MillWheel:Fault-Tolerant Stream Processing at Internet Scale

By FAN Junbo

Introduction

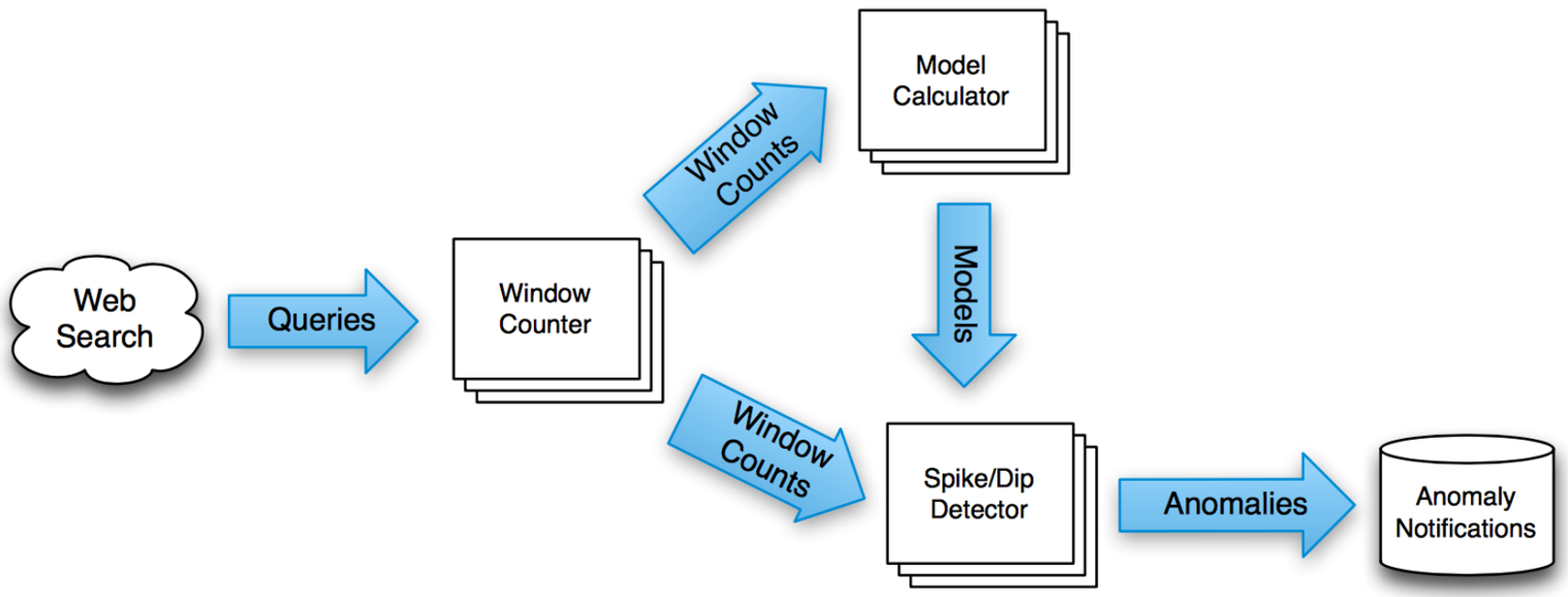
- MillWheel is a low latency data processing framework designed by Google at Internet scale.

Motived by Google Zeitgeist pipeline which is used to track trends of web queries.

0. Persistent Storage

1. Low Watermarks

2. Duplicate Prevention



System Overview

- MillWheel is a graph of user-defined transformations on input data that produces output data.

Computations: the transformations in the graph

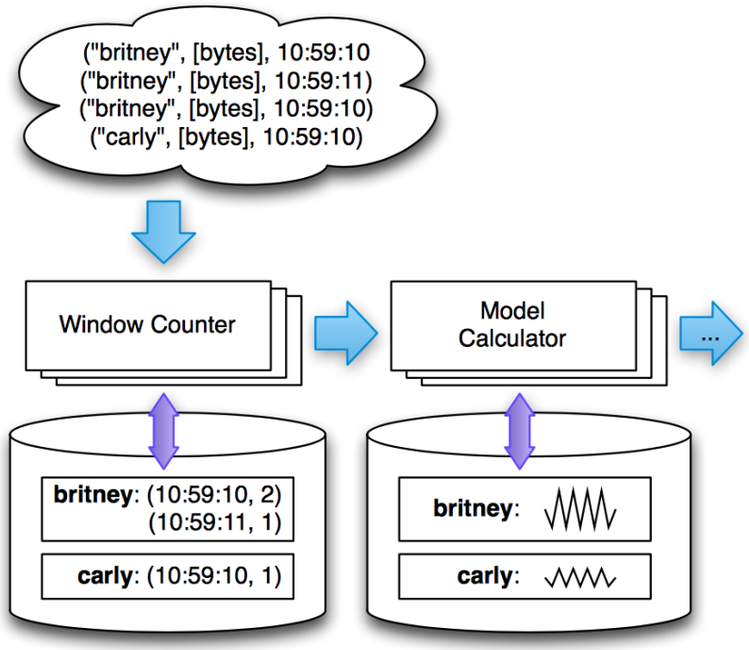
Triples: (key, value, timestamp)

For Zeitgeist:

key => query content

value => arbitrary string bytes

timestamp => the timestamp that the query occurred



Core Concepts

Computations: Application logic lives in computations, which encapsulate arbitrary user code. (Bolt in Storm)

Keys: Keys are the primary abstraction for aggregation and comparison between different records in MillWheel.

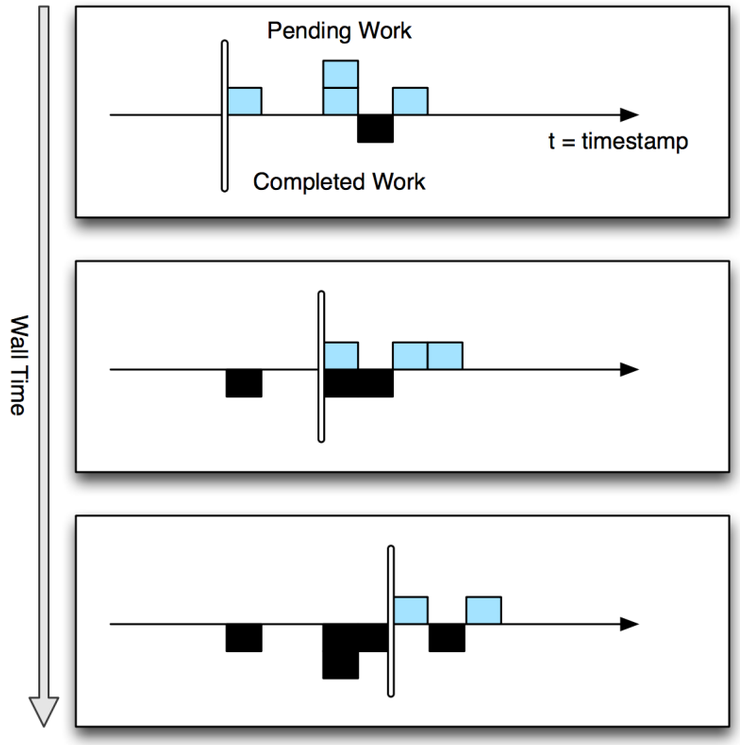
Streams: Streams are the delivery mechanism between different computations in MillWheel.

Persistent State: In its most basic form, persistent state in MillWheel is an opaque byte string that is managed on a per-key basis and is backed up by a highly available data storage such as BigTable.

Core Concepts

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

Low Watermark of A = min(oldest work of A, low watermark of C : C outputs to A)



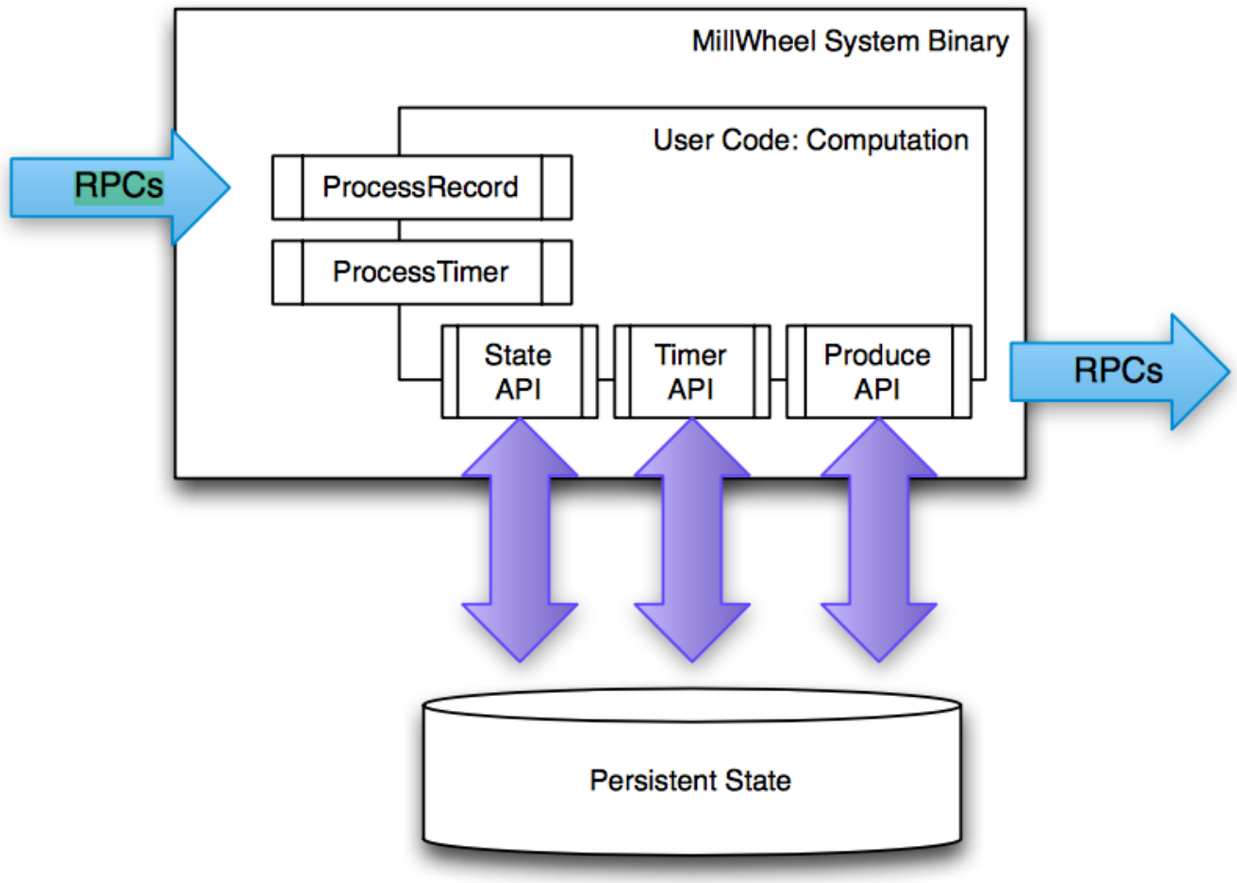
Timer: Timers are per-key programmatic hooks that trigger at a specific wall time or low watermark value. Timers are created and run in the context of a computation, and accordingly can run arbitrary code.

Core Concepts

APIs:

```
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();
};
```



Core Concepts

Injector:

Each computation calculates a low watermark value for all of its pending work

Injectors bring external data into MillWheel and seed low watermark values for the rest of the pipeline

```
// 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.

Zeitgeist Implementation

```
// 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");
  }
}
```

Fault Tolerance

Exactly-Once Guarantee

0. The record is checked against de-duplication data from previous deliveries; duplicates are discarded.

1. User code is run for the input record, possibly resulting in pending changes to timers, state, and productions.

2. Pending changes are committed to the backing store.

3. Senders are ACKed.

4.Pending downstream productions are sent.

(All above actions could be regarded as a **checkpoint**)

The system assigns unique IDs to all records at production time.

Fault Tolerance

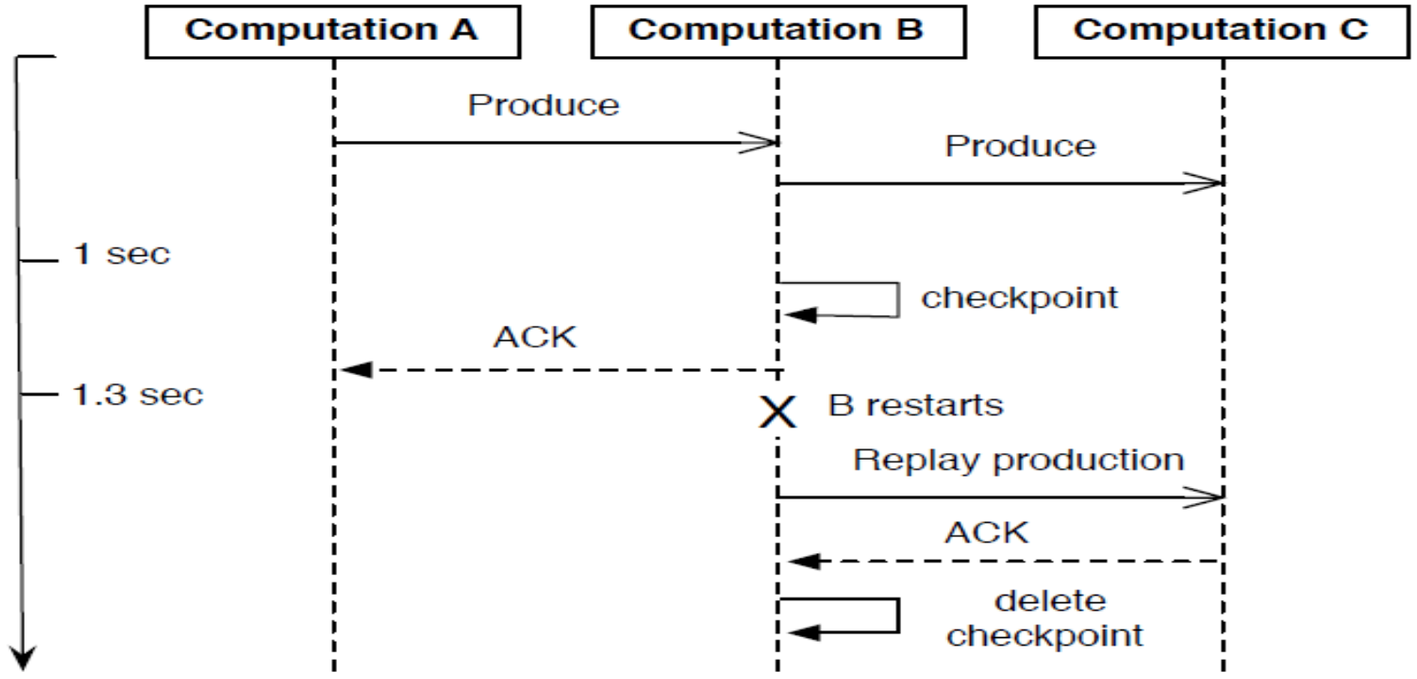
Strong Productions

Checkpoint produced records before delivery in the same atomic write as state modification

Weak Productions

Broadcast downstream deliveries optimistically, prior to persisting state

Each stage waits for the downstream ACKs of records



State Manipulation

Avoid Inconsistencies in Persistent States

Per-key updates are wrapped as single atomic operation

Avoid network remnant stale writes

Sequencer is attached to each write

Mediator of backing store checks before allowing writes

New worker invalidates any extant sequencer

Architecture

MillWheel deployments run as distributed systems on a dynamic set of host servers.

Each computation runs on one or more machines and streams are delivered through RPC.

Load distribution and balancing is handled by a replicated master.

Master divides each computation into a set of owned lexicographic key intervals(collectively covering all keys)

Intervals could be split, combine or merge due to CPU and RAM pressure

Each interval is assigned a unique sequencer.

Architecture

About Low Watermark

Low Watermark is implemented as a sub-system globally available and correct.

Low Watermark is implemented as a central authority tracking all low watermarks in the system.

Low Watermarks are store to persistent state preventing unexpected shutdown

Evaluation

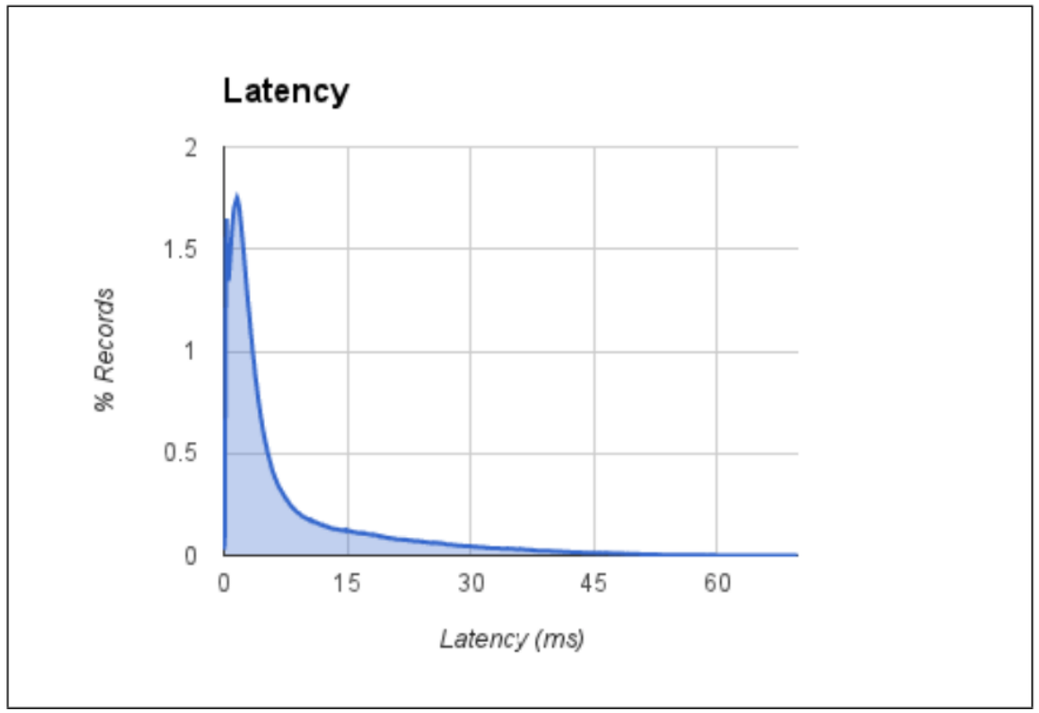


Figure 13: A histogram of single-stage record latencies between two differently-keyed stages.

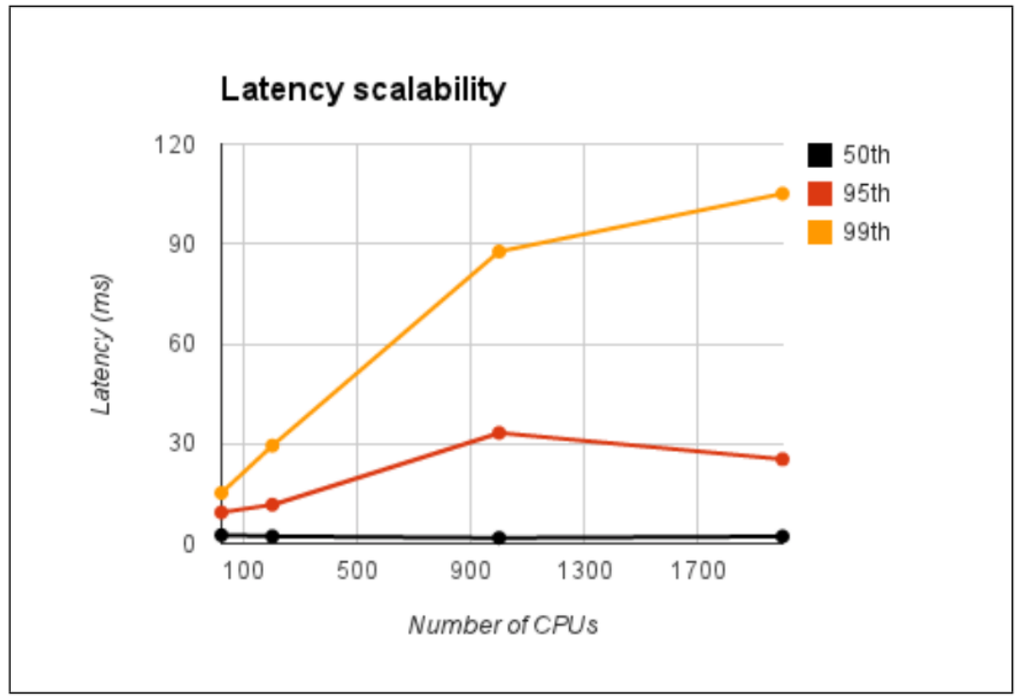


Figure 14: MillWheel's average latency does not noticeably increase as the system's resource footprint scales.

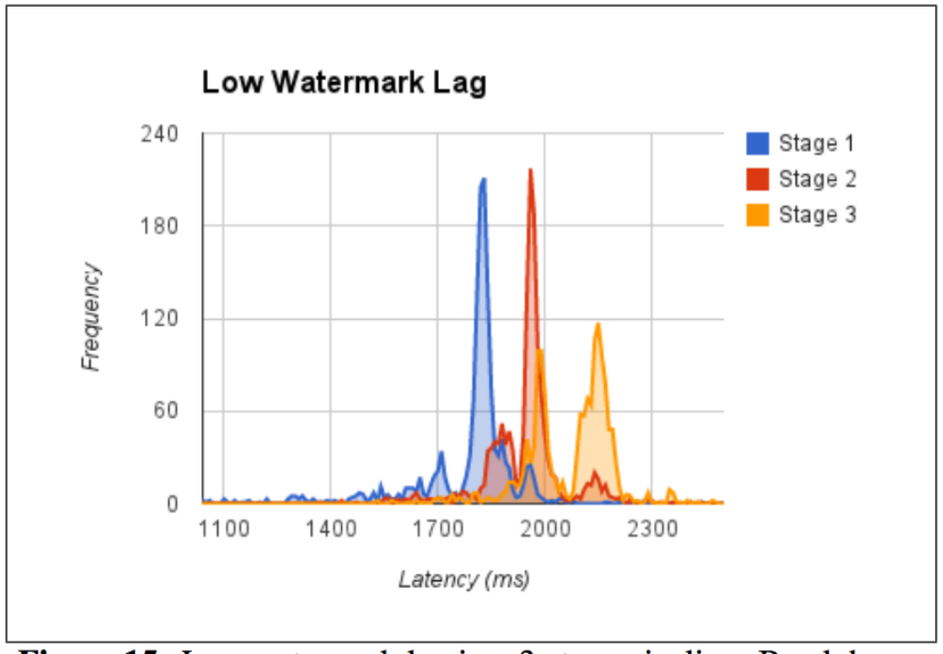


Figure 15: Low watermark lag in a 3-stage pipeline. Breakdown: {stage1: mean 1795, stdev 159. stage2: mean 1954, stdev 127. stage3: mean 2081, stdev 140}

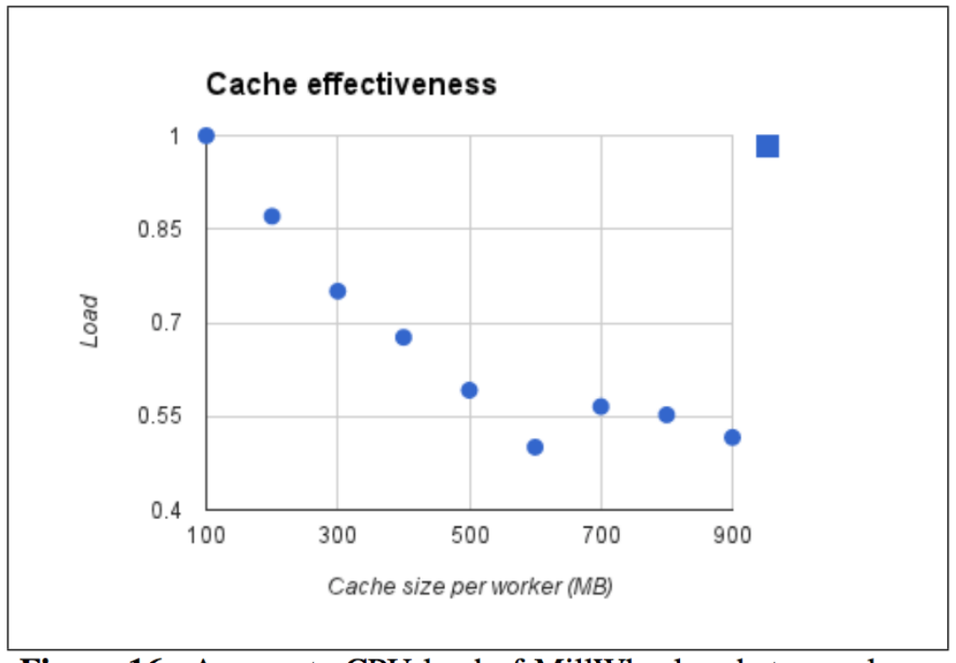


Figure 16: Aggregate CPU load of MillWheel and storage layer v.s. framework cache size.