KIP-328: Ability to suppress updates for KTables

Status

Current state: *Accepted*

Discussion thread: thread

JIRAs:

- Unable to render Jira issues macro, execution error.
- Unable to render Jira issues macro, execution error.
- Unable to render Jira issues macro, execution error.
- Unable to render Jira issues macro, execution error.

- Unable to render Jira issues macro, execution error.
- Unable to render Jira issues macro, execution error.

Release: 2.1

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

Motivation

Kafka Streams represents continuously updating computations over entities via KTable.

Kafka Streams currently provides record caches to control the rate at which updates are emitted from KTables:

- https://www.confluent.io/blog/watermarks-tables-event-time-dataflow-model/
- https://docs.confluent.io/current/streams/architecture.html#record-caches

The relevant configuration parameters (`cache.max.bytes.buffering` and `commit.interval.ms`) apply to the entire Streams app. This is based on the theory that (1) rate-limiting is primarily an operational concern for the app operator and (2) cache size is best configured at the machine level so that the total memory footprint can be managed. In terms of stream processing logic, rate-limiting does not affect the semantics of the application, so it didn't seem to belong in the Streams DSL directly.

Request shaping

The current design does not account for the effect of rate-limiting on downstream consumers. For example, some applications will contact an external system for every event, so they may prefer to limit this stream more aggressively than the rest of the application.

Easier config

Users of Streams have reported difficulty in achieving their desired results with the existing cache-based model. This is variously due to confusion (because the model yields complex interactions with the various operators) or because of actual bugs in the implementation (but the mentioned complexity makes it difficult to decide if observed behavior is a bug or not). Having a more explicit and semantically tighter controls will make improve both of these struggles.

"Final result" of a window aggregation

Windowed computations in particular present a unique opportunity. The output of a windowed computation is a KTable in which the keys are annotated with window ids. Because the window has a defined end, we know that the keys belonging to that window can never be modified again once that end time has passed, except by out-of-order events. If we bound the "maximum allowed delay" of events, we can know the "final value" of the window. This is especially useful for taking irrevocable actions in response to window results, such as sending an email or alert when a windowed aggregation is below a certain threshold.

See Unable to render Jira issues macro, execution error. for one example of this.

Proposed Changes

New operator: "suppress"

I'm proposing to add a new KTable operator: `suppress`.

`Suppress` is for dropping some updates from a `KTable`'s changelog stream in a semantically sound way.

At the moment, I'm proposing to provide two main kinds of suppressions:

1. **All but the final result for windows**. You can use suppress get exactly one final result per window/key for windowed computations. This includes both time and session windows.
    a. This feature requires adding **a "grace period" parameter for windows**.
2. **Intermediate Events**. You can use suppress to wait for more updates on the same key before emitting them downstream. There are two main configuration knobs to tune here:
    a. **How long to wait for more updates before emitting**. This is an amount of time, measured either from the event time (for regular KTables) or from the window start (for windowed KTables), to buffer up each key before emitting them downstream.
    b. **How much memory to use for buffering events (and what to do when it fills up)**. Suppress needs to remember the last observed event for each key, which takes memory. You can configure the amount of memory either by number of keys or by raw heap size. You can configure what to do when the buffer runs out of memory:
        i. **Emit when full**. This option is basically "best effort" suppression. If `suppress` runs out of memory, it will emit the oldest event.
        ii. **Spill to disk**. This option is for when you need suppression to be strict. If the operator runs out of memory, it will allocate a RocksDB database and begin storing new records there. This may be complicated to implement, and consequently might be a future addition.
        iii. **Shut down**. There is an option to run with the assumption that your "live set" of suppression events will fit within memory. If this assumption turns out to be false, the application will run out of memory and shut down. This may sound strange, but it's no different than using an `InMemoryKeyValueStore` or even an `ArrayList`. In the case of suppress, you can either leave the memory buffer unconstrained, which will cause an application crash if it runs out of memory, or you can configure it with a memory constraint and tell it to shut down gracefully when it runs out. Either one is safe from a correctness standpoint, but the latter will result in faster recovery.

Suppression patterns

Final window results per key

You can achieve exactly one event per key per window by configuring:

- A grace period on the window. This would be a balance of how late you expect events to be and how long you're willing to wait before finalizing the window and emitting the result.
- Suppress with "untilWindowClose". There's a specific "Suppressed" configuration for this feature. It allows you to configure the memory buffer and decide whether to shut down or switch to disk.

Best-effort rate limit per key

Configure:

- Emit-when-full memory strategy
- Suppress with "untilWindowClose": the inverse of the desired rate. If the desired rate is R (events per second), the inverse is the buffering time: 1 /R (seconds per event).

Strict rate limit per key

Configure:

- Unlimited memory (either with spill-to-disk or just not limiting the memory)
- Suppress with "untilWindowClose": the inverse of the desired rate. If the desired rate is R (events per second), the inverse is the buffering time: 1 /R (seconds per event).

Details and Public Interfaces

There are two primary changes we're proposing: Altering the window definition and adding a KTable#suppress operator.

Add Grace Period to Window Spec Builders

For our new Suppress operator to support window final results, we need to define a point at which window results are actually final!

Currently, the only hint we have about this is the retention period (called until/maintainMs/retentionPeriod). Once we drop an old window, we can obviously never update it again. However, this isn't a convenient "final" point for our purpose. Retention time is typically very large (the default is one day), and users of IQ may need to keep the retention time large in order to support queries even over final windows. Plus, how long a window is retained is really a property of the window store implementation. In principle, an in-memory implementation might choose to retain events for a short time while a remote distributed store may keep them forever. This shouldn't prohibit the usage of final results, though.

To resolve this conflict, we're adding a new concept to the window spec: grace period. This is an amount of time that the window should accept out-of-order events after the window ends. After the grace period passes, the window is considered "closed" and will never be updated again. The grace period places a lower-bound constraint on the retention time, but otherwise has no implication on retention.

We will add a "grace" configuration to the window spec builders (TimeWindows, JoinWindows, UnlimitedWindows, and SessionWindows).

We are also deprecating the retention time and segment interval as specified on the window spec itself and moving them to the lower-level store configurations.

```
On TimeWindows, JoinWindows, SessionWindows:
/**
 * Reject late events that arrive more than {@code millisAfterWindowEnd}
 * after the end of its window.
 *
 * Lateness is defined as (stream_time - record_timestamp).
 *
 * @param millisAfterWindowEnd The grace period to admit late-arriving events to a window.
 * @return this updated builder
 */
@SuppressWarnings("deprecation") // will be fixed when we remove segments from Windows
public XWindows grace(final long millisAfterWindowEnd); // where XWindows is the appropriate builder


On Windows (and SessionWindows, although it's not abstract there):
+ /**
+  * Return the window grace period (the time to admit
+  * late-arriving events after the end of the window.
+  */
+ public abstract long gracePeriodMs();


/**
 ...
+ * @deprecated since 2.1. Use {@link Joined#retention()}
+ * or {@link Materialized#retention}
+ * or directly configure the retention in a store supplier and use
+ * {@link Materialized#as(WindowBytesStoreSupplier)}.
 */
+ @Deprecated
public Windows<W> until(final long durationMs);

/**
 ...
 * @deprecated since 2.1. Use {@link Joined#retention()} or {@link Materialized#retention} instead.
 */
@Deprecated
public long maintainMs();


/**
 ...
 * @deprecated since 2.1. Instead, directly configure the segment interval in a store supplier and use {@link
Materialized#as(WindowBytesStoreSupplier)}.
 */
@Deprecated
public long segmentInterval();
```

The Window/Session BytesStoreSupplier interface already includes retention period. The existing behavior is that that retention period overrides maintainMs if set on the window. We'll preserve this behavior.

We will add retention to Materialized:

```
/**
 * Configure retention period for window and session stores. Ignored for key/value stores.
 *
 * Overridden by pre-configured store suppliers
 * ({@link Materialized#as(SessionBytesStoreSupplier)} or {@link Materialized#as(WindowBytesStoreSupplier)}).
 *
 * @return itself
 */
public Materialized<K, V, S> withRetention(final long retentionMs);
```

New Suppress Operator

We will add the new operator to KTable:

```
public interface KTable<K, V> {


 /**
 * Suppress some updates from this changelog stream, determined by the supplied {@link Suppressed}.
 *
 * This controls what updates downstream table and stream operations will receive.
 *
 * @param suppressed Configuration object determining what, if any, updates to suppress.
 * @return A new KTable with the desired suppress characteristics.
 */
 KTable<K, V> suppress(final Suppressed<? super K> suppress);


}
```

Note the absence of a variant taking Materialized. The result of a suppression will always be (eventually) consistent with the source KTable, so I'm thinking right now that it would be "bad advice" to present the option to materialize it.

We will also create the config object Suppressed:

```
public interface Suppressed<K> {

/**
 * Marker interface for a buffer configuration that is "strict" in the sense that it will strictly
 * enforce the time bound and never emit early.
 */
interface StrictBufferConfig extends BufferConfig<StrictBufferConfig> {}

/**
 * Marker interface for a buffer configuration that will strictly enforce size constraints
 * (bytes and/or number of records) on the buffer, so it is suitable for reducing duplicate
 * results downstream, but does not promise to eliminate them entirely.
 */
interface EagerBufferConfig extends BufferConfig<EagerBufferConfig> {}

interface BufferConfig<BC extends BufferConfig<BC>> {

        /**
         * Create a size-constrained buffer in terms of the maximum number of keys it will store.
         */
        static EagerBufferConfig maxRecords(final long recordLimit);

        /**
         * Set a size constraint on the buffer in terms of the maximum number of keys it will store.
         */
        BC withMaxRecords(final long recordLimit);

        /**
         * Create a size-constrained buffer in terms of the maximum number of bytes it will use.
         */
        static EagerBufferConfig maxBytes(final long byteLimit);

        /**
         * Set a size constraint on the buffer, the maximum number of bytes it will use.
         */
        BC withMaxBytes(final long byteLimit);

        /**
         * Create a buffer unconstrained by size (either keys or bytes).
         *
         * As a result, the buffer will consume as much memory as it needs, dictated by the time bound.
         *
         * If there isn't enough heap available to meet the demand, the application will encounter an
         * {@link OutOfMemoryError} and shut down (not guaranteed to be a graceful exit). Also, note that
         * JVM processes under extreme memory pressure may exhibit poor GC behavior.
         *
```

```
         * This is a convenient option if you doubt that your buffer will be that large, but also don't
         * wish to pick particular constraints, such as in testing.
         *
         * This buffer is "strict" in the sense that it will enforce the time bound or crash.
         * It will never emit early.
         */
        static StrictBufferConfig unbounded();

        /**
         * Set the buffer to be unconstrained by size (either keys or bytes).
         *
         * As a result, the buffer will consume as much memory as it needs, dictated by the time bound.
         *
         * If there isn't enough heap available to meet the demand, the application will encounter an
         * {@link OutOfMemoryError} and shut down (not guaranteed to be a graceful exit). Also, note that
         * JVM processes under extreme memory pressure may exhibit poor GC behavior.
         *
         * This is a convenient option if you doubt that your buffer will be that large, but also don't
         * wish to pick particular constraints, such as in testing.
         *
         * This buffer is "strict" in the sense that it will enforce the time bound or crash.
         * It will never emit early.
         */
        StrictBufferConfig withNoBound();

        /**
         * Set the buffer to gracefully shut down the application when any of its constraints are violated
         *
         * This buffer is "strict" in the sense that it will enforce the time bound or shut down.
         * It will never emit early.
         */
        StrictBufferConfig shutDownWhenFull();

        /**
         * Sets the buffer to use on-disk storage if it requires more memory than the constraints allow.
         *
         * This buffer is "strict" in the sense that it will never emit early.
         */
        StrictBufferConfig spillToDiskWhenFull();

        /**
         * Set the buffer to just emit the oldest records when any of its constraints are violated.
         *
         * This buffer is "not strict" in the sense that it may emit early, so it is suitable for reducing
         * duplicate results downstream, but does not promise to eliminate them.
         */
        EagerBufferConfig emitEarlyWhenFull();
}

/**
 * Configure the suppression to emit only the "final results" from the window.
 *
 * By default all Streams operators emit results whenever new results are available.
 * This includes windowed operations.
 *
 * This configuration will instead emit just one result per key for each window, guaranteeing
 * to deliver only the final result. This option is suitable for use cases in which the business logic
 * requires a hard guarantee that only the final result is propagated. For example, sending alerts.
 *
 * To accomplish this, the operator will buffer events from the window until the window close (that is,
 * until the end-time passes, and additionally until the grace period expires). Since windowed operators
 * are required to reject late events for a window whose grace period is expired, there is an additional
 * guarantee that the final results emitted from this suppression are eventually consistent with the upstream
 * operator and its queriable state, if enabled.
 *
 * @param bufferConfig A configuration specifying how much space to use for buffering intermediate results.
 * This is required to be a "strict" config, since it would violate the "final results"
 * property to emit early and then issue an update later.
 * @param <K> The key type for the KTable to apply this suppression to. "Final results" mode is only available
 * on Windowed KTables (this is enforced by the type parameter).
 * @return a "final results" mode suppression configuration
```

```
 */
static Suppressed<Windowed> untilWindowCloses(final StrictBufferConfig bufferConfig);

/**
 * Configure the suppression to wait {@code timeToWaitForMoreEvents} amount of time after receiving a record
 * before emitting it further downstream. If another record for the same key arrives in the mean time, it
replaces
 * the first record in the buffer but does <em>not</em> re-start the timer.
 *
 * @param timeToWaitForMoreEvents The amount of time to wait, per record, for new events.
 * @param bufferConfig A configuration specifying how much space to use for buffering intermediate results.
 * @param <K> The key type for the KTable to apply this suppression to.
 * @return a suppression configuration
 */
 static <K> Suppressed<K> untilTimeLimit(final Duration timeToWaitForMoreEvents, final BufferConfig
bufferConfig);


/**
 * Use the specified name for the suppression node in the topology.
 * <p>
 * This can be used to insert a suppression without changing the rest of the topology names
 * (and therefore not requiring an application reset).
 * <p>
 * Note however, that once a suppression has buffered some records, removing it from the topology would cause
 * the loss of those records.
 * <p>
 * A suppression can be "disabled" with the configuration {@code untilTimeLimit(Duration.ZERO, ...}.
 *
 * @param name The name to be used for the suppression node and changelog topic
 * @return The same configuration with the addition of the given {@code name}.
 */
 Suppressed<K> withName(final String name);
}
```

Along with the suppression operator, we will add several metrics. Note that suppress will **not** add to the **skipped-records** metrics. "Skipped" records are records that are for one reason or another invalid. "Suppressed" records are intentionally dropped, just like filtered records. Likewise with events arriving later than the grace period for windows.

Metrics

Note: I'm not proposing roll-up metrics for these. They would be reported at the processor-node level. I suspect this is actually fine, and roll-ups can easily be added later if necessary.

Metrics we'll add:

- late records (new records older than the grace period) are currently metered as "**skipped-records**" and logged at WARN level. As noted, this is not correct, so we will change the logs to DEBUG level and add new metrics:
  - average and max observed lateness of all records: to help configure the grace period
    - `(DEBUG) record-lateness-[avg | max] type=stream-processor-node-metrics client-id=<threadId> task-id=<taskId> processor-node-id=<processorNodeId>`
  - rate and total of dropped events for closed windows
    - `(INFO) late-record-drop-[rate | total] type=stream-processor-node-metrics client-id=<threadId> task-id=<taskId> processor-node-id=<processorNodeId>`
- expired records (new records for segments older than the state store's retention period) are currently not metered and logged at DEBUG level. Since the grace period is currently equivalent to the retention period, this should currently be rare, as such events would never reach the state store but be marked as skipped and dropped in the processor. However, since we are deprecating `Windows.until` and defining retention only on the state store, it would become much more common. Therefore, we'll add some new state store metrics:
  - rate and total of events for expired windows
    - `(INFO) expired-window-record-drop-[rate | total] type=stream-[storeType]-state-metrics client-id=<threadId> task-id=<taskId> [storeType]-state-id=[storeName]`
- intermediate event suppression
  - current, average, and peak intermediate suppression buffer size
    - `(DEBUG) suppression-buffer-size-[current | avg | max] type=stream-buffer-metrics client-id=<threadId> task-id=<taskId> buffer-id=<bufferName>`
  - current, average, and peak number of records in the suppression buffer

        - `(DEBUG) suppression-buffer-count-[current | avg | max] type=stream-buffer-metrics client-id=<threadId> task-id=<taskId> buffer-id=<bufferName>`

    - intermediate suppression emit rate and total: to how often events are emitted
        - `(DEBUG) suppression-emit-[rate | total] type=stream-processor-node-metrics client-id=<threadId> task-id=<taskId> processor-node-id=<processorNodeId>`

    - min and average intermediate suppression overtime: to determine whether the intermediate suppression emitAfter is delaying longer than necessary. **This metric may be unnecessary, since it's equivalent to (timeToWaitForMoreEventsConfig - observedLatenessMetric)**.
        - `(INFO) intermediate-result-suppression-overtime-[min | avg] type=stream-processor-node-metrics client-id=<threadId> task-id=<taskId> processor-node-id=<processorNodeId>`

Examples

Here are some examples of programs using the new operator to achieve various goals.

Final window results per key

Imagine we wish to send an alert if a user has fewer than 3 events in an hour. For the example, we'll wait up to 10 minutes after the hour ends for late-arriving events.

```
builder.<Integer, String>stream("events")
    .groupByKey()
    .windowedBy(TimeWindows.of(3600_000).grace(600_000)
    .count()
    .suppress(untilWindowCloses(BufferConfig.unbounded()))
    .toStream()
    .filter((key, value) -> value < 3)
    .foreach((key, value) -> sendAlert("User " + key.key() + " had fewer than 3 events in window " + key.
window()));
```

Note that we can handle limited memory in a few different ways:

```
// Option 1: expect not to run out of memory
windowCounts
  .suppress(untilWindowCloses(unbounded()))


// Option 2: shut down gracefully if we need too much memory
windowCounts
  .suppress(untilWindowCloses(maxBytes(5_000_000).shutDownWhenFull()))


// Option 3: Start using disk if we need too much memory
windowCounts
  .suppress(untilWindowCloses(maxBytes(5_000_000).spillToDiskWhenFull()))
```

Any of these constructions yield a strict guarantee that each windowed key will emit exactly one event.

Note about the "shut down when full" option:

This may seem like a strange option for production code, but consider that in practice there is limited heap size available. As with all data structures, if you need to store more data than fits in memory, then you will run out of memory and crash. For Java in particular, as the available heap approaches 0, the garbage collector will consume more and more CPU, which can cause the application to become unresponsive long before an actual OutOfMemoryError occurs.

The idea of this option is to pick some reasonably large bound to allow a graceful and performant shutdown before this occurs.

Then, the operator can increase the heap size and restart, or the developer can decrease the grace period (decreasing the required heap), or choose another strategy, such as on-disk buffering.

Rate-limited updates

Suppose we wish to reduce the rate of updates from a KTable to roughly one update every 30s per key. We don't want to use too much memory for this, and we don't think we'll have updates for more than 1000 keys at any one time.

```
table
  .suppress(untilTimeLimit(Duration.ofSeconds(30), maxRecords(1000)))
  .toStream(); // etc.
```

Buffer Eviction Behavior (aka Suppress Emit Behavior)

I propose to offer the following **constraints**:

1. **at all times, the buffer enforces the #keys bound**
2. **at all times, the buffer enforces the size (#bytes) bound**
3. **at all times, the buffer enforces the time bound**

Inserting a new record may violate any or all of the constraints, and advancing stream time may violate constraint 3.

If any of the constraints are violated, the buffer will evict (and emit) records until all constraints are again satisfied.

In evicting (and emitting) records, the buffer will use the following **eviction strategy**:

- **the oldest record (by timestamp) in the buffer gets evicted**

In updating a record whose key is always in the buffer, offset order is respected.

This can be visualized as the following algorithm:

```
process(key, value, timestamp):
  buffer.insertOrUpdate(key -> (value, timestamp))
  while(buffer.constraintsViolated()) {
    (key -> (value, timestamp)) := buffer.evictOldest()
    emit(key, value, timestamp)
  }
```

This spec will maintain offset ordering for all updates to a key, but may re-order events between keys.

Here are some examples to illustrate the dynamics:

Offset order is maintained when a key is updated, regardless of timestamp:

offset	key	value	timestamp	buffer-slot-0	buffer-slot-1	emit	notes
0	A	x	0	(A,x,0)	---	---	
1	A	y	1	(A,x,1)	---	---	Subsequent update to A overwrites the prior value

offset	key	value	timestamp	buffer-slot-0	buffer-slot-1	emit	notes
0	A	x	1	(A,x,1)	---	---	
1	A	w	0	(A,w,0)	---	---	Subsequent update to A overwrites the prior value (even though this is an earlier event by time)

The behavior is straightforward with no late events

Enforcing a key limit of 2:

offset	key	value	timestamp	buffer-slot-0	buffer-slot-1	emit	notes
0	A	w	0	(A,w,0)	---	---	
1	A	x	1	(A,x,1)	---	---	
2	B	y	2	(A,x,1)	(B,y,2)	---	
3	C	z	3	(B,y,2)	(C,z,3)	(A,x,1)	A is the oldest when we violate the key constraint

Enforcing a byte limit of 3 (each character is one byte)

offset	key	value	timestamp	buffer-slot-0	buffer-slot-1	buffer-slot-2	emit	notes
0	A	xx	0	(A,xx,0)	---	---	---	
1	A	yy	1	(A,yy,1)	---	---	---	
2	B	zz	2	(B,zz,2)	---	---	(A,yy,1)	A is the oldest entry when the size constraint is violated, so we emit it.

Enforcing "emit after 2ms":

offset	key	value	timestamp	buffer-slot-0	buffer-slot-1	buffer-slot-2	emit	notes
0	A	w	0	(A,w,0)	---	---	---	
1	A	x	1	(A,x,1)	---	---	---	
2	B	y	2	(A,x,1)	(B,y,2)	---	---	
3	C	z	3	(B,y,2)	(C,z,3)	---	(A,x,1)	The stream time is now 3, so we emit all the records up to time 1 (this is just A)

Note: newly added late events can be immediately evicted.

Enforcing "emit after 2ms":

offset	key	value	timestamp	buffer-slot-0	buffer-slot-1	buffer-slot-2	emit	notes
0	A	w	3	(A,w,3)	---	---	---	
1	A	x	1	---	---	---	(A,x,1)	The stream time is 3, and the timestamp of A is now 1, so we have to emit it.
2	B	y	1	---	---	---	(B,y,1)	The stream time is 3, and the timestamp of B is 1, so we have to emit it.

Likewise with a key constraint of 2:

offset	key	value	timestamp	buffer-slot-0	buffer-slot-1	emit	notes
0	A	w	0	(A,w,0)	---	---	
1	A	x	1	(A,x,1)	---	---	
2	B	y	2	(A,x,1)	(B,y,2)	---	
3	C	z	0	(A,x,1)	(B,y,2)	(C,z,0)	Even though it is the most recently added, C is still the oldest event when the key constraint is violated

And of course with a size constraint of 3 bytes as well:

offset	key	value	timestamp	buffer-slot-0	buffer-slot-1	buffer-slot-2	emit	notes
0	A	xx	0	(A,xx,0)	---	---	---	
1	A	yy	1	(A,yy,1)	---	---	---	
2	B	zz	0	(A,yy,1)	---	---	(B,zz,0)	Even though B is the most recently added event, it is still the oldest one by timestamp when the size constraint is violated

Big records can push multiple events out of the buffer

Size constraint of 3 bytes:

offset	key	value	timestamp	buffer-slot-0	buffer-slot-1	buffer-slot-2	emit	notes
0	A	x	0	(A,x,0)	---	---	---	
1	B	y	1	(A,x,0)	(B,y,1)	---	---	

2	C	zzz	2	(C,zzz,2)	---	---	(A,x,0),(B,y,1)	No other records can fit in the buffer with C, and A and B are both older than C

In fact, events can be so big they don't fit in the buffer at all.

Still 3 bytes:

offset	key	value	timestamp	buffer-slot-0	buffer-slot-1	buffer-slot-2	emit	notes
0	A	x	0	(A,x,0)	---	---	---	
1	B	y	1	(A,x,0)	(B,y,1)	---	---	
2	C	zzzz	2	---	---	---	(A,x,0),(B,y,1),(C,zzzz,2)	A and B are both older than C, so they must be emitted before C, and C itself doesn't fit in the buffer, so it must then be immediately emitted.

Rejected alternative: evicting by offset instead of timestamp.

This causes strange behavior when there is a time constraint involved:

offset	key	value	timestamp	buffer-slot-0	buffer-slot-1	buffer-slot-2	emit	notes
1	A	x	2	(A,x,2)	---	---	---	
2	B	y	1	(A,x,2)	(B,y,1)	---	---	
3	C	z	3	(A,x,2)	(B,y,1)	(C,z,3)	---	Even though we B is old enough to emit, it's not at the head of the queue, so we can't emit it
4	C	zz	4	(C,zz,4)	---	---	(A,x,2),(B,y,1)	It's now time to emit A, and once we do, it's no longer blocking B, so we can emit it as well.

Rejected alternative: evicting by timestamp only when the buffer is time constrained, and using offset order otherwise

Having just one eviction strategy simplifies everything: documentation, explanations, the code, the testing, etc.

I'd also argue that neither strategy is any more or less surprising for key- or size-constrained buffers. All the "flushing" behaviors above are also present in offset-ordered eviction.

Time-based eviction is perfectly legal, as we are only required to maintain a partial order over the partition (order only needs to be maintained within each key). Such partial reordering happens when we repartition anyway.

Rejected alternative (maybe a future option): "reset the timeout" when updating a key:

The current behavior is that, if you specify some timeout, say 5 minutes, any record that gets buffered is guaranteed to be emitted at the 5 minute mark, regardless of how often it's updated. Aka, it is emitted 5 minutes after the **first** update.

An alternative is to effectively reset the timer on each update, so the record would be emitted 5 minutes after the **last** update. This behavior is analogous to session windowing; it allows suppression to turn a high-frequency "run" up updates to a record into a single emitted record.

This was rejected in the current API simply because it may suppress a record **forever**, if that record is updated with a frequency higher than the suppression timeout.

For the default behavior, it seemed better to go with the one that is guaranteed to emit after the specified timeout.

But this could easily be added as an alternative in the future, if there is demand for it.

Compatibility, Deprecation, and Migration Plan

The only part of the KIP that's relevant to existing APIs is the deprecation of Windows#until/maintainMs. I've described above how the deprecation warnings will look, and also what new APIs will replace them. All the implementations will be done in such a way that existing Streams applications will have exactly the same semantics before and after this KIP, so there's no concern about continuing to use the deprecated APIs.

One other change we could consider in the future is to revisit the state store caching mechanism, but that also serves the function of limiting i/o to the state store, so I think that should be a separate discussion.

Rejected Alternatives

There are many alternative ways to achieve this behavior.

At the top of the list, I also considered having dedicated operators for final events on windowed tables and update suppression on non-windowed ones. But the current proposal supports the same behavior with just one new operator.

We also considered having windowed computations directly provide the "final results" feature via an "Emitted" config object, but ultimately settled on adding the grace period to the window and letting "suppress" deal with suppressing all but the final result.

In fact, I previously proposed not to support "final results" directly, but instead to allow a suppression with an upper bound on lateness. using the same time for this lateness bound and intermediate suppression would naturally yield final results only. But we judged that this API was too esoteric. The version we have now is much more straightforward for this use case.