Timestamp tokens: a better coordination primitive for data-processing systems

Andrea Lattuada
ETH Zurich

Frank McSherry
Materialize Inc.

Abstract
Distributed data processing systems have advanced through models that expose more and more opportunities for concurrency within a computation. The scheduling of these increasingly sophisticated models has become the bottleneck for improved throughput and reduced latency.

We present a new coordination primitive for dataflow systems, the timestamp token, which minimizes the volume of information shared between the computation and host system, without surrendering precision about concurrency. Several projects have now used timestamp tokens, and were able to explore computational idioms that could not be expressed easily, if at all, in other platforms. Importantly, these projects did not need to design and implement whole systems to support their research.

1 Introduction
Systems for data-intensive computation have advanced through programming models that allow programs to reveal progressively more opportunities for concurrency. Frameworks like MPI [2] allow programmers only to explicitly sequence data-parallel computations. Systems like DryadLINQ [24] and Spark [25] use data-dependence graphs to allow programs to express task parallelism. Stream processors like Flink [9] and Naiad [22] (following [1, 7, 10]) add a temporal dataflow dimension to represent pipeline parallelism. In each case, new runtimes extract more detailed information about the computations, allowing them greater flexibility in their execution. Figure 1 demonstrates the forms of parallelism that can be expressed in these systems.

Dataflow systems have become limited by the complexity of the boundary between system and computation. Specifically, as computations provide progressively more fine-grained and detailed information about concurrency opportunities, the scalability and sophistication of the system schedulers must increase. In our experience, system complexity has increased to the point that scheduling rather than computation becomes the bottleneck that prevents higher throughputs and lower latencies.

System designers have the opportunity to reduce the volume of coordination by reconsidering the interface between system and operator. For example, where Spark Streaming [26] must schedule distinct events to implement distinct logical times, Flink (and other stream processors) allow operators to retire batches of events corresponding to blocks of logical times, substantially improving throughput. Where Flink (and other stream processors) requires continual interaction with operators to confirm that they have no output at a logical time, Naiad asks operators to explicitly identify future times at which the operator should be notified, which is necessary to support cyclic dataflows. These interfaces reduce the volume of coordination, but require a deeper involvement of the system itself: continually invoking operators in Flink and sequencing notifications in Naiad.

We propose a simple dataflow coordination primitive, timestamp tokens, which can dramatically simplify the design of advanced dataflow systems. Drawing inspiration from work on capability systems, a timestamp token is an in-memory object that can be held by an operator and provides the ability to produce timestamped data messages on a specific dataflow edge. A timestamp token does not require repeated interaction between system and operator to confirm, exercise, or release this ability. Instead, an operator accumulates and summarizes its interactions with its timestamp tokens. The system collects this information when most convenient, maintains a view of outstanding timestamp tokens, and provides summaries of potential input timestamps to each operator.

Timestamp tokens make it relatively simple to introduce dataflow idioms that would be complicated or impossible in other systems. Although we have not previously reported on them since we designed and implemented them in an open-source data-processor, timestamp tokens have been in use for several years in various research and production projects. Faucet [18] uses timestamp tokens to allow opera-
A dataflow program is expressed as a directed graph $(V,E)$ where nodes $V$ represent data transformations and edges $E$ represent the communication channels between the nodes. A dataflow system instantiates multiple workers and provides each with the dataflow graph. At runtime, the system exchanges data messages between the nodes, as the messages cross dataflow edges, and each worker independently applies data transformations in response to received data, producing output messages that are further exchanged and processed.

In modern dataflow systems, messages bear a logical timestamp $t$, and dataflow operators maintain or advance timestamps as they process messages. The system and operators collaborate to track outstanding messages by timestamp, so that operators can learn when certain input timestamps are complete and it is appropriate to produce the corresponding output. Most commonly, the system provides the operator with a “watermark” or “frontier” indicating a lower bound on future timestamps the operator may observe, and the operator communicates to the system a lower bound on the timestamps it might still need to produce as output. The system is responsible for collecting and integrating the information from all operators, as well as the messages produced and retired, to provide correct lower bounds to the operators.

2.1 Representative dataflow systems

We now walk through several representative systems and relate their moving parts to dataflow coordination.

**Spark** models a computation as an acyclic dataflow graph, but without distinct logical times: inputs in Spark are either "complete" or "not yet complete". The Spark system tracks which inputs are complete and signals operators when their inputs are all complete and the operator can run to completion. Operators report back to the system as they complete their outputs.

**Flink** models computations as an acyclic dataflow graph, with integer logical times. Flink streams (dataflow edges) report an increasing integer “watermark” lower-bounding the timestamps the stream may yet produce. These watermarks are interleaved in the stream of data itself, and each operator is required to produce them in their output streams as well. Flink does not have a centralized scheduler, and maintains a fresh view of its outputs only through the continued introduction of new watermarks in the dataflow inputs.

**Naiad** models computations as a potentially cyclic dataflow graph, with partially ordered logical times. Naiad operators request “notifications” at specified logical times, and Naiad invokes a callback only once it determines that all messages bearing that logical time have been delivered. Naiad does not present operators with lower bounds for their inputs, and instead requires operators to defer the responsibility of scheduling to the system itself, in part because the logic for doing so requires a holistic view of the dataflow graph and all other pending notifications.

Each of these systems introduce new opportunities for concurrency, and corresponding performance gains on important tasks. However, no one system unifies the work of the others. We believe that unifying this work, and laying the groundwork for more advanced behav-
3 Timestamp tokens

We propose that dataflow systems and operator logic can coordinate precisely, efficiently, and ergonomically by explicitly handling in-memory tokens that represent their ability to produce outgoing data in the future. We borrow and adapt this idiom from capability systems (e.g., object-capability systems [13, 15], capability-based protection and security [12, 21], hardware capabilities [5, 11]). Similarly to capabilities\(^1\), a timestamp token represents a computing object – an operator output – and the actions that can be performed with respect to that object: the production of data at timestamp \(t\) and dataflow location \(l\).

Following Naiad we refer to the pair of timestamp \(t\) and location \(l\) as a pointstamp \((t, l)\). A location can be either a node in \(V\) or an edge in \(E\).

Definition. A timestamp token is a coordination primitive that names an associated pointstamp \((t, l)\), and which gives its holder the ability to produce messages with timestamp \(t\) at location \(l\).

The location for a timestamp token is typically one of the output edges of the operator that holds it.

Notwithstanding any other similarities to capabilities, our interest is in the information that holding timestamp tokens communicates to others. The system tracks the set of live timestamp tokens and summarizes this information to operators as frontiers: lower bounds on the timestamps that operators may yet observe in their inputs. By downgrading (to future timestamps) or discarding their held timestamp tokens, operators allow frontiers to advance and the computation as a whole to make forward progress.

3.1 The timestamp token life-cycle

Each dataflow operator is initially provided with a timestamp token for each of its output edges, each bearing some minimal “zero” timestamp. This gives each operator the opportunity to be a source of timestamped messages, even without receiving input messages. For many operators, their first actions will be to discard these timestamp tokens, by which they release their ability to produce output messages unprompted, and unblock the dataflow system at the same time.

\(^1\)“Each capability [...] locates by means of a pointer some computing object, and indicates the actions that the computation may perform with respect to that object.” [13]

As a dataflow operator executes, it can receive, exercise, downgrade, and discard timestamp tokens (Figure 2). Operators receive timestamped input messages, each of which provides a timestamp token at that timestamp for each of the operator’s outputs. Operators can produce timestamped output messages as long as they hold a timestamp token with the corresponding timestamp and output edge. Lastly, operators can arbitrarily hold, downgrade (to future timestamps), and discard their timestamp tokens as their logic dictates.

The dataflow system is informed of the net changes to the number of timestamp tokens for each pointstamp, but only passively in response to operator actions, rather than actively as a gatekeeper. Through this information the system can inform dataflow operators about the consequences of operator actions, without the specific details of the reasons for those actions.

3.2 Coordination

The coordination state of the dataflow system is the set of timestamp tokens, which when combined with the dataflow graph determines lower bounds for the timestamps at each operator input. As the set of timestamp tokens evolves these lower bounds advance, and the dataflow system has the responsibility of informing operators as this happens. The difference with timestamp tokens is that operators drive the production of this information, instead of the system itself.\(^2\)

Operators have a great deal of flexibility in how (or even if) they respond to changes in their input frontiers (timestamp lower bounds). Certain streaming operators like map and filter can be oblivious to this information and process data as it arrives. Synchronous reduction operators like reduce should await the indication that they have received all inputs for a timestamp before they apply their reduction function and produce output. Hybrid operators like count may per-

\(^2\)For example, Naiad does not allow operators to hold tokens across invocations; Timely Dataflow (without timestamp tokens) does, by allowing operators to participate directly (and often incorrectly) in the coordination protocol. Here, timestamp tokens are respectively more expressive, and safer.
form some accumulation in place and await the frontier advancing before producing the final tally for each timestamp. In each case the operator responds to input data and changes in its input frontiers, with output data and changes in its held timestamp tokens, but does not otherwise expose complexity to the system.

4 Implementation

We implemented timestamp tokens for Timely Dataflow [4] in the Rust programming language [3][19]. In our implementation, timestamp tokens are Rust types that wrap a timestamp, a location, and a bookkeeping data structure shared with the system. Operator logic manipulates timestamp tokens through their methods—cloning, downgrading, and dropping them—which update the shared data structure with integer changes to the numbers of timestamp tokens at each timestamp and location.

The timely dataflow system drains shared bookkeeping data structures outside of operator logic but on the same thread of control, which ensures the changes reflect atomic operator actions. Following Naiad’s progress tracking protocol, these collected changes are broadcast among unsynchronized workers. Any subset of atomic updates forms a conservative view of the coordination state (the outstanding timestamp tokens) and is sufficient to maintain a conservative view of timestamp lower bounds for each operator across the otherwise asynchronous workers.

The Rust [23] language provides several features that simplify our implementation. Rust is type-safe, and users cannot fabricate timestamp tokens outside of unsafe code. Rust also does not allow users to destruct private struct fields, ensuring that we protect the shared bookkeeping data structure from direct user manipulation. Rust’s affine type system ensures that users cannot casually copy timestamp tokens without explicit method calls, which allow us to interpose and increment counts. Finally, Rust eagerly invokes destructor logic, so that dropping a timestamp token is immediately visible to the system.

Our implementation is general enough to reproduce idioms from other systems, with no overhead. We have implemented Naiad notifications in library operator logic, and if in each invocation an operator processes only their least timestamp they reproduce Naiad’s notification behavior. We can also implement Flink-style watermarks, with operators that explicitly hold timestamp tokens for their output watermarks and downgrade them whenever these watermarks advance. Both these idioms are helpful but restrictive, and they are enforced system-wide in prior work. Our intent is that operators should be able to choose the most appealing idiom, or new idioms as appropriate, without requiring the system to change as well.

This generality is not without some ergonomic cost; prior systems could more easily encourage operators make forward progress. Flink operators should eventually bring their output watermarks in line with their input watermarks, and Naiad operators should respond to notifications with something other than a re-notification request for the same time. From experience, user operators can more easily “lose track” of a timestamp token, for example when used as a key in a hash map and not discarded once its associated values have been processed. We use Rust’s type system to raise the programmers awareness, by providing operators only a “timestamp token option”, which the operator must then specifically retain to receive a timestamp token. Rust’s lifetime system ensures at compile time that the options themselves can not be held by an operator, forcing it to explicitly retain or pass on timestamp token options.

4.1 Timestamp tokens in code

We present an extract of the main definitions of the timestamp token Rust API and implementation in Figure 3. A `TimestampToken` wraps a `Timestamp` and a bookkeeping data structure shared with the system. These fields are private and the operator code cannot directly access or mutate them. The bookkeeping data structure records the location for which the `TimestampToken` is valid, which will be checked by the system should the `TimestampToken` be exercised to send data. Operators may hold any number of `TimestampToken`s.

Three methods, `downgrade`, `clone`, and `drop`, are the only ways user code can directly manipulate the number of timestamp tokens at a pointstamp (without the use of Rust’s `unsafe` keyword). The number of timestamp tokens at a pointstamp is indirectly manipulated by sending timestamped messages to the location of that pointstamp, through the `session` method.

Operator code can directly downgrade a timestamp token to a later timestamp with `downgrade`. This reduces the operator’s ability to produce output at the wrapped timestamp, potentially to the point that the system can unblock downstream operators, though not beyond the timestamp downgraded to. The implementation of `downgrade` updates the bookkeeping datastructure to inform the system of the net changes to the number of timestamp tokens for each pointstamp.

Operator code can also call into Rust’s `clone` (deep copy) and `drop` (destructor) methods on
Custom implementations of these two methods respectively increment and decrement pointstamp counts for the wrapped timestamp in the bookkeeping data-structure. A drop call is automatically inserted by the Rust compiler whenever an object goes out of scope, and makes it much less likely that an operator will fail to release a timestamp token.

In order to transmit data along an output dataflow edge, an operator must express a timestamp token. Access to outputs is guarded by an OutputHandle, whose method session will create an active Session only when presented a reference to a TimestampToken. The Session is only valid for the wrapped timestamp, or timestamps greater than it. Rust’s lifetime system ensures that the TimestampToken cannot be modified or dropped as long as the Session is active (until it is dropped). As long as the Session is available to user code, the TimestampToken is guaranteed to still exist unmodified. Sent data arrive at the destination bearing a timestamp token that can be used by the recipient.

### 4.2 Ergonomic modifications

The core timestamp token code is explained in the previous section, but we have also made several ergonomic improvements in an attempt to minimize the chance of unintended errors.

In addition to TimestampToken objects, which are owned by the code and data structures that reference them, we also provide a TimestampTokenRef structure that cannot be held longer than a fairly narrow lexical scope. To acquire an owned token, user code must explicitly call retain which then results in a TimestampToken. We have found this reduces the incidences of user code unintentionally capturing and indefinitely holding a timestamp token, thereby stalling out dataflows.

Both TimestampToken and TimestampTokenRef implement a Rust trait TimestampTokenTrait that allows system code (specifically session) to accept either. This allows users to bypass the retain method and create a Session from a token reference, avoiding some syntax but importantly also avoiding bookkeeping when timestamp token ownership is not needed.

Timestamp tokens by default update shared bookkeeping data structures, but do not force the system to immediately act upon the changes they reflect. The operators that house an OutputHandle inform the system that it should consult the shared bookkeeping, when the operator yields control. Several variants of TimestampToken take specific action when modified, including notifying the system that it should accept any updates and act on them. This allows these timestamp tokens to be used outside of the operators their pointstamps reference, and are especially useful for manual control of inputs to a dataflow when the logic cannot easily be encapsulated in an operator.

These modifications do not change the core behavior of timestamp tokens, but instead demonstrate how rough edges can be sanded down using layers atop timestamp tokens.

```rust
/// The ability to send data with a /// certain timestamp on a dataflow edge.
pub struct TimestampToken<T: Timestamp> { 
  time: T,
  bookkeeping: Bookkeeping<T>,
}

impl<T: Timestamp> TimestampToken<T> { 
  /// The timestamp associated with this /// timestamp token.
  pub fn time(&self) -> &T { ... } 
  
  /// Downgrades the timestamp token to /// one corresponding to 'new_time'.
  pub fn downgrade(&mut self, new_time: &T) { ... }
}

impl<T: Timestamp> Clone for TimestampToken<T> {
  fn clone(&self) -> TimestampToken<T> { ... }
}

impl<T: Timestamp> Drop for TimestampToken<T> {
  fn drop(&mut self) { ... }
}

impl<T: Timestamp, ...> OutputHandle<T, ...> { 
  /// Obtains a session that can send data /// at the timestamp associated with /// timestamp token 'tok'.
  pub fn session(&mut self, tok: &TimestampToken) -> Session<T, ...> { ... }
}
```

Figure 3: An extract of the timestamp token API and implementation in timely dataflow. We use circled letters, similar to ①, to mark points of interest in the code.

### 5 Example

We use the example of tumbling windowed average to demonstrate the life-cycle of timestamp tokens and how it generates coordination information. This operator re-
receives timestamped integer-valued messages and reports the average every 10 timestamp units, at the timestamp of the start of the next window. The operator produces no output for windows which contain no data. Figure 5 list the example code.

Importantly, this is code that one can write to introduce the behavior of a tumbling window to a system. It is not code that an end user should be expected to write each time they want a tumbling window. Rather, it can be written once, and then end users can simply invoke the method with appropriate parameters.

Figure 4 is a snapshot of the execution after the output for the window [0, 10) has been produced. At this stage the operator maintains the current average for open windows (for which some data has been received but not necessarily all data) and a timestamp token to produce the output at the timestamp of the next open window (in the Figure, time 20).

The operator has great flexibility in how it implements its specification. For example, the operator can choose to retain only the timestamp tokens for timestamps that are not greater than some other held timestamp token, reducing system interaction at the cost of local bookkeeping. The operator can use ordered data structures to efficiently retire multiple windows at once, should the frontier advance suddenly. The operator can maintain partial aggregations for out-of-order data while still being clear at which times they might emerge.

We walk through the sample code in 5.1 and call out the benefits timestamp tokens provide in 5.2.

### 5.1 Example code

Figure 5 shows the code listing for one of the many possible implementation of the tumbling windowed average operator described in § 5. The code presented closely resembles the real implementation of the operator, with some minor syntax modifications to aid readability and avoid Rust-isms that can be unfamiliar to the reader. Although detailed, this is the implementation expected of the system implementor; we expect end users would then access this functionality through a layer of abstraction rather than write it themselves.

The outer anonymous function \( \odot \) is invoked once by the system to initialize the operator with a default timestamp token \( \odot \) at time 0 \( \odot \), which is immediately dropped \( \odot \). The operator initializes an ordered map \( \odot \) to store partial state for open windows: the timestamp for the end of the window maps to a tuple carrying the corresponding timestamp token and the partial WindowData \( \odot \) (the partial sum and count).

The inner anonymous function \( \odot \) contains the operator logic that is invoked every time the operator is scheduled. For each batch of input messages at a certain timestamp \( \odot \), it computes the end-of-window timestamp \( \odot \) from the message timestamp wrapped in the timestamp token \( \odot \) (in the form of a TimestampTokenRef). If it has not seen data for this window before \( \odot \), it captures \( \odot \) the timestamp token, immediately downgrades it to the end-of-window timestamp, and stores it along with initialized empty window data into the windows map.

The timestamp tokens stored in the map implicitly inform the coordination state of the operator: the system is informed of pointstamp changes after each invocation of the operator logic caused by retain, downgrade, and drop (when a timestamp token is finally removed from the map and dropped).

For each batch of input messages the operator logic obtains a mutable reference \( \odot \) to the corresponding window data in the map, and updates the partial sum and count with each data point. Processing of new input concludes here.

The operator logic then needs to determine which windows have closed and emit the computed averages for them. This information is based on the set of live timestamp tokens in the system and is summarized by the system as per-input frontiers at each operator: input.frontier() \( \odot \). In general, timestamps in timely dataflow can be multidimensional and result in frontiers defined by multiple minima, but in this case we know that timestamps, and consequently frontiers, are represented by a single unsigned integer value. The frontier value represents the lower bound on timestamps that may still appear on the input: consequently we can safely retire all windows with end-of-window timestamps up to, but excluding, the frontier (target.ns).

We leverage the map order to iterate over all open windows up to target.ns \( \odot \), and because we stored timestamp tokens alongside the window data, we obtain them during iteration \( \odot \) and can immediately lever-
User-defined structure to maintain window data:

```
/// struct WindowData { pub sum: u64, pub count: u64 }
```

```
fn singleton_frontier(frontier: &MutableAntichain<u64>) -> u64 {
    frontier.frontier().first().cloned().unwrap_or(u64::MAX)
}
```

```
// The 'unary_frontier' method defines a new operator from a anonymous function that specifies its logic.
stream.unary_frontier(
    Exchange::new(|x| x % (peers as u64)), "tumbling_window", tok, _info | { 
        assert!(*tok.time() == 0);
        std::mem::drop(tok);
    let mut windows: BTreeMap<u64, (TimestampToken<u64>, WindowData)> = BTreeMap::new();
    for (tok_ref, batch) in input {
        let window_ts = round_up_to_multiple(*tok_ref.time(), WINDOW_SIZE);
        if !windows.contains_key(&window_ts) {
            let mut window_tok = tok_ref.retain();
            window_tok.downgrade(&window_ts);
            windows.insert(window_ts, (window_tok, WindowData { sum: 0, count: 0 }));
        }
        let (_, ref mut window_data) = windows.get_mut(&window_ts).unwrap();
        for d in batch {
            window_data.sum += *d; window_data.count += 1;
        }
    }
    let target_ts = singleton_frontier(input.frontier());
    for (_, (tok, window)) in windows.range(0..target_ts) {
        output.session(&tok).give(window.sum as f64 / window.count as f64);
    }
    windows.remove_range(0..target_ts);
})
```

Figure 5: A possible implementation of the tumbling window average operator described in § 5. We use circled letters, similar to Z, to mark points of interest in the code.

5.2 Benefits

The operator implementation above has several benefits that are prevented in other systems.

In a Spark-like system, where an operator is scheduled for each distinct timestamp, the operator would be unable to retire blocks of times concurrently. This limitation harms the throughput of data loading, and lowers the operator's throughput when bursts of differently timestamped data arrive. With timestamp tokens entire intervals of time can be closed at once, and the operator can perform all consequent work concurrently.

In a Flink-like system, the operator must be continually interrogated to advance its output watermark. Even if the operator input is idle for periods of time, the operator must remain active to inform downstream operators that there is no data. This scenario is more common than it might seem, with monitoring applications like fraud detection in which one wants to quickly confirm the absence of results. With timestamp tokens the system can bypass the operator entirely, reducing compute load and the critical path latency.

In a Naiad-like system, the operator must defer scheduling to the system. Should a batch of times be retired at once, as when a watermark finally arrives, the operator must repeatedly yield to the system and be re-
invoked with advancing timestamps. With timestamp
tokens the operator can perform this work on its own,
using an efficient ordered data structure.

In addition, timestamp tokens avoid restrictions on
dataflow structure, for example the requirement (seen
in Spark and Flink) that dataflow graphs be acyclic.
Each of these benefits derive from involving the system
less, instead providing the operator with both more in-
formation and more agency.

6 Building with timestamp to-
kens

Timestamp tokens have been in use for several years.
In this section, we relate examples where developers
and researchers found timestamp tokens to be espe-
cially helpful in building frameworks that implement
new dataflow programming patterns. In each case,
timestamp tokens and specialized operator logic allowed
projects to avoid re-implementing parts of the timely
dataflow system itself.

6.1 Co-operative control flow

Dataflow operators may run for a long time or produce
large amounts of output data, and should yield con-
trol so that other operators can execute and potentially
retire some of the output data. However, Naiad’s exe-
cution model asks an operator to run to completion for
each notification, and the return of control is an indica-
tion that the operator has completed its task. Times-
tamp tokens allow operators to yield control without
yielding the right to resume execution and produce out-
put in the future.

Faucet [18] uses timestamp tokens to implement user-
level flow control. This mechanism supports dataflow
operators that may produce unboundedly large num-
bers of output messages for each input. Faucet oper-
ators produce outputs up to a certain limit and then
yield control until these messages are retired. When-
ever an operator yields due to a reached limit, it retains
the timestamp token to indicate it has further output
to produce. This design allowed the Faucet authors to
implement flow control in user code, without requiring
modifications to the underlying system.

6.2 Fine-grained timestamps

Systems that track real time may process events with
timestamps denominated in nanoseconds. Naiad as-
sumes responsibility for ordering all events with distinct
timestamps, and for high-resolution timestamps this can
overwhelm the system. Timestamp tokens provide a
mechanism for the operator to determine the granular-
ity at which it reports outstanding timestamps to the
system, without involving the system in each timestamp
that is processed.

In DD [20], each event has a potentially unique times-
tamp, and operators receive and must react to a stream
of such events. Rather than present each timestamp to
the timely dataflow system, DD’s operator implement-
ations batch messages into “intervals”. An operator
retains the least timestamp tokens for the times of un-
batched messages it holds, and as the operator’s frontier
advances the operator creates new batches containing all
events whose timestamps are not in advance of the new
frontier. The operator uses its current timestamp tokens
to produce any output corresponding to the batch, and
then downgrades its timestamp tokens once, to the new
lower envelope of its un-batched messages. This design
allows the operators to interact with the host timely
dataflow system at a coarse granularity, independent of
the timestamp granularity.

6.3 Optimized scheduling

Timely dataflow computations may act on general par-
tially ordered timestamps, and with large numbers of
outstanding events it may be unclear which events
should be processed next. A system like Naiad stores all
events in an unsorted list and performs a sequential pass
through this list in each scheduling round, limiting the
minimum latency. Alternately, stream processors that
only act on totally ordered timestamps can use prior-
ity queues to quickly extract only the relevant events.
Timestamp tokens provide operators the ability to orga-
nize their schedulable work themselves, without pushing
their implementation into the system itself.

In Megaphone [16], a migration mechanism for timely
dataflow, its authors implemented the NEXMark bench-
mark which contains a variety of streaming computa-
tions, and in particular a variety of windowed computa-
tions. These computations have timestamps that are de-
nominated in nanoseconds, and in one case a windowed
computation with a 12 hour continuous slide (and so, an
effectively unbounded number of distinct timestamps in
play at any time). Their implementation uses prior-
ity queues of timestamp tokens to schedule the work
in these specific operators, providing millisecond laten-
cies without compromising the ability of the rest of the
system to handle partially-ordered timestamps.

6.4 Formalisation and safety proof

Timestamp tokens express a clean interface between op-
erators and the system. An effort to formalize and verify
the safety of the core coordination component of timely
dataflow\cite{8} uses timestamp tokens as a basis for the formalization: the authors can precisely model what actions each instance of an operator can perform, in contrast with previous formalisation work\cite{6} that pre-dates timestamp tokens.

\section{Evaluation}

Our hypothesis is that by reducing systems complexity and granting more control on scheduling to individual operators, timestamp tokens remove the scheduling bottleneck that prevents modern data processing systems from reaching higher throughputs and lower latencies. We evaluate this hypothesis with a set of microbenchmarks designed to compare the different coordination mechanisms in prior art with timestamp tokens (§ 7.2 and § 7.3) and with more complex workloads that attempt to replicate real-world operating conditions (§ 7.4). We hope to observe that timestamp tokens operate robustly in all settings where any coordination mechanism avoids collapse, and is never substantially worse than the best coordination mechanism.

We compare timestamp tokens against the Naiad-style notification API already available in Timely Dataflow. In order to compare with Flink-style watermarks without the confounding factor of running on a different platform (like Flink’s), we re-implemented Flink’s watermarks technique on the same communication and scheduling framework provided by Timely Dataflow. In some of the experiments (Figures 6 and 7), where the technique selected has limited impact on performance, timestamp tokens and Flink-style watermarks achieve nearly identical latency, showing that our implementation does not unfairly disadvantage watermarks.

We observe that timestamp tokens avoid the collapse that notifications experience for high numbers of distinct timestamps (Figure 7), and the collapse that watermarks experience for complex dataflows (Figure 8). In all cases, timestamp tokens remain among the best approaches.

\subsection{Experimental setup}

We run all experiments on a CloudLab\cite{14} server with one AMD EPYC 7452 with 32 physical cores and 128GB of RAM. We disable simultaneous multithreading (SMT) and we pin each timely dataflow worker to a distinct physical core.

Our open-loop testing harness supplies the input at a specified rate, even if the system itself becomes less responsive. We record the observed latency in units of nanoseconds in a histogram of logarithmically-sized bins. If the system becomes overloaded and end-to-end latency becomes greater than 1 second, the testing harness regards the experiment as failed.

\subsection{Microbenchmarks}

Our microbenchmarks use a simple dataflow program that consists of a single stateful operator that computes the overall rolling count of unique words observed on the inputs. Every time the operator receives a word, it updates the internal count, and sends an output message with the updated value.

To determine the effectiveness of handling fine-grained timestamps with various techniques, we generate input at a given constant rate and assign different timestamps to each input tuple based on when it was generated. The assigned timestamps are quantized to powers-of-two ranging from $2^8$ to $2^{16}$ nanoseconds (“ns” in the following). A timestamp quantum of $2^8$ns means that regardless of the input rate, there can be at most $1 \times 10^6$ distinct timestamps in the ingested data per second. For example, with a timestamp quantum of $2^3$ns (256ns), at most 4 million timestamps per second can be generated.

Varying the size of the quantum allows us to evaluate how well a mechanism can handle coarser or finer timestamp granularities. With a smaller timestamp quantum, the system can provide higher time resolution in the output it produces. As previously discussed, with Naiad-style notifications, the operator needs to interact with the system for each logical time it processes, and for which it requires a notification.

\subsubsection{Varying timestamp granularity}

Figure 6 shows the achieved median, p999 (99.9%), and maximum latency when we vary the granularity of the timestamp quantization under different offered loads: 32 million tuples/sec is below the maximum throughput achievable with fine timestamp granularity by at least some of the coordination mechanisms, and 64 million tuples/sec represent a very high load that all mechanisms cannot sustain with a timestamp quantum of $2^{13}$ns or finer. The performance pattern at lower loads is similar to what we report for 32 million tuples/sec, but with lower latency.

All mechanisms display similar performance characteristics when not overloaded, with two notable exceptions. First, notifications are unable to handle a timestamp granularity below $2^{13}$ns; this is because they require an interaction between the operator logic and the system for each timestamp. That is not the case for both watermarks and tokens, that can handle any timestamp quantization. Second, the maximum latency for watermarks is 2x smaller than timestamp tokens for times-
Figure 6: Latency for a single-operator ("word-count") dataflow with Flink-style watermarks, Naiad-style notifications, and with timestamp tokens. We run the workloads on 8 physical cores at three different offered loads. We report median, p999, and maximum latency as we vary the timestamp quantization. Note the different scales on the y axes of the plots.

tamp quantization above 2^{14}: for this extremely simple single-operator dataflow, watermarks can have slightly lower overhead at the tail.

At very high load (64 million tuples/sec) (i) all mechanisms have significantly higher tail latency and cannot handle the finest timestamp granularities, (ii) both watermarks and timestamp tokens can handle timestamp granularities finer than notifications, (iii) notifications achieve better p999 (when they are able to sustain the load) possibly due to additional synchronization imposed by the mechanism, and (iv) watermarks display slightly higher median latency at this load.

In this microbenchmark, timestamp tokens perform essentially on par with watermarks when not over- loaded, and behave better when the system is over- loaded. Notifications are unable to handle highly granular timestamps in the input data even at lower loads, because every timestamp requires an interaction between the operator logic and the system.

7.2.2 Scaling

Figure 7 shows the scaling behaviour of the microbenchmark word-count dataflow. At the coarser timestamp quantization granularity, all techniques display nearly identical scaling characteristics. In both strong and weak scaling we can see the system’s and techniques’ minor inefficiencies starting to affect the reported latency above around 6 workers. At the finer timestamp granularity, Naiad-style notifications fail to keep up with load at any scale, while watermarks and timestamp tokens display similar behaviour. This demonstrates that timestamp tokens do not negatively affect scaling.

7.3 Complex dataflow fragments

As discussed in § 5.2, timestamp tokens do not require continual interaction between the operator and the system to retire timestamps, in particular when an operator is idle for a period of time. To measure the performance benefit of not having to invoke each operator for each successive timestamp, even if no work needs to be performed, we construct a dataflow with a variable sequence of no-op operators (from 8 to 256 no-op operators connected as a sequential pipeline).

Timestamp tokens and Naiad-style notifications always calculate operator input frontiers (low watermarks) as if each channel between two consecutive operators may exchange data between workers. For Flink-style watermarks we need to distinguish between a scenario where a cross-worker exchange happens at each step (and watermarks are broadcast) and an additional (unrealistic) scenario at the other end of the spectrum where no cross-worker data exchange takes place. A real-world dataflow is likely to have a mix of worker-local and cross-worker channels, and would likely sit somewhere between these two extremes.

Figure 8 shows the performance impact of handling timestamps for a sequence of idle operators of varying length. Timestamp tokens, and Naiad-style notifications, and the Flink-style watermark configuration without cross-worker exchange (watermarks-P) have almost identical performance that is only marginally affected by the length of the operator chain (Figure 8a) and by the workload scale (Figure 8b). In this scenario watermarks-P has an unrealistic advantage because no coordination information is ever exchanged between workers: each processor operates as a separate unit, and thus does not incur any coordination cost.

When configured to perform exchanges for every inter-operator channel (watermarks-X) the latency for Flink-style watermarks degrades linearly with the number of operators in the sequence (Figure 8a) because each operator has to be invoked to forward the watermark which then needs to be broadcast to all other
(a) Weak scaling. We vary the number of workers and the offered load, which is fixed at 2 million tuples per second per worker. Note that Naiad-style notification fail to keep up with load for timestamp quantum = 2^{16}\text{ns}.

(b) Strong scaling. We vary the number of workers while keeping the offered load fixed at 20 million tuples per second. For quantum = 2^{16}\text{ns}, notifications fail to keep up with load with less than 2 workers, and other mechanisms fail with less than 3 workers. For quantum = 2^{8}\text{ns}, all configurations fail with less than 3 workers, and notifications fail with any number of workers.

Figure 7: Weak and strong scaling for the word-count workload. Note the different scales on the y axes of the plots. For both weak and strong scaling we report results with the timestamp quantization set to either 2^{16}\text{ns} or 2^{8}\text{ns}.

(a) Sequence of no-op operators. We vary the number of operators in the sequence and the offered load in terms of timestamps/sec. We run the workloads on 8 physical cores.

(b) Weak scaling for an operator sequence of 256 no-op operators. We vary the number of workers while keeping the offered load fixed at 15K and 250K timestamps per second, per worker.

Figure 8: Impact of a long sequence of operators in the dataflow graph. For Flink-style watermarks we consider two dataflows: one with all-worker exchanges at every stage (watermarks-X) and one where operators form pipelines that are connected locally on each worker (watermarks-P). Note the different scales on the y axes of the plots.
operators. This also fundamentally limits scalability: `watermarks-X` has to process watermarks proportional to the length of the sequence times the number of workers, resulting in high latency even at moderate scale.

By not requiring interaction with each operator for each timestamp, timestamp tokens matches or outperforms other techniques when handling complex inactive dataflow fragments.

7.4 NEXMark

To evaluate timestamp tokens’ performance impact on a realistic, albeit simple, data processing use case, we extended the timely dataflow implementation of the NEXMark queries open sourced[17] by the authors of Megaphone[16]. The original implementation leverages timestamp tokens as described in § 6.3. We augmented it by writing the same queries with Naiad-style notifications and Flink-style watermarks.

The NEXMark suite models an auction site in which a high-volume stream of users, auctions, and bids arrive, and standing queries are maintained reflecting a variety of relational queries. For the purpose of this experiment, we focus on queries that result in multi-operator dataflows (Q4 and Q7). Megaphone [16] describes the query semantics; for our purposes we only need to highlight that Q4 has a two-stage dataflow where one of the operators handles tokens to calculate a data-dependent windowed maximum, and Q7 has two stateful operators with two consecutive data exchanges.

Timestamp tokens avoid the collapse that notifications exhibit for Q4 due to overwhelming numbers of distinct timestamps, and are competitive with watermarks (improving on them slightly for Q7). These queries are relatively simple, only a few dataflow stages, and timestamp tokens do not have much room to distinguish themselves from watermarks.

8 Conclusions

We introduced timestamp tokens, a coordination primitive for dataflow systems. Timestamp tokens decouple the sophistication of operator scheduling logic from the task of system-wide coordination. Operators can add sophistication to their own implementations, including flow control, fine-grained timestamps, and optimized data structures. At the same time, timestamp tokens simplify the surrounding system, whose role in scheduling no longer needs to be the bottleneck it once was.

Looking forward, we think timestamp tokens have potential to drive other new dataflow programming idioms, without increasing system complexity. We are especially interested in timestamp tokens as dataflow breakpoints, and how holding timestamp tokens provides external agents the opportunity to suspend execution without fundamentally restructuring dataflow programs.

Finally, we’ve been delighted by the force multiplier of investing in general dataflow primitives. Many projects quickly and safely implemented new system behavior writing only application-level code. We should have more well-considered primitives and fewer systems.

References

[1] Apache storm. http://storm.apache.org. Accessed: 2021-12-14.

[2] Mpi forum, mpi-3.1. https://www.mpi-forum.org/docs/. Accessed: 2021-12-14.

[3] Rust programming language. https://www.rust-lang.org. Accessed: 2021-12-14.

[4] Timely dataflow. https://github.com/TimelyDataflow/timely-dataflow. Accessed: 2021-12-14.

[5] Edward A. Feustel. On the advantages of tagged architecture. Computers, IEEE Transactions on, C-22:644 – 656, 08 1973.

[6] Martín Abadi, Frank McSherry, Derek Gordon Murray, and Thomas L. Rodeheffer. Formal analysis of a distributed algorithm for tracking progress. In Dirk Beyer and Michele Boreale, editors, Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMODDS/FORTE 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings, volume 7892 of Lecture Notes in Computer Science, pages 5–19. Springer, 2013.

[7] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In Very Large Data Bases, pages 734–746, 2013.

[8] Matthias Brun, Sára Decova, Andrea Lattuada, and Dmitriy Traytel. Verified progress tracking for timely dataflow. In Liron Cohen and Cezary Kaliszyk, editors, 12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference), volume 193 of LIPIcs, pages 10:1–10:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
| NEXmark Q4 | latency (milliseconds) | tokens | notifications | watermarks |
|-----------|------------------------|--------|--------------|------------|
| tuples/sec | workers | p50  | p999 | max | p50  | p999 | max | p50  | p999 | max |
| 4M | 4 | 0.62  | 1.25 | 1.9 | DNF  | 0.25  | 0.59  | 1.25 |
| 4M | 8 | 0.52  | 0.98  | 1.51 | DNF  | 0.29  | 0.56  | 1.44 |
| 4M | 12 | 0.59  | 1.02 | 5.77 | DNF  | 0.38  | 0.56  | 2.49 |
| 6M | 4 | 1.31  | 2.62  | 4.19 | DNF  | 0.72  | 2.36  | 4.19 |
| 6M | 8 | 1.25  | 2.36  | 2.88 | DNF  | 0.51  | 1.02  | 3.54 |
| 8M | 4 | 2.03  | 3.93  | 11.53 | DNF  | 0.95  | 2.62  | 3.67 |
| 8M | 8 | DNF  | DNF  | DNF | DNF  | DNF  | DNF  | DNF |
| 8M | 12 | DNF  | DNF  | DNF | DNF  | DNF  | DNF  | DNF |
| NEXmark Q7 | latency (milliseconds) | tokens | notifications | watermarks |
| tuples/sec | workers | p50  | p999 | max | p50  | p999 | max | p50  | p999 | max |
| 4M | 4 | 0.06  | 0.09  | 0.31 | 0.06  | 0.09  | 0.22 | 0.07  | 0.11  | 0.36 |
| 4M | 8 | 0.06  | 0.1  | 0.46 | 0.06  | 0.09  | 0.41 | 0.08  | 0.13  | 0.66 |
| 4M | 12 | 0.06  | 0.11  | 0.82 | 0.06  | 0.1  | 0.72 | 0.1  | 0.17  | 0.79 |
| 6M | 4 | 0.06  | 0.1  | 0.23 | 0.06  | 0.1  | 0.38 | 0.07  | 0.11  | 0.26 |
| 6M | 8 | 0.06  | 0.1  | 0.46 | 0.06  | 0.1  | 0.44 | 0.09  | 0.13  | 0.66 |
| 6M | 12 | 0.07  | 0.11  | 0.92 | 0.06  | 0.11  | 0.95 | 0.11  | 0.18  | 0.82 |
| 8M | 4 | 0.07  | 0.1  | 0.39 | 0.07  | 0.11  | 0.24 | 0.07  | 0.11  | 0.62 |
| 8M | 8 | 0.07  | 0.11  | 0.56 | 0.06  | 0.1  | 0.44 | 0.09  | 0.15  | 0.69 |
| 8M | 12 | 0.07  | 0.11  | 1.02 | 0.07  | 0.11  | 0.92 | 0.11  | 0.19  | 1.31 |

Figure 9: End-to-end processing latency for NEXmark query 4 and query 6. We scale the number of workers while keeping the total load fixed at 4, 6, and 8 million tuples/sec. We report median, p999, and maximum latency in milliseconds. For Q4 note that Naiad-style notifications cannot sustain the load for any of the configurations and timestamp tokens and Flink-style watermarks cannot sustain higher loads with 4-8 workers.

[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 36*(4), 2015.

[10] Badrish Chandramouli, Jonathan Goldstein, and David Maier. On-the-fly progress detection in iterative stream queries. In *International Conference on Very Large Databases (VLDB), Lyon, France*, August 2009.

[11] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the pdp-11: Architectural support for a memory-safe c abstract machine. *SIGPLAN Not.*, 50(4):117–130, March 2015.

[12] Ellis Cohen and David Jefferson. Protection in the hydra operating system. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles, SOSP ’75*, pages 141–160. New York, NY, USA, 1975. ACM.

[13] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM, 9*(3):143–155, March 1966.

[14] Dmitry Duplyakin, Robert Ricci, Aleksander Marcic, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[15] R. S. Fabry. Capability-based addressing. *Commun. ACM, 17*(7):403–412, July 1974.

[16] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. Megaphone: Latency-conscious state mi-
igration for distributed streaming dataflows. *Proc. VLDB Endow.*, 12(9):1002–1015, 2019.

[17] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, and Lorenzo Selvatici. Github: strymon-system / megaphone. https://github.com/strymon-system/megaphone, 2020. Accessed 2021-12-14.

[18] Andrea Lattuada, Frank McSherry, and Zaheer Chothia. Faucet: A user-level, modular technique for flow control in dataflow engines. In *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond - BeyondMR ’16*. ACM Press, 2016.

[19] Nicholas D. Matsakis and Felix S. Klock, II. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT ’14, pages 103–104, New York, NY, USA, 2014. ACM.

[20] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. Shared arrangements: practical inter-query sharing for streaming dataflows. *Proc. VLDB Endow.*, 13(10):1793–1806, 2020.

[21] S. J. Mullender, G. van Rossum, A. S. Tananbaum, R. van Renesse, and H. van Staveren. Amoeba: a distributed operating system for the 1990s. *Computer*, 23(5):44–53, May 1990.

[22] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP ’13*. ACM Press, 2013.

[23] Aaron Weiss. Oxide: The essence of rust. *CoRR*, abs/1903.00982, 2019.

[24] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulf Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI’08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[25] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[26] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP ’13, pages 423–438, New York, NY, USA, 2013. ACM.