MillWheel: Fault-Tolerant Stream Processing at Internet Scale

Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, Sam Whittle
Google
{takidau, alexgb, kayab, chernyak, haberman, relax, sgmc, millsd, pgn, samuelw}@google.com

ABSTRACT

MillWheel is a framework for building low-latency data-processing applications that is widely used at Google. Users specify a directed computation graph and application code for individual nodes, and the system manages persistent state and the continuous flow of records, all within the envelope of the framework's fault-tolerance guarantees.

This paper describes MillWheel's programming model as well as its implementation. The case study of a continuous anomaly detector in use at Google serves to motivate how many of MillWheel's features are used. MillWheel's programming model provides a notion of logical time, making it simple to write time-based aggregations. MillWheel was designed from the outset with fault tolerance and scalability in mind. In practice, we find that MillWheel's unique combination of scalability, fault tolerance, and a versatile programming model lends itself to a wide variety of problems at Google.

1. INTRODUCTION

Stream processing systems are critical to providing content to users and allowing organizations to make faster and better decisions, particularly because of their ability to provide low latency results. Users want real-time news about the world around them. Businesses are likewise interested in the value provided by real-time intelligence sources such as spam filtering and intrusion detection. Similarly, scientists must cull noteworthy results from immense streams of raw data.

Streaming systems at Google require fault tolerance, persistent state, and scalability. Distributed systems run on thousands of shared machines, any of which can fail at any time. Model-based streaming systems, like anomaly detectors, depend on predictions that are generated from weeks of data, and their models must be updated on-the-fly as new data arrives. Scaling these systems by orders of magnitude should not cause a commensurate increase in the operational cost of building and maintaining the system.

Programming models for distributed systems, like MapReduce [11], hide the framework's implementation details in the background, allowing users to create massive distributed systems that are simply expressed. By allowing users to focus solely on their application logic, this kind of programming model allows users to reason about the semantics of their system without being distributed systems experts. In particular, users are able to depend on framework-level correctness and fault-tolerance guarantees as axiomatic, vastly restricting the surface area over which bugs and errors can manifest. Supporting a variety of common programming languages further drives adoption, as users can leverage the utility and convenience of existing libraries in a familiar idiom, rather than being restricted to a domain-specific language.

MillWheel is such a programming model, tailored specifically to streaming, low-latency systems. Users write application logic as individual nodes in a directed compute graph, for which they can define an arbitrary, dynamic topology. Records are delivered continuously along edges in the graph. MillWheel provides fault tolerance at the framework level, where any node or any edge in the topology can fail at any time without affecting the correctness of the result. As part of this fault tolerance, every record in the system is guaranteed to be delivered to its consumers. Furthermore, the API that MillWheel provides for record processing handles each record in an idempotent fashion, making record delivery occur exactly once from the user's perspective. MillWheel checkpoints its progress at fine granularity, eliminating any need to buffer pending data at external senders for long periods between checkpoints.

Other streaming systems do not provide this combination of fault tolerance, versatility, and scalability. Spark Streaming [34] and Sonora [32] do excellent jobs of efficient checkpointing, but limit the space of operators that are available to user code. S4 [26] does not provide fully fault-tolerant persistent state, while Storm's [23] exactly-once mechanism for record delivery, Trident [22], requires strict transaction ordering to operate. Attempts to extend the batch-processing model of MapReduce and Hadoop [4] to provide low-latency systems result in compromised flexibility, such as the operator-specific dependence on Replicated Distributed Datasets [33] in Spark Streaming. Streaming SQL systems [1] [2] [5] [6] [21] [24] provide succinct and simple solutions to many streaming problems, but intuitive state abstractions and complex application logic (e.g. matrix multiplication) are more naturally expressed using the operational flow of an imperative language rather than a declarative language like SQL.

Our contributions are a programming model for streaming systems and an implementation of the MillWheel framework.

- We have designed a programming model that allows for complex streaming systems to be created without distributed systems expertise.
- We have built an efficient implementation of the MillWheel

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy.
Proceedings of the VLDB Endowment, Vol. 6, No. 11
Copyright 2013 VLDB Endowment 2150-8097/13/09... $ 10.00.

framework that proves its viability as both a scalable and fault-tolerant system.

The rest of this paper is organized as follows. Section 2 outlines a motivating example for the development of MillWheel, and the corresponding requirements that it imposes. Section 3 provides a high-level overview of the system. Section 4 defines the fundamental abstractions of the MillWheel model and Section 5 discusses the API that MillWheel exposes. Section 6 outlines the implementation of fault tolerance in MillWheel, and Section 7 covers the general implementation. Section 8 provides experimental results to illustrate the performance of MillWheel, and Section 9 discusses related work.

2. MOTIVATION AND REQUIREMENTS

Google's Zeitgeist pipeline is used to track trends in web queries. To demonstrate the utility of MillWheel's feature set, we will examine the requirements of the Zeitgeist system. This pipeline ingests a continuous input of search queries and performs anomaly detection, outputting queries which are spiking or dipping as quickly as possible. The system builds a historical model of each query, so that expected changes in traffic (e.g. for "television listings" in the early evening) will not cause false positives. It is important that spiking or dipping queries be identified as quickly as possible. For example, Zeitgeist helps power Google's Hot Trends service, which depends on fresh information. The basic topology of this pipeline is shown in Figure 1.

In order to implement the Zeitgeist system, our approach is to bucket records into one-second intervals and to compare the actual traffic for each time bucket to the expected traffic that the model predicts. If these quantities are consistently different over a non-trivial number of buckets, then we have high confidence that a query is spiking or dipping. In parallel, we update the model with the newly received data and store it for future use.

Persistent Storage: It is important to note that this implementation requires both short- and long-term storage. A spike may only last a few seconds, and thus depend on state from a small window of time, whereas model data can correspond to months of continuous updates.

Low Watermarks: Some Zeitgeist users are interested in detecting dips in traffic, where the volume for a query is uncharacteristically low (e.g. if the Egyptian government turns off the Internet). In a distributed system with inputs from all over the world, data arrival time does not strictly correspond to its generation time (the search time, in this case), so it is important to be able to distinguish whether a flurry of expected Arabic queries at $t = 1296167641$ is simply delayed on the wire, or actually not there. MillWheel addresses this by providing a low watermark for incoming data for each processing stage (e.g. Window Counter, Model Calculator), which indicates that all data up to a given timestamp has been received. The low watermark tracks all pending events in the distributed system. Using the low watermark, we are able to distinguish between the two example cases – if the low watermark advances past time t without the queries arriving, then we have high confidence that the queries were not recorded, and are not simply delayed. This semantic also obviates any requirement of strict monotonicity for inputs – out-of-order streams are the norm.

Duplicate Prevention: For Zeitgeist, duplicate record deliveries could cause spurious spikes. Further, exactly-once processing is a requirement for MillWheel's many revenue-processing customers, all of whom can depend on the correctness of the framework implementation rather than reinventing their own deduplication mechanism. Users do not have to write code to manually roll back state updates or deal with a variety of failure scenarios to maintain correctness.

With the above in mind, we offer our requirements for a stream processing framework at Google, which are reflected in MillWheel:

- Data should be available to consumers as soon as it is published (i.e. there are no system-intrinsic barriers to ingesting inputs and providing output data).
- Persistent state abstractions should be available to user code, and should be integrated into the system's overall consistency model.
- Out-of-order data should be handled gracefully by the system.
- A monotonically increasing low watermark of data timestamps should be computed by the system.
- Latency should stay constant as the system scales to more machines.
- The system should provide exactly-once delivery of records.

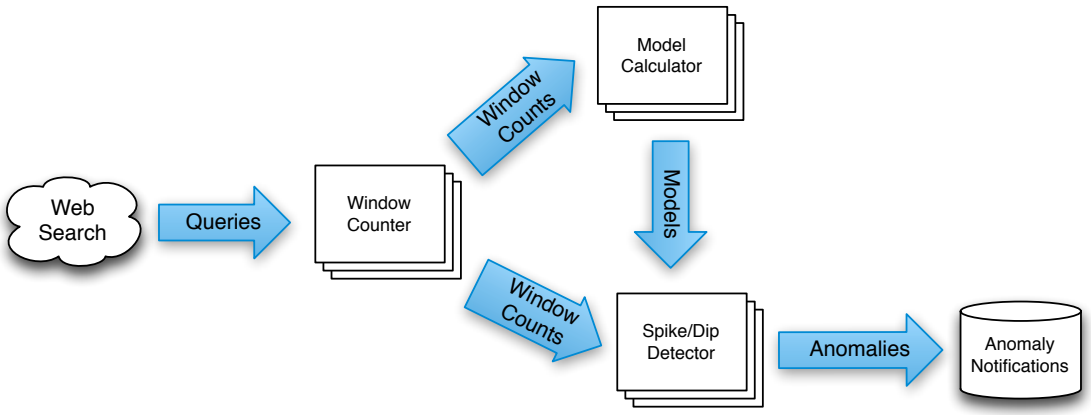

Figure 1: Input data (search queries) goes through a series of MillWheel computations, shown as distributed processes. The output of the system is consumed by an external anomaly notification system.

3. SYSTEM OVERVIEW

At a high level, MillWheel is a graph of user-defined transformations on input data that produces output data. We call these transformations *computations*, and define them more extensively below. Each of these transformations can be parallelized across an arbitrary number of machines, such that the user does not have to concern themselves with load-balancing at a fine-grained level. In the case of Zeitgeist, shown in Figure 1, our input would be a continuously arriving set of search queries, and our output would be the set of queries that are spiking or dipping.

Abstractly, inputs and outputs in MillWheel are represented by (key, value, timestamp) triples. While the *key* is a metadata field with semantic meaning in the system, the *value* can be an arbitrary byte string, corresponding to the entire record. The context in which user code runs is scoped to a specific key, and each computation can define the keying for each input source, depending on its logical needs. For example, certain computations in Zeitgeist would likely select the search term (e.g. "cat videos") as the key, in order to compute statistics on a per-query basis, while other computations might select geographic origin as the key, in order to aggregate on a per-locale basis. The *timestamps* in these triples can be assigned an arbitrary value by the MillWheel user (but they are typically close to wall clock time when the event occurred), and MillWheel will calculate low watermarks according to these values. If a user were aggregating per-second counts of search terms (as in Zeitgeist, illustrated in Figure 2), then they would want to assign a timestamp value corresponding to the time at which the search was performed.

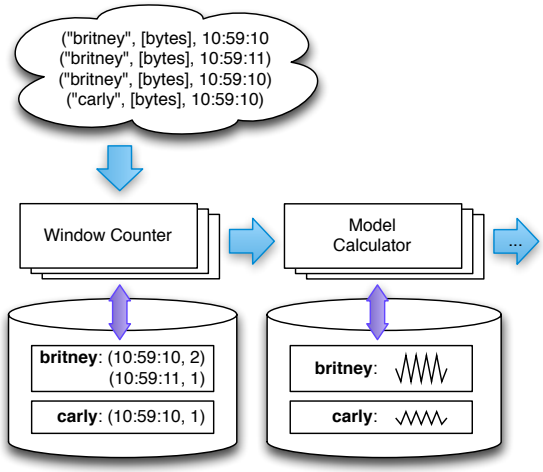

Figure 2: Aggregating web searches into one-second buckets and updating models using persistent per-key state. Each computation has access to its own per-key state, which it updates in response to input records.

Collectively, a pipeline of user computations will form a data flow graph, as outputs from one computation become inputs for another, and so on. Users can add and remove computations from a topology dynamically, without needing to restart the entire system. In manipulating data and outputting records, a computation can combine, modify, create, and drop records arbitrarily.

MillWheel makes record processing idempotent with regard to the framework API. As long as applications use the state and communication abstractions provided by the system, failures and retries are hidden from user code. This keeps user code simple and understandable, and allows users to focus on their application logic. In the context of a computation, user code can access a per-key, per-computation persistent store, which allows for powerful per-key aggregations to take place, as illustrated by the Zeitgeist example. The fundamental guarantee that underlies this simplicity follows:

Delivery Guarantee: All internal updates within the MillWheel framework resulting from record processing are atomically checkpointed per-key and records are delivered exactly once. This guarantee does not extend to external systems.

With this high-level concept of the system in mind, we will expand upon the individual abstractions that make up MillWheel in the next section.

4. CORE CONCEPTS

MillWheel surfaces the essential elements of a streaming system, while providing clean abstractions. Data traverses our system via a user-defined, directed graph of computations (Figure 3), each of which can manipulate and emit data independently.

```
computation SpikeDetector {
  input_streams {
    stream model_updates {
      key_extractor = 'SearchQuery'
    }
    stream window_counts {
      key_extractor = 'SearchQuery'
    }
  }
  output_streams {
    stream anomalies {
      record_format = 'AnomalyMessage'
    }
  }
}
```

Figure 3: Definition of a single node in a MillWheel topology. Input streams and output streams correspond to directed edges in the graph.

4.1 Computations

Application logic lives in computations, which encapsulate arbitrary user code. Computation code is invoked upon receipt of input data, at which point user-defined actions are triggered, including contacting external systems, manipulating other MillWheel primitives, or outputting data. If external systems are contacted, it is up to the user to ensure that the effects of their code on these systems is idempotent. Computation code is written to operate in the context of a single key, and is agnostic to the distribution of keys among different machines. As illustrated in Figure 4, processing is serialized per-key, but can be parallelized over distinct keys.

4.2 Keys

Keys are the primary abstraction for aggregation and comparison between different records in MillWheel. For every record in the system, the consumer specifies a *key extraction* function, which assigns a key to the record. Computation code is run in the context of a specific key and is only granted access to state for that specific key. For example, in the Zeitgeist system, a good choice of key for query records would be the text of the query itself, since we need to aggregate counts and compute models on a per-query basis. Alternately, a spam detector might choose a cookie fingerprint as a key, in order to block abusive behavior. Figure 5 shows different consumers extracting different keys from the same input stream.

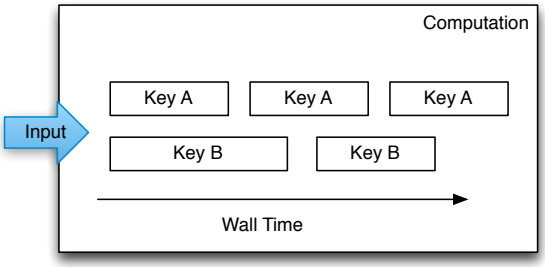

Figure 4: Per-key processing is serialized over time, such that only one record can be processed for a given key at once. Multiple keys can be run in parallel.

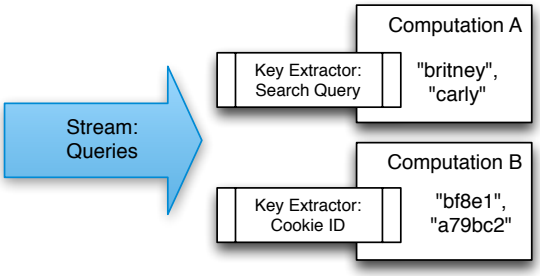

Figure 5: Multiple computations can extract different keys from the same stream. Key extractors are specified by the consumer of a stream.

4.3 Streams

Streams are the delivery mechanism between different computations in MillWheel. A computation subscribes to zero or more input streams and publishes one or more output streams, and the system guarantees delivery along these channels. Key-extraction functions are specified by each consumer on a per-stream basis, such that multiple consumers can subscribe to the same stream and aggregate its data in different ways. Streams are uniquely identified by their names, with no other qualifications – any computation can subscribe to any stream, and can *produce* records (*productions*) to any stream.

4.4 Persistent State

In its most basic form, persistent state in MillWheel is an opaque byte string that is managed on a per-key basis. The user provides serialization and deserialization routines (such as translating a rich data structure in and out of its wire format), for which a variety of convenient mechanisms (e.g. Protocol Buffers [13]) exist. Persistent state is backed by a replicated, highly available data store (e.g. Bigtable [7] or Spanner [9]), which ensures data integrity in a way that is completely transparent to the end user. Common uses of state include counters aggregated over windows of records and buffered data for a join.

4.5 Low Watermarks

The low watermark for a computation provides a bound on the timestamps of future records arriving at that computation.

Definition: We provide a recursive definition of low watermarks based on a pipeline's data flow. Given a computation, A, let the *oldest work* of A be a timestamp corresponding to the oldest unfinished (in-flight, stored, or pending-delivery) record in A. Given this, we define the *low watermark* of A to be

$$\min(\text{oldest work of A}, \text{low watermark of C} : \text{C outputs to A})$$

If there are no input streams, the low watermark and oldest work values are equivalent.

Low watermark values are seeded by *injectors*, which send data into MillWheel from external systems. Measurement of pending work in external systems is often an estimate, so in practice, computations should expect a small rate of *late records* – records behind the low watermark – from such systems. Zeitgeist deals with this by dropping such data, while keeping track of how much data was dropped (empirically around 0.001% of records). Other pipelines retroactively correct their aggregates if late records arrive. Though this is not reflected in the above definition, the system guarantees that a computation's low watermark is monotonic even in the face of late data.

By waiting for the low watermark of a computation to advance past a certain value, the user can determine that they have a complete picture of their data up to that time, as previously illustrated by Zeitgeist's dip detection. When assigning timestamps to new or aggregate records, it is up to the user to pick a timestamp no smaller than any of the source records. The low watermark reported by the MillWheel framework measures known work in the system, shown in Figure 6.

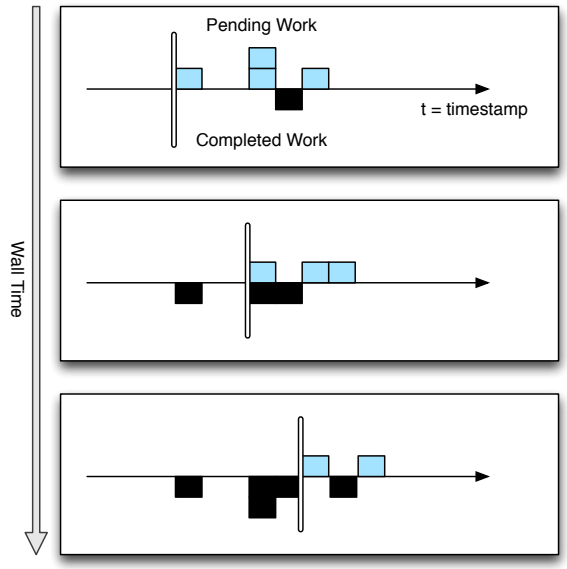

Figure 6: The low watermark advances as records move through the system. In each snapshot, pending records are shown above the timestamp axis, and completed records are shown below. New records appear as pending work in successive snapshots, with timestamp values ahead of the watermark. Data is not necessarily processed in-order, and the low watermark reflects *all* pending work in the system.

4.6 Timers

Timers are per-key programmatic hooks that trigger at a specific wall time or low watermark value. Timers are created and run in the context of a computation, and accordingly can run arbitrary code. The decision to use a wall time or low watermark value is dependent on the application – a heuristic monitoring system that

wants to push hourly emails (on the hour, regardless of whether data was delayed) might use wall time timers, while an analytics system performing windowed aggregates could use low watermark timers. Once set, timers are guaranteed to fire in increasing timestamp order. They are journaled in persistent state and can survive process restarts and machine failures. When a timer fires, it runs the specified user function and has the same exactly-once guarantee as input records. A simple implementation of dips in Zeitgeist would set a low watermark timer for the end of a given time bucket, and report a dip if the observed traffic falls well below the model's prediction.

The use of timers is optional – applications that do not have the need for time-based barrier semantics can skip them. For example, Zeitgeist can detect spiking queries without timers, since a spike may be evident even without a complete picture of the data. If the observed traffic already exceeds the model's prediction, delayed data would only add to the total and increase the magnitude of the spike.

5. API

In this section, we give an overview of our API as it relates to the abstractions in Section 4. Users implement a custom subclass of the Computation class, shown in Figure 7, which provides methods for accessing all of the MillWheel abstractions (state, timers, and productions). Once provided by the user, this code is then run automatically by the framework. Per-key serialization is handled at the framework level, and users do not need to construct any per-key locking semantics.

```
class Computation {
  // Hooks called by the system.
  void ProcessRecord(Record data);
  void ProcessTimer(Timer timer);

  // Accessors for other abstractions.
  void SetTimer(string tag, int64 time);
  void ProduceRecord(
      Record data, string stream);
  StateType MutablePersistentState();
};
```

Figure 7: The MillWheel API consists of a parent Computation class with access to per-key timers, state, and productions. Users implement application logic by overriding ProcessRecord and ProcessTimer.

5.1 Computation API

The two main entry points into user code are provided by the ProcessRecord and ProcessTimer hooks, depicted in Figure 8, which are triggered in reaction to record receipt and timer expiration, respectively. Collectively, these constitute the application logic of a computation.

Within the execution of these hooks, MillWheel provides system functions to fetch and manipulate per-key state, produce additional records, and set timers. Figure 9 illustrates the interaction between these mechanisms. It draws upon our Zeitgeist system to show the use of persistent state and timers in detecting dips in the query stream. Again, note the absence of failure-recovery logic, which is all handled automatically by the framework.

5.2 Injector and Low Watermark API

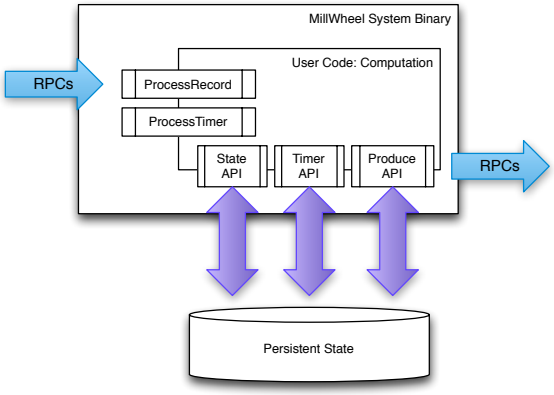

Figure 8: The MillWheel system invokes user-defined processing hooks in response to incoming RPCs. User code accesses state, timers, and productions through the framework API. The framework performs any actual RPCs and state modifications.

At the system layer, each computation calculates a low watermark value for all of its pending work (in-progress and queued deliveries). Persistent state can also be assigned a timestamp value (e.g. the trailing edge of an aggregation window). This is rolled up automatically by the system in order to provide API semantics for timers in a transparent way – users rarely interact with low watermarks in computation code, but rather manipulate them indirectly through timestamp assignation to records.

Injectors: Injectors bring external data into MillWheel. Since injectors seed low watermark values for the rest of the pipeline, they are able to publish an *injector low watermark* that propagates to any subscribers among their output streams, reflecting their potential deliveries along those streams. For example, if an injector were ingesting log files, it could publish a low watermark value that corresponded to the minimum file creation time among its unfinished files, as shown in Figure 10.

An injector can be distributed across multiple processes, such that the aggregate low watermark of those processes is used as the injector low watermark. The user can specify an expected set of injector processes, making this metric robust against process failures and network outages. In practice, library implementations exist for common input types at Google (log files, pubsub service feeds, etc.), such that normal users do not need to write their own injectors. If an injector violates the low watermark semantics and sends a late record behind the low watermark, the user's application code chooses whether to discard the record or incorporate it into an update of an existing aggregate.

6. FAULT TOLERANCE

6.1 Delivery Guarantees

Much of the conceptual simplicity of MillWheel's programming model hinges upon its ability to take non-idempotent user code and run it as if it were idempotent. By removing this requirement from computation authors, we relieve them of a significant implementation burden.

6.1.1 Exactly-Once Delivery

Upon receipt of an input record for a computation, the MillWheel framework performs the following steps:

```
// Upon receipt of a record, update the running
// total for its timestamp bucket, and set a
// timer to fire when we have received all
// of the data for that bucket.
void Windower::ProcessRecord(Record input) {
  WindowState state(MutablePersistentState());
  state.UpdateBucketCount(input.timestamp());
  string id = WindowID(input.timestamp())
  SetTimer(id, WindowBoundary(input.timestamp()));
}

// Once we have all of the data for a given
// window, produce the window.
void Windower::ProcessTimer(Timer timer) {
  Record record =
      WindowCount(timer.tag(),
                  MutablePersistentState());
  record.SetTimestamp(timer.timestamp());
  // DipDetector subscribes to this stream.
  ProduceRecord(record, "windows");
}

// Given a bucket count, compare it to the
// expected traffic, and emit a Dip event
// if we have high enough confidence.
void DipDetector::ProcessRecord(Record input) {
  DipState state(MutablePersistentState());
  int prediction =
    state.GetPrediction(input.timestamp());
  int actual = GetBucketCount(input.data());
  state.UpdateConfidence(prediction, actual);
  if (state.confidence() >
       kConfidenceThreshold) {
    Record record =
        Dip(key(), state.confidence());
    record.SetTimestamp(input.timestamp());
    ProduceRecord(record, "dip-stream");
  }
}
```

Figure 9: ProcessRecord and ProcessTimer definitions for computations which compute window counts and dips based on an existing model using low watermark timers.

- The record is checked against deduplication data from previous deliveries; duplicates are discarded.
- User code is run for the input record, possibly resulting in pending changes to timers, state, and productions.
- Pending changes are committed to the backing store.
- Senders are ACKed.
- Pending downstream productions are sent.

As an optimization, the above operations may be coalesced into a single checkpoint for multiple records. Deliveries in MillWheel are retried until they are ACKed in order to meet our at-least-once requirement, which is a prerequisite for exactly-once. We retry because of the possibility of networking issues and machine failures on the receiver side. However, this introduces the case where a receiver may crash before it has a chance to ACK the input record, even if it has persisted the state corresponding to successful processing of that record. In this case, we must prevent duplicate processing when the sender retries its delivery.

The system assigns unique IDs to all records at production time. We identify duplicate records by including this unique ID for the record in the same atomic write as the state modification. If the same record is later retried, we can compare it to the journaled ID, and discard and ACK the duplicate (lest it continue to retry indefinitely). Since we cannot necessarily store all duplication data in-memory, we maintain a Bloom filter of known record fingerprints, to provide a fast path for records that we have provably never seen before. In the event of a filter miss, we must read the backing store to determine whether a record is a duplicate. Record IDs for past deliveries are garbage collected after MillWheel can guarantee that all internal senders have finished retrying. For injectors that frequently deliver late data, we delay this garbage collection by a corresponding slack value (typically on the order of a few hours). However, exactly-once data can generally be cleaned up within a few minutes of production time.

```
// Upon finishing a file or receiving a new
// one, we update the low watermark to be the
// minimum creation time.
void OnFileEvent() {
  int64 watermark = kint64max;
  for (file : files) {
    if (!file.AtEOF())
      watermark =
        min(watermark, file.GetCreationTime());
  }
  if (watermark != kint64max)
    UpdateInjectorWatermark(watermark);
}
```

Figure 10: A simple file injector reports a low watermark value that corresponds to the oldest unfinished file.

6.1.2 *Strong Productions*

Since MillWheel handles inputs that are not necessarily ordered or deterministic, we checkpoint produced records before delivery in the same atomic write as state modification. We call this pattern of checkpointing before record production *strong productions*. Take the example of a computation that aggregates by wall time, that is emitting counts downstream. Without a checkpoint, it would be possible for that computation to produce a window count downstream, but crash before saving its state. Once the computation came back up, it might receive another record (and add it to the count) before producing the same aggregate, creating a record that was bit-wise distinct from its predecessor but corresponded to the same logical window! In order to handle this case correctly, the downstream consumer would need complex conflict resolution logic. With MillWheel, however, the simple solution just works, because the user's application logic has been made into an idempotent operation by the system guarantees.

We use a storage system such as Bigtable [7], which efficiently implements blind writes (as opposed to read-modify-write operations), making checkpoints mimic the behavior of a log. When a process restarts, the checkpoints are scanned into memory and replayed. Checkpoint data is deleted once these productions are successful.

6.1.3 *Weak Productions and Idempotency*

Taken together, the combination of strong productions and exactly-once delivery makes many computations idempotent with regard to system-level retries. However, some computations may already be idempotent, regardless of the presence of these guarantees (which come with a resource and latency cost). Depending on the semantic needs of an application, strong productions and/or exactly-once can be disabled by the user at their discretion. At the system level, disabling exactly-once can be accomplished simply by skipping the deduplication pass, but disabling strong productions requires more attention to performance.

For *weak productions*, rather than checkpointing record productions before delivery, we broadcast downstream deliveries optimistically, prior to persisting state. Empirically, this introduces a new problem, in that the completion times of consecutive stages of the pipeline are now strictly coupled as they wait for downstream ACKs of records. Combined with the possibility of machine failure, this can greatly increase end-to-end latency for straggler productions as pipeline depth increases. For example, if we assume (rather pessimistically) that there is a 1% chance that any machine will fail during a given minute, then the probability that we will be waiting on at least one failure increases disastrously with pipeline depth – for a pipeline of depth 5, a given production could have nearly a 5% chance of experiencing a failure every minute! We ameliorate this by checkpointing a small percentage of straggler pending productions, allowing those stages to ACK their senders. By selectively checkpointing in this way, we can both improve end-to-end latency and reduce overall resource consumption.

In Figure 11, we show this checkpointing mechanism in action. Computation A produces to Computation B, which immediately produces to Computation C. However, Computation C is slow to ACK, so Computation B checkpoints the production after a 1-second delay. Thus, Computation B can ACK the delivery from Computation A, allowing A to free any resources associated with the production. Even when Computation B subsequently restarts, it is able to recover the record from the checkpoint and retry delivery to Computation C, with no data loss.

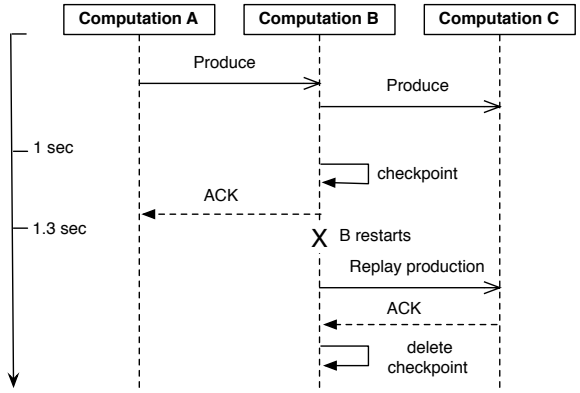

Figure 11: Weak production checkpointing prevents straggler productions from occupying undue resources in the sender (Computation A) by saving a checkpoint for Computation B.

The above relaxations would be appropriate in the case of a pipeline with idempotent computations, since retries would not affect correctness, and downstream productions would also be retry-agnostic. A real-world example of an idempotent computation is a stateless filter, where repeated deliveries along input streams will not change the result.

6.2 State Manipulation

In implementing mechanisms to manipulate user state in MillWheel, we discuss both the "hard" state that is persisted to our backing store and the "soft" state which includes any in-memory caches or aggregates. We must satisfy the following user-visible guarantees:

- The system does not lose data.
- Updates to state must obey exactly-once semantics.
- All persisted data throughout the system must be consistent at any given point in time.
- Low watermarks must reflect all pending state in the system.
- Timers must fire in-order for a given key.

To avoid inconsistencies in persisted state (e.g. between timers, user state, and production checkpoints), we wrap all per-key updates in a single atomic operation. This results in resiliency against process failures and other unpredictable events that may interrupt the process at any given time. As mentioned previously, exactly-once data is updated in this same operation, adding it to the per-key consistency envelope.

As work may shift between machines (due to load balancing, failures, or other reasons) a major threat to our data consistency is the possibility of zombie writers and network remnants issuing stale writes to our backing store. To address this possibility, we attach a sequencer token to each write, which the mediator of the backing store checks for validity before allowing the write to commit. New workers invalidate any extant sequencers before starting work, so that no remnant writes can succeed thereafter. The sequencer is functioning as a lease enforcement mechanism, in a similar manner to the Centrifuge [3] system. Thus, we can guarantee that, for a given key, only a single worker can write to that key at a particular point in time.

This single-writer guarantee is also critical to the maintenance of soft state, and it cannot be guaranteed by depending on transactions. Take the case of a cache of pending timers: if a remnant write from another process could alter the persisted timer state after said cache was built, the cache would be inconsistent. This situation is illustrated by Figure 12, where a zombie process (B) issues a transaction that is delayed on the wire, in response to a production from A. Before the transaction begins, B's successor, B-prime, performs its initial scan of pending timers. After this scan completes, the transaction is applied and A is ACKed, leaving B-prime with incomplete timer state. The lost timer could be orphaned indefinitely, delaying any of its output actions by an arbitrary amount of time. Clearly, this is unacceptable for a latency-sensitive system.

Furthermore, this same situation could occur with a checkpointed production, where it would remain unknown to the system by eluding an initial scan of the backing store. This production would then not be accounted for in the low watermark until it was discovered, and in the intervening time, we might be reporting an erroneous low watermark value to consumers. Furthermore, since our low watermarks are monotonically increasing, we are unable to correct an erroneous advancement in the value. By violating our low watermark guarantees, a variety of correctness violations could occur, including premature timer firings and incomplete window productions.

In order to quickly recover from unplanned process failures, each computation worker in MillWheel can checkpoint its state at an arbitrarily fine granularity (in practice, sub-second or per-record granularity is standard, depending on input volume). Our use of always-consistent soft state allows us to minimize the number of occasions when we must scan these checkpoints to specific cases – machine failures or load-balancing events. When we do perform scans, these can often be asynchronous, allowing the computation to continue processing input records while the scan progresses.

7. SYSTEM IMPLEMENTATION

7.1 Architecture

MillWheel deployments run as distributed systems on a dynamic set of host servers. Each computation in a pipeline runs on one or

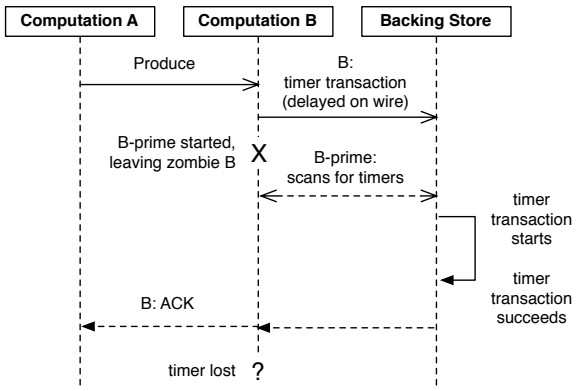

Figure 12: Transactions cannot prevent inconsistencies in soft state. Orphaned transactions may commit after a read-only scan has completed, causing inconsistent state in MillWheel's timer system.

more machines, and streams are delivered via RPC. On each machine, the MillWheel system marshals incoming work and manages process-level metadata, delegating to the appropriate user computation as necessary.

Load distribution and balancing is handled by a replicated master, which divides each computation into a set of owned lexicographic key intervals (collectively covering all key possibilities) and assigns these intervals to a set of machines. In response to increased CPU load or memory pressure (reported by a standard per-process monitor), it can move these intervals around, split them, or merge them. Each interval is assigned a unique sequencer, which is invalidated whenever the interval is moved, split, or merged. The importance of this sequencer was discussed in Section 6.2.

For persistent state, MillWheel uses a database like Bigtable [7] or Spanner [9], which provides atomic, single-row updates. Timers, pending productions, and persistent state for a given key are all stored in the same row in the data store.

MillWheel recovers from machine failures efficiently by scanning metadata from this backing store whenever a key interval is assigned to a new owner. This initial scan populates in-memory structures like the heap of pending timers and the queue of checkpointed productions, which are then assumed to be consistent with the backing store for the lifetime of the interval assignment. To support this assumption, we enforce single-writer semantics (per computation worker) that are detailed in Section 6.2.

7.2 Low Watermarks

In order to ensure data consistency, low watermarks must be implemented as a sub-system that is globally available and correct. We have implemented this as a central authority (similar to OOP [19]), which tracks all low watermark values in the system and journals them to persistent state, preventing the reporting of erroneous values in cases of process failure.

When reporting to the central authority, each process aggregates timestamp information for all of its owned work. This includes any checkpointed or pending productions, as well as any pending timers or persisted state. Each process is able to do this efficiently by depending on the consistency of our in-memory data structures, eliminating the need to perform any expensive queries over the backing data store. Since processes are assigned work based on key intervals, low watermark updates are also bucketed into key intervals, and sent to the central authority.

To accurately compute system low watermarks, this authority must have access to low watermark information for all pending and persisted work in the system. When aggregating per-process updates, it tracks the completeness of its information for each computation by building an interval map of low watermark values for the computation. If any interval is missing, then the low watermark corresponds to the last known value for the missing interval until it reports a new value. The authority then broadcasts low watermark values for all computations in the system.

Interested consumer computations subscribe to low watermark values for each of their sender computations, and thus compute the low watermark of their input as the minimum over these values. The reason that these minima are computed by the workers, rather than the central authority, is one of consistency: the central authority's low watermark values should always be at least as conservative as those of the workers. Accordingly, by having workers compute the minima of their respective inputs, the authority's low watermark never leads the workers', and this property is preserved.

To maintain consistency at the central authority, we attach sequencers to all low watermark updates. In a similar manner to our single-writer scheme for local updates to key interval state, these sequencers ensure that only the latest owner of a given key interval can update its low watermark value. For scalability, the authority can be sharded across multiple machines, with one or more computations on each worker. Empirically, this can scale to 500,000 key intervals with no loss in performance.

Given a global summary of work in the system, we are able to optionally strip away outliers and offer heuristic low watermark values for pipelines that are more interested in speed than accuracy. For example, we can compute a 99% low watermark that corresponds to the progress of 99% of the record timestamps in the system. A windowing consumer that is only interested in approximate results could then use these low watermark values to operate with lower latency, having eliminated its need to wait on stragglers.

In summary, our implementation of low watermarks does not require any sort of strict time ordering on streams in the system. Low watermarks reflect both in-flight and persisted state. By establishing a global source of truth for low watermark values, we prevent logical inconsistencies, like low watermarks moving backwards.

8. EVALUATION

To illustrate the performance of MillWheel, we provide experimental results that are tailored towards key metrics of stream-processing systems.

8.1 Output Latency

A critical metric for the performance of streaming systems is latency. The MillWheel framework supports low latency results, and it keeps latency low as the distributed system scales to more machines. To demonstrate the performance of MillWheel, we measured record-delivery latency using a simple, single-stage MillWheel pipeline that buckets and sorts numbers. This resembles the many-to-many shuffle that occurs between successive computations that are keyed differently, and thus is a worst case of sorts for record delivery in MillWheel. Figure 13 shows the latency distribution for records when running over 200 CPUs. Median record delay is 3.6 milliseconds and 95th-percentile latency is 30 milliseconds, which easily fulfills the requirements for many streaming systems at Google (even 95th percentile is within human reaction time).

This test was performed with strong productions and exactly-once disabled. With both of these features enabled, median latency jumps up to 33.7 milliseconds and 95th-percentile latency to 93.8 milliseconds. This is a succinct demonstration of how idempotent computations can decrease their latency by disabling these two features.

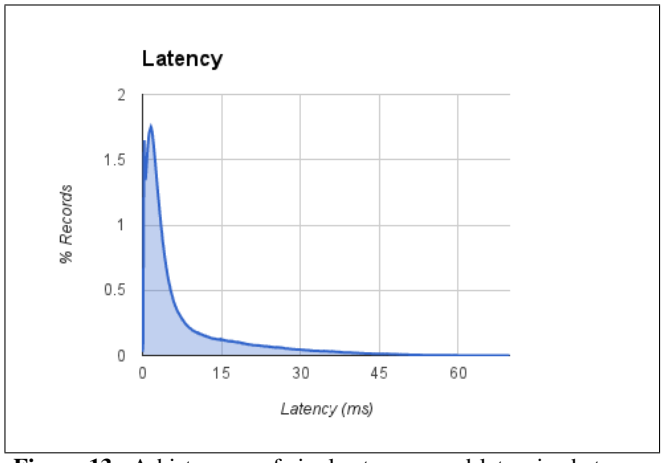

Figure 13: A histogram of single-stage record latencies between two differently-keyed stages.

To verify that MillWheel's latency profile scales well with the system's resource footprint, we ran the single-stage latency experiment with setups ranging in size from 20 CPUs to 2000 CPUs, scaling input proportionally. Figure 14 shows that median latency stays roughly constant, regardless of system size. 99th-percentile latency does get significantly worse (though still on the order of 100ms). However, tail latency is expected to degrade with scale – more machines mean that there are more opportunities for things to go wrong.

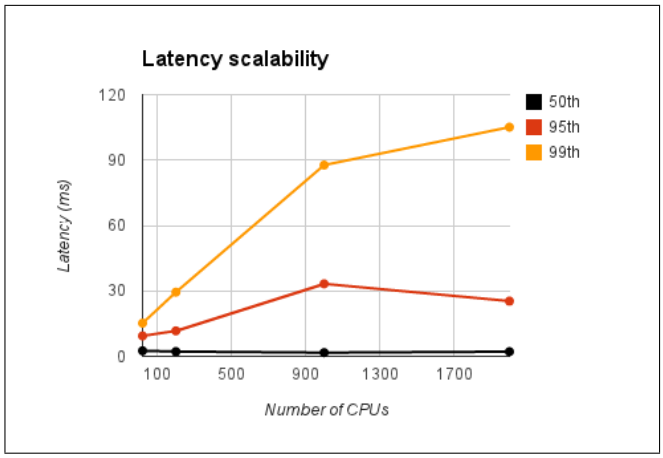

Figure 14: MillWheel's average latency does not noticeably increase as the system's resource footprint scales.

8.2 Watermark Lag

While some computations (like spike detection in Zeitgeist) do not need timers, many computations (like dip detection) use timers to wait for the low watermark to advance before outputting aggregates. For these computations, the low watermark's lag behind real time bounds the freshness of these aggregates. Since the low watermark propagates from injectors through the computation graph, we expect the lag of a computation's low watermark to be proportional to its maximum pipeline distance from an injector. We ran a simple three-stage MillWheel pipeline on 200 CPUs, and polled each computation's low watermark value once per second. In Figure 15, we can see that the first stage's watermark lagged real time by 1.8 seconds, however, for subsequent stages, the lag increased per stage by less than 200ms. Reducing watermark lag is an active area of development.

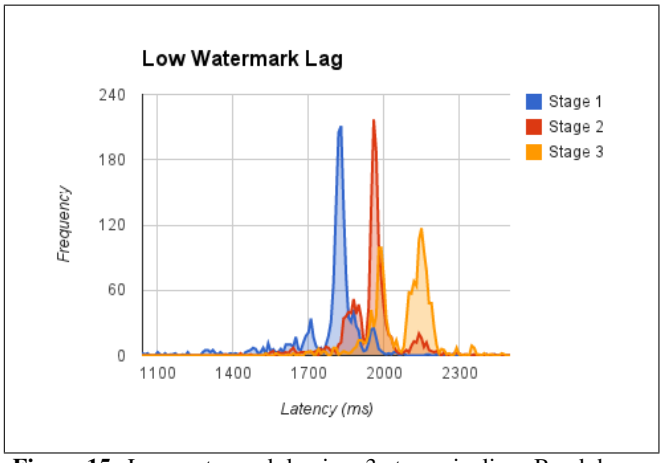

Figure 15: Low watermark lag in a 3-stage pipeline. Breakdown: {stage1: mean 1795, stdev 159. stage2: mean 1954, stdev 127. stage3: mean 2081, stdev 140}

8.3 Framework-Level Caching

Due to its high rate of checkpointing, MillWheel generates significant amounts of traffic to the storage layer. When using a storage system such as Bigtable, reads incur a higher cost than writes, and MillWheel alleviates this with a framework-level cache. A common use case for MillWheel is to buffer data in storage until the low watermark has passed a window boundary and then to fetch the data for aggregation. This usage pattern is hostile to the LRU caches commonly found in storage systems, as the most recently modified row is the one least likely to be fetched soon. MillWheel knows how this data is likely to be used and can provide a better cache-eviction policy. In Figure 16 we measure the combined CPU usage of the MillWheel workers and the storage layer, relative to maximum cache size (for corporate-secrecy reasons, CPU usage has been normalized). Increasing available cache linearly improves CPU usage (after 550MB most data is cached, so further increases were not helpful). In this experiment, MillWheel's cache was able to decrease CPU usage by a factor of two.

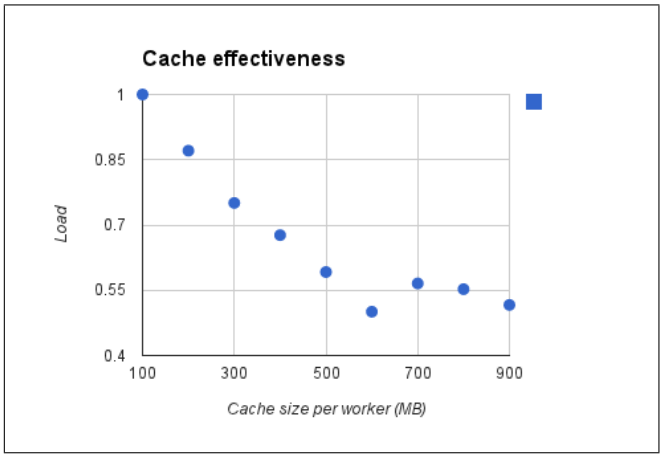

Figure 16: Aggregate CPU load of MillWheel and storage layer v.s. framework cache size.

8.4 Real-world Deployments

MillWheel powers a diverse set of internal Google systems. It performs streaming joins for a variety of Ads customers, many of whom require low latency updates to customer-visible dashboards. Billing pipelines depend on MillWheel's exactly-once guarantees. Beyond Zeitgeist, MillWheel powers a generalized anomaly-detection service that is used as a turnkey solution by many different teams. Other deployments include network switch and cluster health monitoring. MillWheel also powers user-facing tools like image panorama generation and image processing for Google Street View.

There are problems that MillWheel is poorly suited for. Monolithic operations that are inherently resistant to checkpointing are poor candidates for inclusion in computation code, since the system's stability depends on dynamic load balancing. If the load balancer encounters a hot spot that coincides with such an operation, it must choose to either interrupt the operation, forcing it to restart, or wait until it finishes. The former wastes resources, and the latter risks overloading a machine. As a distributed system, MillWheel does not perform well on problems that are not easily parallelized between different keys. If 90% of a pipeline's traffic is assigned a single key, then one machine must handle 90% of the overall system load for that stream, which is clearly inadvisable. Computation authors are advised to avoid keys that are high-traffic enough to bottleneck on a single machine (such as a customer's language or user-agent string), or build a two-phase aggregator.

If a computation is performing an aggregation based on low watermark timers, MillWheel's performance degrades if data delays hold back low watermarks for large amounts of time. This can result in hours of skew over buffered records in the system. Oftentimes memory usage is proportional to skew, because an application depends on low watermarks to flush this buffered data. To prevent memory usage from growing without bound, an effective remedy is to limit the total skew in the system, by waiting to inject newer records until the low watermarks have advanced.

9. RELATED WORK

Our motivation for building a general abstraction for streaming systems was heavily influenced by the success seen by MapReduce [11] in transforming the world of batch processing, as illustrated by the widespread adoption of Apache Hadoop [4]. Comparing MillWheel to existing models for streaming systems, such as Yahoo! S4 [26], Storm [23], and Sonora [32], we find that their models are insufficiently general for our desired class of problems. In particular, S4 and Sonora do not address exactly-once processing and fault-tolerant persistent state, while Storm has only recently added such support through Trident [22], which imposes strict ordering requirements on transaction IDs in order to function. Logothetis, et al, make similar arguments for the necessity of first-class user state [20]. Ciel [25] targets general data processing problems, while dynamically generating the dataflow graph. Like MapReduce Online [8], we see tremendous utility in making "early returns" available to the user. Google's Percolator [27] also targets incremental updates to large datasets, but expects latencies on the order of minutes.

In evaluating our abstraction, we note that we fulfill the requirements for a streaming system as enumerated by Stonebraker, et al [30]. Our flexibility toward out-of-order data is similar to the OOP approach [19], which makes compelling arguments for the necessity of a global low watermark calculation (rather than an operator-level one) and convincingly denies the viability of static slack values as a means of compensating for out-of-order data. While we appreciate the operator-specific unification of streaming and batch systems proposed by Spark Streaming [34], we believe that MillWheel addresses a more general set of problems, and that the microbatch model is not tenable without restricting users to pre-defined operators. Specifically, this model depends heavily on RDDs [33], which limit users to rollback-capable operators.

Checkpointing and recovery is a crucial aspect of any streaming system, and our approach echoes many that came before it. Our use of sender-side buffering resembles "upstream backup" in [14], which also defines recovery semantics (precise, rollback, and gap) that mirror our own flexible options for data delivery. While naïve upstream backup solutions can consume undue resources, as mentioned in Spark Streaming [34], our use of checkpoints and persistent state eliminates these disadvantages. Furthermore, our system is capable of significantly finer checkpoints than Spark Streaming [34], which proposes backups every minute, and depends on application idempotency and system slack for recovery. Similarly, S-Guard [17] utilizes checkpoints more infrequently than MillWheel (again, once per minute), though its operator partitioning scheme resembles our key-based sharding.

Our low watermark mechanism parallels the punctuation [31] or heartbeats [16] used in other streaming systems, like Gigascope [10]. However, we do not interleave heartbeats with standard tuples in our system, opting instead for a global aggregator, as in the OOP system [19]. Our concept of low watermarks is also similar to the low watermarks defined in OOP. We agree with their analysis that aggregating heartbeats at individual operators is inefficient and is better left to a global authority. This inefficiency is highlighted by Srivastava, et al, in [29], which discusses the maintenance of per-stream heartbeat arrays at every operator. Additionally, it establishes a similar distinction between the concepts of user-defined timestamps ("application time") and wall-clock time ("system time"), which we have found to be tremendously useful. We note the work of Lamport [18] and others [12] [15] in developing compelling time semantics for distributed systems.

Much of the inspiration for streaming systems can be traced back to the pioneering work done on streaming database systems, such as TelegraphCQ [6], Aurora [2], and STREAM [24]. We observe similarities between components of our implementation and their counterparts in streaming SQL, such as the use of partitioned operators in Flux [28] for load balancing. While we believe our low watermark semantic to be more robust than the slack semantic in [2], we see some similarities between our concept of percentile low watermarks and the QoS system in [1].

Acknowledgements

We take this opportunity to thank the many people who have worked on the MillWheel project over the years, including Atul Adya, Alexander Amato, Grzegorz Czajkowski, Oliver Dain, Anthony Feddersen, Joseph Hellerstein, Tim Hollingsworth, and Zhengbo Zhou. This paper is the beneficiary of many comments and suggestions from numerous Googlers, including Atul Adya, Matt Austern, Craig Chambers, Ken Goldman, Sunghwan Ihm, Xiaozhou Li, Tudor Marian, Daniel Myers, Michael Piatek, and Jerry Zhao.

10. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *In CIDR*, pages 277–289, 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[3] A. Adya, J. Dunagan, and A. Wolman. Centrifuge: Integrated lease management and partitioning for cloud services. In *NSDI*, pages 1–16. USENIX Association, 2010.

[4] Apache. Apache hadoop. `http://hadoop.apache.org`, 2012.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668. ACM, 2003.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26:4:1–4:26, June 2008.

[8] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. Technical report, University of California, Berkeley, 2009.

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally-distributed database. *To appear in Proceedings of OSDI*, page 1, 2012.

[10] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: High performance network monitoring with an sql interface. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 623–623. ACM, 2002.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, Jan. 2008.

[12] E. Deelman and B. K. Szymanski. Continuously monitored global virtual time. Technical report, in Intern. Conf. Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, 1996.

[13] Google. Protocol buffers. `http://code.google.com/p/protobuf/`, 2012.

[14] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 779–790. IEEE, 2005.

[15] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7:404–425, 1985.

[16] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. A heartbeat mechanism and its application in gigascope. In *Proceedings of the 31st international conference on Very large data bases*, pages 1079–1088. VLDB Endowment, 2005.

[17] Y. Kwon, M. Balazinska, and A. Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *Proceedings of the VLDB Endowment*, 1(1):574–585, 2008.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[19] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: a new architecture for high-performance stream systems. *Proceedings of the VLDB Endowment*, 1(1):274–288, 2008.

[20] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 51–62. ACM, 2010.

[21] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 555–566. IEEE, 2002.

[22] N. Marz. Trident. `https://github.com/nathanmarz/storm/wiki/Trident-tutorial`, 2012.

[23] N. Marz. Twitter storm. `https://github.com/nathanmarz/storm/wiki`, 2012.

[24] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. Technical Report 2002-41, Stanford InfoLab, 2002.

[25] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, page 9. USENIX Association, 2011.

[26] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170 –177, dec. 2010.

[27] D. Peng, F. Dabek, and G. Inc. Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[28] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 25–36. IEEE, 2003.

[29] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 263–274. ACM, 2004.

[30] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

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

[32] F. Yang, Z. Qian, X. Chen, I. Beschastnikh, L. Zhuang, L. Zhou, and J. Shen. Sonora: A platform for continuous mobile-cloud computing. Technical report, Technical Report. Microsoft Research Asia.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. 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*, 2011.

[34] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10. USENIX Association, 2012.