FASTER State Management for Timely Dataflow

Matthew Brookes
ETH Zürich
brookesm@student.ethz.ch

Vasiliki Kalavri
ETH Zürich
vasiliki.kalavri@inf.ethz.ch

John Liagouris
ETH Zürich
liagos@inf.ethz.ch

ABSTRACT

We explore the performance and resource trade-offs of two alternative approaches to streaming state management. When the state size exceeds the amount of available memory, systems can either scale out and partition the state across distributed computing nodes or rely on secondary storage and divide the state into 'hot' and 'cold' sets. Scaling out a streaming computation might introduce coordination overhead among parallel workers, while flushing state to disk requires efficient data structures and careful caching policies to minimise expensive I/O.

To study the characteristics of these state management approaches, we present an integration of the Timely Dataflow stream processing engine with the FASTER embedded key-value store. We demonstrate a prototype that allows users to transparently maintain arbitrary larger-than-memory state with low overhead by making only minimal changes to application code. Our preliminary experimental results show that managed state incurs acceptable overhead over built-in in-memory data structures and, in some cases, performs better when relying on secondary storage in a single node as opposed to scaling out to multiple nodes.

ACM Reference Format:
Matthew Brookes, Vasiliki Kalavri, and John Liagouris. 2019. FASTER State Management for Timely Dataflow. In *Real-Time Business Intelligence and Analytics (BIRTE 2019), August 26, 2019, Los Angeles, CA, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3350489.3350493

1 INTRODUCTION

Any non-trivial streaming computation maintains and continuously updates state: running aggregations, synopses of the input stream, sets of events grouped in windows, triggers and timers. Efficient state management in stream processing is crucial for two reasons. First, due to low latency requirements, streaming engines must process incoming events and update internal state as quickly as possible. Second, due to their long-running nature, streaming applications accumulate state which can exceed the available memory provided by a single node. To support both low latency and growing state, modern streaming dataflow systems employ data parallelism and state partitioning so that parallel workers are responsible for disjoint input streams and state partitions.

Deploying streaming jobs across multiple distributed machines allows for larger state but might incur performance penalties due to higher coordination among parallel workers. This trade-off between supported total state size and coordination overhead is evident in distributed stream processors.

This demo presents an integration of Timely Dataflow [3] with FASTER [2], an embedded key-value store that combines a highly optimised cache with a hybrid log spanning main memory and storage. Offloading the task of state management to FASTER enables Timely Dataflow to support large state while keeping the number of parallel workers low and still use commodity hardware.

2 BACKGROUND

Timely Dataflow is a stream processor, written in Rust, based upon the Naiad[5] system. Timely computations are expressed as directed, possibly cyclic, graphs of operators that process timestamped data. Timely Dataflow programs can be executed by multiple workers, either on a single machine or in a networked cluster. Each worker executes a copy of the dataflow program on partitioned input data and exchanges intermediate results and progress information with other workers as needed. Timely operators may construct, access, and modify built-in and custom data structures. For example, the `Aggregation` operator in Listing 2 modifies an aggregate value every time it is invoked. Each Timely worker maintains its own separate state during processing and cannot directly access the state of other workers.

FASTER is an embedded Key-Value store that supports in-place updates and concurrent access to a log-structured record store for larger-than-memory data via a `HybridLog`. It provides point reads, blind updates, and read-modify-write operations for arbitrary Key-Value types. FASTER exploits the strong temporal locality exhibited by state in streaming workloads. Records are separated into a changing *hot*

BIRTE 2019, August 26, 2019, Los Angeles, CA, USA
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7660-0/19/08.
https://doi.org/10.1145/3350489.3350493

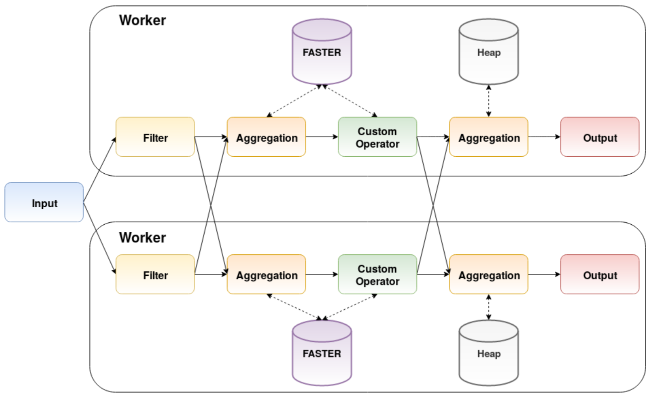


Figure 1: Integration Overview

state kept in-memory and a larger on-disk *cold* state. The HybridLog record allocator organises records across both portions such that newer records can be updated in-place and older records use a copy-on-write strategy. By providing in-place updates, workloads can benefit from records being in higher levels of the cache as well as from not copying entire records when making modifications. FASTER is implemented as a C++/C# library that allows users to define their own Key and Value types.

3 INTEGRATING TIMELY DATAFLOW WITH FASTER

Figure 1 showcases the integration of Timely Dataflow and FASTER. In the following, we describe the state primitives currently supported, how state is exposed to users, and how users can define different state backends.

We have implemented a library, *faster-rs* [1], that allows any Rust application, including Timely Dataflow programs, to use FASTER as an embedded Key-Value store. *faster-rs* is a wrapper around FASTER's C++ library [4].

3.1 Managed State Primitives

We have created several basic state primitives that can be used by developers of Timely programs. These include a `ManagedCount`, `ManagedValue`, and `ManagedMap`. Listing 1 shows the methods available for each primitive. The state primitives are backend-agnostic, meaning that dataflow operators can be configured to use any custom state backend with any of the aforementioned primitives. State primitives are accessed via a `StateHandle`, as we explain in the following.

3.2 Accessing State

Timely Dataflow provides several generic operators that can be used to implement arbitrary logic. These generic operators provide several handles for accessing their input(s), output(s), notificator (used for coordination with other workers), and

```
pub trait ManagedCount {
    fn decrease(amount: i64);
    fn increase(amount: i64);
    fn get() -> i64;
    fn set(value: i64);
}

pub trait ManagedValue<V> {
    fn set(value: V);
    fn get() -> Option<Rc<V>>;
    fn take() -> Option<V>;
    fn rmw(modification: V);
}

pub trait ManagedMap<K, V>  {
    fn insert(key: K, value: V);
    fn get(key: &K) -> Option<Rc<V>>;
    fn remove(key: &K) -> Option<V>;
    fn rmw(key: K, modification: V);
    fn contains(key: &K) -> bool;
}
```

Listing 1: Managed State Primitive Methods

internal state. By exposing the `StateHandle` through the generic operators' API, one can implement individual operators with their own managed state primitives. Listing 2 shows how a generic Timely Dataflow operator (`unary_notify()`) can access a `ManagedMap` state primitive. State primitives can also be accessed through `StateHandles` exposed through the node and worker's API. Instances of state primitives are given a name ("aggs" in the example below) when created or accessed. This serves as a key for storing the values on the state backend.

```
fn aggregate(...) -> Stream {
  self.unary_notify(...,
    move |input, output, notificator, state_handle| {
      let mut aggregates // a ManagedMap (Listing 1)
          = state_handle.get_managed_map("aggs");
         ...
    }
  )
}
```

Listing 2: Aggregation Operator with Managed State

3.3 Defining State Backends

Currently, our implementation provides an in-memory and a FASTER backend, however, there is no restriction against custom backends as long as they implement the required interface for the various state primitives.

State backends are defined at the node, worker or operator level. The FASTER state backend at the node level allows workers on the same node to share state. At the worker level, all stateful operator instances executed by the particular worker will use the defined backend. Defining a state backend for a worker is compulsory but can be overridden for individual operators if needed. In Figure 1, each worker is configured to use FASTER as the default backend for stateful operators; however, the last Aggregation operator (just before the Output) is set to use Heap for storing its state. Listing 3 shows how state backends are defined at the worker and operator levels. The node level backend cannot be changed.

```
worker.dataflow::<_,_,_,FASTERBackend>(
|scope, worker_state_handle| {
  /// Worker is configured to use the FASTERBackend
  input
  .to_stream(scope)
  .unary_notify_core::<_,_,_,InMemoryBackend> (
    /// This operator overrides the configured
    /// state backend and uses the InMemoryBackend
    // Operator logic
    ...
  )
});
```

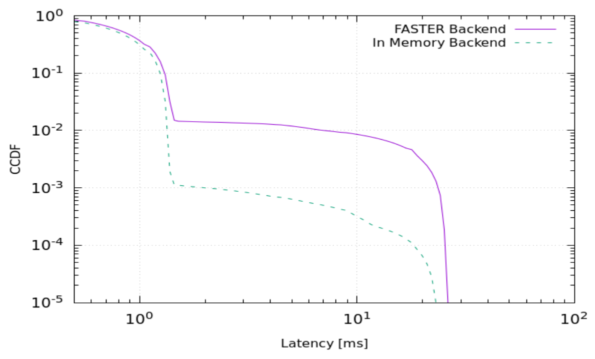
Listing 3: Configuring State Backends

3.4 Preliminary Results

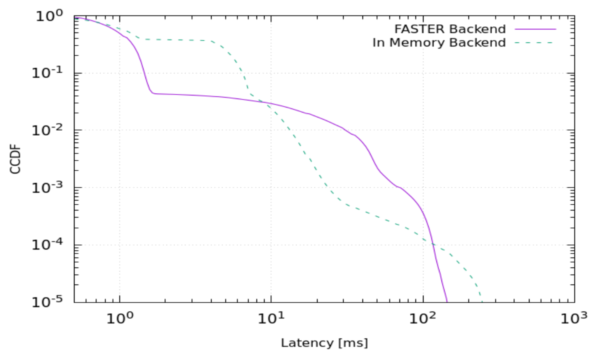
We present the results of two experiments on commodity machines, each having Intel Xeon Platinum 8000 series processors and 16GB of available memory. Figure 2a shows the complementary cumulative distribution function (CCDF) of per-record latency for Query 3 of the Nexmark benchmark [6] with either the FASTER or in-memory backend. The state fits within available memory in both configurations. Currently, using FASTER incurs a 1.1x overhead until the p98 latency and a 16x overhead afterwards. The latter overhead comes from FASTER dynamically allocating memory as the in-memory buffer fills. Despite this overhead, the maximum absolute latency when using FASTER is $26ms$. Figure 2b shows the CCDF of per-record latency for the same query and $22GB$ of total state, when using FASTER on one worker compared to scaling out using in-memory on four nodes with 24 workers in total. We show that until the p97 latency and after the p9999, a single worker with larger-than-memory state achieves better performance.

4 DEMONSTRATION OUTLINE

In this demonstration we will show the implementation of several Nexmark queries in both Timely Dataflow's original form and with managed state. We will demonstrate the



(a) FASTER overhead compared to in-memory backend



(b) Single-node FASTER compared to distributed in-memory workload

Figure 2: CCDFs of per-record latency for Nexmark Query 3 (incremental join)

modification of Timely Dataflow programs, using various state primitives and switching between state backends on a worker or operator level. We will also demonstrate our results showing when it is preferable to rely on secondary storage for storing larger-than-memory state rather than scaling out to more nodes.

REFERENCES

[1] Matthew Brookes and Max Meldrum. 2019. faster-rs. Retrieved June 01, 2019 from https://github.com/faster-rs/faster-rs
[2] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. *Sigmod 2018* (2018).
[3] Frank McSherry. 2019. Timely Dataflow. Retrieved June 01, 2019 from https://github.com/TimelyDataflow/timely-dataflow
[4] Microsoft. 2019. FASTER. Retrieved June 06, 2019 from https://github.com/microsoft/FASTER
[5] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *SOSP '13: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.*
[6] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2002. *NEXMark—A Benchmark for Queries over Data Streams DRAFT.* Technical Report. OGI School of Science & Engineering at OHSU.