Structured Streaming Programming Abstraction, Semantics, and APIs - Apache JIRA

Authors: Matei Zaharia <matei@databricks.com>, Michael Armbrust <michael@databricks.com>, Tathagata Das <tdas@databricks.com>, Reynold Xin <rxin@databricks.com>

History:

2016-03-14: First draft

Introduction

Having learned from building and deploying pre-existing Spark Streaming in the last three years, we are starting an effort, dubbed structured streaming, to rebuild streaming functionalities on top of Catalyst and DataFrames. Since it is a very large effort, we will decompose the design of structured streaming into a series of design docs.

This document describes the programming abstraction and APIs for Structured Streaming (aka Streaming DataFrames) in Spark 2.0+. The goal is to create an abstraction that is easy to understand yet with well defined semantics. This document does not cover how the underlying execution works, and the semantics should be independent of the underlying execution engine.

The most important question is probably the **semantics** of streaming programs: what does a streaming program “mean”, and what will it output? Many systems today have complicated semantics, or do not specify their semantics at all. Well-defined semantics are essential both to make the system easy to understand for users and to ensure its APIs work together.

Some of the questions answered in this document are:

1. What is the overall **semantics** of streaming?
2. Are streams a different **class** than DataFrame?
3. How does **windowing** happen?
4. Is **event time** a special concept?
5. Is **processing time** a special concept / column?

6. How does **sessionization** happen?
7. How to specify **triggering**?
8. How to specify **output** behavior, including output to user code (e.g. foreach)?
9. How to do **program setup**, e.g. adding queries and restarting on failure?
10. How to handle apps that want to **feed data into themselves**, e.g. ML model updates?

Semantics

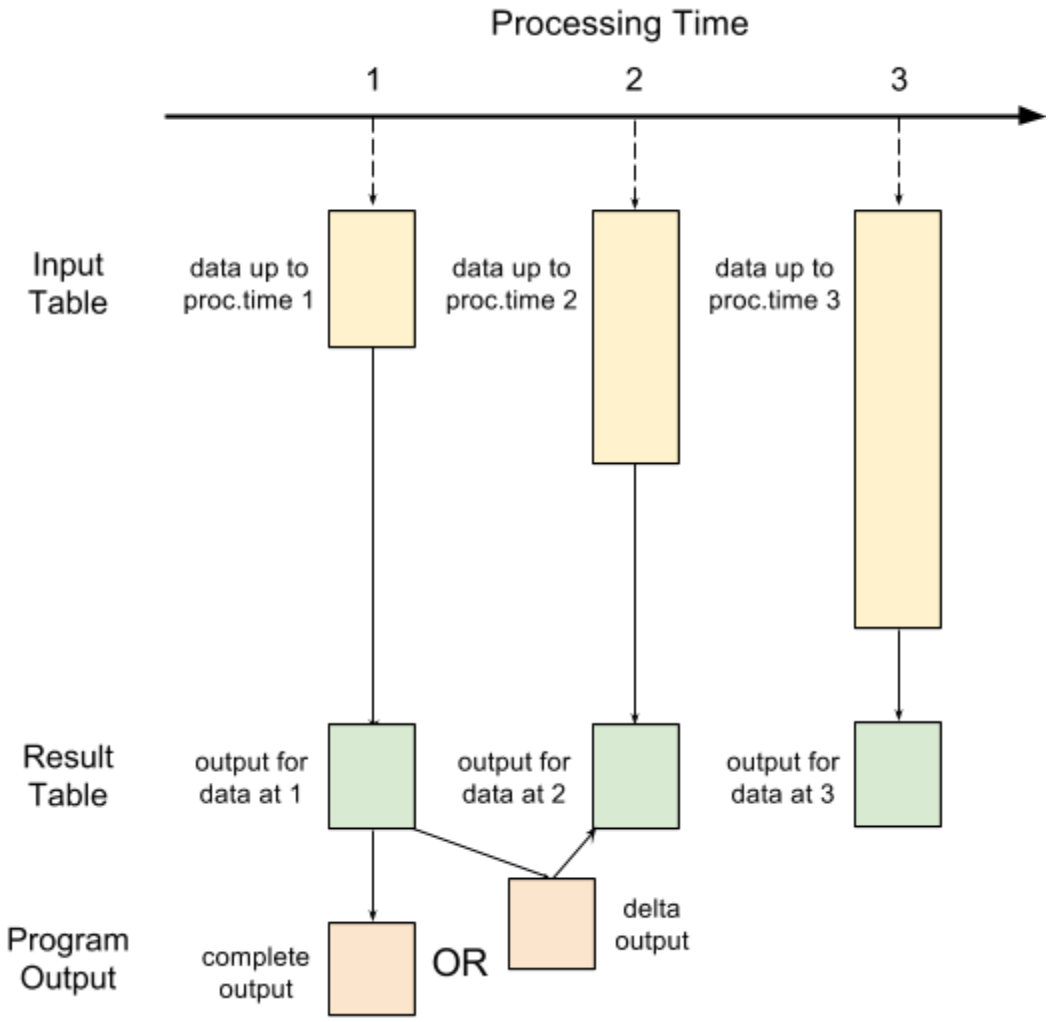
We propose a simple model, “repeated queries” (RQ). In this model, users reason about this abstraction as if they are reasoning about static tables, and apply all the previous knowledge they have with respect to SQL/DataFrames.

It works like this:

- Logically, each input stream is an **append-only *table*** (i.e. DataFrame), where records can arrive in any order across our system’s processing time (event time is just a field in each record).
- Users define ***queries*** as just traditional SQL or DataFrame queries on this whole table, which return a new table whenever they are executed in processing time.
- Users set a ***trigger*** to say when to run each query and output, which is based purely on processing time. The system makes best effort to meet such requirement. In some cases, the trigger can be “as soon as possible”.
- Finally, users set an ***output mode*** for each query. The different output modes are:
    - **Delta**: Although logically the output of each query is always a table, users can set a “delta” output mode that only writes the records from the query result changed from the last firing of the trigger.
        - These are physical deltas and not logical deltas. That is to say, they specify what rows were added and removed, but not the logical difference for some row.
        - Users must specify a primary key (can be composite) for the records. The output schema would include an extra “status” field to indicate whether this is an “add”, “remove”, or “update” delta record for the primary key.
    - **Append**: A special case of the Delta mode that does not include removals. There is no need to specify a primary key, and the output would not include the status field.
    - **Update(**-in-place**)**: Update the result directly in place (e.g. update a MySQL table). Similar to delta, a primary key must be specified.
    - **Complete**: For each run of the query, create a complete snapshot of the query result.

That's it: in short, the results from RQ are identical to running each query manually at various points in time, and saving either its whole output or just the difference from the last time it was executed. Each execution works like standard Spark SQL.

Physically (under the hood), the optimizer should incrementalize the query execution. In cases incrementalization is not possible (i.e. query execution requires unbounded data or state), an exception (AnalysisException) should be thrown.



Common Operations

In this section we discuss how RQ supports some common streaming operations:.

Map-only ETL job

- Query is a map over the input table
- Trigger is processing time (e.g. every 1 minute, or as fast as possible)
- Output mode is append (write a new HDFS file with the added records).

Infinite aggregation / landmark window (track user visits by page)

- Query is “select count(*) from visits group by page”
- Trigger is processing time, as soon as possible
- Output mode is in-place updates to a table in MySQL (update just changed records)
- For the same query, we could request writing the whole table of counts to a file in HDFS instead by just changing the output mode and sink.

Sliding window aggregation by event time (count visits by page and window)

A sliding window is defined with 4 parameters:

1. Time column
2. Size of the window
3. Sliding interval
4. When the window starts

A tumbling window is just a special case of sliding window where the sliding interval is the same as the window size. As an example, we can define a window using the eventTime column, where the window size is 10 mins, and it moves forward every 1 min, and it should start at 00:00.

To specify the following query, we can do:

- Query is a count with a special (multi) group-by operator that maps each event to all the windows it is part of (would be similar to a flatMap in Spark)
- Trigger is processing time, every 1 min
- Output mode is in-place updates in MySQL; if an already outputted window gets a late event, we can also update its old record.

Query on top of windows (find most popular window in past hour)

- Query: add a top k operator over the previous windowed count query
- Trigger is processing time, every 1 min

- Output mode can be a flat file in HDFS that we update, or a Kafka stream that we write the top K to (both with “complete table” output mode)

Session statistics (count number and average length of sessions)

- Query groups data using a special operator that assigns a session ID (combination of session start, end, and session grouping key) to each record; can then do count(*), max(time) - min(time), etc grouping by session ID.
- Trigger is processing time, every 1 min
- Output mode can be flat file or table to update.

Output Modes

The most common outputs (sinks) will be some file system (HDFS / S3) or an external database system.

We should support append, delta, and complete modes for file systems, but not update-in-place, since those are difficult to do.

For external database systems, we should be able to support all modes. In most cases, when dealing with external database systems, user applications would want the update-in-place mode.

Details of Time and Triggers

To specify what the RQ model means, in particular with respect to event time, we give a slightly more formal definition:

- We assume there is a system-level concept of “processing time” (e.g. current wall clock time) that increases monotonically, even across restarts of Spark. Also, each input record is assigned a processing time reflecting its order of ingest. Once assigned, the associated processing time with each record is fixed across retries or Spark restarts. For example, the system might log what processing time it assigned to a Kafka offset.
- Records may also have an event time field, on which we make no assumptions.
- Each run of a query can see the current processing time and all data up to that time (the current processing time can be accessed through a function, processing_time()).
- Programs may also add an operator to drop events whose event time is too far behind, which will then be considered in optimization (e.g. dropping old state).

The point of these details is to make each run of a query deterministic given its processing time. Specifically, at a given processing time, we have a known prefix of each input stream. Thus, we can deterministically compute the right output table, or a delta from the previous output table.

Alternative: We can also introduce event time triggers in the future. To define event time triggers, we have an “event time watermark”, which is a function of the current event table and current processing time that says what event time we have “likely” gotten most of the data up to. This is a global function, EVENT_NOW. It can only increase as data arrives. Event time triggers fire when this passes various values. Queries are allowed to output results for “old” windows regardless of EVENT_NOW. Note that this is actually very similar as applying a filter on event time column using processing time triggers.

Out-of-order Data

In streams, data might arrive late for a variety of reasons.

This discard delay is the only extra configuration user applications need to worry about. It is used by the system to make the state tractable. If the system needs to handle arbitrarily late data, then the system state might grow indefinitely.

In RQ’s style of aggregation, late records that arrive before the discard delay will simply be part of the query runs once the records arrive, and then the output will correct itself (update-in-place) or inject deltas to indicate updates.

In some cases, applications might want to hold off for a period of time before emitting any output to avoid writing out partial outputs. This can be accomplished at the user level by simply injecting a filter on event time.

Comparison with Other Models

The benefits of the repeated query model are:

1. There is no special concept of a stream -- everything is a table and a SQL query. If you understand “normal” Spark SQL, you can understand this model.
2. Unlike Google Dataflow, triggers and outputs are independent from the query itself. In Dataflow, a window (which is really a group-by from the SQL point of view) must also select an output mode and trigger, which is confusing. In RQ, one can also have these for queries that don’t logically use windows, and each concept is simpler.
3. Easy to share code with batch processing.
4. Many of our desired features (sessions, feedback loops, etc) are easy to express.

The main downside of RQ is that incrementalization of queries is done by the planner, so our planner must support common combinations of queries, output modes and triggers. For example, it should know when it can drop old data/state. Users get little control over this.

We also compare this model with other systems:

Storm

Storm exposes a lower level API that requires users to explicitly specify low level data flow topology. This provides some of the low level primitives to build streaming applications, but are too low level for most users to reason about.

Storm assumes a monotonic system (processing) time metric, and it is difficult to deal with event time, and is also difficult to build/reason about stateful operations or fault tolerance.

Discretized streams (aka dstream)

Unlike Storm, dstream exposes a higher level API similar to RDDs. There are two main challenges with dstream:

1. Similar to Storm, it exposes a monotonic system (processing) time metric, and makes support for event time difficult.
2. Its APIs are tied to the underlying microbatch execution model, and as a result lead to inflexibilities such as changing the underlying batch interval would require changing the window size.

RQ addresses the above:

1. RQ operations support both system time and event time.
2. RQ APIs are decoupled from the underlying execution model. As a matter of fact, it is possible to implement an alternative engine that is not microbatch-based for RQ.
3. In addition, due to the declarative specification of operations, RQ leverages a relational query optimizer and can often generate more efficient query plans.

CQL (Streams + Tables)

CQL, Calcite and some other streaming DBs have separate concepts of streams and tables. These typically assume a monotonic time metric, so event time is tricky with out-of-order data. For most of these systems, once an output is generated, it cannot be corrected.

Dataflow

Dataflow treats the input as a big table and separates “what” to compute from “when” to compute it, making it easier to think about event time (we can compute the same query later and see late events). The model is pretty complex because the concept of a window ties together grouping, triggers, incremental output, and late data policies. RQ can be viewed as a simpler version of Dataflow, where we separated these concepts and made trigger/output a 1st class citizen of the RQ specification. Late data policies in most cases are just part of the query itself (e.g. users can add an explicit filter in the query to ignore data later than a specific late).

Code Examples

Per-Record Transformation (ETL)

In this case, we incrementally ETL data from one location to another, every 5 secs.

```
val records = sqlContext.read.format(“json”).stream(“hdfs://input”)
val urls = records.map(lower($“url”))
urls.write
  .trigger(“5 sec”)
  .outputMode(Append)
  .format(“parquet”)
  .startStream(“hdfs://output”)
```

The output should look identical to the input, except we lower case url and convert all the data from JSON to Parquet format.

Infinite Aggregation (Landmark Window)

In this case, we compute some counter for each user and write the output directly to a table in MySQL through JDBC. The table in MySQL will always have the latest result. Note that there is no special handling of late data. As soon as late data is processed, the result table would include it.

```
val records = sqlContext.read.format(“json”).stream(“hdfs://input”)
val counts = records.groupBy(“user”).count()
counts.write
  .trigger(ProcessingTime(“5 sec”))
  .outputMode(UpdateInPlace(“user”))
  .format(“jdbc”)
```

```
  .startStream("mysql://...")
```

The output should look like the following table in MySQL:

user	count
matei	2
rxin	5
...	...

Initially, this API would require explicit primary key specification. In some cases, the system should be able to infer the primary key automatically. We are not going to auto-infer to avoid having to explain when the system can or cannot automatically infer.

Windowed Aggregation on Processing Time

In this case, we compute the the running 5-min count every minute.

```
val records = sqlContext.read.format("json").stream("hdfs://input")
val counts =
  records.groupBy(
    window(
      timeColumn = processing_time(),
      windowSize = "5 min",
      slidingInterval = "1 min"))
  .count()
counts.write
  .trigger("1 min")
  .outputMode(Append)
  .format("parquet")
  .startStream("hdfs://output")
```

For the following input:

0 min: record 1
5 min 1 sec: record 2
5 min 2 sec: record 3
6 min 1 sec: record 4
6 min 2 sec: record 5
...

The output should look like:

window	count
0 min - 5 min	1
1 min - 6 min	3
2 min - 7 min	5
...	...

Windowed Aggregation on Event Time

In this case, we compute the running 5-min count every minute, but avoid producing any output for the last window, in order to deal with out-of-order data.

```
val records = sqlContext.read.format(“json”)
  .withEventTime(“eventTime”, discardDelay=“5 min”)
  .stream(“hdfs://input”)
val windows = records.groupBy(window(“eventTime”, “5 min”, “1 min”))
val counts = windows.count().fitler(“window.time < processing_time() - 5 min”)
counts.write
  .trigger(EventTime(“1 min”))
  .outputMode(Append)
  .format(“parquet”)
  .startStream(“hdfs://output”)
```

The output looks similar to aggregation on processing time.

Sessions

In this case, we count the number of events for each user session, and also output the start and end session time. Sessions are considered to end if there is no activity for 30 seconds, or if there is an explicit “logout” event.

```
val records = sqlContext.read.format(“json”)
  .withEventTime(“eventTime”, discardDelay=“30 sec”)
  .stream(“hdfs://input”)
val sessions = records.sessionize(“user”, “eventTime”, “30 sec”, $“type” == “logout”)
val stats = sessions.agg(count(“*”), min(“eventTime”), max(“eventTime”))
stats.write
  .trigger(EventTime(“1 min”))
  .outputMode(Append)
```

```
  .format(“kafka”)
  .stream(“...”)
```

For the following input:

0 sec: user 1
1 sec: user 2
10 sec: user 1
40 sec: user 3
45 sec: user 3 logout
...

The output should look like the following in Kafka:

deltaStatus	user	count	min(eventTime)	max(eventTime)
add	User 1	1	0	10
add	User 2	3	1	31
add	User 3	5	40	45
...	...	...	...	...

Sessions with User Code

In this case, we are running some arbitrary user defined code on the sessions.

```
val records = sqlContext.read.format(“json”)
  .withEventTime(“eventTime”, discardDelay=“30 sec”)
  .stream(“hdfs://input”)
val sessions = records.sessionize(“user”, “eventTime”, “30 sec”, $“type” == “logout”)
val stats = sessions.agg(count(“*”), min(“eventTime”), max(“eventTime”))
stats.write
  .trigger(EventTime(“1 min”))
  .outputMode(Deltas)
  .foreachPartition { /* user code on deltas */ }
```

API Details

Overview: DataFrame as Streams

At a high level, the API works as follows:

1. Create one or more streams using methods of SQLContext. These are logical plans to read and transform data.
2. Transform streams to give new streams representing a query. Apart from standard SQL operators, there are new ones for windows, sessions and adding time fields.
3. Launch a ContinuousQuery by assigning a trigger, output mode and sink to a stream. This object can then be used to stop the execution.

This document proposes simply using DataFrame (and also Dataset) to represent streams. We will introduce a new isStreaming method:

```
class DataFrame {
  ...
  def isStreaming: Boolean
  ...
}
```

All existing actions that return non-DataFrames (e.g. DataFrame.count(), head()) should throw runtime exceptions for streaming data.

Unlike pre-existing Spark Streaming, we don't introduce any top level concepts (i.e. there is no special context for streaming or special programming abstraction). An alternative is to create separate type hierachies for streaming DataFrames, similar to pre-existing Spark Streaming. The following table summarizes the pros and cons:

Streams are DataFrames	**Streams are separate classes**
+ Easily share methods between streaming and batch code + Simpler class hierarchy overall - Some methods and libraries will throw exception when called on streams	+ More type safety - Need a bunch of new classes, at least StreamDataFrame & GroupedStreamData - Harder to write libraries that use both (may need a superclass, but that's 2 new classes)

DataFrameReader: Creating Streams

Streams will be created through DataFrameReader (access through context.read), similar to the data source API today. The following methods are added to support streams:

```
class DataFrameReader {
  …
  def withEventTime(column: Column, discardDelayMs: Long): DataFrameReader
  def withEventTime(column: Column, discardDelay: String): DataFrameReader

  def stream(): DataFrame
  def stream(path: String): DataFrame
  ...
}
```

An event time column can be set by the user with a discard delay.

One additional feature of streams over data sources will be rate limiting (API TBD).

DataFrame: Transforming Streams

Relational Operators

All of the existing relational operators work the same way they do on DataFrames, under the RQ model (i.e. assume they see all data since the beginning of each stream). In cases where this would be too expensive, the planner may throw an exception (e.g. if we do a sort on an input stream, or cartesian product of two input streams).

Time

Processing time for each event is tracked by the system implicitly, and is available as a function processing_time(). As discussed earlier, event time is just a normal column.

Event time is specified in the stream definition in DataFrameReader, as discussed earlier. Note that by putting event time specification in the stream definition, all queries against a stream would have the same view of data at any given processing time.

Windowing

Windowing will be implemented as a multi-group-by (possible to map each record to multiple windows, e.g. for sliding windows). It has the following API:

```
def window(timeColumn, windowSize, slidingInterval, startTime): Column
```

Parameters:

- timeColumn is a column in the data to window by; it must be a time column (either processing time or event time column).
- windowSize is the size of each window.
- slidingInterval is the time to advance before starting a new window.
- startTime is an optional parameter giving an initial offset past multiples of slidingInterval; for example, if we have a window advancing every hour, we may still want to do it 5 seconds “past the hour”. It is equivalent to applying a filter on the input stream before windowing.

The result of window() is a Column that can be included in a group-by function. The key is a time column specifying the start of each window (e.g. if our window has slidingInterval = 1 minute and offset = 0, there is one key for each minute), and the value is the original event. One can then do aggregates, mapGroups, etc. Each event can appear in multiple keys if the windows overlap.

Sessionization

Sessionization is also implemented as a special kind of grouping operator, which assigns a *session key* to each input record by reconstructing sessions from the stream of events. The idea in sessionization is that you get a stream of events with a key in them, e.g. user ID browsing a page. You then want to construct sessions out of the events that occurred close enough in time: for example, you say events for the same user within 30 minutes of each other count as the same session, but if the user logs in again hours later, that’s a new session. Sessions may also be ended when we see particular events, e.g. “logout”. We propose the following API:

```
def sessionize(keyColumn, timeColumn, timeout, isCloseEvent): Colum
```

Parameters:

- keyColumn is the column to map events to the same session (e.g. user ID above)
- timeColumn is either a processing or event time column
- timeout is the duration of inactivity for closing a session, in the timeColumn metric
- isCloseEvent is an optional expression to decide whether a given event marks an end of session (in which case we’d close it before its timeout)

The result of sessionize() is a Column that can be used in a group-by function. The key is {keyColumn, sessionStartTime} and the value is an event. That is, we assign each event to one session, and identify each session by its keyColumn and the time of its first event (which is a unique way to identify nonoverlapping sessions).

Together with this group-by, users can call aggregates, window functions and UDAFs to perform common session operations. For example, it is very easy to count the total number of sessions, sessions started in the last minute, etc.

Note that if late events come in, some sessions' keys will change (if we see an earlier event) and some sessions may be merged; this is the same in Dataflow.

In the future, if we were to support pattern matching on events (e.g. CEP style), we would need to decide how to deal with late data. That might require buffering enough to account for discard delay, and then pass data into pattern matching engine sorted by event time.

mapWithState

Using the sessionization operator, we should also be able to expose an interface similar to Spark Streaming's mapWithState that lets users track a state for each key. The trickiest part will be whether their user-defined function is guaranteed to see the records in a particular order, or any order. It's probably simplest to just tell it the records will come in any order, including possibly out of order w.r.t. event time. In the case where two sessions are merged, we'd have to run their UDF again from the beginning of the new session to compute the state. Thus, maybe we shouldn't even allow sessionization on event time columns if you use mapWithState.

DataFrameWriter: Writing Output

Every stream can be turned into a running ContinuousQuery by assigning a trigger, output mode and sink. The ContinuousQuery object acts as a handle for stopping it too. The following functions are added to DataFrameWriter (access through df.write) to support streams:

```
class DataFrameWriter {
  ...
  def trigger(trigger: Trigger): DataFrameReader
  def outputMode(mode: OutputMode): DataFrameReader
  def startStream(): ContinuousQuery
  def startStream(path: String): ContinuousQuery
  ...
}
```

Note that DataFrameWriter already contains a “mode” for indicating save mode (e.g. append, overwrite, error if exists, ignore). We should also investigate whether we could reconcile the two and have only one “mode”.

Triggers

Initially, triggers are based on processing time.

1. ProcessingTime(period): fires at multiples of period in processing time. If the cluster is overloaded, we will skip some firings and wait until the next multiple of period. It would be great to be able to warn users of system overloading (e.g. in logs, UIs).
    - **Alternative:** we could also make this fire ASAP if the cluster is overloaded, but that makes it harder to align outputs with processing time windows.
2. ASAP (or ProcessingTime(0)): fires as quickly as possible

In the future, we may add triggers based on the amount of data (e.g. don’t fire unless there are at least 100 MB or data or you’ve waited at least 10 seconds) or event time.

Output Modes

Each stream execution can use one of several output modes, though not all modes are supported for all sinks:

- Complete: compute and write the whole output table anew each time. For file systems, this will create a new file, but we can add a flag to reuse the same file.
- Update(keyCols): update, add or delete records in a key-value sink (e.g. MySQL) using the given columns as the key.
- Append: output only the new (added) records. For file systems, this will create a new file, and for tabular sinks, it will just insert records.
- Deltas(keyCols): output a set of {deltaType, data} records where the deltaType says whether this record is new, modified or deleted since last time. This also requires some key columns to figure out which records are “modified”.

Sinks

There will be a pluggable sink API similar to data sources. Not all sinks will support all output modes -- the Update one can only be used on key-value or otherwise indexed sinks.

Supported Execution Configurations

Some combinations of queries and output modes might not be allowed by the planner if they will require overly expensive work (e.g. query is a map and output mode is Complete) or unbounded state (e.g. median since start of stream). We should decide on a policy for these.

One strawman is to assume that all aggregations not based on time use a finite number of keys, and see whether that results in the work per input record and total state being bounded.

Advanced Features

This part is not fully fleshed-out yet, but we think the API can support a few interesting features:

- **Ad-hoc queries on streams:** by adding a special “Spark SQL temp table” sink, we can let users query quickly-updating data through JDBC or standard Spark SQL. For example, one could set up a table that shows all events in the last hour of data, or all active sessions, or aggregates that can then be queried. We optimize the representation of these kind of temp tables because they are updated so often.

- **Feedback within an application:** for use cases like online learning, where we want to train an ML model and simultaneously apply it to new data, one easy option is to have streaming queries that depend on each other through a temp table. For example, one query can write / update the model parameters to a table, and another can read it, or the same query can join new data against the old table of parameters. Some API may be needed to say which query runs first (e.g. add dependencies between these).

- **Dynamically changing queries:** the API supports adding and removing queries at runtime, so we’d just have to implement it.

Detailed Answers to Questions

This section proposes some more specific answers the ten questions at the start.

- **Semantics:** RQ model above
    - Results are equivalent to running a given query on the whole table of input data at particular times, and writing either a full result or deltas from last run
- **Classes:** streams are the same class as DataFrame
    - Add new methods such as isStreaming; disallow some actions on streams
- **Windowing** is a group-by, possibly mapping each record to multiple groups
    - Returns some kind of GroupedData, with a “window” column for the key
    - New feature: can do another groupBy on top of GroupedData to add keys

    - Window column's value might just be start time of each window
    - Windows can be either on processing time or event time
    - Later, may have a limited form of count-based windows, at least on event time (sessions may also provide this right now)
- **Event time:** special operators let you mark a column in each input stream as event time and discard highly delayed events
    - Each stream with event time has two duration settings: a "watermark delay" that says when it's allowed to fire watermarks (how long it waits after the last event for a given time), and a "discard delay" saying that that it can discard old records after waiting this much time past the watermark; this may be infinite. When the watermark delay >= the discard delay then we will never revoke an answer, but if it's smaller than we might output an answer we will later change.
    - Event time triggers are based on watermarks -- all input streams must pass a given watermark for the trigger to fire
    - Analyzer tracks that this column means event time throughout
- **Processing time:**
    - Processing time is assigned when a record is actually ready to be processed; rate limiting may therefore artificially increase the processing time. This processing time should be consistent across task retries or Spark restarts.
    - For each record it is implicitly tracked by the system, and can be made available using function processing_time()
    - For unit tests, allow specifying a column in the data as processing time; this will need to be increasing across input records, and will replace system time.
- **Sessionization:** special operator creates session windows similar to a group-by:
    - Params: stream.sessionize(keyColumn, timeColumn, timeout, isCloseEvent)
        - keyColumn is the column to map events to the same session
        - timeColumn is either a processing or event time column
        - timeout in the relevant time metric for dropping old events
        - isCloseEvent is an optional expression to decide whether a given event marks an end of session (in which case we'd end it before its timeout)
    - Returns a GroupedData of {{key, startTime}, event}
    - Can use window functions and UDAFs on this to aggregate
        - Some complexity with UDAFs and late records: do we sort the records by time in each group, even if they are late?
- **Triggering:** periodic triggers, either in processing time or event time
    - For both types, system overload may cause some firings to be missed; in that case, we fire again the next time the time field reaches a multiple of the period
    - Alternative: processing time triggers only, and use EVENT_NOW to figure out which event windows are ready and only output those
- **Output:** we will support several output modes:
    - Output to storage systems (HDFS, MySQL, Kafka, etc) in several ways:
        - Complete tables: write each output table once to a new file / table / etc
        - Update in place: replace a file / table atomically (or just changed rows)

        - Deltas: write records added, records changed and records deleted and mark them somehow; might want a special case for added-only
    - Output to user code: we invoke a foreach function at least once on each partition, passing either complete tables or deltas as above
    - Output to a Spark SQL temp table, which one can query / expose through JDBC
- **Program setup:** a strawman is the following:
    - User program starts by initializing a SqlContext, setting up UDFs, etc
    - Next, it creates one or more streaming queries; these can be given explicit names or will otherwise be called Query1, Query2, etc; this happens on all runs
    - Finally, it starts with a given checkpoint "context" set; it will recover from the checkpoint if it ran before and the queries match what was there earlier
    - Unlike Spark Streaming, init code runs every time, and UDFs may change
- **Data flow within app:** one proposal here is to have a special sink for Spark SQL temp tables. Then, other streaming queries could join against these tables.
    - Example: ML training query builds the "modelparams" table, then ML serving reads from those to predict.
    - Can add a notion of priority to control order of triggers.