across machines is found to be negligible. This allows D-CAPE to perform reallocation at a high frequency, if deemed necessary.

- Even strategies that achieve good initial distribution patterns such as the Grouping Distribution can still experience a further performance boost when undergoing runtime redistribution.

- On-line operator re-allocation has been shown to improve performance over time compared to only working with static distributions.

- Initial static distribution decisions significantly affect the performance in the long-term even when continuing to dynamically apply reallocation.

7. Conclusion

In this chapter, we have presented a streaming query processing system named CAPE. We reviewed the query optimization techniques that are unique to

CAPE, including the heterogeneous-grained adaptation framework and constraint-exploiting techniques. The adaptation technologies we illustrated include the adaptive operator execution logic, self-healing adaptive operator scheduling, runtime query plan re-optimization and migration, and self-adjusting query plan distribution across machines. CAPE employs these technologies to effectively evaluate continuous queries in highly dynamic streaming environments.

References

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, August 2003.

[2] A. Arasu, S. Babu, and J. Widom. CQL: A language for continuous queries over streams and relations. In *DBPL*, pages 1–19, Sep 2003.

[3] 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.

[4] S. Babu and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transactions on Database Systems*, 39(3), Sep 2004.

[5] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, August 2002.

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, pages 269–280, Jan 2003.

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

[8] D. Carney and U. Cetintemel and A. Rasin et al. Operator scheduling in a data stream manager. In *VLDB*, pages 838–849, 2003.

[9] A. Dan and D. Towsley. An approximate analysis of the lru and fifo buffer replacement schemes. In *ACM SIGMETRICS*, pages 143–152, 1990.

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

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

[12] L. Ding, E. A. Rundensteiner, and G. T. Heineman. MJoin: A metadata-aware stream join operator. In *DEBS*, June 2003.

[13] L. Golab and M. T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, Sep 2003.

[14] P. Haas and J. Hellerstein. Ripple joins for online aggregation. In *ACM SIGMOD*, pages 287–298, June 1999.

[15] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*, pages 297–308, Sep 2003.

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

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

[18] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[19] M. F. Mokbel, M. Lu, and W. G. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *ICDE*, pages 251–262, Mar/Apr 2004.

[20] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, pages 245–256, Jan 2003.

[21] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB Demo*, Aug/Sep 2004, to appear.

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

[23] Stanford University. Stream query repository. http://www-db.stanford.edu/stream/sqr/, Dec 2002.

[24] T. Sutherland, B. Pielech, and E. A. Rundensteiner. Adaptive scheduling framework for a continuous query system. Technical Report WPI-CS-TR-04-16, Worcester Polytechnic Institute, April 2004.

[25] T. Sutherland and E. A. Rundensteiner. D-cape: A self-tuning continuous query plan distribution architecture. Technical Report WPI-CS-TR-04-18, Worcester Polytechnic Institute, July 2004.

[26] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, May/June 2003.

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

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

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

[30] J. Zahorjan and C. McCann. Processor scheduling in shared memory multiprocessors. In *ACM SIGMETRICS*, pages 214–225, 1990.

[31] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *ACM SIGMOD*, June 2004.