FLIP-326: Enhance Watermark to Support Processing-Time Temporal Join

<table>
<thead>
<tr>
<th>Discussion thread</th>
<th><a href="https://lists.apache.org/thread/6m47ggxn9sl0qgfgyq20ytyzhw42crxj">https://lists.apache.org/thread/6m47ggxn9sl0qgfgyq20ytyzhw42crxj</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vote thread</td>
<td>-</td>
</tr>
<tr>
<td>JIRA</td>
<td>-</td>
</tr>
<tr>
<td>Release</td>
<td>-</td>
</tr>
</tbody>
</table>

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

[This FLIP proposal is a joint work between Xuannan Su and Dong Lin]

• Motivation
• Terminology and Background
• Public Interfaces
• Proposed Changes
• Example Usage
• Compatibility, Deprecation, and Migration Plan
• Test Plan
• Future Work

Motivation

Assuming the user needs to perform a processing-time temporal join where the Probe Side records are obtained from a Kafka Source and the Build Side records are obtained from a MySQL CDC Source, which consists of a snapshot reading phase followed by a binlog reading phase. Notably, all input records lack event-time information. The user's requirement is that each record on the Probe Side must be joined with at least the records from the Build Side's snapshot phase. In other words, the Join operator needs to wait for the completion of the Build Side's snapshot phase before processing the Probe Side's data.

Currently, Flink does not support the aforementioned use-case. In Flink SQL, the "SYSTEM_TIME AS OF" syntax, used for temporal joins with the latest version of any view/table, is not supported. Although the TemporalProcessTimeJoinOperator enables temporal joins based on processing time, it does not support the Probe Side waiting for records from the Build Side. Consequently, there is a risk that the operator may commence processing the Probe Side's data before reading the data from the Build Side's snapshot phase. This can result in situations where the Probe Side's data cannot be joined with any records, leading to output that fails to meet the user's requirements. For further details, please refer to FLINK-19830.

This document proposes the introduction of APIs that allow source operators (e.g., HybridSource, MySQL CDC Source) to send watermarks to downstream operators, indicating that the watermark should start increasing according to the system time. In addition to supporting processing-time temporal joins, this FLIP provides the fundation to simplifies DataStream APIs such as KeyedStream#window(...), such that users would no longer need to explicitly differentiate between TumblingEventTimeWindows and TumblingProcessingTimeWindows, leading to a more intuitive experience.

Terminology and Background

The FLIP proposes changes to Flink's watermark and timestamp concepts. To better understand the underlying design, let's recap the relevant concepts in this section.

**Probe Side**: Refers to the left side of the stream in a temporal join, also known as the Fact Table. Typically, data on the Probe Side doesn't need to be retained after processing.

**Build Side**: Represents the right side of the stream in a temporal join, often referred to as the Dimension Table. It can be a versioned table and typically contains the latest data for each key.

**Watermark**: Serves as a signal to operators that no elements with a timestamp older than or equal to the watermark timestamp should arrive at the operator.

**TimestampAssigner**: Responsible for assigning event-time timestamps to elements. These timestamps are utilized by functions operating on event time, such as event time windows.

Here is the 'TimestampAssigner' interface:
### TimestampAssigner

```java
public interface TimestampAssigner<T> {
    long extractTimestamp(T element, long recordTimestamp);
}
```

### WatermarkGenerator

Generates watermarks either based on events or at regular intervals.

Here is the `WatermarkGenerator` interface:

```java
@Public
public interface WatermarkGenerator<T> {
    void onEvent(T event, long eventTimestamp, WatermarkOutput output);
    void onPeriodicEmit(WatermarkOutput output);
}
```

In a DataStream program, the determination of event time and watermark values follows these steps:

- When creating a source, the user provides a `WatermarkStrategy to StreamExecutionEnvironment#fromSource`.
- If the source natively supports event time (e.g., KafkaSource) or the user provides a custom `TimestampAssigner in the WatermarkStrategy` to extract the timestamp from the record, Flink will add the timestamp to the record. Otherwise, the timestamp on the record will be set to `Long.MIN_VALUE`.
- If the user employs `NoWatermarkGenerator in the WatermarkStrategy`, the job will not generate watermarks. Otherwise, the job will periodically emit watermarks, and the watermark value depends on event time. The frequency of watermark emission is determined by `pipeline.auto-watermark-interval`, with a default value of 200ms.

### Public Interfaces


```java
```
Watermark

/**
 * Watermarks are the progress indicators in the data streams. A watermark signifies that no events
 * with a timestamp smaller than or equal to the watermark's time will occur after the watermark.
 * <ul>
 *   <li>A watermark with a timestamp <i>T</i> and useProcessingTime set to false indicates that the
 *       event time of the stream has progressed to time <i>T</i>.</li>
 *   <li>A watermark with a timestamp <i>T</i> and useProcessingTime set to true indicates that the
 *       event time of the stream progresses in synchronization with the system time. The timestamp
 *       <i>T</i> must be less than the current system time. Otherwise, an exception will be thrown.</li>
 * </ul>
 * <p>Watermarks are created at the sources and propagate through the streams and operators.</p>
 * <p>In some cases a watermark is only a heuristic, meaning some events with a lower timestamp may
 * still follow. In that case, it is up to the logic of the operators to decide what to do with the
 * "late events". Operators can for example ignore these late events, route them to a different
 * stream, or send update to their previously emitted results.</p>
 * <p>When a source reaches the end of the input, it emits a final watermark with timestamp (@code
 * Long.MAX_VALUE), indicating the "end of time".</p>
 * <p>Note: A stream's time starts with a watermark of (@code Long.MIN_VALUE). That means that all
 * records in the stream with a timestamp of (@code Long.MIN_VALUE) are immediately late.</p>
 * <p>Note: After sending a watermark with useProcessingTime set to true, the source should only send
 * subsequent watermarks with useProcessingTime set to true. Sending a watermark with
 * useProcessingTime set to false will result in an exception.</p>
 */

@Public
public final class Watermark implements Serializable {
    ...

    /**
     * If useProcessingTime set to false, this is the time of the watermark in milliseconds. If
     * useProcessingTime set to true, this is the last effective time of the watermark in
     * milliseconds.
     */
    private final long timestamp;

    /**
     * If this is true, this watermark indicates the event time of the stream progresses in
     * synchronization with the system time.
     */
    private final boolean useProcessingTime;

    public Watermark(long timestamp) {
        this(timestamp, false);
    }

    public Watermark(long timestamp, boolean useProcessingTime) {
        this.timestamp = timestamp;
        this.useProcessingTime = useProcessingTime;
    }

    /** Returns whether the time of watermark can be determined by the system time. */
    public boolean useProcessingTime() {
        return useProcessingTime;
    }
}
Please note that the proposed change enhances the expressiveness of the Watermark class without altering its core abstraction. When `useProcessingTime` is set to false, the watermark becomes a static value represented by the `Watermark#timestamp`. Conversely, when `useProcessingTime` is set to true, the watermark should be dynamically derived using a function such as `System#currentTimeMillis`. In both cases, the Watermark class serves the purpose of instructing downstream operators on how to determine the appropriate watermark value.

2) Update `AbstractStreamOperator/AbstractStreamOperatorV2` to handle `Watermark#useProcessingTime` correctly:

**Original Behavior:**
- Upon receiving an instance of Watermark, the operator triggers event-time timers whose scheduled time is less than or equal to `watermark#timestamp`.

**Updated Behavior:**
- Upon receiving a watermark:
  - If `watermark#useProcessingTime` is false, the operator triggers event-time timers whose scheduled time is less than or equal to `watermark#timestamp`.
  - If `watermark#useProcessingTime` is true, the operator starts a scheduler to dynamically trigger event-time timers whose scheduled time is less than or equal to `System#currentTimeMillis`.

3) Update `NoWatermarksGenerator#onPeriodicEmit` to emit `Watermark(timestamp=Long.MIN_VALUE, useProcessingTime=true)`.

Note that `NoWatermarksGenerator` is currently only used when every operator in the job operates in "processing time mode" and these operators do not rely on the watermark value. Therefore, the proposed change will not disrupt existing jobs that utilize `NoWatermarksGenerator`.

4) Add `delegateWatermarkPeriodicEmit()` to the `SourceReader` interface.

**Proposed Changes**
1) Update SourceOperator behavior

- The SourceOperator should invoke `SourceReader#delegateWatermarkPeriodicEmit` with a runnable that invokes `WatermarkGenerator#onPeriodicEmit`. If `delegateWatermarkPeriodicEmit()` returns true, the SourceOperator should not start any scheduler that invokes `WatermarkGenerator#onPeriodicEmit`.
- Otherwise, if `emitProgressiveWatermarks` is false, the SourceOperator should invoke `WatermarkGenerator#onPeriodicEmit` once before emitting the first record.
- Otherwise, the SourceOperator will maintain its existing behavior of starting a scheduler to periodically invoke `WatermarkGenerator#onPeriodicEmit`.

2) Update sources with bounded/unbounded phases (e.g., HybridSource, MySQL CDC Source) to override `SourceReader#delegateWatermarkPeriodicEmit`.

These sources should invoke `onWatermarkEmit` when they want to notify downstream operators of the watermark value, such as at the beginning of the MySQL CDC binlog phase.

3) Update TemporalRowTimeJoinOperator to optimize the case where watermarks from both inputs have `useProcessingTime` set to true.

After the modifications proposed in this FLIP, the TemporalRowTimeJoinOperator, as a subclass of AbstractStreamOperator, can support temporal joins based on processing time when both the build side and probe side send watermarks with `useProcessingTime` set to true.

To optimize performance, when both the probe side and build side receive a watermark with `useProcessingTime` set to true, the operator can process the data in pure processing time mode without timers. The probe side data can directly join with the build side, and the build side only needs to keep the latest record without triggering clean-up based on event time.

4) Update Flink SQL Planner to support processing-time temporal join and remove `TemporalProcessTimeJoinOperator`.

The Flink SQL planner will be updated to select the appropriate temporal join operator.

**Original behavior:**

- For temporal joins based on row time (event time), use `TemporalRowTimeJoinOperator`.
- For temporal joins based on processing time:
  - If the `TemporalFunctionJoin` syntax is used, use `TemporalProcessTimeJoinOperator`.
  - Otherwise, throw an exception.

**Updated behavior:**

- Use `TemporalRowTimeJoinOperator` for all temporal joins.

And we can remove `TemporalProcessTimeJoinOperator` because its functionality is covered by `TemporalRowTimeJoinOperator`.

5) Update `WatermarkToDataOutput` to handle `Watermark#useProcessingTime`.

The `WatermarkToDataOutput` is responsible for sending the watermark from the `SourceReader` downstream and ensuring that the watermark never decreases. The proposed change will maintain this semantics.

**Original behavior:**

- When the `WatermarkToDataOutput` receives a watermark, it checks the timestamp of the watermark. It only sends the watermark downstream if the timestamp of the watermark is greater than the timestamp of the most recently sent watermark.

**Updated behavior:**

- When the `WatermarkToDataOutput` receives a watermark, it checks the timestamp and the `useProcessingTime` field of the watermark:
  - If the watermark to be sent has `useProcessingTime` set to true and the current system time is less than the timestamp of the most recently sent watermark, an exception is thrown.
  - If a watermark with `useProcessingTime` set to true has been previously sent, and the watermark to be sent has `useProcessingTime` set to false, an exception is thrown to maintain consistency.
  - It sends the watermark downstream if the timestamp of the watermark is greater than the timestamp of the most recently sent watermark or if the `useProcessingTime` field is set to true.

Overall, these updates in the `WatermarkToDataOutput` ensure that when `useProcessingTime` is false, the watermark timestamp never decreases. It also guarantees that information with `useProcessingTime` set to true is sent downstream and that the `useProcessingTime` flag remains true once it has been set.
6) Update StatusWatermarkValve to handle Watermark#useProcessingTime.

The purpose of the StatusWatermarkValve is to calculate the current watermark for an input with multiple channels and invoke the processWatermark method of the operator. The primary objective is to ensure that the watermark passed to the processWatermark method is the minimum among all the input channels of that input and that it never decreases. The proposed change will maintain this semantics.

Original behavior:
- Each input channel can be either active or idle. The StatusWatermarkValve updates the status of each input channel when it receives a WatermarkStatus from the upstream.
- Each input channel can be either aligned or unaligned. An input channel is considered aligned if it is active and its watermark is greater than or equal to the last watermark timestamp of the input.
- The StatusWatermarkValve calculates the minimum watermark timestamp among all the aligned input channels and uses it as the watermark for the input. If the new watermark timestamp is greater than the previous watermark timestamp, it invokes the processWatermark method.

Updated Behavior
- Each input channel can be either active or idle. The StatusWatermarkValve updates the status of each input channel when it receives a WatermarkStatus from the upstream. If the input channel has useProcessingTime set to true, it is considered active.
- Each input channel can be either aligned or unaligned. If the useProcessingTime of the last watermark of the input is set to false, an input channel is considered aligned if it is active and its watermark is greater than or equal to the last watermark timestamp of the input. If the useProcessingTime of the last watermark of the input is set to true, an input channel is considered aligned if it is active and its useProcessingTime is set to true.
- If there exists any input channel with useProcessingTime set to false, the watermark of the input is the minimum watermark timestamp among all the aligned input channels whose useProcessingTime is false. Otherwise, the watermark of the input is Watermark(timestamp=Long.MIN_VALUE, useProcessingTime=true).

With these updates, we can still ensure that the effective watermark never decreases. The chart below illustrates the effective watermark of the input given the watermark of the two input channels.

<table>
<thead>
<tr>
<th>InputChannel1</th>
<th>InputChannel2</th>
<th>currentTimestamp = t2</th>
<th>currentTimestamp = t2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>useProcessingTime = true</td>
<td>useProcessingTime = false</td>
</tr>
<tr>
<td>currentTimestamp = t1</td>
<td>Watermark(Long.MIN_VALUE, true)</td>
<td>Watermark(t2, false)</td>
<td></td>
</tr>
<tr>
<td>useProcessingTime = true</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>currentTimestamp = t1</td>
<td>Watermark(t1, false)</td>
<td>Watermark(MIN(t1, t2), false)</td>
<td></td>
</tr>
<tr>
<td>useProcessingTime = false</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: If any operator emits Watermark(timestamp=t, useProcessingTime=true), it is required that t <= System.currentTimeMillis. This effectively makes Watermark(timestamp=t, useProcessingTime=true) equivalent to Watermark(timestamp=Long.MIN_VALUE, useProcessingTime=true).

7) Update IndexedCombinedWatermarkStatus to handle Watermark#useProcessingTime.

The IndexedCombinedWatermarkStatus represents the combined value and status of a watermark for a set number of input partial watermarks. The operator advances the event time of timeServiceManager based on the combined watermark. The objective of the IndexedCombinedWatermarkStatus is to ensure that the watermark timestamp of the operator, which has multiple inputs, is the minimum timestamp among all the inputs and that it never decreases. The proposed change will maintain this semantics.

Original behavior:
- Each channel can be either active or idle. The IndexedCombinedWatermarkStatus updates the status of the input when it receives a WatermarkStatus from that input.
- The IndexedCombinedWatermarkStatus calculates the minimum watermark timestamp among all the active inputs and uses it as the watermark of the operator. If the new watermark timestamp is greater than the previous watermark timestamp, it advances the event time of timeServiceManager.

Updated Behavior
- Each channel can be either active or idle. The IndexedCombinedWatermarkStatus updates the status of an input when it receives a WatermarkStatus from that input. If the useProcessingTime of an input is set to true, it is considered active.
- If there exists any input with useProcessingTime set to false, the watermark timestamp of the operator is determined as the minimum watermark timestamp among all the active inputs. Otherwise, all inputs have useProcessingTime set to true, indicating the use of processing time instead of event time, the watermark of the operator is set to Watermark(timestamp=Long.MIN_VALUE, useProcessingTime=true).

By incorporating these updates, we ensure that the IndexedCombinedWatermarkStatus maintains the desired behavior of having the minimum watermark timestamp among all inputs while preventing any decrease in the watermark.
Example Usage

Here is the Flink SQL example that demonstrates how to perform processing time temporal join after the FLIP.

```sql
-- Create mysql cdc source table (dimension table)
CREATE TEMPORARY TABLE user_info {
    user_id INTEGER PRIMARY KEY NOT ENFORCED,
    gender STRING
} WITH {
    'connector' = 'mysql-cdc',
    'database-name' = 'example_database',
    'hostname' = 'localhost',
    'username' = 'root',
    'password' = 'root',
    'table-name' = 'user_info'
};

-- Create datagen source table (fact table)
CREATE TEMPORARY TABLE click_event {
    user_id INTEGER,
    item_id INTEGER,
    proctime AS PROCTIME()
} WITH {
    'connector' = 'datagen',
    'rows-per-second' = '1',
    'fields.user_id.min' = '0',
    'fields.user_id.max' = '9'
};

-- Create a print sink table
CREATE TEMPORARY TABLE print_sink {
    user_id INTEGER,
    item_id INTEGER,
    gender STRING
} WITH {
    'connector' = 'print'
};

-- Processing time temporal join
INSERT INTO print_sink
SELECT click_event.user_id AS user_id, item_id, gender FROM click_event
LEFT JOIN user_info FOR SYSTEM_TIME AS OF click_event.proctime
ON click_event.user_id = user_info.user_id;
```

Compatibility, Deprecation, and Migration Plan

The proposed change might be negative impact user experience in the following scenario:

- User writes a Flink SQL program with a temporal join in processing time that does not use the TemporalFunctionJoin syntax,
- The source on the Build Side employs a bounded + unbounded phase internally (e.g., HybridSource, MySQL CDC) but has not been updated to overwrite SourceReader#delegateWatermarkPeriodicEmit as specified in this FLIP
- User prefers to have the job fail fast rather than producing results where the probe-side records fail to join with the records from the bounded phase of the build-side source.

However, we believe that the benefits of this FLIP outweigh the impact of the above scenario. In the common case, users can identify the data quality issue and upgrade the source library version to resolve the problem. We will explain this change in the Flink release notice to ensure users are aware of it.

Apart from the issue mentioned above, the changes made in this FLIP are backward compatible for the following reasons:

- We only introduce new APIs, which do not cause existing code handling Watermarks to fail. Moreover, the default setting for the useProcessingTime parameter in Watermark instances is false, preserving the existing semantics.
With the updates to AbstractStreamOperator/AbstractStreamOperatorV2 based on this FLIP, all operators can now support Watermarks with the useProcessingTime field and correctly trigger the operator's timer based on event time or system time. For sources that have not been updated, the Watermarks they send always have useProcessingTime set to false. In this case, the behavior of the operators remains unchanged, ensuring compatibility with existing jobs.

Test Plan

The change will be covered with unit and integration tests.

Future Work

After this FLIP, we can unify the API for processing time and event time. The following are some examples of the APIs pairs that distinguish between event time and processing time. Currently, in the DataStream API, users need to explicitly differentiate between Processing Time and Event Time in several places when expressing job logic.

When invoking methods like `DataStream.windowAll` or `KeyedDataStream.window`, users need to select the appropriate `WindowAssigner` based on processing time or event time.

- org.apache.flink.streaming.api.windowing.assigners.WindowAssigner
  - TumblingEventTimeWindows vs TumblingProcessingTimeWindows
  - SlidingEventTimeWindows vs SlidingProcessingTimeWindows
  - DynamicEventTimeSessionWindows vs DynamicProcessingTimeSessionWindows
  - EventTimeSessionWindows vs ProcessingTimeSessionWindows

When implementing custom `ProcessFunction` or `KeyedProcessFunction`, users need to differentiate between registering a timer for processing time or event time using the `TimerService`.

- org.apache.flink.streaming.api.TimerService
  - registerProcessingTimeTimer vs registerEventTimeTimer