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
  • Timestamp 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

- Key extraction function: specified by the stream consumer to assign keys to 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 computations over the same key are serialized
- Computations over different keys run in parallel
  - MillWheel distributes key ranges to different workers
Persistent State

- **Per-key** state is stored in a row in Bigtable or Spanner
- 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 stream data has timestamps

• In practice, out-of-order streams 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
  • Assumes that all data before low watermark has been received
  • Guarantees that a computation’s low 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 have timestamps > low watermark
• They 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

Returns 
(window1, count1)

Produce 
(window1, count1, boundary_time) 
for DipDetector
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

• **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)
Atleast-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

• 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
Atleast-Once Record Processing

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

- Solution: checkpoint outgoing messages at selected computations
- A can delete its checkpoint after receiving ACK
Evaluation

<table>
<thead>
<tr>
<th>Latency (ms)</th>
<th>Median</th>
<th>95th percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak productions</td>
<td>3.6</td>
<td>30</td>
</tr>
<tr>
<td>Strong productions</td>
<td>33.7</td>
<td>93.8</td>
</tr>
</tbody>
</table>
Conclusions

• MillWheel provides a dataflow-based programming model for stream processing

• Each computation runs in the context of a single key and persists per-key state

• Low watermarks enable handling out-of-order events

• Exactly-one delivery simplifies the programming model
Discussion
Q1

- What can Millwheel do about records that arrive behind the low watermark?
Q2

• 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)
Q3

• How does the 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

- Would intra-key parallelism be beneficial? How could it be implemented?