Managing Parallelism for Stream Processing in the Cloud

Nathan Backman, Rodrigo Fonseca, Uğur Çetintemel
Brown University
{backman, rfonseca, ugur}@cs.brown.edu

Abstract

Stream processing applications run continuously and have varying load. Cloud infrastructures present an attractive option to meet these fluctuating computational demands. Coordinating such resources to meet end-to-end latency objectives efficiently is important in preventing the frivolous use of cloud resources. We present a framework that parallelizes and schedules workflows of stream operators, in real-time, to meet latency objectives. It supports data- and task-parallel processing of all workflow operators, by all computing nodes, while maintaining the ordering properties of sorted data streams. We show that a latency-oriented operator scheduling policy coupled with the diversification of computing node responsibilities encourages parallelism models that achieve end-to-end latency-minimization goals. We demonstrate the effectiveness of our framework with preliminary experimental results using a variety of real-world applications on heterogeneous clusters.

Categories and Subject Descriptors H.2.4 [*Database Management*]: Systems – query processing, parallel databases

General Terms Design, Performance, Experimentation

Keywords Stream processing, Parallelism management, Heterogeneous clusters

1. Introduction

Many stream processing applications are time-critical, producing urgent results which require immediate attention. Examples include automated stock trading, real-time video processing, and network traffic monitoring, and their utility rapidly decreases as results are delayed. In this context, minimizing end-to-end latency critically depends on parallelization, workload balancing, and operator scheduling.

Distributed stream processing engines often employ rigid operator allocation strategies: they allocate the entire workload of a subset of operators [2, 5, 6] and/or a portion of the workload from a single but expensive operator [1, 3] to each computing node. "Pinning" nodes to operators, in this way, results in coarse-grained workload balancing — on the scale of nodes to operators. Additionally, pinning nodes to a specific set of operators prevents those nodes from contributing to other operators across the workflow in times of need to withstand stream or resource volatility.

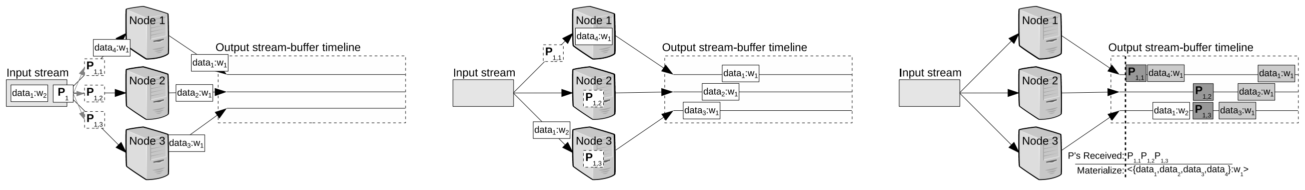
Statically employing data, task, and/or pipeline parallelism by pinning operators to nodes may not efficiently meet end-to-end latency objectives. The forms of parallelism enacted by nodes, at any moment, should be motivated by latency-oriented objectives. To this end, we present a stream processing workload allocation and scheduling framework to manage parallelism in real-time. Instead of pinning nodes to operators, we *partition* all operator workloads for parallel processing while maintaining the integrity of sorted streams. We then distribute those partitions across the set of all computing nodes to encourage forms of parallelism which reduce latency (§3), and employ a latency-oriented, workflow-wide operator scheduler to concentrate the nodes' collective efforts on the portions of the workflow that are most important to minimizing latency at any given time (§2). To determine the level of parallelism applied to each workflow operator, we use a simulation-based search heuristic (§4) to discover stream partitioning granularities which result in low end-to-end latencies.

Our framework supports the concurrent use of heterogeneous and non-dedicated computing nodes while providing resilience to stream and resource volatility through the diversification of computing responsibilities. This is an important step towards pushing time-critical stream applications into the cloud; the acquisition and use of cloud resources must be motivated by end-to-end latency objectives when facilitating time-critical stream processing applications.

2. Supporting & Guiding Parallelism

Task and pipeline parallelism are normally achieved by pinning nodes to different workflow operators. Achieving data parallelism requires partitioning the input stream of an operator into distinct and independent sub-streams for nodes to process concurrently. We define these sub-streams as *partitions* of an operator's workload and our framework supports

HotCDP 2012 - 1st International Workshop on Hot Topics in Cloud Data Processing
April 10, 2012, Bern, Switzerland.



(a) (b) (c)

Figure 1: Punctuation replication (a), propagation (b), and merging (c) enable stream re-ordering for data-parallel processing.

various partitioning strategies to apply to the data values (e.g., range, hash, and spatial partitioning for multidimensional data). Operators that cannot be executed in parallel will have only a single partition.

2.1 Maintaining Stream Order

When stream applications analyze the temporal relation between data in a stream, they do so by processing *windows* (temporal subdivisions of a stream). Windowed data streams have been studied for years but not in the context of data-parallel stream processing. In such scenarios, nodes can consume and emit data from a parallelized operation at different rates; if care is not taken, the merged output stream of an operator may not retain its sorted order, thereby ruining the integrity of the stream.

We define a novel adaption of punctuations [4] to maintain stream order in the midst of data-parallel operations. Punctuations are "control" tuples which are inserted into a stream amidst data to denote window boundaries. We replicate punctuations at stream partitioning points within a workflow (Figure 1a) and send copies to nodes that will receive the partitions (encoding the total replica count in each copy). Those nodes will processes data from their partitions and propagate the punctuations to the merged output stream (Figure 1b). We buffer the data in the merged output stream while waiting for the receipt of all replicas. Upon the arrival of the last replica (as determined by the encoded replica count), the window can be re-materialized and propagated along the workflow (Figure 1c). Inserting materialized windows back into the workflow results in a sorted stream.

2.2 Workflow-Wide Scheduling

To increase the impact of a node's operator scheduler, we diversify the types of workloads it can receive. This allows each node's scheduler to better identify regions of the workflow with the highest priority to execute first. With the same scheduling objectives, nodes will collectively target high-priority areas helping to quickly mitigate the effects of bottlenecks and hotspots.

Our latency-oriented scheduler prioritizes data by age, choosing to process the "oldest" data a node has. This decreases the average latency of each result — conversely, spending time processing "new" data would result in a higher latency for the old data that was not selected. Therefore, nodes will collectively concentrate on the "backside" of the workflow, nearest to the outputs, when possible. This is where we will see data parallelism become prevalent and also task parallelism (among parallel branches of operators having data of similar priorities). Pipeline parallelism, on the other hand, will only occur as a last resort in which case we must force a node to process new data, having no old data left to process, and while other nodes still have the means to process old data.

Nodes collectively asserting a common, workflow-wide scheduling strategy means that:

1. Nodes will be less likely to sit idly waiting for data as their supply of data has been diversified to include all operator workloads.
2. The operator scheduling scope of a computing node has been increased such that it can make better scheduling decisions. Thus, high-priority data are processed sooner.
3. Workflow bottlenecks no longer impact (subsequently overloading) only a subset of computing nodes but can now be addressed and relieved collectively by all nodes.

This workflow-wide scheduling technique only works when we diversify a node's processing opportunities. The allocation of operator workload partitions, along with the scheduling objective, is what allows the scheduler to guide the types of parallelism that occur. Therefore, we must efficiently balance workload partitions across the set of available computing nodes.

3. Workload Balancing

Bin-packing algorithms have traditionally been the tool of choice for balancing the load of a parallelized operation onto a set of computing nodes [1, 3]. These algorithms assign workload partitions a "height," proportional to the processing time they require, and are "packed" into computing nodes which represent bins. These bin-packing strategies migrate partitions between bins with the goal of minimizing the cumulative partition height of the tallest bin/node that spends the most time processing. The minimization objective intends to mitigate the effect of stragglers by reallocating partitions to nodes which are likely to finish processing them earlier.

3.1 Balancing Both Load and Parallelism

When balancing the set of *all* operator workloads, however, the standard bin-packing procedure alone is not very effective at minimizing end-to-end latency. Consider two identically expensive operators, A and B, in a serial chain and two identical nodes to distribute that work to. We can simply pack all partitions of A onto one node and all of B's onto the other. While the load would be perfectly balanced, the resulting pipeline parallelism would de-couple the latency minimization efforts of the nodes' schedulers.

A better solution would involve each node taking half of each operator's partitions. Load would again be perfectly balanced but each node would now have the opportunity to focus on high-priority data. Balancing load alone does not encourage the goal of minimizing end-to-end latency; it is necessary to distribute workloads so that operator schedulers can leverage the types of parallelism that will reduce latency.

In fact, it is by modifying the scope of the traditional bin-packing algorithms that we are able to achieve workflow-wide, latency-oriented scheduling. Consider these identical bin-packing strategies of differing scopes:

- **Global Bin-Packing:** Packing all operator workloads together across all available computing nodes.
- **Local Bin-Packing:** Packing each operator's workload across all nodes, independently of all other operator workloads.
- **Tiered Bin-Packing:** Packing simultaneously the workloads of operators which are the same distance from the outputs (and therefore of relatively similar priorities).

The Global strategy balances loads the best as it can access all workloads simultaneously for fine-grained placement. However, it does not discourage pipeline parallelism nor does it encourage the diversification of responsibilities for resilience to stream volatility. The Local strategy provides exactly this kind of diversification as it encouraging nodes to receive partitions from each operator, thereby supporting workflow-wide scheduling. Likewise, the Tiered strategy provides diversification between operators of differing priorities while allowing for better load balancing opportunities across operators of similar priorities.

In practice, the Local and Tiered strategies only differ when faced with non-trivial workflows containing parallel branches of operators. In such workflows, Tiered will always have the upper-hand for balancing load (with the opportunity to balance more at once), but Local will be more resilient to stream and resource volatility as it enforces strict diversification of processing opportunities. Depending on the characteristics of the application and its workload, both algorithms will be useful for objectively reducing end-to-end latency.

3.2 Evaluating & Distributing Assignments

The Global strategy is a centralized algorithm (running once and disseminating the results to all parallelized operators), whereas the Local and Tiered strategies consist of multiple bin-packing invocations with each executed independently. The new partition assignments that are produced can be implemented at will for non-windowed operators. For window operators, however, new partition assignments can only be enacted on window boundaries to ensure that all data pertaining to a window is received at the same node. In our tests, we evaluate and disseminate partition strategies for window-based operators after a variable number of windows have been evaluated (e.g., every 10 windows) while we use time-based evaluation and dissemination of partition assignments for non-windowed operations (e.g., every 5 seconds).

3.3 Balancing Work Across Heterogeneous Nodes

When using heterogeneous computing nodes, the concept of partition height, for bin-packing, becomes relative. Partition height corresponds to time spent processing, thus, the size of a partition will be proportional to a node's ability to process it. We profile the performance of participating node types (such as those that can be acquired in cloud infrastructures) by observing the rate at which they are able to process the data corresponding to the operations in the workflow. Using such observed statistics, and by having known quantities of data within each partition, we can adjust the height of a partition relative to the bin/node it might be assigned to.

Not only can this strategy be used to characterize the performance of various node types, with regard to workflow operators, but we also apply the same strategy to leverage non-dedicated computing nodes which allow shared-processing of other applications. In such a scenario, individual node profiling must continue over time to monitor external load fluctuations to adequately balance partitions across those computing resources.

4. Simulation-Based Latency Evaluation

Workflow-wide scheduling requires that computing nodes be allocated work from various operators. Picking the partition granularity for each operator's workload is not a trivial task. The partition granularities and subsequent partition-to-node allocations we pick must be motivated by the requirement to achieve low end-to-end latencies.

4.1 Simulation Environment

We estimate the impact of potential partitioning schemes through discrete event simulation. Our simulator replicates the workflow layout, data arrival rates, observed node processing speeds, stream partitioning, workload balancing, and network transmission times to produce end-to-end latency measurements. Repeated invocations of the simulator take a partition granularity for each operator as input. The simulator balances the resulting partitions across the simulated nodes and then propagates a stream of data through the workflow to record end-to-end latencies.

4.2 Heuristic Search via Simulation

The produced latencies provide a means to navigate a search space of possible partitioning configurations. We use a genetic algorithm to methodically explore this space for configurations which produce low latencies. Our genetic algorithm fosters a population of candidates such that each candidate describes the partition granularity for each operator. The set is initially randomly generated and then we:

1. Simulate and rank candidates by their latencies.
2. Prune off a percentage of the poorest performing candidates to cull the population.
3. Breed a percentage of the best performers (randomly sampling partition granularities from "parent" nodes to produce a child) to increase the population.
4. Clone and mutate (randomizing a subset of partition granularities) a percentage of the best performers to increase the population.
5. Randomly generate new candidates until the original population size is restored.

We loop this algorithm to exploit good features of winners while maintaining randomness to avoid local minima. The algorithm continues until a stopping criterion has been reached (e.g., time spent searching, iterations elapsed, minimal additional benefit found, a combination of these methods, etc.). Then the highest ranked configuration can be deployed onto the real-time system.

Candidate simulations are independent and easily distributed across nodes; they can run while nodes are idle or may actively compete for processing time. Static workloads will not require re-evaluation of partitioning granularities and therefore the regularity of re-evaluation should be proportional to the volatility of each application's workload.

5. Experimental Results

We demonstrate our framework's ability to withstand both stream and resource volatility, distribute and balance workloads, and automatically pick operator partition granularity all with respect to latency objectives.

5.1 Withstanding Volatility

We used a moving average stock price application to show resiliency to computing resource and stream workload volatility. The workflow consists of an operator that filters non-relevant stock symbols followed by an aggregation operator to calculate a moving average per stock symbol. The application was deployed on a dual Intel Xeon E5410 computer, leveraging a total of 6 physical cores to act as computing nodes. The computational complexity of both operators was artificially increased to emphasize the impact of stream and resource fluctuations.

This experiment contrasts the use of unpinned computing nodes (able to process data from any operator) with pinned computing nodes. Because a variety of configurations of pinned nodes could be used with two operators, we enumerated and included all configurations. Our test included two different fluctuations over the testing interval: 1) a 10-second volume burst at the 10 second mark; and 2) a 10-second cycle-stealing event (simulating external workloads) occurring on one core at the 40 second mark. The results of this experiment are depicted in Figure 2.

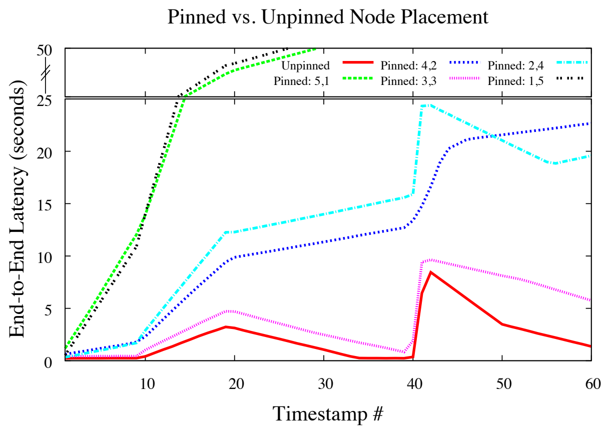


Figure 2: Unpinned nodes better withstand workload surges (time [10,20]) and mitigate bottlenecks caused by external workloads (time [40,50]).

The allocation of 3 nodes between each operator (3,3) performed the best out of the pinned strategies when the application was stable and during the stream burst, however the (2,4) allocation was able to more quickly overcome the node-overload situation. In all cases, the unpinned node strategy resulted in lower overall latencies. The unpinned framework allows for the automatic adjustment of parallelism and node responsibilities to adapt to variances of load across resources and across operators. The best we can do with pinned strategies is to jump between configurations (coarse-grained load balancing) to something more suitable at the moment.

5.2 Modifying Scope for Load Balancing

This test utilized an image processing application (i.e., a people counting application) in which two video streams (providing background and foreground imagery) were independently processed, then joined, and finally post-processed. For the purpose of load balancing, we used 3 heterogeneous computers: a quad-core 2.4 GHz Intel, a dual-core 2.0 GHz AMD, and a single-core 1.6 GHz AMD processor.

By varying bin-packing scope, we achieve different goals. Figure 3 shows that Global, while most balanced, exhibits high latencies. Local and Tiered perform similarly, as the two objectively try to reduce latency through specific regions of the workflow. Tiered slightly outperformed Local via better load balancing despite lesser diversification of node responsibilities — this workload was stable without load fluctuations, lending itself to the objectives of Tiered.

5.3 Picking Parallelism Granularity

We contrived a large, asymmetric workflow of 20 operators (7 attached to inputs, 6 joins, and both parallel and serialized segments) with varying complexities. We used 15 comput-

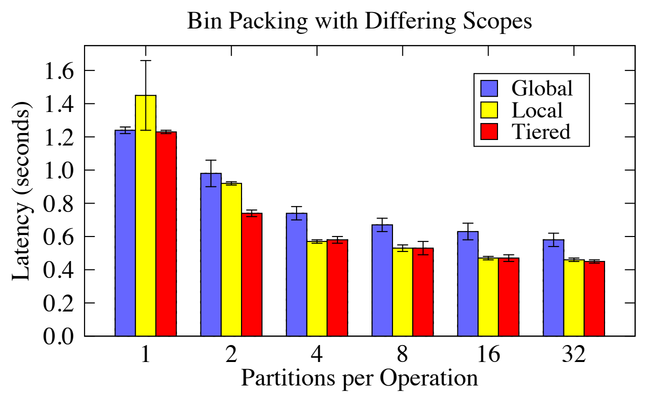


Figure 3: Local and Tiered load balancing scopes improve latency by encouraging diversified data parallelism.

ers (5 of the quad-core processors, 5 of the dual-cores, and 5 single-core processors) and issued the workflow and set of resources to our search algorithm to discover good partitioning granularities.

Figure 4a shows the progress of the heuristic search to find better results over multiple iterations when starting from scratch (a randomized configuration). We additionally show that, given a seed (a previously deployed partitioning strategy), the search is able to much more quickly converge to lower latencies.

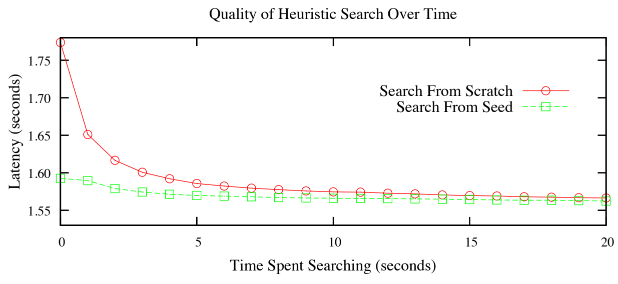
For heuristic searches, it is difficult to prove optimality. Therefore, we contrived an additional workload, such that the optimal partitioning strategies and workload assignments were known, to see how well the genetic algorithm would approximate the optimal results. The workflow simply consisted of a long chain of 10 identical operators. While the workflow was simple, the genetic algorithm is completely agnostic to the structure of a workflow or the similarity of operators. The results from the approximation of the optimal plan are shown in 4b. Even though none of the simulated configurations exactly matched the optimal configuration, the genetic algorithm was able to, on average, come within 2.5 milliseconds of the optimal plan.

6. Conclusion

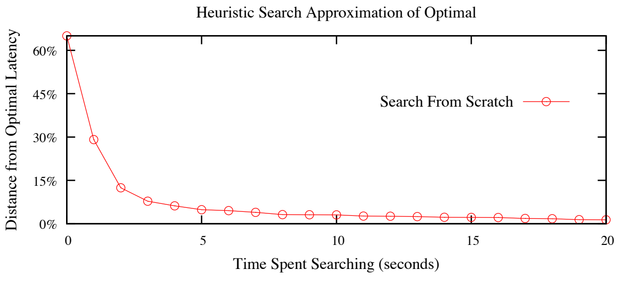
We have shown that parallelism can be orchestrated to objectively improve end-to-end latency by diversifying processing opportunities and enabling workflow-wide scheduling.

Our framework makes data-parallel processing a possibility for window-based stream applications and supports the use of heterogeneous and non-dedicated computing nodes to meet end-to-end latency objectives. Our workflow-wide scheduler allows nodes to collectively focus on the areas of a workflow deemed a priority while gracefully withstanding stream and resource fluctuations.

Additionally, our simulation-based search can easily be extended to simulate the incremental addition/removal of known (previously profiled) computing node types to leverage the elastic properties of the cloud. Also, our framework



(a)



(b)

Figure 4: Our simulation-based search discovers low-latency combinations of operator partitioning granularities. Seeding the search speeds this process up.

could be used to support workload allocation and scheduling policies that are targeted towards other stream application objectives such as throughput maximization.

References

[1] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA, 1999. ACM Press.

[2] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 49, Washington, DC, USA, 2006. IEEE Computer Society.

[3] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An adaptive partitioning operator for continuous query systems. Technical Report UCB/CSD-2-1205, U.C. Berkeley, 2002.

[4] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. on Knowl. and Data Eng.*, 15(3):555–568, 2003.

[5] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the Borealis stream processor. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 791–802. IEEE Computer Society, 2005.

[6] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *VLDB '06: Proceedings of the 32nd international conference on Very Large Data Bases*, pages 775–786. VLDB Endowment, 2006.