MillWheel: Fault-Tolerant Stream Processing at Internet Scale

Presented by Rui Zhang

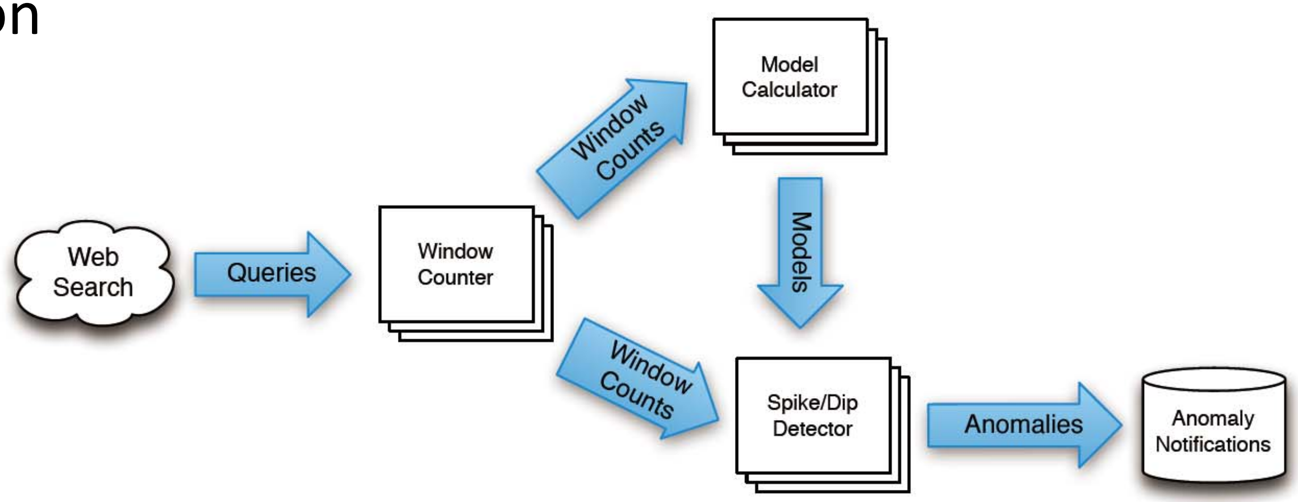
October 28, 2013

What is MillWheel?

- Stream processing framework
- Simple programming models
- User-specified directed computation graph
- Fault-tolerance guarantees
- Scalability

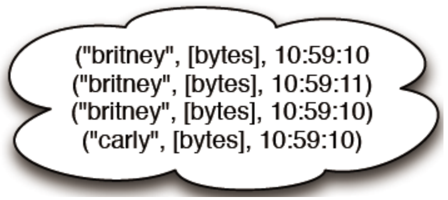
Requirements by example

- Persistent Storage
  - Short-term and long-term
- Low Watermarks
  - Distinguish late records
- Duplicate Prevention



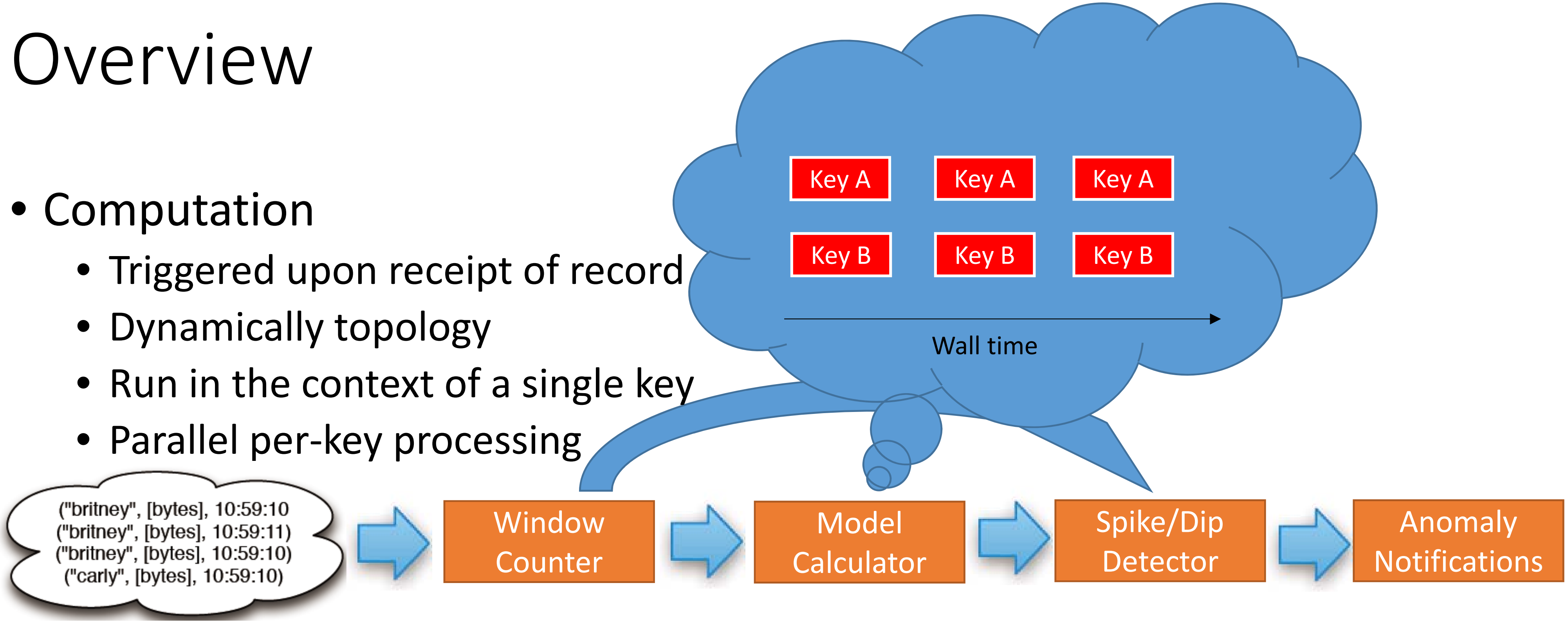
Overview

- Input and output triple
    - (key, value, timestamp)



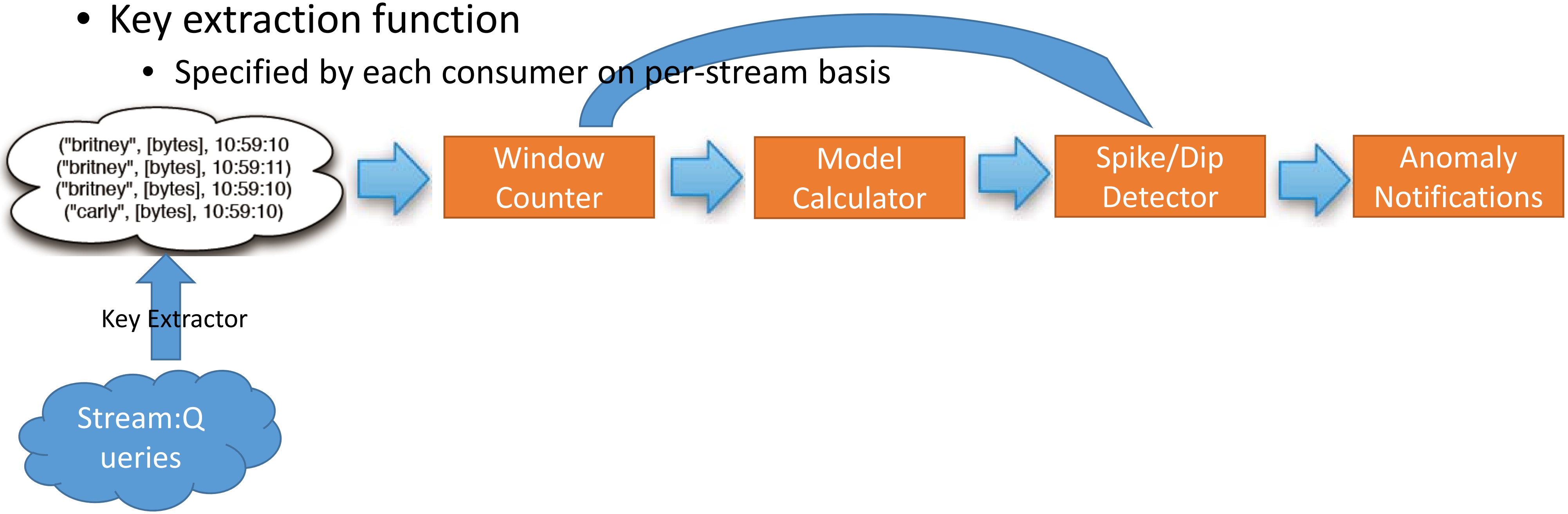
Overview

- Computation
  - Triggered upon receipt of record
  - Dynamically topology
  - Run in the context of a single key
  - Parallel per-key processing



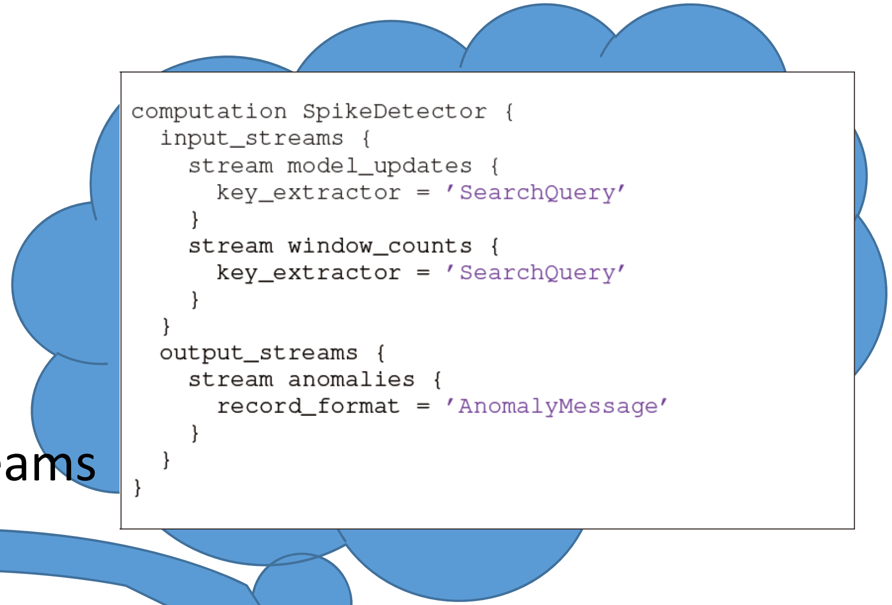
Overview

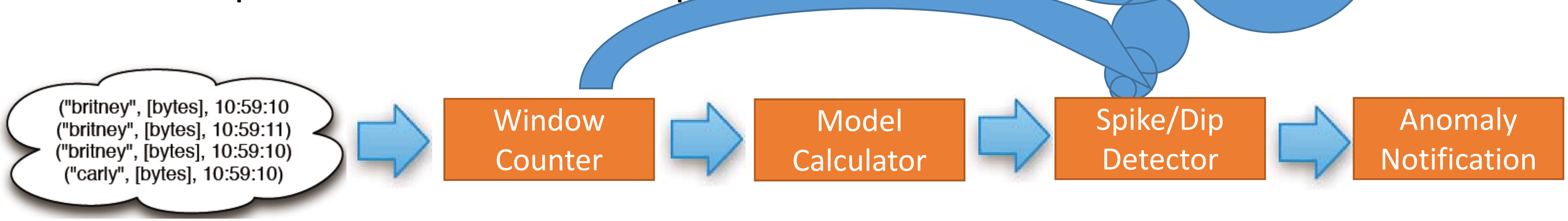
- Keys
  - Abstraction for record aggregation and comparison
  - Computation can only access state for the specific key
  - Key extraction function
    - Specified by each consumer on per-stream basis



Overview

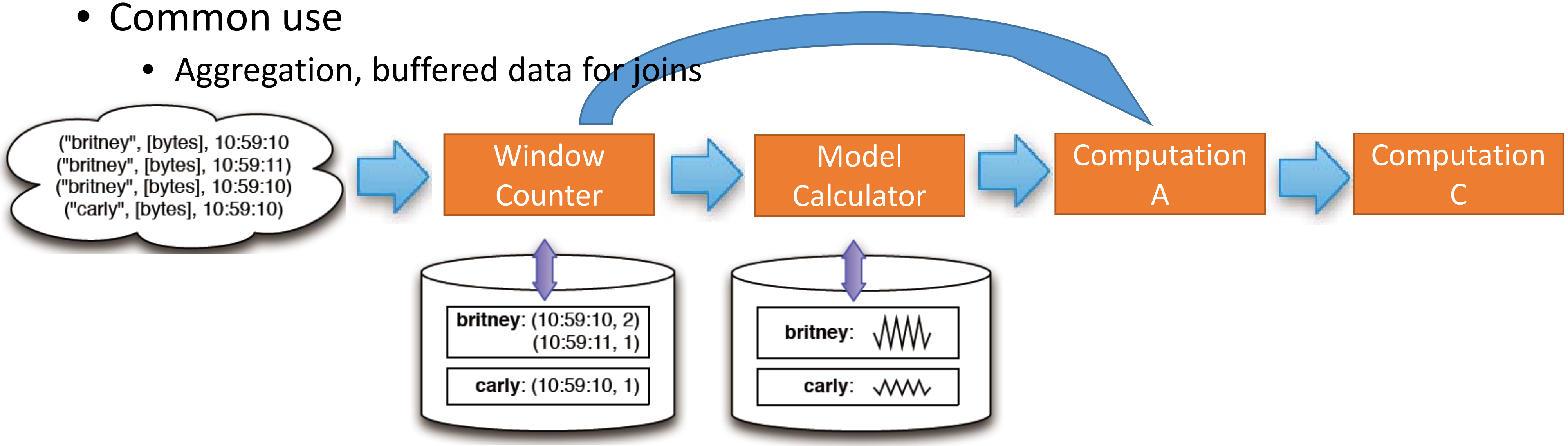
- Streams
  - Delivery mechanism between computations
  - Computation can get input from multiple streams and also produce records to multiple streams





Overview

- Persistent State
  - Managed on per-key basis
  - Stored in Bigtable or Spanner
  - Common use
    - Aggregation, buffered data for joins



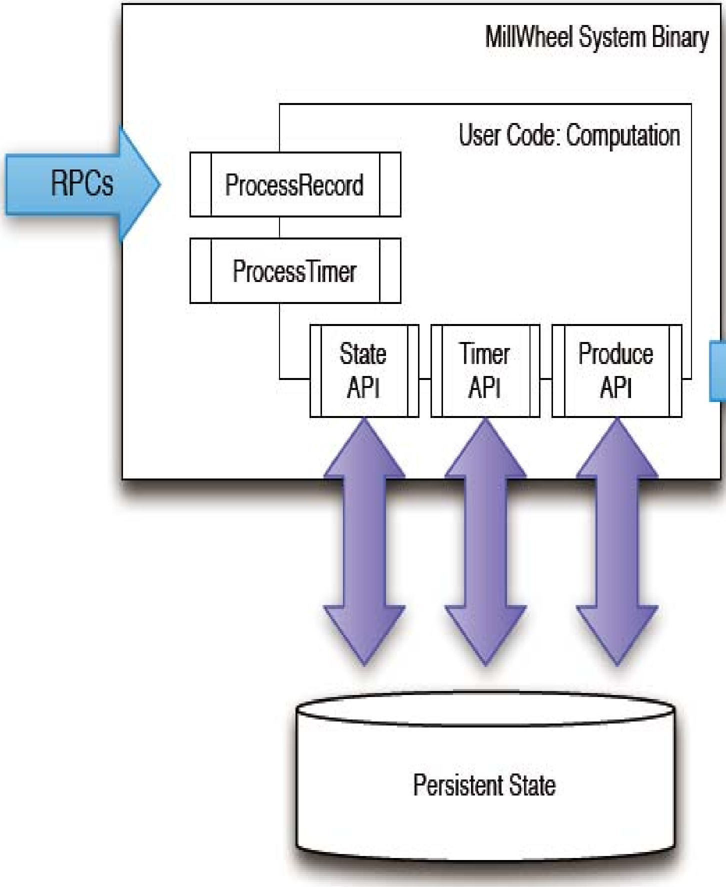
API

- Computation API
  - ProcessRecord
    - Triggered when receiving a record
  - ProcessTimer
    - Triggered at a specific value or low watermark value
    - Timers are stored in persistent state
    - Not necessary

```
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();
};
```

API



Fetch and manipulate state

Set Timer

Produce Record

```
pon receipt of a record, update the running
otal for its timestamp bucket, and set a
imer 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");
  }
}
```

API

- Low Watermark
  - At the system layer
  - Compute the low watermark value for all the pending work
  - Computation code rarely communicate with low watermarks

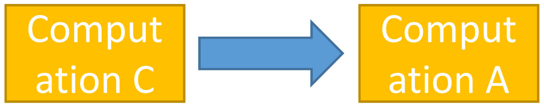
API

- Injectors
  - Bring external data into MillWheel
  - Publish the injector low watermark
  - Distributed across many processes
    - Injector low watermark is determined among those processes

```
// 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);
}
```

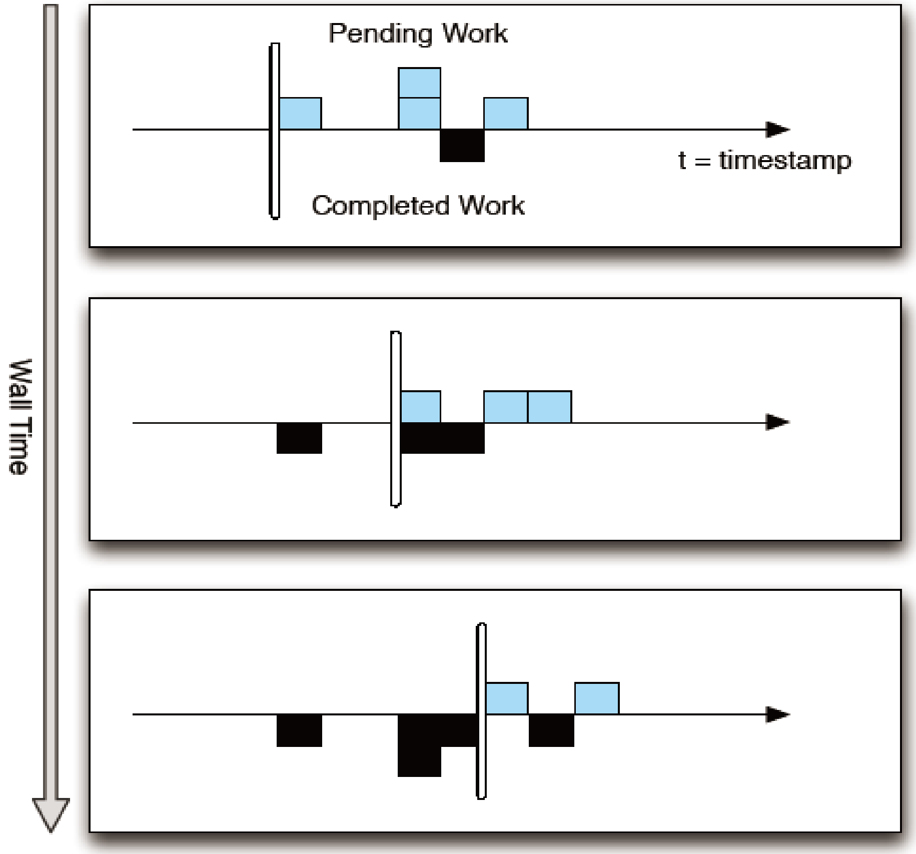
Key Features

- Low Watermark
  - Min(oldest work of A, low watermark of C)
  - Late records
    - Records behind the low watermark
    - Process them according to application (discard or correct the result)
  - Monotonic in the face of late data



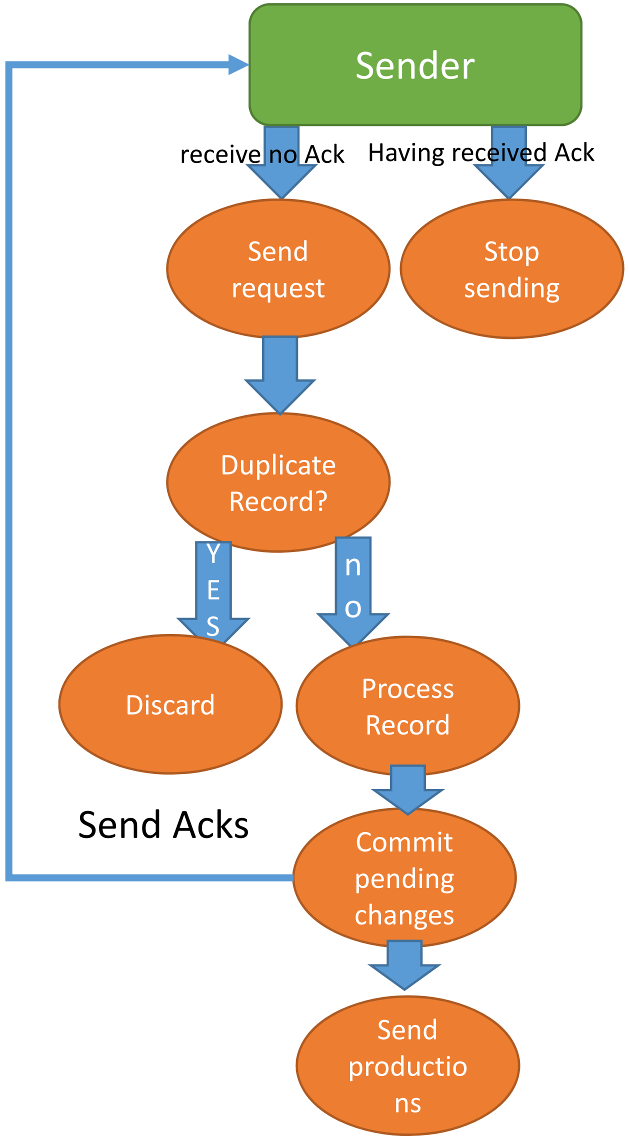
Key Features

- Low Watermark



Key Features

- Delivery Guarantees
  - Exactly-Once Delivery
    - Unique ID for every record
    - Bloom filter to provide fast path
    - Garbage collection for record IDs
      - Delay for those frequently delivering late data
    - Duplicate checking can be disabled



Key Features

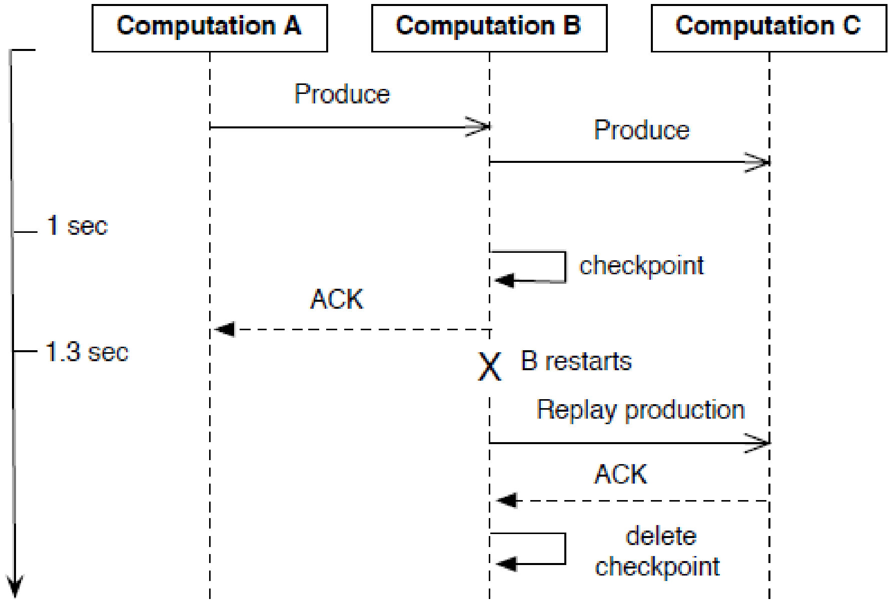
- Delivery Guarantees
  - Strong Productions
    - Checkpoint before delivering productions
    - Checkpoint data will be deleted once productions succeed

Key Features

- Delivery Guarantees
  - Weak Productions
    - For computations inherently idempotent
    - Broadcast downstream without checkpointing
    - End-to-end latency
    - Partial checkpointing

Key Features

- Delivery Guarantees
  - Weak Productions

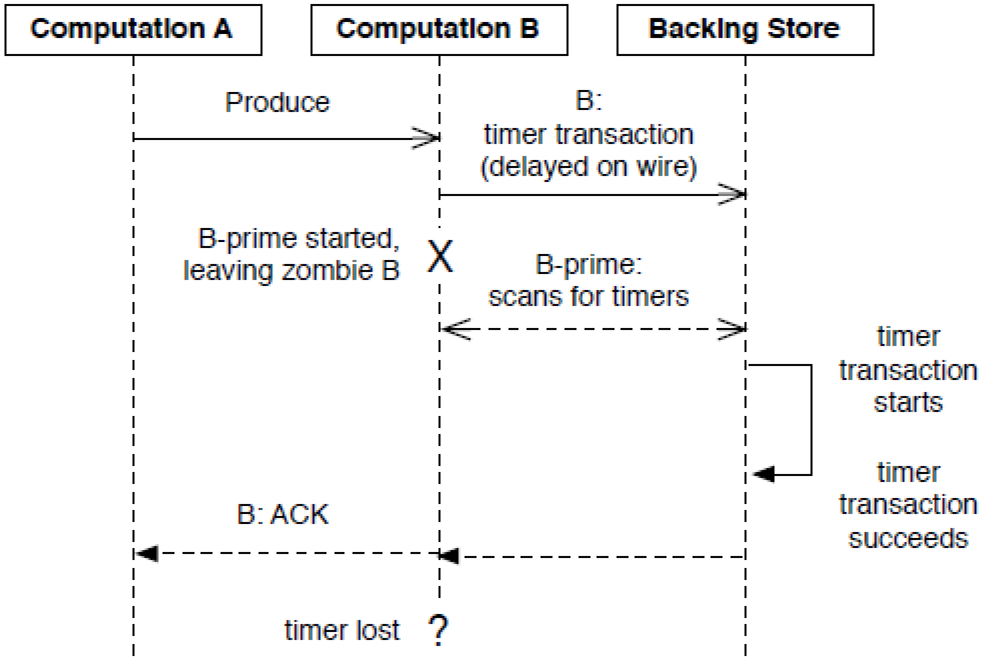


Key Features

- State Manipulation
  - Wrap all per-key updates into an atomic operation in case of crash
    - Per-key consistency
    - timer, user state, production checkpoints
  - Single-writer guarantee
    - Avoid zombie writers and network remnants issuing stale writes
    - Sequencer token
      - Check the validity before committing writes
    - Critical for both hard state and soft state

Key Features

- State Manipulation

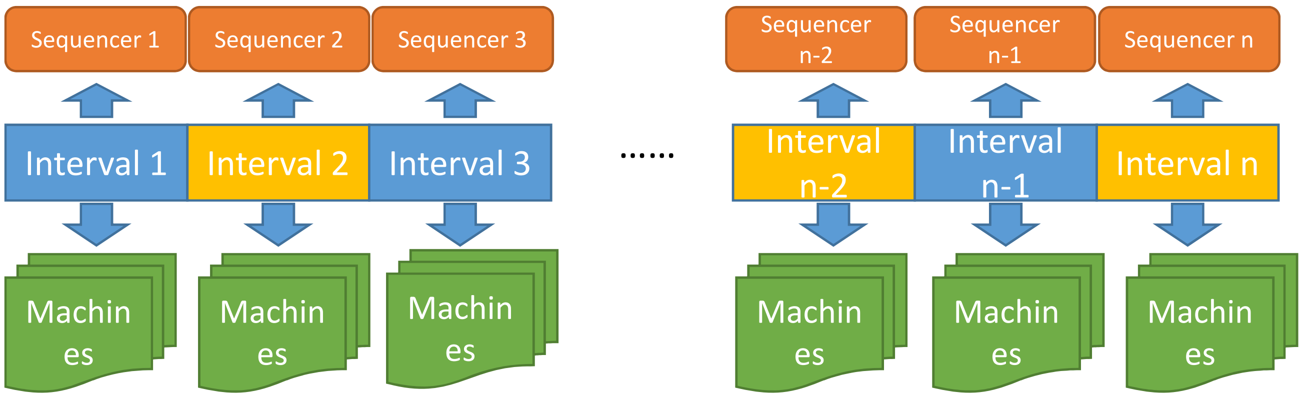


Implementation

- Architecture
  - Each computation runs on one or more machines
  - Streams are delivered through RPC
  - On each machine:
    - Marshals incoming work
    - Manages process-level metadata
    - Delegates to corresponding computation

Implementation

- Architecture
  - Load distribution and balancing
    - Handled by replicated master
    - Key intervals
      - Keep changing according to CPU load and memory pressure

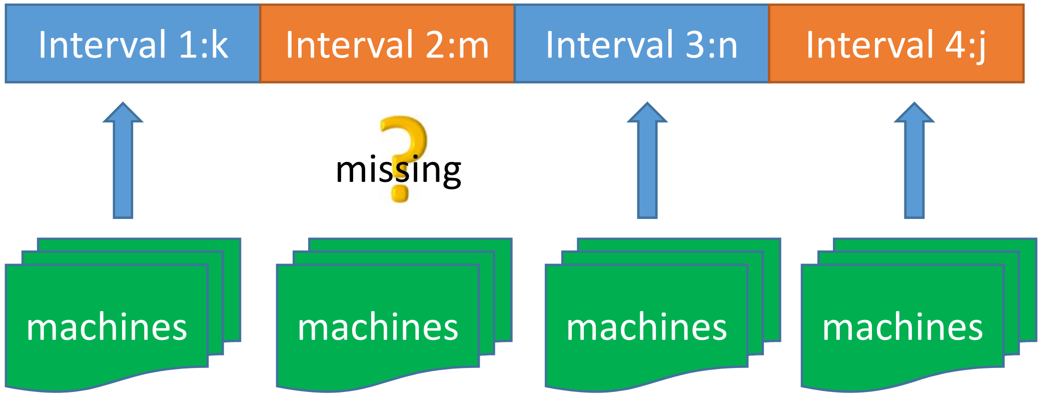


Implementation

- Architecture
    - Persistent state
        - Bigtable or Spanner
        - Data for a particular key are stored in the same row
            - Timers, pending productions, persistent state
        - Recover from failure efficiently by scanning metadata
            - Consistency is important

Implementation

- Low Watermark
  - Central authority
    - Track all low watermark values across the system
    - Store them in persistent state in case of failure
    - Each process aggregates their own timestamp information and send to central authority
      - Bucketed into key intervals



Implementation

- Low Watermark
  - Central authority
    - Minima are computed by workers
    - Sequencer for low watermark updates
    - Scalability
      - Sharded across multiple machines

Evaluation

- Output latency
    - Idempotent guarantee can increase latency a lot
- Watermark lag
    - Proportional to the pipeline distance from the injector
- Framework-level caching
    - Increasing available cache improves the CPU usage linearly

Comparison

- Punctuation-based system
  - Use special annotations embedded in data streams to specify the end of a subset of data
  - Indicate no more records will come which match the punctuation
- Gigascope
  - Heartbeat based system
  - Heartbeats carry temporal update tuples
  - Heartbeats monitor the system performance and check the node failure
- Drawbacks of these systems
  - Need to generate artificial messages even though there are no new records
  - Utilize a more aggressive checkpointing protocol where they track every record processed