MillWheel: Fault-Tolerant Stream Processing at Internet Scale

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

ECE1724

Authors:

Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, Sam Whittle (Google)

Many slides adapted from Amir H. Payberah

Motivation

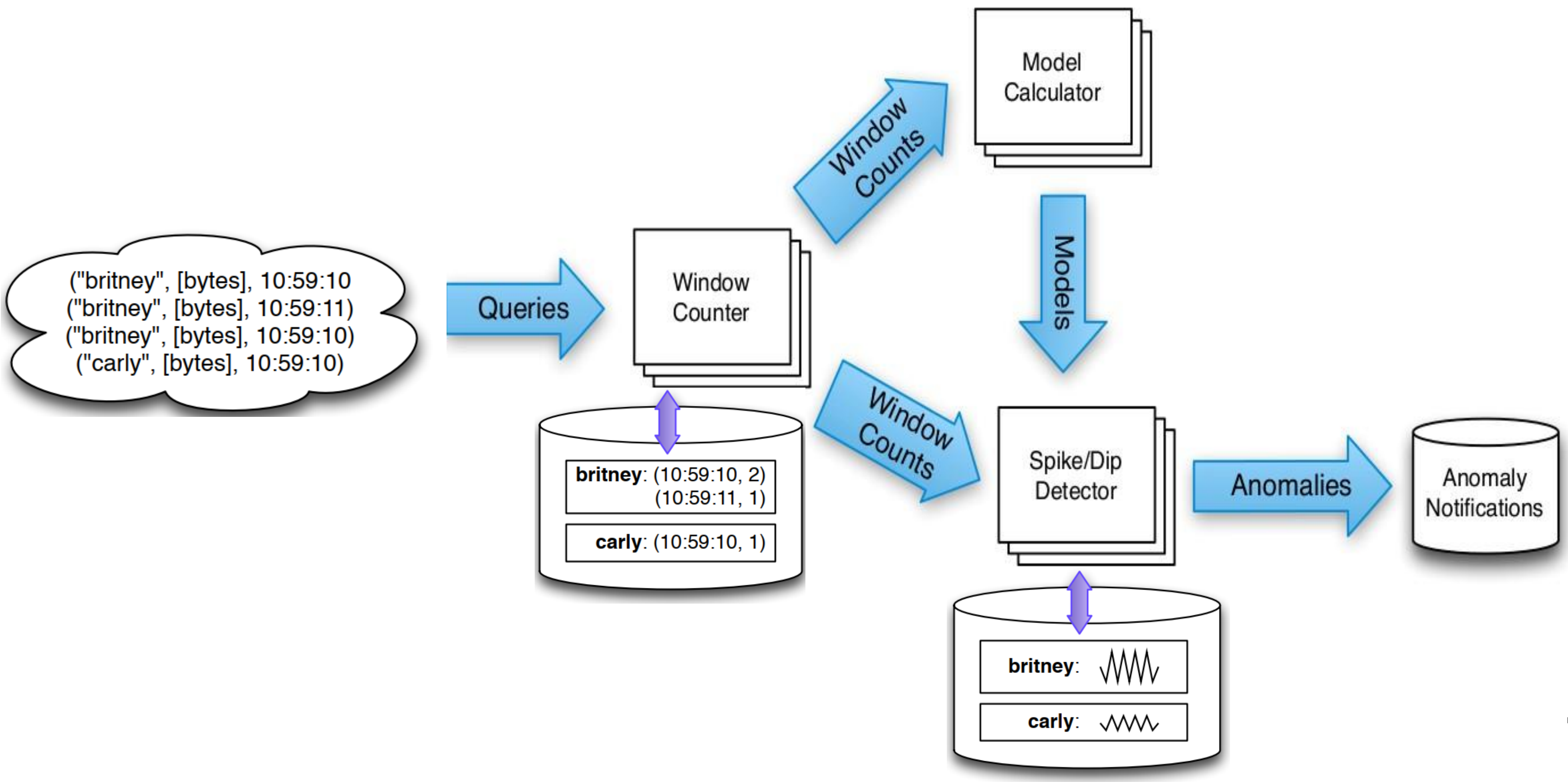
- Google's Zeitgeist pipeline tracks trends in web queries
- Builds a historical model of each query
- Shows queries that are spiking in real time

Millwheel DataFlow

- A graph of user-defined computations connected by streams
- Computations perform application logic
- Stream is a sequence of (key, value, timestamp) records
  - Timestamps are user defined but typically close to wall clock time when the event occurred (event-time)
- A computation subscribes to zero or more input streams and publishes one or more output streams
  - Keys of these streams may be same or different
- Computations can be added or removed from the graph dynamically

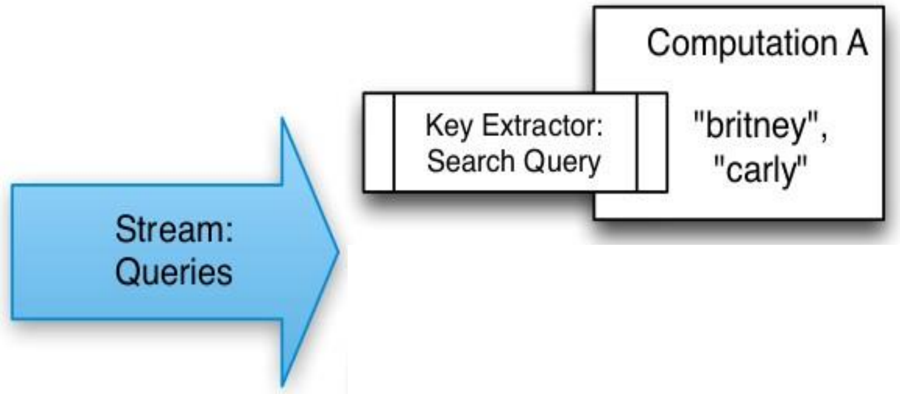
Zeitgeist

- Input is continuously arriving search queries
- Output is the set of queries that are spiking or dipping



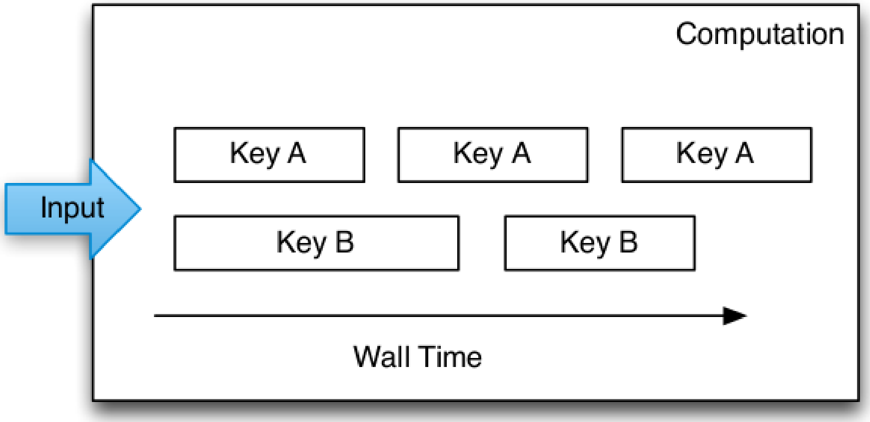
Key Extraction Function

- A computation specifies a key extraction function for each input stream to assign keys to input records
- Multiple computations can extract different keys from the same stream



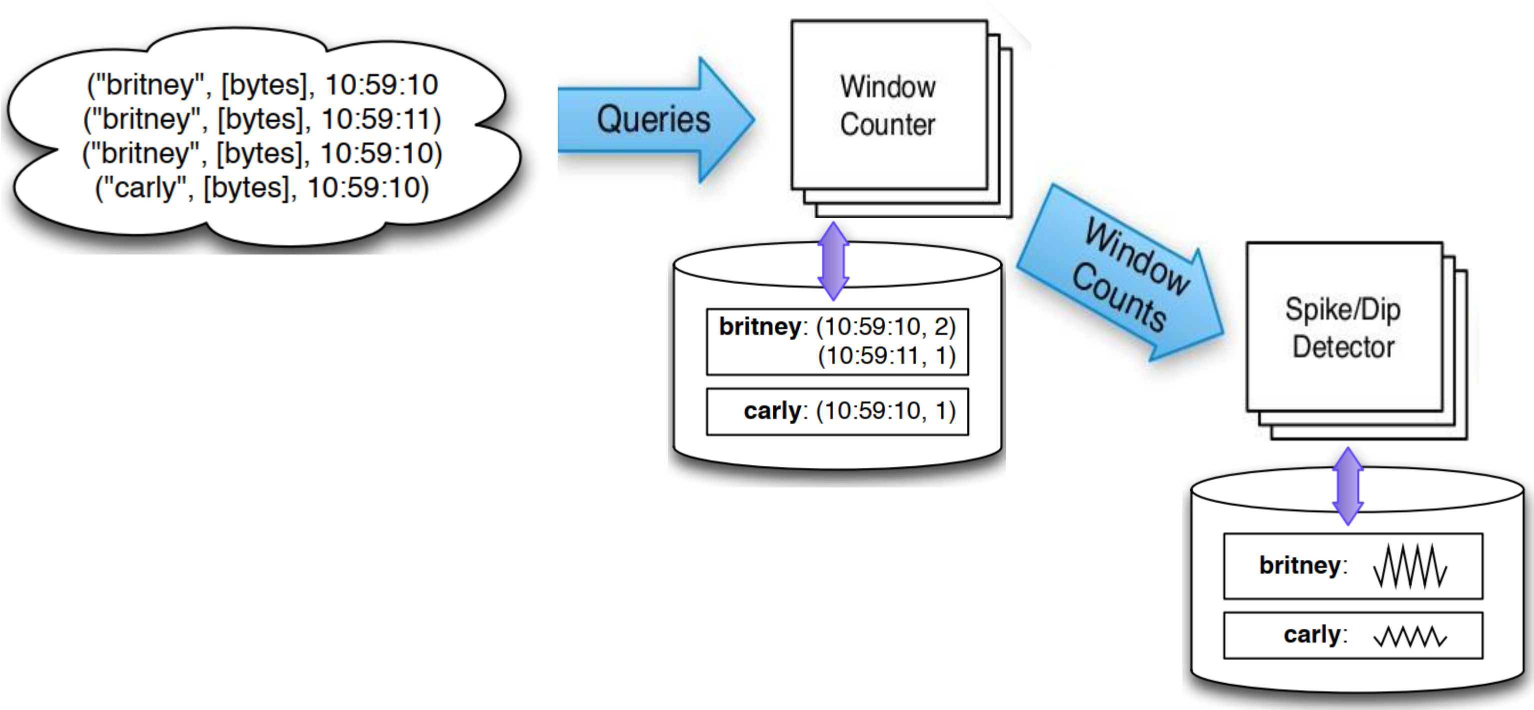
Computation

- Each computation
  - Runs in the context of a single (extracted) key
  - Can only access state that is associated with the key
- All processing over the same key is serialized
- Processing over different keys can run in parallel
  - MillWheel distributes key ranges to different workers

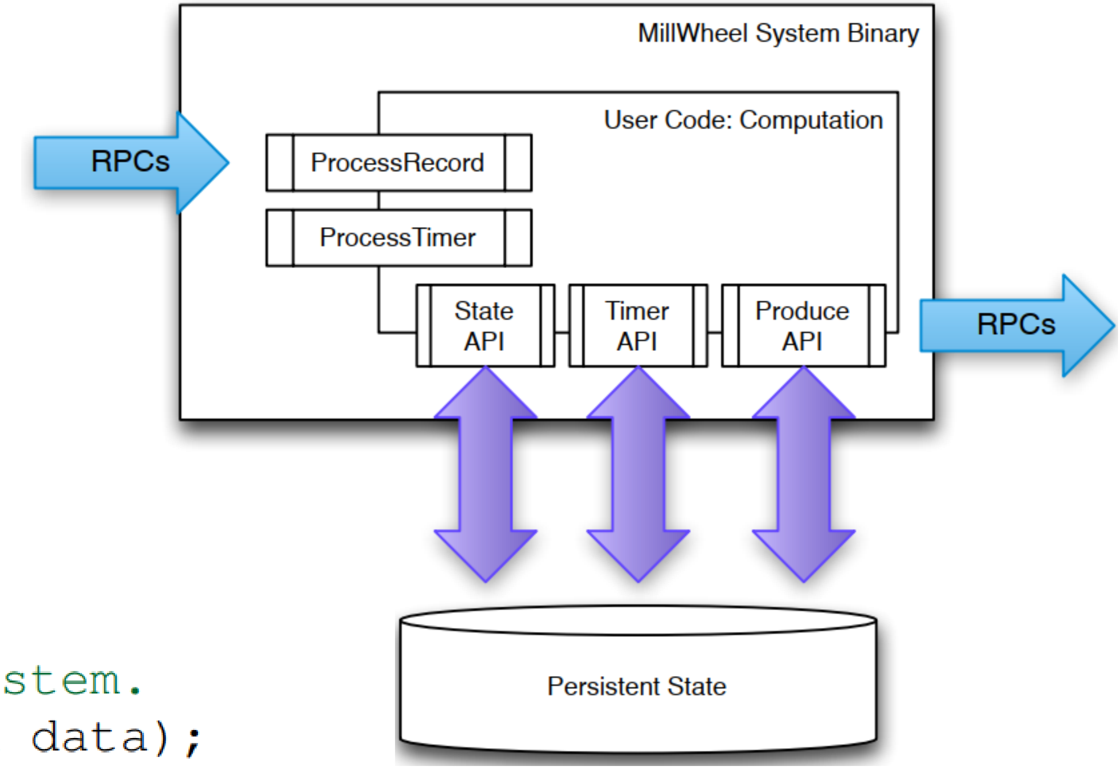


Persistent State

- For each computation, per-key state is stored in a row in Bigtable or Spanner
  - Allows atomic state updates
- Common use: per-key aggregation, joins, ...



Computation API



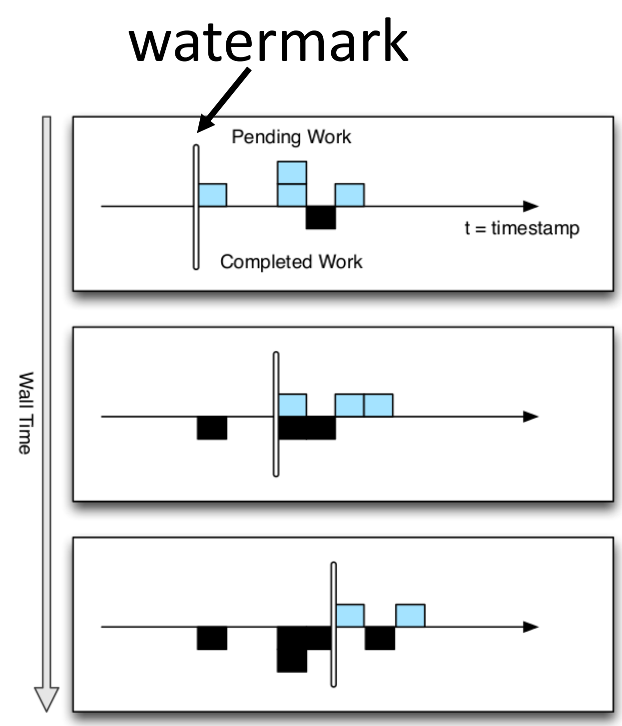

```
class Computation {
  // Hooks called by the system.
  void ProcessRecord(Record data);
  void ProcessTimer(Timer timer);

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

Low Watermarks

- Recall records has timestamps
- In practice, out-of-order records are the norm
  - Need to distinguish between events that were not generated versus events that are delayed in some time interval
- Millwheel provides each computation a low watermark
  - Sets watermark so most input records have larger timestamps
    - Also, guarantees each computation's watermark is monotonic
  - Computations can use low watermark to perform operations
    - E.g., output window counts when low watermark crosses window boundary, which ensures that all data within window is processed

Low Watermarks

- Low watermarks are "seeded" by injectors
- New records appear as pending work in the system
- A computation may perform pending work out-of-order
- As work completes in the system, low watermark is increased
  - At each node, minimum of:
    1. pending work at the node
    2. watermarks of upstream nodes



Read and update per-key state:
[(window1, count1),
 (window2, count2), ...]

Set timer to fire when low watermark crosses window_boundary

When timer for window1 fires, returns count1

Produce (count1, window_boundary) for DipDetector

```
// 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

- MillWheel ensures that computations are processed exactly-once
  - Greatly simplifies programming model because user code can be non-idempotent (system ensures it behaves idempotently)
  - A requirement for MillWheel's revenue-processing customers
- MillWheel guarantees that each computation
  - Performs per-key update atomically
  - Delivers records exactly once
- Together, guarantees same behavior as failure-free operation

Exactly-Once Record Processing

- Exactly-once record processing:
  1. Check duplicate incoming record ID
  2. Perform computation
  3. Atomically checkpoint
     1. Incoming record ID
     2. Updated per-key state
     3. All outgoing records
  4. ACK incoming record to upstream node
  5. Send outgoing records to downstream node(s)
- On failure:
  1. Restore consistent state of the computation from checkpoint
  2. Replay outgoing messages (downstream will filter duplicates)

At-Least-Once Record Processing

- Exactly-once record processing is expensive
  - Requires 1) duplicate detection, 2) checkpointing before sending outgoing records
  - If user code is idempotent, both can be avoided

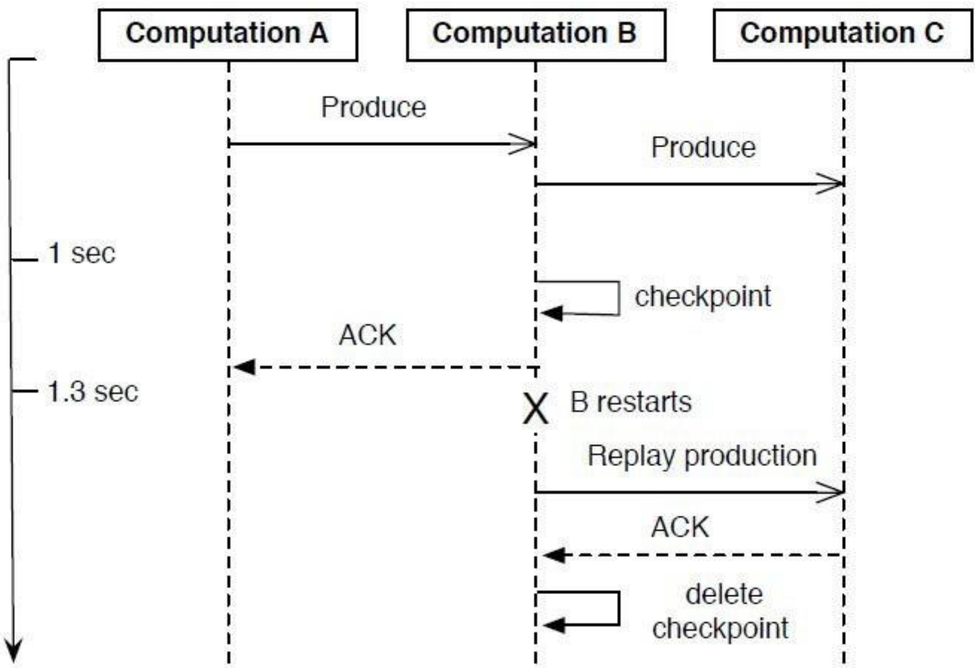
- At-least-once record processing:
  1. Perform computation
  2. Send outgoing records to downstream node(s)
  3. Atomically write per-key state
  4. Wait for ACKs of outgoing messages
  5. ACK incoming messages

At-Least-Once Record Processing

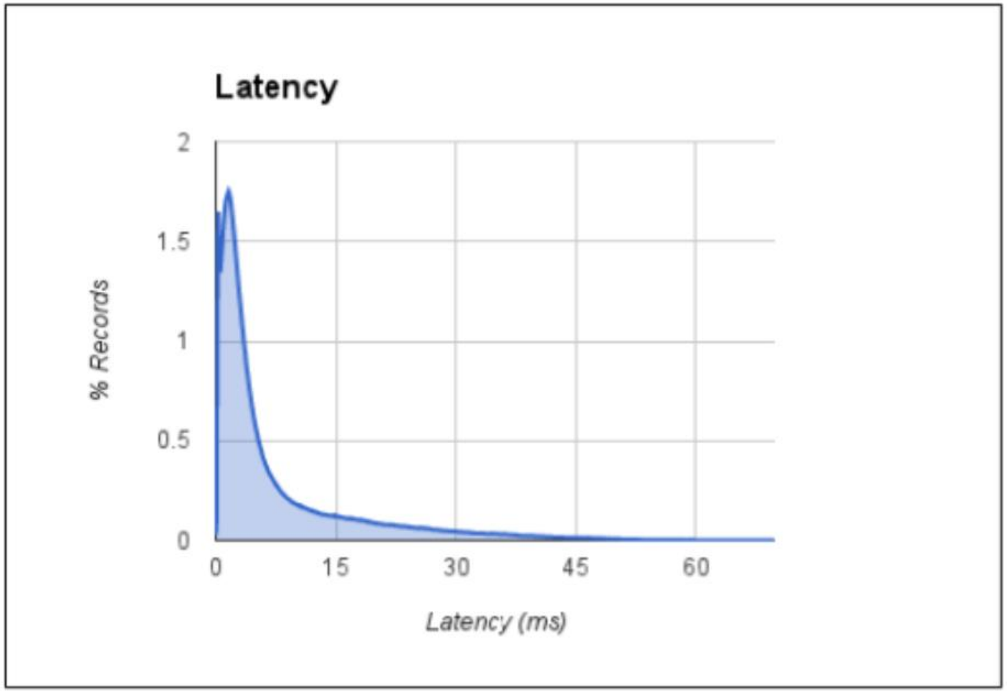
- However, with at-least-once processing, every node waits for the completion of all downstream nodes
  - Increases end-to-end latency since failures may require resending more data
  - Increases resource requirements (e.g., state in memory)

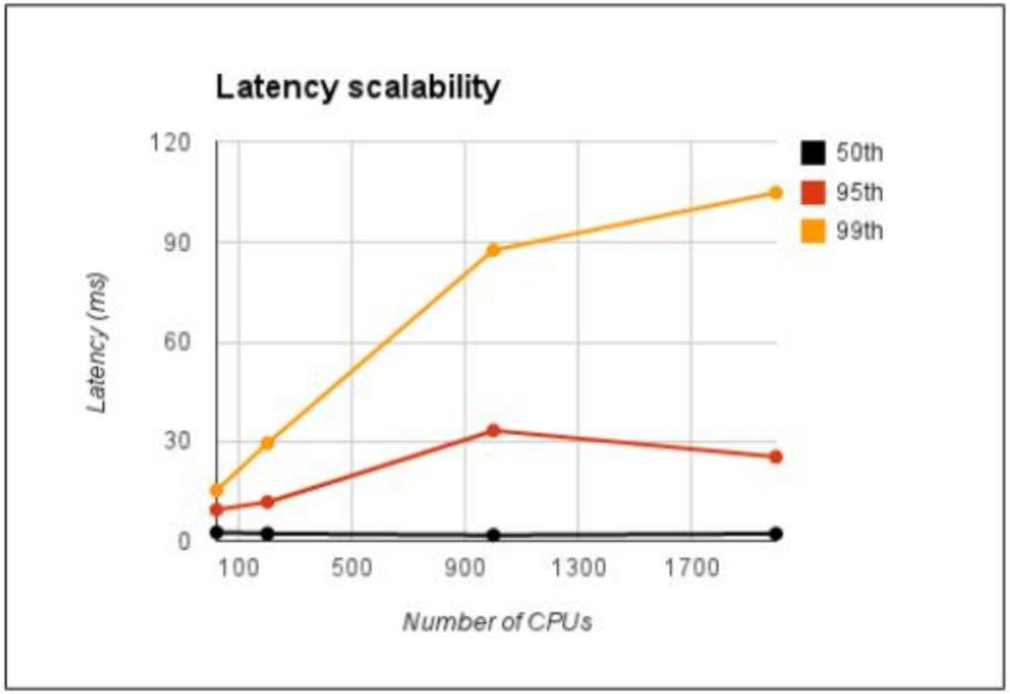
At-Least-Once Record Processing

- Solution: checkpoint outgoing messages at selected computations
  - Computation A can free resources after receiving ACK



Evaluation





Latency (ms)	Median	95^{th} percentile
Weak productions	3.6	30
Strong productions	33.7	93.8

Conclusions

- MillWheel provides a dataflow-based programming model for stream processing
- Each computation runs in the context of a single key
  - Enables low-latency processing
  - Enables parallelizing operations across keys
- Low watermarks enable processing events out-of-order
- Uses fine-grained (per-key) checkpoints to provide exactly-once delivery semantics
  - Simplifies programming model

Discussion

Q1

- Say upstream node U has a low watermark Wu and downstream node D has a low watermark Wd. Does Millwheel ensure any relation between Wu and Wd?

Q2

- What can Millwheel do about records that arrive behind the low watermark?

Q3

- How does this ordering ensure exactly-once semantics?

1. Check duplicate incoming record ID
2. Perform computation
3. Atomically checkpoint
    1. Incoming record ID
    2. Updated per-key state
    3. All outgoing records
4. Ack incoming record to upstream node
5. Send outgoing records to downstream node(s)

Q4

- What is the purpose of Step 4 below?

1. Check duplicate incoming record ID
2. Perform computation
3. Atomically checkpoint
    1. Incoming record ID
    2. Updated per-key state
    3. All outgoing records
4. Ack incoming record to upstream node
5. Send outgoing records to downstream node(s)

Q5

- The incoming record ID and outgoing records in the checkpoint need to be garbage collected. When can that be done?

1. Check duplicate incoming record ID
2. Perform computation
3. Atomically checkpoint
    1. Incoming record ID
    2. Updated per-key state
    3. All outgoing records
4. Ack incoming record to upstream node
5. Send outgoing records to downstream node(s)