Latency-conscious dataflow reconfiguration

Moritz Hoffmann
ETH Zurich
moritz.hoffmann@inf.ethz.ch

Frank McSherry
ETH Zurich
frank.mcsherry@inf.ethz.ch

Andrea Lattuada
ETH Zurich
andrea.lattuada@inf.ethz.ch

ABSTRACT

We propose a prototype incremental data migration mechanism for stateful distributed data-parallel dataflow engines with latency objectives. When compared to existing scaling mechanisms, our prototype has the following differentiating characteristics: (i) the mechanism provides tunable granularity for avoiding latency spikes, (ii) reconfigurations can be prepared ahead of time to avoid runtime coordination, and (iii) the implementation only relies on existing dataflow APIs and need not require system modifications.

We demonstrate our proposal on example computations with varying amounts of state that needs to be migrated, which is a non-trivial task for systems like Dhalion and Flink. Our implementation, prototyped on Timely Dataflow, provides a scalable stateful operator template compatible with existing APIs that carefully reorganizes data to minimize migration overhead. Compared to naïve approaches we reduce service latencies by orders of magnitude.

ACM Reference Format:
Moritz Hoffmann, Frank McSherry, and Andrea Lattuada. 2018. Latency-conscious dataflow reconfiguration. In *BeyondMR'18: Algorithms and Systems for MapReduce and Beyond , June 15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3206333.3206334

1 BACKGROUND

Distributed streaming applications are by nature long-running and may face varying and largely unpredictable workloads. A robust system is required to *adapt* to changes in order to provide low-latency, high-throughput, reliable and cost-efficient streaming services.

In this paper we consider adapting *stateful* streaming distributed dataflow computations. A stateful dataflow computation has operators that rely on access to non-transient state, distributed across computing elements, which may need to be migrated when adapting to changes in the workload. Specifically, we design, implement, and evaluate a mechanism for state migration among dataflow workers that is efficient, responsive, and minimally invasive.

The current state-of-the-art systems have three approaches to provide scaling mechanisms. Firstly, systems such as Flink [2] and Dhalion [4] rely on their fault-tolerance mechanism to capture the state of the system and then restart their computations with a different configuration. Secondly, systems such as SEEP [3] stop and restart parts of a computation to limit the overall performance impact of scaling a dataflow. Thirdly, systems such as MillWheel [1] externalize state to turn stateful operators into stateless variants at the cost of reduced performance.

We propose a state migration mechanism that substantially reduces latency spikes during reconfiguration, by using the dataflow system itself to perform progressive state migration. Our implementation is built on Timely dataflow[1], and is implemented purely in library code, requiring no modifications to the underlying system.

We should stress that while our prototype does migrate data, the bulk of the effort in reconfiguration, it does not perform *system* reconfigurations like adding or removing workers. This is a non-trivial operation the existing systems do perform, and one which shouldn't be ignored. Our approach is aimed at a setting where a large set of dataflow workers are available but otherwise idle, and we bring them into the computation by migrating data to them.[2]

2 MECHANISM

We illustrate our mechanism in the context of a data-parallel `fold` operator: given an ordered stream of (key, val) pairs, we want to independently apply a function $f : (state, val) \rightarrow (state, result)$ to state associated with each key, and produce the $(key, result)$ stream as output. The `fold` operator is commonly seen in the form of data-parallel aggregation, but it is general enough to implement more advanced state machine logic.

A typical implementation of `fold` distributes responsibility for keys across a set of workers, and processes (key, val) pairs by routing them to the appropriate worker based on key. Each record is somehow timestamped, and the timestamp order describes the order in which (key, val) pairs from multiple sources should be applied. As long as the distribution of keys is consistent, a single worker receives all values for any one key and can correctly maintain the associated state. However, any *change* to the distribution of keys has the potential to result in incorrect results: initial state may be missing and (key, val) pairs may be transiently mis-routed.

In modern dataflow systems the input timestamps do not need to reflect the current system time, but can instead be *logical*, chosen by the same external source that provides the data. Most dataflow systems also do not require timestamps to be presented in order, and instead require only that the source of data commit to advancing the lower bound of input timestamps in order to make corresponding progress in the outputs. At each location in the dataflow graph, the dataflow system is able to conservatively report the timestamps that might still be seen on input messages.

The main idea behind our mechanism is to use the timestamps of the records themselves, and the progress information provided by the system, to coordinate clean transitions from one distribution of keys to another. We introduce a new stream of records describing

ACM ISBN 978-1-4503-5703-6/18/06…$15.00
https://doi.org/10.1145/3206333.3206334

[1]`https://github.com/frankmcsherry/timely-dataflow`
[2]Timely dataflow workers which process no data do not participate in the underlying coordination protocol, and can in principle remain idle until they do receive data.

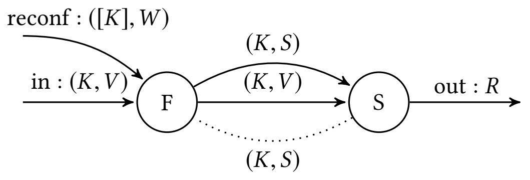


Figure 1: We replace a single distributed `fold` operator with distributed `fixer` (F) and `state` (S) operators, where `fixer` has (i) an additional reconfiguration input, (ii) a $(key, state)$ dataflow edge to `state`, and (iii) access to the state of keys managed by the instance of `state` on the same worker.

reconfiguration requests, each timestamped from the same domain as the inputs to the `fold` operator. We then use the dataflow mechanisms themselves to both coordinate and effect the migration: (i) the timestamps on (key, val) data indicate when we should switch the routing of values to new workers, (ii) the system-reported progress reveals when `fold` state reflects all values with pre-reconfiguration timestamps (and is therefore safe to migrate), and (iii) the dataflow channels themselves transport migrating state with timestamps corresponding to the migration time.

This approach results in clean transitions, but it has the potential to be disruptive in the moments of transition. Fortunately, we show that the per-reconfiguration overheads are small enough that we can perform a sequence of fine-grained migrations. Rather than migrate all keys at once, stalling the computation for the duration, we refine the space of keys to a granularity of our choosing and trade off the latency of each reconfiguration against the total number of reconfigurations.

2.1 Detailed description

A Timely dataflow operator like `fold` is specified from a stream of input data by announcing a partitioning function for the stream, and supplying the operator logic that should be invoked within each worker. The Timely dataflow system instantiates multiple copies of the operator, and supplies each instance with a steady stream of data partitioned by key. Further, the Timely dataflow system provides information about which timestamps have expired and will not be seen in the input again. In the case of `fold` the partitioning function should be a function of the *key* field, and the operator logic likely contains a hash map from keys to state, and the logic to apply received values to the state of the associated key once their timestamp has expired.

Our implementation replaces a single `fold` operator with two operators: `fixer` and `state`, sketched in Figure 1. These two operators are fully connected, in that each instance of `fixer` can send data to each instance of `state`.

In addition to the stream of (key, val) pairs the operator takes a stream of timestamped reconfiguration statements, of the form $(keys, worker)$ indicating that a set of keys should now be mapped to a specific worker (as of the timestamp). This stream represents control signals that would normally be system-level interventions as dataflow streams themselves. The `fixer` operator is responsible for distributing the stream of (key, val) pairs to the `state` operator in accordance with the current mapping from keys to workers. The `state` operator behaves as if it were `fold`, maintaining state for each key according to the values it receives.

In addition, the `fixer` operator has a second output along which it may send $(key, state)$ messages, to effect the migration of state from its worker to another worker. The `fixer` operator has a handle to the state of the `state` operator on the same worker, and can inspect it to see the state that it may need to send to another worker. The `state` operator has a second input on which it receives state from the `fixer` operators and which it installs and then maintains (until a future reconfiguration directs the state to another worker).

2.2 Migration logic

While the `fixer` and `state` operators now have sufficient dataflow connections to effect the migration of state, we must carefully detail the operator logic to ensure that this migration happens only once all state updates are included in any migrated state.

Timely dataflow moves timestamped data among workers on dataflow channels, but is also able to report the minimal outstanding timestamps at any point in the dataflow graph. This information tells operators about "progress" in the computation, and provides guarantees that all updates with certain timestamps have been retired from the computation. We will have the `fixer` operator monitor these timestamps in the output of the `state` operator to be certain that it only migrates state that reflects all updates through the timestamps indicated by the reconfiguration requests.

Specifically, the `fixer` operator receives as input both timestamped (key, val) pair and $(keys, worker)$ pairs, but acts on neither until Timely dataflow indicates that the timestamp is no longer possible from the reconfiguration channel. This progress informs us that `fixer` now knows the final destination for a timestamped (key, val) pair, and will not prematurely migrate state that may be pre-empted by an out-of-order reconfiguration request. Any (key, val) data may now be routed according to the configuration as of their timestamp, and any reconfiguration request is marked as pending and awaits finalization before being effected.

The `fixer` operator also monitors the timestamps that might result from the output of the `state` operator, which warns the operator about the potential existence of (key, val) and $(key, state)$ records in flight elsewhere in the system. Once the potential output timestamps of `state` match that of a pending reconfiguration, the `fixer` operator can effect the migration with the certainty that the state cannot experience a state transition at an earlier timestamp. The `fixer` operator captures the state of the keys indicated by the reconfiguration and sends them to the indicated worker with the timestamp of the reconfiguration request.

If the `fixer` operator receives several reconfiguration requests for future times, it can route any (key, val) tuples at any timestamps t for which it knows it will receive no further reconfigurations (as indicated by Timely dataflow's view of the reconfiguration input). Even in the absence of migrations, the reconfiguration input can indicate that there will be no further reconfigurations for the foreseeable future, removing the `fixer` queueing for the common case of (key, val) records timestamped within the foresworn time. This trades off latency (buffering records) against adaptivity (foreswearing reconfigurations for a time).

2.3 An Example

Before evaluating our proposed mechanism, let's walk through a simplified example. Imagine a wordcount example where keys are strings, and both values and state are integers that simply accumulate the total associated with each string.

Imagine we have two workers, and initially all data are at worker 0. Perhaps we have as input data (“*dog*”, 10) and (“*cat*”, 5) at timestamp 100, and (“*dog*”, 13) and (“*cat*”, 23) at timestamp 200. At timestamp 150 we have a reconfiguration request (“*dog*”, 1) and at timestamp 200 we have a similar reconfiguration request of (“*cat*”, 1).

Initially the `fixer` operator may receive the (“*dog*”, 10) and (“*cat*”, 5) records, which it enqueues until its reconfiguration input passes timestamp 100. There are no reconfigurations at or before timestamp 100; this progress is eventually noticed, and the records are distributed according to the current configuration. The `state` operator observes that it will not receive inputs with timestamp 100 (neither other values, nor migrated state) and it applies the updates.

Now `fixer` receives the reconfiguration request (“*dog*”, 1) bearing timestamp 150. At this point `fixer` has a potential migration to perform, but it must await progress in its inputs and the output of `state`. The inputs to `fixer` should advance past timestamp 150 at which point the `fixer` operator can observe that `state` operator's output reaches timestamp 150 (it does not *pass* timestamp 150, because the `fixer` retains the ability to send it data with timestamp 150). The `fixer` operator can now migrate the state for “dog", by peeking at the maintained state and producing an output (“*dog*”, *state*) that it sent to worker 1. Worker 1 consumes this input, and soon thereafter its `state` operator learns that its inputs have now passed timestamp 150, and so it installs the state. The timestamps in the output of `state` now increase beyond 150, as neither messages nor operators may send at that time.

Perhaps `fixer` now receives the reconfiguration request (“*cat*”, 1) bearing the timestamp 200, but not yet the data (“*dog*”, 13) and (“*cat*”, 23) (also bearing timestamp 200). The reconfiguration must await the output of `state` reaching timestamp 200, but it does not need the timestamp to pass 200. The reconfiguration can be initiated even before the recently received data are routed; a reconfiguration only co-locates values and state at the indicated time, and only needs to ensure that the state reflects *prior* values.

Once the data input to `fixer` reaches 200 and the reconfiguration input passes 200, the migration can be initiated. At this point, the data timestamped with 200 can be immediately routed and both keys should arrive at worker 1, where they will be correctly applied to the accumulation of strictly prior values.

2.4 Discussion

Many stateful dataflow operators can be independently migrated, especially when downstream operators make no assumptions about from which workers their input records will arrive. There are however situations in which multiple operators may need to be simultaneously migrated. Systems like Flink and Naiad [5], among others, reveal that the outputs of operators like `fold` are now partitioned by their key, and downstream operators may exploit this information (avoiding redundant data-exchange). If we migrate only the upstream operator, this information becomes invalid and downstream operators may malfunction as a result.

One resolution to this problem is simply to disable optimizations that rely on specific data placement. A second alternative is to simultaneously migrate the state of the other operators whose key distribution needs to match the operator. Because our migration mechanism performs exactly the migration as indicated by the reconfiguration stream, rather than a best-effort approximation, the same stream of reconfiguration requests can be applied to other stateful operators that must be co-partitioned.

3 EVALUATION

Our goal is to evaluate the potential of our migration mechanism, as compared with less carefully coordinated migration strategies. To this end, we have implemented a prototype on Timely dataflow, and a test harness with synthetic data that allows us to demonstrate the mechanisms behavior in several situations. In order to evaluate our proposed mechanism, rather than the performance of specific systems, we create computations emulating the migration strategies of some existing approaches.[3]

We consider executions in which the reconfiguration is either *sudden*, in which all keys are redistributed at the same time, or *fluid*, in which small sets of keys are migrated completely before initiating the migration of the next batch of keys. The sudden approach represents a system that is unavailable for as long as it takes to migrate all state between workers; unlike Dhalion or Flink we don't shut down the system, but the migration does stall all requests. The fluid approach represents our contribution, in which the timestamps guide a progressive migration that regularly returns to a consistent configuration that can service requests. We also consider a *batched fluid* approach that migrates sets of keys so that no worker is simultaneously involved in two migrations.

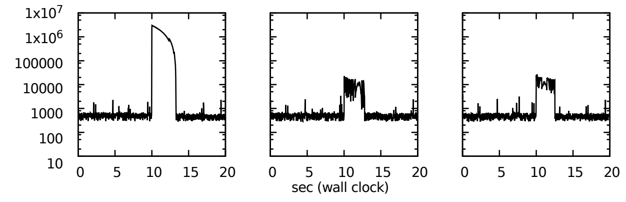
All of our experiments use a relatively simple word-counting `fold` dataflow, in which the keys are strings and the values are integers. We exploit none of the algebraic properties of counting (*e.g.* pre-aggregation), and use counting only as a computation-light state machine, to better study the effects of migration. In all our experiments we bin keys into 256 groups and use this as the finest granularity of migration, but this choice can be made differently (or reconfigured itself, in principle).

Our experiments are performed on a 32 core Intel Xeon E5-4640 running at 2.40 GHz running two processes and using loopback TCP to connect the two. In each experiment roughly half of the migrating data changes workers, and all such data crosses between processes. Migration across a more constrained physical network should only make the migration more painful.

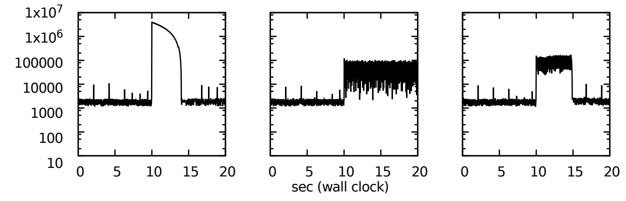
3.1 Sudden vs. fluid migration

For our first experiment, we demonstrate the difference between sudden and fluid migrations. We start with a set of 10 million keys, which are initially on one worker, and at 10 seconds migrate the keys to be uniformly distributed between two workers. We introduce values at a rate of 1 million per second, and record the latency for the response to each as the time between the response returning

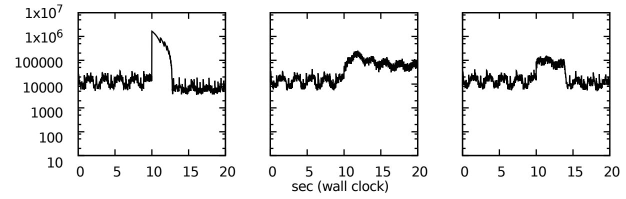
[3]The absolute performance of Timely dataflow can be much higher than Storm, Dhalion, and Flink, and this decision is made to have a stronger baseline that avoids conflating systems benefits with any benefits of our approach to migration.



(a) Ten millions keys migrating from one worker to two workers under a constant one million queries per second.



(b) Forty millions keys migrating from four workers to eight workers under a constant one million queries per second.



(c) Ten millions keys migrating from four workers to eight workers under a variable 2.6 million queries per second.

Figure 2: Timelines for sudden, fluid, and batched fluid migration, each at 10 seconds, with elapsed seconds on the x-axis and microseconds latency on the y-axis.

and the moment at which the input should have been available (even if the system was not available to accept it yet).

Figure 2a reports the observed latencies when we migrate 10 million keys from one worker to two workers, using sudden migration and twice using fluid migration. The observed latency is substantially reduced, from 4 seconds for the sudden migration to roughly 20ms for the fluid migration. Additionally, time to return to a stable latency is also reduced, from 4 seconds to 2.5 seconds, in part because long queues do not build up.

3.2 Batched migration

For our second experiment, we increase the number of workers from one and two to four and eight, and compare sudden and fluid migration. With multiple workers, sudden migration may take less time, but fluid migration as described above will still perform a sequence of 256 migrations each of which involve only two workers, and each of which may take no less time with additional workers.

To test this, we also consider a migration strategy that concurrently migrates sets of bins so that no worker is simultaneously involved in two migrations, again retiring input (key, val) data between the migrations. The intent is that these concurrent migrations may come at little incremental latency, as the system is briefly stalled for any migration, but will reduce the elapsed migration time (for which the system experiences degraded latency) by keeping all workers equally busy.

Figure 2b demonstrates these three migrations—sudden, fluid, and batched fluid—among from four to eight workers with a proportionately increased state size of 40 million keys. We see that the sudden migration takes no more time to migrate (despite the increased number of keys), whereas the fluid migration takes longer to perform the complete migration as each individual bin is migrated no more quickly. The batched fluid migration takes as long as the sudden migration, with the latency of the fluid migration.

3.3 Loaded migration

For our last experiment, we consider the ability of the migration mechanisms to migrate to more workers when under barely sustainable input loads. Specifically, instead of a fixed load we generate load using a square wave where the high load exceeds and the low load is within the capacity of four workers and both are within the capacity of eight workers. Our migration mechanisms respond differently to load, and we might expect that the more responsive progressive migration more quickly reaches a sustainable configuration (once enough bins migrate to the additional workers).

Figure 2c demonstrates sudden, fluid, and batched fluid migration under barely sustainable load, using a square-wave load generator with a period of two seconds. Although batched fluid migration does maintain lower latencies, the faster stabilization does not occur, suggesting that there may be more intelligent policy decisions about how aggressively to migrate data versus service existing load.

4 CONCLUSION

We introduce a progressive migration mechanism that can be used by dataflow systems to reconfigure the layout of state without substantial interruptions to the availability of the system itself.

Importantly, we were able to do this without invasive changes to the dataflow system. Our migration mechanism demonstrates that one can use dataflow coordination mechanisms (timestamp progress information) and dataflow channels themselves to effect the migration. The only feature specific to Timely dataflow, not commonly found in other systems, is the ability to observe state in other local dataflow operators. This feature can be emulated with a dataflow back-edge, creating a cycle in the dataflow, another feature that is currently specific to Timely dataflow. Other systems should be able to support a similar progressive migration mechanism, but may require some extra-dataflow coordination.

REFERENCES

[1] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam Mcveety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.

[2] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and batch processing in a single engine. *Data Engineering* 38, 4 (2015).

[3] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. 725–736. https://doi.org/10.1145/2463676.2465282

[4] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: Self-Regulating Stream Processing in Heron. *PVLDB* (2017).

[5] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM. http://research.microsoft.com/apps/pubs/default.aspx?id=201100