Faucet: a user-level, modular technique for flow control in dataflow engines

Andrea Lattuada
Systems Group, ETH
Zuerich, Switzerland
andreal@
student.ethz.ch
andrea@lattuada.me

Frank McSherry
Unaffiliated
fmcsherry@me.com

Zaheer Chothia
Systems Group, ETH
Zuerich, Switzerland
zchothia@inf.ethz.ch

ABSTRACT

This document presents Faucet, a modular flow control approach for distributed data-parallel dataflow engines with support for arbitrary (cyclic) topologies. When compared to existing *backpressure* techniques Faucet has the following differentiating characteristics: (i) the implementation only relies on existing progress information exposed by the system and does not require changes to the underlying dataflow system, (ii) it can be applied selectively to certain parts of the dataflow graph, and (iii) it is designed to support a wide variety of use cases, topologies and workloads.

We demonstrate Faucet on an example computation for efficiently determining a cyclic join of relations, whose variability in rates of produced and consumed tuples challenges the flow control techniques employed by systems like Storm, Heron, and Spark. Our implementation, prototyped in Timely Dataflow, introduces flow control at critical locations in the computation, keeping the computation stable and resource-bound while introducing at most 20% runtime overhead over an unconstrained implementation.

Our experience is that the information Timely Dataflow provides to user logic is sufficient for a variety of flow control and scheduling tasks, and merits further investigation.

CCS Concepts

•Software and its engineering → Scheduling; Data flow architectures;

1. INTRODUCTION

Dataflow system such as Storm[5] or Timely Dataflow[7][8] represent computation as a graph of operators connected with communication channels; each operator repeatedly consumes some number of input tuples, and produces some number of output tuples. Due to the generality of the programming model, input and output rates need not be tied by a simple, statically determined function; this prevents the construction of a static schedule which guarantees that the number of tuples in-flight on each communication channel is kept small. In order to prevent memory exhaustion, most modern systems employ dynamic flow control techniques, often mandated by the most common use case for the system.

Storm[5], Heron[4] and Spark[2] employ variations of a technique that relies on overflow signals from various components of the system to limit the data rate of the most upstream data sources, typically the inputs to the computation. While effective when dealing with an increased input data rate, this technique is not sufficient to protect the system from a surge in the number of messages produced by one of the internal operators in response to a single tuple from its input.

The edge-by-edge *backpressure* technique employed in IBM Streams[3], Flink[1], and Reactive Streams[10] accomplishes stable operation in many common cases but intrinsically introduces the risk of deadlock in cyclical topologies; its effective operation relies on a system-level implementation that propagates information upstream on a *backpressure* graph which corresponds to the full dataflow graph with the edges inverted.

The generality of Timely Dataflow's programming model enables a variety of programs with widely different workload characteristics. For this reason, we strived to devise a flexible technique that could be appropriately tuned and successfully applied when dealing with a diverse set of use cases, ranging from streaming analytics to scalable database algorithms.

2. AN EXAMPLE TOPOLOGY

As a motivating example of a non-analytics computation that benefits from Faucet, we describe a dataflow implementation[6] of a specialization of Ngo et al.'s GenericJoin[9] algorithm for relational joins. Our experiments consider the enumeration of 3- and 4-cliques in a directed graph (N, E), viewed as a repeated self-join of the relation of edges

$$A_{i,j} = \{(a_i, a_j) | a_i, a_j \in N, (a_i, a_j) \in E\}$$

The flow control issues and techniques apply to the more general GenericJoin implementation.

Our specific implementation of GenericJoin produces the streams of tuples in the result join when projected onto each length-k prefix of the attributes. This is initially the stream of the empty tuple, and for each k the stream is extended to the stream of length-$(k+1)$ prefixes by having each relation (i) propose the number of extensions to each tuple, (ii) extend each tuple using the relation with the fewest proposal,

BeyondMR'16, June 26-July 01 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-4311-4/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2926534.2926544

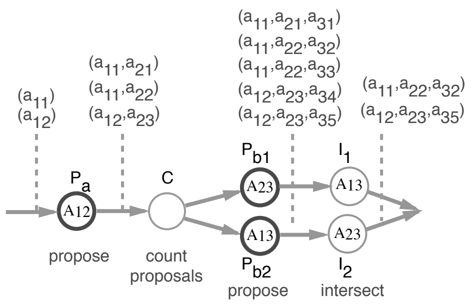

Figure 1: Logical topology for the dataflow-join computation.

and (iii) intersect the proposals against each relation to drop extensions not supported by each relation.

For example, a tuple (a_1, a_2) could indicate a 2-clique (an edge) and would be extended to a 3-clique by having each relation (of A_{12}, A_{23}, A_{13}) propose values for a_3. Only relations A_{23} and A_{13} have opinions, and for each (a_1, a_2) the proposals for a_3 come from the neighbors of the vertex (a_1 or a_2) with lower degree. They are then intersected against the neighbors of the other vertex.

Figure 1 depicts a simplified version of the logical dataflow topology. Each edge in the logical graph corresponds to a full interconnection between physical workers, each of which perform some fraction of the work for each operator. The edges are annotated with the tuples that cross them, at any point in the computation, as a result of injecting the two singleton tuples (a_{11}) and (a_{12}).

An uncontrolled, naïve execution schedule may attempt running operators P_a, C, P_{b1} and P_{b2} to completion before the intersect steps starts reading tuples. This would result in the entirety of the intermediate state produced by the two propose steps to be buffered on the edges from operators P_{bi} to I_i. Needless to say, this amount of data can be enormous.

Conversely, a hypothetical memory-optimal single-threaded schedule for the computation would limit all the input operator buffer sizes to a single tuple: for example, P_1 would transfer control to C and P_{2i} immediately after producing tuple (a_{11}, a_{21}) and P_{2i} would, in turn, yield to C_i with (a_{11}, a_{22}, a_{31}) and so on. Such an approach, while minimizing memory usage, would, in turn, starve most workers in a distributed setting.

3. PROGRESS TRACKING BASICS

This section introduces the core concepts of Timely Dataflow's progress tracking subsystem, which is a core primitive that enables the implementation of Faucet as a user-level pattern that does not require changes to the underlying engine.

In Timely Dataflow tuples are shipped within small batches that carry additional metadata: a logical timestamp is always associated to a batch of tuples and this information is used to keep track of the computation's overall progress. When traversing an operator, input tuples can turn into output tuples which have an equal or greater timestamp.

Operators of the computation are organized in possibly nested scopes which enclose logical computation subgraphs, and the timestamps in each scope have a corresponding nested structure: often timestamps are tuples of integers, indicating an integer timestamp in each of its nested scopes.

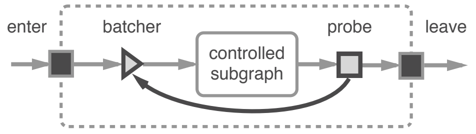

Figure 2: Diagram of the Faucet's topology pattern.

Scopes can have entry and exit nodes that respectively append or remove timestamp coordinates: these vertices adjust the metadata of tuples traversing them without affecting the contents. For example, the timestamp (a_1, a_2) for a tuple entering a subscope would become (a_1, a_2, a_3); conversely, the last coordinate would be stripped off when exiting the subscope.

Timely dataflow's progress tracking machinery aggregates information about pending timestamps, corresponding to unconsumed messages and work in the system. This allows operators to understand when they have received all records that are annotated with a given timestamp. This information is aggregated into a per-input frontier at each operator. A frontier $F = \{t_1, ..., t_x, t_i \in T\}$ implicitly defines the set S_F of logical timestamps that could be attached to future messages that could reach the operator's inputs,

$$S_F = \{\ t_x \mid \exists\ t_f \in F \mid t_x \geq t_f\ \}\ .$$

Conversely, any timestamp $t_i \notin S_F$ is guaranteed to never appear at the operator's input in the future.

Typically, this information is used to provide a high level signal, a *notification* at some logical timestamp t_i, that an operator may assume that it is never going to see any other message at any timestamp $t_x \leq t_i$. Timely provides an operator, *probe* that acts as a pass-through for tuples, but also permits user-level code to inspect progress tracking information: in particular, it permits to query whether a certain timestamp can still appear at the inputs of the probe operator, i.e., whether $t_i \in S_F$. This information does not flow along a dataflow edge, and an operator can inspect the progress state of other operators, even if distant in the dataflow graph.

4. USER-LEVEL FLOW CONTROL

Faucet, the computation pattern we propose, wraps a subgraph of the computation by introducing an additional scope, an operator with control logic (the *batcher*) at the subgraph ingress edge, and a probe at the egress edge. Figure 2 shows the resulting topology. The probe handle, that permits inspecting the frontier at the probe, is injected in the batcher, which uses that information to regulate the rate of admission of tuples into the controlled subgraph. When entering the new scope, data has to cross an *enter* ingress operator that appends a coordinate to the logical timestamp. Thus, all data with outer timestamp $t_{outer} = (t_1, ..., t_n)$ appears at the batcher's input with timestamp $t = (t_1, ..., t_n, t_b)$. The new coordinate, t_b will be used to annotate batches of data in the inner topology in order to track their progress through the controlled subgraph.

The batcher processes its input and forwards small batches of the data for each incoming logical timestamp t_e and annotates them with a batch counter t_b, unique within the same t_e. By inspecting the probe at the egress of the flow-control subgraph, the batcher is able to maintain a constant number

$N_{batches}$ of batches in-flight within the controlled subgraph. If the input data rate is higher than the rate at which the subgraph is able to process data, the batcher stops forwarding data and buffers it. The number of batches $N_{batches}$ and the number of tuples per batch B allowed in the controlled subgraph are configurable: how to select values for these parameters will be discussed in section 5.1.

The batcher can also support a custom transformation of the data traversing it as long as the operator logic can be expressed as a function from an incoming tuple to an iterator. With a lazy iterator, execution can be suspended after a set number of tuples has been produced and only resumed once the batch has reached the end of the controlled subgraph.

Notably, the implementation of Faucet does not require changes to the underlying system and only relies on existing abstractions available to the user: namely expressive operator logic, nested scopes and timestamps, and progress tracking information. Due to its composable structure, this flow-control mechanism can be nested arbitrarily to permit fine-grained control of the amount of data allowed to be in flight in any controlled subgraph, as demonstrated in section 4.1. This proves to be a powerful primitive for the dataflow implementor, with adaptable guarantees and overhead.

4.1 Example Usage

Figure 3 depicts the topology described in section 2 with Faucet applied to guard the two nested subgraphs originating from the two *propose* steps. We remark how, thanks to its modularity, the pattern can be applied in a nested fashion, where necessary, to control resource consumption and scheduling in subgraphs that may generate large amounts of intermediate state.

Each of the black rectangles represents a batch of tuples tagged with a certain synthetic logical timestamp (in square brackets) and for which completion can be detected by the probe at the end of the flow-control block.

An example for $N_{batches} = 1$ and $B = 1$ follows. The input tuple (a_{11}) is the only one allowed in the outer scope at first, as part of a batch with timestamp $(..., u_1)$. (a_{12}) will only be injected once processing for (a_{11}) is complete. The inner instance of Faucet only admits (a_{11}, a_{21}) in the inner scope as part of the batch annotated with timestamp $(..., u_1, v_1)$. Once this batch is completed (resulting in no tuples on the output), (a_{11}, a_{22}) is admitted into the inner scope with timestamp $(..., u_1, v_2)$. The process repeats till exhaustion of the input.

By limiting the batch size and the number of parallel batches, Faucet limits the amount of intermediate state being generated and stabilizes the buffer sizes.

5. EVALUATION

The following measurements refer to the dataflow-join computation described in the example, when executed on the *livejournal* dataset and with a varying number of worker threads allocated evenly on two machines with an Intel Xeon E5-2650 @ 2.00GHz with 16 physical cores and connected by a 10Gbps link.

5.1 Sensitivity to Parameter Choice

When devising an execution schedule, the trade-off between minimizing the peak message queue size between operators and the likelihood that no single worker is subject to starvation can have a critical effect on overall throughput and latency.

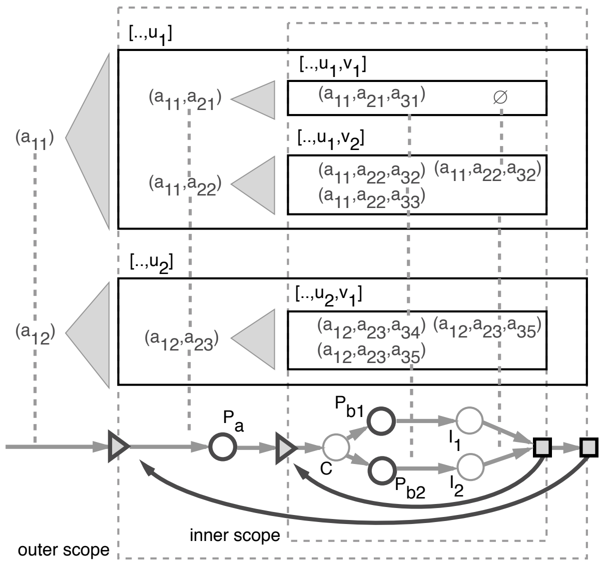

Figure 3: Dataflow-join topology with the Faucet pattern applied.

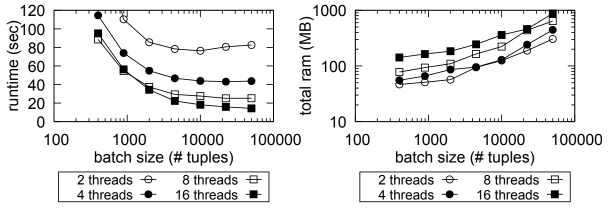

Figure 4: Effect of batch size B on total runtime and memory consumption, with $N_{batches} = 4$

In Faucet, large enough values for $N_{batches}$ (the number of parallel batches) and B (the batch size) result in close-to-optimal performance, rendering the selection of optimal values for the pattern's parameters less of a concern for the end user. In our experiments, switching from $N_{batches} = 1$ to $N_{batches} = 2$ improves performance multiple-fold by minimizing the adverse effect of stragglers which delay injection of future batches by inhibiting the completion of a previous batch. Notably, values of $N_{batches} \geq 2$ have very limited effect on runtime, especially when running with a larger number of threads.

Similarly, when a $N_{batches} \geq 2$ is selected, an exceedingly small value for B causes dramatic performance losses likely due to inefficient processing and synchronization overhead. The left chart in Figure 4 shows that, for $N_{batches} = 4$ and values of B above a certain threshold the total runtime is not affected by B. For the topology under test, the threshold value is roughly $1000 \cdot C$ where C is the number of worker threads. On the right of Figure 4 we show the effect of different values for B on peak total memory consumption. Graphs for other values of $N_{batches} \geq 2$ display a similar pattern.

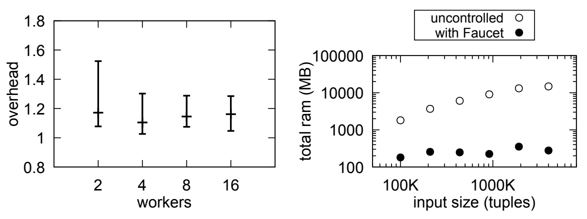

Figure 5: Overhead and peak memory footprint of Faucet when compared to a non-controlled execution.

5.2 Overhead

In order to estimate the overhead caused by the additional synchronization and inefficiencies introduced by the Faucet pattern, we compare the total execution time of the computation with and without the pattern applied. We artificially limited the size of the input dataset to prevent the failure of the non-controlled topology due to resource exhaustion.

The left chart in figure 5 shows overhead as the ratio of total execution time with and without the pattern applied: we display minimum, maximum and average overhead when we select a range of values for the parameters that achieve close to optimal performance. In distributed settings the measured overhead averages around 15% and rarely peaks above 20%.

5.3 Buffer Size

The Faucet pattern is effective in limiting the sizes of operator input buffers within the controlled subgraphs without imposing hard limits. This ensures that computations producing large amounts of intermediate state can run within a limited memory footprint.

We rely again on an artificially reduced input data set to provide a useful comparison. The right chart in Figure 5 displays how, when the input data set size is increased, (i) peak memory usage grows in the uncontrolled executions, and (ii) it remains constrained and one to two orders of magnitude lower with Faucet.

In additional experiments, Faucet was successful in limiting peak memory usage in computations that produce orders of magnitude more intermediate state, *e.g.* dataflow-join topologies with additional propose and intersect steps to extract 4-cliques.

6. RELATED WORK

The problem of flow control is inherent to dataflow systems. Flink[1], IBM Streams[3], and Reactive Streams[10] employ a system-wide, always-on edge-by-edge backpressure technique similar to TCP flow control. The strategies employed in Storm[5], Heron[4], and Spark[2] can only affect the ingestion rate. In contrast, Faucet's approach is modular and arbitrarily nestable.

The pattern described in this document relies on primitives provided by Timely Dataflow's progress tracking mechanism. Other systems may be able to provide similar signals for user-level flow-control. For example: customizable, fine-grained punctuation[11], exposed by the system or generated at the operator level, can be sufficient to keep track of the completion of tuple batches among operators, even if distant in the dataflow graph.

7. CONCLUSION AND FUTURE WORK

We introduced Faucet, a flow control pattern for distributed data-parallel dataflow systems that produces efficient, stable computation schedules with limited synchronization overhead.

Thanks to its modularity and low sensitivity to parameter choice, the technique can be effectively applied to stabilise computations, where necessary, with limited burden on the user. Controlled memory usage may also enable performance gains through buffer re-use in a broader set of topologies.

We remark how Faucet is implemented by relying solely on the progress-tracking information Timely Dataflow provides to user logic; we expect that a similar approach would enable prototyping a diverse family of scheduling and optimization patterns for modern dataflow systems.

8. REFERENCES

[1] U. Celebi, K. Tzoumas, and S. Ewen. How flink handles backpressure. http://data-artisans.com/how-flink-handles-backpressure/. Online; accessed 23 February 2016.

[2] F. Garillot et al. Spark streaming back-pressure signaling. https://docs.google.com/document/d/1ZhiP_yBHcbjifz8nJEyPJpHqxB1FT6s8-Zk7sAfayQw/edit. Online; accessed 23 February 2016.

[3] M. Hirzel et al. Ibm streams processing language: Analyzing big data in motion. *IBM J. Res. Dev.*, 57(3-4):1:7–1:7, May 2013.

[4] S. Kulkarni et al. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[5] Z. Liu et al. [storm-886] automatic back pressure (abp). https://github.com/apache/storm/pull/700. Online; accessed 23 February 2016.

[6] F. McSherry. Generic join. http://www.frankmcsherry.org/dataflow/relational/join/2015/04/11/genericjoin.html. Online; accessed 24 February 2016.

[7] F. McSherry et al. Timely dataflow. https://github.com/frankmcsherry/timely-dataflow. Online; accessed 1 March 2016.

[8] D. G. Murray et al. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, November 2013.

[9] H. Q. Ngo, C. Re, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. *CoRR*, abs/1310.3314, 2013.

[10] Reactive Streams Special Interest Group. Reactive streams. http://www.reactive-streams.org/. Online; accessed 23 February 2016.

[11] 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, Mar. 2003.