Tony Nguyen

Dr. Shawn Bowers

CPSC 324 01

6 April 2024

Reading 4

Part 1: Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., Whittle, S. (2013) *MillWheel: Fault-Tolerant Stream Processing at Internet Scale*. Proc. of VLDB, vol. 6, no. 11, pp. 1033–1044.

Reading Notes

1. Introduction
    - Stream processing provides content and allows organizations to make faster and better decisions, particularly in terms of their ability to provide low-latency results.
    - In short, people want real-time data.
    - Requires fault tolerance, persistent state, and scalability.
    - MillWhell is a programming model specifically tailored 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.
    - It provides fault tolerance at the framework level, allowing any node or edge topology to fail at any time without affecting the correctness of results.
    - MillWheel checkpoints its progress at fine granularity, alleviating buffers.
    - MillWheel vs other frameworks

- Allows complex streaming systems to be created without the experience of distributed systems.
- Proven to be scalable and fault-tolerant

2. Motivation and Requirements

- Persistent storage: The experiment requires both long-term and short-term storage. Data involved can just be instant (a few seconds) or can continuously last for months.
- Low watermarks: be able to detect whether a query is delayed or if it is not there.
- Duplicate prevention: MillWheel needs each instance to be processed one only to not cause duplication.
- Requirements
  - 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.

3. System Overview

- MillWheel is a graph of user-defined transformations (or computations) on input data that produces output data.
  - The computations can be parallelized across many machines, so we don't have to care about load balancing or fine grain level.
  - Input: continuously arriving set of search queries

    - Output: a set of queries that are spiking or dipping
- In MillWheel, the inputs and outputs are represented by (key, value, timestamp) triples.
    - Key: metadata field with semantic meaning in the system
    - Value: arbitrary byte string corresponding to the record.
    - The context where user code runs is scoped to a specific key, and each computation can define the keying for each input source.
- A pipeline of user computations will form a data flow graph, where its topology can be modified dynamically without having to restart
- MillWheel makes record processing idempotent with regard to the framework API, meaning it should not change. This should encapsulate the errors from the user codes.
- Delivery Guarantee: All internal updates within the MillWheel framework resulting from record processing are atomically checkpointed per key, and records are delivered exactly once

4. Core Concepts
    - MillWheel: allows essential elements of a streaming system while providing clean abstractions.
    - Computations
        - Application logic lives in computations
        - Invoked upon receipt of input data
        - Written to operate in the context of a single key and is agnostic to the distribution of keys among different machines
    - Keys

    - Keys are the primary abstraction for aggregation and comparison between different records in MillWheel.
    - For every record, the consumer specifies a key extraction function, which assigns a key to the record.
        - The code runs in the context of a specific key and is only granted access to state for that specific key.

- Streams
    - The delivery mechanism between different computations in MillWheel
    - Subscribe to zero or more input streams and publish one or more output streams, and the system guarantees delivery along these channels.
- Persistent state
    - An opaque byte string that is managed on a per-key basis
    - The user provides serialization and deserialization routines for which a variety of convenient mechanisms
    - The data system is backed by a replicated, highly available data store, ensuring data integrity that is completely transparent to the end user.
    - Common use: aggregated over windows of records and buffered data for a join
- Low watermarks
    - Provides a bound on the timestamps of future records arriving at the computation
    - Definition

        - Provides a recursive definition of low watermarks based on a pipelined's data flow
        - See paper for algorithm
        - Min(oldest work of A, low watermark of C: C outputs to A)
        - If there are no input streams, the low watermark and oldest work values are equivalent.
    - Seeded by injectors, which send data into MillWheel from external systems
        - They are only estimates, so computations could expect a small rate of late records.
        - Dropping such data while keeping track of how much data was dropped
        - Other pipelines retroactively correct their aggregates if late records arrive
- Timers
    - Per-key programmatic hooks that trigger at a specific wall time or low watermark value
    - Once set, timers are guaranteed to fire at increasing timestamp order

5. API

- Computation API
    - Two main entry points
        - ProcessRecord
        - ProcessTimer hooks
        - Triggers in reaction to record receipt and timer expiration

- Injector and Low Watermark API
    - At the system layer, each computation calculates a low watermark value for all of its pending work.
    - Rolled up automatically to provide API semantics
    - Injectors
        - Brings external data into MillWheel
        - Since injectors seed low watermark values into the pipeline, they are able to publish an injector low watermark that propagates any subscriber.
    - It can be distributed across multiple processes, such that the aggregate low watermark is used as the injector low watermark.
    - Users don’t have to write their own injector.

6. Fault Tolerance

- Delivery guarantees
    - It is possible to take non-idempotent user code and run it as if it were idempotent.
        - We relieve it
    - Exactly-once delivery
        - When receiving a record for a computation
            - 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.
    - The operations may be coalesced into a single checkpoint for multiple records.
    - The system assigns unique IDs to all records at production time.
        - Use the ID to record and discard the ACK
    - See papers
- Strong productions
    - Since MillWheel handles inputs that are not ordered or deterministic, we checkpoint-produced records before delivery in the same atomic write as state modification.
    - Without checkpointing, the session might be crashed first anyway.
        - However, with MillWheel, the user's application logic just works because they have been made into an idempotent operation by the system guarantees.
        - Use BigTable
- Weak production and Idempotency
    - Strong productions and exactly-once delivery make many computations idempotent with regard to system-level retires.
    - 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.

        - Weak productions
            - Broadcast downstream deliveries optimistically prior to persisting state.
            - Problems:
                - Completion times of the consecutive stages of the pipeline are now strictly coupled as they wait for downstream ACKs of records.

- State manipulation
    - Must satisfy the following
        - The system does not lose data
        - Updates to state must obey exactly once semantics
        - All persisted data must always be consistent
        - Low watermarks must reflect all pending states in the system
        - Timers must fire in order for a given key
    - Wrap all per-key updates in a single atomic operation
    - Results in resiliency against failures and unpredictable events
    - To address zombie writers and network issues, we attach a sequencer token for each write.
    - We can guarantee that, for a given key, only a single worker can write to that key at a particular point in time.
    - The single-writer guarantee is also critical to the maintenance of the soft state.

7. System Implementation

- Architecture

  - Runs as distributed systems on a dynamic set of host servers. Each computation runs on one or more machines, and streams are delivered via RPC.
  - A replicated master handles load balancing and distribution and divides each computation into a set of owned lexicographic key intervals.
    - Each interval has a sequencer that tracks its stages
  - Recover from failures efficiently by scanning metadata from this backing store whenever a key interval is assigned to a new owner.
- Low watermarks
  - It must be implemented as a sub-system that is globally available and correct
  - Implements as a central authority, which tracks all low watermark values in the systems and journals them to a persistent state, preventing the reporting of erroneous values in cases of process failures.
  - When reporting to a central authority
    - Each process aggregates timestamp information for all of its owned work
    - Low watermark updates are also bucketed into key intervals and sent to the central authority.
  - The central authority should be conservative with its workers
  - To maintain consistency, we attach sequencers to all low-watermark updates.
  - This implementation does not require any strict time ordering on streams in the system, reflecting both in-flight and persisted state.

- We prevent logical inconsistencies by establishing a global source of truth for low watermark values.

Answers to Questions

1. Question 1
    - A low watermark is a metric or marker that assists in identifying when all data up to a specific point in time has been received or processed by a specific stage of the system. It helps the system to better manage data completeness and integrity, which is especially helpful in remote systems where data may come out of order or at various times.
    - For example, to process financial transactions in a bank, a low watermark, for instance, could be used to make sure that all transactions up to a specific timestamp have been completed before creating daily reports or carrying out batch operations, like calculating interest. In spite of transactions arriving at different times or out of order as a result of variable processing speeds or network delays, the bank would be guaranteed to operate on a complete dataset and retain accuracy in its daily summary and customer account updates.
2. Question 2
    - In stream processing systems, a "timer" is a scheduling mechanism that sets triggers for the execution of particular actions at predefined intervals. Timers are very helpful because they allow the system to operate based on temporal conditions, guaranteeing that tasks are completed at the appropriate times without requiring constant human

monitoring. They enhance the effectiveness and dependability of system processes by assisting in the management of time-dependent events.

- For example, consider a smart home system. Timers are used to control various devices based on user-defined schedules. During the winter, a timer may be programmed to activate the heating system every day at 6:00 PM, ensuring that the house is warm enough when the user returns from works.

3. Question 3
    - Weak production checkpointing addresses the problem of maintaining high throughput and efficiency in stream processing systems while ensuring data consistency and reliability during failures or restarts.
    - It is helpful, especially when dealing with systems where components might fail after having sent out data but before this data is permanently recorded. Without it, the results may have lost data or produced duplicated resources.
    - By broadcasting data to downstream processes without waiting for the data to be fully persisted or recognized, weak production checkpointing acts as a fallback if the sending process fails after broadcasting but before the data is verified and saved. This approach improves the system's overall efficiency by reducing the latency. It also selectively checkpoints critical or slow-responding operations, helps maintain faster data flow and lowers latency across the system.

Additional Questions and Observations

1. I don’t have any questions regarding this case

2. I like how versatile and flexible can be when designing a system. In different cases, streaming data comes in many different sizes and formats, which have is quite challenging to take care of that. And now, with MillWheel, things are getting easier.

Part 2: Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R.J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., Whittle, S. (2015) *The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing*. Proc. of VLDB, vol. 8, no. 12, pp. 1792–1803

Reading Notes

1. Introduction
    - General notes
        - Current systems suffer from latency problems inherent in collecting all input data into a batch before processing it.
        - Those that provide scalability and fault tolerance fall short on expressiveness or correctness vectors.
        - MillWheel and Spark Streaming are both sufficiently scalable, fault-tolerant, and low-latency but lack high-level programming models that make calculating event time sessions straightforward.
        - Suggest a new abstracted away beneath a model of sufficient generality and flexibility. The choice of execution engine comes down to latency and resource cost.
        - Required models

        - Windowing model: supports unaligned event time windows and a simple API.
            - Event time means the time at which events occur, not processed.
        - Triggering model: binds the output times of results to runtime characteristics of the pipeline with a powerful and flexible declarative API
        - Incremental processing model: integrates retractions and updates into the windowing and triggering models
        - Scalable implementations: implements the above atop the MillWheel and FlumeJava, with an external reimplementation for Google Cloud Dataflow

- Unbounded/Bounded v. Streaming/Batch
    - Prefer unbounded/bounded over the other because the latter carries the implication of using one engine over another.
- Windowing
    - Windowing slices up a dataset into finite chunks for processing as a group.
    - Always time-based, aligned (applied to all windows in sessions) v. unaligned (applied across only specified subset of data)
    - Fixed windows: defined by a static window size, generally aligned. However, they don't align sometimes when spreading the window completion load evenly across time.

    - Sliding windows: defined by a window size and slide period. The period may be less than the size, meaning we may have an overlap. Typically aligned. A fixed window is really a special case of Sliding when size equals period.
    - Sessions: windows that capture some period of activity over a subset of the data, or per key in this case.
        - Typically defined by a timeout gap, including those that have a span of time <= the timeout.
        - Unaligned windows.
- Time Domains
    - Two domains of interest
        - Event time: the time at which the event itself actually occurred.
        - Processing time: the time at which an event is observed at any given point during processing within the pipeline
    - Event time never changes, but processing time changes constantly for each event.
    - The realities of these systems result in an inherent and dynamically changing amount of skew between the two domains.
    - Completeness is different from correctness, so we won't rely on a watermark.
    - The time domain skew would always be zero. We would process all events immediately as they happen.
    - The dynamic variance in skew is very common in distributed data processing systems and will play a big role in defining what functionality is necessary for providing correct, repeatable results.

2. DataFlow Model

- Core Primitives
    - The DataFlow SDK has two core transforms:
        - ParDo: for generic parallel processing. Each input is provided to a user-defined function. The ParDo operation operates element-wise on each input element and thus translates naturally to unbounded data
        - GroupByKey: kor key-grouping pairs. It collects all data for a given key before sending them downstream for reduction.
    - If the input source is unbounded, we don't know when it will end. Solution: window the data.
- Windowing
    - Typically, the GroupByKey operation is redefined to essentially be GroupByKeyAndWindow.
    - Support unaligned windows, which have two key insights
        - Treat all windowing strategies as unaligned from the perspective of the model. Allows underlying implementations to apply optimization relevant to the aligned cases where applicable.
        - Broken into two operations

- `Set<Window>` **`AssignWindows`**`(T datum)`, which assigns the element to zero or more windows. This is essentially the Bucket Operator from Li [22].
- `Set<Window>` **`MergeWindows`**`(Set<Window> windows)`, which merges windows at grouping time. This allows data-driven windows to be constructed over time as data arrive and are grouped together.
-

        - Two operations are intimately related – sliding window assignment requires sliding window merging. Sessions window assignment requires session window merging.
    - Window assignment
        - Creates a new copy of the element in each of the windows to which it has been assigned
        - Each of the two (key, value) pairs is duplicated to exist in both windows that overlap the element's timestamp.
    - Window merging
        - Occurs as part of the GroupByKeyAndWindow operations
            - DropTimestamps: Drop element timestamps
            - GroupByKey: groups (value, window) tuples by key
            - MergeWindows: Merge the set of currently buffered windows for a key.
            - GroupAlsoByWindow: for each key, group values by window
            - ExpandToElements: expand per-key, per-window groups of values into tuples with new per-window timestamps.
            - See paper for diagram
- Triggers & Incremental Processing
    - Watermark alone is insufficient.
    - Feature triggers:
        - Allow the specification of when to trigger the output results for a given window.

- The mechanism for stimulating the production of GroupByKeyAndWindow results in response to internal or external signals
- Complementary to the windowing model. They affect system behavior along a different axis of time
    - Windowing: determines where in event time data are grouped together for processing
    - Triggering determines when in processing time, the results of groupings are emitted as panes
- Three different refinement modes to control how panes for the same window relate to each other
    - Discarding
        - Upon triggering, window contents are discarded, and later results are nonrelated to previous results.
        - This mode is useful where the downstream consumer of the data expects the values from various trigger fires to be independent.
        - Most efficient in terms of the amount of data buffered though for associative and commutative
    - Accumulating
        - Upon triggering, the window contents are left intact in a persistent state, and later results become a refinement of previous results.

        - Useful when the downstream consumer expects to overwrite old values with new ones when receiving multiple results for the same window.
    - Accumulating & Retracting
        - Upon triggering, in addition to the Accumulating semantics, a copy of the emitted value is also stored in a persistent state.
        - When the window triggers again in the future, a retraction for the previous value will be emitted first, followed by the new value as a normal datum.
        - Retractions are necessary for pipelines with multiple serial GroupByKeyAndWindow operations since the multiple results generated by a simple window over subsequent trigger fires may end up on separate keys when grouped down downstream.
- Examples
    - See papers for notes.

3. Implementation & Design
    - Implementation
        - We implemented this model internally in FlumeJava, with MillWheel used as the underlying execution engine for streaming mode
        - An external reimplementation of Cloud Dataflow is largely complete at the time of writing

    - The core windowing and triggering code is quite general, and a significant portion of it is shared across batch and streaming implementations; that system itself is worthy of a more detailed analysis.

- Design Principles
    - Set of principles
        - Never rely on any notion of completeness.
        - Be flexible to accommodate the diversity of known use cases and those to come in the future.
        - Not only makes sense but also adds value in the context of each of the envisioned execution engines
        - Encourage clarity of implementation
        - Support robust analysis of data in the context in which they occurred
- Motivating Experiences
    - Large-scale backfills & The Lambda Architecture: Unified Model
        - A better setup would be to have a single implementation written in a unified model that could run in both streaming and batch mode without modification.
        - Customers stopped trusting the weakly consistent results over time, and as a result, reimplemented around strong consistency mode with a nightly MR to generate truth.
    - Unaligned Windows
        - Very important use case within Google
        - It is trivial

- Billing: Triggers, Accumulation, & Retraction
  - Watermark lags cause issues in the workflow
  - Two folds result
    - Triggers: allowing the concise and flexible specification of when results are materialized
- Statistics Calculation: Watermark Triggers
  - Calculates aggregate statistics
  - Watermark allows us to achieve a high level of accuracy
- Recommendations: Processing Time Triggers
- Anomaly Detection: Data-Driven & Composite Triggers
  - Motivates data-driven triggers.
  - Observe the stream of queries and calculate statistical estimates of whether a spike exists or not.
  - Emit a start record when a spike is happening and then a stop when it ceases.
  - Ok, but not suitable for a large number of use cases.

Answers to Questions

1. Question 1
   - Sessions are windows that capture some period of activity over a subset of the data or per key in this implementation. Each session is defined by periods of activity separated by a specified gap of inactivity.
   - How are they different?

- Fixed windows: defined by a static window size, generally aligned.
- Sliding windows: defined by a window size and slide period. The eriod may be less than the size, meaning we may have an overlap.
- In contrast, session windows are not based on set intervals but rather on the behavior of the data. A session starts when an event occurs and extends until a specified duration of inactivity occurs. If another event occurs before the inactivity period ends, the window extends further.

2. Question 2

- Premature watermark
  - The watermarks progress too quickly, which can cause them to overlook relevant but late-arriving data. This oversight can compromise the correctness of the output because the system disregards valid data inputs, assuming that all necessary data has already been processed.
  - This can lead to incomplete or incorrect results.
- Delayed watermark
  - Watermarks may progress very slowly, usually due to anomalies or stragglers within the data stream.
  - This approach ensures that no potentially relevant data is missed, thus maintaining data integrity and correctness. However, the trade-off is increased latency, making the streaming process less efficient.

3. Question 3

- A "pane" is a subset of its results. It is the outputs produced each time the window's triggering criteria are satisfied. Panes allow for the incremental processing and output

of windowed data in a streaming system, which can be crucial when handling large or continuously updating datasets.

- Three refinement modes
    - Discarding mode:
        - This mode discards all data in a pane after its results are emitted. Subsequent results are processed as if starting fresh, which minimizes state storage and is useful when only the latest results are needed.
    - Accumulating mode
        - In this mode, data within a pane is retained after it's emitted. Each new pane includes results from all previous panes, continuously building a comprehensive result over time. This is beneficial for ongoing aggregations where each output adds to the previous data.
    - Accumulating and retracting mode:
        - Extending the accumulating mode. This mode also issues retractions.
        - Before issuing new results, the system retracts the previous pane’s results and then updates them with a new result that includes all cumulative data.
        - This ensures downstream systems can update their states accurately, accommodating changes or late data.

4. Question 4
    - The transition from a batch pipeline to one handling unbounded input results in no output when using watermarks and a global window.

- This happens because, with unbounded data, the watermark that triggers output can never advance past the global window, which covers all of the event time, effectively preventing any output production. The watermark is expected to signal the completeness of data up to a certain point, and the data is unbounded (theoretically infinite); the watermark cannot fulfill its role in triggering the output, leading to a scenario where no output is ever produced.
- To fix this
    - Change the Trigger: Adopt a trigger that doesn't rely only on the watermark, such as one based on processing time, which would allow for periodic outputs regardless of data completeness.
    - Change the Windowing Strategy: Instead of a global window, use fixed or sliding windows based on event time, which enable the watermark to advance past these smaller windows, thus facilitating regular outputs.

5. Question 5
    - 3.3.3
        - Billing pipelines on MillWheel required adjustments for accurate and timely billing based on streaming data
        - Features supported
            - Triggers facilitated the flexible emission of outputs.
            - Accumulation and Retraction allowed updates and corrections to previous outputs as new data arrived, ensuring accuracy.
    - 3.3.4

    - Pipelines calculating aggregate statistics, like latency averages, where complete accuracy wasn't critical, but speed was.
    - Features supported
        - Watermark Triggers timed the output of aggregates based on a largely complete data view, balancing speed with representativeness.

- 3.3.5
    - Watermark Triggers timed the output of aggregates based on a largely complete data view, balancing speed with representativeness.
    - Features supported
        - Watermark Triggers timed the output of aggregates based on a largely complete data view, balancing speed with representativeness.
- 3.3.6
    - Watermark Triggers timed the output of aggregates based on a largely complete data view, balancing speed with representativeness.
    - Features supported
        - Data-Driven Triggers: responded to particular patterns, like spikes in queries, enabling real-time alerts.
        - Composite Triggers supported complex decision-making by combining multiple trigger conditions for contextually relevant outputs.

Additional Questions and Observations

1. I don’t really have any questions.

2. I think this system is really flexible, adapting to data behavior based on data-driven and composite triggers. This is great as it provides the users with more use cases that can actually well fit into their needs.